实时渲染 第4版

第1章 概述

Real-time rendering is concerned with rapidly making images on the computer. It

is the most highly interactive area of computer graphics. An image appears on the

screen, the viewer acts or reacts, and this feedback affects what is generated next.

This cycle of reaction and rendering happens at a rapid enough rate that the viewer

does not see individual images, but rather becomes immersed in a dynamic process.

实时渲染涉及计算机快速绘制图像。它是计算机图形学最具交互性领域。一张图像出现在屏幕上，观众的行为或者反应，这种反馈会影响下一步产生的内容。这种反应和渲染的循环发生在一个快速的频率以至于观众看到的不是独立的静态图像，而是沉浸在一个动态的过程中。

The rate at which images are displayed is measured in frames per second (FPS)

or Hertz (Hz). At one frame per second, there is little sense of interactivity; the user

is painfully aware of the arrival of each new image. At around 6 FPS, a sense of

interactivity starts to grow. Video games aim for 30, 60, 72, or higher FPS; at these

speeds the user focuses on action and reaction.

图像显示的速率一般用帧速率或者赫兹衡量。一秒一帧的情况下，几乎没有交互感，用户往往看到的是每副独立的图像。大概在每秒6帧的时候，交互感开始提升。视频游戏的帧速率目标为30,60,72或者更高的帧速率，在这种帧速率下，用户的焦点才会集中在行为以及反应上。

Movie projectors show frames at 24 FPS but use a shutter system to display each

frame two to four times to avoid flicker. This refresh rate is separate from the display

rate and is expressed in Hertz (Hz). A shutter that illuminates the frame three times

has a 72 Hz refresh rate. LCD monitors also separate refresh rate from display rate.

电影放映的帧速率一般是24，但是会使用快门 系统每帧显示2到4次去避免闪烁。这种刷新的频率区分于帧速率，被称为赫兹。每个快门使每帧显示3次的话，那么就是72赫兹的刷新率。LCD监视器同样区别开刷新率和帧速率。

Watching images appear on a screen at 24 FPS might be acceptable, but a higher

rate is important for minimizing response time. As little as 15 milliseconds of

temporal delay can slow and interfere with interaction [1849]. As an example, head-

mounted displays for virtual reality often require 90 FPS to minimize latency.

以24的帧速率观看屏幕上的图像是可以被接受的，但是更高的速率对于减少反应时间非常重要。15毫秒的延迟将会减慢和影响交互。举个例子，头戴式沉浸虚拟现实常常需要90的帧速率去减少延迟。

There is more to real-time rendering than interactivity. If speed was the only

criterion, any application that rapidly responded to user commands and drew

anything on the screen would qualify. Rendering in real time normally means

producing three dimensional images.

实时渲染比交互需要更多东西。如果速度是惟一的标准，那么任何一个可以快速响应用户命令以及在屏幕上显示图形的应用都是合格的。实时渲染通常意味着产生三维图形。  
Interactivity and some sense of connection to three-dimensional space are sufficient

conditions for real-time rendering, but a third element has become a part of

its definition: graphics acceleration hardware. Many consider the introduction of the

3Dfx Voodoo 1 card in 1996 the real beginning of consumer-level three-dimensional

graphics [408]. With the rapid advances in this market, every computer, tablet, and

mobile phone now comes with a graphics processor built in. Some excellent

examples of the results of real-time rendering made possible by hardware

acceleration are shown in Figures 1.1 and 1.2.

交互感和三维空间的感觉就足够实时渲染的条件，但是第三种元素慢慢成为实时渲染描述的一部分：图像加速硬件。在1996年，消费者级别的三维图像加速器刚开始发展的时候，很多人考虑引入3Dfx Voodoo 1 卡。随着市场的高速发展，每一台电脑，平板，手机都内嵌了图形处理器。一些很好的例子显示了图形加速器在处理实时渲染的效果。（图1.1和图1.2）



Figure 1.1. A shot from Forza Motorsport 7. (Image courtesy of Turn 10 Studios, Microsoft.)

图1.1 来自Forza Motorsport 7 的镜头（图片微软Turn 10工作室提供）



Figure 1.2 The city of Beauclair rendered in The Witcher 3

图1.2 巫师3中渲染的Beauclair城市

Advances in graphics hardware have fueled an explosion of research in the field

of interactive computer graphics. We will focus on providing methods to increase

speed and improve image quality, while also describing the features and limitations

of acceleration algorithms and graphics APIs. We will not be able to cover every topic

in depth, so our goal is to present key concepts and terminology, explain the most

robust and practical algorithms in the field, and provide pointers to the best places to

go for more information. We hope our attempts to provide you with tools for

understanding this field prove to be worth the time and effort you spend with our

book.

图像处理硬件的发展给交互式计算机图像领域的研究提供了爆炸式增长的燃料。我们的目标集中在提供方法图增加帧速率以及改善图形的质量，同时也描述了加速算法和图形APIs的特征以及局限性。我们并不会涵盖所有的主题的深度，因为我们的目标是呈现关键概念和术语，解释该领域最强大的和最实用的算法，同时提供更多信息的关键点。我们希望我们能够去提供给你理解这个领域的工具去证明值得你花时间和精力去阅读这本书。

* 1. 目录预览

What follows is a brief overview of the chapters ahead.

以下是对各章节的简要介绍。

Chapter 2, The Graphics Rendering Pipeline. The heart of real-time rendering is the

set of steps that takes a scene description and converts it into something we can see.

第2章：图形渲染管线。实时渲染的核心就是一系列的步骤将对场景的描述转换成一些我们能看到的东西。

Chapter 3, The Graphics Processing Unit. The modern GPU implements the stages of

the rendering pipeline using a combination of fixed-function and programmable units.

第3章：图形处理器。现代GPU实现了固定渲染管线和可编程片段的结合。

Chapter 4, Transforms. Transforms are the basic tools for manipulating the position,

orientation, size, and shape of objects and the location and view of the camera.

第4章：变换。变换是操作物体位置、方向、大小、形状以及本地位置、摄像机试图的基本工具

Due to space constraints, we have made a chapter about Collision Detection free

for download at realtimerendering.com, along with appendices on linear algebra and

trigonometry.

由于空间的限制，我们提供一个章节关于碰撞检测、线性代数以及三角函数。可以在realtimerendering.com网站免费下载。

1.2 Notation and Definitions 符号和定义

First, we shall explain the mathematical notation used in this book. For a more

thorough explanation of many of the terms used in this section, and throughout this

book, get our linear algebra appendix at realtimerendering.com.

首先，我们将解释本书所用的数学符号。对着这部分更多详尽的解释可以在realtimerendering.com网站上得到有关线性代数的内容。

1.2.1 Mathematical Notation 数学符号

Table1.1 summarizes most of the mathematical notation we will use. Some of the

concepts will be described at some length here.

表1.1 总结了我们将用到的大部分数学符号。其中一些概念会得到一些描述。

Note that there are some exceptions to the rules in the table, primarily shading

equations using notation that is extremely well established in the literature, e.g., L

for radiance, E for irradiance, and σs for scattering coefficient.

请注意，表中规则有一些例外，只要是着色方程所用的符号，在其他文献中是非常确定的。比如L代表辐射度，E代表辐照度，σs代表散射度 。

The angles and the scalars are taken from R, i.e., they are real numbers. Vectors

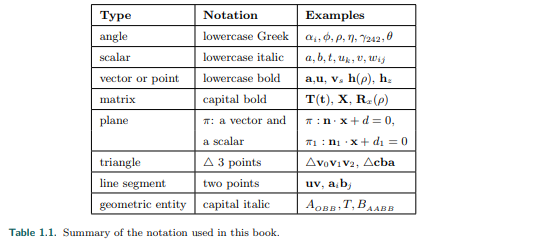
and points are denoted by bold lowercase letters, and the components are accessed

as v=(vx,vy,vz)（竖写）, that is, in column vector format, which is commonly used

in the computer graphics world. At some places in the text we use (vx, vy, vz)

instead of the formally more correct (vx vy vz)T, since the former is easier to read.

角度和标量取自R,即，他们都是实数。向量和点用粗体小写表示，组件表示为v=(vx,vy,vz) （竖写）,那是，列向量格式，经常用在计算机图形学中。在本书的一些地方我们用(vx, vy, vz)来代替更为正式准确的(vx vy vz)T,因为前者更便于阅读。



Using homogeneous notation, a coordinate is represented by four values v =

(vx vy vz vw)T, where a vector is v = (vx vy vz 0)T and a point is v = (vx vy vz 1)T

. Sometimes we use only three-element vectors and points, but we try to avoid any

ambiguity as to which type is being used. For matrix manipulations,it is extremely

advantageous to have the same notation for vectors as for points. For more

information, see Chapter 4 on transforms. In some algorithms, it will be convenient

to use numeric indices instead of x, y, and z, for example v = (v0 v1 v2)T. All

these rules for vectors and points also hold for two-element vectors; in that case, we

simply skip the last component of a three-element vector.

使用一致的表示法，一个坐标表示为 v =(vx vy vz vw)T，向量表示为v = (vx vy vz 0)T，点表示为：v = (vx vy vz 1)T。有时候我们只用3个元素表示向量和点，但是无论我们用哪一种类型，会尽量避免产生歧义。对于矩阵操作，向量和点使用统一的表示法会有很大的优点。更多信息，参见第4章变换。在一些算法中，使用数字下标代替x,y,z会很便利，比如v = (v0 v1 v2)T.向量和点的所有规则适用于二维向量，在那种情况下，我们仅仅是跳过了三维向量的最后的一个元素。

The matrix deserves a bit more explanation. The common sizes that will be used

are 2 × 2, 3 × 3, and 4 × 4. We will review the manner of accessing a 3 × 3 matrix

M, and it is simple to extend this process to the other sizes. The (scalar) elements of

M are denoted mij , 0 ≤ (i, j) ≤ 2, where i denotes the row and j the column, as in

Equation 1.1．

矩阵值得更多解释。我们常用的大小是2x2,3x3和4x4。我们将会重述3x3矩阵，因为他扩展到其他尺寸会非常简单。矩阵的元素（标量）表示为Mij, 0 ≤ (i, j) ≤ 2,i表示行，j表示列。如公式1.1（常规的3x3矩阵表示）

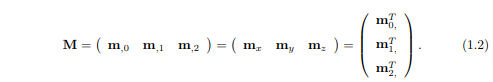
The following notation, shown in Equation 1.2 for a 3 × 3 matrix, is used to isolate

vectors from the matrix M: m,j represents the jth column vector and mi, represents

the ith row vector (in column vector form). As with vectors and points, indexing

the column vectors can also be done with x, y, z, and sometimes w, if that is more

convenient:



下面的符号，公式1.2所示的3x3矩阵，是从矩阵M独立出向量组表示：m,j表示第j列向量，mi,表示第i行向量（以列向量的形式）。对于向量和点来说，索引列向量也可以用x,y,z,w表示，如果那样更方便的话。

A plane is denoted π : n · x + d = 0 and contains its mathematical formula, the

plane normal n and the scalar d. The normal is a vector describing what direction

the plane faces. More generally (e.g., for curved surfaces), a normal describes this

direction for a particular point on the surface. For a plane the same normal happens

to apply to all its points. π is the common mathematical notation for a plane. The

plane π is said to divide the space into a positive half-space, where n · x + d > 0,

and a negative half-space, where n · x + d < 0. All other points are said to lie in the

plane。

平面表示为π：n·x+d=0和他的数学表达式包含了平面法线n和标量d。法线是一个向量，描述了平面朝向的方向。更一般的讲（例如曲面）,法线描述了表面的一个特定的点的方向。对于一个平面来讲，所有的点具有同样的法线。π是平面常用的数学符号。一个平面可以说成分割空间成为一个正方向空间n · x + d > 0，和一个负方向空间

n · x + d < 0。其他所有的点平铺在平面上。

A triangle can be defined by three points v0, v1, and v2 and is denoted by

△v0v1v2.

三角形可以被3个点v0, v1, 和 v2定义，用△v0v1v2表示。

Table 1.2 presents some additional mathematical operators and their notation.

The dot, cross, determinant, and length operators are explained in our downloadable

linear algebra appendix at realtimerendering.com. The transpose operator turns a

column vector into a row vector and vice versa. Thus a column vector can be written

in compressed form in a block of text as v = (vx vy vz)T. Operator 4, introduced

in Graphics Gems IV [735], is a unary operator on a two-dimensional vector. Lettiing

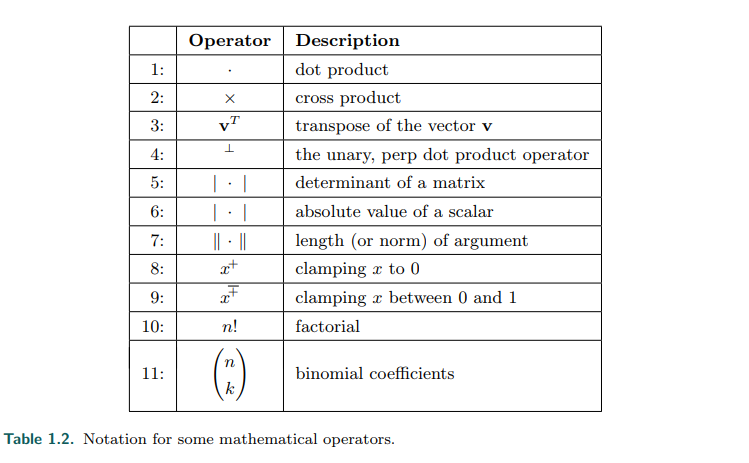
this operator work on a vector v = (vx vy)T gives a vector that is perpendicular to v,

i.e., v⊥ = (−vy vx)T. We use |a| to denote the absolute value of the scalar a, while

|A| means the determinant of the matrix A. Sometimes, we also use |A| = |a b c| =

det(a, b, c), where a, b, and c are column vectors of the matrix A.

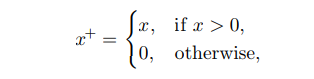
表1.2呈现了一些额外的数学运算符和他们的符号。点乘，叉乘、行列式和长度运算符是在下载资料realtimerendering.com线性代数里面有解释。转置运算符可以翻转一个列向量成为一个行向量，反之亦然。因此列向量可以写成v = (vx vy vz)T.操作符4，在图形宝典4[735]有介绍，是二维向量的一元运算符。这个运算符作用于v = (vx vy)T会得到一个垂直于v的向量，也就是：v⊥ = (−vy vx)T.我们用|a|去表示标量a的绝对值，|A|的意思是矩阵A的行列式。有时候，我们也使用|A| = |a b c| =det(a, b, c),a,b,c是矩阵的列向量。



Operators 8 and 9 are clamping operators, commonly used in shading calculations.

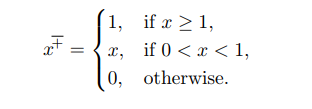
操作符8和9是钳位运算符，通常用在着色计算中。

Operator 8 clamps negative values to 0:



操作符8将负值钳位为0。

and operator 9 clamps values between 0 and 1:



操作符9将值钳位为0到1之间。

The tenth operator, factorial, is defined as shown below, and note that 0! = 1:



第10个运算符，阶乘，定义为如下所示，注意0! = 1。

The eleventh operator, the binomial factor, is defined as shown in Equation 1.6:



第11个运算符，二项式因子，定义如公式1.6所示。

Further on, we call the common planes x = 0, y = 0, and z = 0 the coordinate

planes or axis-aligned planes. The axes ex = (1 0 0)T, ey = (0 1 0)T, and

ez = (0 0 1)T are called main axes or main directions and individually called the

x-axis, y-axis, and z-axis. This set of axes is often called the standard basis. Unless

otherwise noted, we will use orthonormal bases (consisting of mutually perpendicular

unit vectors).

接下来，我们称公共界面x = 0, y = 0, 和 z = 0坐标平面或者是轴对齐平面。轴ex = (1 0 0)T, ey = (0 1 0)T, 和ez = (0 0 1)T称为主轴或主方向，分别称为x轴，y轴和z轴。这组轴常被称为标准基础。除非是特别提到，我们将使用正交基（由互相垂直的单位向量组成）。

The notation for a range that includes both a and b, and all numbers in between, is

[a, b]. If we want all number between a and b, but not a and b themselves, then we

write (a, b). Combinations of these can also be made, e.g., [a, b) means all numbers

between a and b including a but not b.

范围的符号包含a和b，所有的数字在其中间是[a, b]。如果我们想要所有的数字在a,b之间，但是不包括a,b自己是(a, b)。结合上述我们可以得到例[a, b)，意思是所有的数字在a,b之间，包含a但是不包含b。

The C-math function atan2(y,x) is often used in this text, and so deserves some

attention. It is an extension of the mathematical function arctan(x). The main

differences between them are that − π /2 < arctan(x) < π/ 2 , that 0 ≤ atan2(y, x)

< 2π, and that an extra argument has been added to the latter function. A common

use for arctan is to compute arctan(y/x), but when x = 0, division by zero results.

The extra argument for atan2(y,x) avoids this.

C-数学函数atan2(y,x)，经常在我们的文章中用到，因此知道更多注意。他是数学函数arctan(x)的一个扩展。主要不同的地方是：− π /2 < arctan(x) < π/ 2 , 而 0 ≤ atan2(y, x) < 2π,并且在后一个函数里有一个额外的参数。常见的arctan的用法是计算arctan(y/x),但是当x=0时，会得到除以0的结果。额外的参数atan2(y,x)就是为了避免这种情况。

In this volume the notation log(n) always means the natural logarithm, loge (n), not

the base-10 logarithm, log10(n).

在本书中，符号log(n)意思是自然对数loge (n)，而不是底为10的对数，log10(n)。

We use a right-hand coordinate system since this is the standard system for three

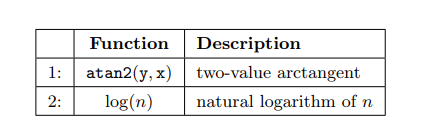
dimensional geometry in the field of computer graphics.

我们使用右手坐标系因为在计算机图形学中这是三维几何的标准系统。

Colors are represented by a three-element vector, such as (red, green, blue), where

each element has the range [0, 1].

颜色是用三原色表示法，如（红、绿、蓝），每个分量都是在范围[0, 1].



1.2.2 Geometrical Definitions 几何定义

The basic rendering primitives (also called drawing primitives) used by almost all

graphics hardware are points, lines, and triangles.

几乎所有的图形处理硬件使用的基本渲染图元（也称为绘制图元）是：点、线、三角形。

Throughout this book, we will refer to a collection of geometric entities as either a

model or an object. A scene is a collection of models comprising everything that is

included in the environment to be rendered. A scene can also include material

descriptions, lighting, and viewing specifications.

在本书中，我们称几何实体为模型或者对象。一个场景是所有的模型集合，包含了要渲染的环境锁包含的所有东西，同样也包含了材质描述，光照和视角规则等。

Examples of objects are a car, a building, and even a line. In practice, an object

often consists of a set of drawing primitives, but this may not always be the case; an

object may have a higher kind of geometrical representation, such as Bezier curves

or surfaces, or subdivision surfaces. Also, objects can consist of other objects, e.g., a

car object includes four door objects, four wheel objects, and so on.

比如对象是一辆汽车，一个建筑物甚至一条线。实际上，一个物体常常是一组绘制图元组成，但并不是都是这种情况。一个物体也可能具有更高类型的几何表示，比如贝塞尔曲线或曲面，或者细分曲面。当然，对象可能包含其他对象，比如一辆车包含了4个门对象，4个轮子对象等。

1.2.3 Shading 着色

Following well-established computer graphics usage, in this book terms derived from

“shading,” “shader,” and related words are used to refer to two distinct but related

concepts: computer-generated visual appearance (e.g., “shading model,” “shading

equation,” “toon shading”) or a programmable component of a rendering system

(e.g., “vertex shader,” “shading language”). In both cases, the intended meaning

should be clear from the context.

在完善的计算机图形运用下，在这本书从“着色”，“着色器”，和相关的用到的词语衍生出两个不同但有联系的概念：计算机生成视觉外观（如“着色模型”，“着色方程”，“卡通着色”）或者渲染系统可编程组件（如“顶点着色器”，“着色器语言”）。在两种情况下，从上下文应该清楚其指的含义。

Further Reading and Resources 更多阅读和资源

The most important resource we can refer you to is the website for this book:

realtimerendering.com. It contains links to the latest information and websites

relevant to each chapter. The field of real-time rendering is changing with real-time

speed. In the book we have attempted to focus on concepts that are fundamental

and techniques that are unlikely to go out of style. On the website we have the

opportunity to present information that is relevant to today’s software developer, and

we have the ability to keep it up-to-date.

最重要的资源我们可以提供给你的是本书的网站：realtimerendering.com。他包含了对应各章节最新的信息和网站链接。实时渲染领域随着实时的速度更新。在这本书中，我们试图把目标集中在基本的概念和不太会过时的技术。在网站上，我们有机会去呈现与现在软件开发者相关的信息，我们有能力使其保持最新状态。

Chapter 2 The Graphics Rendering Pipeline

第2章 图形渲染管线

This chapter presents the core component of real-time graphics, namely the graphics

rendering pipeline, also known simply as “the pipeline.” The main function of the

pipeline is to generate, or render, a two-dimensional image, given a virtual camera,

three-dimensional objects, light sources, and more. The rendering pipeline is thus

the underlying tool for real-time rendering. The process of using the pipeline is

depicted in Figure 2.1. The locations and shapes of the objects in the image are

determined by their geometry, the characteristics of the environment, and the

placement of the camera in that environment. The appearance of the objects is

affected by material properties, light sources, textures (images applied to surfaces),

and shading equations.

这一章主要展现了实时渲染的核心组件：图形渲染管线，简称为“管线”。管线的主要功能是给定虚拟的摄像机，三维物体，光源等，生成或渲染成一个二维图像。图形渲染管线因此是实时渲染的基础工具。图2.1描述了图形渲染管线的过程。物体的位置和形状是由他们的几何体、环境特征和环境中摄像机的位置决定的。物体的外观是由材质属性，光源、纹理（应用于表面的图片）和着色方程式影响。

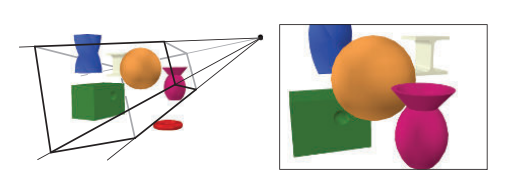


Figure 2.1. In the left image, a virtual camera is located at the tip of the pyramid

(where four lines converge). Only the primitives inside the view volume are rendered.

For an image that is rendered in perspective (as is the case here), the view volume is

a frustum (plural: frusta), i.e., a truncated pyramid with a rectangular base. The

right image shows what the camera “sees.” Note that the red donut shape in the left

image is not in the rendering to the right because it is located outside the view

frustum. Also, the twisted blue prism in the left image is clipped against the top

plane of the frustum.

图2.1 左边图形，一个虚拟摄像机位于金字塔的顶端（4条线的交点）。只有图元在视锥体内才会被渲染。对于透视渲染的图像（如图所示），视锥体是一个平截头体（复数：截头体）即切去了头的金字塔有一个矩形基底。右侧图像显示了摄像机所看到的。注意左侧图像中红色圆环形状并不在右边渲染图像中，因为他的位置位于视锥体外。同样，左侧图像中蓝色扭曲角柱被截头体的上部平面截断了。

We will explain the different stages of the rendering pipeline, with a focus on

function rather than implementation. Relevant details for applying these stages will

be covered in later chapters.

我们将解释渲染管线的不同阶段，并且把焦点放在功能上而不是实现。应用不同阶段的相应细节将在最后章节讲述。

2.1 Architecture 结构

In the physical world, the pipeline concept manifests itself in many different forms,

from factory assembly lines to fast food kitchens. It also applies to graphics

rendering. A pipeline consists of several stages [715], each of which performs part of

a larger task.

在物理世界，流水管线概念有很多不同的方式呈现，从工厂装配线到快餐餐厅。他同样应用于图形渲染领域。一个流水管线包含了几个阶段，每个阶段执行较大任务的一部分。

The pipeline stages execute in parallel, with each stage dependent upon the result of

the previous stage. Ideally, a nonpipelined system that is then divided into n

pipelined stages could give a speedup of a factor of n. This increase in performance

is the main reason to use pipelining. For example, a large number of sandwiches can

be prepared quickly by a series of people—one preparing the bread, another adding

meat, another adding toppings. Each passes the result to the next person in line and

immediately starts work on the next sandwich. If each person takes twenty seconds

to perform their task, a maximum rate of one sandwich every twenty seconds, three

a minute, is possible. The pipeline stages execute in parallel, but they are stalled

until the slowest stage has finished its task. For example, say the meat addition

stage becomes more involved, taking thirty seconds. Now the best rate that can be

achieved is two sandwiches a minute. For this particular pipeline, the meat stage is

the bottleneck, since it determines the speed of the entire production. The toppings

stage is said to be starved (and the customer, too) during the time it waits for the

meat stage to be done.

流水线阶段之间的执行是并行的，并且每个阶段都依赖于前一阶段的结果。理想的话，一个无流水线的系统被分割为n个流水阶段那么会提升n倍的执行速度。性能上的增长是使用流水线的主要原因。举个例子：很大数量的三明治会被一系列的工人快速制作——一个人准备面包，另一个加肉，另一个添加酱，每个人完成之后就传递给流水线上的下一个人并立即开始下一个三明治的工作。如果每个人花费20s时间去执行他们的任务，那么最大的速率就是每20s制作一个三明治，可能的话，每分钟制作3个。流水管线阶段之间的执行是并行的，但是他们会停止直到最慢的阶段完成他的任务。举个例子，加肉阶段变得更复杂，需要花费30s，现在最快的速率是每分钟能完成2个三明治。对于这个特殊的流水线，加肉阶段就是一个瓶颈，因为他决定了完成整个产品的速度。加酱阶段需要等待（顾客也是一样）加肉阶段完成他的任务。

This kind of pipeline construction is also found in the context of real-time computer

graphics. A coarse division of the real-time rendering pipeline into four main stages—

application, geometry processing, rasterization, and pixel processing—is shown in

Figure 2.2. This structure is the core—the engine of the rendering pipeline—which is

used in real-time computer graphics applications and is thus an essential base for

discussion in subsequent chapters. Each of these stages is usually a pipeline in itself,

which means that it consists of several substages. We differentiate between the

functional stages shown here and the structure of their implementation. A functional

stage has a certain task to perform but does not specify the way that task is

executed in the pipeline. A given implementation may combine two functional stages

into one unit or execute using programmable cores, while it divides another, more

time-consuming, functional stage into several hardware units.

在实时渲染计算机图像中也能看到这种流水线结构。实时渲染流水线可以粗略的划分为4个主要阶段——应用阶段，几何阶段，光栅化和像素处理——在图2.2显示。这个结构是图形渲染流水管线引擎的核心，用在实时渲染计算机图形应用上，因此是后续章节讨论的必要基础。每个阶段内部同样存在流水管线，意思是他包含了几个子阶段。我们区别于这里展示的功能阶段和他们实行的结构。功能阶段有一定的任务要执行，但没有指定任务在管道中执行的方式。 给定的实现可以将两个功能阶段组合成一个单元或者使用可编程核执行，同时将另一个更耗时的功能阶段划分为多个硬件单元。

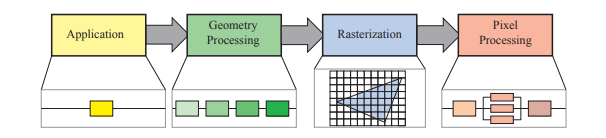


Figure 2.2. The basic construction of the rendering pipeline, consisting of four stages:

application, geometry processing, rasterization, and pixel processing. Each of these

stages may be a pipeline in itself, as illustrated below the geometry processing stage,

or a stage may be (partly) parallelized, as shown below the pixel processing stage.

In this illustration, the application stage is a single process, but this stage could also

be pipelined or parallelized. Note that rasterization finds the pixels inside a primitive,

e.g., a triangle.

图2.2 图形渲染管线的基本结构，包括了4个阶段：应用阶段，几何阶段，光栅化和像素处理。每个阶段内部可能只有一个流水线，比如几何阶段下面的插画，或者阶段内部是并行的，像像素处理阶段下面所示。在这幅插图中，应用阶段是一个单一的过程，但是这个阶段同样是可以被流水化或并行化。注意光栅化会寻找图元内部的像素，比如三角形图元。

The rendering speed may be expressed in frames per second (FPS), that is, the

number of images rendered per second. It can also be represented using Hertz (Hz),

which is simply the notation for 1/seconds, i.e., the frequency of update. It is also

common to just state the time, in milliseconds (ms), that it takes to render an image.

The time to generate an image usually varies, depending on the complexity of the

computations performed during each frame. Frames per second is used to express

either the rate for a particular frame, or the average performance over some

duration of use. Hertz is used for hardware, such as a display, which is set to a fixed

rate.

渲染的速度可以用每秒帧率FPS表示，意思是每一秒渲染图像的数量。同样可以使用赫兹表示，简单用符号表示为1/s，即刷新的频率。通常仅仅用毫秒时间来表示一副图片的渲染时间。生成一张图片时间是多变的，依赖于每一帧的复杂计算和执行。帧速率常常用来表示特定帧的速率，或者是持续一段时间内的平均性能。赫兹常常用在硬件，比如显示器，被设定为固定的速率。

As the name implies, the application stage is driven by the application and is

therefore typically implemented in software running on general-purpose CPUs. These

CPUs commonly include multiple cores that are capable of processing multiple

threads of execution in parallel. This enables the CPUs to efficiently run the large

variety of tasks that are the responsibility of the application stage. Some of the tasks

traditionally performed on the CPU include collision detection, global acceleration

algorithms, animation, physics simulation, and many others, depending on the type

of application. The next main stage is geometry processing, which deals with

transforms, projections, and all other types of geometry handling. This stage

computes what is to be drawn, how it should be drawn, and where it should be

drawn. The geometry stage is typically performed on a graphics processing unit

(GPU) that contains many programmable cores as well as fixed-operation hardware.

The rasterization stage typically takes as input three vertices, forming a triangle, and

finds all pixels that are considered inside that triangle, then forwards these to the

next stage. Finally, the pixel processing stage executes a program per pixel to

determine its color and may perform depth testing to see whether it is visible or not.

It may also perform per-pixel operations such as blending the newly computed color

with a previous color. The rasterization and pixel processing stages are also

processed entirely on the GPU. All these stages and their internal pipelines will be

discussed in the next four sections. More details on how the GPU processes these

stages are given in Chapter 3.

顾名思义，应用阶段是被应用程序执行的，因此通常是在运行了CPU的软件上实现。这些CPU通常是有多核组成因此他们有能力去处理并行多线程程序。这使CPU能够高效的运行应用阶段负责的各种任务。通常在CPU上执行的任务包括：碰撞检测，全局加速算法，动画，物理模拟以及其他的任务，取决于应用程序的种类。下一个主要阶段是几何阶段，决定了转换，投影和其他的几何处理。这个阶段计算机会计算绘制什么东西，如何绘制，以及在哪绘制。几何阶段主要是在图形处理器（GPU）中执行的，GPU包含了一些可编程内核和固定操作硬件。光栅化阶段通常把输入的三个顶点形成三角形，并且寻找三角形内部的所有像素，然后传递给下一阶段。最后，像素处理阶段执行一个程序决定每个像素的颜色以及执行深度测试去看这个像素是否需要显示。同样也会对每个像素执行操作比如新的计算颜色与之前的颜色混合。光栅化和像素处理阶段同样是在GPU上完成的。所有的阶段和内部流水线将会在下面4个部分讨论。更多GPU阶段的处理详情第3章节。

2.2 The Application Stage 应用阶段

The developer has full control over what happens in the application stage, since it

usually executes on the CPU. Therefore, the developer can entirely determine the

implementation and can later modify it in order to improve performance. Changes

here can also affect the performance of subsequent stages. For example, an

application stage algorithm or setting could decrease the number of triangles to be

rendered.

因为应用阶段是在CPU中执行的，开发者有全权管理权。因此，开发者可以完全决定其执行并且可以更改以便改善性能。改变也会影响后续阶段。举个例子，一个应用阶段算法或者设定可能减少渲染的三角形数量。

All this said, some application work can be performed by the GPU, using a separate

mode called a compute shader. This mode treats the GPU as a highly parallel general

processor, ignoring its special functionality meant specifically for rendering graphics.

所有这些说，一些应用工作使用一种独立的模式可以被GPU执行，称作计算着色器。这种模式将GPU看作是高效并行通用处理器，忽略了他专门渲染图像的特殊功能。

At the end of the application stage, the geometry to be rendered is fed to the

geometry processing stage. These are the rendering primitives, i.e., points, lines,

and triangles, that might eventually end up on the screen (or whatever output device

is being used). This is the most important task of the application stage.

在应用阶段的结束，要被渲染的几何体将被提交到几何阶段。这些是渲染图元，比如点、线和三角形，这些可能最终会显示在屏幕上（或者其他正在使用的输出设备）。这是应用阶段最重要的任务。

A consequence of the software-based implementation of this stage is that it is not

divided into substages, as are the geometry processing, rasterization, and pixel

processing stages. However, to increase performance, this stage is often executed

in parallel on several processor cores. In CPU design, this is called a superscalar

construction, since it is able to execute several processes at the same time in the

same stage. Section 18.5 presents various methods for using multiple processor

cores.

基于软件执行的结果就是不像几何阶段、光栅化和像素处理阶段，这个阶段并没有划分子阶段。然而为了提升性能，这个阶段经常会在几个处理器核心上并行执行。在CPU设计上，这称为超标量结构，因为他能够在同一阶段同一时间执行多个进程。第18.5章节将介绍使用多核处理器的各种方法。

One process commonly implemented in this stage is collision detection. After a

collision is detected between two objects, a response may be generated and sent

back to the colliding objects, as well as to a force feedback device. The application

stage is also the place to take care of input from other sources, such as the keyboard,

the mouse, or a head-mounted display. Depending on this input, several different

kinds of actions may be taken. Acceleration algorithms, such as particular culling

algorithms (Chapter 19), are also implemented here, along with whatever else the

rest of the pipeline cannot handle.

这个阶段通常要执行的一个过程是碰撞检测，在检测到两个物体之间的碰撞后，会有一种回应产生并返回给碰撞的物体，同样也会返回给力反馈装备。应用阶段也是处理从其他资源接收输入的地方，比如键盘，鼠标，或者是头戴式设备。根据输入，不同的反应会产生。加速算法，比如特殊裁剪算法（第19章），也会在这里实现，以及其他流水线无法处理的其他任何内容。

2.3 Geometry Processing 几何阶段

The geometry processing stage on the GPU is responsible for most of the per-

triangle and per-vertex operations. This stage is further divided into the following

functional stages: vertex shading, projection, clipping, and screen mapping (Figure

2.3).

在GPU上几何阶段主要负责绝大多数的逐三角形和逐顶点操作。这个阶段更深入的分成子功能阶段：顶点着色，投影，裁剪和屏幕映射（图2.3）。

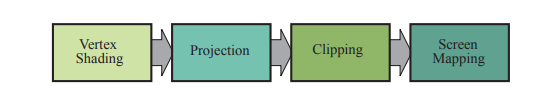


Figure 2.3. The geometry processing stage divided into a pipeline of functional

stages.

图2.3 几何阶段根据阶段功能划分为流水线

2.3.1 Vertex Shading 顶点着色

There are two main tasks of vertex shading, namely, to compute the position for a

vertex and to evaluate whatever the programmer may like to have as vertex output

data, such as a normal and texture coordinates. Traditionally much of the shade of

an object was computed by applying lights to each vertex’s location and normal and

storing only the resulting color at the vertex. These colors were then interpolated

across the triangle. For this reason, this programmable vertex processing unit was

named the vertex shader [1049]. With the advent of the modern GPU, along with

some or all of the shading taking place per pixel, this vertex shading stage is more

general and may not evaluate any shading equations at all, depending on the

programmer’s intent. The vertex shader is now a more general unit dedicated to

setting up the data associated with each vertex. As an example, the vertex shader

can animate an object using the methods in Sections 4.4 and 4.5.

顶点着色有两项大的任务，即，计算顶点的位置以及评估开发者想要的任何顶点输出数据，比如法线和纹理坐标。传统上，一个物体的大部分着色是通过每个顶点的位置和法线应用光线计算的以及仅仅储存在顶点的颜色。这些颜色接下来会在三角形中插值运算。这个部分，可编程的顶点处理单元为称为顶点着色器。随着现代GPU的出现，以及每个像素的部分和全部着色，顶点着色阶段更加普遍以及可能不会有任何的着色方程，这取决于开发者的意愿。顶点着色器现在是一个更加普遍的单元用于设置每个顶点之间的数据联系。比如，在4.4和4.5章节使用方法，顶点着色器可以使物体产生动画。

We start by describing how the vertex position is computed, a set of coordinates that

is always required. On its way to the screen, a model is transformed into several

different spaces or coordinate systems. Originally, a model resides in its own model

space, which simply means that it has not been transformed at all. Each model can

be associated with a model transform so that it can be positioned and oriented. It is

possible to have several model transforms associated with a single model. This

allows several copies (called instances) of the same model to have different locations,

orientations, and sizes in the same scene, without requiring replication of the basic

geometry.

我们开始描述顶点位置是如何计算的，这需要一系列的坐标。在显示到屏幕的过程中，模型会被转换到几个不同的空间或者坐标系统。最开始，模型仅仅处于他自己的模型空间，简单来讲就是他没有被转换。每个模型可以通过模型转换联系起来以便于定位和定向。单个模型也可能有几个模型转换。这允许同一场景同一模型的几个副本（实例）有不同的位置，方向和尺寸，而不需要对基础几何体的复制。

It is the vertices and the normals of the model that are transformed by the model

transform. The coordinates of an object are called model coordinates, and after the

model transform has been applied to these coordinates, the model is said to be

located in world coordinates or in world space. The world space is unique, and after

the models have been transformed with their respective model transforms, all

models exist in this same space.

模型顶点和法线根据模型转化而转换。物体的坐标称为模型坐标，在模型变换应用了这些坐标后，模型可以说成位于世界坐标或者世界空间中。世界空间是唯一的，所有的模型在他们独自的模型转换后，他们就存在同一空间。

As mentioned previously, only the models that the camera (or observer) sees are

rendered. The camera has a location in world space and a direction, which are used

to place and aim the camera. To facilitate projection and clipping, the camera and all

the models are transformed with the view transform. The purpose of the view

transform is to place the camera at the origin and aim it, to make it look in the

direction of the negative z-axis, with the y-axis pointing upward and the x-axis

pointing to the right. We use the −z-axis convention; some texts prefer looking down

the +z-axis. The difference is mostly semantic, as transform between one and the

other is simple. The actual position and direction after the view transform has been

applied are dependent on the underlying application programming interface (API).

The space thus delineated is called camera space, or more commonly, view space or

eye space. An example of the way in which the view transform affects the camera

and the models is shown in Figure 2.4. Both the model transform and the view

transform may be implemented as 4×4 matrices, which is the topic of Chapter 4.

However, it is important to realize that the position and normal of a vertex can be

computed in whatever way the programmer prefers.

正如前面提到的，只有在摄像机（或观察者）视线内的模型才会被渲染。摄像机在世界空间有一个位置和一个方向，用来放置和瞄准摄像机。为了投影和裁剪，摄像机和所有的模型都要转换到视角空间。视图转换的目的为了使摄像机处于原点并指向他，使他看起来是在Z轴负方向，Y轴指向上方，X轴指向右方。我们使用-Z轴约定，有的文章更喜欢使用+Z轴。这种差异主要是语义上的，因为两者之间的转换很简单。试图转换应用后的真正位置和方向依赖于底层应用程序编程接口（API）。这个空间因此被称为摄像机空间，或者更普遍的是视图空间或者视角空间。图2.4显示的例子表示了视图转换应用于摄像机和模型的方式。模型转换和视图转换都可以用4x4矩阵实现，这是第4章的主题。然而，重要的是要意识到顶点的位置和法线是可以用开发者喜欢的任何方式计算。

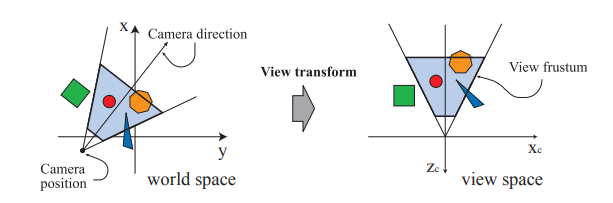


Figure 2.4. In the left illustration, a top-down view shows the camera located and

oriented as the user wants it to be, in a world where the +z-axis is up. The view

transform reorients the world so that the camera is at the origin, looking along its

negative z-axis, with the camera’s +y-axis up, as shown on the right. This is done to

make the clipping and projection operations simpler and faster. The light blue area is

the view volume. Here, perspective viewing is assumed, since the view volume is a

frustum. Similar techniques apply to any kind of projection.

图2.4 在左侧插图中，俯视图显示了用户设置的摄像机的位置和方向，在世界空间，Z轴是向上的。视图转换调整世界空间使摄像机在原点，向下看是负Z轴，并且Y轴正方向是朝上的，如右图所示。这么做会使裁剪和投影操作更简单和快速。蓝色区域是视锥体。这里，假定是透视视角，因为视锥体是平截头体。任何其他类别的投影使用相同的技术。

Next, we describe the second type of output from vertex shading. To produce a

realistic scene, it is not sufficient to render the shape and position of objects, but

their appearance must be modeled as well. This description includes each object’s

material, as well as the effect of any light sources shining on the object. Materials

and lights can be modeled in any number of ways, from simple colors to elaborate

representations of physical descriptions.

接下来，我们描述顶点着色的第二种形式。为了产生真实的场景，仅仅渲染物体的形状和位置是不够的，他们的外观也要被建模。这种描述包括了每个物体的材质，以及光源照射在物体上会产生的效果。材质和光源可以被多种方式建模，从简单的颜色到复杂的物理形式呈现。

This operation of determining the effect of a light on a material is known as shading.

It involves computing a shading equation at various points on the object. Typically,

some of these computations are performed during geometry processing on a model’s

vertices, and others may be performed during per-pixel processing. A variety of

material data can be stored at each vertex, such as the point’s location, a normal, a

color, or any other numerical information that is needed to evaluate the shading

equation. Vertex shading results (which can be colors, vectors, texture coordinates,

along with any other kind of shading data) are then sent to the rasterization and

pixel processing stages to be interpolated and used to compute the shading of the

surface.

这种决定光源在材质上的效果操作称为着色。他包含了计算物体上各种各样的顶点着色方程。特别的，一些计算是在模型顶点的几何阶段执行，其他也可以在逐像素过程执行。每个顶点可以储存各种各样的材质数据，比如点的位置，法线，颜色，或者一些其他的着色方程所需要的数值信息。顶点着色结果（可以是颜色、向量、纹理坐标，甚至一些其他类型的着色数据）接下来会传送到光栅化和像素处理阶段进行插值运算和计算物体表面的着色。

Vertex shading in the form of the GPU vertex shader is discussed in more depth

throughout this book and most specifically in Chapters 3 and 5.

GPU顶点着色器的顶点着色形式更深入的讨论将贯穿整本书，并且第3章和第5章会给出更多明确的信息。

As part of vertex shading, rendering systems perform projection and then clipping,

which transforms the view volume into a unit cube with its extreme points at (−1,

−1, −1) and (1, 1, 1). Different ranges defining the same volume can and are used,

for example, 0 ≤ z ≤ 1. The unit cube is called the canonical view volume. Projection

is done first, and on the GPU it is done by the vertex shader. There are two

commonly used projection methods, namely orthographic (also called parallel) and

perspective projection. See Figure 2.5. In truth, orthographic is just one type of

parallel projection. Several others find use, particularly in the field of architecture,

such as oblique and axonometric projections. The old arcade game Zaxxon is named

from the latter.

作为顶点着色的一部分，渲染系统执行投影然后裁剪，转换视锥体到极值为t (−1, −1, −1) 和(1, 1, 1)的单元立方体。不同的范围适用于同样的视锥体，例如0 ≤ z ≤ 1。单元立方体被称为正规化可视空间。首先进行的是投影，这是在GPU顶点着色阶段完成的。共有两种投影方法，即正交视透（也称为平行投影）和透视投影。如图2.5,。实际上，正交投影只是平行投影的一种。也有其他的投影，特别是建筑行业，比如斜视投影和轴向投影。老街机游戏Zaxxon就是以后者命名的。

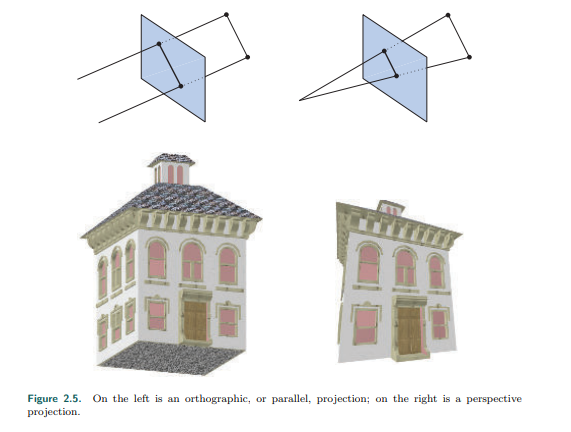


图2.5 左侧是正交投影或平行投影。右侧是透视投影。

Note that projection is expressed as a matrix (Section 4.7) and so it may sometimes

be concatenated with the rest of the geometry transform.

注意投影表示为一个矩阵（章节4.7），因此他常常和几何阶段的剩余部分串联一起。

The view volume of orthographic viewing is normally a rectangular box, and the

orthographic projection transforms this view volume into the unit cube. The main

characteristic of orthographic projection is that parallel lines remain parallel after the

transform. This transformation is a combination of a translation and a scaling.

正交视透的视锥体通常是一个矩形，正交视透投影转换这个视锥体为单元正方体。最主要的特点是正交投影转换之后平行线仍然是平行线。这个转变综合了转换和缩放。

The perspective projection is a bit more complex. In this type of projection, the

farther away an object lies from the camera, the smaller it appears after projection.

In addition, parallel lines may converge at the horizon. The perspective transform

thus mimics the way we perceive objects’ size. Geometrically, the view volume,

called a frustum, is a truncated pyramid with rectangular base. The frustum is

transformed into the unit cube as well. Both orthographic and perspective transforms

can be constructed with 4 × 4 matrices (Chapter 4), and after either transform, the

models are said to be in clip coordinates. These are in fact homogeneous coordinates,

discussed in Chapter 4, and so this occurs before division by w. The GPU’s vertex

shader must always output coordinates of this type in order for the next functional

stage, clipping, to work correctly.

透视投影略微有些复杂。在这种方式的投影下，物体离摄像机越远，投影后，看起来就越小。另外，水平的平行线有可能会相交。透视变换因此模仿了我们感知物体尺寸的方式。几何上，这个视锥体，称为平截头体，是一个切了头的金字塔附带一个矩形基础。平截头体也要转换进单元立方体。正交视透和透视投影转换都可以由4x4矩阵构造，在任意一个转换之后，模型坐标被称为裁剪坐标。他们实际上是齐次坐标，会在第4章讨论，这出现在除以w之前。GPU顶点着色器必须输出这种形式的坐标，为了下一个功能阶段，裁剪能够正确工作。

Although these matrices transform one volume into another, they are called

projections because after display, the z-coordinate is not stored in the image

generated but is stored in a z-buffer, described in Section 2.5. In this way, the

models are projected from three to two dimensions.

虽然这些矩阵变换让一个体积成为了另一个，这称为投影，因为在显示之后，z坐标并没有储存在生成的图像中，而是储存在z缓冲区，将在2.5章节介绍。这种方式模型从三维投影成了两维。

2.3.2 Optional Vertex Processing 可选的顶点处理

Every pipeline has the vertex processing just described. Once this processing is done,

there are a few optional stages that can take place on the GPU, in this order:

tessellation, geometry shading, and stream output. Their use depends both on the

capabilities of the hardware—not all GPUs have them—and the desires of the

programmer. They are independent of each other, and in general they are not

commonly used. More will be said about each in Chapter 3.

每个流水管线都有刚刚描述的顶点处理。顶点处理结束后，GPU上会有几个可选阶段按以下顺序执行：镶嵌、几何着色和流输出。这些阶段依赖于硬件的能力——并不都是GPU处理——也取决于开发者的意愿。他们是互相独立的，通常很少用到。在第3章会有更多的介绍。

The first optional stage is tessellation. Imagine you have a bouncing ball object. If

you represent it with a single set of triangles, you can run into problems with quality

or performance. Your ball may look good from 5 meters away, but up close the

individual triangles, especially along the silhouette, become visible. If you make the

ball with more triangles to improve quality, you may waste considerable processing

time and memory when the ball is far away and covers only a few pixels on the

screen. With tessellation, a curved surface can be generated with an appropriate

number of triangles.

第一个可选阶段是镶嵌。假设你有一个弹跳的球物体。如果你用一组三角形表现他的时候，可能会遇到质量或者性能问题。球在5m开外看起来还不错，但是近距离看，尤其是沿着轮廓，能看到独立的三角形。如果你用更多的三角形去改善质量，但是当球距离很远的时候，仅仅只覆盖了屏幕上的几个像素，那么将会浪费了大量的处理时间和内存。通过镶嵌，可以用适量的三角形生成一个曲面。

We have talked a bit about triangles, but up to this point in the pipeline we have just

processed vertices. These could be used to represent points, lines, triangles, or other

objects. Vertices can be used to describe a curved surface, such as a ball. Such

surfaces can be specified by a set of patches, and each patch is made of a set of

vertices. The tessellation stage consists of a series of stages itself—hull shader,

tessellator, and domain shader—that converts these sets of patch vertices into

(normally) larger sets of vertices that are then used to make new sets of triangles.

The camera for the scene can be used to determine how many triangles are

generated: many when the patch is close, few when it is far away.

我们已经讨论了一点三角形，但是到目前为止，流水管线我们仅仅处理了顶点。这些也可以用来表示点、线、三角形或者其他的东西。顶点也可以用来描述一个曲面，比如一个球。这样的曲面可以由一系列的斑块表示，并且每个斑块由一系列的顶点组成。镶嵌阶段本身包含了一系列的子阶段——外壳着色器，镶嵌器，和域着色器——这些阶段将这一系列的斑块顶点转换成更到系列的顶点，然后用于创建新的三角形集。场景的摄像机决定了产生多少的三角形，当离得很近的时候会产生很多三角形，离的很远的时候就产生较少的三角形。

The next optional stage is the geometry shader. This shader predates the tessellation

shader and so is more commonly found on GPUs. It is like the tessellation shader in

that it takes in primitives of various sorts and can produce new vertices. It is a much

simpler stage in that this creation is limited in scope and the types of output

primitives are much more limited. Geometry shaders have several uses, with one of

the most popular being particle generation. Imagine simulating a fireworks explosion.

Each fireball could be represented by a point, a single vertex. The geometry shader

can take each point and turn it into a square (made of two triangles) that faces the

viewer and covers several pixels, so providing a more convincing primitive for us to

shade.

下一个可选阶段是几何着色器。这个着色器在镶嵌着色器更早之前出现，因此在GPU上更常见。它就像镶嵌着色器，接受各种各样的片元并产生新的顶点。这是一个简单的多的阶段，因为这种创建受限于视野，输出的片元受到更多的限制。几何着色器有几个用途，其中最流行的应用是粒子生成器。想象一下模拟烟花爆炸。每一个火球都可以由一个点、一个顶点表示。几何着色器可以将每一个点转换成一个正方形（由两个三角形组成），并且面向观察者，覆盖几个像素，因此提供了更让人更信服的片元去着色。

The last optional stage is called stream output. This stage lets us use the GPU as a

geometry engine. Instead of sending our processed vertices down the rest of the

pipeline to be rendered to the screen, at this point we can optionally output these to

an array for further processing. These data can be used by the CPU, or the GPU itself,

in a later pass. This stage is typically used for particle simulations, such as our

fireworks example.

最后一个可选阶段称为流输出。这个阶段让我们把GPU当做一个几何引擎来用。处理过的顶点可以可选用的输出到一个数组去做更多的处理，而不是直接传送到接下来的流水管线去渲染在屏幕上。在接下来的过程，这些数据可以被CPU使用，也可以被GPU自己使用。这个阶段通常用在粒子模拟，比如烟花的例子。

These three stages are performed in this order—tessellation, geometry shading, and

stream output—and each is optional. Regardless of which (if any) options are used, if

we continue down the pipeline we have a set of vertices with homogeneous

coordinates that will be checked for whether the camera views them.

这三个阶段的执行顺序是——镶嵌，几何着色，和流输出——每个阶段都是可选的。不管我们用的哪些可选阶段，如果我们继续沿着流水管线，我们将有一系列的齐次坐标顶点要被检查是否摄像机可以看到他们。

2.3.3 Clipping 裁剪

Only the primitives wholly or partially inside the view volume need to be passed on

to the rasterization stage (and the subsequent pixel processing stage), which then

draws them on the screen. A primitive that lies fully inside the view volume will be

passed on to the next stage as is. Primitives entirely outside the view volume are not

passed on further, since they are not rendered. It is the primitives that are partially

inside the view volume that require clipping. For example, a line that has one vertex

outside and one inside the view volume should be clipped against the view volume,

so that the vertex that is outside is replaced by a new vertex that is located at the

intersection between the line and the view volume. The use of a projection matrix

means that the transformed primitives are clipped against the unit cube. The

advantage of performing the view transformation and projection before clipping is

that it makes the clipping problem consistent; primitives are always clipped against

the unit cube.

只有片元的整体或者在视锥体中的部分才会传送到光栅化阶段（以及随后的像素处理阶段），然后绘制在屏幕上。片元完全处于视锥体中时将会传递到下一个阶段。片元完全处于视锥体外部的将不会传递到接下来的阶段，因为他们不被渲染。片元部分位于视锥体内的需要被裁剪。举个例子，一条线有一个顶点在视锥体外部，一个顶点在视锥体内部，这条线应该根据视锥体进行裁剪，外部的顶点会被线与视锥体的相交处的新顶点代替。使用投影矩阵意味着转换后的图元将被单元立方体进行裁剪。在剪裁之前执行视图转换和投影的优点在于，他使裁剪问题得到了统一:图元总是被单元立方体裁剪。

The clipping process is depicted in Figure 2.6. In addition to the six clipping planes of

the view volume, the user can define additional clipping planes to visibly chop

objects. An image showing this type of visualization, called sectioning, is shown in

Figure 19.1 on page 818.

图2.6描述了裁剪过程。除了视锥体的6个裁剪平面以外，用户可以自定义额外的裁剪屏幕去切割物体。818页的图19.1展示了这种类型的可视化，称为分割。

The clipping step uses the 4-value homogeneous coordinates produced by projection

to perform clipping. Values do not normally interpolate linearly across a triangle in

perspective space. The fourth coordinate is needed so that data are properly

interpolated and clipped when a perspective projection is used. Finally, perspective

division is performed, which places the resulting triangles’ positions into three-

dimensional normalized device coordinates. As mentioned earlier, this view volume

ranges from (−1, −1, −1) to (1, 1, 1). The last step in the geometry stage is to

convert from this space to window coordinates.

裁剪过程使用了由投影产生了4值齐次坐标进行裁剪。在透视空间值通常不能对三角形进行线性插值。当透视投影使用后，第四个坐标是必须的以便数据可以能够正确的插值和裁剪。最后，执行透视分割，将得到的三角形放置在三维归一化设备坐标。正如之前提到的，视锥体范围是(−1, −1, −1) 到 (1, 1, 1)。几何阶段的最后阶段是把这个空间转换到窗口坐标。

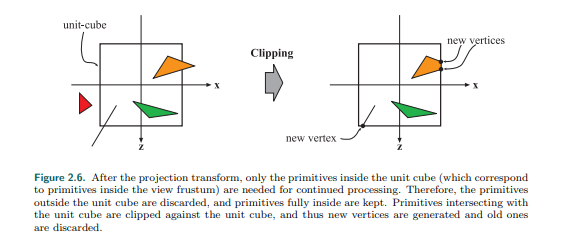


图2.6 在投影转换之后，只有在单位立方体内的图元（对应视同中的图元）才需要继续处理。因此，单元立方体外部的图元会被丢弃，图元完全在内部的被保留。和单位立方体相交的图元会根据单元立方体裁剪，会生成新的顶点，旧的顶点被丢弃。

2.3.4 Screen Mapping 屏幕映射

Only the (clipped) primitives inside the view volume are passed on to the screen

mapping stage, and the coordinates are still three-dimensional when entering this

stage. The x- and y-coordinates of each primitive are transformed to form screen

coordinates. Screen coordinates together with the z-coordinates are also called

window coordinates. Assume that the scene should be rendered into a window with

the minimum corner at (x1, y1) and the maximum corner at (x2, y2), where x1 < x2

and y1 < y2. Then the screen mapping is a translation followed by a scaling

operation. The new x- and y-coordinates are said to be screen coordinates. The z-

coordinate ([−1, +1] for OpenGL and [0, 1] for DirectX) is also mapped to [z1, z2],

with z1 = 0 and z2 = 1 as the default values. These can be changed with the API,

however. The window coordinates along with this remapped z-value are passed on

to the rasterizer stage. The screen mapping process is depicted in Figure 2.7.

只有（裁剪过的）视锥体内部的图元才会传到屏幕映射阶段，进入这个阶段的时候，坐标仍然是三维的。每个图元的x,y轴会被转换成屏幕坐标格式。屏幕坐标和z轴一起也称为窗口坐标。假设场景要被渲染到窗口，他的最小点是（x1,y2），最大点是（x2,y2），其中x1<x2，y1<y2。屏幕映射是缩放操作的一个转换。新的x,y坐标被称为屏幕坐标。Z坐标（OpenGL的是[−1, +1] ， DirectX的是[0, 1]），同样被映射为[z1, z2]，z1 = 0和z2 = 1是默认值。但是这些可以通过API进行更改。窗口坐标以及重新映射的z值会一起传递到光栅化阶段。图2.7描述了屏幕映射的过程。

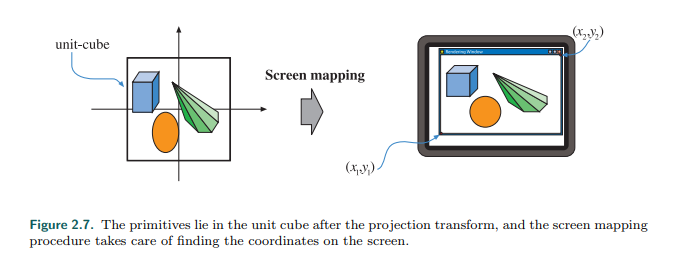


图2.7 单元立方体内的图元在经过了投影转换之后，屏幕映射程序复杂找到他们的屏幕坐标。

Next, we describe how integer and floating point values relate to pixels (and texture

coordinates). Given a horizontal array of pixels and using Cartesian coordinates, the

left edge of the leftmost pixel is 0.0 in floating point coordinates. OpenGL has always

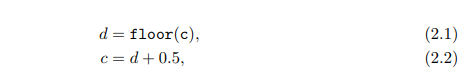
used this scheme, and DirectX 10 and its successors use it. The center of this pixel is

at 0.5. So, a range of pixels [0, 9] cover a span from [0.0, 10.0). The conversions

are simply where d is the discrete (integer) index of the pixel and c is the continuous

(floating point) value within the pixel.

接下来，我们将描述整形和浮点数据如何和像素（和纹理坐标）联系起来。用笛卡尔坐标系构建一个像素的水平数组，最左边像素的左边缘是浮点坐标的0.0。OpenGL一直使用这种方案，DirectX 10和后续版本也会用到这个方案。像素的中心点坐标为0.5。因此像素[0, 9]覆盖范围是坐标[0.0, 10.0)。这种转换是简单的，d是像素的离散（整形）下标，c是像素的连续的（浮点）坐标。



While all APIs have pixel location values that increase going from left to right, the

location of zero for the top and bottom edges is inconsistent in some cases between

OpenGL and DirectX. OpenGL favors the Cartesian system throughout, treating the

lower left corner as the lowest-valued element, while DirectX sometimes defines the

upper left corner as this element, depending on the context. There is a logic to each,

and no right answer exists where they differ. As an example, (0, 0) is located at the

lower left corner of an image in OpenGL, while it is upper left for DirectX. This

difference is important to take into account when moving from one API to the other.

尽管所有的API都规定像素位置从左到右增加，在OpenGL和DirectX之间，同样情况下，顶部和底部的0位置会有些不同。OpenGL从始至终都使用笛卡尔坐标系统，将左下角作为最下元素，而DirectX有时候会根据上下文定义左上角为最小元素。每一个都很有逻辑，尽管他们不同，但是并没有对错之分。举个例子：在OpenGL 中，(0, 0)位于图像的左下角，而DirectX，他是位于图像的右上角。当从一个API移植到另一个API时，考虑到这种差异会非常重要。

2.4 Rasterization 光栅化

Given the transformed and projected vertices with their associated shading data (all

from geometry processing), the goal of the next stage is to find all pixels—short for

picture elements—that are inside the primitive, e.g., a triangle, being rendered. We

call this process rasterization, and it is split up into two functional substages: triangle

setup (also called primitive assembly) and triangle traversal. These are shown to the

left in Figure 2.8. Note that these can handle points and lines as well, but since

triangles are most common, the substages have “triangle” in their names.

Rasterization, also called scan conversion, is thus the conversion from two-

dimensional vertices in screen space—each with a z-value (depth value) and various

shading information associated with each vertex—into pixels on the screen.

Rasterization can also be thought of as a synchronization point between geometry

processing and pixel processing, since it is here that triangles are formed from three

vertices and eventually sent down to pixel processing.

给定转换过和投影过的顶点以及他们相关联的着色数据（都来自几何处理阶段），下一阶段的目标是找到所有的像素——图片元素的简称——他们在图元内部，比如三角形，并开始被渲染。我们称这个处理为光栅化，并且他可以被划分为两个子功能阶段：三角形设置（也被称为图元装配）和三角形遍历。如图2.8所示。注意这里也可以处理点和线，但是由于三角形更为常见，因此子阶段名字里含有三角形。光栅化也称为扫描转换，因为他将二维屏幕顶点——每个都有z值（深度值）并且各种关联每个顶点的着色信息——转换成屏幕上的像素。光栅化也可以认为是几何阶段和像素处理阶段的同步点，因为在这里，三个顶点形成三角形并接下来传递给像素处理阶段。

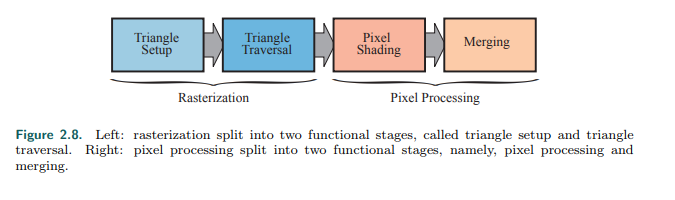


图2.8 左侧：光栅化划分为两个子阶段，称为三角形设置和三角形遍历。

右侧：像素处理阶段划分为两个子功能阶段，称为像素处理和合并

Whether the triangle is considered to overlap the pixel depends on how you have set

up the GPU’s pipeline. For example, you may use point sampling to determine

“insideness.” The simplest case uses a single point sample in the center of each pixel,

and so if that center point is inside the triangle then the corresponding pixel is

considered inside the triangle as well. You may also use more than one sample per

pixel using supersampling or multisampling antialiasing techniques (Section 5.4.2).

Yet another way is to use conservative rasterization, where the definition is that a

pixel is “inside” the triangle if at least part of the pixel overlaps with the triangle

(Section 23.1.2).

三角形是否是被认为覆盖了像素取决于你如何设置GPU的流水管线。举个例子，你可以使用点取样去决定“内在性”。最简单的情况是使用每个像素的中心点取样，所以如果那个中心点在三角形内部那么相应的像素也被认为是在三角形的内部。你也可以使用超级采样或者多重采样抗锯齿技术（章节5.4.2）来使每个像素采取多个样本。还有另外一种方式是使用保守光栅化，定义可像素至少一部分覆盖了三角形那么这个三角形就是在三角形内部（章节23.1.2）。

2.4.1 Triangle Setup 三角形设置

In this stage the differentials, edge equations, and other data for the triangle are

computed. These data may be used for triangle traversal (Section 2.4.2), as well as

for interpolation of the various shading data produced by the geometry stage. Fixed-

function hardware is used for this task.

在这个阶段计算了三角形的微分，边缘方程和其他的数据。这些数据也会被三角形遍历用到，也用到了几何阶段生成的各种着色数据的插值。固定功能硬件用于此任务。

2.4.2 Triangle Traversal 三角形遍历

Here is where each pixel that has its center (or a sample) covered by the triangle is

checked and a fragment generated for the part of the pixel that overlaps the triangle.

More elaborate sampling methods can be found in Section 5.4. Finding which

samples or pixels are inside a triangle is often called triangle traversal. Each triangle

fragment’s properties are generated using data interpolated among the three triangle

vertices (Chapter 5). These properties include the fragment’s depth, as well as any

shading data from the geometry stage. McCormack et al. [1162] offer more

information on triangle traversal. It is also here that perspective-correct interpolation

over the triangles is performed [694] (Section 23.1.1). All pixels or samples that are

inside a primitive are then sent to the pixel processing stage, described next.

这里检查每个被三角形覆盖了中心的像素，并且与三角形重叠的部分生成一个片段。更复杂的采样方法详见章节5.4。寻找三角形内部的样本或者像素称为三角形遍历。每个三角形片段属性都是使用三个三角形顶点插值数据生成的（第5章）。这些属性包括了片段的深度，和几何阶段的任何着色数据。McCormack等[1162]提供了更多的三角形遍历的信息。也正是在这里，执行了三角形上的正确透视插值[694]（章节23.1.1）。图元内部所有的像素或样本传送到了像素处理阶段，接下来进行描述。

2.5 Pixel Processing 像素处理阶段

At this point, all the pixels that are considered inside a triangle or other primitive

have been found as a consequence of the combination of all the previous stages.

The pixel processing stage is divided into pixel shading and merging, shown to the

right in Figure 2.8. Pixel processing is the stage where per-pixel or per-sample

computations and operations are performed on pixels or samples that are inside a

primitive.

此时，综合之前所有阶段的结果，三角形或者其他图元内部的所有像素都被找到了。像素处理阶段分为像素着色和合并，如图2.8所示。像素处理阶段是图元内部像素或样本执行逐像素或逐样本计算和操作的阶段。

2.5.1 Pixel Shading 像素着色

Any per-pixel shading computations are performed here, using the interpolated

shading data as input. The end result is one or more colors to be passed on to the

next stage. Unlike the triangle setup and traversal stages, which are usually

performed by dedicated, hardwired silicon, the pixel shading stage is executed by

programmable GPU cores. To that end, the programmer supplies a program for the

pixel shader (or fragment shader, as it is known in OpenGL), which can contain any

desired computations. A large variety of techniques can be employed here, one of

the most important of which is texturing. Texturing is treated in more detail in

Chapter 6. Simply put, texturing an object means “gluing” one or more images onto

that object, for a variety of purposes. A simple example of this process is depicted in

Figure 2.9. The image may be one-, two-, or three-dimensional, with two-

dimensional images being the most common. At its simplest, the end product is a

color value for each fragment, and these are passed on to the next substage。

使用插值的着色数据作为输入，任何逐像素计算都在这里执行。执行的结果就是一种或多种颜色传递到下一个阶段。与三角形设置和遍历阶段不同的是，三角形设置和遍历阶段通常是固定执行的，像素着色阶段则是由可编程的GPU核心执行的。为了那个目的，程序员提供一个像素着色器程序（或者片段着色器，OpenGL中所述），里面包含了任何期望的计算。大量的技术在这里被使用，最重要的其中之一是贴纹理。纹理详情第6章。简单的讲，纹理对象就是贴一张或更多的图像在物体上，为了各种目的。图2.9一个简单的例子描述了这个过程。图像可以是一维，二维或者三维，二维是最常见的。最简单的，最终产品是每个片段的颜色值，这些数据将被传递到下一个子阶段。

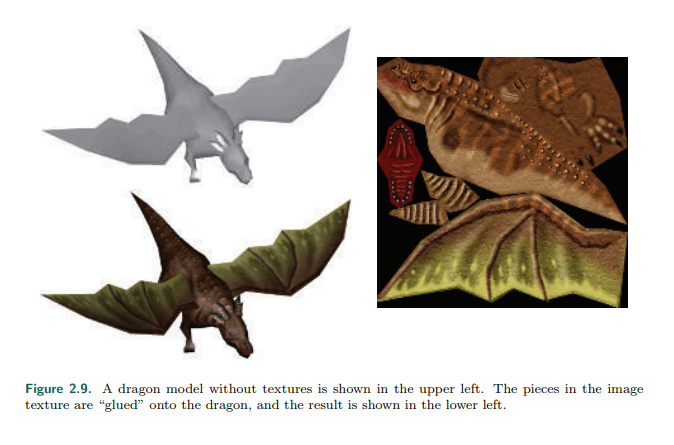


图2.9 一个没有纹理的龙模型在左上方所示。一个图像纹理贴在了龙上，结果如左下角所示。

2.5.2 Merging 合并

The information for each pixel is stored in the color buffer, which is a rectangular

array of colors (a red, a green, and a blue component for each color). It is the

responsibility of the merging stage to combine the fragment color produced by the

pixel shading stage with the color currently stored in the buffer. This stage is also

called ROP, standing for “raster operations (pipeline)” or “render output unit,”

depending on who you ask. Unlike the shading stage, the GPU subunit that performs

this stage is typically not fully programmable. However, it is highly configurable,

enabling various effects.

每个像素的信息储存在颜色缓冲区，颜色的矩形数组（每个颜色由红、绿、蓝分量组成）。合并阶段的职责就是合并由像素着色阶段产生的片元颜色和已经储存在颜色缓冲区的颜色。这个阶段也称为ROP，表示“光栅操作（流水线）”或“渲染输出单元”，依赖于你询问的对象。不像着色阶段，执行这个阶段的GPU子单元并不完全可编程的，但是是高度可配置的，支持各种特效。

This stage is also responsible for resolving visibility. This means that when the whole

scene has been rendered, the color buffer should contain the colors of the primitives

in the scene that are visible from the point of view of the camera. For most or even

all graphics hardware, this is done with the z-buffer (also called depth buffer)

algorithm [238]. A z-buffer is the same size and shape as the color buffer, and for

each pixel it stores the z-value to the currently closest primitive. This means that

when a primitive is being rendered to a certain pixel, the z-value on that primitive at

that pixel is being computed and compared to the contents of the z-buffer at the

same pixel. If the new z-value is smaller than the z-value in the z-buffer, then the

primitive that is being rendered is closer to the camera than the primitive that was

previously closest to the camera at that pixel. Therefore, the z-value and the color of

that pixel are updated with the z-value and color from the primitive that is being

drawn. If the computed z-value is greater than the z-value in the z-buffer, then the

color buffer and the z-buffer are left untouched. The z-buffer algorithm is simple, has

O(n) convergence (where n is the number of primitives being rendered), and works

for any drawing primitive for which a z-value can be computed for each (relevant)

pixel. Also note that this algorithm allows most primitives to be rendered in any order,

which is another reason for its popularity. However, the z-buffer stores only a single

depth at each point on the screen, so it cannot be used for partially transparent

primitives. These must be rendered after all opaque primitives, and in back-to-front

order, or using a separate order-independent algorithm (Section 5.5). Transparency

is one of the major weaknesses of the basic z-buffer.

这个阶段还负责解决可视化问题。那意味着整个场景已经被渲染了，颜色缓冲区应该包含了从摄像机视角观看可见的场景中图元的颜色。对于大部分甚至所有的图像硬件来说，这是使用了z缓冲（也称为深度缓冲）算法[238]完成的。一个z缓冲和颜色缓冲具有同样大小和形状，为每个像素储存了当前最近图元的z值。这意味着当一个图元将要被渲染到某一像素时，图元位于该像素上的z值将被计算并和z缓冲区中同一像素的z值进行比较。如果新的z值比z缓冲区中的z值更小，那么在那个像素上，这个将要被渲染的图元将要比之前离摄像机最近的图元要更加靠近摄像机。因此，像素上的z值和颜色将会更新为将要被绘制图元的z值的颜色。如果计算的z值比z缓冲区中的z值更大，那么颜色缓冲区和z缓冲区保持不变。z缓冲算法很简单，具有O(n)的复杂度（n是要被渲染图元的数量），并为任何可计算每（相关）像素z值的绘制图元工作。同时注意，这个算法允许大部分图元以任何顺序渲染，这也是他备受欢迎的另一个原因。然而，z缓冲仅仅储存了屏幕上每个店的一个深度值，因此他不能用在部分透明的图元。这些必须在所有不透明的图元之后进行渲染，并且是从后向前的顺序，或者是使用单独的顺序无关的算法（章节5.5）。透明度是基本z缓冲的主要缺点之一。

We have mentioned that the color buffer is used to store colors and that the z-buffer

stores z-values for each pixel. However, there are other channels and buffers that

can be used to filter and capture fragment information. The alpha channel is

associated with the color buffer and stores a related opacity value for each pixel

(Section 5.5). In older APIs, the alpha channel was also used to discard pixels

selectively via the alpha test feature. Nowadays a discard operation can be inserted

into the pixel shader program and any type of computation can be used to trigger a

discard. This type of test can be used to ensure that fully transparent fragments do

not affect the z-buffer (Section 6.6).

我们已经提到颜色缓冲区是用来储存每个像素的颜色，z缓冲区储存z值。然而，还有其他的通道和缓冲区用来过滤和捕捉片元信息。Alpha通道与颜色缓冲区有紧密联系，储存了每个像素的相关的不透明值（章节5.5）。在早些的API中，alpha通道通常用来通过alpha测试特性去有选择性的去除像素。现在，丢弃操作可以插入到像素着色器程序中，任何类型的计算都可以用来触发丢弃操作。这种类型的测试可以用来确保完全透明的片元不会影响z缓冲区（章节6.6）。

The stencil buffer is an offscreen buffer used to record the locations of the rendered

primitive. It typically contains 8 bits per pixel. Primitives can be rendered into the

stencil buffer using various functions, and the buffer’s contents can then be used to

control rendering into the color buffer and z-buffer. As an example, assume that a

filled circle has been drawn into the stencil buffer. This can be combined with an

operator that allows rendering of subsequent primitives into the color buffer only

where the circle is present. The stencil buffer can be a powerful tool for generating

some special effects. All these functions at the end of the pipeline are called raster

operations (ROP) or blend operations. It is possible to mix the color currently in the

color buffer with the color of the pixel being processed inside a triangle. This can

enable effects such as transparency or the accumulation of color samples. As

mentioned, blending is typically configurable using the API and not fully

programmable. However, some APIs have support for raster order views, also called

pixel shader ordering, which enable programmable blending capabilities.

模板缓冲区是屏幕之外的缓冲区，用来记录渲染图元的位置。每个像素通常包含8位。图元可以被通过各种函数渲染进模板缓冲区，缓冲区中的内容可以被用来控制颜色缓冲区和z缓冲区的渲染。举个例子，假设模板缓冲区已经画了一个填充的圆。他可以联合操作只允许圆形区域的颜色缓冲区渲染接下来的图元。模板缓冲区是生成一些特效的强大工具。流水管线末端的所有这些函数称为光栅操作（ROP）或者混合操作。也是可能的，混合颜色缓冲区的当前颜色和当前正在处理的三角形内部像素的颜色。这可以实现一些效果比如透明度或者颜色样本的积累。正如提到的，混合是可以使用API高度可配置的并不完全可编程。然而，一些API支持光栅顺序视图，也称为像素着色顺序，他支持了可编程的混合功能。

The framebuffer generally consists of all the buffers on a system.

帧缓冲通常是由系统中所有的缓冲组成。

When the primitives have reached and passed the rasterizer stage, those that are

visible from the point of view of the camera are displayed on screen. The screen

displays the contents of the color buffer. To avoid allowing the human viewer to see

the primitives as they are being rasterized and sent to the screen, double buffering is

used. This means that the rendering of a scene takes place off screen, in a back

buffer. Once the scene has been rendered in the back buffer, the contents of the

back buffer are swapped with the contents of the front buffer that was previously

displayed on the screen. The swapping often occurs during vertical retrace, a time

when it is safe to do so.

当图元到达并通过了光栅化阶段，代表他们从摄像机的角度看是可见的，可以显示在屏幕上的。屏幕将呈现出颜色缓冲区上面的颜色。为了避免人们视角看到图元光栅化和传送到屏幕上的过程，双缓冲区是必须的。那意味着场景的渲染发生在屏幕之外，是在后置缓冲区。一旦场景在后置缓冲区上渲染结束，那么后置缓冲区上的内容将会替换前置缓冲区的内容，就是之前在屏幕上看到的东西。交换是发生在垂直追踪期间，这个时间这么做是安全的。

For more information on different buffers and buffering methods, see Sections 5.4.2,

23.6, and 23.7.

更多不同缓冲区和缓冲方法的信息，请看章节5.4.2,23.6和23.7。

2.6 Through the Pipeline 通过流水管线

Points, lines, and triangles are the rendering primitives from which a model or an

object is built. Imagine that the application is an interactive computer aided design

(CAD) application, and that the user is examining a design for a waffle maker. Here

we will follow this model through the entire graphics rendering pipeline, consisting of

the four major stages: application, geometry, rasterization, and pixel processing. The

scene is rendered with perspective into a window on the screen. In this simple

example, the waffle maker model includes both lines (to show the edges of parts)

and triangles (to show the surfaces). The waffle maker has a lid that can be opened.

Some of the triangles are textured by a two-dimensional image with the

manufacturer’s logo. For this example, surface shading is computed completely in

the geometry stage, except for application of the texture, which occurs in the

rasterization stage.

点、线和三角形构成了模型或者对象的渲染图元。想象一下，应用程序是交互计算机设计（CAD）程序，用户正在检查华夫饼机的设计。这里，我们跟着这个模型通过完整的图形渲染管线，由四个阶段组成：应用阶段，几何阶段，光栅化阶段和像素处理阶段。场景通过透视被渲染进了屏幕窗口。在这个简单的例子中，华夫饼机的制作模型包含了两条线（去显示零件的边缘）和三角形（显示表面）。华夫饼机有一个可以打开的盖子。其中一些三角形由带有工厂logo的二维图像构成纹理。在这个例子中，除了应用纹理是由光栅化阶段完成，表面着色全部都是在几何阶段计算。

Application 应用阶段

CAD applications allow the user to select and move parts of the model. For example,

the user might select the lid and then move the mouse to open it. The application

stage must translate the mouse move to a corresponding rotation matrix, then see to

it that this matrix is properly applied to the lid when it is rendered. Another example:

An animation is played that moves the camera along a predefined path to show the

waffle maker from different views. The camera parameters, such as position and

view direction, must then be updated by the application, dependent upon time. For

each frame to be rendered, the application stage feeds the camera position, lighting,

and primitives of the model to the next major stage in the pipeline—the geometry

stage.

CAD应用程序允许用户选择并移动模型的零件。举个例子，用户可以选择顶盖然后移动鼠标打开他。应用阶段必须转换鼠标的移动为相应的旋转矩阵，然后确保顶盖在渲染时候矩阵能够正确的应用上。另一个例子：有一个动画是播放按预定义的路径移动摄像机去显示不同视角的华夫饼机。摄像机的参数，比如位置，视角方向，必须在应用阶段根据时间更新。为了渲染每一帧，应用阶段将摄像机位置，光源和模型图元数据提供给流水线的下一个主要阶段——几何阶段。

Geometry Processing 几何阶段

For perspective viewing, we assume here that the application has supplied a

projection matrix. Also, for each object, the application has computed a matrix that

describes both the view transform and the location and orientation of the object in

itself. In our example, the waffle maker’s base would have one matrix, the lid

another. In the geometry stage the vertices and normals of the object are

transformed with this matrix, putting the object into view space. Then shading or

other calculations at the vertices may be computed, using material and light source

properties. Projection is then performed using a separate user-supplied projection

matrix, transforming the object into a unit cube’s space that represents what the eye

sees. All primitives outside the cube are discarded. All primitives intersecting this unit

cube are clipped against the cube in order to obtain a set of primitives that lies

entirely inside the unit cube. The vertices then are mapped into the window on the

screen. After all these per-triangle and per-vertex operations have been performed,

the resulting data are passed on to the rasterization stage.

对于透视视图，我们假设应用阶段提供了一个投影矩阵。同样，对于每一个对象，应用阶段都会计算一个矩阵描述了视图转换和物体本身的位置和方向。在我们例子中，华夫饼机基础有一个矩阵，顶盖有另一个。在几何阶段，物体的顶点和法线会被矩阵转换，把物体放置在视图空间。接着着色或者顶点的其他运算会被计算，包括材质和光源属性。然后使用一个单独的用户提供投影矩阵进行投影，将物体转换进我们眼睛看得到的单元立方体。所有的立方体之外的图元将被丢弃。所有的与单元立方体相交的图元会被裁剪，为了得到一系列的图元完全在立方体之内。顶点接着映射到屏幕的窗口上。在所有的逐三角和逐顶点操作执行后，数据结果会被传送到光栅化阶段。

Rasterization 光栅化

All the primitives that survive clipping in the previous stage are then rasterized,

which means that all pixels that are inside a primitive are found and sent further

down the pipeline to pixel processing.

所有经历过之前阶段裁剪的保留下来的图元接着被光栅化，意味着，找到图元内的所有像素，然后传送到流水管线更深处——像素处理阶段。

Pixel Processing 像素处理阶段

The goal here is to compute the color of each pixel of each visible primitive. Those

triangles that have been associated with any textures (images) are rendered with

these images applied to them as desired. Visibility is resolved via the z-buffer

algorithm, along with optional discard and stencil tests. Each object is processed in

turn, and the final image is then displayed on the screen.

这个阶段的目标是计算每个可见片元中每个像素的颜色。那些跟一些纹理（图片）相关的三角形，将会根据需要应用图片去渲染。可见性会通过z缓冲算法，以及可选的丢弃和模板测试解决。每个图像会依次处理，最后的图像会显示在屏幕上。

Conclusion 结语

This pipeline resulted from decades of API and graphics hardware evolution targeted

to real-time rendering applications. It is important to note that this is not the only

possible rendering pipeline; offline rendering pipelines have undergone different

evolutionary paths. Rendering for film production was often done with micropolygon

pipelines [289, 1734], but ray tracing and path tracing have taken over lately. These

techniques, covered in Section 11.2.2, may also be used in architectural and design

previsualization.

这条流水管线是数十年API和面向实时渲染应用程序的图形硬件发展的结果。重要的是需要注意这不是仅有渲染管道；离线渲染管道经历了不同的发展路径。电影产品的渲染经常采用微多边形流水管线[289,1734]，但是射线追踪和路径追踪最近开始流行起来。这些技术，也可以用在建筑和视觉化设计领域上，会在章节11.2.2中讲解。

For many years, the only way for application developers to use the process described

here was through a fixed-function pipeline defined by the graphics API in use. The

fixed-function pipeline is so named because the graphics hardware that implements

it consists of elements that cannot be programmed in a flexible way. The last

example of a major fixed-function machine is Nintendo’s Wii, introduced in 2006.

Programmable GPUs, on the other hand, make it possible to determine exactly what

operations are applied in various sub-stages throughout the pipeline. For the fourth

edition of the book, we assume that all development is done using programmable

GPUs.

多年来，应用程序开发者描述过程的唯一方式是通过使用图形API定义的固定功能流水管线。固定功能流水管线之所以如此命名，是因为图形硬件执行其下面流程是只能使用固定的方式，不可编程的。最后一个专业固定功能流水管线机型的例子是2006年任天堂推出的Wii。另一方面，可编程的GPU，能够准确的确定在流水管线各个子阶段都应用了哪些操作。对于本书的第4版，我们假设所有的开发者都是使用可编程的GPU完成的。

Further Reading and Resources 进一步的阅读和参考资料

Blinn’s book A Trip Down the Graphics Pipeline [165] is an older book about writing a

software renderer from scratch. It is a good resource for learning about some of the

subtleties of implementing a rendering pipeline, explaining key algorithms such as

clipping and perspective interpolation. The venerable (yet frequently updated)

OpenGL Programming Guide (a.k.a. the “Red Book”) [885] provides a thorough

description of the graphics pipeline and algorithms related to its use. Our book’s

website, realtimerendering.com, gives links to a variety of pipeline diagrams,

rendering engine implementations, and more.

Blinn’s的书“Trip Down the Graphics Pipeline”[165]是一本关于从0开始编写软件渲染器的老书。这是一个很好的学习资源，介绍了实现渲染管道的微妙之处，解释了关键算法比如裁剪和透视插值。历史悠久（经常更新）的OpenGL编程指南（也称为红皮书）[885]提供了图形渲染管线的详细描述和相关的使用算法。我们书的网站，realtimerendering.com，给出了各种渲染管道图，渲染引擎实现和更多内容的链接。

Chapter 3 The Graphics Processing Unit

第3章 GPU

Historically, graphics acceleration started with interpolating colors on each pixel

scanline overlapping a triangle and then displaying these values. Including the ability

to access image data allowed textures to be applied to surfaces. Adding hardware for

interpolating and testing z-depths provided built-in visibility checking. Because of

their frequent use, such processes were committed to dedicated hardware to

increase performance. More parts of the rendering pipeline, and much more

functionality for each, were added in successive generations. Dedicated graphics

hardware’s only computational advantage over the CPU is speed, but speed is critical.

历史上，图形学加速发展开始于插值计算覆盖三角形每个像素的颜色并显示出来。包括访问图像数据并允许在其表面贴纹理的能力。添加了支持插值运算和z深度的硬件提供了像素的可见性测试。由于频繁使用，这样的过程委托与专用的硬件去提升性能。渲染管道的大部分，和每个部分的更多功能，都是连续的几代中添加的。专用图形硬件相对于CPU的唯一计算优势是速度，但是速度也是最关键因素。

Over the past two decades, graphics hardware has undergone an incredible

transformation. The first consumer graphics chip to include hardware vertex

processing(NVIDIA’s GeForce256) shipped in 1999. NVIDIA coined the term graphics

processing unit (GPU) to differentiate the GeForce 256 from the previously available

rasterization only chips, and it stuck. During the next few years, the GPU evolved

from configurable implementations of a complex fixed-function pipeline to highly

programmable blank slates where developers could implement their own algorithms.

Programmable shaders of various kinds are the primary means by which the GPU is

controlled. For efficiency, some parts of the pipeline remain configurable, not

programmable, but the trend is toward programmability and flexibility [175].

最近20年，图形硬件经历了令人难以想象的转变。第一个包括硬件顶点处理的消费图形芯片（NVIDIA’s GeForce256）在1999年上市。NVIDIA创造了术语图形处理器（GPU），区分开了先前的只能光栅处理而且已经到达瓶颈的芯片GeForce 256。接下来几年里，GPU从可配置执行的复杂固定功能渲染管线进化到了高度可编程的空白阶段，甚至开发者可以实现自己的算法。各种各样的可编程着色器是控制GPU的主要方式。为了提高效率，流水管线的一部分仍然是可配置而非可编程的，但是这个趋势是朝向可编程性和灵活性[175]。

GPUs gain their great speed from a focus on a narrow set of highly parallelizable

tasks. They have custom silicon dedicated to implementing the z-buffer, to rapidly

accessing texture images and other buffers, and to finding which pixels are covered

by a triangle, for example. How these elements perform their functions is covered in

Chapter 23. More important to know early on is how the GPU achieves parallelism for

its programmable shaders.

GPU的巨大速度来自集中一组狭窄高速并行的任务。他们有自定义专用硅去执行z缓冲，且高速的访问纹理图像和其他缓冲区，哪些像素被三角形覆盖。23章会介绍这些元素是如何执行他们的功能。更重要的是尽早了解GPU是如果实现可编程着色器的并行执行。

Section 3.3 explains how shaders function. For now, what you need to know is that a

shader core is a small processor that does some relatively isolated task, such as

transforming a vertex from its location in the world to a screen coordinate, or

computing the color of a pixel covered by a triangle. With thousands or millions of

triangles being sent to the screen each frame, every second there can be billions of

shader invocations, that is, separate instances where shader programs are run.

章节3.3部分介绍了着色器的功能。但是现在，你需要知道的是着色器核心是一个做一些相应独立任务的小处理器，比如转换一个顶点从世界空间到屏幕坐标，或者计算一个三角形覆盖的像素颜色。每一帧都有成千上百万的三角形传送到屏幕上，每秒可能有数十亿的着色器调用，也就是说着色器程序运行的独立实例。

To begin with, latency is a concern that all processors face. Accessing data takes

some amount of time. A basic way to think about latency is that the farther away the

information is from the processor, the longer the wait. Section 23.3 covers latency in

more detail. Information stored in memory chips will take longer to access than that

in local registers. Section 18.4.1 discusses memory access in more depth. The key

point is that waiting for data to be retrieved means the processor is stalled, which

reduces performance.

首先，延迟是所有处理器面对的问题。访问数据要花费一部分时间。考虑延迟的一个基本方式是，信息离处理器越远，需要等的时间越长。章节23.3更详细的介绍延迟。信息储存在内存芯片将花费更多的时间比当地寄存器访问。章节18.4.1更深入的讲解内存访问。关键点在于等待数据被检索意味着处理器停止运行，这会降低性能。

3.1 Data-Parallel Architectures 数据并行架构

Various strategies are used by different processor architectures to avoid stalls. A CPU

is optimized to handle a wide variety of data structures and large code bases. CPUs

can have multiple processors, but each runs code in a mostly serial fashion, limited

SIMD vector processing being the minor exception. To minimize the effect of latency,

much of a CPU’s chip consists of fast local caches, memory that is filled with data

likely to be needed next. CPUs also avoid stalls by using clever techniques such as

branch prediction, instruction reordering, register renaming, and cache prefetching

[715].

不同的处理器架构使用不同的策略来避免停滞。CPU经过优化可以处理大量的数据结构和大量的代码库。CPU可以有多个处理器，但是每个处理器以连续方式运行代码，这是有限制的SIMD向量处理的一个小的例外。为了减少延迟的影响，大部分CPU芯片包含了快速的本地缓存，内存很有可能填满了接下来需要的数据。CPU还通过使用智能技术比如预分支，指令重排，寄存命名和缓存预取等避免停滞。

GPUs take a different approach. Much of a GPU’s chip area is dedicated to a large set

of processors, called shader cores, often numbering in the thousands. The GPU is a

stream processor, in which ordered sets of similar data are processed in turn.

Because of this similarity—a set of vertices or pixels, for example—the GPU can

process these data in a massively parallel fashion. One other important element is

that these invocations are as independent as possible, such that they have no need

for information from neighboring invocations and do not share writable memory

locations. This rule is sometimes broken to allow new and useful functionality, but

such exceptions come at a price of potential delays, as one processor may wait on

another processor to finish its work.

GPU采用不同的方法。大部分GPU芯片区域专用于一组大的处理器，称为着色器内核，经常有上千个。GPU是流处理器，他依次有序的处理相似数据集。由于这种相似性（例如一系列顶点或像素），GPU可以大规模并行的方式处理数据。另外一个重要的元素是这些调用尽可能的独立，因此他们不需要来自相邻调用的信息以及也不共享可写内存位置。这个准则有时候会被打破，以允许新的和有用的功能，但是这种例外是以潜在的延迟为代价，因为一个处理器可能会等待另一个处理器完成它的工作。

The GPU is optimized for throughput, defined as the maximum rate at which data

can be processed. However, this rapid processing has a cost. With less chip area

dedicated to cache memory and control logic, latency for each shader core is

generally considerably higher than what a CPU processor encounters [462].

GPU对吞吐量进行优化，定义为数据处理的最大速率。然而这种快速处理是有代价的，更少的芯片去专一缓存和处理逻辑，每个着色器内核的延迟通常比CPU处理器遇到的延迟要高的多[462]。

Say a mesh is rasterized and two thousand pixels have fragments to be processed; a

pixel shader program is to be invoked two thousand times. Imagine there is only a

single shader processor, the world’s weakest GPU. It starts to execute the shader

program for the first fragment of the two thousand. The shader processor performs

a few arithmetic operations on values in registers. Registers are local and quick to

access, so no stall occurs. The shader processor then comes to an instruction such

as a texture access; e.g., for a given surface location the program needs to know the

pixel color of the image applied to the mesh. A texture is an entirely separate

resource, not a part of the pixel program’s local memory, and texture access can be

somewhat involved. A memory fetch can take hundreds to thousands of clock cycles,

during which time the GPU processor is doing nothing. At this point the shader

processor would stall, waiting for the texture’s color value to be returned.

一个网格被光栅化，他有2000像素片元被处理；一个像素着色器程序被调用了2000次。想象一下这里只有一个着色器处理器，世界上最弱的GPU。他开始执行着色器程序中2000个片元的第一个。着色器处理器对寄存器中的值执行一些算术操作。寄存器是本地的可以快速访问，因此不会发生停机。着色器处理器接下来要执行一条指令去访问纹理；例如：给定一个程序需要知道网格表面应用的像素的位置。纹理是一个完全独立的资源，并不是像素程序本地内存的一部分，并且纹理访问可能涉及到一些其他的内容。一次内存获取可能花费成百上千的时间，在这段时间GPU处理器什么也不做。此时着色器处理器将停止工作，等待纹理颜色值的返回。

To make this terrible GPU into something considerably better, give each fragment a

little storage space for its local registers. Now, instead of stalling on a texture fetch,

the shader processor is allowed to switch and execute another fragment, number

two of two thousand. This switch is extremely fast, nothing in the first or second

fragment is affected other than noting which instruction was executing on the first.

Now the second fragment is executed. Same as with the first, a few arithmetic

functions are performed, then a texture fetch is again encountered. The shader core

now switches to another fragment, number three. Eventually all two thousand

fragments are processed in this way. At this point the shader processor returns to

fragment number one. By this time the texture color has been fetched and is

available for use, so the shader program can then continue executing. The processor

proceeds in the same fashion until another instruction that is known to stall

execution is encountered, or the program completes. A single fragment will take

longer to execute than if the shader processor stayed focused on it, but overall

execution time for the fragments as a whole is dramatically reduced.

为了使糟糕的GPU变得更好，现在给每个片元一段本地寄存器的储存空间。现在，着色器处理器将不再停在纹理获得，而是开始转向处理另一个片元，2000个片元的第二个。这种转换是非常快的，第一个或者第二个片元的任何内容都不会受到影响，只会注意在第一个片元上执行的是哪条指令。现在第二个片元被执行。跟第一个相同，一些算术运算被执行，接着又一次遇到了纹理的获得。着色器核心转向的另一个片元，第三个片元。所有2000个片元用这种方式处理。此时着色器处理器转向片元一。这个过程纹理颜色已经获得并可以使用，因此着色器程序可以继续执行。处理器以相同的方式进行直到遇到另一个已知会导致执行停顿的指令，或者程序完成。单个片元将花费比着色器处理器一直关注更长的时间执行，但是片元作为一个整体的整体执行时间将显著减少。

In this architecture, latency is hidden by having the GPU stay busy by switching to

another fragment. GPUs take this design a step further by separating the instruction

execution logic from the data. Called single instruction, multiple data (SIMD), this

arrangement executes the same command in lock-step on a fixed number of shader

programs. The advantage of SIMD is that considerably less silicon (and power) needs

to be dedicated to processing data and switching, compared to using an individual

logic and dispatch unit to run each program. Translating our two-thousand fragment

example into modern GPU terms, each pixel shader invocation for a fragment is

called a thread. This type of thread is unlike a CPU thread. It consists of a bit of

memory for the input values to the shader, along with any register space needed for

the shader’s execution. Threads that use the same shader program are bundled into

groups, called warps by NVIDIA and wavefronts by AMD. A warp/wavefront is

scheduled for execution by some number GPU shader cores, anywhere from 8 to 64,

using SIMD-processing. Each thread is mapped to a SIMD lane.

在这种架构下，通过使GPU忙于切换到另一个片元来隐藏延迟。通过从数据中分离逻辑执行指令，GPU进一步实现了这一设计。称为单一指令，多重数据（SIMD），这种管理使得固定数量的着色器程序锁步执行同一命令。与单一逻辑和分发单元去运行每个程序相比，SIMD的优点在于处理数据和转换所用的硅（和电能）更少。将我们2000个片元例子转换成现代GPU术语，每个片元的像素着色器调用称为一个线程。这种类型的线程不像CPU线程。他包含了一点内存把数据传送到着色器，以及一些供着色器执行的寄存器空间。使同样着色器程序的线程被捆绑成组，NVIDIA称为翘曲，AMD称为波前。翘曲/波前计划由一些GPU着色核心执行，从8到64，使用SIMD处理，每个线程映射为一个SIMD道。

Say we have two thousand threads to be executed. Warps on NVIDIA GPUs contain

32 threads. This yields 2000/32 = 62.5 warps, which means that 63 warps are

allocated, one warp being half empty. A warp’s execution is similar to our single GPU

processor example. The shader program is executed in lock-step on all 32 processors.

When a memory fetch is encountered, all threads encounter it at the same time,

because the same instruction is executed for all. The fetch signals that this warp of

threads will stall, all waiting for their (different) results. Instead of stalling, the warp

is swapped out for a different warp of 32 threads, which is then executed by the 32

cores. This swapping is just as fast as with our single processor system, as no data

within each thread is touched when a warp is swapped in or out. Each thread has its

own registers, and each warp keeps track of which instruction it is executing.

Swapping in a new warp is just a matter of pointing the set of cores at a different set

of threads to execute; there is no other overhead. Warps execute or swap out until

all are completed. See Figure 3.1.

我们有2000个线程要被执行。NVIDIA GPU的翘曲包含了32个线程。这产生了2000/32 = 62.5个翘曲，意味着63个翘曲被分配，其中一个为半空。一个翘曲的执行同我们单一GPU处理器例子相同。在32个处理器上，着色器程序锁步执行。当遇到获得一段内存，所有的线程会在同一时间遭遇，因为所有线程执行依靠同一指令。获取内存的信号表明这个线程的翘曲将要停止，所有的线程等待他们（不同的）结果。为了避免停顿，这个翘曲被另一个32线程的翘曲替换，然后被32个核心执行。这种交换速度与我们单一处理器系统一样快，因为在翘曲交换过程中每个线程内的数据都不会受到影响。每个线程都有自己的寄存器，每个翘曲都持续追踪自己执行的指令。交换一个新的翘曲仅仅只一组核心指向另一组线程的执行的问题，并没有其他的开销。翘曲执行或交换直到所有翘曲完成。见图3.1。

In our simple example the latency of a memory fetch for a texture can cause a warp

to swap out. In reality warps could be swapped out for shorter delays, since the cost

of swapping is so low. There are several other techniques used to optimize execution

[945], but warp-swapping is the major latency-hiding mechanism used by all GPUs.

Several factors are involved in how efficiently this process works. For example, if

there are few threads, then few warps can be created, making latency hiding

problematic.

在我们简单的例子中，获取纹理内存的延迟会导致翘曲转换到另一个。实际上由于交换的成本非常低，可以用更短的延迟替换掉翘曲。有几个其他的技术用来优化执行[945]，但是翘曲交换是所有的GPU的主要减少延迟机制。这个过程的效率涉及到几个因素。例如，如果只有很少的线程，那么只有很少的翘曲创建，从而导致延迟隐藏问题。

The shader program’s structure is an important characteristic that influences

efficiency. A major factor is the amount of register use for each thread. In our

example we assume that two thousand threads can all be resident on the GPU at

one time. The more registers needed by the shader program associated with each

thread, the fewer threads, and thus the fewer warps, can be resident in the GPU. A

shortage of warps can mean that a stall cannot be mitigated by swapping. Warps

that are resident are said to be “in flight,” and this number is called the occupancy.

High occupancy means that there are many warps available for processing, so that

idle processors are less likely. Low occupancy will often lead to poor performance.

The frequency of memory fetches also affects how much latency hiding is needed.

Lauritzen [993] outlines how occupancy is affected by the number of registers and

the shared memory that a shader uses. Wronski [1911, 1914] discusses how the

ideal occupancy rate can vary depending on the type of operations a shader

performs.

着色器程序结构是影响效率的重要角色。主要的因素是每个线程中的寄存器使用数量。在我们的例子中，我们假定GPU上同时驻留了2000个线程。着色器程序每个线程相联系寄存器需求更多，那么可以驻留在GPU上的线程越少，翘曲越少。翘曲的不足意味着延迟不能通过交换来减缓。驻留的翘曲称为“在飞行中”，这个数量称为占用。高占用率意味着很多可用的翘曲要被处理，因此空闲处理器的可能性很小。低占用常常导致较差的性能。内存获取的频率也影响多少延迟需要被隐藏。Lauritzen [993]概述了占有率是如何被寄存器数量和着色器使用共享内存影响。Wronski [1911, 1914]讨论了理想的占有率如何根据着色器执行的操作类型而变化。

Another factor affecting overall efficiency is dynamic branching, caused by “if”

statements and loops. Say an “if” statement is encountered in a shader program. If

all the threads evaluate and take the same branch, the warp can continue without

any concern about the other branch. However, if some threads, or even one thread,

take the alternate path, then the warp must execute both branches, throwing away

the results not needed by each particular thread [530, 945]. This problem is called

thread divergence, where a few threads may need to execute a loop iteration or

perform an “if” path that the other threads in the warp do not, leaving them idle

during this time.

另一个因素影响整体效率是动态分支，由“if”语句和循环引起。在着色器程序中遇到了“if”语句。如果所有的线程评估并去同一个分支，那么翘曲会继续执行而不去关心另一个分支。然而，如果一些线程，甚至是一个线程，选择了另一个路径，那么翘曲必须执行两个分支，每个独立的线程将丢弃掉不需要的结果[530,945]。这个问题成为线程分散，当一些线程需要执行循环迭代或者执行“if”路径，其他的翘曲线程不需要时候，那么他们将在这个时间闲置。

All GPUs implement these architectural ideas, resulting in systems with strict

limitations but massive amounts of compute power per watt. Understanding how this

system operates will help you as a programmer make more efficient use of the

power it provides. In the sections that follow we discuss how the GPU implements

the rendering pipeline, how programmable shaders operate, and the evolution and

function of each GPU stage.

所有的GPU实现了这种架构思想，导致系统有严格的限制，但是每瓦特的计算能力非常大。理解这种系统操作将帮助你作为一个开发者更有效的利用他提供的功能。接下来的部分，我们将讨论GPU如何实现渲染管道，可编程着色器如何操作，以及每个GPU阶段的发现和功能。

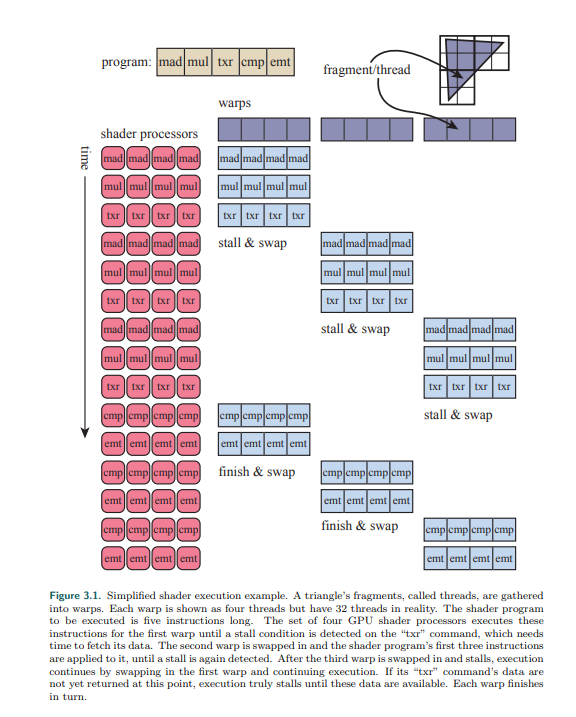


图3.1 简化的着色器执行例子。一个三角形片元，称为线程，集合成翘曲。每个翘曲显示有4个线程，实际上是32个线程。着色器程序由5个指令执行。四个GPU着色处理器执行第一个翘曲的指令直到执行“txr”指令遇到了停顿，需要时间取得数据。第二个翘曲交换进来，执行着色器程序的前3个指令，直到又一次检测到停顿。在第三个翘曲交换进来以及停顿，将交换进第一个翘曲继续执行指令。如果此时他的“txr”命令数据还没有得到，那么执行将真正停顿，直到这些数据可用为止。每个翘曲轮流结束。

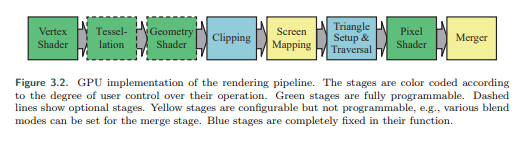


图3.2 GPU实现的渲染管线。依据操作的可控性涂上了不同的颜色。绿色是完全可编程的。点线代表可选阶段。黄色是可配置的但是不可编程，比如各种混合模式可以在合并阶段设置。蓝色是完全固定功能。

3.2 GPU Pipeline Overview GPU流水管线概述

The GPU implements the conceptual geometry processing, rasterization, and pixel

processing pipeline stages described in Chapter 2. These are divided into several

hardware stages with varying degrees of configurability or programmability. Figure

3.2 shows the various stages color coded according to how programmable or

configurable they are. Note that these physical stages are split up somewhat

differently than the functional stages presented in Chapter 2.

在第2章提到了GPU实现了概念上的几何阶段、光栅化和像素处理阶段。这些阶段根据不同程度的可配置和编程性被划分几个硬件阶段。图3.2根据可编程或可配置程度显示了不同的颜色。注意：这些物理阶段的划分与第2章介绍的功能阶段有所不同。

We describe here the logical model of the GPU, the one that is exposed to you as a

programmer by an API. As Chapters 18 and 23 discuss, the implementation of this

logical pipeline, the physical model, is up to the hardware vendor. A stage that is

fixed-function in the logical model may be executed on the GPU by adding

commands to an adjacent programmable stage. A single program in the pipeline may

be split into elements executed by separate sub-units, or be executed by a separate

pass entirely. The logical model can help you reason about what affects performance,

but it should not be mistaken for the way the GPU actually implements the pipeline.

我们在这里描述GPU的逻辑模型，他通过API对开发者公开。正如18章和23章讨论的，逻辑流水管线的实现，物理模型取决于硬件供应商。逻辑模型的固定功能阶段可以通过相邻的可编程阶段添加命令在GPU中执行。流水管线的单个程序可以被分割由单独的子单元执行的元素，或者完全单独执行。逻辑模型可以帮助你分析英雄性能的原因，而不是将其误认为GPU实际实现管道的方式。

The vertex shader is a fully programmable stage that is used to implement the

geometry processing stage. The geometry shader is a fully programmable stage that

operates on the vertices of a primitive (point, line, or triangle). It can be used to

perform per-primitive shading operations, to destroy primitives, or to create new

ones. The tessellation stage and geometry shader are both optional, and not all

GPUs support them, especially on mobile devices.

顶点着色器是完全可编程的阶段，用来实现几何阶段。几何着色器是完全可编程阶段用来操作图元（点，线或者三角形）上的顶点。同样可以用来执行每图元着色操作，去销毁图元，或者创建新的图元。镶嵌阶段和几何着色器都是可选的，并不是所有的GPU都支持这些，尤其在移动设备上。

The clipping, triangle setup, and triangle traversal stages are implemented by ﬁxed-

function hardware. Screen mapping is aﬀected by window and viewport settings,

internally forming a simple scale and repositioning. The pixel shader stage is fully

programmable. Although the merger stage is not programmable, it is highly

conﬁgurable and can be set to perform a wide variety of operations. It implements

the “merging” functional stage, in charge of modifying the color, z-buffer, blend,

stencil, and any other output-related buffers. The pixel shader execution together

with the merger stage form the conceptual pixel processing stage presented in

Chapter 2.

裁剪，三角形设置和三角形遍历阶段是被固定功能硬件执行。屏幕映射受窗口和视口设置所影响，内部形成一个简单的缩放和重定位。像素着色器阶段是完全可编程的。尽管合并阶段不是可编程的，但是他是高度可配置的，以及可以设置执行各种操作。执行合并功能阶段，主要负责颜色，Z缓冲区，混合，模板以及其他相关输出缓冲区数据的更新。像素着色器的执行和合并阶段构成了第2章描述的像素处理阶段概念。

Over time, the GPU pipeline has evolved away from hard-coded operation and

toward increasing ﬂexibility and control. The introduction of programmable shader

stages was the most important step in this evolution. The next section describes the

features common to the various programmable stages.

随着时间的推移，GPU流水管线从硬编码操作朝着增加更灵活和控制的方向发展。引入可编程着色器阶段是演变中最重要的一步。下个部分将表述各种可编程阶段的共同特征。

3.3 The Programmable Shader Stage 可编程着色器阶段

Modern shader programs use a unified shader design. This means that the vertex,

pixel, geometry, and tessellation-related shaders share a common programming

model. Internally they have the same instruction set architecture (ISA). A processor

that implements this model is called a common-shader core in DirectX, and a GPU

with such cores is said to have a unified shader architecture. The idea behind this

type of architecture is that shader processors are usable in a variety of roles, and the

GPU can allocate these as it sees fit. For example, a set of meshes with tiny triangles

will need more vertex shader processing than large squares each made of two

triangles. A GPU with separate pools of vertex and pixel shader cores means that the

ideal work distribution to keep all the cores busy is rigidly predetermined. With

unified shader cores, the GPU can decide how to balance this load.

现代着色器程序使用统一的着色器设计。意思是顶点、像素、几何和镶嵌相关着色器共享同一编程模型。内部他们具有同一指令系统体系结构（ISA）。在DirectX中，执行这个模型的处理器称为共同着色器核心，一个GPU具有这样的核心被称为有统一着色器结构。这种架构类型背后的想法是着色器处理器可以用在各种角色，GPU可以根据自己的需要分配这些处理器。举个例子，一系列具有小三角形的网格比2个三角形组成的矩阵需要更多的顶点着色器处理。一个拥有独立的顶点和像素着色核心的GPU意味着保持所有核心忙碌的理想工作分配要严格预先确定的。具有统一着色器核心，GPU可以决定如何平衡这种负载。

Describing the entire shader programming model is well beyond the scope of this

book, and there are many documents, books, and websites that already do so.

Shaders are programmed using C-like shading languages such as DirectX’s High-

Level Shading Language (HLSL) and the OpenGL Shading Language (GLSL). DirectX’s

HLSL can be compiled to virtual machine bytecode, also called the intermediate

language (IL or DXIL), to provide hardware independence. An intermediate

representation can also allow shader programs to be compiled and stored offline.

This intermediate language is converted to the ISA of the specific GPU by the driver.

Console programming usually avoids the intermediate language step, since there is

then only one ISA for the system.

描述全部的可编程着色器模型超出了本书的范围，已经有很多文档、书和网站已经这样做了。着色器使用类C着色语言编程，比如DirectX的高级着色语言（HLSL）和OpenGL着色语言（GLSL）。DirectX的HLSL可以编译为虚拟机字节码，也称为中间语言（IL或者DXIL），以提供硬件独立性。中间码也允许脱机编译和存储着色器程序。中间语言被驱动程序转换为特定GPU的ISA。控制台程序通常避免中间语言步骤，因为这里仅仅只有一个ISA用于系统。

The basic data types are 32-bit single-precision floating point scalars and vectors,

though vectors are only part of the shader code and are not supported in hardware

as outlined above. On modern GPUs 32-bit integers and 64-bit floats are also

supported natively. Floating point vectors typically contain data such as positions

(xyzw), normals, matrix rows, colors (rgba), or texture coordinates (uvwq). Integers

are most often used to represent counters, indices, or bitmasks. Aggregate data

types such as structures, arrays, and matrices are also supported.

基础数据类型是32位单精度浮点值标量和向量，尽管向量只是着色器代码的一部分，并且在上述概述的硬件中并不支持。在现代GPU上，32位整形和64位浮点值也得到了支持。浮点值向量通常包括位置（xyzw），法线，矩阵行，颜色（rgba），或者纹理坐标（uvwq）。整形数据通常用在计数器，下标或者位掩码。还支持聚合数据类型比如结构、数组和矩阵。

A draw call invokes the graphics API to draw a group of primitives, so causing the

graphics pipeline to execute and run its shaders. Each programmable shader stage

has two types of inputs: uniform inputs, with values that remain constant throughout

a draw call (but can be changed between draw calls), and varying inputs, data that

come from the triangle’s vertices or from rasterization. For example, a pixel shader

may provide the color of a light source as a uniform value, and the triangle surface’s

location changes per pixel and so is varying. A texture is a special kind of uniform

input that once was always a color image applied to a surface, but that now can be

thought of as any large array of data.

一个绘制命令调用了图形应用接口去绘制一组图元，驱使图形渲染管线执行，运行着色器。每一个可编程着色器阶段有两种输入类型：统一输入，一个绘制命令中值保持不变（但是可能在绘制命令之间改变），和不同输入，数据来自三角形顶点或者光栅化。举个例子，像素着色器光源颜色作为统一数值，并且三角形表面的位置随着每个像素的变化而变化。纹理是一种特殊的统一输入，曾经只是用在表面的一张颜色图像，但是现在可以看作任何大的数据数组。

The underlying virtual machine provides special registers for the different types of

inputs and outputs. The number of available constant registers for uniforms is much

larger than those registers available for varying inputs or outputs. This happens

because the varying inputs and outputs need to be stored separately for each vertex

or pixel, so there is a natural limit as to how many are needed. The uniform inputs

are stored once and reused across all the vertices or pixels in the draw call. The

virtual machine also has general-purpose temporary registers, which are used for

scratch space. All types of registers can be array-indexed using integer values in

temporary registers. The inputs and outputs of the shader virtual machine can be

seen in Figure 3.3.

底层虚拟机为不同的输入输出数据提供了特殊的寄存器。统一的可用常量寄存器的数量要比可变的输入输出可用寄存器要多。是因为变化的输入输出需要为每个顶点或像素独立存储，因此需要多少是有限制的。统一输入只存储一次，在绘制命令中通过所有的顶点和像素重复使用。虚拟机还具有通用的临时寄存器，用在临时存储空间。所有类型的寄存器可以使用临时寄存器中的整数数值进行数值索引。着色器虚拟机的输入和输出如图3.3所示。

Operations that are common in graphics computations are efficiently executed on

modern GPUs. Shading languages expose the most common of these operations

(such as additions and multiplications) via operators such as \* and +. The rest are

exposed through intrinsic functions, e.g., atan(), sqrt(), log(), and many others,

optimized for the GPU. Functions also exist for more complex operations, such as

vector normalization and reflection, the cross product, and matrix transpose and

determinant computations.

图形计算中常见的操作可以在现在GPU上高效的执行。着色器语言通过比如\*和+执行最常见的操作（比如加法和乘法）。其余部分通过固有的函数，比如atan(), sqrt(), log(),和其他的，这些函数都为GPU优化过。函数也适用于更复杂的操作，比如向量归一化和反射，叉乘，和矩阵转置和行列式计算。

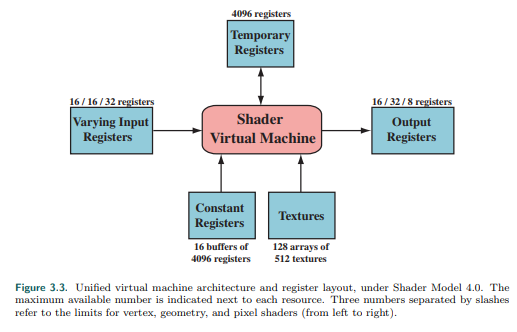


图3.3：统一的虚拟机架构和寄存器布局。在着色器模型4.0下，最大数量可用数据显示在每个资源旁边。斜线隔开的三个数据参考顶点、几何和像素着色器的限制（从左到右）。

The term flow control refers to the use of branching instructions to change the flow

of code execution. Instructions related to flow control are used to implement high-

level language constructs such as “if” and “case” statements, as well as various types

of loops. Shaders support two types of flow control. Static flow control branches are

based on the values of uniform inputs. This means that the flow of the code is

constant over the draw call. The primary benefit of static flow control is to allow the

same shader to be used in a variety of different situations (e.g., a varying numbers

of lights). There is no thread divergence, since all invocations take the same code

path. Dynamic flow control is based on the values of varying inputs, meaning that

each fragment can execute the code differently. This is much more powerful than

static flow control but can cost performance, especially if the code flow changes

erratically between shader invocations.

流程控制术语指的是使用分支指令去改变代码流程执行。流程控制相关的指令被用来执行高级语言结构比如“if”和“case”语句，以及各种类型的循环。着色器支持两种流程控制类型。静态流程控制分支建立在统一输入数据。意味着在绘制命令过程中代码流程是固定的。静态流程控制的优点在于可以使同一着色器可用在不同的情况（比如不同数量的灯光）。没有线程发散，因为所有的调用采用同样的代码路径。动态流程控制是基于不同的输入数据，意思是每个片元可以以不同的的方式执行代码。这比静态流程控制强大的多但是更加消耗性能，尤其是当代码流在着色器调用中不规律的变化。

3.4 The Evolution of Programmable Shading and APIs 可编程着色器和应用程序接口的演变

The idea of a framework for programmable shading dates back to 1984 with Cook’s

shade trees [287]. A simple shader and its corresponding shade tree are shown in

Figure 3.4. The RenderMan Shading Language [63, 1804] was developed from this

idea in the late 1980s. It is still used today for film production rendering, along with

other evolving specifications, such as the Open Shading Language (OSL) project

[608].

可编程着色器的概念追溯到1984年Cook的着色树[287]。一个简单的着色器和他对应的着色树如图3.4所示。RenderMan着色器语言[63,1804]在1980年晚期从这种思想发展而来。他仍然用于今天的电影渲染，以及其他不断发展的规范，如开放着色器语言（OSL）项目[608]。

Consumer-level graphics hardware was first successfully introduced by 3dfx

Interactive on October 1, 1996. See Figure 3.5 for a timeline from this year. Their

Voodoo graphics card’s ability to render the game Quake with high quality and

performance led to its quick adoption. This hardware implemented a fixed-function

pipeline throughout. Before GPUs supported programmable shaders natively, there

were several attempts to implement programmable shading operations in real time

via multiple rendering passes. The Quake III: Arena scripting language was the first

widespread commercial success in this area in 1999. As mentioned at the beginning

of the chapter, NVIDIA’s GeForce256 was the first hardware to be called a GPU, but

it was not programmable. However, it was configurable.

1996年10月1日，3dfx Interactive首次成功的推出了顾客级图形硬件。从那年开始的时间线见图3.5.巫毒显卡的能力是高质量和性能渲染了游戏Quake，致使他迅速被采用。该硬件实现了固定功能渲染管线。在GPU原生支持可编程着色器之前，有一些人试图通过多重渲染通道实现实时可编程着色操作。Quake III：竞技场脚本语言在1999年首次在这个领取取得广泛的商业成功。正如本章开始提到的，NVIDIA的GeForce256是第一个被称为GPU的硬件，但是他不是可编程的，而是可配置的。

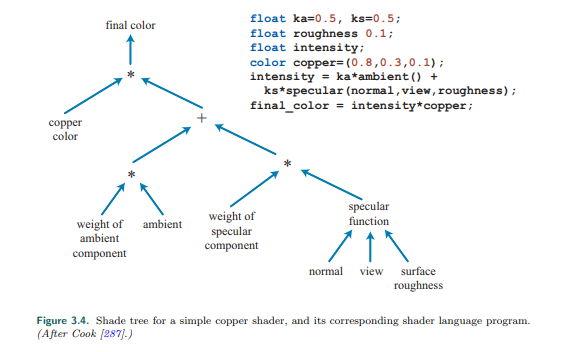


图3.4 简单copper着色器的着色树，以及他对应的着色器程序（cook[287]）

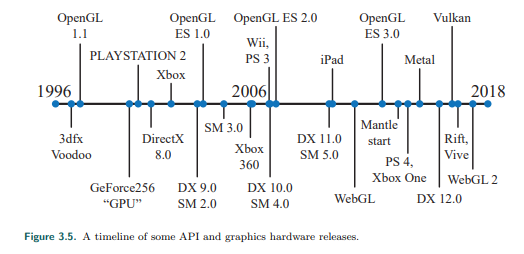


图3.5 应用程序接口和图像硬件发布的时间线

In early 2001, NVIDIA’s GeForce 3 was the first GPU to support programmable

vertex shaders [1049], exposed through DirectX 8.0 and extensions to OpenGL.

These shaders were programmed in an assembly-like language that was converted

by the drivers into microcode on the fly. Pixel shaders were also included in DirectX

8.0, but pixel shaders fell short of actual programmability—the limited “programs”

supported were converted into texture blending states by the driver, which in turn

wired together hardware “register combiners.” These “programs” were not only

limited in length (12 instructions or less) but also lacked important functionality.

Dependent texture reads and floating point data were identified by Peercy et al.

[1363] as crucial to true programmability, from their study of RenderMan.

在2001年早期，NVIDIA的GeForce 3是第一个支持可编程顶点着色器的GPU[1049]，通过DirectX 8.0和OpenGL拓展公开。这些着色器是以一种类似程序集的语言编写，驱动程序可以动态将其转换成微代码。DirectX 8.0同样也包含了像素着色器，但是像素着色器缺少实际的可编程性——驱动所支持的程序转换纹理混合状态的限制，从而将硬件“寄存器组合器”连接在一起。这些程序不仅仅是长度的限制（12个指令或更少），同样缺乏重要的功能性。Peercy等人通过对RenderMan的研究发现，依赖纹理读取以及浮点数据的鉴别对于真正的可编程性至关重要。

Shaders at this time did not allow for flow control (branching), so conditionals had to

be emulated by computing both terms and selecting or interpolating between the

results. DirectX defined the concept of a Shader Model (SM) to distinguish hardware

with different shader capabilities. The year 2002 saw the release of DirectX 9.0

including Shader Model 2.0, which featured truly programmable vertex and pixel

shaders. Similar functionality was also exposed under OpenGL using various

extensions. Support for arbitrary dependent texture reads and storage of 16-bit

floating point values was added, finally completing the set of requirements identified

by Peercy et al. Limits on shader resources such as instructions, textures, and

registers were increased, so shaders became capable of more complex effects.

Support for flow control was also added. The growing length and complexity of

shaders made the assembly programming model increasingly cumbersome.

Fortunately, DirectX 9.0 also included HLSL. This shading language was developed

by Microsoft in collaboration with NVIDIA. Around the same time, the OpenGL ARB

(Architecture Review Board) released GLSL, a fairly similar language for OpenGL

[885]. These languages were heavily influenced by the syntax and design philosophy

the C programming language and included elements from the RenderMan Shading

Language.

着色器在这个时间是不允许流程控制的，因此坐标不得不双边计算选择或者是结果之间插值。DirectX定义了着色器模型（SM）的概念去区分不同着色器能力的硬件。2002年发布了包括着色器模型2.0的DirectX 9.0，他具有真正可编程的顶点和像素着色器。相同的功能同样随着OpenGL的各种拓展公开。添加了对任意依赖纹理读取和16位浮点值的存储的支持，最终完成了Peercy等人确定的需求。着色器资源上的限制比如指令、纹理、和寄存器增加了，因此着色器有能力处理更复杂的效果。添加了对流程控制的支持。随着着色器长度和复杂性的增长使得程序集模型变得更加笨重。幸运的是，DirectX 9.0包含了HLSL。这种着色器语言是微软和NVIDIA共同合作开发的。在同时，OpenGL ARB（Architecture Review Board）发布了GLSL，这是一种非常类似的OpenGL语言[885]。这些语言深受C编程语言的语法和设计理念的影响，并包含RenderMan着色语言的元素。

Shader Model 3.0 was introduced in 2004 and added dynamic flow control, making

shaders considerably more powerful. It also turned optional features into

requirements, further increased resource limits and added limited support for texture

reads in vertex shaders. When a new generation of game consoles was introduced in

late 2005 (Microsoft’s Xbox 360) and late 2006 (Sony Computer Entertainment’s

PLAYSTATION 3 system), they were equipped with Shader Model 3.0–level GPUs.

Nintendo’s Wii console was one of the last notable fixed-function GPUs, which initially

shipped in late 2006. The purely fixed-function pipeline is long gone at this point.

Shader languages have evolved to a point where a variety of tools are used to create

and manage them. A screenshot of one such tool, using Cook’s shade tree concept,

is shown in Figure 3.6.

着色器模型3.0是在2004年被提出，他增加了动态流程控制，使得着色器更加强大。他也将可选特征转变成需求，更进一步的增加了资源限制以及增加了在顶点着色器中对纹理读取的支持。新一代的游戏控制台发布比如2005年年末的(Microsoft’s Xbox 360)和2006年末的(Sony Computer Entertainment’s PLAYSTATION 3 system)，他们装备了着色器模型3.0级别的GPU。任天堂的Wii控制台是最后一批引人注目的固定功能GPU，最早于2006年末上市。此时，纯粹的固定功能流水管线消失了。着色器语言已经演变到各种各样的工具去创建和管理他们。图3.6显示了使用Cook着色器树概念的此类工具的截图。

The next large step in programmability also came near the end of 2006. Shader

Model 4.0, included in DirectX 10.0 [175], introduced several major features, such as

the geometry shader and stream output. Shader Model 4.0 included a uniform

programming model for all shaders (vertex, pixel, and geometry), the unified shader

design described earlier. Resource limits were further increased, and support for

integer data types (including bitwise operations) was added. The introduction of

GLSL 3.30 in OpenGL 3.3 provided a similar shader model.

下一个可编程的重大进展也出现在2006年底。包含着着色器模型4.0的DirectX 10.0[175]发布了几个主要的功能，比如几何着色器和流输出。着色器模型4.0包括了一个统一的所有着色器的可编程模型（顶点、像素、和几何），这是前面描述的统一着色器设计。资源限制更进一步增加，并且增加了支持整形数据类型（包括位操作）。OpenGL 3.3引入的GLSL 3.30提供了相似的着色器模型。

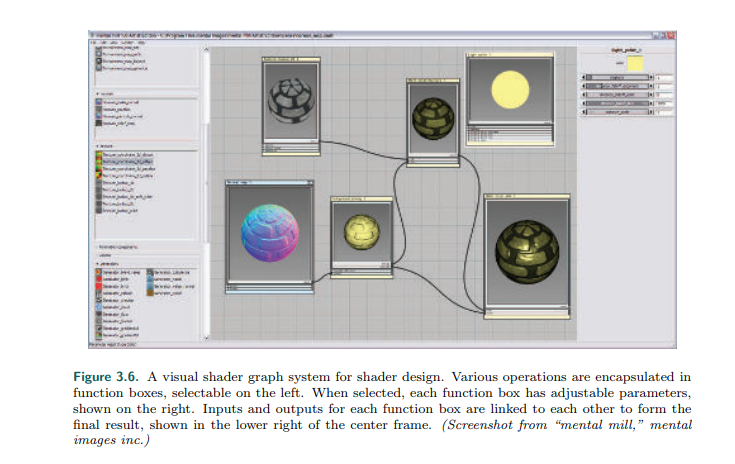


图3.6 着色器设计的一个可视化着色器图形系统。各种操作被封装在左边可选的函数箱子里面。当选择时，右边显示了每个函数盒子的可用参数。每个函数框的输入和输出相互廉洁，形成结果，如图中右下角所示。（截图来自《心智磨坊》）

In 2009 DirectX 11 and Shader Model 5.0 were released, adding the tessellation

stage shaders and the compute shader, also called DirectCompute. The release also

focused on supporting CPU multiprocessing more effectively, a topic discussed in

Section 18.5. OpenGL added tessellation in version 4.0 and compute shaders in 4.3.

DirectX and OpenGL evolve differently. Both set a certain level of hardware support

needed for a particular version release. Microsoft controls the DirectX API and so

works directly with independent hardware vendors (IHVs) such as AMD, NVIDIA, and

Intel, as well as game developers and computer-aided design software firms, to

determine what features to expose. OpenGL is developed by a consortium of

hardware and software vendors, managed by the nonprofit Khronos Group. Because

of the number of companies involved, the API features often appear in a release of

OpenGL some time after their introduction in DirectX. However, OpenGL allows

extensions, vendor-specific or more general, that allow the latest GPU functions to

be used before official support in a release.

在2009年DirectX 11和着色器模型5.0发布，增加了镶嵌阶段着色器和计算着色器，也被称为并行计算。发布同样聚焦在支持CPU更高效的多重处理，这是章节18.5讨论的主题。OpenGL在版本4.0增加了镶嵌，4.3版本增加了计算着色器。DirectX和OpenGL发展不同。两者都设置了特定版本发布所需的特定级别的硬件支持。Microsoft掌控者DirectX API因此直接与独立硬件开发商（IHVS）比如AMD、NVIDIA和Intel以及游戏开发者和计算机辅助设计开发商合作，去决定公开哪些特征。OpenGL是由硬件和软件开发商共同开发，由非盈利机构Khronos管理。由于包含的公司众多，API特征常常在DirectX引入之后的某个版本发布。然而，OpenGL允许拓展，供应商特定和或通用的，允许最新的GPU功能在正式支持发布之前使用。

The next significant change in APIs was led by AMD’s introduction of the Mantle API

in 2013. Developed in partnership with video game developer DICE, the idea of

Mantle was to strip out much of the graphics driver’s overhead and give this control

directly to the developer. Alongside this refactoring was further support for effective

CPU multiprocessing. This new class of APIs focuses on vastly reducing the time the

CPU spends in the driver, along with more efficient CPU multiprocessor support

(Chapter 18). The ideas pioneered in Mantle were picked up by Microsoft and

released as DirectX 12 in 2015. Note that DirectX 12 is not focused on exposing new

GPU functionality—DirectX 11.3 exposed the same hardware features. Both APIs can

be used to send graphics to virtual reality systems such as the Oculus Rift and HTC

Vive. However, DirectX 12 is a radical redesign of the API, one that better maps to

modern GPU architectures. Low-overhead drivers are useful for applications where

the CPU driver cost is causing a bottleneck, or where using more CPU processors for

graphics could benefit performance [946]. Porting from earlier APIs can be difficult,

and a naive implementation can result in lower performance [249, 699, 1438].

API下一个标志性的变化是AMD在2013年引入了Mantle API。Mantle是与视频游戏开发者DICE合作开发的，Mantle的理念是去电大部分的图形驱动的开销并把控制权直接交给开发者。除了这种重构，更有效的支持CPU多核处理。这种新型API着重极大的减少了CPU在驱动中的花费时间，以及支持更有效的CPU多重处理（章节18）。在Mantle中率先提出的理念是在2015年被微软采取，并随着DirectX 12发布。注意DirectX 12并不关注公开新的GPU功能——DirectX 11.3公开了相同的硬件特性。两种API都可以用来发送图像到虚拟现实系统比如Oculus Rift和HTC Vive。然而，DirectX 12彻底的重新设计了API，能够更好的映射到现代GPU架构。低开销驱动适用于CPU驱动消耗导致瓶颈的应用程序，或者用在多CPU处理图形可以提高性能的应用程序[946]。从早期的API移植会有一定困难，简单的实现可能导致降低性能[249，699，1438]。

Apple released its own low-overhead API called Metal in 2014. Metal was first

available on mobile devices such as the iPhone 5S and iPad Air, with newer

Macintoshes given access a year later through OS X El Capitan. Beyond efficiency,

reducing CPU usage saves power, an important factor on mobile devices. This API

has its own shading language, meant for both graphics and GPU compute programs.

苹果在2014年发布了他自己的低开销的API称为Metal。Metal是第一个可用在移动设备比如苹果5s和平板Air，一年后Macintoshes电脑可以通过OS X El Capitan操作系统使用。除了高效，减少CPU使用还可以节省电量，这是在移动设备上重要的因素。这个API有他自己的着色器语言，用于图像和GPU计算程序。

AMD donated its Mantle work to the Khronos Group, which released its own new API

in early 2016, called Vulkan. As with OpenGL, Vulkan works on multiple operating

systems. Vulkan uses a new high-level intermediate language called SPIRV, which is

used for both shader representation and for general GPU computing. Precompiled

shaders are portable and so can be used on any GPU supporting the capabilities

needed [885]. Vulkan can also be used for non-graphical GPU computation, as it

does not need a display window [946]. One notable difference of Vulkan from other

low-overhead drivers is that it is meant to work with a wide range of systems, from

workstations to mobile devices.

AMD将Mantle捐献给Khronos集团工作，后者在2016年初发布了他自己新的API，称为Vulkan。与OpenGL一样，Vulkan可工作在多个操作系统。Vulkan使用了新的高级中间语言称为SPIRV，他即用于着色器表示也用于通用GPU的计算。预编译着色器是可移植的，因此可以在任何支持所需功能的GPU上使用[885]。Vulkan同样可以用在非图像GPU计算，正如他不需要显示窗口[946]。Vulkan与其他低开销驱动的值得注意的不同点是，他适用于从工作站到移动设备的广泛系统。

On mobile devices the norm has been to use OpenGL ES. “ES” stands for Embedded

Systems, as this API was developed with mobile devices in mind. Standard OpenGL

at the time was rather bulky and slow in some of its call structures, as well as

requiring support for rarely used functionality. Released in 2003, OpenGL ES 1.0 was

a stripped-down version of OpenGL 1.3, describing a fixed-function pipeline. While

releases of DirectX are timed with those of graphics hardware that support them,

developing graphics support for mobile devices did not proceed in the same fashion.

For example, the first iPad, released in 2010, implemented OpenGL ES 1.1. In 2007

the OpenGL ES 2.0 specification was released, providing programmable shading. It

was based on OpenGL 2.0, but without the fixed-function component, and so was

not backward-compatible with OpenGL ES 1.1. OpenGL ES 3.0 was released in 2012,

providing functionality such as multiple render targets, texture compression,

transform feedback, instancing, and a much wider range of texture formats and

modes, as well as shader language improvements. OpenGL ES 3.1 adds compute

shaders, and 3.2 adds geometry and tessellation shaders, among other features.

Chapter 23 discusses mobile device architectures in more detail.

移动设备标准开始使用OpenGL ES。“ES”代表着嵌入式系统，因为这个API是为了移动设备开发的。当时标准OpenGL调用结构相当笨重、缓慢，并且支持很少的使用功能。OpenGL ES 1.0发布于2003年，是OpenGL 1.3的精简版，描述了固定功能的流水管线。虽然DirectX的发布与支持他们的图形硬件同步，但是为移动设备开发图形支持并不是以同一方式进行的。例如，第一个平板，发布于2010年，实现了OpenGL ES 1.1。在2007年OpenGL ES 2.0标准发布，提供了可编程着色器。他是基OpenGL 2.0，但是并没有固定功能组件，因此不能向后兼容OpenGL ES 1.1。OpenGL ES 3.0于2012年发布，提供了比如多重渲染目标，纹理压缩，转换反馈，实例化和更大范围的纹理格式和模型功能，以及着色器语言的改善。OpenGL ES 3.1增加了计算着色器，3.2增加了几何和镶嵌着色器，以及其他特性。第23章将更详细的讨论移动设备架构。

An offshoot of OpenGL ES is the browser-based API WebGL, called through

JavaScript. Released in 2011, the first version of this API is usable on most mobile

devices, as it is equivalent to OpenGL ES 2.0 in functionality. As with OpenGL,

extensions give access to more advanced GPU features. WebGL 2 assumes OpenGL

ES 3.0 support.

OpenGL ES的衍生物是基于浏览器的API WebGL,通过JavaScript调用。2011年发布的第一个版本用于大部分移动设备，功能上相当于OpenGL ES 2.0。与OpenGL一样，扩展允许访问更高级的GPU特性。WebGL 2假定支持OpenGL ES 3.0。

WebGL is particularly well suited for experimenting with features or use in the

classroom:

• It is cross-platform, working on all personal computers and almost all mobile devices.

• Driver approval is handled by the browsers. Even if one browser does not support a particular GPU or extension, often another browser does.

• Code is interpreted, not compiled, and only a text editor is needed for development.

• A debugger is built in to most browsers, and code running at any website can be examined.

• Programs can be deployed by uploading them to a website or Github, for example.

WebGL特别适合用在以下场景：

1. 他是跨平台的，可以工作在所有的私人电脑和几乎所有移动设备。
2. 驱动支持是由浏览器掌控。即使一个浏览器不支持一个独特的GPU或拓展，另一个浏览可能就支持。
3. 编码是解释型的，不是编译型的，开发者只需要一个文本编辑器。
4. 调试器建立在大部分浏览器，运行在任何网站的编码都可被检测。
5. 程序通过上传到一个网站或Github上部署。

Higher-level scene-graph and effects libraries such as three.js [218] give easy access

to code for a variety of more involved effects such as shadow algorithms, post-

processing effects, physically based shading, and deferred rendering.

更高层级的场景图和效果库比如three.js[218]为各种复杂的效果比如阴影算法，后处理效果，基于物理渲染和延迟渲染提供了更方便的代码访问。

3.5 The Vertex Shader 顶点着色器

The vertex shader is the first stage in the functional pipeline shown in Figure 3.2.

While this is the first stage directly under programmer control, it is worth noting that

some data manipulation happens before this stage. In what DirectX calls the input

assembler [175, 530, 1208], several streams of data can be woven together to form

the sets of vertices and primitives sent down the pipeline. For example, an object

could be represented by one array of positions and one array of colors. The input

assembler would create this object’s triangles (or lines or points) by creating vertices

with positions and colors. A second object could use the same array of positions

(along with a different model transform matrix) and a different array of colors for its

representation. Data representation is discussed in detail in Section 16.4.5. There is

also support in the input assembler to perform instancing. This allows an object to be

drawn several times with some varying data per instance, all with a single draw call.

The use of instancing is covered in Section 18.4.2.

图3.2所示的顶点着色器是功能流水管线的第一个阶段。虽然这是程序员控制下的第一个阶段，但是值得注意的是，一些数据操作是在这个阶段之前发生的。在DirectX称为输入汇编程序中[175,530,1208]，几个数据流可以被交织在一起形成一系列顶点和片元发送到流水管线下端。例如，一个物体可以被一个位置数组和一个颜色数组表示。输入汇编程序可以通过创建带有位置和颜色的顶点创建这个物体的三角形（或者线或者点集）。第二个物体可以使用同样的位置数组（一个不同的模型转换矩阵）和不同颜色数组。数据表示将在章节16.4.5详细讨论。输入汇编程序同样支持执行实例化。这允许每个实例使用不同的数据多次绘制物体，所有数据使用一个绘制命令。实例化的使用详见章节18.4.2。

A triangle mesh is represented by a set of vertices, each associated with a specific

position on the model surface. Besides position, there are other optional properties

associated with each vertex, such as a color or texture coordinates. Surface normals

are defined at mesh vertices as well, which may seem like an odd choice.

Mathematically, each triangle has a well-defined surface normal, and it may seem to

make more sense to use the triangle’s normal directly for shading. However, when

rendering, triangle meshes are often used to represent an underlying curved surface,

and vertex normals are used to represent the orientation of this surface, rather than

that of the triangle mesh itself. Section 16.3.4 will discuss methods to compute

vertex normals. Figure 3.7 shows side views of two triangle meshes that represent

curved surfaces, one smooth and one with a sharp crease.

一个三角形由一组顶点表示，每一个都与模型表面的一个特定位置相联系。除了位置，每个顶点还有其他的可选的属性，比如颜色或者纹理坐标。表面法线也定义在网格顶点中，可能看起来是一个奇怪的选择。数学上，每个三角形有一个定义分明的表面法线，使用三角形法线直接进行着色似乎更有意义。然而，在渲染中，三角形网格通常用来显示一个底层曲面，并且顶点法线通常用来显示表面的方向，而不是三角形网格自己。章节16.3.4将讨论方法计算顶点法线。图3.7显示了两个三角形显示的曲面，一个平滑一个尖锐。

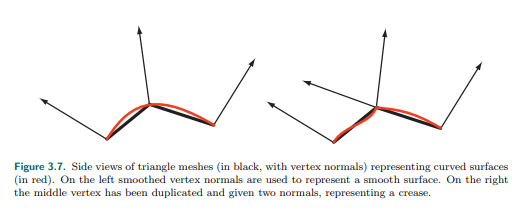


图3.7 三角形网格（黑色，以及他的顶点法线）表示曲面（红色）的侧视图。左侧顶点法线用来显示一个平滑的表面，右侧顶点被复制且给出了两个法系，表示一个折痕。

The vertex shader is the first stage to process the triangle mesh. The data describing

what triangles are formed is unavailable to the vertex shader. As its name implies, it

deals exclusively with the incoming vertices. The vertex shader provides a way to

modify, create, or ignore values associated with each triangle’s vertex, such as its

color, normal, texture coordinates, and position. Normally the vertex shader program

transforms vertices from model space to homogeneous clip space (Section 4.7). At a

minimum, a vertex shader must always output this location.

顶点着色器是第一个阶段处理三角形网格。对于顶点着色器来说三角形形成的数据描述是难以获得的。正如其名字所示，他只处理传入的顶点。顶点着色器提供了一个方法去修改、创建或者忽略与每个三角形顶点相联系的值，例如颜色，法线，纹理坐标或者位置。通常顶点着色器程序将顶点从模型空间转换到齐次裁剪空间（章节4.7）。至少，一个顶点着色器必须输出他的位置。

A vertex shader is much the same as the unified shader described earlier. Every

vertex passed in is processed by the vertex shader program, which then outputs a

number of values that are interpolated across a triangle or line. The vertex shader

can neither create nor destroy vertices, and results generated by one vertex cannot

be passed on to another vertex. Since each vertex is treated independently, any

number of shader processors on the GPU can be applied in parallel to the incoming

stream of vertices.

顶点着色器与之前描述的同一着色器非常相似。每个顶点通过顶点着色器程序处理，然后输出一些值，这些值在三角形或线上进行插值运算。顶点着色器既不能创建和销毁顶点，也不能从一个顶点生成的结果传递给另外一个顶点。由于每个顶点都是被独立处理，GPU上的任何数量的着色器程序都可以并行的应用于传入的顶点流。

Input assembly is usually presented as a process that happens before the vertex

shader is executed. This is an example where the physical model often differs from

the logical. Physically, the fetching of data to create a vertex might happen in the

vertex shader and the driver will quietly prepend every shader with the appropriate

instructions, invisible to the programmer.

输入程序集通常表示为顶点着色器执行之前发生的一个过程。在这个例子中，物理模型通常与逻辑模型不同。物理上，获取数据来创建一个顶点可能发生在顶点着色器中，驱动程序会悄悄地在每个着色器前加上适当的指令，而程序员是看不到这些指令的。

Chapters that follow explain several vertex shader effects, such as vertex blending

for animating joints, and silhouette rendering. Other uses for the vertex shader

include:

• Object generation, by creating a mesh only once and having it be deformed by the

vertex shader.

• Animating character’s bodies and faces using skinning and morphing techniques.

• Procedural deformations, such as the movement of flags, cloth, or water [802, 943].

• Particle creation, by sending degenerate (no area) meshes down the pipeline and

having these be given an area as needed.

• Lens distortion, heat haze, water ripples, page curls, and other effects, by using

the entire framebuffer’s contents as a texture on a screen-aligned mesh undergoing

procedural deformation.

• Applying terrain height fields by using vertex texture fetch [40, 1227].

接下来的章节解释了几个顶点着色器特效，比如动画关节的顶点混合，和轮廓渲染。其他的顶点着色器应用包括：

1. 对象生成，通过创建一个网格一次并是他被顶点着色器变形。
2. 使用变形和皮肤技术动画角色的身体和脸。
3. 程序上的变形，比如旗帜、布或者水的移动。
4. 粒子生成，通过发送退化（无区域）网格到流水管线下端，并依需要给定一定区域。
5. 镜头变形，热雾，水波纹，页面卷曲以及其他的效果，通过使用整个帧缓冲生成一个屏幕对齐的纹理去进行程序变形。
6. 通过顶点纹理获取应用地形高度领域[40,1227]。

Some deformations done using a vertex shader are shown in Figure 3.8.

图3.8展示了一个使用顶点着色器变形的例子。

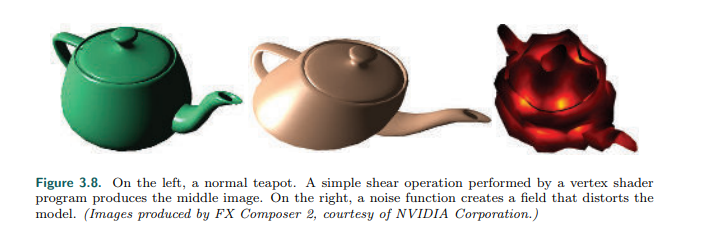


图3.8 在左侧，普通的茶壶。一个顶点着色器程序通过简单的裁剪操作执行造成了中间的图像。在右侧，一个噪声函数创建一个字段扭曲了模型。（图片由FX Composer2制作，由英伟达公司提供。）

The output of the vertex shader can be consumed in several different ways. The

usual path is for each instance’s primitives, e.g., triangles, to then be generated and

rasterized, and the individual pixel fragments produced to be sent to the pixel shader

program for continued processing. On some GPUs the data can also be sent to the

tessellation stage or the geometry shader or be stored in memory. These optional

stages are discussed in the following sections.

顶点着色器的输出有几种不同的方式使用。通常的路径是每个图元实例，比如三角形，然后生成和光栅化，并且生成的单个像素片段发送到像素着色器程序去继续处理。在一些GPU上，数据也可以被送到镶嵌阶段或几何着色器或者存储在内存中。这些可选的阶段会在接下来的部分讨论。

3.6 The Tessellation Stage 镶嵌阶段

The tessellation stage allows us to render curved surfaces. The GPU’s task is to take

each surface description and turn it into a representative set of triangles. This stage

is an optional GPU feature that first became available in (and is required by) DirectX

11. It is also supported in OpenGL 4.0 and OpenGL ES 3.2.

镶嵌阶段允许我们渲染曲面。GPU的任务是将每个表面描述转换成一组代表性的三角形。这个阶段是一个可选的GPU功能，第一次是在DirectX 11中可用（并且是所要求）。他同样被OpenGL 4.0 和OpenGL ES 3.2支持。

There are several advantages to using the tessellation stage. The curved surface

description is often more compact than providing the corresponding triangles

themselves. Beyond memory savings, this feature can keep the bus between CPU

and GPU from becoming the bottleneck for an animated character or object whose

shape is changing each frame. The surfaces can be rendered efficiently by having an

appropriate number of triangles generated for the given view. For example, if a ball

is far from the camera, only a few triangles are needed. Up close, it may look best

represented with thousands of triangles. This ability to control the level of detail can

also allow an application to control its performance, e.g., using a lower-quality mesh

on weaker GPUs in order to maintain frame rate. Models normally represented by flat

surfaces can be converted to fine meshes of triangles and then warped as desired

[1493], or they can be tessellated in order to perform expensive shading

computations less frequently [225].

使用镶嵌阶段有几个优点。曲面描述常常比提供相应的三角形更加紧密。除了节省内存，这种特性在角色或物体动画在每一帧都更新形状的情况下可以防止CPU和GPU之间运转称为角色或物体动画的瓶颈。通过给定视图生成适当数量的三角形可以有效的呈现曲面。例如，如果一个球距离摄像机很远，仅仅只需要显示一些三角形。近距离观看，为了更好的呈现，需要上千的三角形。这种控制细节等级的能力同样也允许应用程序控制他的性能。比如，在低性能GPU上使用低质量的网格来保持帧率。通常由平面表示的模型可以转换为由三角形精细网格表示，然后根据我们的需要进行弯曲[1493]，或者他们可以被镶嵌以便不用那么频繁的执行昂贵的着色计算[225]。

The tessellation stage always consists of three elements. Using DirectX’s terminology,

these are the hull shader, tessellator, and domain shader. In OpenGL the hull shader

is the tessellation control shader and the domain shader the tessellation evaluation

shader, which are a bit more descriptive, though verbose. The fixed-function

tessellator is called the primitive generator in OpenGL, and as will be seen, that is

indeed what it does.

镶嵌阶段通常包括三个元素。用DirectX的术语，他们是外壳着色器，曲面细分和域着色器。在OpenGL外壳着色器是镶嵌控制着色器以及域着色器是镶嵌评估着色器，有更多一点的描述，尽管冗杂。固定功能曲面细分在OpenGL中称为图元生成器，我们将看到，它确实是这样做的。

How to specify and tessellate curves and surfaces is discussed at length in Chapter

17. Here we give a brief summary of each tessellation stage’s purpose. To begin, the

input to the hull shader is a special patch primitive. This consists of several control

points defining a subdivision surface, B´ezier patch, or other type of curved element.

The hull shader has two functions. First, it tells the tessellator how many triangles

should be generated, and in what configuration. Second, it performs processing on

each of the control points. Also, optionally, the hull shader can modify the incoming

patch description, adding or removing control points as desired. The hull shader

outputs its set of control points, along with the tessellation control data, to the

domain shader. See Figure 3.9.

详细说明和镶嵌曲面和表面将在第17章说明。这里我们给出每个镶嵌阶段目的简明总结。首先输入外壳着色器的是一个特殊的补丁图元。他包含了一些定义了曲面的控制点，B´ezier补丁，或者其他类型的曲面元素。外壳着色器有两个功能。首先，他告诉曲面细分有多少三角形需要生成，以及他们的结构。第二，他对每个控制点执行处理。同样，可选择性的，外壳着色器可以更改传进来的补丁描述，如愿增加或者删除控制点。外壳着色器输出一系列的控制点，以及曲面细分控制数据，传到域着色器。见图3.9。

The tessellator is a fixed-function stage in the pipeline, only used with tessellation

shaders. It has the task of adding several new vertices for the domain shader to

process. The hull shader sends the tessellator information about what type of

tessellation surface is desired: triangle, quadrilateral, or isoline. Isolines are sets of

line strips, sometimes used for hair rendering [1954]. The other important values

sent by the hull shader are the tessellation factors (tessellation levels in OpenGL).

These are of two types: inner and outer edge. The two inner factors determine how

much tessellation occurs inside the triangle or quadrilateral. The outer factors

determine how much each exterior edge is split (Section 17.6). An example of

increasing tessellation factors is shown in Figure 3.10. By allowing separate controls,

we can have adjacent curved surfaces’ edges match in tessellation, regardless of

how the interiors are tessellated. Matching edges avoids cracks or other shading

artifacts where patches meet. The vertices are assigned barycentric coordinates

(Section 22.8), which are values that specify a relative location for each point on the

desired surface.

曲面细分在流水管线中是固定功能阶段，只与镶嵌着色器一起使用。他的任务是为了域着色器的处理添加新的顶点。外壳着色器发送有关曲面细分表面所需的曲面细分信息：三角形、四边形或者等值线。等值线是一组线带，有时候用于头发渲染[1954]。其他被外壳着色器发送的重要的值是曲面细分因子（OpenGL称为曲面细分等级）。有两个种类：内部或者外部边缘。两个内部因素决定了三角形或四边形发生多少曲面细分。外部因素决定了每条外部边缘被分割多少（章节17.6）。一个关于增加曲面细分因子的例子如图3.10所示。通过允许独立控制，我们可以让相邻的曲面边缘在曲面区分中匹配，忽略内部是如何曲面细分的。边缘匹配避免了裂缝或者或者补丁遇到的其他着色问题。顶点被重心坐标分配（章节22.8），这些值详细指定了所需表面的每个点的相对位置。

The hull shader always outputs a patch, a set of control point locations. However, it

can signal that a patch is to be discarded by sending the tessellator an outer

tessellation level of zero or less (or not-a-number, NaN). Otherwise, the tessellator

generates a mesh and sends it to the domain shader. The control points for the

curved surface from the hull shader are used by each invocation of the domain

shader to compute the output values for each vertex. The domain shader has a data

flow pattern like that of a vertex shader, with each input vertex from the tessellator

being processed and generating a corresponding output vertex. The triangles formed

are then passed on down the pipeline.

外壳着色器常常输出补丁，一系列控制点的位置。但是，他可以通过发送曲面细分一个外部曲面细分等级为0或者更小（或者不是数字，NaN）来发出一个要丢弃补丁的信号。否则，曲面细分生成网格并将它发送给域着色器。外壳着色器中的曲面控制点被用来域着色器的每次调用去计算每个顶点的输出值。域着色器有一个类似顶点着色器的数据流模式，每个从曲面细分输入的顶点都会被处理并生成相应的输出顶点。然后形成的三角形沿着管道传递下去。

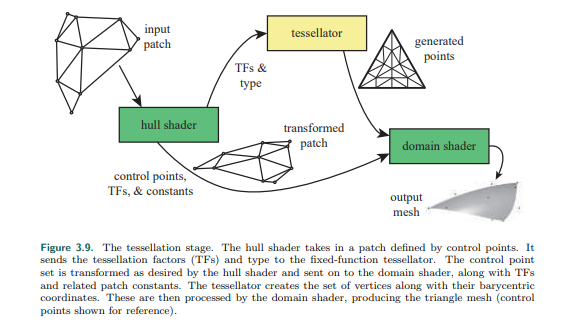


图3.9 镶嵌阶段。外壳着色器接受一个由控制点定义的补丁。他发送曲面细分因子（TFs）和类型到固定功能曲面细分。控制点可以如愿被外壳着色器转换传送到域着色器，以及包括曲面细分因子和相关的补丁。曲面细分创建了一系列的顶点以及他们的重心坐标。这些接下来要被域着色器处理，产生三角形网格（显示的控制点以供参考）。

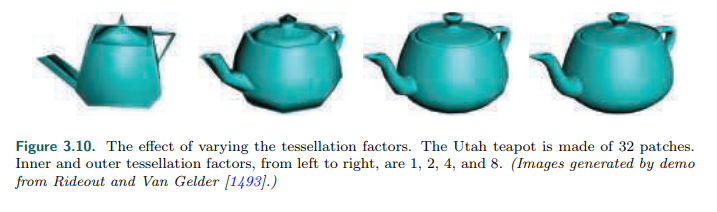


图3.10 不同的曲面细分因子的效果。茶壶是由32个补丁组成。内部和外部曲面细分因子从左到右分别是1,2,4和8.（Rideout 和Van Gelder的图例[1493]）

While this system sounds complex, it is structured this way for efficiency, and each

shader can be fairly simple. The patch passed into a hull shader will often undergo

little or no modification. This shader may also use the patch’s estimated distance or

screen size to compute tessellation factors on the fly, as for terrain rendering [466].

Alternately, the hull shader may simply pass on a fixed set of values for all patches

that the application computes and provides. The tessellator performs an involved but

fixed-function process of generating the vertices, giving them positions, and

specifying what triangles or lines they form. This data amplification step is performed

outside of a shader for computational efficiency [530]. The domain shader takes the

barycentric coordinates generated for each point and uses these in the patch’s

evaluation equation to generate the position, normal, texture coordinates, and other

vertex information desired. See Figure 3.11 for an example.

虽然这个系统听起来很复杂，但是为了提高效率，他的结构是这样的，而且每个着色器都相当简单。传递到外壳着色器的补丁常常很少甚至没有修改。着色器常常使用补丁预估距离或者屏幕尺寸去实时计算曲面细分因子，比如地形渲染[466]。另外外壳着色器可以简单的为应用程序计算和提供的所有补丁传递一组固定的值。曲面细分执行复杂但是固定功能的过程，生成顶点，给他们位置以及详细说明形成的三角形或线为了计算效率，这些数据详述步骤是在着色器外部执行的[530]。域着色器接受每个顶点生成的重心坐标并用这些在补丁评估方程中生成位置、发现、纹理坐标和其他需要的顶点信息。如图3.11所示。

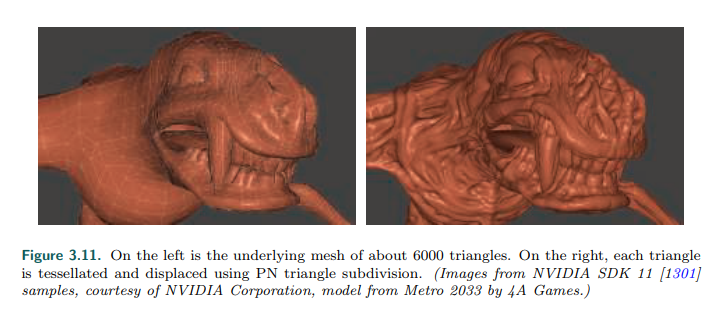


图3.11 在坐标潜在的网格大概是6000个三角形。造右边，每个三角形被曲面细分并用PN三角形细分曲面显示。（图片来自英伟达SDK11[1301]，样品由英伟达公司提供，模型来自4A Games的Metro2033）

3.7 The Geometry Shader 几何着色器

The geometry shader can turn primitives into other primitives, something the

tessellation stage cannot do. For example, a triangle mesh could be transformed to a

wireframe view by having each triangle create line edges. Alternately, the lines could

be replaced by quadrilaterals facing the viewer, so making a wireframe rendering

with thicker edges [1492]. The geometry shader was added to the hardware-

accelerated graphics pipeline with the release of DirectX 10, in late 2006. It is

located after the tessellation shader in the pipeline, and its use is optional. While a

required part of Shader Model 4.0, it is not used in earlier shader models. OpenGL

3.2 and OpenGL ES 3.2 support this type of shader as well.

几何着色器可以将图元转换成其他图元，一些曲面细分阶段不能做的。例如，一个三角形网格可以通过每个三角形创建线边缘转换成线框视角。另外，线可以为四边形替代，使线框由粗的边缘渲染[1492]。几何着色器是随着2006年底DirectX 10的发布被加进硬件加速图形管线。在流水管线中，他位于曲面细分着色器的后面，并且是可选的。他是着色器模型4.0的需求部分，更早的着色器模型并不能用。OpenGL3.2和OpenGL ES3.2同样支持这种类型的着色器。

The input to the geometry shader is a single object and its associated vertices. The

object typically consists of triangles in a strip, a line segment, or simply a point.

Extended primitives can be defined and processed by the geometry shader. In

particular, three additional vertices outside of a triangle can be passed in, and the

two adjacent vertices on a polyline can be used. See Figure 3.12. With DirectX 11

and Shader Model 5.0, you can pass in more elaborate patches, with up to 32 control

points. That said, the tessellation stage is more efficient for patch generation [175].

输入到几何着色器的是单个的模型以及他关联的顶点。对象通常由三角形带、线段或者简单的点组成。拓展图元可以有几何着色器定义和处理。特别的，三角形外的三个额外顶点可以传入，并且多线段上的两个相邻顶点可以使用。如图3.12,。通过DirectX 11和着色器模型5.0，你可以传入更复杂的补丁，以及上限32个控制点。也就是说，曲面细分阶段对于补丁生成更有效。

The geometry shader processes this primitive and outputs zero or more vertices,

which are treated as points, polylines, or strips of triangles. Note that no output at all

can be generated by the geometry shader. In this way, a mesh can be selectively

modified by editing vertices, adding new primitives, and removing others.

几何着色器处理图元并输出0或者更多的顶点，这些顶点被视为点、折线或三角形带。注意，几何着色器可以不生成任何输出。在这种方式下，网格可以通过编辑顶点，增加新的图元或删除去修改。

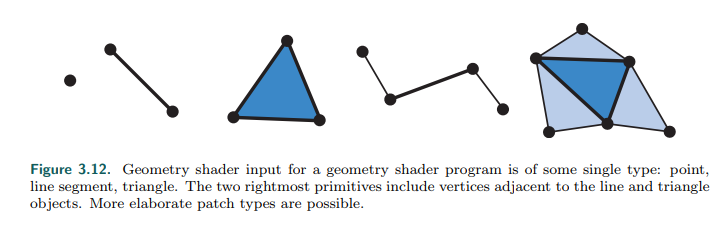


图3.12 几何着色器输入一些单一类型：点、线段、三角形。2个右侧图元包括线的相邻顶点和三角形对象。更复杂的补丁类型是可能的。

The geometry shader is designed for modifying incoming data or making a limited

number of copies. For example, one use is to generate six transformed copies of

data to simultaneously render the six faces of a cube map; see Section 10.4.3. It can

also be used to efficiently create cascaded shadow maps for high-quality shadow

generation. Other algorithms that take advantage of the geometry shader include

creating variablesized particles from point data, extruding fins along silhouettes for

fur rendering, and finding object edges for shadow algorithms. See Figure 3.13 for

more examples. These and other uses are discussed throughout the rest of the book.

几何着色器是为了更改输入的数据或者制作有限数量的副本而设计的。比如，一个用途是生成6个转换后的数据副本去同时渲染立方体纹理的6面，见章节10.4.3.他也可以用在高效创建串联的阴影纹理以生成高质量阴影。其他利用几何着色器的算法包括从点数据创建可变的粒子，沿着轮廓挤压鳍片用于皮毛渲染，以及为阴影算法寻找物体边缘。图3.13显示更多例子。在本书的其余部分，我们将讨论这些和其他用途。

DirectX 11 added the ability for the geometry shader to use instancing, where the

geometry shader can be run a set number of times on any given primitive [530,

1971]. In OpenGL 4.0 this is specified with an invocation count. The geometry

shader can also output up to four streams. One stream can be sent on down the

rendering pipeline for further processing. All these streams can optionally be sent to

stream output render targets.

DirectX 11增加了几何着色器使用实例化的能力，其中几何着色器可以在任何给定的图元上运行一组数据次数[530,1971]。在OpenGL 4.0，这是用一个调用计数指定的。几何着色器可以输出多达4个流。一个流可以沿着渲染管道发送，以便进一步处理。所有的流可以选择性的被传送到流输出去渲染目标。

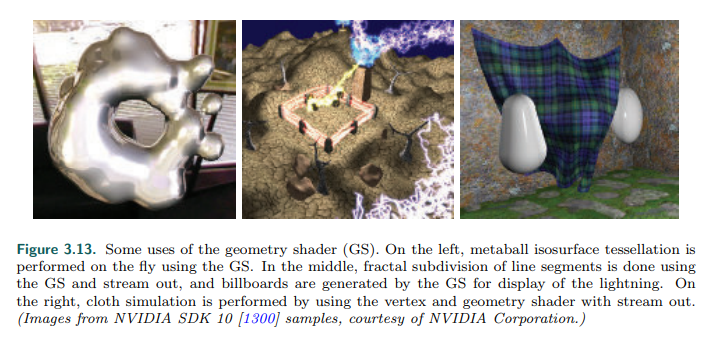


图3.13 一些使用几何着色器的例子（GS）。左侧，金属球等值面曲面细分被GS动态执行。中间，使用GS进行线段不规则曲面细分以及流输出，广告牌被GS生成以显示闪电。右侧，通过使用顶点和几何着色器和流输出执行布料模拟（图片来自英伟达SDK10[1300]样本，由英伟达公司提供）

The geometry shader is guaranteed to output results from primitives in the same

order that they are input. This affects performance, because if several shader cores

run in parallel, results must be saved and ordered. This and other factors work

against the geometry shader being used to replicate or create a large amount of

geometry in a single call [175, 530].

几何着色器确保输出图元的结果与输入时候的顺序是一样的。这会影响性能，因为如果一些着色器代码是在并行中运行，结果必须被保存以及排序。这一点和其他因素都不利于几何着色器在一次调用中辅助或创建大量几何图形。

After a draw call is issued, there are only three places in the pipeline where work can

be created on the GPU: rasterization, the tessellation stage, and the geometry

shader. Of these, the geometry shader’s behavior is the least predictable when

considering resources and memory needed, since it is fully programmable. In

practice the geometry shader usually sees little use, as it does not map well to the

GPU’s strengths. On some mobile devices it is implemented in software, so its use is

actively discouraged there [69].

在绘制命令发出后，在GPU中流水管线中只有三个位置可以创建工作：光栅化，曲面细分阶段和几何着色器。其中，几何着色器的行为是最不可预测的，需要考虑资源和内存，因为他是完全可编程的。实际上，几何着色器通常很少使用，因为他不能很好的映射出GPU的优势。在一些移动设备上，他是在软件中实现的，因此不鼓励使用他[69]。

3.7.1 Stream Output 流输出

The standard use of the GPU’s pipeline is to send data through the vertex shader,

then rasterize the resulting triangles and process these in the pixel shader. It used to

be that the data always passed through the pipeline and intermediate results could

not be accessed. The idea of stream output was introduced in Shader Model 4.0.

After vertices are processed by the vertex shader (and, optionally, the tessellation

and geometry shaders), these can be output in a stream, i.e., an ordered array, in

addition to being sent on to the rasterization stage. Rasterization could, in fact, be

turned off entirely and the pipeline then used purely as a non-graphical stream

processor. Data processed in this way can be sent back through the pipeline, thus

allowing iterative processing. This type of operation can be useful for simulating

flowing water or other particle effects, as discussed in Section 13.8. It could also be

used to skin a model and then have these vertices available for reuse (Section 4.4).

GPU流水管线的标准使用是发送数据通过顶点着色器，然后光栅化结果三角形然后再像素着色器中处理。过去，数据总是通过流水管线传递并且中间结果不能被访问。着色器模型4.0引入了流输出的理念。在顶点着色器处理过顶点之后（以及，可选的着色器，曲面细分着色器和几何着色器），这些顶点数据可以输出到一个流中，即，一个顺序数组，附加的传送到光栅化阶段。实际上，光栅化可以完全关闭，流水管线纯粹作为一个无图像流处理器。这种方式处理的数据可以通过流水管线返回，从而允许迭代处理。这种类型的操作可以由于模拟水流动或者其他的粒子效果，如章节13.8讨论。同样可以用在模型皮肤，然后可以重复使用这些顶点（章节4.4）。

Stream output returns data only in the form of floating point numbers, so it can have

a noticeable memory cost. Stream output works on primitives, not directly on

vertices. If meshes are sent down the pipeline, each triangle generates its own set of

three output vertices. Any vertex sharing in the original mesh is lost. For this reason

a more typical use is to send just the vertices through the pipeline as a point set

primitive. In OpenGL the stream output stage is called transform feedback, since the

focus of much of its use is transforming vertices and returning them for further

processing. Primitives are guaranteed to be sent to the stream output target in the

order that they were input, meaning the vertex order will be maintained [530].

流输出返回的数据都是浮点类型的数据，因此他会有显著的内存消耗。流输出是在图元上工作，而不是直接在顶点上。如果网格被传到流水管线下端，每个三角形都会生成他自己的三个输出顶点集。原始网格上的任何共享顶点都将丢失。由于这个原因，更典型的使用是仅仅将顶点作为顶点图元集通过流水管线传送。在OpenGL中流输出阶段被称为转换反馈，因为他的主要用途是转换顶点并返回他们以做进一步的处理。确保图元发送到流输出目标是以他们输入的顺序，意味着顶点顺序将维持不变[530]。

3.8 The Pixel Shader 像素着色器

After the vertex, tessellation, and geometry shaders perform their operations, the

primitive is clipped and set up for rasterization, as explained in the previous chapter.

This section of the pipeline is relatively fixed in its processing steps, i.e., not

programmable but somewhat configurable. Each triangle is traversed to determine

which pixels it covers. The rasterizer may also roughly calculate how much the

triangle covers each pixel’s cell area (Section 5.4.2). This piece of a triangle partially

or fully overlapping the pixel is called a fragment.

在顶点、曲面细分和几何着色器执行他们的操作之后，图元将被裁剪以及为光栅化设置，正如之前章节所介绍的。流水管线的这一部分在他的处理步骤上是相对固定的，即不可变成但是一些可以配置。每个三角形将被遍历去决定他们覆盖的像素。光栅化也可以粗略的计算出三角形覆盖每个像素单元的面积（章节5.4.2）。部分或完全重叠像素的三角形这一部分称为片元。

The values at the triangle’s vertices, including the z-value used in the z-buffer, are

interpolated across the triangle’s surface for each pixel. These values are passed to

the pixel shader, which then processes the fragment. In OpenGL the pixel shader is

known as the fragment shader, which is perhaps a better name. We use “pixel

shader” throughout this book for consistency. Point and line primitives sent down the

pipeline also create fragments for the pixels covered.

在三角形顶点上的值，包括用在z缓冲区的z值，将会通过三角形表面，为每个像素插值运算。这些值会传递到像素着色器，然后处理片元。在OpenGL中像素着色器也被称为片元着色器，也许是一个更好的名字。在这本书中，我们使用像素着色器。点和线片元也会通过流水管线传送并生成覆盖每个像素的片元。

The type of interpolation performed across the triangle is specified by the pixel

shader program. Normally we use perspective-correct interpolation, so that the

worldspace distances between pixel surface locations increase as an object recedes

in the distance. An example is rendering railroad tracks extending to the horizon.

Railroad ties are more closely spaced where the rails are farther away, as more

distance is traveled for each successive pixel approaching the horizon. Other

interpolation options are available, such as screen-space interpolation, where

perspective projection is not taken into account. DirectX 11 gives further control over

when and how interpolation is performed [530].

通过三角形执行的插值类型是被像素着色器程序指定。通常我们使用透视校正插值，这样随着物体距离上后退，像素表面位置之间的世界空间距离将增加。一个例子是渲染铁路铁轨在地平线上延伸。铁轨越远他们之间的距离就越近，因为每一个接近地平线的连续像素移动的距离就越大。其他插值运算选项是可用的，比如屏幕空间插值，其中不考虑透视投影。DirectX11在什么时间以及如何插值执行给了进一步的控制[530]。

In programming terms, the vertex shader program’s outputs, interpolated across the

triangle (or line), effectively become the pixel shader program’s inputs. As the GPU

has evolved, other inputs have been exposed. For example, the screen position of

the fragment is available to the pixel shader in Shader Model 3.0 and beyond. Also,

which side of a triangle is visible is an input flag. This knowledge is important for

rendering a different material on the front versus back of each triangle in a single

pass.

在编程术语中，顶点着色器程序的输出，通过三角形（或线）的插值运算，有效的变成像素着色器程序的输入。随着GPU的演变，其他输出也被公开。举个例子，在着色器模型3.0或者之上版本，像素着色器可以使用片元的屏幕位置。同样，三角形的哪条边是可见的是输入标志。在单一通道中，这一知识对于每个三角形的正面和背面不同材质的渲染是重要的。

With inputs in hand, typically the pixel shader computes and outputs a fragment’s

color. It can also possibly produce an opacity value and optionally modify its z-depth.

During merging, these values are used to modify what is stored at the pixel. The

depth value generated in the rasterization stage can also be modified by the pixel

shader. The stencil buffer value is usually not modifiable, but rather it is passed

through to the merge stage. DirectX 11.3 allows the shader to change this value.

Operations such as fog computation and alpha testing have moved from being

merge operations to being pixel shader computations in SM 4.0 [175].

使用输入，通常像素着色器计算并输出片元的颜色。他同样可能产生不透明值以及可选择性的更改他的z深度。在合并过程中，这些值被用来修改像素中储存的值。光栅化阶段生成的深度值同样可以被像素着色器修改。模板缓冲区的值通常是不可修改的，但是他会被传递到合并阶段。DirectX11.3允许着色器更改他的值。在着色器模型4.0，比如雾的计算和透明度测试已经从合并操作转变为像素着色器计算。

A pixel shader also has the unique ability to discard an incoming fragment, i.e.,

generate no output. One example of how fragment discard can be used is shown in

Figure 3.14. Clip plane functionality used to be a configurable element in the fixed

function pipeline and was later specified in the vertex shader. With fragment discard

available, this functionality could then be implemented in any way desired in the

pixel shader, such as deciding whether clipping volumes should be AND’ed or OR’ed

together.

像素着色器具有独有的能力是丢弃一个到来的片元，即，生成但是不输出。一个例子是如图3.14所示，片元丢弃是如何被使用的。裁剪平面功能过去在固定功能流水管线中是可配置的元素，后来在顶点着色器中指定。随着片元丢弃可用，这个功能在像素着色器中可以被任何所需的方式执行，比如决定裁剪视锥体是否和ed或者或ed一起。

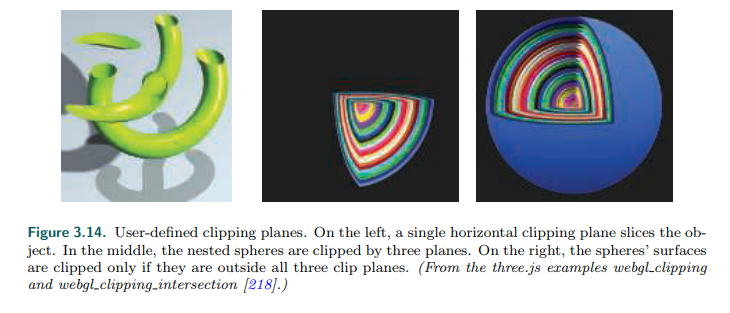


图3.14 用户定义的裁剪平面。在左边，单一水平裁剪平面裁剪物体。中间部分，网格球被三个平面裁剪。在右边，球表面只有当他们位于三个裁减平面外部时才会被裁减（来自three.js例子，webgl\_裁减和webgl裁减交集）

Initially the pixel shader could output to only the merging stage, for eventual display.

The number of instructions a pixel shader can execute has grown considerably over

time. This increase gave rise to the idea of multiple render targets (MRT). Instead of

sending results of a pixel shader’s program to just the color and z-buffer, multiple

sets of values could be generated for each fragment and saved to different buffers,

each called a render target. Render targets generally have the same x- and y-

dimensions; some APIs allow different sizes, but the rendered area will be the

smallest of these. Some architectures require render targets to each have the same

bit depth, and possibly even identical data formats. Depending on the GPU, the

number of render targets available is four or eight.

最初，像素着色器仅仅只能输出到合并阶段为了最终的显示。随着时间的推移，像素着色器可以执行的指令数量在增加。这种增加产生了多种渲染目标的想法。不同于仅仅将像素着色器程序的结果发送到颜色和深度缓冲区，可以为每个片元生成多重设置数值并保存到不同的缓冲区，每个缓冲区成为渲染目标。渲染目标通常具有相同的x和y轴，一些API允许不同的尺寸，但是渲染区域将是这些中最小的。一些架构要求每个渲染目标具有相同的深度位，并且甚至是相同的数据格式。依据GPU，可用的渲染目标是4个或8个。

Even with these limitations, MRT functionality is a powerful aid in performing

rendering algorithms more efficiently. A single rendering pass could generate a color

image in one target, object identifiers in another, and world-space distances in a

third. This ability has also given rise to a different type of rendering pipeline, called

deferred shading, where visibility and shading are done in separate passes. The first

pass stores data about an object’s location and material at each pixel. Successive

passes can then efficiently apply illumination and other effects. This class of

rendering methods is described in Section 20.1.

尽管具有这些限制，多重渲染目标功能在高效执行渲染算法上是强有力的辅助工具。一个渲染通道可以在一个目标生成颜色图像，另一个生成对象标识符，第三个生成世界空间距离。这种能力产生了不同类型的渲染管道，称为延迟着色，其中可见性和着色是在单独的通道中完成的。第一个通道储存关于每个像素的物体位置和材质数据。接下来的通道可以有效的利用光照和其他特效。这种类型的渲染方法将在章节20.1讲述。

The pixel shader’s limitation is that it can normally write to a render target at only

the fragment location handed to it, and cannot read current results from neighboring

pixels. That is, when a pixel shader program executes, it cannot send its output

directly to neighboring pixels, nor can it access others’ recent changes. Rather, it

computes results that affect only its own pixel. However, this limitation is not as

severe as it sounds. An output image created in one pass can have any of its data

accessed by a pixel shader in a later pass. Neighboring pixels can be processed using

image processing techniques, described in Section 12.1.

像素着色器的限制是他通常只可以将对应位置处理的片元写进一个渲染目标，而不能从相邻像素读取结果。也就是说，当像素着色器程序执行，他不能将输出直接发送给相邻像素，也不能访问其他像素最近的更改。相反，他的计算记过只能影响他自己的像素。然而，这种限制并没有听起来那个严重。在一个通道中创建出的输出图像可以被像素着色器在接下来的通道访问它的任何数据。使用图像处理技术处理相邻像素如章节12.1所述。

There are exceptions to the rule that a pixel shader cannot know or affect

neighboring pixels’ results. One is that the pixel shader can immediately access

information for adjacent fragments (albeit indirectly) during the computation of

gradient or derivative information. The pixel shader is provided with the amounts by

which any interpolated value changes per pixel along the x and y screen axes. Such

values are useful for various computations and texture addressing. These gradients

are particularly important for operations such as texture filtering (Section 6.2.2),

where we want to know how much of an image covers a pixel. All modern GPUs

implement this feature by processing fragments in groups of 2 × 2, called a quad.

When the pixel shader requests a gradient value, the difference between adjacent

fragments is returned. See Figure 3.15. A unified core has this capability to access

neighboring data—kept in different threads on the same warp—and so can compute

gradients for use in the pixel shader. One consequence of this implementation is that

gradient information cannot be accessed in parts of the shader affected by dynamic

flow control, i.e., an “if” statement or loop with a variable number of iterations. All

the fragments in a group must be processed using the same set of instructions so

that all four pixels’ results are meaningful for computing gradients. This is a

fundamental limitation that exists even in offline rendering systems [64].

也有像素着色器不能访问或影响相邻像素结果例外规则。一种就是像素着色器在梯度或者导数信息计算时可以立即访问相邻片元的信息（尽管是间接）。像素着色器提供了沿着x和y屏幕轴每像素内插值的变化量。这类数据对于各种计算和纹理寻址都非常有用。这些梯度对于像纹理过滤（章节6.2.2）的操作特别重要，我们想知道一个图片的多少片元覆盖一个像素。所有现代GPU通过处理2x2组的片元成为四方来执行这个特性。当像素着色器需要一个梯度值，将返回相邻片元之间的差值。见图3.15。一个统一的内核具有访问相邻数据的能力——保存在同一线程束的不同线程中——因此可以在像素着色器中使用计算梯度。这种实现的一种结果是梯度信息不能被受动态流程控制影响的像素着色器访问，即if语句或可变数量的迭代循环。所有组内的片元必须使用同一设置的指令进行处理，这样对于计算梯度所有4个像素结果才是有意义的。这是一个基本的限制，即使在离线渲染系统[64]。

DirectX 11 introduced a buffer type that allows write access to any location, the

unordered access view (UAV). Originally for only pixel and compute shaders, access

to UAVs was extended to all shaders in DirectX 11.1 [146]. OpenGL 4.3 calls this a

shader storage buffer object (SSBO). Both names are descriptive in their own way.

Pixel shaders are run in parallel, in an arbitrary order, and this storage buffer is

shared among them.

DirectX 11引入了一种缓冲区类型允许对任何位置进行写访问，即无序访问视图（UAV）。最初只是为了像素和计算着色器服务，在DirectX 11.1中访问UAV拓展到所有的着色器。OpenGL 4.3称这种为着色存储缓冲区对象（SSBO）。两个名字都以各自的方式描述。像素着色器以任意顺序并行运行，并且他们之间共享这个存储缓冲区。

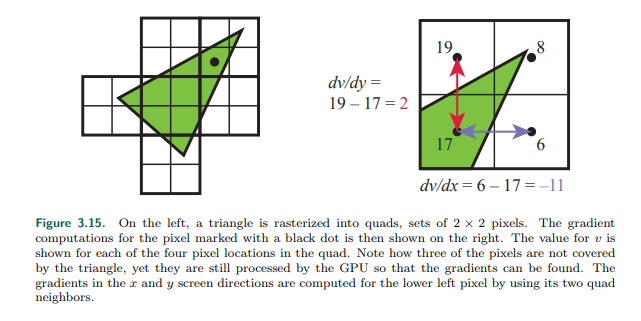
****

图3.15 在左边，一个三角形光栅化进四边形，2x2像素序列。像素梯度计算结果用黑点标识如右边所示。在四方内4个像素位置显示了V的值。注意三个像素没有被三角形覆盖，但他们仍然被GPU处理，这样梯度就可以被找到。左下角像素通过使用它的两个四边形相邻梯度值计算其在x和y屏幕坐标方向的梯度值。

Often some mechanism is needed to avoid a data race condition (a.k.a. a data

hazard), where both shader programs are “racing” to influence the same value,

possibly leading to arbitrary results. As an example, an error could occur if two

invocations of a pixel shader tried to, say, add to the same retrieved value at about

the same time. Both would retrieve the original value, both would modify it locally,

but then whichever invocation wrote its result last would wipe out the contribution of

the other invocation—only one addition would occur. GPUs avoid this problem by

having dedicated atomic units that the shader can access [530]. However, atomics

mean that some shaders may stall as they wait to access a memory location

undergoing read/modify/write by another shader.

一些机制需要去避免数据紊乱（也称为数据风险），当两个着色器程序在竞争影响相同的值，可能会导致任意结果。举个例子，像素着色器的两个指令试图在同一时间添加相同的索引值，可能会产生错误。两者都将检索原始值，都将在本地更改他的值，但是最后一次写入结果的调用会擦除另一个调用的贡献——只有一个添加会成功。GPU通过使用着色器可以访问的专用原子单元来避免问题[530]。然而，原子意味着一些着色器将停滞等待去访问内存位置在其他着色器读/更改/写的时候。

While atomics avoid data hazards, many algorithms require a specific order of

execution. For example, you may want to draw a more distant transparent blue

triangle before overlaying it with a red transparent triangle, blending the red atop

the blue. It is possible for a pixel to have two pixel shader invocations for a pixel,

one for each triangle, executing in such a way that the red triangle’s shader

completes before the blue’s. In the standard pipeline, the fragment results are sorted

in the merger stage before being processed. Rasterizer order views (ROVs) were

introduced in DirectX 11.3 to enforce an order of execution. These are like UAVs;

they can be read and written by shaders in the same fashion. The key difference is

that ROVs guarantee that the data are accessed in the proper order. This increases

the usefulness of these shader-accessible buffers considerably [327, 328]. For

example, ROVs make it possible for the pixel shader to write its own blending

methods, since it can directly access and write to any location in the ROV, and thus

no merging stage is needed [176]. The price is that, if an out-of-order access is

detected, a pixel shader invocation may stall until triangles drawn earlier are

processed.

当原子避免数据风险，程序的一些算法需要一个特定顺序。举个例子，在透明的三角形覆盖之前，你可能想要绘制一个更遥远的透明蓝三角形，将红色混合在蓝色之上。很有可能对于一个像素有两个像素着色器调用一个像素，每个三角形一个，执行这样的方式，红色三角形着色器在蓝色之前完成。在标准流水管线，片元结果在像素处理之前的合并阶段排序。在DirectX11.3引入的光栅化顺序视图来强制执行顺序。就像UAV，他们可以以相同的方式被着色器读写。关键不同点在于ROV确保了数据以正确顺序。这大大的增加了这些着色器可访问缓冲区的有用性[327,328]。举个例子，ROV使像素着色器写自己的混合方法称为可能，因为他可以直接访问ROV的任何位置并对其进行写入，因此不需要合并阶段[176]。代价就是，如果检测到无序访问，像素着色器调用将停滞知道处理前面绘制的三角形。

3.9 The Merging Stage 合并阶段

As discussed in Section 2.5.2, the merging stage is where the depths and colors of

the individual fragments (generated in the pixel shader) are combined with the

framebuffer. DirectX calls this stage the output merger; OpenGL refers to it as per-

sample operations. On most traditional pipeline diagrams (including our own), this

stage is where stencil-buffer and z-buffer operations occur. If the fragment is visible,

another operation that takes place in this stage is color blending. For opaque

surfaces there is no real blending involved, as the fragment’s color simply replaces

the previously stored color. Actual blending of the fragment and stored color is

commonly used for transparency and compositing operations (Section 5.5).

正如章节2.5.2所讨论的，合并阶段是处理单独的片元深度以及颜色值（像素着色器生成）并与帧缓冲区合并的地方。DirectX称这个阶段为输出合并；OpenGL称为逐片元操作。在大部分传统的流水管线图（包括我们自己的），这个阶段就是模板缓冲和深度缓冲操作发生的地方。如果一个片元可见，在这个阶段另一个发生的操作就是颜色混合。对于表面不透明的，没有真正的混合，片元只是简单的替换先前存储的颜色。片元和存储颜色的真正混合通常用在透明度和合成操作（章节5.5）。

Imagine that a fragment generated by rasterization is run through the pixel shader

and then is found to be hidden by some previously rendered fragment when the

zbuffer is applied. All the processing done in the pixel shader was then unnecessary.

To avoid this waste, many GPUs perform some merge testing before the pixel shader

is executed [530]. The fragment’s z-depth (and whatever else is in use, such as the

stencil buffer or scissoring) is used for testing visibility. The fragment is culled if

hidden. This functionality is called early-z [1220, 1542]. The pixel shader has the

ability to change the z-depth of the fragment or to discard the fragment entirely. If

either type of operation is found to exist in a pixel shader program, early-z then

generally cannot be used and is turned off, usually making the pipeline less efficient.

DirectX 11 and OpenGL 4.2 allow the pixel shader to force early-z testing to be on,

though with a number of limitations [530]. See Section 23.7 for more about early-z

and other z-buffer optimizations. Using early-z effectively can have a large effect on

performance, which is discussed in detail in Section 18.4.5.

想象一下一个光栅化生成的片元通过像素着色器当应用了深度发现他会被之前渲染的片元所遮盖。所有的像素着色器中的处理就不是必须的了。为了避免这种浪费，一些GPU在像素着色器执行之前执行一些合并测试[530]。片元的Z-深度值（以及使用的其他东西，比如模板缓冲或裁减）用来测试可见性。如果隐藏则片元被剔除掉。这种功能称为early-z技术[1220,1542].像素着色器具有改变片元Z-深度值或者完全剔除片元的能力。如果任何一种操作存在在像素着色器程序中，early-z通常不能使用，并且被关闭，这通常会降低流水管线的效率。DirectX 11 和 OpenGL 4.2允许像素着色器强制执行early-z测试，尽管有一些限制[530]。有关early-z和其他z缓冲区的优化信息详见章节23.7。使用early-z技术可以对性能产生很大的影响，将在章节18.4.5进行详细讨论。

The merging stage occupies the middle ground between fixed-function stages, such

as triangle setup, and the fully programmable shader stages. Although it is not

programmable, its operation is highly configurable. Color blending in particular can

be set up to perform a large number of different operations. The most common are

combinations of multiplication, addition, and subtraction involving the color and

alpha values, but other operations are possible, such as minimum and maximum, as

well as bitwise logic operations. DirectX 10 added the capability to blend two colors

from the pixel shader with the framebuffer color. This capability is called dual source-

color blending and cannot be used in conjunction with multiple render targets. MRT

does otherwise support blending, and DirectX 10.1 introduced the capability to

perform different blend operations on each separate buffer.

合并阶段位于固定功能阶段之间，比如三角形设置和完全可编程的着色器阶段。尽管他不是可编程的，但是他的操作是高度可配置的。特别是颜色混合可以设置为执行大量不同的操作。最通用的是包含颜色和alpha值得乘法、加法和减法的组合，但是其他的操作也是可能的，比如最小值和最大值，以及位逻辑操作。DirectX 10增加了混合像素着色器和帧缓冲区两种颜色的能力。这种能力称为双源颜色混合，不能和多重渲染目标一起使用。MRT还支持混合，DirectX 10.1引入了在每个单独缓冲区执行不同混合操作的能力。

As mentioned at the end of the previous section, DirectX 11.3 provided a way to

make blending programmable through ROVs, though at a price in performance.

ROVs and the merging stage both guarantee draw order, a.k.a. output invariance.

Regardless of the order in which pixel shader results are generated, it is an API

requirement that results are sorted and sent to the merging stage in the order in

which they are input, object by object and triangle by triangle.

正如之前章节末尾所说，DirectX 11.3提供了一种通过ROV使混合和编程的方法，尽管在性能是有代价的。ROV和合并阶段都确保了绘制顺序，也就是输出不变性。不管生成像素着色器结果的顺序如何，API要求输入的顺序，对象对对象，三角形对三角形对结果进行排序并将结果发送到合并阶段。

3.10 The Compute Shader 计算着色器

The GPU can be used for more than implementing the traditional graphics pipeline.

There are many non-graphical uses in fields as varied as computing the estimated

value of stock options and training neural nets for deep learning. Using hardware in

this way is called GPU computing. Platforms such as CUDA and OpenCL are used to

control the GPU as a massive parallel processor, with no real need or access to

graphics-specific functionality. These frameworks often use languages such as C or

C++ with extensions, along with libraries made for the GPU.

GPU不仅仅可以用来执行传统的图形渲染管线。在计算股票期权的估计值和训练用于深度学习的神经网络领域有很多非图形化的用途。以这种方式使用硬件被称为GPU计算。比如CUDA和OpenCL平台用来控制GPU作为庞大的并行处理器，不需要或访问卫星特定的功能。这些框架通常使用C或C++拓展以及为GPU创建的库。

Introduced in DirectX 11, the compute shader is a form of GPU computing, in that it

is a shader that is not locked into a location in the graphics pipeline. It is closely tied

to the process of rendering in that it is invoked by the graphics API. It is used

alongside vertex, pixel, and other shaders. It draws upon the same pool of unified

shader processors as those used in the pipeline. It is a shader like the others, in that

it has some set of input data and can access buffers (such as textures) for input and

output. Warps and threads are more visible in a compute shader. For example, each

invocation gets a thread index that it can access. There is also the concept of a

thread group, which consists of 1 to 1024 threads in DirectX 11. These thread groups

are specified by x-, y-, and z-coordinates, mostly for simplicity of use in shader code.

Each thread group has a small amount of memory that is shared among threads. In

DirectX 11, this amounts to 32 kB. Compute shaders are executed by thread group,

so that all threads in the group are guaranteed to run concurrently [1971].

在DirectX 11引入了GPU计算的方式：计算着色器，他是一个不锁定在图形管线位置的着色器。他与渲染处理密切相关，因为他是通过图形API调用。他与顶点、像素和其他着色器一起使用。他使用与流水管线中使用的相同的统一着色器处理器池。他是一个像其他一样的着色器，因为他有相同数据输入集并且可以访问缓冲区（比如纹理）用来输入输出。线程束和线程在计算着色器中更为可见。例如，每次调用可以得到一个线程索引，他可以访问这个索引。这同样是线程组的一个概念，在DirectX 11中包含了1到1024个线程。这些线程由X-,Y-和Z-坐标指定，主要是方便着色器代码使用。每个线程组都有一段小内存可以在线程之间共享。在DirectX 11中，内存数量是32kB。计算着色器被线程组执行，因此组中的所有线程都确保并发运行[1971]。

One important advantage of compute shaders is that they can access data generated

on the GPU. Sending data from the GPU to the CPU incurs a delay, so performance

can be improved if processing and results can be kept resident on the GPU [1403].

Post-processing, where a rendered image is modified in some way, is a common use

of compute shaders. The shared memory means that intermediate results from

sampling image pixels can be shared with neighboring threads. Using a compute

shader to determine the distribution or average luminance of an image, for example,

has been found to run twice as fast as performing this operation on a pixel shader

[530].

计算着色器最重要的优点是他们可以访问GPU生成的数据。从GPU中发送数据到CPU可以产生一个延迟，因此如果处理和结果可以保存在GPU上的话，性能会得到改善[1403]。后处理是指渲染的图形以一些方式更改，是计算着色器的常见用途。共享内存意味着从采样图形像素得到的中间结果可以与相邻线程共享。例如，使用计算着色器去确定一个图像的分布或平均亮度，其运行速度实在像素着色器上执行此操作的两倍[530]。

Compute shaders are also useful for particle systems, mesh processing such as facial

animation [134], culling [1883, 1884], image filtering [1102, 1710], improving depth

precision [991], shadows [865], depth of field [764], and any other tasks where a

set of GPU processors can be brought to bear. Wihlidal [1884] discusses how

compute shaders can be more efficient than tessellation hull shaders. See Figure

3.16 for other uses.

计算着色器同样用于粒子系统、网格处理比如面部动画[134]，裁减[1883,1884]，图像过滤[1102,1710]，改善深度精度[991]，阴影[865]，区域深度[764]，以及其他的任何一组GPU处理器可以处理的任务。Wihlidal[1884]讨论了如何计算着色器比镶嵌外壳着色器更有效。其他用途见图3.16。

This ends our review of the GPU’s implementation of the rendering pipeline. There

are many ways in which the GPUs functions can be used and combined to perform

various rendering-related processes. Relevant theory and algorithms tuned to take

advantage of these capabilities are the central subjects of this book. Our focus now

moves on to transforms and shading.

这就结束了GPU渲染流水管线实现的回顾。有很多方式可以使用GPU函数和组合执行各种渲染相关处理。相关的理论和算法，以及利用这些能力是本书的主题。现在我们我的重点是转换和着色。

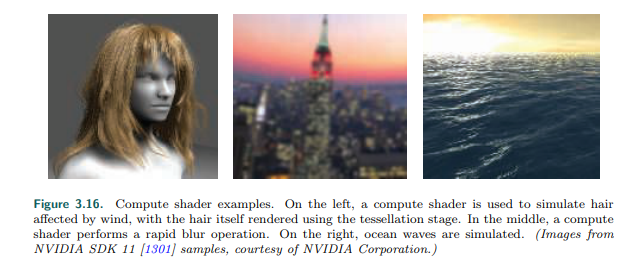


图3.16 计算着色器例子。左侧，计算着色器用在模拟风中头发效果，头发本身使用了曲面细分着色器。在中间，计算着色器执行了快速模糊操作。右边是模拟波浪。（图片来自英伟达SDK 11[1301]样例，英伟达公司提供）。

Further Reading and Resources 更深度的阅读和资源

Giesen’s tour of the graphics pipeline [530] discusses many facets of the GPU at

length, explaining why elements work the way they do. The course by Fatahalian

and Bryant [462] discusses GPU parallelism in a series of detailed lecture slide sets.

While focused on GPU computing using CUDA, the introductory part of Kirk and

Hwa’s book [903] discusses the evolution and design philosophy for the GPU.

Giesen的图形渲染管线之旅[530]讨论了GPU的许多方面，解释了单元是如何工作的。Fatahalian和Bryant[462]的课程在一系列详细的幻灯片中讨论了GPU并行性。Kirk和Hwa的书中介绍了使用CUDA进行GPU计算的同时，也讨论了GPU的发展和设计理念。

To learn the formal aspects of shader programming takes some work. Books such as

the OpenGL Superbible [1606] and OpenGL Programming Guide [885] include

material on shader programming. The older book OpenGL Shading Language [1512]

does not cover more recent shader stages, such as the geometry and tessellation

shaders, but does focus specifically on shader-related algorithms. See this book’s

website, realtimerendering.com, for recent and recommended books.

去学习着色器编程的正式方面需要做一些工作。像OpenGL超级盛典[1606]和OpenGL编程指南[885]包含了着色器编程的材料。较老的OpenGL着色语言[1512]一书没有涵盖最近的着色器阶段，比如集合阶段和曲面细分阶段，但是特别关注与着色器相关的算法。查看本书的网站realtimerendering.com，了解最新推荐书籍。