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

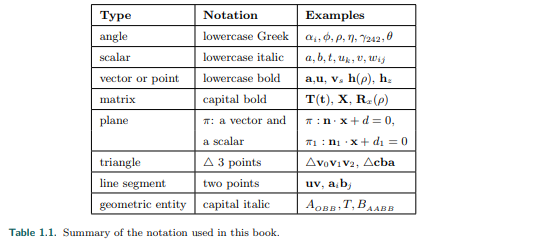
and points are denoted by bold lowercase letters, and the components are accessed

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）

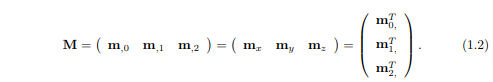
The following notation, shown in Equation 1.2 for a 3 × 3 matrix, is used to isolate

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

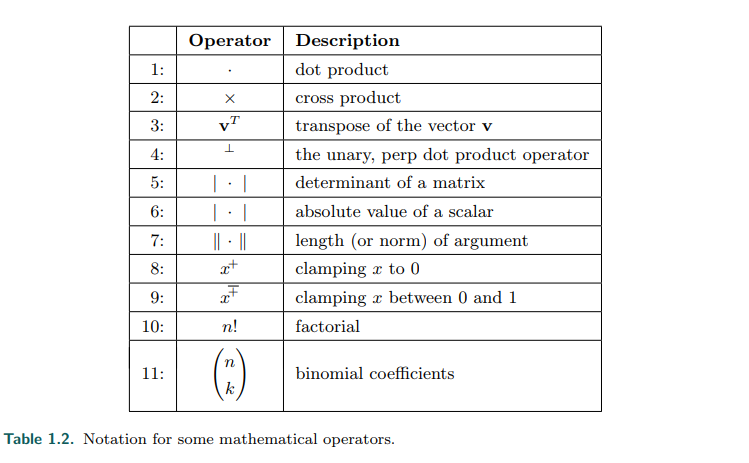
this operator work on a vector v = (vx vy)T gives a vector that is perpendicular to v,

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.

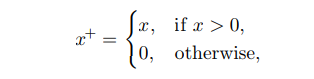
表1.2呈现了一些额外的数学运算符和他们的符号。点乘，叉乘、行列式和长度运算符是在下载资料realtimerendering.com线性代数里面有解释。转置运算符可以翻转一个列向量成为一个行向量，反之亦然。因此列向量可以写成v = (vx vy vz)T.操作符4，在图形宝典4[735]有介绍，是二维向量的一元运算符。这个运算符作用于v = (vx vy)T会得到一个垂直于v的向量，也就是：v⊥ = (−vy vx)T.我们用|a|去表示标量a的绝对值，|A|的意思是矩阵A的行列式。有时候，我们也使用|A| = |a b c| =det(a, b, c),a,b,c是矩阵的列向量。



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

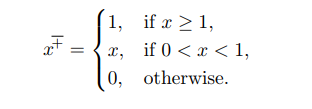
操作符8和9是钳位运算符，通常用在着色计算中。

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

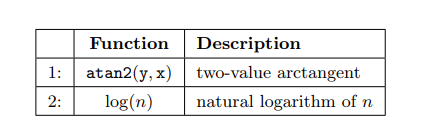
dimensional geometry in the field of computer graphics.

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。

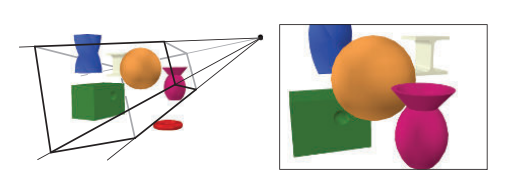


Figure 2.1. In the left image, a virtual camera is located at the tip of the pyramid

(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。

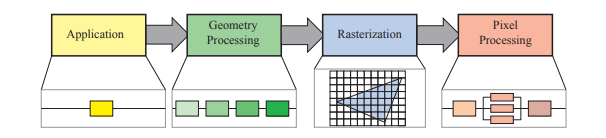


Figure 2.2. The basic construction of the rendering pipeline, consisting of four stages:

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）。

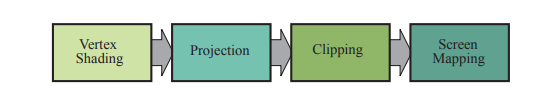


Figure 2.3. The geometry processing stage divided into a pipeline of functional

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。

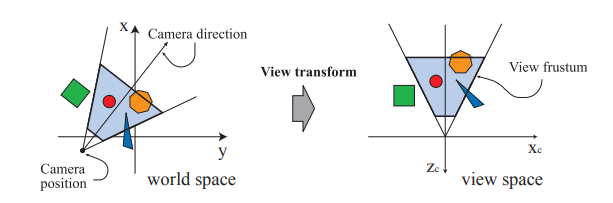


Figure 2.4. In the left illustration, a top-down view shows the camera located and

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。

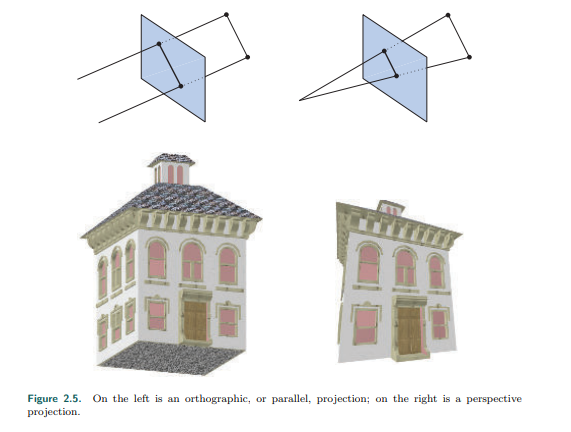


图2.5 左侧是正交投影或平行投影。右侧是透视投影。

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。

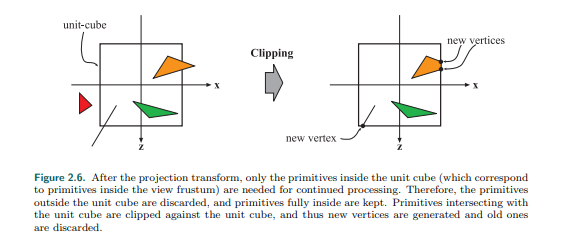


图2.6 在投影转换之后，只有在单位立方体内的图元（对应视同中的图元）才需要继续处理。因此，单元立方体外部的图元会被丢弃，图元完全在内部的被保留。和单位立方体相交的图元会根据单元立方体裁剪，会生成新的顶点，旧的顶点被丢弃。

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。

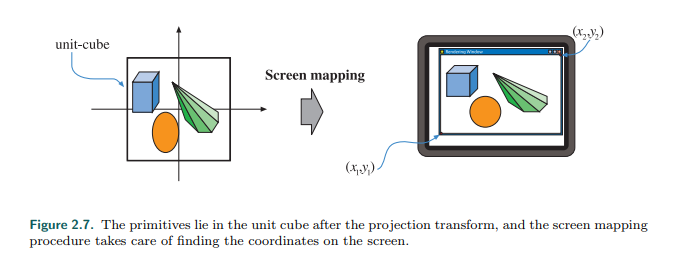


图2.7 单元立方体内的图元在经过了投影转换之后，屏幕映射程序复杂找到他们的屏幕坐标。

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

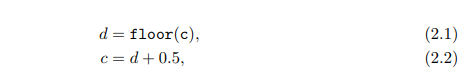
used this scheme, and DirectX 10 and its successors use it. The center of this pixel is

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。

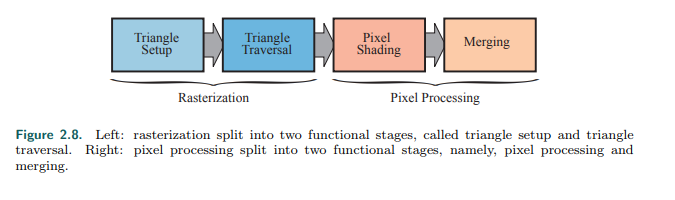


图2.8 左侧：光栅化划分为两个子阶段，称为三角形设置和三角形遍历。

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。

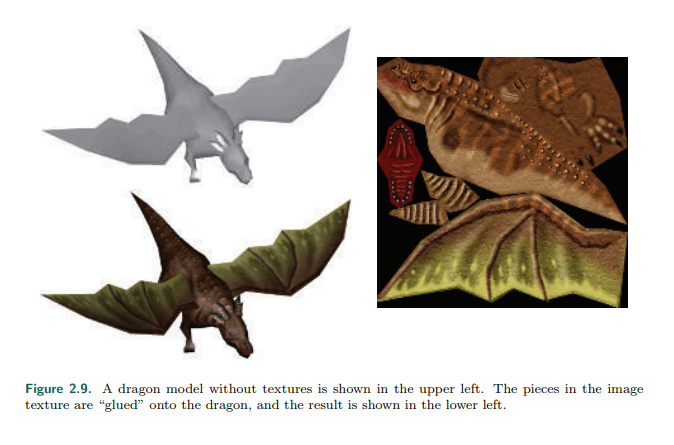


图2.9 一个没有纹理的龙模型在左上方所示。一个图像纹理贴在了龙上，结果如左下角所示。

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代码是一个做一些相应独立任务的小程序，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，日过只有很少的线程，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。

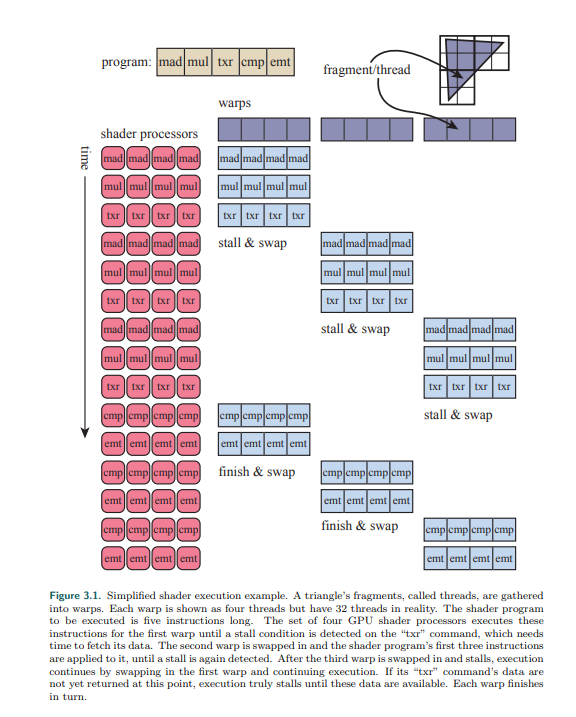


图3.1 简化的着色器执行例子。一个三角形片元，称为线程，集合成翘曲。每个翘曲显示有4个线程，实际上是32个线程。着色器程序由5个指令执行。四个GPU着色处理器执行第一个翘曲的指令直到执行“txr”指令遇到了停顿，需要时间取得数据。第二个翘曲交换进来，执行着色器程序的前3个指令，直到又一次检测到停顿。在第三个翘曲交换进来以及停顿，将交换进第一个翘曲继续执行指令。如果此时他的“txr”命令数据还没有得到，那么执行将真正停顿，直到这些数据可用为止。每个翘曲轮流结束。

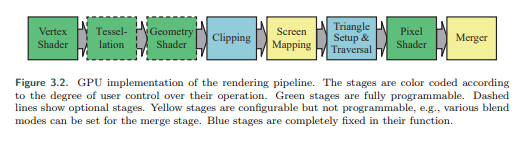


图3.2 GPU实现的渲染管线。依据操作的可控性涂上了不同的颜色。绿色是完全可编程的。点线代表可选阶段。黄色是可配置的但是不可编程，比如各种混合模式可以在合并阶段设置。蓝色是完全固定功能。

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。