1 Introduction

The term computer graphics describes any use of computers to create and manipulate images. This book introduces the algorithmic and mathematical tools that can be used to create all kinds of images—realistic visual effects, informative technical illustrations, or beautiful computer animations. Graphics can be two- or three-dimensional; images can be completely synthetic or can be produced by manipulating photographs. This book is about the fundamental algorithms and mathematics, especially those used to produce synthetic images of three-dimensional objects and scenes.

计算机图形一词是指使用计算机创建和处理图像。本书介绍了可用于创建各种图像—真实视觉效果、内容丰富的技术插图或精美的计算机动画--的算法和数学工具。图形可以是二维的，也可以是三维的；图像可以是完全合成的，也可以通过处理照片来制作。本书主要介绍基本算法和数学，尤其是用于制作三维物体和场景合成图像的算法和数学。

Actually doing computer graphics inevitably requires knowing about specific hardware, file formats, and usually a graphics API (see Section 1.3) or two. Computer graphics is a rapidly evolving field, so the specifics of that knowledge are a moving target. Therefore, in this book we do our best to avoid depending on any specific hardware or API. Readers are encouraged to supplement the text with relevant documentation for their software and hardware environment. Fortunately, the culture of computer graphics has enough standard terminology and concepts that the discussion in this book should map nicely to most environments.

在实际操作计算机图形学时，不可避免地需要了解特定的硬件、文件格式，通常还需要了解一两个图形应用程序接口（见第 1.3 节）。计算机图形学是一个快速发展的领域，因此这些知识的具体内容是一个不断变化的目标。因此，在本书中，我们尽力避免依赖任何特定的硬件或 API。我们鼓励读者为自己的软件和硬件环境补充相关文档。幸运的是，计算机图形文化有足够多的标准术语和概念，本书的讨论应该可以很好地映射到大多数环境中。

This chapter defines some basic terminology and provides some historical background, as well as information sources related to computer graphics.

本章定义了一些基本术语，介绍了一些历史背景以及与计算机图形学有关的信息来源。

1.1 Graphics Areas

Imposing categories on any field is dangerous, but most graphics practitioners would agree on the following major areas of computer graphics:

• Modeling deals with the mathematical specification of shape and appearance properties in a way that can be stored on the computer. For example, a coffee mug might be described as a set of ordered 3D points along with some interpolation rule to connect the points and a reflection model that describes how light interacts with the mug.

• Rendering is a term inherited from art and deals with the creation of shaded images from 3D computer models.

• Animation is a technique to create an illusion of motion through sequences of images. Animation uses modeling and rendering but adds the key issue of movement over time, which is not usually dealt with in basic modeling and rendering.

对任何领域进行分类都是危险的，但大多数图形从业人员都会同意计算机图形学的以下主要领域：

- 建模（Modeling）：建模是用数学的方式来规范形状和外观属性，并将其存储在计算机中。例如，一个咖啡杯可以描述为一组有序的三维点，以及连接这些点的插值规则和描述光线如何与咖啡杯相互作用的反射模型。

- 渲染（Rendering）是从艺术中继承下来的一个术语，用于根据三维计算机模型创建阴影图像。

- 动画是一种通过图像序列创造运动幻觉的技术。动画使用建模和渲染，但增加了随时间运动的关键问题，而基本建模和渲染通常不会处理这个问题。

There are many other areas that involve computer graphics, and whether they are core graphics areas is a matter of opinion. These will all be at least touched on in the text. Such related areas include the following:

涉及计算机图形学的领域还有很多，至于它们是否属于核心图形学领域，则见仁见智。文中至少会涉及到这些领域。这些相关领域包括:

• User interaction deals with the interface between input devices such as mice and tablets, the application, feedback to the user in imagery, and other sensory feedback. Historically, this area is associated with graphics largely because graphics researchers had some of the earliest access to the input/output devices that are now ubiquitous.

用户交互涉及诸如鼠标和平板电脑等输入设备、应用程序、以图像形式向用户提供的反馈以及其他感官反馈之间的接口。从历史上看，这一领域在很大程度上与图形学相关，主要是因为图形学研究人员最早接触到了如今已无处不在的输入 / 输出设备

• Virtual reality attempts to immerse the user into a 3D virtual world. This typically requires at least stereo graphics and response to head motion. For true virtual reality, sound and force feedback should be provided as well. Because this area requires advanced 3D graphics and advanced display technology, it is often closely associated with graphics.

虚拟现实试图让用户沉浸在三维虚拟世界中。这通常至少需要立体图形和对头部运动的响应。对于真正的虚拟现实，还应提供声音和力反馈。由于这一领域需要先进的三维图形和先进的显示技术，因此往往与图形技术密切相关。

1.3 Graphics APIs

A key part of using graphics libraries is dealing with a graphics API. An application program interface (API) is a standard collection of functions to perform a set of related operations, and a graphics API is a set of functions that perform basic operations such as drawing images and 3D surfaces into windows on the screen.

使用图形库的一个关键部分是处理图形应用程序接口。应用程序接口（API）是执行一系列相关操作的标准函数集，而图形 API 是执行基本操作（如在屏幕上的窗口中绘制图像和 3D 曲面）的函数集。

Every graphics program needs to be able to use two related APIs: a graphics API for visual output and a user-interface API to get input from the user. There are currently two dominant paradigms for graphics and user-interface APIs. The first is the integrated approach, exemplified by Java, where the graphics and user interface toolkits are integrated and portable packages that are fully standardized and supported as part of the language. The second is represented by Direct3D and OpenGL, where the drawing commands are part of a software library tied to a language such as C++, and the user-interface software is an independent entity that might vary from system to system. In this latter approach, it is problematic to write portable code, although for simple programs, it may be possible to use a portable library layer to encapsulate the system specific user-interface code.

每个图形程序都需要能够使用两个相关的应用程序接口：一个是用于视觉输出的图形应用程序接口，另一个是用于获取用户输入的用户界面应用程序接口。目前，图形和用户界面 API 有两种主流模式。第一种是以 Java 为代表的集成方法，其中图形和用户界面工具包是集成的、可移植的软件包，作为语言的一部分得到完全标准化和支持。第二种方法以 Direct3D 和 OpenGL 为代表，绘图命令是与 C++ 等语言相关联的软件库的一部分，而用户界面软件则是独立的实体，可能因系统而异。在后一种方法中，编写可移植代码是个问题，尽管对于简单的程序来说，可以使用可移植库层来封装系统特定的用户界面代码。

Whatever your choice of API, the basic graphics calls will be largely the same, and the concepts of this book will apply.

无论您选择何种应用程序接口，基本的图形调用大体相同，本书的概念也适用。

1.4 Graphics Pipeline

Every desktop computer today has a powerful 3D graphics pipeline. This is a special software/hardware subsystem that efficiently draws 3D primitives in perspective. Usually, these systems are optimized for processing 3D triangles with shared vertices. The basic operations in the pipeline map the 3D vertex locations to 2D screen positions and shade the triangles so that they both look realistic and appear in proper back-to-front order.

如今，每台台式电脑都有一个功能强大的 3D 图形管线。这是一个特殊的软件/硬件子系统，可以高效地绘制透视三维基元。通常，这些系统经过优化，可处理具有共享顶点的三维三角形。管线中的基本操作是将三维顶点位置映射到二维屏幕位置，并对三角形进行阴影处理，使它们看起来既逼真又有适当的前后顺序。

Although drawing the triangles in valid back-to-front order was once the most important research issue in computer graphics, it is now almost always solved using the z-buffer, which uses a special memory buffer to solve the problem in a brute-force manner.

虽然按有效的前后顺序绘制三角形曾经是计算机图形学中最重要的研究课题，但现在几乎都是使用 z 缓冲区来解决这个问题，它使用一个特殊的内存缓冲区，以暴力方式解决问题。

It turns out that the geometric manipulation used in the graphics pipeline can be accomplished almost entirely in a 4D coordinate space composed of three traditional geometric coordinates and a fourth homogeneous coordinate that helps with perspective viewing. These 4D coordinates are manipulated using 4 × 4 matrices and 4-vectors. The graphics pipeline, therefore, contains much machinery for efficiently processing and composing such matrices and vectors. This 4D coordinate system is one of the most subtle and beautiful constructs used in computer science, and it is certainly the biggest intellectual hurdle to jump when learning computer graphics. A big chunk of the first part of every graphics book deals with these coordinates.

事实证明，图形处理流程中使用的几何操作几乎完全可以在 4D 坐标空间中完成，该坐标空间由三个传统几何坐标和有助于透视观察的第四个同质坐标组成。这些 4D 坐标使用 4 × 4 矩阵和 4 向量进行处理。因此，图形管线包含了大量用于高效处理和合成此类矩阵和矢量的机制。4D 坐标系是计算机科学中最精妙、最漂亮的结构之一，也是学习计算机图形学时需要跨越的最大智力障碍。每本图形学书籍的第一部分都有很大篇幅涉及这些坐标。

The speed at which images can be generated depends strongly on the number of triangles being drawn. Because interactivity is more important in many applications than visual quality, it is worthwhile to minimize the number of triangles used to represent a model. In addition, if the model is viewed in the distance, fewer triangles are needed than when the model is viewed from a closer distance. This suggests that it is useful to represent a model with a varying level of detail(LOD).

生成图像的速度在很大程度上取决于绘制三角形的数量。由于在许多应用中，交互性比视觉质量更为重要，因此值得尽量减少用于表示模型的三角形数量。此外，如果从远处观察模型，所需的三角形数量要少于从近处观察模型。这表明，用不同的详细程度（LOD）来表示模型是有用的。

1.5 Numerical Issues

Many graphics programs are really just 3D numerical codes. Numerical issues are often crucial in such programs. In the “old days,” it was very difficult to handle such issues in a robust and portable manner because machines had different internal representations for numbers, and even worse, handled exceptions in different and incompatible ways. Fortunately, almost all modern computers conform to the IEEE floating-point standard (IEEE Standards Association, 1985). This allows the programmer to make many convenient assumptions about how certain numeric conditions will be handled.

许多图形程序实际上只是三维数字代码。在这些程序中，数字问题往往至关重要。在 “过去”，由于计算机内部对数字的表示方法各不相同，更有甚者，处理异常的方法也各不相同，互不兼容，因此很难以稳健、可移植的方式处理此类问题。幸运的是，几乎所有现代计算机都符合 IEEE 浮点标准（IEEE 标准协会，1985 年）。这使得程序员可以就如何处理某些数字条件做出许多方便的假设。

Although IEEE floating-point has many features that are valuable when coding numeric algorithms, there are only a few that are crucial to know for most situations encountered in graphics. First, and most important, is to understand that there are three “special” values for real numbers in IEEE floating-point:

虽然 IEEE 浮点运算有很多特性，对数值算法编码很有价值，但对于图形学中遇到的大多数情况来说，只有少数几个特性是必须了解的。首先，也是最重要的一点是，要了解 IEEE 浮点中的实数有三个 “特殊 ”值：

1. Infinity (∞). This is a valid number that is larger than all other valid numbers.

2. Minus infinity (−∞). This is a valid number that is smaller than all other valid numbers.

3. Not a number (NaN). This is an invalid number that arises from an operation with undefined consequences, such as zero divided by zero.

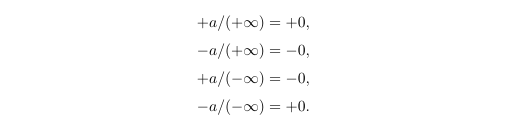
1. 无穷大（∞）。这是一个比所有其他有效数字都大的有效数字。

2. 无穷小 (-∞)。这是一个比所有其他有效数字都小的有效数字。

3. 不是一个数（NaN）。这是一个无效数，由一个具有未定义后果的运算产生，例如零除以零。

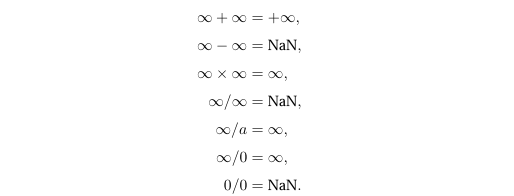
The designers of IEEE floating-point made some decisions that are extremely convenient for programmers. Many of these relate to the three special values above in handling exceptions such as division by zero. In these cases, an exception is logged, but in many cases, the programmer can ignore that. Specifically, for any positive real number a, the following rules involving division by infinite values hold

IEEE 浮点运算的设计者做出了一些对程序员来说非常方便的决定。其中许多都与上述处理异常（如除以零）的三个特殊值有关。在这些情况下，异常会被记录下来，但在很多情况下，程序员可以忽略不计。具体来说，对于任何正实数 a，以下涉及无限值除法的规则都是成立的



Other operations involving infinite values behave the way one would expect. Again for positive a, the behavior is as follows:

其他涉及无穷值的运算与我们预期的情况相同。同样，对于正 a，行为如下：



The rules in a Boolean expression involving infinite values are as expected:

1. All finite valid numbers are less than +∞.

2. All finite valid numbers are greater than −∞.

3. −∞ is less than +∞.

The rules involving expressions that have NaN values are simple:

1. Any arithmetic expression that includes NaN results in NaN.

2. Any Boolean expression involving NaN is false

布尔表达式中涉及无限值的规则与预期的一样：

1. 所有有限有效数字都小于 +∞。

2. 所有有限有效数字都大于 -∞。

3. -∞小于 +∞。

涉及具有 NaN 值的表达式的规则很简单：

1. 任何包含 NaN 的算术表达式的结果都是 NaN。

2. 任何包含 NaN 的布尔表达式都是false。

Perhaps the most useful aspect of IEEE floating-point is how divide-by-zero is handled. For any positive real number a, the following rules involving division by zero values hold:

IEEE 浮点运算最有用的方面可能是如何处理除零值。对于任何正实数 a，以下涉及零值除法的规则都是成立的



There are many numeric computations that become much simpler if the programmer takes advantage of the IEEE rules. For example, consider the expression:

如果程序员利用 IEEE 规则，许多数字计算都会变得简单得多。例如，请看表达式



Such expressions arise with resistors and lenses. If divide-by-zero resulted in a program crash (as was true in many systems before IEEE floating-point), then two if statements would be required to check for small or zero values of b or c. Instead, with IEEE floating-point, if b or c is zero, we will get a zero value for a as desired. Another common technique to avoid special checks is to take advantage of the Boolean properties of NaN. Consider the following code segment:

电阻和透镜就会出现这种表达式，（此处翻译可能有问题）。如果除以零会导致程序崩溃（在 IEEE 浮点运算之前的许多系统中都是如此），那么就需要两个 if 语句来检查 b 或 c 的值是否很小或为零。另一种避免特殊检查的常用方法是利用 NaN 的布尔属性。请看下面的代码段：



Here, the function f may return “ugly” values such as ∞ or NaN, but the if condition is still well-defined: it is false for a = NaN or a = −∞ and true for a =+∞. With care in deciding which values are returned, often the if can make the right choice, with no special checks needed. This makes programs smaller, more robust, and more efficient.

在这里，函数 f 可能返回 “丑陋 ”的值，如 ∞ 或 NaN，但 if 条件仍然定义明确：当 a = NaN 或 a = -∞ 时为假，当 a =+∞ 时为真。只要小心决定返回哪些值，if 通常就能做出正确的选择，而不需要特别的检查。这使得程序更小、更健壮、更高效。

1.6 Efficiency

There are no magic rules for making code more efficient. Efficiency is achieved through careful tradeoffs, and these tradeoffs are different for different architectures. However, for the foreseeable future, a good heuristic is that programmers should pay more attention to memory access patterns than to operation counts. This is the opposite of the best heuristic of two decades ago. **This switch has occurred because the speed of memory has not kept pace with the speed of processors**. Since that trend continues, the importance of limited and coherent memory access for optimization should only increase.

要提高代码的效率，并没有什么神奇的规则。效率是通过谨慎权衡实现的，而这些权衡对于不同的体系结构是不同的。不过，在可预见的未来，一个好的启发式方法是，程序员应更多地关注内存访问模式，而不是操作次数。这与二十年前的最佳启发式正好相反。之所以出现这种变化，是因为**内存的速度跟不上处理器的速度**。由于这一趋势仍在继续，**有限且连贯的内存访问对于优化的重要性只会有增无减**。

A reasonable approach to making code fast is to proceed in the following order, taking only those steps which are needed:

快速编写代码的合理方法是按照以下顺序进行，只采取必要的步骤：

1. Write the code in the most straightforward way possible. Compute intermediate results as needed on the fly rather than storing them. 用最直接的方式编写代码。根据需要即时计算中间结果，而不是存储结果。

2. Compile in optimized mode. 以优化模式编译。

3. Use whatever profiling tools exist to find critical bottlenecks. 使用任何现有的剖析工具来查找关键瓶颈。

4. Examine data structures to look for ways to improve locality. If possible, make data unit sizes match the cache/page size on the target architecture. 检查数据结构，寻找提高局部性的方法。如果可能，使数据单元大小与目标架构的缓存/页面大小相匹配。

5. If profiling reveals bottlenecks in numeric computations, examine the assembly code generated by the compiler for missed efficiencies. Rewrite source code to solve any problems you find. 如果剖析发现数值计算中存在瓶颈，则检查编译器生成的汇编代码，查找遗漏的效率。重写源代码以解决发现的任何问题。

The most important of these steps is the first one. Most “optimizations” make the code harder to read without speeding things up. In addition, time spent upfront optimizing code is usually better spent correcting bugs or adding features. Also, beware of suggestions from old texts; some classic tricks such as using integers instead of reals may no longer yield speed because modern CPUs can usually perform floating-point operations just as fast as they perform integer operations. In all situations, profiling is needed to be sure of the merit of any optimization for a specific machine and compiler.

其中最重要的是第一个步骤。大多数 “优化 ”都会增加代码的阅读难度，却不会加快速度。此外，前期优化代码所花费的时间通常更适合用来修正错误或添加功能。此外，还要谨防旧版本中的建议；一些经典的技巧，如使用整数代替实数，可能不再能提高速度，因为现代 CPU 执行浮点运算的速度通常与执行整数运算的速度相当。在任何情况下，都需要进行剖析，以确定针对特定机器和编译器的任何优化的优点。

1.7 Designing and Coding Graphic Programs

Certain common strategies are often useful in graphics programming. In this section, we provide some advice that you may find helpful as you implement the methods you learn about in this book.

在图形编程中，某些常用策略通常非常有用。在本节中，我们将为您提供一些建议，您在实施本书所学方法时可能会发现这些建议很有帮助。

1.7.1 Class Design

A key part of any graphics program is to have good classes or routines for geometric entities such as vectors and matrices, as well as graphics entities such as RGB colors and images. These routines should be made as clean and efficient as possible. A universal design question is whether locations and displacements should be separate classes because they have different operations; e.g., a location multiplied by one-half makes no geometric sense while one-half of a displacement does (Goldman, 1985; DeRose, 1989). There is little agreement on this question, which can spur hours of heated debate among graphics practitioners, but for the sake of example, let’s assume we will not make the distinction.

任何图形程序的关键部分都是为矢量和矩阵等几何实体以及 RGB 颜色和图像等图形实体建立良好的类或例程。这些例程应尽可能简洁高效。一个普遍性的设计问题是，位置和位移是否应该分门别类，因为它们的运算方式不同；例如，位置乘以二分之一没有几何意义，而位移的二分之一却有几何意义（Goldman, 1985; DeRose, 1989）。在这个问题上几乎没有一致的意见，这可能会引发图形从业人员数小时的激烈争论，但为了举例说明，让我们假设不做区分。

This implies that some basic classes to be written include. 这意味着需要编写的一些基本类包括：

* vector2. A 2D vector class that stores an x- and y- component. It should store these components in a length-2 array so that an indexing operator can be well supported. You should also include operations for vector addition, vector subtraction, dot product, cross product, scalar multiplication, and scalar division. vector2. 二维向量类，用于存储 x 和 y 分量。它应将这些分量存储在一个长度为 2 的数组中，这样就能很好地支持索引操作。此外，还应包括矢量加法、矢量减法、点积、交叉积、标量乘法和标量除法等操作。
* vector3. A 3D vector class analogous to vector2. 与 vector2 类似的三维向量类。
* hvector. A homogeneous vector with four components (see Chapter 8).有四个分量的均质向量（见第 8 章）。
* RGB. An RGB color that stores three components. You should also include operations for RGB addition, RGB subtraction, RGB multiplication, scalar multiplication, and scalar division. 存储三个分量的 RGB 颜色。您还应包含 RGB 加法、RGB 减法、RGB 乘法、标量乘法和标量除法的操作。
* transform. A 4 × 4 matrix for transformations. You should include a matrix multiply and member functions to apply to locations, directions, and surface normal vectors. As shown in Chapter 7, these are all different. 用于变换的 4 × 4 矩阵。应包含矩阵乘法和以应用于位置、方向和表面法向量的成员函数。如第 7 章所示，这些函数都是不同的。
* image. A 2Darray of RGB pixels with an output operation. 具有输出操作的 RGB 像素二维数组。

In addition, you might or might not want to add classes for intervals, orthonormal bases, and coordinate frames. 此外，您还可以添加或不添加区间、正交基和坐标框架的类。

1.7.2 Float vs Double

Modern architecture suggests that keeping memory use down and maintaining coherent memory access are the keys to efficiency. This suggests using single precision data. However, avoiding numerical problems suggests using double precision arithmetic. The tradeoffs depend on the program, but it is nice to have a default in your class definitions.

PS：I suggest using doubles for geometric computation and floats for color computation. For data that occupies a lot of memory, such as triangle meshes, I suggest storing float data, but converting to double when data are accessed through member functions.

现代架构认为，降低内存使用量和保持内存访问的一致性是提高效率的关键。这建议使用单精度数据。然而，为了避免出现数字问题，建议使用双精度算术。如何取舍取决于程序，但在类定义中设置一个默认值还是不错的。

PS：我建议几何计算使用double，颜色计算使用float。对于占用大量内存的数据，如三角形网格，我建议存储float数据，但在通过成员函数访问数据时转换为 double 数据。

1.7.3 Debugging Graphic Programs

If you ask around, you may find that as programmers become more experienced, they use traditional debuggers less and less. One reason for this is that using such debuggers is more awkward for complex programs than for simple programs. Another reason is that the most difficult errors are conceptual ones where the wrong thing is being implemented, and it is easy to waste large amounts of time stepping through variable values without detecting such cases. We have found several debugging strategies to be particularly useful in graphics.

如果你四处打听，可能会发现随着程序员的经验越来越丰富，他们越来越少使用传统的调试器。其中一个原因是，在复杂程序中使用传统调试器比在简单程序中使用传统调试器更麻烦。另一个原因是，最难处理的错误是概念性错误，即执行了错误的东西，而这种情况下很容易浪费大量时间来查看变量值，却无法发现。我们发现有几种调试策略在图形学中特别有用。

The Scientific Method. 科学的方法：

In graphics programs, there is an alternative to traditional debugging that is often very useful. The downside to it is that it is very similar to what computer programmers are taught not to do early in their careers, so you may feel “naughty” if you do it: we create an image and observe what is wrong with it. Then, we develop a hypothesis about what is causing the problem and test it. For example, in a ray-tracing program we might have many somewhat random looking dark pixels. This is the classic “shadow acne” problem that most people run into when they write a ray tracer. Traditional debugging is not helpful here; instead, we must realize that the shadow rays are hitting the surface being shaded. We might notice that the color of the dark spots is the ambient color, so the direct lighting is what is missing. Direct lighting can be turned off in shadow, so you might hypothesize that these points are incorrectly being tagged as in shadow when they are not. To test this hypothesis, we could turn off the shadowing check and recompile. This would indicate that these are false shadow tests, and we could continue our detective work. The key reason that this method can sometimes be good practice is that we never had to spot a false value or really determine our conceptual error. Instead, we just narrowed in on our conceptual error experimentally. Typically, only a few trials are needed to track things down, and this type of debugging is enjoyable.

在图形程序中，有一种替代传统调试的方法，通常非常有用。它的缺点是，它与计算机程序员在职业生涯早期被教导不要做的事情非常相似，所以如果你做了，你可能会觉得 “淘气”：我们创建一个图像，然后观察它有什么问题。然后，我们就问题的起因提出假设并进行测试。例如，在光线跟踪程序中，我们可能会看到许多看起来有些随意的暗像素。这是典型的 “阴影失真 ”问题，大多数人在编写光线追踪程序时都会遇到这个问题。传统的调试方法在这里毫无用处；相反，我们必须意识到阴影光线是打在被着色的表面上。我们可能会注意到，暗点的颜色是环境色，因此缺少的是直接光照。直接光照能够在阴影中被关闭，因此可以假设这些点被错误地标记为阴影中，而实际上它们并不在阴影中。为了验证这一假设，我们可以关闭阴影检查并重新编译。这将表明这些是错误的阴影测试，我们就可以继续我们的侦查工作了。这种方法有时是一种很好的做法，其关键原因在于我们从未发现过错误值或真正确定我们的概念错误。相反，我们只是通过实验缩小了概念错误的范围。通常情况下，只需要几次试验就能找出问题所在，而且这种类型的调试是令人愉快的。

Images as Coded Debugging Output. 图像作为编码调试输出。

In many cases, the easiest channel by which to get debugging information out of a graphics program is the output image itself. If you want to know the value of some variable for part of a computation that runs for every pixel, you can just modify your program temporarily to copy that value directly to the output image and skip the rest of the calculations that would normally be done. For instance, if you suspect a problem with surface normals is causing a problem with shading, you can copy the normal vectors directly to the image (x goes to red, y goes to green, z goes to blue), resulting in a color-coded illustration of the vectors actually being used in your computation. Or, if you suspect a particular value is sometimes out of its valid range, make your program write bright red pixels where that happens. Other common tricks include drawing the back sides of surfaces with an obvious color (when they are not supposed to be visible), coloring the image by the ID numbers of the objects, or coloring pixels by the amount of work they took to compute.

在许多情况下，从图形程序中获取调试信息的最简单渠道就是输出图像本身。如果您想知道某个变量在每个像素上的计算值，您只需临时修改程序，将该值直接复制到输出图像上，而跳过通常要进行的其他计算。例如，如果您怀疑是表面法线的问题导致了阴影的问题，您可以直接将法线向量复制到图像中（x 为红色，y 为绿色，z 为蓝色），这样就可以用颜色编码说明计算中实际使用的向量。或者，如果你怀疑某个特定值有时超出了有效范围，可以让程序在出现这种情况的地方写入鲜红的像素。其他常见的技巧还包括用明显的颜色绘制表面的背面（当它们不应该可见时）、用对象的 ID 编号为图像着色，或用计算所花费的工作量为像素着色。

Using a Debugger. 使用调试器。

There are still cases, particularly when the scientific method seems to have led to a contradiction, when there’s no substitute for observing exactly what is going on. The trouble is that graphics programs often involve many, many executions of the same code (once per pixel, for instance, or once per triangle), making it completely impractical to step through in the debugger from the start. And the most difficult bugs usually only occur for complicated inputs.

在某些情况下，尤其是当科学方法似乎导致了矛盾时，观察究竟发生了什么是无法替代的。问题在于，图形程序通常会多次执行相同的代码（例如，每个像素执行一次，或每个三角形执行一次），因此从一开始就在调试器中逐步检查是完全不切实际的。而且最难处理的错误通常只出现在复杂输入的情况下。

A useful approach is to “set a trap” for the bug. First, make sure your program is deterministic—run it in a single thread and **make sure that all random numbers are computed from fixed seeds**. Then, find out which pixel or triangle is exhibiting the bug and add a statement before the code you suspect is incorrect that will be executed only for the suspect case. For instance, if you find that pixel (126,247) exhibits the bug, then add

一种有用的方法是为错误 “设陷阱”。首先，确保您的程序是确定性的，在单线程中运行，**并确保所有随机数都是由固定种子计算的**。然后，找出出现错误的像素或三角形，并在您怀疑不正确的代码前添加一条语句，该语句仅在可疑情况下执行。例如，如果您发现像素 (126,247) 出现了错误，那么添加



If you set a breakpoint on the print statement, you can drop into the debugger just before the pixel you’re interested in is computed. Some debuggers have a “conditional breakpoint” feature that can achieve the same thing without modifying the code.

如果在打印语句上设置一个断点，就可以在计算出你感兴趣的像素之前进入调试器。有些调试器具有 “条件断点 ”功能，可以在不修改代码的情况下实现同样的目的。

In the cases where the program crashes, a traditional debugger is useful for pinpointing the site of the crash. You should then start backtracking in the program, using asserts and recompiles, to find where the program went wrong. These asserts should be left in the program for potential future bugs you will add. This again means the traditional step-through process is avoided, because that would not be adding the valuable asserts to your program.

在程序崩溃的情况下，传统的调试器对于精确定位崩溃位置非常有用。然后，你应该开始回溯程序，使用断言和重新编译，找出程序出错的地方。这些断言应保留在程序中，以备将来可能添加的错误。这也意味着避免了传统的逐步处理过程，因为这样就不会在程序中添加有价值的断言。

Data Visualization for Debugging. 调试数据可视化

Often, it is hard to understand what your program is doing, because it computes a lot of intermediate results before it finally goes wrong. The situation is similar to a scientific experiment that measures a lot of data, and one solution is the same: make good plots and illustrations for yourself to understand what the data mean. For instance, in a ray tracer you might write code to visualize ray trees so you can see what paths contributed to a pixel, or in an image resampling routine you might make plots that show all the points where samples are being taken from the input. Time spent writing code to visualize your program’s internal state is also repaid in a better understanding of its behavior when it comes time to optimize it.

通常情况下，你很难理解你的程序在做什么，因为它在最终出错之前计算了很多中间结果。这种情况类似于测量大量数据的科学实验，解决方法也是一样的：为自己制作良好的图表和插图，以便理解数据的含义。例如，在光线跟踪器中，你可以编写代码将光线树可视化，这样你就能看到哪些路径对像素有贡献；在图像重采样程序中，你可以绘制图表显示从输入中采样的所有点。花时间编写可视化程序内部状态的代码，还可以在优化程序时更好地理解程序的行为。

2. Miscellaneous Math

Much of graphics is just translating math directly into code. The cleaner the math, the cleaner the resulting code. So much of this book concentrates on using just the right math for the job. This chapter reviews various tools from high school and college mathematics and is designed to be used more as a reference than as a tutorial. It may appear to be a hodge-podge of topics and indeed it is; each topic is chosen because it is a bit unusual in “standard” math curricula, because it is of central importance in graphics, or because it is not typically treated from a geometric standpoint. In addition to establishing a review of the notation used in this book, this chapter also emphasizes a few points that are sometimes skipped in the standard undergraduate curricula, such as barycentric coordinates on triangles. This chapter is not intended to be a rigorous treatment of the material; instead, intuition and geometric interpretation are emphasized. A discussion of linear algebra is deferred until Chapter 6 just before transformation matrices are discussed. Readers are encouraged to skim this chapter to familiarize themselves with the topics covered and to refer back to it as needed. The exercises at the end of this chapter may be useful in determining which topics need a refresher.

图形学的大部分工作就是将数学直接转化为代码。数学越简洁，生成的代码就越简洁。因此，本书的大部分内容都集中在使用正确的数学方法来完成工作。本章回顾了高中和大学数学中的各种工具，旨在作为参考而非教程使用。每一个主题的选择都是因为它在 “标准 ”数学课程中有点不同寻常，或者因为它在图形学中具有核心意义，或者因为它通常不是从几何的角度来处理的。除了回顾本书使用的符号外，本章还强调了标准本科课程中有时会跳过的几个要点，如三角形的重心坐标。本章无意对教材进行严格处理，而是强调直觉和几何解释。关于线性代数的讨论将推迟到第 6 章讨论变换矩阵之前。我们鼓励读者略读本章，以熟悉所涉及的主题，并在需要时回头查阅。本章最后的练习可能有助于确定哪些主题需要复习。

2.1 Sets and Mappings

Mappings, also called functions, are basic to mathematics and programming. Like a function in a program, a mapping in math takes an argument of one type and maps it to (returns) an object of a particular type. In a program, we say “type”; in math, we would identify the set. When we have an object that is a member of a set, we use the ∈ symbol. For example,

映射也称为函数，是数学和编程的基础。与程序中的函数一样，数学中的映射也是将一种类型的参数映射到（返回）一种特定类型的对象。在程序中，我们说的是 “类型”；而在数学中，我们指的是集合。当我们有一个对象是集合的成员时，我们使用 ∈ 符号。例如

a∈S

can be read “a is a member of set S.” Given any two sets A and B, we can create a third set by taking the Cartesian product of the two sets, denoted A × B. This set A × B is composed of all possible ordered pairs (a ,b) where a ∈ A and b ∈B. As a shorthand, we use the notation A2 to denote A× A. We can extend the Cartesian product to create a set of all possible ordered triples from three sets and so on for arbitrarily long ordered tuples from arbitrarily many sets.

可以理解为 “a 是集合 S 的成员”。给定任意两个集合 A 和 B，我们可以通过取这两个集合的笛卡尔积（表示为 A × B）来创建第三个集合。这个集合 A × B 由所有可能的有序对 (a ,b) 组成，其中 a∈A 和 b∈B. 作为简称，我们用 A2 表示 A× A。我们可以扩展笛卡尔积，从三个集合中创建一个包含所有可能有序三元组的集合，并以此类推，从任意多的集合中创建任意长的有序元组。

Common sets of interest include

* R - the real numbers；实数集
* R+ the nonnegative real numbers (includes zero)；非负实数（包括零）；
* R2 - the ordered pairs in the real 2D plane; 在真实的二维平面上的有序对；
* Rn - the points in n-dimensional Cartesian space; 在 n 维笛卡尔空间中的点；
* Z - the integers;
* S2 - the set of 3D points (points in R3) on the unit sphere. 单位球面上三维点（R3 中的点）的集合。

Note that although S2 is composed of points embedded in three-dimensional space, it is on a surface that can be parameterized with two variables, so it can be thought of as a 2D set. Notation for mappings uses the arrow and a colon, for example, f : R→ Z , which you can read as “There is a function called f that takes a real number as input and maps it to an integer.” Here, the set that comes before the arrow is called the **domain** of the function, and the set on the right-hand side is called the **target**. Computer programmers might be more comfortable with the following equivalent language: “There is a function called f which has one real argument and returns an integer.” In other words, the set notation above is equivalent to the common programming notation: **integer f(real) ← equivalent → f : R → Z**. So the colon-arrow notation can be thought of as a programming syntax. It’s that simple.

请注意，虽然 S2 是由嵌入三维空间的点组成的，但它位于一个可以用两个变量进行参数化的面上，因此可以将其视为一个二维集合。映射的符号使用箭头和冒号，例如，f ：R→Z，可以理解为 “有一个名为 f 的函数，它将实数作为输入，并将其映射为整数”。这里，箭头前的集合称为函数的域，右侧的集合称为目标。计算机程序员可能更喜欢使用下面的等价语言：“有一个函数叫 f，它有一个实数参数，返回一个整数"。换句话说，上述集合符号等同于常见的编程符号：整数 f(实数) ← 等价 → f ：R → Z。因此，冒号箭头符号可以看作是一种编程语法。就是这么简单。

The point f(a) is called the image of a, and the image of a set A (a subset of the domain) is the subset of the target that contains the images of all points in A. The image of the whole domain is called the range of the function.

点 f(a) 称为 a 的image，而集合 A（域的子集）的image是包含 A 中所有点的image的目标子集。整个域的image称为函数的范围。

2.1.1 Inverse Mappings