



Unity Shader

入门精要

冯乐乐 著



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS



[0000](#)

[0000](#)

[00](#)

[0000000000000000](#)

[00000000](#)

[0000000000](#)

[000000](#)

[0000](#)

[00](#)

[010000](#)

[0100000Shader000](#)

[1.1000000000](#)

[1.20000](#)

[02000000](#)

[2.100](#)

[2.2CPUGPU00000](#)

[2.3GPU000](#)

[2.4000000000](#)

[2.5000000000Shader00](#)

[2.60000](#)

[030UnityShader00](#)

[3.1UnityShader00](#)

[3.2UnityShader0000ShaderLab](#)

[3.3UnityShader000](#)

[3.4UnityShader000](#)

[3.500000UnityShader00](#)

[3.60000](#)

[3.70000](#)

[04000Shader00000000](#)

[4.100000000](#)

[4.20000000](#)

[4.30000](#)

[4.400](#)

[4.5 纹理采样函数](#)

[4.6 光照](#)

[4.7 阴影](#)

[4.8 Unity Shader 进阶](#)

[4.9 粒子](#)

[4.10 动画](#)

[4.11 其他](#)

[第2章 基础](#)

[第5章 Unity Shader 入门](#)

[5.1 纹理采样函数](#)

[5.2 光照/阴影](#)

[5.3 粒子Unity 进阶](#)

[5.4 Unity 进阶 Cg/HLSL](#)

[5.5 粒子Debug](#)

[5.6 粒子](#)

[5.7 Shader 进阶](#)

[5.8 其他](#)

[第6章 Unity 进阶](#)

[6.1 纹理采样函数](#)

[6.2 光照](#)

[6.3 Unity 进阶](#)

[6.4 Unity Shader 进阶](#)

[6.5 Unity Shader 进阶](#)

[6.6 粒子Unity 进阶](#)

[第7章 粒子](#)

[7.1 粒子](#)

[7.2 粒子](#)

[7.3 粒子](#)

[7.4 粒子](#)

[第8章 动画](#)

[8.1 粒子](#)

[8.2 Unity Shader 进阶](#)

[8.3 粒子](#)

[8.4 粒子](#)

[8.5 粒子](#)

[8.6 ShaderLab 进阶](#)

8.7 物理渲染

3.1 物理

9.1 物理引擎

9.1 Unity物理引擎

9.2 Unity物理引擎

9.3 Unity物理引擎

9.4 Unity物理

9.5 物理引擎Unity Shader

10.1 物理引擎

10.1 物理引擎

10.2 物理引擎

10.3 物理引擎

11.1 物理引擎

11.1 Unity Shader物理引擎

11.2 物理引擎

11.3 物理引擎

4.1 物理

12.1 物理引擎

12.1 物理引擎物理引擎物理引擎

12.2 物理引擎物理引擎物理引擎

12.3 物理引擎

12.4 物理引擎

12.5 Bloom物理

12.6 物理引擎

12.7 物理引擎

13.1 物理引擎物理引擎

13.1 物理引擎物理引擎

13.2 物理引擎

13.3 物理引擎

13.4 物理引擎

13.5 物理引擎

14.1 物理引擎

14.1 物理引擎物理引擎

14.2 物理引擎物理引擎

14.3 物理引擎

14.4 物理引擎

[15 15.1 15.2 15.3 15.4 15.5](#)

[15.1](#)

[15.2](#)

[15.3](#)

[15.4](#)

[15.5](#)

[16 16.1 16.2 16.3 16.4 16.5 16.6 16.7 16.8 16.9](#)

[16.1](#)

[16.2](#)

[16.3](#)

[16.4](#)

[16.5](#)

[16.6](#)

[16.7](#)

[16.8](#)

[16.9](#)

[5 5.1 5.2 5.3 5.4 5.5 5.6 5.7 5.8 5.9](#)

[17 17.1 17.2 17.3 17.4 17.5 17.6](#)

[17.1](#)

[17.2](#)

[17.3](#)

[17.4](#)

[17.5](#)

[17.6](#)

[18 18.1 18.2 18.3 18.4 18.5 18.6](#)

[18.1](#)

[18.2](#)

[18.3](#)

[18.4](#)

[18.5](#)

[18.6](#)

[19 19.1 19.2 19.3 19.4](#)

[19.1](#)

[19.2](#)

[19.3](#)

[19.4](#)

20□□□□□□□

20.1 □□□□□□□□□□

20.2 □□□□

20.3 □□□

□□□□□□□

□□□□□

□□□□□□

□□□□□

□□□□□□□□

□□□

□□

- □□□□
□□□□ □ □
- □□□□□□□□□□ □□□□□□□□□□11□
□□ 100164 □□□□ 315@ptpress.com.cn
□□ <http://www.ptpress.com.cn>
- □□□□□□(010)81055410
□□□□□□(010)81055315

```

000000000000Unity Shader000000000000Unity000000
00000000Unity Shader0000000000000000000000000000
000000000000000000000000

```

Unity

Unity Shader

- 在Unity Shader中，C++和C#的Shader都可以通过编译生成统一的Shader。Unity Shader的编译过程是在运行时完成的，而不是在编译时完成的。
- Unity Shader的编译过程是在运行时完成的，而不是在编译时完成的。
- Shader的编译过程是在运行时完成的，而不是在编译时完成的。
- C++和C#的Shader都可以通过编译生成统一的Shader。Unity Shader的编译过程是在运行时完成的，而不是在编译时完成的。

[illegible]

目錄

前言

- 本書介紹了Unity Shader的基礎知識，包括Unity Shader的編寫方法、Unity Shader的執行原理、Unity Shader的優化方法等。本書共分3章，第一章介紹了Unity Shader的基礎知識，第二章介紹了Unity Shader的執行原理，第三章介紹了Unity Shader的優化方法。
- 本書介紹了Unity Shader的編寫方法，包括Unity Shader的編寫環境、Unity Shader的編寫步驟、Unity Shader的編寫示例等。本書共分3章，第一章介紹了Unity Shader的編寫環境，第二章介紹了Unity Shader的編寫步驟，第三章介紹了Unity Shader的編寫示例。
- 本書介紹了Unity Shader的執行原理，包括Unity Shader的執行流程、Unity Shader的執行原理、Unity Shader的執行優化等。本書共分3章，第一章介紹了Unity Shader的執行流程，第二章介紹了Unity Shader的執行原理，第三章介紹了Unity Shader的執行優化。
- 本書介紹了Unity Shader的優化方法，包括Unity Shader的優化原理、Unity Shader的優化方法、Unity Shader的優化示例等。本書共分3章，第一章介紹了Unity Shader的優化原理，第二章介紹了Unity Shader的優化方法，第三章介紹了Unity Shader的優化示例。

本書是根據作者在Unity Shader方面的研究心得編寫的，旨在為Unity Shader的學習者提供參考。

目录

本书的所有资源在github上https://github.com/candycat1992/Unity_Shaders_Book 本书的所有资源都在github上
 本书使用Unity 5.2.1版本，Mac 10.9.5版本，Windows 8版本，所有资源都在github上，所有资源都在github上，所有资源都在github上

目录	内容
Assets/Scenes	本书的所有资源都在github上，所有资源都在github上，所有资源都在github上 Assets/Scenes/Chapter7 本书的所有资源都在github上，所有资源都在github上，所有资源都在github上 Scene_7_2_3 本书的所有资源都在github上，所有资源都在github上，所有资源都在github上 Scene_7_1_2_a 本书的所有资源都在github上，所有资源都在github上，所有资源都在github上 Scene_7_1_2_b
Assets/Shaders	本书的所有资源都在github上，所有资源都在github上，所有资源都在github上 Unity Shader 本书的所有资源都在github上，所有资源都在github上，所有资源都在github上 Assets/Shaders/Chapter7 本书的所有资源都在github上，所有资源都在github上，所有资源都在github上 Unity Shader 本书的所有资源都在github上，所有资源都在github上，所有资源都在github上 ChapterX- 本书的所有资源都在github上，所有资源都在github上，所有资源都在github上 Unity Shader 本书的所有资源都在github上，所有资源都在github上，所有资源都在github上 Chapter7-RampTexture
Assets/Materials	本书的所有资源都在github上，所有资源都在github上，所有资源都在github上 Assets/ Scenes/Chapter7 本书的所有资源都在github上，所有资源都在github上，所有资源都在github上 Unity Shader 本书的所有资源都在github上，所有资源都在github上，所有资源都在github上 Mat 本书的所有资源都在github上，所有资源都在github上，所有资源都在github上 Chapter7-RampTexture 本书的所有资源都在github上，所有资源都在github上，所有资源都在github上 Unity Shader 本书的所有资源都在github上，所有资源都在github上，所有资源都在github上 RampTextureMat
Assets/Scripts	本书的所有资源都在github上，所有资源都在github上，所有资源都在github上 C# 本书的所有资源都在github上，所有资源都在github上，所有资源都在github上 Assets/Scripts/Chapter5
Assets/Textures	本书的所有资源都在github上，所有资源都在github上，所有资源都在github上 Assets/Textures/Chapter7

本书的所有资源都在github上，所有资源都在github上，所有资源都在github上
 Assets/Editor 本书的所有资源都在github上，所有资源都在github上，所有资源都在github上
 Assets/Prefabs 本书的所有资源都在github上，所有资源都在github上，所有资源都在github上

□□□□

□ □ □ □

0000000000000000000000000000000000000000000000000
 000000000000000000000000github0000[https://
 github.com/candycat1992/Unity_Shaders_Book/issues](https://github.com/candycat1992/Unity_Shaders_Book/issues) 000000
 0000000000000000000000000000000000000000000000000
 0lelefeng1992@gmail.com0000000000QQ004381030990

□□□□□□□zhangtao@ptpress.com.cn□

11

Unity 3D ShaderLab

[illegible][illegible][illegible]

□ □

[illegible]

11

1 简介

Shader是计算机图形学中的一个重要概念，它定义了物体在光照下的颜色、透明度和反射等属性。在Unity Shader中，Shader是用于渲染3D模型的关键组件。

1.1 Shader的定义

Shader的定义通常包含两个部分：顶点着色器（Vertex Shader）和像素着色器（Pixel Shader）。在Unity Shader中，Shader的定义通常以“.shader"文件的形式存在，其中包含顶点着色器和像素着色器的代码。

2 Shader的编译

Shader的编译是将Shader源代码转换为GPU可以执行的二进制代码的过程。在Unity Shader中，Shader的编译通常由Unity引擎自动完成。

3 Unity Shader

Unity Shader是Unity引擎中用于渲染3D模型的关键组件。它定义了物体在光照下的颜色、透明度和反射等属性。

4 Shader的编写

Shader的编写通常涉及编写顶点着色器和像素着色器的代码。在Unity Shader中，Shader的编写通常以“.shader"文件的形式存在，其中包含顶点着色器和像素着色器的代码。

1 Shader

[illegible]

1.1 Shader 是什么

Shader 是计算机图形学中的一个重要概念，它定义了物体在光照下的颜色、透明度和其他视觉属性。在 Unity 中，Shader 是用于渲染 3D 模型的关键技术。通过 Shader，我们可以控制模型的颜色、透明度、反射率等属性，使其在光照下呈现出逼真的效果。

在 Unity 中，Shader 通常以 .shader 文件的形式存在。这些文件包含了一些用于渲染的指令，如材质属性、光照模型等。Unity 的 Shader 系统允许我们使用不同的 Shader 来渲染不同的模型。例如，我们可以使用 Standard Shader 来渲染具有金属质感的物体，或者使用 Transparent Shader 来渲染透明的玻璃物体。此外，我们还可以使用自定义 Shader 来实现一些特殊的效果，如粒子系统、后处理等。

Shader 的实现通常涉及到一些数学计算，如向量运算、矩阵变换等。在 Unity 中，我们通常使用 C# 语言来编写 Shader 的脚本。这些脚本会调用 Unity 提供的 Shader 系统接口，来实现对模型渲染的控制。此外，我们还可以使用 C++ 语言来编写 Shader 的底层实现，以获得更高的性能和更灵活的控制。总之，Shader 是 Unity 中实现 3D 渲染的核心技术之一，掌握 Shader 的编写和调试对于开发高质量的 3D 游戏至关重要。

在 Unity 中，Shader 的编写和调试是一个复杂的过程。我们需要了解 Shader 的基本原理和 Unity 的 Shader 系统。同时，我们还需要掌握一些数学知识和编程技巧，以便能够编写出高质量的 Shader。通过不断的学习和实践，我们可以逐步掌握 Shader 的编写和调试技巧，为开发高质量的 3D 游戏打下坚实的基础。

1.2 背景

本節主要介紹了 Shader 的相關背景知識。

● 背景

在 Unity 中，Shader 是用來控制物體的顏色、光澤、透明度等屬性。Unity 3.x 版本開始，Unity Shader 被引入。

2 在 Unity 中，GPU 是負責渲染物體的。Shader 是 GPU 用來渲染物體的代碼。

3 **Unity Shader** 是 Unity 提供的一種 Shader 語言。Unity Shader 是 Unity Shader 的一種實現。Unity Shader 是 Unity Shader 的一種實現。

4 **Shader** 是 Shader 的一種實現。Shader 是 Shader 的一種實現。Shader 是 Shader 的一種實現。Shader 是 Shader 的一種實現。

● 背景

在 Unity 中，Unity Shader 是 Unity Shader 的一種實現。Unity Shader 是 Unity Shader 的一種實現。Unity Shader 是 Unity Shader 的一種實現。Unity Shader 是 Unity Shader 的一種實現。

5 **Unity Shader** 是 Unity Shader 的一種實現。Unity Shader 是 Unity Shader 的一種實現。Unity Shader 是 Unity Shader 的一種實現。Unity Shader 是 Unity Shader 的一種實現。

6 **Unity** 是 Unity 的一種實現。Unity 是 Unity 的一種實現。Unity 是 Unity 的一種實現。Unity 是 Unity 的一種實現。

7 本稿では、Unity Shaderの基本的な書き方を紹介します。

8 UnityのShaderは、C#で書かれます。C#は、C++のサブセットです。C++の知識があれば、C#でも問題なく書けます。

● 環境

本稿では、Unity 3.5.4f3を前提としています。

9 Unity 3.5.4f3では、Shaderの書き方が変わりました。以前は、Shaderの書き方が、Unity Shaderと、Unity Shaderの2種類ありました。現在は、Unity Shaderの1種類になりました。

10 本稿では、Unity Shaderの基本的な書き方を紹介します。

11 Shaderの書き方は、Shaderの書き方と、Shaderの書き方の2種類があります。

● 環境

本稿では、Shaderの基本的な書き方を紹介します。

12 UnityのShaderは、C#で書かれます。C#は、C++のサブセットです。C++の知識があれば、C#でも問題なく書けます。

13 UnityのShaderは、C#で書かれます。C#は、C++のサブセットです。C++の知識があれば、C#でも問題なく書けます。

14 UnityのShaderは、C#で書かれます。C#は、C++のサブセットです。C++の知識があれば、C#でも問題なく書けます。

15 如何 在Unity中实现一个自定义的Shader？

16 Unity中如何实现一个自定义的Shader？

● 如何

在Unity中实现一个自定义的Shader？

17 Unity中如何实现一个自定义的Shader？

18 如何在Unity 5中实现一个自定义的Shader？

19 Unity 5中如何实现一个自定义的Shader？

20 如何在Unity 5中实现一个自定义的Shader？

如何在Unity 5中实现一个自定义的Shader？

2 环境

Shader 是 Unity 中用于渲染的着色器。Shader 是 Unity 中用于渲染的着色器。Shader 是 Unity 中用于渲染的着色器。

Shader 是 Unity 中用于渲染的着色器。Shader 是 Unity 中用于渲染的着色器。Shader 是 Unity 中用于渲染的着色器。

Shader 是 Unity 中用于渲染的着色器。Shader 是 Unity 中用于渲染的着色器。Shader 是 Unity 中用于渲染的着色器。

2.1 并行

每个 Shader 单元都是一个 Shader 单元，每个 Shader 单元都是一个 Shader 单元，每个 Shader 单元都是一个 Shader 单元，每个 Shader 单元都是一个 Shader 单元。

2.1.1 并行性

并行性是指多个 Shader 单元可以同时执行相同的操作。

每个 Shader 单元都有一个 4 个输入和 1 个输出，每个 Shader 单元都有一个 2 个输入和 3 个输出，每个 Shader 单元都有一个 4 个输入和 1 个输出。

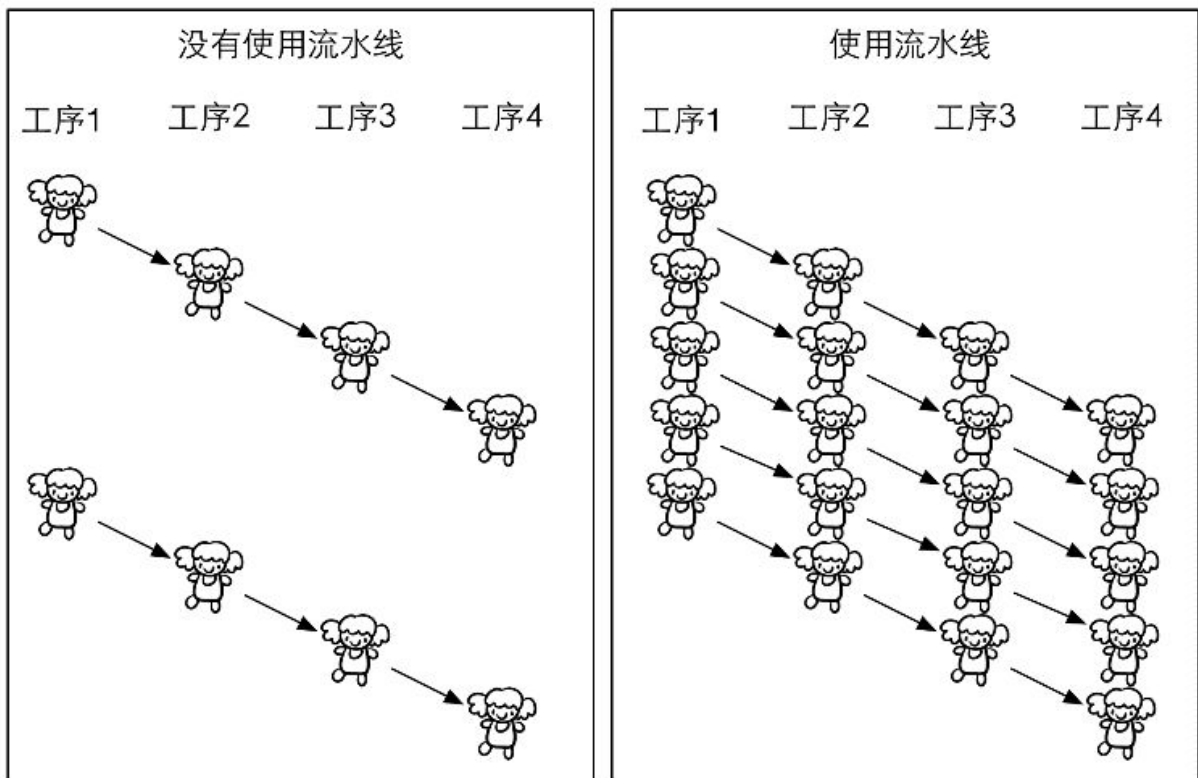
每个 Shader 单元都有一个 4 个输入和 1 个输出，每个 Shader 单元都有一个 1 个输入和 4 个输出。

每个 Shader 单元都有一个 4 个输入和 1 个输出，每个 Shader 单元都有一个 1 个输入和 2 个输出，每个 Shader 单元都有一个 1 个输入和 1 个输出。

每个 Shader 单元都有一个 1 个输入和 1 个输出，每个 Shader 单元都有一个 2.1 个输入和 1 个输出。

每个 Shader 单元都有一个 1 个输入和 1 个输出，每个 Shader 单元都有一个 2 个输入和 1 个输出，每个 Shader 单元都有一个 bottleneck。

每个 Shader 单元都有一个 n 个输入和 1 个输出，每个 Shader 单元都有一个 n 个输入和 1 个输出。



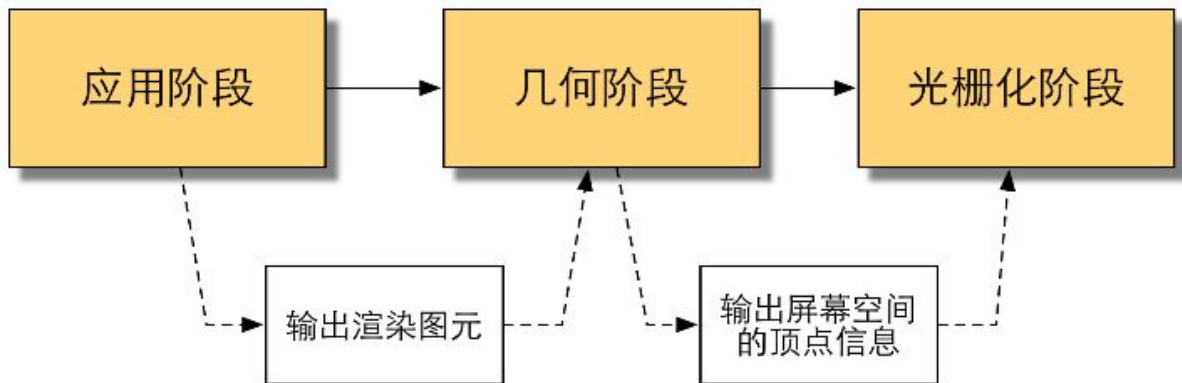
▲图2.1 流水线示意图

2.1.2 流水线模型

流水线模型是一种并行处理模型，它将一个任务分解成多个子任务，每个子任务由一个处理器（CPU或GPU）并行执行。流水线模型广泛应用于图形渲染、信号处理、数据分析和人工智能等领域。

在《Render-Time Rendering, Third Edition》^[1] 中，流水线模型被分为三个阶段：Application Stage、Geometry Stage 和 Rasterizer Stage。

图2.2展示了流水线模型的结构。图3展示了流水线模型的应用。



▲2.2 3D渲染流程

● 应用阶段

应用阶段主要负责将3D模型数据加载到GPU，并设置初始的渲染状态。这个阶段通常由CPU控制，将模型数据（顶点、索引、材质等）上传到GPU，并设置初始的渲染状态（如清除颜色缓冲区、深度缓冲区等）。

应用阶段的主要任务包括：3D模型数据的加载和上传、初始渲染状态的设置、视锥体剔除（**culling**）等。在应用阶段结束后，GPU会开始渲染图元（**rendering primitives**），生成屏幕空间的顶点信息，为后续的几何阶段和光栅化阶段做准备。

应用阶段是3D渲染流程的第一步，也是基础性的工作。它确保了GPU拥有正确的模型数据和初始状态，为后续的渲染工作奠定基础。

● 几何阶段

几何阶段主要负责对渲染图元进行几何变换和投影。这个阶段通常由GPU控制，将应用阶段输出的渲染图元进行几何变换（如平移、旋转、缩放等），并将其投影到屏幕空间。

几何阶段的主要任务包括：几何变换、投影变换、视锥体剔除（**culling**）等。在几何阶段结束后，GPU会生成屏幕空间的顶点信息，为后续的光栅化阶段做准备。

● 環境構築

環境構築は、まずGPUをインストールする。GPUは、
インストールする。GPUは、
インストールする。

環境構築は、まずGPUをインストールする。



環境構築は、まずGPUをインストールする。GPUは、
インストールする。GPUは、
インストールする。

2.2 CPU与GPU的区别

1. CPU与GPU的区别

2. CPU与GPU的区别

3. CPU与GPU的区别

4. CPU与GPU的区别

2.2.1 CPU与GPU的区别

1. CPU与GPU的区别

2. CPU与GPU的区别

3. CPU与GPU的区别

4. CPU与GPU的区别

5. CPU与GPU的区别

6. CPU与GPU的区别

7. CPU与GPU的区别

8. CPU与GPU的区别

9. CPU与GPU的区别

10. CPU与GPU的区别

11. CPU与GPU的区别

12. CPU与GPU的区别

13. CPU与GPU的区别

14. CPU与GPU的区别

15. CPU与GPU的区别

16. CPU与GPU的区别

17. CPU与GPU的区别

18. CPU与GPU的区别

19. CPU与GPU的区别

20. CPU与GPU的区别

21. CPU与GPU的区别

22. CPU与GPU的区别

23. CPU与GPU的区别

24. CPU与GPU的区别

25. CPU与GPU的区别

26. CPU与GPU的区别

27. CPU与GPU的区别

28. CPU与GPU的区别

29. CPU与GPU的区别

30. CPU与GPU的区别

31. CPU与GPU的区别

32. CPU与GPU的区别

33. CPU与GPU的区别

34. CPU与GPU的区别

35. CPU与GPU的区别

36. CPU与GPU的区别

37. CPU与GPU的区别

38. CPU与GPU的区别

39. CPU与GPU的区别

40. CPU与GPU的区别

41. CPU与GPU的区别

42. CPU与GPU的区别

43. CPU与GPU的区别

44. CPU与GPU的区别

45. CPU与GPU的区别

46. CPU与GPU的区别

47. CPU与GPU的区别

48. CPU与GPU的区别

49. CPU与GPU的区别

50. CPU与GPU的区别

51. CPU与GPU的区别

52. CPU与GPU的区别

53. CPU与GPU的区别

54. CPU与GPU的区别

55. CPU与GPU的区别

56. CPU与GPU的区别

57. CPU与GPU的区别

58. CPU与GPU的区别

59. CPU与GPU的区别

60. CPU与GPU的区别

61. CPU与GPU的区别

62. CPU与GPU的区别

63. CPU与GPU的区别

64. CPU与GPU的区别

65. CPU与GPU的区别

66. CPU与GPU的区别

67. CPU与GPU的区别

68. CPU与GPU的区别

69. CPU与GPU的区别

70. CPU与GPU的区别

71. CPU与GPU的区别

72. CPU与GPU的区别

73. CPU与GPU的区别

74. CPU与GPU的区别

75. CPU与GPU的区别

76. CPU与GPU的区别

77. CPU与GPU的区别

78. CPU与GPU的区别

79. CPU与GPU的区别

80. CPU与GPU的区别

81. CPU与GPU的区别

82. CPU与GPU的区别

83. CPU与GPU的区别

84. CPU与GPU的区别

85. CPU与GPU的区别

86. CPU与GPU的区别

87. CPU与GPU的区别

88. CPU与GPU的区别

89. CPU与GPU的区别

90. CPU与GPU的区别

91. CPU与GPU的区别

92. CPU与GPU的区别

93. CPU与GPU的区别

94. CPU与GPU的区别

95. CPU与GPU的区别

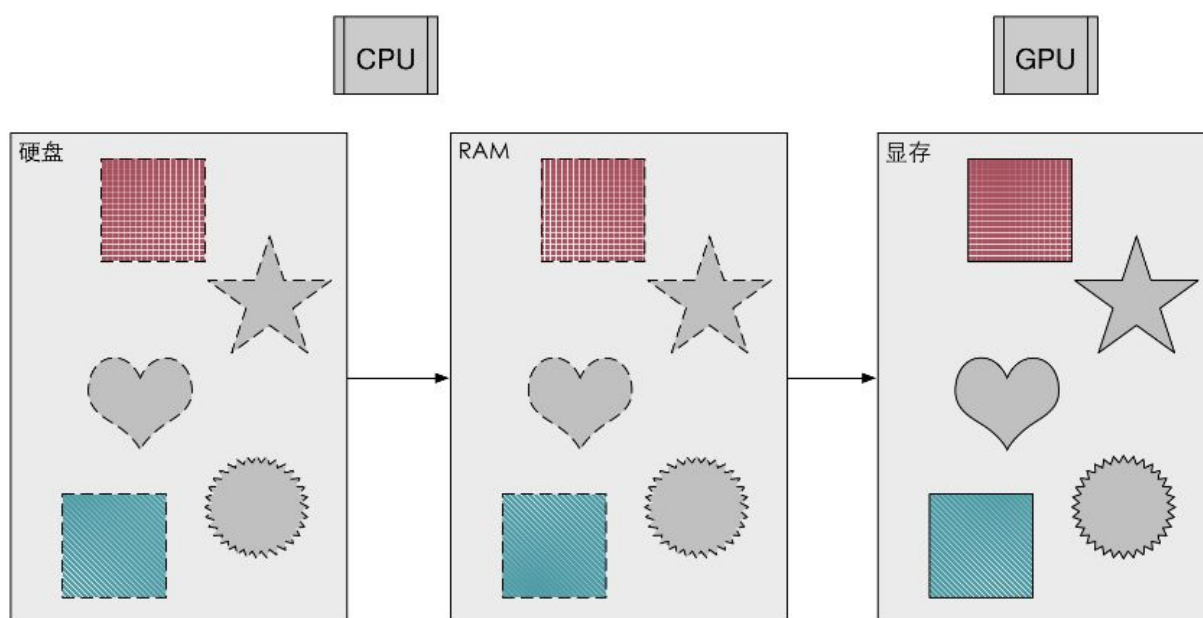
96. CPU与GPU的区别

97. CPU与GPU的区别

98. CPU与GPU的区别

99. CPU与GPU的区别

100. CPU与GPU的区别



2.3 CPU与GPU的区别

1. CPU与GPU的区别

2. CPU与GPU的区别

3. CPU与GPU的区别

4. CPU与GPU的区别

5. CPU与GPU的区别

6. CPU与GPU的区别

7. CPU与GPU的区别

8. CPU与GPU的区别

9. CPU与GPU的区别

10. CPU与GPU的区别

11. CPU与GPU的区别

12. CPU与GPU的区别

13. CPU与GPU的区别

14. CPU与GPU的区别

15. CPU与GPU的区别

16. CPU与GPU的区别

17. CPU与GPU的区别

18. CPU与GPU的区别

19. CPU与GPU的区别

20. CPU与GPU的区别

21. CPU与GPU的区别

22. CPU与GPU的区别

23. CPU与GPU的区别

24. CPU与GPU的区别

25. CPU与GPU的区别

26. CPU与GPU的区别

27. CPU与GPU的区别

28. CPU与GPU的区别

29. CPU与GPU的区别

30. CPU与GPU的区别

31. CPU与GPU的区别

32. CPU与GPU的区别

33. CPU与GPU的区别

34. CPU与GPU的区别

35. CPU与GPU的区别

36. CPU与GPU的区别

37. CPU与GPU的区别

38. CPU与GPU的区别

39. CPU与GPU的区别

40. CPU与GPU的区别

41. CPU与GPU的区别

42. CPU与GPU的区别

43. CPU与GPU的区别

44. CPU与GPU的区别

45. CPU与GPU的区别

46. CPU与GPU的区别

47. CPU与GPU的区别

48. CPU与GPU的区别

49. CPU与GPU的区别

50. CPU与GPU的区别

51. CPU与GPU的区别

52. CPU与GPU的区别

53. CPU与GPU的区别

54. CPU与GPU的区别

55. CPU与GPU的区别

56. CPU与GPU的区别

57. CPU与GPU的区别

58. CPU与GPU的区别

59. CPU与GPU的区别

60. CPU与GPU的区别

61. CPU与GPU的区别

62. CPU与GPU的区别

63. CPU与GPU的区别

64. CPU与GPU的区别

65. CPU与GPU的区别

66. CPU与GPU的区别

67. CPU与GPU的区别

68. CPU与GPU的区别

69. CPU与GPU的区别

70. CPU与GPU的区别

71. CPU与GPU的区别

72. CPU与GPU的区别

73. CPU与GPU的区别

74. CPU与GPU的区别

75. CPU与GPU的区别

76. CPU与GPU的区别

77. CPU与GPU的区别

78. CPU与GPU的区别

79. CPU与GPU的区别

80. CPU与GPU的区别

81. CPU与GPU的区别

82. CPU与GPU的区别

83. CPU与GPU的区别

84. CPU与GPU的区别

85. CPU与GPU的区别

86. CPU与GPU的区别

87. CPU与GPU的区别

88. CPU与GPU的区别

89. CPU与GPU的区别

90. CPU与GPU的区别

91. CPU与GPU的区别

92. CPU与GPU的区别

93. CPU与GPU的区别

94. CPU与GPU的区别

95. CPU与GPU的区别

96. CPU与GPU的区别

97. CPU与GPU的区别

98. CPU与GPU的区别

99. CPU与GPU的区别

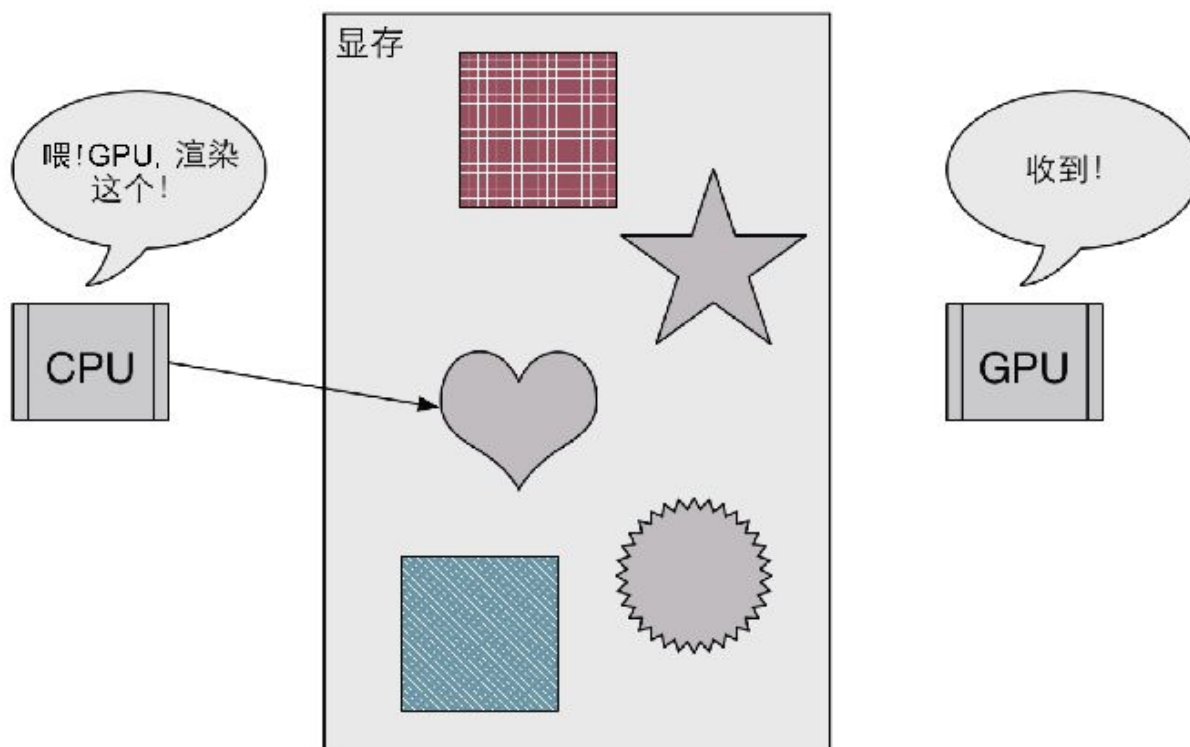
100. CPU与GPU的区别

CPU告诉GPU“我要渲染这个”——这就是Draw Call

2.2.3 Draw Call

Draw Call 告诉GPU要渲染什么primitives。CPU告诉GPU要渲染什么primitives——这就是Draw Call。2.5

Draw Call告诉GPU要渲染什么。GPU收到Draw Call后，开始渲染。GPU



▲2.5 CPU告诉GPU要渲染什么 Draw Call告诉GPU要渲染什么

2.3 GPU

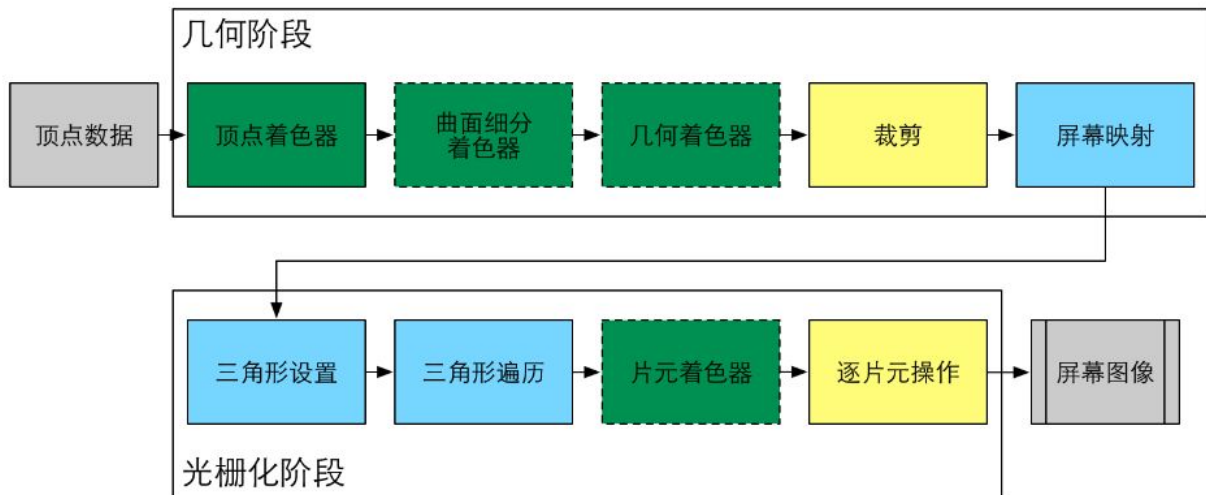
GPU CPU

2.3.1 〇〇

○○○○○○○○○○○○○○○○○○○○CPU○○○○GPU○○○○○○○○○○Draw Call○○○
GPU○○○○○○GPU○○○○○○○○GPU○○○○

The diagram shows three GPU units arranged horizontally. Each unit consists of a square box labeled "GPU". The first two GPUs are connected to each other by a horizontal line. A vertical line extends from the midpoint of this connection to a central horizontal bus line. This bus line connects to the third GPU on the right.

GPU GPU
2.6



▲ 2.6 GPU Shader

GPU Shader

GPU Draw Call

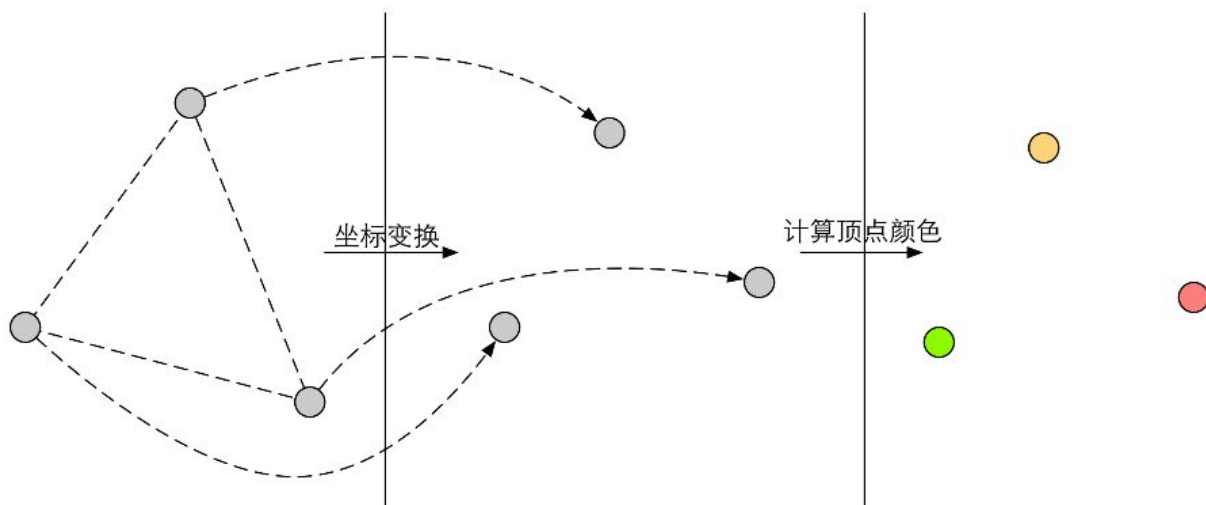
Vertex Shader Tessellation Shader Geometry Shader Per-Primitive Clipping Screen Mapping

Triangle Setup Triangle Traversal Fixed-Function Fragment Shader Per-Fragment Per-Fragment Operations

2.3.2

Vertex Shader CPU GPU

2.7



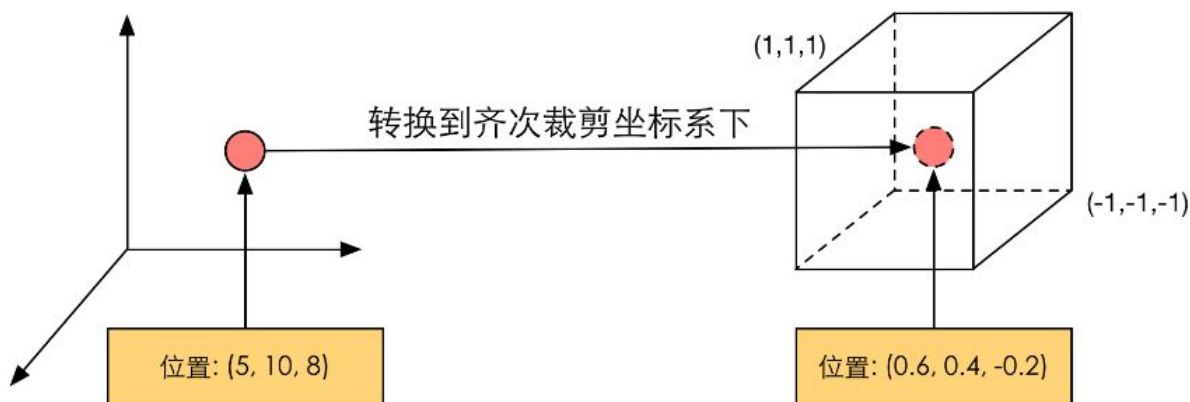
▲2.7 GPU坐标变换与裁剪

● 在GPU中，坐标变换和裁剪是两个非常重要的步骤。坐标变换是将物体从局部坐标系转换到世界坐标系，而裁剪则是将物体裁剪到视锥体内。这两个步骤是GPU渲染流程中的核心部分。

```
o.pos = mul(UNITY_MVP, v.position);
```

在GPU中，坐标变换和裁剪是两个非常重要的步骤。坐标变换是将物体从局部坐标系转换到世界坐标系，而裁剪则是将物体裁剪到视锥体内。这两个步骤是GPU渲染流程中的核心部分。

Normalized Device Coordinates (NDC) 是GPU渲染流程中的一个重要概念。NDC是一个归一化的坐标系统，其范围通常为[-1, 1]。在NDC中，物体的位置可以通过其坐标值来唯一确定。



▲2.8 坐标系的转换和NDC坐标

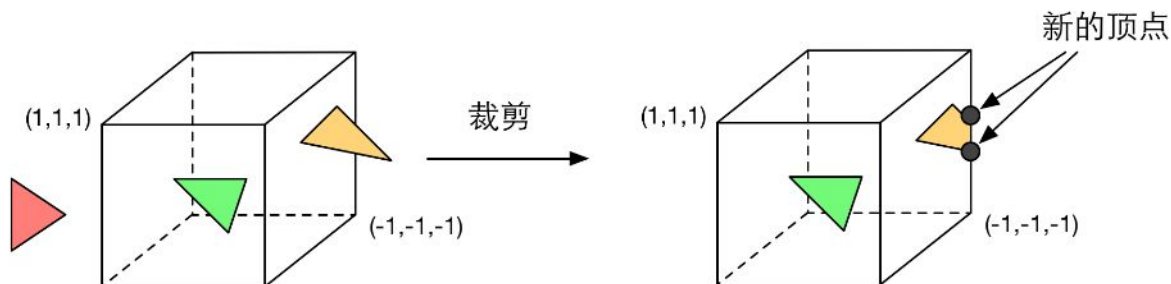
在2.8节中我们介绍了OpenGL和Unity的NDC坐标，z轴在 $[-1, 1]$ 范围内，而DirectX的NDC z轴在 $[0, 1]$ 范围内。在Shader Model 5.0之前，DirectX的NDC z轴在 $[-1, 1]$ 范围内，但在5.0之后，它被更改为 $[0, 1]$ 。因此，在编写Shader时，需要注意这些差异。

2.3.3 裁剪

在3D图形中，裁剪（Clipping）是一个重要的步骤，用于移除那些不在当前视口范围内的几何体。这通常是在NDC空间中进行的操作。

在3D空间中，一个物体可能部分位于视口之外。通过裁剪，我们可以只保留那些在视口内的部分。这通常涉及到计算物体与视口平面的交点，并移除那些在平面之外的部分。

在NDC空间中，裁剪通常是在x、y和z轴上分别进行的。这确保了只有那些在视口范围内的几何体才会被渲染。



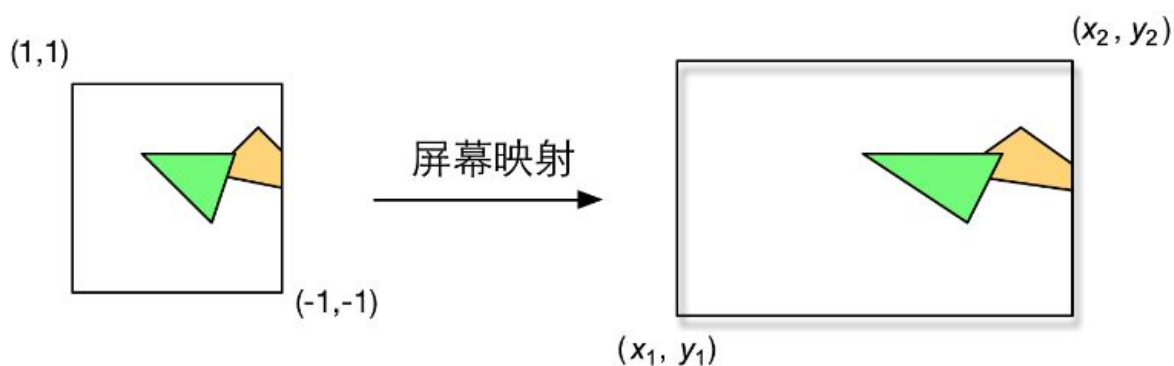
▲2.9 坐标系的转换和NDC坐标

在2.9节中，我们讨论了坐标系的转换和NDC坐标。这涉及到将物体从世界空间转换到NDC空间，以便进行裁剪和渲染。

2.3.4 投影

窗口坐标到屏幕坐标的映射
Screen Mapping 窗口坐标 x, y 到屏幕坐标 **Screen Coordinates** 的映射

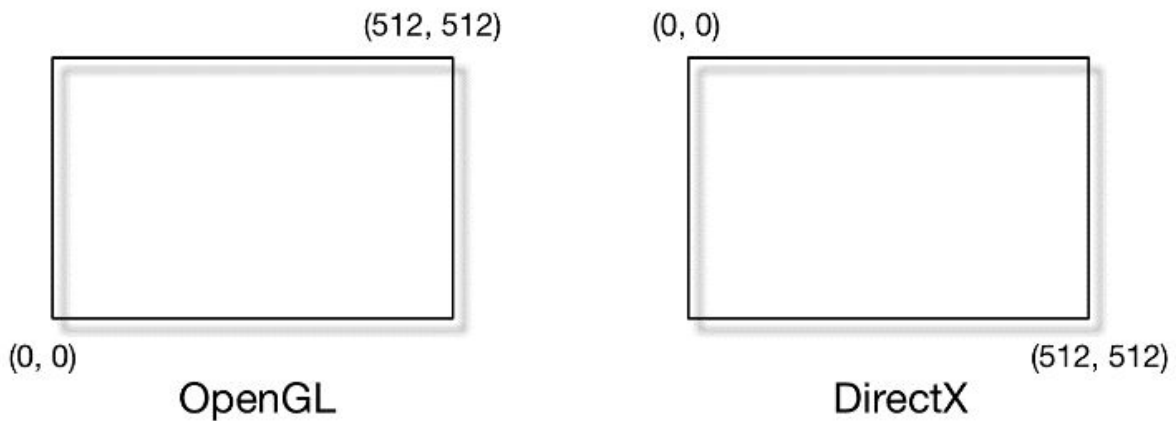
窗口坐标 (x_1, y_1) 到窗口坐标 (x_2, y_2) 的映射 $x_1 < x_2, y_1 < y_2$ 窗口坐标范围 $-1 \sim 1$
 窗口坐标范围 2.10 窗口坐标范围 z 窗口坐标范围 z 窗口坐标范围 z
Window Coordinates 窗口坐标



▲2.10 窗口坐标 x, y 范围 $-1, 1$ 窗口坐标范围

窗口坐标范围

窗口坐标范围 **OpenGL** **DirectX** 窗口坐标范围
OpenGL 窗口坐标范围 **DirectX** 窗口坐标范围
 窗口2.11 窗口坐标范围



▲2.11 OpenGL과 DirectX의 차이점
 OpenGL은 (0, 0)을 원점으로 하고, DirectX은 (0, 0)을 상단좌측으로 한다.

OpenGL은 좌표계를 (0, 0)에서 (512, 512)로 설정한다.
 DirectX은 좌표계를 (0, 0)에서 (512, 512)로 설정한다.

2.3.5 렌더링

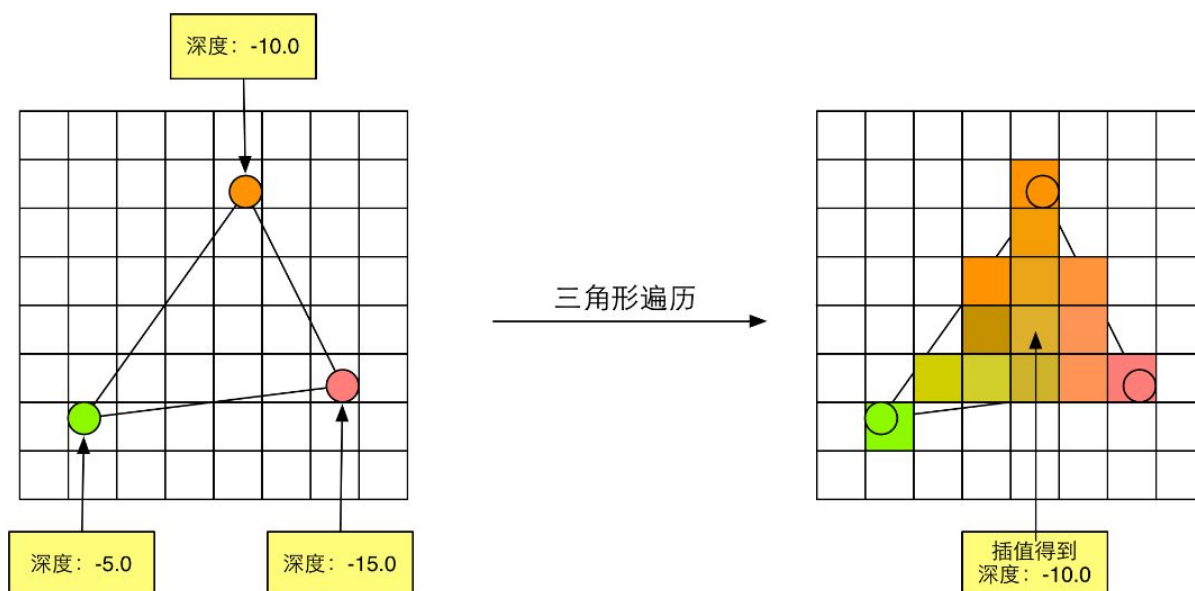
렌더링은 화면에 이미지를 출력하는 과정이다.
 z 버퍼를 사용하여 깊이 정보를 처리한다.

렌더링의 첫 단계는 **Triangle Setup**이다.
 삼각형의 꼭짓점을 설정하고, 색상을 지정한다.

2.3.6 렌더링

렌더링의 두 번째 단계는 **Triangle Traversal**이다.
 fragment를 생성하고, **Scan Conversion**을 수행한다.

深度值插值 3 个顶点的深度值，通过插值计算出三角形内部每个像素的深度值。2.12 三角形遍历



▲2.12 三角形遍历 3 个顶点的深度值，通过插值计算出三角形内部每个像素的深度值。2.12 3 个顶点的深度值，通过插值计算出三角形内部每个像素的深度值。-10.0

深度值插值 3 个顶点的深度值，通过插值计算出三角形内部每个像素的深度值。2.12 3 个顶点的深度值，通过插值计算出三角形内部每个像素的深度值。-10.0

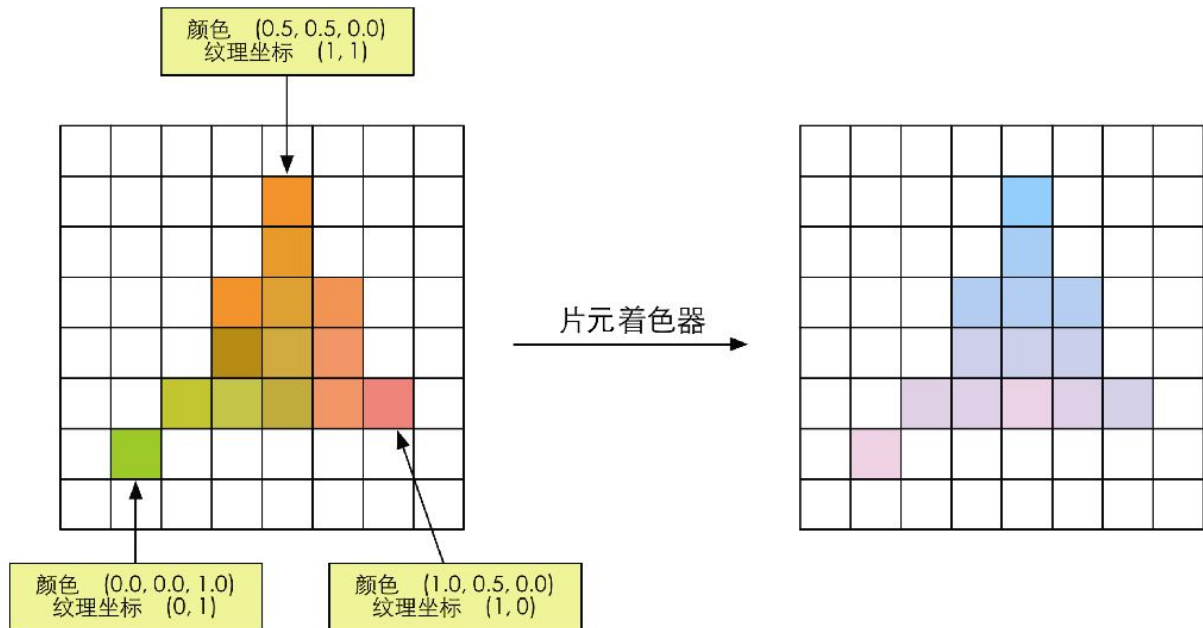
2.3.7 深度值插值

Fragment Shader 像素着色器 DirectX 像素着色器 Pixel Shader 像素着色器

Per-Fragment Operations 像素着色器

2.13 像素着色器

片元着色器
 片元着色器
 片元着色器



▲2.13 片元着色器

片元着色器
 片元着色器
 gradient derivative

2.3.8 片元着色器

片元着色器
Per-Fragment Operations
 OpenGL DirectX
Output-Merger
 Merger OpenGL
 片元着色器

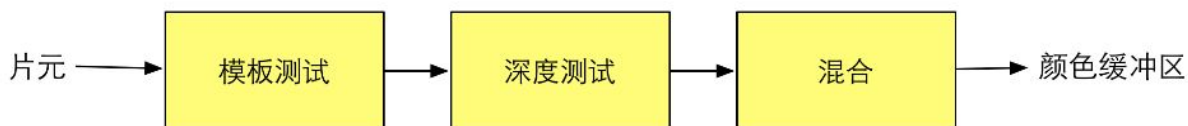
片元着色器

1 片元着色器

2. 片元着色器（Fragment Shader）：对每个片元进行颜色计算，并决定是否丢弃该片元。它接收来自顶点着色器的数据，并根据纹理采样和光照模型进行计算。输出结果存储在颜色缓冲区中。

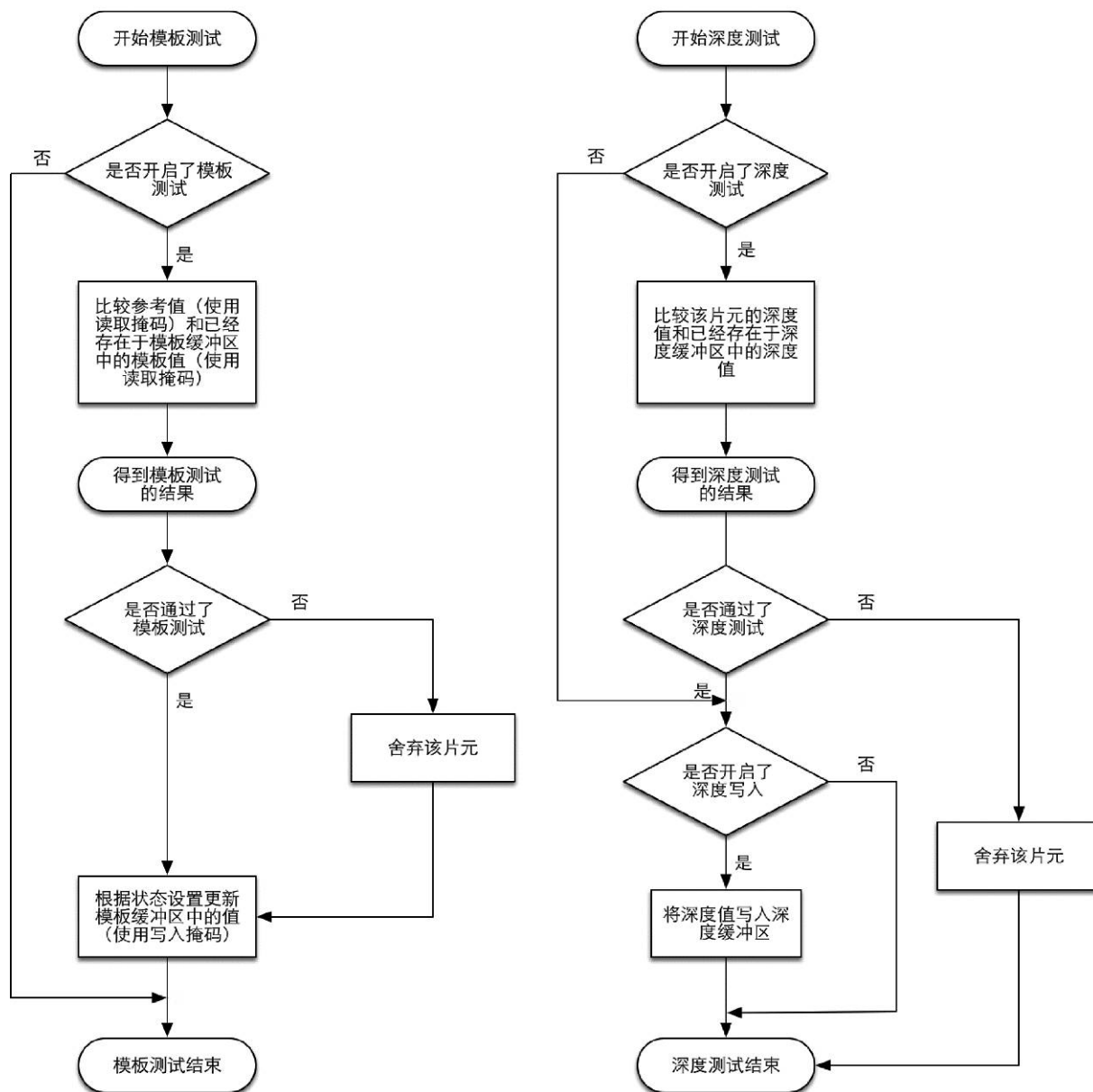
片元着色器通常与顶点着色器一起使用，共同完成图形的渲染。

片元着色器在GPU中执行，其输出结果存储在颜色缓冲区中。在片元着色器中，我们通常使用纹理采样函数来从纹理中获取颜色值。此外，我们还可以通过光照模型来计算片元的最终颜色。图2.14展示了片元着色器的基本流程。



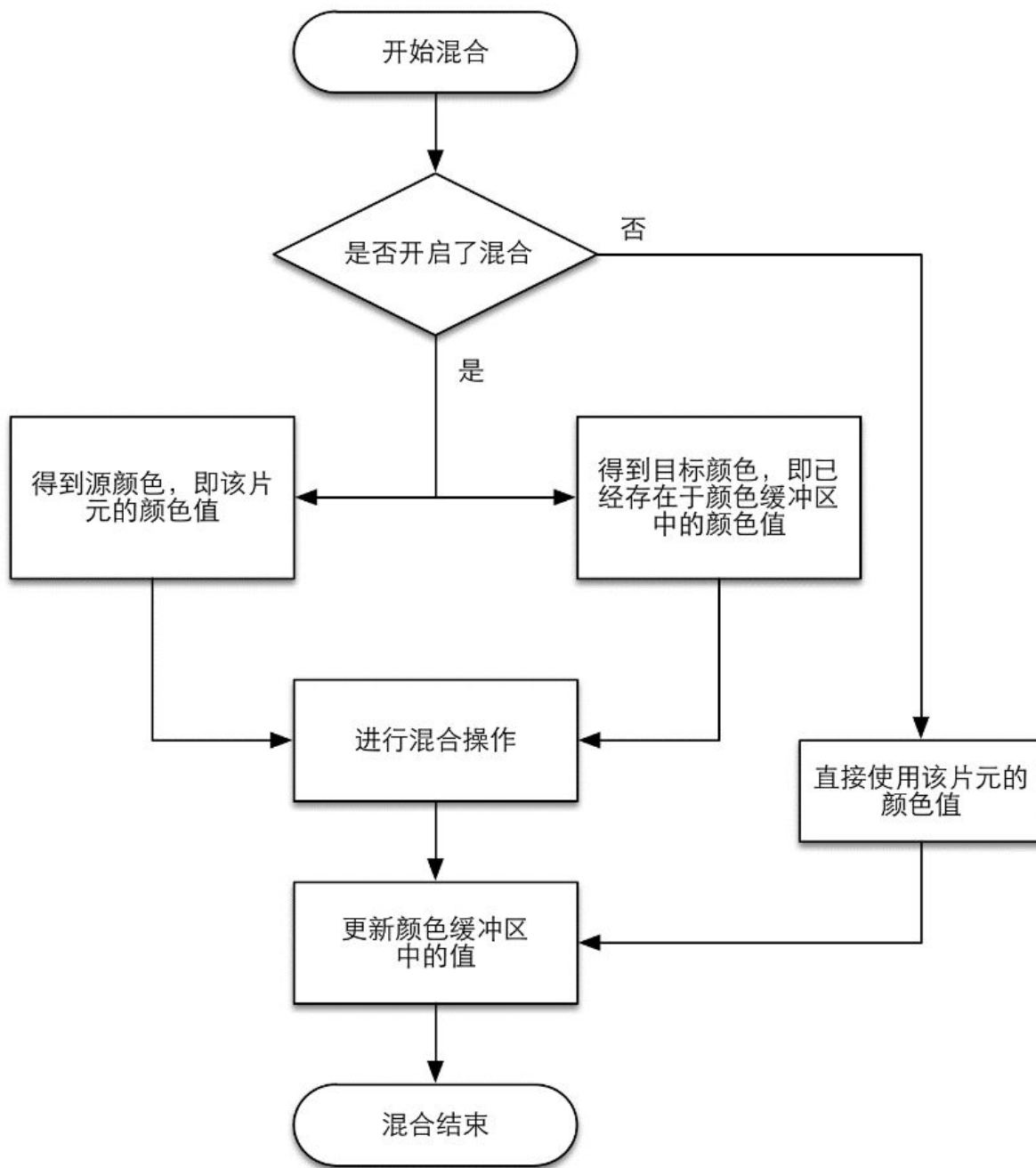
▲图2.14 片元着色器的基本流程

在OpenGL和DirectX等图形API中，片元着色器通常通过顶点着色器来调用。在OpenGL中，我们使用gl_FragData来访问片元着色器的输出。在DirectX中，我们使用OutputMerger来合并片元着色器的输出。图2.15展示了片元着色器在OpenGL中的调用流程。



▲2.15 深度测试与模板测试

深度测试与模板测试是GPU渲染管线中的两个重要步骤。在深度测试中，GPU会比较当前片元的深度值与已经存在于深度缓冲区中的深度值。如果当前片元的深度值大于深度缓冲区中的深度值，则该片元将被舍弃。在模板测试中，GPU会比较当前片元的模板值与已经存在于模板缓冲区中的模板值。如果当前片元的模板值与模板缓冲区中的模板值不匹配，则该片元将被舍弃。模板测试通常用于实现透明物体的正确渲染。

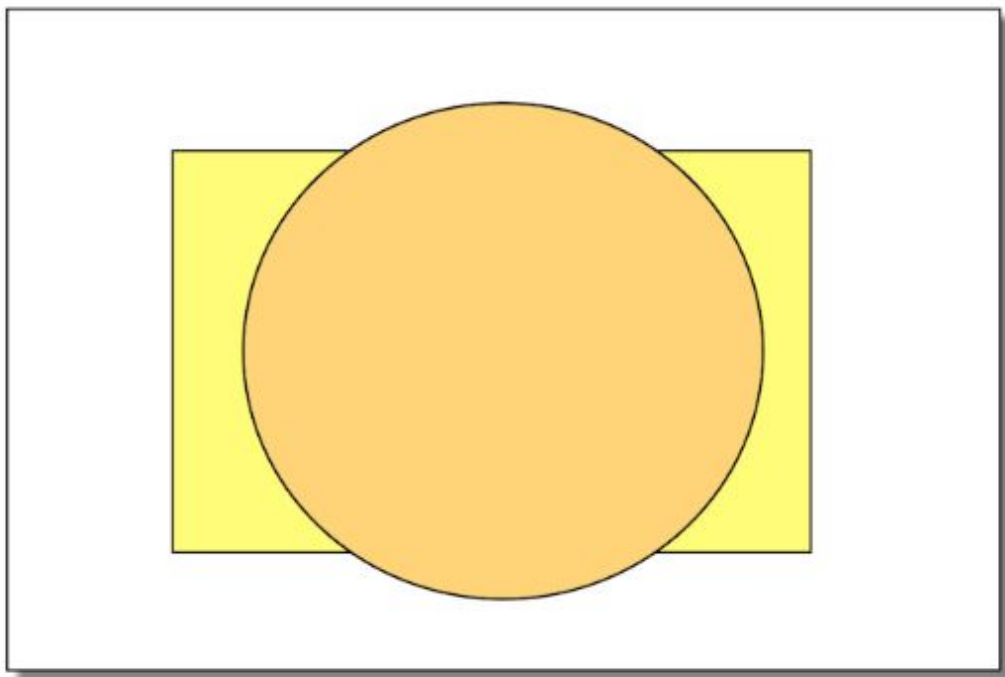


▲图2.16 混合流程图

在图形学中，混合（Blending）是指将两个或多个颜色值按照一定的规则（如透明度）进行组合，以生成最终显示的颜色。这个过程通常由GPU（图形处理单元）完成。在混合过程中，源颜色（Source Color）和目标颜色（Destination Color）是关键因素。源颜色是指当前片元的颜色值，而目标颜色是指已经存在于颜色缓冲区中的颜色值。混合操作的结果会更新颜色缓冲区中的值，以便后续片元的渲染。

PhotoshopのレンダリングエンジンがGPUに切り替わったことについて、
詳しく説明されている記事があります。

この記事は、PhotoshopのレンダリングエンジンがGPUに切り替わったことについて、
GPUの活用方法や、GPUの活用によるメリット、GPUの活用によるデメリット、
GPUの活用による今後の展望について詳しく説明されています。
2.17のバージョンで、GPUの活用がサポートされています。



▲2.17 GPUの活用方法について、
詳しく説明されている記事があります。
GPUの活用方法について詳しく説明されています。

GPUの活用方法について、
UnityのGPUの活用方法について詳しく説明されています。
Early-Zの活用方法について詳しく説明されています。
GPUの活用方法について詳しく説明されています。

GPUの活用方法について、
8.3のバージョンで、GPUの活用がサポートされています。
APIの活用方法について詳しく説明されています。
GPUの活用方法について詳しく説明されています。

GPU 的渲染管线中，顶点着色器（Vertex Shader）和几何着色器（Geometry Shader）是负责处理顶点数据的。它们接收从 CPU 传来的顶点数据，并根据指定的着色器程序进行计算，生成新的顶点数据。这个过程通常被称为“顶点变换”或“顶点着色”。

在 GPU 的渲染管线中，顶点着色器（Vertex Shader）和几何着色器（Geometry Shader）是负责处理顶点数据的。它们接收从 CPU 传来的顶点数据，并根据指定的着色器程序进行计算，生成新的顶点数据。这个过程通常被称为“顶点变换”或“顶点着色”。

Double Buffering 是一种用于避免数据冲突的技术。在 GPU 渲染中，它通常用于管理帧缓冲区（Frame Buffer）。通过维护两个缓冲区，一个用于当前帧的渲染，另一个用于存储前一帧的渲染结果，可以避免在渲染过程中覆盖前一帧的数据，从而确保渲染结果的完整性。

Back Buffer 是 GPU 渲染管线中的一个缓冲区，用于存储前一帧的渲染结果。它通常与 **Front Buffer** 一起使用，以实现双缓冲（Double Buffering）。在渲染过程中，新的帧数据被写入 **Front Buffer**，而前一帧的数据则保留在 **Back Buffer** 中，直到前一帧的渲染完成并显示在屏幕上。

2.3.9 渲染

在 GPU 的渲染管线中，顶点着色器（Vertex Shader）和几何着色器（Geometry Shader）是负责处理顶点数据的。它们接收从 CPU 传来的顶点数据，并根据指定的着色器程序进行计算，生成新的顶点数据。这个过程通常被称为“顶点变换”或“顶点着色”。

OpenGL 和 DirectX 是两种常用的 GPU 渲染 API。它们提供了丰富的接口，用于控制 GPU 的渲染过程。在 OpenGL 中，渲染过程通常涉及创建顶点数组对象（Vertex Array Object, VAO）和顶点缓冲对象（Vertex Buffer Object, VBO），并指定着色器程序。而在 DirectX 中，渲染过程则涉及创建顶点缓冲对象（Vertex Buffer）和索引缓冲对象（Index Buffer），并指定着色器程序。

Unity 是一个流行的游戏引擎，它提供了强大的渲染功能。在 Unity 中，渲染过程通常涉及创建材质（Material）和渲染器（Renderer）。材质定义了物体的外观，包括颜色、纹理和着色器。渲染器则负责将材质应用到模型上，并生成最终的渲染结果。Unity 还提供了丰富的 API，用于控制渲染过程，包括设置渲染质量、启用/禁用渲染功能等。

Unity Shader 是 Unity 引擎中用于定义着色器程序的脚本。它们通常使用 ShaderLab 语言编写，用于描述着色器的输入、输出和计算逻辑。Unity Shader 可以与材质（Material）结合使用，为模型赋予不同的外观。通过编写自定义的 Unity Shader，可以实现各种复杂的渲染效果，如粒子系统、后处理效果等。

在 GPU 的渲染管线中，顶点着色器（Vertex Shader）和几何着色器（Geometry Shader）是负责处理顶点数据的。它们接收从 CPU 传来的顶点数据，并根据指定的着色器程序进行计算，生成新的顶点数据。这个过程通常被称为“顶点变换”或“顶点着色”。

渲染管线（Rendering Pipeline）是 GPU 渲染过程中的核心部分。它定义了从输入数据到最终渲染结果的整个流程。通常，渲染管线包括顶点着色、几何着色、光栅化（Rasterization）和像素着色（Pixel Shader）等阶段。每个阶段都负责处理特定的数据，并最终生成可供显示器显示的像素数据。

2.4 图形接口

图形Shader接口是图形接口的重要组成部分，它定义了图形API与GPU之间的交互方式。

2.4.1 图形API: OpenGL/DirectX

图形API是连接CPU与GPU的桥梁，主要分为OpenGL和DirectX两种。OpenGL是跨平台的图形API，而DirectX是微软开发的图形API，主要用于Windows和Xbox平台。

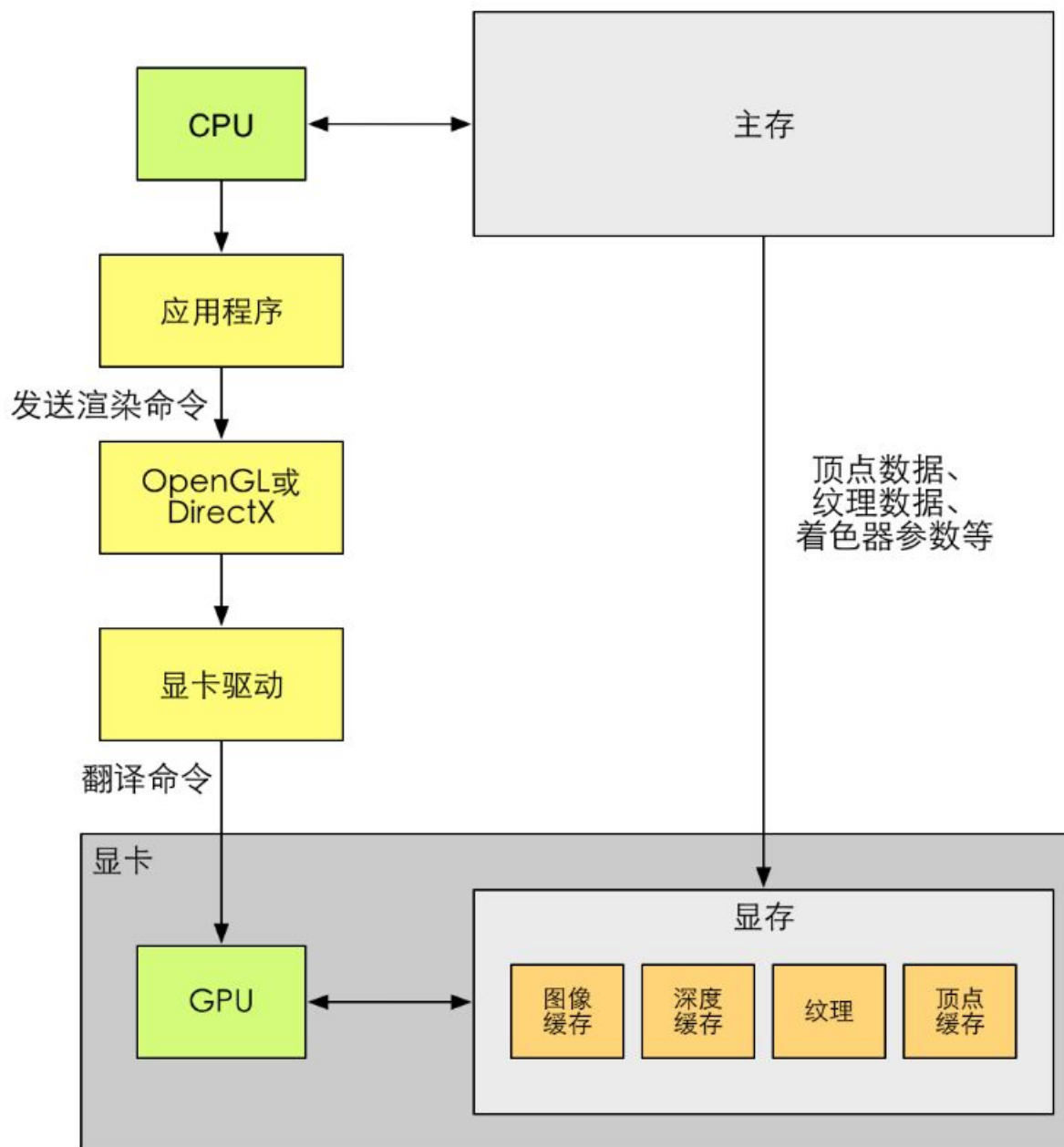
图形API的主要作用是管理GPU资源，包括顶点、纹理、着色器等。它通过调用GPU的接口来实现图形的渲染。

OpenGL和DirectX都提供了丰富的接口，用于管理GPU资源。它们通过调用GPU的接口来实现图形的渲染。Graphics Driver是连接CPU与GPU的桥梁，它负责管理GPU资源，并调用GPU的接口来实现图形的渲染。

CPU通过调用OpenGL或DirectX的接口来管理GPU资源。Draw Call是CPU向GPU发出的命令，用于启动图形的渲染。

2.18节介绍了GPU的Video Random Access Memory (VRAM)。VRAM是GPU专用的内存，用于存储纹理、顶点数据等。

GPU通过OpenGL或DirectX接口与CPU进行通信。OpenGL和DirectX都提供了丰富的接口，用于管理GPU资源。



▲图2.18 CPU与OpenGL/DirectX接口与GPU接口

2.4.2 学习HLSL、GLSL、CG

学习HLSL、GLSL、CG，需要了解一些数学知识，如线性代数、微积分等。C#是学习HLSL、GLSL、CG的常用语言。

Draw Call GPU GPU
“ ” CPU

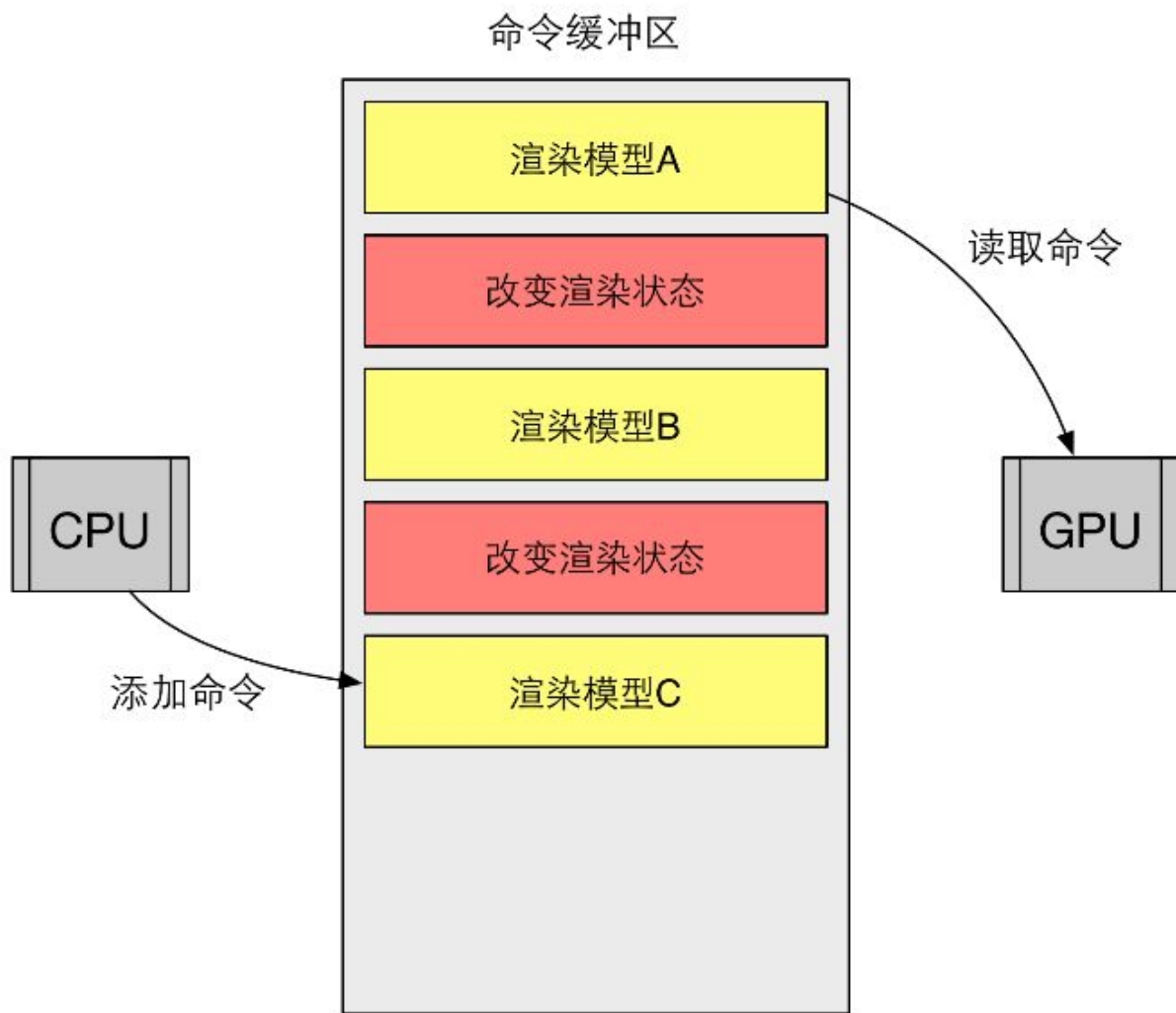
Draw Call CPU GPU

CPU GPU

CPU GPU
CPU GPU
Command Buffer

CPU GPU
CPU GPU CPU
GPU

Draw Call
2.19



▲2.19 CPU命令缓冲区与GPU命令缓冲区之间的Draw Call

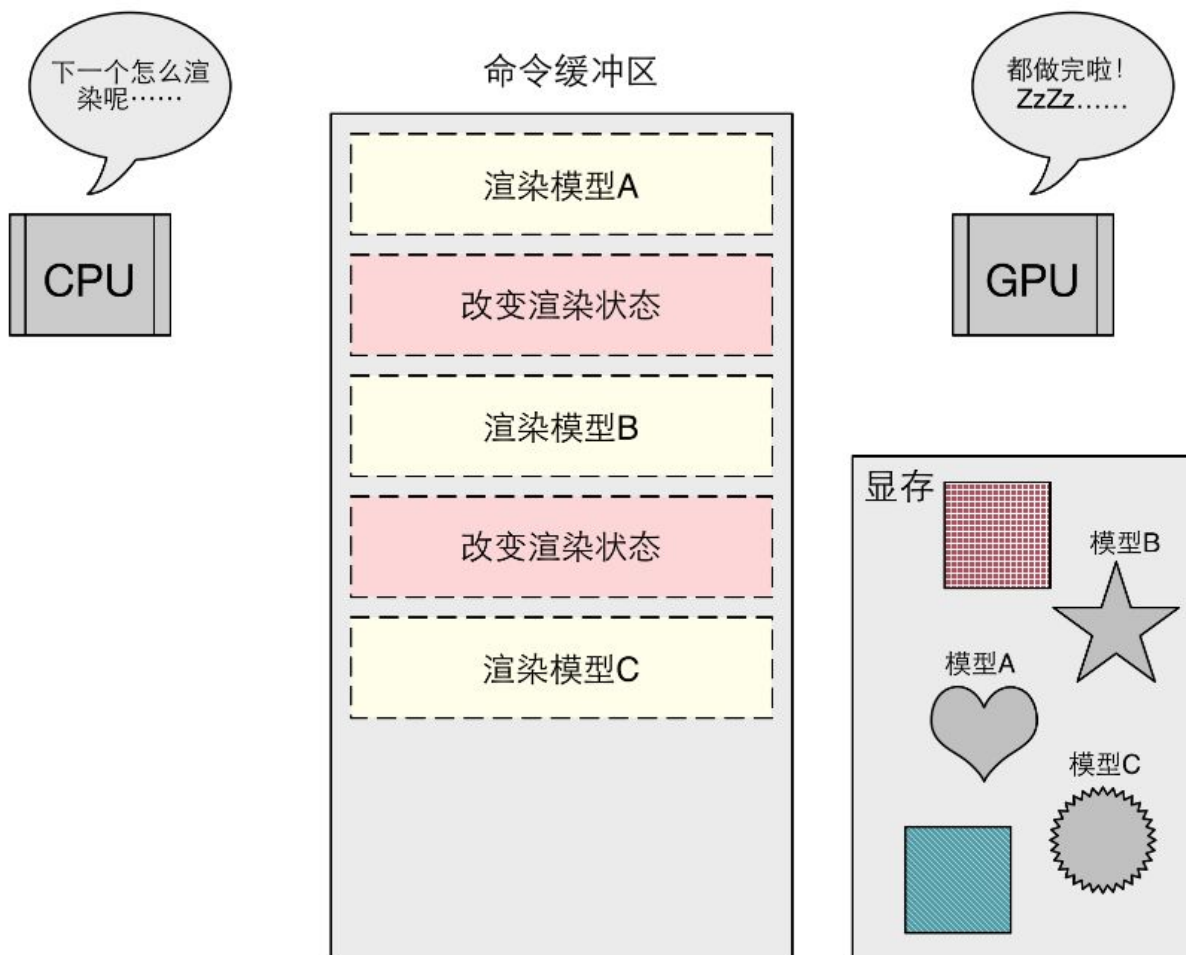
Draw Call

10 000 1KB

10MB

10MB

一个一个的Draw Call
 CPU GPU一个一个一个一个一个一个CPU一个一个一个一个
 一个一个CPU一个一个GPU一个一个GPU一个一个
 200 2 000一个一个一个一个一个一个CPU一个一个
 Draw Call一个一个CPU一个一个Draw Call一个一个CPU一个一个
 2.20一个一个



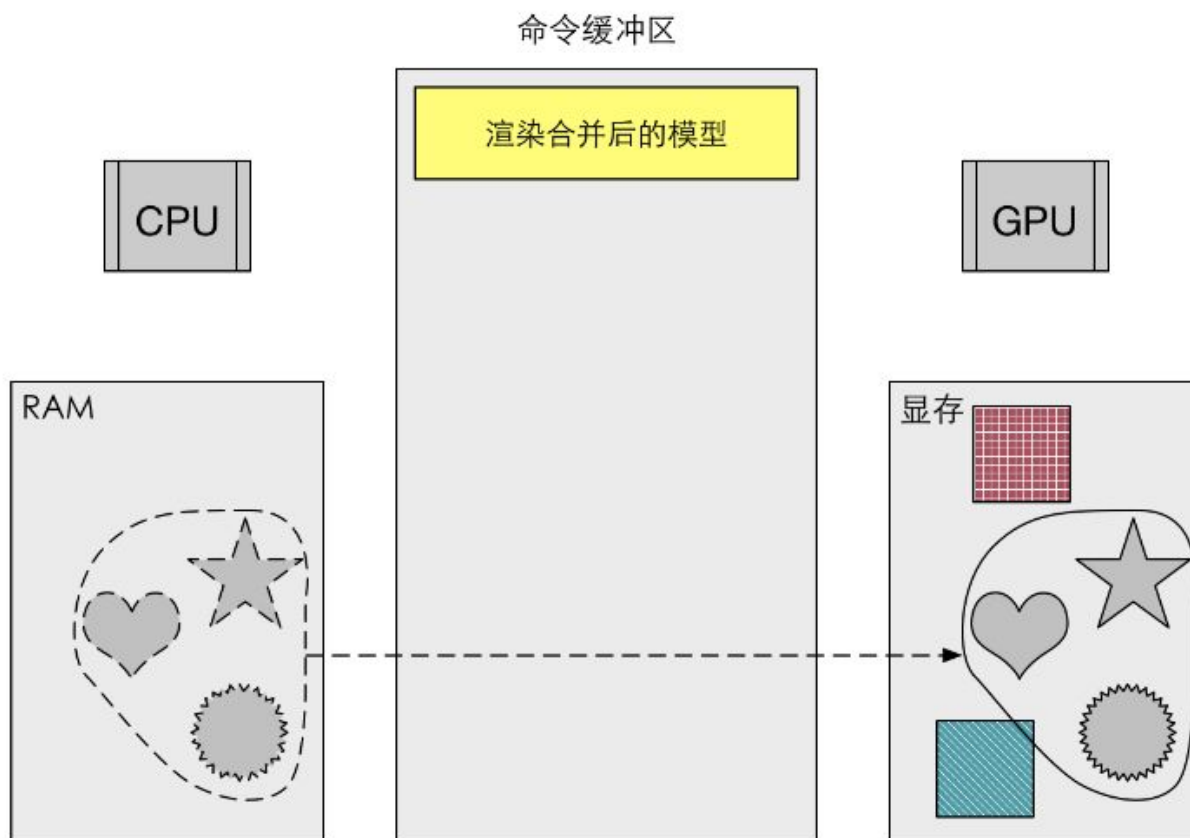
▲2.20 一个一个GPU一个一个一个一个一个一个 GPU一个一个CPU
 一个一个

一个一个**Draw Call**

一个一个Draw Call一个一个一个一个一个一个**Batching** 一个一个

Draw Call CPU CPU
 Draw Call Draw Call
 Draw Call 2.21

CPU
 GPU



▲ 2.21 CPU RAM GPU Draw Call

Draw Call

1

2

16.4 Unity

2.4.4

Fixed-Function Pipeline GPU

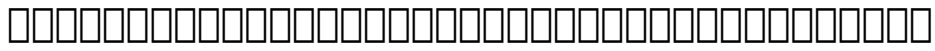
Function Entry Points GPU

GPU 2.1 3

2.1 3

3D API		
OpenGL	1.5	2.0
OpenGL ES	1.1	2.0
DirectX	7.0	8.0

GPU OpenGL 2.0 GPU OpenGL 3.0 OpenGL 3.2 Core Profile



2.5 Shader

GPU Shader

- GPU Shader
- Shader
- Shader

Shader

Unity Shader

2.6 渲染

渲染是图形API中最复杂的部分之一。在OpenGL中，渲染是通过调用`glDraw`系列函数来完成的。在DirectX中，渲染是通过调用`ID3D11DeviceContext::Draw`系列函数来完成的。在NVIDIA的CUDA中，渲染是通过调用`cudaDeviceSynchronize`来完成的。^[1] 在OpenGL中，渲染是通过调用`glDraw`系列函数来完成的。^[2] 在DirectX中，渲染是通过调用`ID3D11DeviceContext::Draw`系列函数来完成的。^[2] 在NVIDIA的CUDA中，渲染是通过调用`cudaDeviceSynchronize`来完成的。^[2] 在OpenGL中，渲染是通过调用`glDraw`系列函数来完成的。^[2] 在DirectX中，渲染是通过调用`ID3D11DeviceContext::Draw`系列函数来完成的。^[2] 在NVIDIA的CUDA中，渲染是通过调用`cudaDeviceSynchronize`来完成的。^[2] https://www.opengl.org/wiki/Rendering_Pipeline_Overview
[https://msdn.microsoft.com/en-us/library/windows/desktop/ff476882\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ff476882(v=vs.85).aspx) 渲染管道

[1] Akenine-Möller T, Haines E, Hoffman N. Real-time rendering[M]. CRC Press, 2008.

[2] Wloka M. Batch, Batch, Batch: What does it really mean?[C]//Presentation at game developers conference. 2003.

3 Unity Shader

Shader를 불러와서 GPU에 업로드하고, vertex shader와 fragment shader를 설정한다.

Unity에서 Shader를 불러오는 방법은 다음과 같다.

```
// Shader를 불러오는 함수
void Initialization() {
    // Vertex Shader를 불러옴
    string vertexShaderCode = LoadShaderFromFile(VertexShader.shader);
    // Fragment Shader를 불러옴
    string fragmentShaderCode = LoadShaderFromFile(FragmentShader.shader);
    // GPU에 Vertex Shader를 업로드
    LoadVertexShaderFromString(vertexShaderCode);
    // GPU에 Fragment Shader를 업로드
    LoadFragmentShaderFromString(fragmentShaderCode);

    // "vertexPosition" 속성을 설정
    SetVertexShaderProperty("vertexPosition", vertices);
    // "MainTex" 속성을 someTexture로 설정
    SetVertexShaderProperty("MainTex", someTexture);
    // "MVP" 속성을 MVP로 설정
    SetVertexShaderProperty("MVP", MVP);

    // Blend를 Disable
    Disable(Blend);
    // Z-Test를 Enable
    Enable(ZTest);
    SetZTestFunction(LessOrEqual);

    // ...
}

// OnRendering 함수
void OnRendering() {
    // DrawCall
    DrawCall();
    // ...
}
```

VertexShader.shader

```
// 顶点着色器MVP矩阵
in float3 vertexPosition;
in sampler2D MainTex;
in Matrix4x4 MVP;

// 输出MVP后的位置
out float4 position;

void main() {
    // 计算MVP后的位置
    position = MVP * vertexPosition;
}
```

FragmentShader.shader

```
// 接收VertexShader输出的position并计算颜色
in float4 position;

// 输出颜色
out float4 fragColor;

void main() {
    // 设置颜色
    fragColor = float4(1.0, 1.0, 1.0, 1.0);
}
```

ShaderLab语言是Unity Shader的配置文件，它定义了Shader的编译选项和渲染管线。ShaderLab语言的基本语法如下：

Unity Shader的配置文件通常以ShaderLab语言编写，并保存在Assets/Shader目录下。Unity Shader的配置文件通常以ShaderLab语言编写，并保存在Assets/Shader目录下。Unity Shader的配置文件通常以ShaderLab语言编写，并保存在Assets/Shader目录下。

3.1 Unity Shader

Unity Shader

3.1.1 Unity Shader

Unity **Material** Unity Shader

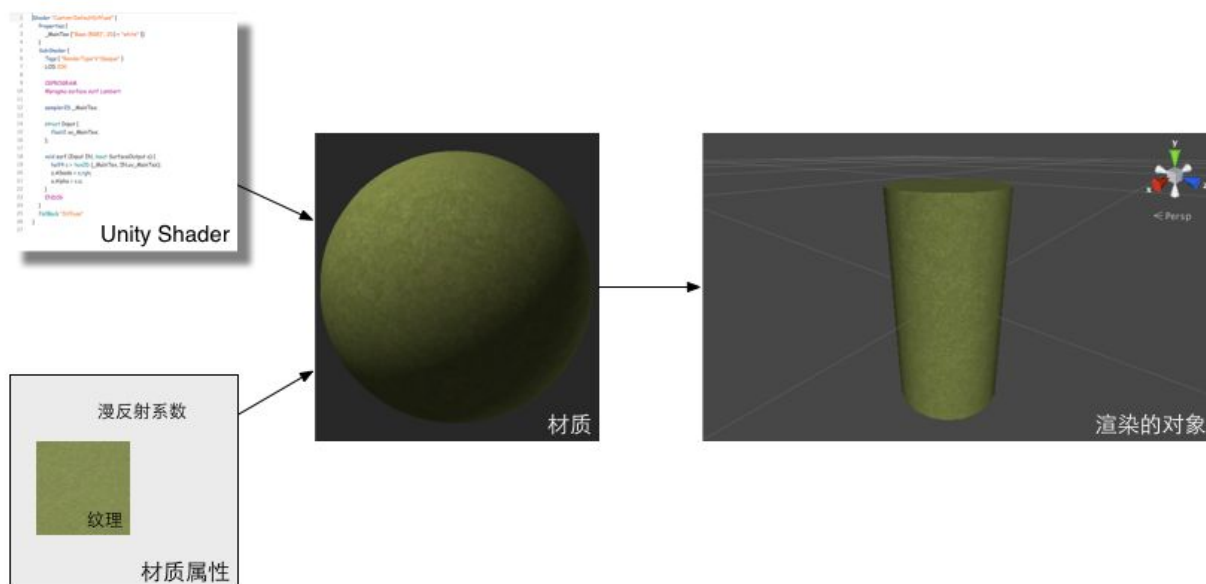
1

2 Unity Shader

3

4 Unity Shader

3.1 Unity Shader



▲ 3.1 Unity Shader

Unity Shader은 Unity에서 사용하는 그래픽 렌더링을 위한 프로그램 언어입니다. Unity Shader은 Unity에서 사용하는 그래픽 렌더링을 위한 프로그램 언어입니다.

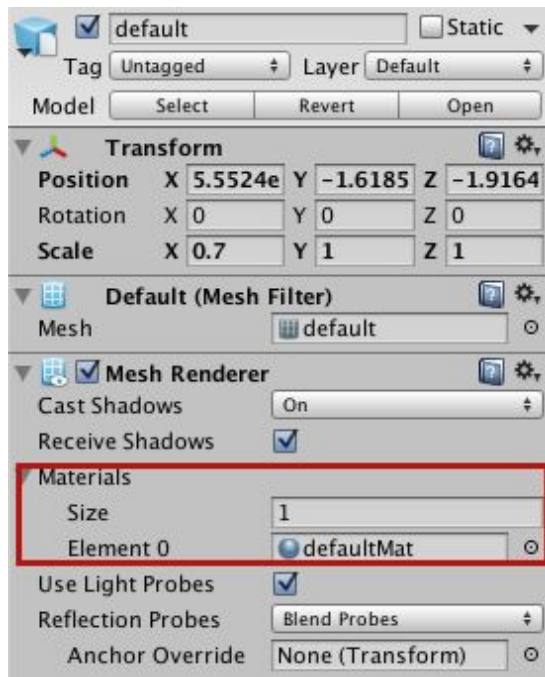
3.1.2 Unity Shader

Unity Shader은 GameObject Mesh Particle Systems Unity Shader Shader

Unity Shader Assets -> Create -> Material Project -> Create -> Material Scene Mesh Renderer 3.2

Unity 5.x Standard Shader 18

Unity Cinema 4D Maya 3.3



▲ 3.2 Mesh Renderer



▲3.3 UnityのShaderプロパティ



「1」UnityのShaderプロパティ「2」Unity ShaderのShaderプロパティ

3.1.3 UnityのShader

UnityのShaderプロパティUnityのShaderプロパティUnity Shader Unity ShaderのShaderプロパティ

3.6.2 数据流图

1. Unity Shader Assets -> Create -> Shader Project -> Create -> Shader Unity 5.2 Unity 4 Unity Shader
 Standard Surface Shader Unlit Shader Image Effect Shader Compute Shader Standard Surface Shader
 Unity 5 18 Unlit Shader / Image Effect Shader 12 Compute Shader Shader Shader GPU Unity
Compute Shader
<http://docs.unity3d.com/Manual/ComputeShaders.html>
 Standard Surface Shader
 Unity / Unlit Shader

Unity Shader

Unity Shader

Unity Shader

Unity Shader

[illegible]

Default Maps

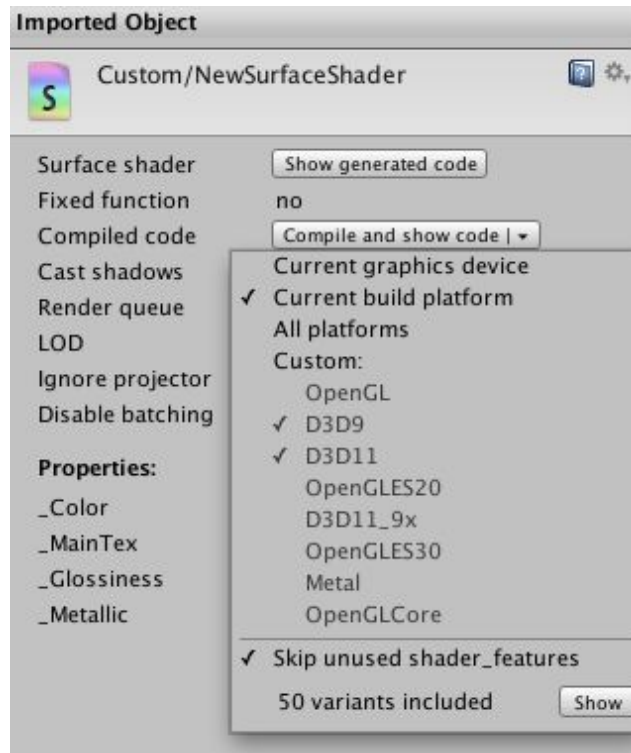
Unity Shader
Unity Shader
Unity Shader
Surface Shader
Fixed Function Shader
Unity Shader 3.3.3
LOD

3.4.1 *Show generated code*
Unity
Unity Shader

Unity ShaderのInspectorでFixed function をShow generated code からCompile and show code に変更し、Unity ShaderのInspectorでOpenGL D3D9 D3D11のShaderを3.5に設定する。



▲3.4 Unity Shader



▲ 3.5 Compile and show code

Unity Shader Render
queue Disable batching Properties

3.2 Unity Shader与ShaderLab

“ShaderLab是Unity Shader的配置文件”

——Nicholas Francis

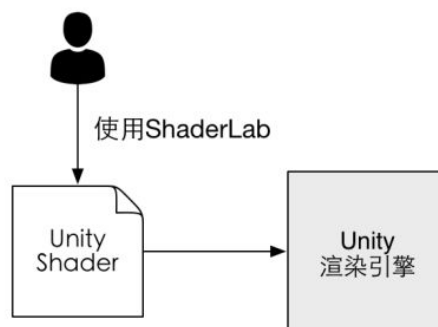
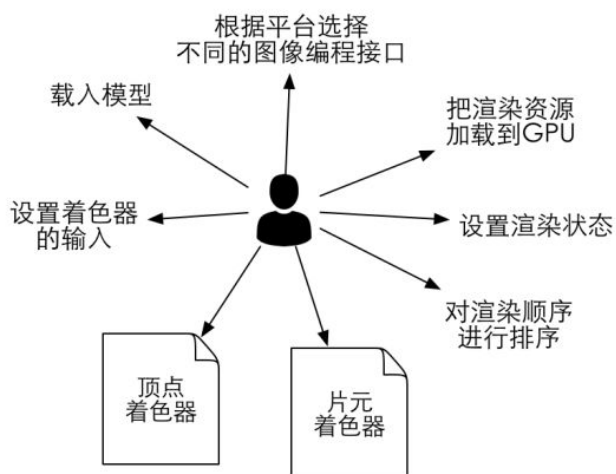
ShaderLab是Unity Shader的配置文件，它定义了Shader的属性，如颜色、纹理、光照等。它通常与Shader一起使用，用于控制Shader的渲染效果。

Unity Shader与ShaderLab是Unity引擎的重要组成部分。Unity Shader用于定义物体的渲染效果，而ShaderLab则用于配置Shader的属性。ShaderLab的配置文件通常以“.shader”为后缀。

ShaderLab

"ShaderLab is a friend you can afford."——Nicholas Francis
Nicholas Francis是Unity的联合创始人兼COO。

Unity Shader与Unity引擎的版本3.6版本开始支持ShaderLab。Unity ShaderLab是Unity ShaderLab的配置文件。



▲ 3.6 Unity Shader의 기본 구조와 ShaderLab의 기본 구조

Unity의 Shader는 ShaderLab에서 정의된다. ShaderLab은 Unity의 Shader를 정의하는 언어이다. ShaderLab의 기본 구조는 다음과 같다.

syntax Unity Shader의 기본 구조는 다음과 같다.

*Properties*는 Shader의 속성을 정의하는 부분이다. ShaderLab은 CgFX Direct3D Effects의 FX를 기반으로 작성된 언어이다.

Unity Shader의 기본 구조

```
Shader "ShaderName" {
    Properties {
        // ...
    }
    SubShader {
        // ...
    }
    SubShader {
        // ...
    }
    Fallback "VertexLit"
}
```

Unity의 Shader는 ShaderLab에서 정의된다. ShaderLab은 Unity의 Shader를 정의하는 언어이다.

3.3 Unity Shader

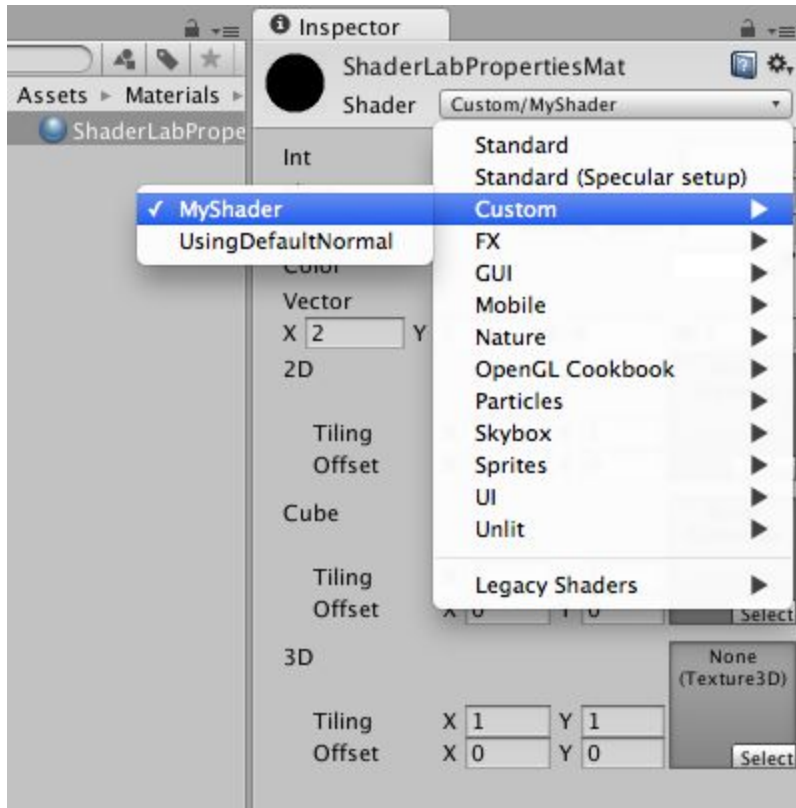
ShaderLab *Properties* *SubShader* *Fallback* Unity Shader Unity
Unity Shader

3.3.1 Shader

Unity Shader *Shader* Unity Shader
"MyShader" Unity
Shader "/"
Unity Shader

```
Shader "Custom/MyShader" { }
```

Unity Shader *Shader -> Custom -> MyShader* 3.7



▲ 3.7 Unity Shaderのインストールと設定

3.3.2 Unity ShaderのProperties

Properties は、Shader の **property** を定義するためのブロックです。

Properties は、Shader の

```
Properties {
    Name ("display name", PropertyType) = DefaultValue
    Name ("display name", PropertyType) = DefaultValue
    // ...
}
```

Shader の **Name** は、Unity の Shader の **display name** を指定するためのプロパティです。

1 **PropertyType** 3.1
 2 Unity Shader

3.1 Properties

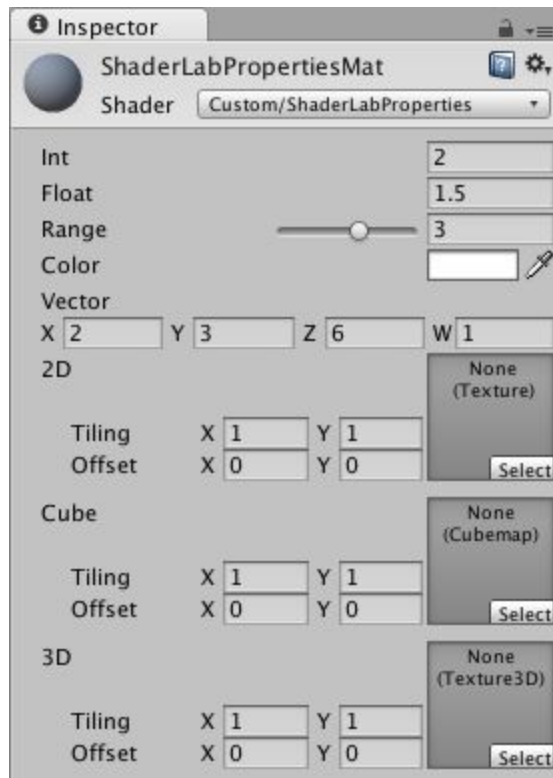
PropertyType	PropertyType	PropertyType
Int	number	_Int ("Int", Int) = 2
Float	number	_Float ("Float", Float) = 1.5
Range(min, max)	number	_Range("Range", Range(0.0, 5.0)) = 3.0
Color	(number,number,number,number)	_Color ("Color", Color) = (1,1,1,1)
Vector	(number,number,number,number)	_Vector ("Vector", Vector) = (2, 3, 6, 1)
2D	"defaulttexture" {}	_2D ("2D", 2D) = "" {}
Cube	"defaulttexture" {}	_Cube ("Cube", Cube) = "white" {}
3D	"defaulttexture" {}	_3D ("3D", 3D) = "black" {}

1 **Int** **Float** **Range**
 2 **Color** **Vector**
 3 **2D** **Cube**
 4 **3D**

このチュートリアルでは、ShaderLabの“white”“black”“gray”と“bump”の
テクスチャ生成方法を紹介します。Unity 5.0では、
TexGen CubeReflect と **TexGen CubeNormal** の2つの
テクスチャ生成方法が追加されました。Unity 5.0
では、これらの方法をShaderLabで使用する
ことができます。

このチュートリアルでは、

```
Shader "Custom/ShaderLabProperties" {  
    Properties {  
        // Numbers and Sliders  
        _Int ("Int", Int) = 2  
        _Float ("Float", Float) = 1.5  
        _Range ("Range", Range(0.0, 5.0)) = 3.0  
        // Colors and Vectors  
        _Color ("Color", Color) = (1,1,1,1)  
        _Vector ("Vector", Vector) = (2, 3, 6, 1)  
        // Textures  
        _2D ("2D", 2D) = "" {}  
        _Cube ("Cube", Cube) = "white" {}  
        _3D ("3D", 3D) = "black" {}  
    }  
  
    FallBack "Diffuse"  
}
```



▲3.8 ShaderLabPropertiesMat

3.8 ShaderLabPropertiesMat

ShaderLabPropertiesMatは、UnityのShaderLabのCustom Shader GUIの例として、Assets -> Materials -> Chapter3 -> RedifyMatに配置されています。
 Custom Shader GUIの例として、<http://docs.unity3d.com/Manual/SL-CustomShaderGUI.html>を参照してください。

ShaderLabPropertiesMatは、CgのShaderのPropertiesを定義するためのCustom Shader GUIの例として、Assets -> Materials -> Chapter3 -> RedifyMatに配置されています。

3.3.3 SubShader

Unity Shader 的 *SubShader* 可以包含一个或多个 *Unity Shader* 的 *SubShader*。如果 *SubShader* 包含一个 *Unity Shader* 的 *Fallback*，那么 *Unity Shader* 将使用 *Fallback*。

Unity 的 *SubShader* 可以包含一个或多个 *Pass*。如果 *SubShader* 包含一个 *Pass*，那么 *SubShader* 将使用 *Pass*。如果 *SubShader* 包含多个 *Pass*，那么 *SubShader* 将使用第一个 *Pass*。

SubShader 的语法如下：

```
SubShader {  
    // 名称  
    [Tags]  
  
    // 渲染设置  
    [RenderSetup]  
  
    Pass {  
    }  
    // Other Passes  
}
```

SubShader 的 *Pass* 可以包含一个或多个 *RenderSetup*。如果 *SubShader* 包含一个 *Pass*，那么 *SubShader* 将使用 *Pass*。如果 *SubShader* 包含多个 *Pass*，那么 *SubShader* 将使用第一个 *Pass*。如果 *SubShader* 包含一个 *Pass*，那么 *SubShader* 将使用 *Pass*。如果 *SubShader* 包含多个 *Pass*，那么 *SubShader* 将使用第一个 *Pass*。

● 名称

ShaderLab 的 *SubShader* 可以包含一个或多个 *Pass*。如果 *SubShader* 包含一个 *Pass*，那么 *SubShader* 将使用 *Pass*。如果 *SubShader* 包含多个 *Pass*，那么 *SubShader* 将使用第一个 *Pass*。

3.2 名称

プロパティ	デフォルト値	説明
Cull	Cull Back ☐ Front ☐ Off	背面/前面/オフのいずれかを指定
ZTest	ZTest Less Greater ☐ LEqual ☐ GEqual ☐ Equal ☐ NotEqual ☐ Always	比較モードを指定
ZWrite	ZWrite On ☐ Off	Z書き込みを指定
Blend	Blend SrcFactor DstFactor	ブレンドモードを指定

SubShader は、複数の Pass を含む。Pass は、シェディングの各ステップを定義する。Pass は、Material の Shader を定義する。Pass は、Shader の各ステップを定義する。Pass は、Shader の各ステップを定義する。

● SubShader のタグ

SubShader は、Tags プロパティを持つ。Key/Value Pair の形式で、SubShader を識別する。Unity は、Tags を使用して、SubShader を識別する。

例:

```
Tags { "TagName1" = "Value1" "TagName2" = "Value2" }
```

SubShader は、Unity 3.3 からサポートされている。

3.3 SubShader のタグ

プロパティ	デフォルト値	説明

属性名	説明	デフォルト値
Queue	描画キューに属する。透明オブジェクトは「Transparent」キュー、そうでないものは「Default」キューに属する。8つのキューがあり、各キューは異なるレンダリングパスを持つ。	Tags { "Queue" = "Transparent" }
RenderType	レンダリングタイプ。透明オブジェクトは「Transparent」、そうでないものは「Opaque」。 Shader Replacement	Tags { "RenderType" = "Opaque" }
DisableBatching	SubShader が Unity 11.3 以降のバージョンで、SubShader が	Tags { "DisableBatching" = "True" }
ForceNoShadowCasting	SubShader が 8.4 以降のバージョンで、	Tags { "ForceNoShadowCasting" = "True" }
IgnoreProjector	“True” の場合、SubShader が Projector を無視する。	Tags { "IgnoreProjector" = "True" }
CanUseSpriteAtlas	SubShader が sprites を使用しない場合、 “False”	Tags { "CanUseSpriteAtlas" = "False" }

PreviewType	PreviewType	Tags
PreviewType	<pre> ShaderLab { "Plane" "SkyBox" } </pre>	<pre> Tags { "PreviewType" = "Plane" } </pre>

ShaderLab

ShaderLab SubShader Pass Pass
 ShaderLab SubShader

● Pass

Pass

```

Pass {
    [Name]
    [Tags]
    [RenderSetup]
    // Other code
}

```

Pass Pass

Name "MyPassName"

ShaderLab UsePass Unity
 Shader Pass

UsePass "MyShader/MYPASSNAME"

Unity Pass 使用 Pass 使用 Pass 使用 Pass

Pass SubShader Pass 3.4.3

Pass SubShader 3.4 Pass

3.4 Pass

Property	Value	Tags
LightMode	Pass Unity	Tags { "LightMode" = "ForwardBase" }
RequireOptions	Pass Unity SoftVegetation	Tags { "RequireOptions" = "SoftVegetation" }

Pass Unity Shader Pass

- **UsePass** Unity Shader Pass
- **GrabPass** Pass 10.2.2

3.3.4 Fallback

3.4 Unity Shader

Unity Shader ShaderLab Unity Shader SubShader Pass

Unity 3 Unity Shader Shader ShaderLab

```
Shader "MyShader" {
    Properties {
        //
    }
    SubShader {
        // Shader
        // Surface Shader
        // Vertex/Fragment Shader
        // Fixed Function Shader
    }
    SubShader {
        // SubShader
    }
}
```

3.4.1 Unity

Surface Shader Unity Unity Unity Unity Unity Unity

```
Shader "Custom/Simple Surface Shader" {
    SubShader {
        Tags { "RenderType" = "Opaque" }
```



```

        fixed4 frag() : SV_Target {
            return fixed4(1.0,0.0,0.0,1.0);
        }

    ENDCG
}
}
}

```

シェーダーファイル/パスを指定する `CGPROGRAM` `ENDCG` の
 シェーダーファイル/パスを指定する `Pass` シェーダー `SubShader` シェーダー
 シェーダー `Pass` シェーダー `Shader` シェーダー `CGPROGRAM` `ENDCG` シェー
 シェーダー `Cg/HLSL` シェーダー

3.4.3 シェーダーの作成

シェーダー `Unity Shader` シェーダー `GPU` シェー
`DirectX 7.0` `OpenGL 1.5` `OpenGL ES 1.1` シェーダー `iPhone 3` シェー
 シェーダー `Fixed Function Shader` シェー

シェーダー `Shader` シェー

```

Shader "Tutorial/Basic" {
    Properties {
        _Color ("Main Color", Color) = (1,0.5,0.5,1)
    }
    SubShader {
        Pass {
            Material {
                Diffuse [_Color]
            }
            Lighting On
        }
    }
}

```

Pass Pass Pass
3.3.3

ShaderLab ShaderLab
Cg/HLSL

GPU
Unity 5.2 Unity /

3.4.4 Unity Shader

Unity Shader

- /
-
- /
- /

3.5 如何编写Unity Shader

如何编写Unity Shader，在Unity Shader中，我们使用“Shader”来定义一个Shader，

在Unity Shader中，我们使用“Shader”来定义一个Shader，
在Unity Shader中，我们使用“Shader”来定义一个Shader，17

3.6 ☐ ☐ ☐ ☐

[illegible]

3.6.1 Unity Shader != GLSL Shader

00000000Unity Shader0000002000000Shader000Unity
 Shader000000Unity000000Unity00Unity Shader0000000000
 ShaderLab00——0000**.shader** 00000000000000

```

    Unity Shader
    ShaderLab
    Shader

```

- Shader Shader Unity Shader
- Shader Unity Shader
- Shader Unity Shader Unity Shader

Unity Shader Shader Shader Tessellation Shader Geometry Shader Unity Unity 4.x DirectX 11 OpenGL Shader Unity Shader

[illegible]

3.6.2 Unity Shader Cg/HLSL

Unity Shader ShaderLab ShaderLab Cg/HLSL CGPROGRAM ENDCG Cg DX9 HLSL Unity Cg HLSL Cg/HLSL ShaderLab

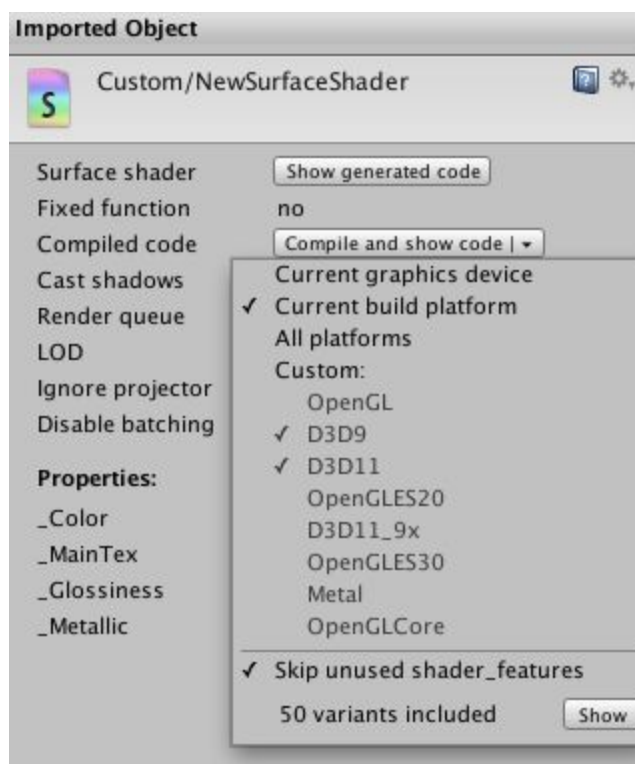
Cg Pass

```
Pass {  
    // Pass  
  
    CGPROGRAM  
    //  
    #pragma vertex vert  
    #pragma fragment frag  
  
    // Cg  
  
    ENDCG  
    //  
}
```

“Cg/HLSL SubShader Pass” Cg/HLSL SubShader Unity/ ShaderLab Unity Pass/ Unity Shader Show generated code Unity Shader Unity 5.2 Unity

Unity Cg Unity Cg OpenGL Direct3D 11 OpenGL ES Unity Cg

3.1.3 Unity Shaderのデバッグ
Debug 3.9



▲3.9 Unity Shaderのデバッグ
Compile and show code UnityのCG
Compile and show code のデバッグ
のデバッグ

のデバッグ
Mac OS X DirectXのデバッグ

3.6.3 GLSL

のデバッグ
“Cg/HLSLのデバッグGLSL”
Mac OS X OpenGL ES 2.0 Linux PC
Xbox 360 DirectXのデバッグ

のデバッグ
GLSL Unity ShaderのデバッグCg/HLSL
CGPROGRAM ENDCG のデバッグGLSL
GLSLPROGRAM ENDGLSL

Unity Shader GLSL Unity GLSL
Shader Programs <http://docs.unity3d.com/Manual/SL-GLSLShaderPrograms.html>

3.7 参考

UnityのUnity Shaderの参考としてUnityの
ドキュメント, 及びUnity Shaderの
<http://docs.unity3d.com/Manual/SL-Reference.html> 及び
Unityの
<http://docs.unity3d.com/Manual/ShaderTut1.html> 及び
<http://docs.unity3d.com/Manual/ShaderTut2.html> 及びUnity
ShaderのCgのNVIDIAのCg
<http://http.developer.nvidia.com/Cg/> 及びNVIDIAの
http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html 及びCgの

4 Shader

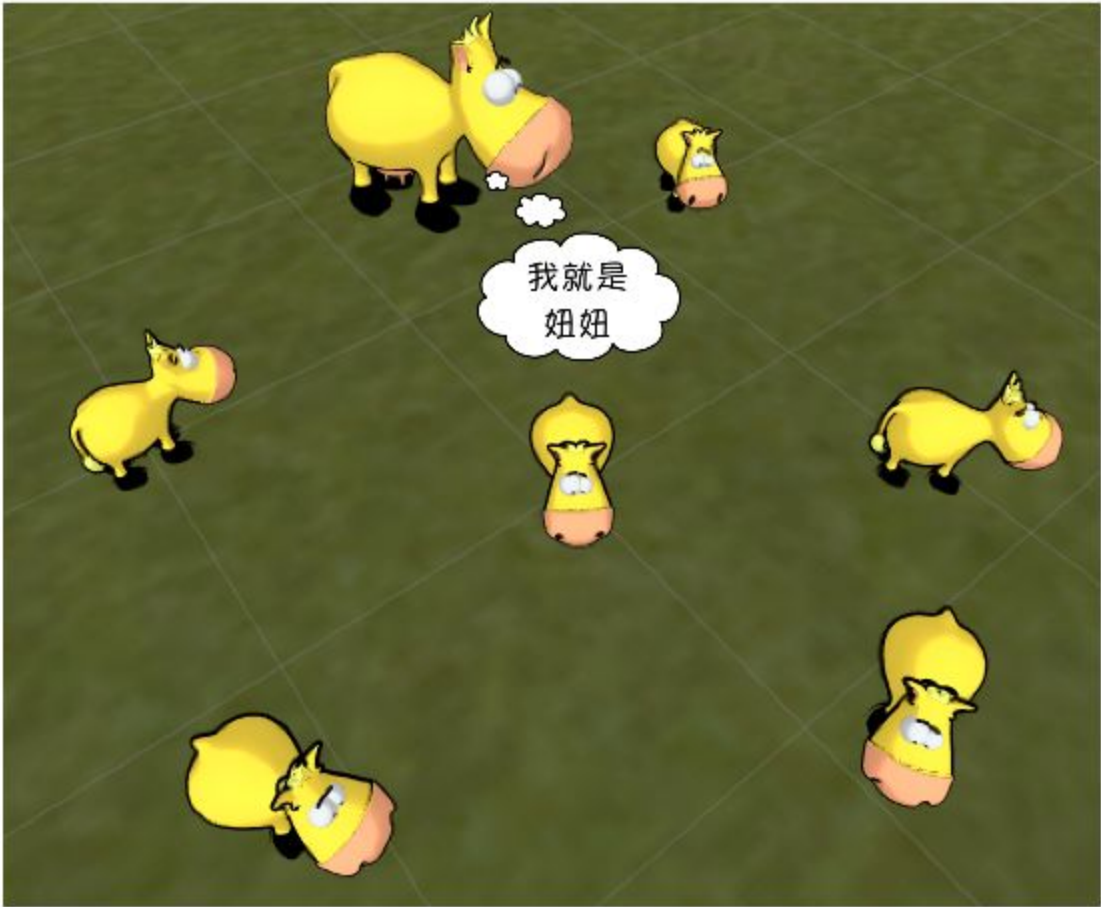
——

Shader Shader
——

Shader “”

4.1 自我介绍

小朋友们，你们好！我是来自幼儿园的张老师。今天我要给大家讲一个关于自我介绍的故事。故事的名字叫《我就是妞妞》。故事的主人公是一个叫妞妞的小黄鸭。妞妞是一个非常有礼貌、非常爱干净、非常爱学习的小黄鸭。妞妞的故事告诉我们，每个人都要学会自我介绍，学会与人交往，学会尊重他人。故事的名字叫《我就是妞妞》。故事的主人公是一个叫妞妞的小黄鸭。妞妞是一个非常有礼貌、非常爱干净、非常爱学习的小黄鸭。妞妞的故事告诉我们，每个人都要学会自我介绍，学会与人交往，学会尊重他人。



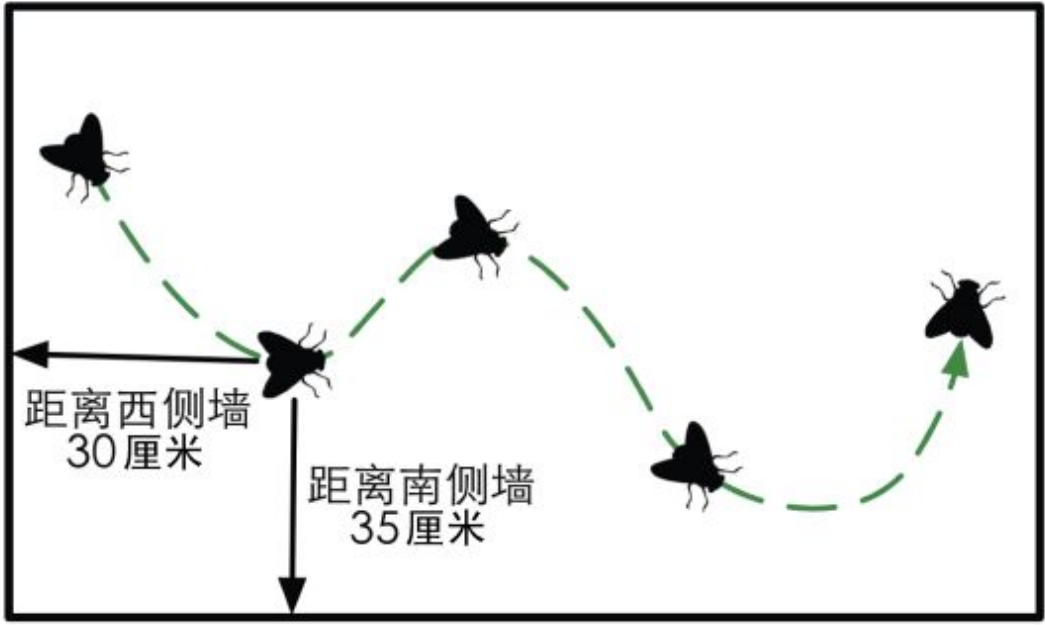
▲4.1 自我介绍

小朋友们，你们好！我是来自幼儿园的张老师。今天我要给大家讲一个关于自我介绍的故事。故事的名字叫《我就是妞妞》。故事的主人公是一个叫妞妞的小黄鸭。妞妞是一个非常有礼貌、非常爱干净、非常爱学习的小黄鸭。妞妞的故事告诉我们，每个人都要学会自我介绍，学会与人交往，学会尊重他人。故事的名字叫《我就是妞妞》。故事的主人公是一个叫妞妞的小黄鸭。妞妞是一个非常有礼貌、非常爱干净、非常爱学习的小黄鸭。妞妞的故事告诉我们，每个人都要学会自我介绍，学会与人交往，学会尊重他人。

4.2 坐标系

在上一节中，我们介绍了如何描述一个物体在二维平面上的位置。今天，我们将介绍一个更通用的方法，即**Cartesian Coordinate System**（笛卡尔坐标系）。这个坐标系是由法国数学家René Descartes提出的。

在笛卡尔坐标系中，我们使用两个互相垂直的轴来描述一个点的位置。通常，我们使用x轴和y轴。x轴表示水平方向，y轴表示垂直方向。一个点的坐标可以用一个有序对(x, y)来表示。例如，点(3, 4)表示在x轴上距离原点3个单位，在y轴上距离原点4个单位的点。在本节中，我们将使用这个坐标系来描述蚂蚁在蚁巢中的位置。

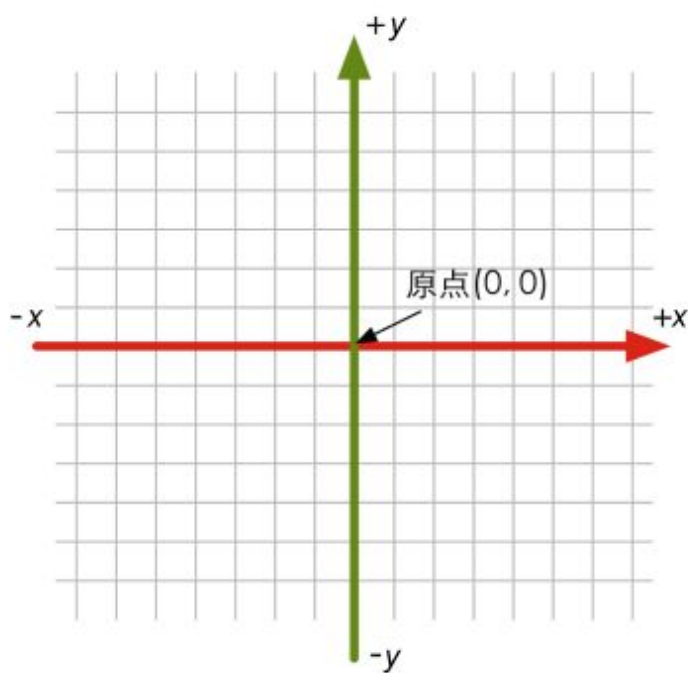


▲图4.2 蚂蚁在蚁巢中的位置。图中展示了蚂蚁在蚁巢中的移动路径。坐标系的引入使得我们可以精确地描述蚂蚁的位置。图中蚂蚁的位置可以用坐标(x, y)来表示。例如，图中蚂蚁的位置可以用(30, 35)来表示，其中30表示距离西侧墙的距离，35表示距离南侧墙的距离。

4.2.1 坐标系

在计算机图形学中，坐标系是描述图形位置的基础。通常，我们使用笛卡尔坐标系来定义图形在屏幕上的位置。在2D图形中，我们通常使用x和y轴来定义一个点的位置。在3D图形中，我们通常使用x、y和z轴来定义一个点的位置。

4.3 坐标系



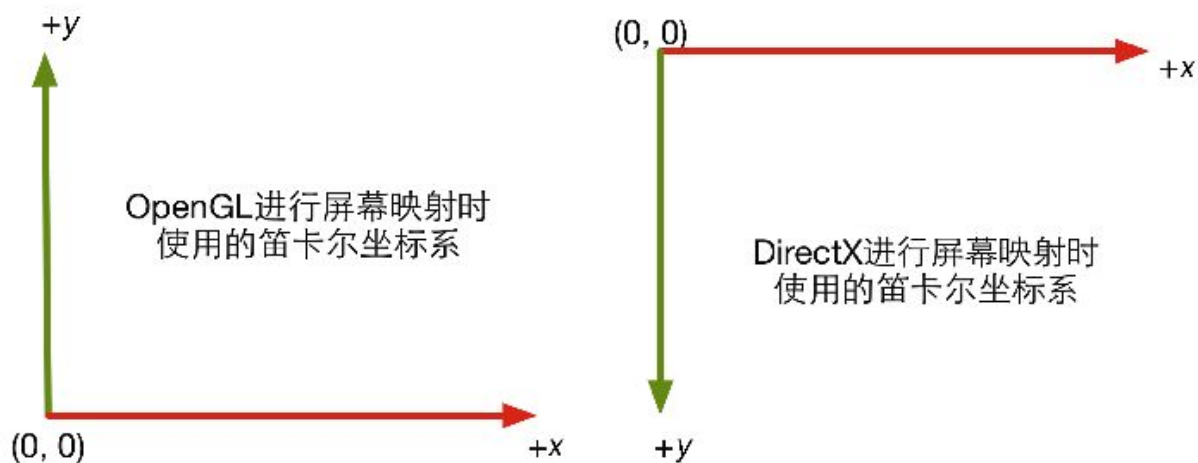
▲4.3 坐标系

在计算机图形学中，坐标系是描述图形位置的基础。通常，我们使用笛卡尔坐标系来定义图形在屏幕上的位置。在2D图形中，我们通常使用x和y轴来定义一个点的位置。在3D图形中，我们通常使用x、y和z轴来定义一个点的位置。

- 在2D图形中，我们通常使用x和y轴来定义一个点的位置。
- 在3D图形中，我们通常使用x、y和z轴来定义一个点的位置。

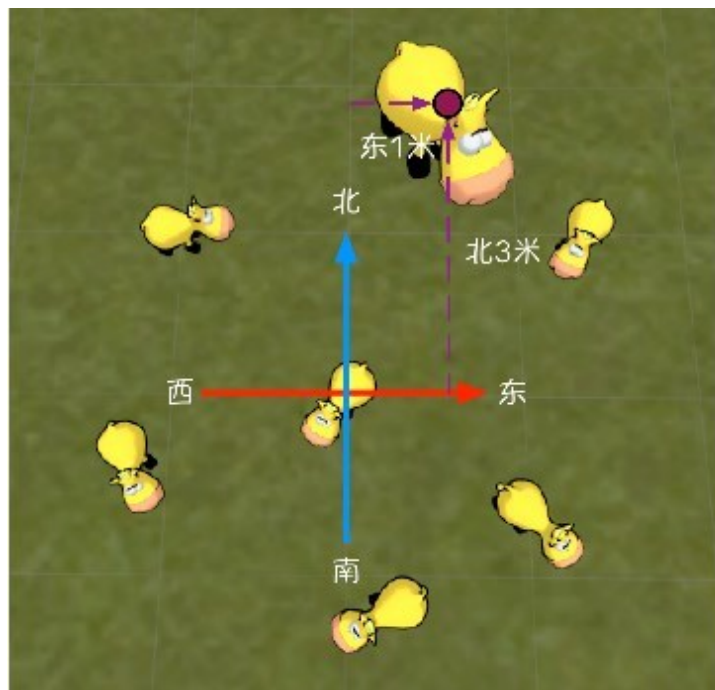
在4.3节中，我们介绍了x和y轴的定义。在3D图形中，我们通常使用x、y和z轴来定义一个点的位置。在2.3.4节中，我们介绍了OpenGL和DirectX的坐标系。在4.4节中，我们将讨论如何将这些坐标系应用到图形渲染中。

在4.4节中，我们将讨论如何将这些坐标系应用到图形渲染中。通常，我们使用1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 622, 623, 624, 625, 626, 627, 628, 629, 630, 631, 632, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 643, 644, 645, 646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 659, 660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 675, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696, 697, 698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709, 710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720, 721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795, 796, 797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 810, 811, 812, 813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 830, 831, 832, 833, 834, 835, 836, 837, 838, 839, 840, 841, 842, 843, 844, 845, 846, 847, 848, 849, 850, 851, 852, 853, 854, 855, 856, 857, 858, 859, 860, 861, 862, 863, 864, 865, 866, 867, 868, 869, 870, 871, 872, 873, 874, 875, 876, 877, 878, 879, 880, 881, 882, 883, 884, 885, 886, 887, 888, 889, 890, 891, 892, 893, 894, 895, 896, 897, 898, 899, 900, 901, 902, 903, 904, 905, 906, 907, 908, 909, 910, 911, 912, 913, 914, 915, 916, 917, 918, 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, 929, 930, 931, 932, 933, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 968, 969, 970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000.



▲4.4 比较OpenGL与DirectX坐标系

在OpenGL和DirectX中，坐标系的定义是不同的。在OpenGL中，坐标系的定义是：原点在左下角，x轴向右，y轴向上。而在DirectX中，坐标系的定义是：原点在左上角，x轴向右，y轴向下。这种差异导致了在屏幕映射时，OpenGL和DirectX的坐标系需要进行转换。



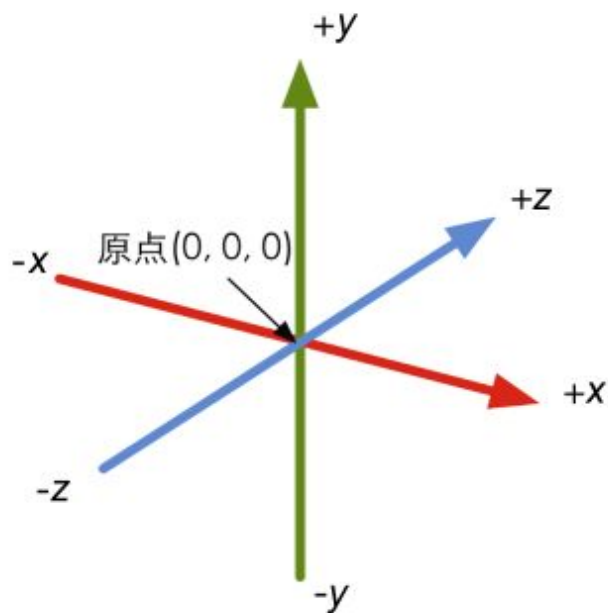
▲4.5 比较OpenGL与DirectX坐标系

4.2.2 基底

基底是线性代数中非常重要的概念，它描述了向量空间的一组线性无关的向量。在3D空间中，基底通常由三个相互正交的向量组成，它们可以表示空间中的任意向量。基底的选择通常基于方便性和物理意义，例如在物理学中，常用的基底是笛卡尔坐标系中的单位向量。

在3D空间中，基底的选择通常基于方便性和物理意义。例如，在物理学中，常用的基底是笛卡尔坐标系中的单位向量。这些单位向量通常表示为 $\hat{x}$, $\hat{y}$, 和 $\hat{z}$ ，它们分别指向x, y, 和z轴的正方向。这些单位向量的模长为1，且彼此正交。

图4.6展示了3D坐标系中的基底向量。图中显示了三个相互正交的向量，分别指向x, y, 和z轴的正方向。这些向量构成了3D空间的一个基底。



▲图4.6 3D坐标系中的基底

3D空间中的基底通常由三个相互正交的向量组成，它们可以表示空间中的任意向量。这组基底被称为**basis vector**。如果这组基底向量的模长为1，且彼此正交，则它们被称为**orthonormal basis**。如果这组基底向量彼此正交，但不一定模长为1，则它们被称为**orthogonal basis**。

在3D空间中，基底的选择通常基于方便性和物理意义。

图4.6展示了3D坐标系中的基底向量。图中显示了三个相互正交的向量，分别指向x, y, 和z轴的正方向。这些向量构成了3D空间的一个基底。

Figure 4.6 illustrates the difference between a left-handed coordinate space and a right-handed coordinate space. In a left-handed coordinate space, the z -axis points in the same direction as the x and y axes. In a right-handed coordinate space, the z -axis points in the opposite direction of the x and y axes.

4.2.3 Coordinate Systems

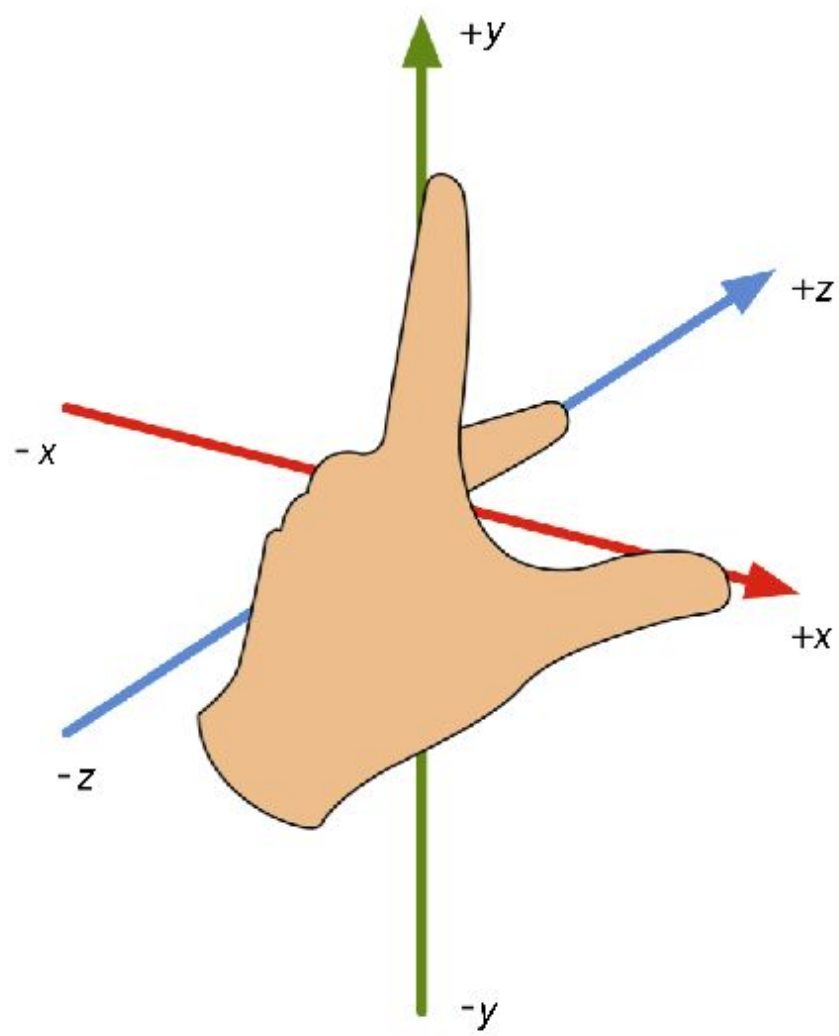
Figure 4.4 shows the coordinate systems used in OpenGL and DirectX. In OpenGL, the z -axis points in the negative z direction. In DirectX, the z -axis points in the positive z direction. The x and y axes are the same in both systems.

Figure 4.6 shows the difference between a left-handed coordinate space and a right-handed coordinate space. In a left-handed coordinate space, the z -axis points in the same direction as the x and y axes. In a right-handed coordinate space, the z -axis points in the opposite direction of the x and y axes.

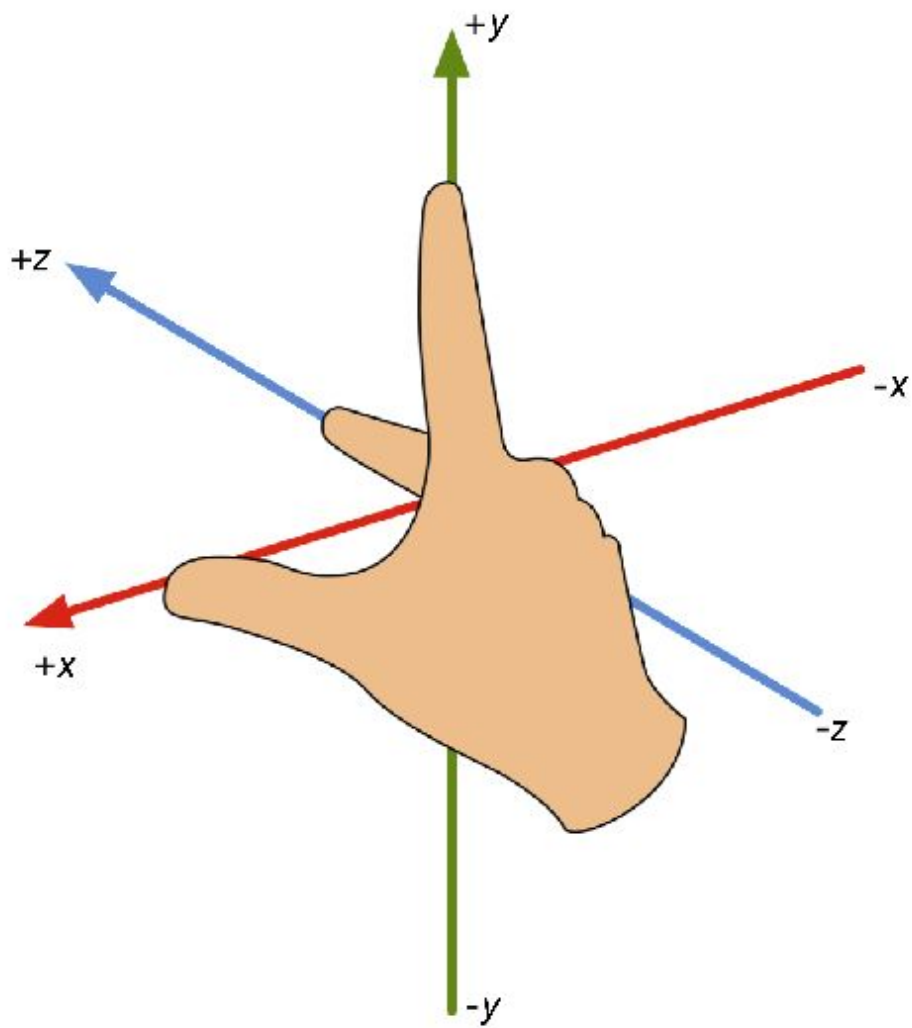
Figure 4.7 shows the difference between a left-handed coordinate space and a right-handed coordinate space. In a left-handed coordinate space, the z -axis points in the same direction as the x and y axes. In a right-handed coordinate space, the z -axis points in the opposite direction of the x and y axes.

Figure 4.8 shows the difference between a left-handed coordinate space and a right-handed coordinate space. In a left-handed coordinate space, the z -axis points in the same direction as the x and y axes. In a right-handed coordinate space, the z -axis points in the opposite direction of the x and y axes.

Figure 4.8 shows the difference between a left-handed coordinate space and a right-handed coordinate space. In a left-handed coordinate space, the z -axis points in the same direction as the x and y axes. In a right-handed coordinate space, the z -axis points in the opposite direction of the x and y axes.



▲ 4.7



▲图4.8 右手定则

在图4.8中，右手拇指指向+y轴，食指指向+z轴，中指指向+x轴，此时四指弯曲的方向即为角速度的方向。

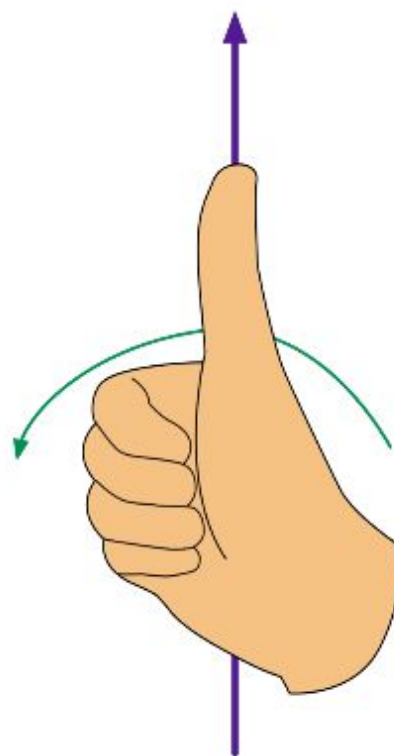
在图4.9中，右手拇指指向+forward轴，食指指向+right轴，中指指向+up轴，此时四指弯曲的方向即为角速度的方向。

在图4.10中，右手拇指指向+forward轴，食指指向+right轴，中指指向+up轴，此时四指弯曲的方向即为角速度的方向。

“**4.10**”

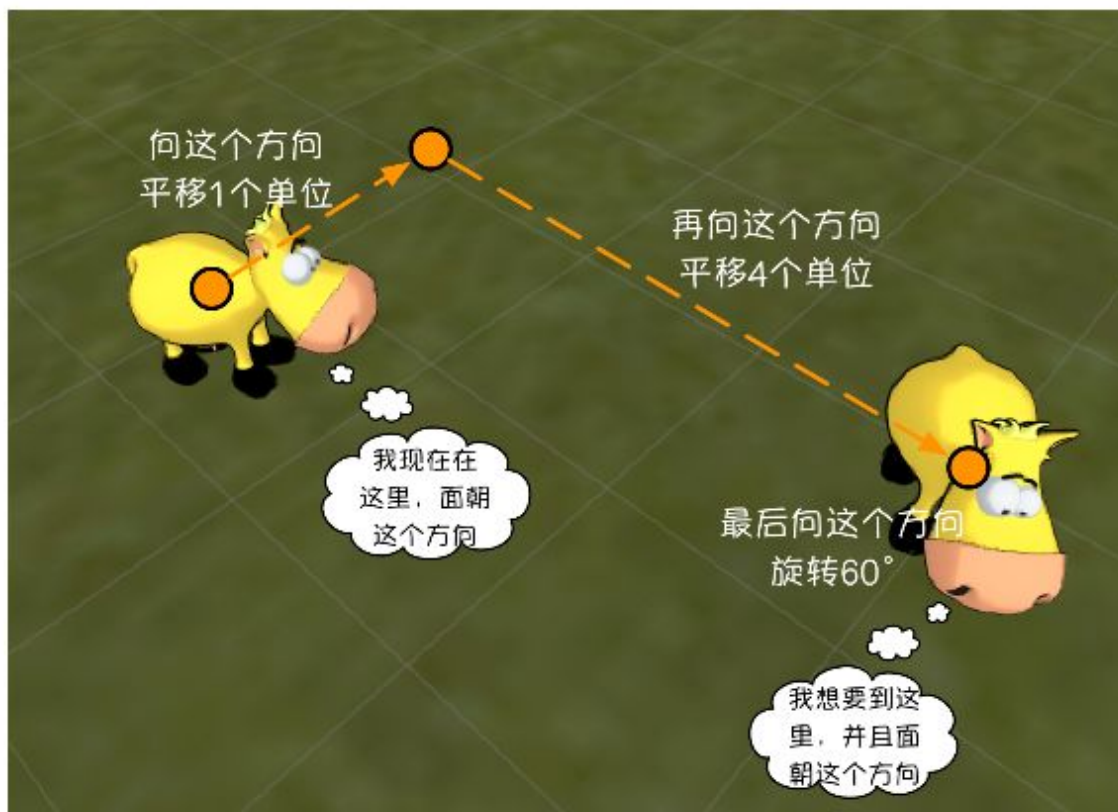


左手法则



右手法则

▲4.9 □□□□□□□□□□□□□□□□



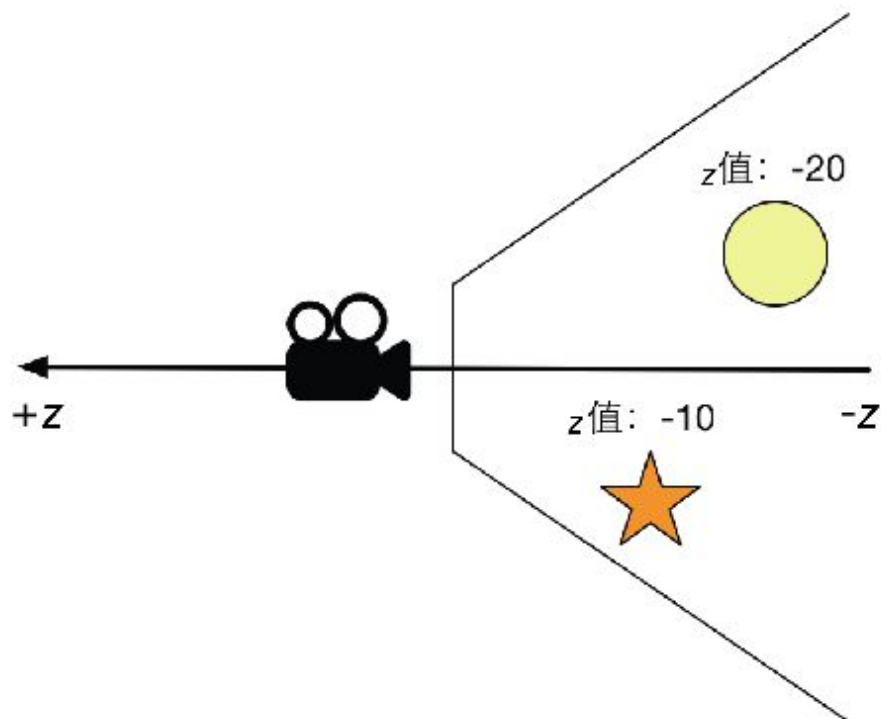
▲图4.10 从初始位置开始，先沿x轴方向平移1个单位，再沿x轴方向平移4个单位，最后沿x轴方向旋转60°

图4.10展示了从初始位置开始，先沿x轴方向平移1个单位，再沿x轴方向平移4个单位，最后沿x轴方向旋转60°的过程。图4.11展示了从初始位置开始，先沿x轴方向平移1个单位，再沿z轴方向平移4个单位，最后沿z轴方向旋转-60°的过程。



▲图4.11 从初始位置开始，先沿x轴方向平移1个单位，再沿z轴方向平移4个单位，最后沿z轴方向旋转-60°

图4.13所示



▲图4.13 Unity中的Z轴

Unity中的Z轴

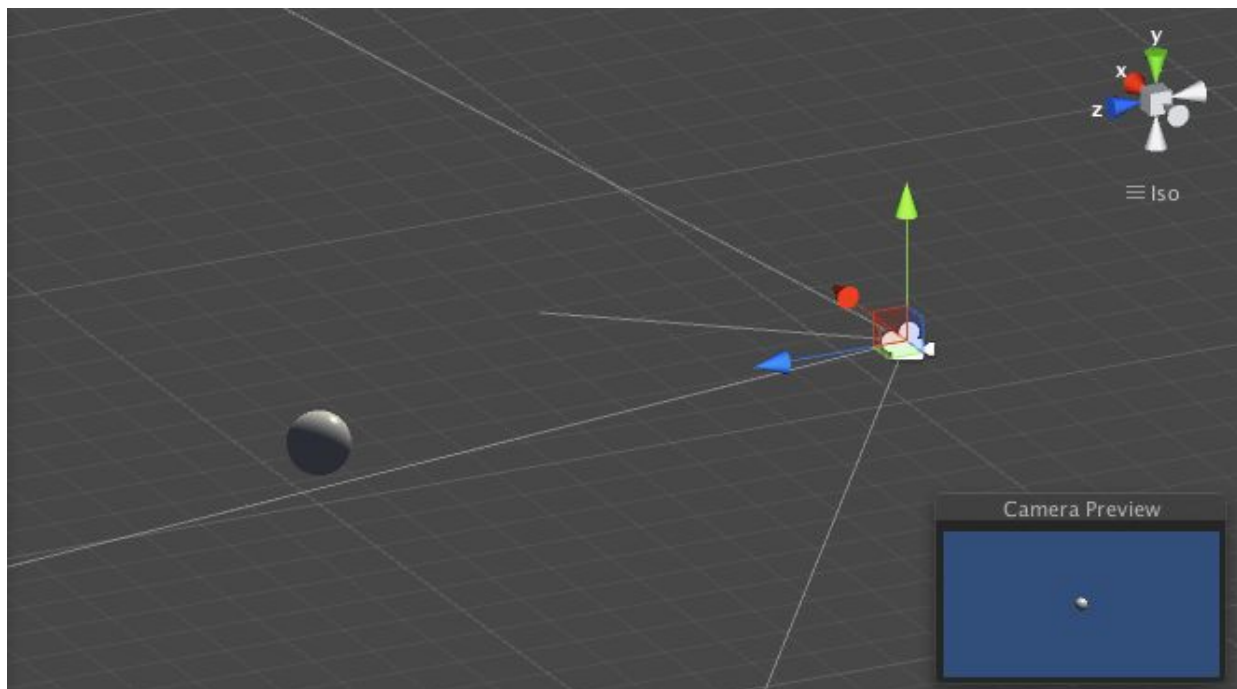
4.2.5 坐标

坐标

1. 在3ds Max中，x轴指向右，y轴指向上，z轴指向里。

2. 在Unity中，x轴指向右，y轴指向上，z轴指向里。

3. 在Unity中，x轴指向右，y轴指向上，z轴指向里。



▲4.14 位置ベクトル $(0, 1, -10)$ の位置に位置ベクトル $(0, 1, 0)$

の位置ベクトルを z 軸の位置ベクトルとして表示する

4.3 點與向

點 **point** 在 n 維空間中，點可以用 n 個坐標來表示。在二維空間中，點 $P = (P_x, P_y)$ 可以用 $P = (P_x, P_y, P_z)$ 來表示。

向 **vector** 是既有大小又有方向的量。向可以用 **scalar** 來表示其大小，用 **magnitude** 來表示其方向。例如，速度 **velocity** 是一個向，其大小是 80km/h ，方向是向北。距離 **distance** 是一個標量，其大小是 200 。

點與向

- 點可以用 (x, y) 來表示。
- 向可以用 (x, y, z) 來表示。

向 $\mathbf{v} = (x, y)$ 可以用 $\mathbf{v} = (x, y, z)$ 來表示。向 $\mathbf{v} = (x, y, z, w)$ 可以用 $\mathbf{v} = (x, y, z, w)$ 來表示。

向與點

- 向 $\mathbf{a}$ 可以用 $\mathbf{a} = b \mathbf{x} + c \mathbf{y} + d \mathbf{z} + e \mathbf{w}$ 來表示。
- 向 $\mathbf{a}$ 可以用 $\mathbf{a} = b \mathbf{u} + c \mathbf{v}$ 來表示。
- 向 $\mathbf{A}$ 可以用 $\mathbf{A} = \mathbf{B} + \mathbf{S} + \mathbf{M} + \mathbf{R}$ 來表示。

圖 4.15 顯示了向 $\mathbf{v}$ 的 **head** 和 **tail**。向 $\mathbf{v}$ 的 **head** 是向的終點，**tail** 是向的起點。

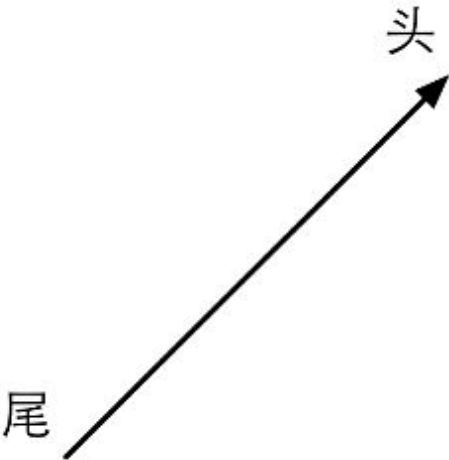
向 $\mathbf{v}$ 的大小是 100km/h ，方向是向北。向 $\mathbf{v}$ 的 **displacement** 是向的終點與起點之間的距離。

4.3.1 向量的表示

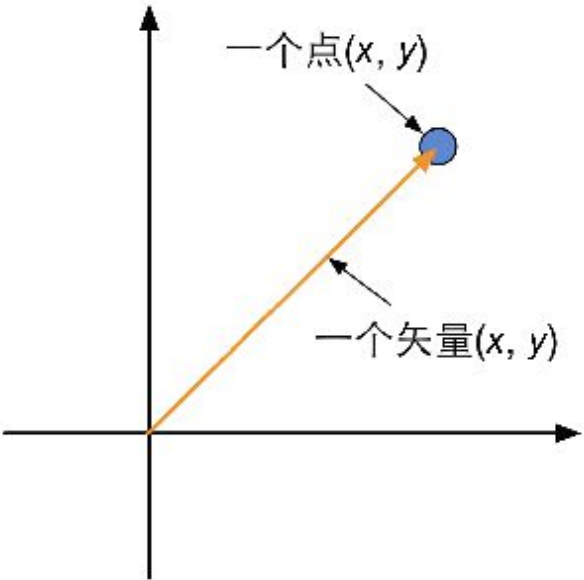
向量的表示方法有多种，最常见的是用有向线段表示。向量的大小（模）可以用线段的长度表示，向量的方向可以用线段的指向表示。

在平面直角坐标系中，向量可以用坐标表示。设向量 $\vec{a}$ 的起点为 (x_1, y_1) ，终点为 (x_2, y_2) ，则向量 $\vec{a}$ 的坐标表示为 $(x_2 - x_1, y_2 - y_1)$ 。

4.16 向量的坐标表示



▲4.15 向量的表示



▲4.16 空間ベクトル

空間ベクトルとは、空間内の任意の点を原点として、3次元の座標系で表すことができる。このとき、空間内の任意の点を (x, y, z) と表すことができる。また、空間内の任意の点を (x, y, z) と表すことができる。また、空間内の任意の点を (x, y, z) と表すことができる。

4.3.2 空間ベクトル

空間ベクトルとは、空間内の任意の点を原点として、3次元の座標系で表すことができる。このとき、空間内の任意の点を (x, y, z) と表すことができる。また、空間内の任意の点を (x, y, z) と表すことができる。また、空間内の任意の点を (x, y, z) と表すことができる。

1. 空間ベクトルの定義

空間ベクトルとは、空間内の任意の点を原点として、3次元の座標系で表すことができる。このとき、空間内の任意の点を (x, y, z) と表すことができる。また、空間内の任意の点を (x, y, z) と表すことができる。また、空間内の任意の点を (x, y, z) と表すことができる。

空間ベクトルとは、空間内の任意の点を原点として、3次元の座標系で表すことができる。

$$k\mathbf{v} = (kv_x, kv_y, kv_z)$$

空間ベクトルとは、空間内の任意の点を原点として、3次元の座標系で表すことができる。

$$\frac{\mathbf{v}}{k} = \frac{(x, y, z)}{k} = \frac{1}{k}(x, y, z) = \left(\frac{x}{k}, \frac{y}{k}, \frac{z}{k}\right), k \neq 0$$

空間ベクトルとは、空間内の任意の点を原点として、3次元の座標系で表すことができる。

$$2(1, 2, 3) = (2, 4, 6)$$

$$-3.5(2, 0) = (-7, 0)$$

$$\frac{(1, 2, 3)}{2} = (0.5, 1, 1.5)$$

空間ベクトルとは、空間内の任意の点を原点として、3次元の座標系で表すことができる。このとき、空間内の任意の点を (x, y, z) と表すことができる。また、空間内の任意の点を (x, y, z) と表すことができる。また、空間内の任意の点を (x, y, z) と表すことができる。

任意のベクトル $\mathbf{v}$ に対して k が任意のベクトル $\mathbf{v}$ に対して k が任意のベクトル
 任意のベクトル $\mathbf{v}$ に対して $2\|\mathbf{v}\| \leq \|\mathbf{v}\| + \|\mathbf{v}\|$ であることは、4.17 の結果から
 容易に示される。

2. ベクトルの加法と減法

任意のベクトル $\mathbf{a}$ と $\mathbf{b}$ に対して、 $\mathbf{a} + \mathbf{b}$ と $\mathbf{a} - \mathbf{b}$ はベクトルである。

任意のベクトル $\mathbf{a}$ と $\mathbf{b}$ に対して、 $\mathbf{a} + \mathbf{b}$ と $\mathbf{a} - \mathbf{b}$ はベクトルである。

$$\mathbf{a} + \mathbf{b} = (a_x + b_x, a_y + b_y, a_z + b_z)$$

$$\mathbf{a} - \mathbf{b} = (a_x - b_x, a_y - b_y, a_z - b_z)$$

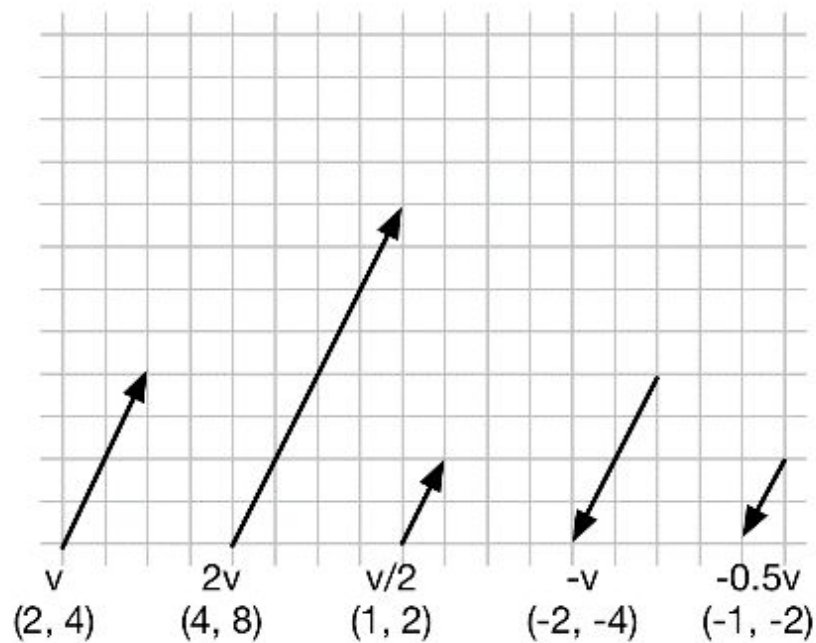
任意のベクトル $\mathbf{a}$ と $\mathbf{b}$ に対して、 $\mathbf{a} + \mathbf{b}$ と $\mathbf{a} - \mathbf{b}$ はベクトルである。

$$(1, 2, 3) + (4, 5, 6) = (5, 7, 9)$$

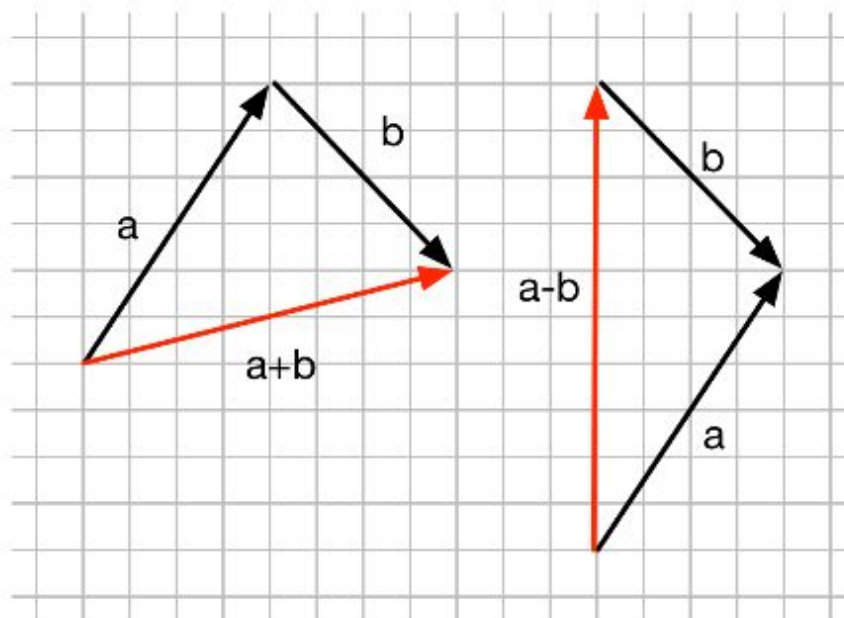
$$(5, 2, 7) - (3, 8, 4) = (2, -6, 3)$$

任意のベクトル $\mathbf{a}$ と $\mathbf{b}$ に対して、 $\mathbf{a} + \mathbf{b}$ と $\mathbf{a} - \mathbf{b}$ はベクトルである。

任意のベクトル $\mathbf{a}$ と $\mathbf{b}$ に対して、 $\mathbf{a} + \mathbf{b}$ と $\mathbf{a} - \mathbf{b}$ はベクトルである。
 任意のベクトル $\mathbf{a}$ と $\mathbf{b}$ に対して、 $\mathbf{a} + \mathbf{b}$ と $\mathbf{a} - \mathbf{b}$ はベクトルである。
 任意のベクトル $\mathbf{a}$ と $\mathbf{b}$ に対して、 $\mathbf{a} + \mathbf{b}$ と $\mathbf{a} - \mathbf{b}$ はベクトルである。
 triangle rule 任意のベクトル $\mathbf{a}$ と $\mathbf{b}$ に対して、 $\mathbf{a} + \mathbf{b}$ と $\mathbf{a} - \mathbf{b}$ はベクトルである。



▲4.17 向量的数乘



▲4.18 向量的加法

向量的加法满足交换律和结合律，即对于任意向量 a, b, c ，有 $a+b=b+a$ 和 $(a+b)+c=a+(b+c)$ 。

1. 若已知向量 a 与 b 的模长分别为 $|a| = 3$ 与 $|b| = 4$ ，且它们的夹角为 120° ，求 $|a + b|$ 与 $|a - b|$ 的值。

3. 计算

2. 已知向量 $a = (1, 2, 3)$ 与 $b = (3, 4, 5)$ ，求 $a \cdot b$ 与 $|a|$ 的值。

$$|v| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

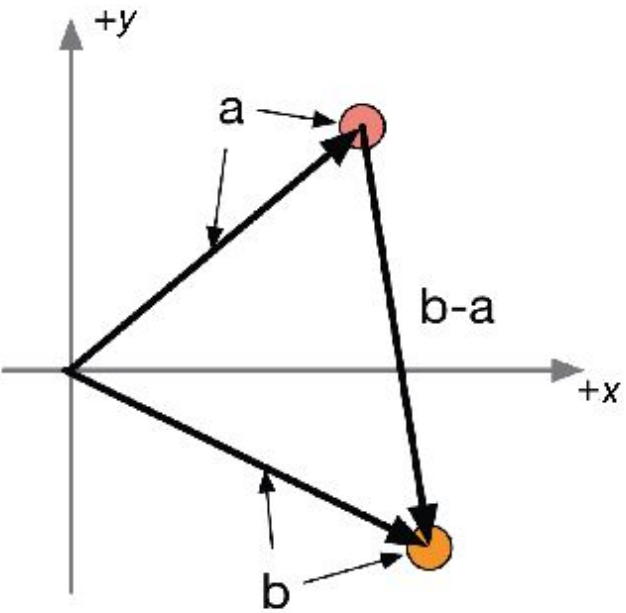
3. 已知向量 $a = (1, 2, 3)$ 与 $b = (3, 4, 5)$ ，求 $a \cdot b$ 与 $|a|$ 的值。

4. 已知向量 $a = (1, 2, 3)$ 与 $b = (3, 4, 5)$ ，求 $a \cdot b$ 与 $|a|$ 的值。

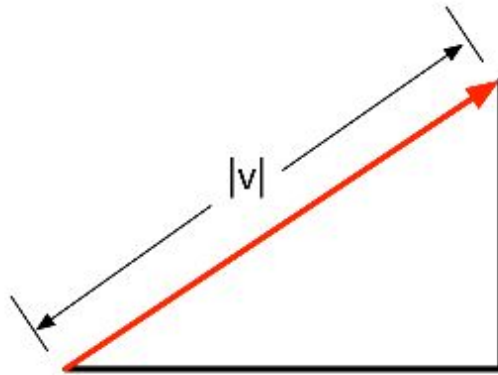
$$|(1, 2, 3)| = \sqrt{1^2 + 2^2 + 3^2} = \sqrt{1 + 4 + 9} = \sqrt{14} \approx 3.742$$

$$|(3, 4)| = \sqrt{3^2 + 4^2} = \sqrt{9 + 16} = \sqrt{25} = 5$$

4.20 已知向量 $a = (1, 2, 3)$ 与 $b = (3, 4, 5)$ ，求 $a \cdot b$ 与 $|a|$ 的值。



▲4.19 向量的模长a和b



▲4.20 单位向量

4.20 向量的模长为1的向量称为单位向量。单位向量的模长为1，方向与原向量相同。单位向量通常用 $\hat{v}$ 表示。

4.21 单位向量

单位向量的模长为1，方向与原向量相同。单位向量通常用 $\hat{v}$ 表示。单位向量的模长为1，方向与原向量相同。单位向量通常用 $\hat{v}$ 表示。单位向量的模长为1，方向与原向量相同。单位向量通常用 $\hat{v}$ 表示。

单位向量的模长为1，方向与原向量相同。单位向量通常用 $\hat{v}$ 表示。单位向量的模长为1，方向与原向量相同。单位向量通常用 $\hat{v}$ 表示。单位向量的模长为1，方向与原向量相同。单位向量通常用 $\hat{v}$ 表示。

单位向量的模长为1，方向与原向量相同。单位向量通常用 $\hat{v}$ 表示。单位向量的模长为1，方向与原向量相同。单位向量通常用 $\hat{v}$ 表示。单位向量的模长为1，方向与原向量相同。单位向量通常用 $\hat{v}$ 表示。

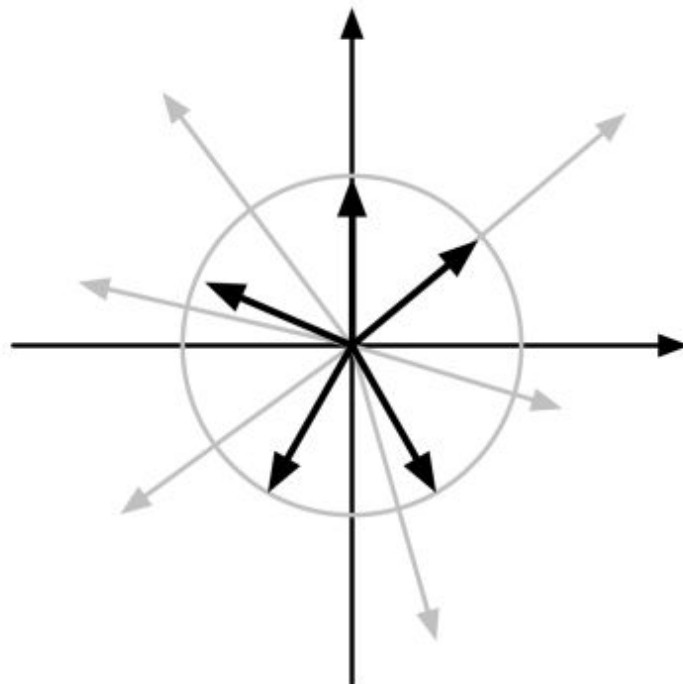
$$\hat{v} = \frac{v}{|v|}, v \text{ 为任意向量}$$

单位向量的模长为1

$$\frac{(3, -4)}{|(3, -4)|} = \frac{(3, -4)}{\sqrt{3^2 + (-4)^2}} = \frac{(3, -4)}{\sqrt{25}} = \frac{(3, -4)}{5} = \left(\frac{3}{5}, \frac{-4}{5}\right) = (0.6, -0.8)$$

那么 零向量 $\mathbf{0} = (0,0,0)$ 的内积就是 0

那么，我们如何求两个向量的内积呢？
 我们来看图 4.21 所示的向量。



▲图 4.21 向量的内积

我们来看图 4.21 所示的向量。图中显示了两个向量 $\mathbf{a}$ 和 $\mathbf{b}$ ，以及它们的内积。图中还显示了其他一些向量，用于说明内积的几何意义。

5 向量的内积

我们来看图 4.21 所示的向量。图中显示了两个向量 $\mathbf{a}$ 和 $\mathbf{b}$ ，以及它们的内积。图中还显示了其他一些向量，用于说明内积的几何意义。

dot product 内积 **inner product** 点积 **cross product** 叉积 **outer product** 外积

我们来看图 4.21 所示的向量。图中显示了两个向量 $\mathbf{a}$ 和 $\mathbf{b}$ ，以及它们的内积。图中还显示了其他一些向量，用于说明内积的几何意义。

Unity Shader 的 dot(a, b) 函数

Unity Shader 的 dot(a, b) 函数

Unity Shader 的 dot(a, b) 函数

$$\mathbf{a} \cdot \mathbf{b} = (a_x, a_y, a_z) \cdot (b_x, b_y, b_z) = a_x b_x + a_y b_y + a_z b_z$$

Unity Shader 的 dot(a, b) 函数

$$(1, 2, 3) \cdot (0.5, 4, 2.5) = 0.5 + 8 + 7.5 = 16$$

$$(-3, 4, 0) \cdot (5, -1, 7) = -15 + -4 + 0 = -19$$

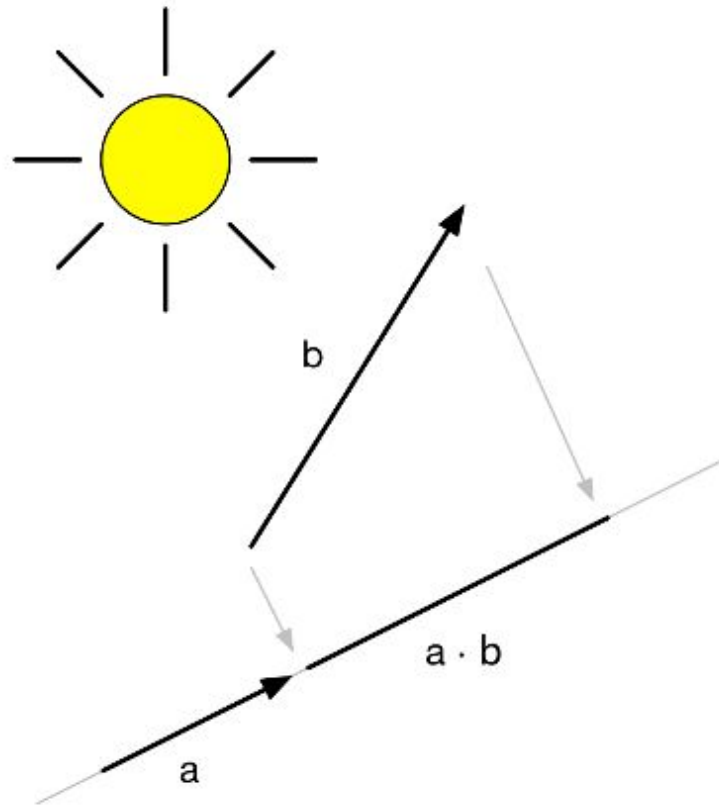
$$\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$$

Unity Shader 的 projection 函数

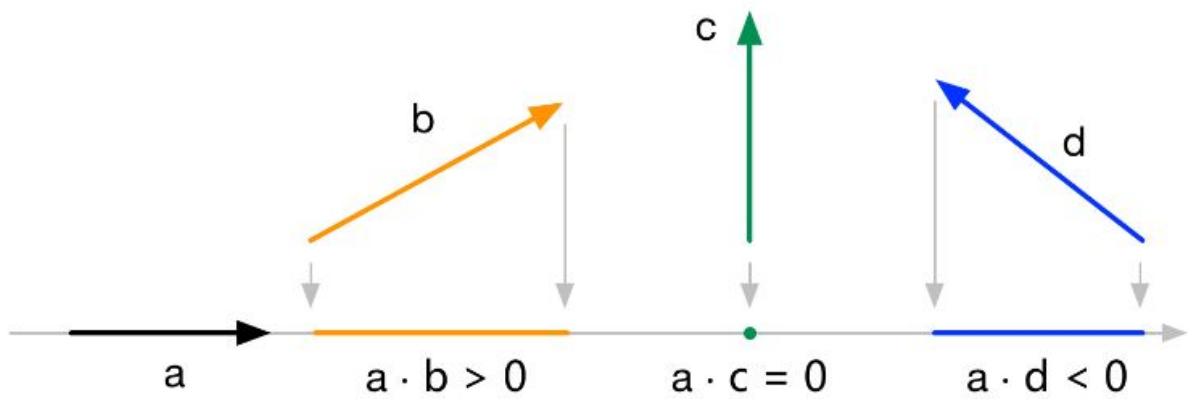
Unity Shader 的 projection 函数

Unity Shader 的 projection 函数

Unity Shader 的 projection 函数



▲ 4.22 b 在 a 上的投影



▲ 4.23 点积的符号

点积的符号与两向量的夹角有关

向量的单位化 $\hat{a}$ 向量的点积 $\mathbf{a} \cdot \mathbf{b}$ 向量的模 $|\mathbf{a}|$ 向量的叉积 $\mathbf{a} \times \mathbf{b}$

向量的点积在 Shader 中的应用

向量的叉积在 Shader 中的应用

向量的点积在 Shader 中的应用

$$(k \mathbf{a}) \cdot \mathbf{b} = \mathbf{a} \cdot (k \mathbf{b}) = k (\mathbf{a} \cdot \mathbf{b})$$

向量的点积在 Shader 中的应用

向量的叉积在 Shader 中的应用

向量的点积在 Shader 中的应用

$$\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}$$

向量的点积在 Shader 中的应用

向量的叉积在 Shader 中的应用

向量的点积在 Shader 中的应用

$$\mathbf{v} \cdot \mathbf{v} = v_x v_x + v_y v_y + v_z v_z = |\mathbf{v}|^2$$

向量的点积在 Shader 中的应用

向量的叉积在 Shader 中的应用

向量的点积在 Shader 中的应用

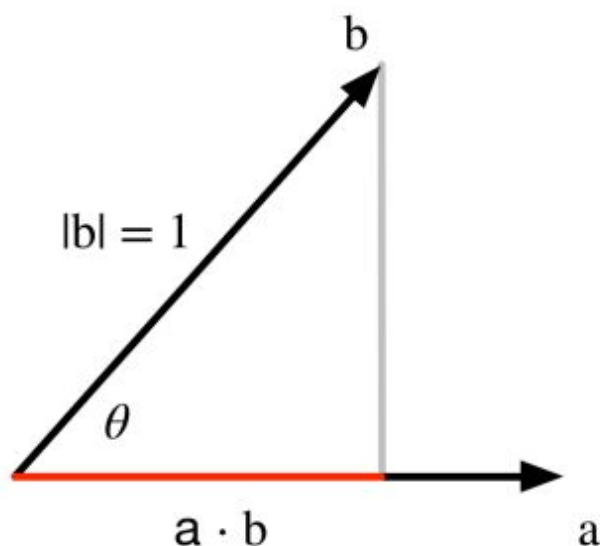
向量的叉积在 Shader 中的应用

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos \theta$$

我们假设 $\mathbf{a}$ 和 $\mathbf{b}$ 都是单位向量，那么 $|\mathbf{a}| = |\mathbf{b}| = 1$ ，于是式 (4.24) 变为

我们假设 $\mathbf{a}$ 和 $\mathbf{b}$ 都是单位向量，那么 $|\mathbf{a}| = |\mathbf{b}| = 1$ ，于是式 (4.24) 变为

$$\hat{\mathbf{a}} \cdot \hat{\mathbf{b}} = \frac{\text{直角边}}{\text{斜边}} = \cos \theta$$



▲ 4.24 单位向量的点积

我们假设 $\mathbf{a}$ 和 $\mathbf{b}$ 都是单位向量，那么 $|\mathbf{a}| = |\mathbf{b}| = 1$ ，于是式 (4.24) 变为

$$\mathbf{a} \cdot \mathbf{b} = (|\mathbf{a}| \hat{\mathbf{a}}) \cdot (|\mathbf{b}| \hat{\mathbf{b}}) = |\mathbf{a}| |\mathbf{b}| (\hat{\mathbf{a}} \cdot \hat{\mathbf{b}}) = |\mathbf{a}| |\mathbf{b}| \cos \theta$$

我们假设 $\mathbf{a}$ 和 $\mathbf{b}$ 都是单位向量，那么 $|\mathbf{a}| = |\mathbf{b}| = 1$ ，于是式 (4.24) 变为

两个向量的夹角在 0° 到 180° 之间

$$\theta = \arccos(\hat{a} \cdot \hat{b})$$

其中 $\hat{a}$ 和 $\hat{b}$ 是单位向量

即 $\arccos$ 的输入

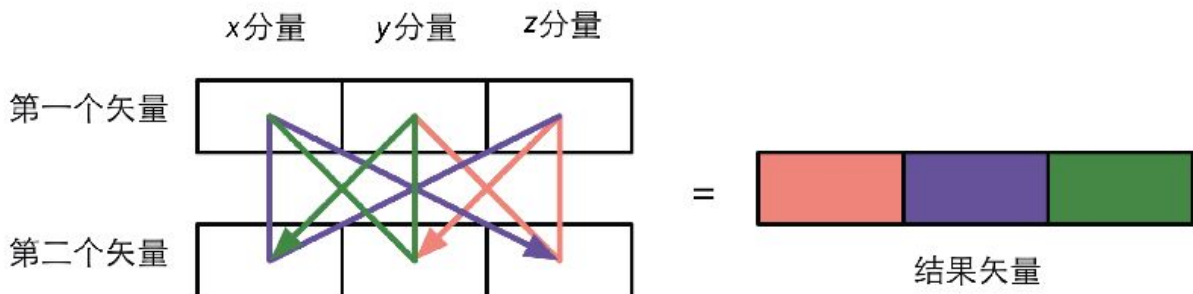
6. 叉积

叉积 (cross product) 和点积 (dot product) 是向量运算中的两个重要概念

两个向量 $\mathbf{a}$ 和 $\mathbf{b}$ 的叉积 $\mathbf{a} \times \mathbf{b}$ 是一个向量，其方向垂直于 $\mathbf{a}$ 和 $\mathbf{b}$ 所在的平面

$$\mathbf{a} \times \mathbf{b} = (a_x, a_y, a_z) \times (b_x, b_y, b_z) = (a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x)$$

例如，两个向量的叉积的大小为 4.25



▲ 4.25 两个向量的叉积的大小为 4.25

即

$$(1, 2, 3) \times (-2, -1, 4) = ((2)(4) - (3)(-1), (3)(-2) - (1)(4), (1)(-1) - (2)(-2))$$

$$= (8 - (-3), -6 - 4, -1 - (-4)) = (11, -10, 3)$$

$\mathbf{a} \times \mathbf{b} \neq \mathbf{b} \times \mathbf{a}$ 且 $\mathbf{a} \times \mathbf{b} = -(\mathbf{b} \times \mathbf{a})$
 $(\mathbf{a} \times \mathbf{b}) \times \mathbf{c} \neq \mathbf{a} \times (\mathbf{b} \times \mathbf{c})$

向量的叉积运算满足分配律：
 $\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c}$

向量的叉积运算满足结合律：
 $\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) = (\mathbf{a} \times \mathbf{b}) \times \mathbf{c}$

$$|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}| |\mathbf{b}| \sin \theta$$

向量的叉积运算满足反交换律：
 $\mathbf{a} \times \mathbf{b} = -(\mathbf{b} \times \mathbf{a})$
 向量的叉积运算满足分配律：
 $\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c}$

例4.26 设 $\mathbf{a}$ 与 $\mathbf{b}$ 的夹角为 θ ，求 $|\mathbf{a} \times \mathbf{b}|$ 。

解：由向量的叉积定义可知， $\mathbf{a} \times \mathbf{b}$ 的模等于 $|\mathbf{a}| |\mathbf{b}| \sin \theta$ ，即

$$|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}| |\mathbf{b}| \sin \theta$$

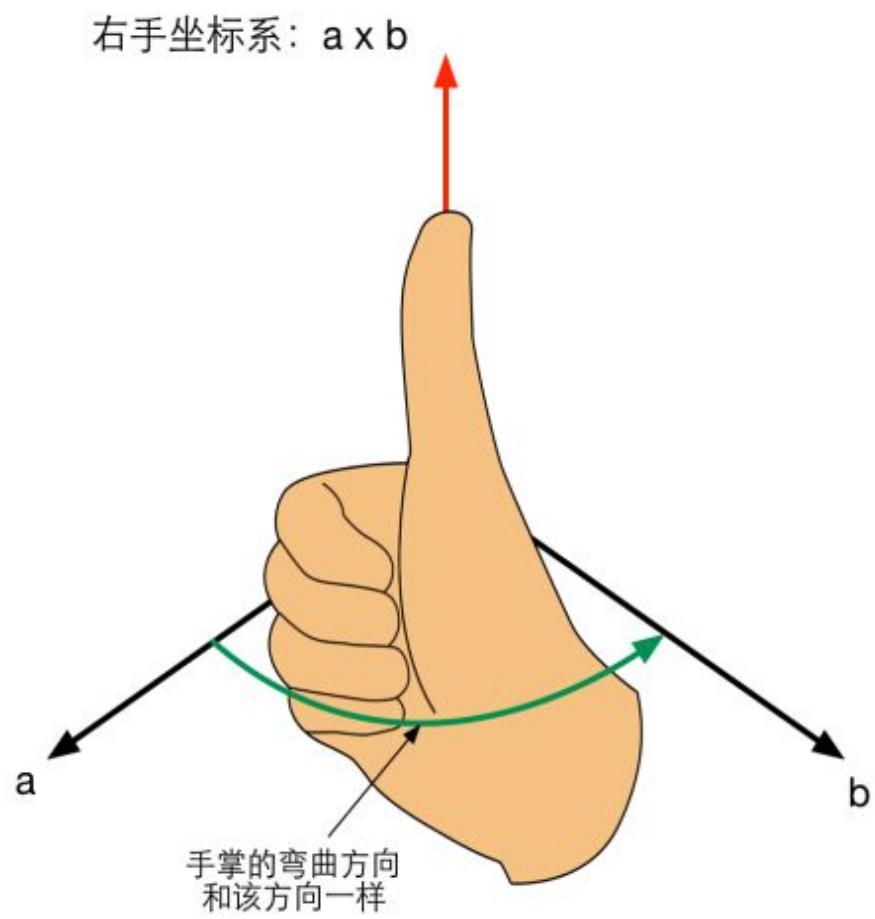
当 $\theta = 0$ 或 $\theta = \pi$ 时， $\sin \theta = 0$ ，故 $\mathbf{a} \times \mathbf{b} = \mathbf{0}$ 。

向量的叉积运算满足结合律：
 $\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) = (\mathbf{a} \times \mathbf{b}) \times \mathbf{c}$

例4.27 设 $\mathbf{a}$ 与 $\mathbf{b}$ 的夹角为 θ ，求 $|\mathbf{a} \times \mathbf{b}|$ 。



..... $\mathbf{a} \times \mathbf{b}$
 $\mathbf{a} \cdot \mathbf{b}$
 $\mathbf{a} \cdot \mathbf{b}$
 $\mathbf{a} \times \mathbf{b}$ 4.28



▲4.28 如何确定 $\mathbf{a} \times \mathbf{b}$ 的方向

如何确定 $\mathbf{a} \times \mathbf{b}$ 的方向？

如何确定 $\mathbf{a} \times \mathbf{b}$ 的方向？“右手定则”是确定 $\mathbf{a} \times \mathbf{b}$ 方向的方法。将右手四指从 $\mathbf{a}$ 沿小于 π 的角度转向 $\mathbf{b}$ ，此时大拇指所指的方向就是 $\mathbf{a} \times \mathbf{b}$ 的方向。

如何确定 $\mathbf{a} \times \mathbf{b}$ 的方向？“右手定则”是确定 $\mathbf{a} \times \mathbf{b}$ 方向的方法。

4.3.3 〇〇〇

[illegible]

10000

1

[illegible]

3

2□□□□□□□□□□

1(2,7,3)

$$\square 2\square 2.5(5,4,10)$$
$$\square 3 \square \frac{(3, 4)}{2}$$

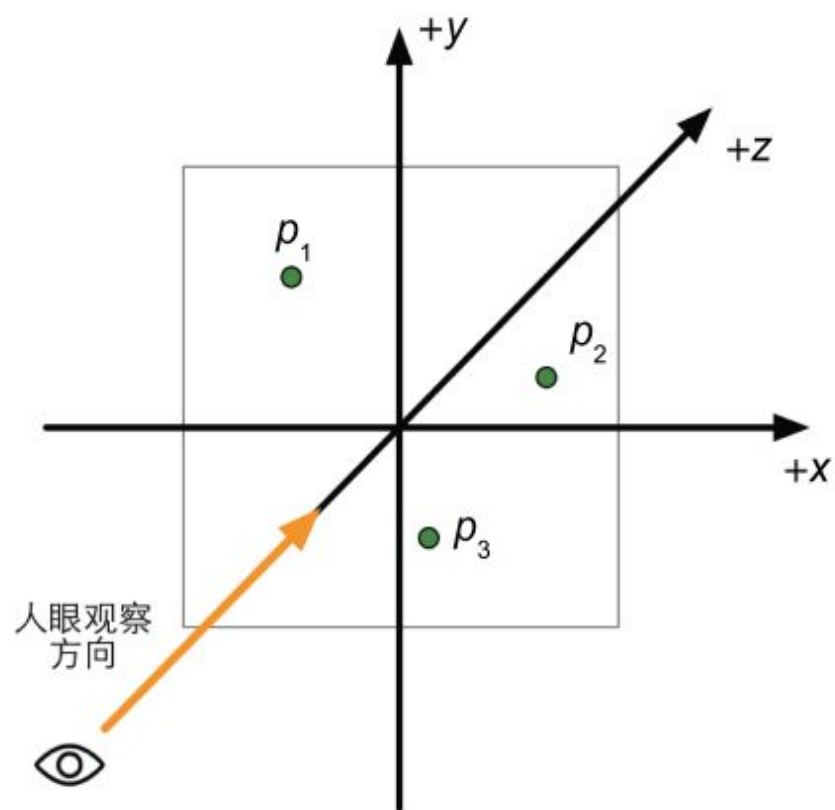
□4□□(5,12)□□□□□

□5□(1,1,1)□□□□□

$$\square 6\square(7,4)+(3,5)$$
$$\square 7\square(9,4,13)-(15,3,11)$$
[illegible]

4□□□□□□□□

$$\square 1 \square (4,7) \cdot (3,9)$$
$$\square 2\square(2,5,6)\cdot(3,1,2)-10$$
$$= 3 - 0.5(-3, 4) \cdot (-2, 5)$$



▲图4.29 在 xy 平面内观察 z 轴上的点

4.4 矩阵

矩阵(matrix)是线性代数中最重要的概念之一

——《The Matrix》

矩阵(matrix)是线性代数中最重要的概念之一。The Matrix 是一个由数字组成的矩形阵列，通常用于表示线性变换、求解方程组等。

矩阵的运算包括加法、减法、乘法等。2 阶矩阵的行列式可以用来判断矩阵是否可逆。

矩阵的转置是指将矩阵的行和列互换。矩阵的逆矩阵是指与原矩阵相乘得到单位矩阵的矩阵。

4.4.1 矩阵的表示

矩阵通常用大写字母表示，如 A 、 B 等。

$$\begin{bmatrix} 1 & 0.5 & 3 & 2 \\ 2.3 & 5 & \sqrt{3} & 10 \\ 4 & 8 & 11 & 5 \end{bmatrix}$$

矩阵的维数是指矩阵的行数和列数，通常表示为 $m \times n$ 。矩阵的行列式是一个标量值，可以用来判断矩阵是否可逆。

矩阵的行和列分别称为行向量和列向量。3×4 矩阵的行向量有 3 个，列向量有 4 个。3×3 矩阵的行列式可以用来判断矩阵是否可逆。

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

m_{ij} 是矩阵 M 中 i 行 j 列

元素

4.4.2 列向量

由 n 个元素组成的向量称为 $n \times 1$ 列向量，记为 **column matrix**。
由 $1 \times n$ 个元素组成的向量称为 $1 \times n$ 行向量，记为 **row matrix**。例如 $\mathbf{v} = (3, 8, 6)$ 是行向量。

$[3 \ 8 \ 6]$

列向量

$\begin{bmatrix} 3 \\ 8 \\ 6 \end{bmatrix}$

在 OpenGL 中，列向量通常用 $\mathbf{v}$ 表示，行向量通常用 $\mathbf{v}^T$ 表示。

在 Shader 中，列向量通常用 $\mathbf{v}$ 表示，行向量通常用 $\mathbf{v}^T$ 表示。

4.4.3 标量

标量是一个数值，通常用 s 表示。在 Shader 中，标量通常用 s 表示。

1. 标量

1. 假設 M 為 3×3 的矩陣，則 kM 為 3×3 的矩陣。

$$kM = Mk = k \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} = \begin{bmatrix} km_{11} & km_{12} & km_{13} \\ km_{21} & km_{22} & km_{23} \\ km_{31} & km_{32} & km_{33} \end{bmatrix}$$

2. 矩陣的乘法

假設 A 為 $r \times n$ 的矩陣， B 為 $n \times c$ 的矩陣，則 AB 為 $r \times c$ 的矩陣。

例如：假設 A 為 4×3 的矩陣， B 為 3×6 的矩陣，則 AB 為 4×6 的矩陣。

矩陣的乘法是左結合的，即 $(AB)C = A(BC)$ 。

假設 A 為 $r \times n$ 的矩陣， B 為 $n \times c$ 的矩陣， C 為 $r \times c$ 的矩陣，則 $C = AB$ 。

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{in}b_{nj} = \sum_{k=1}^n a_{ik}b_{kj}$$

假設 A 為 $r \times n$ 的矩陣， B 為 $n \times c$ 的矩陣，則 c_{ij} 為 A 的第 i 行與 B 的第 j 列的內積。

例如：假設 A 為 4×2 的矩陣， B 為 2×4 的矩陣，則 C 為 4×4 的矩陣。

$$c_{23} = a_{21}b_{13} + a_{22}b_{23}$$

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} \quad \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

▲4.30 c_{23} 的

Shader 的 4×4 的

的

的

的

$$\mathbf{AB} \neq \mathbf{BA}$$

的

的

$$(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$$

的

$$\mathbf{ABCDE} = ((\mathbf{A}(\mathbf{BC}))\mathbf{D})\mathbf{E} = (\mathbf{AB})(\mathbf{CD})\mathbf{E}$$

的

4.4.4 矩阵

Shader 中 Shader 矩阵

1. 矩阵

square matrix 3x3 4x4

diagonal elements m_{11} m_{22} m_{33} 0 diagonal matrix 4x4

$$\begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 7 \end{bmatrix}$$

2. 矩阵

identity matrix I_n 3x3

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$MI = IM = M$$

1

3. 矩阵

我们称 **transposed matrix** 为转置矩阵。
 若 $r \times c$ 矩阵 $\mathbf{M}$ 的转置矩阵 $\mathbf{M}^T$ 为 $c \times r$ 矩阵，
 且 $\mathbf{M}$ 中第 i 行第 j 列的元素等于 $\mathbf{M}^T$ 中第 j 行第 i 列的元素。

$$M_{ij}^T = M_{ji}$$

例如

$$\begin{bmatrix} 6 & 2 & 10 & 3 \\ 7 & 5 & 4 & 9 \end{bmatrix}^T = \begin{bmatrix} 6 & 7 \\ 2 & 5 \\ 10 & 4 \\ 3 & 9 \end{bmatrix}$$

转置矩阵的性质

$$\begin{bmatrix} x & y & z \end{bmatrix}^T = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}^T = \begin{bmatrix} x & y & z \end{bmatrix}$$

转置矩阵的性质

转置矩阵的性质

转置矩阵的性质

$$(\mathbf{M}^T)^T = \mathbf{M}$$

转置矩阵的性质

转置矩阵的性质

$$(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$$

转置矩阵的性质

4 逆矩阵

逆矩阵 **inverse matrix** 是指对于任意一个非奇异矩阵 M ，存在一个唯一的矩阵 M^{-1} ，使得 $MM^{-1} = M^{-1}M = I$ ，其中 I 是单位矩阵。

对于任意一个非奇异矩阵 M ，存在一个唯一的矩阵 M^{-1} ，使得 $MM^{-1} = M^{-1}M = I$ ，其中 I 是单位矩阵。

$$MM^{-1} = M^{-1}M = I$$

如果一个矩阵 M 的行列式不为 0，那么它是可逆的，称为 **invertible** 或 **nonsingular**。如果一个矩阵 M 的行列式为 0，那么它是不可逆的，称为 **noninvertible** 或 **singular**。

行列式 **determinant** 是一个标量值，可以用来判断一个矩阵是否可逆。如果行列式不为 0，则矩阵可逆；如果行列式为 0，则矩阵不可逆。在 Shader 编程中，可以使用 C++ 的 Eigen 库或 Unity 的 Matrix 类来计算行列式。Unity 的 Matrix 类提供了 `determinant` 方法。

4.8 逆矩阵

逆矩阵的定义

对于任意一个非奇异矩阵 M ，存在一个唯一的矩阵 M^{-1} ，使得 $MM^{-1} = M^{-1}M = I$ ，其中 I 是单位矩阵。

逆矩阵 M^{-1} 的性质

$$(M^{-1})^{-1} = M$$

逆矩阵的计算

逆矩阵的计算

$$I^{-1} = I$$

我们假设 M 是一个 $n \times n$ 的实数矩阵，且 M 是可逆的。那么， M 的逆矩阵 M^{-1} 满足 $MM^{-1} = I$ ，其中 I 是 $n \times n$ 的单位矩阵。

我们考虑一个 3×3 的实数矩阵 M ，其元素为 c_1, c_2, c_3 。

$$\begin{aligned}
 M^T M &= \begin{bmatrix} - & c_1 & - \\ - & c_2 & - \\ - & c_3 & - \end{bmatrix} \begin{bmatrix} | & | & | \\ c_1 & c_2 & c_3 \\ | & | & | \end{bmatrix} \\
 &= \begin{bmatrix} c_1 \cdot c_1 & c_1 \cdot c_2 & c_1 \cdot c_3 \\ c_2 \cdot c_1 & c_2 \cdot c_2 & c_2 \cdot c_3 \\ c_3 \cdot c_1 & c_3 \cdot c_2 & c_3 \cdot c_3 \end{bmatrix} \\
 &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = I
 \end{aligned}$$

我们假设 M 是一个 3×3 的实数矩阵。

$$c_1 \cdot c_1 = 1, c_1 \cdot c_2 = 0, c_1 \cdot c_3 = 0$$

$$c_2 \cdot c_1 = 0, c_2 \cdot c_2 = 1, c_2 \cdot c_3 = 0$$

$$c_3 \cdot c_1 = 0, c_3 \cdot c_2 = 0, c_3 \cdot c_3 = 1$$

我们假设 M 是一个 3×3 的实数矩阵。

- 我们假设 c_1, c_2, c_3 是 M 的列向量，且 c_1, c_2, c_3 是正交的。
- 我们假设 c_1, c_2, c_3 是 M 的列向量，且 c_1, c_2, c_3 是正交的。
- 我们假设 M 是一个 3×3 的实数矩阵，且 M 是可逆的。

我们假设 M 是一个 3×3 的实数矩阵，且 M 是可逆的。那么， M 的逆矩阵 M^{-1} 满足 $MM^{-1} = I$ ，其中 I 是 3×3 的单位矩阵。

orthogonal basis

orthonormal basis

4.4.5

$\mathbf{v} = (x, y, z)$ $\mathbf{v} = [xyz]$ $\mathbf{v} = [x \ y \ z]^T$ $\mathbf{M}$

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

$\mathbf{M}$

$$vM = [xm_{11} + ym_{21} + zm_{31} \quad xm_{12} + ym_{22} + zm_{32} \quad xm_{13} + ym_{23} + zm_{33}]$$

$$Mv = \begin{bmatrix} xm_{11} + ym_{12} + zm_{13} \\ xm_{21} + ym_{22} + zm_{23} \\ xm_{31} + ym_{32} + zm_{33} \end{bmatrix}$$

Unity

$$CBAv = (C(B(Av)))$$

$\mathbf{v} = \mathbf{A} \mathbf{B} \mathbf{C}$

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □

$$vA^T B^T C^T = (((vA^T)B^T)C^T)$$

□□□□□□□□□□□□□□□□□□3□

4.4.6 ☐☐☐

1□□□□□□□□□□□□□□□□□□□□□□□□

$$\mathbf{1} \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} -1 & 5 \\ 0 & 2 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 4 & 3 \\ 2 & 1 & 4 \end{bmatrix} \begin{bmatrix} -1 & 5 \\ 0 & 2 \\ 3 & 10 \\ 4 & 5 \end{bmatrix}$$

$$\boxed{3} \begin{bmatrix} 1 & -2 & 3 \\ 5 & 1 & 4 \\ 6 & 0 & 3 \end{bmatrix} \begin{bmatrix} -5 \\ 4 \\ 8 \end{bmatrix}$$

2□□□□□□□□□□□□□□

$$\mathbf{1} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

□2□

□3□

[illegible]

1

□2□

□3□

4.5 线性变换

线性变换的定义

- 线性变换的定义
- 线性变换的性质
- 线性变换的表示
- 线性变换的逆

线性变换的表示

线性变换的表示矩阵

线性变换的表示矩阵

4.5.1 线性变换

线性变换的定义

线性变换的定义

$$f(x) + f(y) = f(x + y)$$

$$k f(x) = f(k x)$$

线性变换的定义

項目	項目	項目	項目	項目
項目	Y	Y	Y	Y
項目	Y	Y	N	N
項目	N	N	N	N

4.6.7

4.5.2

3×3 4×4 homogeneous coordinate

w 1 w 0 4×4

4.5.3

4×4 4

$$\begin{bmatrix} M_{3 \times 3} & t_{3 \times 3} \\ 0_{1 \times 3} & 1 \end{bmatrix}$$

$\begin{bmatrix} 0 & 0 & 0 \end{bmatrix}_{1 \times 3} \mathbf{M}_{3 \times 3} \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}_{1 \times 3} \mathbf{t}_{3 \times 1} + \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}_{1 \times 3} \mathbf{I}_3 \begin{bmatrix} 1 \end{bmatrix}_{1 \times 1}$

$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}_{4 \times 4}$

4.5.4 变换

$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}$$

$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$ 是 3D 空间中的点 (x, y, z) 的齐次坐标 (t_x, t_y, t_z)

$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$

$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$

$\mathbf{M}_{3 \times 3}$ 是 3×3 的矩阵 $\mathbf{I}_3$

$\begin{bmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$

$$\begin{bmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$\begin{bmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$

4.5.5 坐标轴

坐标轴通常用x y z 表示，坐标轴的单位向量为

$$\begin{bmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} k_x x \\ k_y y \\ k_z z \\ 1 \end{bmatrix}$$

坐标轴的单位向量为

$$\begin{bmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \begin{bmatrix} k_x x \\ k_y y \\ k_z z \\ 0 \end{bmatrix}$$

当 $k_x = k_y = k_z$ 时，坐标轴称为 **uniform scale** ；
当 $k_x \neq k_y \neq k_z$ 时，坐标轴称为 **nonuniform scale** 。

坐标轴的单位向量为

$$\begin{bmatrix} \frac{1}{k_x} & 0 & 0 & 0 \\ 0 & \frac{1}{k_y} & 0 & 0 \\ 0 & 0 & \frac{1}{k_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

坐标轴的单位向量为

坐标轴的单位向量为

4.5.6 坐标轴

1. 在 3D 空间中，我们通常使用齐次坐标来表示点。假设我们有一个点 P ，其坐标为 (x, y, z) 。

2. 绕 x 轴旋转 θ 角度的变换矩阵 $R_x(\theta)$ 为：

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3. 绕 y 轴旋转 θ 角度的变换矩阵 $R_y(\theta)$ 为：

$$R_y(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

4. 绕 z 轴旋转 θ 角度的变换矩阵 $R_z(\theta)$ 为：

$$R_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5. 假设我们有一个点 $P(2, 2, 2)$ ，我们想要计算它在绕 x 轴旋转 30° 后的新位置 P' 。

4.5.7 练习

1. 计算点 $P(2, 2, 2)$ 绕 y 轴旋转 30° 后的新位置 P' 。

$$p_{new} = M_{translation} M_{rotation} M_{scale} p_{old}$$

2. 计算点 $P(2, 2, 2)$ 绕 z 轴旋转 45° 后的新位置 P' 。

1. 在 MATLAB 中，如何定义一个 4x4 的矩阵？

2. 如何计算两个 4x4 矩阵的乘积？

3. 如何求解一个 4x4 矩阵的逆？

4. 假设有一个 3D 点 (0, 0, 5)，如何将其平移 5 个单位沿 x 轴，2 个单位沿 y 轴，2 个单位沿 z 轴？

5. 假设有一个 3D 点 (0, 0, 10)，如何将其绕 y 轴旋转 30 度？

$$\begin{aligned}
 M_{translation} M_{rotation} M_{scale} &= \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} k_x \cos \theta & 0 & k_z \sin \theta & t_x \\ 0 & k_y & 0 & t_y \\ -k_x \sin \theta & 0 & k_z \cos \theta & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}
 \end{aligned}$$

6. 如何计算上述矩阵的逆？

$$\begin{aligned}
 M_{rotation} M_{scale} M_{translation} &= \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} k_x \cos \theta & 0 & k_z \sin \theta & t_x k_x \cos \theta + t_z k_z \sin \theta \\ 0 & k_y & 0 & t_y k_y \\ -k_x \sin \theta & 0 & k_z \cos \theta & -t_x k_x \sin \theta + t_z k_z \cos \theta \\ 0 & 0 & 0 & 1 \end{bmatrix}
 \end{aligned}$$

7. 如何验证上述结果？

4.5.6

 x y z 3 x y z

$(\theta_x, \theta_y, \theta_z)$ Unity

 zxy API $(\theta_x, \theta_y, \theta_z)$

$$[M_{rotate_z} M_{rotate_x} M_{rotate_y}] = \begin{bmatrix} \cos \theta_z & -\sin \theta_z & 0 & 0 \\ \sin \theta_z & \cos \theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta_x & -\sin \theta_x & 0 \\ 0 & \sin \theta_x & \cos \theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta_y & 0 & \sin \theta_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta_y & 0 & \cos \theta_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

zxy $(\theta_x, \theta_y, \theta_z)$

- E z θ_z E y θ_y E x θ_x
- E z θ_z E z θ_z E' y θ_y E' y θ_y E' x θ_x

zxy yxz Unity

4.6 纹理

纹理是计算机图形学中的一个重要概念，它用于为模型表面赋予颜色和图案。在 OpenGL 中，纹理通常以二维数组的形式存储，每个元素代表一个像素的颜色值。

纹理的加载和绑定是图形编程中的基础操作。在 OpenGL 中，我们通常使用 `glTexEnvf` 和 `glTexEnvfv` 函数来设置纹理的环境参数，如颜色、透明度等。

纹理的采样和过滤也是图形编程中的关键部分。在 OpenGL 中，我们通常使用 `glTexGenf` 和 `glTexGenfv` 函数来设置纹理的生成参数，如坐标、范围等。

4.6.1 纹理的加载和绑定

在 OpenGL 中，纹理的加载和绑定是一个复杂的过程。首先，我们需要将纹理数据加载到 GPU 的显存中。这通常通过 `glTexSubImage2D` 函数来完成。然后，我们需要将纹理绑定到当前的渲染目标上，以便在渲染时能够正确显示。

在 OpenGL 中，纹理的加载和绑定通常是通过调用 `glTexEnvf` 和 `glTexEnvfv` 函数来实现的。这些函数允许我们设置纹理的环境参数，如颜色、透明度等。

在 OpenGL 中，纹理的采样和过滤也是一个重要的环节。在 OpenGL 中，我们通常使用 `glTexGenf` 和 `glTexGenfv` 函数来设置纹理的生成参数，如坐标、范围等。这些参数决定了纹理在渲染时的采样方式和过滤效果。

在 OpenGL 中，纹理的加载和绑定通常是通过调用 `glTexEnvf` 和 `glTexEnvfv` 函数来实现的。这些函数允许我们设置纹理的环境参数，如颜色、透明度等。在 OpenGL 4.6.5 版本中，我们还可以通过调用 `glTexSubImage2D` 函数来设置纹理的生成参数，如坐标、范围等。

1. 點 A 在直線 l 上

2. x 軸上的點 a

3. y 軸上的點 b

4. z 軸上的點 c

點 A 在直線 l 上，且 A 在 x 軸、 y 軸、 z 軸上的射影分別為 a 、 b 、 c 。設 O 為原點， $\mathbf{O}_c$ 為點 O 在 z 軸上的射影， $\mathbf{P}$ 為點 A 在 xy 平面上的射影， $\mathbf{x}_c$ 、 $\mathbf{y}_c$ 、 $\mathbf{z}_c$ 分別為 $\mathbf{O}_c$ 在 x 、 y 、 z 軸上的射影。求 A 的坐標。

1. 點 A 在直線 l 上

點 A 在直線 l 上，且 A 在 x 軸、 y 軸、 z 軸上的射影分別為 a 、 b 、 c 。設 O 為原點， $\mathbf{O}_c$ 為點 O 在 z 軸上的射影， $\mathbf{P}$ 為點 A 在 xy 平面上的射影， $\mathbf{x}_c$ 、 $\mathbf{y}_c$ 、 $\mathbf{z}_c$ 分別為 $\mathbf{O}_c$ 在 x 、 y 、 z 軸上的射影。求 A 的坐標。

2. x 軸上的點 a

點 A 在直線 l 上，且 A 在 x 軸、 y 軸、 z 軸上的射影分別為 a 、 b 、 c 。設 O 為原點， $\mathbf{O}_c$ 為點 O 在 z 軸上的射影， $\mathbf{P}$ 為點 A 在 xy 平面上的射影， $\mathbf{x}_c$ 、 $\mathbf{y}_c$ 、 $\mathbf{z}_c$ 分別為 $\mathbf{O}_c$ 在 x 、 y 、 z 軸上的射影。求 A 的坐標。

$$O_c + ax_c$$

3. y 軸上的點 b

點 A 在直線 l 上，且 A 在 x 軸、 y 軸、 z 軸上的射影分別為 a 、 b 、 c 。設 O 為原點， $\mathbf{O}_c$ 為點 O 在 z 軸上的射影， $\mathbf{P}$ 為點 A 在 xy 平面上的射影， $\mathbf{x}_c$ 、 $\mathbf{y}_c$ 、 $\mathbf{z}_c$ 分別為 $\mathbf{O}_c$ 在 x 、 y 、 z 軸上的射影。求 A 的坐標。

$$O_c + ax_c + by_c$$

4. z 軸上的點 c

點 A 在直線 l 上，且 A 在 x 軸、 y 軸、 z 軸上的射影分別為 a 、 b 、 c 。設 O 為原點， $\mathbf{O}_c$ 為點 O 在 z 軸上的射影， $\mathbf{P}$ 為點 A 在 xy 平面上的射影， $\mathbf{x}_c$ 、 $\mathbf{y}_c$ 、 $\mathbf{z}_c$ 分別為 $\mathbf{O}_c$ 在 x 、 y 、 z 軸上的射影。求 A 的坐標。

$$O_c + ax_c + by_c + cz_c$$

點 A 在直線 l 上，且 A 在 x 軸、 y 軸、 z 軸上的射影分別為 a 、 b 、 c 。設 O 為原點， $\mathbf{O}_c$ 為點 O 在 z 軸上的射影， $\mathbf{P}$ 為點 A 在 xy 平面上的射影， $\mathbf{x}_c$ 、 $\mathbf{y}_c$ 、 $\mathbf{z}_c$ 分別為 $\mathbf{O}_c$ 在 x 、 y 、 z 軸上的射影。求 A 的坐標。

$$A_p = O_c + ax_c + by_c + cz_c$$

我们假设在平面上的任意一点 P 的坐标为 (x, y, z) ，那么它到原点的距离为

$$\begin{aligned}
 A_p &= O_c + ax_c + by_c + cz_c \\
 &= (x_{O_c}, y_{O_c}, z_{O_c}) + a(x_{x_c}, y_{x_c}, z_{x_c}) + b(x_{y_c}, y_{y_c}, z_{y_c}) + c(x_{z_c}, y_{z_c}, z_{z_c}) \\
 &= (x_{O_c}, y_{O_c}, z_{O_c}) + \begin{bmatrix} x_{x_c} & x_{y_c} & x_{z_c} \\ y_{x_c} & y_{y_c} & y_{z_c} \\ z_{x_c} & z_{y_c} & z_{z_c} \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} \\
 &= (x_{O_c}, y_{O_c}, z_{O_c}) + \begin{bmatrix} | & | & | \\ x_c & y_c & z_c \\ | & | & | \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix}
 \end{aligned}$$

我们假设在平面上的任意一点 P 的坐标为 (x, y, z) ，那么它到原点的距离为

我们假设在平面上的任意一点 P 的坐标为 (x, y, z) ，那么它到原点的距离为

$$\begin{aligned}
 A_p &= (x_{O_c}, y_{O_c}, z_{O_c}, 1) + \begin{bmatrix} | & | & | & 0 \\ x_c & y_c & z_c & 0 \\ | & | & | & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix} \\
 &= \begin{bmatrix} 1 & 0 & 0 & x_{O_c} \\ 0 & 1 & 0 & y_{O_c} \\ 0 & 0 & 1 & z_{O_c} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} | & | & | & 0 \\ x_c & y_c & z_c & 0 \\ | & | & | & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix} \\
 &= \begin{bmatrix} | & | & | & x_{O_c} \\ x_c & y_c & z_c & y_{O_c} \\ | & | & | & z_{O_c} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix} \\
 &= \begin{bmatrix} | & | & | & | \\ x_c & y_c & z_c & O_c \\ | & | & | & | \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix}
 \end{aligned}$$

我们假设在平面上的任意一点 P 的坐标为 (x, y, z) ，那么它到原点的距离为

[illegible]

$\mathbf{M}_{c \rightarrow p} \mathbf{M}_{p \rightarrow c}$ $\mathbf{C}$ $\mathbf{P}$ $\mathbf{P}$ $\mathbf{C}$

$\mathbf{M}_{c \rightarrow p}$ $\mathbf{C}$ $\mathbf{P}$
 3 3 0 1

3D point cloud $\mathbf{x}_c, \mathbf{y}_c, \mathbf{z}_c$ is generated by the 3D point cloud generator.

$\mathbf{M}_{c \rightarrow p}$ is a 4×4 matrix that maps the coordinates x, y, z of the point $\mathbf{C}$ in the camera coordinate system to the coordinates x', y', z' of the point $\mathbf{C}$ in the point cloud coordinate system.

A 3x3 grid of squares. The bottom-left square contains the text "3x3".

Shader 3x3 4x4 5x5 6x6 7x7 8x8 9x9 10x10 11x11 12x12 13x13 14x14 15x15 16x16 17x17 18x18 19x19 20x20 21x21 22x22 23x23 24x24 25x25 26x26 27x27 28x28 29x29 30x30 31x31 32x32 33x33 34x34 35x35 36x36 37x37 38x38 39x39 40x40 41x41 42x42 43x43 44x44 45x45 46x46 47x47 48x48 49x49 50x50 51x51 52x52 53x53 54x54 55x55 56x56 57x57 58x58 59x59 60x60 61x61 62x62 63x63 64x64 65x65 66x66 67x67 68x68 69x69 70x70 71x71 72x72 73x73 74x74 75x75 76x76 77x77 78x78 79x79 80x80 81x81 82x82 83x83 84x84 85x85 86x86 87x87 88x88 89x89 90x90 91x91 92x92 93x93 94x94 95x95 96x96 97x97 98x98 99x99 100x100 101x101 102x102 103x103 104x104 105x105 106x106 107x107 108x108 109x109 110x110 111x111 112x112 113x113 114x114 115x115 116x116 117x117 118x118 119x119 120x120 121x121 122x122 123x123 124x124 125x125 126x126 127x127 128x128 129x129 130x130 131x131 132x132 133x133 134x134 135x135 136x136 137x137 138x138 139x139 140x140 141x141 142x142 143x143 144x144 145x145 146x146 147x147 148x148 149x149 150x150 151x151 152x152 153x153 154x154 155x155 156x156 157x157 158x158 159x159 160x160 161x161 162x162 163x163 164x164 165x165 166x166 167x167 168x168 169x169 170x170 171x171 172x172 173x173 174x174 175x175 176x176 177x177 178x178 179x179 180x180 181x181 182x182 183x183 184x184 185x185 186x186 187x187 188x188 189x189 190x190 191x191 192x192 193x193 194x194 195x195 196x196 197x197 198x198 199x199 200x200 201x201 202x202 203x203 204x204 205x205 206x206 207x207 208x208 209x209 210x210 211x211 212x212 213x213 214x214 215x215 216x216 217x217 218x218 219x219 220x220 221x221 222x222 223x223 224x224 225x225 226x226 227x227 228x228 229x229 230x230 231x231 232x232 233x233 234x234 235x235 236x236 237x237 238x238 239x239 240x240 241x241 242x242 243x243 244x244 245x245 246x246 247x247 248x248 249x249 250x250 251x251 252x252 253x253 254x254 255x255 256x256 257x257 258x258 259x259 260x260 261x261 262x262 263x263 264x264 265x265 266x266 267x267 268x268 269x269 270x270 271x271 272x272 273x273 274x274 275x275 276x276 277x277 278x278 279x279 280x280 281x281 282x282 283x283 284x284 285x285 286x286 287x287 288x288 289x289 290x290 291x291 292x292 293x293 294x294 295x295 296x296 297x297 298x298 299x299 300x300 301x301 302x302 303x303 304x304 305x305 306x306 307x307 308x308 309x309 310x310 311x311 312x312 313x313 314x314 315x315 316x316 317x317 318x318 319x319 320x320 321x321 322x322 323x323 324x324 325x325 326x326 327x327 328x328 329x329 330x330 331x331 332x332 333x333 334x334 335x335 336x336 337x337 338x338 339x339 340x340 341x341 342x342 343x343 344x344 345x345 346x346 347x347 348x348 349x349 350x350 351x351 352x352 353x353 354x354 355x355 356x356 357x357 358x358 359x359 360x360 361x361 362x362 363x363 364x364 365x365 366x366 367x367 368x368 369x369 370x370 371x371 372x372 373x373 374x374 375x375 376x376 377x377 378x378 379x379 380x380 381x381 382x382 383x383 384x384 385x385 386x386 387x387 388x388 389x389 390x390 391x391 392x392 393x393 394x394 395x395 396x396 397x397 398x398 399x399 400x400 401x401 402x402 403x403 404x404 405x405 406x406 407x407 408x408 409x409 410x410 411x411 412x412 413x413 414x414 415x415 416x416 417x417 418x418 419x419 420x420 421x421 422x422 423x423 424x424 425x425 426x426 427x427 428x428 429x429 430x430 431x431 432x432 433x433 434x434 435x435 436x436 437x437 438x438 439x439 440x440 441x441 442x442 443x443 444x444 445x445 446x446 447x447 448x448 449x449 450x450 451x451 452x452 453x453 454x454 455x455 456x456 457x457 458x458 459x459 460x460 461x461 462x462 463x463 464x464 465x465 466x466 467x467 468x468 469x469 470x470 471x471 472x472 473x473 474x474 475x475 476x476 477x477 478x478 479x479 480x480 481x481 482x482 483x483 484x484 485x485 486x486 487x487 488x488 489x489 490x490 491x491 492x492 493x493 494x494 495x495 496x496 497x497 498x498 499x499 500x500 501x501 502x502 503x503 504x504 505x505 506x506 507x507 508x508 509x509 510x510 511x511 512x512 513x513 514x514 515x515 516x516 517x517 518x518 519x519 520x520 521x521 522x522 523x523 524x524 525x525 526x526 527x527 528x528 529x529 530x530 531x531 532x532 533x533 534x534 535x535 536x536 53

我们假设命题 $\mathbf{M}_{p \rightarrow c}$ 的命题变元为 $\mathbf{M}_{c \rightarrow p}$ ，我们假设命题变元 $\mathbf{M}_{p \rightarrow c}$ 的命题变元为 $\mathbf{M}_{p \rightarrow c}$ ，我们假设命题变元 $\mathbf{M}_{c \rightarrow p}$ 的命题变元为 $\mathbf{M}_{c \rightarrow p}$ 。

$$M_{p \rightarrow c} = \begin{bmatrix} | & | & | \\ x_p & y_p & z_p \\ | & | & | \end{bmatrix} = M_{c \rightarrow p}^{-1} = M_{c \rightarrow p}^T$$

$$= \begin{bmatrix} - & x_c & - \\ - & y_c & - \\ - & z_c & - \end{bmatrix}$$

我们假设命题变元 $\mathbf{M}_{c \rightarrow p}$ 的命题变元为 $\mathbf{x}_c, \mathbf{y}_c, \mathbf{z}_c$ ，我们假设命题变元 $\mathbf{x}_p, \mathbf{y}_p, \mathbf{z}_p$ 的命题变元为 $\mathbf{M}_{c \rightarrow p}$ ，我们假设命题变元 $\mathbf{M}_{A \rightarrow B}$ 的命题变元为 $\mathbf{A}, \mathbf{x}, \mathbf{B}$ ，我们假设命题变元 $\mathbf{B}, \mathbf{x}$ 的命题变元为 $\mathbf{A}, \mathbf{B}, \mathbf{x}, \mathbf{y}, \mathbf{z}$ ，我们假设命题变元 $\mathbf{A}, \mathbf{B}$ 的命题变元为 $\mathbf{A}, \mathbf{B}$ 。

我们假设命题变元 $\mathbf{P}, \mathbf{C}$ 的命题变元为 $\mathbf{P}, \mathbf{C}$ ，我们假设命题变元 $\mathbf{P}, \mathbf{C}$ 的命题变元为 $\mathbf{P}, \mathbf{C}$ 。

我们假设命题变元 $\mathbf{P}, \mathbf{C}$ 的命题变元为 $\mathbf{P}, \mathbf{C}$ ，我们假设命题变元 $\mathbf{P}, \mathbf{C}$ 的命题变元为 $\mathbf{P}, \mathbf{C}$ 。

我们假设命题变元 $\mathbf{A}, \mathbf{B}$ 的命题变元为 $\mathbf{A}, \mathbf{B}$ ，我们假设命题变元 $\mathbf{B}, \mathbf{x}, \mathbf{y}, \mathbf{z}$ 的命题变元为 $\mathbf{A}, \mathbf{B}, \mathbf{x}_B, \mathbf{y}_B, \mathbf{z}_B$ ，我们假设命题变元 $\mathbf{A}, \mathbf{B}$ 的命题变元为 $\mathbf{M}_{A \rightarrow B}$ ，我们假设命题变元 $\mathbf{A}, \mathbf{B}$ 的命题变元为 $\mathbf{A}, \mathbf{B}$ 。

$$M_{A \rightarrow B} = \begin{bmatrix} | & | & | \\ x_B & y_B & z_B \\ | & | & | \end{bmatrix}$$

我们考虑从基 $\mathbf{B}$ 到基 $\mathbf{A}$ 的过渡矩阵 $\mathbf{M}_{A \rightarrow B}$ 。设 $\mathbf{x}_B$ 是基 $\mathbf{B}$ 下的坐标向量， $\mathbf{x}_A$ 是基 $\mathbf{A}$ 下的坐标向量。我们有：

$$M_{A \rightarrow B} x_B = \begin{bmatrix} | & | & | \\ x_B & y_B & z_B \\ | & | & | \end{bmatrix} x_B$$

我们考虑从基 $\mathbf{B}$ 到基 $\mathbf{A}$ 的过渡矩阵 $\mathbf{M}_{A \rightarrow B}$ 。设 $\mathbf{x}_B$ 是基 $\mathbf{B}$ 下的坐标向量， $\mathbf{x}_A$ 是基 $\mathbf{A}$ 下的坐标向量。我们有：

$$M_{A \rightarrow B} x_B = \begin{bmatrix} - & x_B & - \\ - & y_B & - \\ - & z_B & - \end{bmatrix} x_B = \begin{bmatrix} x_B \cdot x_B \\ y_B \cdot x_B \\ z_B \cdot x_B \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

我们考虑从基 $\mathbf{B}$ 到基 $\mathbf{A}$ 的过渡矩阵 $\mathbf{M}_{A \rightarrow B}$ 。

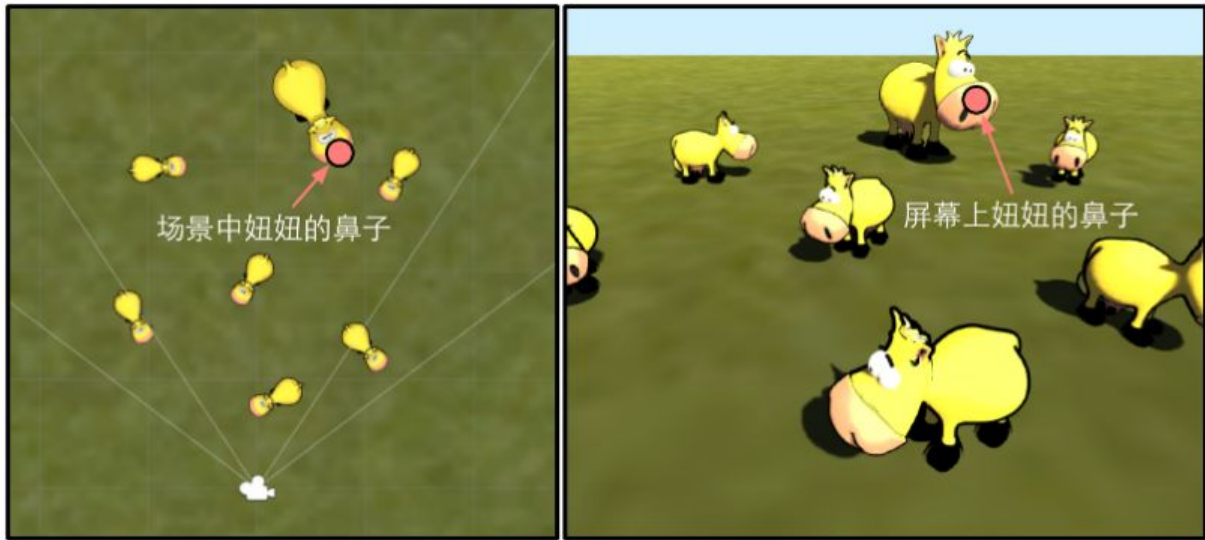
我们考虑从基 $\mathbf{B}$ 到基 $\mathbf{A}$ 的过渡矩阵 $\mathbf{M}_{A \rightarrow B}$ 。

4.6.3 从基 $\mathbf{B}$ 到基 $\mathbf{A}$ 的过渡矩阵

我们考虑从基 $\mathbf{B}$ 到基 $\mathbf{A}$ 的过渡矩阵 $\mathbf{M}_{A \rightarrow B}$ 。设 $\mathbf{x}_B$ 是基 $\mathbf{B}$ 下的坐标向量， $\mathbf{x}_A$ 是基 $\mathbf{A}$ 下的坐标向量。我们有：

我们考虑从基 $\mathbf{B}$ 到基 $\mathbf{A}$ 的过渡矩阵 $\mathbf{M}_{A \rightarrow B}$ 。

我们考虑从基 $\mathbf{B}$ 到基 $\mathbf{A}$ 的过渡矩阵 $\mathbf{M}_{A \rightarrow B}$ 。



▲图4.31 场景中妞妞的鼻子和屏幕上妞妞的鼻子

在Unity中，我们通常使用两种方式来指定物体的位置：

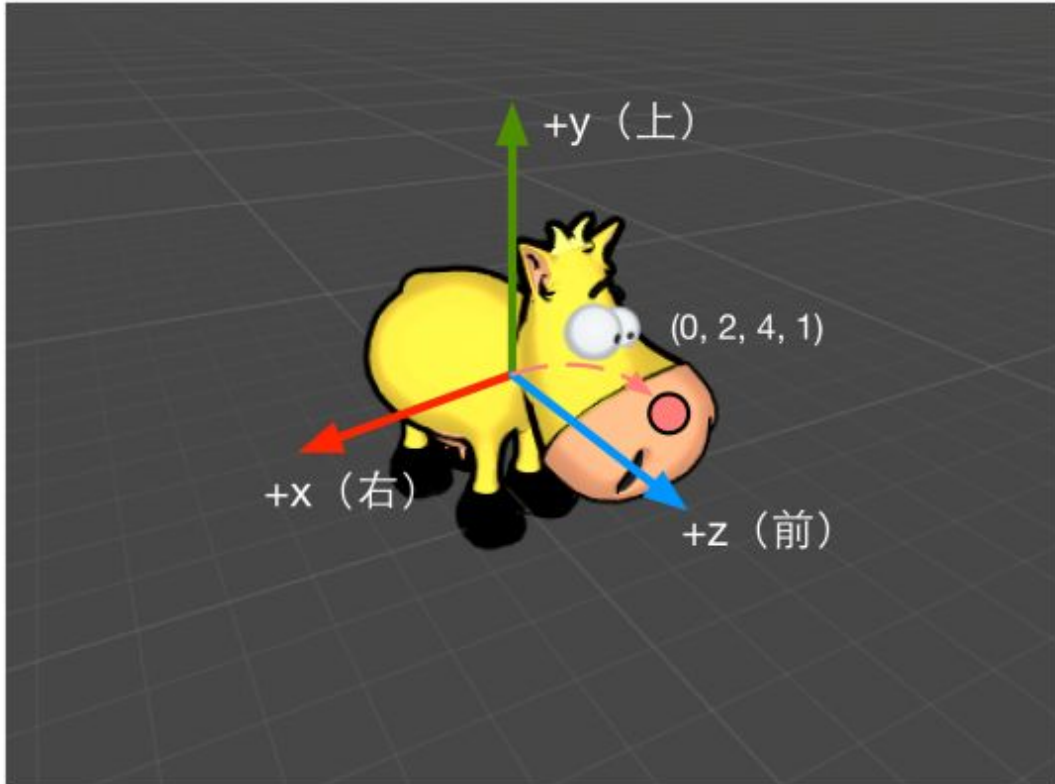
4.6.4 坐标系

在Unity中，我们通常使用两种方式来指定物体的位置：
一种是使用**model space**（模型空间）来指定物体的位置，另一种是使用**object space**（对象空间）来指定物体的位置。
在**model space**中，物体的位置是相对于其自身的局部坐标系来指定的。而在**object space**中，物体的位置是相对于其在场景中的父级对象来指定的。

在Unity中，我们通常使用两种方式来指定物体的位置：
一种是使用**forward**（前）、**back**（后）、**left**（左）、**right**（右）、**up**（上）、**down**（下）来指定物体的位置。
另一种是使用**x**、**y**、**z**来指定物体的位置。
在Unity中，我们通常使用两种方式来指定物体的位置：
一种是使用**forward**（前）、**back**（后）、**left**（左）、**right**（右）、**up**（上）、**down**（下）来指定物体的位置。
另一种是使用**x**、**y**、**z**来指定物体的位置。

在Unity中，我们通常使用两种方式来指定物体的位置：
一种是使用**forward**（前）、**back**（后）、**left**（左）、**right**（右）、**up**（上）、**down**（下）来指定物体的位置。
另一种是使用**x**、**y**、**z**来指定物体的位置。

このように、座標空間の原点を(0, 0, 0)とすると、キャラクターの位置は(0, 2, 4)と表すことができます。
また、キャラクターの向きは(0, 2, 4, 1)と表すことができます(図4.32参照)。



▲図4.32 座標空間の原点を(0, 0, 0)とすると、キャラクターの位置は(0, 2, 4, 1)と表すことができます。

4.6.5 座標空間

座標空間は **world space** と呼ばれる空間で、この空間でキャラクターの位置や向きを指定します。この空間の原点は(0, 0, 0)と表され、この空間の軸は+x (右)、+y (上)、+z (前)と表されます。

この空間の原点は(0, 0, 0)と表され、この空間の軸は+x (右)、+y (上)、+z (前)と表されます。この空間の原点は(0, 0, 0)と表され、この空間の軸は+x (右)、+y (上)、+z (前)と表されます。

Unityでは、座標空間の位置は `x`、`y`、`z` の3つの値で表されます。Unityでは、座標空間の位置は `Transform` の `Position` プロパティで指定します。

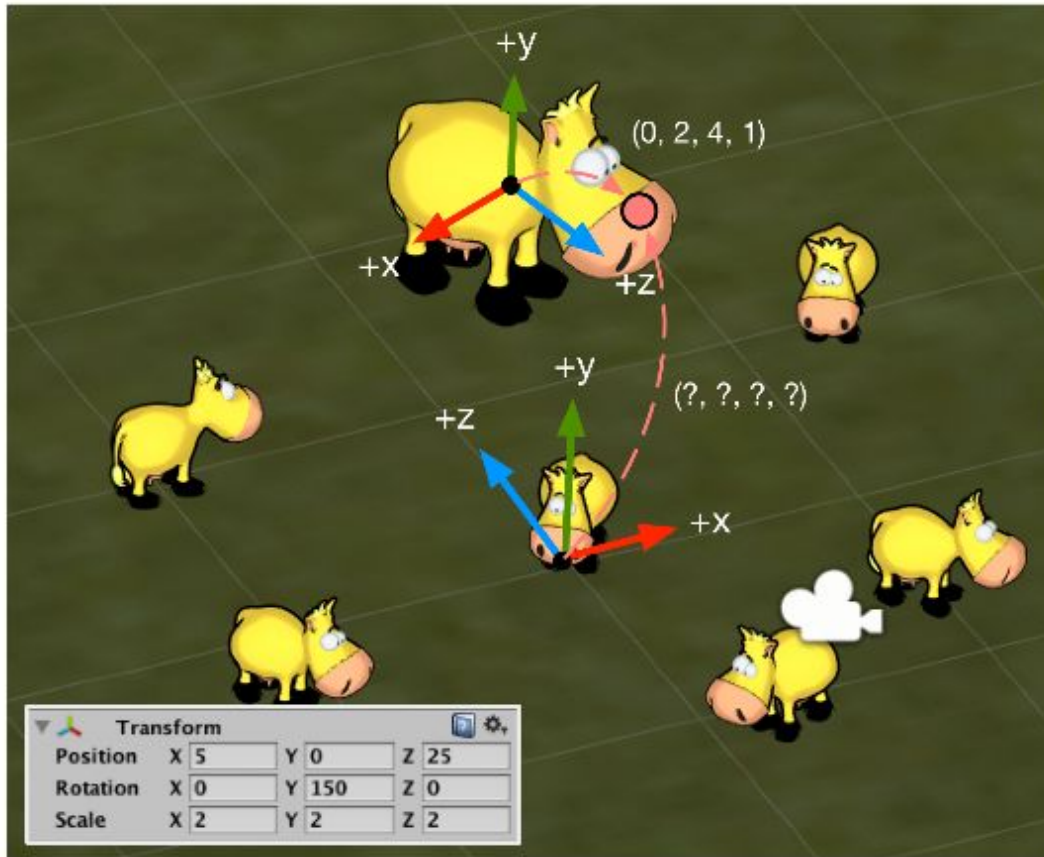
Transform 的 parent 属性指向它的父 Transform 对象。图 4.33 展示了 Transform 的 Rotation 和 Scale 属性。

图 4.34 展示了 model transform 的概念。

图 4.34 展示了 Transform 的 Position 属性。图 4.34



▲图 4.33 Unity 的 Transform 属性。Transform 的 Position 属性指向 "Mesh" 的 Position 属性。"Cow" 的 Position 属性指向 "Cow" 的 Position 属性。



▲ 4.34 3D 모델의 변환 (Translation, Rotation, Scale)을 나타내는 행렬을 구성하는 방법을 설명하십시오.

3D Transform 행렬은 Translation, Rotation, Scale을 포함합니다. 예를 들어, Translation은 (2, 2, 2)로 이동, Rotation은 (0, 150, 0)로 회전, Scale은 (5, 0, 25)로 크기 조절을 나타냅니다.

$$M_{model} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} k_x & 0 & 0 & 0 \\ 0 & k_y & 0 & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 25 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -0.866 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \\ -0.5 & 0 & -0.866 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} -1.732 & 0 & 1 & 5 \\ 0 & 2 & 0 & 0 \\ -1 & 0 & -1.732 & 25 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

この行列を計算すると、

$$\mathbf{P}_{world} = \mathbf{M}_{model} \mathbf{P}_{model}$$

$$= \begin{bmatrix} -1.732 & 0 & 1 & 5 \\ 0 & 2 & 0 & 0 \\ -1 & 0 & -1.732 & 25 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 9 \\ 4 \\ 18.072 \\ 1 \end{bmatrix}$$

この結果から、カメラの位置は (9, 4, 18.072) であることがわかる。この結果を Unity の `Camera` クラスの `position` プロパティに設定する。

4.6.6 カメラの向き

カメラの向きは、**view space** と **camera space** の関係で決まる。view space は、カメラの位置を原点とした座標系で、camera space は、カメラの向きを基準とした座標系である。この2つの座標系を関係付けるために、カメラの向きを表現する行列が必要となる。

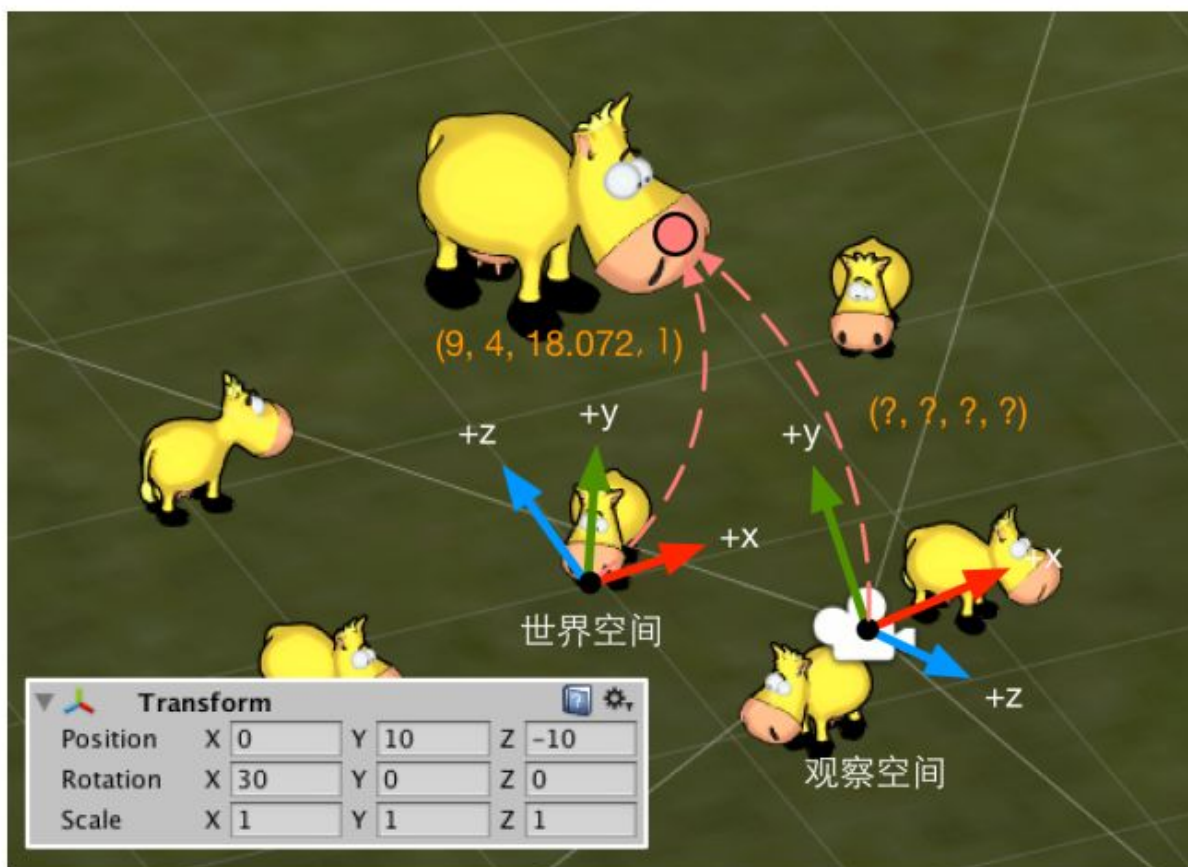
カメラの向きを表現する行列は、カメラの位置を原点とした座標系 (view space) と、カメラの向きを基準とした座標系 (camera space) の関係で決まる。この2つの座標系を関係付けるために、カメラの向きを表現する行列が必要となる。Unity では、`Camera` クラスの `transform` プロパティを使用して、カメラの向きを設定することができる。また、OpenGL の `glLookAt` 関数も、カメラの向きを設定するための関数である。

Unity では、`Camera` クラスの `transform` プロパティを使用して、カメラの向きを設定することができる。また、`Camera` クラスの `cameraToWorldMatrix` プロパティと `worldToCameraMatrix` プロパティも、カメラの向きを設定するための行列である。

4.6.8 投影变换
 projection 变换

view transform

Transform 4.35



▲4.35 观察空间中的坐标轴与z轴
 Unity 观察空间中的坐标轴与z轴

4.6.2 投影变换

1. 在OpenGL中，我们使用gluLookAt()函数来设置视图。该函数的原型如下：

void gluLookAt(
 double eyex, double eyey, double ezez, // 观察者的眼睛位置
 double cx, double cy, double cz, // 观察的目标点
 double ex, double ey, double ez // 观察的方向向量
);

$$\begin{aligned}
 M_{view} &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.866 & 0.5 & 0 \\ 0 & -0.5 & 0.866 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -10 \\ 0 & 0 & 1 & 10 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.866 & 0.5 & -3.66 \\ 0 & -0.5 & 0.866 & 13.66 \\ 0 & 0 & 0 & 1 \end{bmatrix}
 \end{aligned}$$

2. 在OpenGL中，我们使用gluProject()函数来将视图中的点投影到窗口坐标。该函数的原型如下：

$$\mathbf{M}_{view} = \mathbf{M}_{negate\ z} \mathbf{M}_{view}$$

$$\begin{aligned}
 &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.866 & 0.5 & -3.66 \\ 0 & -0.5 & 0.866 & 13.66 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.866 & 0.5 & -3.66 \\ 0 & 0.5 & -0.866 & -13.66 \\ 0 & 0 & 0 & 1 \end{bmatrix}
 \end{aligned}$$

3. 在OpenGL中，我们使用gluUnProject()函数来将窗口坐标反投影到视图坐标。该函数的原型如下：

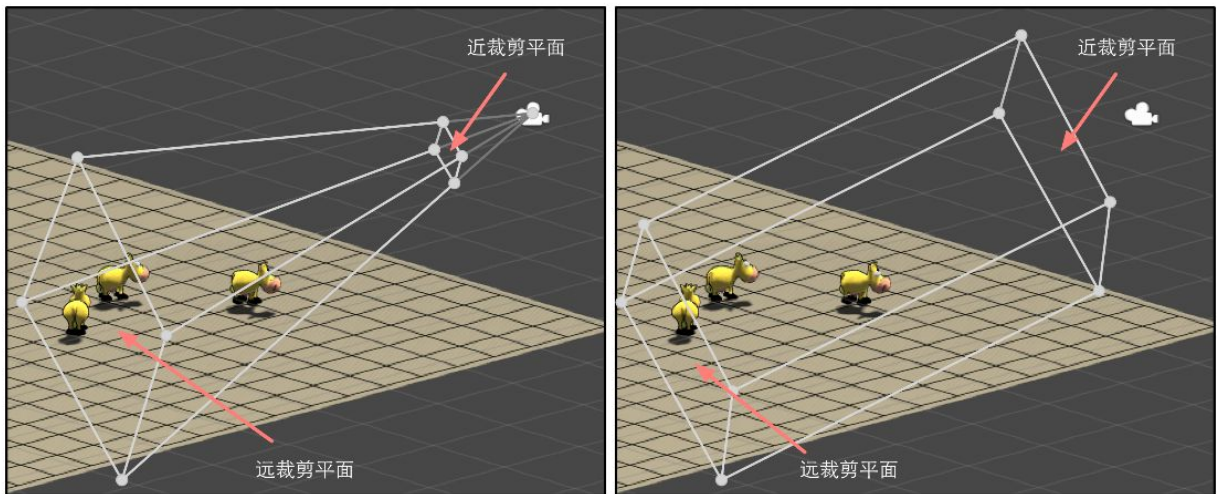
$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.866 & 0.5 & -3.66 \\ 0 & 0.5 & -0.866 & -13.66 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 9 \\ 4 \\ 18.072 \\ 1 \end{bmatrix} = \begin{bmatrix} 9 \\ 8.84 \\ -27.31 \\ 1 \end{bmatrix}$$

4.6.7 □□□□

[illegible][illegible]

在3D图形引擎中，我们通常使用近裁剪平面（near clip plane）和远裁剪平面（far clip plane）来定义视锥体的范围。近裁剪平面通常位于相机前方，而远裁剪平面通常位于视锥体的远端。在2D图形引擎中，我们通常使用HUD（Heads-Up Display）来显示游戏中的信息。

在6.1节中，我们讨论了近裁剪平面（near clip plane）和远裁剪平面（far clip plane）的定义。在4.37节中，我们将讨论如何计算近裁剪平面和远裁剪平面的位置。



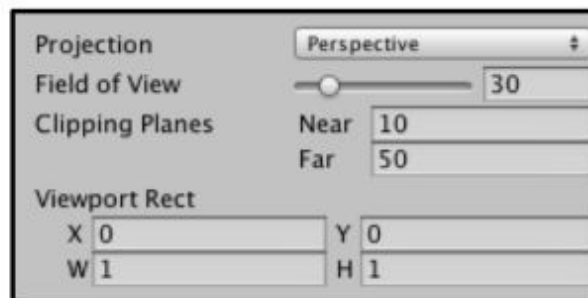
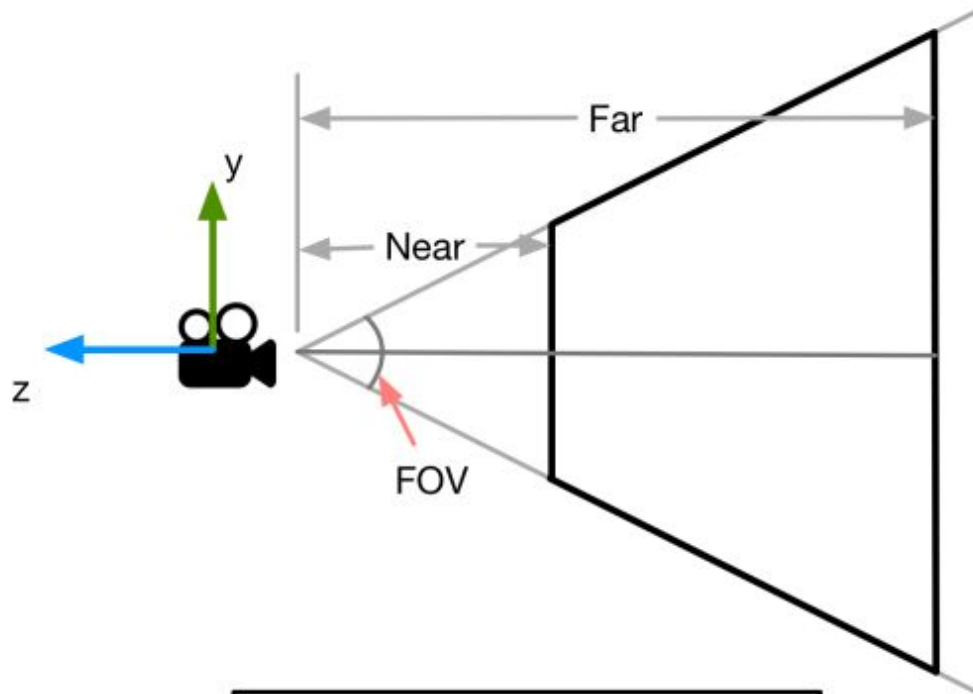
▲4.37 近裁剪平面和远裁剪平面的位置

在4.37节中，我们将讨论如何计算近裁剪平面和远裁剪平面的位置。在4.37节中，我们将讨论如何计算近裁剪平面和远裁剪平面的位置。

在4.37节中，我们将讨论如何计算近裁剪平面和远裁剪平面的位置。

- 在4.37节中，我们将讨论如何计算近裁剪平面和远裁剪平面的位置。在4.37节中，我们将讨论如何计算近裁剪平面和远裁剪平面的位置。

在4.37节中，我们将讨论如何计算近裁剪平面和远裁剪平面的位置。



▲图4.38 摄像机视场角和剪辑平面

图4.38展示了摄像机的Field of View (FOV) 和剪辑平面 (Clipping Planes) 的设置。Near 和 Far 分别表示近剪辑平面和远剪辑平面的距离。

$$\begin{aligned} nearClipPlaneHeight &= 2 \cdot Near \cdot \tan \frac{FOV}{2} \\ farClipPlaneHeight &= 2 \cdot Far \cdot \tan \frac{FOV}{2} \end{aligned}$$

在Unity中，摄像机的View Rect (W, H) 可以通过 `Camera.aspect` 属性来设置，该属性返回了摄像机的Aspect (宽高比)。

$$\begin{aligned} Aspect &= \frac{nearClipPlaneWidth}{nearClipPlaneHeight} \\ Aspect &= \frac{farClipPlaneWidth}{farClipPlaneHeight} \end{aligned}$$

Near Far FOV Aspect

$$M_{frustum} = \begin{bmatrix} \frac{\cot \frac{FOV}{2}}{Aspect} & 0 & 0 & 0 \\ 0 & \cot \frac{FOV}{2} & 0 & 0 \\ 0 & 0 & -\frac{Far+Near}{Far-Near} & -\frac{2 \cdot Near \cdot Far}{Far-Near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Unity
 z
 [-w , w] DirectX z
 [0, w]

Direct3D

$$\mathbf{P}_{clip} = \mathbf{M}_{frustum} \mathbf{P}_{view}$$

$$= \begin{bmatrix} \frac{\cot \frac{FoV}{2}}{Aspect} & 0 & 0 & 0 \\ 0 & \cot \frac{FoV}{2} & 0 & 0 \\ 0 & 0 & -\frac{Far+Near}{Far-Near} & -\frac{2 \cdot Near \cdot Far}{Far-Near} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} x \frac{\cot \frac{FoV}{2}}{Aspect} \\ y \cot \frac{FoV}{2} \\ -z \frac{Far+Near}{Far-Near} - \frac{2 \cdot Near \cdot Far}{Far-Near} \\ -z \end{bmatrix}$$

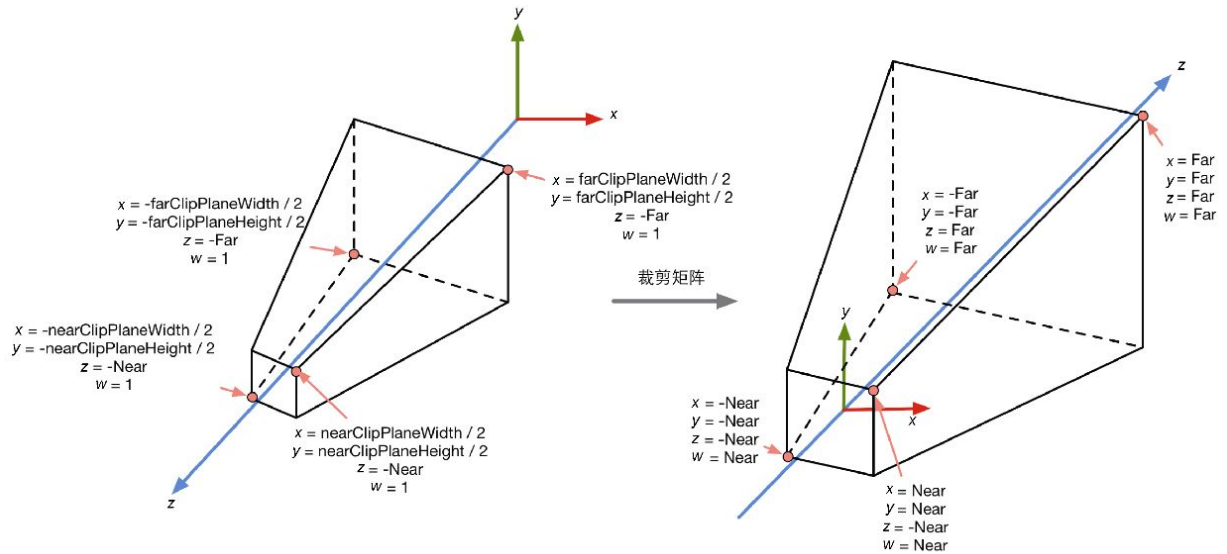
x y z
 w 1 z
 ,

$$-w \leq x \leq w$$

$$-w \leq y \leq w$$

$$-w \leq z \leq w$$

4.39



▲4.39 裁剪矩阵

4.39

2

Unity Camera Game 4.40

Near, Far, Size, Aspect
 00

$$M_{ortho} = \begin{bmatrix} \frac{1}{Aspect \cdot Size} & 0 & 0 & 0 \\ 0 & \frac{1}{Size} & 0 & 0 \\ 0 & 0 & -\frac{2}{Far - Near} & -\frac{Far + Near}{Far - Near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Unity
 0000

0000000000000000

$$\mathbf{P}_{clip} = \mathbf{M}_{ortho} \mathbf{P}_{view}$$

$$= \begin{bmatrix} \frac{1}{Aspect \cdot Size} & 0 & 0 & 0 \\ 0 & \frac{1}{Size} & 0 & 0 \\ 0 & 0 & -\frac{2}{Far - Near} & -\frac{Far + Near}{Far - Near} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{x}{Aspect \cdot Size} \\ \frac{y}{Size} \\ -\frac{2z}{Far - Near} - \frac{Far + Near}{Far - Near} \\ 1 \end{bmatrix}$$

w 00001
 [0 0 -1 0]
 [0 0 0 1]

4.41

00

4.6.6
 (9, 8.84, -27.31)

The diagram illustrates the clipping process in a 3D coordinate system. It shows a cube being transformed by a clipping matrix to fit within a normalized viewing frustum.

Left Cube (World Space):

- Top-back corner: $x = -\text{farClipPlaneWidth} / 2$, $y = -\text{farClipPlaneHeight} / 2$, $z = -\text{Far}$, $w = 1$
- Top-front corner: $x = \text{farClipPlaneWidth} / 2$, $y = \text{farClipPlaneHeight} / 2$, $z = -\text{Far}$, $w = 1$
- Bottom-back corner: $x = -\text{nearClipPlaneWidth} / 2$, $y = -\text{nearClipPlaneHeight} / 2$, $z = -\text{Near}$, $w = 1$
- Bottom-front corner: $x = \text{nearClipPlaneWidth} / 2$, $y = \text{nearClipPlaneHeight} / 2$, $z = -\text{Near}$, $w = 1$

Transformation: 裁剪矩阵 (Clipping Matrix)

Right Cube (Normalized Space):

- Top-back corner: $x = -1$, $y = -1$, $z = 1$, $w = 1$
- Top-front corner: $x = 1$, $y = 1$, $z = 1$, $w = 1$
- Bottom-back corner: $x = -1$, $y = -1$, $z = -1$, $w = 1$
- Bottom-front corner: $x = 1$, $y = 1$, $z = -1$, $w = 1$

A screenshot of a Unity 3D environment. The scene features a flat green ground plane under a light blue sky. Several yellow, cartoonish cows with large noses and small horns are scattered across the field. In the bottom-left corner, a grey UI panel displays camera settings: 'Projection' is set to 'Perspective', 'Field of View' is 60, 'Clipping Planes' shows 'Near' at 5 and 'Far' at 40, and 'Viewport Rect' is defined by 'X 0', 'Y 0', 'W 1', and 'H 1'. The top of the window has a title bar with '4:3' and buttons for 'Maximize on Play', 'Mute audio', 'Stats', and 'Gizmos'.

▲4.42 正交投影变换矩阵

已知视场角 $FOV = 60^\circ$ ，近裁剪面 $Near = 5$ ，远裁剪面 $Far = 40$ ，纵横比 $Aspect = 4/3 = 1.333$ ，求正交投影变换矩阵。

$$M_{frustum} = \begin{bmatrix} \frac{\cot \frac{FOV}{2}}{Aspect} & 0 & 0 & 0 \\ 0 & \cot \frac{FOV}{2} & 0 & 0 \\ 0 & 0 & -\frac{Far+Near}{Far-Near} & -\frac{2 \cdot Near \cdot Far}{Far-Near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1.299 & 0 & 0 & 0 \\ 0 & 1.732 & 0 & 0 \\ 0 & 0 & -1.286 & -11.429 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

将视图中的点变换到裁剪空间。

$$\mathbf{P}_{clip} = \mathbf{M}_{frustum} \mathbf{P}_{view}$$

$$= \begin{bmatrix} 1.299 & 0 & 0 & 0 \\ 0 & 1.732 & 0 & 0 \\ 0 & 0 & -1.286 & -11.429 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 9 \\ 8.84 \\ -27.31 \\ 1 \end{bmatrix} = \begin{bmatrix} 11.691 \\ 15.311 \\ 23.692 \\ 27.31 \end{bmatrix}$$

将裁剪空间中的点变换到归一化设备坐标空间——(11.691, 15.311, 23.692, 27.31) 在 Unity 引擎中，裁剪空间的点坐标范围是 $[-1, 1]$ 。

$$-w \leq x \leq w \rightarrow -27.31 \leq 11.691 \leq 27.31$$

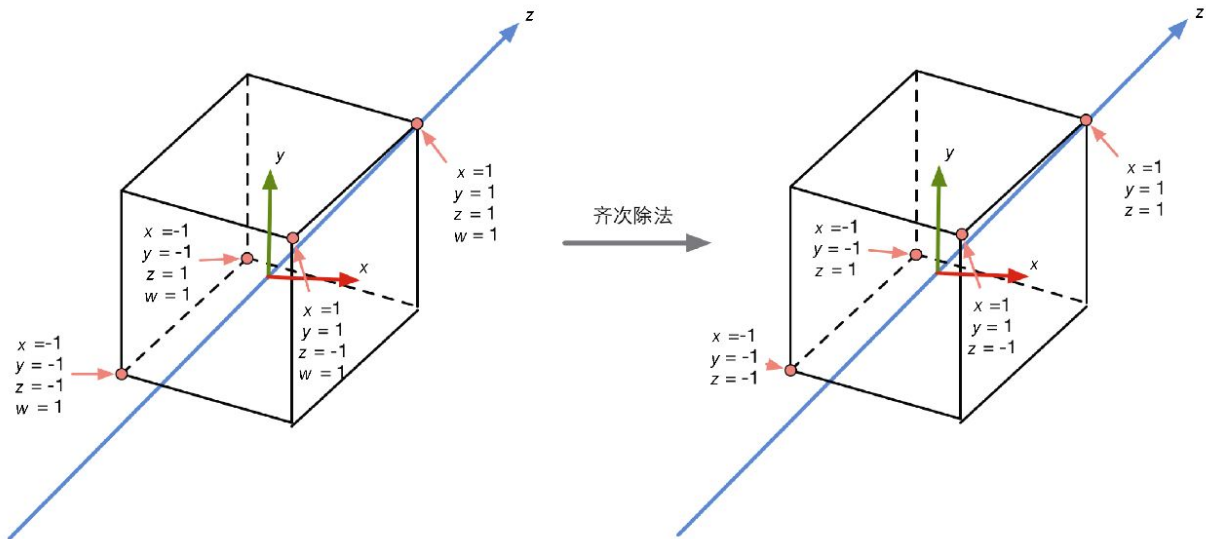
$$-w \leq x \leq y \rightarrow -27.31 \leq 15.311 \leq 27.31$$

$$-w \leq z \leq w \rightarrow -27.31 \leq 23.692 \leq 27.31$$

将归一化设备坐标空间的点变换到屏幕空间。

4.6.8 屏幕空间

将归一化设备坐标空间的点变换到屏幕空间。屏幕空间的点坐标范围是 $[0, 1]$ ，通常称为 **screen space**。



▲4.44 齐次除法

齐次除法是将归一化坐标转换为NDC坐标的过程。在Unity中，归一化坐标的范围是[-1, 1]，而NDC坐标的范围是[0, 1]。

Unity中，归一化坐标的范围是[-1, 1]，而NDC坐标的范围是[0, 1]。Unity中，归一化坐标的范围是[-1, 1]，而NDC坐标的范围是[0, 1]。

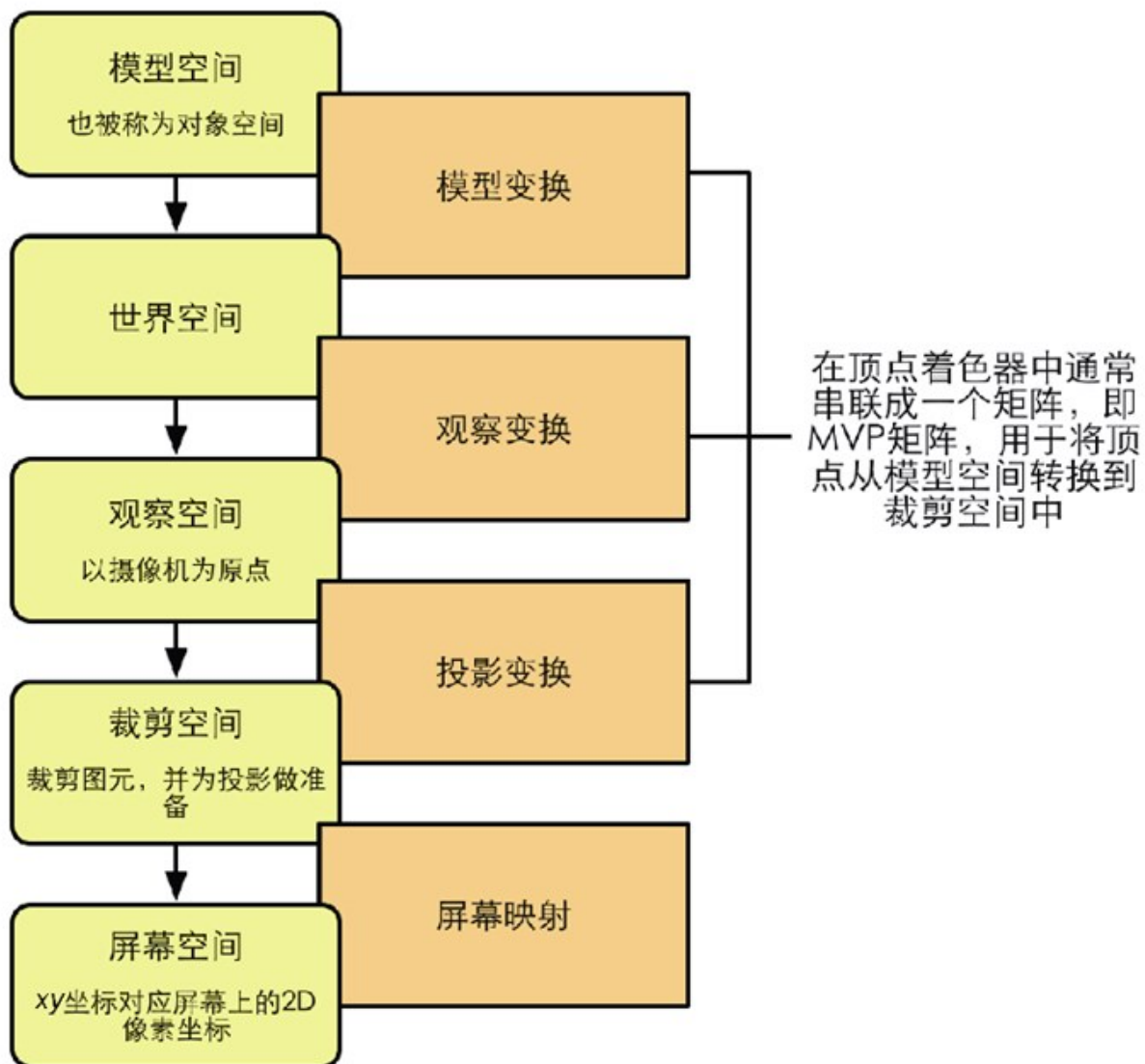
齐次除法的公式如下：

$$\begin{aligned} screen_x &= \frac{clip_x \cdot pixelWidth}{2 \cdot clip_w} + \frac{pixelWidth}{2} \\ screen_y &= \frac{clip_y \cdot pixelHeight}{2 \cdot clip_w} + \frac{pixelHeight}{2} \end{aligned}$$

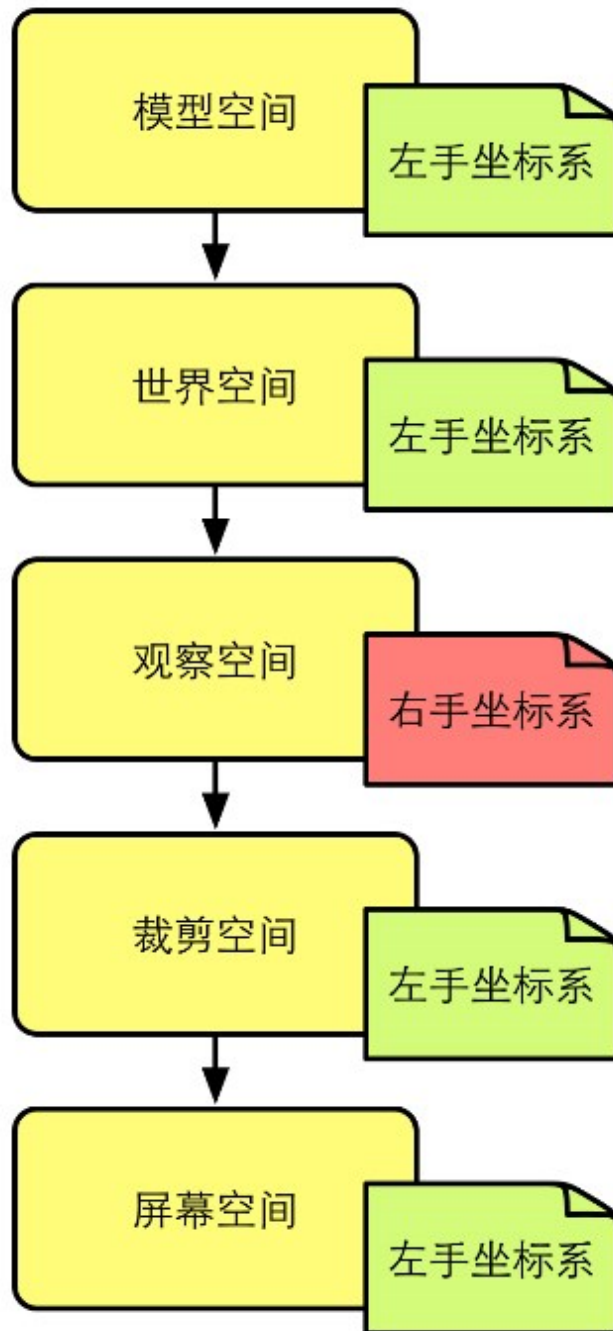
齐次除法是将归一化坐标转换为NDC坐标的过程。在Unity中，归一化坐标的范围是[-1, 1]，而NDC坐标的范围是[0, 1]。齐次除法的公式如下：

Unity中，归一化坐标的范围是[-1, 1]，而NDC坐标的范围是[0, 1]。Unity中，归一化坐标的范围是[-1, 1]，而NDC坐标的范围是[0, 1]。

齐次除法是将归一化坐标转换为NDC坐标的过程。在Unity中，归一化坐标的范围是[-1, 1]，而NDC坐标的范围是[0, 1]。齐次除法的公式如下：



▲ 4.45 模型变换与观察变换



▲图4.46 Unity坐标空间转换

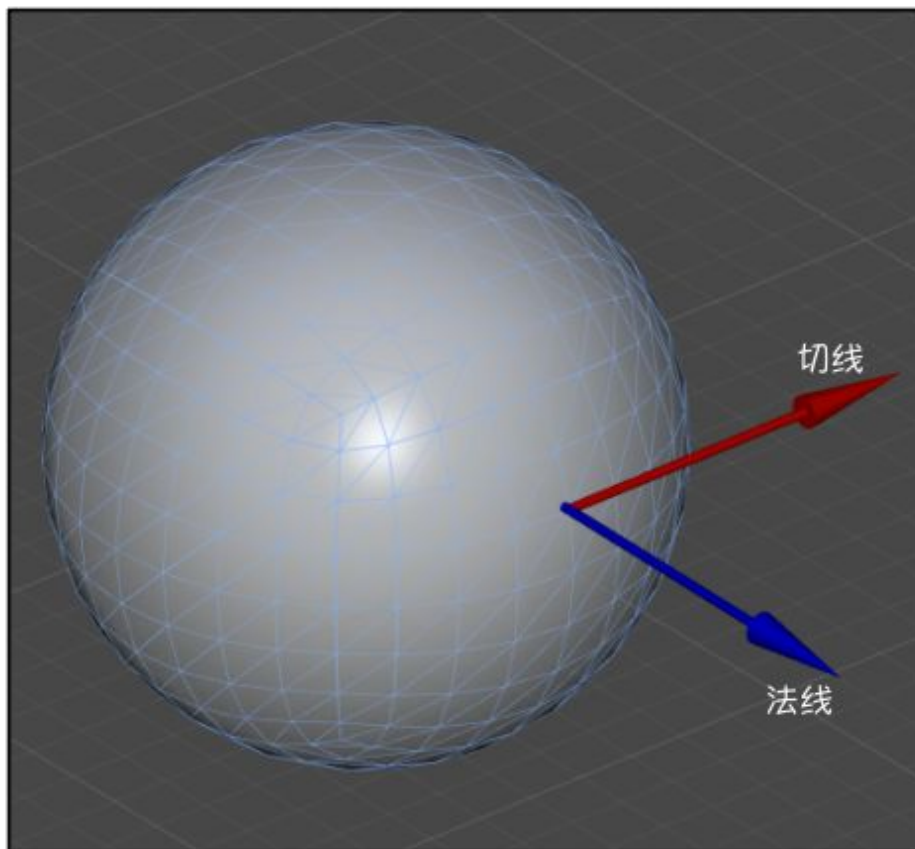
4.7 变换

变换是图形学中的一个重要概念。

一个 **normal** 向量通常是一个 **normal vector**，它垂直于一个平面。在图形学中，normal 向量用于计算光照效果。例如，在渲染一个立方体时，我们需要为每个面指定一个 normal 向量，以便根据光源的方向计算出该面的亮度。

变换矩阵通常是一个 4×4 或 3×3 的矩阵 $\mathbf{M}_{A \rightarrow B}$ ，它表示从空间 A 到空间 B 的变换。例如，平移、旋转和缩放都可以用变换矩阵来表示。

一个 **tangent** 向量通常是一个 **tangent vector**，它与一个平面的法向量垂直。在图形学中，tangent 向量用于计算纹理映射。例如，在渲染一个球体时，我们需要为球面上的每个点指定一个 tangent 向量，以便根据纹理的坐标计算出该点的颜色。

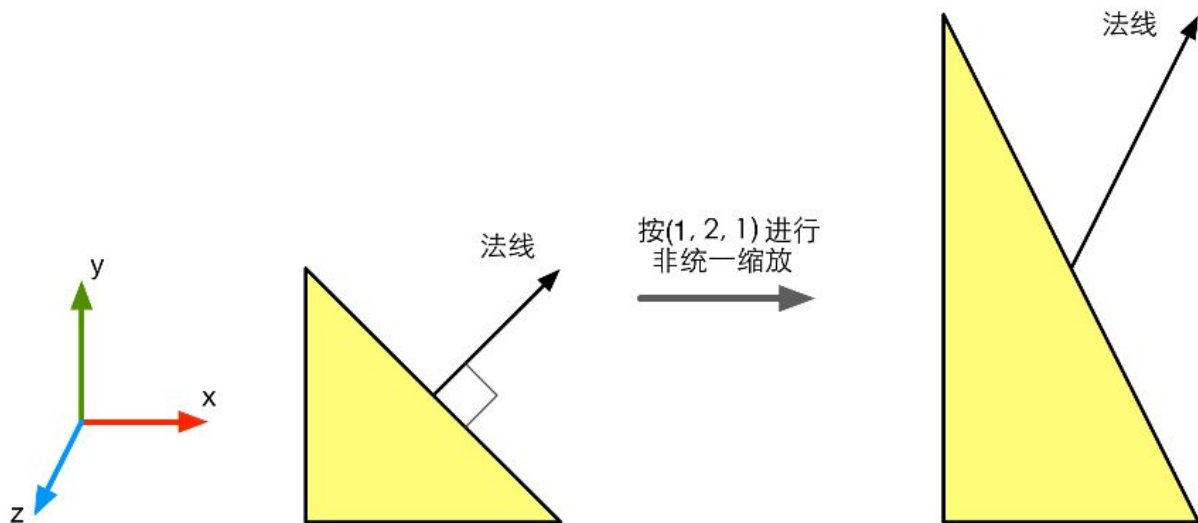


▲图4.47 切线及法线

假设在物体A上的点A处有一个切向量 $\mathbf{T}_A$ ，在物体B上的点B处有一个切向量 $\mathbf{T}_B$ 。如果物体A和物体B之间存在一个变换 $\mathbf{M}_{A \rightarrow B}$ ，那么物体A上的切向量 $\mathbf{T}_A$ 在物体B上的切向量 $\mathbf{T}_B$ 可以通过以下公式计算：

$$\mathbf{T}_B = \mathbf{M}_{A \rightarrow B} \mathbf{T}_A$$

其中 $\mathbf{T}_A$ 和 $\mathbf{T}_B$ 都是3x1的列向量， $\mathbf{M}_{A \rightarrow B}$ 是一个3x3的变换矩阵。图4.48展示了切向量的变换过程。



▲4.48 在刚体运动中，刚体内部任意一点A的加速度与刚体内部任意一点B的加速度之间的关系为

在刚体运动中，刚体内部任意一点A的加速度与刚体内部任意一点B的加速度之间的关系为
 $\mathbf{T}_A$ 和 $\mathbf{T}_B$ 是刚体内部任意一点A和B的加速度向量， $\mathbf{N}_A$ 和 $\mathbf{N}_B$ 是刚体内部任意一点A和B的法向量， $\mathbf{T}_A \cdot \mathbf{N}_A = 0$ ， $\mathbf{T}_B \cdot \mathbf{N}_B = 0$ ， $\mathbf{M}_{A \rightarrow B}$ 是刚体内部任意一点A和B的惯性张量， $\mathbf{G}$ 是刚体内部任意一点A和B的引力张量， $\mathbf{N}_A$ 是刚体内部任意一点A的法向量， $\mathbf{N}_B$ 是刚体内部任意一点B的法向量。

$$\mathbf{T}_B \cdot \mathbf{N}_B = (\mathbf{M}_{A \rightarrow B} \mathbf{T}_A) \cdot (\mathbf{G} \mathbf{N}_A) = 0$$

在刚体运动中，刚体内部任意一点A的加速度与刚体内部任意一点B的加速度之间的关系为

$$(M_{A \rightarrow B} T_A) \cdot (G N_A) = (M_{A \rightarrow B} T_A)^T (G N_A) = T_A^T M_{A \rightarrow B}^T G N_A = T_A^T (M_{A \rightarrow B}^T G) N_A = 0$$

$$\mathbf{T}_A \cdot \mathbf{N}_A = 0 \implies M_{A \rightarrow B}^T G = I \implies G = (M_{A \rightarrow B}^T)^{-1} = (M_{A \rightarrow B}^{-1})^T$$

$$\mathbf{M}_{A \rightarrow B} \implies M_{A \rightarrow B}^{-1} = M_{A \rightarrow B}^T \implies (M_{A \rightarrow B}^T)^{-1} = M_{A \rightarrow B} \implies (M_{A \rightarrow B}^T)^{-1} = \frac{1}{k} M_{A \rightarrow B}$$

4.8 Unity Shader

Unity Shader
Unity
UnityShaderVariables.cginc

4.8.1

4.2 Unity 5.2
float4x4

Unity

Unity
5.2 4.x

4.2 Unity

UNITY_MATRIX_MVP	
UNITY_MATRIX_MV	
UNITY_MATRIX_V	
UNITY_MATRIX_P	
UNITY_MATRIX_VP	
UNITY_MATRIX_T_MV	UNITY_MATRIX_MV

——

```
// transposeUNITY_MATRIX_IT_MV
// UNITY_MATRIX_MV
// 
float4 modelPos = mul(transpose(UNITY_MATRIX_IT_MV), viewPos);

// transposemul
// 
float4 modelPos = mul(viewPos, UNITY_MATRIX_IT_MV);
```

mul4.9.2

4.8.2

Unity
Camera4.3Unity 5.2

4.3 Unity

_WorldSpaceCameraPos	float3	
_ProjectionParams	float4	$x = 1.0 - 1.0$ $y = \text{Near}$ $z = \text{Far}$ $w = 1.0 + 1.0/\text{Far}$ $\text{Near} \cdot \text{Far}$
_ScreenParams	float4	$x = \text{width}$ $y = \text{height}$ $z = 1.0 + 1.0/\text{width}$ $w = 1.0 + 1.0/\text{height}$ $\text{width} \cdot \text{height}$ render target

変数	型	説明
_ZBufferParams	float4	$x = 1 - \text{Far}/\text{Near}$ $y = \text{Far}/\text{Near}$ $z = x/\text{Far}$ $w = y/\text{Far}$ Zバッファの値を13.1で割る
unity_OrthoParams	float4	$x = \text{width}$ $y = \text{height}$ $z = 1.0$ $w = 0.0$ width, height, 1.0, 0.0
unity_CameraProjection	float4x4	カメラの投影行列
unity_CameraInvProjection	float4x4	カメラの逆投影行列
unity_CameraWorldClipPlanes[6]	float4	6つの世界クリップ平面の係数

4.9 矩阵

矩阵是计算机图形学中非常重要的数学工具，用于表示和变换几何对象。在本节中，我们将介绍矩阵的基本概念和运算。

4.9.1 3×3 和 4×4 矩阵

3×3 矩阵通常用于表示二维平面上的变换，如平移、旋转和缩放。4×4 矩阵则用于表示三维空间中的变换，包括上述二维变换以及沿 z 轴的平移和旋转。在 Unity Shader 中，我们通常使用 4×4 矩阵来定义模型的变换。

4.9.2 Cg 中的矩阵

在 Unity Shader 中，Cg 语言提供了对矩阵的内置支持。我们可以使用 `float3x3` 和 `float4x4` 类型来声明矩阵变量。

Cg 中的 `float3x3` 和 `float4x4` 类型分别表示 3x3 和 4x4 的浮点矩阵。我们可以使用 `dot` 函数来计算两个向量的点积，使用 `mul` 函数来计算两个矩阵的乘积。

```
float4 a = float4(1.0, 2.0, 3.0, 4.0);
float4 b = float4(1.0, 2.0, 3.0, 4.0);
// 计算点积
float result = dot(a, b);
```

在 Cg 中，我们可以使用 `mul` 函数来计算两个矩阵的乘积。例如，我们可以将模型矩阵 `M` 与向量 `v` 相乘，得到变换后的向量。

```
float4 v = float4(1.0, 2.0, 3.0, 4.0);
float4x4 M = float4x4(1.0, 0.0, 0.0, 0.0,
                      0.0, 2.0, 0.0, 0.0,
                      0.0, 0.0, 3.0, 0.0,
                      0.0, 0.0, 0.0, 4.0);
// 计算 v * M
float4 column_mul_result = mul(M, v);
// 计算 v * M^T
float4 row_mul_result = mul(v, M);
// 验证 column_mul_result == row_mul_result
// mul(M,v) == mul(v, transpose(M))
```

```
// mul(v,M) == mul(tranpose(M), v)
```

Unity 的 UNITY_MATRIX_MVP 是 4x4 的矩阵，由 12 个 float 组成，表示模型在视图空间中的位置、旋转和缩放。

Cg 的 float4x4 矩阵，表示模型在视图空间中的位置、旋转和缩放。Unity 的 UNITY_MATRIX_MVP 是 4x4 的矩阵，由 12 个 float 组成，表示模型在视图空间中的位置、旋转和缩放。

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Unity 的 UNITY_MATRIX_MVP 是 4x4 的矩阵，由 12 个 float 组成，表示模型在视图空间中的位置、旋转和缩放。

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

Cg 的 float4x4 矩阵，表示模型在视图空间中的位置、旋转和缩放。Unity 的 UNITY_MATRIX_MVP 是 4x4 的矩阵，由 12 个 float 组成，表示模型在视图空间中的位置、旋转和缩放。

Unity 的 UNITY_MATRIX_MVP 是 4x4 的矩阵，由 12 个 float 组成，表示模型在视图空间中的位置、旋转和缩放。

```
// 定义 3x3 矩阵 M
float3x3 M = float3x3(1.0, 2.0, 3.0,
                      4.0, 5.0, 6.0,
                      7.0, 8.0, 9.0);
// 获取 M 的第一行 (1.0, 2.0, 3.0)
float3 row = M[0];

// 获取 M 的第二列的第一个元素 4.0
float ele = M[1][0];
```

Unity Shader 中的 Cg 语言，用于编写 GPU 上的着色器程序。Cg 语言是 C 语言的子集，用于编写 GPU 上的着色器程序。

Unity API Unity Matrix4x4
Unity Shader

4.9.3 Unity ComputeScreenPos/VPOS/WPOS

4.6.8 Shader

/

VPOS **WPOS** 5.4
VPOS HLSL WPOS Cg Unity Shader
HLSL/Cg /
VPOS
WPOS

```
fixed4 frag(float4 sp : VPOS) : SV_Target {  
    //   
    return fixed4(sp.xy/_ScreenParams.xy,0.0,1.0);  
}
```

4.49



▲4.49

VPOS/WPOS 是 `float4` 类型，`xy` 表示在 `400 x 300` 的 `x` 轴范围 `[0.5, 400.5]` 和 `y` 轴范围 `[0.5, 300.5]` 上的归一化坐标。在 OpenGL 和 DirectX 10 中，`z` 轴范围是 `[0.5, 1]`，而在 Unity 中，VPOS/WPOS 的 `z` 轴范围是 `[0, 1]`。在 `z` 轴范围 `[0, 1]` 中，`z = 0` 表示近平面，`z = 1` 表示远平面。`w` 表示深度，其范围是 `[1/Near, 1/Far]`，其中 `Near` 和 `Far` 是 Camera 的近平面和远平面。在 `w` 轴范围 `[1, 0]` 中，`w = 1` 表示近平面，`w = 0` 表示远平面。`viewport space` 是指从 `(0, 0)` 到 `(1, 1)` 的 `xy` 平面上的归一化坐标。

在 Unity 中，`ComputeScreenPos` 是 `UnityCG.cginc` 中的一个函数，用于计算屏幕上的位置。它的原型是：
`ComputeScreenPos(float4 pos)`
 它返回一个 `float4` 类型的值，表示在屏幕上的位置。

```
struct vertOut {
    float4 pos:SV_POSITION;
    float4 scrPos : TEXCOORD0;
};

vertOut vert(appdata_base v) {
    vertOut o;
    o.pos = mul (UNITY_MATRIX_MVP, v.vertex);
    // 调用ComputeScreenPos函数计算scrPos
    o.scrPos = ComputeScreenPos(o.pos);
    return o;
}

fixed4 frag(vertOut i) : SV_Target {
    // 使用scrPos.xy除以scrPos.w得到归一化坐标
    float2 wcoord = (i.scrPos.xy/i.scrPos.w);
    return fixed4(wcoord,0.0,1.0);
}
```

在 4.49 节中，我们讨论了如何计算屏幕上的位置。在 3.6.8 节中，我们讨论了如何计算屏幕上的位置。在 3.6.8 节中，我们讨论了如何计算屏幕上的位置。

$$viewport_x = \frac{clip_x}{2 \cdot clip_w} + \frac{1}{2}$$

$$viewport_y = \frac{clip_y}{2 \cdot clip_w} + \frac{1}{2}$$

```
inline float4 ComputeScreenPos (float4 pos) {
    float4 o = pos * 0.5f;
    #if defined(UNITY_HALF_TEXEL_OFFSET)
    o.xy = float2(o.x, o.y*_ProjectionParams.x) + o.w * _ScreenParams.zw;
    #else
    o.xy = float2(o.x, o.y*_ProjectionParams.x) + o.w;
    #endif

    o.zw = pos.zw;
    return o;
}
```

$$\begin{aligned} Output_x &= \frac{clip_x}{2} + \frac{clip_w}{2} \\ Output_y &= \frac{clip_y}{2} + \frac{clip_w}{2} \\ Output_z &= clip_z \\ Output_w &= clip_w \end{aligned}$$

```

    xy = zw * z
    zw = z * w
    z = [-Near, Far] * w
    w = [Near, Far] * z
    z = [-1, 1] * w
    w = 1

```

4.10 □□□□

[illegible]

[1] Fletcher Dunn, Ian Parberry. 3D Math Primer for Graphics and Game Development (2nd Edition). November 2, 2011 by A K Peters/CRC Press□

[2] Eric Lengyel. Mathematics for 3D game programming and computer graphics (3rd Edition). 2011 by Charles River Media

[3] David Eberly. Conversion of Left-Handed Coordinates to Right-Handed Coordinates□

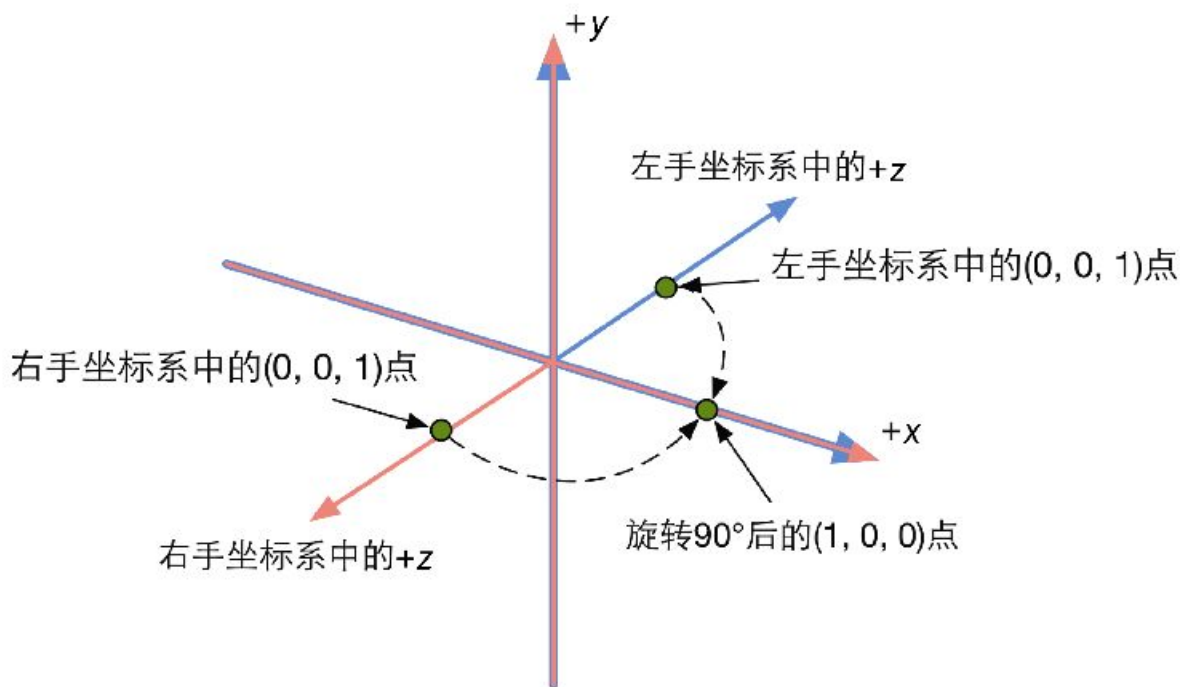
[4] <http://www.humus.name/temp/Linearize%20depth.txt> □

4.11 坐标系

4.2.5

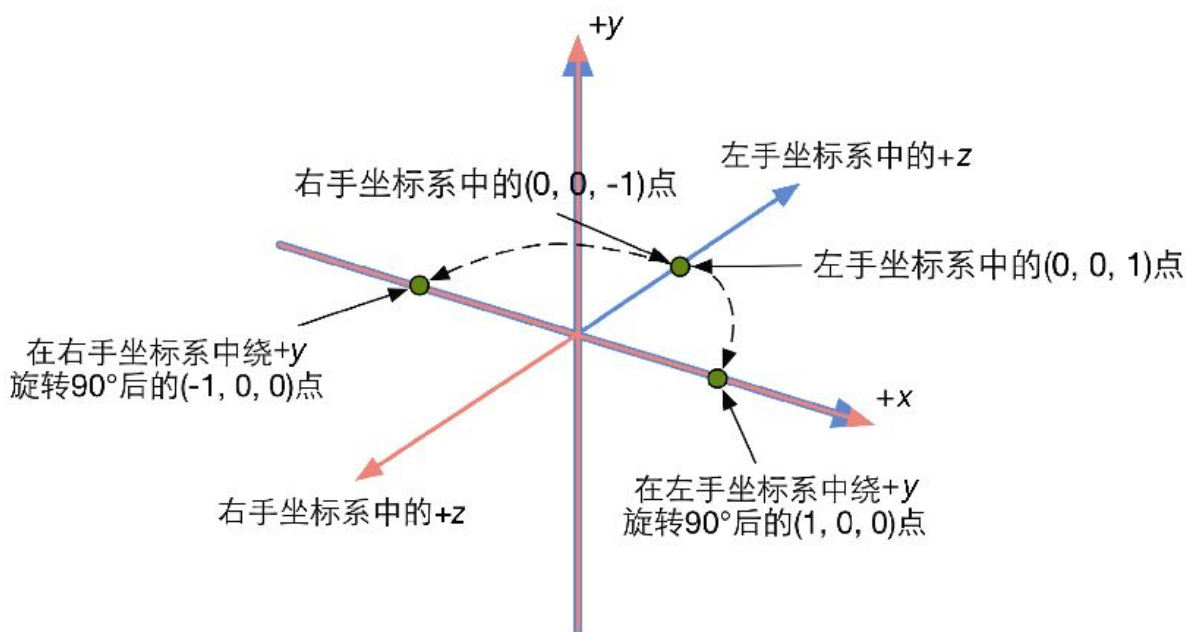
1

2 $(1, 0, 0) \rightarrow (1, 0, 0)$ 右手坐标系中的 $(0, 0, 1)$ 点
左手坐标系中的 $(0, 0, 1)$ 点
旋转 90° 后的 $(1, 0, 0)$ 点
4.50



▲4.50 右手坐标系中的 x 轴 y 轴 z 轴 左手坐标系中的 $(0, 0, 1)$ 点 $(0, 0, 1)$ 点
旋转 90° 后的 $(1, 0, 0)$ 点

右手坐标系中的 $(0, 0, 1)$ 点 $(0, 0, -1)$ 点
左手坐标系中的 $(0, 0, 1)$ 点 $(0, 0, -1)$ 点
旋转 90° 后的 $(1, 0, 0)$ 点
4.51



▲4.51 在右手坐标系中绕+y轴旋转90°后的(-1, 0, 0)点

3-1010Unity在左手坐标系中绕+y轴旋转90°后的(1, 0, 0)点

4.3.3

1

1在右手坐标系中绕+y轴旋转90°后的(-1, 0, 0)点

2

3在左手坐标系中绕+y轴旋转90°后的(1, 0, 0)点

$$\mathbf{a} \times \mathbf{b} = (a_x, a_y, a_z) \times (b_x, b_y, b_z)$$

$$= (a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x)$$

[illegible]

20

$\boxed{1}\sqrt{62} \approx 7.874$

$\mu_2(12.5, 10, 25)$

$\square 3\square(1.5,2)$

$$\boxed{4} \left(\frac{5}{13} \frac{12}{13} \right) \approx 0.3850.923$$

$$\boxed{5} \left(\frac{1}{\sqrt{3}} \frac{1}{\sqrt{3}} \frac{1}{\sqrt{3}} \right) \approx 0.5770.5770.577$$

$$\square 6 \square (10, 9)$$

□7□(-6,1,2)

$$3\sqrt{308} \approx 17.55$$

40

175

□2□13

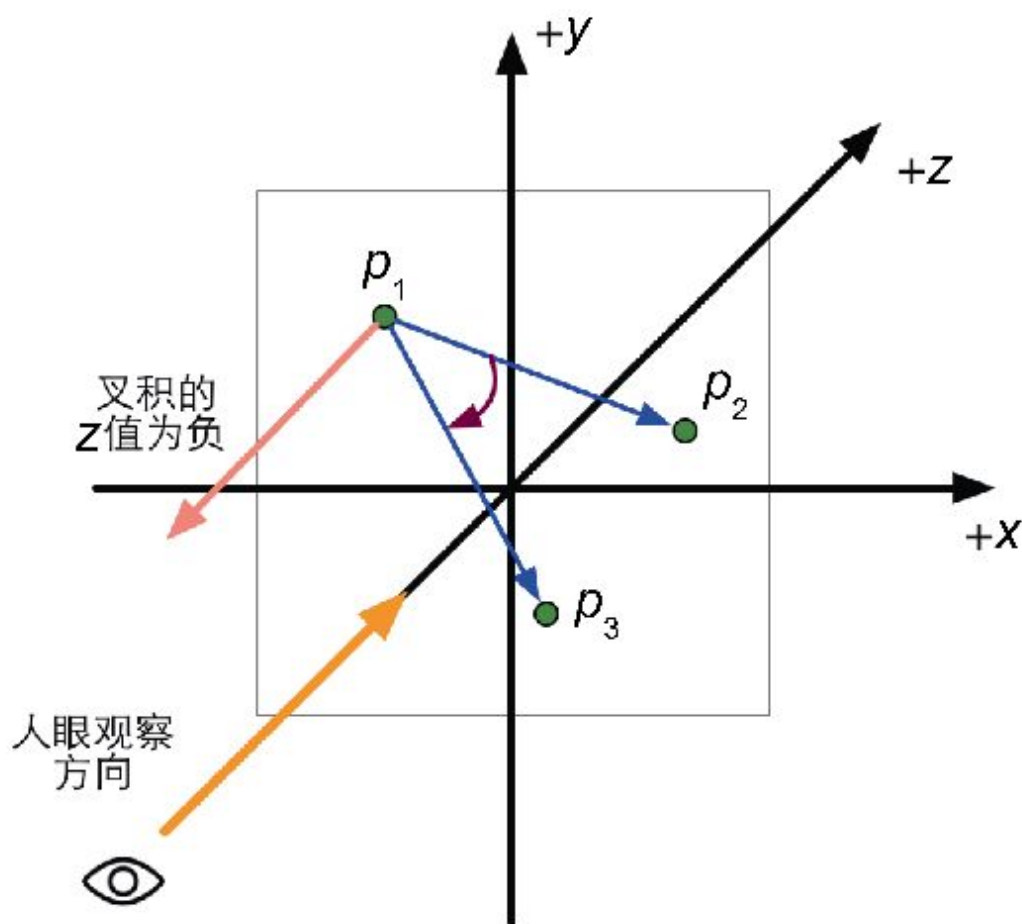
□3□13

$\square 4 \square (-9, -13, 7)$

[illegible]

50

□1□12



▲图4.52 用右手定则判断叉积的方向

4.4.6 例题

1.

$$\text{例1} \quad \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} -1 & 5 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} (1)(-1) + (3)(0) & (1)(5) + (3)(2) \\ (2)(-1) + (4)(0) & (2)(5) + (4)(2) \end{bmatrix} = \begin{bmatrix} -1 & 11 \\ -2 & 18 \end{bmatrix}$$

例2 计算下列两个向量的叉积，并判断其方向。已知向量 $\mathbf{a} = \begin{bmatrix} 1 \\ 5 \\ 6 \end{bmatrix}$ ， $\mathbf{b} = \begin{bmatrix} -2 \\ 1 \\ 0 \end{bmatrix}$ ， $\mathbf{c} = \begin{bmatrix} 3 \\ 4 \\ 3 \end{bmatrix}$ 。

$$\text{解} \quad \begin{bmatrix} 1 & -2 & 3 \\ 5 & 1 & 4 \\ 6 & 0 & 3 \end{bmatrix} \begin{bmatrix} -5 \\ 4 \\ 8 \end{bmatrix} = \begin{bmatrix} (1)(-5) + (-2)(4) + (3)(8) \\ (5)(-5) + (1)(4) + (4)(8) \\ (6)(-5) + (0)(4) + (3)(8) \end{bmatrix} = \begin{bmatrix} 11 \\ 11 \\ -6 \end{bmatrix}$$

2

1. Find the inverse of the matrix $A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ using the formula $A^{-1} = \frac{1}{\det(A)} \text{adj}(A)$.

2. Find the inverse of the matrix $B = \begin{bmatrix} 1 & 2 & 6 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ using the formula $B^{-1} = \frac{1}{\det(B)} \text{adj}(B)$.

3. Find the inverse of the matrix $C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ using the formula $C^{-1} = \frac{1}{\det(C)} \text{adj}(C)$.

3

1. Find the inverse of the matrix $A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ using the formula $A^{-1} = \frac{1}{\det(A)} \text{adj}(A)$.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \\ 6 \end{bmatrix} = \begin{bmatrix} (1)(3) + (0)(2) + (0)(6) \\ (0)(3) + (1)(2) + (0)(6) \\ (3)(0) + (2)(0) + (6)(1) \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \\ 6 \end{bmatrix}$$

Find the inverse of the matrix $B = \begin{bmatrix} 1 & 2 & 6 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ using the formula $B^{-1} = \frac{1}{\det(B)} \text{adj}(B)$.

2. Find the inverse of the matrix $B = \begin{bmatrix} 1 & 2 & 6 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ using the formula $B^{-1} = \frac{1}{\det(B)} \text{adj}(B)$.

$$\begin{bmatrix} 1 & 2 & 6 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \\ 6 \end{bmatrix} = \begin{bmatrix} (1)(3) + (2)(2) + (6)(6) \\ (0)(3) + (1)(2) + (0)(6) \\ (0)(3) + (0)(2) + (1)(6) \end{bmatrix} = \begin{bmatrix} 15 \\ 2 \\ 6 \end{bmatrix}$$

Find the inverse of the matrix $C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ using the formula $C^{-1} = \frac{1}{\det(C)} \text{adj}(C)$.

$$\begin{aligned}
 [3 \quad 2 \quad 6] \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & -3 \\ 0 & 0 & 3 \end{bmatrix}^T &= [3 \quad 2 \quad 6] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & -3 & 3 \end{bmatrix} \\
 &= \begin{bmatrix} (3)(1) + (2)(0) + (6)(2) \\ (3)(0) + (2)(1) + (6)(-3) \\ (3)(0) + (2)(0) + (6)(3) \end{bmatrix} = [15 \quad -16 \quad 18]
 \end{aligned}$$

$$\begin{aligned}
 &\square 3 \square \\
 [3 \quad 2 \quad 6] \begin{bmatrix} 2 & -1 & 3 \\ -1 & 5 & -3 \\ 3 & -3 & 4 \end{bmatrix} &= \begin{bmatrix} (3)(2) + (2)(-1) + (6)(3) \\ (3)(-1) + (2)(5) + (6)(-3) \\ (3)(3) + (2)(-3) + (6)(4) \end{bmatrix} = [22 \quad -11 \quad 27]
 \end{aligned}$$

$$\begin{bmatrix} 2 & -1 & 3 \\ -1 & 5 & -3 \\ 3 & -3 & 4 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \\ 6 \end{bmatrix} = \begin{bmatrix} (2)(3) + (-1)(2) + (3)(6) \\ (-1)(3) + (5)(2) + (-3)(6) \\ (3)(3) + (-3)(2) + (4)(6) \end{bmatrix} = \begin{bmatrix} 22 \\ -11 \\ 27 \end{bmatrix}$$

□□□□□□□□□□(22,-11,27)□□□□□□□□□□□□□□□□□□□□
 □**symmetric matrix**□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

2 背景

Unity Shaderの背景を説明する。Shaderの背景を説明する。Unity Shaderの背景を説明する。9の背景を説明する。Unity Shaderの背景を説明する。

5 Unity Shaderの背景

Unity Shaderの背景を説明する。Unity Shaderの背景を説明する。

6 Unityの背景

Unityの背景を説明する。

7 背景

Unity Shaderの背景を説明する。

8 背景

Unityの背景を説明する。Unityの背景を説明する。

第5章 Unity Shader入门

本章共分2节——第一节介绍Unity Shader入门知识，第二节介绍Unity Shader入门知识。

第一节介绍Unity Shader入门知识，第二节介绍Unity Shader入门知识。5.1节介绍Unity Shader入门知识，5.2节介绍Unity Shader入门知识，5.3节介绍Unity Shader入门知识，5.4节介绍Unity Shader入门知识，5.5节介绍Unity Shader入门知识，5.6节介绍Unity Shader入门知识，5.7节介绍Unity Shader入门知识。

5.1

[illegible]

Mac OS X 10.9.5
 Mac
 OpenGL Windows DirectX
 OpenGL Render Texture (0, 0) DirectX
 (0, 0) 5.6

5.2 Shader/Renderer

RendererはUnity ShaderRenderer/Renderer
を

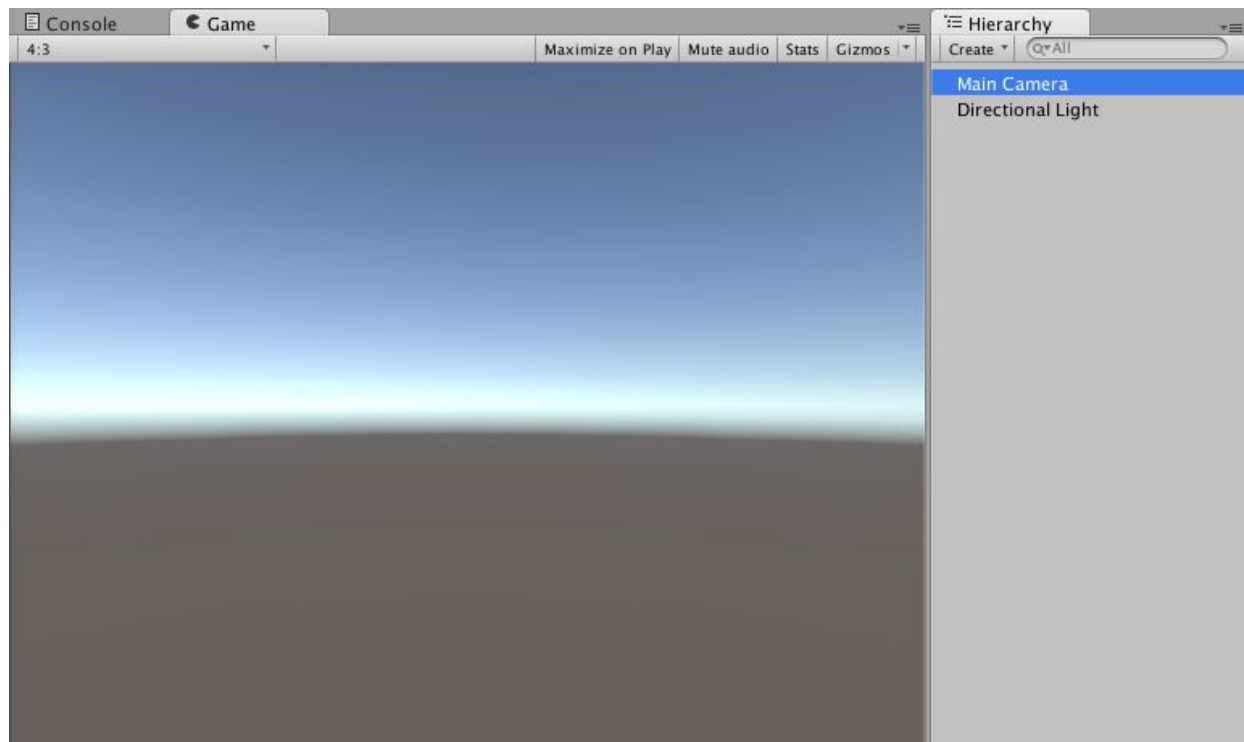
5.2.1 Shader/Renderer

3.3Unity ShaderRendererShader
Properties SubShader Fallback Shader
Renderer/Renderer
Renderer

```
Shader "MyShaderName" {  
    Properties {  
        //  
    }  
    SubShader {  
        // A SubShader  
        Pass {  
            //  
  
            // Cg  
            CGPROGRAM  
            //  
            #pragma vertex vert  
            #pragma fragment frag  
  
            // Cg  
  
            ENDCG  
  
            //  
        }  
        // Pass  
    }  
    SubShader {  
        // B SubShader  
    }  
  
    // SubShaderUnity Shader  
    Fallback "VertexLit"  
}
```

このPassは、シーン全体の背景を生成するためのPassです。
Skybox/背景生成

1. シーンをScene_5_2としてUnity 5.1で実行



▲5.1 Unity 5.1で実行

このPassは、シーン全体の背景を生成するためのPassです。
SkyboxはUnity 5.xで導入された新しいPassです。Unityの
Window -> Lighting -> Skyboxを選択すると、Unity 4.x
のSkyboxを生成できます。

2. Unity ShaderのChapter5-SimpleShader

3. SimpleShaderMatの2つのUnity
Shader

4. Gameの3つのPass

第5章第2节 Unity Shader 的编写

```
Shader "Unity Shaders Book/Chapter 5/Simple Shader" {
    SubShader {
        Pass {
            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

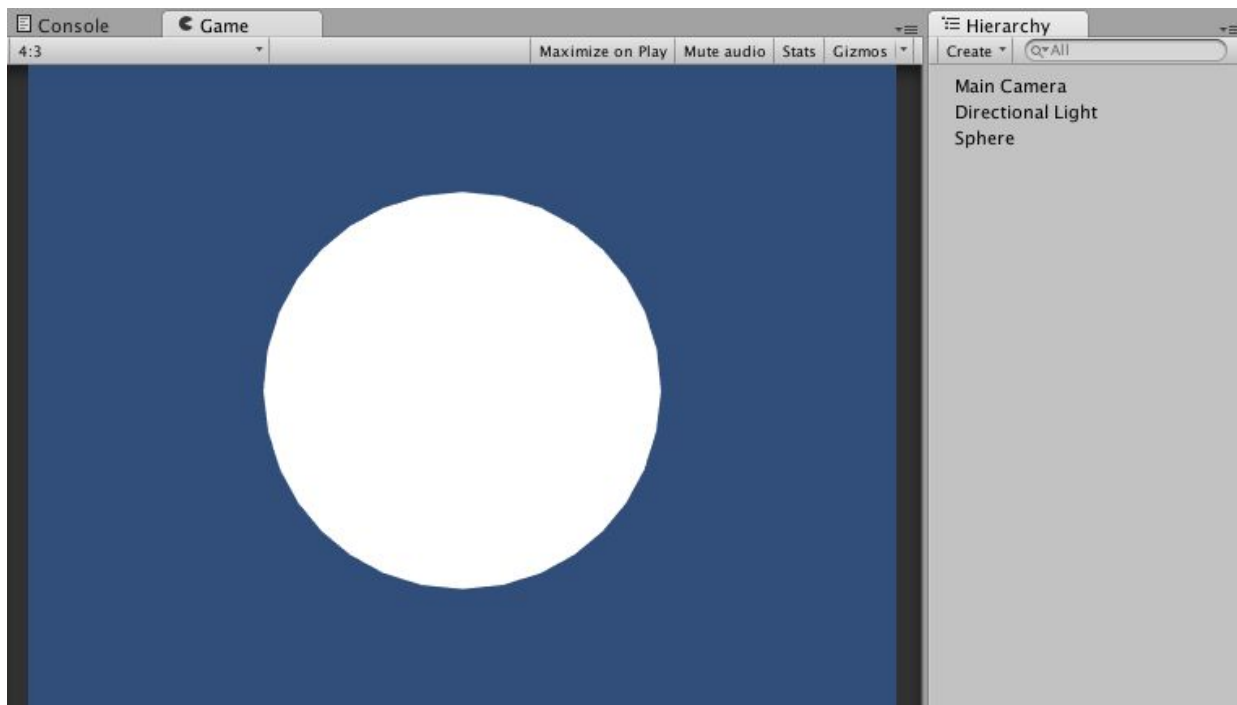
            float4 vert(float4 v : POSITION) : SV_POSITION {
                return mul (UNITY_MATRIX_MVP, v);
            }

            fixed4 frag() : SV_Target {
                return fixed4(1.0, 1.0, 1.0, 1.0);
            }

            ENDCG
        }
    }
}
```

Unity 的 Shader

5.2 节



▲5.2 シェーダー/シェーダープログラム

シェーダー/シェーダープログラム

シェーダーはUnity Shader——“Unity Shaders Book/Chapter 5/Simple Shader”で説明されている。シェーダーはUnity ShaderのPropertiesで定義される。PropertiesはShaderのPropertiesで定義される。

ShaderはSubShader Passで定義される。SubShaderはSubShader Passで定義される。PassはPass Passで定義される。

CGPROGRAM ENDCG CGプログラムはCGプログラムで定義される。

```
#pragma vertex vert
#pragma fragment frag
```

Unityはシェーダープログラムを定義する。

```
#pragma vertex name
#pragma fragment name
```

name 是顶点着色器或片段着色器的名称。顶点着色器使用 `vert` 关键字，片段着色器使用 `frag` 关键字。

顶点着色器使用 `vert` 关键字。

```
float4 vert(float4 v : POSITION) : SV_POSITION {
    return mul (UNITY_MATRIX_MVP, v);
}
```

顶点着色器接收一个 `float4` 类型的输入 `v`，其语义为 `POSITION`。顶点着色器返回一个 `float4` 类型的输出，其语义为 `SV_POSITION`。在 Cg/HLSL 中，`SV_POSITION` 的语义是 `POSITION`。Unity 使用 `SV_POSITION` 语义来指定顶点的位置。在 Unity 4.8 之前，顶点着色器返回的 `SV_POSITION` 值必须乘以 `UNITY_MATRIX_MVP`。从 Unity 4.8 开始，顶点着色器返回的 `SV_POSITION` 值可以直接使用，无需乘以 `UNITY_MATRIX_MVP`。

片段着色器使用 `frag` 关键字。

```
fixed4 frag() : SV_Target {
    return fixed4(1.0, 1.0, 1.0, 1.0);
}
```

片段着色器接收一个 `fixed4` 类型的输入，其语义为 `SV_Target`。片段着色器返回一个 `fixed4` 类型的输出，其语义为 `SV_Target`。在 HLSL 中，`SV_Target` 的语义是 `render target`。Unity 使用 `SV_Target` 语义来指定片段的颜色。在 Unity 4.8 之前，片段着色器返回的 `SV_Target` 值必须乘以 `UNITY_MATRIX_MVP`。从 Unity 4.8 开始，片段着色器返回的 `SV_Target` 值可以直接使用，无需乘以 `UNITY_MATRIX_MVP`。

この関数は、a2vの戻り値をa2vの戻り値としてUnityのNORMAL TEXCOORD0の戻り値としてUnityのPOSITION, TANGENT NORMAL TEXCOORD0 TEXCOORD1 TEXCOORD2 TEXCOORD3 COLORの戻り値として返す。

この関数は、a2vの戻り値をa2vの戻り値として返す。

```
struct StructName {  
    Type Name : Semantic;  
    Type Name : Semantic;  
    .....  
};
```

この関数は、5.4の戻り値をa2vの戻り値として返す。

この関数は、vertの戻り値をa2vの戻り値として返す。

この関数は、a2vの戻り値を返す。

この関数は、applicationの戻り値をvertex shaderのa2vの戻り値として返す。

この関数は、POSITION TANGENT NORMALの戻り値をUnityのMesh RenderのDraw CallのMesh RenderのUnity Shaderの戻り値として返す。

5.2.3 この関数は、a2vの戻り値を返す。

この関数は、a2vの戻り値を返す。

この関数は、a2vの戻り値を返す。

```

Shader "Unity Shaders Book/Chapter 5/Simple Shader" {
    SubShader {
        Pass {
            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            struct a2v {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
                float4 texcoord : TEXCOORD;
            };

            // 顶点着色器输出
            struct v2f {
                // SV_POSITION 顶点位置
                float4 pos : SV_POSITION;
                // COLOR0 顶点颜色
                fixed3 color : COLOR0;
            };

            v2f vert(a2v v) : SV_POSITION {
                // 顶点位置
                v2f o;
                o.pos = mul(UNITY_MATRIX_MVP, v.vertex);
                // v.normal 顶点法线
                // 顶点颜色
                o.color = v.normal * 0.5 + fixed3(0.5, 0.5, 0.5);
                return o;
            }

            fixed4 frag(v2f i) : SV_Target {
                // 顶点颜色
                return fixed4(i.color, 1.0);
            }

            ENDCG
        }
    }
}

```

顶点着色器输出 `v2f` 结构体，包含 `SV_POSITION` 和 `COLOR0` 属性。


```
v2f vert(a2v v) : SV_POSITION {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);
    o.color = v.normal * 0.5 + fixed3(0.5, 0.5, 0.5);
    return o;
}

fixed4 frag(v2f i) : SV_Target {
    fixed3 c = i.color;
    // _Color
    c *= _Color.rgb;
    return fixed4(c, 1.0);
}

ENDCG

}
```

ShaderLabのPropertiesは、_ColorやColor(1.0,1.0,1.0,1.0)やCgのPropertiesを参照する。

ShaderLabのCgの5.1

5.1 ShaderLabのCg

ShaderLab	Cg
ColorVector	float4, half4, fixed4
RangeFloat	float, half, fixed
2D	sampler2D
Cube	samplerCube

ShaderLab	Cg
3D	sampler3D

CG uniform

```
uniform fixed4 _Color;
```

CG uniform
uniform Unity
Shader uniform

5.3 如何集成Unity渲染器

如何集成Unity渲染器/如何集成渲染器/如何集成渲染器
如何集成“渲染器”如何集成渲染器/如何集成渲染器/如何集成渲染器
如何集成Unity渲染器/如何集成渲染器/如何集成渲染器/如何集成渲染器
如何集成Unity Shader如何集成渲染器/如何集成渲染器/如何集成渲染器
如何集成渲染器/如何集成Unity渲染器/如何集成渲染器

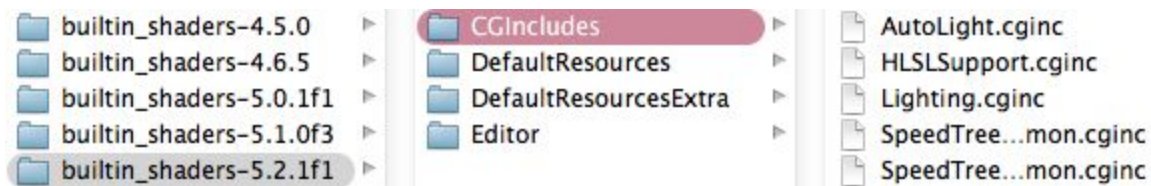
如何集成渲染器/如何集成渲染器

5.3.1 如何集成

如何集成 **include file** 如何集成C++如何集成Unity如何集成
如何集成.cginc如何集成Shader如何集成#include如何集成渲染器/如何集成
如何集成Unity如何集成渲染器/如何集成渲染器/如何集成渲染器

```
CGPROGRAM
// ...
#include "UnityCG.cginc"
// ...
ENDCG
```

如何集成渲染器/如何集成渲染器/如何集成渲染器 <http://unity3d.com/cn/get-unity/download/archive> 如何集成 -> 如何集成 如何集成渲染器5.3如何集成渲染器/如何集成渲染器



▲5.3 Unity如何集成

如何5.3如何集成渲染器/如何集成渲染器/如何集成渲染器/如何集成CGIncludes如何集成
如何集成渲染器/如何集成DefaultResources如何集成渲染器/如何集成Unity
Shader如何集成GUI如何集成Shader如何集成DefaultResourcesExtra如何集成

UnityはUnity Shader Editorで簡単にShaderを作成できる。Unity 5ではStandard Shaderが18のMaterialをサポートしている。CGIncludesはUnityのCGIncludesディレクトリに存在する。

UnityのCGIncludesディレクトリはMacではMac/Unity/Applications/Unity/Unity.app/Contents/CGIncludes、WindowsではUnity/Data/CGIncludesに存在する。

5.2 CGIncludesのディレクトリ

5.2 UnityのCGIncludesディレクトリ

ファイル名	説明
UnityCG.cginc	UnityのCGIncludesディレクトリに存在する。
UnityShaderVariables.cginc	Unity ShaderのCGIncludesディレクトリに存在する。UNITY_MATRIX_MVPを定義する。
Lighting.cginc	Unity ShaderのCGIncludesディレクトリに存在する。Surface ShaderのCGIncludesディレクトリに存在する。
HLSLSupport.cginc	Unity ShaderのCGIncludesディレクトリに存在する。HLSLのCGIncludesディレクトリに存在する。

Unity ShaderのCGIncludesディレクトリには、`#include`で他のCGIncludesファイルを読み込むことができる。UnityShaderVariables.cgincはUnity ShaderのCGIncludesディレクトリに存在する。UNITY_MATRIX_MVPはUnity ShaderのCGIncludesディレクトリに存在する。5.2ではUnity 5のCGIncludesディレクトリにUnityStandardBRDF.cgincとUnityStandardCore.cgincが存在する。18のMaterialをサポートしている。

UnityCG.cgincの関数と定数の一覧をまとめたドキュメントを公開しました。
このドキュメントは、UnityCG.cgincの関数と定数の一覧をまとめたドキュメントです。
5.3のバージョンのUnityCG.cgincの関数と定数の一覧をまとめたドキュメントです。

5.3 UnityCG.cgincの関数と定数の一覧

関数名	戻り値	説明
appdata_base	float3	アプリケーションのベースディレクトリ
appdata_tan	float3	アプリケーションのタンジェントディレクトリ
appdata_full	float3	アプリケーションのフルディレクトリ
appdata_img	float3	アプリケーションのイメージディレクトリ
v2f_img	float3	アプリケーションのv2fディレクトリ

UnityCG.cgincの関数と定数の一覧をまとめたドキュメントを公開しました。
Unityのバージョン5.3のUnityCG.cgincの関数と定数の一覧をまとめたドキュメントです。

UnityCG.cgincの関数と定数の一覧をまとめたドキュメントを公開しました。
5.4のバージョンのUnityCG.cgincの関数と定数の一覧をまとめたドキュメントです。

5.4 UnityCG.cgincの関数と定数の一覧

関数名	戻り値
float3 WorldSpaceViewDir (float4 v)	ワールド空間での視線方向ベクトル

変数名	説明
float3 ObjSpaceViewDir (float4 v)	オブジェクト空間からの視点方向ベクトル
float3 WorldSpaceLightDir (float4 v)	ワールド空間からの照明方向ベクトル
float3 ObjSpaceLightDir (float4 v)	オブジェクト空間からの照明方向ベクトル
float3 UnityObjectToWorldNormal (float3 norm)	オブジェクト空間からの法線ベクトル
float3 UnityObjectToWorldDir (float3 dir)	オブジェクト空間からの視点方向ベクトル
float3 UnityWorldToObjectDir(float3 dir)	ワールド空間からの視点方向ベクトル

UnityCG.cgincは、UnityObjectToWorldDir、UnityWorldToObjectDir、UnityCG.cgincは、UnityCG.cginc

5.3.2 変数

4.8は、Unity、UnityShader Variables.cginc、Lighting.cginc、AutoLight.cginc

5.4 UnityとCg/HLSL

Shaderは5.2でSV_POSITION POSITION COLOR 0

5.4.1

Cg/HLSL semantics DirectX [https://msdn.microsoft.com/en-us/library/windows/desktop/bb509647\(v=vs.85\).aspx#VS.aspx#VS](https://msdn.microsoft.com/en-us/library/windows/desktop/bb509647(v=vs.85).aspx#VS.aspx#VS) Shader Cg/HLSL Shader Unity

COLOR 0 color color Shader

Unity a2v TEXCOORD 0 texcoord Unity TEXCOORD 0 texcoord TEXCOORD 0 a2v v2v a2f TEXCOORD 0 v2f TEXCOORD 0

DirectX 10 system-value semantics SV SV system-value SV_POSITION pos pos SV_POSITION Shader Shader POSITION

SV_POSITION の型は *SV_POSITION* DirectX 10 の型は *POSITION* PS4 の型は *SV_POSITION* Shader の型は *COLOR* *SV_Target* Shader の型は *SV* 5.6 の型は

5.4.2 Unity

5.5 Unity SV Unity

5.5 Unity

名前	型
POSITION	float4
NORMAL	float3
TANGENT	float4
TEXCOORDn n TEXCOORD0 TEXCOORD1	TEXCOORD0 float2 float4
COLOR	fixed4 float4

TEXCOORDn n Shader Model Shader Model 2 Unity Shader Model Shader Model 3 n 8 Shader Model 4 Shader Model 5 n 16 TEXCOORD0 TEXCOORD1 Unity appdata_full 6

5.6 Unity

5.6 5.6 Unity

SV_POSITION	POSITION
COLOR0	
COLOR1	
TEXCOORD0	
TEXCOORD7	

SV_POSITION POSITION
 COLOR0
 COLOR1
 TEXCOORD0 TEXCOORD7

5.7 5.7 Unity

5.7 5.7 Unity

SV_Target	render target

5.4.3 5.4.3

fixed2 float float4
 fixed4


```
struct v2f {
    float4 pos : SV_POSITION;
    fixed3 color0 : COLOR0;
    fixed4 color1 : COLOR1;
    half value0 : TEXCOORD0;
    float2 value1 : TEXCOORD1;
};
```

5.7.1

4 float float3x4 float4x4 float4x4 float4 float4

5.5 □□□□□□Debug

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □

>

> `00000000000000000000000000000000bug00`

```

00000000 debug Shader
Shader-----
Shader

```

Unity Shader

5.5.1 □□□□□□

```

    [false-color image] [
    [true-color image] [
    Shader

```

[illegible]

```

    [0, 1] Shader
    1 1 0 0
    0 1

```

[illegible][illegible]

```
Shader "Unity Shaders Book/Chapter 5/False Color" {
    SubShader {
        Pass {
            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            struct v2f {
                float4 pos : SV_POSITION;
                fixed4 color : COLOR0;
            };
        }
    }
}
```

```

};

v2f vert(appdata_full v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    // 法線ベクトル
    o.color = fixed4(v.normal * 0.5 + fixed3(0.5, 0.5, 0.5), 1.0);

    // 接線ベクトル
    o.color = fixed4(v.tangent.xyz * 0.5 + fixed3(0.5, 0.5, 0.5), 1.0);

    // バイノーマルベクトル
    fixed3 binormal = cross(v.normal, v.tangent.xyz) * v.tangent.w;
    o.color = fixed4(binormal * 0.5 + fixed3(0.5, 0.5, 0.5), 1.0);

    // テクスチャ座標
    o.color = fixed4(v.texcoord.xy, 0.0, 1.0);

    // テクスチャ座標1
    o.color = fixed4(v.texcoord1.xy, 0.0, 1.0);

    // テクスチャ座標の飽和
    o.color = frac(v.texcoord);
    if (any(saturate(v.texcoord) - v.texcoord)) {
        o.color.b = 0.5;
    }
    o.color.a = 1.0;

    // テクスチャ座標1の飽和
    o.color = frac(v.texcoord1);
    if (any(saturate(v.texcoord1) - v.texcoord1)) {
        o.color.b = 0.5;
    }
    o.color.a = 1.0;

    // カラー
    //o.color = v.color;

    return o;
}

fixed4 frag(v2f i): SV_Target {
    return i.color;
}

ENDCG
}
}
}

```

法線ベクトル、接線ベクトル、バイノーマルベクトルを計算する。——*appdata_full* はUnity 5.3から追加された構造体。
 参照: `UnityCG.cginc`

```

struct appdata_full {
    float4 vertex : POSITION;

```

```

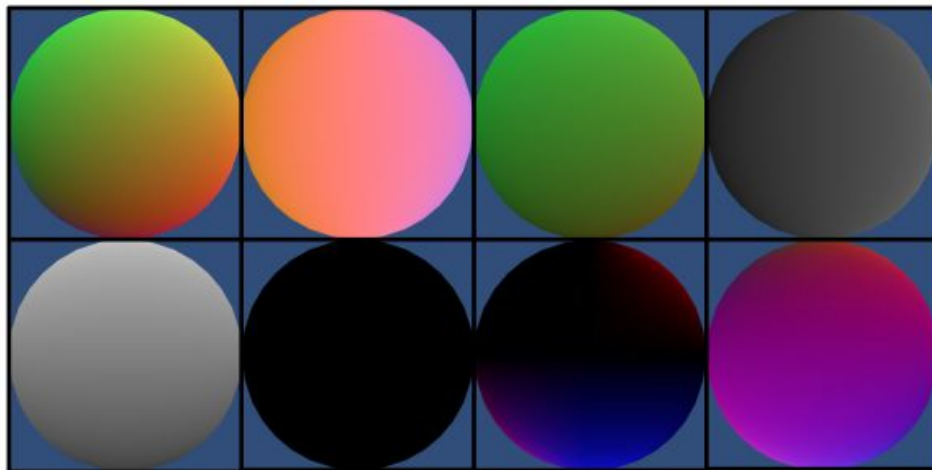
float4 tangent : TANGENT;
float3 normal : NORMAL;
float4 texcoord : TEXCOORD0;
float4 texcoord1 : TEXCOORD1;
float4 texcoord2 : TEXCOORD2;
float4 texcoord3 : TEXCOORD3;
#if defined(SHADER_API_XBOX360)
half4 texcoord4 : TEXCOORD4;
half4 texcoord5 : TEXCOORD5;
#endif
fixed4 color : COLOR;
};

```

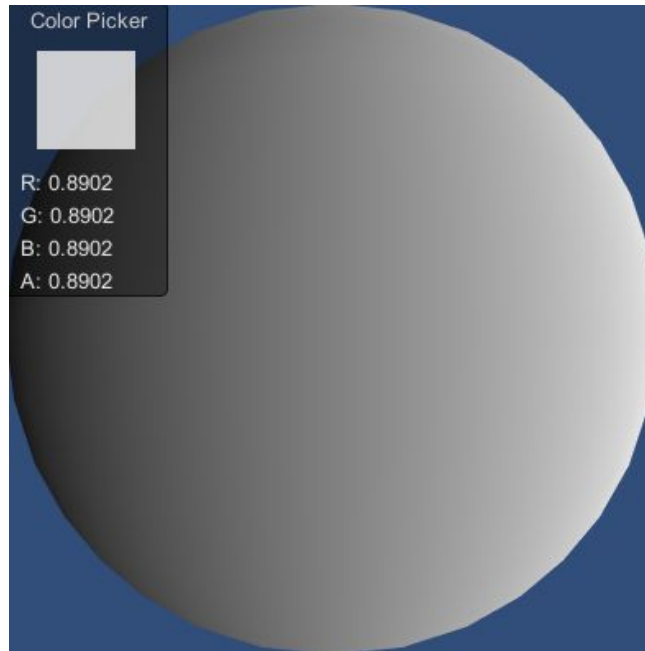
appdata_full

——v2f color
5.4
Unity

RGBA
Assets -> Scripts -> Chapter 5 ->
ColorPicker.cs 5.5



▲5.4 Unity Shader



▲5.5 色相空間の表現

5.5.2 色相空間Visual Studio

Windows/Mac両プラットフォームでVisual Studio/Windows/Mac両プラットフォームでVisual Studio 2012/Unity Shaderのデバッグ——**Graphics Debugger**

Graphics Debuggerは、UnityのShaderをDirectX 11のShaderとして実行し、その実行結果をVisual StudioのGraphics Debuggerでデバッグするためのツールです。UnityのShaderをDirectX 11のShaderとして実行するための設定は、<http://docs.unity3d.com/Manual/SL-DebuggingD3D11ShadersWithVS.html>を参照してください。

UnityのDirectX 11のShaderをGraphics Debuggerでデバッグするための設定は、<http://docs.unity3d.com/Manual/SL-DebuggingD3D11ShadersWithVS.html>を参照してください。

5.5.3 色相空間の表現

Mac/Mac両プラットフォームでVisual Studio/Unity 5のShaderをUIのShaderとして実行し、その実行結果をVisual StudioのGraphics Debuggerでデバッグするためのツールです。UnityのShaderをUIのShaderとして実行するための設定は、<http://docs.unity3d.com/Manual/SL-DebuggingD3D11ShadersWithVS.html>を参照してください。

UnityのGraphics DebuggerのWindow -> Frame Debuggerでデバッグするための設定は、<http://docs.unity3d.com/Manual/SL-DebuggingD3D11ShadersWithVS.html>を参照してください。

▲5.7 KnotGameHierarchyKnot

Draw CallRenderTextureGameGameRGB

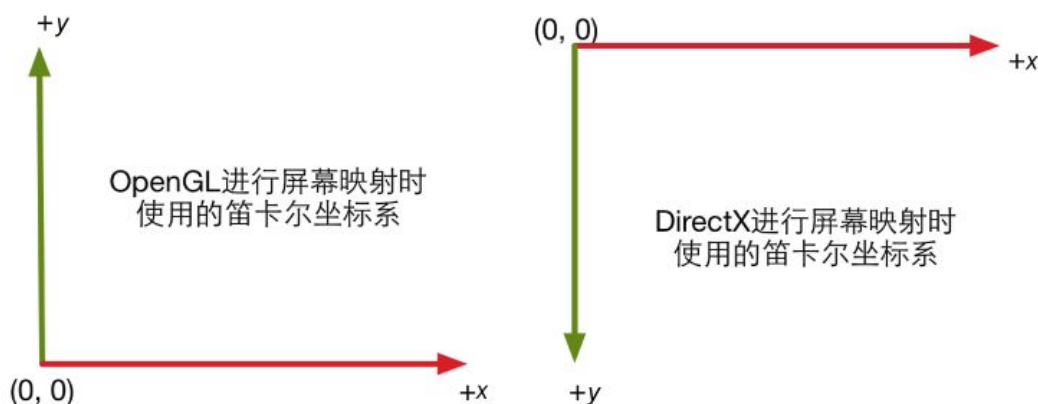
Unity 5frame capture
4Draw Call4Draw Call
5.2
Visual StudioIntel GPARenderDocNVIDIA NSightAMD GPU
PerfStudio

5.6 坐标系

Unity坐标系——坐标系Unity坐标系

5.6.1 坐标系

2.3.44.2.2OpenGLDirectXOpenGLOpenGL ES(0, 0)DirectXMetal(0, 0)5.8



▲5.8 OpenGLDirectX

Render TargetRender Texture12

UnityDirectXUnity

UnityEdit -> Project Settings -> Quality -> Anti AliasingUnityDirectXGraphics.BlitUnityDirectXDirectX


```
#if UNITY_UV_STARTS_AT_TOP
if (_MainTex_TexelSize.y > 0)
    uv.y = 1-uv.y;
#endif
```

UNITY_UV_STARTS_AT_TOP DirectX
_MainTex_TexelSize.y 0
13

12
Unity 12.5
Pass Bloom
DirectX
13
15.3
Pass

5.6.2 Shader

Windows Mac Shader

```
incorrect number of arguments to numeric-type constructor (compiling for d3d11)
```

```
output parameter 'o' not completely initialized (compiling for d3d11)
```

DirectX 9/11 Shader
Shader

```
// v float4  
float4 v = float4(0.0);
```

OpenGL 4 0.0 float4
DirectX 11

```
float4 v = float4(0.0, 0.0, 0.0, 0.0);
```

out
out
out

```
void vert (inout appdata_full v, out Input o) {  
    // UnityUNITY_INITIALIZE_OUTPUTo  
    UNITY_INITIALIZE_OUTPUT(Input,o);  
    // ...  
}
```

DirectX 9 / 11tex2Dtex2D
DirectX 9 / 11tex2D
ShaderUVtex2D
tex2Dlod

```
tex2Dlod(tex, float4(uv, 0, 0)).
```

#pragma target 3.0tex2DlodShader Model 3.0

5.6.3 Shader

5.4Shader
SV_POSITION POSITION Shader
Shader

- SV_POSITION ShaderPOSITION ShaderPS4
- SV_Target ShaderCOLORCOLOR0 ShaderPS4

5.6.4

ShaderA
BUnity<http://docs.unity3d.com/Manual/SL-PlatformDifferences.html>

5.7 Shader

Shader

5.7.1 float half fixed

Cg/HLSL Unity Shader Cg/HLSL 3 float half fixed 5.8 3

5.8 Cg/HLSL 3

float	32
half	16 -60 000 +60 000
fixed	11 -2.0 +2.0

- GPU
- GPU float half fixed PC half fixed
 - GPU Shader
 - fixed GPU fixed half

Shader fixed half float Shader 16

5.7.2

5.6.2 DirectX Shader DirectX

5.7.3

Shader Unity

temporary register limit of 8 exceeded

Arithmetic instruction limit of 64 exceeded; 65 arithmetic instructions needed to compile program

Shader Shader Target Shader Target

Shader Target 5.9 Unity Shader Target

5.9 Unity Shader Target

Shader Target	Shader Model
#pragma target 2.0	Shader Target Direct3D 9 Shader Model 2.0 LOD
#pragma target 3.0	Direct3D 9 Shader Model 3.0
#pragma target 4.0	Direct3D 10 Shader Model 4.0
#pragma target 5.0	Direct3D 11 Shader Model 5.0

Unity Shader Target

Shader Model

Shader Model Shader feature capability Shader Shader Model Shader

Shader Target Shader

5.7.4

Shader

GPU GPU if-else for while GPU CPU GPU CPU Shader GPU GPU

Shader Shader CPU Shader

- Shader
-
-

5.7.5 0

Shader
C#

Shader

```
fixed4 frag(v2f i) : SV_Target
{
    return fixed4(0.0/0.0,0.0/0.0, 0.0/0.0, 1.0);
}
```

Shader
1.0
Shader

0
if
0.000001
0

5.8 □□□□

这篇文章GPU的2个GPU 文章 第^[1] 这篇文章详细描述了GPU
 GPU的5.7.3 Shader Shader Model
 Shader Model
 Wiki^[2] HLSL^[3]

[1]Mark Harris, Ian Buck. "GPU Flow-Control Idioms." In GPU Gems 2. Addison-Wesley, 2007.

[2]High-Level Shading LanguageWiki
https://en.wikipedia.org/wiki/High-Level_Shading_Language
 []

[3]Shader Models vs Shader Profiles - HLSL -
[https://msdn.microsoft.com/en-us/library/windows/desktop/bb509626\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/bb509626(v=vs.85).aspx)

第6章 Unity 入门

本章主要介绍 Unity 的入门知识，包括 Unity 的安装、运行、基本操作等。

本章分为 6.1 节到 6.6 节，分别介绍 Unity 的入门知识、Unity Shader 的入门知识、Unity 的基本操作、Unity 的进阶操作、Unity 的进阶操作、Unity 的进阶操作。

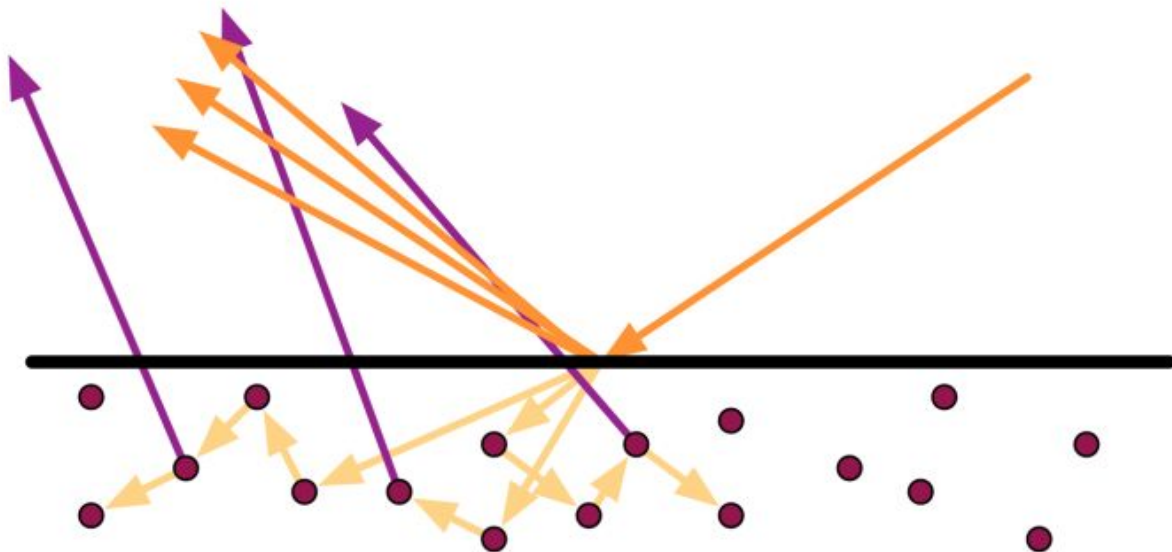
本章分为 9.5 节，分别介绍 Shader 的入门知识、Unity Shader 的入门知识。

$\cos \theta_t = \frac{d}{\cos \theta_i} \cos \theta_r$
 $\cos \theta_r = \frac{n_1 \cos \theta_i}{n_2}$

6.1.2 光の透過

光が物質に当たると、**scattering**（散乱）や **absorption**（吸収）が起こる。

光が物質に当たると、**refraction**（屈折）や **transmission**（透過）や **reflection**（反射）が起こる。
 6.2 光の透過



▲6.2 光の透過

光が物質に当たると、**specular**（鏡面反射）や **diffuse**（拡散反射）や **exitance**（放射率）が起こる。

Figure 6.1.3: A diagram illustrating the relationship between shading and lighting models. It shows a hierarchy where shading is a subset of lighting models.

6.1.3 Shading

The shading process is a subset of the lighting model. The lighting model defines how light interacts with a surface, while shading is the process of calculating the color and intensity of the surface based on the lighting model.

6.1.4 BRDF

The Bidirectional Reflectance Distribution Function (BRDF) is a function that describes the relationship between the incident light and the reflected light. It is a function of the incident direction, the reflected direction, and the surface properties. The BRDF is used to calculate the color and intensity of the surface based on the lighting model. The BRDF is a function of the incident direction, the reflected direction, and the surface properties. The BRDF is used to calculate the color and intensity of the surface based on the lighting model. The BRDF is a function of the incident direction, the reflected direction, and the surface properties. The BRDF is used to calculate the color and intensity of the surface based on the lighting model.

The BRDF is a function of the incident direction, the reflected direction, and the surface properties.

The BRDF is a function of the incident direction, the reflected direction, and the surface properties. The BRDF is used to calculate the color and intensity of the surface based on the lighting model. The BRDF is a function of the incident direction, the reflected direction, and the surface properties. The BRDF is used to calculate the color and intensity of the surface based on the lighting model.

6.2 物理模型

物理模型是描述物体表面反射特性的数学模型。它通常表示为双向反射分布函数 (BRDF) 的形式。

1975年，Bui Tuong Phong 提出了著名的 Phong 模型，该模型考虑了直接光 (direct light) 和间接光 (indirect light) 的贡献。

该模型通常表示为以下形式，其中 $c_{emissive}$ 和 $c_{specular}$ 是材料属性， $c_{diffuse}$ 和 $c_{ambient}$ 是环境属性。

- **emissive** 属性 $c_{emissive}$ 表示物体自身发出的光，即全局照明 (global illumination)。
- **specular** 属性 $c_{specular}$ 表示物体表面的镜面反射特性。
- **diffuse** 属性 $c_{diffuse}$ 表示物体表面的漫反射特性。
- **ambient** 属性 $c_{ambient}$ 表示环境光的贡献。

6.2.1 间接光

间接光 (indirect light) 是指光线在场景中经过多次反射后到达观察者的光。它通常由环境光 (ambient light) 和全局照明 (global illumination) 组成。

在 Phong 模型中，间接光的贡献通常通过环境光属性 $c_{ambient}$ 来近似表示。

$$c_{ambient} = g_{ambient}$$

6.2.2 发射

发射是指物体自身发出光线的现象。在计算机图形学中，发射体（emissive object）是指那些能够自行发光的物体，如光源、火焰、霓虹灯等。发射体的颜色通常由其发射的光谱分布决定，而亮度则由发射强度控制。

$$C_{emissive} = m_{emissive}$$

Unity 5 中，发射体的颜色可以通过 `Color` 属性来设置，而发射强度则通过 `emissionIntensity` 属性来控制。在 Shader 中，发射体的颜色通常表示为 `Color` 变量，而发射强度则表示为 `float` 变量。

6.2.3 漫反射

漫反射是指光线照射到物体表面后，向各个方向均匀散射的现象。在计算机图形学中，漫反射是物体表面最基本的反射类型。漫反射的颜色通常由物体的表面颜色决定，而亮度则由入射光的强度和物体表面的反射率决定。

在 Lambert's law 中，漫反射的亮度与入射光的强度和物体表面的反射率成正比。在 Shader 中，漫反射的颜色通常表示为 `Color` 变量，而反射率则表示为 `float` 变量。

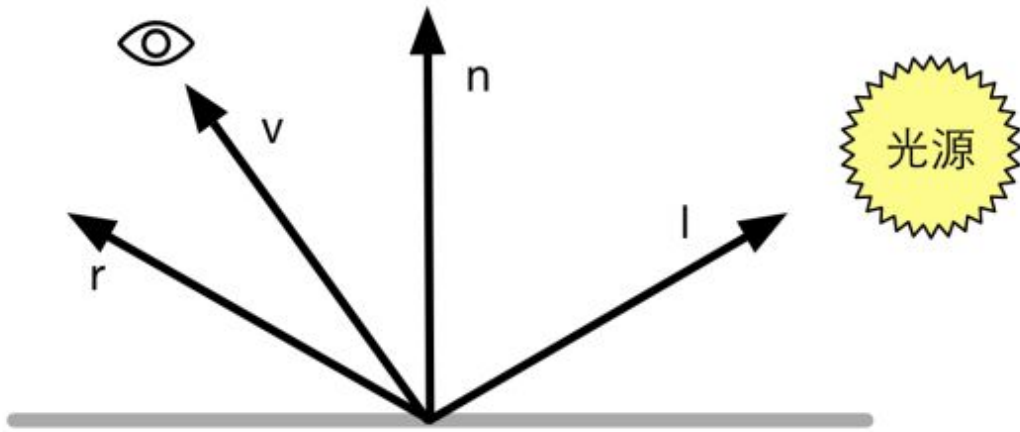
$$C_{diffuse} = (C_{light} \cdot m_{diffuse}) \max(0, \hat{n} \cdot \hat{l})$$

在公式中， $\hat{n}$ 表示物体表面的法向量， $\hat{l}$ 表示入射光的方向向量， $m_{diffuse}$ 表示物体表面的漫反射系数， C_{light} 表示入射光的颜色。当 $\hat{n} \cdot \hat{l} < 0$ 时，表示光线是从物体背面照射过来的，此时漫反射亮度为 0。

6.2.4 折射

折射是指光线从一种介质进入另一种介质时，传播方向发生改变的现象。在计算机图形学中，折射是透明物体（如玻璃、水）的基本属性。折射的强度由两种介质的折射率之比决定。

在 Shader 中，折射通常通过 `Refract` 函数来实现。在 Unity 5 中，折射的强度可以通过 `refractStrength` 属性来控制。在 Shader 中，折射的强度通常表示为 `float` 变量。



▲图6.3 Phong着色模型

Phong着色模型是3D图形学中最常用的着色模型之一。它通过计算反射光线的方向来模拟光滑表面的反射效果。

$$\hat{\mathbf{r}} = 2(\hat{\mathbf{n}} \cdot \hat{\mathbf{l}})\hat{\mathbf{n}} - \hat{\mathbf{l}}$$

Phong着色模型的计算公式如下：

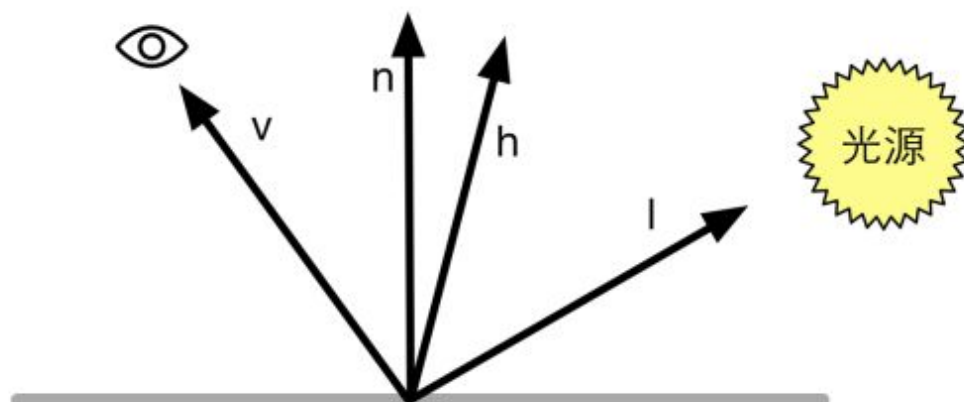
$$\mathbf{c}_{specular} = (\mathbf{c}_{light} \cdot \mathbf{m}_{specular}) \max(0, \hat{\mathbf{v}} \cdot \hat{\mathbf{r}})^{m_{gloss}}$$

其中 $\mathbf{m}_{gloss}$ 是材质的 **gloss** 属性，通常称为 **shininess**。它决定了反射光的强度。公式中的 $\hat{\mathbf{v}} \cdot \hat{\mathbf{r}}$ 是观察方向与反射方向的点积，用于计算反射光的强度。当观察方向与反射方向一致时，反射光的强度达到最大值。

Phong着色模型与Blinn着色模型非常相似。Blinn着色模型使用半反射向量 $\hat{\mathbf{h}}$ 来计算反射光的强度，而Phong着色模型使用反射方向 $\hat{\mathbf{r}}$ 。Blinn着色模型的公式为 $\hat{\mathbf{v}} \cdot \hat{\mathbf{h}}$ ，而Phong着色模型的公式为 $\hat{\mathbf{v}} \cdot \hat{\mathbf{r}}$ 。

$$\hat{h} = \frac{\hat{v} + \hat{l}}{|\hat{v} + \hat{l}|}$$

注意 $\hat{n} \cdot \hat{h}$ 和 $\hat{v} \cdot \hat{r}$ 都是标量6.4



▲6.4 Blinn

Blinn

$$c_{specular} = (c_{light} \cdot m_{specular}) \max(0, \hat{n} \cdot \hat{h})^{m_{gloss}}$$

Blinn Phong
 $\hat{v} \cdot \hat{l}$ $\hat{h}$ $\hat{v} \cdot \hat{l}$ Phong
 Blinn “
 ”Phong Blinn

6.2.5 光照模型

光照模型是计算机图形学中的一个重要概念，它描述了光线与物体表面相互作用的过程。光照模型可以分为两大类：基于像素的光照模型（**per-pixel lighting**）和基于顶点的光照模型（**per-vertex lighting**）。

基于像素的光照模型（**Phong shading**）是计算机图形学中最常用的光照模型之一。它通过计算每个像素的法向量，并应用 Phong 光照模型来计算该像素的颜色。Phong 光照模型考虑了环境光、漫反射光和镜面反射光的贡献。Phong 光照模型的计算复杂度较高，因为它需要对每个像素进行法向量的插值和光照计算。

基于顶点的光照模型（**Gouraud shading**）是另一种常用的光照模型。它通过计算每个顶点的法向量，并应用 Gouraud 光照模型来计算该顶点的颜色。Gouraud 光照模型考虑了环境光和漫反射光的贡献，但不考虑镜面反射光的贡献。Gouraud 光照模型的计算复杂度较低，因为它只需要对每个顶点进行法向量的插值和光照计算。

6.2.6 反射

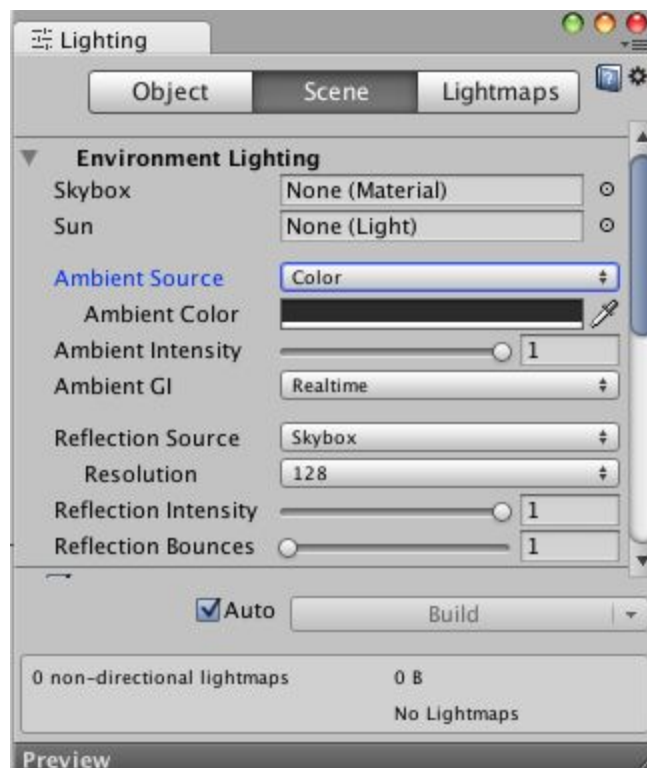
反射是光照模型中的一个重要概念，它描述了光线在物体表面发生反射的过程。反射可以分为两大类：漫反射和镜面反射。漫反射是指光线在物体表面发生无规则的反射，而镜面反射是指光线在物体表面发生有规则的反射。在计算机图形学中，反射模型可以分为两大类：基于顶点的反射模型（**Blinn-Phong**）和基于像素的反射模型（**Blinn-Phong**）。

基于顶点的反射模型（**Blinn-Phong**）是计算机图形学中最常用的反射模型之一。它通过计算每个顶点的法向量，并应用 Blinn-Phong 反射模型来计算该顶点的颜色。Blinn-Phong 反射模型考虑了环境光、漫反射光和镜面反射光的贡献。Blinn-Phong 反射模型的计算复杂度较高，因为它需要对每个顶点进行法向量的插值和反射计算。Blinn-Phong 反射模型还考虑了 Fresnel reflection 和 anisotropic 反射。Fresnel reflection 是指光线在物体表面发生反射时，反射光的强度与入射光的极化状态有关。anisotropic 反射是指光线在物体表面发生反射时，反射光的强度与入射光的方向有关。Blinn-Phong 反射模型还考虑了 isotropic 反射。isotropic 反射是指光线在物体表面发生反射时，反射光的强度与入射光的方向无关。Blinn-Phong 反射模型还考虑了 18 种不同的反射模型。

6.3 Unity環境構築

環境構築の準備作業

Unity環境構築の準備作業
Window -> Lighting -> Ambient Source/Ambient Color/Ambient Intensity の設定
6.5のShaderの準備作業
UnityのUNITY_LIGHTMODEL_AMBIENTの設定



▲6.5 Unityの*Window -> Lighting* の設定
Ambient Source/Ambient Color/Ambient Intensity の設定

Shaderの準備作業
UnityのUNITY_LIGHTMODEL_AMBIENTの設定

6.4 Unity Shaderの基礎

この章では、UnityのShaderの基礎を説明します。

6.2.3の章で、次のような式を説明しました。

$$\mathbf{c}_{diffuse} = (\mathbf{c}_{light} \cdot \mathbf{m}_{diffuse}) \max(0, \hat{\mathbf{n}} \cdot \hat{\mathbf{l}})$$

ここで、 $\mathbf{c}_{light}$ は4次元ベクトル、 $\mathbf{m}_{diffuse}$ は3次元ベクトル、 $\hat{\mathbf{n}}$ は法線ベクトル、 $\hat{\mathbf{l}}$ は照明ベクトルです。

max関数は、Cgのsat関数と同じです。Cgのsat関数は、次のように定義されています。

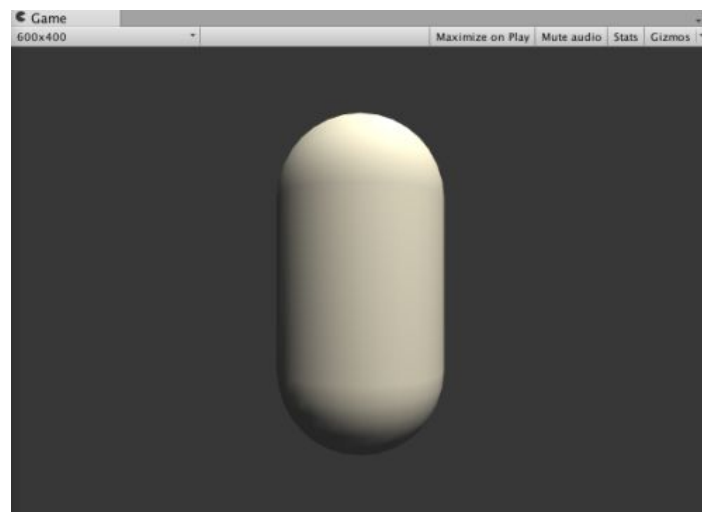
sat(x) =

if (x < 0) return 0; else return x;

つまり、xが0以下であれば0を返す、それ以外の場合はxを返す関数です。

6.4.1 Shaderの作成

この章では、UnityのShaderを作成する方法を説明します。



▲6.6 UnityのShader

この章では、UnityのShaderを作成する方法を説明します。

1. UnityのプロジェクトにScene_6_4という名前で作成したSceneをUnity 5.2のプロジェクトにインポートします。Window -> Lighting -> Skyboxを選択します。

2. Shaderの作成には、DiffuseVertexLevelMatを使用します。

7. 顶点着色器中，我们使用以下结构体来定义顶点和法线数据：

```
struct a2v {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct v2f {
    float4 pos : SV_POSITION;
    fixed3 color : COLOR;
};
```

在顶点着色器中，我们使用 `a2v` 结构体来定义顶点位置 `vertex` 和法线 `normal`。我们使用 `Unity` 提供的 `UNITY_MATRIX_MVP` 矩阵来将顶点位置从对象空间转换到投影空间。我们使用 `v2f` 结构体来定义顶点位置 `pos` 和颜色 `color`。我们使用 `UNITY_MATRIX_MVP` 矩阵来将顶点位置从对象空间转换到投影空间。我们使用 `UNITY_LIGHTMODEL_AMBIENT` 来定义环境光颜色。

8. 在顶点着色器中，我们使用以下函数来计算顶点的颜色：

```
v2f vert(a2v v) {
    v2f o;
    // Transform the vertex from object space to projection space
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    // Get ambient term
    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

    // Transform the normal from object space to world space
    fixed3 worldNormal = normalize(mul(v.normal, (float3x3)_World2Object));
    // Get the light direction in world space
    fixed3 worldLight = normalize(_WorldSpaceLightPos0.xyz);
    // Compute diffuse term
    fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb * saturate(dot(worldNormal, worldLight));

    o.color = ambient + diffuse;

    return o;
}
```

在顶点着色器中，我们使用 `o` 结构体来定义顶点位置 `pos` 和颜色 `color`。我们使用 `UNITY_MATRIX_MVP` 矩阵来将顶点位置从对象空间转换到投影空间。我们使用 `UNITY_LIGHTMODEL_AMBIENT` 来定义环境光颜色。

在顶点着色器中，我们使用 `4` 个通道来定义顶点的颜色。我们使用 `_Diffuse` 来定义顶点的颜色。我们使用 `v.normal` 来定义顶点的法线。我们使用 `Unity` 提供的 `_LightColor0` 来定义顶点的颜色。我们使用 `Pass` 来定义顶点的颜色。我们使用 `LightMode` 来定义顶点的颜色。我们使用 `_WorldSpaceLightPos0` 来定义顶点的颜色。我们使用 `_WorldSpaceLightPos0` 来定义顶点的颜色。我们使用 `6.6` 来定义顶点的颜色。

在顶点着色器中，我们使用 `a2v` 结构体来定义顶点位置 `vertex` 和法线 `normal`。我们使用 `4.7` 来定义顶点的颜色。我们使用 `_World2Object` 来定义顶点的颜色。我们使用 `mul` 来定义顶点的颜色。我们使用 `_World2Object` 来定义顶点的颜色。

在顶点着色器中，我们使用 `saturate` 来定义顶点的颜色。我们使用 `saturate` 来定义顶点的颜色。我们使用 `Cg` 来定义顶点的颜色。我们使用 `[0, 1]` 来定义顶点的颜色。我们使用 `saturate` 来定义顶点的颜色。

9

```
fixed4 frag(v2f i) : SV_Target {
    return fixed4(i.color, 1.0);
}
```

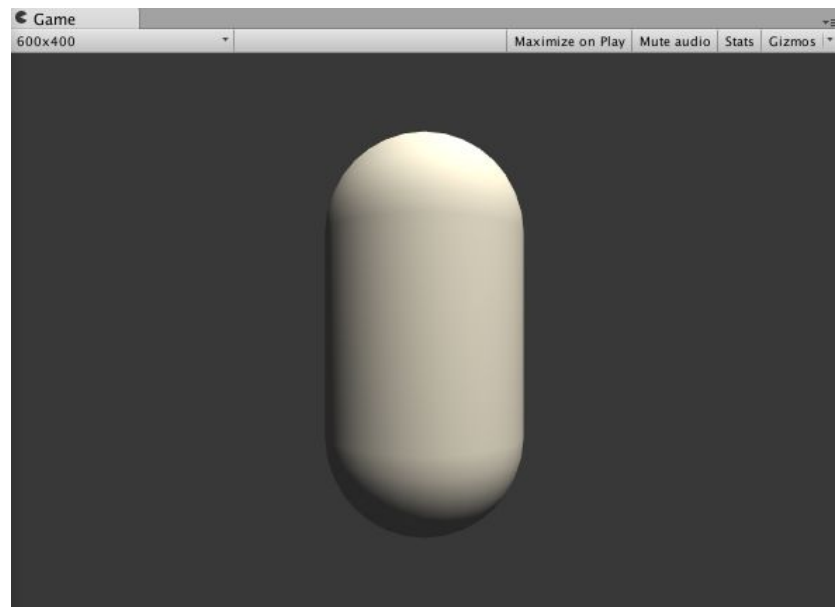
10 Unity Shader shader Diffuse

Fallback "Diffuse"

6.6

6.4.2 〇〇〇〇〇〇〇〇

Shader 6.7



▲ 6.7

□ □ □ □ □ □ □ □ □ □ □ □ □

16.4.1

DiffusePixelLevelMat

3 Unity Shader Shader Chapter6-DiffusePixelLevel Shader

04002000000000000000

Chapter6-DiffusePixelLevel 6.4.1 6.4.1 Chapter6-DiffusePixelLevel

1 v2f

```
struct v2f {
    float4 pos : SV_POSITION;
    float3 worldNormal : TEXCOORD0;
};
```

2

```
v2f vert(a2v v) {
    v2f o;
    // Transform the vertex from object space to projection space
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    // Transform the normal from object space to world space
    o.worldNormal = mul(v.normal, (float3x3)_World2Object);

    return o;
}
```

3

```
fixed4 frag(v2f i) : SV_Target {
    // Get ambient term
    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

    // Get the normal in world space
    fixed3 worldNormal = normalize(i.worldNormal);
    // Get the light direction in world space
    fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz);

    // Compute diffuse term
    fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb * saturate(dot(worldNormal, worldLightDir));

    fixed3 color = ambient + diffuse;

    return fixed4(color, 1.0);
}
```

6.4.1

Half Lambert

6.4.3

6.4.1 Valve

$$\mathbf{c}_{diffuse} = (\mathbf{c}_{light} \cdot \mathbf{m}_{diffuse})(\alpha(\hat{\mathbf{n}} \cdot \hat{\mathbf{l}}) + \beta)$$

$\max_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^n \log \frac{1}{\sum_{j=1}^n \exp(\mathbf{w}^T \mathbf{x}_i^j)}$

$$\mathbf{c}_{diffuse} = (\mathbf{c}_{light} \cdot \mathbf{m}_{diffuse})(0.5(\hat{\mathbf{n}} \cdot \hat{\mathbf{l}}) + 0.5)$$

$\hat{n} \cdot \hat{l}$ $[-1, 1]$ $[0, 1]$
 0

□ □

6.4.2

16.4.1 16.4.1 16.4.1

2 HalfLambertMat

3D Unity Shader Shader Chapter6-HalfLambert Shader 2D

[illegible]

Chapter6-HalfLambertShader6.4.2Chapter6-DiffusePixelLevel

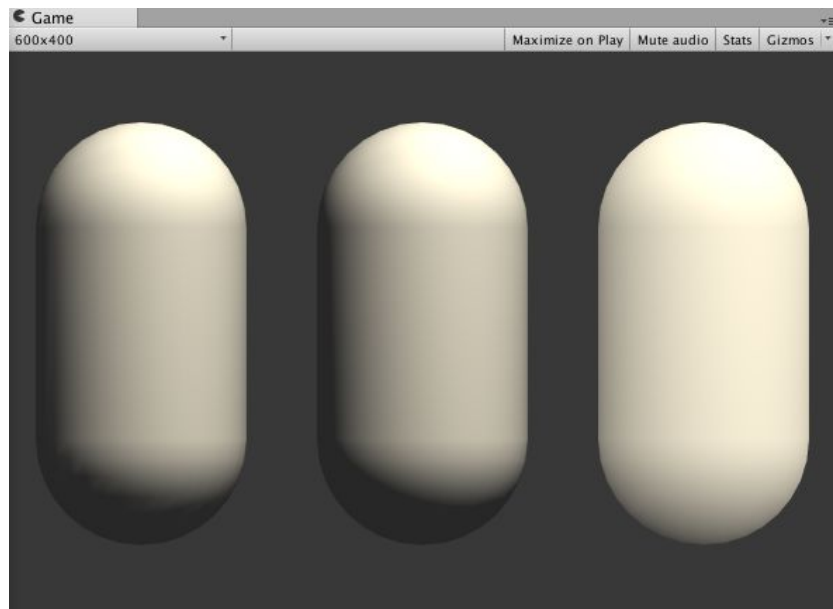
```
fixed4 frag(v2f i) : SV_Target {
    ...

    // Compute diffuse term
    fixed halfLambert = dot(worldNormal, worldLightDir) * 0.5 + 0.5;
    fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb * halfLambert;

    fixed3 color = ambient + diffuse;

    return fixed4(color, 1.0);
}
```

6.8



▲6.8

6.5 Unity Shader

6.2.4

c_specular = (c_light · m_specular) max(0, v · r)^m_gloss

4 c_light m_specular v r n l

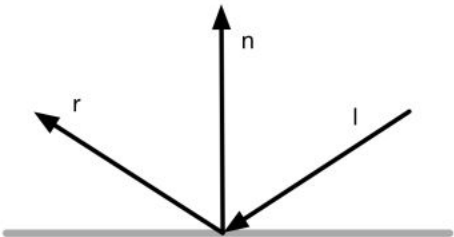
r = 2(n · l)n - l

Cgreflect

reflect(i, n)

i n float float2 float3

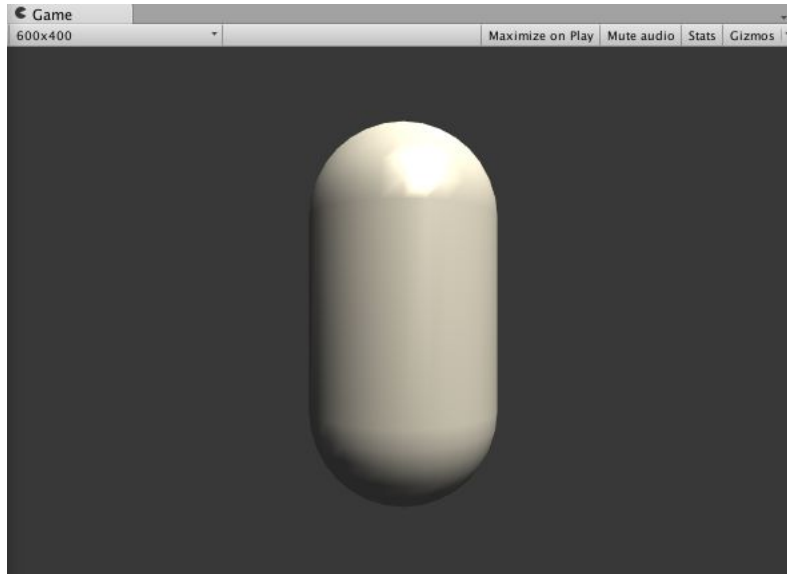
i n reflect 6.9



▲6.9 Cgreflect

6.5.1

6.10



▲6.10 Specular Vertex Level Shader

Specular Vertex Level Shader

1. Unity 5.2.10 Scene_6_5 Unity 5.2.10 Window -> Lighting -> Skybox

2. SpecularVertexLevelMat

3. Unity Shader Chapter6-SpecularVertexLevel Shader 2

4. SpecularVertexLevelMat

5. SpecularVertexLevelMat

SpecularVertexLevelMat Shader Chapter6-SpecularVertexLevel Shader

1. Shader

```
Shader "Unity Shaders Book/Chapter 6/Specular Vertex-Level" {
```

2. Shader Properties

```
Properties {
    _Diffuse ("Diffuse", Color) = (1, 1, 1, 1)
    _Specular ("Specular", Color) = (1, 1, 1, 1)
    _Gloss ("Gloss", Range(8.0, 256)) = 20
}
```

_Specular _Gloss

3. SubShader Pass Pass Pass

```
SubShader {
    Pass {
        Tags { "LightMode"="ForwardBase" }
```

```
        LightMode Pass Pass Unity 9
        LightMode Unity _LightColor0
```

```
    4 CGPROGRAM ENDCG CG
    #pragma Unity vert frag
```

```
CGPROGRAM
```

```
#pragma vertex vert
#pragma fragment frag
```

```
    5 Unity _LightColor0 Unity Lighting.cginc
```

```
#include "Lighting.cginc"
```

```
    6 Shader Properties
```

```
fixed4 _Diffuse;
fixed4 _Specular;
float _Gloss;
```

```
    7 0 1 _Diffuse _Specular fixed _Gloss
    float
```

```
struct a2v {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};

struct v2f {
    float4 pos : SV_POSITION;
    fixed3 color : COLOR;
};
```

```
    8
```

```
v2f vert(a2v v) {
    v2f o;
    // Transform the vertex from object space to projection space
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    // Get ambient term
    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

    // Transform the normal from object space to world space
    fixed3 worldNormal = normalize(mul(v.normal, (float3x3)_World2Object));
    // Get the light direction in world space
    fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz);

    // Compute diffuse term
    fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb * saturate(dot(worldNormal, worldLightDir));
```

```

// Get the reflect direction in world space
fixed3 reflectDir = normalize(reflect(-worldLightDir, worldNormal));
// Get the view direction in world space
fixed3 viewDir = normalize(_WorldSpaceCameraPos.xyz - mul(_Object2World, v.vertex).xyz);

// Compute specular term
fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(saturate(dot(reflectDir, viewDir)), _Gloss);

o.color = ambient + diffuse + specular;

return o;
}

```

6.4 `reflectDir` `Cg`
`reflect` `worldLightDir` `reflect`
`_WorldSpaceCameraPos`
`_WorldSpaceCameraPos`

4
 9

```

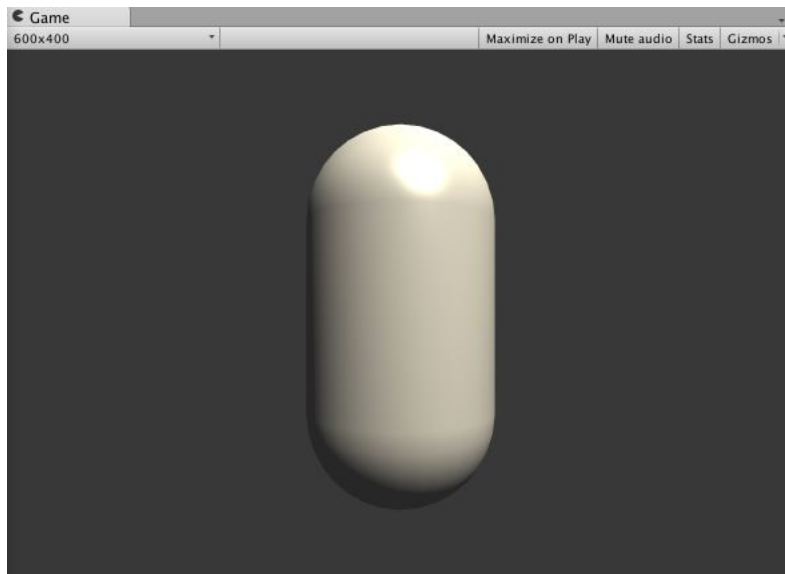
fixed4 frag(v2f i) : SV_Target {
    return fixed4(i.color, 1.0);
}

```

10 Unity Shader `Shader` `Specular`

Fallback "Specular"

6.10



▲ 6.11

6.5.2

6.11

6.5.1

SpecularPixelLevelMat

Unity Shader Shader Chapter6-SpecularPixelLevel Shader 2

4 2

Chapter6-SpecularPixelLevel Shader 6.5.1

1 v2f

```
struct v2f {
    float4 pos : SV_POSITION;
    float3 worldNormal : TEXCOORD0;
    float3 worldPos : TEXCOORD1;
};
```

2

```
v2f vert(a2v v) {
    v2f o;
    // Transform the vertex from object space to projection space
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    // Transform the normal from object space to world space
    o.worldNormal = mul(v.normal, (float3x3)_World2Object);
    // Transform the vertex from object space to world space
    o.worldPos = mul(_Object2World, v.vertex).xyz;

    return o;
}
```

3

```
fixed4 frag(v2f i) : SV_Target {
    // Get ambient term
    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

    fixed3 worldNormal = normalize(i.worldNormal);
    fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz);

    // Compute diffuse term
    fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb * saturate(dot(worldNormal, worldLightDir));

    // Get the reflect direction in world space
    fixed3 reflectDir = normalize(reflect(-worldLightDir, worldNormal));
    // Get the view direction in world space
    fixed3 viewDir = normalize(_WorldSpaceCameraPos.xyz - i.worldPos.xyz);
    // Compute specular term
    fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(saturate(dot(reflectDir, viewDir)), _Gloss);

    return fixed4(ambient + diffuse + specular, 1.0);
}
```

6.5.1 Phong shading

Phong shading

6.5.3 Blinn-Phong shading

6.5.2 Phong shading in Unity 6.2.4 — Blinn shading
Blinn shading uses the half vector $\hat{h}$ instead of the view vector $\hat{v}$ to compute the specular reflection.

$$\hat{h} = \frac{\hat{v} + \hat{l}}{|\hat{v} + \hat{l}|}$$

Blinn shading

$$c_{specular} = (c_{light} \cdot m_{specular}) \max(0, \hat{n} \cdot \hat{h})^{m_{gloss}}$$

Blinn-Phong shading 6.5.2

1. Create a new shader

2. Create a new material named BlinnPhongMat

3. Create a new Unity Shader named Chapter6-BlinnPhongShader in the Shader folder
4. Create a new material named Chapter6-BlinnPhongMat in the Materials folder

4. Create a new material named Chapter6-BlinnPhongMat in the Materials folder

Chapter6-BlinnPhongShader Shader 6.5.2 Chapter6-Specular PixelLevel shading

```
fixed4 frag(v2f i) : SV_Target {  
    ...  
    // Get the view direction in world space  
    fixed3 viewDir = normalize(_WorldSpaceCameraPos.xyz - i.worldPos.xyz);  
    // Get the half direction in world space  
    fixed3 halfDir = normalize(worldLightDir + viewDir);  
    // Compute specular term  
    fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0, dot(worldNormal, halfDir)), _Gloss);  
    return fixed4(ambient + diffuse + specular, 1.0);  
}
```

6.12 Phong shading and Blinn-Phong shading

6.6 使用Unity

使用Unity时，需要将LightPos0.xyz和CameraPos.xyz - i.worldPosition.xyz归一化，以便在9.2中使用。

Unity 5.3.1版本中，UnityCG.cginc文件中定义了以下函数：

6.1 UnityCG.cginc

函数名	描述
float3 WorldSpaceViewDir (float4 v)	返回世界空间中的视图方向，即UnityWorldSpaceViewDir。
float3 UnityWorldSpaceViewDir (float4 v)	返回世界空间中的视图方向。
float3 ObjSpaceViewDir (float4 v)	返回物体空间中的视图方向。
float3 WorldSpaceLightDir (float4 v)	返回世界空间中的光照方向，即UnityWorldSpaceLightDir。
float3 UnityWorldSpaceLightDir (float4 v)	返回世界空间中的光照方向。
float3 ObjSpaceLightDir (float4 v)	返回物体空间中的光照方向。
float3 UnityObjectToWorldNormal (float3 norm)	返回物体空间中的法线方向。

関 数	説 明
float3 UnityObjectToWorldDir (float3 dir)	オブジェクト空間の方向ベクトルをワールド空間に変換する
float3 UnityWorldToObjectDir(float3 dir)	ワールド空間の方向ベクトルをオブジェクト空間に変換する

UnityXXXはUnity 5で導入された関数で、Unity 5.9以前は5.9.1で導入されたWorldSpaceViewDir関数で代替されていた。

```
// Computes world space view direction, from object space position
inline float3 UnityWorldSpaceViewDir( in float3 worldPos )
{
    return _WorldSpaceCameraPos.xyz - worldPos;
}
```

Unity 5.9.1以前は、WorldSpaceLightDir関数で代替されていた。

Unity 5.9.1以前は、WorldSpaceLightDir関数はUnity 5.9.1で導入された。Unity 5.9.1以前は、_WorldSpaceLightPos0で代替されていた。Unity 5.9.1以前は、_WorldSpaceLightPos0で代替されていた。

Unity Shader

Unity Shaderは、Unity 5.9.1以前は、Unity Shaderで代替されていた。

Unity Shader 6.5.3は、Unity Shaderで代替されていた。

Unity 5.2以前は、Scene_6_6 Unity 5.2以前は、Window -> Lighting -> Skybox で代替されていた。

Unity Shader 6.5.3は、BlinnPhongUseBuildInFunctionMatで代替されていた。

3 Unity Shader Shader Chapter6-
BlinnPhongUseBuildIn function Shader 2

4 2

Chapter6-BlinnPhongUseBuildInFunction Chapter6-
BlinnPhong Unity

1 UnityObjectToWorldNormal

```
v2f vert(a2v v) {  
    v2f o;  
    ...  
    // Use the build-in function to compute the normal in world space  
    o.worldNormal = UnityObjectToWorldNormal(v.normal);  
    ...  
    return o;  
}
```

2 UnityWorldSpaceLightDir
UnityWorldSpaceView
Dir

```
fixed4 frag(v2f i) : SV_Target {  
    ...  
  
    fixed3 worldNormal = normalize(i.worldNormal);  
    // Use the build-in function to compute the light direction in world space  
    // Remember to normalize the result  
    fixed3 worldLightDir = normalize(UnityWorldSpaceLightDir(i.worldPos));  
  
    ...  
  
    // Use the build-in function to compute the view direction in world space  
    // Remember to normalize the result  
    fixed3 viewDir = normalize(UnityWorldSpaceViewDir(i.worldPos));  
    ...  
}
```

normalize

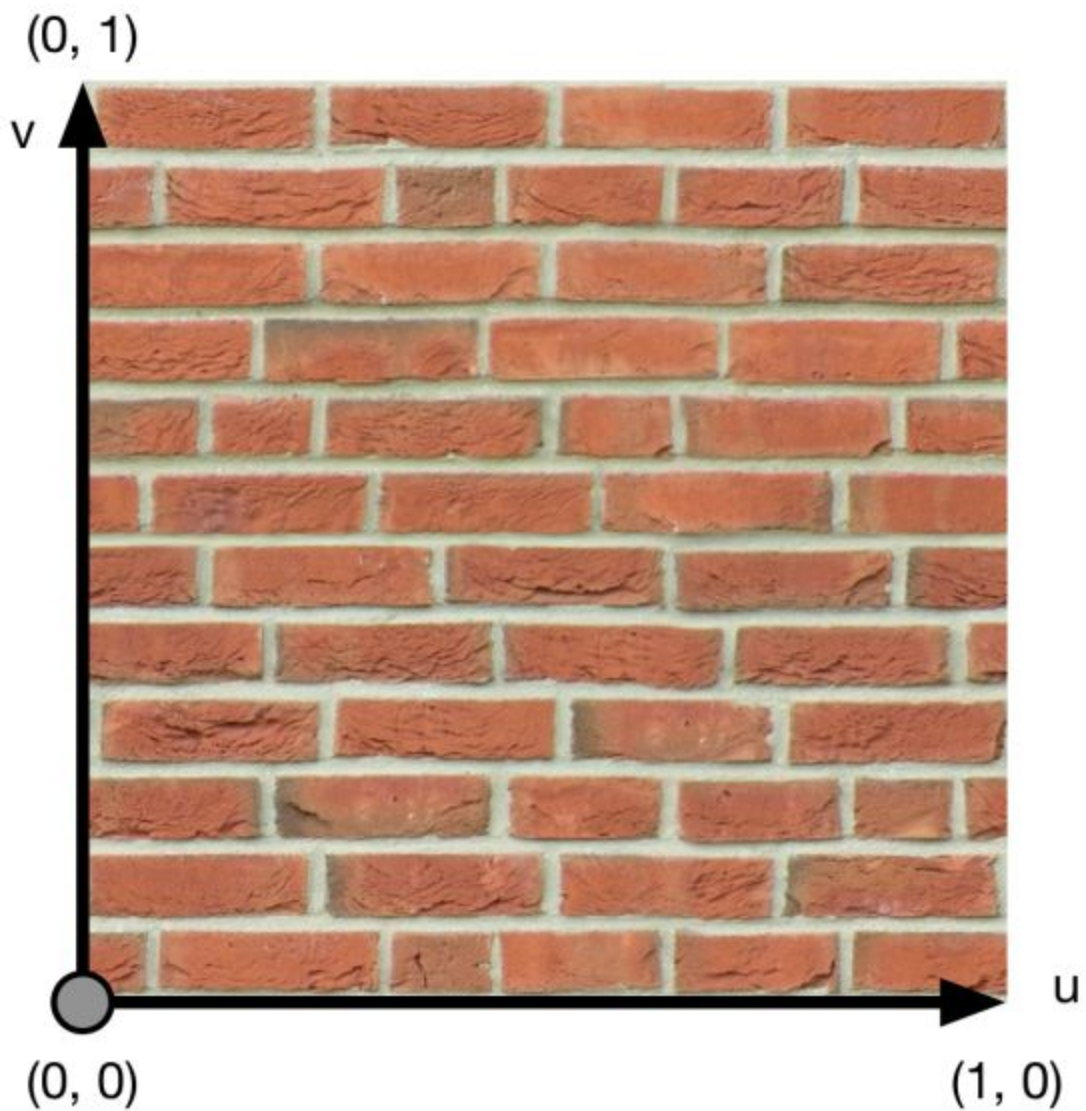
7 纹理

纹理映射是计算机图形学中的一个重要概念。在 OpenGL 中，纹理映射使用 `texture mapping` 函数来指定纹理坐标。在 DirectX 中，纹理映射使用 `texel` 函数来指定纹理坐标。

纹理映射坐标通常表示为 `texture-mapping coordinates`。在 2D 纹理映射中，纹理坐标 (u, v) 通常表示为 `uv` 坐标。

纹理映射坐标的范围通常为 256×256 或 1024×1024 。在 OpenGL 中，纹理坐标 $[0, 1]$ 表示纹理的左上角到右下角。在 DirectX 中，纹理坐标 $[0, 1]$ 表示纹理的左下角到右上角。在 Unity 中，纹理坐标 $[0, 1]$ 表示纹理的左上角到右下角。7.1.2 纹理映射

在 OpenGL 和 DirectX 中，纹理映射是一个重要的概念。在 OpenGL 中，纹理映射使用 `texture mapping` 函数来指定纹理坐标。在 DirectX 中，纹理映射使用 `texel` 函数来指定纹理坐标。Unity 5.6 版本中，纹理映射使用 `texture mapping` 函数来指定纹理坐标。在 OpenGL 和 DirectX 中，纹理映射是一个重要的概念。Unity 5.6 版本中，纹理映射使用 `texture mapping` 函数来指定纹理坐标。7.1 纹理



▲7.1 Unity

Unity 7.1
Unity Shader 7.2
Unity 7.3 7.4

Shader 9.5
Unity Shader

```

\documentclass[MainTex]{3.3.2}{2D}{
"white"
Color

```

3 SubShader Pass Pass Pass

```
SubShader {
    Pass {
        Tags { "LightMode"="ForwardBase" }
    }
}
```

LightMode Pass Pass Unity

4 CGPROGRAM ENDCG Cg
#pragma Unity vert frag

```
CGPROGRAM

#pragma vertex vert
#pragma fragment frag
```

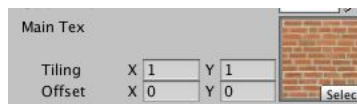
5 Unity _LightColor0 Unity Lighting.cginc

```
#include "Lighting.cginc"
```

6 Cg

```
fixed4 _Color;
sampler2D _MainTex;
float4 _MainTex_ST;
fixed4 _Specular;
float _Gloss;
```

float4 _MainTex_ST
Unity _ST ST scale translation
_MainTex_ST xy zw
7.3 7.1.2



7.3

7

```
struct a2v {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
};

struct v2f {
    float4 pos : SV_POSITION;
    float3 worldNormal : TEXCOORD0;
    float3 worldPos : TEXCOORD1;
    float2 uv : TEXCOORD2;
};
```

a2v TEXCOORD0 texcoord Unity
v2f uv

8. 顶点着色器

```
v2f vert(a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.worldNormal = UnityObjectToWorldNormal(v.normal);

    o.worldPos = mul(_Object2World, v.vertex).xyz;

    o.uv = v.texcoord.xy * _MainTex_ST.xy + _MainTex_ST.zw;
    // Or just call the built-in function
    // o.uv = TRANSFORM_TEX(v.texcoord, _MainTex);

    return o;
}
```

在顶点着色器中，我们使用 `_MainTex_ST` 属性来指定纹理的缩放和偏移。该属性是一个 `float2` 类型的变量，其 `xy` 分量表示纹理在 `uv` 平面上的缩放，而 `zw` 分量表示纹理在 `uv` 平面上的偏移。Unity 引擎会自动计算该属性，我们可以在 `TRANSFORM_TEX` 宏中使用它。

```
// Transforms 2D UV by scale/bias property
#define TRANSFORM_TEX(tex,name) (tex.xy * name##_ST.xy + name##_ST.zw)
```

在顶点着色器中，我们使用 `_ST` 属性来指定纹理的缩放和偏移。

9. 片段着色器

```
fixed4 frag(v2f i) : SV_Target {
    fixed3 worldNormal = normalize(i.worldNormal);
    fixed3 worldLightDir = normalize(UnityWorldSpaceLightDir(i.worldPos));

    // Use the texture to sample the diffuse color
    fixed3 albedo = tex2D(_MainTex, i.uv).rgb * _Color.rgb;

    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;

    fixed3 diffuse = _LightColor0.rgb * albedo * max(0, dot(worldNormal, worldLightDir));

    fixed3 viewDir = normalize(UnityWorldSpaceViewDir(i.worldPos));
    fixed3 halfDir = normalize(worldLightDir + viewDir);
    fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0, dot(worldNormal, halfDir)), _Gloss);

    return fixed4(ambient + diffuse + specular, 1.0);
}
```

在片段着色器中，我们使用 `Cg` 宏来指定纹理的缩放和偏移。该宏是一个 `float2` 类型的变量，其 `xy` 分量表示纹理在 `uv` 平面上的缩放，而 `zw` 分量表示纹理在 `uv` 平面上的偏移。我们使用 `tex2D` 函数来采样纹理，并将结果与 `_Color` 属性相乘，得到最终的 `albedo` 值。

10. Shader 的 Fallback

```
Fallback "Specular"
```

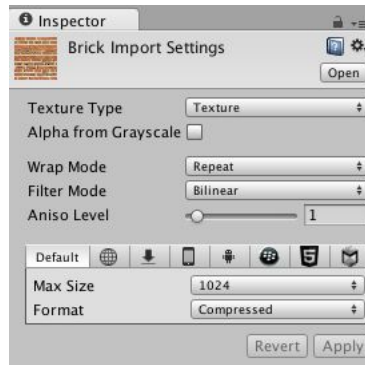
在 Unity 引擎中，我们可以使用 `SingleTextureMat` 属性来指定纹理的缩放和偏移。该属性是一个 `float2` 类型的变量，其 `xy` 分量表示纹理在 `uv` 平面上的缩放，而 `zw` 分量表示纹理在 `uv` 平面上的偏移。我们可以在 `Brick_Diffuse.jpg` 纹理上使用 `Main Tex` 属性。

7.1.2 纹理

UnityのInspectorでtex2DのTextureを指定する。UnityのInspectorで

UnityのInspectorで7.4のInspector

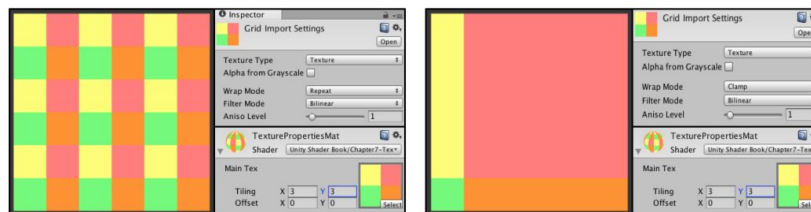
Texture Normal map Cubemap Unity Shader Unity



▲7.4 Inspector

Texture Alpha from Grayscale 8

Wrap Mode [0, 1] Wrap Mode Repeat Clamp Scene_7_1_2_a



▲7.5 Wrap Mode [0, 1]

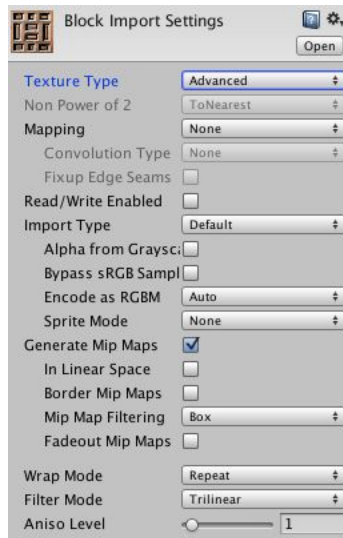
7.5 Tiling (3, 3) Wrap Mode Repeat Clamp

_MainTex_ST Unity Shader

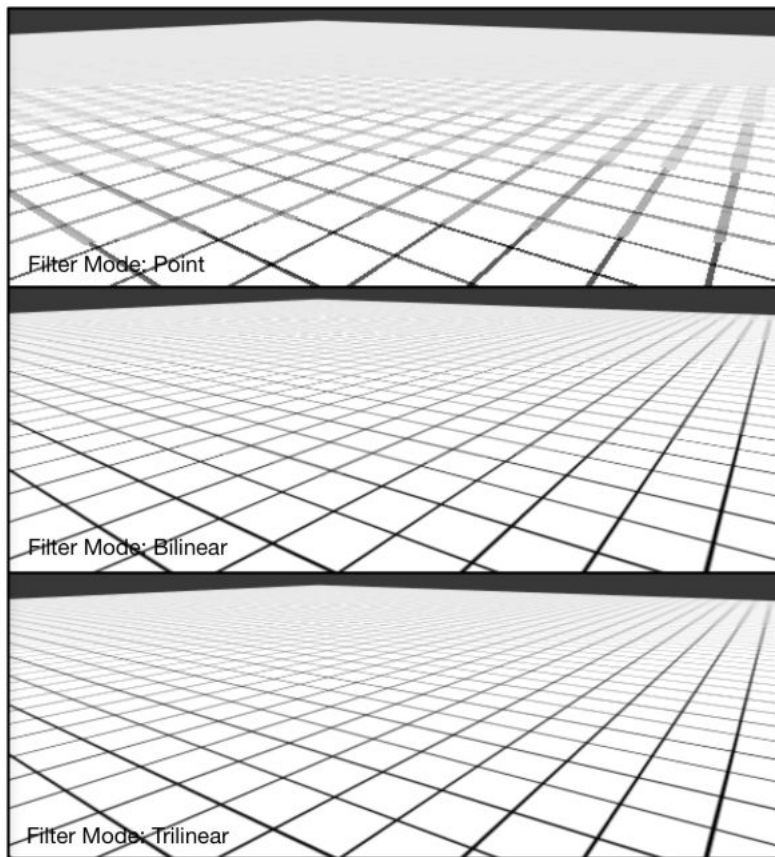
```
o.uv = v.texcoord.xy * _MainTex_ST.xy + _MainTex_ST.zw;  
// Or just call the built-in function  
o.uv = TRANSFORM_TEX(v.texcoord, _MainTex);
```

7.6

7.6 (0.2, 0.6) Wrap Mode Repeat Clamp



▲7.8 Advancedのテクスチャ設定



▲7.9 Point + Bilinear + Trilinear + Anisotropic

Unityのテクスチャ設定は7.10



```

Format UnityTexture TypeAdvancedFormat
Truecolor
Truecolor

```

7.2 □□□□

bump mapping

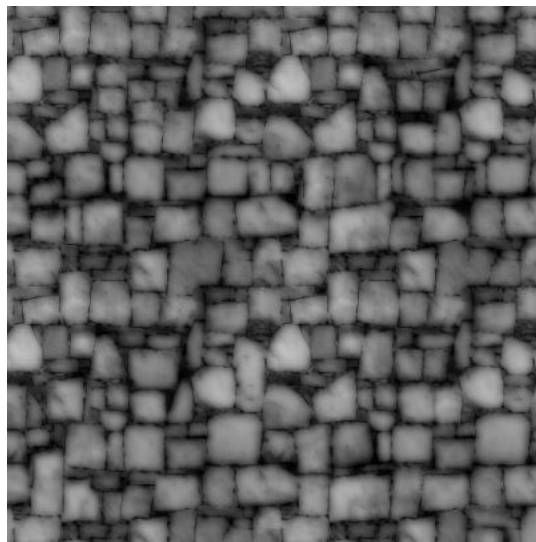
height map displacement
height mapping normal map
normal mapping

7.2.1 〇〇〇〇

```

#####intensity#####
#####
#####
#####7.11#####

```



▲ 7.11

[illegible]

7.2.2 四个空

[illegible]

$$pixel = \frac{normal + 1}{2}$$

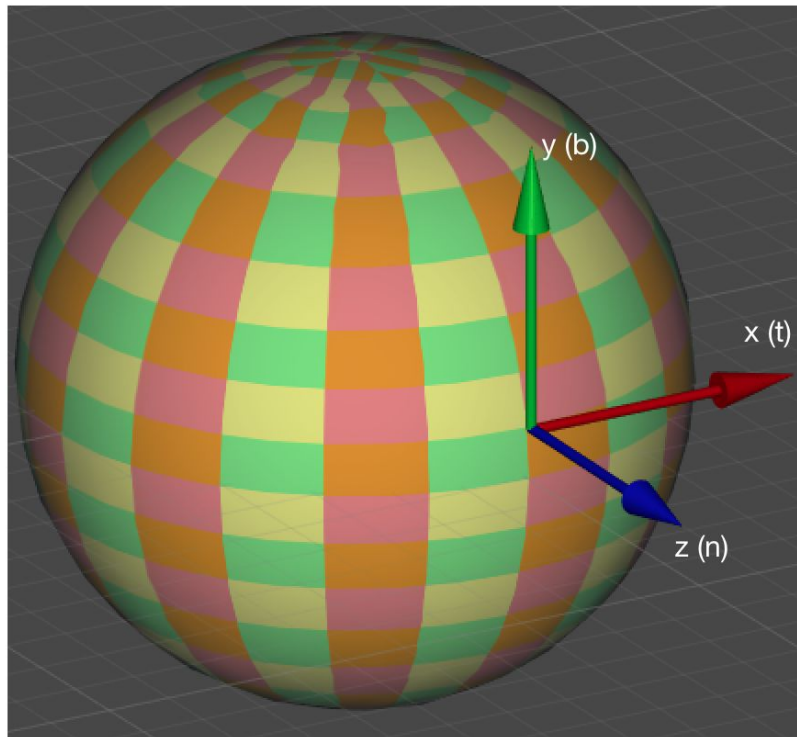
[illegible]

$$normal = pixel \times 2 - 1$$

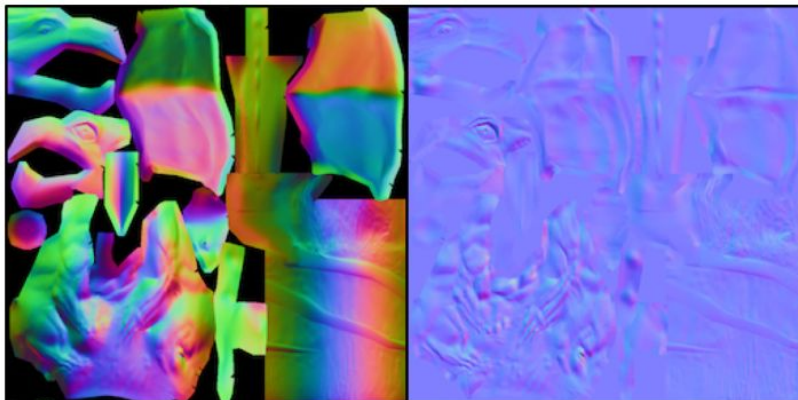
object-space normal map
 tangent space
 z n x t y
 bitangent b 7.12

7.13 **tangent-space normal map** 7.13 <http://www.surlybird.com/tutorials/TangentSpace/>

7.13 “ ” (0, 1, 0) RGB(0.5, 1, 0.5) (0, -1, 0) (0.5, 0, 0.5) z (0, 0, 1) RGB(0.5, 0.5, 1)

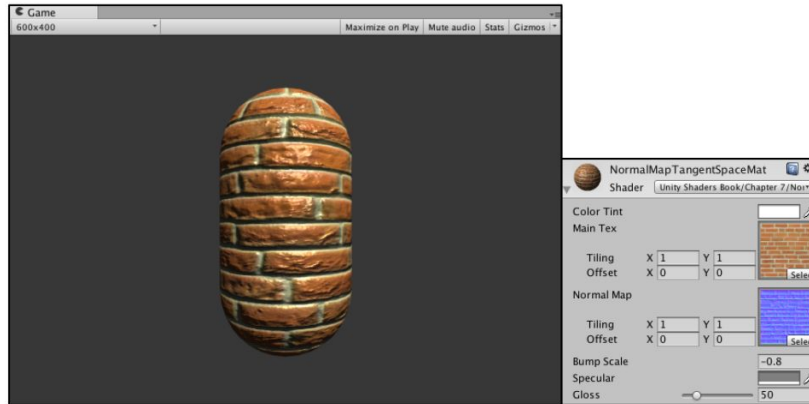


7.12 $x(t)$ $y(b)$ $z(n)$



7.13

“ ”



▲ 7.14

[illegible]

05000000

Chapter7-NormalMapTangentSpace

1 Unity Shader

```
Shader "Unity Shaders Book/Chapter 7/Normal Map In Tangent Space" {
```

2 Properties

```
Properties {
  _Color ("Color Tint", Color) = (1,1,1,1)
  _MainTex ("Main Tex", 2D) = "white" {}
  _BumpMap ("Normal Map", 2D) = "bump" {}
  _BumpScale ("Bump Scale", Float) = 1.0
  _Specular ("Specular", Color) = (1, 1, 1, 1)
  _Gloss ("Gloss", Range(8.0, 256)) = 20
}
```

```

    [0]_BumpMap[0]"bump"[0]"bump"[0]Unity[0]
    [0]"bump"[0]_BumpScale[0]0[0]

```

3 SubShader Pass Pass Pass

```
SubShader {
    Pass {
        Tags { "LightMode"="ForwardBase" }
    }
}
```

LightMode Pass Pass Unity

```
#pragma pack(push, 1)
CGPROGRAM
ENDCG
Cg
#pragma pack(pop)
Unity
vert frag
```

```
CGPROGRAM
#pragma vertex vert
#pragma fragment frag
```

5UnityLightColor0UnityLighting.cginc

```
#include "Lighting.cginc"
```

6PropertiesCg

```
fixed4 _Color;
sampler2D _MainTex;
float4 _MainTex_ST;
sampler2D _BumpMap;
float4 _BumpMap_ST;
float _BumpScale;
fixed4 _Specular;
float _Gloss;
```

MainTex_BumpMap_MainTex_ST_BumpMap_ST

7a2v

```
struct a2v {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float4 tangent : TANGENT;
    float4 texcoord : TEXCOORD0;
};
```

TANGENTfloat4tangentUnitytangent
normaltangentfloat4float3tangent.w
—
—

8v2f

```
struct v2f {
    float4 pos : SV_POSITION;
    float4 uv : TEXCOORD0;
    float3 lightDir: TEXCOORD1;
    float3 viewDir : TEXCOORD2;
};
```

9

```
v2f vert(a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.uv.xy = v.texcoord.xy * _MainTex_ST.xy + _MainTex_ST.zw;
    o.uv.zw = v.texcoord.xy * _BumpMap_ST.xy + _BumpMap_ST.zw;

    // Compute the binormal
    // float3 binormal = cross( normalize(v.normal), normalize(v.tangent.xyz) ) * v.tangent.w;
    // // Construct a matrix which transform vectors from object space to tangent space
    // float3x3 rotation = float3x3(v.tangent.xyz, binormal, v.normal);
    // Or just use the built-in macro
    TANGENT_SPACE_ROTATION;

    // Transform the light direction from object space to tangent space
    o.lightDir = mul(rotation, ObjSpaceLightDir(v.vertex)).xyz;
    // Transform the view direction from object space to tangent space
    o.viewDir = mul(rotation, ObjSpaceViewDir(v.vertex)).xyz;

    return o;
}
```

```
}
```

```

// Get the texel in the normal map
fixed4 packedNormal = tex2D(_BumpMap, i.uv.zw);
fixed3 tangentNormal;
// If the texture is not marked as "Normal map"
// tangentNormal.xy = (packedNormal.xy * 2 - 1) * _BumpScale;
// tangentNormal.z = sqrt(1.0 - saturate(dot(tangentNormal.xy, tangentNormal.xy)));

// Or mark the texture as "Normal map", and use the built-in function
tangentNormal = UnpackNormal(packedNormal);
tangentNormal.xy *= _BumpScale;
tangentNormal.z = sqrt(1.0 - saturate(dot(tangentNormal.xy, tangentNormal.xy)));

fixed3 albedo = tex2D(_MainTex, i.uv).rgb * _Color.rgb;

fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;

fixed3 diffuse = _LightColor0.rgb * albedo * max(0, dot(tangentNormal, tangentLightDir));

fixed3 halfDir = normalize(tangentLightDir + tangentViewDir);
fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0, dot(tangentNormal, halfDir)), _Gloss);

return fixed4(ambient + diffuse + specular, 1.0);
}
```

10

```

fixed4 frag(v2f i) : SV_Target {
    fixed3 tangentLightDir = normalize(i.lightDir);
    fixed3 tangentViewDir = normalize(i.viewDir);

    // Get the texel in the normal map
    fixed4 packedNormal = tex2D(_BumpMap, i.uv.zw);
    fixed3 tangentNormal;
    // If the texture is not marked as "Normal map"
    // tangentNormal.xy = (packedNormal.xy * 2 - 1) * _BumpScale;
    // tangentNormal.z = sqrt(1.0 - saturate(dot(tangentNormal.xy, tangentNormal.xy)));

    // Or mark the texture as "Normal map", and use the built-in function
    tangentNormal = UnpackNormal(packedNormal);
    tangentNormal.xy *= _BumpScale;
    tangentNormal.z = sqrt(1.0 - saturate(dot(tangentNormal.xy, tangentNormal.xy)));

    fixed3 albedo = tex2D(_MainTex, i.uv).rgb * _Color.rgb;

    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;

    fixed3 diffuse = _LightColor0.rgb * albedo * max(0, dot(tangentNormal, tangentLightDir));

    fixed3 halfDir = normalize(tangentLightDir + tangentViewDir);
    fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0, dot(tangentNormal, halfDir)), _Gloss);

    return fixed4(ambient + diffuse + specular, 1.0);
}
```

```

// Get the texel in the normal map
fixed4 packedNormal = tex2D(_BumpMap, i.uv.zw);
fixed3 tangentNormal;
// If the texture is not marked as "Normal map"
// tangentNormal.xy = (packedNormal.xy * 2 - 1) * _BumpScale;
// tangentNormal.z = sqrt(1.0 - saturate(dot(tangentNormal.xy, tangentNormal.xy)));

// Or mark the texture as "Normal map", and use the built-in function
tangentNormal = UnpackNormal(packedNormal);
tangentNormal.xy *= _BumpScale;
tangentNormal.z = sqrt(1.0 - saturate(dot(tangentNormal.xy, tangentNormal.xy)));

fixed3 albedo = tex2D(_MainTex, i.uv).rgb * _Color.rgb;

fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;

fixed3 diffuse = _LightColor0.rgb * albedo * max(0, dot(tangentNormal, tangentLightDir));

fixed3 halfDir = normalize(tangentLightDir + tangentViewDir);
fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0, dot(tangentNormal, halfDir)), _Gloss);

return fixed4(ambient + diffuse + specular, 1.0);
}
```

11 Unity Shader Fallback

Fallback "Specular"

```

// Get the texel in the normal map
fixed4 packedNormal = tex2D(_BumpMap, i.uv.zw);
fixed3 tangentNormal;
// If the texture is not marked as "Normal map"
// tangentNormal.xy = (packedNormal.xy * 2 - 1) * _BumpScale;
// tangentNormal.z = sqrt(1.0 - saturate(dot(tangentNormal.xy, tangentNormal.xy)));

// Or mark the texture as "Normal map", and use the built-in function
tangentNormal = UnpackNormal(packedNormal);
tangentNormal.xy *= _BumpScale;
tangentNormal.z = sqrt(1.0 - saturate(dot(tangentNormal.xy, tangentNormal.xy)));

fixed3 albedo = tex2D(_MainTex, i.uv).rgb * _Color.rgb;

fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;

fixed3 diffuse = _LightColor0.rgb * albedo * max(0, dot(tangentNormal, tangentLightDir));

fixed3 halfDir = normalize(tangentLightDir + tangentViewDir);
fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0, dot(tangentNormal, halfDir)), _Gloss);

return fixed4(ambient + diffuse + specular, 1.0);
}
```




▲図7.15 Bump Scaleの値によるレンダリング結果

2. Cubemap

この章では、Cubemapのレンダリングについて説明します。Cubemapは、6つの面を持つ立方体のテクスチャで、環境映写に使用されます。Cubemapのレンダリングには、Shaderの書き換えが必要です。

1. Cubemapの定義

2. Shaderの書き換え

3. NormalMapWorldSpaceMatの定義

4. Unity Shaderの書き換え (Chapter7-NormalMapWorldSpace Shader)

5. Cubemapのレンダリング

6. Chapter7-NormalMapWorldSpace Shaderの書き換え

7. Shaderの書き換え (v2f)

```
struct v2f {
    float4 pos : SV_POSITION;
    float4 uv : TEXCOORD0;
    float4 TtoW0 : TEXCOORD1;
    float4 TtoW1 : TEXCOORD2;
    float4 TtoW2 : TEXCOORD3;
};
```

3.3.2 Cubemapのレンダリング (float4 TtoW0, TtoW1, TtoW2)

8. Cubemapのレンダリング (w)

```
v2f vert(a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.uv.xy = v.texcoord.xy * _MainTex_ST.xy + _MainTex_ST.zw;
    o.uv.zw = v.texcoord.xy * _BumpMap_ST.xy + _BumpMap_ST.zw;

    float3 worldPos = mul(_Object2World, v.vertex).xyz;
    fixed3 worldNormal = UnityObjectToWorldNormal(v.normal);
    fixed3 worldTangent = UnityObjectToWorldDir(v.tangent.xyz);
    fixed3 worldBinormal = cross(worldNormal, worldTangent) * v.tangent.w;
```

```
// Compute the matrix that transform directions from tangent space to world space
// Put the world position in w component for optimization
o.TtoW0 = float4(worldTangent.x, worldBinormal.x, worldNormal.x, worldPos.x);
o.TtoW1 = float4(worldTangent.y, worldBinormal.y, worldNormal.y, worldPos.y);
o.TtoW2 = float4(worldTangent.z, worldBinormal.z, worldNormal.z, worldPos.z);

return o;
}
```

TtoW0TtoW1TtoW2xyzw

3

```
fixed4 frag(v2f i) : SV_Target {
    // Get the position in world space
    float3 worldPos = float3(i.TtoW0.w, i.TtoW1.w, i.TtoW2.w);
    // Compute the light and view dir in world space
    fixed3 lightDir = normalize(UnityWorldSpaceLightDir(worldPos));
    fixed3 viewDir = normalize(UnityWorldSpaceViewDir(worldPos));

    // Get the normal in tangent space
    fixed3 bump = UnpackNormal(tex2D(_BumpMap, i.uv.zw));
    bump.xy *= _BumpScale;
    bump.z = sqrt(1.0 - saturate(dot(bump.xy, bump.xy)));
    // Transform the normal from tangent space to world space
    bump = normalize(half3(dot(i.TtoW0.xyz, bump), dot(i.TtoW1.xyz, bump), dot(i.TtoW2.xyz, bump)));

    ...
}
```

```

TtoW0TtoW1TtoW2wUnityWorldSpaceLightDir
UnityWorldSpaceViewDirUnpackNormal
Normal map_BumpScaleTtoW0TtoW1TtoW2

```

Unity 4.x Cubemap
Unity Shader
Unity 5.x
Unity Shader
Unity 5.x
19.2

7.2.4 Unity

Normal mapUnityUnpackNormal



▲7.16 UnpackNormal Normal map

```

    00000000000000000000Unity Shader00000000000000000000Normal map000000000000
    00000000Unity00000000000000000000Unity Shader00000000UnpackNormal0000000000
    00000000000000Normal map00000000000000000000

```

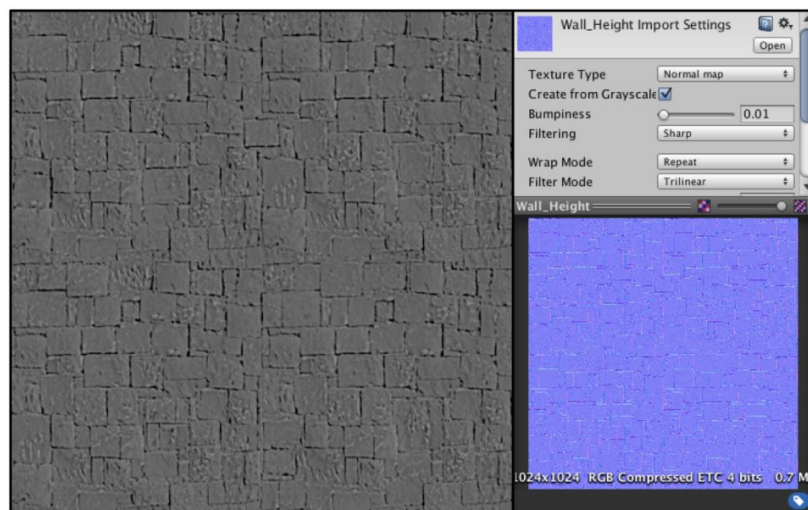
Unity DXT5nm http://tech-artists.org/wiki/Normal_map_compression UnpackNormal
UnityCG.cginc UnpackNormal

```
inline fixed3 UnpackNormalDXT5nm (fixed4 packednormal)
{
    fixed3 normal;
    normal.xy = packednormal.wy * 2 - 1;
    normal.z = sqrt(1 - saturate(dot(normal.xy, normal.xy)));
    return normal;
}

inline fixed3 UnpackNormal(fixed4 packednormal)
{
#ifdef UNITY_NO_DXT5nm
    return packednormal.xyz * 2 - 1;
#else
    return UnpackNormalDXT5nm(packednormal);
#endif
}
```

DXT5nm
 a
 w x g y r b z xy
 DXT5nm
 z
 z

Normal map Create from Grayscale
 Unity Normal map Create from Grayscale
 7.17



▲ 7.17 Create from Grayscale Unity

```

#####Create from Grayscale#####—Bumpiness  Filtering  Bumpiness#####
#####Filtering#####Smooth  #####Sharp  #####
#####Sobel#####Sobel#####3×3#####x×y#####
#####x×y#####
#####r×g#####

```

7.3 □□□□

Team Fortress 2 Valve

Gooch1998A Non-Photorealistic Lighting Model For Automatic Technical Illustration **cool-to-warm tones** 14.1 Unity Shader

000000000000000000000000000000000000000000000000000007.180000



▲ 7.18

2

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □

```
01UnityScene_7_3Unity 5.2
Window -> Lighting -> Skybox
```

2 RampTextureMat

03000000Unity Shader0000000000Unity Shader00Chapter7-RampTexture0000Unity Shader00002000000000

4 Suzanne 2

05000000

Chapter7-RampTexture

1 Shader

```
Shader "Unity Shaders Book/Chapter 7/Ramp Texture" {
```

2 Properties

```
Properties {
    _Color ("Color Tint", Color) = (1,1,1,1)
    _RampTex ("Ramp Tex", 2D) = "white" {}
    Specular ("Specular", Color) = (1, 1, 1, 1)
```

```
}    _Gloss ("Gloss", Range(8.0, 256)) = 20
```

3 SubShader Pass Pass Pass

```
SubShader {  
    Pass {  
        Tags { "LightMode"="ForwardBase" }    }}
```

LightMode Pass Pass Unity

4 CGPROGRAM ENDCG Cg #pragma
Unity vert frag

```
CGPROGRAM  
  
#pragma vertex vert  
#pragma fragment frag
```

5 Unity _LightColor0 Unity Lighting.cginc

```
#include "Lighting.cginc"
```

6 Properties

```
fixed4 _Color;  
sampler2D _RampTex;  
float4 _RampTex_ST;  
fixed4 _Specular;  
float _Gloss;
```

_RampTex _RampTex_ST

7

```
struct a2v {  
    float4 vertex : POSITION;  
    float3 normal : NORMAL;  
    float4 texcoord : TEXCOORD0;  
};  
  
struct v2f {  
    float4 pos : SV_POSITION;  
    float3 worldNormal : TEXCOORD0;  
    float3 worldPos : TEXCOORD1;  
    float2 uv : TEXCOORD2;  
};
```

8

```
v2f vert(a2v v) {  
    v2f o;  
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);  
  
    o.worldNormal = UnityObjectToWorldNormal(v.normal);  
  
    o.worldPos = mul(_Object2World, v.vertex).xyz;
```

```
o.uv = TRANSFORM_TEX(v.texcoord, _RampTex);  
return o;  
}
```

TRANSFORM_TEX

9

```
fixed4 frag(v2f i) : SV_Target {  
    fixed3 worldNormal = normalize(i.worldNormal);  
    fixed3 worldLightDir = normalize(UnityWorldSpaceLightDir(i.worldPos));  
  
    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;  
  
    // Use the texture to sample the diffuse color  
    fixed halfLambert = 0.5 * dot(worldNormal, worldLightDir) + 0.5;  
    fixed3 diffuseColor = tex2D(_RampTex, fixed2(halfLambert, halfLambert)).rgb * _Color.rgb;  
  
    fixed3 diffuse = _LightColor0.rgb * diffuseColor;  
  
    fixed3 viewDir = normalize(UnityWorldSpaceViewDir(i.worldPos));  
    fixed3 halfDir = normalize(worldLightDir + viewDir);  
    fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0, dot(worldNormal, halfDir)), _Gloss);  
  
    return fixed4(ambient + diffuse + specular, 1.0);  
}
```

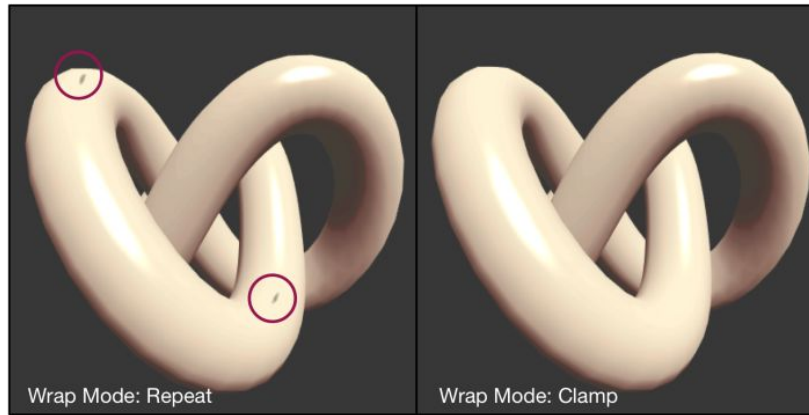
6.4.3 0.5 0.5
halfLambert halfLambert [0,1] halfLambert
_RampTex _RampTex u v
halfLambert _Color

10 Unity Shader Fallback

Fallback "Specular"

Ramp_Texture0.psd Ramp_Texture1.psd

Wrap Mode Clamp 7.19
Wrap Mode Repeat Clamp



▲7.19 Wrap Mode Repeat Clamp

Repeat fixed2(halfLambert,
halfLambert)[0, 1] 1.000 01
Repeat 0.000 01
Wrap Mode Clamp

7.4 遮罩

遮罩 **mask texture** 是用于控制纹理的可见性的。它通常是一个黑白纹理，其中白色表示纹理的可见部分，黑色表示不可见部分。在着色器中，我们通常使用 `texel.r` 来获取纹理的红色通道，并将其与 `0` 进行比较，以确定是否应该显示该像素。

在着色器中，我们通常使用 `texel.r` 来获取纹理的红色通道，并将其与 `0` 进行比较，以确定是否应该显示该像素。

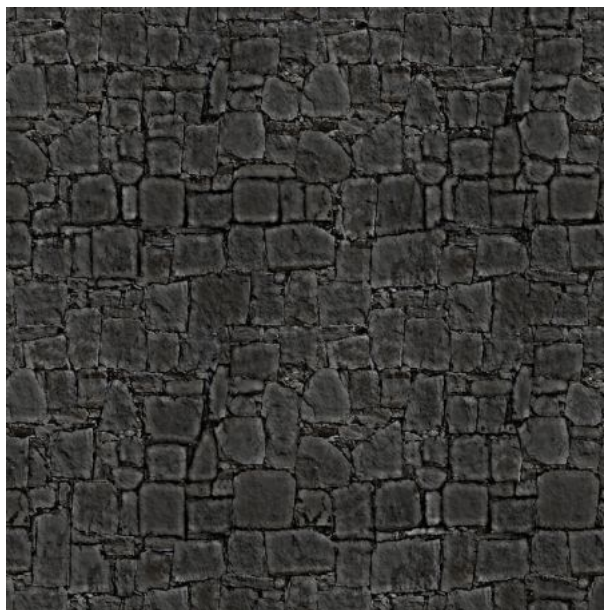
7.4.1 遮罩

在 Unity 7.20 版本中，我们引入了一个新的属性 `maskTexture`，它允许我们在材质中指定一个遮罩纹理。这个属性可以用于控制纹理的可见性，从而实现更复杂的视觉效果。



▲7.20 遮罩纹理的示例

在 Unity 7.21 版本中，我们引入了一个新的属性 `maskTexture`，它允许我们在材质中指定一个遮罩纹理。这个属性可以用于控制纹理的可见性，从而实现更复杂的视觉效果。



▲7.21 遮罩纹理的示例

在 Unity Shader 中，我们通常使用 `maskTexture` 属性来控制纹理的可见性。这个属性可以用于实现更复杂的视觉效果，如遮罩、混合等。

1 Unity Scene_7_4 Unity 5.2 Window -> Lighting -> Skybox

2 MaskTextureMat

3 Unity Shader Unity Shader Chapter7-MaskTexture Unity Shader 2

4 2

5

Chapter7-MaskTexture

1 Shader

```
Shader "Unity Shaders Book/Chapter 7/Mask Texture" {
```

2 Properties

```
Properties {
    _Color ("Color Tint", Color) = (1,1,1,1)
    _MainTex ("Main Tex", 2D) = "white" {}
    _BumpMap ("Normal Map", 2D) = "bump" {}
    _BumpScale("Bump Scale", Float) = 1.0
    _SpecularMask ("Specular Mask", 2D) = "white" {}
    _SpecularScale ("Specular Scale", Float) = 1.0
    _Specular ("Specular", Color) = (1, 1, 1, 1)
    _Gloss ("Gloss", Range(8.0, 256)) = 20
}
```

_SpecularMask _SpecularScale

3 SubShader Pass Pass Pass

```
SubShader {
    Pass {
        Tags { "LightMode"="ForwardBase" }
```

LightMode Pass Pass Unity

4 CGPROGRAM ENDCG Cg #pragma Unity vert frag

```
CGPROGRAM

#pragma vertex vert
#pragma fragment frag
```

5 Unity _LightColor0 Unity Lighting.cginc

```
#include "Lighting.cginc"
```

6 Properties

```
fixed4 _Color;
sampler2D _MainTex;
float4 _MainTex_ST;
sampler2D _BumpMap;
float _BumpScale;
sampler2D _SpecularMask;
float _SpecularScale;
fixed4 _Specular;
float _Gloss;
```

```
        _MainTex_BumpMap_SpecularMask_MainTex_ST
        3
        TextureName_ST
```

7

```
struct a2v {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float4 tangent : TANGENT;
    float4 texcoord : TEXCOORD0;
};

struct v2f {
    float4 pos : SV_POSITION;
    float2 uv : TEXCOORD0;
    float3 lightDir: TEXCOORD1;
    float3 viewDir : TEXCOORD2;
};
```

8

```
v2f vert(a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.uv.xy = v.texcoord.xy * _MainTex_ST.xy + _MainTex_ST.zw;

    TANGENT_SPACE_ROTATION;
    o.lightDir = mul(rotation, ObjSpaceLightDir(v.vertex)).xyz;
    o.viewDir = mul(rotation, ObjSpaceViewDir(v.vertex)).xyz;

    return o;
}
```

9

```
fixed4 frag(v2f i) : SV_Target {
    fixed3 tangentLightDir = normalize(i.lightDir);
    fixed3 tangentViewDir = normalize(i.viewDir);

    fixed3 tangentNormal = UnpackNormal(tex2D(_BumpMap, i.uv));
    tangentNormal.xy *= _BumpScale;
    tangentNormal.z = sqrt(1.0 - saturate(dot(tangentNormal.xy, tangentNormal.xy)));

    fixed3 albedo = tex2D(_MainTex, i.uv).rgb * _Color.rgb;

    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;

    fixed3 diffuse = _LightColor0.rgb * albedo * max(0, dot(tangentNormal, tangentLightDir));

    fixed3 halfDir = normalize(tangentLightDir + tangentViewDir);
```

```

// Get the mask value
fixed specularMask = tex2D(_SpecularMask, i.uv).r * _SpecularScale;
// Compute specular term with the specular mask
fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0, dot(tangentNormal, halfDir)), _Gloss);

return fixed4(ambient + diffuse + specular, 1.0);
}

```

我们使用 `_SpecularMask` 纹理的 `r` 通道乘以 `_SpecularScale` 属性来得到 `specularMask`。

然后我们使用 `specularMask` 乘以 `_LightColor0.rgb` 和 `_Specular.rgb` 的乘积，再乘以 `pow(max(0, dot(tangentNormal, halfDir)), _Gloss)`。

最后我们返回 `fixed4(ambient + diffuse + specular, 1.0)`。

Fallback "Specular"

7.4.2 材质属性

在 `Shader` 中，我们使用 `fixed4` 来定义颜色。在 `Shader` 中，我们使用 `fixed4` 来定义颜色。在 `Shader` 中，我们使用 `fixed4` 来定义颜色。

在 `DOTA 2` 中，我们使用 `fixed4` 来定义颜色。在 `DOTA 2` 中，我们使用 `fixed4` 来定义颜色。在 `DOTA 2` 中，我们使用 `fixed4` 来定义颜色。

8 透明

透明は、Alpha Channel を用いて表現される。Alpha Channel は、0 から 1 の範囲で値を持つ。0 は完全に透明、1 は完全不透明を表す。

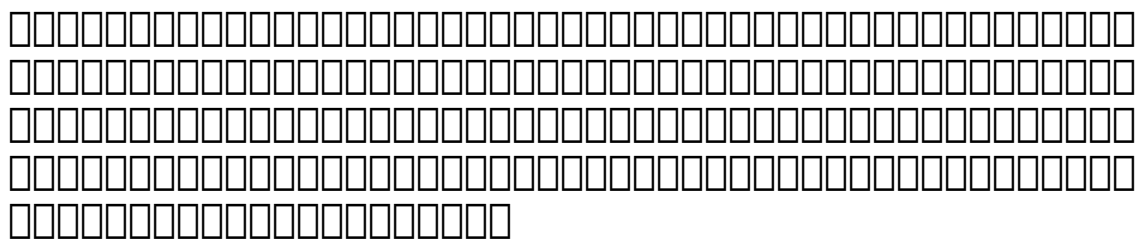
Unity では、Alpha Test と Alpha Blending が提供されている。

Alpha Test は、Alpha Channel の値に基づいて、ピクセルを透明か不透明かに分類する。Alpha Blending は、透明ピクセルと不透明ピクセルの色を混合する。depth buffer と z-buffer は、奥行き情報を保持し、visibility を確保するための仕組みである。

Alpha Blending の基本的な式は、 $A \cdot B + (1 - A) \cdot B$ である。ZWrite は、奥行き書き込みを制御する。

透明ピクセルの処理フローは以下の通り。

- 透明ピクセルの Alpha Channel の値を読み取る。
- Alpha Channel の値が 0 より大きい場合、ピクセルの色を depth buffer に書き込み、visibility を確保する。
- Alpha Channel の値が 0 より小さい場合、ピクセルの色を depth buffer から読み取り、Alpha Blending を行う。



8.1 命题逻辑

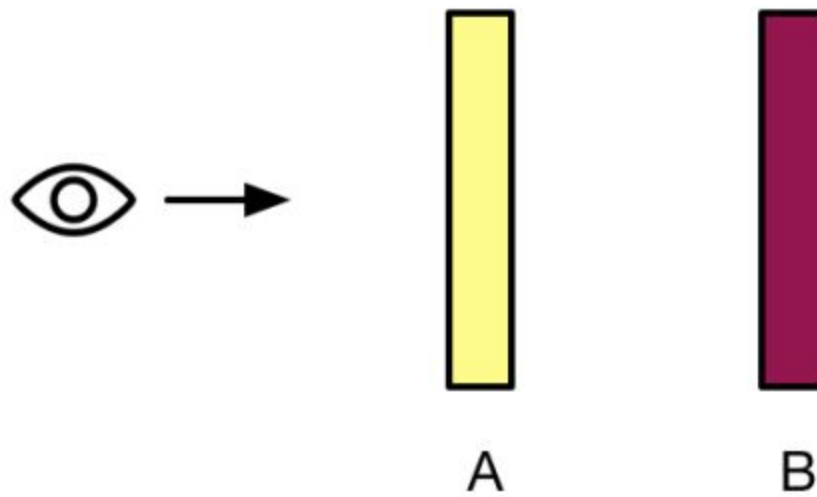
命题逻辑是研究命题及其推理的逻辑。命题是指具有真假值的陈述句。命题逻辑的基本符号包括命题变元、逻辑联结词和量词。命题变元通常用小写字母表示，逻辑联结词包括否定、合取、析取、蕴含和等价。量词包括全称量词和存在量词。命题逻辑的推理规则包括分离规则、假言推理规则、归谬法等。命题逻辑的推理过程通常通过真值表或推理树来完成。命题逻辑在计算机科学、数学和哲学等领域都有广泛的应用。

命题逻辑的基本符号包括命题变元、逻辑联结词和量词。命题变元通常用小写字母表示，逻辑联结词包括否定、合取、析取、蕴含和等价。量词包括全称量词和存在量词。命题逻辑的推理规则包括分离规则、假言推理规则、归谬法等。命题逻辑的推理过程通常通过真值表或推理树来完成。命题逻辑在计算机科学、数学和哲学等领域都有广泛的应用。

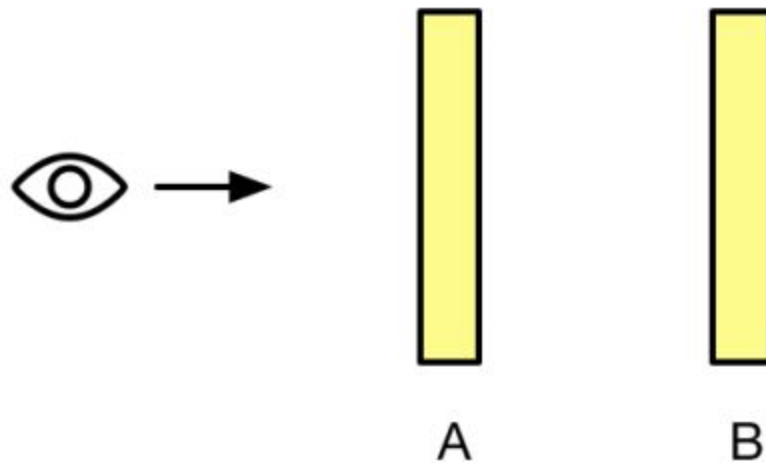
命题逻辑的基本符号包括命题变元、逻辑联结词和量词。命题变元通常用小写字母表示，逻辑联结词包括否定、合取、析取、蕴含和等价。量词包括全称量词和存在量词。命题逻辑的推理规则包括分离规则、假言推理规则、归谬法等。命题逻辑的推理过程通常通过真值表或推理树来完成。命题逻辑在计算机科学、数学和哲学等领域都有广泛的应用。

- 命题逻辑的基本符号包括命题变元、逻辑联结词和量词。命题变元通常用小写字母表示，逻辑联结词包括否定、合取、析取、蕴含和等价。量词包括全称量词和存在量词。命题逻辑的推理规则包括分离规则、假言推理规则、归谬法等。命题逻辑的推理过程通常通过真值表或推理树来完成。命题逻辑在计算机科学、数学和哲学等领域都有广泛的应用。
- 命题逻辑的基本符号包括命题变元、逻辑联结词和量词。命题变元通常用小写字母表示，逻辑联结词包括否定、合取、析取、蕴含和等价。量词包括全称量词和存在量词。命题逻辑的推理规则包括分离规则、假言推理规则、归谬法等。命题逻辑的推理过程通常通过真值表或推理树来完成。命题逻辑在计算机科学、数学和哲学等领域都有广泛的应用。

命题逻辑的基本符号包括命题变元、逻辑联结词和量词。命题变元通常用小写字母表示，逻辑联结词包括否定、合取、析取、蕴含和等价。量词包括全称量词和存在量词。命题逻辑的推理规则包括分离规则、假言推理规则、归谬法等。命题逻辑的推理过程通常通过真值表或推理树来完成。命题逻辑在计算机科学、数学和哲学等领域都有广泛的应用。



▲8.1 □□□□□□□□A□□□□□□□□B□□□□□□□□

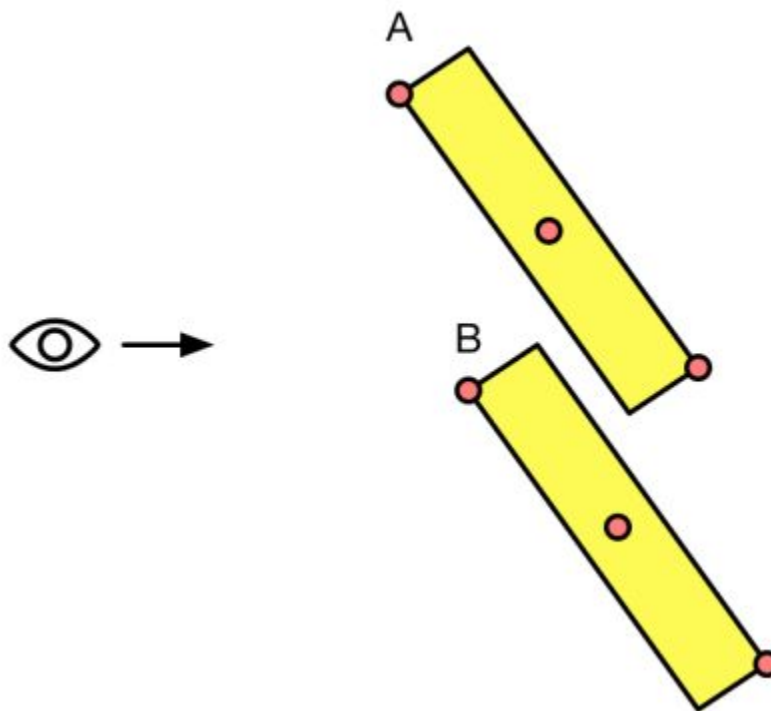


▲ 8.2 □□□□□□□□□□A□B□□□□□□□□

[illegible]

- [illegible]

[illegible]



▲8.4 物体の位置と向きを指定する方法

この章では、物体の位置と向きを指定する方法について説明します。8.4節では、物体の位置と向きを指定する方法について説明します。AとBの位置と向きを指定する方法について説明します。

この章では、物体の位置と向きを指定する方法について説明します。8.5節では、物体の位置と向きを指定する方法について説明します。

この章では、Unityの位置と向きを指定する方法について説明します。

8.2 Unity Shader

Unity render queue SubShader Queue Unity 5 AlphaTest 8.1 5

8.1 Unity 5

Background	1000	
Geometry	2000	
AlphaTest	2450	
Transparent	3000	
Overlay	4000	

```
SubShader {
  Tags { "Queue"="AlphaTest" }
  Pass {
    ...
  }
}
```

}	

[illegible]

```
SubShader {
    Tags { "Queue"="Transparent" }
    Pass {
        ZWrite Off
        ...
    }
}
```

ZWrite Off Pass
 SubShader SubShader Pass

8.3 剪贴测试

剪贴测试是Unity引擎中用于剔除不在视锥体内的几何体的重要技术。

剪贴测试的原理是：对于每个像素，计算其对应的几何体在视锥体内的投影面积。如果投影面积为零，则该几何体不在视锥体内，可以被剔除。否则，该几何体在视锥体内，需要进行渲染。

剪贴测试的实现通常依赖于clip函数。Cg语言提供了clip函数，用于对顶点坐标进行剪贴。

例如，void clip(float4 x); void clip(float3 x); void clip(float2 x); void clip(float1 x); void clip(float x);

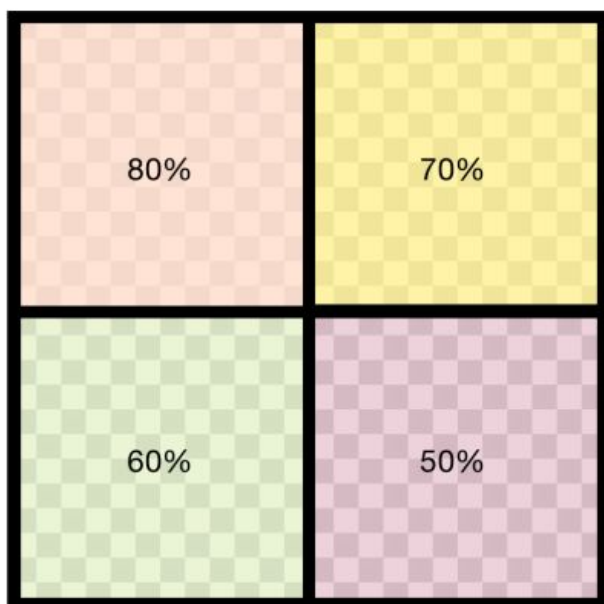
这些函数用于对顶点坐标进行剪贴。

剪贴测试的实现通常依赖于clip函数。Cg语言提供了clip函数，用于对顶点坐标进行剪贴。

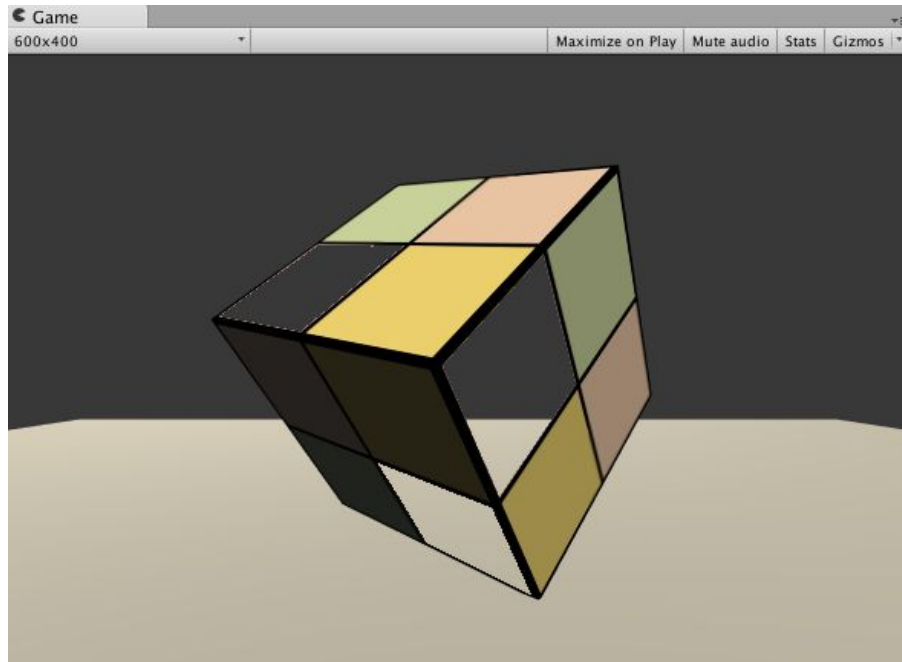
```
void clip(float4 x)
{
    if (any(x < 0))
        discard;
}
```

剪贴测试的原理是：对于每个像素，计算其对应的几何体在视锥体内的投影面积。如果投影面积为零，则该几何体不在视锥体内，可以被剔除。否则，该几何体在视锥体内，需要进行渲染。

剪贴测试的实现通常依赖于clip函数。Cg语言提供了clip函数，用于对顶点坐标进行剪贴。



▲8.5 剪贴测试的原理



▲8.6 透過

透過の仕組み

1. UnityのプロジェクトのAssetsフォルダにScene_8_3というUnity 5.2のシーンを作成し、
 実行メニューからWindow -> Lighting -> Skyboxを選択し、Skyboxを設定する。

2. ShaderのAssetsフォルダにAlphaTestMatを作成する。

3. Unity ShaderのAssetsフォルダにUnity ShaderのChapter8-AlphaTestを作成し、
 Unity Shaderの2番目のShaderを選択する。

4. ShaderのAssetsフォルダに2番目のShaderのChapter8-AlphaTestを作成する。

5. ShaderのAssetsフォルダにChapter8-AlphaTestを作成する。

6. ShaderのAssetsフォルダにChapter8-AlphaTestを作成する。

7. ShaderのAssetsフォルダにChapter8-AlphaTestを作成する。

```
Shader "Unity Shaders Book/Chapter 8/Alpha Test" {
```

2. ShaderのAssetsフォルダにChapter8-AlphaTestを作成する。
 Propertiesの[0, 1]の範囲に
 _Cutoff

```
Properties {
    _Color ("Main Tint", Color) = (1,1,1,1)
    _MainTex ("Main Tex", 2D) = "white" {}
    _Cutoff ("Alpha Cutoff", Range(0, 1)) = 0.5
```

```
}
```

3. `_Cutoff` 0 到 1 的浮点型变量，用于控制透明度的截止值。

4. `SubShader` 块，用于定义子着色器。

```
SubShader {  
    Tags { "Queue"="AlphaTest" "IgnoreProjector"="True" "RenderType"="TransparentCutout" }  
  
    Pass {  
        Tags { "LightMode"="ForwardBase" }  
    }
```

5. `8.2` 版本开始，Unity 引入了 `AlphaTest` 渲染模式，用于实现透明度的测试。在 `SubShader` 块中，我们使用 `Tags` 属性来指定 `Queue` 为 `AlphaTest`，`IgnoreProjector` 为 `True`，以及 `RenderType` 为 `TransparentCutout`。在 `Pass` 块中，我们使用 `Tags` 属性来指定 `LightMode` 为 `ForwardBase`。

6. `CGPROGRAM` 和 `ENDCG` 是 Cg 语言的标记，用于定义着色器程序。在 `CGPROGRAM` 和 `ENDCG` 之间，我们使用 `#pragma` 来指定顶点着色器（`vertex vert`）和片段着色器（`fragment frag`）。

```
CGPROGRAM  
  
#pragma vertex vert  
#pragma fragment frag
```

7. `Unity` 引擎提供了 `_LightColor0` 变量，用于获取光源的颜色。在 `Unity` 引擎中，我们使用 `Lighting.cginc` 文件来定义这些变量。

```
#include "Lighting.cginc"
```

8. `Properties` 块，用于定义着色器的属性。

```
fixed4 _Color;  
sampler2D _MainTex;  
float4 _MainTex_ST;  
fixed _Cutoff;
```

9. `_Cutoff` 0 到 1 的浮点型变量，用于控制透明度的截止值。

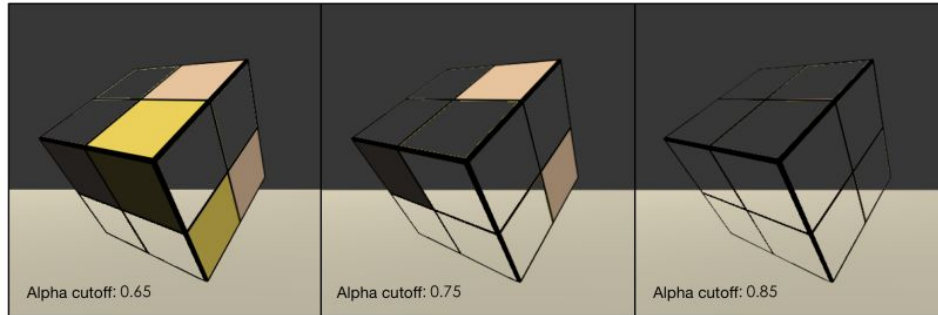
10. `struct a2v` 块，用于定义顶点着色器的输入变量。

```
struct a2v {  
    float4 vertex : POSITION;  
    float3 normal : NORMAL;
```


Fallback "Transparent/Cutout/VertexLit"

DiffuseSpecularTransparent/Cutout/VertexLit Shader SubShader Shader 9.4.5

Alpha cutoff 8.7



▲8.7 Alpha cutoff

8.68.7 “” —— 8.68.7

8.4 混合

混合是指将两个或多个颜色值按照一定的比例混合在一起，以产生新的颜色值。在图形学中，混合通常用于实现透明效果、阴影、反射等。

混合操作通常涉及到源颜色（Src）和目标颜色（Dst）的混合。混合的结果取决于混合因子（Blend Factor）和混合操作（Blend Operation）。

Unity 引擎提供了多种混合模式，包括 Blend、Blend Unity、Blend OneMinusSrcAlpha 等。这些模式可以通过 ShaderLab 中的 Blend 指令来指定。

8.2 ShaderLab 混合

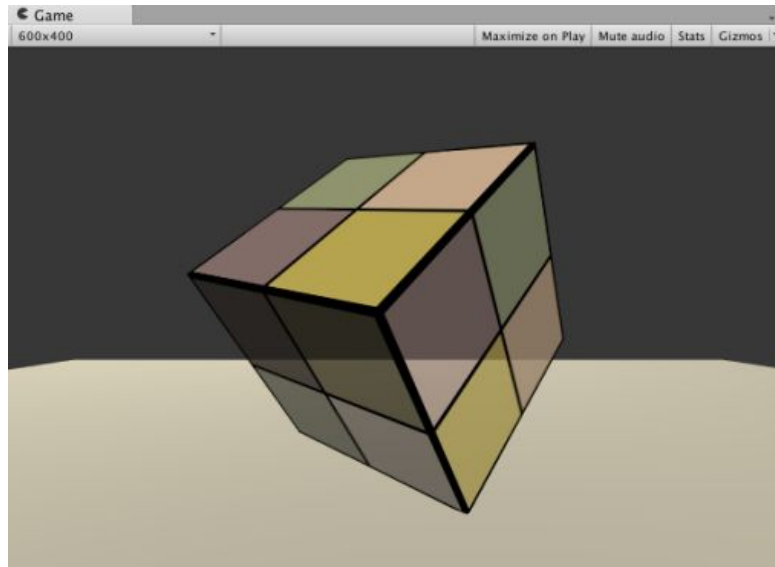
名称	描述
Blend Off	关闭混合
Blend SrcFactor DstFactor	使用 SrcFactor 和 DstFactor 进行混合
Blend SrcFactor DstFactor, SrcFactorA DstFactorA	使用 SrcFactor 和 DstFactor 进行混合，并指定源和目标混合因子
BlendOp BlendOperation	指定混合操作 BlendOperation

在 ShaderLab 中，混合操作通常通过 Blend 指令来指定。例如，Blend SrcFactor DstFactor 表示使用 SrcFactor 和 DstFactor 进行混合。Unity 引擎还提供了 Blend OneMinusSrcAlpha 等模式，用于实现透明效果。

$$DstColor_{new} = SrcAlpha \times SrcColor + (1 - SrcAlpha) \times DstColor_{old}$$

混合操作通常涉及到源颜色（Src）和目标颜色（Dst）的混合。混合的结果取决于混合因子（Blend Factor）和混合操作（Blend Operation）。

8.3 混合操作 8.8 混合操作



▲8.8 半透明

Unityの半透明のレンダリング

1 Unityの半透明のレンダリング Scene_8_4 Unity 5.2の半透明のレンダリング
Window -> Lighting -> Skyboxの半透明のレンダリング

2 半透明のレンダリングのAlphaBlendMat

3 Unity Shaderの半透明のレンダリング Unity Shader Chapter8-AlphaBlendの半透明
Unity Shaderの2つの半透明のレンダリング

4 半透明のレンダリングの2つの半透明のレンダリングの半透明のレンダリング

5 半透明のレンダリング

Chapter8-AlphaBlendの半透明のレンダリング 8.3 Chapter8-AlphaTestの半透明のレンダリング
半透明のレンダリングの半透明のレンダリング

1 Propertiesの半透明のレンダリング

```
Properties {
    _Color ("Main Tint", Color) = (1,1,1,1)
    _MainTex ("Main Tex", 2D) = "white" {}
    _AlphaScale ("Alpha Scale", Range(0, 1)) = 1
}
```

半透明のレンダリングのAlphaScaleの半透明のレンダリングのAlphaScaleの半透明のレンダリング
半透明のレンダリングのPassの半透明のレンダリング

```
fixed4 _Color;
sampler2D _MainTex;
float4 _MainTex_ST;
```

```
fixed _AlphaScale;
```

2 SubShader

```
SubShader {  
    Tags { "Queue"="Transparent" "IgnoreProjector"="True" "RenderType"="Transparent" }
```

Transparent Queue
Transparent RenderType Unity Shader Transparent
Shader Shader RenderType
IgnoreProjector True Shader Projectors
Shader SubShader 3

3 Pass

```
Pass {  
    Tags { "LightMode"="ForwardBase" }  
  
    ZWrite Off  
    Blend SrcAlpha OneMinusSrcAlpha
```

Pass LightMode ForwardBase Unity
Pass ZWrite Off
Pass
SrcAlpha OneMinusSrcAlpha

4

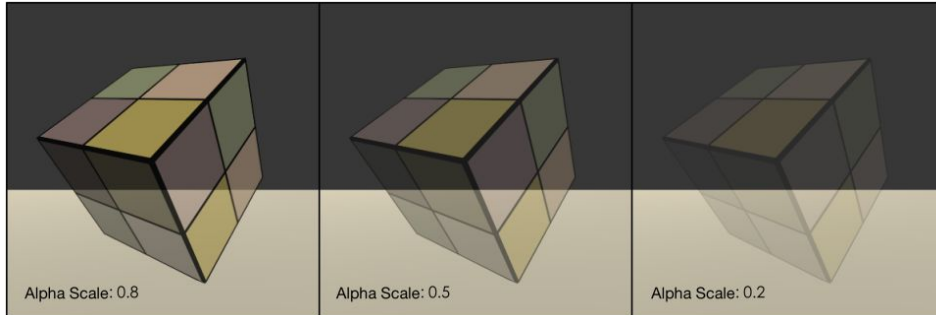
```
fixed4 frag(v2f i) : SV_Target {  
    fixed3 worldNormal = normalize(i.worldNormal);  
    fixed3 worldLightDir = normalize(UnityWorldSpaceLightDir(i.worldPos));  
  
    fixed4 texColor = tex2D(_MainTex, i.uv);  
  
    fixed3 albedo = texColor.rgb * _Color.rgb;  
  
    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;  
  
    fixed3 diffuse = _LightColor0.rgb * albedo * max(0, dot(worldNormal, worldLightDir));  
  
    return fixed4(ambient + diffuse, texColor.a * _AlphaScale);  
}
```

8.3
_AlphaScale Blend

5 Unity Shader Fallback

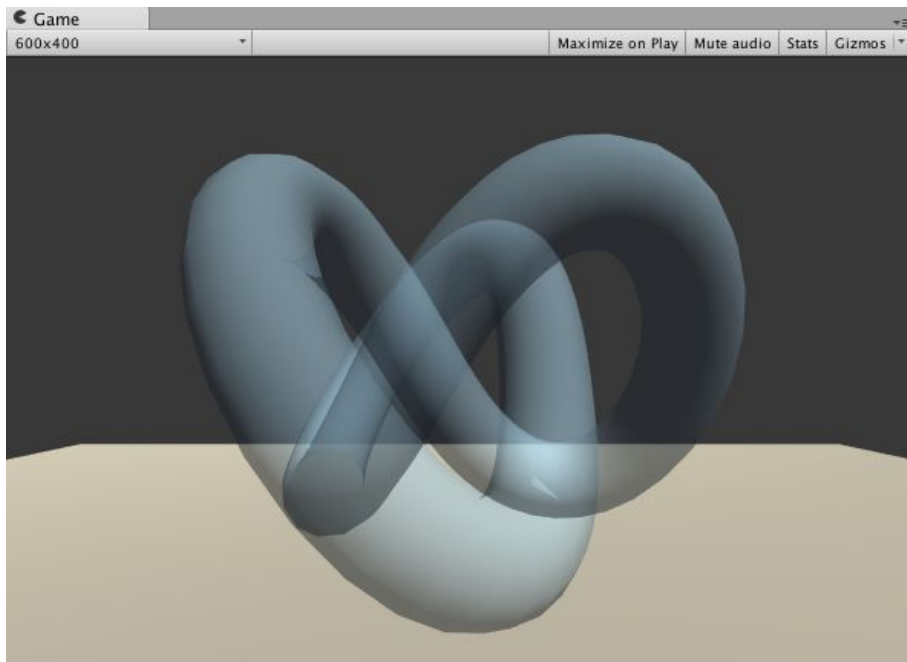
Fallback "Transparent/VertexLit"

Alpha Scale 8.9 Alpha Scale



8.9 Alpha Scale

8.1 8.10 Unity Shader Knot

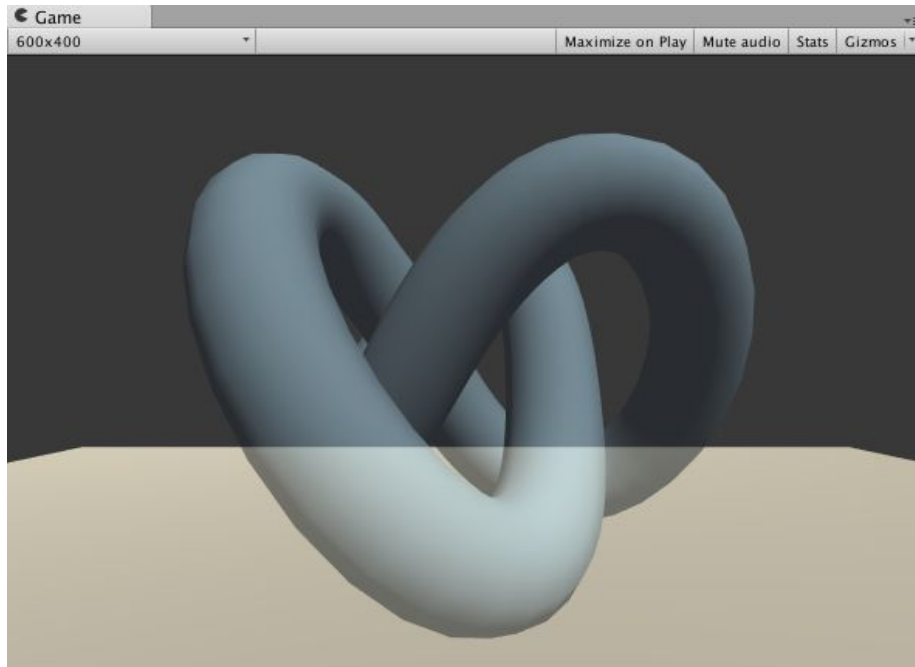


8.10

8.1 “ ”

8.5 透明オブジェクトのレンダリング

8.4の透明オブジェクトのレンダリングは、**Pass** を使用して、
Passを使用して透明オブジェクトのレンダリングを行います。
Passを使用して透明オブジェクトのレンダリングを行います。
Passを使用して透明オブジェクトのレンダリングを行います。
Passを使用して透明オブジェクトのレンダリングを行います。
8.11の透明オブジェクトのレンダリングは、
透明オブジェクトのレンダリングを行います。



▲8.11 透明オブジェクトのレンダリング

透明オブジェクトのレンダリング

1. UnityのプロジェクトをScene_8_5に開き、Unity 5.2のバージョンを確認します。
Window -> Lighting -> Skyboxを選択します。

2. AlphaBlendZWriteMatを選択します。

3. Unity Shaderを選択します。Unity ShaderのChapter8-AlphaBlendZWriteを選択します。Unity Shaderの2を選択します。

4. 透明オブジェクトのレンダリングは、2の透明オブジェクトのレンダリングを行います。

5. 透明オブジェクト

8.4の透明オブジェクトのレンダリングは、Chapter8-AlphaBlendの透明オブジェクトのレンダリングを行います。
Chapter8-AlphaBlendZWriteの透明オブジェクトのレンダリングを行います。Passを使用して透明オブジェクトのレンダリングを行います。

8.6 ShaderLab

8.4 Blend

source color **destination color** **S** **D** **O** RGBA RGB

Unity Blend Blend Off API OpenGL glEnable(GL_BLEND) Unity

8.6.1

2.3.8

blend equation RGB A RGB A 4 8.3 ShaderLab

8.3 ShaderLab

項目	説明
Blend SrcFactor DstFactor	SrcFactorとDstFactorの値を指定する。値は0から1の範囲である。
Blend SrcFactor DstFactor, SrcFactorA DstFactorA	SrcFactorAとDstFactorAの値を指定する。値は0から1の範囲である。

Blend SrcFactor DstFactor, SrcFactorA DstFactorAの値を指定する。値は0から1の範囲である。

$$O_{rgb} = SrcFactor \times S_{rgb} + DstFactor \times D_{rgb}$$

$$O_a = SrcFactorA \times S_a + DstFactorA \times D_a$$

8.4 ShaderLabのBlendモード

8.4 ShaderLabのBlendモード

項目	説明
One	値は1
Zero	値は0
SrcColor	SrcColorのRGB成分を指定する。値は0から1の範囲である。
SrcAlpha	SrcColorのA成分を指定する。値は0から1の範囲である。

項目	説明
DstColor	目的側のRGB値とSrcColorのRGB値を、 目的側のAlpha値とSrcColorのAlpha値を、
DstAlpha	目的側のAlpha値
OneMinusSrcColor	(1-SrcColor)のRGB値とSrcColorのRGB値を、 (1-SrcAlpha)のAlpha値とSrcColorのAlpha値を、
OneMinusSrcAlpha	(1-SrcAlpha)
OneMinusDstColor	(1-DstColor)のRGB値とDstColorのRGB値を、 (1-DstAlpha)のAlpha値とDstColorのAlpha値を、
OneMinusDstAlpha	(1-DstAlpha)

目的側のRGB値とSrcColorのAlpha値を、
目的側のAlpha値とSrcColorのAlpha値を、
Blend SrcFactor DstFactor, SrcFactorA DstFactorA 目的側のAlpha値とSrcColorのAlpha値を、
目的側のAlpha値とSrcColorのAlpha値を、

Blend SrcAlpha OneMinusSrcAlpha, One Zero

8.6.2 目的側

目的側のAlpha値とSrcColorのAlpha値を、
目的側のAlpha値とSrcColorのAlpha値を、
ShaderLabの**BlendOp BlendOperation** 目的側のAlpha値とSrcColorのAlpha値を8.5
ShaderLabの目的側のAlpha値

8.5 ShaderLabの目的側

項目	説明
Add	加算混合 $O_{rgb} = SrcFactor \times S_{rgb} + DstFactor \times D_{rgb}$ $O_a = SrcFactor A \times S_a + DstFactor A \times D_a$
Sub	減算混合 $O_{rgb} = SrcFactor \times S_{rgb} - DstFactor \times D_{rgb}$ $O_a = SrcFactor A \times S_a - DstFactor A \times D_a$
RevSub	反転減算混合 $O_{rgb} = DstFactor \times D_{rgb} - SrcFactor \times S_{rgb}$ $O_a = DstFactor A \times D_a - SrcFactor A \times S_a$
Min	最小値混合 $O_{rgba} = (\min(S_r, D_r), \min(S_g, D_g), \min(S_b, D_b), \min(S_a, D_a))$
Max	最大値混合 $O_{rgba} = (\max(S_r, D_r), \max(S_g, D_g), \max(S_b, D_b), \max(S_a, D_a))$
備考	DirectX 11.1

Min Max の値は、0 から 1 の範囲で指定します。

8.6.3 乗算混合

Photoshop の乗算混合モード

```

// 乗算混合
Blend SrcAlpha OneMinusSrcAlpha

// ソフト加算混合
Blend OneMinusDstColor One

// 乗算混合

```

```
Blend DstColor Zero

// 2x Multiply
Blend DstColor SrcColor

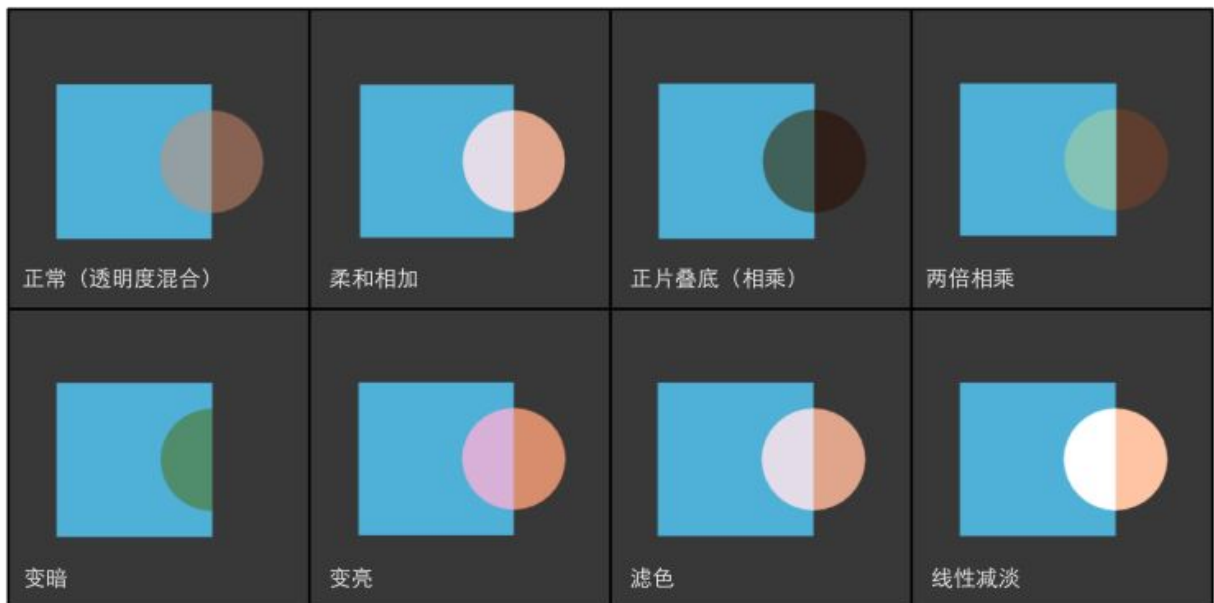
// Darken
BlendOp Min
Blend One One

// Lighten
BlendOp Max
Blend One One

// Screen
Blend OneMinusDstColor One
// 
Blend One OneMinusSrcColor

// Linear Dodge
Blend One One
```

8.12 Scene_8_6_3



▲8.12 □□□□□□□□□□□□□□□□

MinMaxBlendOp Add

8.7 背面剔除

背面剔除是渲染过程中的一项重要技术，用于提高渲染效率。在Unity中，可以通过设置Cull属性来实现背面剔除。Cull属性的取值可以是Back、Front或Off。Back表示只渲染正面，Front表示只渲染背面，Off表示不进行剔除。

```
Cull Back | Front | Off
```

在Unity中，可以通过设置Cull属性来控制剔除。例如，将Cull设置为Back，则只会渲染物体的正面。如果设置为Front，则只会渲染物体的背面。如果设置为Off，则不会进行剔除。

8.7.1 背面剔除

在Unity Shader中，可以通过设置Cull属性来控制剔除。例如，在AlphaTestBothSidedMat Shader中，可以通过设置Cull为Off来禁用剔除。在Chapter8-AlphaTestBothSided Shader中，可以通过设置Cull为Off来禁用剔除。

```
Pass {
    Tags { "LightMode"="ForwardBase" }

    // Turn off culling
    Cull Off
}
```

在Unity Shader中，可以通过设置Cull属性来控制剔除。例如，在Chapter8-AlphaTestBothSided Shader中，可以通过设置Cull为Off来禁用剔除。


```

    Pass {
        Tags { "LightMode"="ForwardBase" }

        // First pass renders only back faces
        Cull Front

        // 背面のみ描画
    }

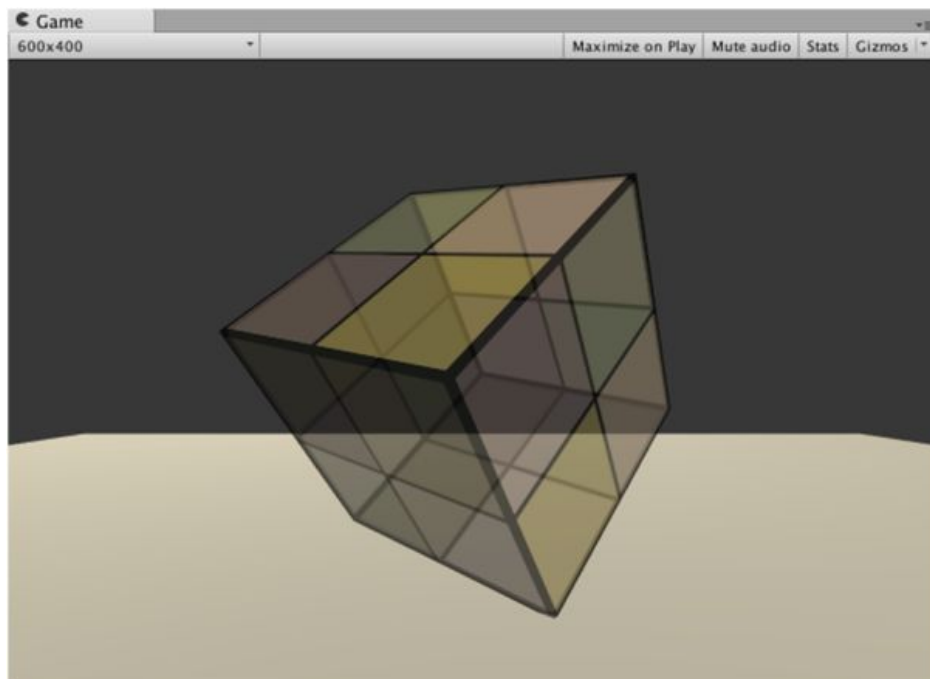
    Pass {
        Tags { "LightMode"="ForwardBase" }

        // Second pass renders only front faces
        Cull Back

        // 前面のみ描画
    }
}
Fallback "Transparent/VertexLit"
}

```

透明な立方体のレンダリング8.14



▲8.14 透明な立方体のレンダリング

3 3

Unity
3

9

Unity 3 3
Unity
Shader

10

Unity Shader

11

Shader

9 本章小结

本章主要介绍了Unity Shader的编写方法，包括Shader的编写、编译和运行。本章还介绍了Unity Shader的优化方法，包括Shader的编译选项、Shader的编译时间、Shader的运行时间等。本章还介绍了Unity Shader的调试方法，包括Shader的编译选项、Shader的编译时间、Shader的运行时间等。

本章还介绍了Unity Shader的优化方法，包括Shader的编译选项、Shader的编译时间、Shader的运行时间等。本章还介绍了Unity Shader的调试方法，包括Shader的编译选项、Shader的编译时间、Shader的运行时间等。

本章还介绍了Unity Shader的优化方法，包括Shader的编译选项、Shader的编译时间、Shader的运行时间等。本章还介绍了Unity Shader的调试方法，包括Shader的编译选项、Shader的编译时间、Shader的运行时间等。

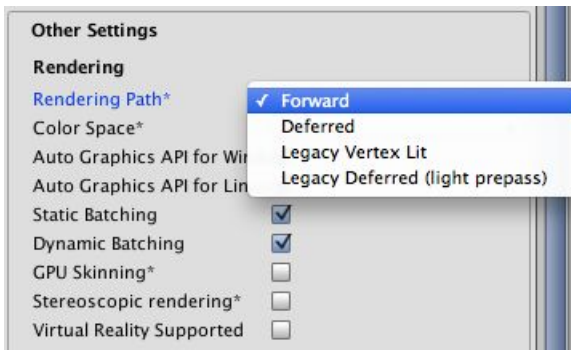
9.1 Unity

`Unity.Renderer.RenderPath` の中で Unity Shader Pass を呼び出す処理が、`UnityShaderPass` というクラスで定義されている。このクラスは、`UnityShaderPass` という名前空間に属している。また、このクラスは、`UnityShaderPass` という名前空間に属している。

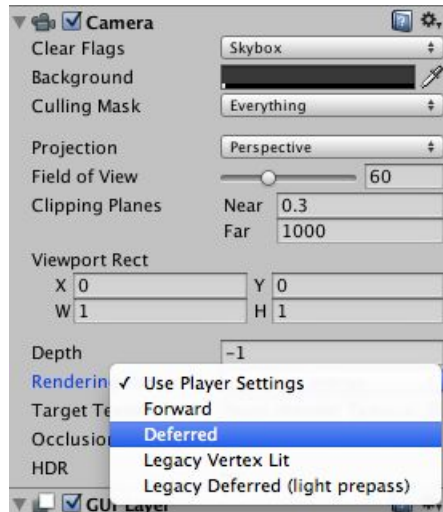
Unity 5.0 3D Forward
Rendering Path Deferred Rendering Path
Vertex Lit Rendering Path Unity 5.0 Unity
 Unity Shader

Unity Edit → Project Settings → Player → Other Settings → Rendering Path → **Deferred** → **9.1**

A B
Project Settings 9.2



▲9.1 Unity



▲9.2 Rendering PathとProject Settings

UnityのProject Settingsには、Rendering Pathの設定があります。この設定は、UnityのGPUに最適化されたレンダリングパスを選択するためのものです。

レンダリングパスは、Passの順序で構成されます。Passの順序は、LightModeによって決定されます。

```
Pass {
    Tags { "LightMode" = "ForwardBase" }
```

UnityのPassは、ForwardBase、ForwardAdd、Deferred、ShadowCasterなどがあります。ForwardBaseは、9.1のPassのLightModeで定義されています。

9.1 LightMode

Pass	説明
Always	すべてのPassに適用されます。
ForwardBase	ForwardBase Passの最初のPassです。SHやLightmapsを使用します。
ForwardAdd	ForwardBase Passの2番目のPassです。
Deferred	Deferred Passの最初のPassです。G-bufferを使用します。
ShadowCaster	ShadowCaster Passの最初のPassです。shadowmapを使用します。

名前	説明
PrepassBase	Prepassの基底Pass
PrepassFinal	Prepassの最終Pass
VertexLMRGBM, VertexLM	Vertexのシェディング

Unityのシェーダーは、Passで構成されています。Unityは、シェーダーを「Vertex Pass」「Fragment Pass」のように分類しています。Unity 5.xでは、Vertex PassとFragment Passの区別がなくなり、Vertex PassとFragment Passの両方を「Pass」として扱うようになりました。

Unityのシェーダーは、Vertex PassとFragment Passの両方から構成されています。

9.1.1 シェーダー

Unityのシェーダーは、Unity Shaderとして記述されます。

1. シェーダー

Unityのシェーダーは、Vertex PassとFragment Passの両方から構成されています。

```

Pass {
    for (each primitive in this model) {
        for (each fragment covered by this primitive) {
            if (failed in depth test) {
                // 深度テストに失敗
                discard;
            } else {
                // シェーディング
                float4 color = Shading(materialInfo, pos, normal, lightDir, viewDir);
                // フレームバッファに書き込み
                writeFrameBuffer(fragment, color);
            }
        }
    }
}

```


- `Unity Shader` `Base Pass` `Base Pass` `Additional Pass` `Base Pass` `Base Pass` `Additional Pass` `Base Pass` `Additional Pass`

9.4 `Pass` `Unity` `Pass` `LightColor0` `Unity` `Base Pass` `Additional Pass`

3

`Pass` `LightMode` `Unity` `Shader`

`Unity 5` `LightMode` `ForwardBase` `ForwardAdd` `9.2` `Shader`

9.2

変数名	型	説明
<code>_LightColor0</code>	<code>float4</code>	<code>Pass</code>
<code>_WorldSpaceLightPos0</code>	<code>float4</code>	<code>_WorldSpaceLightPos0.xyz</code> <code>Pass</code> <code>_WorldSpaceLightPos0.w</code> 0 1
<code>_LightMatrix0</code>	<code>float4x4</code>	cookie <code>attenuation</code>
<code>unity_4LightPosX0</code> , <code>unity_4LightPosY0</code> , <code>unity_4LightPosZ0</code>	<code>float4</code>	<code>Base Pass</code> 4
<code>unity_4LightAtten0</code>	<code>float4</code>	<code>Base Pass</code> 4
<code>unity_LightColor</code>	<code>half4[4]</code>	<code>Base Pass</code> 4

6.6 `WorldSpaceLightDir` `UnityWorldSpaceLightDir` `ObjSpaceLightDir` 9.3

9.3

変数名	説明
float3 WorldSpaceLightDir (float4 v)	ワールド空間の照明方向ベクトル UnityWorldSpaceLightDir
float3 UnityWorldSpaceLightDir (float4 v)	ワールド空間の照明方向ベクトル UnityWorldSpaceLightDir
float3 ObjSpaceLightDir (float4 v)	オブジェクト空間の照明方向ベクトル ObjSpaceLightDir
float3 Shade4PointLights (...)	4点光源の位置と色、減衰係数 unity_4LightPosX0, unity_4LightPosY0, unity_4LightPosZ0 unity_LightColor, unity_4LightAtten0

この変数は、Unityのバージョンによって、異なる値を持つ場合があります。

9.1.2 変数名

Unityのバージョンによって、変数名が変更された場合があります。Unity 5.0では、Unity 4.7までの変数名が使用されています。Unity 5.0では、Unity 4.7までの変数名が使用されています。

1.Unityのバージョン

Unityのバージョンによって、変数名が変更された場合があります。Unity 5.0では、Unity 4.7までの変数名が使用されています。

Unity 5.0では、Unity 4.7までの変数名が使用されています。Unity 5.0では、Unity 4.7までの変数名が使用されています。

2.変数名

Unityのバージョンによって、変数名が変更された場合があります。Unity 5.0では、Unity 4.7までの変数名が使用されています。

9.4 点光源 (Point Light)

変数名	型	説明
unity_LightColor	half4[8]	色
unity_LightPosition	float4[8]	xyz座標 (z座標は0から1)
unity_LightAtten	half4[8]	点光源の照射範囲 (spotAngle/2) の cos 値、 $1/\cos(\text{spotAngle}/4)$ の逆余弦値、x座標 - 1、y座標 1、z座標 w座標
unity_SpotDirection	float4[8]	点光源の照射方向 (0, 0, 1, 0)

点光源の照射範囲 (spotAngle/2) の cos 値、 $1/\cos(\text{spotAngle}/4)$ の逆余弦値、x座標 - 1、y座標 1、z座標 w座標

9.5 点光源 (Point Light)

9.5 点光源 (Point Light)

変数名	型
float3 ShadeVertexLights (float4 vertex, float3 normal)	点光源の照射範囲 (spotAngle/2) の cos 値、 $1/\cos(\text{spotAngle}/4)$ の逆余弦値、x座標 - 1、y座標 1、z座標 w座標
float3 ShadeVertexLightsFull (float4 vertex, float3 normal, int lightCount, bool spotLight)	点光源の照射範囲 (spotAngle/2) の cos 値、 $1/\cos(\text{spotAngle}/4)$ の逆余弦値、x座標 - 1、y座標 1、z座標 w座標

9.1.3 点光源

点光源の照射範囲 (spotAngle/2) の cos 値、 $1/\cos(\text{spotAngle}/4)$ の逆余弦値、x座標 - 1、y座標 1、z座標 w座標

点光源の照射範囲 (spotAngle/2) の cos 値、 $1/\cos(\text{spotAngle}/4)$ の逆余弦値、x座標 - 1、y座標 1、z座標 w座標

1 点光源

```
00000000Pass0000Pass00000000000000000000000000000000  
0000000000000000000000000000G0000000000Pass0000G00000  
000000000000000000000000000000
```

□ □

```

Pass 1 {
    // Pass primitive
    // primitiveG

    for (each primitive in this model) {
        for (each fragment covered by this primitive) {
            if (failed in depth test) {
                // 
                discard;
            } else {
                // 
                // primitiveG
                writeGBuffer(materialInfo, pos, normal, lightDir, viewDir);
            }
        }
    }
}

Pass 2 {
    // G
    for (each pixel in the screen) {
        if (the pixel is valid) {
            // 
            // primitiveG
            readGBuffer(pixel, materialInfo, pos, normal, lightDir, viewDir);

            // 
            float4 color = Shading(materialInfo, pos, normal, lightDir, viewDir);
            // 
            writeFrameBuffer(pixel, color);
        }
    }
}

```

```

    Pass
    2D

```

2 Unity

UnityUnity 5Unity5.x

Unity 5
Standard Shader

<http://docs.unity3d.com/Manual/RenderTech-DeferredLighting.html> 这篇文章

这篇文章主要介绍了Unity的Deferred Lighting渲染技术，包括其原理、实现以及相关的性能优化。

- 这篇文章主要介绍了anti-aliasing技术
- 这篇文章主要介绍了
- 这篇文章主要介绍了MRT (Multiple Render Targets) Shader Mode 3.0技术

Unity的Deferred Lighting渲染技术

1. Deferred Pass: 在Deferred Pass中，我们首先渲染所有物体，并将它们的颜色、法线、深度等信息存储到GBuffer中。GBuffer是Deferred Lighting渲染技术的关键，它存储了所有物体在渲染过程中需要的信息。

2. Deferred Pass: 在Deferred Pass中，我们首先渲染所有物体，并将它们的颜色、法线、深度等信息存储到GBuffer中。GBuffer是Deferred Lighting渲染技术的关键，它存储了所有物体在渲染过程中需要的信息。

3. Deferred Pass: 在Deferred Pass中，我们首先渲染所有物体，并将它们的颜色、法线、深度等信息存储到GBuffer中。GBuffer是Deferred Lighting渲染技术的关键，它存储了所有物体在渲染过程中需要的信息。

- RT0: ARGB32 (RGBA) 存储物体的颜色、法线、深度等信息
- RT1: ARGB32 (RGBA) 存储物体的颜色、法线、深度等信息
- RT2: ARGB2101010 (RGBA) 存储物体的颜色、法线、深度等信息
- RT3: ARGB32 (RGBA) HDR (ARGBHalf) HDR (ARGBHalf) + lightmap + reflection probes
- RT4: ARGB32 (RGBA) HDR (ARGBHalf) HDR (ARGBHalf) + lightmap + reflection probes

Unity的Deferred Lighting渲染技术，主要依赖于GBuffer和Deferred Shading。GBuffer存储了所有物体在渲染过程中需要的信息，而Deferred Shading则利用这些信息来计算每个像素的最终颜色。这篇文章详细介绍了Unity的Deferred Lighting渲染技术，包括其原理、实现以及相关的性能优化。

3. Deferred Lighting

9.6 Deferred Lighting: 在Deferred Lighting中，我们首先渲染所有物体，并将它们的颜色、法线、深度等信息存储到GBuffer中。GBuffer是Deferred Lighting渲染技术的关键，它存储了所有物体在渲染过程中需要的信息。

9.6 Deferred Lighting

名称	数据类型	描述
_LightColor	float4	光源颜色
_LightMatrix0	float4x4	光源矩阵，用于计算cookie的投影

9.1.4

Unity <http://docs.unity3d.com/Manual/RenderingPaths.html>

Unity

□□□□□□□□□□Unity□□□□□□□□□□

9.2 Unityのライティング

Unityのライティングは、Unity Shaderのライティングモデルによって決定される。Unityのライティングモデルは、**point light**、**spot light**、**area light**の3種類がある。

Unityのライティングモデルは、4種類のライティングモデルに分かれる。そのうち、**area light**は、Unityのライティングモデルの中で、最もリアルなライティングモデルである。

9.2.1 ライティングモデル

Unityのライティングモデルは、Shaderのライティングモデルによって決定される。Unityのライティングモデルは、5種類のライティングモデルに分かれる。

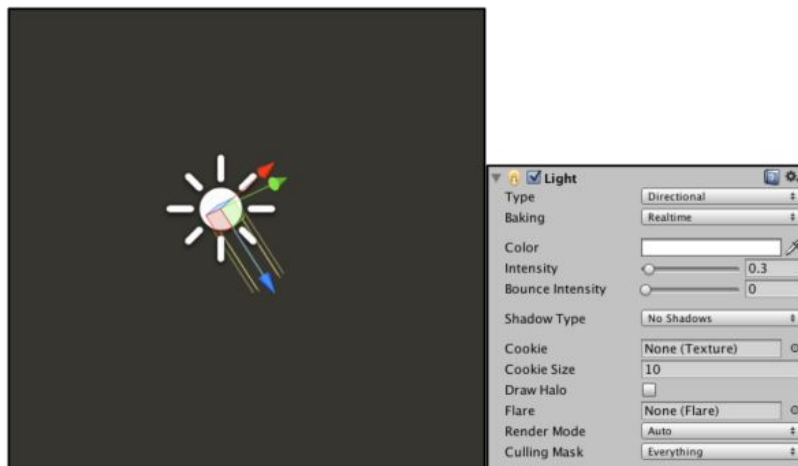
1. ライティングモデル

Unityのライティングモデルは、9.5のUnityのSceneのLightのライティングモデルである。

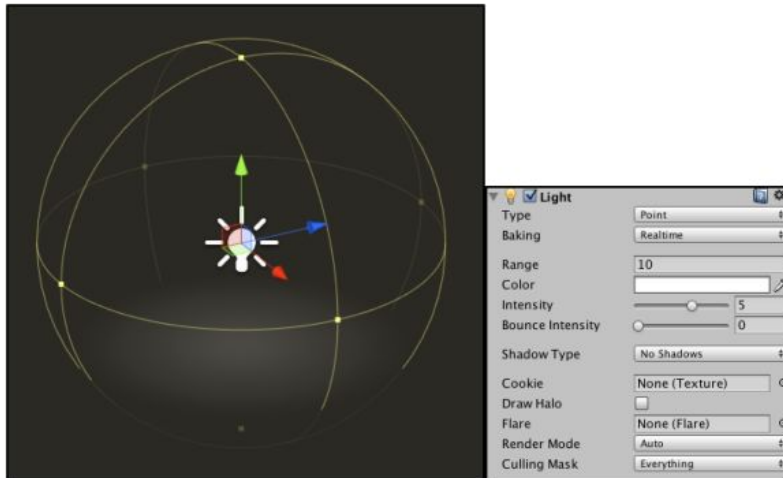
Unityのライティングモデルは、TransformのRotationのライティングモデルである。

2. ライティングモデル

Unityのライティングモデルは、9.6のUnityのSceneのLightのライティングモデルである。



▲9.5 ライティングモデル



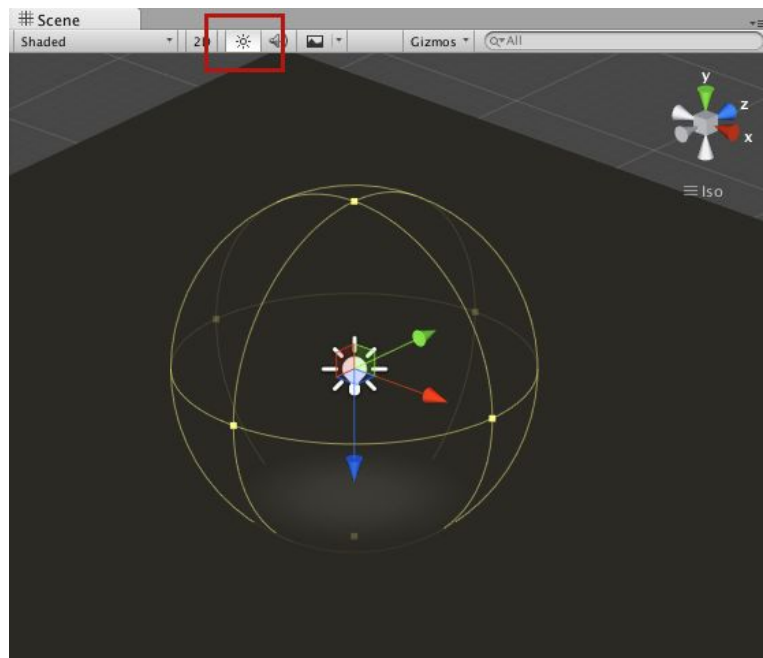
▲9.6 点光源

点光源の位置と範囲をSceneビューで確認する方法は、9.7のSceneビューで確認する。

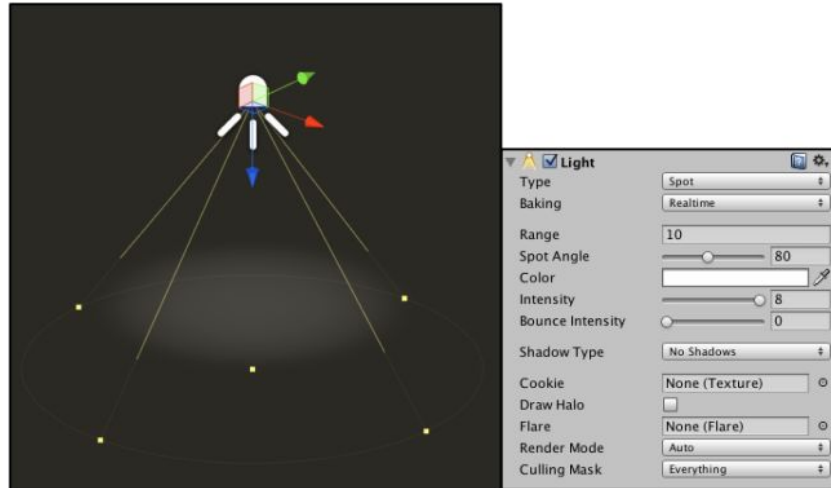
点光源の範囲をRangeプロパティでSceneビューで確認する方法は、TransformのPositionプロパティで確認する。点光源の位置はLightのPositionプロパティで確認する。点光源の範囲は0で確認する。

3. 点光源

点光源の位置と範囲を確認する方法は、9.8のUnityのSceneビューで確認する。LightのInspectorで確認する。



▲9.7 Sceneビューの確認



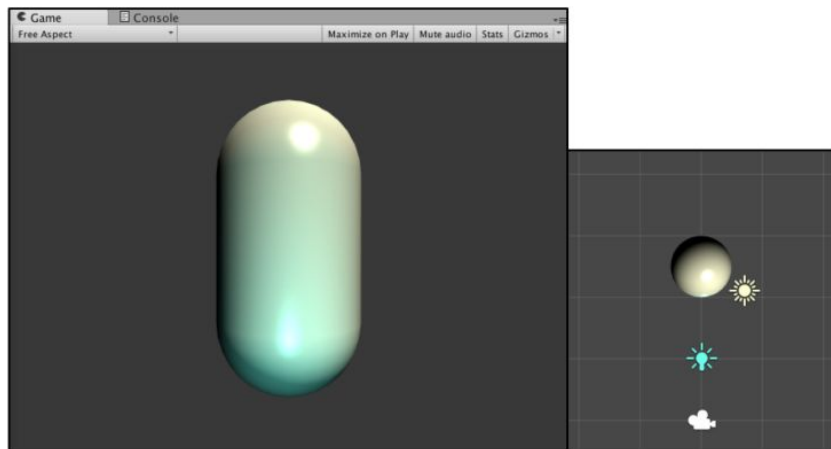
▲9.8 スポットライト

Rangeはライトの範囲、Spot Angleはスポットライトの角度、Sceneはシーン、Transformは変換、Positionは位置、Intensityは強度、Bounce Intensityは跳ね返り強度、Shadow Typeは影の種類、CookieはCookie、Draw Haloはハロを引くかどうか、Flareはフレア、Render Modeはレンダリングモード、Culling Maskはカullingマスク。

9.2.2 環境光

Unity Shaderの5番目のパラメータは、環境光。環境光は、シーン全体の明るさを決める。

9.9環境光



▲9.9 環境光

1. 環境光

環境光は、シーン全体の明るさを決める。

1. UnityのInspectorでScene_9_2_2_1のInspectorで、Window → Lighting → Skyboxを選択。

2ForwardRenderingMat

3Unity ShaderShaderChapter9-ForwardRenderingUnity Shader2

42

5

6

Blinn-PhongBase PassAdditional Pass

1Pass—Base PassPass

```
Pass {
    // Pass for ambient light & first pixel light (directional light)
    Tags { "LightMode"="ForwardBase" }

    CGPROGRAM

    // Apparently need to add this declaration
    #pragma multi_compile_fwdbase
```

#pragma #pragma multi_compile_fwdbase Shader

2Base Pass

```
// Get ambient term
fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;
```

Additional Pass

3Base PassUnityBase PassAdditional PassBase Pass5Base PassWorldSpaceLightPos0_LightColor0_LightColor01.0

```
// Compute diffuse term
fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb * max(0, dot(worldNormal, worldLightDir));

...

// Compute specular term
fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0, dot(worldNormal, halfDir)), _Gloss);

// The attenuation of directional light is always 1
fixed atten = 1.0;

return fixed4(ambient + (diffuse + specular) * atten, 1.0);
```

Base Pass

4 Additional Pass Pass

```
Pass {
    // Pass for other pixel lights
    Tags { "LightMode"="ForwardAdd" }

    Blend One One

    CGPROGRAM

    // Apparently need to add this declaration
    #pragma multi_compile_fwdadd
```

#pragma multi_compile_fwdadd Additional Pass Base Pass Blend Additional Pass Blend Additional Pass **Blend One One** Unity **Blend SrcAlpha One**

5 Additional Pass Base Pass Base Pass Additional Pass Base Pass SH Additional Pass Additional Pass Additional Pass 5 — — LightColor0

```
#ifdef USING_DIRECTIONAL_LIGHT
    fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz);
#else
    fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz - i.worldPosition.xyz);
#endif
```

#ifdef USING_DIRECTIONAL_LIGHT Pass Unity USING_DIRECTIONAL_LIGHT _WorldSpaceLightPos0.xyz _WorldSpaceLightPos0.xyz

6

```
#ifdef USING_DIRECTIONAL_LIGHT
    fixed atten = 1.0;
#else
    float3 lightCoord = mul(_LightMatrix0, float4(i.worldPosition, 1)).xyz;
    fixed atten = tex2D(_LightTexture0, dot(lightCoord, lightCoord).rr).UNITY_ATTEN_CHANNEL;
#endif
```

USING_DIRECTIONAL_LIGHT 1.0 Unity Lookup Table LUT Unity 9.3

9.5 Unity Shader

2 Base Pass Additional Pass

9.1.1 Unity SH

1 Unity Scene_9_2_2_2 Unity 5.2 Window -> Lighting -> Skybox

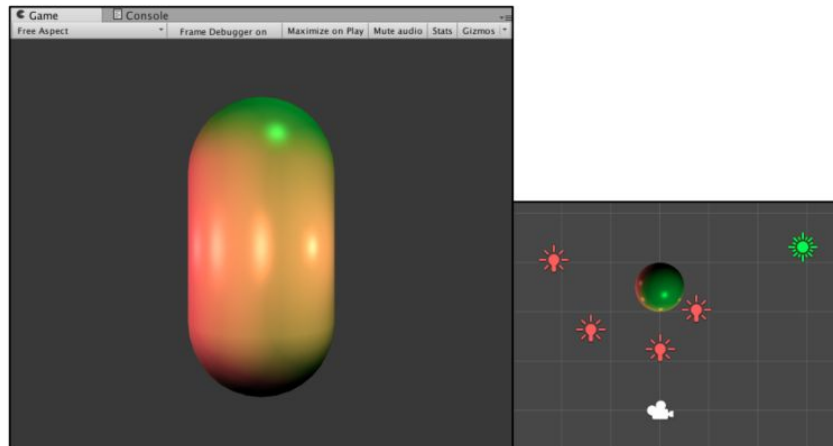
2

3 ForwardRenderingMat

4 4

5

9.10

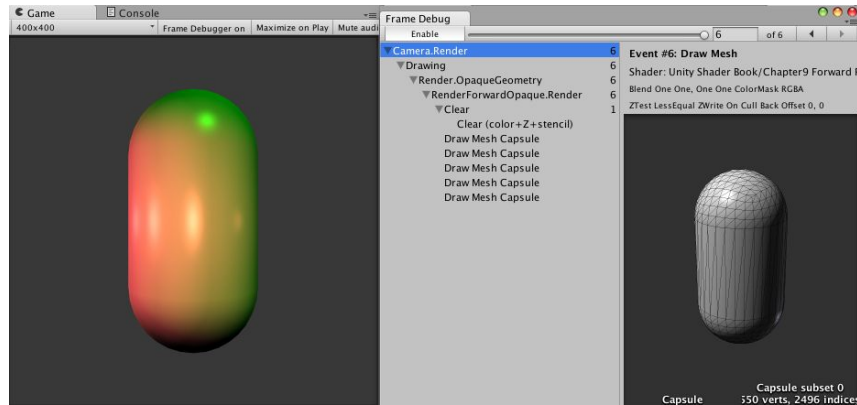


▲9.10 1 + 4

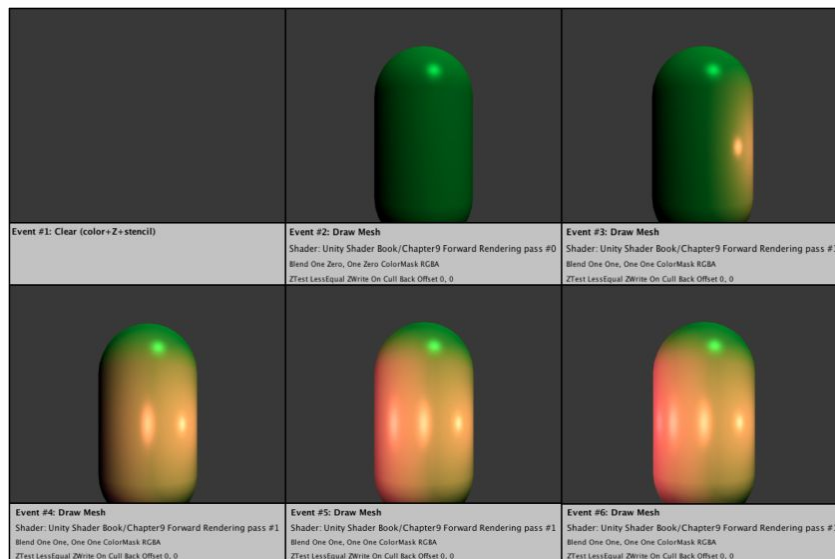
Render Mode Light Auto
Unity SH Edit → Project Settings → Quality → Pixel Light Count 4 5
Chapter9-Forward Rendering Base Pass 4
Render Mode Auto 4 Chapter9-ForwardRendering Additional Pass
Additional Pass

Unity 5 Frame Debugger Window -> Frame Debugger 9.11

Unity 6 6 9.12 Unity 6



▲9.11 画面のレンダリング



▲9.12 画面の6つのレンダリングパス

9.12のUnityのレンダリングパスは、Unityのレンダリングパスの1つである。Unityのレンダリングパスは、UnityのChapter9-ForwardRenderingのPassのBase Passと、4つのAdditional Passの4つのパスから構成されている。

Unityのレンダリングパスは、9.12のレンダリングパスの1つである。Unityのレンダリングパスは、0.2のレンダリングパスの1つである。Unityのレンダリングパスは、Unityのレンダリングパスの1つである。

Unityのレンダリングパスは、Unityのレンダリングパスの1つである。Unityのレンダリングパスは、9.13のレンダリングパスの1つである。Unityのレンダリングパスは、Unityのレンダリングパスの1つである。

9.3 Unity Shader

9.2节介绍了Unity Shader的基本概念，本节将介绍Unity Shader的编写方法。

- 了解Unity Shader的编写方法
- 了解Unity Shader的编译方法

Unity Shader的编写方法，主要涉及Shader的编写、编译和运行。Unity Shader的编译方法，主要涉及Shader的编译选项和编译结果。

9.3.1 Shader的编写

Unity Shader的编写，主要涉及Shader的编写、编译和运行。Unity Shader的编译方法，主要涉及Shader的编译选项和编译结果。

Unity Shader的编写，主要涉及Shader的编写、编译和运行。Unity Shader的编译方法，主要涉及Shader的编译选项和编译结果。

```
float3 lightCoord = mul(_LightMatrix0, float4(i.worldPosition, 1)).xyz;
```

Unity Shader的编写，主要涉及Shader的编写、编译和运行。Unity Shader的编译方法，主要涉及Shader的编译选项和编译结果。

```
fixed atten = tex2D(_LightTexture0, dot(lightCoord, lightCoord).rr).UNITY_ATTEN_CHANNEL;
```

Unity Shader的编写，主要涉及Shader的编写、编译和运行。Unity Shader的编译方法，主要涉及Shader的编译选项和编译结果。

9.3.2 Shader的编译

Unity Shader的编译，主要涉及Shader的编译选项和编译结果。

```
float distance = length(_WorldSpaceLightPos0.xyz - i.worldPosition.xyz);  
atten = 1.0 / distance; // linear attenuation
```

Unity Shader的编译，主要涉及Shader的编译选项和编译结果。

ShaderUnity
ShaderUnity

9.4 Unity

Unityは、ゲームエンジンとして知られる。この章では、Unityのレンダリングパイプラインについて説明する。

9.4.1 レンダリングパイプライン

Unityのレンダリングパイプラインは、以下の順序で実行される。

1. **Shadow Map** の生成。これは、ShadowCaster Passで実行される。

2. **Base Pass**。これは、Unityのレンダリングパイプラインのメインパスである。

3. **Additional Pass**。これは、Base Passの後に実行される。Unityは、このパスで、**LightMode** と **ShadowCaster** の2つのパスを実行する。Unity Shaderは、**LightMode** と **ShadowCaster** の2つのパスを実行する。Unity Shaderは、**LightMode** と **ShadowCaster** の2つのパスを実行する。Unity Shaderは、**LightMode** と **ShadowCaster** の2つのパスを実行する。

4. **ScreenSpace Shadow Map**。これは、Unity 5.0で導入された。これは、Unityのレンダリングパイプラインのメインパスである。

5. **Post Processing**。これは、Unityのレンダリングパイプラインのメインパスである。

Unityのレンダリングパイプラインは、以下の順序で実行される。

- **Shadow Map** の生成。これは、ShadowCaster Passで実行される。
- **Base Pass**。これは、Unityのレンダリングパイプラインのメインパスである。

Unityのレンダリングパイプラインは、以下の順序で実行される。

9.4.2 阴影设置

阴影设置步骤如下：

1. 在Unity Hierarchy视图中，选中要设置阴影的物体，在Inspector视图中，将Scene_9_4_2下的Unity 5.2下的Light属性设置为Window → Lighting → Skybox。

2. 在Unity Hierarchy视图中，选中要设置阴影的物体，在Inspector视图中，将ShadowMat属性设置为9.2下的Chapter9-ForwardRendering。

3. 在Unity Hierarchy视图中，选中要设置阴影的物体，在Inspector视图中，将2下的Standard属性设置为Standard。

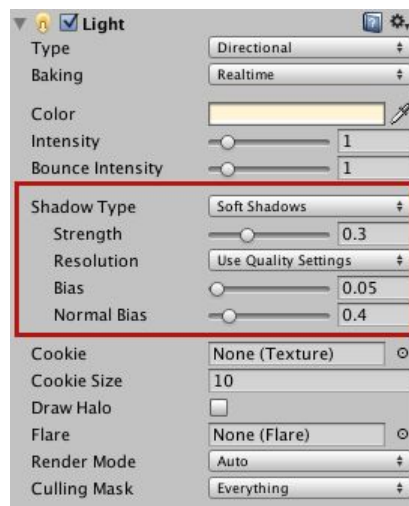
4. 设置阴影。

在Unity Hierarchy视图中，选中要设置阴影的物体，在Inspector视图中，将Light属性设置为9.15下的Light。

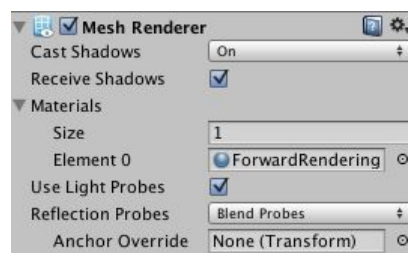
在Unity Hierarchy视图中，选中要设置阴影的物体，在Inspector视图中，将Soft Shadows属性设置为Soft Shadows。

1. 阴影设置

在Unity Hierarchy视图中，选中要设置阴影的物体，在Inspector视图中，将Mesh Renderer属性设置为Cast Shadows，Receive Shadows属性设置为9.16下的Receive Shadows。



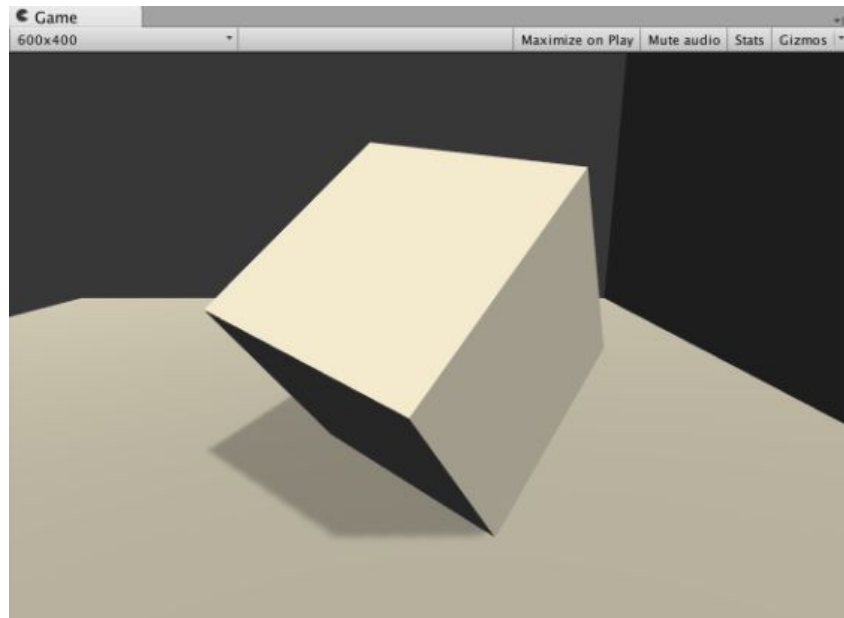
▲9.15 阴影设置



▲9.16 Mesh Renderer的Cast Shadows和Receive Shadows属性设置

Cast Shadows OnOff Cast Shadows Unity
 LightMode ShadowCaster Pass Receive Shadows
 Receive Shadows Unity

Cast Shadows Receive Shadows 9.17



▲ 9.17 Cast Shadows Receive Shadows

9.17 Chapter9-ForwardRendering
 LightMode ShadowCaster Pass
 Chapter9-ForwardRendering Pass
 Chapter9-ForwardRendering SubShader Pass—Base Pass Additional Pass
 Chapter9-ForwardRendering Fallback

Fallback "Specular"

Chapter9-ForwardRendering Fallback Unity Shader
 Specular Specular Pass Fallback VertexLit
 VertexLit Unity builtin-shaders-xxx-
 >DefaultResourcesExtra->Normal- VertexLit.shader
 LightMode ShadowCaster Pass

```
// Pass to render object as a shadow caster
Pass {
    Name "ShadowCaster"
    Tags { "LightMode" = "ShadowCaster" }

    CGPROGRAM
```

```

#pragma vertex vert
#pragma fragment frag
#pragma multi_compile_shadowcaster
#include "UnityCG.cginc"

struct v2f {
    V2F_SHADOW_CASTER;
};

v2f vert( appdata_base v )
{
    v2f o;
    TRANSFER_SHADOW_CASTER_NORMALOFFSET(o)
    return o;
}

float4 frag( v2f i ) : SV_Target
{
    SHADOW_CASTER_FRAGMENT(i)
}

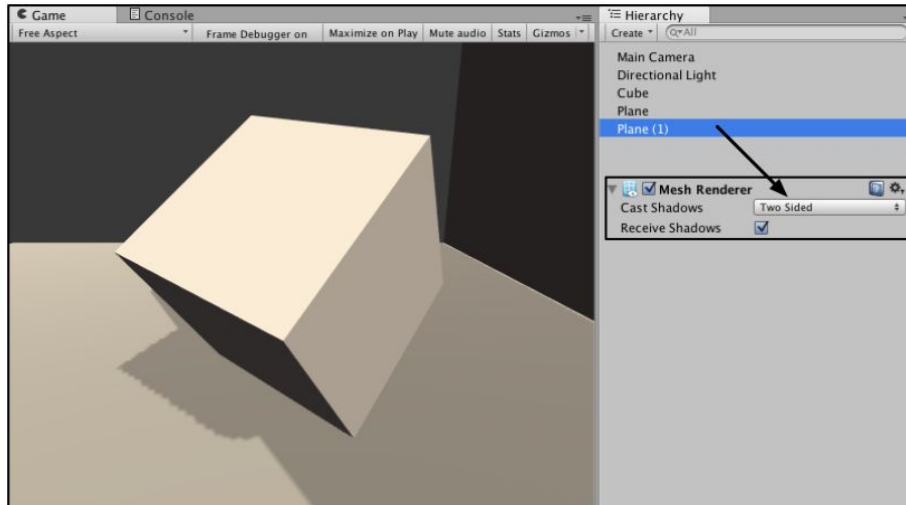
ENDCG
}

```

이 코드는 Unity 5의 Shader Pass를 정의하는 예제입니다.

Chapter9-ForwardRendering에서 Fallback Shader를 정의하는 예제를 보았습니다. Fallback Shader는 SubShader의 LightMode가 ShadowCaster Pass를 호출하는 Pass입니다. Unity Shader에서 Fallback Shader는 Diffuse Shader를 호출합니다. Pass에서 Fallback VertexLit Unity LightMode ShadowCaster Pass를 호출합니다. 9.4.2에서 LightMode ShadowCaster Pass를 보았습니다.

9.17에서 Cast Shadows를 보았습니다. frontface Cast Shadows Two Sided 9.18에서 Cast Shadows Two Sided를 보았습니다.



▲9.18 Cast Shadows Two Sided

この章では、Unity ShaderのStandard ShaderのShaderのCast ShadowsをTwo Sidedに設定し、Chapter9-ForwardRenderingでShadowMatを生成する。

2. Shader

Unity ShaderのChapter9-ShadowのShaderのCast ShadowsをTwo Sidedに設定し、Chapter9-ShadowMatを生成する。

1. Base Pass

```
#include "AutoLight.cginc"
```

この章では、Unity ShaderのChapter9-ShadowのShaderのCast ShadowsをTwo Sidedに設定し、Chapter9-ShadowMatを生成する。

2. v2f SHADOW_COORDS

```
struct v2f {
    float4 pos : SV_POSITION;
    float3 worldNormal : TEXCOORD0;
    float3 worldPos : TEXCOORD1;
    SHADOW_COORDS(2)
};
```

この章では、Unity ShaderのChapter9-ShadowのShaderのCast ShadowsをTwo Sidedに設定し、Chapter9-ShadowMatを生成する。

3. TRANSFER_SHADOW

```
v2f vert(a2v v) {
    v2f o;
```

```

    ...
    // Pass shadow coordinates to pixel shader
    TRANSFER_SHADOW(o);

    return o;
}

```

SHADOW_COORDS TRANSFER_SHADOW SHADOW_ATTENUATION

4 SHADOW_ATTENUATION

```

// Use shadow coordinates to sample shadow map
fixed shadow = SHADOW_ATTENUATION(i);

```

SHADOW_COORDS TRANSFER_SHADOW SHADOW_ATTENUATION
 "AutoLight.cginc"

```

// -----
// Shadow helpers
// -----

// ---- Screen space shadows
#if defined (SHADOWS_SCREEN)
    UNITY_DECLARE_SHADOWMAP(_ShadowMapTexture);
    #define SHADOW_COORDS(idx1) unityShadowCoord4 _ShadowCoord : TEXCOORD##idx1;
    #if defined(UNITY_NO_SCREENSPACE_SHADOWS)
        #define TRANSFER_SHADOW(a) a._ShadowCoord = mul( unity_World2Shadow[0],
            mul( _Object2World, v.vertex ) );
        inline fixed unitySampleShadow (unityShadowCoord4 shadowCoord)
        {
            ...
        }
    #else // UNITY_NO_SCREENSPACE_SHADOWS
        #define TRANSFER_SHADOW(a) a._ShadowCoord = ComputeScreenPos(a.pos);
        inline fixed unitySampleShadow (unityShadowCoord4 shadowCoord)
        {
            fixed shadow = tex2Dproj( _ShadowMapTexture, UNITY_PROJ_COORD(shadowCoord) ).r;
            return shadow;
        }
    #endif
    #define SHADOW_ATTENUATION(a) unitySampleShadow(a._ShadowCoord)
#endif

// ---- Spot light shadows
#if defined (SHADOWS_DEPTH) && defined (SPOT)
    ...
#endif

// ---- Point light shadows
#if defined (SHADOWS_CUBE)
    ...
#endif

// ---- Shadows off
#if !defined (SHADOWS_SCREEN) && !defined (SHADOWS_DEPTH) && !defined (SHADOWS_CUBE)
    #define SHADOW_COORDS(idx1)
    #define TRANSFER_SHADOW(a)
    #define SHADOW_ATTENUATION(a) 1.0

```

```
#endif
```

```

// Unity shader compilation defines
SHADOW_COORDS _ShadowCoord
TRANSFER_SHADOW
UNITY_NO_SCREENSPACE_SHADOWS
ComputeScreenPos _ShadowCoord
TRANSFER_SHADOW _ShadowCoord
SHADOW_ATTENUATION _ShadowCoord

```

```

// SHADOW_COORDS
TRANSFER_SHADOW SHADOW_ATTENUATION 1

```

```

// TRANSFER_SHADOW
v.vertex
a.pos
a2f
vertex
v2f
v
v2f
pos

```

```

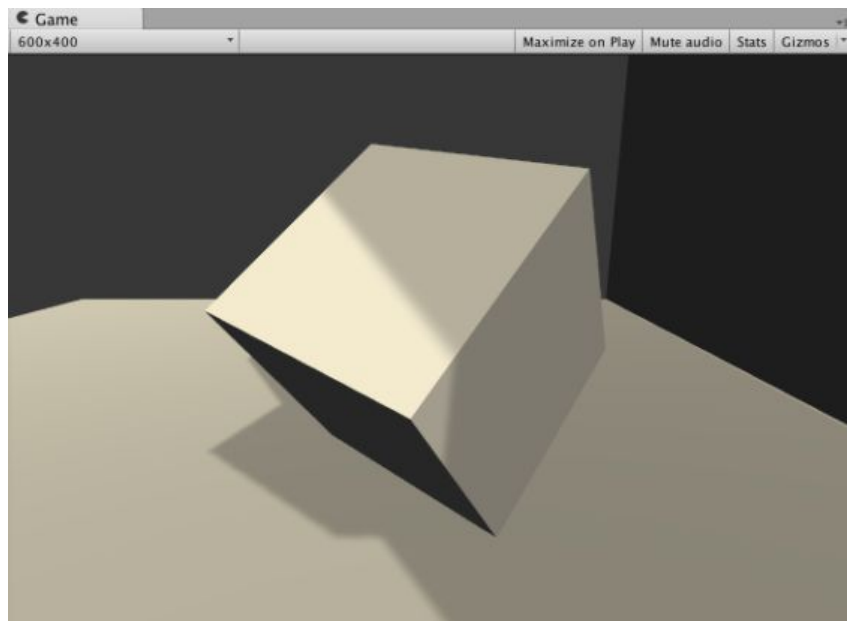
5 shadow

```

```

Unity 9.19

```



▲9.19 阴影效果

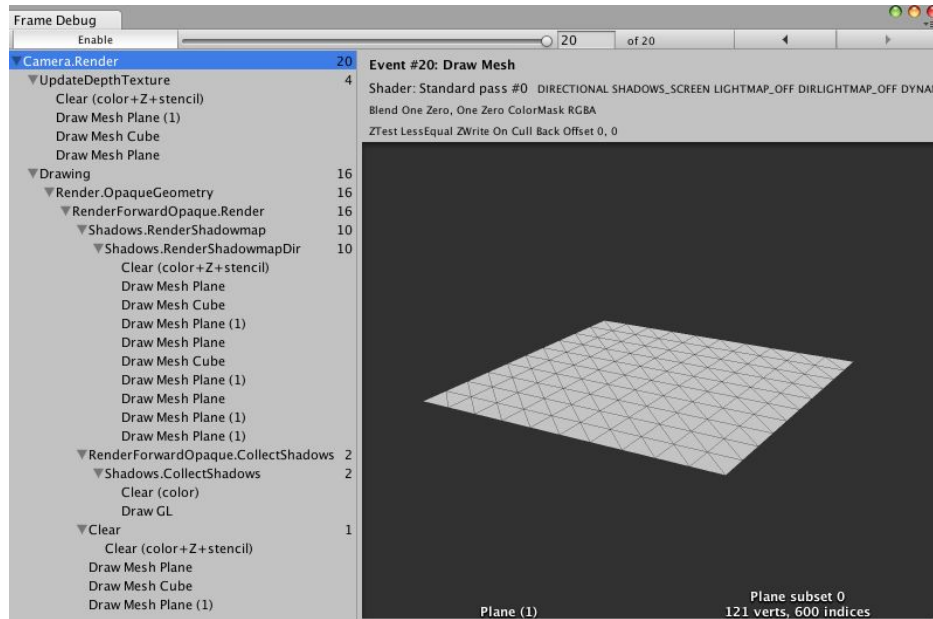
Base Pass 和 Additional Pass 是 Unity Shader 的两个主要部分。Base Pass 负责渲染物体的基本属性，而 Additional Pass 则用于实现更复杂的视觉效果。在 9.4.4 节中，我们将讨论如何编写 Additional Pass。在 9.5 节中，我们将讨论如何编写 Unity Shader。

9.4.3 阴影效果

———— Unity 5
9.2.2 Pass

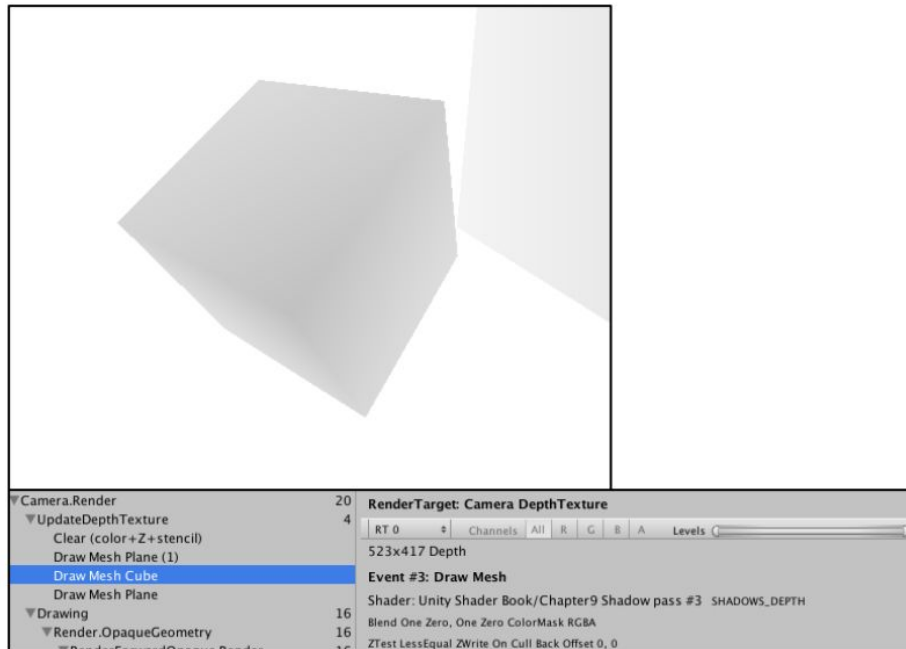
Window -> Frame Debugger 9.20 Scene_9_4_2

9.20 20 4 UpdateDepthTexture
RenderShadowmap CollectShadows



▲ 9.20

9.21

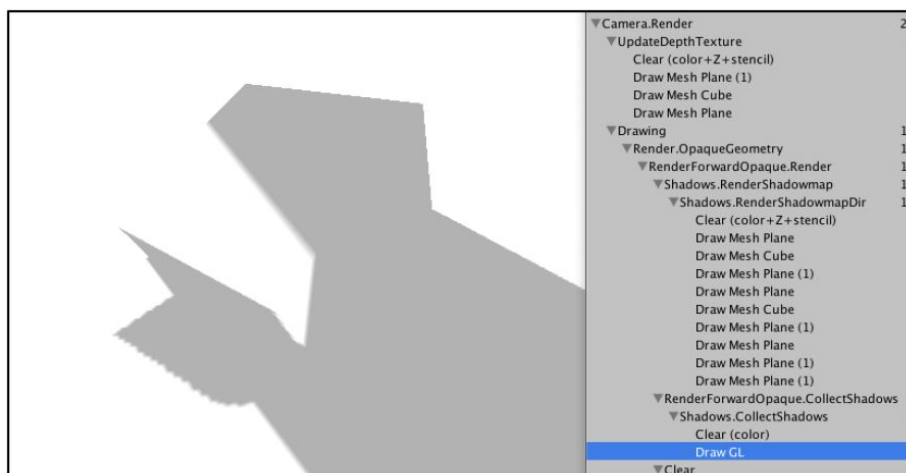


▲9.21 画面の構成

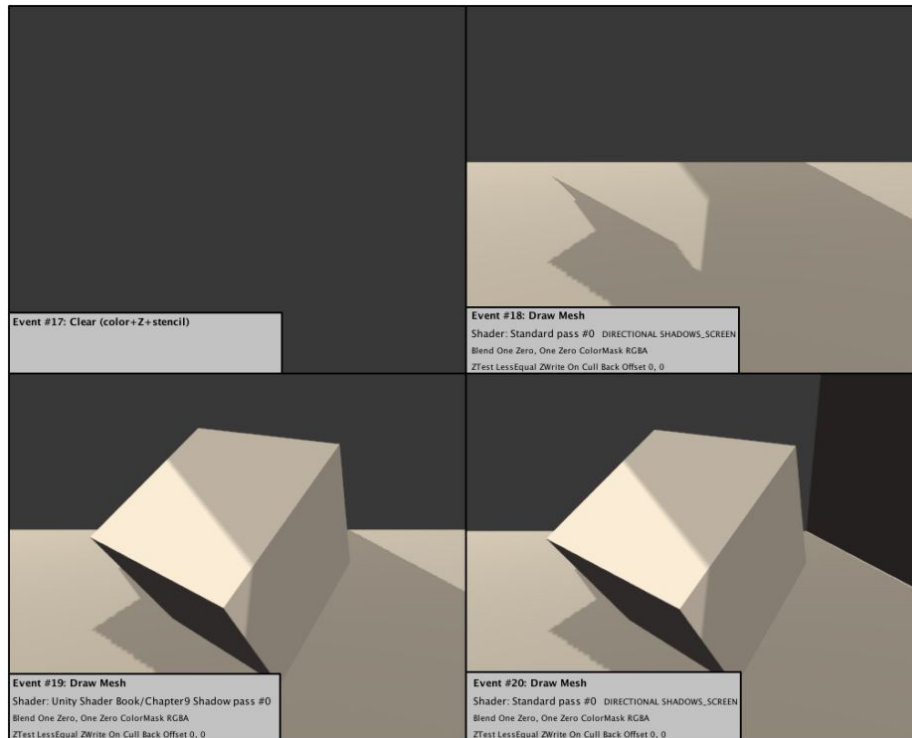
画面の構成を確認するため、9.21の画面の構成を確認する。UnityのShader: **Unity Shader Book/Chapter9 Shadow pass #3** の画面の構成を確認する。Chapter9-ShadowのPassのChapter9-ShadowのPassのUnityのFallbackのPassの**LightMode** の**ShadowCaster** のPassの画面の構成を確認する。UnityのPassの画面の構成を確認する。

画面の構成を確認するため、9.22の画面の構成を確認する。

画面の構成を確認するため、9.23の画面の構成を確認する。Unityの画面の構成を確認する。



▲9.22 画面の構成



▲9.23 Unityのレンダリングパス

9.4.4 影の計算

9.2と9.3のレンダリングパスでUnity Shaderのレンダリングパスは——Base PassとAdditional Passの2つに分かれている。Base Passは1つのPassで、Additional Passは複数のPassで構成されている。このように、Unity Shaderのレンダリングパスは、Base PassとAdditional Passの2つに分かれている。

Unity Shaderのレンダリングパスは、Base PassとAdditional Passの2つに分かれている。

1. 9.4.2のレンダリングパスで、Scene_9_4_4のレンダリングパスは、Base PassとAdditional Passの2つに分かれている。

2. 9.4.2のレンダリングパスで、AttenuationAndShadowUseBuildInFunctionsMatのレンダリングパスは、Base PassとAdditional Passの2つに分かれている。

3. 9.4.2のUnity Shaderのレンダリングパスは、Chapter9-AttenuationAndShadowUse BuildInFunctionsの2つのPassに分かれている。

4. 9.4.2のレンダリングパスで、Chapter9-Shadowのレンダリングパスは、Base PassとAdditional Passの2つに分かれている。

5. 9.4.2のレンダリングパスで、Chapter9-Shadowのレンダリングパスは、Base PassとAdditional Passの2つに分かれている。

Chapter9-AttenuationAndShadowUseBuildInFunctionsとChapter9-Shadowの2つのPassは、Chapter9-Shadowのレンダリングパスで、Unityのレンダリングパスの一部として実行される。

1

```
// Need these files to get built-in macros
#include "Lighting.cginc"
#include "AutoLight.cginc"
```

2 `v2f` `SHADOW_COORDS`

```
struct v2f {
    float4 pos : SV_POSITION;
    float3 worldNormal : TEXCOORD0;
    float3 worldPos : TEXCOORD1;
    SHADOW_COORDS(2)
};
```

3 `TRANSFER_SHADOW`

```
v2f vert(a2v v) {
    v2f o;
    ...
    TRANSFER_SHADOW(o);

    return o;
}
```

4 9.4.2 `UNITY_LIGHT_ATTENUATION`

```
fixed4 frag(v2f i) : SV_Target {
    ...

    // UNITY_LIGHT_ATTENUATION not only compute attenuation, but also shadow infos
    UNITY_LIGHT_ATTENUATION(atten, i, i.worldPos);

    return fixed4(ambient + (diffuse + specular) * atten, 1.0);
}
```

UNITY_LIGHT_ATTENUATION `Unity`
`AutoLight.cginc` 3
`atten` **UNITY_LIGHT_ATTENUATION**
`v2f` 9.4.2 **SHADOW_ATTENUATION**
9.3
`AutoLight.cginc` **UNITY_LIGHT_ATTENUATION** `Unity`
`cookie` **UNITY_LIGHT_ATTENUATION**

5 **UNITY_LIGHT_ATTENUATION** `Base Pass` `Additional Pass`
— `Base Pass` `Additional Pass`
UNITY_LIGHT_ATTENUATION `Unity` `Additional Pass` `#pragma multi_compile_fwdadd_fullshadows`

Additional Pass `#pragma multi_compile_fwdadd` `Unity` Shader

9.4.5

`Unity` Shader `LightMode` `ShadowCaster` `Pass` `VertexLit` `ShadowCaster` `VertexLit` `ShadowCaster`

`Fallback` `VertexLit` `Fallback`

`VertexLit` `Diffuse` `Specular` `Scene_9_4_5_a` `AlphaTestWithShadowMat` `Chapter9-AlphaTestWithShadow` `Chapter9-AlphaTestWithShadow` 8.3

1

```
#include "Lighting.cginc"
#include "AutoLight.cginc"
```

2 `v2f` `SHADOW_COORDS`

```
struct v2f {
    float4 pos : SV_POSITION;
    float3 worldNormal : TEXCOORD0;
    float3 worldPos : TEXCOORD1;
    float2 uv : TEXCOORD2;
    SHADOW_COORDS(3)
};
```

3 `SHADOW_COORDS` `TEXCOORD0` `TEXCOORD1` `TEXCOORD2` `TEXCOORD3`

3 `TRANSFER_SHADOW`

```
v2f vert(a2v v) {
    v2f o;
    ...
    // Pass shadow coordinates to pixel shade
    TRANSFER_SHADOW(o);

    return o;
}
```

4 `UNITY_LIGHT_ATTENUATION`

```
fixed4 frag(v2f i) : SV_Target {
    ...
    // UNITY_LIGHT_ATTENUATION not only compute attenuation, but also shadow infos
    UNITY_LIGHT_ATTENUATION(atten, i, i.worldPos);

    return fixed4(ambient + diffuse * atten, 1.0);
}
```

5FallbackVertexLitShader

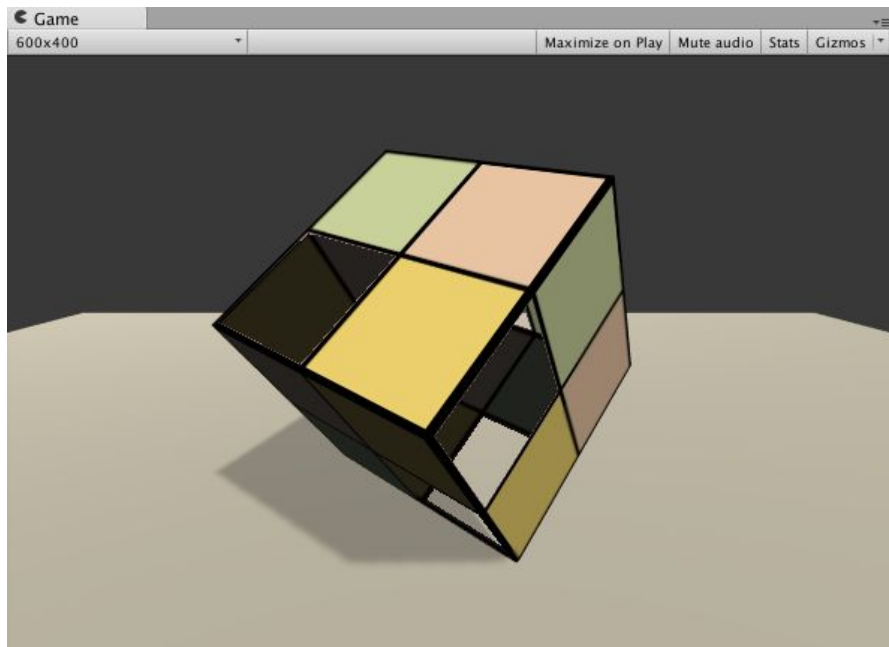
Fallback "VertexLit"

transparent_texture.psd9.24

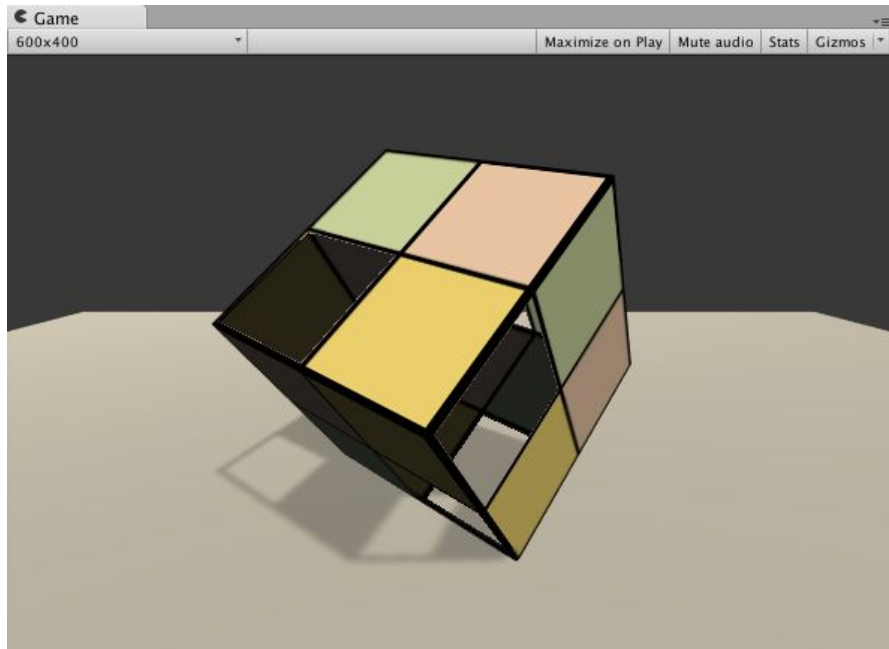
VertexLit ShadowCaster PassShadowCaster Pass PassUnity Shader

Unity ShaderFallbackTransparent/Cutout/VertexLit8.3Unity ShaderShadowCaster PassTransparent/Cutout/VertexLit_CutoffShader_Cutoff

Fallback9.25

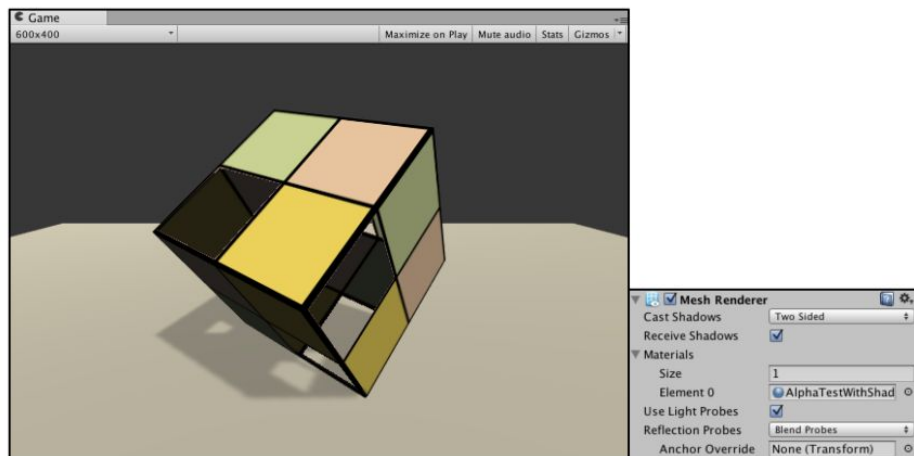


▲9.24



▲9.25 Fallback

Unity 9.25 版本开始，Fallback 材质不再支持。在 Unity 9.25 之前，Fallback 材质是用于在透明物体后面显示其他物体的。现在，Unity 9.25 引入了新的 Mesh Renderer 的 Cast Shadows 属性，可以设置物体是否投射阴影。在 Unity 9.26 版本中，Fallback 材质被移除，取而代之的是新的 Two Sided 属性。



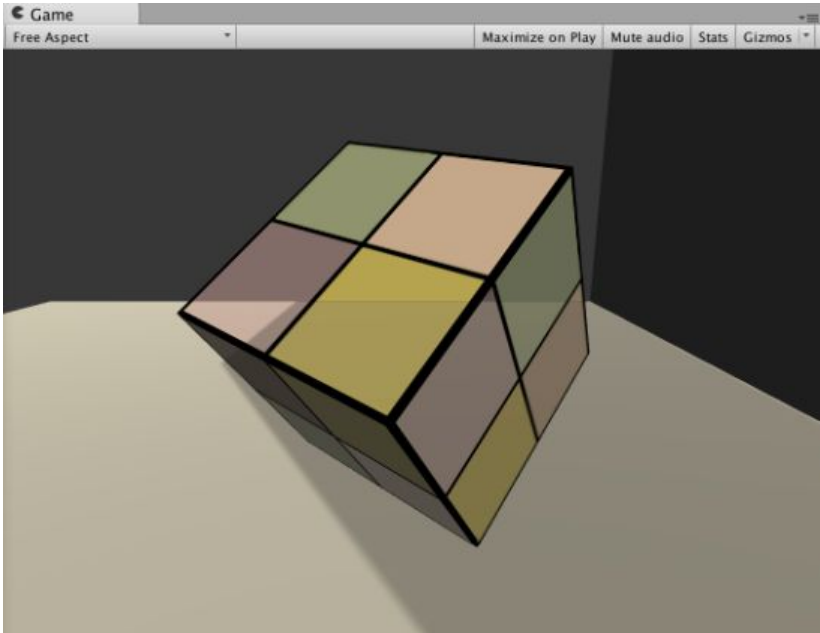
▲9.26 Cast Shadow

Unity Shader 的 Transparent/VertexLit 渲染模式不再支持。在 Unity 9.25 之前，Transparent/VertexLit 渲染模式是用于在透明物体后面显示其他物体的。现在，Unity 9.25 引入了新的 Mesh Renderer 的 Cast Shadows 属性，可以设置物体是否投射阴影。在 Unity 9.26 版本中，Fallback 材质被移除，取而代之的是新的 Two Sided 属性。在 Unity 9.27 版本中，Transparent/VertexLit 渲染模式被移除，取而代之的是新的 AlphaBlendWithShadowMat 材质。

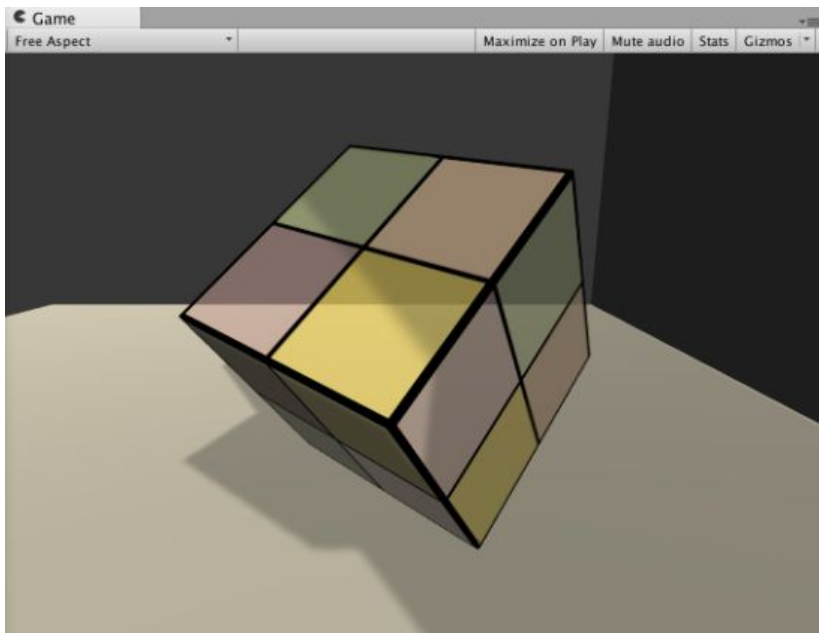
```

Unity
Shader
dirty trick
Fallback
VertexLit
Diffuse
Unity Shader
Unity
Fallback
Pass
Mesh Renderer
Cast Shadows
Receive
Shadows
9.28
Fallback
VertexLit

```



▲9.27 Unity ShaderFallbackTransparent/VertexLit



▲ 9.28 Fallback VertexLit

[illegible]

9.5 如何设置Unity Shader

Unity 提供了许多内置的 Shader，这些 Shader 位于 Unity 的 Assets/ Shaders/Common 目录下。Unity Shader 包括 BumpedDiffuse、BumpedSpecular、Unity Shader、BumpedDiffuse、Phong、BumpedSpecular、Blinn-Phong 等。Unity Shader 位于 Assets/ Material/ Objects 目录下。Assets/ Material/ Walls 目录下也有 Unity Shader。Unity Shader 位于 Unity Shader 目录下。

10

7

Cubemap

Render Texture

Procedure Texture

10.1 □□□□□

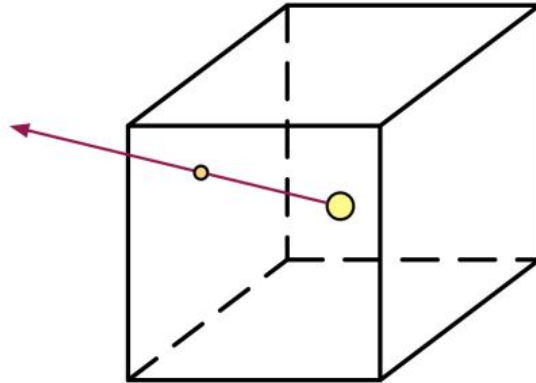
環境マップの環境マッピング

6

3D

6

10.1



▲10.1 □□□□□□□□

Unity 5

☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ ☐ Skybox ☐ ☐ ☐ ☐

10.1.1 〇〇〇〇

Skybox

Unity Skybox

Skybox

1 SkyboxMat

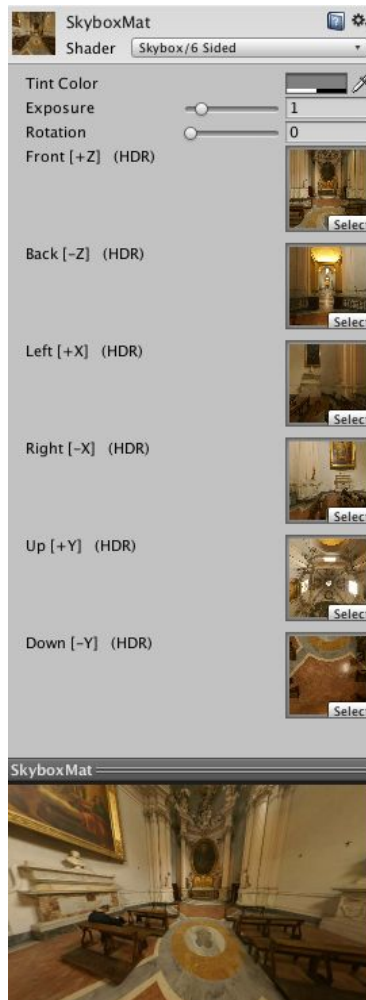
2 SkyboxMat Unity Shader Unity Skybox/6 Sided 6

```

    3Assets/Textures/Chapter10/Cubemaps626
    posFront [+Z] 6Wrap Mode Clamp

```

□□□□□□□□□□10.2□□□



▲10.2 環境

環境は6面から3面の **Tint Color** 環境は **Exposure** 環境は **Rotation** 環境は +y 環境は

環境は Skybox

1 環境は Scene_10_1_1

2 Window → Lighting 環境は SkyboxMat 環境は Skybox 環境は 10.3

環境は Camera 環境は Clear Flags 環境は **Skybox** 環境は 10.4



▲10.3 環境照明の設定



▲10.4 3Dレンダリング結果

UnityのWindow → Lighting → SkyboxメニューからSkybox Component → Rendering → Skyboxを選択する。

UnityのSkybox ComponentのInspectorで、SkyboxMatを選択する。

10.1.2 環境反射の設定

UnityのSkybox ComponentのInspectorで、Reflection SourceをSkyboxに設定する。

Unity 5では、CubemapのResolutionを6に設定する。

UnityのInspectorで、Texture TypeをCubemapに設定する。UnityはHDRをサポートしているため、CubemapのFormatを18に設定する。<http://docs.unity3d.com/Manual/class-Cubemap.html> を参照。

Unity 5では、CubemapのResolutionを6に設定する。Unity 5では、glossy reflectionをサポートしている。

`Camera.RenderToCubemap`

```
void OnWizardCreate () {
    // create temporary camera for rendering
    GameObject go = new GameObject( "CubemapCamera");
    go.AddComponent<Camera>();
    // place it on the object
    go.transform.position = renderFromPosition.position;
    // render into cubemap
    go.GetComponent<Camera>().RenderToCubemap(cubemap);

    // destroy temporary camera
    DestroyImmediate( go );
}
```

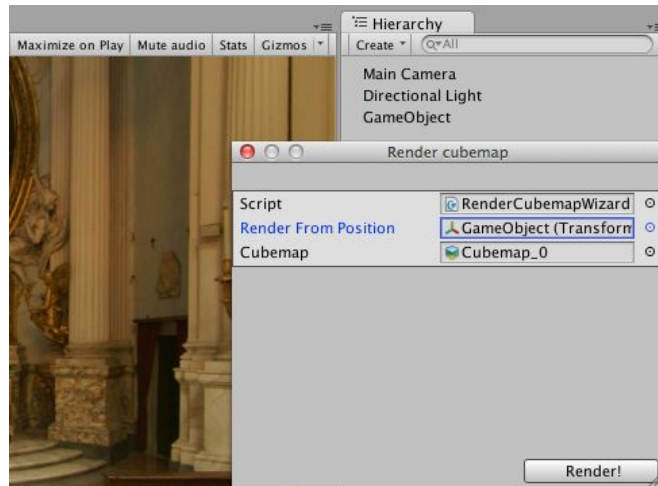
立方体映射Cubemap

```

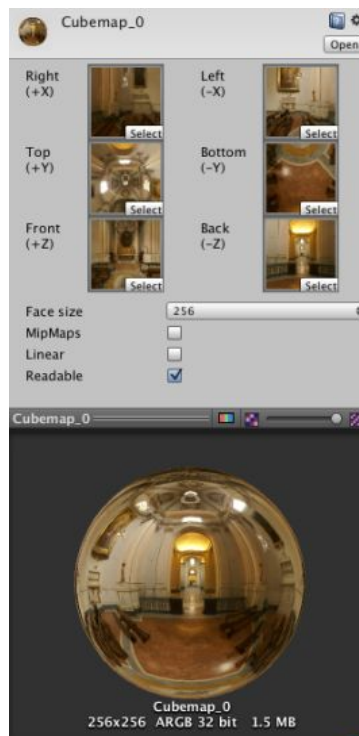
3 UnityGameObject -> Render into Cubemap
1 GameObject 2 Cubemap_0 Render From Position Cubemap
10.5

```

4 Render! 6 Cubemap 0 10.6



▲10.5 立方体マップの作成



▲10.6 立方体マップの適用

立方体マップの作成が完了したら、CubemapのInspectorで10.6の**Face size**とFace sizeの値を指定して、立方体マップを適用します。

立方体マップの適用が完了したら、Inspectorで立方体マップの値を確認します。

10.1.3 立方体マップ

立方体マップの作成が完了したら、Inspectorで立方体マップの値を確認します。10.7の図のように、Inspectorで立方体マップの値を確認します。



▲10.7 反射オブジェクトTeapot

制作環境

1. 制作環境はScene_10_1_3でUnity 5.10.11、Window → Lighting → Skybox → Skyboxを選択

2. Teapotを10.1.2のCubemap_0のGameObjectに追加

3. ReflectionMatを2つのTeapotに適用

4. Unity ShaderのShaderをChapter10-ReflectionのChapter10-Reflection_3を選択

Chapter10-Reflectionの環境設定

1. 3つの

```
Properties {
    _Color ("Color Tint", Color) = (1, 1, 1, 1)
    _ReflectColor ("Reflection Color", Color) = (1, 1, 1, 1)
    _ReflectAmount ("Reflect Amount", Range(0, 1)) = 1
    _Cubemap ("Reflection Cubemap", Cube) = "_Skybox" {}
}
```

_ReflectColor、_ReflectAmount、_Cubemapの値を設定

2. CGのreflect関数

```
v2f vert(a2v v) {
    v2f o;

    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.worldNormal = UnityObjectToWorldNormal(v.normal);
```

```

o.worldPos = mul(_Object2World, v.vertex).xyz;

o.worldViewDir = UnityWorldSpaceViewDir(o.worldPos);

// Compute the reflect dir in world space
o.worldRefl = reflect(-o.worldViewDir, o.worldNormal);

TRANSFER_SHADOW(o);

return o;
}

```

このシェーダーは、反射方向を計算し、反射色を計算する。

3つの反射方向を計算する。

```

fixed4 frag(v2f i) : SV_Target {
    fixed3 worldNormal = normalize(i.worldNormal);
    fixed3 worldLightDir = normalize(UnityWorldSpaceLightDir(i.worldPos));
    fixed3 worldViewDir = normalize(i.worldViewDir);

    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

    fixed3 diffuse = _LightColor0.rgb * _Color.rgb * max(0, dot(worldNormal, worldLightDir));

    // Use the reflect dir in world space to access the cubemap
    fixed3 reflection = texCUBE(_Cubemap, i.worldRefl).rgb * _ReflectColor.rgb;

    UNITY_LIGHT_ATTENUATION(atten, i, i.worldPos);

    // Mix the diffuse color with the reflected color
    fixed3 color = ambient + lerp(diffuse, reflection, _ReflectAmount) * atten;

    return fixed4(color, 1.0);
}

```

このシェーダーは、CGのtexCUBE関数を使用して、i.worldReflの値を Cubemap から取得し、_ReflectAmountの値を乗算して、反射色を計算する。

このシェーダーは、UnityのLightModelAmbientの値を使用して、環境光を計算する。

このシェーダーは、Cubemap_0のReflection Cubemapを使用して、10.7の値を乗算して、反射色を計算する。

10.1.4 屈折

このシェーダーは、Unity ShaderのRefract関数を使用して、屈折を計算する。

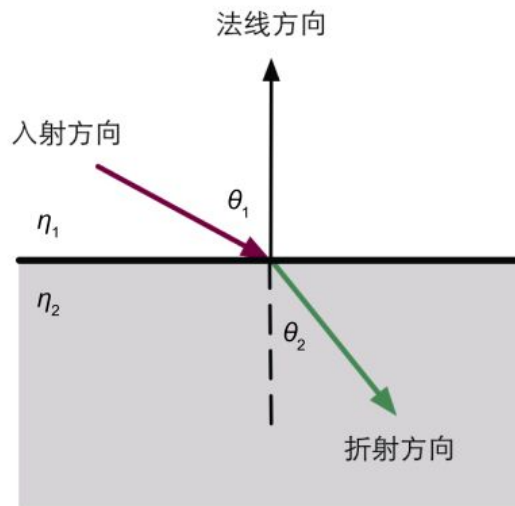
このシェーダーは、Snell's Lawを使用して、1つの角度θ₁と2つの角度θ₂を計算する。

$$\eta_1 \sin \theta_1 = \eta_2 \sin \theta_2$$

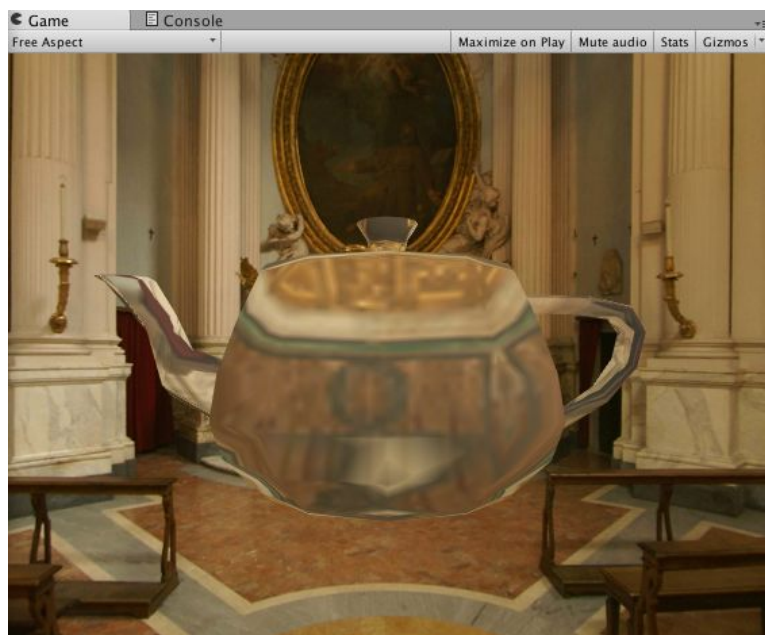
このη₁とη₂は、index of refractionの値であり、1.5と10.8の値を乗算して、屈折を計算する。

1. 在 Unity 5 中，我们使用 Ray Tracing 来模拟光线折射。在 Unity 5 中，我们使用 Ray Tracing 来模拟光线折射。在 Unity 5 中，我们使用 Ray Tracing 来模拟光线折射。

10.9 折射



▲10.8 折射



▲10.9 透明玻璃Teapot

1. 在 Unity 5 中，我们使用 Ray Tracing 来模拟光线折射。

1. 在 Unity 5 中，我们使用 Ray Tracing 来模拟光线折射。在 Unity 5 中，我们使用 Ray Tracing 来模拟光线折射。在 Unity 5 中，我们使用 Ray Tracing 来模拟光线折射。

2. 在 Unity 5 中，我们使用 Ray Tracing 来模拟光线折射。

3 RefractionMat 2 Teapot

4 Unity Shader Shader Chapter10-Refract Chapter10-Refract 3

Chapter10-Refract

1 4

```
Properties {
    _Color ("Color Tint", Color) = (1, 1, 1, 1)
    _RefractColor ("Refraction Color", Color) = (1, 1, 1, 1)
    _RefractAmount ("Refraction Amount", Range(0, 1)) = 1
    _RefractRatio ("Refraction Ratio", Range(0.1, 1)) = 0.5
    _Cubemap ("Refraction Cubemap", Cube) = "_Skybox" {}
}
```

_RefractColor _RefractAmount _Cubemap 10.1.3
_RefractRatio

2

```
v2f vert(a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.worldNormal = UnityObjectToWorldNormal(v.normal);

    o.worldPos = mul(_Object2World, v.vertex).xyz;

    o.worldViewDir = UnityWorldSpaceViewDir(o.worldPos);

    // Compute the refract dir in world space
    o.worldRefr = refract(-normalize(o.worldViewDir), normalize(o.worldNormal), _RefractRatio);

    TRANSFER_SHADOW(o);

    return o;
}
```

CG **refract**
1/1.5

3

```
fixed4 frag(v2f i) : SV_Target {
    fixed3 worldNormal = normalize(i.worldNormal);
    fixed3 worldLightDir = normalize(UnityWorldSpaceLightDir(i.worldPos));
    fixed3 worldViewDir = normalize(i.worldViewDir);

    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

    fixed3 diffuse = _LightColor0.rgb * _Color.rgb * max(0, dot(worldNormal, worldLightDir));

    // Use the refract dir in world space to access the cubemap
    fixed3 refraction = texCUBE(_Cubemap, i.worldRefr).rgb * _RefractColor.rgb;

    UNITY_LIGHT_ATTENUATION(atten, i, i.worldPos);

    // Mix the diffuse color with the refract color
    fixed3 color = ambient + lerp(diffuse, refraction, _RefractAmount) * atten;
}
```

```
}    return fixed4(color, 1.0);
```

return i.worldRefractAmount * i.RefractAmount + (1 - i.worldRefractAmount) * i.RefractAmount;

Cubemap_0 Reflection Cubemap 10.9

10.1.5

Fresnel reflection 18 John Hable *Everything Has Fresnel* <http://filmicgames.com/archives/557>

Schlick

$$F_{Schlick}(\mathbf{v}, \mathbf{n}) = F_0 + (1 - F_0)(1 - \mathbf{v} \cdot \mathbf{n})^5$$

F_0 $\mathbf{v}$ $\mathbf{n}$ **Empirical**

$$F_{Empirical}(\mathbf{v}, \mathbf{n}) = \max(0, \min(1, \text{bias} + \text{scale} \times (1 - \mathbf{v} \cdot \mathbf{n})^{\text{power}}))$$

bias scale power

Schlick 10.10



▲10.10 Teapot

10.10 Teapot

1. Scene_10_1_5 Unity 5 10.1.1
Window → Lighting → Skybox

2. Teapot

3. FresnelMat 2 Teapot

4. Unity Shader Shader Chapter10-Fresnel Chapter10- Fresnel 3

Chapter10-Fresnel

1. Properties Cubemap

```
Properties {
    _Color ("Color Tint", Color) = (1, 1, 1, 1)
    _FresnelScale ("Fresnel Scale", Range(0, 1)) = 0.5
    _Cubemap ("Reflection Cubemap", Cube) = "_Skybox" {}
}
```

2.

```
v2f vert(a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.worldNormal = mul(v.normal, (float3x3)_World2Object);
    o.worldPos = mul(_Object2World, v.vertex).xyz;
    o.worldViewDir = UnityWorldSpaceViewDir(o.worldPos);
    o.worldRefl = reflect(-o.worldViewDir, o.worldNormal);
    TRANSFER_SHADOW(o);
}
```

```
    return o;
}
```

3

```
fixed4 frag(v2f i) : SV_Target {
    fixed3 worldNormal = normalize(i.worldNormal);
    fixed3 worldLightDir = normalize(UnityWorldSpaceLightDir(i.worldPos));
    fixed3 worldViewDir = normalize(i.worldViewDir);

    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

    UNITY_LIGHT_ATTENUATION(atten, i, i.worldPos);

    fixed3 reflection = texCUBE(_Cubemap, i.worldRefl).rgb;

    fixed fresnel = _FresnelScale + (1 - _FresnelScale) * pow(1 - dot(worldViewDir, worldNormal), 5);

    fixed3 diffuse = _LightColor0.rgb * _Color.rgb * max(0, dot(worldNormal, worldLightDir));

    fixed3 color = ambient + lerp(diffuse, reflection, saturate(fresnel)) * atten;

    return fixed4(color, 1.0);
}
```

Schlick fresnel fresnel

Cubemap_0 **Cubemap** 10.10
_FresnelScale 1 Cubemap _FresnelScale 0
15.2

10.2 鏡面

鏡面表現は、通常のレンダリングとは異なり、GPUのバックバッファに描画されたものを、別のRenderTargetTexture(RTT)に書き出し、それをback bufferに読み込み、再度描画する。この処理をMultiple Render Target(MRT)と呼ぶ。GPUの性能が向上し、この処理がより簡単にできるようになった。

Unityでは、MirrorMatというシェADERで鏡面表現を実現している。このシェADERは、UnityのProject Inspectorで、ShaderのGrabPassをOnRenderImageに設定し、UnityのOnRenderImageメソッドで呼び出す。Passは12個ある。

10.2.1 鏡面

鏡面表現は、通常のレンダリングとは異なり、GPUのバックバッファに描画されたものを、別のRenderTargetTexture(RTT)に書き出し、それをback bufferに読み込み、再度描画する。この処理をMultiple Render Target(MRT)と呼ぶ。GPUの性能が向上し、この処理がより簡単にできるようになった。

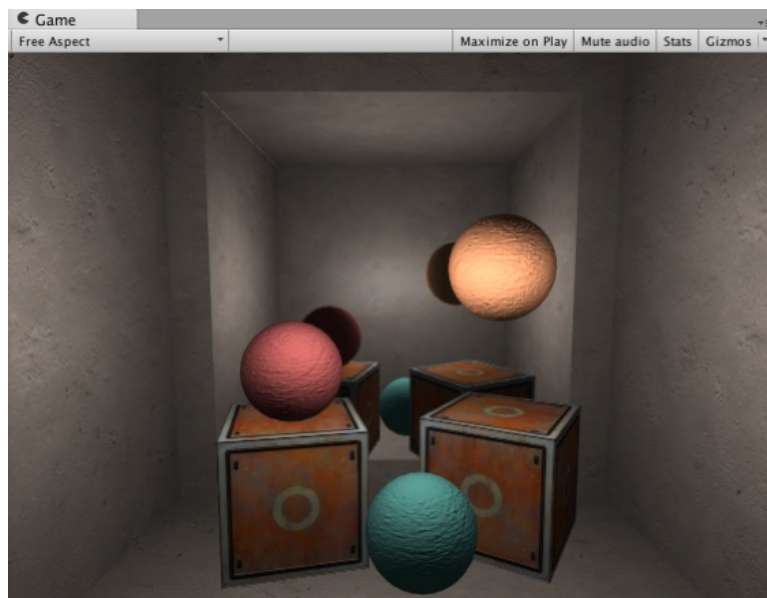


図10.11 鏡面

鏡面表現は、通常のレンダリングとは異なり、GPUのバックバッファに描画されたものを、別のRenderTargetTexture(RTT)に書き出し、それをback bufferに読み込み、再度描画する。この処理をMultiple Render Target(MRT)と呼ぶ。GPUの性能が向上し、この処理がより簡単にできるようになった。

1. Unity 5.2のProject Inspectorで、ShaderのGrabPassをOnRenderImageに設定し、UnityのOnRenderImageメソッドで呼び出す。Passは12個ある。

2. MirrorMatというシェADERで鏡面表現を実現している。

3. Unity ShaderのChapter10-MirrorというシェADERで鏡面表現を実現している。

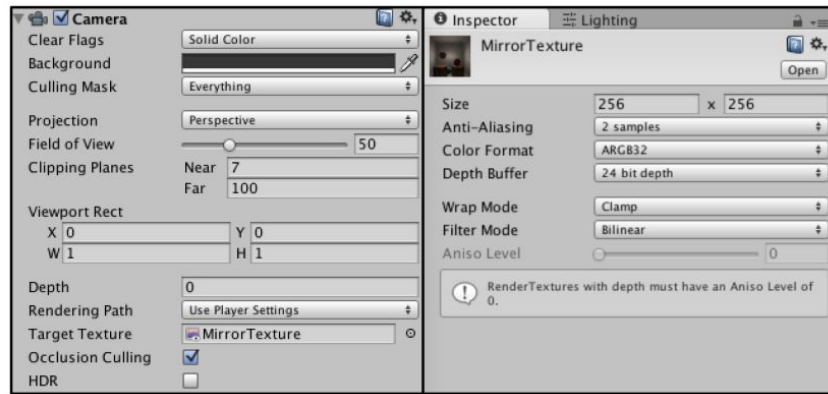
4. 6つの鏡面表現を実現している。6つの鏡面表現は、9.5の鏡面表現と異なり、3つの鏡面表現を実現している。

5. 3つの鏡面表現を実現している。9.5の鏡面表現と異なり、3つの鏡面表現を実現している。

6. QuadというシェADERで鏡面表現を実現している。2つの鏡面表現を実現している。

7 Project Create → Render Texture MirrorTexture 10.12

8 MirrorTexture Target Texture 7 MirrorTexture Target Texture 10.12



10.12 Target Texture

Chapter10-Mirror

1 Properties

```
Properties {
    _MainTex ("Main Tex", 2D) = "white" {}
}
```

2

```
v2f vert(a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.uv = v.texcoord;
    // Mirror needs to flip x
    o.uv.x = 1 - o.uv.x;

    return o;
}
```

x

3

```
fixed4 frag(v2f i) : SV_Target {
    return tex2D(_MainTex, i.uv);
}
```

MirrorTexture Main Tex 10.11

256×256のテクスチャを生成し、それをCubeMapとして使用します。

10.2.2 実装

UnityのShaderのPassでGrabPassを使用する。ShaderのGrabPassでUnityのTextureのPassでGrabPassを使用する。

GrabPassのQueueをTransparentに設定する。

GrabPassの10.13のCubeMapのGrabPassを使用する。

実装

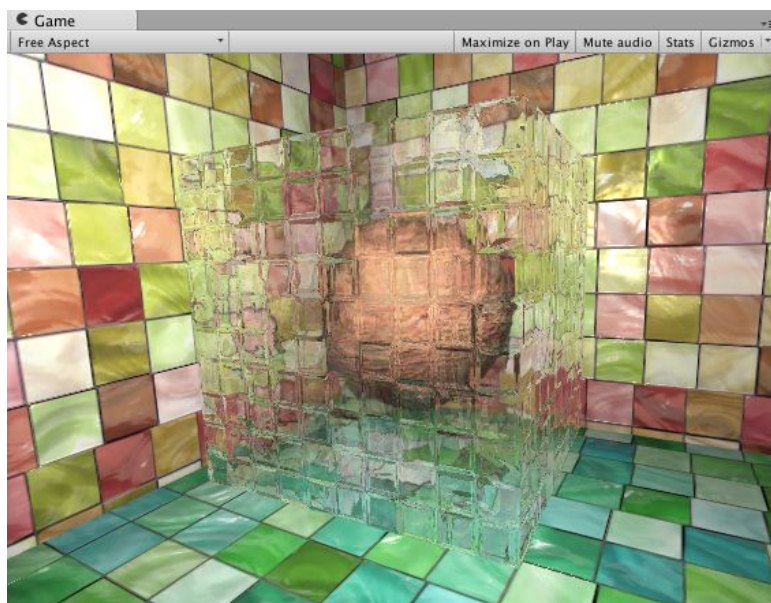
1. Scene_10_2_2のUnity 5.2のWindow → Lighting → Skybox

2. GlassRefractionMat

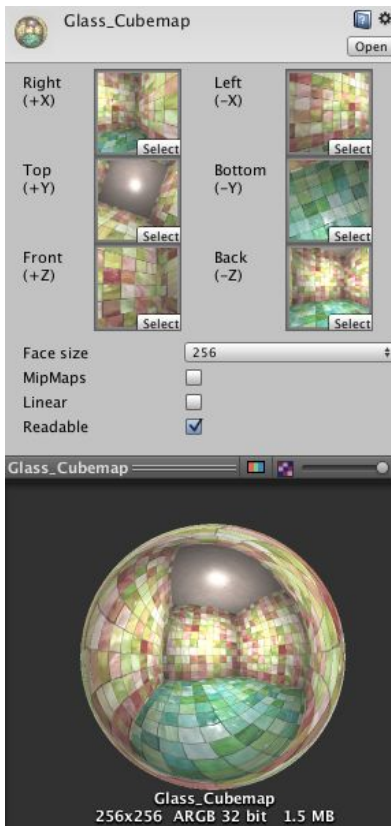
3. Unity ShaderのChapter10-GlassRefractionのUnity Shader

4. 6の2

5. 10.1.2のGameObject → Render into Cubemapの10.14のGlass_Cubemap



▲10.13 実装



▲ 10.14 □□□□□□□□

Chapter10-GlassRefraction

1 Shader

```
Properties {
    _MainTex ("Main Tex", 2D) = "white" {}
    _BumpMap ("Normal Map", 2D) = "bump" {}
    _Cubemap ("Environment Cubemap", Cube) = "_Skybox" {}
    _Distortion ("Distortion", Range(0, 100)) = 10
    _RefractAmount ("Refract Amount", Range(0.0, 1.0)) = 1.0
}
```

```

    _MainTex_BumpMap_Cubemap
    _Distortion_RefractAmount_RefractAmount0
    RefractAmount1

```

2 GrabPass

```
SubShader {
    // We must be transparent, so other objects are drawn before this one.
    Tags { "Queue"="Transparent" "RenderType"="Opaque" }

    // This pass grabs the screen behind the object into a texture.
    // We can access the result in the next pass as _RefractionTex
    GrabPass { " RefractionTex" }
```

SubShaderTransparentRenderTypeOpaqueQueueTransparentRenderTypeShader Replacement13

GrabPassPassPass

3PassShader

```
sampler2D _MainTex;
float4 _MainTex_ST;
sampler2D _BumpMap;
float4 _BumpMap_ST;
samplerCUBE _Cubemap;
float _Distortion;
fixed _RefractAmount;
sampler2D _RefractionTex;
float4 _RefractionTex_TexelSize;
```

_RefractionTex_RefractAmountGrabPass_RefractAmount_RefractAmount256x512(1/256, 1/512)

4

```
v2f vert (a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.scrPos = ComputeGrabScreenPos(o.pos);

    o.uv.xy = TRANSFORM_TEX(v.texcoord, _MainTex);
    o.uv.zw = TRANSFORM_TEX(v.texcoord, _BumpMap);

    float3 worldPos = mul(_Object2World, v.vertex).xyz;
    fixed3 worldNormal = UnityObjectToWorldNormal(v.normal);
    fixed3 worldTangent = UnityObjectToWorldDir(v.tangent.xyz);
    fixed3 worldBinormal = cross(worldNormal, worldTangent) * v.tangent.w;

    o.TtoW0 = float4(worldTangent.x, worldBinormal.x, worldNormal.x, worldPos.x);
    o.TtoW1 = float4(worldTangent.y, worldBinormal.y, worldNormal.y, worldPos.y);
    o.TtoW2 = float4(worldTangent.z, worldBinormal.z, worldNormal.z, worldPos.z);

    return o;
}
```

ComputeGrabScreenPosUnityCG.cgincComputeScreenPos_MainTex_BumpMapfloat4xyzzwCubemapTtoW0TtoW1TtoW2xyz3xyzTtoW0w

5

```
fixed4 frag (v2f i) : SV_Target {
    float3 worldPos = float3(i.TtoW0.w, i.TtoW1.w, i.TtoW2.w);
```

```

fixed3 worldViewDir = normalize(UnityWorldSpaceViewDir(worldPos));

// Get the normal in tangent space
fixed3 bump = UnpackNormal(tex2D(_BumpMap, i.uv.zw));

// Compute the offset in tangent space
float2 offset = bump.xy * _Distortion * _RefractionTex_TexelSize.xy;
i.scrPos.xy = offset + i.scrPos.xy;
fixed3 refrCol = tex2D(_RefractionTex, i.scrPos.xy/i.scrPos.w).rgb;

// Convert the normal to world space
bump = normalize(half3(dot(i.TtoW0.xyz, bump), dot(i.TtoW1.xyz, bump), dot(i.TtoW2.xyz, bump)));
fixed3 reflDir = reflect(-worldViewDir, bump);
fixed4 texColor = tex2D(_MainTex, i.uv.xy);
fixed3 reflCol = texCUBE(_Cubemap, reflDir).rgb * texColor.rgb;

fixed3 finalColor = reflCol * (1 - _RefractAmount) + refrCol * _RefractAmount;

return fixed4(finalColor, 1);
}

```

TtoW0 w TtoW1 TtoW2
 _Distortion _RefractionTex_TexelSize _Distortion
 scrPos 4.9.3 _RefractionTex

TtoW0 TtoW1 TtoW2
 Cubemap

_RefractAmount

Glass_Diffuse.jpg Glass_Normal.jpg Main Tex Normal Map
 Glass_Cubemap Environment Cubemap _RefractAmount 10.13

GrabPass GrabPass

- GrabPass { } Pass _GrabTexture Unity
- GrabPass { "TextureName" } Pass TextureName Unity

10.2.3 vs. GrabPass

GrabPass 10.2.1 + GrabPass
 Shader Render Target Shader

GrabPass
 GrabPass
 GrabPass
 CPU back buffer CPU GPU

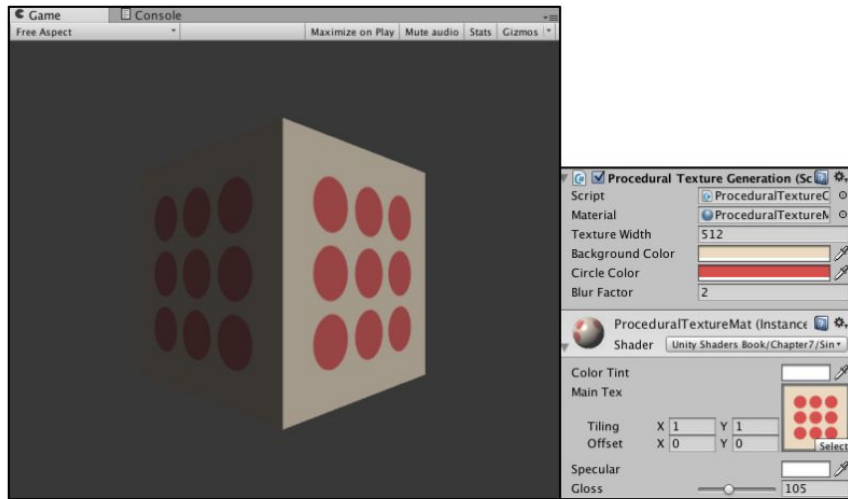
Unity 5 Unity **Command Buffers** Unity
Unity
<http://docs.unity3d.com/Manual/Graphics CommandBuffers.html> Unity
Unity

10.3 □□□□

Procedural Texture

10.3.1 Unity

10.15



▲10.15 □□□□□□□□

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □

```
1Scene_10_3_1Unity 5.2
Window → Lighting → Skybox
```

ProceduralTextureMat

3D Unity Shader——Chapter7-SingleTexture2

□4□□□□□□□□□□2□□□□□□□□□□

```
05ProceduralTextureMatProceduralTextureGeneration.cs4
```

ProceduralTextureGeneration.cs

1

```
[ExecuteInEditMode]
public class ProceduralTextureGeneration : MonoBehaviour {
```

2

```
public Material material = null;
```

3

```
#region Material properties
[SerializeField, SetProperty("textureWidth")]
private int m_textureWidth = 512;
public int textureWidth {
    get {
        return m_textureWidth;
    }
    set {
        m_textureWidth = value;
        _UpdateMaterial();
    }
}

[SerializeField, SetProperty("backgroundColor")]
private Color m_backgroundColor = Color.white;
public Color backgroundColor {
    get {
        return m_backgroundColor;
    }
    set {
        m_backgroundColor = value;
        _UpdateMaterial();
    }
}

[SerializeField, SetProperty("circleColor")]
private Color m_circleColor = Color.yellow;
public Color circleColor {
    get {
        return m_circleColor;
    }
    set {
        m_circleColor = value;
        _UpdateMaterial();
    }
}

[SerializeField, SetProperty("blurFactor")]
private float m_blurFactor = 2.0f;
public float blurFactor {
    get {
        return m_blurFactor;
    }
    set {
        m_blurFactor = value;
        _UpdateMaterial();
    }
}
}
#endregion
```

#region #endregion

```
0004get/setsetSetProperty
https://github.com/LMNRY/SetProperty/blob/master/Scripts/SetPropertyExample.cs
UpdateMaterial
```

4 Texture2D

```
private Texture2D m_generatedTexture = null;
```

5 Start

```
void Start () {
    if (material == null) {
        Renderer renderer = gameObject.GetComponent<Renderer>();
        if (renderer == null) {
            Debug.LogWarning("Cannot find a renderer.");
            return;
        }

        material = renderer.sharedMaterial;
    }

    _UpdateMaterial();
}
```

material
_UpdateMaterial

6 _UpdateMaterial

```
private void _UpdateMaterial() {
    if (material != null) {
        m_generatedTexture = _GenerateProceduralTexture();
        material.SetTexture("_MainTex", m_generatedTexture);
    }
}
```

material _GenerateProceduralTexture
m_generatedTexture Material.SetTexture material
_MainTex

7 _GenerateProceduralTexture

```
private Texture2D _GenerateProceduralTexture() {
    Texture2D proceduralTexture = new Texture2D(textureWidth, textureWidth);

    // 
    float circleInterval = textureWidth / 4.0f;
    // 
    float radius = textureWidth / 10.0f;
    // 
    float edgeBlur = 1.0f / blurFactor;

    for (int w = 0; w < textureWidth; w++) {
        for (int h = 0; h < textureWidth; h++) {
            // 
            Color pixel = backgroundColor;

            // 9
            for (int i = 0; i < 3; i++) {
                for (int j = 0; j < 3; j++) {
                    // 
                    Vector2 circleCenter = new Vector2(circleInterval * (i + 1), circleInterval * (j + 1));
```

```

        // 距離を計算
        float dist = Vector2.Distance(new Vector2(w, h), circleCenter) - radius;

        // カラーを計算
        Color color = _MixColor(circleColor, new Color(pixel.r, pixel.g,
        pixel.b, 0.0f), Mathf.SmoothStep(0f, 1.0f, dist * edgeBlur));

        // 最終カラーを計算
        pixel = _MixColor(pixel, color, color.a);
    }
}

proceduralTexture.SetPixel(w, h, pixel);
}

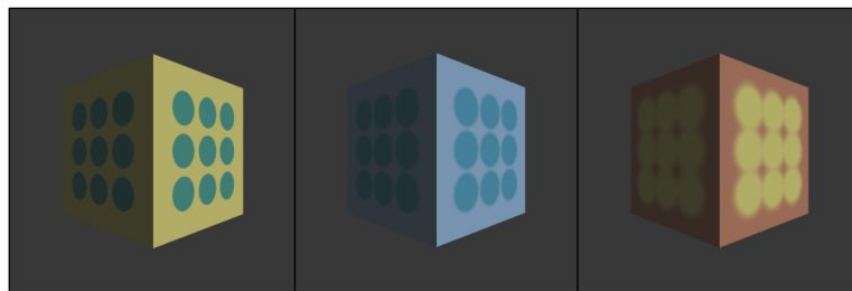
proceduralTexture.Apply();

return proceduralTexture;
}

```

このコードを実行すると、テクスチャが生成され、Texture2D.Apply()で適用されます。

このコードはUnity 10.15で動作しますが、Unity 10.16では動作しません。



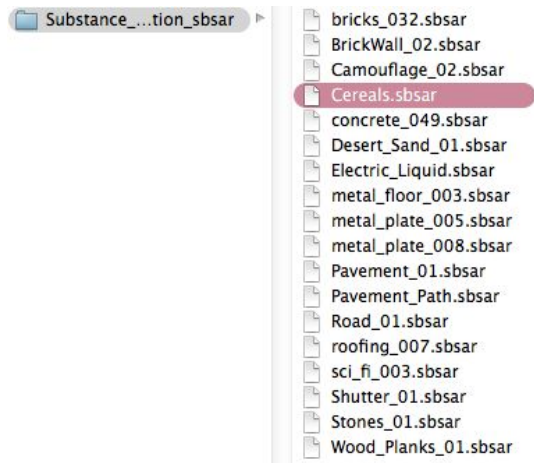
▲10.16 距離を計算するコード

このコードはUnity 10.16で動作しますが、Unity 10.17では動作しません。

10.3.2 Unity Procedural Materials

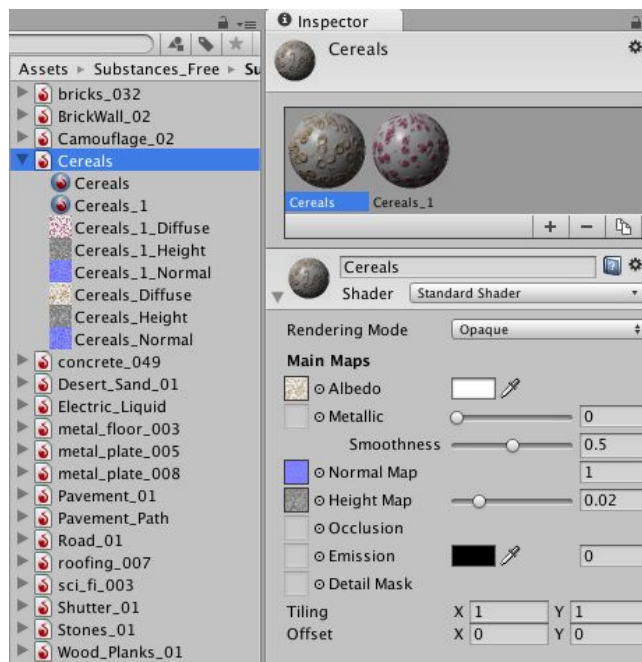
Unity 10.15で導入されたProcedural Materialsは、UnityのProcedural Materials Systemの一部です。UnityのProcedural Materials Systemは、UnityのProcedural Materials Systemの一部です。

Substance Designerは、UnityのProcedural Materials Systemの一部です。Substance Designerは、UnityのProcedural Materials Systemの一部です。Substance Designerは、UnityのProcedural Materials Systemの一部です。



▲10.17 .sbsarをSubstanceに

UnityのUnityにはProcedural Material Assetという機能があります。10.18のようにCerealsのCereals_1のDiffuseをUnityのCereals_DiffuseにCereals_1_Diffuseに



▲10.18 Unity Shader

Unity ShaderのInspectorでScene_10_3_2のCerealsのInspectorで

Scene_10_3_2のInspectorでCerealsのInspectorで10.19のようにCerealsのInspectorで



▲10.19

Shader

11 本章小结

本章主要介绍了Unity Shader的编写方法，包括11.1节中的Unity Shader的编写方法，11.2节中的Unity Shader的编写方法，以及11.3节中的Unity Shader的编写方法。

11.1 Unity Shader

Unity Shader Shader Shader 11.1

11.1 Unity

_Time	float4	t(t/20, t, 2t, 3t)
_SinTime	float4	t(t/8, t/4, t/2, t)
_CosTime	float4	t(t/8, t/4, t/2, t)
unity_DeltaTime	float4	dt(dt, 1/dt, smoothDt, 1/smoothDt)

11.2 贴图

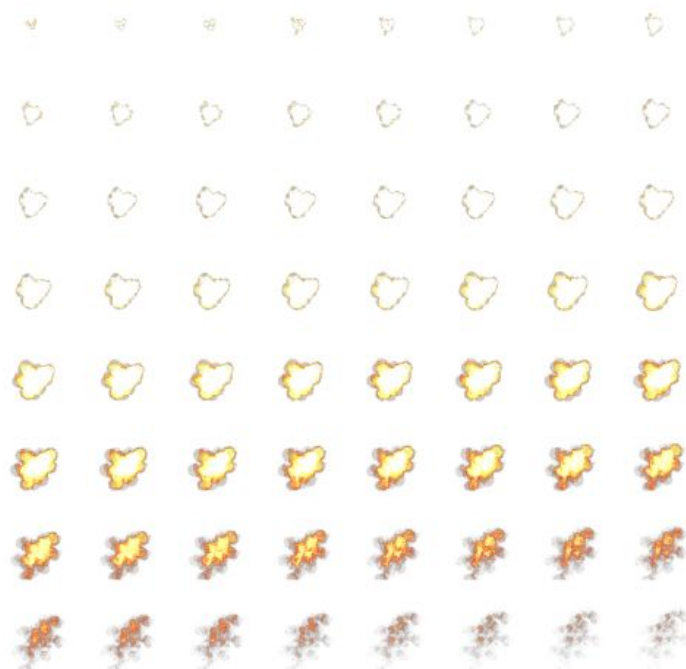
在本章中，我们将学习如何为游戏中的物体添加贴图，并了解贴图的加载和渲染过程。

11.2.1 贴图加载

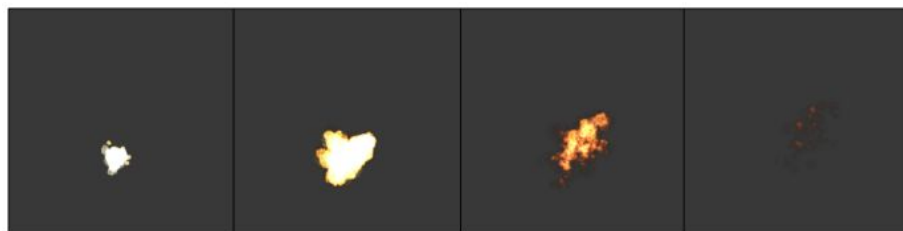
在Unity中，贴图是通过Texture类来管理的。我们可以使用Texture.Load方法将贴图加载到内存中，或者使用Texture2D类来创建和管理贴图。在本节中，我们将学习如何使用Texture2D类来加载和管理贴图。

首先，我们需要将贴图文件放入项目的Assets/Textures/Chapter11/Boom.png文件中。然后，我们可以使用以下代码来加载贴图：

我们将使用8 × 8的贴图来创建爆炸效果。在11.2节中，我们将学习如何使用贴图来创建爆炸效果。



▲图11.1 爆炸效果贴图



▲图11.2 爆炸效果动画

在Unity中，我们可以使用以下代码来加载和管理贴图：

1. Scene_11_2_1 Unity 5.2 Window → Lighting → Skybox

2. ImageSequenceAnimationMat

3. Unity Shader Chapter11-ImageSequenceAnimation Shader 2

4. Quad 2

Chapter11-ImageSequenceAnimation

1.

```
Properties {
    _Color ("Color Tint", Color) = (1, 1, 1, 1)
    _MainTex ("Image Sequence", 2D) = "white" {}
    _HorizontalAmount ("Horizontal Amount", Float) = 4
    _VerticalAmount ("Vertical Amount", Float) = 4
    _Speed ("Speed", Range(1, 100)) = 30
}
```

_MainTex _HorizontalAmount _VerticalAmount _Speed

2. Pass

```
SubShader {
    Tags {"Queue"="Transparent" "IgnoreProjector"="True" "RenderType"="Transparent"}

    Pass {
        Tags { "LightMode"="ForwardBase" }

        ZWrite Off
        Blend SrcAlpha OneMinusSrcAlpha
    }
}
```

Queue RenderType Transparent IgnoreProjector True Pass Blend

3. v2f

```
v2f vert (a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);
    o.uv = TRANSFORM_TEX(v.texcoord, _MainTex);
    return o;
}
```

4.



▲11.3 forest-background © 2012-2013 Julien Jorge julien.jorge@stuff-o-matic.com

森林背景

1. 在 Unity 5.2 中，通过 Window → Lighting → Skybox 设置 2D 背景。

2. 使用 ScrollingBackgroundMat。

3. 使用 Unity Shader Shader Chapter11-ScrollingBackground。

4. 使用 Quad 设置 2D 背景。

Chapter11-ScrollingBackground 设置。

1. 设置。

```
Properties {
    _MainTex ("Base Layer (RGB)", 2D) = "white" {}
    _DetailTex ("2nd Layer (RGB)", 2D) = "white" {}
    _ScrollX ("Base layer Scroll Speed", Float) = 1.0
    _Scroll2X ("2nd layer Scroll Speed", Float) = 1.0
    _Multiplier ("Layer Multiplier", Float) = 1
}
```

设置 _MainTex、_DetailTex、_ScrollX、_Scroll2X、_Multiplier。

2. 设置。


```

v2f vert (a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.uv.xy = TRANSFORM_TEX(v.texcoord, _MainTex) + frac(float2(_ScrollX, 0.0) * _Time.y);
    o.uv.zw = TRANSFORM_TEX(v.texcoord, _DetailTex) + frac(float2(_Scroll2X, 0.0) * _Time.y);

    return o;
}

```

TRANSFORM_TEX
 o.uv

3

```

fixed4 frag (v2f i) : SV_Target {
    fixed4 firstLayer = tex2D(_MainTex, i.uv.xy);
    fixed4 secondLayer = tex2D(_DetailTex, i.uv.zw);

    fixed4 c = lerp(firstLayer, secondLayer, secondLayer.a);
    c.rgb *= _Multiplier;

    return c;
}

```

i.uv.xy i.uv.zw CG lerp
 _Multiplier

4 Fallback VertexLit Fallback

Fallback "VertexLit"

Assets/Textures/Chapter11/Far_Background.png
 Assets/Textures/Chapter11/Near_Background.png Base Layer 2nd Layer
 Base Layer 2nd Layer Base Layer
 2nd Layer 11.3

11.3 水面

水面は、ゲームエンジンで最も重要な要素の一つです。水面の表現は、ゲームのリアル感と美しさを大きく左右します。本章では、Unity 5.2で水面を表現する方法について詳しく説明します。

11.3.1 水面の表現

水面の表現には、2Dと3Dの2つの方法があります。本章では、2Dの水面表現について詳しく説明します。



▲図11.4 2D水面の表現

1. 水面の表現方法

1. Unity 5.2のScene_11_3_1で、Window → Lighting → Skyboxを選択し、2DのSkyboxを選択します。

2. WaterMat1とWaterMat2の2つのMaterialを作成します。

3. Unity ShaderのShaderをChapter11-WaterのShaderに設定します。

4. Waterの2つのMaterialを作成します。

Chapter11-WaterのShaderのProperties

1. Properties

```
Properties {
    _MainTex ("Main Tex", 2D) = "white" {}
    _Color ("Color Tint", Color) = (1, 1, 1, 1)
    _Magnitude ("Distortion Magnitude", Float) = 1
    _Frequency ("Distortion Frequency", Float) = 1
    _InvWaveLength ("Distortion Inverse Wave Length", Float) = 10
    _Speed ("Speed", Float) = 0.5
}
```

2. _MainTex、_Color、_Magnitude、_Frequency、_InvWaveLength、_Speedの値を設定します。

3. SubShaderの値を設定します。

```
SubShader {
    // Need to disable batching because of the vertex animation
    Tags {"Queue"="Transparent" "IgnoreProjector"="True" "RenderType"="Transparent" "DisableBatching"="True"}
```

QueueIgnoreProjectorRenderType——
DisableBatching 3.3.3SubShaderUnitySubShaderShaderShaderShader
 Shader

3Pass

```
Pass {
    Tags { "LightMode"="ForwardBase" }

    ZWrite Off
    Blend SrcAlpha OneMinusSrcAlpha
    Cull Off
```

4

```
v2f vert(a2v v) {
    v2f o;

    float4 offset;
    offset.yzw = float3(0.0, 0.0, 0.0);
    offset.x = sin(_Frequency * _Time.y + v.vertex.x * _InvWaveLength + v.vertex.y * _InvWaveLength + v.vertex.z * _InvWaveLength) * _Magnitude;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex + offset);

    o.uv = TRANSFORM_TEX(v.texcoord, _MainTex);
    o.uv += float2(0.0, _Time.y * _Speed);

    return o;
}
```

xzyzw0_Frequency_Time.y
 _InvWaveLength
 _Magnitude

_Time.y_Speed

5

```
fixed4 frag(v2f i) : SV_Target {
    fixed4 c = tex2D(_MainTex, i.uv);
    c.rgb *= _Color.rgb;

    return c;
}
```

6FallbackTransparent/VertexLitFallback

Fallback "Transparent/VertexLit"

Assets/Textures/Chapter11/Water.psd Main Tex Water 11.4

11.3.2

Billboarding

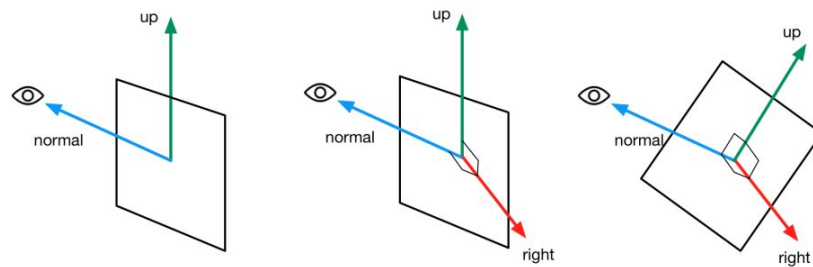
normal up right anchor location

3 (0, 1, 0)

$$\text{right} = \text{up} \times \text{normal}$$

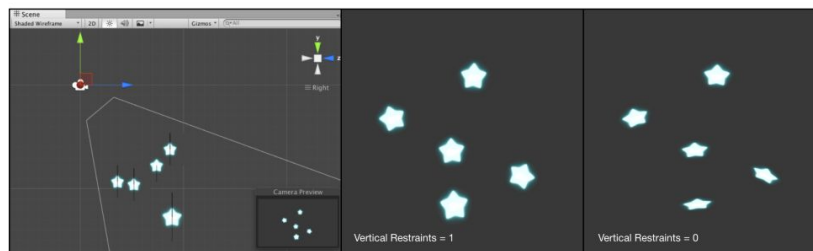
$$\text{up}' = \text{normal} \times \text{right}$$

3 11.5



▲11.5

Unity 11.6



▲11.6 Vertical Restraints 1 0 (0, 1, 0)

Scene_11_3_2 Unity 5.2 Window → Lighting → Skybox

2BillboardMat

3Unity ShaderShaderChapter11-BillboardShader2

4Quad2

Chapter11-Billboard

1

```
Properties {
    _MainTex ("Main Tex", 2D) = "white" {}
    _Color ("Color Tint", Color) = (1, 1, 1, 1)
    _VerticalBillboarding ("Vertical Restraints", Range(0, 1)) = 1
}
```

_MainTex_Color_VerticallBillboarding

2SubShader

```
SubShader {
    // Need to disable batching because of the vertex animation
    Tags { "Queue"="Transparent" "IgnoreProjector"="True" "RenderType"="Transparent" "DisableBatching"=
```

QueueIgnoreProjectorRenderType

DisableBatching 3.3.3SubShaderUnitySubShaderShaderShaderShader

3Pass

```
Pass {
    Tags { "LightMode"="ForwardBase" }

    ZWrite Off
    Blend SrcAlpha OneMinusSrcAlpha
    Cull Off
```

4

```
// Suppose the center in object space is fixed
float3 center = float3(0, 0, 0);
float3 viewer = mul(_World2Object,float4(_WorldSpaceCameraPos, 1));
```

3_VerticallBillboarding

```
float3 normalDir = viewer - center;
// If _VerticalBillboarding equals 1, we use the desired view dir as the normal dir
// Which means the normal dir is fixed
// Or if _VerticalBillboarding equals 0, the y of normal is 0
// Which means the up dir is fixed
```

```
normalDir.y = normalDir.y * _VerticalBillboarding;
normalDir = normalize(normalDir);
```

`_VerticalBillboarding` 1 0 (0, 1, 0)

`y`

```
// Get the approximate up dir
// If normal dir is already towards up, then the up dir is towards front
float3 upDir = abs(normalDir.y) > 0.999 ? float3(0, 0, 1) : float3(0, 1, 0);
float3 rightDir = normalize(cross(upDir, normalDir));
upDir = normalize(cross(normalDir, rightDir));
```

3 3

```
float3 centerOffs = v.vertex.xyz - center;
float3 localPos = center + rightDir * centerOffs.x + upDir * centerOffs.y + normalDir * centerOffs.z;
```

```
o.pos = mul(UNITY_MATRIX_MVP, float4(localPos, 1));
```

5

```
fixed4 frag (v2f i) : SV_Target {
    fixed4 c = tex2D (_MainTex, i.uv);
    c.rgb *= _Color.rgb;

    return c;
}
```

6 Fallback Transparent/VertexLit Fallback

Fallback "Transparent/VertexLit"

Unity Quad Plane `v.vertex`

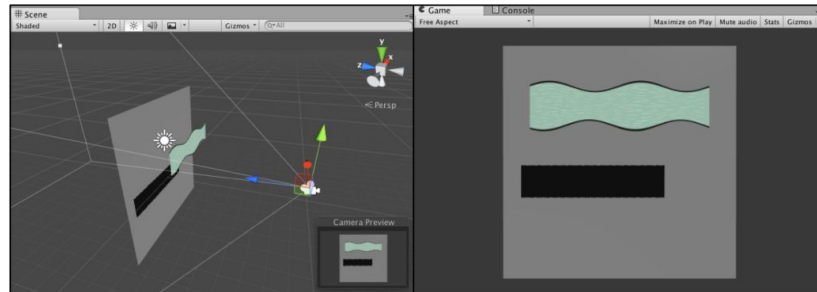
Assets/Textures/Chapter11/star.png Main Tex 11.6

11.3.3

11.3.2 SubShader DisableBatching Unity Shader Draw Call

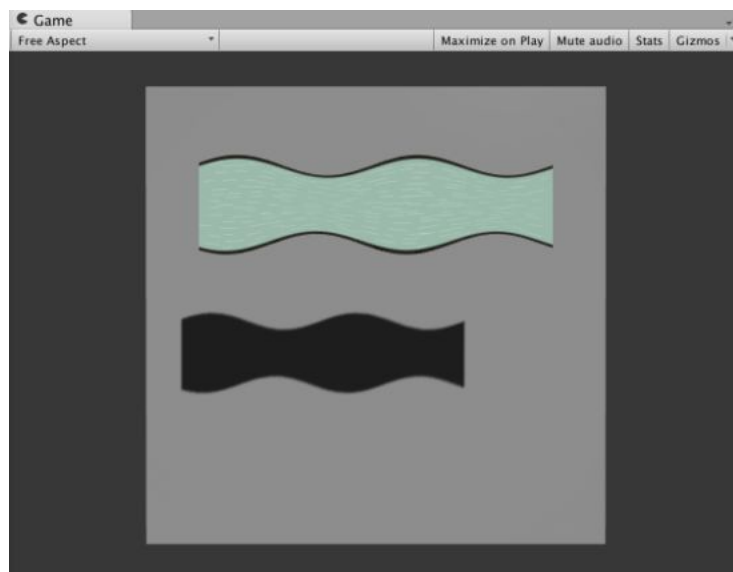
9.4 Diffuse Pass
 Unity ShadowCaster Pass
 ShadowCaster Pass Pass Unity
 ShadowCaster Pass Pass
 Fallback Transparent/VertexLit Transparent/VertexLit ShadowCaster Pass
 9.4.5

Scene_11_3_3 11.3.1
 “ ” Unity Shader Fallback
 VertexLit Unity Fallback VertexLit ShadowCaster Pass 11.7



▲ 11.7 ShadowCaster Pass

Water
 ShadowCaster Pass Chapter11-VertexAnimationWithShadow Unity
 Shader Shader 11.8



▲ 11.8 ShadowCaster Pass

Shader ShadowCaster Pass

```
// Pass to render object as a shadow caster
Pass {
    Tags { "LightMode" = "ShadowCaster" }

    CGPROGRAM
    #pragma vertex vert
```

```

#pragma fragment frag

#pragma multi_compile_shadowcaster

#include "UnityCG.cginc"

float _Magnitude;
float _Frequency;
float _InvWaveLength;
float _Speed;

struct a2v {
    float4 vertex : POSITION;
    float4 texcoord : TEXCOORD0;
};

struct v2f {
    V2F_SHADOW_CASTER;
};

v2f vert(a2v i) {
    v2f o;

    float4 offset;

    offset.yzw = float3(0.0, 0.0, 0.0);

    offset.x = sin(_Frequency * _Time.y + v.vertex.x * _InvWaveLength + v.vertex.y
        * _InvWaveLength + v.vertex.z * _InvWaveLength) * _Magnitude;

    v.vertex = v.vertex + offset;

    TRANSFER_SHADOW_CASTER_NORMALOFFSET(o)

    return o;
}

fixed4 frag(v2f i) : SV_Target {
    SHADOW_CASTER_FRAGMENT(i)
}
ENDCG
}

```

Pass
 UnityV2F_SHADOW_CASTERTRANSFER_SHADOW_CASTER_NORMALOFFSET
 TRANSFER_SHADOW_CASTERSHADOW_CASTER_FRAGMENT
 v2fV2F_SHADOW_CASTER
 TRANSFER_SHADOW_CASTER_NORMALOFFSETUnity
 SHADOW_CASTER_FRAGMENTUnity

Unity3UnityCG.cgincPass
 TRANSFER_SHADOW_CASTER_NORMALOFFSETv
 vvertexv.normalappdata_base
 v.vertex
 TRANSFER_SHADOW_CASTER_NORMALOFFSET15.1Pass

4 4

Shader 4
4

12 4

Unity 4
4

13 4

Unity 4

14 4

4

15 4

“ ” 4

16 Unity 4

Unity 4

12 屏幕后处理

本章主要介绍 **screen post-processing effects** 的相关内容。
12.1 介绍在 Unity 中如何实现屏幕后处理。
12.2 介绍在 Unity 中如何实现屏幕后处理。
12.3 介绍在 Unity 中如何实现 Sobel 算子。
12.4 介绍在 Unity 中如何实现 12.5 12.6 介绍在 Unity 中实现 Bloom 效果。

12.1 後處理效果器基礎

後處理效果器是 Unity 中非常重要的一部分，它負責在渲染完場景後，對渲染結果進行後處理。本章將介紹後處理效果器的基礎知識，包括 Depth of Field、Motion Blur 等。

本章將介紹 Unity 中後處理效果器的基礎知識，包括 **OnRenderImage** 的用法。

```
MonoBehaviour.OnRenderImage (RenderTexture src, RenderTexture dest)
```

Unity 中後處理效果器的基礎知識，包括 **OnRenderImage** 的用法。Graphics.Blit 是 Unity 中用於後處理效果器的重要方法，它用於將一個 RenderTexture 渲染到另一個 RenderTexture 上。Graphics.Blit 的用法如下：

```
public static void Blit(Texture src, RenderTexture dest);  
public static void Blit(Texture src, RenderTexture dest, Material mat, int pass = -1);  
public static void Blit(Texture src, Material mat, int pass = -1);
```

src 是源 RenderTexture，dest 是目標 RenderTexture。mat 是後處理效果器所使用的 Shader。pass 是 Shader 的 pass 索引，默認為 -1。Graphics.Blit 的用法如下：

OnRenderImage 是 MonoBehaviour 的一個方法，它用於在渲染完場景後，對渲染結果進行後處理。OnRenderImage 的用法如下：

Unity 中後處理效果器的基礎知識，包括 Graphics.Blit 的用法。Graphics.Blit 是 Unity 中用於後處理效果器的重要方法，它用於將一個 RenderTexture 渲染到另一個 RenderTexture 上。Graphics.Blit 的用法如下：

Unity Shader 是 Unity 中用於後處理效果器的重要方法，它用於將一個 RenderTexture 渲染到另一個 RenderTexture 上。Unity Shader 的用法如下：

PostEffectsBase.cs 是 Unity 中用於後處理效果器的重要方法，它用於將一個 RenderTexture 渲染到另一個 RenderTexture 上。PostEffectsBase.cs 的用法如下：

1. 在 Unity 中，後處理效果器是通過 MonoBehaviour 類來實現的。後處理效果器的基礎知識，包括 Graphics.Blit 的用法。Graphics.Blit 是 Unity 中用於後處理效果器的重要方法，它用於將一個 RenderTexture 渲染到另一個 RenderTexture 上。Graphics.Blit 的用法如下：

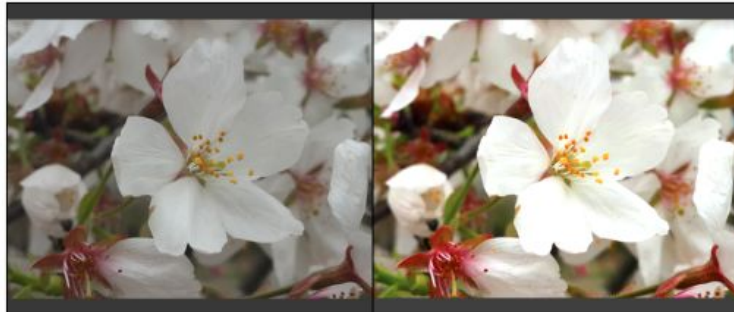
```
[ExecuteInEditMode]  
[RequireComponent (typeof(Camera))]  
public class PostEffectsBase : MonoBehaviour {
```

2. 在 Unity 中，後處理效果器是通過 MonoBehaviour 類來實現的。後處理效果器的基礎知識，包括 Graphics.Blit 的用法。Graphics.Blit 是 Unity 中用於後處理效果器的重要方法，它用於將一個 RenderTexture 渲染到另一個 RenderTexture 上。Graphics.Blit 的用法如下：

```
// Called when start  
protected void CheckResources() {  
    bool isSupported = CheckSupport();
```


12.2 画面の調整

12.1 画面の調整
12.1 画面の調整



▲12.1 画面の調整

画面の調整

1 画面の調整
Scene_12_2 Unity 5.2 画面の調整
Window → Lighting → Skybox

2 画面の調整
Assets/Textures/Chapter12/Sakura0.jpg 画面の調整
Sakura0.jpg Sprite

3 画面の調整
BrightnessSaturationAndContrast.cs

4 Unity Shader
Chapter12-BrightnessSaturationAndContrast

BrightnessSaturationAndContrast.cs

1 画面の調整

```
public class BrightnessSaturationAndContrast : PostEffectsBase {
```

2 画面の調整
Shader

```
public Shader briSatConShader;  
private Material briSatConMaterial;  
public Material material {  
    get {  
        briSatConMaterial = CheckShaderAndCreateMaterial(briSatConShader, briSatConMaterial);  
        return briSatConMaterial;  
    }  
}
```

画面の調整
briSatConShader Shader
Chapter12-BrightnessSaturationAndContrast
briSatConMaterial material

material.getCheckShaderAndCreateMaterial

3

```
[Range(0.0f, 3.0f)]
public float brightness = 1.0f;

[Range(0.0f, 3.0f)]
public float saturation = 1.0f;

[Range(0.0f, 3.0f)]
public float contrast = 1.0f;
```

Unity.Range

4 OnRenderImage

```
void OnRenderImage(RenderTexture src, RenderTexture dest) {
    if (material != null) {
        material.SetFloat("_Brightness", brightness);
        material.SetFloat("_Saturation", saturation);
        material.SetFloat("_Contrast", contrast);

        Graphics.Blit(src, dest, material);
    } else {
        Graphics.Blit(src, dest);
    }
}
```

OnRenderImage Graphics.Blit

Shader Chapter12-BrightnessSaturationAndContrast

1

```
Properties {
    _MainTex ("Base (RGB)", 2D) = "white" {}
    _Brightness ("Brightness", Float) = 1
    _Saturation ("Saturation", Float) = 1
    _Contrast ("Contrast", Float) = 1
}
```

12.1 Graphics.Blit(src, dest, material) Shader _MainTex
_MainTex
Properties Properties
Unity Shader

2 Pass

```
SubShader {
    Pass {
        ZTest Always Cull Off ZWrite Off
```

OnRenderImage Pass Shader “ ”

3 CG

```
sampler2D _MainTex;  
half _Brightness;  
half _Saturation;  
half _Contrast;
```

4

```
struct v2f {  
    float4 pos : SV_POSITION;  
    half2 uv: TEXCOORD0;  
};  
  
v2f vert(appdata_img v) {  
    v2f o;  
  
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);  
  
    o.uv = v.texcoord;  
  
    return o;  
}
```

Unity appdata_img UnityCG.cginc

5

```
fixed4 frag(v2f i) : SV_Target {  
    fixed4 renderTex = tex2D(_MainTex, i.uv);  
  
    // Apply brightness  
    fixed3 finalColor = renderTex.rgb * _Brightness;  
  
    // Apply saturation  
    fixed luminance = 0.2125 * renderTex.r + 0.7154 * renderTex.g + 0.0721 * renderTex.b;  
    fixed3 luminanceColor = fixed3(luminance, luminance, luminance);  
    finalColor = lerp(luminanceColor, finalColor, _Saturation);  
  
    // Apply contrast  
    fixed3 avgColor = fixed3(0.5, 0.5, 0.5);  
    finalColor = lerp(avgColor, finalColor, _Contrast);  
  
    return fixed4(finalColor, renderTex.a);  
}
```

_MainTex renderTex _Brightness luminance 0 _Saturation

6 Unity Shader Fallback

Chapter12-BrightnessSaturationAndContrast
SaturationAndContrast.cs
briSatConShader
12.1



Shader Shader
Shader Shader Shader 12.2

12.3 □□□□

12.2 12.3

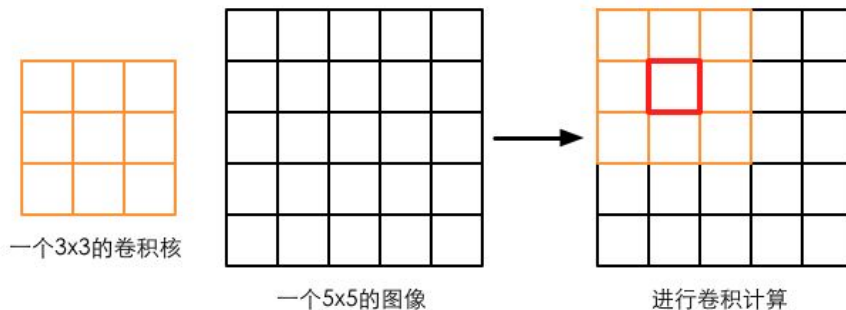


▲ 12.3 12.2

convolution

12.3.1 ☐☐☐☐☐

kernel
2x2 3x3 12.4



▲ 12.4 3×3 5×5

3x3 1/9

12.3.2 实验目的

[illegible]

Roberts

-1	0
0	1

 G_x

0	-1
1	0

 G_y

Prewitt

-1	-1	-1
0	0	0
1	1	1

 G_x

-1	0	1
-1	0	1
-1	0	1

 G_y

Sobel

-1	-2	-1
0	0	0
1	2	1

 G_x

-1	0	1
-2	0	2
-1	0	1

 G_y

12.5 3Dエッジ検出

3Dエッジ検出は、12.5節で説明する。この節では、2Dエッジ検出の結果を3Dエッジ検出に適用する方法を説明する。2Dエッジ検出の結果は、 G_x と G_y の値として得られる。

$$G = \sqrt{G_x^2 + G_y^2}$$

この式は、2Dエッジ検出の結果を3Dエッジ検出に適用するための式である。

$$G = |G_x| + |G_y|$$

この式は、2Dエッジ検出の結果を3Dエッジ検出に適用するための式である。

12.3.3 3Dエッジ検出

3Dエッジ検出は、Sobelエッジ検出の結果を3Dエッジ検出に適用する方法を説明する。

1. 3Dエッジ検出の結果をScene_12_3に適用する。Unity 5.2のバージョンでは、3Dエッジ検出の結果をScene_12_3に適用する。Window -> Lighting -> Skyboxを選択する。

2. 3Dエッジ検出の結果をAssets/Textures/Chapter12/Sakura0.jpgに適用する。Sakura0.jpgを選択する。Spriteを選択する。

3. 3Dエッジ検出の結果をEdgeDetection.csに適用する。

4. 3Dエッジ検出の結果をUnity Shaderに適用する。Shaderを選択する。Chapter12-EdgeDetectionを選択する。

5. 3Dエッジ検出の結果をEdgeDetection.csに適用する。

1. 3Dエッジ検出の結果を12.1に適用する。

```
public class EdgeDetection : PostEffectsBase {
```

2. 3Dエッジ検出の結果をShaderに適用する。

```
public Shader edgeDetectShader;
private Material edgeDetectMaterial = null;
public Material material {
    get {
        edgeDetectMaterial = CheckShaderAndCreateMaterial(edgeDetectShader, edgeDetectMaterial);
        return edgeDetectMaterial;
    }
}
```

ShaderChapter12-EdgeDetection

3

```
[Range(0.0f, 1.0f)]
public float edgesOnly = 0.0f;

public Color edgeColor = Color.black;

public Color backgroundColor = Color.white;
```

edgesOnly0edgesOnly1
backgroundColoredgeColor

4OnRenderImage

```
void OnRenderImage (RenderTexture src, RenderTexture dest) {
    if (material != null) {
        material.SetFloat("_EdgeOnly", edgesOnly);
        material.SetColor("_EdgeColor", edgeColor);
        material.SetColor("_BackgroundColor", backgroundColor);

        Graphics.Blit(src, dest, material);
    } else {
        Graphics.Blit(src, dest);
    }
}
```

OnRenderImageGraphics.Blit

ShaderChapter12-EdgeDetection

1

```
Properties {
    _MainTex ("Base (RGB)", 2D) = "white" {}
    _EdgeOnly ("Edge Only", Float) = 1.0
    _EdgeColor ("Edge Color", Color) = (0, 0, 0, 1)
    _BackgroundColor ("Background Color", Color) = (1, 1, 1, 1)
}
```

_MainTex

2Pass

```
SubShader {
    Pass {
        ZTest Always Cull Off ZWrite Off
    }
}
```

3CG

```
sampler2D _MainTex;
half4 _MainTex_TexelSize;
fixed _EdgeOnly;
```



```
        -2,  0,  2,
        -1,  0,  1};

half texColor;
half edgeX = 0;
half edgeY = 0;
for (int it = 0; it < 9; it++) {
    texColor = luminance(tex2D(_MainTex, i.uv[it]));
    edgeX += texColor * Gx[it];
    edgeY += texColor * Gy[it];
}

half edge = 1 - abs(edgeX) - abs(edgeY);

return edge;
}
```

このコードは、テクスチャの9つのサンプルからGxとGyの加重和を計算し、edgeXとedgeYの絶対値の和を1から引いてedge値を算出しています。

6番目のShaderはFallbackとしてFallback Offを設定しています。

Chapter12-EdgeDetectionのEdgeDetection.csでは、edgeDetectShaderをEdgeDetection.csで定義し、edgeDetectShaderをChapter12-EdgeDetectionの12.6でedgeOnlyとして1回呼び出しています。



▲12.6 エッジ検出結果

この結果は、Chapter13.4で説明されているように、エッジ検出の結果を示しています。

12.4 □□□□

12.3 — 12.7



▲ 12.7

12.4.1 □□□□

[illegible]

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

[illegible][illegible]

The diagram shows a 5x5 matrix on the left, an arrow pointing to the right, and a 1x5 vector on the right. The 5x5 matrix has the following values:

0.0030	0.0133	0.0219	0.0133	0.0030
0.0133	0.0596	0.0983	0.0596	0.0133
0.0219	0.0983	0.1621	0.0983	0.0219
0.0133	0.0596	0.0983	0.0596	0.0133
0.0030	0.0133	0.0219	0.0133	0.0030

The 1x5 vector has the following values:

0.0545	0.2442	0.4026	0.2442	0.0545

▲12.8 5×5 1

```

    5x5PassPass
Pass

```

12.4.2 実装

実装の準備

1 Scene_12_4 Unity 5.2のSkyboxをSkyboxとして設定する
Window → Lighting → Skybox

2 Assets/Textures/Chapter12/Sakura1.jpgをSpriteとして設定する
Sakura1.jpgをSpriteとして設定する

3 GaussianBlur.csを作成する

4 Unity ShaderのChapter12-GaussianBlurを作成する

GaussianBlur.csを作成する

12.1の実装

```
public class GaussianBlur : PostEffectsBase {
```

2 Shaderを作成する

```
public Shader gaussianBlurShader;
private Material gaussianBlurMaterial = null;

public Material material {
    get {
        gaussianBlurMaterial = CheckShaderAndCreateMaterial(gaussianBlurShader, gaussianBlurMaterial);
        return gaussianBlurMaterial;
    }
}
```

gaussianBlurShaderをshaderとしてChapter12-GaussianBlurを作成する

3 実装

```
// Blur iterations - larger number means more blur.
[Range(0, 4)]
public int iterations = 3;

// Blur spread for each iteration - larger value means more blur
[Range(0.2f, 3.0f)]
public float blurSpread = 0.6f;

[Range(1, 8)]
public int downSample = 2;
```

blurSpread、downSample、_BlurSizeをOnRenderImageで設定する
_BlurSize、downSample、_BlurSizeをOnRenderImageで設定する
downSample

4 OnRenderImageを実装する

```
/// 1st edition: just apply blur
void OnRenderImage(RenderTexture src, RenderTexture dest) {
    if (material != null) {
        int rtW = src.width;
        int rtH = src.height;
```

```

    RenderTexture buffer = RenderTexture.GetTemporary(rtW, rtH, 0);

    // Render the vertical pass
    Graphics.Blit(src, buffer, material, 0);
    // Render the horizontal pass
    Graphics.Blit(buffer, dest, material, 1);

    RenderTexture.ReleaseTemporary(buffer);
} else {
    Graphics.Blit(src, dest);
}
}

```

1. `RenderTexture.GetTemporary` creates a temporary render texture.
 2. `Graphics.Blit(src, buffer, material, 0)` renders the source texture to the buffer using the first pass of the shader.
 3. `Graphics.Blit(buffer, dest, material, 1)` renders the buffer to the destination using the second pass of the shader.
 4. `RenderTexture.ReleaseTemporary(buffer)` releases the temporary render texture.

5. `OnRenderImage` is a callback function that is called when the image is rendered.

```

/// 2nd edition: scale the render texture
void OnRenderImage (RenderTexture src, RenderTexture dest) {
    if (material != null) {
        int rtW = src.width/downSample;
        int rtH = src.height/downSample;
        RenderTexture buffer = RenderTexture.GetTemporary(rtW, rtH, 0);
        buffer.filterMode = FilterMode.Bilinear;

        // Render the vertical pass
        Graphics.Blit(src, buffer, material, 0);
        // Render the horizontal pass
        Graphics.Blit(buffer, dest, material, 1);

        RenderTexture.ReleaseTemporary(buffer);
    } else {
        Graphics.Blit(src, dest);
    }
}

```

6. `downSample` is a variable that controls the scaling of the render texture.

6. `downSample` is a variable that controls the scaling of the render texture.

```

/// 3rd edition: use iterations for larger blur
void OnRenderImage (RenderTexture src, RenderTexture dest) {
    if (material != null) {
        int rtW = src.width/downSample;
        int rtH = src.height/downSample;

        RenderTexture buffer0 = RenderTexture.GetTemporary(rtW, rtH, 0);
        buffer0.filterMode = FilterMode.Bilinear;

        Graphics.Blit(src, buffer0);

        for (int i = 0; i < iterations; i++) {
            material.SetFloat("_BlurSize", 1.0f + i * blurSpread);

            RenderTexture buffer1 = RenderTexture.GetTemporary(rtW, rtH, 0);

            // Render the vertical pass

```



```

struct v2f {
    float4 pos : SV_POSITION;
    half2 uv[5]: TEXCOORD0;
};

v2f vertBlurVertical(appdata_img v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    half2 uv = v.texcoord;

    o.uv[0] = uv;
    o.uv[1] = uv + float2(0.0, _MainTex_TexelSize.y * 1.0) * _BlurSize;
    o.uv[2] = uv - float2(0.0, _MainTex_TexelSize.y * 1.0) * _BlurSize;
    o.uv[3] = uv + float2(0.0, _MainTex_TexelSize.y * 2.0) * _BlurSize;
    o.uv[4] = uv - float2(0.0, _MainTex_TexelSize.y * 2.0) * _BlurSize;

    return o;
}

```

12.4.1 5x5 5 3 weight weight sum sum

4

5 Pass

```

fixed4 fragBlur(v2f i) : SV_Target {
    float weight[3] = {0.4026, 0.2442, 0.0545};

    fixed3 sum = tex2D(_MainTex, i.uv[0]).rgb * weight[0];

    for (int it = 1; it < 3; it++) {
        sum += tex2D(_MainTex, i.uv[it]).rgb * weight[it];
        sum += tex2D(_MainTex, i.uv[2*it]).rgb * weight[it];
    }

    return fixed4(sum, 1.0);
}

```

12.4.1 5x5 5 3 weight weight sum sum

6 Pass

```

ZTest Always Cull Off ZWrite Off

Pass {
    NAME "GAUSSIAN_BLUR_VERTICAL"

    CGPROGRAM

    #pragma vertex vertBlurVertical
    #pragma fragment fragBlur

    ENDCG
}

Pass {

```

```
NAME "GAUSSIAN_BLUR_HORIZONTAL"

CGPROGRAM

#pragma vertex vertBlurHorizontal
#pragma fragment fragBlur

ENDCG
}
```

PassNAME3.3.3Bloom12.5PassShaderPassPass

7ShaderFallback

Fallback Off

Chapter12-GaussianBlurGaussianBlur.csgaussianBlurShaderGaussianBlur.csgaussianBlurShaderChapter12-GaussianBlur

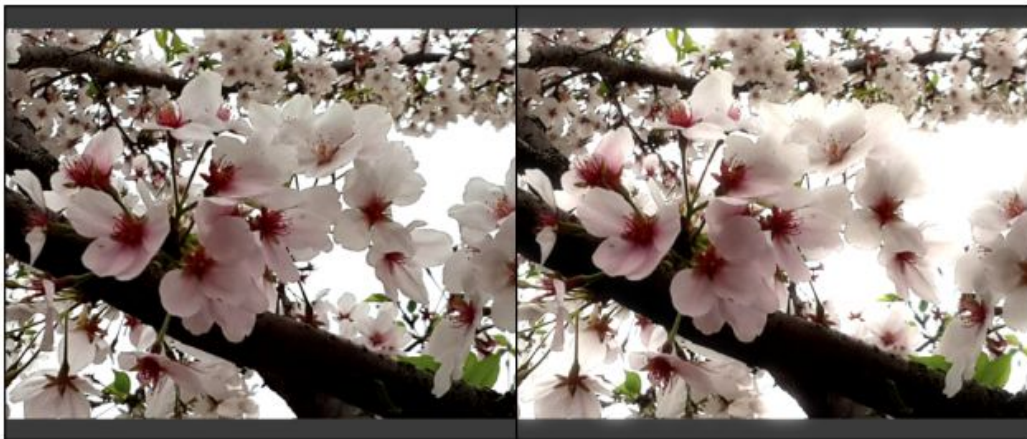
12.5 Bloom

Bloomは、レンダリング時に、テクスチャの輝度情報を、周囲のピクセルにぼかし込む効果を実現する。この効果は、映画『Elephants Dream』のBloomシーンで、

レンダリングされた。Bloomは、2003年12月10日に発表された。



▲12.9 レンダリングされたBloom効果のシーン



▲12.10 レンダリングされたBloom効果のシーン

Bloomは、レンダリング時に、テクスチャの輝度情報を、周囲のピクセルにぼかし込む効果を実現する。この効果は、映画『Elephants Dream』のBloomシーンで、

レンダリングされた。

`_BloomGraphics.Blit (src, dest, material, 3) Shader Pass`
`dest`

Shader Chapter12-Bloom

1

```
Properties {
    _MainTex ("Base (RGB)", 2D) = "white" {}
    _Bloom ("Bloom (RGB)", 2D) = "black" {}
    _LuminanceThreshold ("Luminance Threshold", Float) = 0.5
    _BlurSize ("Blur Size", Float) = 1.0
}
```

`_MainTex`
`_Bloom`
`_LuminanceThreshold`
`_BlurSize` 12.4

2 `CGINCLUDE` SubShader `CGINCLUDE`
`ENDCG`

```
SubShader {
    CGINCLUDE
    ...
    ENDCG
    ...
}
```

3

```
sampler2D _MainTex;
half4 _MainTex_TexelSize;
sampler2D _Bloom;
float _LuminanceThreshold;
float _BlurSize;
```

4

```
struct v2f {
    float4 pos : SV_POSITION;
    half2 uv : TEXCOORD0;
};

v2f vertExtractBright(appdata_img v) {
    v2f o;

    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.uv = v.texcoord;
```


6 6 Pass Bloom 4 Pass

```
ZTest Always Cull Off ZWrite Off

Pass {
    CGPROGRAM
    #pragma vertex vertExtractBright
    #pragma fragment fragExtractBright

    ENDCG
}

UsePass "Unity Shaders Book/Chapter 12/Gaussian Blur/GAUSSIAN_BLUR_VERTICAL"

UsePass "Unity Shaders Book/Chapter 12/Gaussian Blur/GAUSSIAN_BLUR_HORIZONTAL"

Pass {
    CGPROGRAM
    #pragma vertex vertBloom
    #pragma fragment fragBloom

    ENDCG
}
```

Pass 12.4 Pass UsePass
Pass Unity Pass Name
UsePass

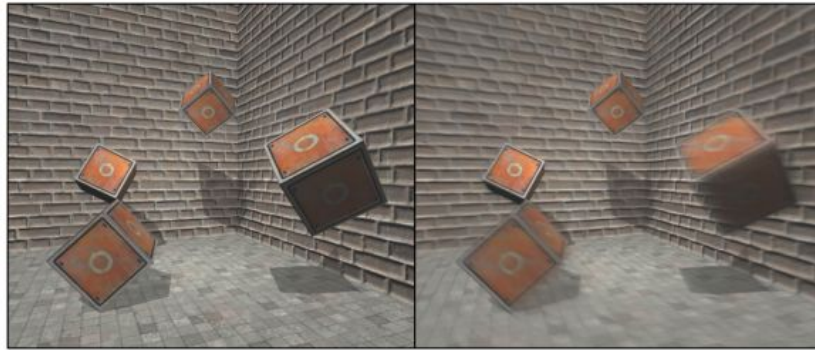
7 Shader Fallback

```
Fallback Off
```

Chapter12-Bloom Bloom.cs bloomShader
Bloom.cs bloomShader Chapter12-Bloom

12.6 実装

この章では、運動ぼかしを実装するためのシェーダーとC#スクリプトを作成します。12.11のイメージを参考に、運動ぼかしを実装します。



▲12.11 運動ぼかしの実装イメージ

運動ぼかしを実装するために、**accumulation buffer** と **velocity buffer** を使用します。accumulation buffer は、過去のフレームの画像を蓄積するためのバッファです。velocity buffer は、オブジェクトの移動速度を蓄積するためのバッファです。

運動ぼかしを実装するために、accumulation buffer と velocity buffer を使用します。accumulation buffer は、過去のフレームの画像を蓄積するためのバッファです。velocity buffer は、オブジェクトの移動速度を蓄積するためのバッファです。

運動ぼかしを実装するために、accumulation buffer と velocity buffer を使用します。

1. Unity 5.2 の Scene_12_6 を開きます。Window → Lighting → Skybox を開きます。

2. 3D ビューポートで、オブジェクトの移動速度を蓄積するための velocity buffer を作成します。9.5 の Translating.cs を参照してください。

3. MotionBlur.cs を作成します。

4. Unity Shader を作成します。Shader を Chapter12-MotionBlur と命名します。

MotionBlur.cs を作成します。

1. 12.1 のイメージを参考に、運動ぼかしを実装します。

```
public class MotionBlur : PostEffectsBase {
```

2. Shader を作成します。

```
public Shader motionBlurShader;  
private Material motionBlurMaterial = null;
```

```

public Material material {
    get {
        motionBlurMaterial = CheckShaderAndCreateMaterial(motionBlurShader, motionBlurMaterial);
        return motionBlurMaterial;
    }
}

```

3. 添加模糊效果

```

[Range(0.0f, 0.9f)]
public float blurAmount = 0.5f;

```

blurAmount 属性，范围从 0.0 到 0.9

4. 添加 RenderTexture 属性

```

private RenderTexture accumulationTexture;

void OnDisable() {
    DestroyImmediate(accumulationTexture);
}

```

OnDisable 方法中，销毁 accumulation Texture。

5. 添加 OnRenderImage 方法

```

void OnRenderImage (RenderTexture src, RenderTexture dest) {
    if (material != null) {
        // Create the accumulation texture
        if (accumulationTexture == null || accumulationTexture.width != src.width ||
            accumulationTexture.height != src.height) {
            DestroyImmediate(accumulationTexture);
            accumulationTexture = new RenderTexture(src.width, src.height, 0);
            accumulationTexture.hideFlags = HideFlags.HideAndDontSave;
            Graphics.Blit(src, accumulationTexture);
        }

        // We are accumulating motion over frames without clear/discard
        // by design, so silence any performance warnings from Unity
        accumulationTexture.MarkRestoreExpected();

        material.SetFloat("_BlurAmount", 1.0f - blurAmount);

        Graphics.Blit (src, accumulationTexture, material);
        Graphics.Blit (accumulationTexture, dest);
    } else {
        Graphics.Blit(src, dest);
    }
}

```

accumulationTexture 属性，用于存储模糊后的图像。
 accumulationTexture.hideFlags 属性，设置为 HideFlags.HideAndDontSave。

accumulation Texture Graphics.Blit(src, accumulationTexture)

accumulationTexture.accumulationTexture.MarkRestoreExpected **restore operation** OnRenderImage accumulationTexture accumulationTexture Graphics.Blit (src, accumulationTexture, material) src accumulationTexture Graphics.Blit (accumulationTexture, dest)

ShaderChapter12-MotionBlur

1

```
Properties {
    _MainTex ("Base (RGB)", 2D) = "white" {}
    _BlurAmount ("Blur Amount", Float) = 1.0
}
```

_MainTex_BlurAmount

2CGINCLUDESubShaderCGINCLUDEENDCG

```
SubShader {
    CGINCLUDE
    ...
    ENDCG
    ...
}
```

3

```
sampler2D _MainTex;
fixed _BlurAmount;
```

4

```
struct v2f {
    float4 pos : SV_POSITION;
    half2 uv : TEXCOORD0;
};

v2f vert(appdata_img v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.uv = v.texcoord;

    return o;
}
```

5RGB A

```
fixed4 fragRGB (v2f i) : SV_Target {
    return fixed4(tex2D(_MainTex, i.uv).rgb, _BlurAmount);
}

half4 fragA (v2f i) : SV_Target {
    return tex2D(_MainTex, i.uv);
}
```

RGB Shader A _BlurAmount A
A
6 Pass Pass RGB A
A RGB RGB A A

```
ZTest Always Cull Off ZWrite Off

Pass {
    Blend SrcAlpha OneMinusSrcAlpha
    ColorMask RGB

    CGPROGRAM

    #pragma vertex vert
    #pragma fragment fragRGB

    ENDCG
}

Pass {
    Blend One Zero
    ColorMask A

    CGPROGRAM

    #pragma vertex vert
    #pragma fragment fragA

    ENDCG
}
```

7 Shader Fallback

Fallback Off

Chapter12-MotionBlur MotionBlur.cs motionBlurShader
MotionBlur.cs motionBlurShader Chapter12-MotionBlur

13

12.7 参考

Unityのコンパイルオプションに関する情報は、UnityのImage Effectの<http://docs.unity3d.com/Manual/comp-ImageEffects.html>で確認できます。GPU Gemsの<https://developer.nvidia.com/gpugems/GPUGems>も、GPU Gems 3の27章で、Unityのコンパイルオプションに関する情報が提供されています。

13 環境保護

12 環境保護
環境保護是企業社會責任的重要組成部分。本公司一直致力於減少生產過程中的環境污染，並積極參與社會公益活動。我們通過採用先進的生產技術和設備，不斷提高資源利用效率，減少廢棄物的排放。同時，我們還加強了員工的環境保護意識培訓，確保每一位員工都能為環境保護貢獻力量。此外，我們還積極參與各種環保公益活動，如植樹造林、環保宣傳等，為改善環境質量做出貢獻。

環境保護是企業社會責任的重要組成部分。Unity 環境保護
13.1 環境保護
13.2 環境保護
13.3 環境保護
13.4 環境保護

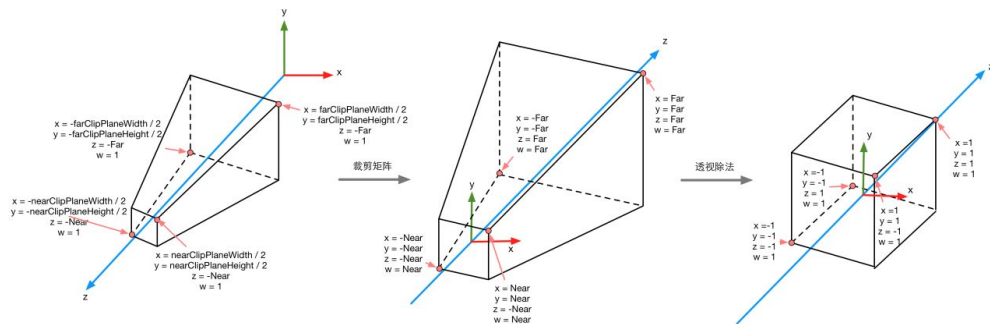
13.1 坐标变换

Unity 坐标变换

13.1.1 坐标变换

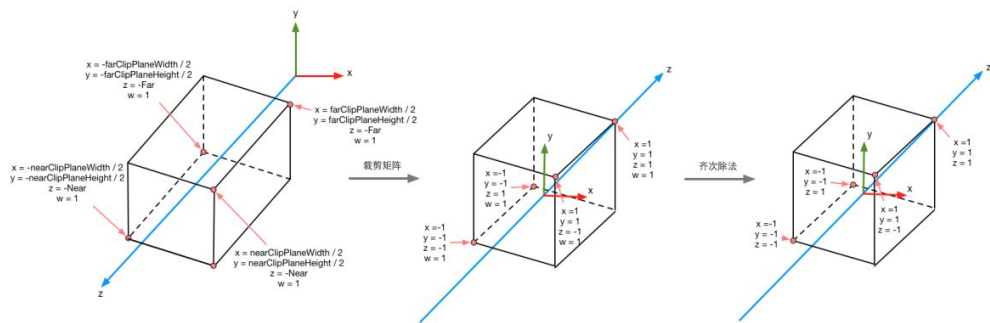
坐标变换是将物体在 3D 空间中的位置、旋转、缩放等信息，通过一系列数学运算，转换为在 2D 屏幕上的位置。这个过程通常分为几个步骤：首先，将物体在 3D 空间中的位置（ x, y, z ）转换为在 4D 空间中的位置（ x, y, z, w ），这个过程称为“齐次坐标变换”。然后，将 4D 坐标转换为 3D 坐标，这个过程称为“投影变换”。最后，将 3D 坐标转换为 2D 坐标，这个过程称为“裁剪变换”。在 Unity 中，这些变换通常通过“Normalized Device Coordinates”（NDC）来实现。NDC 是一个标准化的 3D 坐标系统，其范围通常为 $[-1, 1]$ 。在 Unity 中，NDC 的 z 轴范围通常为 $[0, 1]$ ，而在 DirectX 中， z 轴范围通常为 $[-1, 1]$ 。在 Unity 4.6.7 版本中，NDC 的 z 轴范围被调整为 $[0, 1]$ ，以与 DirectX 保持一致。

在 Unity 4.6.7 版本中，坐标变换的过程如下：首先，将物体在 3D 空间中的位置（ x, y, z ）转换为在 4D 空间中的位置（ x, y, z, w ），这个过程称为“齐次坐标变换”。然后，将 4D 坐标转换为 3D 坐标，这个过程称为“投影变换”。最后，将 3D 坐标转换为 2D 坐标，这个过程称为“裁剪变换”。在 Unity 中，这些变换通常通过“Normalized Device Coordinates”（NDC）来实现。NDC 是一个标准化的 3D 坐标系统，其范围通常为 $[-1, 1]$ 。在 Unity 中，NDC 的 z 轴范围通常为 $[0, 1]$ ，而在 DirectX 中， z 轴范围通常为 $[-1, 1]$ 。在 Unity 4.6.7 版本中，NDC 的 z 轴范围被调整为 $[0, 1]$ ，以与 DirectX 保持一致。



▲ 13.1 坐标变换

13.2 坐标变换



▲13.2 深度纹理的生成和读取

在NDC空间中，深度值 z 的范围是 $[-1, 1]$ ，而深度纹理的存储范围是 $[0, 1]$ 。

$$d = 0.5 \cdot z_{ndc} + 0.5$$

其中 d 是深度值， z_{ndc} 是NDC空间中的深度值。

在Unity中，可以通过`Shader Replacement`来生成深度纹理。在`SubShader`中，将`RenderType`设置为`Opaque`，并添加`AlphaTest`。在`Pass`中，使用`Shader`标签来指定要使用的着色器。在`RenderType`中，将`RenderType`设置为`Opaque`。

在Unity中，可以通过`Shader Replacement`来生成深度纹理。在`SubShader`中，将`RenderType`设置为`Opaque`，并添加`AlphaTest`。在`Pass`中，使用`Shader`标签来指定要使用的着色器。在`RenderType`中，将`RenderType`设置为`Opaque`。

13.1.2 深度纹理

在Unity中，可以通过`Shader Replacement`来生成深度纹理。在`SubShader`中，将`RenderType`设置为`Opaque`，并添加`AlphaTest`。在`Pass`中，使用`Shader`标签来指定要使用的着色器。在`RenderType`中，将`RenderType`设置为`Opaque`。

```
camera.depthTextureMode = DepthTextureMode.Depth;
```

在Unity中，可以通过`Shader Replacement`来生成深度纹理。在`SubShader`中，将`RenderType`设置为`Opaque`，并添加`AlphaTest`。在`Pass`中，使用`Shader`标签来指定要使用的着色器。在`RenderType`中，将`RenderType`设置为`Opaque`。

在Unity中，可以通过`Shader Replacement`来生成深度纹理。在`SubShader`中，将`RenderType`设置为`Opaque`，并添加`AlphaTest`。在`Pass`中，使用`Shader`标签来指定要使用的着色器。在`RenderType`中，将`RenderType`设置为`Opaque`。

```
camera.depthTextureMode = DepthTextureMode.DepthNormals;
```

在Unity中，可以通过`Shader Replacement`来生成深度纹理。在`SubShader`中，将`RenderType`设置为`Opaque`，并添加`AlphaTest`。在`Pass`中，使用`Shader`标签来指定要使用的着色器。在`RenderType`中，将`RenderType`设置为`Opaque`。

深度纹理的生成和读取+深度纹理

```
camera.depthTextureMode |= DepthTextureMode.Depth;  
camera.depthTextureMode |= DepthTextureMode.DepthNormals;
```

在Unity 5中，我们可以在Camera的RenderTexture属性中勾选生成深度纹理。在Shader中，我们使用_CameraDepthTexture来访问深度纹理。在PS3和PSP2上，我们使用Unity提供的SAMPLE_DEPTH_TEXTURE宏来访问深度纹理。在Shader中，我们使用SAMPLE_DEPTH_TEXTURE来访问深度纹理。

```
float d = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, i.uv);
```

在i.uv中，我们使用float2来访问深度纹理。在SAMPLE_DEPTH_TEXTURE_PROJ宏中，我们使用SAMPLE_DEPTH_TEXTURE_LOD宏来访问深度纹理。在SAMPLE_DEPTH_TEXTURE_PROJ宏中，我们使用float3和float4来访问深度纹理。在tex2Dproj宏中，我们使用float3和float4来访问深度纹理。在SAMPLE_DEPTH_TEXTURE_PROJ宏中，我们使用float3和float4来访问深度纹理。

```
float d = SAMPLE_DEPTH_TEXTURE_PROJ(_CameraDepthTexture, UNITY_PROJ_COORD(i.scrPos));
```

在i.scrPos中，我们使用ComputeScreenPos(o.pos)来访问深度纹理。在Unity中，我们使用HLSLSupport.cginc来访问深度纹理。

在Unity 5中，我们可以在Camera的RenderTexture属性中勾选生成深度纹理。在Shader中，我们使用_CameraDepthTexture来访问深度纹理。在PS3和PSP2上，我们使用Unity提供的SAMPLE_DEPTH_TEXTURE宏来访问深度纹理。在Shader中，我们使用SAMPLE_DEPTH_TEXTURE来访问深度纹理。

4.6.7 深度纹理的生成和读取 P_{clip} 深度纹理的生成和读取 z w

$$z_{clip} = -z_{view} \frac{Far + Near}{Far - Near} - \frac{2 \cdot Near \cdot Far}{Far - Near}$$

$$w_{clip} = -z_{view}$$

在Far和Near之间，我们使用NDC的z值。

$$z_{ndc} = \frac{z_{clip}}{w_{clip}} = \frac{Far + Near}{Far - Near} + \frac{2 \cdot Near \cdot Far}{(Far - Near) \cdot z_{view}}$$

13.1.1 从 NDC 到深度平面

$$d = 0.5 \cdot z_{ndc} + 0.5$$

深度平面上的深度 d 与 z_{view} 的关系

$$z_{view} = \frac{1}{\frac{Far - Near}{Near \cdot Far} d - \frac{1}{Near}}$$

Unity 中 z 轴的范围是 $[0, 1]$ ，而 z_{view} 的范围是 $[-Near, Far]$ 。

$$z'_{view} = \frac{1}{\frac{Near - Far}{Near \cdot Far} d + \frac{1}{Near}}$$

将 z_{view} 映射到 $[0, 1]$ 的范围，即 z_{01} 。

$$z_{01} = \frac{1}{\frac{Near - Far}{Near} d + \frac{Far}{Near}}$$

Unity 中的 `LinearEyeDepth` 函数用于将 `Linear01Depth` 转换为 `LinearEyeDepth`。

Unity 中的 `DecodeDepthNormal` 函数用于从深度和法线纹理中解码深度和法线。

```
inline void DecodeDepthNormal( float4 enc, out float depth, out float3 normal )
{
    depth = DecodeFloatRG (enc.zw);
    normal = DecodeViewNormalStereo (enc);
}
```

```
}
```

DecodeDepthNormal 0.5 + 0.5 * DecodeDepthNormal
0.5 * xy 0.5 * zw DecodeDepthNormal
[0, 1] DecodeFloatRG
DecodeViewNormalStereo 0.5 + 0.5 * DecodeDepthNormal

Unity 5.0.0.0

13.1.3 0.5 * 0.5

Shader Unity 5.0.0.0
Frame Debugger 13.3
+



13.3 Frame Debugger 0.5 * 0.5

Unity 5.0.0.0
13.4

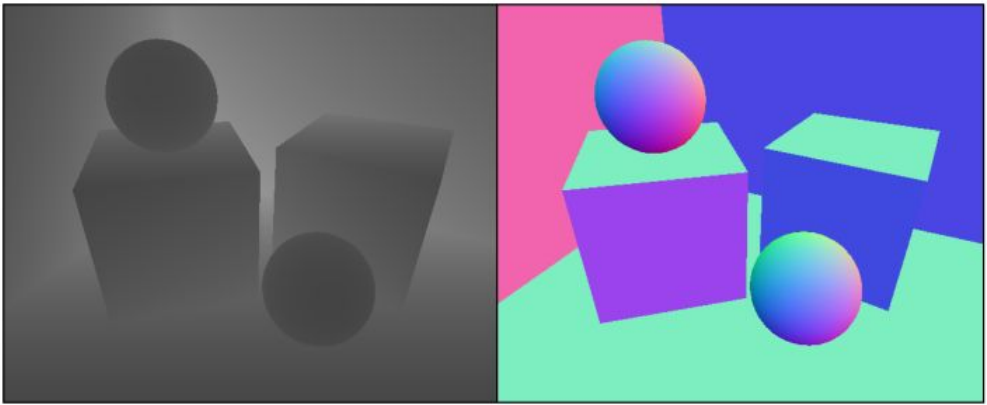
```
float depth = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, i.uv);  
float linearDepth = Linear01Depth(depth);  
return fixed4(linearDepth, linearDepth, linearDepth, 1.0);
```

0.5 * 0.5

```
fixed3 normal = DecodeViewNormalStereo(tex2D(_CameraDepthNormalsTexture, i.uv).xy);  
return fixed4(normal * 0.5 + 0.5, 1.0);
```

Unity 1.0.0.0

13.4



▲13.4 0, 1

13.2 运动模糊

12.6 运动模糊的实现原理。运动模糊的实现原理是，在每一帧渲染时，将当前帧的渲染结果与前一帧的渲染结果进行混合，从而实现运动模糊效果。Shader 是实现运动模糊的关键。

GPU Gems3 第 27 章 http://http.developer.nvidia.com/GPUGems3/gpugems3_ch27.html 详细介绍了运动模糊的实现原理。* 运动模糊的实现原理是，在每一帧渲染时，将当前帧的渲染结果与前一帧的渲染结果进行混合，从而实现运动模糊效果。* 运动模糊的实现原理是，在每一帧渲染时，将当前帧的渲染结果与前一帧的渲染结果进行混合，从而实现运动模糊效果。

运动模糊的实现原理

1 运动模糊的实现原理。Scene_13_2 Unity 5.2 运动模糊的实现原理。Window → Lighting → Skybox 运动模糊的实现原理。

2 运动模糊的实现原理。3 运动模糊的实现原理。4 运动模糊的实现原理。9.5 运动模糊的实现原理。Translating.cs 运动模糊的实现原理。

3 运动模糊的实现原理。MotionBlurWithDepthTexture.cs 运动模糊的实现原理。

4 运动模糊的实现原理。Unity Shader 运动模糊的实现原理。Chapter13-MotionBlurWithDepthTexture

运动模糊的实现原理。MotionBlurWithDepthTexture.cs 运动模糊的实现原理。

1 运动模糊的实现原理。12.1 运动模糊的实现原理。

```
public class MotionBlurWithDepthTexture : PostEffectsBase {
```

2 运动模糊的实现原理。Shader 运动模糊的实现原理。

```
public Shader motionBlurShader;
private Material motionBlurMaterial = null;

public Material material {
    get {
        motionBlurMaterial = CheckShaderAndCreateMaterial(motionBlurShader, motionBlurMaterial);
        return motionBlurMaterial;
    }
}
```

3 运动模糊的实现原理。

```
[Range(0.0f, 1.0f)]
public float blurSize = 0.5f;
```

4 运动模糊的实现原理。Camera 运动模糊的实现原理。

```
private Camera myCamera;
public Camera camera {
    get {
        if (myCamera == null) {
            myCamera = GetComponent<Camera>();
        }
    }
}
```

```

        return myCamera;
    }
}

```

5. 在 OnRenderImage 方法中，将当前的投影矩阵与之前的投影矩阵相乘。

```
private Matrix4x4 previousViewProjectionMatrix;
```

6. 在 OnEnable 方法中，将 DepthTextureMode 设置为 Depth。

```

void OnEnable() {
    camera.depthTextureMode |= DepthTextureMode.Depth;
}

```

7. 在 OnRenderImage 方法中，将当前的投影矩阵与之前的投影矩阵相乘。

```

void OnRenderImage (RenderTexture src, RenderTexture dest) {
    if (material != null) {
        material.SetFloat("_BlurSize", blurSize);

        material.SetMatrix("_PreviousViewProjectionMatrix", previousViewProjectionMatrix);
        Matrix4x4 currentViewProjectionMatrix = camera.projectionMatrix * camera.worldToCameraMatrix;
        Matrix4x4 currentViewProjectionInverseMatrix = currentViewProjectionMatrix.inverse;
        material.SetMatrix("_CurrentViewProjectionInverseMatrix", currentViewProjectionInverseMatrix);
        previousViewProjectionMatrix = currentViewProjectionMatrix;

        Graphics.Blit (src, dest, material);
    } else {
        Graphics.Blit(src, dest);
    }
}

```

8. 在 OnRenderImage 方法中，将当前的投影矩阵与之前的投影矩阵相乘。—— 将当前的投影矩阵与之前的投影矩阵相乘，然后将结果与当前的投影矩阵相乘。将当前的投影矩阵与之前的投影矩阵相乘，然后将结果与当前的投影矩阵相乘。将当前的投影矩阵与之前的投影矩阵相乘，然后将结果与当前的投影矩阵相乘。

9. 在 Shader 中，将 Chapter13-MotionBlurWithDepthTexture 设置为 BlurSize。

10. 在 Shader 中，将 Chapter13-MotionBlurWithDepthTexture 设置为 BlurSize。

```

Properties {
    _MainTex ("Base (RGB)", 2D) = "white" {}
    _BlurSize ("Blur Size", Float) = 1.0
}

```

11. 在 Shader 中，将 Chapter13-MotionBlurWithDepthTexture 设置为 BlurSize。将当前的投影矩阵与之前的投影矩阵相乘，然后将结果与当前的投影矩阵相乘。将当前的投影矩阵与之前的投影矩阵相乘，然后将结果与当前的投影矩阵相乘。将当前的投影矩阵与之前的投影矩阵相乘，然后将结果与当前的投影矩阵相乘。

12. 在 Shader 中，将 Chapter13-MotionBlurWithDepthTexture 设置为 BlurSize。将当前的投影矩阵与之前的投影矩阵相乘，然后将结果与当前的投影矩阵相乘。将当前的投影矩阵与之前的投影矩阵相乘，然后将结果与当前的投影矩阵相乘。将当前的投影矩阵与之前的投影矩阵相乘，然后将结果与当前的投影矩阵相乘。

```

SubShader {
    CGINCLUDE
    ...
    ENDCG
}

```

...

3. Shader Properties

```
sampler2D _MainTex;  
half4 _MainTex_TexelSize;  
sampler2D _CameraDepthTexture;  
float4x4 _CurrentViewProjectionInverseMatrix;  
float4x4 _PreviousViewProjectionMatrix;  
half _BlurSize;
```

Shader Properties: `_MainTex`, `_BlurSize`, `_CameraDepthTexture`, `Unity_
_CurrentViewProjectionInverseMatrix`, `_PreviousViewProjectionMatrix`, `_MainTex_TexelSize`.
Version: 5.6.1

4. Vertex Shader

```
struct v2f {  
    float4 pos : SV_POSITION;  
    half2 uv : TEXCOORD0;  
    half2 uv_depth : TEXCOORD1;  
};  
  
v2f vert(appdata_img v) {  
    v2f o;  
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);  
  
    o.uv = v.texcoord;  
    o.uv_depth = v.texcoord;  
  
    #if UNITY_UV_STARTS_AT_TOP  
    if (_MainTex_TexelSize.y < 0)  
        o.uv_depth.y = 1 - o.uv_depth.y;  
    #endif  
  
    return o;  
}
```

Shader Properties: `DirectX`, `DirectX`.

5. Fragment Shader

```
fixed4 frag(v2f i) : SV_Target {  
    // Get the depth buffer value at this pixel.  
    float d = SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, i.uv_depth);  
    // H is the viewport position at this pixel in the range -1 to 1.  
    float4 H = float4(i.uv.x * 2 - 1, i.uv.y * 2 - 1, d * 2 - 1, 1);  
    // Transform by the view-projection inverse.  
    float4 D = mul(_CurrentViewProjectionInverseMatrix, H);  
    // Divide by w to get the world position.  
    float4 worldPos = D / D.w;  
  
    // Current viewport position  
    float4 currentPos = H;  
    // Use the world position, and transform by the previous view-projection matrix.  
    float4 previousPos = mul(_PreviousViewProjectionMatrix, worldPos);  
    // Convert to nonhomogeneous points [-1,1] by dividing by w.  
    previousPos /= previousPos.w;  
  
    // Use this frame's position and last frame's to compute the pixel velocity.  
    float2 velocity = (currentPos.xy - previousPos.xy)/2.0f;
```



```

float2 uv = i.uv;
float4 c = tex2D(_MainTex, uv);
uv += velocity * _BlurSize;
for (int it = 1; it < 3; it++, uv += velocity * _BlurSize) {
    float4 currentColor = tex2D(_MainTex, uv);
    c += currentColor;
}
c /= 3;

return fixed4(c.rgb, 1.0);
}

```

*
 SAMPLE_DEPTH_TEXTURE d 13.1.2 d NDC H NDC $d * 2 - 1$ NDC xy NDC xyz $[-1, 1]$ NDC H * w worldPos

* NDC previousPos velocity

_BlurSize

6 Pass

```

Pass {
    ZTest Always Cull Off ZWrite Off

    CGPROGRAM

    #pragma vertex vert
    #pragma fragment frag

    ENDCG
}

```

7 shader Fallback

Fallback Off

Chapter13-MotionBlurWithDepthTexture MotionBlur
 WithDepthTexture.cs motionBlurShader MotionBlurWith DepthTexture.cs
 motionBlurShader Chapter13-MotionBlur WithDepthTexture

Unity
 ImageEffect

13.3

13.3 □□□□

```

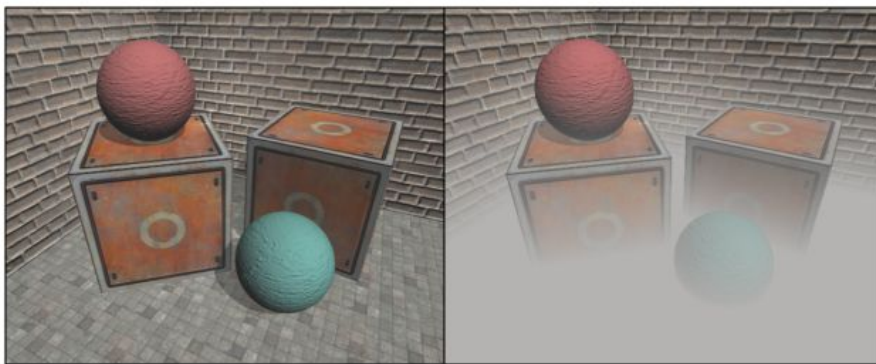
    Fog Unity
    Shader #pragma multi_compile_fog
    UNITY_FOG_COORDS UNITY_TRANSFER_FOG UNITY_APPLY_FOG
    Unity

```

```

Shader
/
13.5

```

[illegible]

13.2 NDC*

13.3.1

[illegible]

```
float4 worldPos = _WorldSpaceCameraPos + linearDepth * interpolatedRay;
```

```

    _WorldSpaceCameraPos * interpolatedRay * linearDepth * linearDepth *
    interpolatedRay * linearDepth *
    13.1.2 * interpolatedRay

```

```
interpolatedRay[0][0] = 4; interpolatedRay[0][1] = 4; interpolatedRay[0][2] = 0;
interpolatedRay[1][0] = FOV[0][0] * 13.6; interpolatedRay[1][1] = 0; interpolatedRay[1][2] = 1;
toRight[0][0] = 0; toRight[0][1] = 0; toRight[0][2] = 0;
```

```

    toTop = camera.up × halfHeight
    toRight = camera.right × halfHeight · aspect
    NearFOV camera.up camera.right

```

4 13.6 TL

3

$$BL = camera.forward \cdot Near - toTop - toRight$$

$$BR = camera.forward \cdot Near - toTop + toRight$$

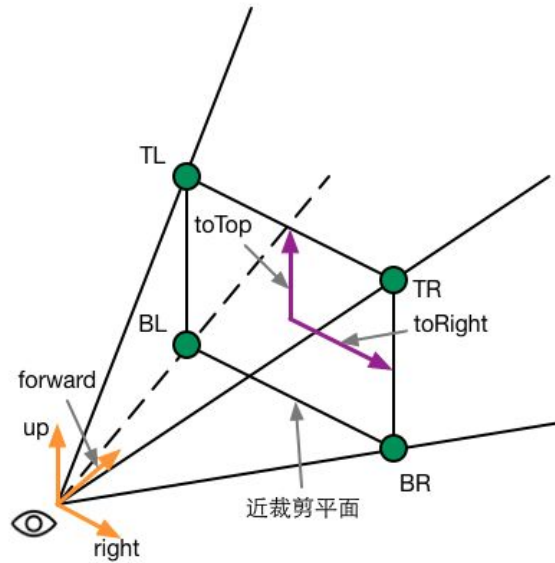
0000000040000000000000000000400000000000000000000000000
0000z 00000000000000000000004000000000000000000000013.7000000000
0000000000000000TL000000000000TL00000000000000000000000000
0TL00000000

TL	dist
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0
18	0
19	0
20	0
21	0
22	0
23	0
24	0
25	0
26	0
27	0
28	0
29	0
30	0
31	0
32	0
33	0
34	0
35	0
36	0
37	0
38	0
39	0
40	0
41	0
42	0
43	0
44	0
45	0
46	0
47	0
48	0
49	0
50	0
51	0
52	0
53	0
54	0
55	0
56	0
57	0
58	0
59	0
60	0
61	0
62	0
63	0
64	0
65	0
66	0
67	0
68	0
69	0
70	0
71	0
72	0
73	0
74	0
75	0
76	0
77	0
78	0
79	0
80	0
81	0
82	0
83	0
84	0
85	0
86	0
87	0
88	0
89	0
90	0
91	0
92	0
93	0
94	0
95	0
96	0
97	0
98	0
99	0

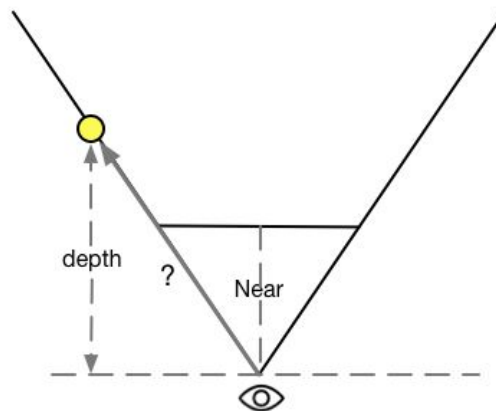
0040000000000000300000TL0000000000000000000000000000

$$Ray_{TL} = \frac{TL}{|TL|} \times scale, Ray_{TR} = \frac{TR}{|TR|} \times scale$$

$$Ray_{BL} = \frac{BL}{|BL|} \times scale, Ray_{BR} = \frac{BR}{|BR|} \times scale$$



▲13.6 插值射线 (interpolatedRay)



▲13.7 深度插值 (depth interpolation)

深度插值 (depth interpolation) 是渲染过程中非常重要的一步。它用于计算每个像素的深度值，以便进行正确的遮挡剔除。在 Unity 中，我们使用 `interpolatedRay` 来计算深度值。

13.3.2 雾效 (Fog)

雾效 (Fog) 是一种常用的视觉效果，用于模拟大气中的雾。在 Unity 中，我们使用 `f` 来计算雾效。

```
float3 afterFog = f * fogColor + (1 - f) * origColor;
```

在 Unity 中，雾效的计算方式有多种，包括 Linear、Exponential、Exponential Squared 等。其中，Linear 是最常用的方式。它通过计算深度 z 和雾效因子 f 来计算雾效。

Linear

$$f = \frac{d_{max} - |z|}{d_{max} - d_{min}} \quad d_{min} \leq d_{max} \quad \text{Exponential}$$

Exponential

$$f = [e^{-d \cdot |z|}] \quad d \quad \text{Exponential Squared}$$

Exponential Squared

$$f = e^{-(d \cdot |z|)^2} \quad d \quad \text{Exponential Squared}$$

Exponential Squared $y = f$

$$f = \frac{H_{end} - y}{H_{end} - H_{start}} \quad H_{start} \leq H_{end} \quad \text{Exponential Squared}$$

13.3.3

Unity

1 Scene_13_3 Unity 5.2 Window -< Lighting -< Skybox

2 3 9.5 Translating.cs

3 FogWithDepthTexture.cs

4 Unity Shader Shader Chapter13-FogWithDepthTexture

FogWithDepthTexture.cs

1 12.1

```
public class FogWithDepthTexture : PostEffectsBase {
```

2 Shader

```
public Shader fogShader;
private Material fogMaterial = null;

public Material material {
    get {
        fogMaterial = CheckShaderAndCreateMaterial(fogShader, fogMaterial);
        return fogMaterial;
    }
}
```

3 FOV Camera Transform

```
private Camera myCamera;
public Camera camera {
```


3. 変数宣言

```
float4x4 _FrustumCornersRay;

sampler2D _MainTex;
half4 _MainTex_TexelSize;
sampler2D _CameraDepthTexture;
half _FogDensity;
fixed4 _FogColor;
float _FogStart;
float _FogEnd;
```

`_FrustumCornersRay` は `Properties` の `Shader Properties` の `_CameraDepthTexture` の `Unity` の `(0, 0)` から `(1, 1)` までの範囲で

4. 変数宣言

```
struct v2f {
    float4 pos : SV_POSITION;
    half2 uv : TEXCOORD0;
    half2 uv_depth : TEXCOORD1;
    float4 interpolatedRay : TEXCOORD2;
};

v2f vert(appdata_img v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.uv = v.texcoord;
    o.uv_depth = v.texcoord;

    #if UNITY_UV_STARTS_AT_TOP
    if (_MainTex_TexelSize.y < 0)
        o.uv_depth.y = 1 - o.uv_depth.y;
    #endif

    int index = 0;
    if (v.texcoord.x < 0.5 && v.texcoord.y < 0.5) {
        index = 0;
    } else if (v.texcoord.x > 0.5 && v.texcoord.y < 0.5) {
        index = 1;
    } else if (v.texcoord.x > 0.5 && v.texcoord.y > 0.5) {
        index = 2;
    } else {
        index = 3;
    }

    #if UNITY_UV_STARTS_AT_TOP
    if (_MainTex_TexelSize.y < 0)
        index = 3 - index;
    #endif

    o.interpolatedRay = _FrustumCornersRay[index];

    return o;
}
```

`v2f` の `interpolatedRay` は `4` の `Unity` の `(0, 0)` から `(1, 1)` までの範囲で

frustumCorners은 DirectX Metal에서 (0, 0)에서 Unity에서 DirectX에서 Unity에서 5.6.1에서 _FrustumCornersRay interpolatedRay

4

5

```
fixed4 frag(v2f i) : SV_Target {
    float linearDepth = LinearEyeDepth(SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, i. uv_depth));
    float3 worldPos = _WorldSpaceCameraPos + linearDepth * i.interpolatedRay.xyz;

    float fogDensity = (_FogEnd - worldPos.y) / (_FogEnd - _FogStart);
    fogDensity = saturate(fogDensity * _FogDensity);

    fixed4 finalColor = tex2D(_MainTex, i.uv);
    finalColor.rgb = lerp(finalColor.rgb, _FogColor.rgb, fogDensity);

    return finalColor;
}
```

SAMPLE_DEPTH_TEXTURE LinearEyeDepth interpolatedRay

13.3.2 _FogEnd _FogStart worldPos.y fogDensity _FogDensity saturate [0, 1]

6 Pass

```
Pass {
    ZTest Always Cull Off ZWrite Off

    CGPROGRAM

    #pragma vertex vert
    #pragma fragment frag

    ENDCG
}
```

7 Shader Fallback

Fallback Off

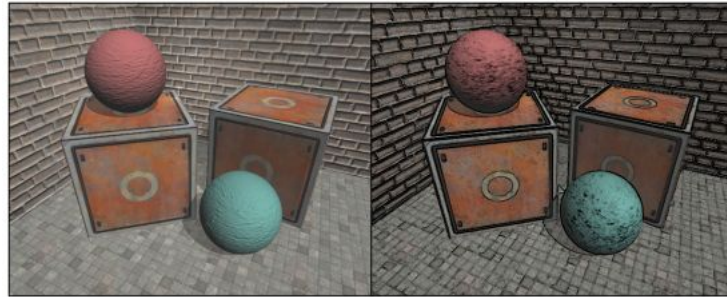
Chapter13-FogWithDepthTexture FogWithDepthTexture.cs fogShader FogWithDepthTexture.cs fogShader Chapter13-FogWithDepthTexture

<http://www.derschmale.com/2014/03/19/reconstructing-positions-from-the-depth-buffer-pt-2-perspective-and-orthographic-general-case/>

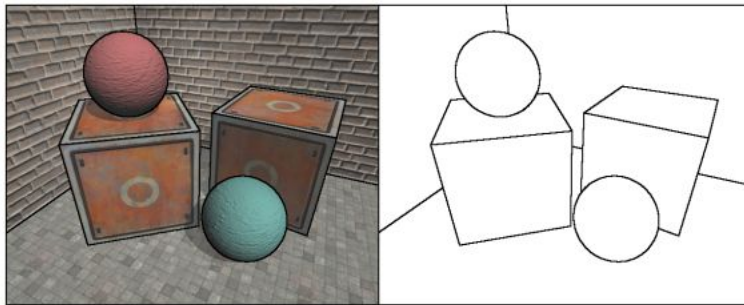
13.4 边缘检测

12.3 节我们学习了 Sobel 算子，它用于检测图像中的边缘。在本节中，我们将学习 Roberts 算子，它也是一种边缘检测算子。

Roberts 算子是一种简单的边缘检测算子，它通过计算图像中相邻像素的差值来检测边缘。在本节中，我们将学习 Roberts 算子的原理和实现。



▲ 13.8 边缘检测的结果



▲ 13.9 边缘检测的结果

12.3 节我们学习了 Sobel 算子，在本节中，我们将学习 Roberts 算子。

Roberts

-1	0	0	-1
0	1	1	0
G_x		G_y	

▲ 13.10 Roberts 算子

Roberts 算子是一种简单的边缘检测算子，它通过计算图像中相邻像素的差值来检测边缘。在本节中，我们将学习 Roberts 算子的原理和实现。

边缘检测的结果

1 节我们学习了 Scene_13_4 在 Unity 5.2 中的实现。在本节中，我们将学习 Roberts 算子的实现。

2. 3. 9.5. Translating.cs

3. EdgeDetectNormalsAndDepth.cs

4. Unity Shader Shader Chapter13-EdgeDetectNormalAndDepth

EdgeDetectNormalsAndDepth.cs 12.3 EdgeDetection.cs

1. 12.1

```
public class EdgeDetectNormalsAndDepth : PostEffectsBase {
```

2. Shader

```
public Shader edgeDetectShader;
private Material edgeDetectMaterial = null;
public Material material {
    get {
        edgeDetectMaterial = CheckShaderAndCreateMaterial(edgeDetectShader, edgeDetectMaterial);
        return edgeDetectMaterial;
    }
}
```

3.

```
[Range(0.0f, 1.0f)]
public float edgesOnly = 0.0f;

public Color edgeColor = Color.black;

public Color backgroundColor = Color.white;

public float sampleDistance = 1.0f;

public float sensitivityDepth = 1.0f;

public float sensitivityNormals = 1.0f;
```

sampleDistance + sampleDistance
sensitivityDepth sensitivityNormals

4. OnEnable

```
void OnEnable() {
    GetComponent<Camera>().depthTextureMode |= DepthTextureMode.DepthNormals;
}
```

5. OnRenderImage

```
[ImageEffectOpaque]
void OnRenderImage (RenderTexture src, RenderTexture dest) {
    if (material != null) {
        material.SetFloat("_EdgeOnly", edgesOnly);
        material.SetColor("_EdgeColor", edgeColor);
        material.SetColor("_BackgroundColor", backgroundColor);
        material.SetFloat("_SampleDistance", sampleDistance);
    }
}
```

```

        material.SetVector("_Sensitivity", new Vector4(sensitivityNormals, sensitivityDepth, 0.0f, 0.0f));
        Graphics.Blit(src, dest, material);
    } else {
        Graphics.Blit(src, dest);
    }
}

```

OnRenderImage[ImageEffectOpaque]12.1
 OnRenderImagePassPass2 500PassBackgroundGeometryAlphaTestTransparentPassOnRenderImageImageEffectOpaque

ShaderChapter13-EdgeDetectNormalAndDepth

1

```

Properties {
    _MainTex ("Base (RGB)", 2D) = "white" {}
    _EdgeOnly ("Edge Only", Float) = 1.0
    _EdgeColor ("Edge Color", Color) = (0, 0, 0, 1)
    _BackgroundColor ("Background Color", Color) = (1, 1, 1, 1)
    _SampleDistance ("Sample Distance", Float) = 1.0
    _Sensitivity ("Sensitivity", Vector) = (1, 1, 1, 1)
}

```

_Sensitivityxyzw

2CGINCLUDESubShaderCGINCLUDEENDCG

```

SubShader {
    CGINCLUDE
    ...
    ENDCG
    ...
}

```

3CG

```

sampler2D _MainTex;
half4 _MainTex_TexelSize;
fixed _EdgeOnly;
fixed4 _EdgeColor;
fixed4 _BackgroundColor;
float _SampleDistance;
half4 _Sensitivity;
sampler2D _CameraDepthNormalsTexture;

```

+_CameraDepthNormalsTexture
 _MainTex_TexelSize

4

```

struct v2f {
    float4 pos : SV_POSITION;
    half2 uv[5]: TEXCOORD0;
};

v2f vert(appdata_img v) {

```

```

v2f o;
o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

half2 uv = v.texcoord;
o.uv[0] = uv;

#if UNITY_UV_STARTS_AT_TOP
if (_MainTex_TexelSize.y < 0)
    uv.y = 1 - uv.y;
#endif

o.uv[1] = uv + _MainTex_TexelSize.xy * half2(1,1) * _SampleDistance;
o.uv[2] = uv + _MainTex_TexelSize.xy * half2(-1,-1) * _SampleDistance;
o.uv[3] = uv + _MainTex_TexelSize.xy * half2(-1,1) * _SampleDistance;
o.uv[4] = uv + _MainTex_TexelSize.xy * half2(1,-1) * _SampleDistance;

return o;
}

```

5个uv值，其中uv[0]是主uv，uv[1]~uv[4]是 Roberts 交叉采样点，每个点都乘以 SampleDistance。

5个采样点

```

fixed4 fragRobertsCrossDepthAndNormal(v2f i) : SV_Target {
    half4 sample1 = tex2D(_CameraDepthNormalsTexture, i.uv[1]);
    half4 sample2 = tex2D(_CameraDepthNormalsTexture, i.uv[2]);
    half4 sample3 = tex2D(_CameraDepthNormalsTexture, i.uv[3]);
    half4 sample4 = tex2D(_CameraDepthNormalsTexture, i.uv[4]);

    half edge = 1.0;

    edge *= CheckSame(sample1, sample2);
    edge *= CheckSame(sample3, sample4);

    fixed4 withEdgeColor = lerp(_EdgeColor, tex2D(_MainTex, i.uv[0]), edge);
    fixed4 onlyEdgeColor = lerp(_EdgeColor, _BackgroundColor, edge);

    return lerp(withEdgeColor, onlyEdgeColor, _EdgeOnly);
}

```

4个采样点 + CheckSame 函数，用于比较两个采样点的深度和法线是否相似。

```

half CheckSame(half4 center, half4 sample) {
    half2 centerNormal = center.xy;
    float centerDepth = DecodeFloatRG(center.zw);
    half2 sampleNormal = sample.xy;
    float sampleDepth = DecodeFloatRG(sample.zw);

    // difference in normals
    // do not bother decoding normals - there's no need here
    half2 diffNormal = abs(centerNormal - sampleNormal) * _Sensitivity.x;
    int isSameNormal = (diffNormal.x + diffNormal.y) < 0.1;
    // difference in depth
    float diffDepth = abs(centerDepth - sampleDepth) * _Sensitivity.y;
    // scale the required threshold by the distance
    int isSameDepth = diffDepth < 0.1 * centerDepth;

    // return:
    // 1 - if normals and depth are similar enough
    // 0 - otherwise
    return isSameNormal * isSameDepth ? 1.0 : 0.0;
}

```

[illegible]

CheckSame12.3

6 Pass

```
Pass {
    ZTest Always Cull Off ZWrite Off

    CGPROGRAM

    #pragma vertex vert
    #pragma fragment fragRobertsCrossDepthAndNormal

    ENDCG
}
```

7 Shader Fallback

Fallback Off

Chapter13-EdgeDetectNormalAndDepthEdgeDetect
NormalsAndDepth.csedgeDetectShaderEdgeDetectNormals
AndDepth.csedgeDetectShaderChapter13-EdgeDetectNormal
AndDepth

[illegible]

13.5 ☐☐☐☐

```
ShaderReplacementTag = new ShaderReplacementTag(shader, replacementTag);  
Camera.RenderWithShader(shader, replacementTag);  
Camera.RenderWithShader(shader, replacementTag);
```

Unity 2011 SIGGRAPH
http://blogs.unity3d.com/2011/09/08/special-effects-with-depth-talk-at-siggraph/
Unity Image Effect
http://docs.unity3d.com/Manual/com-ImageEffects.html
Screen Space Ambient Occlusion SSAO

14 图形学

图形学追求的目标是 photorealism 逼真度
Non-Photorealistic Rendering NPR 非逼真度
渲染技术

14.1 渲染技术
14.2 渲染技术

14.1 图形渲染

图形渲染是计算机图形学中的一个重要分支，它涉及到如何将存储在计算机中的图形数据（如顶点、纹理、颜色等）按照一定的规则，通过数学计算和算法，最终在屏幕上呈现出来。Capcom的Okami游戏就是一个典型的例子，它展示了图形渲染在游戏中的应用。

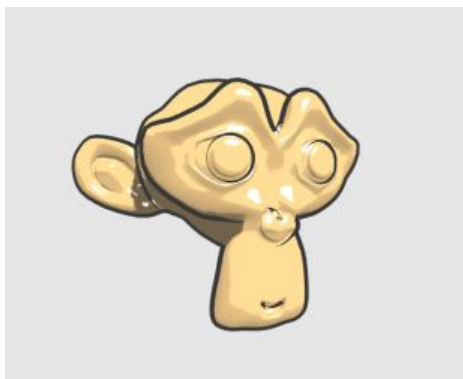
图形渲染技术可以分为多种类型，其中一种重要的技术是**tone-based shading**。Gooch在1998年提出了一种名为**tone-based shading**的渲染技术，它通过调整光源的颜色和强度，来实现对物体颜色的渲染。这种技术在Okami游戏中得到了广泛的应用，使得游戏中的场景呈现出一种独特的、柔和的色调。

图形渲染技术的发展，使得游戏中的场景越来越逼真。从早期的像素风格，到现在的3D渲染，图形渲染技术一直在不断进步。Okami游戏中的场景，就是一个典型的例子，它展示了图形渲染技术在游戏中的应用。

图形渲染技术的发展，使得游戏中的场景越来越逼真。Okami游戏中的场景，就是一个典型的例子，它展示了图形渲染技术在游戏中的应用。



▲14.1 图形渲染Okami游戏中的场景



▲14.2 图形渲染

14.1.1 图形渲染

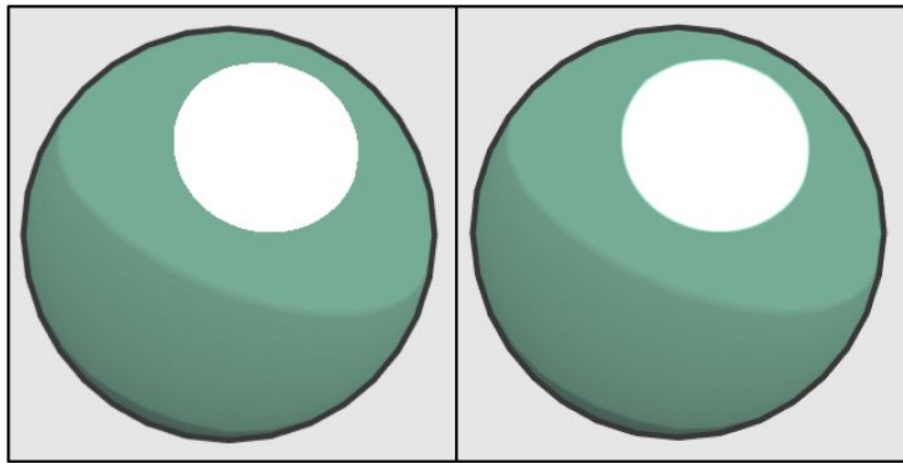
图形渲染技术的发展，使得游戏中的场景越来越逼真。Okami游戏中的场景，就是一个典型的例子，它展示了图形渲染技术在游戏中的应用。Real Time Rendering, third edition

CG **step** 関数を用いて、spec 値を 0 から 1 までの範囲に制限する。

図 14.3: smoothstep 関数を用いた、spec 値の制限。左側は spec 値が 0 から 1 までの範囲に制限された結果、右側は spec 値が 1 以上の範囲に制限された結果。

```
float spec = dot(worldNormal, worldHalfDir);
spec = lerp(0, 1, smoothstep(-w, w, spec - threshold));
```

smoothstep 関数を用いて、spec 値を 0 から 1 までの範囲に制限する。CG **smoothstep** 関数は、 w と $spec - threshold$ の値を $-w$ から w の範囲に制限し、その結果を 0 から 1 の範囲に制限する。つまり、 $spec$ が $spec - threshold + w$ より大きい場合、 $spec$ は 1 となる。CG **fwidth** 関数は、 w の 2 倍の値を返す。



▲図 14.3 smoothstep 関数を用いた、spec 値の制限

この章では、Anjyo [2] (2003) の論文「Toon Shading」を参考に、Unity 5.2 での実装を行う。[2] は <http://blog.csdn.net/candycat1992/article/details/47284289> に掲載されている。Unity での実装は、

14.1.3 実装

Unity での実装は、ToonShadingMat という名前で行う。

1. Unity での実装は、Scene_14_1 という Unity 5.2 のプロジェクトで行う。Window → Lighting → Skybox を ToonSkybox に設定する。

2. ToonShadingMat という名前で行う。

3. Unity Shader を作成する。Unity Shader の Chapter 14-ToonShading を参照する。Unity Shader の 2.1 節を参照する。

4. Suzanne というモデルを 2.1 節で参照する。

5

Chapter14-ToonShading

1

```
Properties {
    _Color ("Color Tint", Color) = (1, 1, 1, 1)
    _MainTex ("Main Tex", 2D) = "white" {}
    _Ramp ("Ramp Texture", 2D) = "white" {}
    _Outline ("Outline", Range(0, 1)) = 0.1
    _OutlineColor ("Outline Color", Color) = (0, 0, 0, 1)
    _Specular ("Specular", Color) = (1, 1, 1, 1)
    _SpecularScale ("Specular Scale", Range(0, 0.1)) = 0.01
}
```

_Ramp _Outline _OutlineColor
_Specular _SpecularScale

2 Pass Pass

```
Pass {
    NAME "OUTLINE"

    Cull Front
}
```

Cull NAME Pass
Pass Pass

3

```
v2f vert (a2v v) {
    v2f o;

    float4 pos = mul(UNITY_MATRIX_MV, v.vertex);
    float3 normal = mul((float3x3)UNITY_MATRIX_IT_MV, v.normal);
    normal.z = -0.5;
    pos = pos + float4(normalize(normal), 0) * _Outline;
    o.pos = mul(UNITY_MATRIX_P, pos);

    return o;
}

float4 frag(v2f i) : SV_Target {
    return float4(_OutlineColor.rgb, 1);
}
```

14.1.1
z
z

4 Pass Unity Pass

```
Pass {
    Tags { "LightMode"="ForwardBase" }

    Cull Back

    CGPROGRAM

    #pragma vertex vert
    #pragma fragment frag

    #pragma multi_compile_fwdbase
```

LightMode ForwardBase #pragma Shader

5

```
struct v2f {
    float4 pos : POSITION;
    float2 uv : TEXCOORD0;
    float3 worldNormal : TEXCOORD1;
    float3 worldPos : TEXCOORD2;
    SHADOW_COORDS(3)
};

v2f vert (a2v v) {
    v2f o;

    o.pos = mul( UNITY_MATRIX_MVP, v.vertex);
    o.uv = TRANSFORM_TEX (v.texcoord, _MainTex);
    o.worldNormal = mul(v.normal, (float3x3)_World2Object);
    o.worldPos = mul(_Object2World, v.vertex).xyz;

    TRANSFER_SHADOW(o);

    return o;
}
```

Unity SHADOW_COORDS TRANSFER_SHADOW 9.4

6

```
float4 frag(v2f i) : SV_Target {
    fixed3 worldNormal = normalize(i.worldNormal);
    fixed3 worldLightDir = normalize(UnityWorldSpaceLightDir(i.worldPos));
    fixed3 worldViewDir = normalize(UnityWorldSpaceViewDir(i.worldPos));
    fixed3 worldHalfDir = normalize(worldLightDir + worldViewDir);

    fixed4 c = tex2D (_MainTex, i.uv);
    fixed3 albedo = c.rgb * _Color.rgb;

    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;
```

```

UNITY_LIGHT_ATTENUATION(atten, i, i.worldPos);

fixed diff = dot(worldNormal, worldLightDir);
diff = (diff * 0.5 + 0.5) * atten;

fixed3 diffuse = _LightColor0.rgb * albedo * tex2D(_Ramp, float2(diff, diff)).rgb;

fixed spec = dot(worldNormal, worldHalfDir);
fixed w = fwidth(spec) * 2.0;
fixed3 specular = _Specular.rgb * lerp(0, 1, smoothstep(-w, w, spec + _SpecularScale
- 1)) * step(0.0001, _SpecularScale);

return fixed4(ambient + diffuse + specular, 1.0);
}

```

14.1.2 Shader Fallback

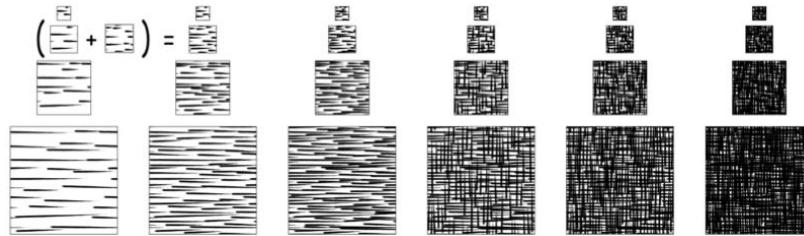
Fallback "Diffuse"

9.4

Team Fortress 2 Valve 2007 [3]

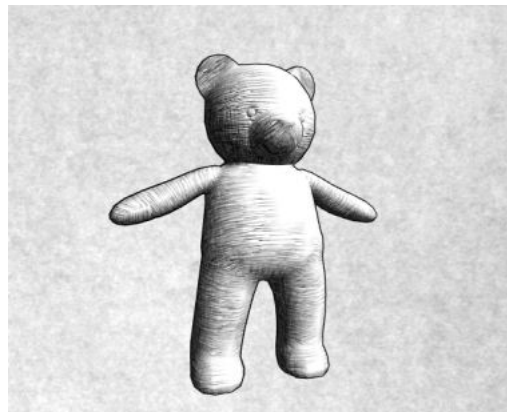
14.2 纹理混合

纹理混合技术最早是由Praun在2001年的SIGGRAPH会议上提出的[4]。该技术通过混合多个纹理来生成新的纹理。在14.4节中，我们将介绍一种名为**Tonal Art Map (TAM)**的纹理混合技术。在14.4节中，我们将介绍一种名为**mipmaps**的纹理混合技术。



▲14.4 TAM纹理混合 Praun E, et al. Real-time hatching[4]

在14.5节中，我们将介绍一种名为**6**的纹理混合技术。在14.5节中，我们将介绍一种名为**6**的纹理混合技术。在14.5节中，我们将介绍一种名为**6**的纹理混合技术。



▲14.5 纹理混合

纹理混合技术

1. Unity 纹理混合 Scene_14_2 Unity 5.2 纹理混合 Window -> Lighting -> Skybox

2. HatchingMat

3. Unity Shader Chapter14-Hatching Unity Shader

4. TeddyBear 2

5.

Chapter14-Hatching

1.


```
Properties {
  _Color ("Color Tint", Color) = (1, 1, 1, 1)
  _TileFactor ("Tile Factor", Float) = 1
  _Outline ("Outline", Range(0, 1)) = 0.1
  _Hatch0 ("Hatch 0", 2D) = "white" {}
  _Hatch1 ("Hatch 1", 2D) = "white" {}
  _Hatch2 ("Hatch 2", 2D) = "white" {}
  _Hatch3 ("Hatch 3", 2D) = "white" {}
  _Hatch4 ("Hatch 4", 2D) = "white" {}
  _Hatch5 ("Hatch 5", 2D) = "white" {}
}
```

```

    _Color_ _TileFactor_ _TileFactor_14.5
    _TileFactor_8_ _Hatch0_ _Hatch5_6

```

214.1Pass

```
SubShader {
    Tags { "RenderType"="Opaque" "Queue"="Geometry" }

    UsePass "Unity Shaders Book/Chapter 14/Toon Shading/OUTLINE"
```

```

    UsePass14.1PassUnity Shaders Book/Chapter 14/Toon Shading
    14.1Chapter14-ToonShadingShaderUnityPass
    UsePassPass

```

3 Pass Pass

```
Pass {
    Tags { "LightMode"="ForwardBase" }

    CGPROGRAM

    #pragma vertex vert
    #pragma fragment frag

    #pragma multi_compile_fwdbase
```

46v2f

```
struct v2f {
    float4 pos : SV_POSITION;
    float2 uv : TEXCOORD0;
    fixed3 hatchWeights0 : TEXCOORD1;
    fixed3 hatchWeights1 : TEXCOORD2;
    float3 worldPos : TEXCOORD3;
    SHADOW_COORDS(4)
};
```

```
00000060000000000000fixed30000hatchWeights0hatchWeights1
00000000000000worldPos0000SHADOW_COORDS0000000000
```

[illegible]

```
v2f vert(a2v v) {
    v2f o;

    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.uv = v.texcoord.xy * _TileFactor;
```


7Fallback

Fallback "Diffuse"

[4] <https://alastaira.wordpress.com/2013/11/01/hand-drawn-shaders-and-creating-tonal-art-maps/> Photoshop

14.3 卡通渲染

卡通渲染是介于真实渲染和2D渲染之间的一种渲染方式，它通过模拟真实世界的物理特性，如光照、阴影、反射等，来生成具有卡通风格的图像。NPAR (Non-Photorealistic Animation and Rendering) 是一种用于生成卡通渲染的算法，其原理基于对真实世界物理特性的模拟。The Algorithms and Principles of Non-photorealistic Graphics^[5] 是一本关于卡通渲染的书籍，详细介绍了相关的算法和原理。

Unity 提供了多种卡通渲染的 Shader，包括 **Toon Shader Free** (<https://www.assetstore.unity3d.com/cn/#!/content/21288>)、**Toon Styles Shader Pack** (<https://www.assetstore.unity3d.com/cn/#!/content/7212>) 和 **Hand-Drawn Shader Pack** (<https://www.assetstore.unity3d.com/cn/#!/content/12465>)。这些 Shader 可以在 Unity 引擎中使用，为游戏或应用提供高质量的卡通渲染效果。

14.4 □□□□

[1] Gooch A, Gooch B, Shirley P, et al. A non-photorealistic lighting model for automatic technical illustration[C]//Proceedings of the 25th annual conference on Computer graphics and interactive techniques. ACM, 1998: 447-452□

[2] Anjyo K, Hiramitsu K. Stylized highlights for cartoon rendering and animation[J]. Computer Graphics and Applications, IEEE, 2003, 23(4): 54-61□

[3] Mitchell J, Francke M, Eng D. Illustrative rendering in Team Fortress 2[C]//Proceedings of the 5th international symposium on Non-photorealistic animation and rendering. ACM, 2007: 71-76□

[4] Praun E, Hoppe H, Webb M, et al. Real-time hatching[C]//Proceedings of the 28th annual conference on Computer graphics and interactive techniques. ACM, 2001: 581□

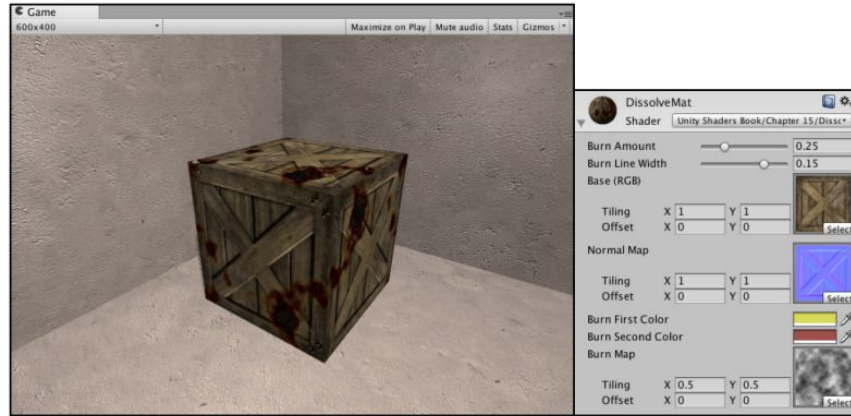
[5] Geng W. The Algorithms and Principles of Non-photorealistic Graphics: Artistic Rendering and Cartoon Animation[M]. Springer Science & Business Media, 2011□

15

2022年12月31日“资产”项目余额为“资产”项目
 2022年12月31日“负债”项目余额为“负债”项目15.1
 2022年12月31日“所有者权益”项目余额为“所有者权益”项目15.2
 2022年12月31日“收入”项目余额为“收入”项目15.3
 2022年12月31日“费用”项目余额为“费用”项目13.3

15.1 溶解

この章でdissolveという効果を実装します。UnityのShader BookのChapter 15.1を参考にします。



▲15.1 溶解

この章では、UnityのShader BookのChapter 15.1を参考に、dissolveという効果を実装します。この効果は、物体の表面を徐々に消滅させる効果です。この効果は、UnityのShader BookのChapter 15.1を参考に、pow関数とclip関数を使用して実装します。

この効果を実装するために、以下のような手順を行います。

1. UnityのScene_15_1を開きます。Unity 5.2のバージョンが必要です。Window -> Lighting -> Skyboxを選択します。

2. ShaderをDissolveMatに設定します。

3. Unity ShaderのInspectorで、Chapter15-DissolveのShaderを選択します。

4. ShaderのInspectorで、Burn Amount、Burn Line Width、Base (RGB)、Tiling、Offset、Normal Map、Burn First Color、Burn Second Color、Burn Mapの値を設定します。

5. シーンを再生します。

Chapter15-DissolveのShaderを実装します。

1. ShaderのInspectorで、Burn Amountの値を設定します。

```
Properties {
    _BurnAmount ("Burn Amount", Range(0.0, 1.0)) = 0.0
    _LineWidth ("Burn Line Width", Range(0.0, 0.2)) = 0.1
    _MainTex ("Base (RGB)", 2D) = "white" {}
    _BumpMap ("Normal Map", 2D) = "bump" {}
    _BurnFirstColor ("Burn First Color", Color) = (1, 0, 0, 1)
    _BurnSecondColor ("Burn Second Color", Color) = (1, 0, 0, 1)
    _BurnMap ("Burn Map", 2D) = "white" {}
}
```

`_BurnAmount` 0 1 `_LineWidth`
`_MainTex` `_BumpMap`
`_BurnFirstColor` `_BurnSecondColor` `_BurnMap`

2 `SubShader` `Pass`

```
Pass {
    Tags { "LightMode"="ForwardBase" }

    Cull Off

    CGPROGRAM

    #include "Lighting.cginc"
    #include "AutoLight.cginc"

    #pragma multi_compile_fwdbase
```

`Pass` `LightMode` `multi_compile_fwdbase` `Cull`
`Shader`

3

```
struct v2f {
    float4 pos : SV_POSITION;
    float2 uvMainTex : TEXCOORD0;
    float2 uvBumpMap : TEXCOORD1;
    float2 uvBurnMap : TEXCOORD2;
    float3 lightDir : TEXCOORD3;
    float3 worldPos : TEXCOORD4;
    SHADOW_COORDS(5)
};

v2f vert(a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.uvMainTex = TRANSFORM_TEX(v.texcoord, _MainTex);
    o.uvBumpMap = TRANSFORM_TEX(v.texcoord, _BumpMap);
    o.uvBurnMap = TRANSFORM_TEX(v.texcoord, _BurnMap);

    TANGENT_SPACE_ROTATION;
    o.lightDir = mul(rotation, ObjSpaceLightDir(v.vertex)).xyz;

    o.worldPos = mul(_Object2World, v.vertex).xyz;

    TRANSFER_SHADOW(o);

    return o;
}
```

`TRANSFORM_TEX`
`TRANSFER_SHADOW` 9.4

4

```
fixed4 frag(v2f i) : SV_Target {
    fixed3 burn = tex2D(_BurnMap, i.uvBurnMap).rgb;

    clip(burn.r - _BurnAmount);
```



```

float3 tangentLightDir = normalize(i.lightDir);
fixed3 tangentNormal = UnpackNormal(tex2D(_BumpMap, i.uvBumpMap));

fixed3 albedo = tex2D(_MainTex, i.uvMainTex).rgb;

fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;

fixed3 diffuse = _LightColor0.rgb * albedo * max(0, dot(tangentNormal, tangentLightDir));

fixed t = 1 - smoothstep(0.0, _LineWidth, burn.r - _BurnAmount);
fixed3 burnColor = lerp(_BurnFirstColor, _BurnSecondColor, t);
burnColor = pow(burnColor, 5);

UNITY_LIGHT_ATTENUATION(atten, i, i.worldPos);
fixed3 finalColor = lerp(ambient + diffuse * atten, burnColor, t * step(0.0001, _BurnAmount));

return fixed4(finalColor, 1);
}

```

_BurnAmount clip 0
 albedo
 burnColor _LineWidth
 smoothstep t 1 0
 _BurnFirstColor _BurnSecondColor
 pow t + step
 _BurnAmount finalColor

Pass 9.4.5
 Pass “”
 Pass

```

// Pass to render object as a shadow caster
Pass {
    Tags { "LightMode" = "ShadowCaster" }

    CGPROGRAM

    #pragma vertex vert
    #pragma fragment frag

    #pragma multi_compile_shadowcaster

```

Unity Pass LightMode ShadowCaster #pragma
 multi_compile_shadowcaster

0

```

struct v2f {
    V2F_SHADOW_CASTER;
    float2 uvBurnMap : TEXCOORD1;
};

v2f vert(appdata_base v) {
    v2f o;

    TRANSFER_SHADOW_CASTER_NORMALOFFSET(o)

    o.uvBurnMap = TRANSFORM_TEX(v.texcoord, _BurnMap);

    return o;
}

```

```

}

fixed4 frag(v2f i) : SV_Target {
    fixed3 burn = tex2D(_BurnMap, i.uvBurnMap).rgb;

    clip(burn.r - _BurnAmount);

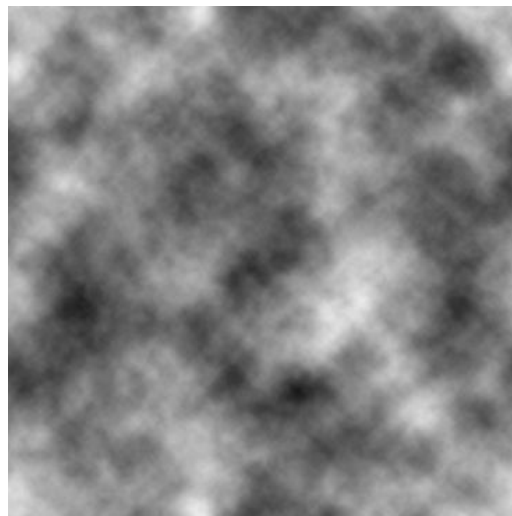
    SHADOW_CASTER_FRAGMENT(i)
}

```

Pass Pass
 Unity V2F_SHADOW_CASTER
 TRANSFER_SHADOW_CASTER_NORMALOFFSET TRANSFER_SHADOW_CASTER
 SHADOW_CASTER_FRAGMENT
 v2f V2F_SHADOW_CASTER
 TRANSFER_SHADOW_CASTER_NORMALOFFSET V2F_SHADOW_CASTER
 Unity uvBurnMap
 SHADOW_CASTER_FRAGMENT Unity

Unity 3 UnityCG.cginc Pass
 TRANSFER_SHADOW_CASTER_NORMALOFFSET
 v v.vortex v.normal appdata_base
 v.vortex TRANSFER_SHADOW_
 CASTER_NORMALOFFSET 11.3.3

Assets/Textures/Chapter15/Burn_Noise.png 15.2
 _BurnMap _BurnAmount
 _BurnAmount



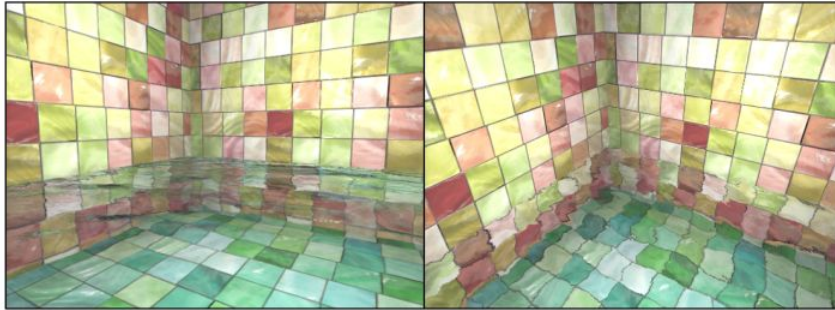
▲ 15.2

Tiling Offset

15.2 □□□□

[illegible]

10.1.5 15.3



▲ 15.3

```

Shader 10.2.2
Cubemap
GrabPass
10.2.2

```

$$fresnel = pow(1 - max(0, \mathbf{v} \cdot \mathbf{n}), 4)$$

v n fresnel

□ □ □ □ □ □ □ □ □ □ □ □ □ □

```

1Scene_15_2Unity 5.2
Window -> Lighting -> Skybox

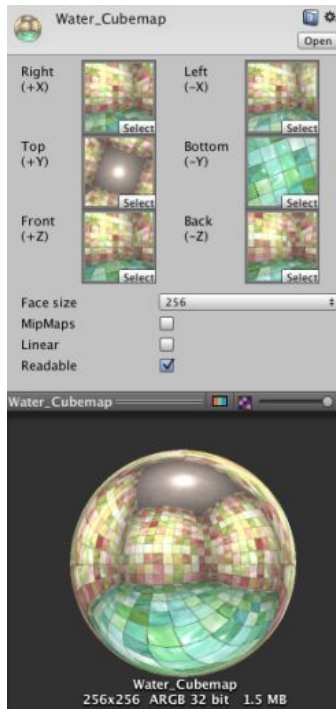
```

`0200000000000000WaterWaveMat`

Chapter15-WaterWaveShader

4. 6. 9.5. 2.

[illegible]



▲15.4 環境映写

Chapter15-WaterWave環境映写

1 Shader

```
Properties {
    _Color ("Main Color", Color) = (0, 0.15, 0.115, 1)
    _MainTex ("Base (RGB)", 2D) = "white" {}
    _WaveMap ("Wave Map", 2D) = "bump" {}
    _Cubemap ("Environment Cubemap", Cube) = "_Skybox" {}
    _WaveXSpeed ("Wave Horizontal Speed", Range(-0.1, 0.1)) = 0.01
    _WaveYSpeed ("Wave Vertical Speed", Range(-0.1, 0.1)) = 0.01
    _Distortion ("Distortion", Range(0, 100)) = 10
}
```

Color、MainTex、WaveMap、Cubemap、Distortion、WaveXSpeed、WaveYSpeed、X、Y

2 GrabPass

```
SubShader {
    // We must be transparent, so other objects are drawn before this one.
    Tags { "Queue"="Transparent" "RenderType"="Opaque" }

    // This pass grabs the screen behind the object into a texture.
    // We can access the result in the next pass as _RefractionTex
    GrabPass { "_RefractionTex" }
```

SubShader Transparent RenderType Opaque Queue Transparent “ ” RenderType Shader Replacement

13 GrabPass Pass Pass 10.2.2

3 Pass Shader

```
fixed4 _Color;
sampler2D _MainTex;
float4 _MainTex_ST;
sampler2D _WaveMap;
float4 _WaveMap_ST;
samplerCUBE _Cubemap;
fixed _WaveXSpeed;
fixed _WaveYSpeed;
float _Distortion;
sampler2D _RefractionTex;
float4 _RefractionTex_TexelSize;
```

_RefractionTex _RefractionTex_TexelSize GrabPass
_RefractionTex_TexelSize 256×512 (1/256, 1/512)

4 10.2.2

```
struct v2f {
    float4 pos : SV_POSITION;
    float4 scrPos : TEXCOORD0;
    float4 uv : TEXCOORD1;
    float4 TtoW0 : TEXCOORD2;
    float4 TtoW1 : TEXCOORD3;
    float4 TtoW2 : TEXCOORD4;
};

v2f vert(a2v v) {
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.scrPos = ComputeGrabScreenPos(o.pos);

    o.uv.xy = TRANSFORM_TEX(v.texcoord, _MainTex);
    o.uv.zw = TRANSFORM_TEX(v.texcoord, _WaveMap);

    float3 worldPos = mul(_Object2World, v.vertex).xyz;
    fixed3 worldNormal = UnityObjectToWorldNormal(v.normal);
    fixed3 worldTangent = UnityObjectToWorldDir(v.tangent.xyz);
    fixed3 worldBinormal = cross(worldNormal, worldTangent) * v.tangent.w;

    o.TtoW0 = float4(worldTangent.x, worldBinormal.x, worldNormal.x, worldPos.x);
    o.TtoW1 = float4(worldTangent.y, worldBinormal.y, worldNormal.y, worldPos.y);
    o.TtoW2 = float4(worldTangent.z, worldBinormal.z, worldNormal.z, worldPos.z);

    return o;
}
```

ComputeGrabScreenPos
UnityCG.cginc ComputeScreenPos
5.6.1 _MainTex _BumpMap float4 xy zw
Cubemap
TtoW0 TtoW1 TtoW2 xyz
3 x y z TtoW0 w

5

```
fixed4 frag(v2f i) : SV_Target {
    float3 worldPos = float3(i.TtoW0.w, i.TtoW1.w, i.TtoW2.w);
    fixed3 viewDir = normalize(UnityWorldSpaceViewDir(worldPos));
    float2 speed = _Time.y * float2(_WaveXSpeed, _WaveYSpeed);

    // Get the normal in tangent space
    fixed3 bump1 = UnpackNormal(tex2D(_WaveMap, i.uv.zw + speed)).rgb;
    fixed3 bump2 = UnpackNormal(tex2D(_WaveMap, i.uv.zw - speed)).rgb;
    fixed3 bump = normalize(bump1 + bump2);

    // Compute the offset in tangent space
    float2 offset = bump.xy * _Distortion * _RefractionTex_TexelSize.xy;
    i.scrPos.xy = offset * i.scrPos.z + i.scrPos.xy;
    fixed3 refrCol = tex2D(_RefractionTex, i.scrPos.xy/i.scrPos.w).rgb;

    // Convert the normal to world space
    bump = normalize(half3(dot(i.TtoW0.xyz, bump), dot(i.TtoW1.xyz, bump), dot(i.TtoW2.xyz, bump)));
    fixed4 texColor = tex2D(_MainTex, i.uv.xy + speed);
    fixed3 reflDir = reflect(-viewDir, bump);
    fixed3 reflCol = texCUBE(_Cubemap, reflDir).rgb * texColor.rgb * _Color.rgb;

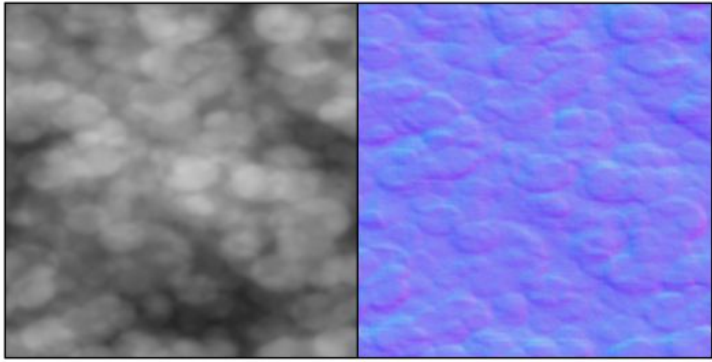
    fixed fresnel = pow(1 - saturate(dot(viewDir, bump)), 4);
    fixed3 finalColor = reflCol * fresnel + refrCol * (1 - fresnel);

    return fixed4(finalColor, 1);
}
```

TtoW0w Time.y WaveXSpeed WaveYSpeed 10.2.2 Distortion RefractionTex_TexelSize Distortion z scrPos RefractionTex

TtoW0TtoW1TtoW2 Cubemap

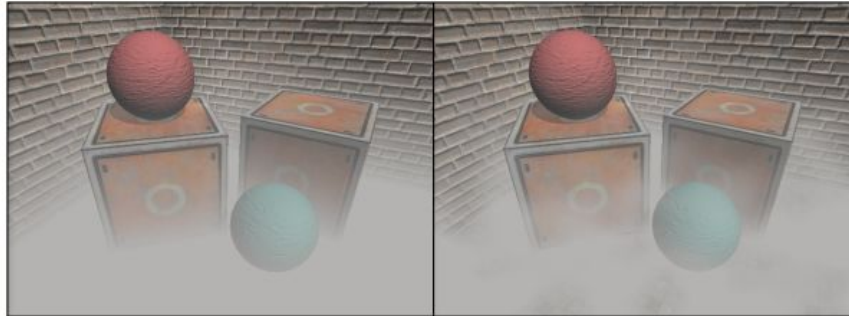
Assets/Textures/Chapter15/Water_Noise.png 15.5 Normal map Create from grayscale 15.5 WaveMap



▲15.5 细胞膜表面结构的模拟与实验对比

15.3 FogWithNoise

13.3 FogWithNoise Shader
13.3 FogWithNoise Shader
15.6 FogWithNoise Shader
15.6 FogWithNoise Shader



▲15.6 FogWithNoise Shader

13.3 FogWithNoise Shader
13.3 FogWithNoise Shader

FogWithNoise

1 Scene_15_3 Unity 5.2
Window -> Lighting -> Skybox

2 3
9.5

FogWithNoise.cs

Unity Shader Shader Chapter15-FogWithNoise

FogWithNoise.cs

12.1

```
public class FogWithNoise : PostEffectsBase {
```

2 Shader

```
public Shader fogShader;
private Material fogMaterial = null;

public Material material {
    get {
        fogMaterial = CheckShaderAndCreateMaterial(fogShader, fogMaterial);
        return fogMaterial;
    }
}
```



```
}
```

3. 3DカメラのFOVをカメラのTransformから取得する

```
private Camera myCamera;
public Camera camera {
    get {
        if (myCamera == null) {
            myCamera = GetComponent<Camera>();
        }
        return myCamera;
    }
}

private Transform myCameraTransform;
public Transform cameraTransform {
    get {
        if (myCameraTransform == null) {
            myCameraTransform = camera.transform;
        }

        return myCameraTransform;
    }
}
```

4. 霧の密度、色、スタート、エンド、ノイズテクスチャ、X、Yの速度、ノイズ量を設定する

```
[Range(0.1f, 3.0f)]
public float fogDensity = 1.0f;

public Color fogColor = Color.white;

public float fogStart = 0.0f;
public float fogEnd = 2.0f;

public Texture noiseTexture;

[Range(-0.5f, 0.5f)]
public float fogXSpeed = 0.1f;

[Range(-0.5f, 0.5f)]
public float fogYSpeed = 0.1f;

[Range(0.0f, 3.0f)]
public float noiseAmount = 1.0f;
```

霧の密度、色、スタート、エンド、ノイズテクスチャ、X、Yの速度、ノイズ量を設定する。fogDensityは霧の密度、fogColorは霧の色、fogStartは霧のスタート、fogEndは霧のエンド、noiseTextureはノイズテクスチャ、fogXSpeedはXの速度、fogYSpeedはYの速度、noiseAmountはノイズ量。noiseAmountは0から3.0fの範囲で設定する。

5. OnEnableメソッドを実装する

```
void OnEnable() {
    camera.depthTextureMode |= DepthTextureMode.Depth;
}
```

6 OnRenderImage

```
void OnRenderImage (RenderTexture src, RenderTexture dest) {
    if (material != null) {
        Matrix4x4 frustumCorners = Matrix4x4.identity;

        // Compute frustumCorners
        ...

        material.SetMatrix("_FrustumCornersRay", frustumCorners);

        material.SetFloat("_FogDensity", fogDensity);
        material.SetColor("_FogColor", fogColor);
        material.SetFloat("_FogStart", fogStart);
        material.SetFloat("_FogEnd", fogEnd);

        material.SetTexture("_NoiseTex", noiseTexture);
        material.SetFloat("_FogXSpeed", fogXSpeed);
        material.SetFloat("_FogYSpeed", fogYSpeed);
        material.SetFloat("_NoiseAmount", noiseAmount);

        Graphics.Blit (src, dest, material);
    } else {
        Graphics.Blit(src, dest);
    }
}
```

13.3 4 frustumCorners
13.3 Graphics.Blit (src, dest, material)

Shader Chapter15-FogWithNoise

1

```
Properties {
    _MainTex ("Base (RGB)", 2D) = "white" {}
    _FogDensity ("Fog Density", Float) = 1.0
    _FogColor ("Fog Color", Color) = (1, 1, 1, 1)
    _FogStart ("Fog Start", Float) = 0.0
    _FogEnd ("Fog End", Float) = 1.0
    _NoiseTex ("Noise Texture", 2D) = "white" {}
    _FogXSpeed ("Fog Horizontal Speed", Float) = 0.1
    _FogYSpeed ("Fog Vertical Speed", Float) = 0.1
    _NoiseAmount ("Noise Amount", Float) = 1
}
```

2 CGINCLUDE SubShader CGINCLUDE ENDCG

```
SubShader {
    CGINCLUDE
    ...
    ENDCG
    ...
}
```

3

```
float4x4 _FrustumCornersRay;

sampler2D _MainTex;
half4 _MainTex_TexelSize;
sampler2D _CameraDepthTexture;
half _FogDensity;
fixed4 _FogColor;
float _FogStart;
float _FogEnd;
sampler2D _NoiseTex;
half _FogXSpeed;
half _FogYSpeed;
half _NoiseAmount;
```

`_FrustumCornersRay` Shader Properties
`_CameraDepthTexture` Unity Shader Properties

4 13.3 13.3

5

```
fixed4 frag(v2f i) : SV_Target {
    float linearDepth = LinearEyeDepth(SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture, i. uv_depth));
    float3 worldPos = _WorldSpaceCameraPos + linearDepth * i.interpolatedRay.xyz;

    float2 speed = _Time.y * float2(_FogXSpeed, _FogYSpeed);
    float noise = (tex2D(_NoiseTex, i.uv + speed).r - 0.5) * _NoiseAmount;

    float fogDensity = (_FogEnd - worldPos.y) / (_FogEnd - _FogStart);
    fogDensity = saturate(fogDensity * _FogDensity * (1 + noise));

    fixed4 finalColor = tex2D(_MainTex, i.uv);
    finalColor.rgb = lerp(finalColor.rgb, _FogColor.rgb, fogDensity);

    return finalColor;
}
```

`_Time.y` `_FogXSpeed` `_FogYSpeed` 0.5 `_NoiseAmount` `fogDensity`

6 Pass

```
Pass {
    CGPROGRAM

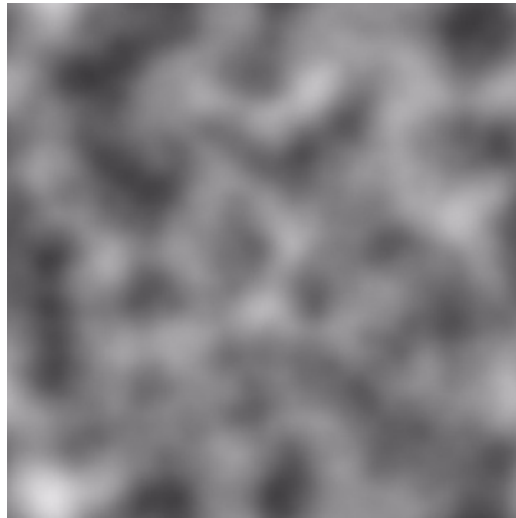
    #pragma vertex vert
    #pragma fragment frag

    ENDCG
}
```

7 Shader Fallback

Fallback Off

Chapter15-FogWithNoise FogWithNoise.cs fogShader
FogWithNoise.cs fogShader Chapter15-FogWithNoise
Assets/Textures/Chapter15/Fog_Noise.jpg 15.7 FogWithNoise.cs
noiseTexture FogWithNoise.cs noiseTexture
Fog_Noise.jpg



▲ 15.7 噪声纹理

15.4 ノイズ

この章では、 procedural texture の中でも、ノイズを用いた procedural texture の生成について解説する。ノイズとは、ランダムな値を生成する関数のことである。 procedural texture の生成には、 Perlin ノイズと Worley ノイズが代表的である。 Perlin ノイズは、 Perlin によって開発されたノイズ関数で、 procedural texture の生成に広く用いられる。 Worley ノイズは、 Worley によって開発されたノイズ関数で、 procedural texture の生成に広く用いられる。 15.3 章では、 Perlin ノイズと Worley ノイズの生成方法について解説した。 Perlin ノイズは、 Perlin によって開発されたノイズ関数で、 procedural texture の生成に広く用いられる。 Worley ノイズは、 Worley によって開発されたノイズ関数で、 procedural texture の生成に広く用いられる。 Photoshop では、 procedural texture の生成に Perlin ノイズと Worley ノイズが広く用いられる。 Perlin ノイズは、 Perlin によって開発されたノイズ関数で、 procedural texture の生成に広く用いられる。 Worley ノイズは、 Worley によって開発されたノイズ関数で、 procedural texture の生成に広く用いられる。 <http://flafla2.github.io/2014/08/09/perlinnoise.html> は、 Perlin ノイズの生成方法について詳しく解説している。 Worley ノイズは、 Worley 1998 によって開発されたノイズ関数で、 procedural texture の生成に広く用いられる。 <http://scrawkblog.com/category/procedural-noise/> は、 Worley ノイズの生成方法について詳しく解説している。 Unity では、 procedural texture の生成に Perlin ノイズと Worley ノイズが広く用いられる。

15.5 □□□□

[1] Worley S. A cellular texture basis
function[C]//Proceedings of the 23rd annual conference on
Computer graphics and interactive techniques. ACM, 1996:
291-294□

16 Unity

Unity is a cross-platform game engine that can be used to develop games for a wide range of platforms, including PC, console, and mobile.

—Michael A. Jackson

Michael A. Jackson is a game developer and the author of the book "Unity: A Beginner's Guide to Game Development". He has been working in the game industry since 1988 and has developed games for a variety of platforms, including PC, console, and mobile.

Unity is a cross-platform game engine that can be used to develop games for a wide range of platforms, including PC, console, and mobile. It is a powerful and flexible engine that allows developers to create high-quality games with a wide range of features and capabilities. Unity is also a very easy-to-use engine, making it a great choice for both beginners and experienced developers.

Unity is a cross-platform game engine that can be used to develop games for a wide range of platforms, including PC, console, and mobile. It is a powerful and flexible engine that allows developers to create high-quality games with a wide range of features and capabilities. Unity is also a very easy-to-use engine, making it a great choice for both beginners and experienced developers.

Unity is a cross-platform game engine that can be used to develop games for a wide range of platforms, including PC, console, and mobile. It is a powerful and flexible engine that allows developers to create high-quality games with a wide range of features and capabilities. Unity is also a very easy-to-use engine, making it a great choice for both beginners and experienced developers.

16.1 性能優化

PC 平台上的 GPU 性能優化通常比移動平台上的 GPU 性能優化要容易得多。PC 平台上的 GPU 性能優化通常比移動平台上的 GPU 性能優化要容易得多。

在移動平台上，overdraw 是一個常見的性能問題。PowerVR 和 iOS 平台上的 Android 平台上的 Tiled-based Deferred Rendering (TBDR) 是一種基於 tile 的 GPU 渲染技術。Adreno 和 Mali ARM 平台上的 Early-Z 是一種基於 GPU 的 Tegra 平台上的 overdraw 性能問題。

在移動平台上，Android 平台上的 Android 平台上的 Android 平台上的 iOS 平台上的 Unity 平台上的 iOS 平台上的 <http://docs.unity3d.com/Manual/iphone-Hardware.html> 是一個常見的性能問題。

16.2 图形渲染流程

图形渲染流程可以分为三个阶段：CPU阶段、GPU阶段和CPU阶段。CPU阶段主要负责数据的准备和上传，GPU阶段主要负责数据的渲染，CPU阶段主要负责数据的清理和更新。

图形渲染流程可以分为三个阶段：

1. CPU阶段

- 调用draw call
- 数据上传

2. GPU阶段

- 渲染
 - 顶点着色
 - 片段着色
- 清理
 - 防止overdraw
 - 深度测试

3. CPU阶段

- 数据清理
- 数据更新

CPU阶段主要负责数据的准备和上传，GPU阶段主要负责数据的渲染，CPU阶段主要负责数据的清理和更新。draw call是CPU阶段的一个重要操作，它负责将数据上传到GPU并启动渲染。draw call的频率会影响CPU的负载，因此优化draw call的数量和频率是提高渲染性能的关键。draw call的频率越高，CPU的负载越大，渲染性能越低。draw call的频率越低，CPU的负载越小，渲染性能越高。draw call的频率应该根据具体的应用场景进行优化。

ゲームエンジンではCPUとGPUの役割分担が非常に重要である。CPUは主にゲームロジック、物理演算、AI処理などを担当し、GPUはグラフィックスのレンダリングを担当する。ShaderはGPU上で実行されるプログラムで、物体の見た目を決定する。

GPUはCPUよりもはるかに高速な処理能力を持つが、メモリ容量が小さい。そのため、GPUに大量のデータを渡すことは避け、必要なデータだけをGPUに渡す必要がある。また、GPUは並列処理に優れているため、複数のオブジェクトを同時にレンダリングできる。

ゲームエンジンの最適化には、CPUとGPUの両方を考慮する必要がある。

1. CPU最適化

- draw callの削減

2. GPU最適化

- バッチ処理
◦ 頂点バッッチ
◦ インスタンスレンダリング
◦ Occlusion Cullingの活用
- テクスチャ最適化
◦ テクスチャ圧縮
◦ テクスチャストリーミング
◦ テクスチャアトラス
- Shader最適化
◦ ShaderのLOD (Level of Detail) 制御
◦ Shaderの再利用

3. その他最適化

- 物理演算の最適化
- AI処理の最適化

Unityはゲームエンジンとして広く使われているが、最適化にはいくつかの注意点がある。例えば、Draw Callの削減にはMeshの分割やバッチ処理の活用が有効である。

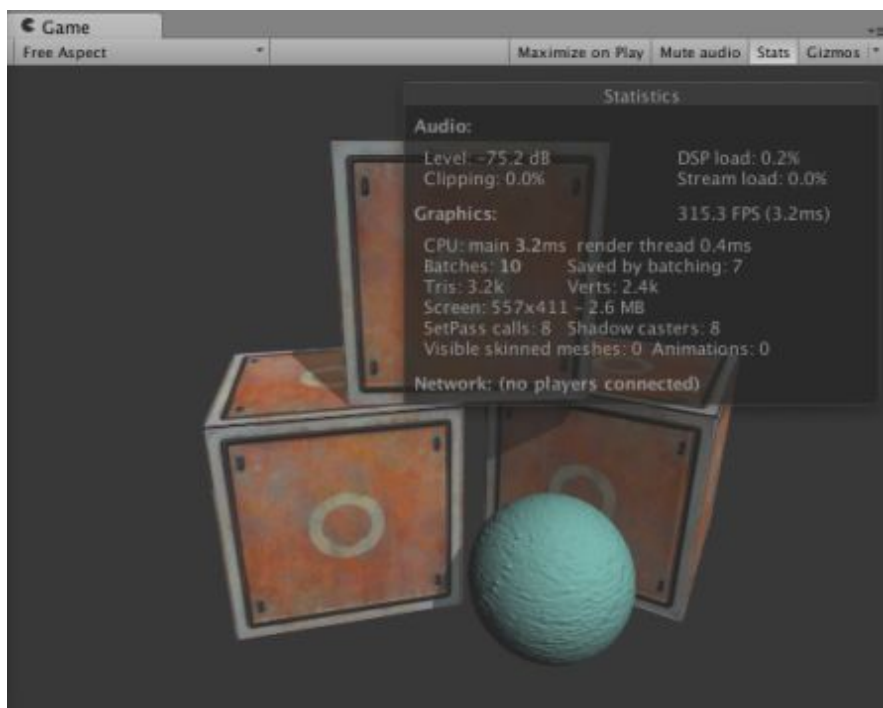
16.3 Unity

Unityは、ゲーム開発のための統合開発環境（IDE）です。Unity 5では、**Rendering Statistics Window**、**Profiler**、**Frame Debugger**などのツールが提供されています。

16.3.1 Unity 5

Unity 5では、**Rendering Statistics Window**が、ゲームのリアルタイムパフォーマンスを監視するためのツールです。Statsウィンドウは、16.1から16.1のバージョンで3つのカテゴリ（Audio、Graphics、Network）に分かれています。

このウィンドウは、FPS（フレームレート）などのリアルタイムパフォーマンス指標を提供します。



▲16.1 Unity 5

□16.1

項目	概要
フレームワークFPS	GraphicフレームワークのFPS
Batches	バッチ数
Saved by batching	バッチングで節約されたdraw call
Tris/Verts	ポリゴン数/頂点数
Screen	スクリーンサイズ
SetPass	Passの回数、Unity runtime ShaderのCPUコスト
Visible Skinned Meshes	可視なスキinnedメッシュの数
Animations	アニメーションの数

```

Unity 5 draw call
Batches Saved by batching
draw call
Unity

```

16.3.2 □□□□□□□□

Window -> Profiler Unity Profiler
Rendering Area 16.2
16.1



▲ 16.2 Unity Profiler

Unity Profilerは、ゲームの性能を監視するためのツールです。Pass、draw call、batchingなどの情報を表示します。

Unity Profilerは、ゲームの性能を監視するためのツールです。draw call、Pass、batchingなどの情報を表示します。Unity 5では、draw callの数を減らすことで、パフォーマンスを向上させることができます。

16.3.3 最適化

Unity 5.5.3のFrame Debuggerは、draw callの数を減らすためのツールです。Shader、Window -> Frame Debugger、draw callなどの情報を表示します。

Android Adreno
NVPerfHUD draw call
GPU shader cycle

iOS Unity GPU PowerVRAM
PVR UniCo shader Xcode OpenGL ES
Driver Instruments
Android iOS PowerVR
draw call GPU

Unity
<http://docs.unity3d.com/Manual/MobileProfiling.html>
Mobile Profiling

16.4 draw call

draw call と batching は、CPU と GPU の間で、draw call と shader の呼び出しを最適化する技術です。CPU は draw call を呼び出す側、GPU は shader を実行する側です。draw call を呼び出すたびに、CPU は GPU に shader を渡す必要があります。 batching は、複数の draw call をまとめて GPU に渡すことで、CPU と GPU の間の通信回数を減らすことで、パフォーマンスを向上させる技術です。 Unity は、 batching を自動的に実行しますが、2.2 と 2.4.3 のバージョンでは、 draw call が多すぎて、パフォーマンスが低下することがあります。 Unity の batching は、 draw call を自動的に実行しますが、 batching のサイズは、 Unity のバージョンによって異なります。

batching は、 draw call を自動的に実行しますが、 batching のサイズは、 Unity のバージョンによって異なります。 batching のサイズは、 Unity のバージョンによって異なります。 batching のサイズは、 Unity のバージョンによって異なります。

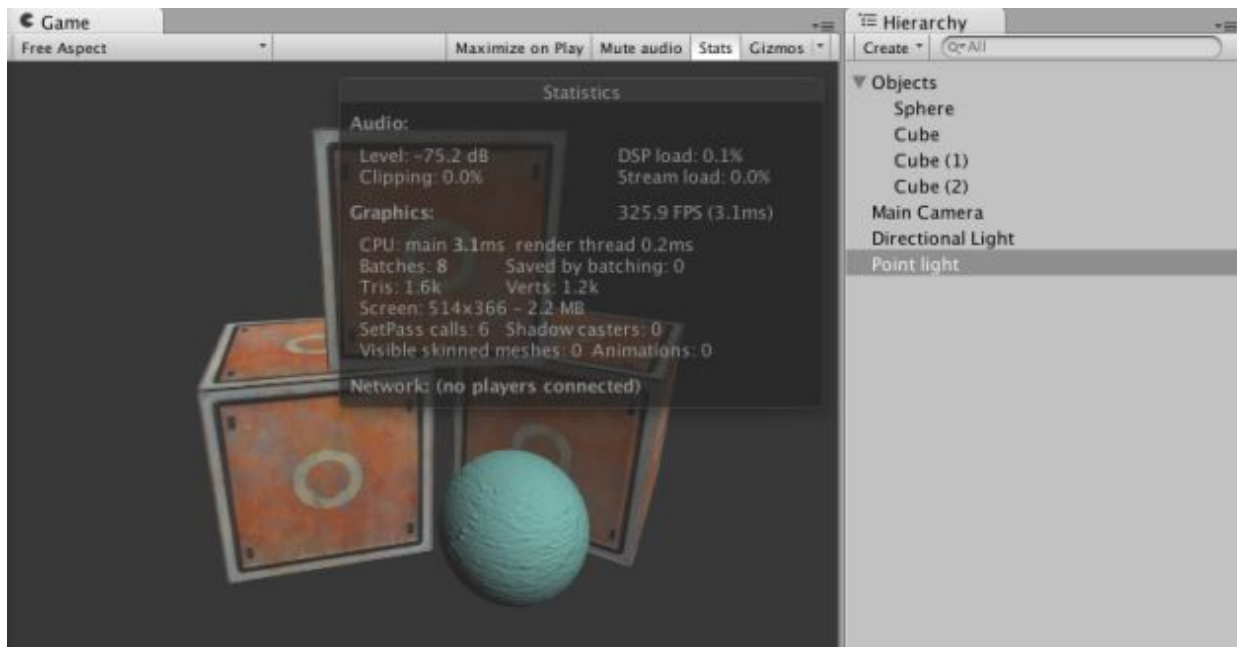
Unity は、 batching を自動的に実行しますが、 batching のサイズは、 Unity のバージョンによって異なります。 batching のサイズは、 Unity のバージョンによって異なります。 batching のサイズは、 Unity のバージョンによって異なります。

16.4.1 batching

batching は、 draw call を自動的に実行しますが、 batching のサイズは、 Unity のバージョンによって異なります。 batching のサイズは、 Unity のバージョンによって異なります。 batching のサイズは、 Unity のバージョンによって異なります。

Unity は、 batching を自動的に実行しますが、 batching のサイズは、 Unity のバージョンによって異なります。 batching のサイズは、 Unity のバージョンによって異なります。 batching のサイズは、 Unity のバージョンによって異なります。

- batching のサイズは、 Unity のバージョンによって異なります。 batching のサイズは、 Unity のバージョンによって異なります。 batching のサイズは、 Unity のバージョンによって異なります。



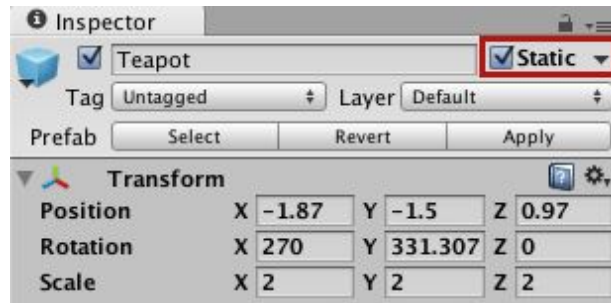
▲ 16.5 パフォーマンス最適化

16.5バージョンでは、バッチング（Batching）が大幅に改善されています。以前は、8つのオブジェクトごとに「Save by batching」が必要でしたが、現在は0回のバッチングで済みます。また、Shaderのバッチングも大幅に改善されています。以前は、4つのShaderごとに2×4回のバッチングが必要でしたが、現在はUnityのバッチングで済みます。Passのバッチングも大幅に改善されています。

バッチングの改善により、draw callの数が大幅に減少しました。以前は、900回のdraw callが必要でしたが、現在はUnityのバッチングで済みます。

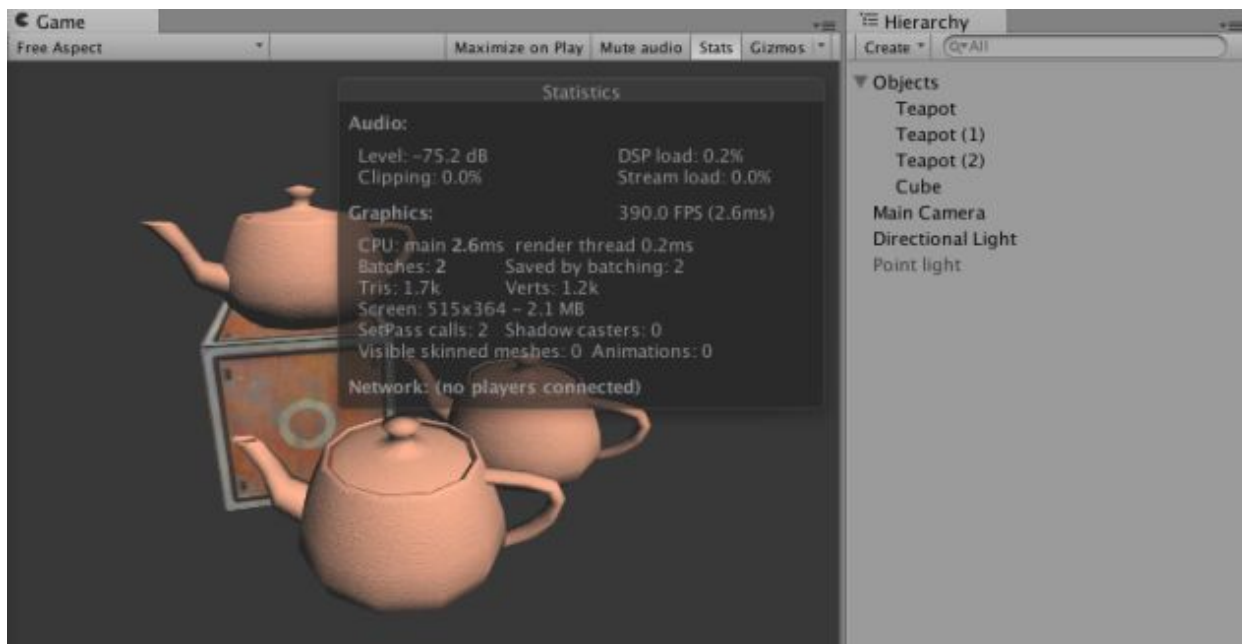
16.4.2 バッチング

Unityのバッチングは、オブジェクトのバッチングとShaderのバッチングの2種類があります。オブジェクトのバッチングは、オブジェクトの位置、サイズ、色などが同じ場合にバッチングされます。Shaderのバッチングは、Shaderの種類が同じ場合にバッチングされます。GPUのバッチングは、GPUのバッチングで済みます。1 000回のバッチングで済みます。1 000回のバッチングで済みます。



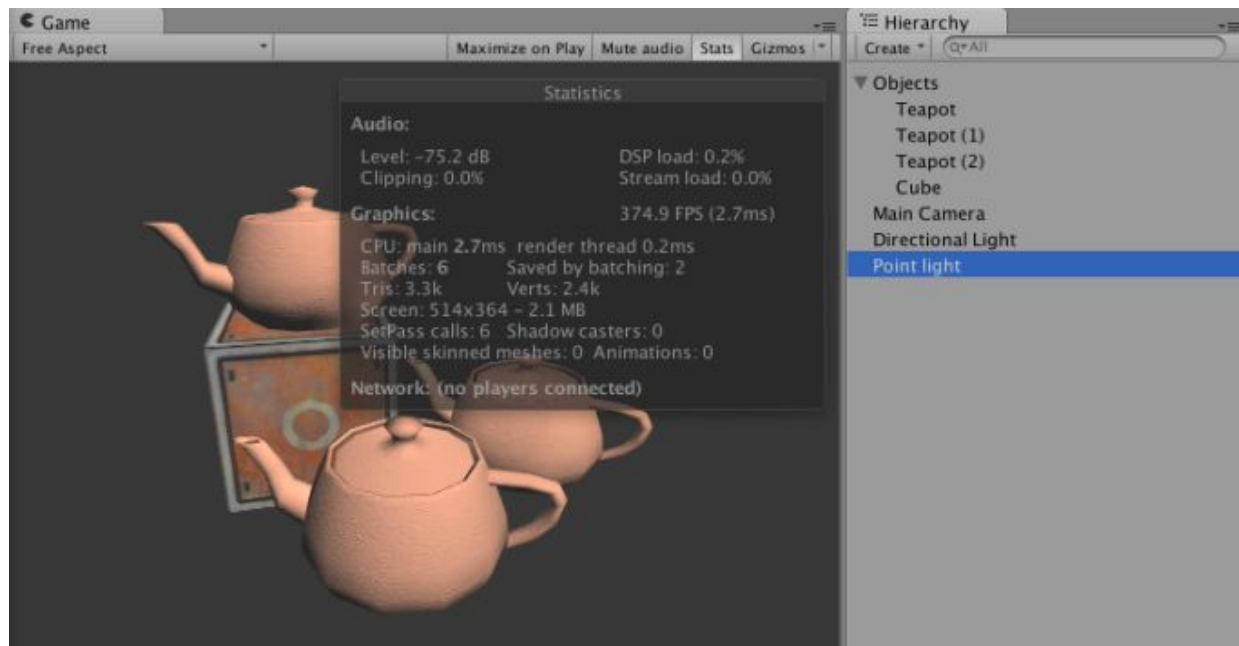
▲ 16.7 Static

16.2 2 **Save by batching** Combined Mesh (root: scene) 16.9 Unity "Static" 3 Teapot 4 16.9 16.9 "4 submeshes" 4 Unity



▲ 16.8

shader Pass
 Pass 16.11 4



▲ 16.11 Pass

Base Pass draw call

16.4.3

atlas

Shader

Renderer.material 是 Unity 的 Material 类，它定义了渲染器使用的着色器。Renderer.sharedMaterial 是 Renderer 类的一个静态属性，它指向一个共享的 Material 对象。Renderer.material 是 Renderer 类的一个属性，它指向一个 Material 对象。Renderer.sharedMaterial 是 Renderer 类的一个静态属性，它指向一个共享的 Material 对象。

Renderer.sharedMaterial 是 Renderer 类的一个静态属性，它指向一个共享的 Material 对象。Renderer.material 是 Renderer 类的一个属性，它指向一个 Material 对象。Renderer.sharedMaterial 是 Renderer 类的一个静态属性，它指向一个共享的 Material 对象。Renderer.material 是 Renderer 类的一个属性，它指向一个 Material 对象。

Renderer.sharedMaterial 是 Renderer 类的一个静态属性，它指向一个共享的 Material 对象。Renderer.material 是 Renderer 类的一个属性，它指向一个 Material 对象。Renderer.sharedMaterial 是 Renderer 类的一个静态属性，它指向一个共享的 Material 对象。Renderer.material 是 Renderer 类的一个属性，它指向一个 Material 对象。

16.4.4 静态渲染器

静态渲染器是指那些不需要通过 Shader 来渲染的物体。它们通常用于渲染一些简单的几何体，如立方体、球体等。

- 静态渲染器不需要通过 Shader 来渲染，它们通常用于渲染一些简单的几何体，如立方体、球体等。
- 静态渲染器不需要通过 Shader 来渲染，它们通常用于渲染一些简单的几何体，如立方体、球体等。
- 静态渲染器不需要通过 Shader 来渲染，它们通常用于渲染一些简单的几何体，如立方体、球体等。
- 静态渲染器不需要通过 Shader 来渲染，它们通常用于渲染一些简单的几何体，如立方体、球体等。

静态渲染器是指那些不需要通过 Shader 来渲染的物体。它们通常用于渲染一些简单的几何体，如立方体、球体等。静态渲染器不需要通过 Shader 来渲染，它们通常用于渲染一些简单的几何体，如立方体、球体等。静态渲染器不需要通过 Shader 来渲染，它们通常用于渲染一些简单的几何体，如立方体、球体等。

Unity 5.2 版本引入了静态渲染器。静态渲染器是指那些不需要通过 Shader 来渲染的物体。它们通常用于渲染一些简单的几何体，如立方体、球体等。

16.5 3D 模型渲染

draw call 是 CPU 和 GPU 之间通信的接口，3D 模型渲染

16.5.1 3D 模型渲染

3D 模型渲染是指将 3D 模型数据转换为 2D 图像的过程。这个过程涉及到模型的几何数据、材质属性、光照信息等。Unity 引擎提供了强大的 3D 渲染系统，可以处理各种复杂的 3D 模型和场景。

Unity 引擎的渲染系统由多个组件组成，包括渲染器、材质系统、光照系统等。Unity 渲染器负责将 3D 模型数据转换为 GPU 可以处理的格式，并生成最终的 2D 图像。Unity 材质系统负责管理模型的材质属性，Unity 光照系统负责计算光照效果。

在 Unity 引擎中，3D 模型渲染涉及到多个关键概念，包括 **uv splits** 和 **smoothing splits**。uv splits 是指将模型的 UV 坐标进行分割，以便在不同的 GPU 实例中进行渲染。smoothing splits 是指将模型的法线进行分割，以便在不同的 GPU 实例中进行渲染。这些概念对于优化 3D 模型的渲染性能至关重要。此外，模型边缘的渲染也是一个重要的考虑因素，包括 hard edge 和 smooth edge 的设置。

GPU 渲染是指将 3D 模型数据发送到 GPU 进行渲染的过程。GPU 渲染器负责将 3D 模型数据转换为 GPU 可以处理的格式，并生成最终的 2D 图像。GPU 渲染系统通常由多个 GPU 实例组成，可以并行处理大量的 3D 模型数据。

16.5.2 LOD 设置

LOD (Level of Detail) 是指根据模型的视距来动态调整模型的细节级别。LOD 设置可以显著提高 3D 模型的渲染性能，特别是在处理大量模型的场景中。LOD 设置通常涉及到模型的几何数据、材质属性、光照信息等。

16.6 如何减少过draw

如何减少GPU的过draw？过draw是指一个像素被多次绘制，这会导致GPU的过draw增加，从而降低性能。

Unity中，过draw是指一个像素被多次绘制，这会导致GPU的过draw增加，从而降低性能。过draw是指一个像素被多次绘制，这会导致GPU的过draw增加，从而降低性能。过draw是指一个像素被多次绘制，这会导致GPU的过draw增加，从而降低性能。

16.6.1 如何减少过draw

如何减少过draw？过draw是指一个像素被多次绘制，这会导致GPU的过draw增加，从而降低性能。过draw是指一个像素被多次绘制，这会导致GPU的过draw增加，从而降低性能。过draw是指一个像素被多次绘制，这会导致GPU的过draw增加，从而降低性能。

Unity中，过draw是指一个像素被多次绘制，这会导致GPU的过draw增加，从而降低性能。过draw是指一个像素被多次绘制，这会导致GPU的过draw增加，从而降低性能。过draw是指一个像素被多次绘制，这会导致GPU的过draw增加，从而降低性能。

如何减少过draw？过draw是指一个像素被多次绘制，这会导致GPU的过draw增加，从而降低性能。过draw是指一个像素被多次绘制，这会导致GPU的过draw增加，从而降低性能。过draw是指一个像素被多次绘制，这会导致GPU的过draw增加，从而降低性能。

如何减少过draw？过draw是指一个像素被多次绘制，这会导致GPU的过draw增加，从而降低性能。过draw是指一个像素被多次绘制，这会导致GPU的过draw增加，从而降低性能。过draw是指一个像素被多次绘制，这会导致GPU的过draw增加，从而降低性能。

16.6.2 如何减少过draw

overdraw GUI GUI overdraw

overdraw

GUI GUI GUI GUI

discard clip PowerVR overdraw discard GPU GPU overdraw

16.6.3

Pass Shader 3 draw call CPU 3 overdraw GPU Pass

lightmap God Ray 1

ShadowGun lookup texture lookup table LUT BRDF NPC LUT ShadowGun LUT LUT

CPU draw call GPU

16.7 ☐☐☐☐

[illegible]

16.7.1 练习

```

    draw call
    2
    Unity 5
    Unity 2
    2

```

mipmapping Unity
Advanced
mipmaps 16.12
Generate Mip Maps Unity
GPU
GUI 2D



▲ 16.12 Unity

Android GPU PowerVR TC Tegra DXT Adreno ATC Unity GUI

16.7.2

GPU

Unity Screen.SetResolution MOMO <http://www.xuanyusong.com/archives/3205>

16.8 性能优化

本章主要介绍在 Shader 性能优化方面的一些技巧和最佳实践。

16.8.1 Shader LOD

在 16.5.2 节中我们介绍了 LOD 的概念。Shader LOD 是指 Shader 在不同 LOD 级别下使用不同的 Shader。Shader LOD 是指 Shader 在不同 LOD 级别下使用不同的 Shader。Shader LOD 是指 Shader 在不同 LOD 级别下使用不同的 Shader。

在 Shader 中，SubShader 块可以指定 shader LOD 值。

```
SubShader {  
    Tags { "RenderType"="Opaque" }  
    LOD 200
```

在 Unity Shader 中，Shader 的 LOD 值是指 Shader 在不同 LOD 级别下使用不同的 Shader。Shader LOD 是指 Shader 在不同 LOD 级别下使用不同的 Shader。Shader LOD 是指 Shader 在不同 LOD 级别下使用不同的 Shader。Shader.maximumLOD 是指 Shader.globalMaximumLOD 的值。

Unity 中 Shader 的 LOD 值是指 Shader 在不同 LOD 级别下使用不同的 Shader。Shader LOD 是指 Shader 在不同 LOD 级别下使用不同的 Shader。Shader LOD 是指 Shader 在不同 LOD 级别下使用不同的 Shader。Diffuse LOD 200 是指 Bumped Specular LOD 400。

16.8.2 性能优化

在 Shader 性能优化方面，有一些技巧和最佳实践。首先，要确保 Shader 的 LOD 值是正确的。其次，要确保 Shader 的 LOD 值是正确的。最后，要确保 Shader 的 LOD 值是正确的。

在 Shader 性能优化方面，有一些技巧和最佳实践。首先，要确保 Shader 的 LOD 值是正确的。其次，要确保 Shader 的 LOD 值是正确的。最后，要确保 Shader 的 LOD 值是正确的。iOS 是指 Shader 在不同 LOD 级别下使用不同的 Shader。

float/mediump和float/lowp的精度。在OpenGL ES 2.0中，float/mediump和float/lowp的精度是固定的，而float/highp的精度是可配置的。

在OpenGL ES 2.0中，float/mediump和float/lowp的精度是固定的，而float/highp的精度是可配置的。在OpenGL ES 2.0中，float/mediump和float/lowp的精度是固定的，而float/highp的精度是可配置的。在OpenGL ES 2.0中，float/mediump和float/lowp的精度是固定的，而float/highp的精度是可配置的。

在OpenGL ES 2.0中，float/mediump和float/lowp的精度是固定的，而float/highp的精度是可配置的。在OpenGL ES 2.0中，float/mediump和float/lowp的精度是固定的，而float/highp的精度是可配置的。在OpenGL ES 2.0中，float/mediump和float/lowp的精度是固定的，而float/highp的精度是可配置的。

在OpenGL ES 2.0中，float/mediump和float/lowp的精度是固定的，而float/highp的精度是可配置的。在OpenGL ES 2.0中，float/mediump和float/lowp的精度是固定的，而float/highp的精度是可配置的。在OpenGL ES 2.0中，float/mediump和float/lowp的精度是固定的，而float/highp的精度是可配置的。

在OpenGL ES 2.0中，float/mediump和float/lowp的精度是固定的，而float/highp的精度是可配置的。

- 在OpenGL ES 2.0中，float/mediump和float/lowp的精度是固定的，而float/highp的精度是可配置的。
- 在OpenGL ES 2.0中，float/mediump和float/lowp的精度是固定的，而float/highp的精度是可配置的。
- 在OpenGL ES 2.0中，float/mediump和float/lowp的精度是固定的，而float/highp的精度是可配置的。

16.8.3 在OpenGL ES 2.0中，float/mediump和float/lowp的精度是固定的，而float/highp的精度是可配置的。

iOSAndroid
10
scaling
“”

16.9 參考

Unity 參考
<http://docs.unity3d.com/Manual/MobileOptimizationPracticalGuide.html> 參考
<http://docs.unity3d.com/Manual/OptimizingGraphicsPerformance.html> 參考
Unity 參考

SIGGRAPH 2011 參考 Unity Shader
<http://blogs.unity3d.com/2011/08/18/fast-mobile-shaders-talk-at-siggraph/> 參考 GPU
shader Unity Unite 2013
Unity **Unity** 參考
GDC 2014 Unity
Youtube SIGGRAPH
2015 Unity Unity ARM
Moving Mobile Graphics Unity Renaldas Zioma PBR Unity SIGGRAPH 2015
Unity

ShadowGun
MadFinger 2011 Unity
Unite 2011 ShadowGun
Youtube 2012 ShadowGun
shader Unity
Angry Bots Unity

5 問題

Unityのインストールと実行環境の構築について説明してください。

17 Unityのインストール

Unityのインストール手順を説明してください。

18 実行環境

Unityの実行環境の構築について説明してください。

19 Unity 5のインストール

Unity 5のインストールと実行環境の構築について説明してください。

20 実行環境

Unity 5の実行環境の構築について説明してください。

17 Unity Shader

2009 Unity 2.x Unity Aras
Aras Pranckevičius 3 Shaders
must die Aras
/ /
Blinn-Phong
Shader
Aras
Unity 2010 Unity 3 **Surface Shader**
Unity

Unity “” **Surface Shader**
/ Aras / “”
Shader “”
”
Pass
Shader Unity “”

17.1

[illegible]

```

1UnityScene_17_1Unity
5.2Window
→Lighting → Skybox

```

2 BumpedSpecularMat

Chapter17-BumpedDiffuse

4 capsules 2

□5□□□□□

Unity
“Legacy Shaders/Bumped Diffuse”
Shader Chapter17-BumpedDiffuse

```
Shader "Unity Shaders Book/Chapter 17/Bumped Diffuse" {
    Properties {
        _Color ("Main Color", Color) = (1,1,1,1)
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _BumpMap ("Normalmap", 2D) = "bump" {}
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 300

        CGPROGRAM
        #pragma surface surf Lambert
        #pragma target 3.0

        sampler2D _MainTex;
        sampler2D _BumpMap;
        fixed4 _Color;
```

```

    struct Input {
        float2 uv_MainTex;
        float2 uv_BumpMap;
    };

    void surf (Input IN, inout SurfaceOutput o) {
        fixed4 tex = tex2D(_MainTex, IN.uv_MainTex);
        o.Albedo = tex.rgb * _Color.rgb;
        o.Alpha = tex.a * _Color.a;
        o.Normal = UnpackNormal(tex2D(_BumpMap, IN.uv_BumpMap));
    }

    ENDCG
}

Fallback "Legacy Shaders/Diffuse"
}

```

```

        UnityBumpedDiffuseMat
Assets/Textures/Chapter17/Mud_Diffuse.tif
Assets/Textures/Chapter17/Mud_Normal.tif_MainTex
_BumpMap17.1
17.1

```



▲17.1

[illegible]

Shaders Book/Common/Bumped Diffuse”
Unity

Unity
<http://docs.unity3d.com/Manual/SL-SurfaceShaderExamples.html>
Unity

Pass CG
SubShader Unity Pass SubShader
Tags Chapter17-BumpedDiffuse
LOD LOD 16.8.1
CGPROGRAM **ENDCG**

Unity

17.2 表面

表面は、UnityのシェーダーでUnityの“表面”を定義するための関数です。表面は、Unityの“表面”を定義するための関数です。

表面は、UnityのシェーダーでUnityの“表面”を定義するための関数です。表面は、Unityの“表面”を定義するための関数です。

```
#pragma surface surfaceFunction lightModel [optionalparams]
```

表面は、UnityのシェーダーでUnityの“表面”を定義するための関数です。表面は、Unityの“表面”を定義するための関数です。

17.2.1 表面

表面は、UnityのシェーダーでUnityの“表面”を定義するための関数です。表面は、Unityの“表面”を定義するための関数です。

```
void surf (Input IN, inout SurfaceOutput o)  
void surf (Input IN, inout SurfaceOutputStandard o)  
void surf (Input IN, inout SurfaceOutputStandardSpecular o)
```

表面は、Unity 5のシェーダーでUnityの“表面”を定義するための関数です。表面は、Unityの“表面”を定義するための関数です。

表面は、UnityのシェーダーでUnityの“表面”を定義するための関数です。表面は、Unityの“表面”を定義するための関数です。

17.2.2 表面

表面は、UnityのシェーダーでUnityの“表面”を定義するための関数です。表面は、Unityの“表面”を定義するための関数です。

Shader 的 Lighting 函数，使用 Surface Shader 的 Lighting 函数。

```
// 使用 Surface Shader 的 Lighting 函数
half4 Lighting<Name> (SurfaceOutput s, half3 lightDir, half atten);
// 使用 Surface Shader 的 Lighting 函数
half4 Lighting<Name> (SurfaceOutput s, half3 lightDir, half3 viewDir, half atten);
```

Unity 的 Shader 文档，请参考 <http://docs.unity3d.com/Manual/SL-SurfaceShaderLighting.html>。Unity 的 Shader 文档，请参考 <http://docs.unity3d.com/Manual/SL-SurfaceShaderLightingExamples.html>。

17.2.3 Shader 的 Pass

Shader 的 Pass 函数，使用 optional params 参数。Unity 的 Shader 文档，请参考 <http://docs.unity3d.com/Manual/SL-SurfaceShaders.html>。

- Shader 的 Pass 函数，使用 optional params 参数。
 vertex:VertexFunction 顶点函数。
 finalcolor:ColorFunction 最终颜色函数。
 Unity 的 Shader 文档，请参考 <http://docs.unity3d.com/Manual/SL-SurfaceShaders.html>。
- Shader 的 Pass 函数，使用 optional params 参数。
 addshadow 添加阴影。
 Pass 函数 Unity 的 Shader 文档，请参考 <http://docs.unity3d.com/Manual/SL-SurfaceShaders.html>。
 9.4 版本，Unity 的 Shader 文档，请参考 <http://docs.unity3d.com/Manual/SL-SurfaceShaders.html>。
 11.3.3 版本，Unity 的 Shader 文档，请参考 <http://docs.unity3d.com/Manual/SL-SurfaceShaders.html>。
 fullforwardshadows 完整的前向阴影。
 Unity 的 Shader 文档，请参考 <http://docs.unity3d.com/Manual/SL-SurfaceShaders.html>。
 noshadow 不添加阴影。
- Shader 的 Pass 函数，使用 optional params 参数。
 alpha 透明度。
 alphatest 透明度测试。
 alphatest:VariableName 透明度测试的变量名。
 Unity 的 Shader 文档，请参考 <http://docs.unity3d.com/Manual/SL-SurfaceShaders.html>。
 addshadow 添加阴影。
 Pass 函数 Unity 的 Shader 文档，请参考 <http://docs.unity3d.com/Manual/SL-SurfaceShaders.html>。
- Shader 的 Pass 函数，使用 optional params 参数。
 noambient 不添加环境光。
 Unity 的 Shader 文档，请参考 <http://docs.unity3d.com/Manual/SL-SurfaceShaders.html>。
 light probe 光照探针。
 novertexlights 不添加顶点光。
 Unity 的 Shader 文档，请参考 <http://docs.unity3d.com/Manual/SL-SurfaceShaders.html>。
 noforwardadd 不添加前向阴影。
 Pass 函数 Unity 的 Shader 文档，请参考 <http://docs.unity3d.com/Manual/SL-SurfaceShaders.html>。
 SH 函数 Unity 的 Shader 文档，请参考 <http://docs.unity3d.com/Manual/SL-SurfaceShaders.html>。
 nolightmap 不添加光照贴图。
 nofog 不添加雾。
- Shader 的 Pass 函数，使用 optional params 参数。
 Unity 的 Shader 文档，请参考 <http://docs.unity3d.com/Manual/SL-SurfaceShaders.html>。
 Pass 函数 Unity 的 Shader 文档，请参考 <http://docs.unity3d.com/Manual/SL-SurfaceShaders.html>。

exclude_path:deferred **exclude_path:forward** **exclude_path:prepass** **Unity**

/ /
/ /
17.6

17.3 变量

本章主要介绍在 Shader 中如何定义和使用变量。在 Shader 中，变量可以分为全局变量和局部变量。全局变量可以在整个 Shader 中被访问，而局部变量只能在定义它的函数或代码块中被访问。本章将详细介绍如何定义和使用这些变量。

在 Shader 中，变量可以分为全局变量和局部变量。全局变量可以在整个 Shader 中被访问，而局部变量只能在定义它的函数或代码块中被访问。本章将详细介绍如何定义和使用这些变量。

17.3.1 全局变量 Input

Input 是 Shader 中用于定义全局变量的关键字。它通常用于定义纹理坐标、法线、颜色等。在 Unity 5 中，Input 变量通常用于定义纹理坐标、法线、颜色等。在 Chapter17-BumpedDiffuse 中，Input 变量用于定义纹理坐标 uv_MainTex 和 uv_BumpMap。在“uv”和“uv2”变量中，uv_MainTex 和 uv_BumpMap 是 Input 变量，用于定义纹理坐标。在 float2 变量 uv_MainTex 中，uv_MainTex 是 Input 变量，用于定义纹理坐标。

表 17.1

变量名	数据类型
float3 viewDir	视图方向向量
COLOR 颜色 float4	颜色值
float4 screenPos	屏幕位置

変数	説明
float3 worldPos	オブジェクトの位置
float3 worldRefl	オブジェクトの反射ベクトルo.Normal
float3 worldRefl; INTERNAL_DATA	オブジェクトの反射ベクトルo.NormalをUnityのWorldReflectionVector(IN, o.Normal)で計算する
float3 worldNormal	オブジェクトの法線ベクトルo.Normal
float3 worldNormal; INTERNAL_DATA	オブジェクトの法線ベクトルo.NormalをUnityのWorldNormalVector(IN, o.Normal)で計算する

オブジェクトの位置、法線、反射ベクトル、色、テクスチャ、Input、Unity、WorldReflectionVector、WorldNormalVector、Input、half fog、Lighting.cginc

17.3.2 SurfaceOutput

Input、SurfaceOutput、SurfaceOutputStandard、SurfaceOutputStandardSpecular、Input、SurfaceOutput、Lighting.cginc

```

struct SurfaceOutput {
    fixed3 Albedo;
    fixed3 Normal;
    fixed3 Emission;
    half Specular;
    fixed Gloss;
    fixed Alpha;
};

```

SurfaceOutputStandard
SurfaceOutputStandardSpecular
 UnityPBSLighting.cginc

```

struct SurfaceOutputStandard
{
    fixed3 Albedo;           // base (diffuse or specular) color
    fixed3 Normal;           // tangent space normal, if written
    half3 Emission;
    half Metallic;           // 0=non-metal, 1=metal
    half Smoothness;         // 0=rough, 1=smooth
    half Occlusion;          // occlusion (default 1)
    fixed Alpha;             // alpha for transparencies
};

struct SurfaceOutputStandardSpecular
{
    fixed3 Albedo;           // diffuse color
    fixed3 Specular;         // specular color
    fixed3 Normal;           // tangent space normal, if written
    half3 Emission;
    half Smoothness;         // 0=rough, 1=smooth
    half Occlusion;          // occlusion (default 1)
    fixed Alpha;             // alpha for transparencies
};

```

Unity
 Unity 5 **Lambert**
BlinnPhong Unity 5 **Standard**
StandardSpecular
SurfaceOutput
Standard **StandardSpecular**

17.4 Unity□□□□□

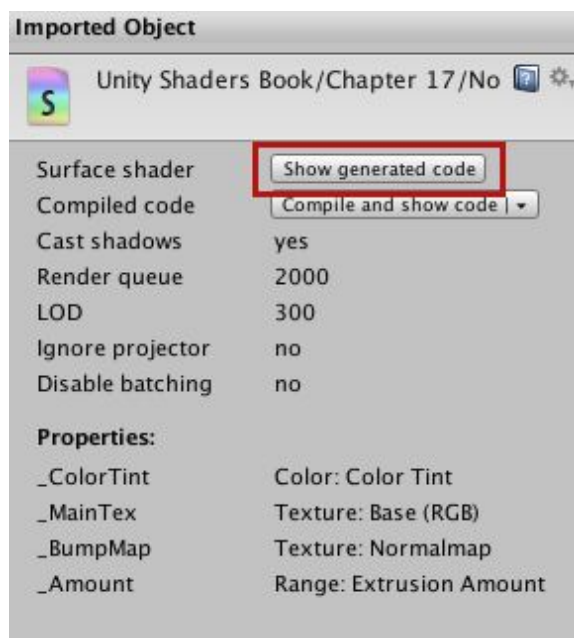
Unity
/

```

    Pass[0] = new UnityPass(Renderer.RenderQueue.PASS_FORWARD / LightMode.ForwardBase);
    Pass[1] = new UnityPass(Renderer.RenderQueue.PASS_FORWARD_ADD / LightMode.ForwardAdd);
    Pass[2] = new UnityPass(5 / LightMode.PrePassBase);
    Pass[3] = new UnityPass(5 / LightMode.Deferred);
    Pass[4] = new UnityPass(LightMode.Meta);
    Pass[5] = new addshadow(Renderer.RenderQueue.PASS_SHADOW_CASTER / LightMode.ShadowCaster);
    Pass[6] = new UnityPass(0);
}

// "Show generated code" button
void ShowGeneratedCode() {
    // 17.2
    UnityEditor.Log("Generated code");
}

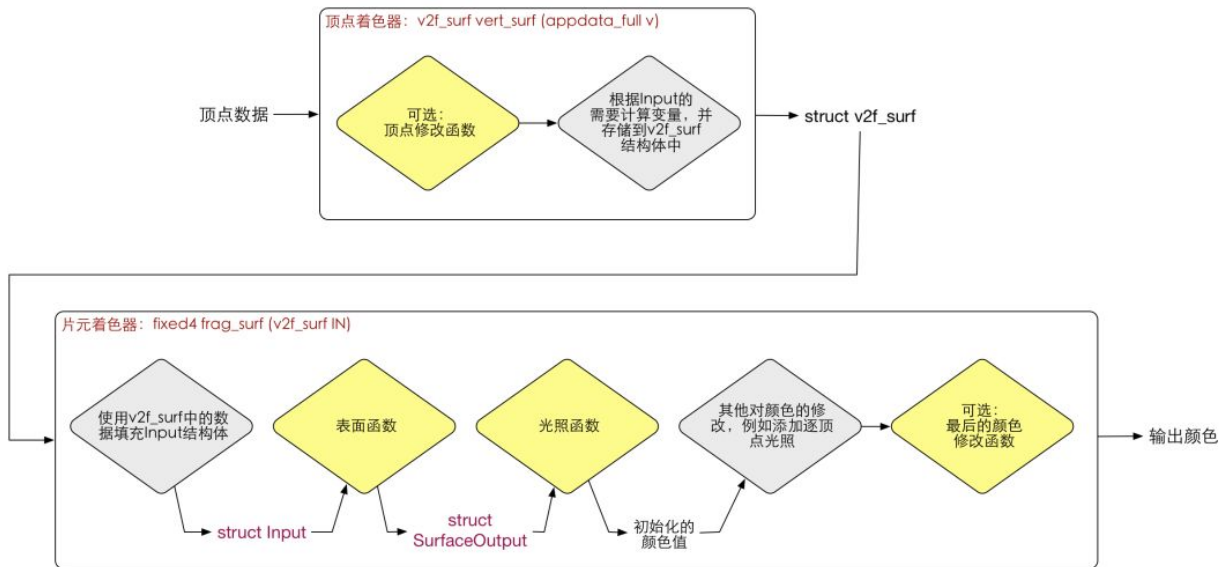
```



Unity Pass
 Unity **LightMode** **ForwardBase** Pass
 17.3 17.3 4

Unity Pass

1 CGPROGRAM ENDCG
 Input



▲ 17.3 Unity

2 Unity — v2f_surf
 Unity
 v2f_surf Input
 Unity
 v2f_surf Unity v2f_surf

3

① Unity
 Input Unity Input
 v2f_surf

② `v2f_surf` shader 的输入输出参数

③ `v2f_surf` shader 的编译选项

④ 编译选项

① `v2f_surf` shader 的 `Input` 参数

② `SurfaceOutput` 参数

③ `Lambert` 和 `BlinnPhong` 模型在 `Unity` 中的实现

④ `Unity` 中的 `Shader` 模型

⑤ `Unity` 中的 `Shader` 模型

⑥ `Pass` 模型

17.5 凹凸贴图

本章主要介绍Unity中的凹凸贴图。

本章主要介绍Scene_17_4中的凹凸贴图效果。17.4



▲图17.4 凹凸贴图效果

本章主要介绍Scene_17_4中的凹凸贴图效果。4
Chapter17-NormalExtrusion

```
Shader "Unity Shaders Book/Chapter 17/Normal Extrusion" {
    Properties {
        _ColorTint ("Color Tint", Color) = (1,1,1,1)
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _BumpMap ("Normalmap", 2D) = "bump" {}
        _Amount ("Extrusion Amount", Range(-0.5, 0.5)) = 0.1
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 300

        CGPROGRAM

        // surf - which surface function.
        // CustomLambert - which lighting model to use.
        // vertex:myvert - use custom vertex modification function.
        // finalcolor:mycolor - use custom final color modification function.
        // addshadow - generate a shadow caster pass. Because we modify the vertex position,
        // the shader needs special shadows handling.
        // exclude_path:deferred/exclude_path:prepass - do not generate passes for
        //deferred/legacy deferred rendering path.
        // nometa - do not generate a "meta" pass (that's used by lightmapping & dynamic
        //global illumination to extract surface information).
        #pragma surface surf CustomLambert vertex:myvert finalcolor:mycolor addshadow
        exclude_path:deferred exclude_path:prepass nometa
        #pragma target 3.0

        fixed4 _ColorTint;
        sampler2D _MainTex;
        sampler2D _BumpMap;
        half _Amount;

        struct Input {
            float2 uv_MainTex;
            float2 uv_BumpMap;
        };

        void myvert (inout appdata_full v) {
            v.vertex.xyz += v.normal * _Amount;
        }
    }
}
```


2

```
// Surface shader code generated based on:  
// vertex modifier: 'myvert'  
// writes to per-pixel normal: YES  
// writes to emission: no  
// needs world space reflection vector: no  
// needs world space normal vector: no  
// needs screen space position: no  
// needs world space position: no  
// needs view direction: no  
// needs world space view direction: no  
// needs world space position for lighting: no  
// needs world space view direction for lighting: no  
// needs world space view direction for lightmaps: no  
// needs vertex color: no  
// needs VFACE: no  
// passes tangent-to-world matrix to pixel shader: YES  
// reads from normal: no  
// 2 texcoords actually used  
// float2 _MainTex  
// float2 _BumpMap
```

Unity

3Unity

```
#define INTERNAL_DATA half3 internalSurfaceTtoW0; half3 internalSurfaceTtoW1; half3 internalSurfaceTtoW2;  
#define WorldReflectionVector(data,normal) reflect (data.worldRefl, half3(dot(data.internalSurfaceTtoW0,normal), dot(data.internalSurfaceTtoW1,normal), dot(data.internalSurfaceTtoW2,normal)))  
#define WorldNormalVector(data,normal) fixed3(dot(data.internalSurfaceTtoW0,normal), dot(data.internalSurfaceTtoW1,normal), dot(data.internalSurfaceTtoW2,normal))
```

Input
17.3.1

4UnityCGPass

5Unityv2f_surfUnity#ifdefSH

```
// vertex-to-fragment interpolation data  
// no lightmaps:  
#ifdef LIGHTMAP_OFF  
struct v2f_surf {  
    float4 pos : SV_POSITION;  
    float4 pack0 : TEXCOORD0; // _MainTex _BumpMap  
    fixed3 tSpace0 : TEXCOORD1;  
    fixed3 tSpace1 : TEXCOORD2;  
    fixed3 tSpace2 : TEXCOORD3;  
    fixed3 vlight : TEXCOORD4; // ambient/SH/vertexlights  
    SHADOW_COORDS(5)  
    #if SHADER_TARGET >= 30  
    float4 lmap : TEXCOORD6;  
    #endif  
};  
#endif  
// with lightmaps:  
#ifndef LIGHTMAP_OFF  
struct v2f_surf {  
    float4 pos : SV_POSITION;  
    float4 pack0 : TEXCOORD0; // _MainTex _BumpMap  
    fixed3 tSpace0 : TEXCOORD1;  
    fixed3 tSpace1 : TEXCOORD2;  
    fixed3 tSpace2 : TEXCOORD3;  
    float4 lmap : TEXCOORD4;
```

```
SHADOW_COORDS(5)
};
#endif
```

pack0
tSpace0 tSpace1 tSpace2
Unity SH vlight

6 Unity

```
// vertex shader
v2f_surf vert_surf (appdata_full v) {
v2f_surf o;
UNITY_INITIALIZE_OUTPUT(v2f_surf,o);
myvert (v);
```

Input
Input

```
void vert(inout appdata_full v, out Input o)
```

v2f_surf MVP TRANSFORM_TEX
o.pack0.xy zw
o.tSpace0
o.tSpace1 o.tSpace2
o.lmap.xy
o.lmap.zw SH o.vlight
4 o.vlight

```
TRANSFER_SHADOW(o); // pass shadow coordinates to pixel shader
return o;
}
```

7 Pass Unity Unity v2f_surf Input

```
// fragment shader
fixed4 frag_surf (v2f_surf IN) : SV_Target {
// prepare and unpack data
Input surfIN;
UNITY_INITIALIZE_OUTPUT(Input,surfIN);
surfIN.uv_MainTex = IN.pack0.xy;
surfIN.uv_BumpMap = IN.pack0.zw;
```

Unity SurfaceOutput

```
#ifdef UNITY_COMPILER_HLSL
SurfaceOutput o = (SurfaceOutput)0;
#else
SurfaceOutput o;
#endif
o.Albedo = 0.0;
o.Emission = 0.0;
o.Specular = 0.0;
o.Alpha = 0.0;
o.Gloss = 0.0;

// call surface function
```

```
surf (surfIN, o);
```

Unity#ifdef HLSLSurfaceOutput
DirectXUnitysurf

Unity

```
// compute lighting & shadowing factor  
UNITY_LIGHT_ATTENUATION(atten, IN, worldPos)  
fixed4 c = 0;  
fixed3 worldN;  
worldN.x = dot(IN.tSpace0.xyz, o.Normal);  
worldN.y = dot(IN.tSpace1.xyz, o.Normal);  
worldN.z = dot(IN.tSpace2.xyz, o.Normal);  
o.Normal = worldN;
```

c 0Unity

```
#ifdef LIGHTMAP_OFF  
c.rgb += o.Albedo * IN.vlight;  
#endif // LIGHTMAP_OFF
```

Unity
Unity

```
// realtime lighting: call lighting function  
#ifdef LIGHTMAP_OFF  
c += LightingCustomLambert (o, lightDir, atten);  
#else  
c.a = o.Alpha;  
#endif
```

Unityc Unity
c

Unityc

```
mycolor (surfIN, o, c);  
UNITY_OPAQUE_ALPHA(c.a);  
return c;  
}
```

UnityUNITY_OPAQUE_ALPHAUnityCG.cginc
1.0
keepalpha

ForwardBase PassForwardAdd PassForwardBase Pass
UnityPass

PassShadowCaster PassPass
11.3.315.1Pass
V2F_SHADOW_CASTERTRANSFER_SHADOW_CASTER_NORMALOFFSET
SHADOW_CASTER_FRAGMENT

17.6 Surface Shader

[illegible]

Unity /
/

```
#####  
#####  
##### Shader ##### Diffuse Bumped Specular #####  
##### Unity ##### Mobile/Diffuse  
Mobile/Bumped Specular ##### Shader #####  
Pass #####  
####
```

10.2.2

/

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □

- **Unity**
- /
- /

18 物理着色

物理着色器通常指 Lambert、Phong、Blinn-Phong 等模型，它们通过数学公式来模拟光线的反射和折射，从而实现逼真的视觉效果。

物理着色（**Physically Based Shading, PBS**）是一种基于物理原理的着色方法，它通过模拟光线的传播和物体表面的物理特性来实现更真实的渲染效果。PBS 模型通常包含材质属性（如颜色、粗糙度、金属度等）和光照模型（如环境光遮蔽、全局光照等）。Unity 2012 版本引入了 PBS 模型，使得开发者可以更轻松地实现物理着色效果。Unity 5 版本进一步改进了 PBS 模型，使其更加稳定和高效。Standard Shader 是 Unity 5 中默认使用的着色器，它支持 PBS 模型，并提供了丰富的材质属性供开发者选择。PBS 模型在渲染过程中需要考虑更多的物理因素，如光线的反射、折射、吸收等，因此其计算复杂度通常高于传统的着色模型。PBS 模型在电影和游戏行业中得到了广泛的应用，因为它能够提供更加逼真和自然的视觉效果。

Unity 5 版本引入了 PBS 模型，使得开发者可以更轻松地实现物理着色效果。Unity 5 版本进一步改进了 PBS 模型，使其更加稳定和高效。Standard Shader 是 Unity 5 中默认使用的着色器，它支持 PBS 模型，并提供了丰富的材质属性供开发者选择。PBS 模型在渲染过程中需要考虑更多的物理因素，如光线的反射、折射、吸收等，因此其计算复杂度通常高于传统的着色模型。PBS 模型在电影和游戏行业中得到了广泛的应用，因为它能够提供更加逼真和自然的视觉效果。

PBS 模型在渲染过程中需要考虑更多的物理因素，如光线的反射、折射、吸收等，因此其计算复杂度通常高于传统的着色模型。PBS 模型在电影和游戏行业中得到了广泛的应用，因为它能够提供更加逼真和自然的视觉效果。Unity 5 版本进一步改进了 PBS 模型，使其更加稳定和高效。Standard Shader 是 Unity 5 中默认使用的着色器，它支持 PBS 模型，并提供了丰富的材质属性供开发者选择。PBS 模型在渲染过程中需要考虑更多的物理因素，如光线的反射、折射、吸收等，因此其计算复杂度通常高于传统的着色模型。PBS 模型在电影和游戏行业中得到了广泛的应用，因为它能够提供更加逼真和自然的视觉效果。

18.1 PBS

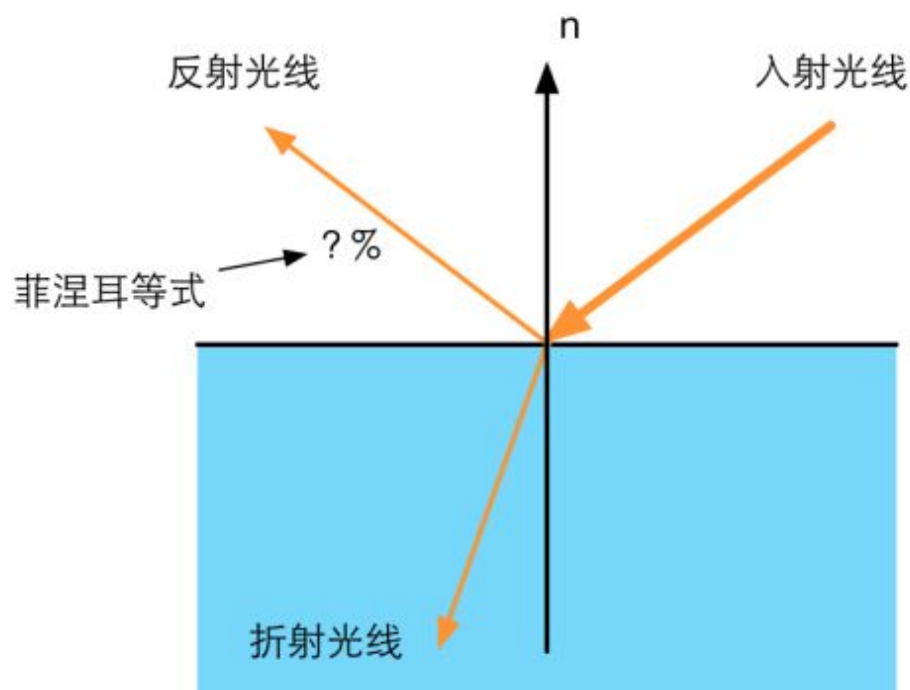
Perceptual Basis of Shading (PBS) is a shading model that is based on the physics of light transport. It is a physically-based shading model that is designed to be a good approximation of the way that light interacts with a surface. It is a shading model that is based on the physics of light transport. It is a shading model that is based on the physics of light transport. Naty Hoffman, SIGGRAPH 2013
Background: Physics and Math of Shading [1]

18.1.1

Light transport is the process by which light travels through a medium. It is the process by which light travels through a medium. It is the process by which light travels through a medium. **absorption** is the process by which light is absorbed by a medium. **scattering** is the process by which light is scattered by a medium.

Light transport is the process by which light travels through a medium. It is the process by which light travels through a medium. It is the process by which light travels through a medium. **refractive index** is the ratio of the speed of light in a vacuum to the speed of light in a medium.

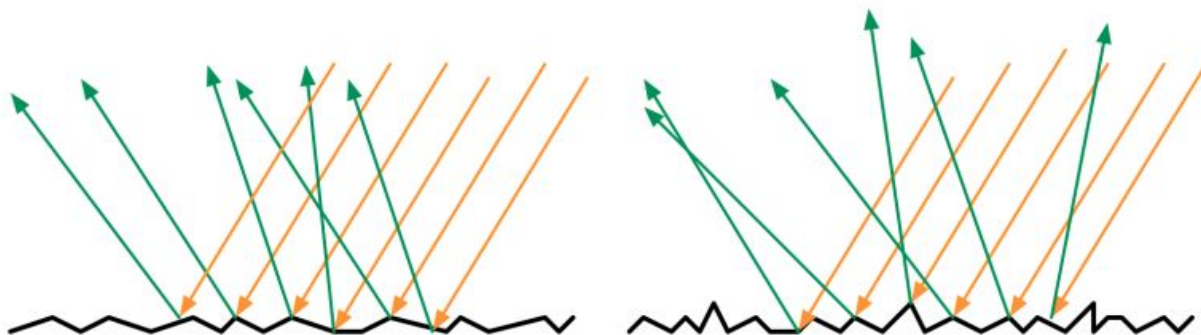
Light transport is the process by which light travels through a medium. It is the process by which light travels through a medium. It is the process by which light travels through a medium. **optically flat** is a surface that is flat to within a fraction of a wavelength of light. **Fresnel equations** are a set of equations that describe the reflection and refraction of light at an interface between two media.



▲18.1 菲涅耳等式

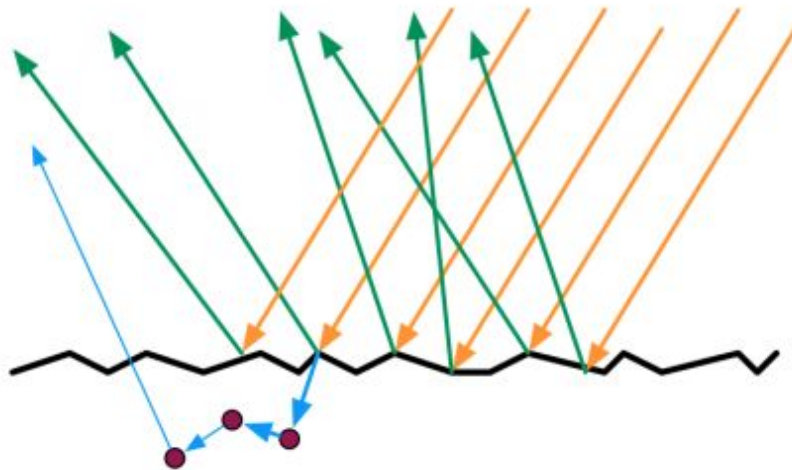
菲涅耳等式描述了光在两种介质的界面上发生反射和折射时，反射光和折射光的强度（或能量）与入射光的强度、入射角以及两种介质的折射率之间的关系。它是一个非常重要的物理公式，广泛应用于光学、电磁学等领域。

菲涅耳等式是1822年由法国物理学家菲涅耳提出的。它描述了光在两种介质的界面上发生反射和折射时，反射光和折射光的强度（或能量）与入射光的强度、入射角以及两种介质的折射率之间的关系。它是一个非常重要的物理公式，广泛应用于光学、电磁学等领域。



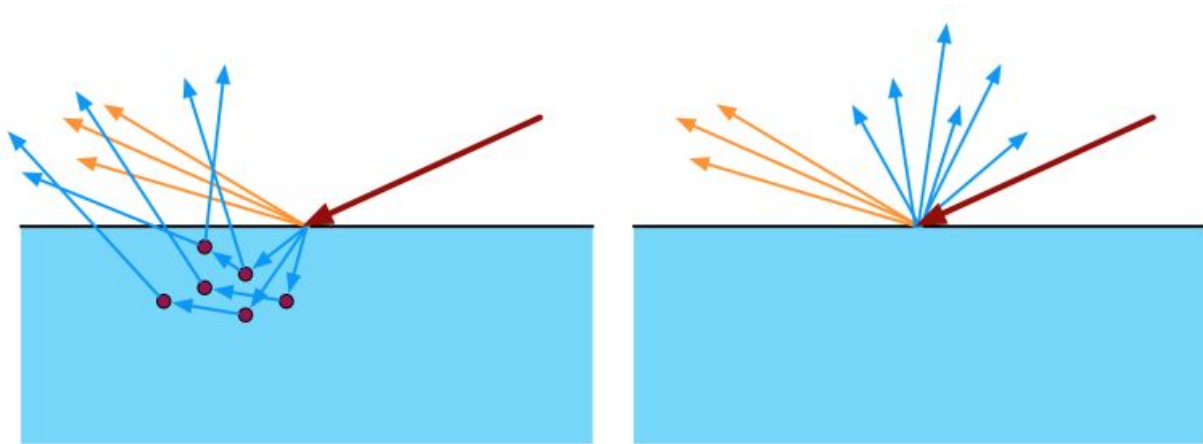
▲18.2 漫反射光与透射光

当光线照射到粗糙表面时，光线会以不同的角度反射回去，这种现象称为漫反射。在透明或半透明材料中，光线会在内部发生多次散射，这种现象称为透射光。在18.3节中，我们将讨论一种特殊的光学现象，称为“subsurface-scattered light”。



▲18.3 漫反射光与透射光

在18.4节中，我们将讨论一种特殊的光学现象，称为“subsurface-scattered light”。在18.4节中，我们将讨论一种特殊的光学现象，称为“subsurface-scattered light”。在18.4节中，我们将讨论一种特殊的光学现象，称为“subsurface-scattered light”。



▲ 18.4

[illegible]

18.1.2 □□□□□□□□BRDF□

[illegible]

radiance

L_i

L_o

shading

[illegible]

BRDF [Bidirectional Reflectance Distribution Function]

BRDF $f(I, \mathbf{v})$ I $\mathbf{v}$ **isotropic** **anisotropic** BRDF

BRDF — BRDF —
BRDF
BRDF
 I $f(I, \mathbf{v})$ $\mathbf{v}$

我们通常把描述物体表面反射特性的方程称为——**反射方程**
reflection equation

$$L_o(\mathbf{v}) = \int_{\Omega} f(\mathbf{l}, \mathbf{v}) \times L_i(\mathbf{l})(\mathbf{n} \cdot \mathbf{l}) d\omega_i$$

这个方程描述了物体表面在任意方向 $\mathbf{v}$ 上的辐射亮度 $L_o(\mathbf{v})$ 是由该表面的 BRDF $f(\mathbf{l}, \mathbf{v})$ 和入射辐射亮度 $L_i(\mathbf{l})$ 以及入射方向 $\mathbf{l}$ 与表面法线 $\mathbf{n}$ 的夹角 $\mathbf{n} \cdot \mathbf{l}$ 决定的。BRDF 是描述物体表面反射特性的函数，它表示入射辐射亮度 $L_i(\mathbf{l})$ 与入射方向 $\mathbf{l}$ 和出射方向 $\mathbf{v}$ 的关系。

我们通常把描述物体表面反射特性的方程称为——**punctual light sources**。这个方程描述了物体表面在任意方向 $\mathbf{v}$ 上的辐射亮度 $L_o(\mathbf{v})$ 是由该表面的 BRDF $f(\mathbf{l}_c, \mathbf{v})$ 和入射辐射亮度 $L_i(\mathbf{l}_c)$ 以及入射方向 $\mathbf{l}_c$ 与表面法线 $\mathbf{n}$ 的夹角 $\mathbf{n} \cdot \mathbf{l}_c$ 决定的。BRDF 是描述物体表面反射特性的函数，它表示入射辐射亮度 $L_i(\mathbf{l}_c)$ 与入射方向 $\mathbf{l}_c$ 和出射方向 $\mathbf{v}$ 的关系。

$$L_o(\mathbf{v}) = \pi f(\mathbf{l}_c, \mathbf{v}) \times c_{light}(\mathbf{n} \cdot \mathbf{l}_c)$$

这个方程描述了物体表面在任意方向 $\mathbf{v}$ 上的辐射亮度 $L_o(\mathbf{v})$ 是由该表面的 BRDF $f(\mathbf{l}_c, \mathbf{v})$ 和入射辐射亮度 $L_i(\mathbf{l}_c)$ 以及入射方向 $\mathbf{l}_c$ 与表面法线 $\mathbf{n}$ 的夹角 $\mathbf{n} \cdot \mathbf{l}_c$ 决定的。BRDF 是描述物体表面反射特性的函数，它表示入射辐射亮度 $L_i(\mathbf{l}_c)$ 与入射方向 $\mathbf{l}_c$ 和出射方向 $\mathbf{v}$ 的关系。

我们通常把描述物体表面反射特性的方程称为——BRDF。这个方程描述了物体表面在任意方向 $\mathbf{v}$ 上的辐射亮度 $L_o(\mathbf{v})$ 是由该表面的 BRDF $f(\mathbf{l}_c, \mathbf{v})$ 和入射辐射亮度 $L_i(\mathbf{l}_c)$ 以及入射方向 $\mathbf{l}_c$ 与表面法线 $\mathbf{n}$ 的夹角 $\mathbf{n} \cdot \mathbf{l}_c$ 决定的。BRDF 是描述物体表面反射特性的函数，它表示入射辐射亮度 $L_i(\mathbf{l}_c)$ 与入射方向 $\mathbf{l}_c$ 和出射方向 $\mathbf{v}$ 的关系。

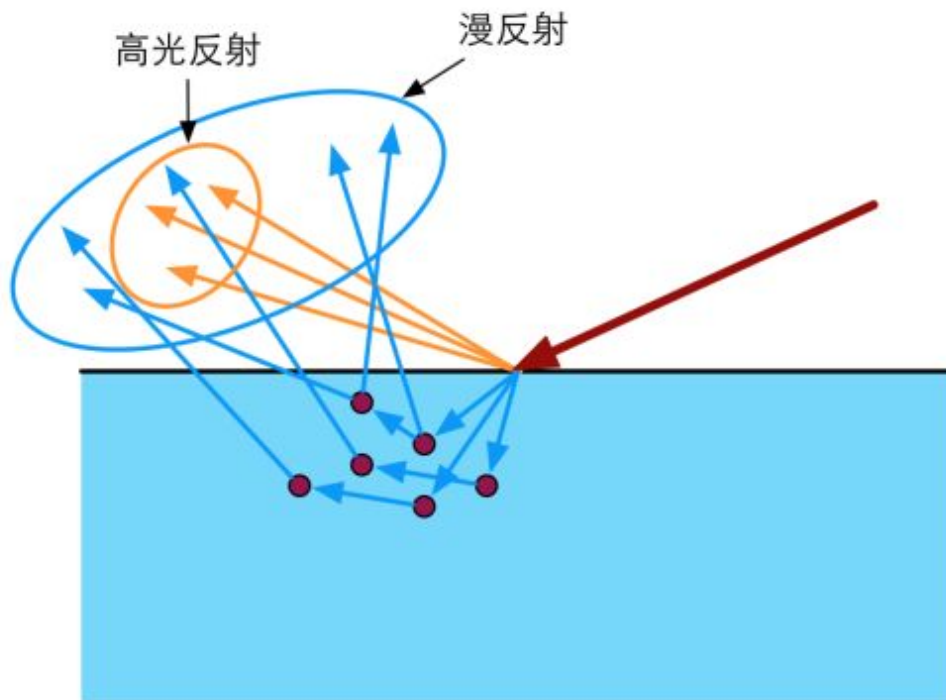
我们通常把描述物体表面反射特性的方程称为——BRDF。这个方程描述了物体表面在任意方向 $\mathbf{v}$ 上的辐射亮度 $L_o(\mathbf{v})$ 是由该表面的 BRDF $f(\mathbf{l}_c, \mathbf{v})$ 和入射辐射亮度 $L_i(\mathbf{l}_c)$ 以及入射方向 $\mathbf{l}_c$ 与表面法线 $\mathbf{n}$ 的夹角 $\mathbf{n} \cdot \mathbf{l}_c$ 决定的。BRDF 是描述物体表面反射特性的函数，它表示入射辐射亮度 $L_i(\mathbf{l}_c)$ 与入射方向 $\mathbf{l}_c$ 和出射方向 $\mathbf{v}$ 的关系。

$$f(l, v) = f(v, l)$$

能量守恒

$$\forall l, \int_{\Omega} f(l, v)(n \cdot l) d\omega_o \leq 1$$

BRDF 能量守恒
BRDF 能量守恒——能量守恒
specular term 漫反射 term
18.5



▲18.5 BRDF 能量守恒

18.1.3 能量守恒

Lambertian BRDF

$$f_{Lambert}(\mathbf{l}, \mathbf{v}) = \frac{c_{diff}}{\pi}$$

```

    vec c_diff = vec(0.5, 0.5, 0.5); // diffuse color
    float refl = 0.7; // Lambert
    vec n = vec(0, 0, 1); // normal
    vec I = vec(1, 1, 1); // incident light
    float BRDF = refl * dot(n, I); // BRDF
    vec R = BRDF * c_diff + (1 - BRDF) * c_spec; // final color
    vec C = R * I; // final color

```

$$L_{diff} = \frac{\mathbf{c}_{diff}}{\pi} \times L_i(\mathbf{l})(\mathbf{n} \cdot \mathbf{l})$$

Lambert
Disney BRDF[2]

$$f_{diff}(\mathbf{l}, \mathbf{v}) = \frac{baseColor}{\pi} (1 + (F_{D90} - 1)(1 - \mathbf{n} \cdot \mathbf{l})^5)(1 + (F_{D90} - 1)(1 - \mathbf{n} \cdot \mathbf{v})^5)$$

111

$$F_{D90} = 0.5 + 2roughness(\mathbf{h} \cdot \mathbf{l})^2$$

[Disney] [baseColor] [roughness]
 [glancing angles]
 Unity 5

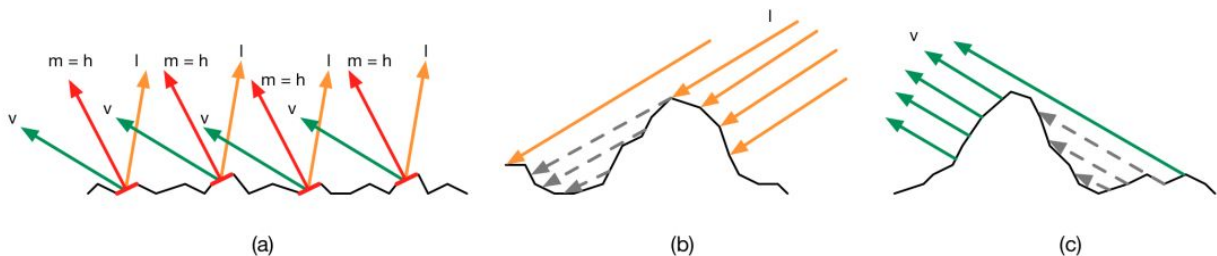
18.1.4 ☐☐☐☐☐

Everything is Shiny Shader BDRF microfacet theory

入射方向 n 表面法線 m 反射方向 n 入射光強度
 入射光強度BRDF 入射光強度 I 入射光強度 v 入射光強度
 入射光強度 m 入射光強度 I 入射光強度 v 入射光強度
 m 入射光強度 I 入射光強度 v 入射光強度 h half-angle vector
 half vector 18.6 a

BRDF

$$f_{spec}(\boldsymbol{l}, \boldsymbol{v}) = \frac{F(\boldsymbol{l}, \boldsymbol{h})G(\boldsymbol{l}, \boldsymbol{v}, \boldsymbol{h})D(\boldsymbol{h})}{4(\boldsymbol{n} \cdot \boldsymbol{l})(\boldsymbol{n} \cdot \boldsymbol{v})}$$



▲18.6a $m = h$ I v BRDF b
 a I c a
 v

Torrance-Sparrow [5] normal distribution function NDF $m =$

$\mathbf{h}$ 是半向量， $\mathbf{l}$ 是入射光方向， $\mathbf{v}$ 是视点方向， $G(\mathbf{l}, \mathbf{v}, \mathbf{h})$ 是遮蔽-阴影函数（shadowing-masking function）， $\mathbf{m} = \mathbf{h}$ 是微表面法线， $F(\mathbf{l}, \mathbf{h})$ 是菲涅尔反射率（Fresnel reflectance）。Everything has Fresnel。公式为： $4(n \cdot \mathbf{l})(n \cdot \mathbf{v})$

BRDF 模型包括 Blinn-Phong [7] 和 $\mathbf{D}(\mathbf{h})$

$$D_{blinn}(\mathbf{h}) = (\mathbf{n} \cdot \mathbf{h})^{gloss}$$

Blinn-Phong 模型，GGX [3]，Beckmann [4]， $G(\mathbf{l}, \mathbf{v}, \mathbf{h})$ ，Smith [6]

BRDF 模型，BRDF 模型，BRDF 模型

Unity 5

18.1.5 Unity 的 PBS

Unity 5 的 PBS 模型，Disney BRDF [2]，Shader，UnityStandardBRDF.cginc

Unity 5 的 PBS 模型，GGX，Blinn-Phong， $\mathbf{D}(\mathbf{h})$ ， $G(\mathbf{l}, \mathbf{v}, \mathbf{h})$ ，Unity 5.2

Phong shader is implemented in Unity 5.3 as a variation of GGX

Unity uses a BRDF

$$f_{diff}(\mathbf{l}, \mathbf{v}) = \frac{baseColor}{\pi} (1 + (F_{D90} - 1)(1 - \mathbf{n} \cdot \mathbf{l})^5)(1 + (F_{D90} - 1)(1 - \mathbf{n} \cdot \mathbf{v})^5)$$

where

$$F_{D90} = 0.5 + 2roughness(\mathbf{h} \cdot \mathbf{l})^2$$

GGX is a variation of Blinn-Phong
UnityStandardBRDF.cginc

Unity uses $D(\mathbf{h})$ GGX

$$D_{GGX} = \frac{\alpha^2}{\pi((\alpha^2 - 1)(\mathbf{n} \cdot \mathbf{h})^2 + 1)^2}$$

where

$$\alpha = roughness^2$$

GGX uses Smith-Schlick

$$G(\mathbf{l}, \mathbf{v}, \mathbf{h}) = \frac{1}{((\mathbf{n} \cdot \mathbf{l})(1 - k) + k)((\mathbf{n} \cdot \mathbf{v})(1 - k) + k)}$$
$$k = \frac{roughness^2}{2}$$

我们使用 $F(l, h)$ 来描述Schlick的模型[7]

$$F(l, h) = F_0 + (1 - F_0)(1 - l \cdot h)^5$$

其中 F_0 是Unity模型中的参数

我们使用Unity模型中的BRDF来描述Schlick的模型

我们使用Unity模型中的Shader来描述Unity模型中的Standard Shader

18.2 Unity 5 Standard Shader

Unity 5 Standard Shader

Unity Standard Shader workflow
Specular workflow
Standard Shader
Standard Specular setup

Standard Shader
Standard Specular setup
Unity Standard Shader

18.2.1

Standard Specular setup Shader
Unity
builtin_shaders-5.x/ DefaultResourcesExtra
Shader builtin_shaders-5.x/CGIncludes
UnityStandardXXX.cginc
PBS
18.1

18.1

ファイル名	機能
UnityPBSLighting.cginc	物理ベースレンダリングのための標準的なSpecular反射の計算とSurfaceOutputの定義
UnityStandardCore.cginc	Standard Shaderの標準的なSpecular setup、Vertex Forward Base、Frag Forward Base、Metallic Setup、Specular Setup、Vertex Output Forward Base、Fragment Common Dataの定義
UnityStandardBRDF.cginc	Unityの標準的なBRDF関数の定義、BRDF1_Unity_PBS、BRDF2_Unity_PBS、BRDF3_Unity_PBS、およびBRDF関数の定義
UnityStandardInput.cginc	Standard Shaderの標準的なshader関数の定義、Vertex Input、TexCoords、Albedo、Occlusion、Specular Glossの定義
UnityStandardUtils.cginc	Standard Shaderの標準的なUnityCG.cginc関数の定義
UnityStandardConfig.cginc	Standard Shaderの標準的なUnity Standard Simple、Blinn-Phong、GGX、BRDF関数の定義、UNITY_STANDARD_SIMPLE、UNITY_BRDF_GGXの定義
UnityStandardMeta.cginc	Standard Shaderの標準的なLight Mode、Meta Passの定義、Light Mode、Meta Passの定義

名前	説明
UnityStandardShadow.cginc	標準 Shader の "LightMode" と "ShadowCaster" Pass を定義する。/ 標準 Shader の "LightMode" と "ShadowCaster" Pass を定義する。
UnityGlobalIllumination.cginc	標準 Shader の UnityGlobalIllumination を定義する。

標準 Shader の Standard.shader と StandardSpecular.shader は、Unity の標準 Shader を定義する。—— 標準 Shader の SubShader と SubShader の定義。#pragma exclude_renderers gles GL ES を定義する。Pass と SubShader の定義。4 Pass と Pass と Pass と Pass と Pass と SubShader と Standard.shader と StandardSpecular.shader の BRDF の定義。—— 標準 Shader の MetallicSetup と SpecularSetup と MetallicSetup と SpecularSetup と UnityStandardCore.cginc 18.7 の Standard Shader の定義。

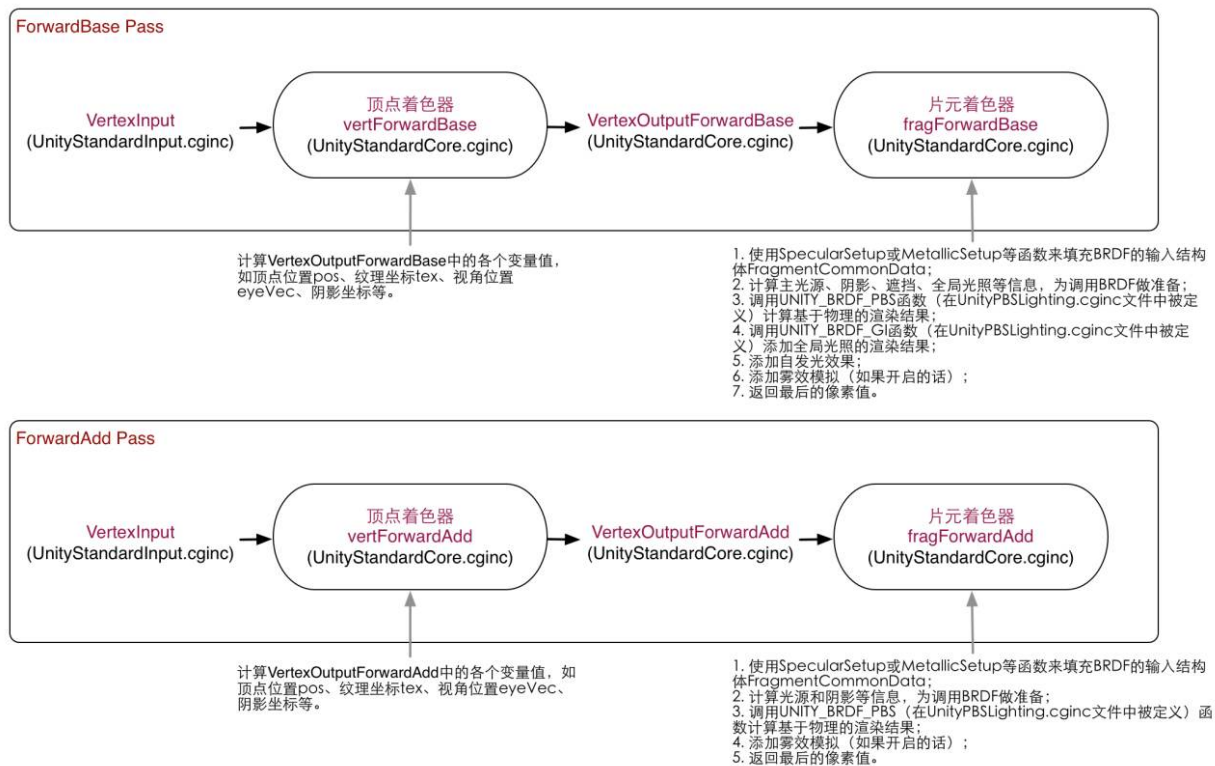
18.7 の Pass と ForwardBase Pass と ForwardAdd Pass の定義。

18.2.2 標準 Standard Shader

Unity 5 の Standard Shader の定義。

18.1 の定義。

1. 使用SpecularSetup或MetallicSetup等函数来填充BRDF的输入结构体FragmentCommonData;
 2. 计算光源、阴影、遮挡、全局光照等信息,为调用BRDF做准备;
 3. 调用UNITY_BRDF_PBS函数(在UnityPBSLighting.cginc文件中被定义)计算基于物理的渲染结果;
 4. 调用UNITY_BRDF_GI函数(在UnityPBSLighting.cginc文件中被定义)添加全局光照的渲染结果;
 5. 添加自发光效果;
 6. 添加雾效模拟(如果开启的话);
 7. 返回最后的像素值。



▲18.7 Standard ShaderPassPBSVertexOutputBaseSimple

1

-
-
-

2

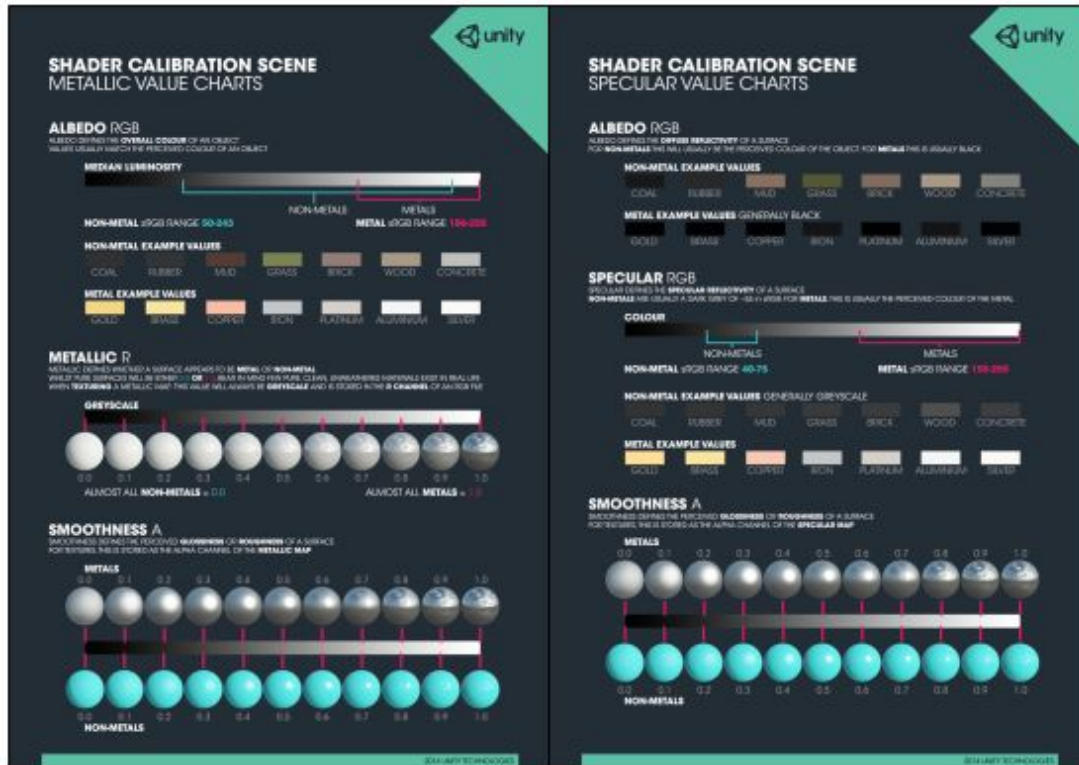
-
-
-

UnityShader Calibration Scene

<https://www.assetstore.unity3d.com/en/#!/content/25422>

 Unity 18.8

 Assets/Textures/Chapter18/Charts



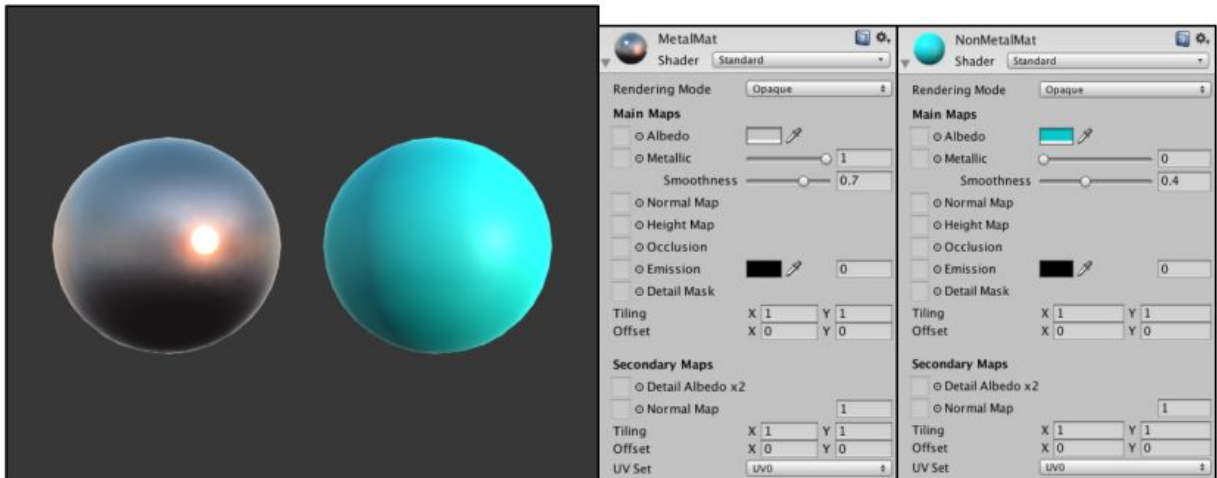
▲ 18.8 Unity

18.8

 Scene_18_2

 18.9 Edit → Project Settings → Player → Other Settings → Color Space → Linear

 18.9 18.3.4



▲18.9 金属と非金属の材質設定

金属と非金属の材質設定は、Albedo、Metallic、Smoothnessの3つのパラメータで制御されます。Unity 18.8では、Albedoは50%243、Metallicは186、255、Smoothnessは0.7で設定されています。18.9では、Albedoは50%243、Metallicは186、255、Smoothnessは0.7で設定されています。Metallicは0から1の範囲で設定され、0は非金属、1は金属を表します。Smoothnessは0.4から0.7の範囲で設定され、0.4は非金属、0.7は金属を表します。

AlbedoはSpecular、MetallicはAlbedo、SmoothnessはSpecular、MetallicはSmoothnessで設定されます。Specularは0.55、Metallicは0.4、Smoothnessは0.7で設定されています。SpecularはSpecular、MetallicはSmoothnessで設定されます。

Specular Specular RGB Specular A Smoothness

Main Maps Main Maps Secondary Maps Main Maps

Standard Shader Render Mode Standard Shader 4 Opaque Cutout Fade Transparent Opaque Albedo A Cutout Albedo A Alpha Cutoff Fade Transparent Transparent Fade

Standard Shader Unity Unity

Albedo Metallic Specular PBS Unity HDR Skybox HDR

18.3 反射の表現

この章では、Unity 5で反射表現を実現する方法について解説します。
Scene_18_3で使用するアセットは、Unity Asset Storeの**Viking Village**
(<https://www.assetstore.unity3d.com/jp/#!/content/29140>)
から取得します。UnityでPBS（Physically Based Rendering）

18.10で解説したように、Unity 5では、Edit →
Project Settings → Player → Color SpaceをLinearに変更し、18.9
で解説したように、18.3.4で解説したように



▲18.10 Unity 5で反射表現を実現する方法について解説します

この章では、Standard ShaderでUnity 5で反射表現を実現する方法について解説します

18.3.1 反射表現

この章では、Unity 5で反射表現を実現する方法について解説します。
Skyboxは、Unity 5でSkyboxをSkyboxとして設定し、
Window → Lighting → SceneでSunsetSkyboxHDRを選択し、
Skyboxで18.11で解説したように

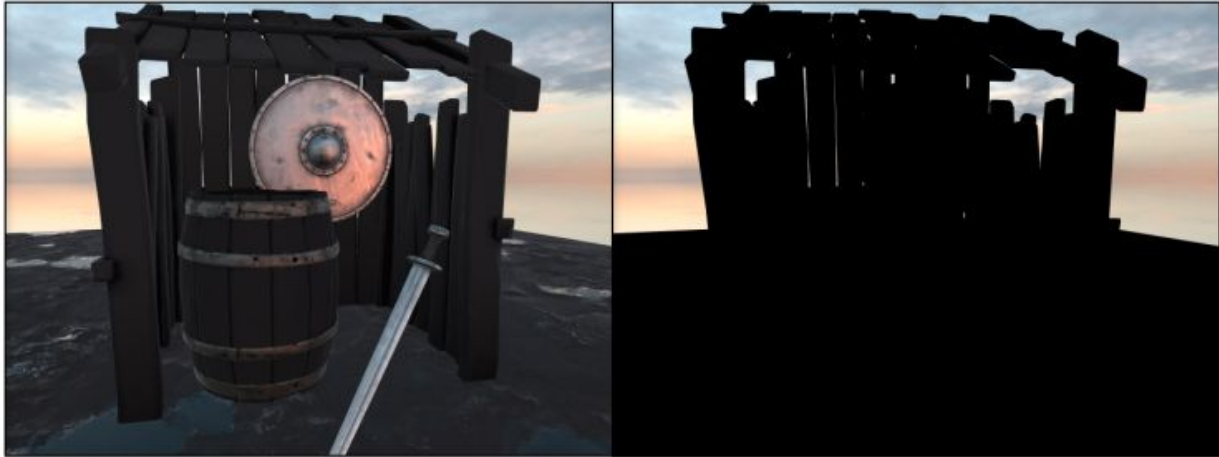
SkyboxはHDR Cubemapとして10.1で解説したように
SkyboxはHDR（High Dynamic Range）として設定します

18.4.3 HDR Skybox

18.11 Ambient Source Ambient Intensity Standard Shader 0 0 18.12



18.11 Scene



▲ 18.12 0 Standard Shader

```

    ██████████ ██████████ Reflection Source ██████████
    ██████████ Skybox ██████████ ██████████ ██████████
    ██████████ Custom ██████████ Cubemap ██████████
    Skybox ██████████ 18.12 ██████████ ██████████
    ██████████ ██████████ Unity ██████████ ForwardBase Pass ██████████
    ██████████ ██████████ ██████████ ██████████ ██████████
    ██████████ Frame Debugger ██████████ ██████████ ██████████
    ██████████ ██████████ Reflection Probes ██████████ 18.3.2 ██████████
    ██████████ Skybox ██████████ Unity ██████████ Skybox ██████████ Cubemap ██████████
    ██████████ Resolution ██████████ ██████████

```

Standard Shader Unity —

Global Illumination GI

Unity

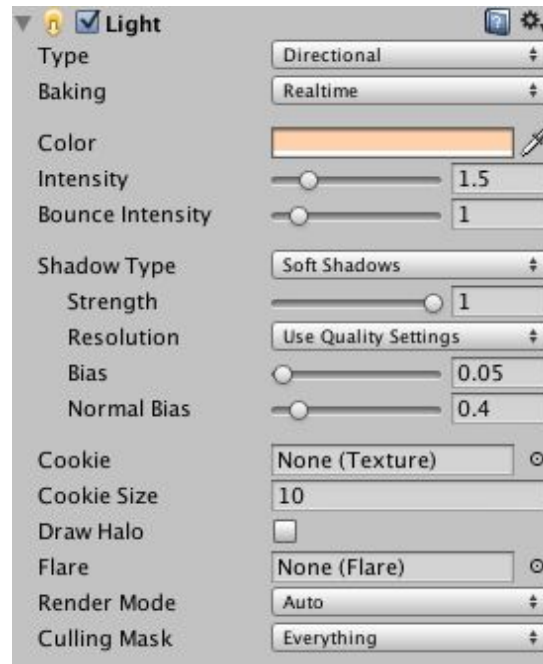
Unity 5

————PBR — Physically Based Rendering
Skybox
Skybox

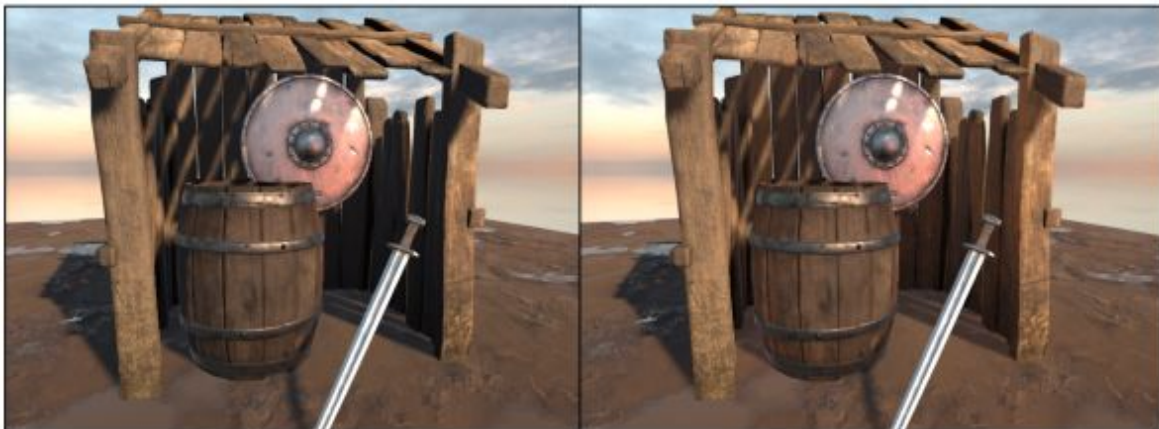
Flare Texture 18.10
18.13 18.10
Skybox

Baking Realtime
Baked Unity lightmap
Mix
Static

Unity 5
GI
18.13 **Bounce Intensity**
0
18.14 Bounce Intensity 0 8



▲18.13 光球



▲18.14 光球Bounce Intensityを0にすると、シーンが暗く、色も暗くなる。Bounce Intensityを8にすると、シーンが明るく、色も明るくなる。

光球の光の強さを調整する。18.11の光球の光の強さを調整する。Sceneの光の強さを調整する。General GIのBounce Boostの値を調整する。Bounce Intensityの値を調整する。Indirect Intensityの値を調整する。

18.3.2 光球

10.1 Cubemap
Cubemap “ ”
Cubemap Unity 5 **Reflection Probes** Light Probes
Unity
Unity Skybox
18.3.1

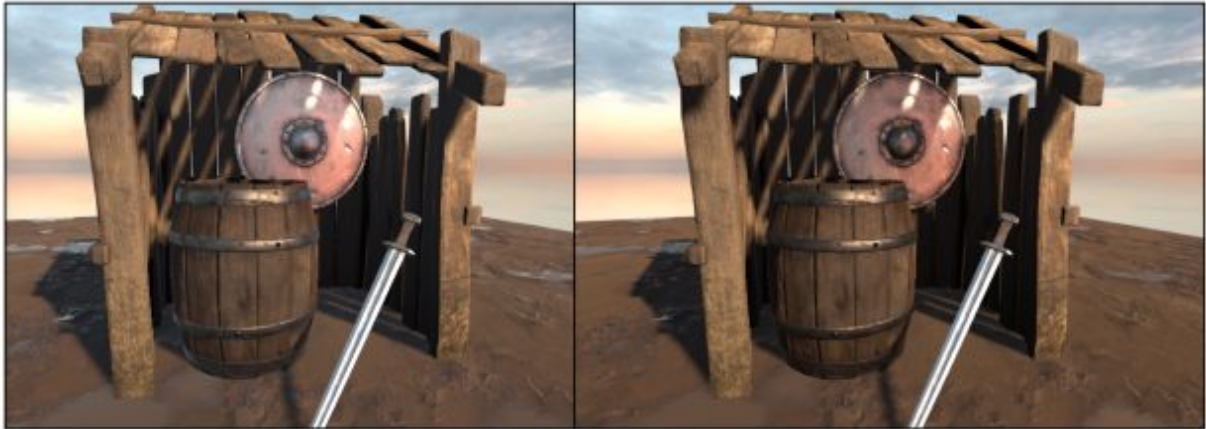
3 Baked Cubemap
Cubemap Cubemap
Reflection Probe Static
Realtime Cubemap
Unity Custom
Cubemap
Cubemap

3 Baked
Static 18.15

Unity
SM 3.0 Unity

Unity
interreflections 10.1 Cubemap

このチュートリアルでは、Unity 5のGI（Global Illumination）を、Scene_18_3_2のシーンで確認します。18.16のシーンで確認します。18.16のシーンで確認します。



▲18.15 3DシーンのGI（Global Illumination）の比較



▲18.16 3Dシーンの反射プローブ

18.16のシーンで、Reflection ProbesのSimpleモードを確認します。Mesh RendererのReflection ProbesのSimpleモードを確認します。このモードでは、反射プローブがシーンの色とテクスチャを正確に反映します。Unityのバージョン18.11から、Reflection Bouncesの値を2に設定することで、よりリアルな反射効果が得られます。

このチュートリアルでは、Cubemapの作成方法を紹介します。Unityの<http://docs.unity3d.com/Manual/ReflectionProbes.html>を参照してください。

18.3.3 Cubemap

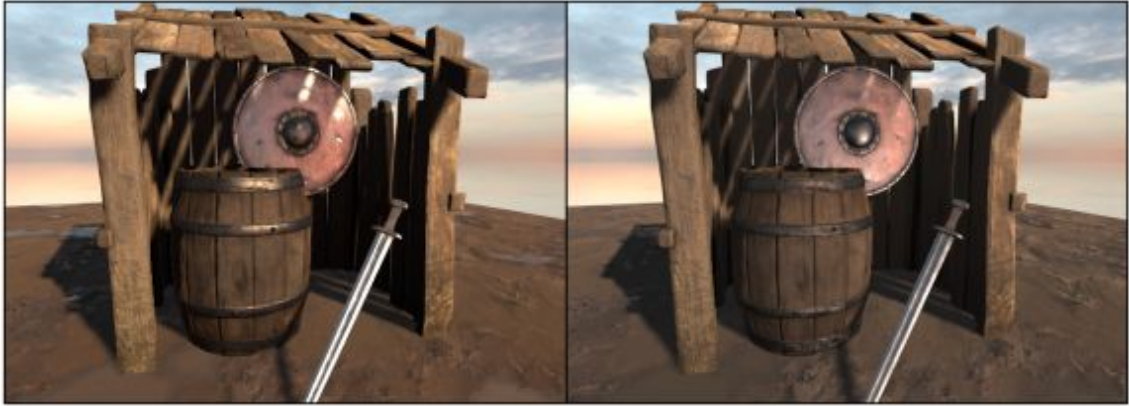
Cubemapは、6つの面から構成される立方体のテクスチャです。各面は、立方体の中心から見たときの周囲のシーンを映し出します。

Unityでは、Standard Shaderの18.2.2セクションで、Cubemapの使用方法が説明されています。Unityのプロジェクト「Viking Village」では、Specular Texture、Occlusion Texture、Normal TextureにCubemapが使用されています。

18.3.4 Color Space

このチュートリアルでは、Color Spaceの概念を説明します。Linear SpaceとGamma Spaceの違いについて説明します。Unityでは、Edit→Project Settings→Player→Other Settings→Color Spaceで、Color Spaceを設定できます。18.17では、Color Spaceの概念を説明します。

18.17では、Color Spaceの概念を説明します。Color Spaceは、色の表現方法です。Linear Spaceは、色の表現方法です。Gamma Spaceは、色の表現方法です。



▲图18.17 伽马校正前后的效果对比

在Unity中，伽马校正（Gamma Correction）是一个非常重要的概念。它用于调整图像的亮度，使其更符合人眼的视觉特性。在默认情况下，Unity会自动启用伽马校正，以确保在不同设备上渲染出的图像亮度一致。然而，在某些特殊情况下，可能需要禁用伽马校正，例如在开发某些特效或进行性能优化时。了解伽马校正的原理和应用，对于掌握Unity的渲染系统至关重要。

在Unity中，可以通过以下步骤来禁用伽马校正：

- 打开Unity Hierarchy面板，选择场景中的任何对象。
- 在Inspector面板中，找到“Rendering”属性。
- 将“Gamma”属性设置为0.0，这将禁用伽马校正。

图18.4.2展示了禁用伽马校正后的效果对比。

18.4 物理引擎

物理引擎是游戏引擎的重要组成部分，Unity 和 Standard Shader 都支持物理引擎。Unity 的物理引擎是 PhysX，而 Standard Shader 的物理引擎是 Unity 的物理引擎。Standard Shader 的物理引擎是 Unity 的物理引擎。Standard Shader 的物理引擎是 Unity 的物理引擎。Standard Shader 的物理引擎是 Unity 的物理引擎。

18.4.1 物理引擎

物理引擎是游戏引擎的重要组成部分，Unity 和 Standard Shader 都支持物理引擎。Unity 的物理引擎是 PhysX，而 Standard Shader 的物理引擎是 Unity 的物理引擎。Standard Shader 的物理引擎是 Unity 的物理引擎。Standard Shader 的物理引擎是 Unity 的物理引擎。

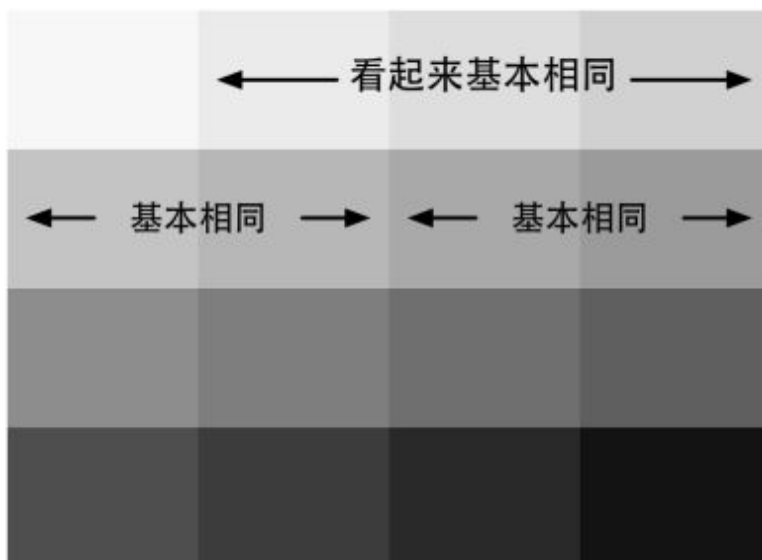
物理引擎是游戏引擎的重要组成部分，Unity 和 Standard Shader 都支持物理引擎。Unity 的物理引擎是 PhysX，而 Standard Shader 的物理引擎是 Unity 的物理引擎。Standard Shader 的物理引擎是 Unity 的物理引擎。Standard Shader 的物理引擎是 Unity 的物理引擎。

Unity 的物理引擎是 Enlighten，而 Unreal Engine 的物理引擎是 3A。Unity 的物理引擎是 Enlighten，而 Unreal Engine 的物理引擎是 3A。Unity 的物理引擎是 Enlighten，而 Unreal Engine 的物理引擎是 3A。Unity 的物理引擎是 Enlighten，而 Unreal Engine 的物理引擎是 3A。

物理引擎是游戏引擎的重要组成部分，Unity 和 Standard Shader 都支持物理引擎。Unity 的物理引擎是 PhysX，而 Standard Shader 的物理引擎是 Unity 的物理引擎。Standard Shader 的物理引擎是 Unity 的物理引擎。Standard Shader 的物理引擎是 Unity 的物理引擎。

物理引擎是游戏引擎的重要组成部分，Unity 和 Standard Shader 都支持物理引擎。Unity 的物理引擎是 PhysX，而 Standard Shader 的物理引擎是 Unity 的物理引擎。Standard Shader 的物理引擎是 Unity 的物理引擎。Standard Shader 的物理引擎是 Unity 的物理引擎。

0 1 0.5 0.5 8
 0 1 256
 18.18



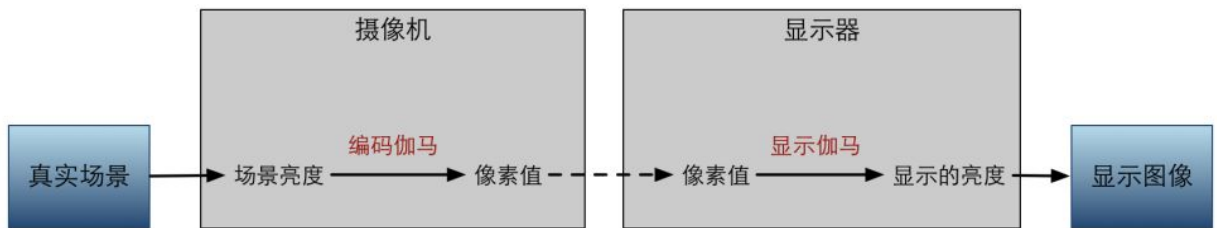
▲ 18.18 色阶

18.18
Color is Broken
 101
 102
 8 0.5 0.5
 128
 8
 0.45 0.5
 0.5 0.22

$$0.5 \approx 0.22^{0.45}$$

CRT
 Cathode Ray Tube
 CRT

在显示设备中，我们通常使用 **display gamma** 来描述 CRT 显示设备的特性。CRT 显示设备的“非线性”特性，即其显示的亮度与输入的像素值之间的关系，通常用 18.19 来描述。



▲ 18.19 伽马校正

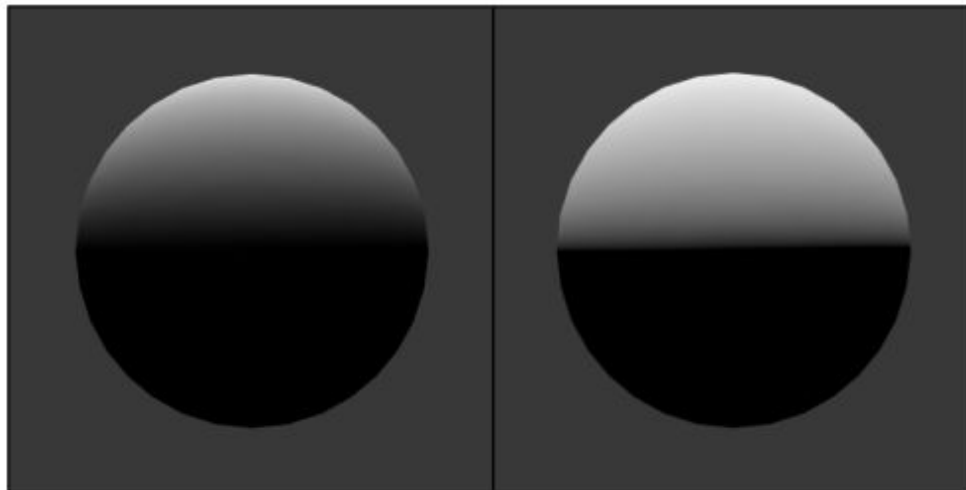
在 sRGB 颜色空间中，我们通常使用 2.2 来描述其伽马值。而 0.45 则是其逆伽马值。因此， $2.2 \times 0.45 \approx 1$ ，这确保了在 PC 上使用 sRGB 颜色空间时，颜色能够正确显示。

在 Unity 引擎中，我们通常使用 `Scene_18_4_2_a` 来描述一个场景。这个场景的伽马值通常设置为 18.20 ，以确保颜色在显示设备上能够正确呈现。

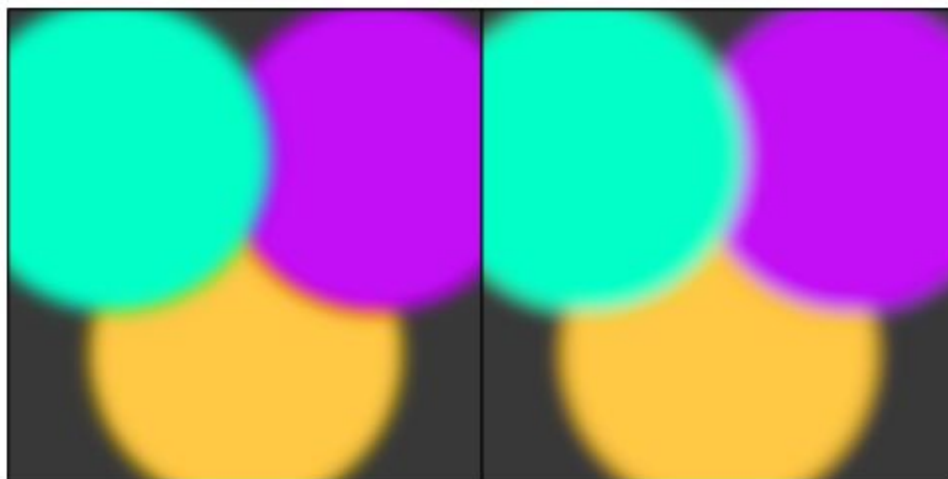
在 Unity 引擎中，我们通常使用 `Scene_18_4_2_a` 来描述一个场景。这个场景的伽马值通常设置为 18.20 ，以确保颜色在显示设备上能够正确呈现。

在 `18.20` 场景中，我们通常使用 `A` 来描述一个场景。这个场景的伽马值通常设置为 60° 。而 `B` 则通常设置为 90° 。在 Shader 中，我们通常使用 `A` 来描述一个场景。这个场景的伽马值通常设置为 $0.5, 0.5, 0.5$ 。而 `B` 则通常设置为 $1.0, 1.0, 1.0$ 。在 `18.20` 场景中，我们通常使用 `A` 来描述一个场景。这个场景的伽马值通常设置为 $1/4$ 。在 `18.20` 场景中，我们通常使用 `Unity` 来描述一个场景。这个场景的伽马值通常设置为 $1/4$ 。

Scene_18_4_2_b
 Scene_18_4_2_b3
 18.21



▲18.20



▲18.21

18.21
 18.21
 18.21

HDR 8-bit 32-bit HDR

8-bit 32-bit HDR

18.4.3 HDR

HDR HDR High Dynamic Range Low Dynamic Range LDR 8-bit 256-bit 18.4.2 HDR HDR 8-bit 32-bit 0-1 LDR Nvidia HDR tonemapping

HDR HDR 18.3 HDR Skybox Skybox HDR 1 HDR 1 HDR tonemapping

HDR HDR Bloom 12.5 Bloom Bloom HDR 1 HDR

HDRの値を1に設定する

HDRの値を1に設定する
HDRの値を1に設定する
HDRの値を1に設定する
UnityのEdit → Project Settings → Quality → Anti Aliasingの項目でHDRの項目をOnに設定する

UnityのCameraの項目でHDRの項目をOnに設定する
HDRの値を0から1に設定する
HDRの値をLDRに設定する
Unityの項目で<http://docs.unity3d.com/Manual/HDR.html>の項目を参照する

18.4.4 PBSの項目

PBSの項目は、PBSの項目をOnに設定する
PBSの項目は、PBSの項目をOnに設定する
PBSの項目は、PBSの項目をOnに設定する
PBSの項目は、PBSの項目をOnに設定する

PBSの項目は、PBSの項目をOnに設定する
PBSの項目は、PBSの項目をOnに設定する
PBSの項目は、PBSの項目をOnに設定する
PBSの項目は、PBSの項目をOnに設定する
PBSの項目は、PBSの項目をOnに設定する
PBSの項目は、PBSの項目をOnに設定する
PBSの項目は、PBSの項目をOnに設定する
PBSの項目は、PBSの項目をOnに設定する

Allegorithmic Substance Painter **Quixel Suite**

18.5 参考文献

Unity 物理光照模型 PBS 模型在 **Unity** 中的实现 global-illumination-in-unity-5 物理光照模型在 Unity 中的实现 working-with-physically-based-Shading-a-practical-approach/ physically-based-shading-in-unity-5-a-primer/ 标准 Shader 模型在 Unity 中的实现 **Viking Village Shader Calibration Scene** **Corridor Lighting Example** Unity 5 标准 Shader 模型在 Unity 5 中的实现 Unity 5 标准 Shader 模型在 Unity 5 中的实现 **The Blacksmith** Shader 模型在 Unity 5 中的实现 Shader 模型在 Unity 5 中的实现 <http://unity3d.com/cn/learn/tutorials/topics/graphics> **Unity 5** <http://unity3d.com/cn/learn/tutorials/modules/beginner/unity-5/unity5-lighting-overview> <https://unity3d.com/cn/learn/tutorials/modules/beginner/graphics/lighting-and-rendering> **Standard Shader** <http://unity3d.com/cn/learn/tutorials/modules/beginner/5-tutorials/standard-shader> Unity 标准 Shader

Unity 物理光照模型 SIGGRAPH 物理光照模型 Unite 2014 Anton Hand 物理光照模型 PBS 物理光照模型 Renaldas Zioma Erland Körner Unity 5 物理光照模型 PBS SIGGRAPH 2015 Unity 物理光照模型 **The Blacksmith**

物理光照模型 PBS 物理光照模型 SIGGRAPH 物理光照模型 SIGGRAPH 2006 物理光照模型 PBS 物理光照模型 4 2012

2015年SIGGRAPH Asia 2015 亚洲计算机图形学大会
Disney 迪士尼 BRDF 双向反射分布函数 Unity 引擎 PBR 渲染
Kostas Anagnostou 的 PBR 渲染
SIGGRAPH 2015

KlayGE 引擎 PBR 渲染
Behind the Pixels
<http://zhuanlan.zhihu.com/graphics> 3 万字
万字

18.6 □□□□

[1] Hoffman N. Background: physics and math of shading[C]//Fourth International Conference and Exhibition on Computer Graphics and Interactive Techniques, Anaheim, USA. 2013: 21-25□

[2] Burley B, Studios W D A. Physically-based shading at disney[C]//ACM SIGGRAPH. 2012: 1-7□

[3] Walter B, Marschner S R, Li H, et al. Microfacet models for refraction through rough surfaces[C]//Proceedings of the 18th Eurographics conference on Rendering Techniques. Eurographics Association, 2007: 195-206□

[4] Beckmann P, Spizzichino A. The scattering of electromagnetic waves from rough surfaces[J]. Norwood, MA, Artech House, Inc., 1987, 511 p., 1987, 1□

[5] Torrance K E, Sparrow E M. Theory for off-specular reflection from roughened surfaces[J]. JOSA, 1967, 57(9): 1105-1112□

[6] Smith B G. Geometrical shadowing of a random rough surface[J]. Antennas and Propagation, IEEE Transactions on, 1967, 15(5): 668-671□

[7] Blinn J F. Models of light reflection for computer synthesized pictures[C]//ACM SIGGRAPH Computer Graphics. ACM, 1977, 11(2): 192-198□

[8] Schlick C. An inexpensive BRDF model for physically-based rendering[C]//Computer graphics forum. 1994, 13(3): 233-246□

19 Unity 5

Unity 5 Shader
Unity 4 Shader Unity 5
Unity 4 Shader Unity 5
Shader Unity Shader

19.1 “ ”

Unity 5 Shader

```
// Unity 5 shader
//  Unity 5 shader x2
c.rgb = s.Albedo * _LightColor0.rgb * (diff * atten * 2);
```

[illegible]

Shader Unity 5 2

19.2 □□□□□□□□“□□□”

```
Unity Shader Graph  
Unity Shader Model 3.0
```

Unity 5 Unity 17
Unity /
Unity 5 Unity
Unity
Unity 5 Unity
Unity 5
Unity Shader Model 2.0
SM 2.0

Shader Model 3.0

```
#pragma target 3.0
```

[illegible]

```
#pragma surface surfaceFunction lightModel noshadow nolightmap nofog
```

19.3

```

Unity 5
4.7
Unity 5
Shader
Shader
Unity 5
CPU
Unity
CPU
Shader

```

```
// #define SCALED_NORMAL (v.normal * unity_Scale.w)
float3 worldNormal = mul((float3x3)_Object2World, SCALED_NORMAL);
```

unity_Scale.w Unity 4.x 1/ Unity 5 CPU

```

Unity Shader
unity_Scale/

```

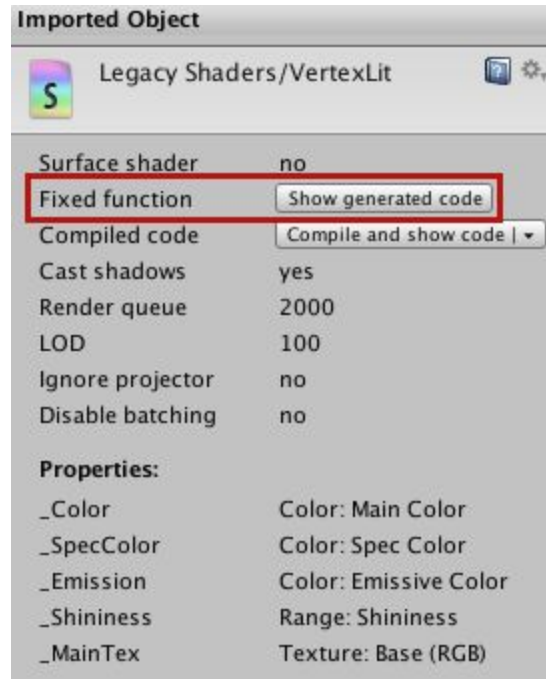
19.4 图形API的演进

在3.4.3节我们介绍了Unity如何与GPU交互。在2005年6月，Unity 1.0.1版本发布，它支持了OpenGL 1.5。在2004年，OpenGL 2.0版本发布，它支持了更高级的图形API。Unity 5.x版本支持了OpenGL 4.5，这为Unity提供了更强大的图形API。

在2004年，OpenGL 2.0版本发布，它支持了更高级的图形API。Unity 5.x版本支持了OpenGL 4.5，这为Unity提供了更强大的图形API。在2004年，OpenGL 2.0版本发布，它支持了更高级的图形API。Unity 5.x版本支持了OpenGL 4.5，这为Unity提供了更强大的图形API。

在2004年，OpenGL 2.0版本发布，它支持了更高级的图形API。Unity 5.x版本支持了OpenGL 4.5，这为Unity提供了更强大的图形API。在2004年，OpenGL 2.0版本发布，它支持了更高级的图形API。Unity 5.x版本支持了OpenGL 4.5，这为Unity提供了更强大的图形API。

Unity 5.x版本支持了OpenGL 4.5，这为Unity提供了更强大的图形API。在2004年，OpenGL 2.0版本发布，它支持了更高级的图形API。Unity 5.x版本支持了OpenGL 4.5，这为Unity提供了更强大的图形API。



▲19.1 shader를 사용하여 Unity를 사용하여 렌더링하는 방법

이 단락에서는 shader를 사용하여 Unity를 사용하여 렌더링하는 방법을 설명합니다. `TexGen`를 사용하여 렌더링하는 방법을 설명합니다. `new Material("fixed function shader string")`를 사용하여 렌더링하는 방법을 설명합니다.

이 단락에서는 Unity를 사용하여 렌더링하는 방법을 설명합니다. Unity 5를 사용하여 렌더링하는 방법을 설명합니다. Unity를 사용하여 `Aras`를 렌더링하는 방법을 설명합니다.

20

20.1 图形API与着色器

Unity Shader与OpenGL、DirectX等API的关系，以及它们在图形API中的地位。OpenGL与DirectX的比较，以及它们在图形API中的地位。OpenGL与DirectX的比较，以及它们在图形API中的地位。^[1] 关于OpenGL与DirectX的比较，以及它们在图形API中的地位。^[2] 关于OpenGL与DirectX的比较，以及它们在图形API中的地位。Milo Yip的API地址：<http://www.douban.com/doulist/1445744/>

GPU与CPU的关系，以及它们在图形API中的地位。GPU Pro与ShaderX的关系，以及它们在图形API中的地位。^[3]^[4]^[5] GPU Pro与ShaderX的关系，以及它们在图形API中的地位。^[6] ShaderX与GPU Pro的关系，以及它们在图形API中的地位。^[7] Gems与GPU Pro的关系，以及它们在图形API中的地位。<http://www.douban.com/doulist/1445745/>

Physically based rendering: From theory to implementation^[8] 关于Physically based rendering: From theory to implementation。PBS

Real-time Rendering, third Edition^[9] 关于Real-time Rendering, third Edition。

SIGGRAPH Asia, Eurographics, Symposium on Interactive 3D Graphics and Games, Ke-Sen Huang, SIGGRAPH Course, 18, Physically Based Shading in Theory and Practice, Advances in Real-Time Rendering, Epic, 2006, SIGGRAPH Course。

Game Developers Conference GDC GDC
2009 GDC China

Shadertoy
GLSL Sandbox Gallery
Shader

20.2 纹理贴图

纹理贴图是Unity Shader中非常重要的一个概念。在Unity Shader中，纹理贴图是指将一张图片（通常是PNG或JPG格式）加载到GPU中，并在渲染时根据UV坐标从该图片中取出对应的像素值。这个过程通常通过纹理采样器（Texture Sampler）来完成。在Unity Shader中，纹理贴图的使用通常涉及到纹理坐标（UV）的计算和纹理采样器的配置。在Unity Shader中，纹理贴图的使用通常涉及到纹理坐标（UV）的计算和纹理采样器的配置。

在Unity Shader中，纹理贴图的使用通常涉及到纹理坐标（UV）的计算和纹理采样器的配置。在Unity Shader中，纹理贴图的使用通常涉及到纹理坐标（UV）的计算和纹理采样器的配置。

在Unity Shader中，纹理贴图的使用通常涉及到纹理坐标（UV）的计算和纹理采样器的配置。在Unity Shader中，纹理贴图的使用通常涉及到纹理坐标（UV）的计算和纹理采样器的配置。

20.3 参考文献

[1] Shreiner D, Sellers G, Kessenich J M, et al. OpenGL programming guide: The Official guide to learning OpenGL, version 4.3[M]. Addison-Wesley, 2013. OpenGL 8

[2] Wright R S, Haemel N, Sellers G M, et al. OpenGL SuperBible: comprehensive tutorial and reference[M]. Pearson Education, 2010. OpenGL 5

[3] Fernando R, Haines E, Sweeney T. GPU gems: programming techniques, tips, and tricks for real-time graphics[J]. Dimensions, 2001, 7(4): 816. GPU 1

[4] Pharr M, Fernando R. Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation[M]. Addison-Wesley Professional, 2005. GPU 2

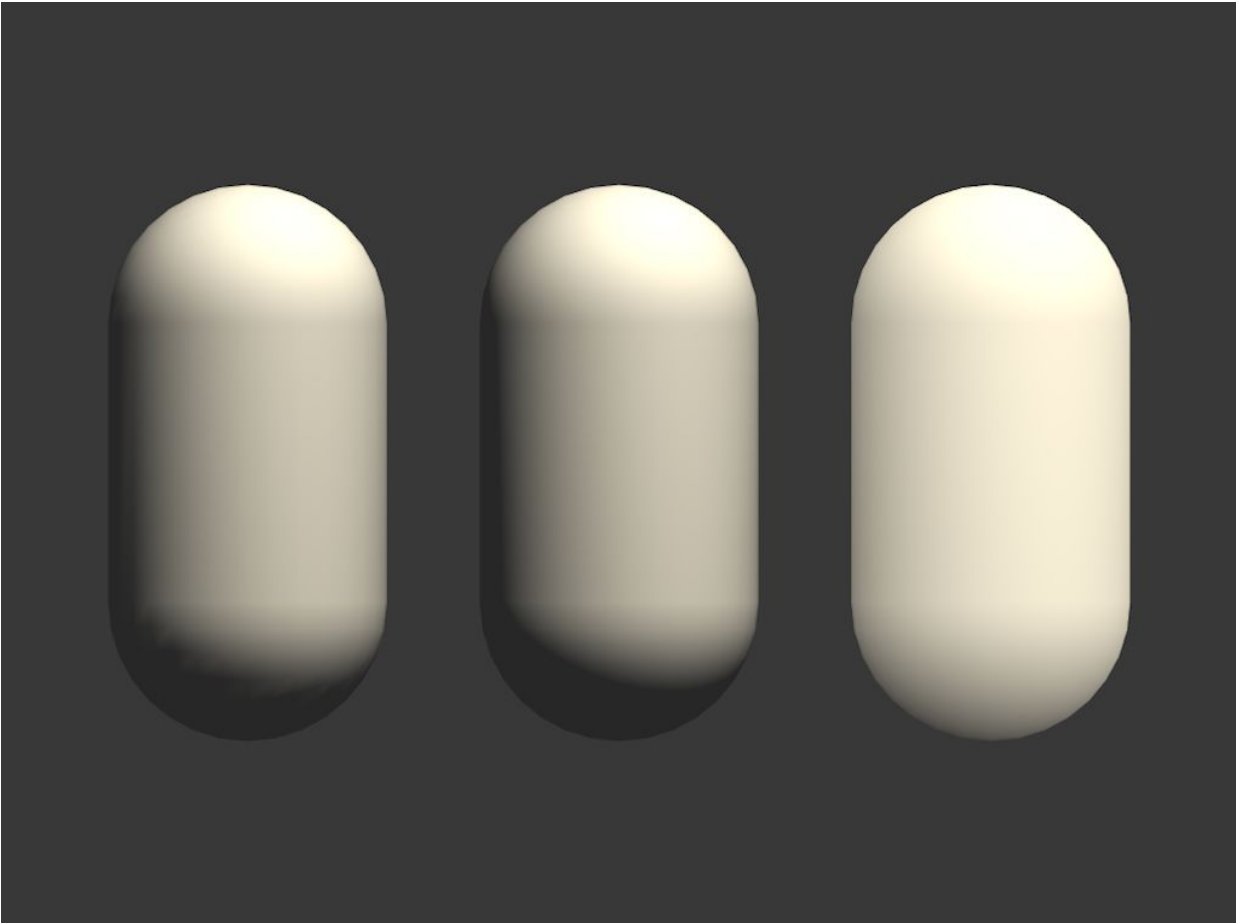
[5] Nguyen H. Gpu gems 3[M]. Addison-Wesley Professional, 2007. GPU 3

[6] GPU Pro 5: Advanced Rendering Techniques[M]. CRC Press, 2014

[7] Engel W. ShaderX7: Advanced Rendering with DirectX and OpenGL (Shaderx Series)[M]. Charles River Media, Inc., 2009

[8] Pharr M, Humphreys G. Physically based rendering: From theory to implementation[M]. Morgan Kaufmann, 2010

[9] Akenine-Möller T, Haines E, Hoffman N. Real-time rendering[M]. CRC Press, 2008□



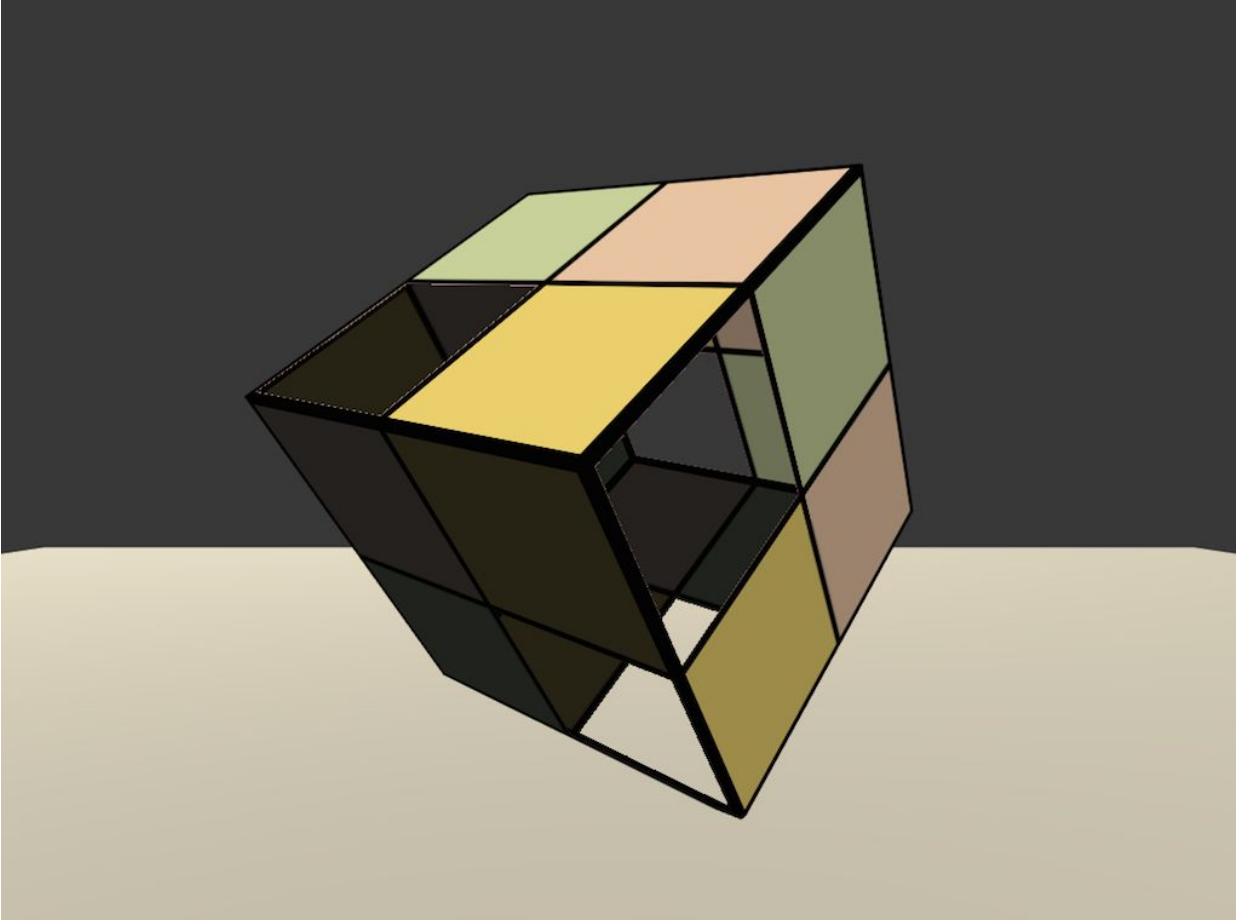
▲6.4



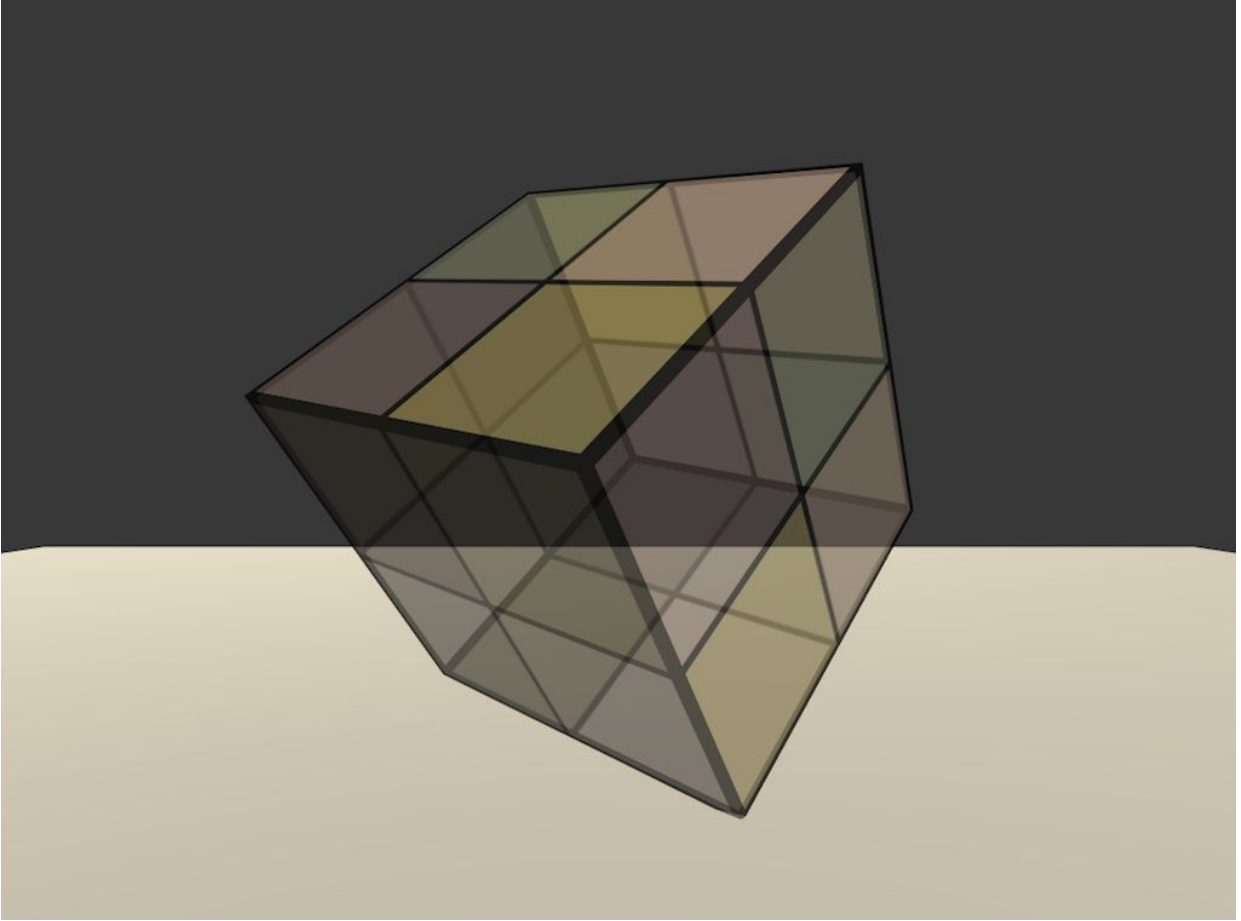
▲7.2□□□□□□□□



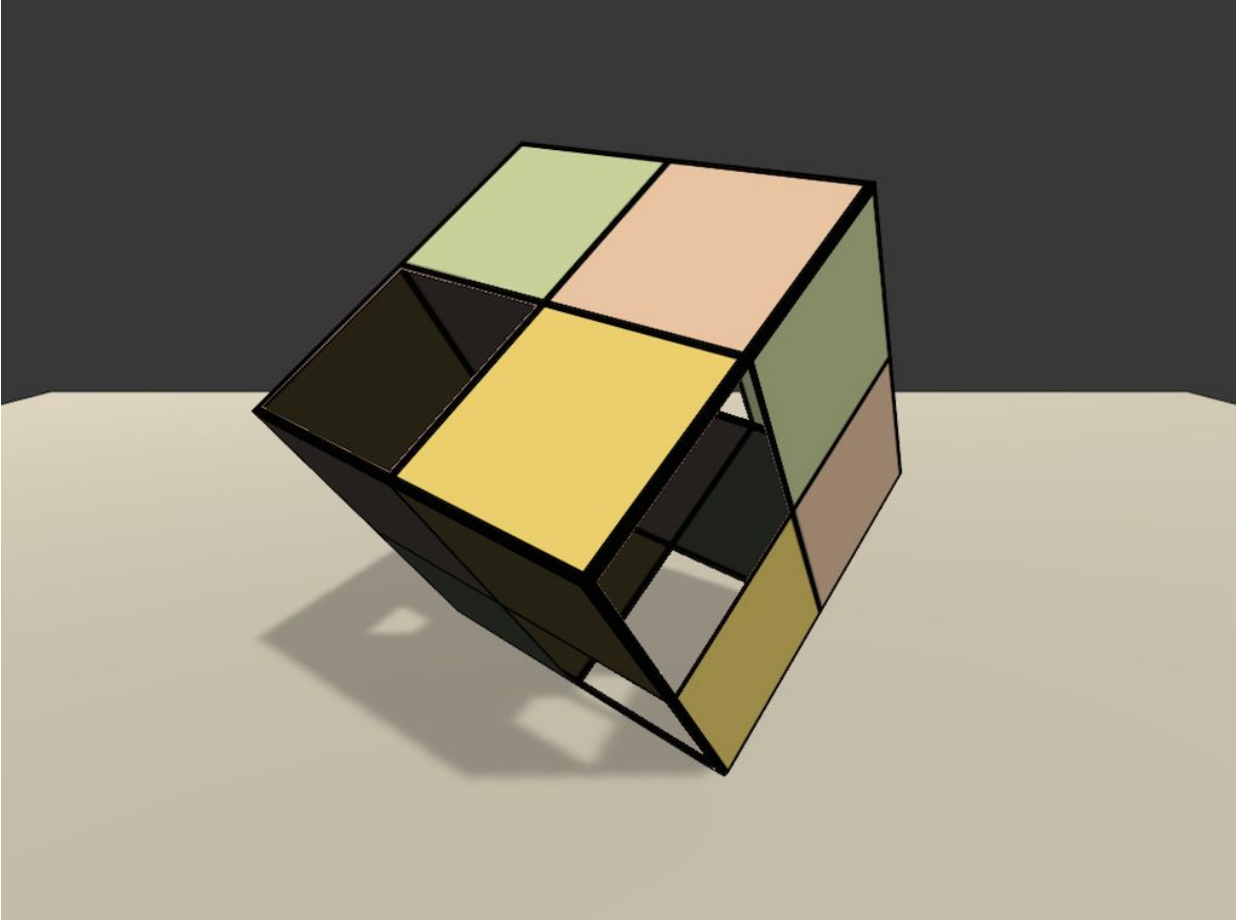
▲7.3□□□□□□□□□□□□□□□□



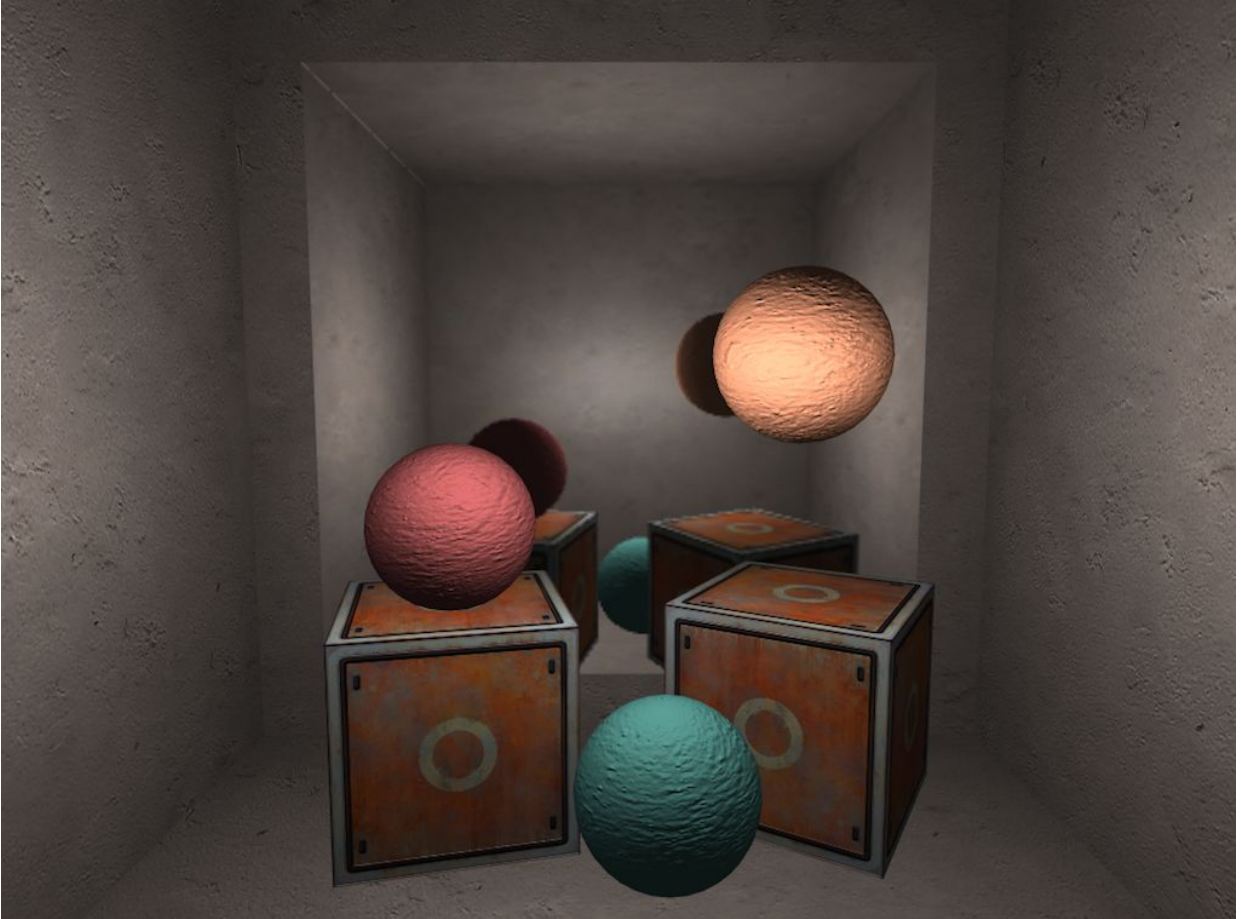
▲8.7.1□□□□□□□□□□□□



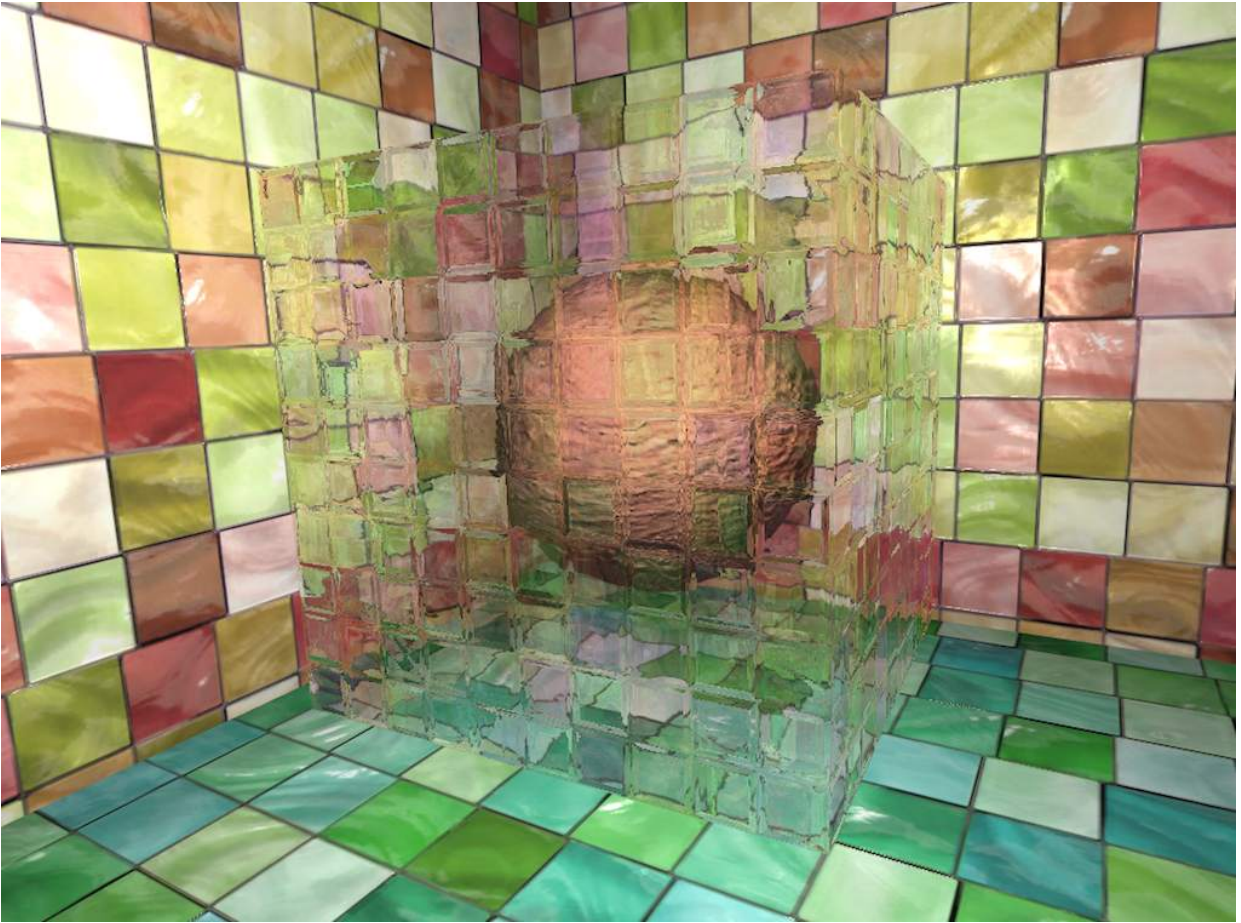
▲8.7.2□□□□□□□□□□□□



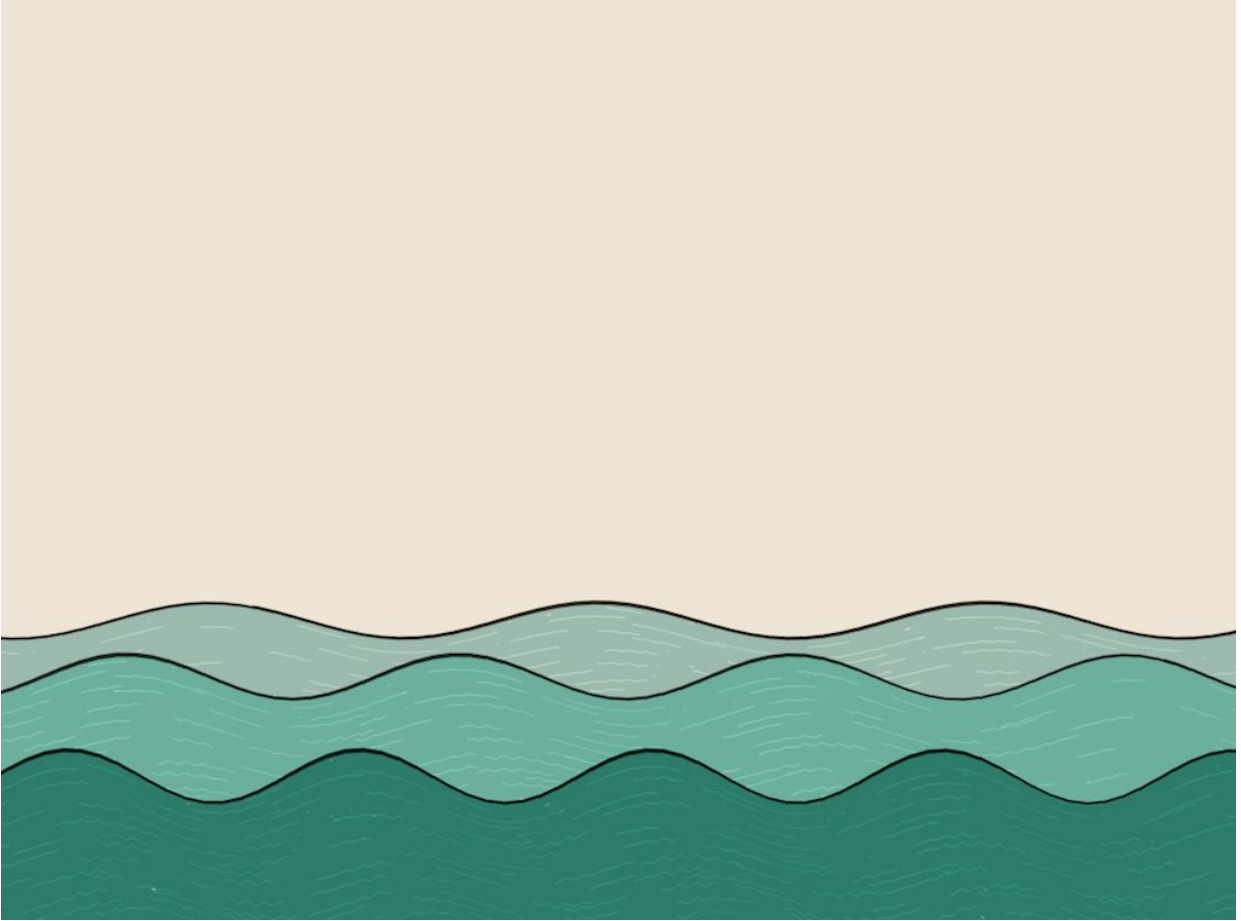
▲9.4□□□□□□□□□□□□



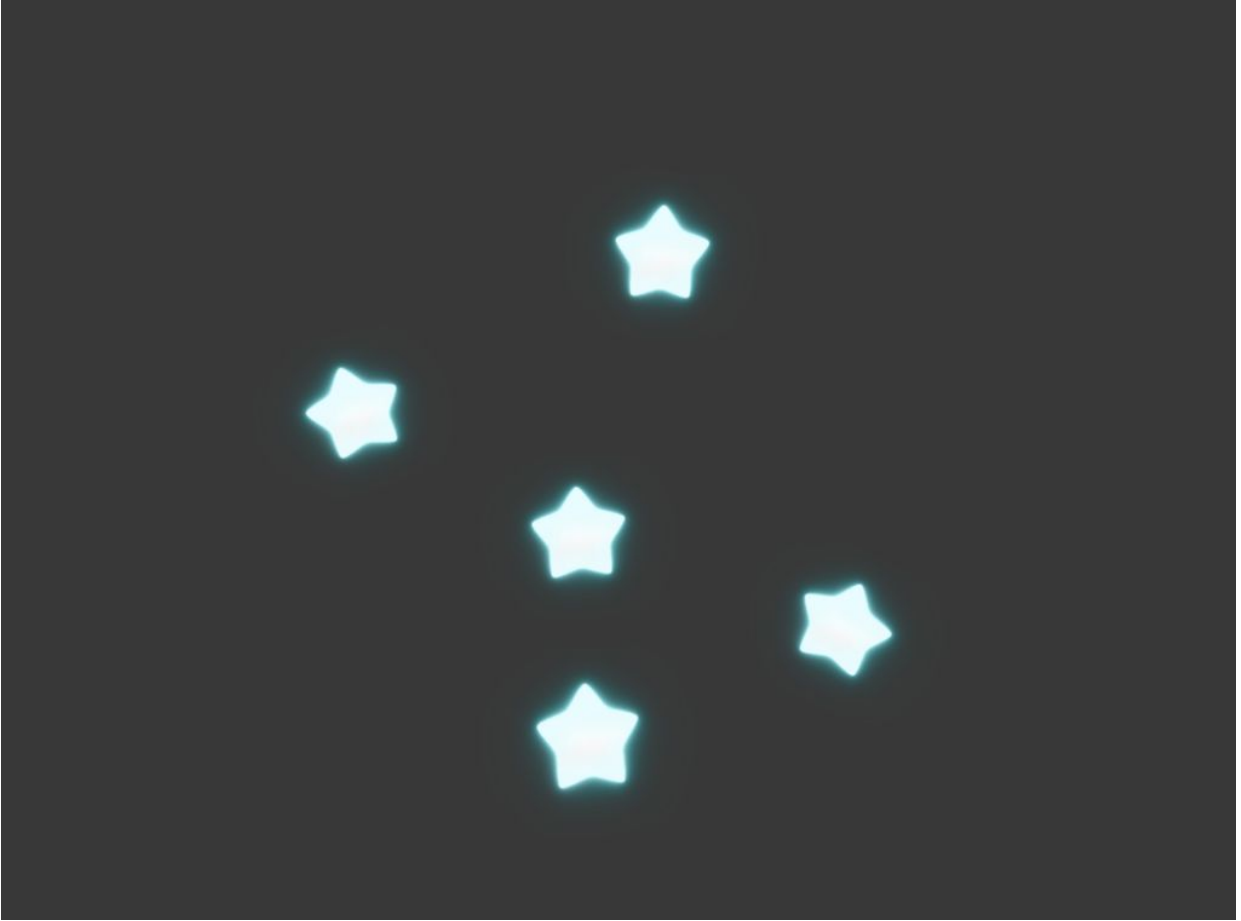
▲10.2.1□□□□□□□□□□□□□□



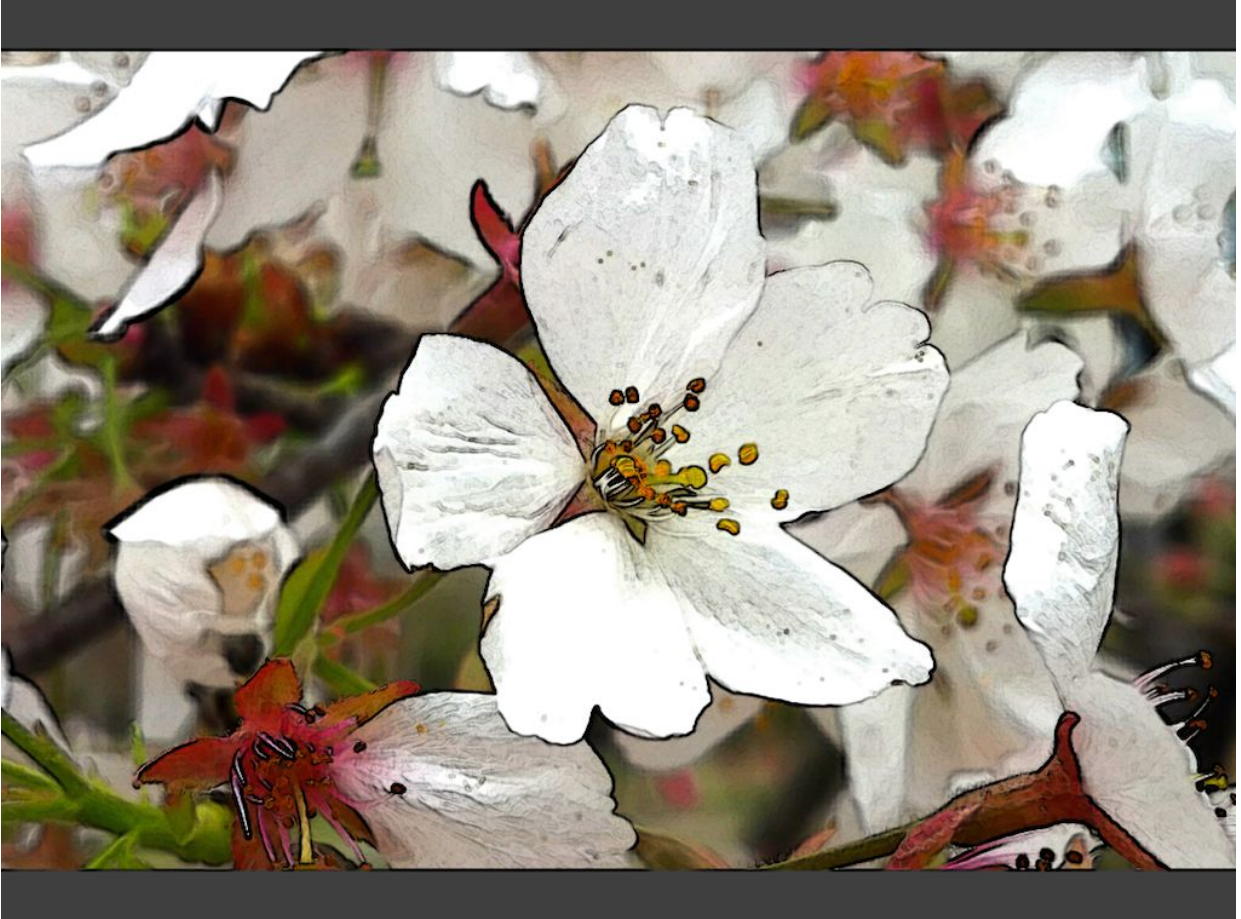
▲10.2 グラブパス (GrabPass)



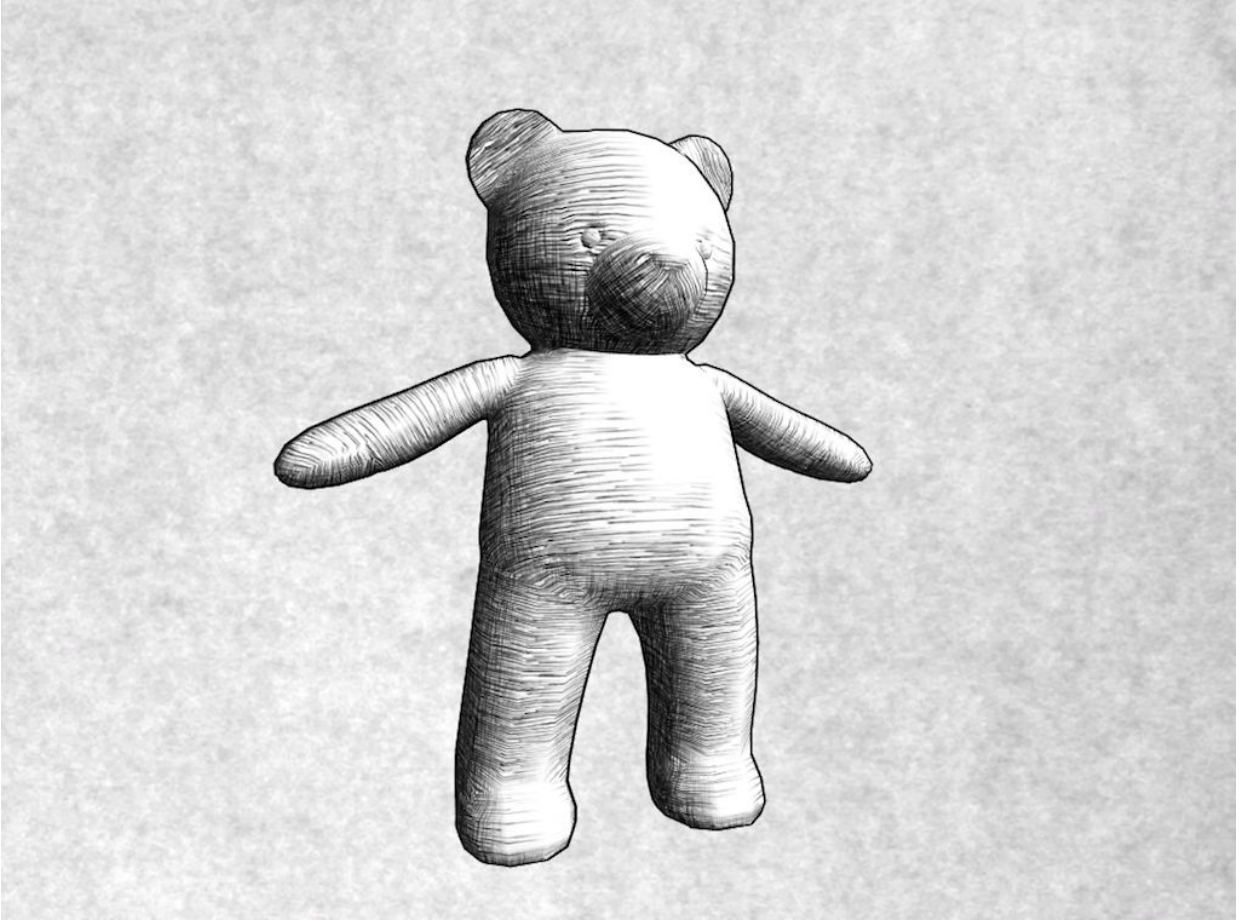
▲11.3.1 2D



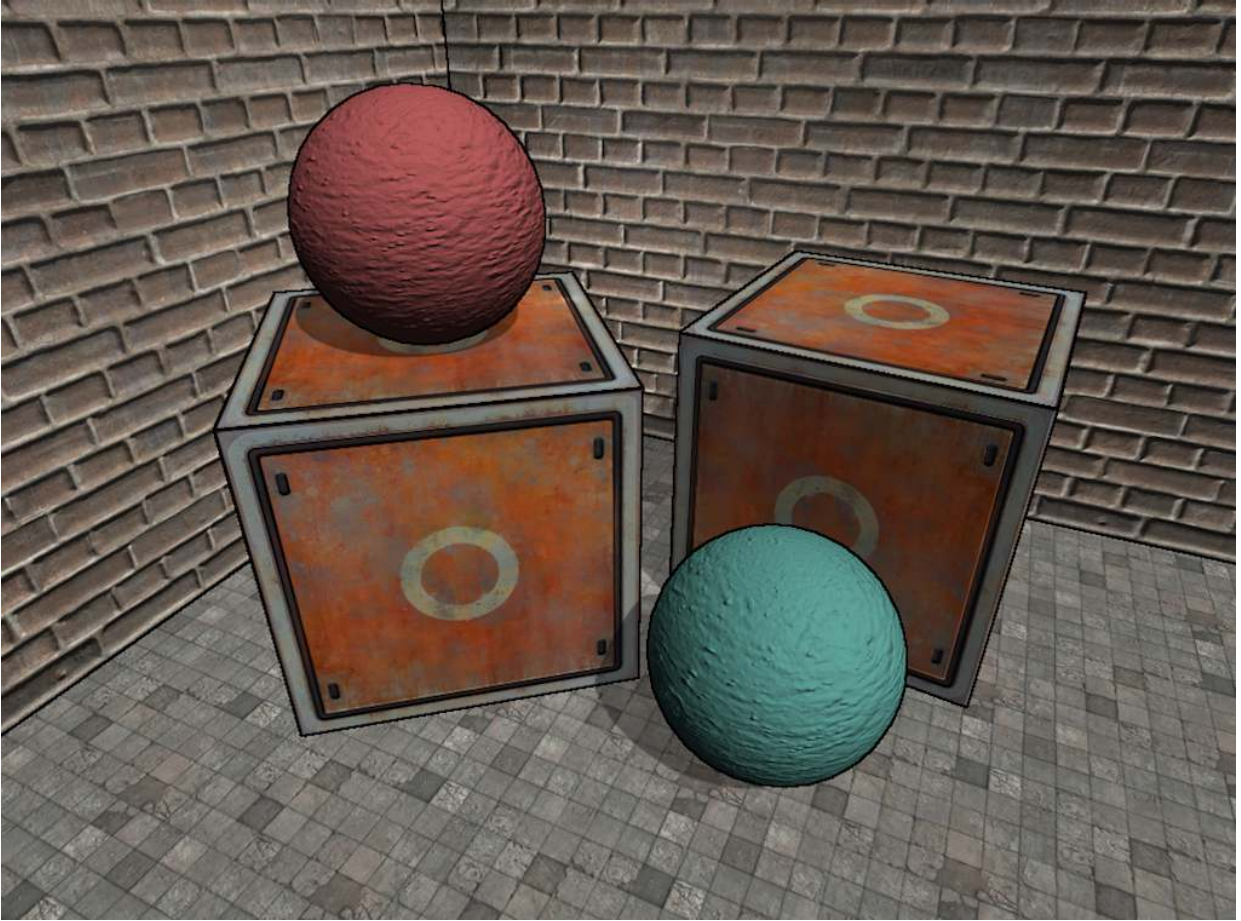
▲11.3.2



▲12.3□□□□□□□□□□□□□□□□



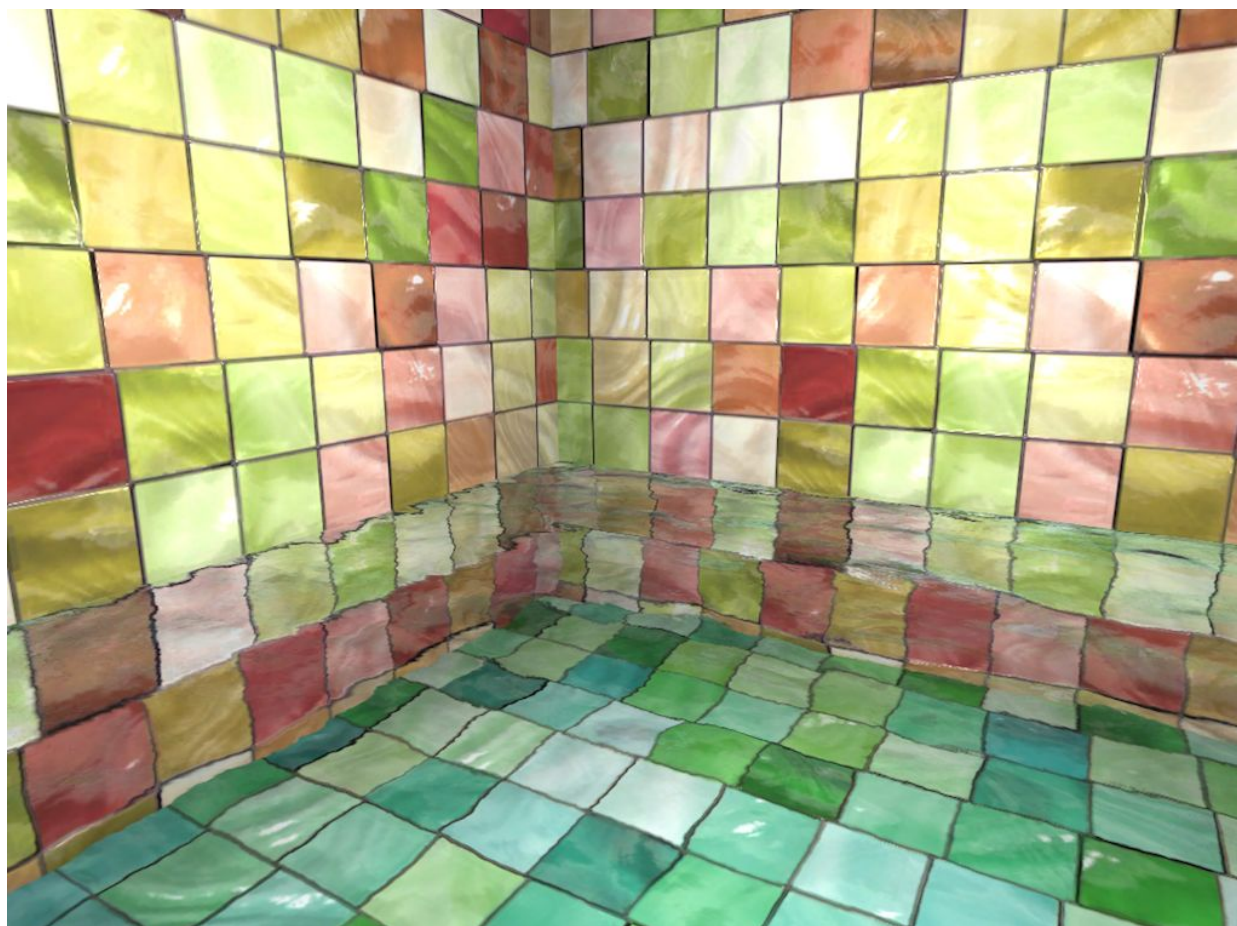
▲14.2□□□□□□□□



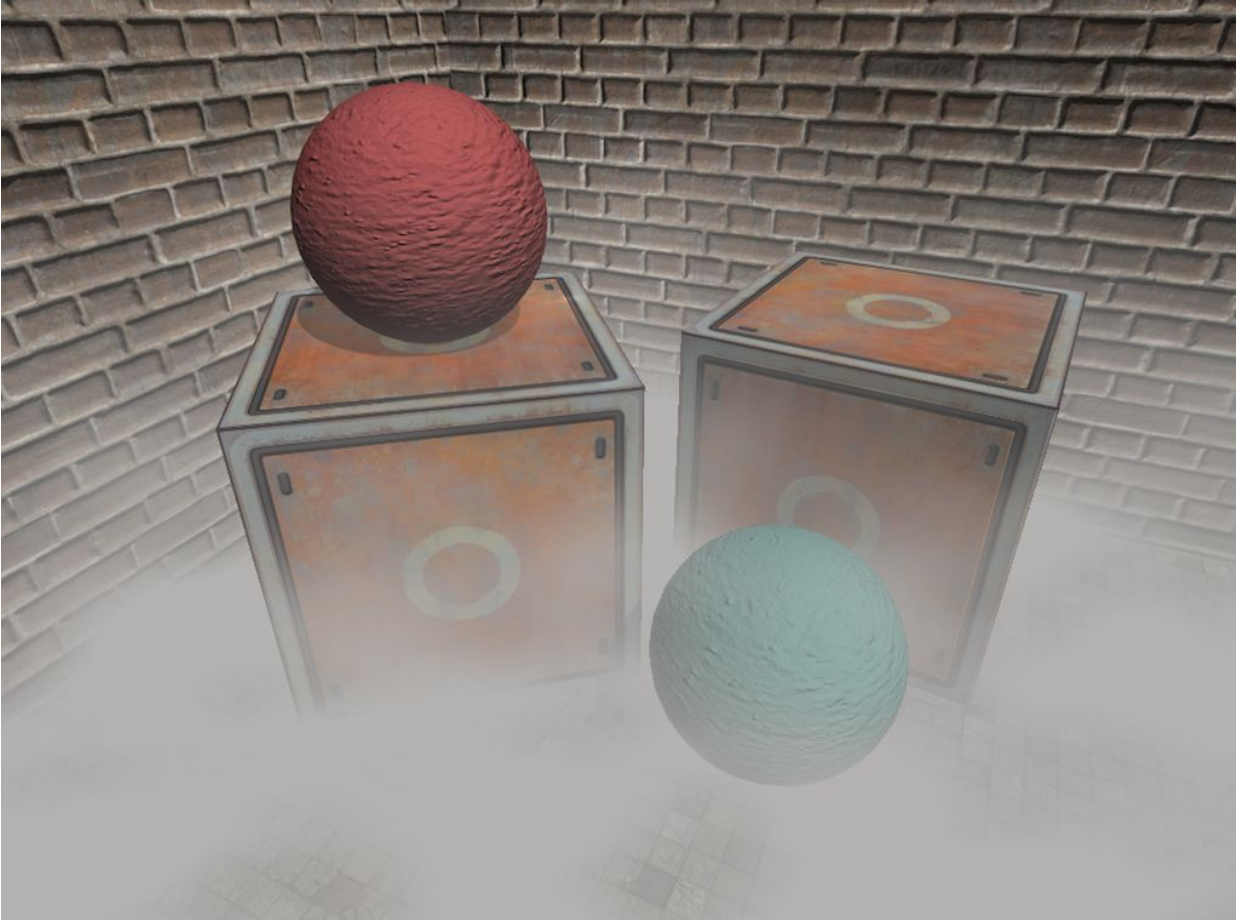
▲13.4□□□□□+□□□□□□□□□□□□□□



▲15.1□□□□□□□□□□□□□□□□



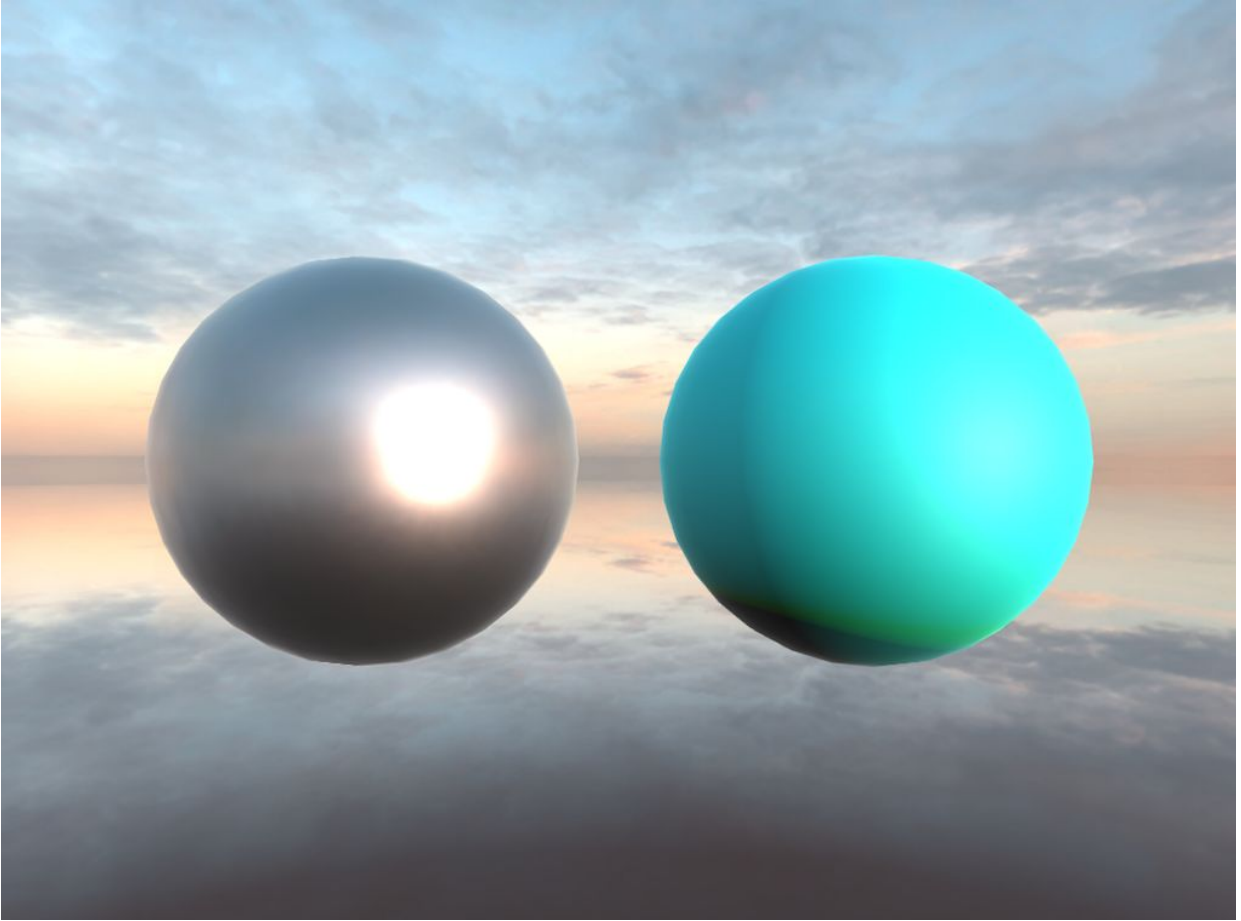
▲15.2□□□□□□□□□□□□□□□□



▲15.3



▲17.1□□□□□□□□



▲18.2□□□□□□□□

□□□□□□□□

□□□□□□□□

□□□□(www.epubit.com.cn)□□□□□□□□IT□□□□□□□□2015
□8□□□□□□

□□□□□□□□□□20□□□IT□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□



异步社区
人民邮电出版社

技术圈 · 图书 电子书 文章

图书/作者/类别



2

写作



异步社区成立一周年大型活动开启

周年庆满减促销 | 满100元减20元、满150元减35元、满200元减50元

+更多



CCIE路由和交换认证考试指南 (第5版) (第1卷)



数据科学实战手册 (R+Python)



软技能: 代码之外的生存指南



Python密码学编程



Python游戏编程快速上手



机器学习项目开发实战



树莓派Python编程入门与实战 (第2版)



像计算机科学家一样思考Python (第2版)



我要写书

近期活动

+更多

异步社区成立一周年大型赠书活动开启！
异步社区的来历 异步社区是人民邮电出版社旗下IT专，业图书旗舰社区，于2015年8月上线运营。异步社区依托于人民邮电出版社20余年的IT专业...
猫叔郭志敬 2016-08-02
阅读 575 推荐 2 收藏 0 评论 8
2016 iWeb峰会北京站即将开启，为HTML5喝彩！
每一次振臂高呼顿时行业的影响，每一天无数人兢兢业业的勤奋，2016雄起！来吧，8月27日，HTML5峰会北京站，我在这里，等你来，为HTML5喝彩！...
猫叔郭志敬 2016-07-29
阅读 60 推荐 1 收藏 0 评论 0

每周半价电子书

+更多



树莓派Python编程入门与实战 (第2版)
[美] Richard Blum 勃鲁姆, Christine Bresnahan 布莱斯纳罕 (作者) 陈晓明 马立新 (译者)

□□□□□□□□

□□□□

□□□□□□□□IT□□□□□□□□Web□□□□□□□□□□□□□□□□□
□□□□□□1000□□□□□□400□□□□□□□□□□□□□□□□□□□□□□□□
□□

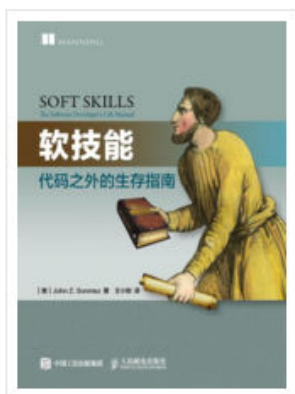
□□□□

□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□



软技能：代码之外的生存指南

[美] 约翰·Z. 森梅兹 (John Z. Sonmez) (作者)

王小刚 (译者)

杨海玲 (责任编辑)



分享

6

推荐



想读

9.0K

阅读

这是一本真正从“人”（而非技术也非管理）的角度关注软件开发人员自身发展的书。书中论述的内容既涉及生活习惯，又包括思维方式，凸显技术中“人”的因素，全面讲解软件行业从业人员所需知道的所有“软技能”。

本书聚焦于软件开发人员生活的方方面面，从揭秘面试的流程到精耕细作出一份杀手级简历，从创建大受欢迎的博客到打造你的个人品牌，从提高工作效率到如何与“拖延症”做斗争，甚至包括如何投资不动产，如何关注自己的健康。

本书共分为职业篇、自我营销篇、学习篇、生产力篇、理财篇、健身篇、精神篇等七篇，概括了软件行业从业人员所需的“软技能”。

● 纸质版 ~~¥59.00~~ **¥46.02** (7.8折)

● 电子版 **¥35.00**

● 电子版 + 纸质版 **¥59.00**

现在购买

下载PDF样章

配套文件下载

□□□□

□□□□□□□□□□□□



□□□□



□□□□



□□□□



二维码



QQ群368449889

访问地址 www.epubit.com.cn

访问地址 二维码

访问地址 @XXXXXXXXXX@XXXXXXXXXX-XXXXXXXXXX

访问地址 & 访问地址 contact@epubit.com.cn

□ □ □

contact@epubit.com.cn

□□□□□□□□□□□□□□□□□□□□ebook@epubit.com.cn□

□ □ □ □ □ □ □ □ □ □

- 1000@1000000
- QQ100368449889