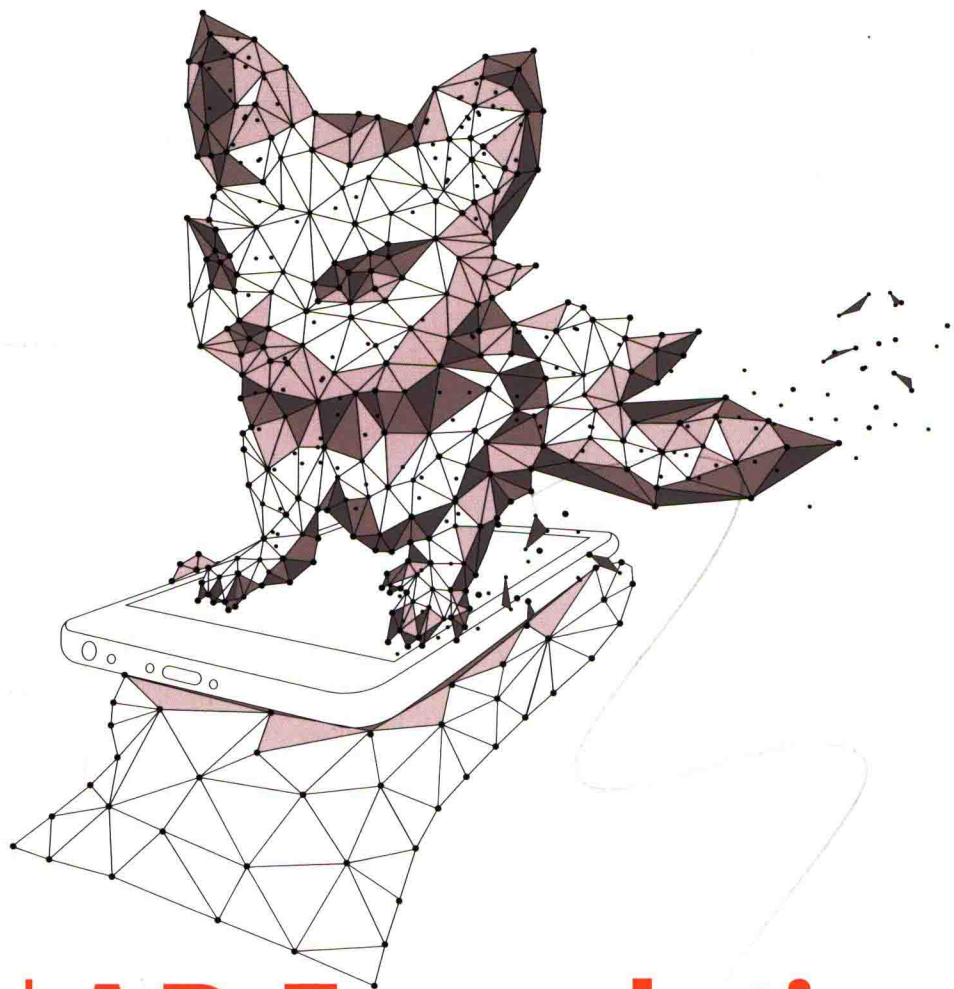




计算机技术开发与应用丛书

详细阐述ARCore底层技术原理，逐一分析ARCore各功能特性
内容翔实，案例丰富，着力解决开发人员在项目开发中面临的难点问题



AR Foundation 增强现实开发实战 (ARCore版)

汪祥春 © 编著



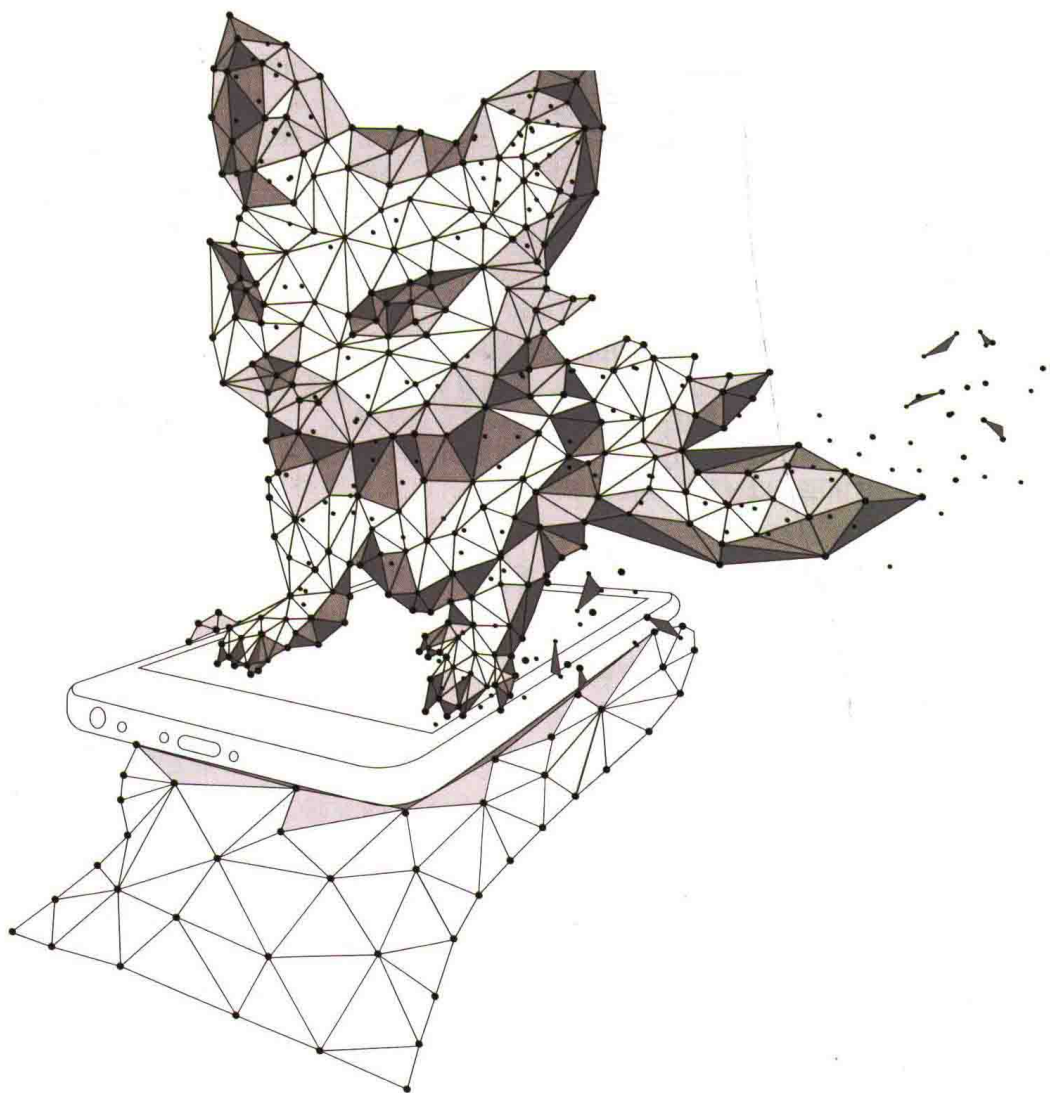
程序代码



清华大学出版社



计算机技术开发与应用丛书



AR Foundation 增强现实开发实战 (ARCore版)

汪祥春 © 编著

清华大学出版社

北京

内 容 简 介

本书基于 AR Foundation 框架,采用 URP 渲染管线,讲述利用 ARCore 进行 Android 平台的 AR 应用开发,从 AR 技术概念、原理、理论脉络到各功能技术点、Android 与 Unity 交互通信、设计原则、性能优化,对 ARCore 应用开发中涉及的技术进行了全方位讲述,用语通俗易懂,阐述深入浅出。

本书共分三部分:第一部分为基础篇,包括第 1 章至第 3 章,从最基础的增强现实概念入手,通过一个实例阐述了 AR 开发从软硬件准备、开发环境搭建、工程设置、发布部署全流程,探讨了 AR Foundation 框架的技术基础、常用组件,并着重探索了 ARCore 功能特性和运动跟踪原理;第二部分为功能技术篇,包括第 4 章至第 12 章,对 ARCore 功能特性各方面进行了详细探索讨论,从平面检测、图像检测、人脸检测到云锚点、光影特效等,全方位地进行了技术剖析、讲解、演示,并对 AR 场景管理、图像信息处理、三维文字音频视频等实用功能进行了阐述和使用操作讲解;第三部分为高级篇,包括第 13 章至第 15 章,主要阐述了 Android 与 Unity 交互通信、AR 应用设计、性能优化相关主题,着力提升开发人员在 AR 应用开发时的实际应用能力和整体把握能力。

本书结构清晰,循序渐进,深浅兼顾,实例丰富,每个技术点都有案例,特别注重对技术原理和实际运用的讲述,力图解决读者在项目开发中面临的难点问题。

本书适合 AR 初学者、Unity 开发人员、程序员、科研人员阅读,也可以作为高校、大专院校相关专业师生的学习用书,以及培训学校的培训教材。

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。

版权所有,侵权必究。举报:010-62782989, beiqinquan@tup.tsinghua.edu.cn。

图书在版编目(CIP)数据

AR Foundation 增强现实开发实战: ARCore 版 / 汪祥春编著. —北京: 清华大学出版社, 2022.9
(计算机技术开发与应用丛书)

ISBN 978-7-302-61218-6

I. ①A… II. ①汪… III. ①虚拟现实—程序设计 IV. ①TP391.98

中国版本图书馆 CIP 数据核字(2022)第 114091 号

责任编辑: 赵佳霓

封面设计: 吴 刚

责任校对: 焦丽丽

责任印制: 曹婉颖

出版发行: 清华大学出版社

网 址: <http://www.wqbook.com>

地 址: 北京清华大学学研大厦 A 座 邮 编: 100084

社 总 机: 010-83470000 邮 购: 010-62786544

投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn

质量反馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn

课件下载: <http://www.tup.com.cn>, 010-83470236

印 装 者: 三河市铭诚印务有限公司

经 销: 全国新华书店

开 本: 186mm × 240mm 印 张: 26.5 字 数: 599 千字

版 次: 2022 年 9 月第 1 版 印 次: 2022 年 9 月第 1 次印刷

印 数: 1~2000

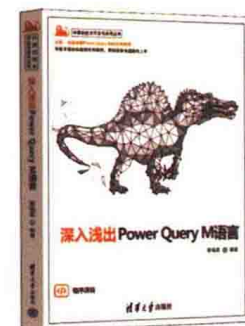
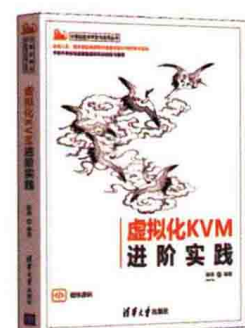
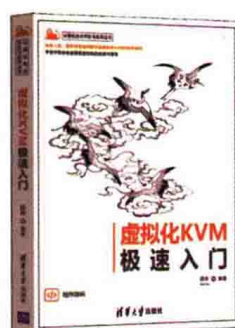
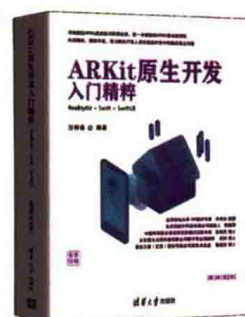
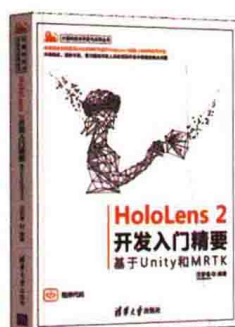
定 价: 109.00 元

产品编号: 097488-01

| 作 | 者 | 简 | 介 |

汪祥春 计算机科学与技术专业硕士，全国信息安全标准化技术委员会虚拟现实与增强现实标准工作组（CNITSC）成员、中国增强现实核心技术产业联盟（CARA）成员、华为HDE（Huawei Developer Experts）、CSDN博客专家，拥有深厚软件工程专业背景，十余年项目实施管理经验，专注AR/MR开发，申请AR专利4项，软件著作权1件。著有《基于Unity的ARCore开发实战详解》《AR开发权威指南：AR Foundation》《ARKit原生开发入门精粹——RealityKit+Swift+SwiftUI》《HoloLens 2开发入门精要——基于Unity和MRTK》。

|图|书|推|荐|



前言

PREFACE

在元宇宙科技浪潮推动下，增强现实（Augmented Reality，AR）技术过去一年来受到了广泛关注，并迅速成为科技巨头和初创企业竞先入局的科技赛道。客观而言，元宇宙仍处于行业发展的初级阶段，无论是底层技术还是应用场景，与预期的成熟形态相比仍有相当大差距，但这也意味着元宇宙相关科技和产业发展空间巨大。作为元宇宙中最核心的增强实现技术是实现虚实融合、新型人机交互模式的关键，必将在接下来的十年中获得长足进步和发展。

AR 技术是一种将虚拟信息与真实世界融合呈现的技术，其广泛运用了人工智能、三维建模、虚实融合、智能交互、传感计算等多种技术手段，将计算机生成的文字、图像、三维模型、声频、视频、动画等虚拟信息模拟仿真后应用到真实世界中。AR 技术同时考虑了真实世界与虚拟信息的相互关系，虚实信息互为补充，从而实现对真实世界的增强。

ARCore 是谷歌公司在前沿科技领域的重大技术布局，是一个高级 AR 应用开发引擎，具备高内聚、易使用的特点，利用其开发 AR 应用快捷高效。ARCore 运动跟踪稳定性好，并且支持多传感器融合（如深度传感器、双目相机），性能消耗低，有利于营造沉浸性更好的 AR 体验。

本书是《AR Foundation 增强现实开发实战（ARKit 版）》的姊妹版，讲述利用 ARCore 进行 Android 平台的 AR 应用开发，从 AR 技术概念、原理、理论脉络到各功能技术点、Android 与 Unity 交互通信、设计原则、性能优化，对 ARCore 应用开发中涉及的技术进行了全方位讲述，旨在帮助开发者系统化掌握 AR 开发的相关知识，建立 ARCore AR 应用开发知识体系。

本书关注 ARCore 功能特性的实际应用，但在讲解技术点的同时对其原理、技术背景进行了较深入的探究，采取循序渐进的方式，使读者知其然更能知其所以然，一步一步地将读者带入 AR 应用开发的殿堂。

前置知识

本书面向 ARCore 应用开发初学者和 Unity 工程师，内容讲述尽力采用通俗易懂的语言、从基础入门，但仍然希望读者能具备以下前置知识。

（1）熟悉 Unity 引擎基本操作，掌握 Unity 开发基本技能，能熟练进行一般性的模型导入导出、属性设置、发布部署。

（2）熟悉 C# 高级语言，掌握基本的 C# 语法及常见数据结构、编码技巧，对常见游戏对象代码操作、事件绑定等有自己的理解。

（3）了解 Visual Studio 开发环境，能进行基本的开发环境设置、功能调试、资源使用。

(4) 了解图形学。数字三维空间是用数学精确描述的虚拟世界,如果读者对坐标系、向量及基本的代数运算有所了解会对理解 AR 应用工作原理、渲染管线有帮助,但本书中没有直接用到复杂的数学计算,读者不用太担心。

读者群体

本书属于技术类书,面向读者人群如下:

- (1) 高等院校及对计算机技术有浓厚兴趣的专科学校学生。
- (2) 对 AR 技术有兴趣的科技工作者。
- (3) 向 AR 方向转行的程序员、工程师。
- (4) 研究讲授 AR 技术的教师。
- (5) 渴望利用新技术的自由职业者或者其他行业人员。

本书特色

(1) 结构清晰。本书共分三个部分:第一部分为基础篇,第二部分为功能技术篇,第三部分为高级篇。紧紧围绕 ARCore 开发,对其功能特性进行了全方位讲述,并从实际应用角度阐述了 Android 与 Unity 交互通信、AR 应用设计准则、性能优化等相关知识。

(2) 循序渐进。本书充分考虑了不同知识背景读者的需求,按知识点循序渐进,通过大量配图、实例进行详细讲解,力求使读者快速掌握 AR Foundation 框架下的 ARCore 开发。

(3) 深浅兼顾。在讲解 ARCore 功能技术点时对其技术原理、理论脉络进行了较深入的探究,用语通俗易懂,对技术阐述深入浅出。

(4) 实用性强。本书实例丰富,每个技术点都有案例,注重对技术的实际运用,力图解决读者在项目开发中面临的难点问题,实用性强。

源代码

本书源代码请扫描下方二维码获取:



本书源代码

读者反馈

尽管编者在本书的编写过程中多次对内容、语言描述的连贯一致性和叙述的准确性进行审查、校正,但由于编者能力水平有限,书中难免存在疏漏,敬请读者批评指正。

致谢

仅以此书献给我的妻子欧阳女士、孩子妍妍及轩轩,是你们让我一直保持昂扬斗志,一往无前,永远爱你们,也感谢清华大学出版社赵佳霓编辑对本书的大力支持。

编者

2022 年 4 月

图书推荐

书 名	作 者
鸿蒙应用程序开发	董昱
HarmonyOS 应用开发实战（JavaScript 版）	徐礼文
鸿蒙操作系统开发入门经典	徐礼文
鸿蒙操作系统应用开发实践	陈美汝、郑森文、武延军、吴敬征
HarmonyOS 移动应用开发	刘安战、余雨萍、李勇军 等
JavaScript 基础语法详解	张旭乾
华为方舟编译器之美——基于开源代码的架构分析与实现	史宁宁
鲲鹏架构入门与实战	张磊
华为 HCIA 路由与交换技术实战	江礼教
Android Runtime 源码解析	史宁宁
深度探索 Go 语言——对象模型与 runtime 的原理、特性及应用	封幼林
Flutter 组件精讲与实战	赵龙
Flutter 组件详解与实战	[加]王浩然（Bradley Wang）
Flutter 实战指南	李楠
Dart 语言实战——基于 Flutter 框架的程序开发（第 2 版）	亢少军
Dart 语言实战——基于 Angular 框架的 Web 开发	刘仕文
IntelliJ IDEA 软件开发与应用	乔国辉
Vue+Spring Boot 前后端分离开发实战	贾志杰
Vue.js 企业开发实战	千锋教育高教产品研发部
Python 从入门到全栈开发	钱超
Python 全栈开发——基础入门	夏正东
Python 游戏编程项目开发实战	李志远
Python 人工智能——原理、实践及应用	杨博雄 主编,于营、肖衡、潘玉霞、高华玲、梁志勇 副主编
Python 深度学习	王志立
Python 预测分析与机器学习	王沁晨
Python 异步编程实战——基于 AIO 的全栈开发技术	陈少佳
Python 数据分析实战——从 Excel 轻松入门 Pandas	曾贤志
Python 数据分析从 0 到 1	邓立文、俞心宇、牛瑶
Python Web 数据分析可视化——基于 Django 框架的开发实战	韩伟、赵盼
Python 玩转数学问题——轻松学习 NumPy、SciPy 和 matplotlib	张赛
Pandas 通关实战	黄福星
深入浅出 Power Query M 语言	黄福星
FFmpeg 入门详解——音视频原理及应用	梅会东
云原生开发实践	高尚衡
虚拟化 KVM 极速入门	陈涛
虚拟化 KVM 进阶实践	陈涛
物联网——嵌入式开发实战	连志安
人工智能算法——原理、技巧及应用	韩龙、张娜、汝洪芳

书 名	作 者
跟我一起学机器学习	王成、黄晓辉
TensorFlow 计算机视觉原理与实战	欧阳鹏程、任浩然
分布式机器学习实战	陈敬雷
计算机视觉——基于 OpenCV 与 TensorFlow 的深度学习方法	余海林、翟中华
深度学习——理论、方法与 PyTorch 实践	翟中华、孟翔宇
深度学习原理与 PyTorch 实战	张伟振
ARKit 原生开发入门精粹——RealityKit + Swift + SwiftUI	汪祥春
HoloLens 2 开发入门精要——基于 Unity 和 MRTK	汪祥春
Altium Designer 20 PCB 设计实战（视频微课版）	白军杰
Cadence 高速 PCB 设计——基于手机高阶板的案例分析与实现	李卫国、张彬、林超文
Octave 程序设计	于红博
ANSYS 19.0 实例详解	李大勇、周宝
AutoCAD 2022 快速入门、进阶与精通	邵为龙
SolidWorks 2020 快速入门与深入实战	邵为龙
SolidWorks 2021 快速入门与深入实战	邵为龙
UG NX 1926 快速入门与深入实战	邵为龙
西门子 S7-200 SMART PLC 编程及应用（视频微课版）	徐宁、赵丽君
三菱 FX3U PLC 编程及应用（视频微课版）	吴文灵
全栈 UI 自动化测试实战	胡胜强、单镜石、李睿
FFmpeg 入门详解——音视频原理及应用	梅会东
pytest 框架与自动化测试应用	房荔枝、梁丽丽
软件测试与面试通识	于晶、张丹
智慧教育技术与应用	[澳]朱佳 (Jia Zhu)
敏捷测试从零开始	陈霖、王富、武夏
智慧建造——物联网在建筑设计与管理中的实践	[美]周晨光 (Timothy Chou) 著；段晨东、柯吉译
深入理解微电子电路设计——电子元器件原理及应用（原书第 5 版）	[美]理查德·C. 耶格 (Richard C. Jaeger)、[美]特拉维斯·N. 布莱洛克 (Travis N. Blalock) 著；宋廷强译
深入理解微电子电路设计——数字电子技术及应用（原书第 5 版）	[美]理查德·C. 耶格 (Richard C. Jaeger)、[美]特拉维斯·N. 布莱洛克 (Travis N. Blalock) 著；宋廷强译
深入理解微电子电路设计——模拟电子技术及应用（原书第 5 版）	[美]理查德·C. 耶格 (Richard C. Jaeger)、[美]特拉维斯·N. 布莱洛克 (Travis N. Blalock) 著；宋廷强译

目录

CONTENTS

基础篇

第1章 AR 开发入门	1
1.1 增强现实技术概述	3
1.1.1 AR 概念	3
1.1.2 AR 技术应用	5
1.2 AR 技术原理	6
1.2.1 传感器数据	6
1.2.2 前端里程计	8
1.2.3 后端优化	8
1.2.4 回环检测	8
1.2.5 建图	9
1.3 AR Foundation 概述	10
1.3.1 AR Foundation 与 ARCore	11
1.3.2 AR Foundation 支持的功能	11
1.3.3 AR Foundation 功能概述	12
1.3.4 AR Foundation 体系架构概述	13
1.3.5 基本术语	13
1.4 支持的设备	15
1.5 开发环境准备	16
1.5.1 所需硬件和软件	17
1.5.2 软件安装	17
1.6 ARCore 初体验	19
1.6.1 工程创建	19
1.6.2 工具包导入	21
1.6.3 工程设置	22
1.6.4 环境搭建	24

1.6.5	代码编写	27
1.6.6	调试准备	29
1.6.7	打包测试	30
1.7	AR 应用调试	31
第 2 章	AR Foundation 基础	36
2.1	AR Foundation 体系架构	36
2.1.1	AR 子系统概念	37
2.1.2	AR 子系统使用	38
2.1.3	跟踪子系统	39
2.2	AR Session & AR Session Origin	40
2.2.1	AR Session	40
2.2.2	AR Session Origin	43
2.3	可跟踪对象	45
2.3.1	可跟踪对象管理器	46
2.3.2	可跟踪对象事件	47
2.3.3	管理可跟踪对象	48
2.4	会话管理	49
第 3 章	ARCore 功能特性与开发基础	52
3.1	ARCore 概述及主要功能	52
3.1.1	运动跟踪	53
3.1.2	环境理解	53
3.1.3	光照估计	54
3.1.4	ARCore 的不足	54
3.2	运动跟踪原理	56
3.2.1	ARCore 坐标系	56
3.2.2	ARCore 运动跟踪分类	56
3.2.3	ARCore 运动跟踪	58
3.2.4	ARCore 使用运动跟踪的注意事项	60
3.3	设备可用性检查	61
3.4	AR 会话生命周期管理与跟踪质量	62
3.5	ARCore 扩展包	65
3.6	相机配置	66
3.7	会话录制与回放	68

3.7.1 AR Foundation 录制与回放	69
3.7.2 ARCore 扩展录制与回放	72
3.8 即时放置	78

功能技术篇

第 4 章 平面检测与锚点管理	85
4.1 平面检测引导	85
4.2 平面管理	88
4.2.1 平面检测	88
4.2.2 可视化平面	89
4.2.3 个性化渲染平面	90
4.2.4 开启与关闭平面检测功能	95
4.2.5 显示与隐藏已检测平面	97
4.2.6 平面遮挡	99
4.3 射线检测	100
4.3.1 射线检测概念	100
4.3.2 射线检测详细讲解	102
4.4 可视化放置点	104
4.5 特征点与点云	106
4.5.1 特征点	107
4.5.2 点云	108
4.5.3 点云数据采集	109
4.6 锚点	113
第 5 章 图像检测跟踪	118
5.1 图像检测跟踪基本操作	118
5.2 图像跟踪功能启用与禁用	123
5.3 多图像跟踪	124
5.4 运行时创建参考图像库	129
5.5 运行时切换参考图像库	130
5.6 运行时添加参考图像	132
5.7 脱卡	136
5.8 图像跟踪优化	137

第 6 章 人脸检测跟踪	140
6.1 人脸检测基础	140
6.1.1 人脸检测概念	140
6.1.2 人脸检测技术基础	141
6.2 人脸姿态与网格	143
6.2.1 人脸姿态	143
6.2.2 人脸网格	145
6.3 人脸区域与多人脸检测	151
6.3.1 人脸区域	151
6.3.2 多人脸检测	155
第 7 章 光影效果	159
7.1 光照基础	159
7.1.1 光源	160
7.1.2 光与材质的交互	160
7.1.3 光照模型	161
7.1.4 三维渲染	162
7.2 光照估计	163
7.2.1 光照一致性	163
7.2.2 光照估计实例	163
7.3 环境反射	167
7.3.1 立方体贴图	167
7.3.2 纹理采样过滤	168
7.3.3 AR Environment Probe Manager 组件	169
7.3.4 使用环境反射	170
7.4 内置实时阴影	171
7.4.1 ShadowMap 技术原理	172
7.4.2 使用实时阴影	173
7.5 Planar 阴影	178
7.5.1 数学原理	178
7.5.2 代码实现	179
7.6 伪阴影	185
第 8 章 持久化存储与多人共享	187
8.1 云锚点	187

8.2	ARCore 云锚点使用实例	188
8.3	ARCore 云锚点使用注意事项	195
8.4	Azure 云服务概述	196
8.5	Azure 空间定位点使用实例	197
第 9 章	场景图像获取与场景深度	206
9.1	获取 GPU 图像	206
9.1.1	获取摄像头原始图像	206
9.1.2	获取屏幕显示图像	208
9.2	获取 CPU 图像	212
9.2.1	AR 摄像头图像数据流	212
9.2.2	从 CPU 中获取摄像头图像	213
9.3	边缘检测原理	222
9.3.1	卷积	223
9.3.2	Sobel 算子	224
9.4	CPU 图像边缘检测实例	226
9.5	Depth API 概述	229
9.6	Depth API 实例	232
9.7	场景深度应用场景	237
第 10 章	相机与手势操作	239
10.1	AR 场景操作	239
10.1.1	场景操作方法	239
10.1.2	场景操作实例	240
10.2	手势操作交互	242
10.2.1	手势检测	242
10.2.2	手势操作控制	244
10.3	XR Interaction Toolkit	247
10.3.1	对象放置	248
10.3.2	对象选择	252
10.3.3	对象操作	254
第 11 章	三维文字与音视频	256
11.1	三维文字	256
11.1.1	文字单位换算	256
11.1.2	中文字体制作	257

11.2	三维声频	258
11.2.1	三维声场原理	259
11.2.2	RA 音频使用	260
11.2.3	RA 组件	260
11.2.4	使用 RA 音频	262
11.2.5	音频效果设置	266
11.2.6	RA 音频 API	268
11.2.7	运行时启用和禁用三维音效	268
11.2.8	使用回响营造空间感	269
11.2.9	三维音效设计原则	272
11.3	三维视频	273
11.3.1	VideoPlayer 组件	274
11.3.2	三维视频播放实现	277
11.3.3	视频音效空间化	280
第 12 章	glTF 与 Model Viewer	281
12.1	glTF 概述	281
12.2	Model Viewer	284
12.2.1	Model Viewer 使用实例	285
12.2.2	Model Viewer 详解	288
12.2.3	使用 Model Viewer 的注意事项	294

高级篇

第 13 章	Android 与 Unity 交互通信	299
13.1	Android 与 Unity 通信原理	299
13.2	Unity 直接调用 Java 代码	302
13.3	模块调用	312
13.4	继承 UnityPlayerActivity 类	317
13.5	AndroidJavaProxy 类详细讲解	320
13.6	订阅 Android 广播	323
13.7	调用 so 库文件	328
13.8	Android 与 Unity 交互注意事项	330

第 14 章	设计原则	334
14.1	移动 AR 带来的挑战	334
14.1.1	三维化思维	334
14.1.2	用户必须移动	335
14.1.3	要求手持设备	335
14.1.4	近脸使用	336
14.1.5	操作手势	336
14.1.6	启动应用	337
14.1.7	色彩运用	337
14.2	移动 AR 设计准则	337
14.2.1	有用或有趣	338
14.2.2	虚实融合必须有意义	338
14.2.3	移动限制	338
14.2.4	心理预期与维度转换	338
14.2.5	环境影响	339
14.2.6	视觉效果	340
14.2.7	UI 设计	340
14.2.8	沉浸式交互	340
14.3	移动 AR 设计指南	341
14.3.1	环境	342
14.3.2	用户细节	343
14.3.3	虚拟内容	346
14.3.4	交互	352
14.3.5	视觉设计	357
14.3.6	真实感	362
14.3.7	三维音效	367
第 15 章	性能优化	369
15.1	性能优化基础	369
15.1.1	影响性能的主要因素	369
15.1.2	AR 应用常用调试方法	371
15.1.3	AR 应用性能优化的一般原则	372
15.2	AR 应用性能调试工具	373
15.2.1	ARCore 性能调试功能	373

15.2.2	Unity 分析器	375
15.2.3	帧调试器	379
15.2.4	UPR 分析器	380
15.3	Unity 分析器使用	382
15.3.1	CPU 使用情况分析器	382
15.3.2	渲染情况分析器	383
15.3.3	内存使用情况分析器	383
15.3.4	物理分析器	384
15.3.5	音视频分析器	385
15.4	性能优化流程	385
15.4.1	收集运行数据	387
15.4.2	分析运行数据	388
15.4.3	确定问题原因	390
15.5	渲染优化	392
15.5.1	渲染流程	392
15.5.2	CPU 瓶颈	393
15.5.3	GPU 瓶颈	396
15.6	代码优化	398
15.6.1	内存管理	398
15.6.2	垃圾回收	399
15.6.3	对象池	404
15.6.4	常见影响性能的代码优化策略	405
15.7	ARCore 优化设置	407
15.7.1	UI/UX 优化	407
15.7.2	常用优化设置	407
15.7.3	AR 应用开发的一般注意事项	408

参考文献	410
------------	-----

基础篇



增强现实是一门新兴技术，它是三维建模渲染、虚实融合、传感计算、人工智能、计算机视觉处理等技术发展的结果，是全新的朝阳技术，也被誉为本十年最重要的技术之一，特别是在元宇宙科技浪潮推动下，应用广泛，前景广阔，正呈现蓬勃发展的态势。

本篇为基础入门篇，从增强现实概念入手，通过一个 HelloWorld 增强现实应用体验 AR 应用开发部署全流程，探讨了 AR Foundation 框架的技术基础、常用组件，并着重探索了 ARCore 功能特性和运动跟踪原理。

基础篇包括以下章节：

第 1 章 AR 开发入门

基础入门章节，简述了增强现实概念、运动跟踪原理、AR Foundation 开发框架，通过一个实例阐述了 AR 开发从软硬件准备、开发环境搭建、工程设置、发布部署全流程，并简要介绍了 AR 应用调试方法和工具。

第 2 章 AR Foundation 基础

简述了 AR Foundation 体系架构、常用功能对象和组件、可跟踪对象及其管理，并对 AR 会话管理进行了使用演示。

第 3 章 ARCore 功能特性与开发基础

从 ARCore 主要功能、优势及不足、运动跟踪到设备可用性检查，对 ARCore 基本功能特性进行了讲述，并对 ARCore 扩展包及其使用进行了演示。

第 1 章



AR 开发入门

科学技术的发展拓展了人类感知的深度与广度，增强了人类对世界的认知能力。高速的数据流使信息的传递与获取前所未有地便捷，虚实融合技术的出现，开创了人类认知领域新的维度，推动着信息获取向更高效、直观、更具真实感的方向发展。

1.1 增强现实技术概述

增强现实技术是一种将虚拟信息与真实世界融合展示的技术，其广泛运用了人工智能、三维建模、运动跟踪、虚实融合、智能交互、传感计算等多种技术手段，将计算机生成的文字、图像、三维模型、声频、视频、动画等虚拟信息模拟仿真后，应用到真实世界中。增强现实技术同时考虑了真实世界与虚拟信息的相互关系，虚实信息互为补充，营造真实与虚拟无痕混合的体验。

1.1.1 AR 概念

VR、AR、XR、MR 这些英文术语缩写有时让初学者感到困惑。VR 是 Virtual Reality 的缩写，即虚拟现实，是一种能够创建和体验纯虚拟世界的计算机仿真技术，它利用计算机生成交互式的全数字三维视场，能够营造全虚拟的环境。AR 是 Augmented Reality 的缩写，即增强现实，是采用以计算机为核心的现代科技手段将生成的文字、图像、视频、三维模型、动画等虚拟信息以视觉、听觉、味觉、嗅觉、触觉等生理感觉方式融合叠加至真实场景中，从而对使用者感知到的真实世界进行增强的技术。VR 是创建完全数字化的世界，隔离真实与虚拟；AR 则是对真实世界的增强，融合了真实与虚拟。近年来，VR 与 AR 技术快速发展，应用越来越广，并且相互关联、相互促进，很多时候会被统称为 XR（Extended Reality，扩展现实）。

MR 是 Mixed Reality 的缩写，即混合现实，是融合真实和虚拟世界的技术，混合现实概念由微软公司提出，强调物理实体和数字对象共存并实时相互作用，如虚实遮挡、环境反射等。比较而言，AR 强调的是对真实世界的增强，MR 则更强调虚实的融合，更关注虚拟数字世界

与真实现实世界之间的交互,如环境遮挡、人形遮挡、场景深度、物理模拟,也更关注以自然、本能的方式操作虚拟对象,但 AR 与 MR 并非两个割裂的概念,其实是对同一事物从不同角度的描述,随着移动端技术的发展,MR 概念中强调的环境遮挡、物理模拟,AR 中也能被高效地实现。

本书主要关注 Android 移动端的 AR 技术,并将详细讲述如何利用 Unity 引擎和 AR Foundation 工具包在 ARCore 基础上开发构建移动设备上的 AR 应用,AR 虚实融合效果如图 1-1 所示。



图 1-1 AR 将虚拟信息叠加在真实环境之上

在增强现实环境中,真实世界环境被计算机生成的文字、图像、视频、三维模型、动画等虚拟信息“增强”,甚至可以跨越视觉、听觉、触觉、体感和嗅觉等多种感官模式。叠加的虚拟信息可以是建设性的(对现实环境的附加),也可以是破坏性的(对现实环境的掩蔽),并与现实世界无缝地交织在一起,让人产生身临其境、真假难辨的感观体验,分不清虚实。通过这种方式,增强现实可以改变用户对真实世界环境的持续感知,这与虚拟现实将虚实隔离,用虚拟环境取代用户真实世界环境完全不一样。

增强现实的主要价值在于它将数字信息带入个人对现实世界的感知中,而不是简单的数据显示,通过与被视为环境自然部分的沉浸式集成实现对现实的增强。借助移动 AR 设备,用户周围的增强世界变得可交互和可操作。简而言之,AR 通过运动跟踪、环境理解等技术手段将现实与虚拟信息进行无缝对接,将在现实中不存在的事物构建在与真实环境一致的同一个三维场景中予以展现、衔接融合。

增强现实技术的发展将改变人们观察世界的方式,世界将不再是我们看到的表面现象的集合,而可以有其更深刻和个性化的内涵,从而引发人类对世界认知方式的变革。想象用户行走或者驱车行驶在路上,通过增强现实显示器(AR 眼镜或者全透明挡风玻璃显示器),信息化图像将出现在用户的视野之内(如路标、导航、提示),这些增强信息实时更新,并且所播放的声音与用户所看到的场景保持同步;或者当我们看到一朵蘑菇时,通过 AR 眼镜即可马上获知其成分和毒性;或者当我们在任何时候需要帮助时,数字人工智能人形助理马上出现在我们面前,以与真人无差异的形象全程为我们服务。

不仅如此,增强现实技术的发展符合更直接、直观、本能的人机交互趋势,必将创造全新的人机交互模式,以更加自然的方式连接虚实世界。

1.1.2 AR 技术应用

AR 技术是一门交叉综合技术,近几年来得到了长足的发足。相比于传统二维矩形框显示信息的方式,通过 AR 可以提供更自然、更直观、更符合人类本能的信息呈现和交互能力。概括而言,AR 系统具有三个突出特点:

- (1) 真实世界与虚拟三维数字世界对齐融合。
- (2) 虚实信息相互影响实时交互。
- (3) 虚实世界均遵循几何一致性和光照一致性原则。

基于标识的增强现实 (Marker Based AR) 实现方案是采用图像作为识别物,即使用专门设计的黑边图或者经过训练的自然图像作为标识物,并在此基础上叠加虚拟信息。该方案在 AR 早期发展中占有重要地位,并在儿童绘本、海报广告 AR 方案中被大量使用,但由于需要预先制作跟踪标识物、识别不稳定、功能有限,不利于推广,也无法实现 6DoF (Degree of Freedom, 自由度) 设备位姿跟踪^①。

无标识增强现实 (Markerless Based AR) 技术的核心基础是实时跟踪设备 6DoF 位姿,在此基础上渲染虚拟元素实现虚实融合。无标识 AR 技术由于可以将虚拟信息叠加到现实世界之上、营造虚实融合一致的体验而在很多领域有广泛的应用前景和潜力^②。无标识增强现实技术诞生后即迅速应用于游戏和娱乐领域,并促进了相关技术的融合,除此之外,AR 技术在文学、艺术、考古、博物、建筑、视觉艺术、零售、应急管理/搜救、教育、社会互动、工业设计、医学、空间沉浸与互动、飞行训练、军事、导航、旅游观光、音乐、虚拟装潢等领域有着广阔的应用前景。

提示

在本书中:①虚拟对象、虚拟信息、虚拟物体均指在真实环境上叠加的由计算机处理生成的文字、图像、三维模型、视频等虚拟非真实信息,严格来讲这三者是有差别的,但有时我们在描述时并不严格区分这三者之间的差异;②Unity、Unity3D 均指 Unity 3D 游戏引擎软件。

① 自由度 (Degrees of Freedom, DoF) 是指物理学中描述一个物理状态,独立对物理状态结果产生影响的变量的数量。运动自由度是确定一个系统在空间中的位置所需要的最小坐标数。在三维坐标系描述一个物体在空间中的位置和朝向信息需要 6 个自由度数据,即 6DoF,指 X、Y、Z 方向上的三维运动 (移动) 加上俯仰 / 偏转 / 滚动 (旋转)。

② 本书主要讲述基于视觉 SLAM 的无标识 ARCore 增强现实技术,后文所述 AR 技术均指提供 6DoF 跟踪能力的增强现实技术。

1.2 AR 技术原理

AR 应用带给使用者奇妙体验的背后是数学、物理学、几何学、人工智能技术、传感器技术、芯片技术、计算机科学技术等基础科学与高新技术的深度融合,对开发人员而言,了解其技术原理有助于理解 AR 整个运行生命周期,理解其优势与不足,更好地服务于应用开发工作。

对 AR 应用而言,最重要也是最基础的功能是感知设备(使用者)位置,将虚拟数字世界与真实物理环境对齐,其核心技术基础是 SLAM (Simultaneous Localization And Mapping, 同时定位与建图) 技术,即搭载特定传感器的主体,在没有环境先验信息的情况下,于运动过程中建立环境模型,同时估计自己的运动。SLAM 技术解决两个问题:一个是估计传感器自身位置;另一个是建立周围环境模型。通俗地讲,SLAM 技术就是在未知环境中确定设备的位置与方向,并逐步建立对未知环境的认知(构建环境的数字地图)。SLAM 技术不仅是 AR/MR 技术的基础,也是自动驾驶、自主导航无人机、机器人等众多需要自主定位技术的基础。经过近 30 年的发展,SLAM 技术理论、算法框架已基本定型,现代典型视觉 SLAM 技术框架如图 1-2 所示。



图 1-2 现代典型视觉 SLAM 技术架构示意图

1.2.1 传感器数据

携带于运动主体上的各类传感器,如激光传感器、相机、轮式编码器、惯性测量单元(Inertial Measurement Unit, IMU)、深度传感器等,它们采集的环境信息为整个 SLAM 系统提供数据来源,其中轮式编码器会测量轮子的转动角度,IMU 会测量运动角速度和加速度,相机、激光传感器、深度传感器会读取环境的某种观测数据,它们测量的都是间接的物理量而不是直接的位置数据,只能通过通过这些间接的物理量推算运动主体的位姿(Pose,包括位置和方向)。就移动端设备而言,我们更关注来自设备摄像头的图像信息、IMU 运动信息(可能还会有深度传感器信息),理论上可以通过这些数据解算出运动主体精确的位姿信息和环境地图,但遗憾的是来自各类传感器的数据都不是完全准确的,都带有一定程度的噪声(误差),这使问题变得复杂化,因为噪声数据会导致计算误差,而这种误差会随着时间推移快速累积(试想一个微小的误差以每秒 60 次的速度迅速放大),很快就会导致定位和建图完全失效。

以图像数据为例,来自摄像头的图像本质上是某个现实场景在相机成像平面上留下的一个投影,它以二维的形式记录了三维的场景,因此,图像与现实场景相比少了一个深度方向

的维度，所以仅凭一幅图像无法恢复（计算）拍摄该图像时相机的位姿，必须通过移动相机获取另一幅图像形成视差才能恢复相机运动，这也就是移动设备在使用 AR 应用时必须移动才能实现正确运动跟踪的原因。

图像在计算机中以矩阵的形式进行存储和描述，为精确匹配图像中的像素与现实世界中的环境点，图像数据还要进行校准才能进入 SLAM 系统中，校准分为相机光度校准与几何校准两部分。

光度校准：光度校准涉及相机底层技术，通常要求 OEM 厂商参与。因为光度校准涉及图像传感器本身的细节特征及内部透镜所用的涂层材料特性等，光度校准一般用于处理色彩和强度的映射。例如，正在拍摄遥远星星的望远镜连接的相机需要知道传感器上一像素光强度的轻微变化是否确实是源于星星的光强变化或者仅仅来源于传感器或透镜中的像差。光度校准对于 AR 运动跟踪的好处是提高了传感器上的像素和真实世界中点的匹配度，因此可使视觉跟踪具有更强的稳健性及更少的错误。

几何校准：以普通相机为例，几何校准使用针孔相机模型来校正镜头的视野和镜筒畸变。由于镜头的加工精度、安装工艺等缘故所有采集到的图像都会产生变形，软件开发人员可以在没有 OEM 厂商帮助的情况下使用棋盘格和公开的相机规格进行几何校正，如图 1-3 所示。

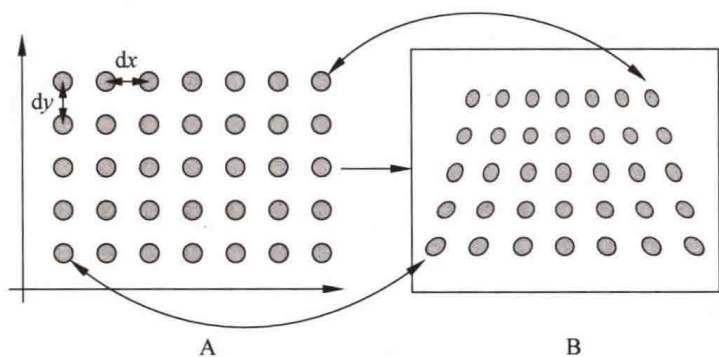


图 1-3 对图像信息进行几何校准示意图

对 SLAM 技术而言，光度校准确定了真实物理点与成像点颜色与强度映射，镜头出厂后不会再发生变化，而由于镜头加工精度、安装工艺等所导致的畸变却会影响真实物理点与成像点的位置对应关系，它们对整个 SLAM 系统的精度影响非常大，必须进行预先处理，常见的畸变有桶形失真和枕形失真两种，如图 1-4 所示。

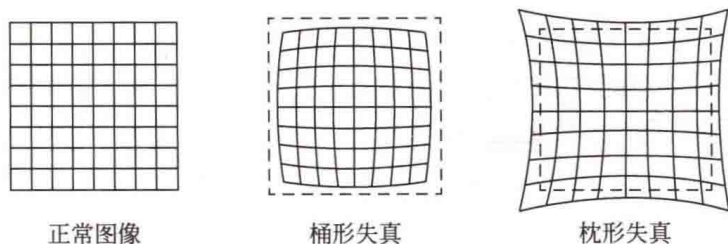


图 1-4 图像畸变类型

除了图像数据噪声和畸变外, IMU 数据也不准确, IMU 产生数据的频率非常高, 通常能达到每秒 100 ~ 1000 次, IMU 误差主要来自噪声 (Bias and Noise)、尺度因子 (Scale Errors) 和轴偏差 (Axis Misalignment), 在极短时间内, 可以信赖 IMU 数据, 但由于 IMU 数据频率高, 误差也会在短时间内快速累积, 因此也需要对 IMU 数据进行校准。

深度传感器也会带来误差, 除了深度传感器本身的系统误差和随机误差, 环境中的透明物体、红外干扰、材质反光属性等都会增加深度值的不确定性, 也需要进行误差校准和外点剔除。

1.2.2 前端里程计

基于视觉的 SLAM 前端里程计通常称为视觉里程计 (Visual Odometry, VO), 其主要任务是估算相邻图像间的运动, 建立局部地图。移动设备通常配备有 IMU 传感器, 因此, 在移动设备中进行的前端里程计融合了视觉、IMU 各传感器数据, 可以提供更高精度和稳健性。

里程计通过定量地分析图像与空间点的几何对应关系、相对运动数据, 能够估计相机运动, 并恢复场景空间结构, 前端里程计只计算相邻时刻的相机运动, 不关联之前的数据, 因此不可避免地会出现累积漂移 (Accumulation Drift), 以这种方式工作, 先前时刻的误差会传递到下一时刻, 导致经过一段时间后, 估计的运动轨迹误差越来越大, 原本直的通道变成了斜的, 原本圆的场景变成了椭圆, 为解决累积漂移问题, 需要引入额外的抑制技术: 后端优化和回环检测。

1.2.3 后端优化

概括地讲, 后端优化主要指处理 SLAM 过程中的噪声问题。虽然希望所有输入的数据都是准确的, 然而现实中, 再精确的传感器也带有一定的噪声。便宜的传感器测量误差较大, 昂贵的则较小, 有的传感器还会受磁场、温度、红外线的影响, 所以除了解决“从相邻时刻估计出相机运动”之外, 还要关心这个估计带有多大的噪声及如何消除这些误差。后端优化要考虑的问题就是如何从这些带有噪声的数据中, 估计整个系统的状态, 以及估计的这种状态不确定性的。这里的状态既包括运动主体本身的轨迹, 也包含地图。在 SLAM 框架中, 前端给后端提供待优化的数据, 以及这些数据的初始值, 而后端负责整体的优化过程, 面对的只有数据, 而不必关心这些数据到底来自什么传感器。在 ARCore 中, 前端和计算机视觉研究领域更为相关, 例如图像的特征提取与匹配等, 也包括图像数据与 IMU 或者 ToF 数据的融合, 后端则主要是滤波与非线性优化算法。通过后端优化, 能够比较有效地抑制误差累积, 将整个 SLAM 系统维持在一个可接受的精度范围之内。

1.2.4 回环检测

回环检测, 又称闭环检测 (Loop Closure Detection), 主要解决位姿估计随时间漂移的问题。

假设实际情况下，运动主体经过一段时间运动后回到了原点，但是由于漂移，它的位置估计值却没有回到原点。如果有某种手段感知到“回到了原点”这件事，或者把“原点”识别出来，再把位置估计值“拉”过去，就可以消除漂移了，这就是所谓的回环检测。回环检测与“定位”和“建图”二者都有密切的关系。事实上，我们认为，地图存在的主要意义是为了让运动主体知晓自己到达过的地方。为了实现回环检测，需要让运动主体具有识别曾到达过场景的能力。实现回环手段有很多，例如可以在环境中某个位置设置一个标志物（如一张二维码图片），只要它识别到了这个标志，就知道自己回到了原点，但是，该标志物实际上是对环境位置的一种人为标识，对应用环境提出了限制，而我们更希望通过运动主体自身携带的传感器——图像数据——来完成这一任务。例如，可以通过判断图像间的相似性来完成回环检测，这一点和人类是相似的，当我们看到两张相似图片时，容易辨认它们来自同一个地方，当然，在计算机中判断两张图像是否相似这个任务比我们想象的要难得多，这是一个典型的人觉得容易而计算机觉得难的命题。

如果回环检测成功，可以显著地减小累积误差，所以回环检测，实际上是一种计算图像数据相似性的算法。由于图像的信息非常丰富，使正确检测回环的难度也降低了不少。在检测到回环之后，我们会把“A与B是同一个点”这样的信息通知后端优化算法，后端根据这些新的信息，把轨迹和地图调整到符合回环检测结果的样子。这样，如果有充分而且正确的回环检测，就可以消除累积误差，得到全局一致的轨迹和地图。

1.2.5 建图

建图（Mapping）是指构建地图的过程。地图是对环境的描述，但这个描述并不是固定的，需要视 SLAM 的应用而定。相比于前端里程计、后端优化和回环检测，建图并没有一个固定的形式和算法。一组空间点的集合可以称为地图，一个三维模型也是地图，一个标记着城市、村庄、铁路、河道的图片亦是地图。地图的形式随 SLAM 的应用场合而定，按照地图用途，地图可以分为度量地图与拓扑地图两种。

度量地图强调精确地表示地图中物体的位置关系，通常用稀疏（Sparse）与稠密（Dense）对它们进行分类。稀疏地图进行了一定程度的抽象，并不需要表达所有的物体。例如，选择一部分具有代表意义的东西，称为路标（Landmark），那么一张稀疏地图就是由路标组成的地图，而不是路标的部分就可以忽略掉。相对地，稠密地图着重于建模所有看到的東西。对于定位来讲，稀疏路标地图就足够了，而对于导航则需要稠密地图。

拓扑地图（Topological Map）是一种保持点与线相对位置关系正确而不一定保持图形形状与面积、距离、方向正确的抽象地图，拓扑地图通常用于路径规划。

在移动端 AR 设备中，主要使用稀疏度量地图实现对物理环境的映射，但在搭载有 ToF 传感器的设备上，可以通过物理方法得到场景深度值，利用这些深度信息，通过算法就可以建立对环境的映射（对环境场景几何表面进行重建），因此可以允许虚拟对象与真实物体发生碰撞及遮挡，如图 1-5 所示。

环境映射是实现真实物体与虚拟物体碰撞与遮挡关系的基础,在对真实环境进行三维重建后虚拟对象就可以与现实世界互动,实现碰撞及遮挡。



图 1-5 对场景重建后实现遮挡示意图

通过 SLAM 技术,不仅能解算出运动主体位姿,还可以建立起周围环境的地图,对齐虚拟数字世界与真实物理世界,因此,就能以正确的视角渲染虚拟元素,实现与真实物体一样的透视、旋转、物理模拟现象,营造虚实融合的 AR 体验。

1.3 AR Foundation 概述

2017 年,苹果(Apple)公司与谷歌(Google)公司相继推出了 AR 开发 SDK 工具包 ARKit 和 ARCore,分别对应 iOS 平台与 Android 平台 AR 开发。ARKit 和 ARCore 推出后,极大地促进了 AR 在移动端的普及发展。自此之后,AR 开始进入普通消费场景中,并迅速在各领域落地应用。由于当前移动手机操作系统主要为 iOS 和 Android 两大阵营,ARKit 和 ARCore 分别服务于各自系统平台,这意味着软件产商在开发移动 AR 应用时必须使用 ARKit 开发一个 iOS 版本、使用 ARCore 开发一个 Android 版本,这无疑增加了开发与成本。

ARCore 提供了 Android、Android NDK、Unity、Unreal 开发包,ARKit 官方只提供了 XCode 开发 API,这也增加了利用其他工具进行开发的开发者学习成本。为进一步提升 XR 开发能力和降低开发成本,Unity 构建了一个 AR 开发平台,这就是 AR Foundation,这个平台架构于 ARKit 和 ARCore 之上,其目的是利用 Unity 良好的跨平台能力构建一种与平台无关的 AR 开发环境,即做了一个中间软件抽象层,换言之,AR Foundation 对 ARKit 和 ARCore 进行了再次封装,提供给开发者一致的开发界面,屏蔽了底层硬件差异,依据 AR 应用发布平台自动选择合适的底层 SDK 版本。

AR Foundation 是 ARKit 和 ARCore 的软件抽象层,虽然最终都使用 ARKit SDK 或 ARCore SDK 实现功能,但已经将二者之间的使用差异抹平,开发者无须关心底层软件及硬件差异。由于考虑到提供公共的使用界面,因此对底层 SDK 特异性的功能部分 AR Foundation 做了取舍,如 ARCore 的云锚点功能,AR Foundation 不提供直接支持,除此之外,函数方法

调用与原生 API 也不同。

AR Foundation 的目标并不局限于整合 ARKit 与 ARCore，它的目标是构建一个统一、开放的 AR/MR 开发平台，经过几年的发展，AR Foundation 也已经纳入 Magic Leap、HoloLens AR 眼镜端开发的相关 SDK，进一步丰富了 AR/MR 开发能力，不仅支持移动端 AR 设备，也支持穿戴式 AR 设备开发。

综上所述，AR Foundation 并不提供 AR/MR 的底层开发 API，这些与平台相关的 API 均由第三方（如 ARKit、ARCore、Magic Leap）提供，它是一个软件抽象层，通过提供统一的使用界面和功能封装，进一步降低了 AR/MR 跨设备、跨平台开发门槛。

1.3.1 AR Foundation 与 ARCore

AR Foundation 提供了独立于平台的脚本 API 和 MonoBehaviour 组件，因此，开发者可以通过 AR Foundation 使用 ARCore 和 ARKit 公有的核心功能构建同时适用于 iOS 和 Android 两个平台的 AR 应用程序，只需一次开发，就可以部署到两个平台的设备上，不必做任何改动。

事实上，谷歌公司也为 Unity 开发提供了 ARCore SDK for Unity 插件包，在 AR Foundation 发展起来以后，谷歌公司已经将对 Unity 的开发支持转移到 AR Foundation 上，因此，笔者不建议再继续使用 ARCore SDK for Unity 插件包进行 ARCore 开发^①。

AR Foundation 架构于 ARCore 和 ARKit 之上，相比于使用原生 SDK 进行开发，多了一个软件抽象层，其与 ARCore、ARKit 的关系如图 1-6 所示。

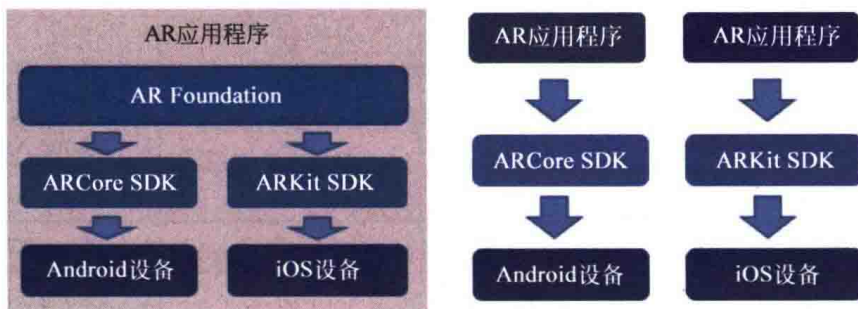


图 1-6 AR Foundation、ARCore、ARKit 相互关系图示

1.3.2 AR Foundation 支持的功能

AR Foundation 与 ARCore、ARKit 都处于快速发展中，支持的功能也从核心的运动跟踪、环境理解向周边扩展，对 AR 应用开发支持越来越完善，它们的主要功能及对比如表 1-1 所示。

^① 谷歌公司目前已经有计划要停止 ARCore SDK for Unity 插件包的继续维护，在 Unity 引擎上开发 ARCore 应用，应当选择 AR Foundation。

表 1-1 AR Foundation、ARCore、ARKit 功能及对比

支持功能	AR Foundation	ARCore	ARKit
垂直平面检测	✓	✓	✓
水平平面检测	✓	✓	✓
特征点检测	✓	✓ + 支持特征点姿态	✓
光照估计	✓	✓ +Color Correction	✓ +Color Temperature
射线检测 (Raycasting, 对特征点与平面)	✓	✓	✓
图像跟踪	✓	✓	✓
三维物体检测与跟踪	✓	—	✓
环境光探头 (Environment Probes)	✓	✓	✓
世界地图 (World Maps)	✓	—	✓
人脸跟踪 (Pose、Mesh、Region、Blendshape)	✓	✓	✓ (iPhone X 及更高型号)
云锚点 (Cloud Anchors)	✓	✓	—
远程调试 (Editor Remoting)	MARS	✓ -Instant Preview	✓ -ARKit Remote
模拟器 (Editor Simulation)	✓	—	—
URP 支持 (支持使用 ShaderGraph)	✓	✓	✓
摄像头图像 API (Camera Image)	✓	✓	—
人体动作捕捉 (Motion Capture)	✓	—	✓ (iPhone XR 及更高型号)
人形遮挡 (People Occlusion)	✓	—	✓ (iPhone XR 及更高型号)
多人脸检测	✓	✓	✓ (iPhone XR 及更高型号)
多人协作 (Collaborative Session)	✓	—	✓ (iPhone XR 及更高型号)
深度图 (Depth API)	✓	✓	✓ (iPhone XR 及更高型号)
地理位置 AR	✓	—	✓ (部分地区)
多图像识别	✓	✓	✓

AR 应用是计算密集型应用,对计算硬件要求较高,AR 应用启动后,即使在应用中不渲染虚拟对象,其底层运动跟踪功能也在对环境图像、特征点跟踪进行实时解算,由于移动端硬件设备资源的限制,一些高级 AR 特性只能在最新的处理器(包括 CPU 和 GPU)上才能运行,如 CPU 纹理图获取、深度 API 等。

1.3.3 AR Foundation 功能概述

AR Foundation 功能的实现依赖于其底层 SDK,同时,AR Foundation 能实现的功能也

与底层 SDK 所在平台相关，如 ARKit 有 WorldMap 功能，而 ARCore 没有，因此，即使 AR Foundation 支持 WorldMap 功能，该功能也只能在支持 ARKit 运行的 iOS 平台上才有效，编译到 Android 平台则会出现错误。对于 ARKit 和 ARCore 都支持的公共功能，AR Foundation 可以在编译时根据平台选择无缝切换所用底层 SDK，达到一次开发跨平台部署的目的。除此之外，AR Foundation 也提供了很多辅助功能和组件以方便开发者使用。

对于 Android 平台的 ARCore 开发，AR Foundation 主要支持的功能如表 1-2 所示。

表 1-2 AR Foundation 支持的功能

功 能	描 述
世界跟踪 (World Tracking)	在物理空间中跟踪用户设备的位置和方向 (姿态)
平面检测 (Plane Detection)	对水平与垂直平面进行检测
锚点 (Anchor)	对特定点的姿态进行跟踪
光照估计 (Light Estimation)	对物理环境中的光照强弱及方向进行估计
人脸跟踪 (Face Tracking)	检测并跟踪人脸
图像跟踪 (Image Tracking)	跟踪物理空间中的二维图像
虚拟物体操作	对 AR 场景中的虚拟物体进行选择、平移、旋转、缩放操作
摄像头图像 API	提供摄像头图像底层支持，方便开发人员开发计算机视觉应用

1.3.4 AR Foundation 体系架构概述

虽然 AR Foundation 是在底层 SDK API 之上进行抽象封装，但为实现 AR 应用开发跨平台能力 (平台无关性)，AR Foundation 搭建了一个开放性的灵活架构体系，使这个架构能够容纳各类底层 SDK，能支持当前及后续其他底层 AR/MR SDK 的加入，宏观上看，AR Foundation 希望构建一个能开发各类 AR/MR 应用的统一平台。

为屏蔽底层软件和硬件差异，AR Foundation 建立在一系列的子系统 (Subsystem) 之上。子系统负责处理与平台相关的工作，进行软件抽象，屏蔽硬件差异，实现功能模块的平台无关性。如 XRPlaneSubsystem 子系统负责实现平面检测、显示功能，在编译时，AR Foundation 根据所选平台自动调用不同底层的 SDK，从开发者角度看，只需调用 XRPlaneSubsystem 功能，而不用关心最终这个实现是基于 iOS 还是 Android，对平台透明化。

这种架构对上提供了与平台无关的功能，对下可以在以后的发展中纳入不同的底层 SDK，从而实现最终的一次开发，跨平台部署的目标，更详细内容将在第 2 章中进行阐述。

1.3.5 基本术语

1. 运动跟踪

在二维和三维空间中跟踪用户的运动并定位其位置是任何 AR 应用的基础，也是营造虚

实融合体验的关键,运动跟踪(Tracking)又称为世界跟踪,指AR设备实时解算其在物理世界中的相对位置和方向。当设备在现实世界中运动时,AR Foundation通过SLAM技术来理解移动设备周边环境、定位设备姿态。SLAM算法会检测从摄像头图像中捕获的视觉差异特征(称为特征点),并使用这些特征来计算其位置变化,这些视觉信息与设备IMU惯性测量结果结合,估测摄像头随着时间推移而相对于周围世界的姿态变化。

通过将渲染三维内容的虚拟相机的姿态与设备摄像头的姿态对齐,就能够从正确的透视角度渲染虚拟内容,渲染的虚拟图像可以叠加到从设备摄像头获取的图像上,让虚拟内容看起来就像真实世界的一部分。

2. 可跟踪对象

可跟踪对象(Trackable)即可以被AR设备检测和/或跟踪的真实特征,例如特征点、平面、人脸、二维图像、锚点等。

3. 特征点

SLAM使用摄像头采集的图像来分析和跟踪环境中用于构建环境地图的特定点(Feature Point),例如木纹表面的纹理点、书本封面的图像,通常是视觉差异点,特征点包含了被观察到的图像视觉特征集合。

4. 会话

会话(Session)用于管理AR系统的状态,是AR功能的入口,是AR应用中非常重要的概念。会话负责处理整个AR应用的生命周期,控制AR系统根据需要开始和暂停相机帧的采集、初始化、释放资源等。在开始使用AR API时,可通过ARSessionState状态值来检查当前设备是否支持AR功能。

5. 会话空间

会话空间(Session Space)即为AR会话初始化后建立的坐标空间,会话空间的原点(0,0,0)为AR应用完成初始化时设备摄像头所在的位置,AR设备跟踪的坐标信息都处在会话空间中,因此在使用时,需要将其从会话空间转换到Unity空间,类似于模型空间和世界空间的转换,AR Foundation默认会执行从会话空间到Unity世界空间的转换。

6. 射线检测

AR Foundation利用射线检测(Ray Casting)获取对应于手机屏幕的(x,y)坐标(通过点按或应用支持的任何其他交互方式),将一条射线投射到摄像头的视野中,返回这条射线贯穿的任何平面或特征点及碰撞位置在现实世界空间中的姿态,通过射线检测可以与AR场景中的物体交互。

7. 增强图像

使用增强图像(Augmented Image)(图像检测)可以构建能够响应特定二维图像(如产

品包装或电影海报)的AR应用,用户将设备摄像头对准特定图像时触发AR体验,如将设备摄像头对准电影海报后加载虚拟模型或者虚拟场景,实现对现实图像的增强。使用增强图像时可离线创建图像数据库,也可以在运行时实时添加参考图像,在AR应用运行时,AR Foundation将检测和跟踪这些图像,并返回其姿态。

8. 体验共享

借助于ARCore中的云锚点(Cloud Anchor)功能,可以创建适用于iOS和Android设备的多人共享AR应用,实现AR体验的多人共享(Sharing)。使用云锚点时,一台设备可以将锚点及其附近的环境特征点信息发送到云端进行托管,通过锚点解析可以与同一环境中Android或iOS设备上的其他用户共享相关数据,从而让用户能够同步拥有相同的AR体验。

9. 平面

为营造更加真实的AR体验,AR场景中大部分内容需要依托于平面(Plane)的建立。AR Foundation中平面可分为水平、垂直两类,平面描述了真实世界中的一个二维平面,包括平面的中心点、平面的X轴和Z轴方向长度、组成平面多边形的顶点等数据。检测到的平面还分为3种状态:正在跟踪、可恢复跟踪、永不恢复跟踪,不在跟踪状态的平面,平面包含的信息可能不准确。随着环境探索的进行,AR Foundation也可能会自动合并两个或者多个平面。

10. 姿态

在AR Foundation中,姿态(Pose)可描述物体位置与方向属性,在AR Foundation API中的姿态可以被认为与OpenGL的模型矩阵或DirectX的世界矩阵一致。随着AR Foundation对环境理解的不断加深,它会自动调整坐标系以便与真实世界保持一致,因此每一帧中,虚拟物体的姿态都有可能发生变化。

11. 光照估计

光照估计(Light Estimate)描述了通过设备摄像头采集的图像信息估计真实环境中光照的过程,通过计算机视觉图像算法可以获取当前设备摄像头采集图像中的光照强度、颜色分量及光照方向,使用光照估计信息渲染虚拟对象照明效果会更真实,并且可以根据光照方向调整AR中虚拟物体的阴影方向,增强虚拟物体的真实感。

1.4 支持的设备

由于SLAM对传感器的极度敏感性和AR对移动设备性能的要求,ARCore并不能在当前所有流行的移动设备上运行(包括手机与平板),而是对设备有强制认证要求,只有符合ARCore要求的设备才能运行AR应用,谷歌公司会在其官网上更新所有受到支持的设备列表,目前,在中国,这些受支持的设备如表1-3所示。

表 1-3 受支持的设备明细表

设备 厂商	设备 型号	型 号 列 表
荣耀	手机	Honor 10、Honor Magic 2、Honor V20、Maimang 7
华为	手机	Mate 20、Mate 20 Pro、Mate 20 X、Nova 3、Nova 3i、Nova 4
		P20、P20 Pro、P30、P30 Pro
		Porsche Design Mate RS、Porsche Design Mate 20 RS
OPPO	手机	A53 5G、A93 5G、Find X3、Find X3 Pro、F19 Pro+、K5、K7x、K9 5G
		Reno、Reno 10x Zoom、Reno Ace、Reno Ace 2、Reno Z、Reno2、Reno2 Z
		Reno4 SE 5G、Reno5 5G、Reno5 K 5G、Reno5 Pro 4G、Reno5 Pro 5G、Reno5 Pro+ 5G、Reno6 Pro+ 5G
三星	手机	Galaxy Note 9、Galaxy S9、Galaxy S9 Exynos、Galaxy S9 Qualcomm、Galaxy S9+ Exynos、Galaxy S9+ Qualcomm、Galaxy S10e Exynos、Galaxy S10e Qualcomm
		Galaxy S10 Exynos、Galaxy S10 Qualcomm、Galaxy S10+ Exynos、Galaxy S10+ Qualcomm、Galaxy S10 5G
VIVO	手机	iQOO、iQOO 5 5G、iQOO 5 Pro 5G、iQOO Neo3 5G、iQOO U1、iQOO Z1x、NEX S、NEX 3 4G/5G
		S7 5G、X23、X50 5G、X50 Pro、X50 Pro+、V2046A、V2047A、V2056A、Y51s、Y70s
Realme	手机	Q、Q2 Pro 5G、Q3 5G、X、X Lite、X2、X2 Pro、X3、X50 Pro Player、X7 5G、X7 Pro 5G
		V5 5G、真我 V13 5G、真我 GT、realme GT Neo
小米	手机	Mi Mix 2S
		Mi Mix 3
		Mi 8、Mi 8 SE、Mi 9、Mi 10 Ultra

表 1-3 是一张动态更新的表，一般而言，新发布的高端 Android 手机都会支持 ARCore，另有一些设备并未在国内标注，但在国际名单上有标注，这些设备也可能会支持 ARCore，但在购买或者使用时，最好先查询相关产品的规格参数^①。

1.5 开发环境准备

开发 ARCore 设备的 AR 应用需要使用较多的工具软件，而且工具软件之间具有相关性，开发环境配置较容易出现問題，因此将详细介绍开发环境所需要的硬件、软件需求和配置。

① 由于 iOS 设备只支持 ARCore 的云锚点及人脸检测跟踪技术，这里并未将其列入。

1.5.1 所需硬件和软件

本书使用 Windows 10、Visual Studio 2019、Unity 2020.3 LTS、Android Studio 2020.3 开发 ARCore 设备的 AR 应用,为确保能高效地进行开发工作^①,建议的开发计算机硬件配置如表 1-4 所示。

表 1-4 开发计算机硬件配置建议表

硬件名称	描 述
CPU	Intel 桌面 i7 第 6 代 (6 核) 或 AMD Ryzen 5 1600 (6 核, 12 线程) 及以上
GPU	支持 DX12 的 NVIDIA GTX 1060 或者 AMD Radeon RX 480 (2GB) 及以上
内存	支持 DDR4 2660 及以上频率的 16GB 内存及以上
硬盘	240GB 固态硬盘作为操作系统及各开发工具安装盘
显示器	1920 × 1080 及以上分辨率的 24 英寸显示器及以上
USB	至少 1 个 USB 接口

开发 AR 应用时本书使用的是 Windows 10 专业版操作系统,各主要工具软件及下载网址如表 1-5 所示。

表 1-5 开发计算机所需软件及下载网址表

软 件	下 载 网 址
Windows 10 专业版 64 位	https://www.microsoft.com/zh-cn/software-download/Windows10
Visual Studio 2019 (16.8 或更高版本)	https://visualstudio.microsoft.com/zh-hans/downloads/
Unity 2020.3 LTS	https://unity.cn/releases/lts/2020
Android Studio 2020.3	https://developer.android.google.cn/studio/
Google Play Services for AR	https://github.com/google-ar/arcore-android-sdk/releases

1.5.2 软件安装

首先正确安装 Windows 10 专业版最新版本,并更新到最新状态,确保硬件均已正确驱动。

1. Visual Studio 2019 安装

使用 Visual Studio Installer 安装最新版本 (16.8 及以上) 的 Visual Studio 2019^② (独立安装

^① 根据笔者使用经验,由于所使用工具均为重量级软件,完整的 AR 应用开发环境搭建完成大约需要 80GB 硬盘空间,使用的计算机性能不好会严重影响开发效率。稳妥起见,本书采用 Unity 2020.3 LTS 版本,建议读者也不要盲目使用 Unity 新版本。

^② 为方便描述,下文中 Visual Studio 2019 和 Visual Studio 简称为 VS。

Visual Studio 2019, 不建议将其作为 Unity 软件工具组件的一部分安装)。由于只使用 VS 进行代码编辑, 并不使用其编译发布, 所以按正常步骤完成 VS 安装即可。

2. Unity 2020.3 安装

建议使用 Unity Hub 安装 Unity 2020.3, Unity Hub 是专用于 Unity 软件各版本安装、管理、卸载的工具, 利用该工具可以同时计算机中安装多个版本的 Unity 软件, 而且可以随时加载或者卸载各版本 Unity 的工作模块。安装 Unity 软件时, 选择好安装的版本, 取消默认的 Microsoft Visual Studio Community 2019 工作模块, 并确保勾选 Android Build Support 工作模块, 如图 1-7 所示。



图 1-7 Unity 软件安装时勾选需要的工作模块

3. Android Studio 2020 安装

对开发 AR 应用而言, 本步骤为可选步骤, 为了方便配置 Android SDK, 建议安装, 并且本书第 13 章使用了 Android Studio。安装过程比较简单, 按正常步骤完成 Android Studio 安装即可。

4. Google Play Services for AR 安装

本步骤为可选步骤, Google Play Services for AR 是 ARCore 运行的服务器端, 它负责 SLAM 运动跟踪、环境理解、人脸检测等所有工作, 一般而言, 受 ARCore 支持的设备 (在谷歌 ARCore 支持设备列表中) 会预先安装 Google Play Services for AR, 该应用没有界面和图标 (可以通过手机的应用列表查看), 如果没有预先安装, 根据手机制造商的不同, 则可以到

其官方应用商店下载并安装，开发人员也可以到表 1-5 中所示地址手动下载并安装。手机端必须安装 Google Play Services for AR 才能运行 ARCore 应用。

1.6 ARCore 初体验

在准备好所有所需软硬件之后，本节将构建一个 ARCore 应用，体验 AR 应用从工程设置到导出部署全过程。同时，为更好利用 Unity URP 渲染管线（Universal Render Pipeline, URP）在移动端应用渲染的优势，本书所有讲述和示例均基于 URP 渲染管线。

1.6.1 工程创建

选择 Windows 开始菜单中 Unity Hub 或者单击桌面的 Unity Hub 图标启动 Unity Hub，单击右侧“新建”下拉菜单，选择 Unity 版本创建一个新项目，如图 1-8 所示。

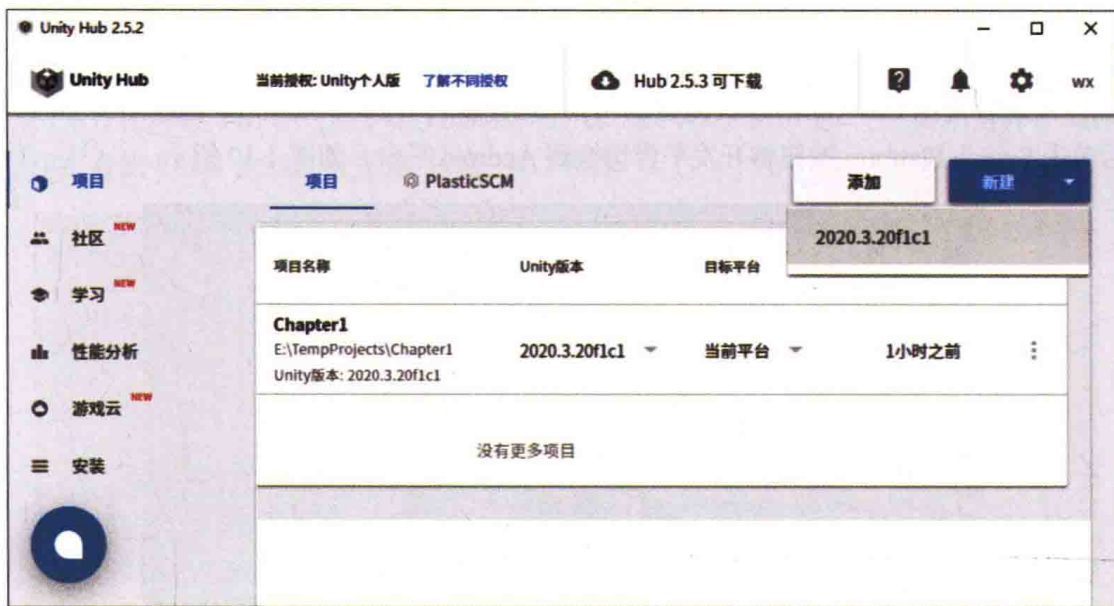


图 1-8 创建新 Unity 工程项目

在弹出的创建新项目面板中，选择 3D 模板、填写项目名称、选择工程保存位置，取消“启用 PlasticSCM”复选框，如图 1-9 所示，单击“创建”按钮开始工程创建^①。

^① 在图 1-9 中，可以看到一个 Universal Render Pipeline 应用模板，也可以直接使用该模板创建 URP 应用，但目前使用这个模板创建应用会引入一个案例场景和若干工程文件，可能会造成初学者困惑，同时，为方便 Unity AR 应用开发者将已有 Built-in 工程转换到 URP 工程，我们选择从 3D 模板创建 URP AR 应用，方便开发者了解详细的 URP 配置过程。

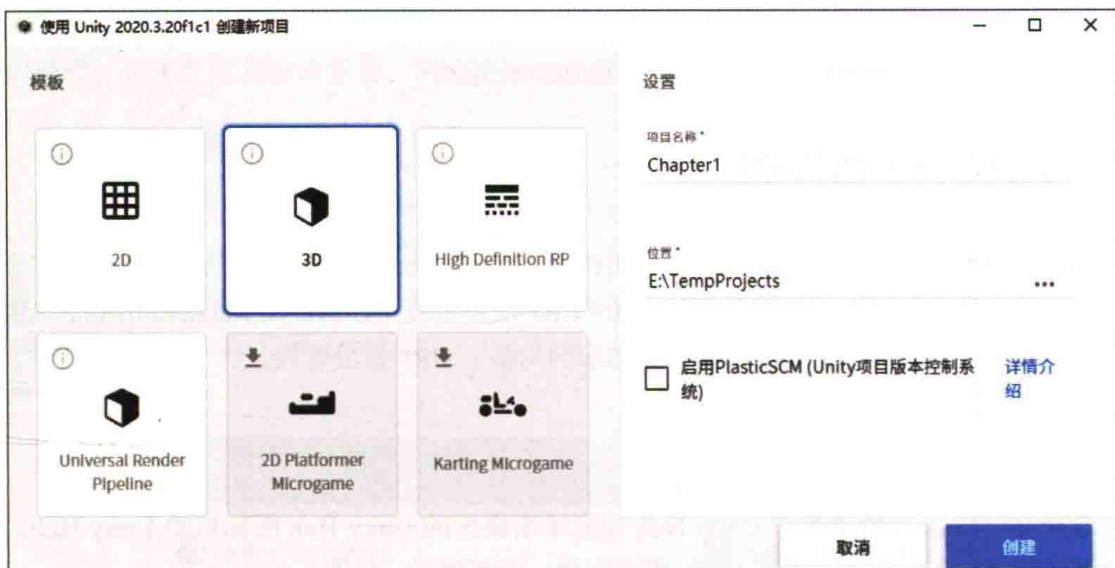


图 1-9 选择 Unity 工程模板，填写项目信息

待 Unity 窗口打开后，将当前场景名修改为 Helloworld，从菜单栏中选择 File → Build Settings (或者使用 Ctrl+Shift+B 快捷键)，打开设置窗口。选择 Platform 下的 Android 选项，然后单击 Switch Platform 按钮将开发平台切换到 Android 平台，如图 1-10 所示。

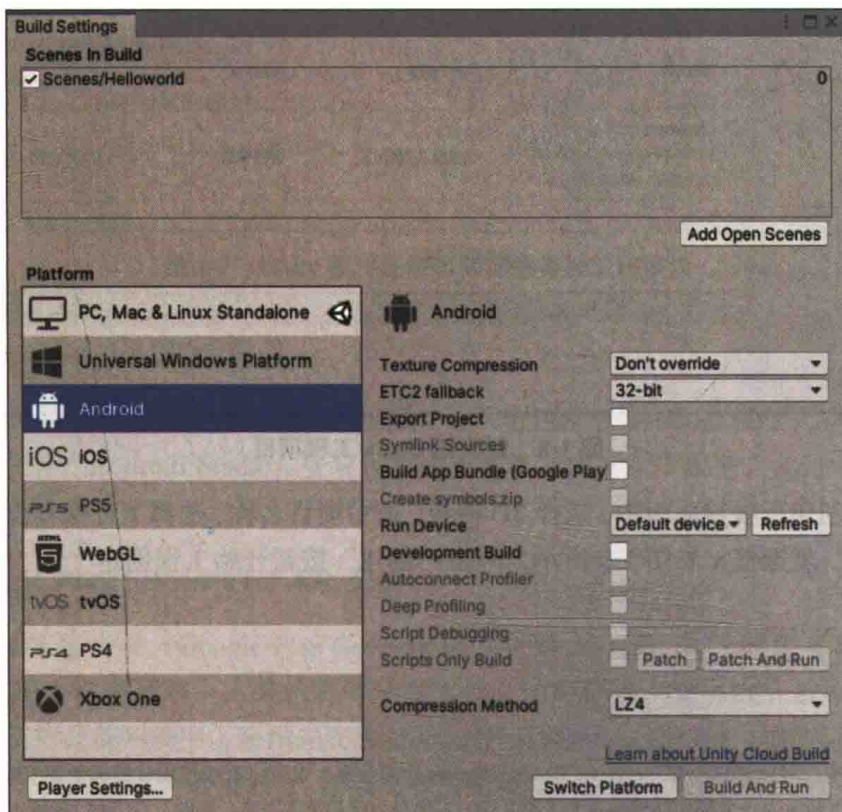


图 1-10 将开发发布平台切换到 Android

1.6.2 工具包导入

在 Unity 菜单栏中, 依次选择 Window → Package Manager, 打开 UPM 管理器窗口, 单击该窗口左上角的下拉菜单, 选择 Unity Registry 选项, 此时会显示所有 Unity 中注册的工具包(插件), 如图 1-11 所示。

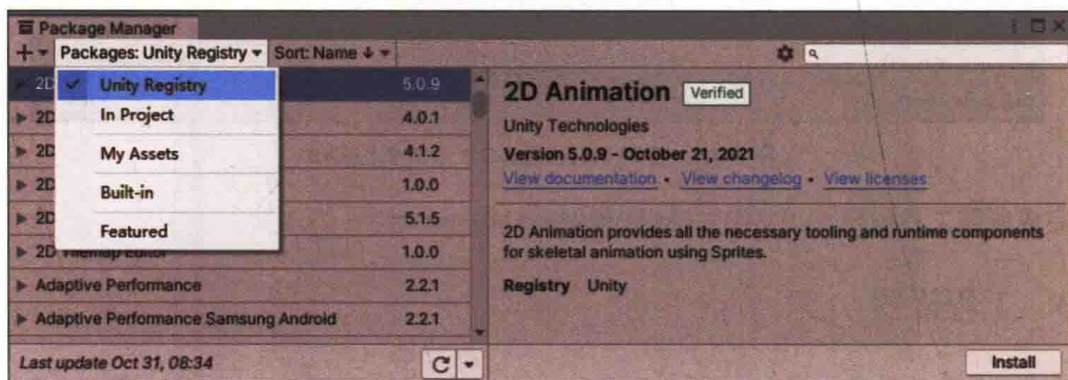


图 1-11 显示 Unity 中注册的所有插件

滑动左侧列表滚动条, 找到 AR Foundation、ARCore XR Plugin 工具包, 分别选择各工具包, 然后单击右侧 Install 按钮进行安装, 如图 1-12 所示^①。

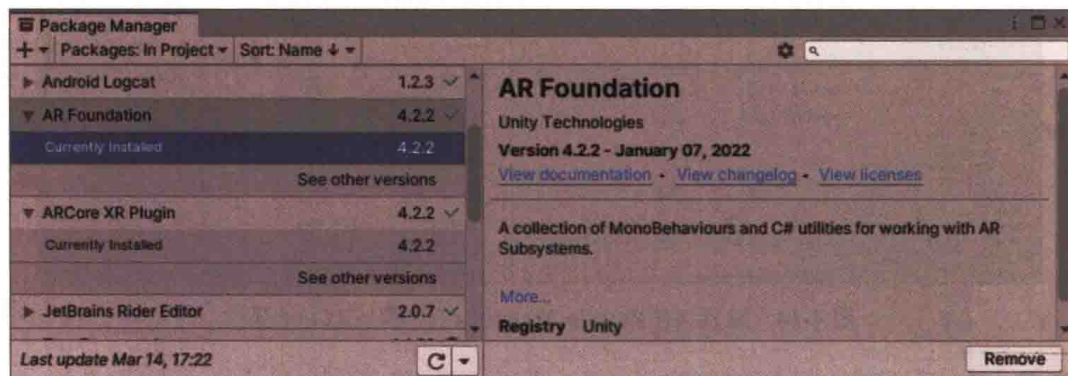


图 1-12 安装 AR Foundation 和 ARCore XR Plugin 插件

由于我们使用的是 URP 管线, 还需要导入 Universal RP 工具包, 滑动 UPM 管理器窗口左侧列表滚动条, 找到 Universal RP 工具包并进行安装, 如图 1-13 所示。

^① 实际项目开发时, 建议选择带 Verified 标识的稳定版本插件。若想体验最新版本, 则可以通过各插件下方的 See other versions 展开其他版本, 选择最新版本安装, 但最新版本可能存在兼容和稳定性问题。



图 1-13 安装 Universal RP 渲染管线插件

完成所需工具包导入，关闭 UPM 管理器窗口。

1.6.3 工程设置

在 Unity 菜单栏中，依次选择 Edit → Project Settings 打开 Project Settings 窗口，选择 XR Plug-in Management 项，勾选 Initialize XR on Startup 和 ARCore 复选框，然后选择 XR Plug-in Management 下的 ARCore 子项，将 Requirement 属性设置为 Required 值、将 Depth 属性设置为 Optional 值，如图 1-14 所示。

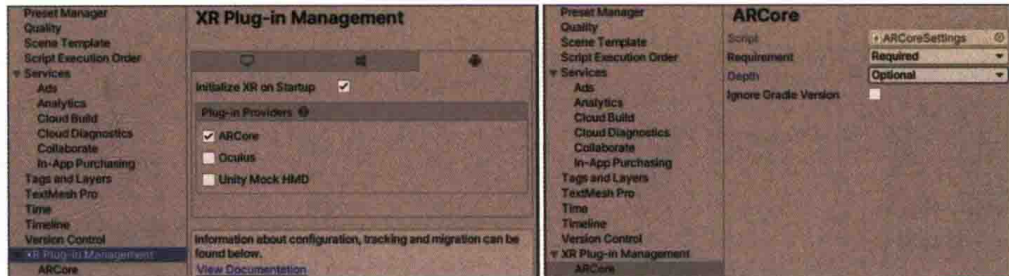


图 1-14 设置 XR Plug-in Management 和 ARCore 属性

在 Unity 工程窗口 (Project 窗口) Assets 目录下新建一个文件夹，命名为 Settings，在 Settings 文件夹上右击，在弹出的级联菜单中依次选择 Create → Rendering → Universal Render Pipeline → Pipeline Asset (Forward Renderer) 创建新的渲染管线，并命名为 ARCorePipeline，如图 1-15 所示。

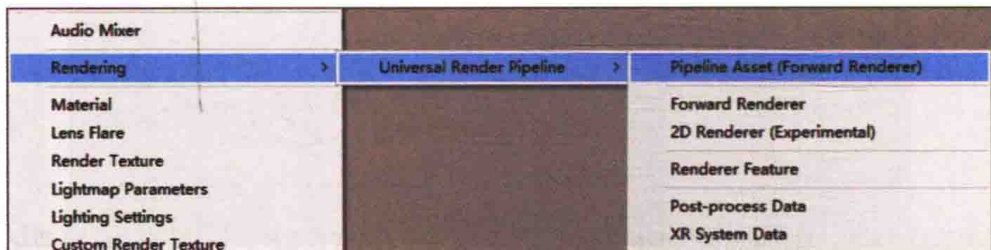


图 1-15 创建 URP 渲染管线

该步骤会新创建两个文件，即 ARCorePipeline.asset 和 ARCorePipeline_Renderer.asset。选择 ARCorePipeline_Renderer.asset 文件，在 Unity 属性窗口（Inspector 窗口）中查看该文件属性，单击其下方的 Add Rednerer Feature 按钮，在弹出的菜单中选择 AR Background Renderer Feature，如图 1-16 所示，通过添加该特性，AR Foundation 就可以将摄像头捕获的场景图像渲染为 AR 应用背景。

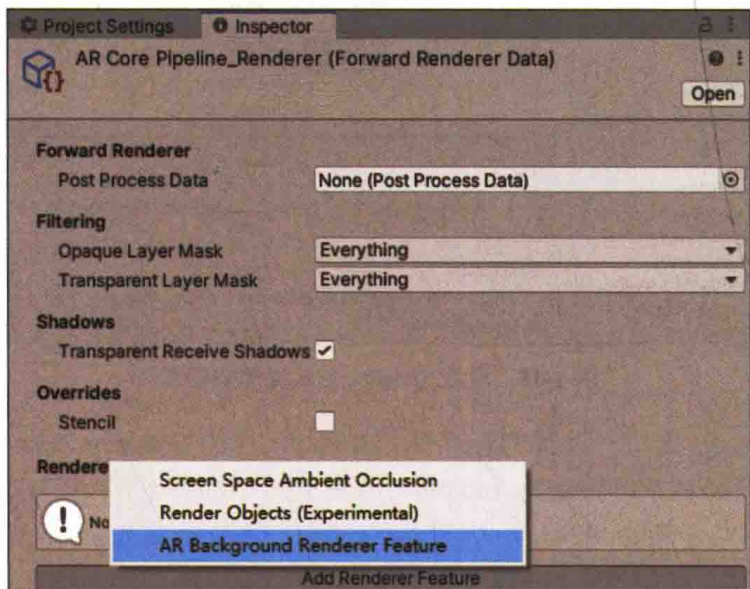


图 1-16 添加 AR Background Renderer Feature 渲染特性

返回 Project Settings 窗口，选择 Graphics 项，将其 Scriptable Render Pipeline Settings 属性值设置为创建的 ARCorePipeline.asset 渲染管线，如图 1-17 所示。

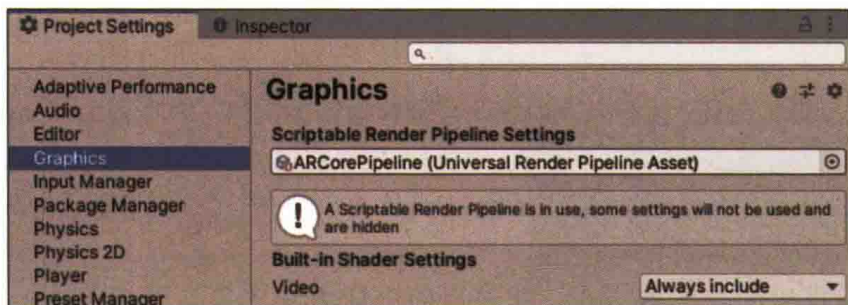


图 1-17 设置 Graphics 自定义渲染管线

在 Project Settings 窗口，选择 Quality 项，在右侧面板中选择 Android 图标下的下拉菜单，将质量设置为 Low 或者 Very Low；将 Rendering 属性值设置为创建的 ARCorePipeline.asset 渲染管线，如图 1-18 所示。

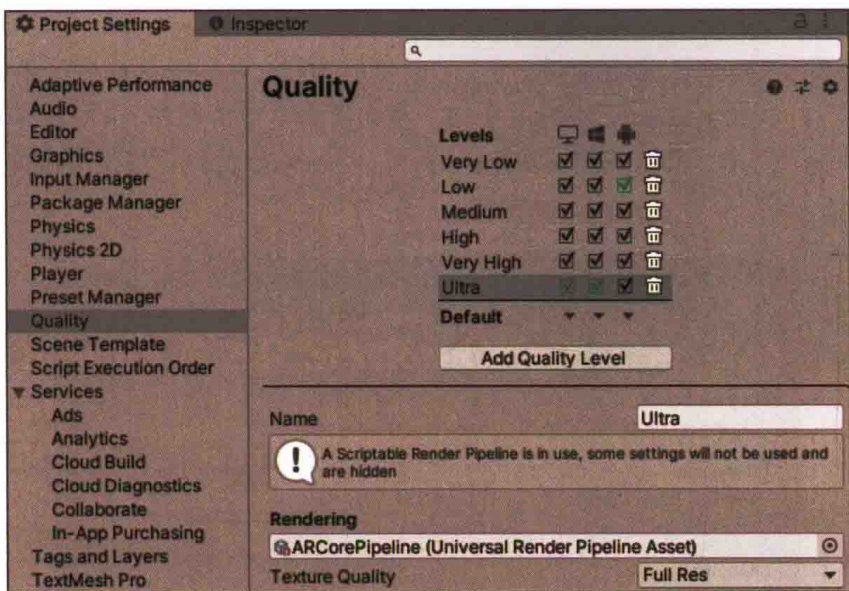


图 1-18 设置 Quality 自定义渲染管线

在 Project Settings 窗口，选择 Player 项，在 Player 栏的 Company Name 与 Product Name 中填写公司名与产品名，然后选择 Android 小图标进入 Adnroid 平台设置，在 Other Settings 选项卡 Graphics APIs 中删除 Vulkan^①，因为 Android 目前不支持 Vulkan，继续向下滚动滚动条，找到 Minimum API Level 属性，单击其下拉菜单，选择 Android 7.0 'nougat'(API Level 24) 或以上版本^②，正如这个选项名字一样，应用程序与 ARCore 将不会在 Nougat 版本之前的 Android 设备上运行，另外可以选择设置 Target API Level（或者使用默认的自动属性），目标版本不得低于 Nougat，不然 AR 应用将无法运行，如图 1-19 所示^③。

1.6.4 环境搭建

设置完成后返回 Unity 主界面，为统一规范管理各类文件，在工程窗口 Assets 目录下新建 Prefabs、Scripts、Materials 3 个文件夹，同时在层级窗口（Hierarchy 窗口）Helloworld 场景中删除 Main Camera 对象（因为 AR Session Origin 对象带有一个 AR 相机），Directional Light 对象可根据需要决定是否删除。

在层级窗口中的空白处右击，在弹出的级联菜单中依次选择 XR → AR Session 和 XR → AR Session Origin，新建这两个 AR 基础游戏对象，如图 1-20 所示。

① 选择 Vulkan，然后单击右下方的减号“-”图标删除该图形 API。

② 本节设置的是“Android 8.0 'Oreo' (API Level 26)”。

③ 当前主流移动终端都支持 64 位，为提高性能，建议 Scripting Backend 选择 IL2CPP，然后 Target Architectures 选择 ARM64。

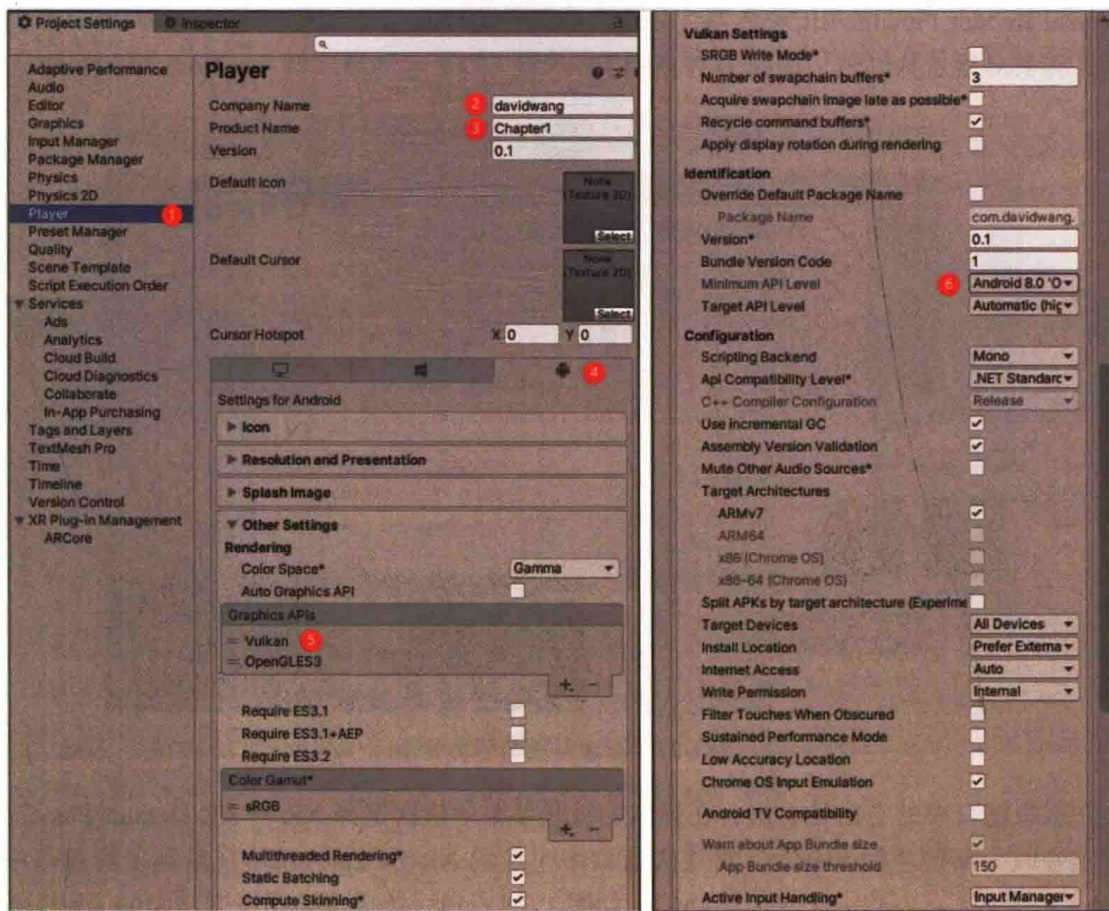


图 1-19 工程设置面板

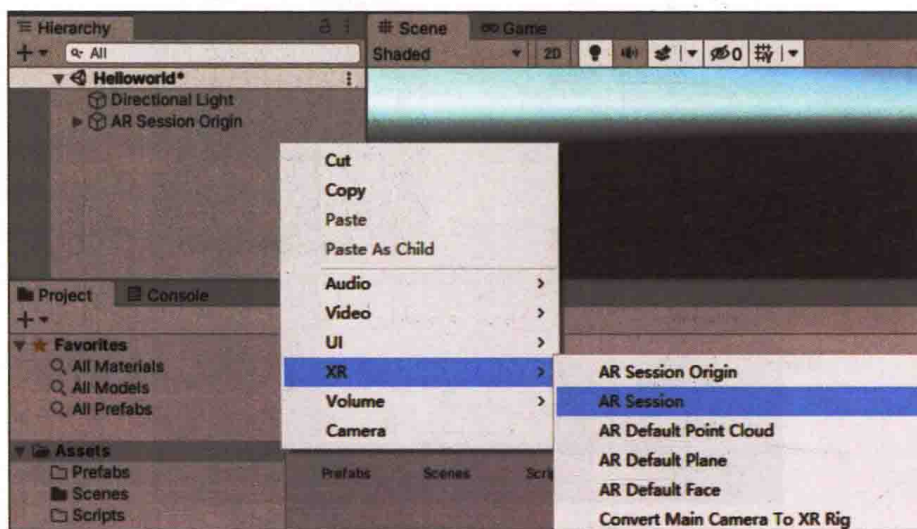


图 1-20 添加 AR 基础游戏对象

在工程窗口 Materials 文件夹中创建两种材质，分别命名为 Plane、Red，材质均使用

Universal Render Pipeline/Lit 着色器, 分别为这两种材质设置不同的颜色, 如将 Plane 设置为绿色, 将 Red 设置为红色, 如图 1-21 所示。



图 1-21 新建平面与球体材质

在层级窗口中的空白处右击, 在弹出的级联菜单中依次选择 XR → AR Default Plane, 将其材质修改为新创建的 Plane^①, 将层级窗口中生成的 AR Default Plane 拖动到工程窗口中的 Prefabs 文件夹下, 制作一个平面预制体, 如图 1-22 所示, 然后删除层级窗口中的 AR Default Plane 对象。



图 1-22 制作平面预制体

① 默认的材质为 Built-in 管线材质, 不适合 URP 管线。

在层级窗口中的空白处右击，在弹出的级联菜单中依次选择 3D Object → Cube，创建一个立方体对象，将 Cube 材质修改为 Red，将其 Scale 属性设置为 (0.1,0.1,0.1)，与制作平面预制体一样，制作一个 Cube 预制体，然后删除层级窗口中的 Cube 游戏对象。

在层级窗口选择 AR Session Origin 对象，然后在属性窗口中单击 Add Component 按钮，并在弹出的搜索框中输入 AR Plane Manager，双击搜索出来的 AR Plane Manager 添加该组件，如图 1-23 所示。

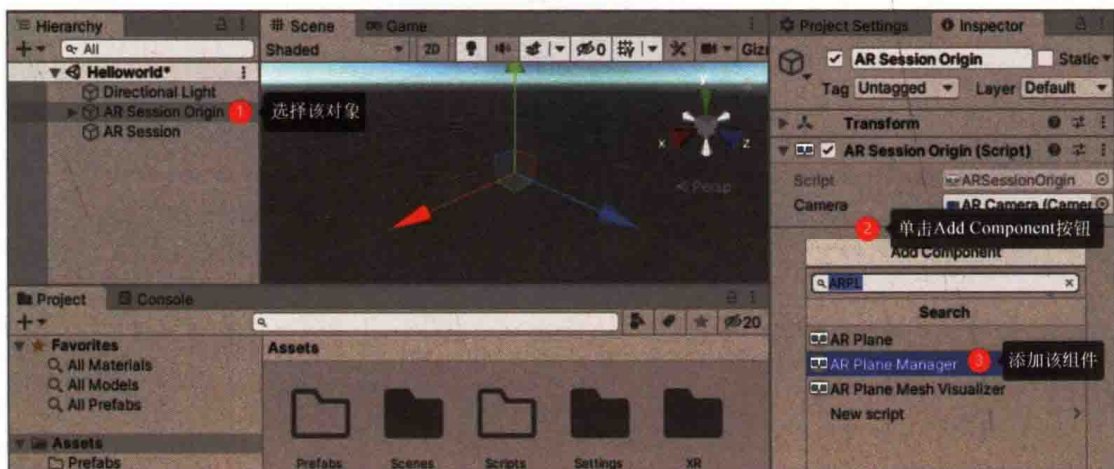


图 1-23 挂载 AR Plane Manager 组件

将工程窗口中 Prefabs 文件夹下的 AR Default Plane 预制体拖动到 AR Plane Manager 组件下的 Plane Prefab 属性框，完成平面预制体的设置，如图 1-24 所示。

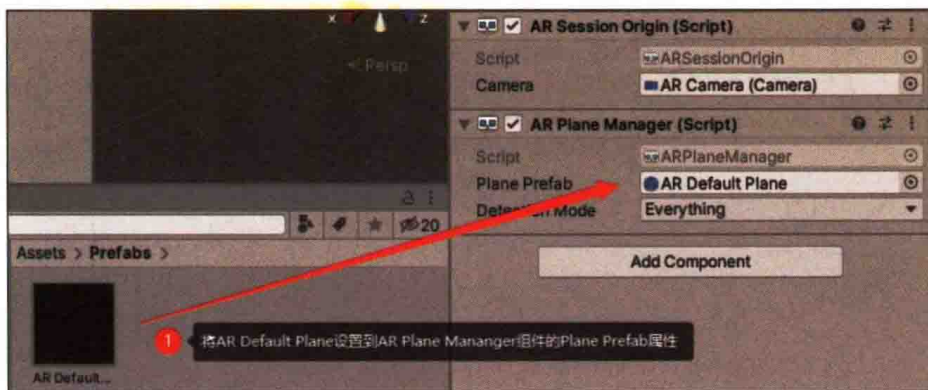


图 1-24 为 AR Plane Manager 组件的 Plane Prefab 属性赋值

1.6.5 代码编写

在工程窗口 Scripts 文件夹下的空白处右击，在弹出的级联菜单中依次选择 Create → C#Script，新建一个脚本文件，并命名为 AppController，双击 AppController 脚本，在 VS 2019

中编辑该脚本，添加代码如下：

```
// 第1章 / 1-1
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;

[RequireComponent(typeof(ARRaycastManager))]
public class AppController: MonoBehaviour
{
    public GameObject spawnPrefab;           // 需要放置的游戏对象预制体
    static List<ARRaycastHit> Hits;          // 碰撞结果
    private ARRaycastManager mRaycastManager;
    private GameObject spawnedObject = null; // 实例化后的游戏对象
    private void Start()
    {
        Hits = new List<ARRaycastHit>();
        mRaycastManager = GetComponent<ARRaycastManager>();
    }
    void Update()
    {
        if (Input.touchCount == 0)
            return;
        var touch = Input.GetTouch(0);
        if (mRaycastManager.Raycast(touch.position, Hits, TrackableType.
PlaneWithinPolygon | TrackableType.PlaneWithinBounds))
        {
            var hitPose = Hits[0].pose;
            // 如未放置游戏对象，则实例化；如果已放置游戏对象，则将游戏对象移动到新的碰撞点位置
            if (spawnedObject == null)
            {
                spawnedObject = Instantiate(spawnPrefab, hitPose.position, hitPose.
rotation);
            }
            else
            {
                spawnedObject.transform.position = hitPose.position;
            }
        }
    }
}
```

在上述代码中，首先使用 `[RequireComponent(typeof(ARRaycastManager))]` 属性确保添加该脚本的对象上必须挂载了 `ARRaycastManager` 组件，因为射线检测需要用到 `ARRaycastManager` 组件。在 `Update()` 方法中，对手势操作进行射线检测，在检测到的平面上放置一个虚拟物体，如果该虚拟物体已存在，则将该虚拟物体移动到新的碰撞点位置。

在层级窗口中选择 AR Session Origin 对象, 为其挂载 AppController 脚本 (可以使用 Add Component 在搜索框中搜索 AppController 添加, 也可以直接把 AppController 脚本拖到 AR Session Origin 对象上), 并将上一步制作的 Cube 预制体拖到 AppController 脚本的 Spawn Prefab 属性框中, 如图 1-25 所示。



图 1-25 为 AppController 脚本组件 Spawn Prefab 属性赋值

1.6.6 调试准备

为了方便将应用编译发布到手机中进行测试, 需要开启手机的“Developer Option (开发者选项)”功能。开发者选项, 顾名思义, 就是供应用开发者开发调试应用的选项, 通过这个功能开发者就可以通过 USB 将手机与开发计算机连接起来, 直接在手机硬件上安装、调试自己的应用程序。通过开发者选项可以在计算机和设备之间复制数据、在设备上安装应用、读取日志数据而不用发送通知, 也可以查看 Android 系统的一些数据和信息。默认情况下, 开发者选项是隐藏的, 不同的手机开启开发者选项的方式不太一样, 这与手机制造商提供的操作系统有关, 但操作大致相同, 略有差异, 可以通过手机制造商获取说明, 本节以华为 P20 Pro 手机为例进行演示。

(1) 进入设置界面, 单击“关于手机”按钮进入手机信息界面, 如图 1-26 (a) 所示。

(2) 连续单击“版本号”按钮 7 次, 触发开发者选项, 要求输入手机密码, 完成后即可开启开发者选项, 如图 1-26 (b) 所示。

(3) 返回设置主界面, 依次选择“系统和更新”→“开发人员选项”进入开发人员选项界面, 打开 USB 调试按钮, 提示“是否允许 USB 调试”弹窗, 单击“确定”按钮完成授权, 如图 1-26 (c) 所示。

(4) 将手机通过 USB 线与开发计算机进行连接, 在弹出的界面中选择“传输文件”功能选项 (默认为“只充电”), 在弹出“是否允许 USB 调试”提示窗口时, 勾选“始终允许使用这台计算机进行调试”复选框, 并单击“确定”按钮完成授权, 如图 1-26 (d) 所示。

开发者选项中有很多参数, 这些设置对应用开发者了解应用的运行状态很有帮助, 详细的参数及功能说明超出本书的范围, 读者可以自行探索了解。



图 1-26 开启开发人员选项并授权 USB 调试

1.6.7 打包测试

确保手机通过 USB 线与开发计算机连接，在 Unity 界面中，从菜单栏依次选择 File → Build Settings（或者使用 Ctrl+Shift+B 快捷键）打开 Build Settings 对话框。在打开的对话框中确保选中当前场景，单击 Build And Run 按钮进行应用构建和发布，如图 1-27 所示，在随

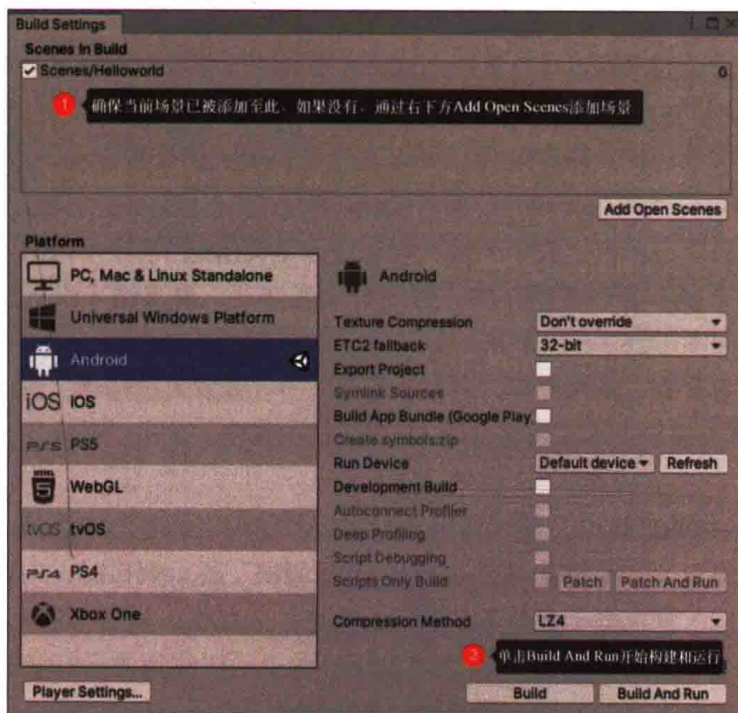


图 1-27 编译并运行应用对话框

后弹出的文件保存对话框中选择应用程序保存路径，单击“保存”按钮开始编译打包 APK 应用。等待 Unity 编译完成后，其会自动将应用安装到手机上并启动应用，当应用启动后，找一个相对平坦且纹理比较丰富的平面左右移动手机进行平面检测，在检测到平面后，手指单击手机屏幕上已检测到的平面，将会在该平面上加载一个小立方体，如图 1-28 所示。至此，使用 AR Foundation 开发的 ARCore Helloworld AR 应用已成功。



图 1-28 应用程序运行效果图

1.7 AR 应用调试

AR 应用运行时需要采集来自设备摄像头的图像数据、设备运动传感器数据，并以此为基础来构建环境感知和估计设备姿态。AR 应用目前无法完全在模拟器中进行测试，而必须采用真机进行测试，这导致 AR 应用的调试测试工作非常缓慢低效，特别是对需要进行现场验证的应用，如导航、实景增强类应用，需要亲自到现场测试，除了天候影响，这还是一个非常烦琐且费时费力的工作。

在使用 USB 或者 WiFi 调试应用时，将 AR 应用编译并推送到手机上需要花费很长时间，但 AR 应用在手机端运行之后，只能看到运行的结果而不能确切地知道在运行过程中发生的事情，作为开发者，我们希望能跟踪代码执行情况，特别是在运行结果与预期不符时，需要查看调试信息查排错误。

1. Logcat 工具

本节将介绍远程调试 AR 应用的一般方法，通过 Android Logcat 工具，可以在开发计算机中远程查看 AR 应用程序的运行状况。

首先通过 USB 或者 WiFi 连接到移动设备（确保已打开开发者选项和允许调试），然

后在开发计算机中打开 Android SDK 安装目录, 进入 SDK 目录下的 Tools 子目录内^①, 双击 monitor.bat 批处理脚本, 这将打开 Android 设备监视器 (Android Device Monitor)。其左侧列表中的设备即为此时连接到计算机的移动设备, 选择该设备, 在 Logcat 窗口将看到设备的日志输出流, 拖动 Logcat 窗口, 使其成为主窗口中的一个选项卡, 如图 1-29 所示。

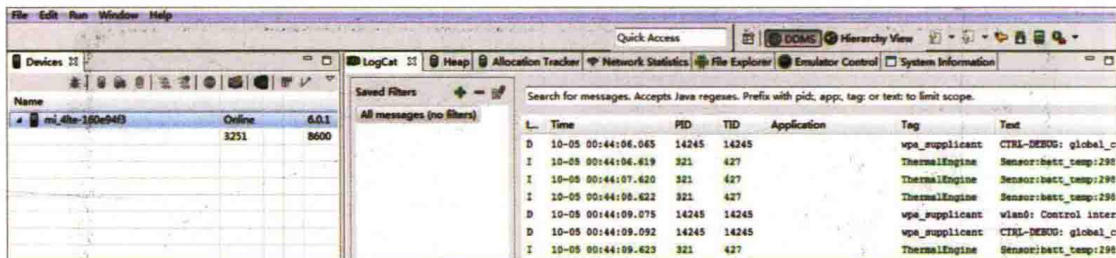


图 1-29 使用 Android Device Monitor 查看 AR 应用日志信息

通过 Logcat 工具, 移动设备运行的系统和所有应用日志都将实时地输出到窗口中, 在 Logcat 窗口中可以看到 AR 应用程序的底层日志, 也可以看到自己编写的 Debug 输出, 通过查看这些日志输出信息, 可以实时了解 AR 应用运行状态、进行 AR 应用远程调试。

默认 Logcat 工具会输出移动设备系统和所有应用运行日志, 信息量多且繁杂, 可以对日志消息进行过滤, 屏蔽无关信息干扰, 操作步骤如下:

- (1) 转到 Android 设备监视器窗口。
- (2) 单击 Logcat → Saved Filters 面板中的绿色加号 (+) 按钮创建过滤器。
- (3) 通过输入过滤器名称 Unity 和日志标记 Unity 创建一个新的过滤器。
- (4) 单击“确定”按钮添加过滤器。

通过添加过滤器, Logcat 窗口只会输出 Unity 应用程序消息, 日志消息更加简洁。除了使用日志标记 (Log Tag) 创建过滤器外, 还可以通过日志消息 (Log Message)、PID、应用名字 (Application)、日志等级 (Log Level) 等来创建过滤器, 或者联合其中的两项或多项创建复杂的过滤器, 使过滤出来的信息更符合预期要求。

事实上, 除了使用 Android SDK 独立的 Logcat 工具, Unity 也已经提供了 Android Logcat 插件包, 可以将该工具集成到开发环境中。在 Unity 菜单中, 依次选择 Window → Package Manager, 打开 UPM 管理器窗口, 检索 Android Logcat 可以直接安装该插件包, 如图 1-30 所示。

在安装好该插件包后, 可以通过 Unity 菜单 Window → Analysis → Logcat 打开该工具的使用面板, 在移动设备连接到开发计算机后, 可以通过该面板查看所有日志输出, 同样也可以通过设置应用程序、关键字等进行日志过滤。

^① 需要先安装 Android SDK, 以笔者计算机为例, 默认 monitor.bat 批处理脚本在 Android SDK 目录下, 所在路径为: C:\Users\Davidwang\AppData\Local\Android\Sdk\tools。



图 1-30 安装 Android Logcat 插件包

2. MARS 工具

MARS (Unity Mixed and Augmented Reality Studio, Unity 混合和增强现实工具) 是 Unity 公司开发的辅助增强和混合现实开发工具套件, MARS 包括一个 Unity 扩展包和一组配套应用程序。通过提供模拟仿真环境, 将 AR 应用测试工作从真机设备迁移到 Unity 开发环境中, MARS 希望构建一个所见即所得的 AR 创作环境。MARS 提供多种室内和室外空间的模拟仿真环境, AR 应用开发时可以不依赖于真机设备, 但却可以测试和展现其在真实世界中运行时的状态。

MARS 以可视化的方式构建 AR 应用模拟仿真环境, 开发人员在 Unity 编辑器中即可对应用进行测试, 模拟仿真环境以 AR 应用运行时的逻辑构建, 可以适应真实环境, 能自动适应物理空间和进行基于真实环境的 AR 应用开发, 帮助开发者在仿真的环境空间中进行创作。

MARS 工具套件建立在 AR Foundation 之上, 针对 AR 应用开发提供了新的 UI 和控件, 并能对 AR 内容修改做出实时的响应, 除此之外, 还配套推出了移动端增强现实伴侣应用程序 (Augmented Reality Companion Apps), 通过该伴侣应用程序能捕获真实世界环境和对象数据, 随后可以导入 Unity 编辑器中, 提供对增强现实场景的编辑功能, 从而减少迭代时间, 提升项目开发效率。

MARS 封装了 AR Foundation 的 API, 将现有项目迁移到 MARS 中需要使用新的 API 重写脚本、使用 MARS 组件从头构建 AR 场景, 即现有 AR Foundation 项目与 MARS 项目不兼容, 因此如果计划采用 MARS, 则建议从项目开始即使用。MARS 使用较为复杂, 关于 MARS 的更多内容, 可查阅官方文档。

3. AR Foundation Editor Remote 工具

AR Foundation Editor Remote 远程调试工具 (以下简称 Remote 工具) 为第三方工具, 可以在 Unity Assets Store 中购买, 该工具支持通过 WiFi 或 USB 连接移动设备进行 AR 功能调试。Remote 工具也是一款 Unity 编辑器的扩展工具, 该工具通过在移动设备端安装一个 AR 伴侣应用 (AR Companion) 采集 AR 应用所需数据并回传到 Unity 编辑器中, 实现对 AR 应用的远

程调试,支持平面检测、面部检测、射线检测、图像检测、光照估计、场景深度检测等功能,也支持远程 AR 会话的启用、禁用、跟踪。

使用 Remote 工具不需要额外的场景设置,与现有项目没有耦合关系,它只是采集真机数据并回传到 Unity 编辑器,方便进行 AR 应用调试。

4. AR Simulation 工具

AR Simulation 工具也由第三方提供,同样构建于 XR 插件体系架构基础上,但它不将真机设备数据回传到 Unity 编辑器,而是完全模拟 AR 运行时的数据,因此,使用该工具时无须借助任何真机设备就可以进行 AR 应用调试,但另一方面,通过生成模拟数据的方式可能不能完全再现真机的某些行为特性,基于此构建的 AR 应用最终依然需要进行真机测试以确保行为逻辑符合预期。使用 AR Simulation 工具时需要对场景进行少许设置。

在进行 AR 应用开发时,除 Logcat 工具经常使用外,通常选择另外三者之一作为主要调试手段,AR Foundation Editor Remote、AR Simulation、Unity MARS 调试特性对比如表 1-6 所示。

表 1-6 AR Foundation Editor Remote、AR Simulation、Unity MARS 调试特性对比^①

AR 调试特性	AR Foundation Editor Remote	AR Simulation	Unity MARS
ARCore 支持	√	√	√
ARKit 支持	√	√	√
平面检测	√	√	√
点云跟踪	√	√	√
图像检测	√	√	√
场景网格	√	-	√
ARKit 场景网格分类	√	-	-
人脸检测	√	-	√/-
会话录制 (Session Recordings)	√	-	+
射线检测	√/-	√/-	-
光照估计	√	√	-
锚点	√	√/-	-
ARCore 云锚点	√	-	-
设备模拟	√	√	-
屏幕单单击触摸模拟	√	√	-
屏幕多单击触摸模拟	√	-	-
ARKit 眼动跟踪	√	-	-
ARKit 脸部 Blendshapes	√	-	-

^① 标用√/-符号为该功能特性有限制或者有额外要求。

续表

AR 调试特性	AR Foundation Editor Remote	AR Simulation	Unity MARS
摄像头视频流	✓	—	—
遮挡	✓	—	—
CPU 图像捕获	✓	—	—
基于位置的服务 (GPS)	✓	—	—
ARKit 人形分割	✓	—	—
ARKit 二维和三维人体跟踪	✓	—	—
ARKit 三维物体检测	✓	—	—
ARKit World Map	✓	—	—
易于集成度	简单	比较简单	复杂
与现有项目耦合度	无耦合	弱耦合	不兼容, 不支持切换
环境模拟	—	✓	✓
对新输入系统的支持	✓	—	✓ / —
最低 AR Foundation 版本	3.0.1	3.0.1	2.1.8
最低 Unity 版本	2019.4	2019.3	2019.3

配置好一个带有远程连接和调试支持的 Unity 开发环境将使我们的开发工作更容易进行, 有利于将主要精力投入应用程序功能开发上, 加快测试调试进程。

第 2 章



AR Foundation 基础

AR Foundation 是 Unity 为跨平台 AR/MR 开发而构建的开放式架构插件包，它本身并不实现任何 AR/MR 特性，是第三方底层 SDK 的软件抽象层，为开发人员提供一个统一、公共、规范的接口。在目标平台上，AR Foundation 会调用该平台下的第三方 SDK 库（如 iOS 上的 ARKit XR 插件或 Android 上的 ARCore XR 插件）实现功能，AR Foundation 整合了各平台独立的第三方 AR/MR 开发包，提供给开发者统一的使用界面，实现一次开发、多端部署的目的。

2.1 AR Foundation 体系架构

在第 1 章中我们已经知道，AR Foundation 体系建立在一系列的子系统（Subsystem）之上，这种架构的好处类似于接口（Interface），将界面与具体的实现分离，开发者只需面向接口编程，而无须关注具体的算法实现（在 AR Foundation 中这个具体的算法提供者就叫 Provider，Provider 这里指实现具体算法的包或者插件，为避免混淆，该英文后文不翻译）。同时，这种软件抽象屏蔽了底层差异并提供更好的可扩展性，允许多个具体实现，支持灵活添加自定义实现，即 AR Foundation 只定义了接口界面，而由 Provider 提供具体的实现，因此具有很强的可扩展能力，如人脸检测，AR Foundation 定义了 XRFaceSubsystem 子系统界面，ARCore 可以提供 Android 平台的具体实现，ARKit 可以提供 iOS 平台的具体实现，HoloLens 可以提供 HoloLens 眼镜的具体实现等，在同一体系架构下可以提供无限的具体实现可能性，并支持随时纳入新的 AR SDK。AR Foundation 在编译时会根据用户选择的平台调用该平台上的一种具体实现，这个过程对开发者透明，开发者无须关注具体的实现方式，从而实现一次开发、多平台部署的目的。AR Foundation 的体系架构如图 2-1 所示。

类似于接口，Provider 在具体实现特定接口（子系统）算法时，可以根据其平台本身的特性进行优化，并可以提供更多功能。由于硬件和软件生态的巨大差异，A 平台具备的特性 B 平台不一定提供，为了让开发者了解这种差异，AR Foundation 要求 Provider 提供子系统描述符（Subsystem Descriptor），子系统描述符描述了 Provider 对特定子系统提供的能力范围，有了这个描述符，开发人员就可以通过遍历查看特定的功能在某特定平台是否提供，为功能开发提供依据。

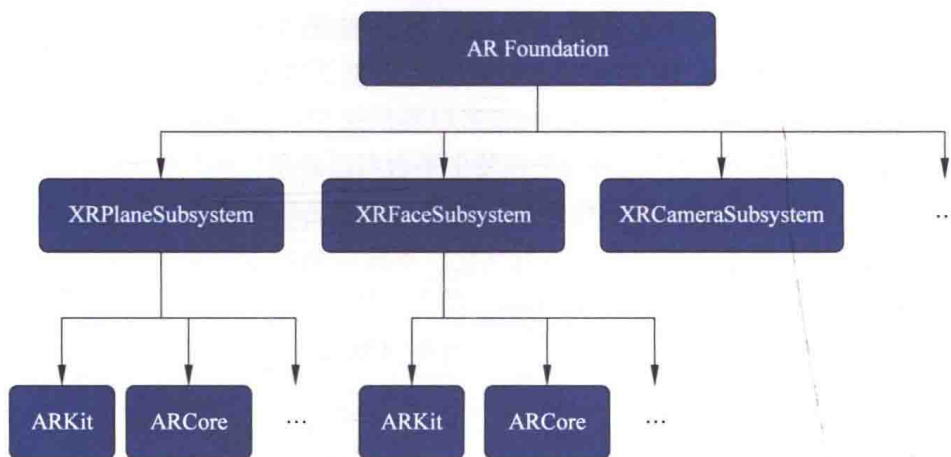


图 2-1 AR Foundation 体系架构

2.1.1 AR 子系统概念

子系统是一个与具体平台无关的软件抽象层，某一子系统定义了某一特定功能的用户界面，AR Foundation 中的子系统只定义了功能接口而没有具体实现，具体实现由 Provider 在其他包或者插件中提供，与 AR 相关的子系统隶属于 UnityEngine.XR.ARSubsystems 命名空间。AR Foundation 目前提供了以下 12 类子系统，具体包括 Session、Raycasting、Camera、Plane Detection、Depth、Image Tracking、Face Tracking、Environment Probes、Object Tracking、Human Body Tracking 2D、Human Body Tracking 3D、Human Segmentation Image。

由于具体的子系统算法实现由 Provider 在其他插件中提供，如果不引入相应的插件，应用编译就会失败，因此，在编译到具体平台时就需要在工程中引入相应的插件，同时引入两个或者多个平台的插件也不会发生冲突，AR Foundation 会根据当前工程的编译平台选择对应的插件。引入插件的方法是在 Unity 菜单中，依次选择 Window → Package Manager 打开 UPM 管理器窗口，然后在左侧的插件包中选择相应的插件进行安装^①，目前 AR Foundation 的 Provider 包括 ARCore XR Plugin、ARKit XR Plugin、Magic Leap XR Plugin、Windows XR Plugin 共 4 种，如图 2-2 所示。

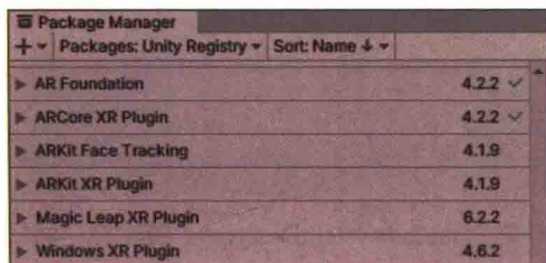


图 2-2 AR Foundation 底层 Provider 插件包

^① 默认只显示本工程使用的插件包，为导入其他插件包，需要在包管理器窗口中选择 Packages:Unity Registry 值显示 Unity 注册的所有插件。

对于 AR Foundation 定义的子系统, Provider 可以实现也可以不实现, 如 Human Body Tracking 3D 子系统, ARCore XR Plugin 目前就没有实现, 因此, Android 平台 Human Body Tracking 3D 功能将不可用。另外, Provider 也可以提供比子系统所定义的更多的功能, 如一些辅助功能, Provider 提供的这些额外功能在特定平台可以使用, 但具体需要查阅 Provider 技术文档。鉴于以上情况, 针对具体平台进行开发时一定要关注该平台 Provider 能够提供的功能范围和能力, 另外, 为了对特定平台进行优化, 不同的 Provider 对参数也有要求, 如在图像检测跟踪功能中, ARKit 要求提供被检测图像的物理尺寸, 而 ARCore 则没有这个要求, 这都需要参阅特定平台 Provider 的相关资料。本书针对 ARCore 开发, 所有 ARCore 相关能力范畴都会涉及。

2.1.2 AR 子系统使用

所有子系统都有相同的生命周期, 即创建 (Created)、开始 (Started)、停止 (Stopped)、销毁 (Destroyed)。每个子系统都有对应的子系统描述符, 可以使用 SubsystemManager 类遍历 Provider 能够提供的功能描述符集, 一旦得到可用的子系统功能描述符后, 就可以使用 Create() 方法创建该子系统, 这也是创建子系统的唯一方式 (在实际使用时, 很多时候由 SubsystemManager 类自动执行这个过程, 但有时我们也可以手动进行代码控制)。下面以创建一个 Plane Subsystem 为例来说明这个过程, 代码如下:

```
// 第 2 章 / 2-1
XRPlaneSubsystem CreatePlaneSubsystem()
{
    // 得到所有可用的 Plane Subsystem
    var descriptors = new List<XRPlaneSubsystemDescriptor>();
    SubsystemManager.GetSubsystemDescriptors(descriptors);
    // 遍历获取一个支持 Boundary Vertices 的功能
    foreach (var descriptor in descriptors)
    {
        if (descriptor.supportsBoundaryVertices)
        {
            // 创建 Plane Subsystem
            return descriptor.Create();
        }
    }
    return null;
}
```

在这个例子中, 首先得到 PlaneSubsystemDescriptor 描述符, 然后遍历这个描述符检查其是否支持 BoundaryVertices 功能特性, 如果支持, 则说明满足需求, 可创建它。子系统创建时其内部实现采用了单例模式, 确保应用中只有一个该类型的子系统, 即使再次调用 Create() 方法, 还是返回第一次创建生成的实例。一旦子系统创建成功后, 就可以通过调用 Start()、Stop() 方法来启 / 停用该子系统, 但需要注意的是, 对不同的子系统, Start()、Stop() 方法执行行为也会

略有不同，但通常执行启用与停用的操作，并且一个子系统可以在其生命周期内调用 Start()、Stop() 方法多次。在使用完某个子系统后，应当调用 Destroy() 方法销毁^① 该子系统以避免无谓的性能消耗，在销毁子系统后，如果需要再次使用，则需要重新创建，示例代码如下：

```
// 第2章 / 2-2
var planeSubsystem = CreatePlaneSubsystem();
if (planeSubsystem != null)
{
    // 开始平面检测
    planeSubsystem.Start();
}

if (planeSubsystem != null)
{
    // 停止平面检测，但这并不会影响已检测到的平面
    planeSubsystem.Stop();
}

if (planeSubsystem != null)
{
    // 销毁该 Subsystem
    planeSubsystem.Destroy();
    planeSubsystem = null;
}
```

再次提醒，针对特定平台的特定功能，在需要时应该查阅该 Provider 的资料以获取更详尽的信息。

2.1.3 跟踪子系统

跟踪子系统 (Tracking Subsystem) 是在物理环境中检测和跟踪某类信息的子系统，如平面跟踪和图像跟踪。跟踪子系统所跟踪的对象被称为可跟踪对象 (Trackable)，如平面跟踪子系统检测跟踪平面，因此平面是可跟踪对象。

在 AR Foundation 中，每一类跟踪子系统都提供了一个名为 getchanges() 的方法，此方法用于检索有关它所跟踪的可跟踪对象的信息数据，getchanges() 方法获取自上次此方法调用以来添加、更新和移除的所有可跟踪对象信息，通过该方法可以获取可跟踪对象的变化情况。

每个可跟踪对象都由标识符 ID (TrackableId，一个 128 位的 GUID 值) 唯一标识，即所有可跟踪对象都是独立可辨识的，可以通过这个 TrackableId 获取某个可跟踪对象。

跟踪子系统可以添加、更新或移除可跟踪对象。只有被添加到跟踪系统中的可跟踪对象才可以被更新或移除，因此在更新或者移除可跟踪对象时先检查该对象是否在跟踪子

^① 事实上，子系统由 AR Foundation 内部管理，调用 Destroy() 方法也不会销毁子系统，但可以释放部分资源。

系统的跟踪中, 如果尝试更新或移除尚未添加的可跟踪对象, 则会引发运行时错误, 同样, 未添加或已移除的可跟踪对象无法更新也无法再次移除。

为了方便使用, 在 AR Foundation 中, 每一类跟踪子系统的管理类组件 (如 AR Plane Manager 组件) 通常会提供一个 xxxChanged 事件 (如 planesChanged 事件), 可以通过其 xxxChangedEventArgs (如 ARPlanesChangedEventArgs) 参数获取 added、updated、removed 可跟踪对象进行后续处理。

2.2 AR Session & AR Session Origin

在第 1 章的案例中, 构建应用程序框架时, 首先在层级窗口中创建了 AR Session 和 AR Session Origin 两个对象, 如图 2-3 所示, 这两个对象构建了 AR 应用最基础的框架, 所有其他工作都在此基础之上展开, 那么这两个对象在整个 AR 应用中起什么作用呢?

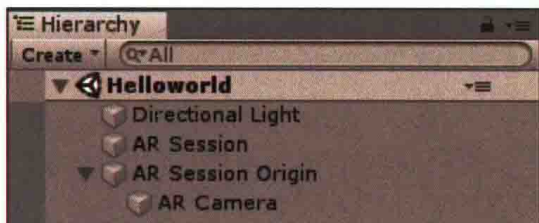


图 2-3 AR Session 和 AR Session Origin 基础对象

2.2.1 AR Session

AR Session 对象上挂载着两个功能组件: 一个是 AR Session 组件, 该组件管理 AR 会话 (Session); 另一个是 AR Input Manager 组件, 主要处理输入相关信息, 如图 2-4 所示。AR 会话用于管理 AR 应用状态、维护 AR 应用生命周期, 控制在目标平台上启用或禁用 AR 功能, 是 AR API 的主要入口。AR 会话根据需要开始或暂停相机图像帧的采集、初始化或释放相关资源、建立会话空间、执行运动跟踪、负责 SLAM 底层逻辑等。AR 场景中必须包含 AR Session 组件, 但 AR Session 组件可以挂载在场景中任何对象上, 一般为了便于管理, 将其挂载在 AR Session 对象上。如果在应用运行期间禁用 AR Session 组件, 则 ARCore 将不再跟踪其环境中的特征点、暂停 SLAM 运动跟踪功能, 但如果稍后再启用, 则 ARCore 将尝试恢复运动跟踪、尝试重定位。

AR Session 组件下有 3 个属性, 即 Attempt Update (尝试更新)、Match Frame Rate (匹配帧率)、Tracking Mode (跟踪模式)。勾选 Attempt Update 复选框, 在 AR 应用启动后, 将在可能的情况下尝试安装 / 更新最新版的 ARCore (Google Play Services for AR), 在 iOS 平台上, 将尝试安装最新版的 ARKit。

勾选 Match Frame Rate 复选框, 则在 AR 应用运行期间会做 3 件事情:

(1) 保持 AR 帧率与渲染帧率一致, AR 帧指完成一次图像采集、IMU 传感器数据读取等

操作的执行周期，因为执行的操作比较多，速度可能会比较慢，从而导致渲染帧率与 AR 帧率不一致，勾选该选项后，则会保持两种帧率的同步。

(2) 将渲染目标帧率设置到 AR 会话预期的更新帧率，有助于提高跟踪质量。

(3) 禁用 VSync，即禁用垂直同步，防止垂直同步影响 AR 帧率与渲染帧率同步。

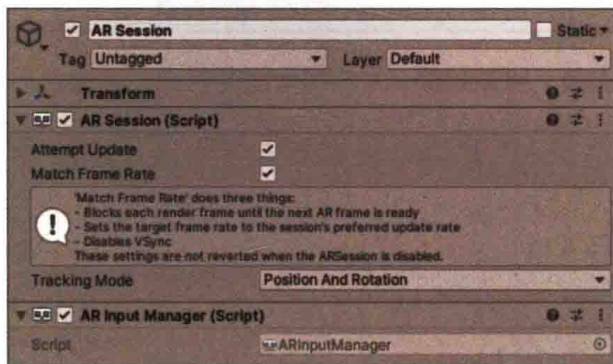


图 2-4 AR Session 对象上挂载的基础组件

Tracking Mode 用于设置 ARCore 运动跟踪使用的模式，支持 Rotation Only、Position And Rotation 两种模式之一，Rotation Only 只支持 3DoF 旋转跟踪，Position And Rotation 支持 6DoF 运动跟踪，一般选择 Position And Rotation。

AR Input Manager 组件是启用 SLAM 运动跟踪所必需的组件，不启用此组件，AR Pose Driver 组件（跟踪姿态驱动）将无法获取设备姿态数据。

特别需要注意的是，在任何一个 AR 应用中，有且仅允许有一个 AR Session，Unity 将 AR Session 组件设置成全局组件，因此如果场景中挂载着多个 AR Session 组件，则这些 AR Session 组件都将尝试管理同一个会话，同样，AR Input Manager 组件也只能有且仅有一个。

Android 移动手机平台设备种类繁多，并不是每一款设备都支持 ARCore，因此需要一些 AR 会话状态来表示设备的功能可用性，以便根据 AR 会话状态进行下一步操作或者在设备不支持 AR 时提供替代方案。

AR Foundation 使用 ARSessionState 枚举类型表示当前设备的 AR 会话状态（例如设备是否支持、是否正在安装 AR 软件及 AR 会话是否工作等），开发人员可以根据 AR 会话状态决定执行的操作，还可以订阅 ARSession.stateChanged 事件，在 AR 会话状态变化时得到通知。

ARSessionState 枚举包括表 2-1 所示枚举值。

表 2-1 ARSessionState 枚举值

枚举值	描述
None	状态未知，未安装 ARCore
Unsupported	当前设备不支持 ARCore
CheckingAvailability	系统正在进行设备支持性检查
NeedsInstall	设备硬件支持 ARCore，但需要安装 ARCore 工具包
Installing	设备正在安装 ARCore

枚举值	描述
Ready	设备支持 ARCore 并且已做好使用准备
SessionInitialized	ARCore 会话正在初始化, 还未建立运动跟踪
SessionTracking	ARCore 会话运动跟踪中

根据 ARSession 状态值, 可以检测 AR 应用当前的状态, 关键代码如下:

```
// 第 2 章 / 2-3
IEnumerator CheckSupport()
{
    Debug.Log(" 检查设备 ...");
    yield return ARSession.CheckAvailability();
    if (ARSession.state == ARSessionState.NeedsInstall)
    {
        Debug.Log(" 设备支持 AR, 但需要更新 ...");
        Debug.Log(" 尝试更新 ...");
        yield return ARSession.Install();
    }
    if (ARSession.state == ARSessionState.Ready)
    {
        Debug.Log(" 设备支持 AR!");
        Debug.Log(" 启动 AR...");

        // 通过设置 enable=true 启动 AR
        m_Session.enabled = true;
    }
    else
    {
        switch (ARSession.state)
        {
            case ARSessionState.Unsupported:
                Debug.Log(" 设备不支持 AR. ");
                break;
            case ARSessionState.NeedsInstall:
                Debug.Log(" 更新失败. ");
                break;
        }
        //
        // 启动非 AR 的替代方案 ...
        //
    }
}

// 控制屏幕上的 UI 按钮
void SetInstallButtonActive(bool active)
{

```



```

    if (m_InstallButton != null)
        m_InstallButton.gameObject.SetActive(active);
}
// 安装 ARCore 支持包
IEnumerator Install()
{
    if (ARSession.state == ARSessionState.NeedsInstall)
    {
        Debug.Log("尝试安装 ARCore 服务 ...");
        yield return ARSession.Install();

        if (ARSession.state == ARSessionState.NeedsInstall)
        {
            Debug.Log("ARCore 服务更新失败。");
            SetInstallButtonActive(true);
        }
        else if (ARSession.state == ARSessionState.Ready)
        {
            Debug.Log("启动 AR...");
            m_Session.enabled = true;
        }
    }
    else
    {
        Debug.Log("无须安装。");
    }
}

void OnEnable()
{
    StartCoroutine(CheckSupport());
}

```

代码中的安装指在手机端安装 ARCore 功能支持包，部分 Android 手机硬件支持 AR，但没有安装相应的 ARCore 功能包，可以通过这种方式进行安装^①。

2.2.2 AR Session Origin

AR Session Origin 对象默认挂载 Transform 组件和 AR Session Origin 组件，如图 2-5 所示。

AR Session Origin 组件的作用是将可跟踪对象（如平面和特征点）姿态信息转换到 Unity 空间，因为 AR 应用由会话管理，因此其获取的各类姿态信息在 Session Space（会话空间）中，这是一个以 AR 会话初始化完成时所在位置为坐标原点的非标度空间，因此由 AR Session

^① 程序可直接到 Google Play 应用商店中搜索下载 ARCore 进行安装，因此需要能通过网络连接到 Google Play。

Origin 组件执行一次坐标空间变换, 将其变换到 Unity 坐标空间, 类似于从模型局部空间到世界空间的变换。例如, 在制作三维模型时, 通常会建立模型自己的坐标系, 模型的所有部件都相对于其自身的坐标系构建, 模型自己的坐标系称为“模型空间”或“局部空间”, 当将这些模型导入 Unity 中时, 模型的所有顶点位置信息都要变换到以 Unity 坐标原点为基准的世界空间中。同样, 由 AR 应用检测到的可跟踪对象姿态在会话空间中, 因此也要做一次坐标变换将其变换到 Unity 世界坐标空间中。

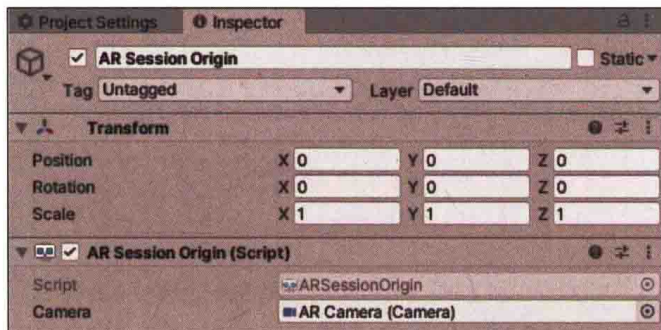


图 2-5 AR Session Origin 对象下的默认组件

AR Session Origin 组件还接受一个相机参数, 这个相机就是 AR 渲染相机, AR 渲染相机与用户设备硬件摄像头通过 AR Pose Driver 组件提供的位姿对齐, 同时与 AR 会话空间对齐, 这样就可以从正确的视角在指定位置渲染虚拟物体了。

AR Session Origin 对象有一个 AR Camera 子对象, AR Camera 对象默认挂载 AR Pose Driver、AR Camera Manager、AR Camera Background 组件, 如图 2-6 所示。

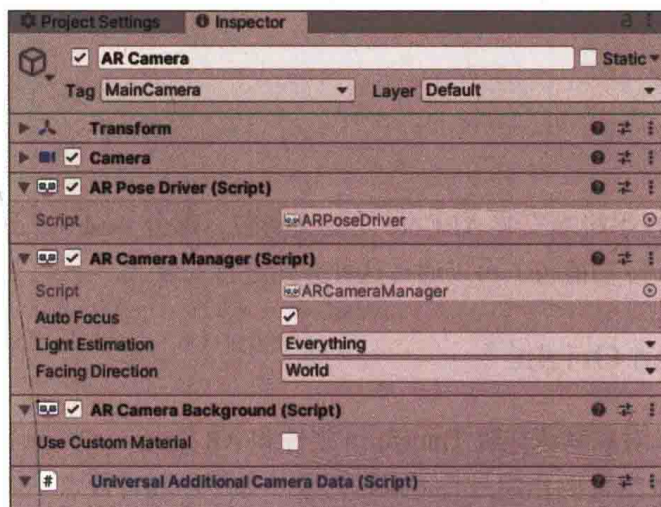


图 2-6 AR Camera 对象下的默认组件

AR Pose Driver 组件的主要作用是将 Unity 中的渲染相机与设备的真实硬件摄像头对齐, 即根据设备硬件摄像头的位姿估计信息调整 Unity 中的渲染相机姿态, 保持虚实相机对齐匹配, 保持虚实相机参数一致 (拥有相同的投影矩阵), 从而确保 Unity 中渲染的虚拟物体在真实世

界中的位置与方向正确。反之，如果没有该组件，Unity 渲染的虚拟物体与真实环境就不会有联系。

AR Camera Manager 组件负责处理及控制摄像头的一些细节参数，其有 3 个参数，即 Auto Focus（自动对焦）、Light Estimation（光照估计）、Facing Direction（使用设备摄像头类型），具体意义如表 2-2 所示。

表 2-2 AR Camera Manager 组件属性

属 性	描 述
Auto Focus	摄像头对焦模式，勾选该值即允许摄像头自动对焦，一般会选择勾选，如果不勾选，则使用固定焦距，固定焦距不会改变设备摄像头焦距，因此，设备与被拍摄物体距离不合适时就会出现采集的图像模糊
Light Estimation	光照估计模式，由于各具体平台光照估计能力不同，这里将 ARCore、ARKit 所有与光照估计相关的能力都列出，在使用时可根据相应平台和所需能力进行选择。光照估计对性能消耗较大，不使用该功能时，建议将其设置为 None 值
Facing Direction	AR 应用使用的设备摄像头类型，World 值为使用后置摄像头，User 值为使用前置摄像头

在 AR 中，以真实环境作为应用背景，因此需要将设备摄像头捕获的图像渲染成背景，使用 AR Camera Background 组件就可以轻松实现这个功能。AR Camera Background 组件还有一个参数 Use Custom Material（使用自定义材质），Use Custom Material 复选框一般情况下不勾选，默认由 Unity 根据应用发布平台进行背景渲染，但如果勾选，就需要提供背景渲染的材质，即实现将摄像头捕获的图像渲染成 AR 应用背景的功能，利用自定义材质可以实现一些高级功能，如背景模糊、描边等。

如果在场景中只有一个 AR Session Origin 对象，为其添加 AR Camera Background 组件就可以实现将来自摄像头的图像渲染成 AR 应用背景，但如果场景中有多多个 AR Session Origin 对象、多个摄像头图像（例如在不同的缩放尺度上渲染 AR 场景），则这时就要为每个 AR Session Origin 对象、每个摄像头指定一个 AR Camera Background 组件并进行相应设置。

AR Session 组件和 AR Session Origin 组件负责 AR 应用整个生命周期管理、AR 相机操作及背景渲染相关工作，这些工作任务每个 AR 应用都需要，但这些工作需要与设备硬件交互，需要与底层 SLAM 通信，通常也会非常复杂，这两个功能组件对此进行了良好封装处理，开发者不用再担心具体的实现细节，极大地降低了 AR 应用开发的难度。

2.3 可跟踪对象

在 AR Foundation 中，平面（Plane）、特征点云（Point Cloud）、锚点（AR Anchor）、跟踪图像（Tracked Image）、环境探头（Environment Probe）、人脸（Face）、跟踪物体（Tracked Object）、共享参与者（AR Participant）这八类对象称为可跟踪对象（trackable），也就是说 AR Foundation 目前可以实时检测跟踪处理这八类对象。当然，在运行时这八类对象是否可用

还与底层 Provider 相关,如三维物体检测跟踪目前只在 ARKit 中被支持。

为了方便管理各类可跟踪对象,AR Foundation 为每一类可跟踪对象建立了一个对应的管理器 (Manager),一种管理器只管理对应的一类可跟踪对象,这种设计大大地提高了灵活性,可以非常方便地添加或者删除某类管理器,从而达到只对特定可跟踪对象进行处理的目的。另外,这八类管理器在界面及功能 (包括事件及 API) 设计上非常相似,掌握一个管理器的用法就可以类推到其他管理器的操作上,提供了统一的使用界面,方便开发人员的操作使用。

需要特别注意的是,可跟踪对象管理器只能挂载在 AR Session Origin 对象上,这是因为 AR Session Origin 组件定义了 AR 会话空间与 Unity 空间的转换关系,可跟踪对象管理器需要利用 AR 会话空间原点及其与 Unity 空间的转换关系才能将可跟踪对象定位在 Unity 空间中的正确位置。

2.3.1 可跟踪对象管理器

在 AR Session Origin 对象上添加一种可跟踪对象管理器即可开始对这种类型的可跟踪对象进行管理 (包括启用、禁用跟踪),删除或者禁用一种可跟踪对象管理器即终止对这种类型的可跟踪对象的管理,AR Foundation 对可跟踪对象的这种处理方式有利于简化对可跟踪对象的管理。各管理器与可跟踪对象的对应关系如表 2-3 所示。

表 2-3 可跟踪对象及其对应管理器

可跟踪对象	可跟踪对象管理器	描 述
ARPlane	ARPlaneManager	检测与管理平面,包括垂直平面与水平平面
ARPointCloud	ARPointCloudManager	检测与管理点云
ARAnchor	ARAnchorManager	检测与管理锚点,可以通过 ARAnchorManager.AddAnchor 和 ARAnchorManager.RemoveAnchor 添加和移除锚点
ARTrackedImage	ARTrackedImageManager	检测与管理二维图像跟踪
AREnvironmentProbe	AREnvironmentProbeManager	管理环境探头生成环境立方体贴图
ARFace	ARFaceManager	检测与管理人脸跟踪
ARTrackedObject	ARTrackedObjectManager	检测与管理三维物体对象跟踪
ARParticipantManager	ARParticipant	在多人协作会话中检测和跟踪参与者
ARRaycast	ARRaycastManager	处理与射线检测相关工作

在 AR Session Origin 对象上添加某类可跟踪对象管理器后,AR Foundation 即开始对该类可跟踪对象的位置、姿态进行跟踪计算,持续地对可跟踪对象的状态进行更新,但管理器并不负责可视化可跟踪对象,即管理器只负责存储可跟踪对象的姿态及状态信息,而不对数据进行其他加工处理。如 AR Plane Manager 管理器,在检测到平面后,它会存储检测到平面的位置、尺寸、边界等相关信息,但并不会可视化已检测到的平面,可视化平面的工作由其他脚本或者组件完成。

如前所述, 启用或者添加可跟踪管理器将启用某类跟踪功能, 例如, 可以通过启用或禁用 AR Plane Manager 组件来启用或者禁用平面检测功能。另外需要注意的是, 跟踪对象是一项非常消耗资源的工作, 因此最好在不使用某类可跟踪对象时禁用或者删除其管理器。

一类可跟踪对象管理器只能管理与其相应的一类可跟踪对象, 但可以管理一个或者多个该类可跟踪对象, 如 AR Plane Manager 管理器可以管理一个或者多个检测到的平面, 每个可跟踪对象管理器都提供了 trackables 属性, 通过该属性可以获取其检测管理的所有可跟踪对象, 典型代码如下:

```
// 第2章 /2-4
var planeManager = GetComponent<ARPlaneManager>();
foreach (ARPlane plane in planeManager.trackables)
{
    // 进行相应处理
}
```

访问 trackables 属性返回一个 TrackableCollection 类型集合对象, 如果知道可跟踪对象 ID, 则可以通过管理器直接检索获取该可跟踪对象, 如 AR Plane Manager 管理器可使用其 GetPlane(TrackableId) 方法获取指定检测到的平面, 某些管理器还提供了 TryGetTrackable() 方法, 在使用时查阅 API 文档可了解更多使用详情。

2.3.2 可跟踪对象事件

每个可跟踪对象都可以被添加、更新、删除。在每一帧中, 管理器会对该类所有可跟踪对象的状态进行检查, 添加新检测到可跟踪对象、更新现跟踪对象姿态、删除陈旧过时的可跟踪对象。有时可能需要在特定事件发生时执行一些操作, 如在检测到平面时自动在检测到的平面上种植小草, 为此, AR Foundation 所有可跟踪对象管理器都会提供一种状态变更事件, 可以通过订阅对应事件进行处理, 如表 2-4 所示。

表 2-4 可跟踪对象管理器常用事件

可跟踪对象管理器	常用事件
ARPlaneManager	planesChanged
ARPointCloudManager	pointCloudsChanged
ARAnchorManager	anchorsChanged
ARTrackedImageManager	trackedImagesChanged
AREnvironmentProbeManager	environmentProbesChanged
ARFaceManager	facesChanged
ARTrackedObjectManager	trackedObjectsChanged
ARParticipantManager	participantsChanged

2.3.3 管理可跟踪对象

AR Foundation 中的可跟踪对象, 有的可以手动添加或删除, 有的则自动管理不需人工参与, 如 AR Anchor 和 Environment Probes, 这两个可跟踪对象可以手动添加或删除, 而 Plane 和 Face 等其他可跟踪对象则完全由 AR Foundation 自动管理。PointCloud 比较特别, 它既可以自动管理, 也可以手动添加或者删除。

由于每一类可跟踪对象都由对应的管理器进行管理, 因此, 不应该直接尝试去销毁 (Destroy) 可跟踪对象, 事实上如果强行使用 Destroy() 方法销毁可跟踪对象, 则会导致运行时错误。对于那些可以手动添加或删除的可跟踪对象, 相应的管理器则提供了移除方法, 例如移除一个 AR Anchor, 只需调用 ARAnchorManager.RemoveAnchor(ARAnchor) 方法, 而不应当直接使用 Destroy() 方法销毁可跟踪对象。

在添加一个可跟踪对象时, AR Foundation 需要做一系列的准备工作, 因此, 在添加一个可跟踪对象后, 它并不会马上就被系统所跟踪, 而是直到系统准备完毕并报告可跟踪对象已经添加到 AR Foundation 中才真正完成可跟踪对象的添加操作, 这个过程和时间因可跟踪对象类型而异。为明确可跟踪对象的状态, 所有的可跟踪对象都有一个 pending 属性, 当 pending 属性为 true 时标识该可跟踪对象已执行添加操作, 但还没有真正添加到 AR Foundation 跟踪系统中, 通过检查这个属性可获取该可跟踪对象的跟踪情况, 典型代码如下:

```
// 第 2 章 / 2-5
var anchor = AnchorManager.AddAnchor(new Pose(position, rotation));
Debug.Log(anchor.pending);           // 返回值为 true

// 该方法不会在当前帧执行, 直到可跟踪对象真正添加到跟踪系统中才执行
void OnAnchorsChanged(ARAnchorsChangedEventArgs eventArgs)
{
    foreach (var anchor in eventArgs.added)
    {
        // 正在跟踪的对象
    }
}
```

在上述代码中, 添加一个 ARAnchor 后马上检查其状态, 其 pending 属性值为 true, 意味着该可跟踪对象还未真正添加到 AR Foundation 跟踪系统中, 因此该管理器的 OnAnchorsChanged 事件不会被触发, 直到真正添加成功 (可能 2 帧或者 3 帧后才会触发)。

有时可能需要禁用某些可跟踪对象的行为, 但又不是完全禁用管理器对可跟踪对象的管理, 例如不需要渲染已被检测到的平面, 但又不禁用平面检测功能, 那么这时可以通过遍历对特定或者全部的正在跟踪的对象进行处理, 典型代码如下:

```
// 第 2 章 / 2-6
var planeManager = GetComponent<ARPlaneManager>();
foreach (var plane in planeManager.trackables)
```



```
{
    plane.gameObject.SetActive(false);
}
```

在上述示例代码中，将所有检测到的平面禁用以阻止其渲染。

当可跟踪对象管理器检测到一个可跟踪对象后会实例化一个该对象的预制体 (prefab)，这个预制体必须挂载着该类可跟踪对象的对应组件，如平面预制体必须有 AR Plane 组件。如果开发人员没有提供这个预制体，则管理器将创建一个空的挂载该类可跟踪对象组件的对象，如果提供的预制体没有挂载该类可跟踪对象的对应组件，则管理器将自动为其添加一个，例如当 AR Plane Manager 管理器检测到一个平面时，如果其 Plane Prefab 属性有赋值，则它将使用这个预制体实例化一个 AR Plane 对象；如果提供的预制体没有 AR Plane 组件，则自动为其挂载一个；如果 Plane Prefab 属性没有赋值，则 AR Plane Manager 组件将创建一个只有 AR Plane 组件的空对象。

2.4 会话管理

在 AR 应用中，AR 会话极为重要，它管理着应用的全生命周期。在 AR 应用运行时，可能需要暂停、重置 AR 执行，甚至可能需要完全重新初始化，这时就需要直接对 AR 会话进行操作，因为 AR 会话负责整个应用生命周期管理，AR 会话的改变将直接导致整个应用行为的改变。

在 AR Foundation 中，提供了 4 种对 AR 会话的管理操作，如表 2-5 所示。

表 2-5 Session 常见管理操作

操 作	常用事件
Pause	暂停 AR 会话，设备将暂停环境检测及可跟踪对象的跟踪，也将暂停图像和运动传感器数据采集，在暂停 AR 会话时，AR 应用不消耗 CPU 及 GPU 资源
Resume	恢复被暂停的 AR 会话，在恢复会话后，设备将尝试恢复环境检测及用户状态跟踪，在这个过程中原可跟踪对象可能会出现漂移
Reset	重置 AR 会话，这将清除所有可跟踪对象并高效地开始新的 AR 会话，用户创建的虚拟物体对象不会被清除，但因为所有可跟踪对象都已被清除，因此虚拟物体会出现漂移
Reload	重新载入一个新的 AR 会话，这个过程用于模拟场景切换，因为需要销毁原 AR 会话重新初始化一个新 AR 会话，执行时间比重置方式长

在 AR Foundation 中，ARSession 类有一个 bool 值类型的 enable 属性，通过它可以控制 AR 会话的暂停和恢复，其还有一个 Reset() 方法，可以直接进行重置操作，典型代码如下：

```
// 第2章 / 2-7
vpublic ARSession mSession;
private void PauseSession()
{
```

```

        if (mSession != null)
            mSession.enabled = false;
    }
    private void ResumeSession()
    {
        if (mSession != null)
            mSession.enabled = true;
    }
    private void ResetSession()
    {
        if (mSession != null)
            mSession.Reset();
    }
    private void ReloadSession()
    {
        if (mSession != null)
        {
            StartCoroutine(DoReload());
        }
    }

    System.Collections.IEnumerator DoReload()
    {
        Destroy(mSession.gameObject);
        yield return null;
        if (sessionPrefab != null)
        {
            mSession = Instantiate(sessionPrefab).GetComponent<ARSession>();
            DeBug.Log("重载 Session 成功!");
        }
    }
}

```

由于执行重载时间较长,上述方法中使用了协程方式防止应用卡顿。

除此之外,AR Foundation 还提供了一个 LoaderUtility 工具类,利用该工具可以方便地在 AR 场景与非 AR 场景之间进行切换。LoaderUtility 工具类提供了 Initialize()、Deinitialize()、GetActiveLoader() 3 种方法协助进行 AR 会话及 AR Foundation 子系统的管理。

在多场景切换时,由于 AR Foundation 中的组件会启用或者禁用子系统,但这些子系统实际上是由 AR Foundation 内部进行管理的,并不允许直接被销毁 (Destroy),从而导致这些子系统生命周期跨多场景,LoaderUtility 工具类就是为了解决此类问题,通过 Initialize()、Deinitialize() 方法可以创建或者彻底销毁子系统对象,从而优化应用性能及表现,典型的应用代码如下:

```

// 第 2 章 / 2-8
// 从非 AR 场景到 AR 场景切换
static void LoadARScene(string sceneName)

```

```
{
LoaderUtility.Initialize();
    SceneManager.LoadScene(sceneName, LoadSceneMode.Single);
}
// 从 AR 场景到非 AR 场景切换
static void LoadNormalScene(string sceneName)
{
    SceneManager.LoadScene(sceneName, LoadSceneMode.Single);
    LoaderUtility.Deinitialize();
}
```




ARCore 是一个高级 AR 应用开发引擎，高内聚、易使用，特别是集成到 AR Foundation 框架后，提供了简洁统一的使用界面，这使利用其开发 AR 应用变得非常高效。ARCore 运动跟踪稳定性好，并且支持多传感器融合（如深度传感器、双目相机），性能消耗低，有利于营造沉浸性更好的 AR 体验。本章主要阐述 ARCore 本身的技术能力和与 ARCore 扩展包相关的知识。

3.1 ARCore 概述及主要功能

2017 年 6 月，苹果公司发布了 ARKit SDK，它能够帮助用户在移动端快速实现 AR 功能。ARKit 的发布推动了 AR 概念的普及，但 ARKit 只能用于苹果公司自家的移动终端，无法应用到 Android 平台。2017 年 8 月，谷歌公司正式发布对标 ARKit SDK 的 ARCore^①，将 AR 能力带入 Android 平台，ARCore 也是一套用来创建 AR 的 SDK，利用该工具包可以为现有及将来的 Android 手机提供 AR 功能，它通过采集环境信息及传感器数据使手机 / 平板具备感知环境、了解现实世界、支持虚实交互的能力，ARCore 还为 Android 和 iOS 平台同时提供了 API，支持 AR 体验共享。

ARCore 主要做两件工作，即跟踪用户设备姿态和构建对现实环境的理解。ARCore 利用 SLAM 技术进行运动跟踪并构建环境三维结构信息，除此之外，ARCore 还支持检测平坦的表面（如桌面或地面）、估计周围环境的光照信息。借助 ARCore 对现实世界的理解，我们能够以一种与现实世界无缝融合的方式添加虚拟物体、注释或其他信息，例如可以将一只打盹的小猫放在咖啡桌的一角，或者利用艺术家的生平信息为一幅画添加注释。

虽然 ARCore 出现时间比 ARKit 晚，但事实上 ARCore 项目开展比 ARKit 早，在 2014 年，谷歌公司展示了其 Tango 项目成果，Tango 是谷歌公司基于 FlyBy 公司 VIO 技术发展起来的 AR 技术，技术比 ARCore 更复杂，融合了更多传感器，需要额外的硬件辅助实现增强现实，过高的门槛使消费者甚至开发者难以触及，这导致 Tango 技术并未能大规模推广。苹

^① ARCore 后来被更名为 Google Play Services for AR，本书遵循原名，仍使用 ARCore 进行描述。

果公司的 ARKit SDK 发布后, 由于其基于现有移动终端硬件平台, 不需要额外硬件支持, 大大降低了 AR 使用者的门槛, 取得了良好的市场反响。为抢占技术高点, 谷歌公司也迅速在 Tango 的基础上推出了 ARCore, 并提供了 Android、iOS、Unity、Unreal、Java 多个开发平台的 API。

ARCore 是在移动手机广泛普及的情况下, 谷歌公司适应时代潮流推出的 AR 开发工具, 但其可以追溯到 Tango 这个已经研究很长时间的 AR 技术项目, 事实上 ARKit 也基于 FlyBy 公司的技术, 因此, ARKit 和 ARCore 技术同源且具有基本相似的功能。

ARCore 通过 3 个主要功能融合虚拟内容与现实场景, 即运动跟踪、环境理解、光照估计。这 3 个功能是 ARCore 实现各项 AR 功能的基础, 也是 ARCore 技术框架中最重要的支柱。

3.1.1 运动跟踪

在三维空间中跟踪设备运动并最终定位其位姿是任何 AR 应用的基础, ARCore 通过 SLAM 算法采集来自设备摄像头的图像数据和来自运动传感器的运动数据, 综合分析并估测设备随着时间推移而相对于周围世界的姿态变化, 如图 3-1 所示。通过将渲染三维内容的虚拟相机姿态与 SLAM 算法计算出的设备摄像头姿态对齐, 就能够从正确的透视角度渲染虚拟内容, 将渲染的虚拟内容叠加到从设备摄像头获取的图像上, 从而实现虚实场景的几何一致性, 使虚拟内容看起来就像真实世界的一部分。



图 3-1 ARCore 运动跟踪示意图

3.1.2 环境理解

ARCore 通过检测环境特征点和平面来不断改进它对现实世界环境的理解, 通过检查位于常见水平或垂直表面 (如桌子或墙壁) 上的成簇特征点构建平面, 并确定平面的边界, 利用这些数据信息就可以将虚拟物体放置于平坦的真实环境表面上, 实现虚拟物体和真实环境无缝融合, 营造虚实一致的体验, 如图 3-2 所示。

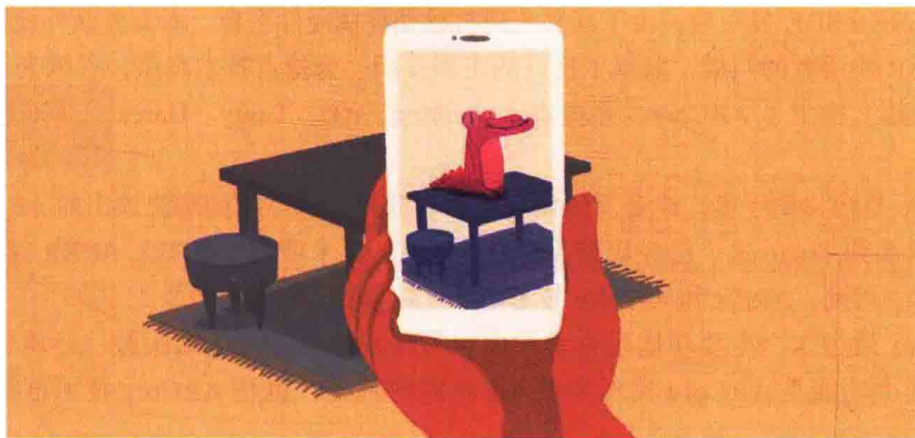


图 3-2 ARCore 通过检测特征点和平面来改进对环境理解

3.1.3 光照估计

ARCore 可以检测设备所在空间环境光的相关信息，估计光照强度和色彩，利用这些光照信息，可以使用与真实环境光相同的光照信息渲染虚拟物体，使虚拟物体和周边真实物体光照效果保持一致，提升虚拟物体的真实感，如图 3-3 所示，处于强光中的虚拟猫模型和处于阴影中的虚拟猫模型光照效果保持了与真实场景中光照的一致性。ARCore 还可以估计光源的位置和方向，从而支持虚拟物体生成与真实光照一致的阴影，进一步提升虚拟物体的真实感。

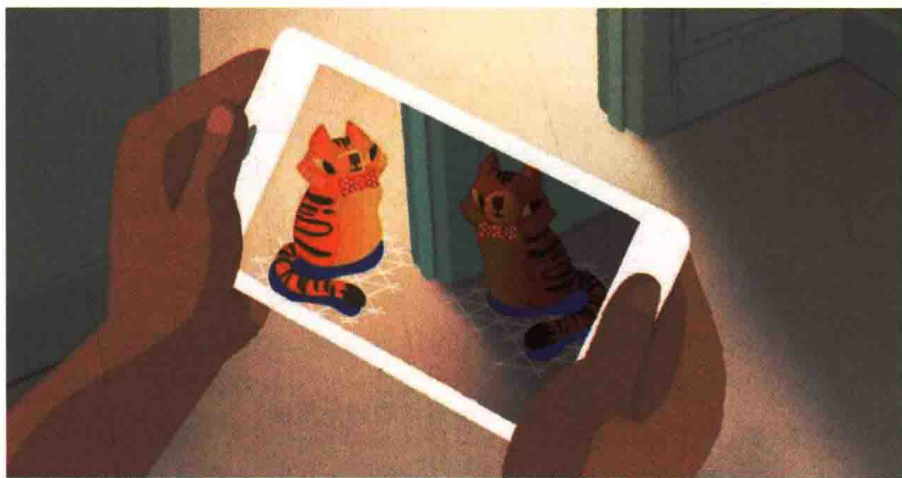


图 3-3 ARCore 光照估计效果示意图

3.1.4 ARCore 的不足

1. 运动跟踪失效

ARCore SDK 提供了稳定的运动跟踪功能，也提供了高精度的环境感知功能，但受限于

视觉 SLAM 技术的技术水平和能力, ARCore 的运动跟踪在以下情况时会失效。

1) 在运动中进行运动跟踪

如果用户在火车或者电梯中使用 ARCore, 这时 IMU 传感器获取的数据不仅包括用户的移动数据(实际是加速度), 也包括火车或者电梯的移动数据(实际是加速度), 这样将导致 SLAM 数据融合问题, 从而引起漂移甚至完全失败。

2) 跟踪动态的环境

如果用户将设备对着波光粼粼的湖面, 这时从设备摄像头采集的图像信息是不稳定的, 这将引起图像特征点提取匹配问题, 进而导致跟踪失败。

3) 热漂移

相机感光元器件与 IMU 传感器都是对温度敏感的元器件, 其校准通常只会针对某一个或者几个特定温度, 但在用户设备使用过程中, 随着时间的推移设备会发热, 发热会影响摄像头采集图像的颜色/强度信息和 IMU 传感器测量的加速度信息的准确性, 从而导致误差, 表现出来的效果就是跟踪的物体漂移。

4) 昏暗环境

基于 VIO 的运动跟踪效果与环境光照条件有很大关系, 昏暗环境采集的图像信息对比度低, 这对提取图像特征点信息非常不利, 进而影响跟踪的准确性, 这也会导致基于 VIO 的运动跟踪失败。

2. ARCore 存在的不足

除运动跟踪问题外, 由于移动设备资源限制或其他问题, ARCore 还存在以下不足。

1) 环境理解需要时间

ARCore 虽然对现实场景表面特征点提取与平面检测进行了非常好的优化, 但还是需要一个相对比较长的时间, 在这个过程中, ARCore 需要收集环境信息构建对现实世界的理解。这是一个容易让不熟悉 AR 的使用者产生困惑的阶段, 因此, AR 应用中应当设计良好的用户引导, 同时帮助 ARCore 更快地完成初始化。

2) 运动处理有滞后

当用户设备移动过快时会导致摄像头采集的图像模糊, 从而影响 ARCore 对环境特征点的提取, 进而造成运动跟踪误差, 表现为跟踪不稳定或者虚拟物体漂移。

3) 弱纹理表面检测问题

ARCore 使用的 VIO 技术很难在光滑、无纹理、反光的场景表面提取所需的特征值, 因而无法构建对环境的理解和检测识别平面, 如很难检测跟踪光滑大理石地面和白墙。

4) 鬼影现象

虽然 ARCore 在机器学习的辅助下对平面边界预测作了很多努力, 但由于现实世界环境的复杂性, 检测到的平面边界仍然还不够准确, 因此, 添加的虚拟物体可能会出现穿越墙壁的现象, 所以对开发者而言, 应当鼓励使用者在开阔的空间或场景中使用 AR 应用程序。

3.2 运动跟踪原理

在第1章中,我们对AR技术原理进行过简要学习,ARCore运动跟踪所采用的技术路线与其他移动端AR SDK相同,也采用VIO与IMU结合的方式进行SLAM定位跟踪。本节将更加深入地学习ARCore运动跟踪原理,从而加深对ARCore在运动跟踪方面优劣势的理解,并在开发中尽量避免劣势或者采取更加友好的方式扬长避短。

3.2.1 ARCore 坐标系

实现虚实融合最基本的要求是实时跟踪用户(设备)的运动,始终保持虚拟相机与设备摄像头的对齐,并依据运动跟踪结果实时更新虚拟元素的姿态,这样才能在现实世界与虚拟世界之间建立稳定精准的联系。运动跟踪的精度与质量直接影响AR的整体效果,任何延时、误差都会造成虚拟元素抖动或者漂移,从而破坏AR的真实感和沉浸性。

在进一步学习运动跟踪之前,先了解一下ARCore空间坐标系,在不了解AR坐标系的情

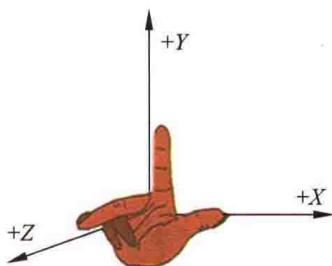


图 3-4 ARCore 采用右手坐标系

况下,阅读或者实现代码可能会感到困惑(如在AR空间中放置的虚拟物体会在Y轴上有 180° 偏转)。ARCore采用右手坐标系(包括世界坐标系、相机坐标系、投影坐标系,这里的相机指渲染虚拟元素的相机),而Unity使用左手坐标系。右手坐标系Y轴正向朝上,Z轴正向指向观察者自己,X轴正向指向观察者右侧,如图3-4所示。

当用户在实际空间中移动时,ARCore坐标系上的距离增减遵循表3-1所示的规律。

表 3-1 ARCore 采用的坐标系与设备移动关系

移动方向	描 述
向右移动	X 增加
向左移动	X 减少
向上移动	Y 增加
向下移动	Y 减少
向前移动	Z 减少
向后移动	Z 增加

3.2.2 ARCore 运动跟踪分类

ARCore运动跟踪支持3DoF和6DoF两种模式,3DoF只跟踪设备旋转,因此是一种受限的运动跟踪方式,通常不会使用这种运动跟踪方式,但在一些特定场合或者6DoF跟踪失效的

情况下,也有可能用到。

DoF 概念与刚体在空间中的运动相关,可以解释为“刚体运动的不同基本方式”。在客观世界或者虚拟世界中,都采用三维坐标系来精确定位一个物体的位置。如一个有尺寸的刚体放置在坐标系的原点,那么这个物体的运动整体上可以分为平移与旋转两类(刚体不考虑缩放),同时,平移又可以分为3个度:前后(沿Z轴移动)、左右(沿X轴移动)、上下(沿Y轴移动);旋转也可以分3个度:俯仰(围绕X轴旋转)、偏航(围绕Y轴旋转)、翻滚(围绕Z轴旋转)。通过分析计算,刚体的任何运动方式均可由这6个基本运动方式来表达,即6DoF的刚体可以完成所有的运动形式,具有6DoF的刚体在空间中的运动是不受限的。

具有6DoF的刚体可以到达三维坐标系的任何位置并且可以朝向任何方向。平移相对来说比较好理解,即刚体沿X、Y、Z3个轴之一运动,旋转其实也是以X、Y、Z3个轴之一为旋转轴进行旋转。在计算机图形学中,常用一些术语来表示特定的运动,这些术语如表3-2所示。

表 3-2 刚体运动术语及其含义

术 语	描 述
Down	向下
Up	向上
Strafe	左右
Walk	前进后退
Pitch	围绕X轴旋转,即上下打量,也叫俯仰角
Rotate	围绕Y轴旋转,即左右打量,也叫偏航角(yaw)
Roll	围绕Z轴旋转,即翻滚,也叫翻滚角

在AR空间中描述物体的位置和方向时经常使用姿态(Pose)这个术语,姿态的数学表达为矩阵,既可以用矩阵来表示物体平移,也可以用矩阵来表示物体旋转。为了更好地平滑及优化内存的使用,通常还会使用四元数来表达旋转,四元数允许以简单的形式定义三维旋转的所有方面。

1. 方向跟踪

在AR Foundation中,可以通过将场景中AR Session对象上的AR Session组件Tracking Mode属性设置为Rotation Only值使用3DoF方向跟踪,即只跟踪设备在X、Y、Z轴上的旋转运动,如图3-5所示,表现出来的效果类似于站立在某个点上下左右观察周围环境。方向跟踪只跟踪方向变化而不跟踪设备位置变化,由于缺少位置信息,无法从后面去观察放置在地面上的桌子,因此这是一种受限的运动跟踪方式。在AR中采用这种跟踪方式时虚拟元素会一直飘浮在摄像头图像之上,即不能固定于现实世界中。

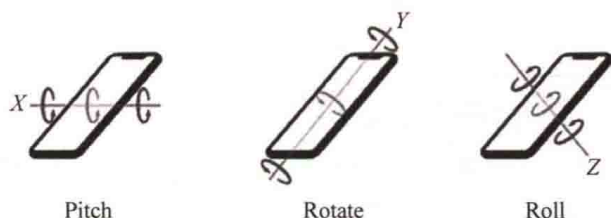


图 3-5 ARCore 中 3DoF 跟踪示意图

当采用 3DoF 进行运动跟踪时, 无法使用平面检测功能, 也无法使用射线检测功能。

2. 世界跟踪

在 AR Foundation 中, 可以通过将场景中 AR Session 对象上的 AR Session 组件 Tracking Mode 属性设置为 Position And Rotation 值使用 6DoF 跟踪, 这是默认跟踪方式, 使用这种方式, 既跟踪设备在 X 、 Y 、 Z 轴上的旋转运动, 也跟踪设备在 X 、 Y 、 Z 轴上的平移运动, 能实现对设备姿态的完全跟踪, 如图 3-6 所示。

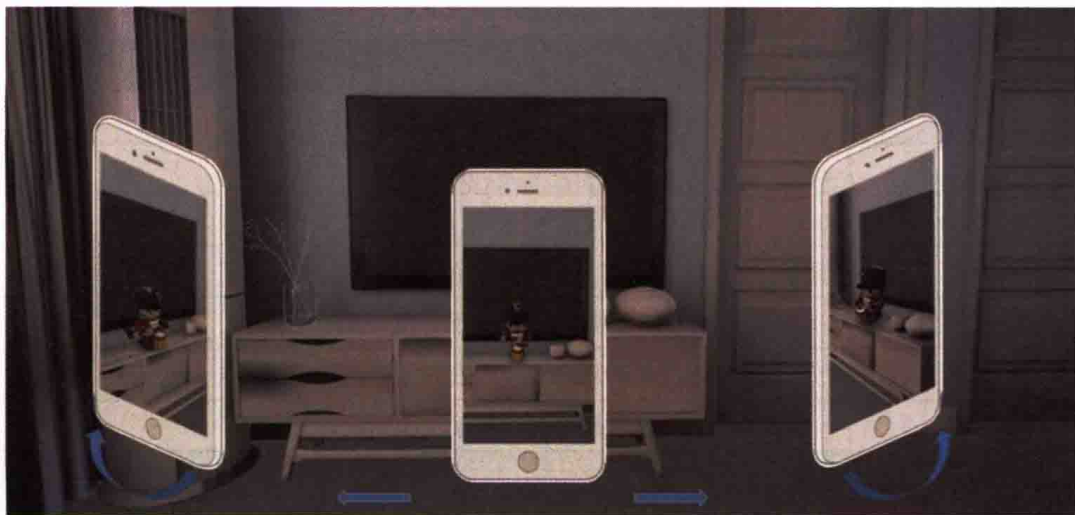


图 3-6 ARCore 中 6DoF 跟踪示意图

6DoF 的运动跟踪方式 (世界跟踪) 可以营造完全真实的 AR 体验, 通过世界跟踪, 能从不同距离、方向、角度观察虚拟物体, 就好像虚拟物体真正存在于现实世界中一样。在 ARCore 中, 通常通过世界跟踪方式创建 AR 应用, 使用世界跟踪时, 支持平面检测、射线检测, 还支持检测识别摄像头采集图像中的二维图像等。

3.2.3 ARCore 运动跟踪

ARCore 通过 VIO + IMU 方式进行运动跟踪, 图像数据来自设备摄像头, IMU 数据来自运动传感器, 包括加速度计和陀螺仪, 它们分别用于测量设备的实时加速度与角速度。运动传感器非常灵敏, 每秒可进行 1000 次以上的数据检测, 能在短时间跨度内提供非常及时准确

的运动信息,但运动传感器也存在测量误差,由于检测速度快,这种误差累积效应就会非常明显(微小的误差以每秒1000次的速度累积会迅速变大),因此,在较长的时间跨度后,跟踪就会变得完全失效。

ARCore 为了消除 IMU 存在的误差累积漂移,采用 VIO 的方式进行跟踪校准,VIO 基于计算机视觉计算,该技术可以提供非常高的计算精度,但付出的代价是计算资源与计算时间。ARCore 为了提高 VIO 跟踪精度采用了机器学习方法,因此,VIO 处理速度相比于 IMU 要慢得多,另外,计算机视觉处理对图像质量要求非常高,对相机运动速度非常敏感,因为快速的相机运动会导致采集的图像模糊。

ARCore 充分吸收利用了 VIO 和 IMU 各自的优势,利用 IMU 的高更新率和高精度进行较小时间跨度的跟踪,利用 VIO 对较长时间跨度 IMU 跟踪进行补偿,融合跟踪数据向上提供运动跟踪服务。IMU 信息来自运动传感器的读数,精度取决于传感器本身。VIO 信息来自于计算机视觉处理结果,因此精度受到较多因素的影响,下面主要讨论 VIO,VIO 进行空间计算的原理如图 3-7 所示。



图 3-7 VIO 进行空间计算原理图

在 AR 应用启动后,ARCore 会不间断地捕获从设备摄像头采集的图像信息,并从图像中提取视觉差异点(特征点),ARCore 会标记每个特征点(每个特征点都有 ID),并会持续地跟踪这些特征点,当设备从另一个角度观察同一空间时(设备移动了位置),特征点就会在两张图像中呈现视差,如图 3-7 所示,利用这些视差信息和设备姿态偏移量就可以构建三角测量,从而计算出特征点缺失的深度信息,换言之,可以通过从图像中提取的二维特征进行三维重建,进而实现跟踪用户设备的目的。

从 VIO 工作原理可以看到,如果从设备摄像头采集的图像不能提供足够的视差信息,则无法进行空间三角计算,从而无法解算出空间信息,因此,若要在 AR 应用中使用 VIO,则设备必须移动位置(X 、 Y 、 Z 方向均可),无法仅仅通过旋转达到目的。

在通过空间三角计算后,特征点的位置信息被解算出来,这些位置信息会存储到对应特征点上。随着用户在现实世界中探索的进行,一些不稳定的特征点被剔除,一些新的特征点

被加入,并逐渐形成稳定的特征点集合,这个特征点集合称为点云,点云坐标原点为 ARCore 初始化时的设备位置,点云地图就是现实世界在 ARCore 中的数字表达。

VIO 跟踪的流程图如图 3-8 所示,从流程图可以看到,为了优化性能,计算机视觉计算并不是每帧都执行。VIO 跟踪主要用于校正补偿 IMU 在时间跨度较长时存在的误差累积,每帧执行视觉计算不仅会消耗大量计算资源,而且没有必要。

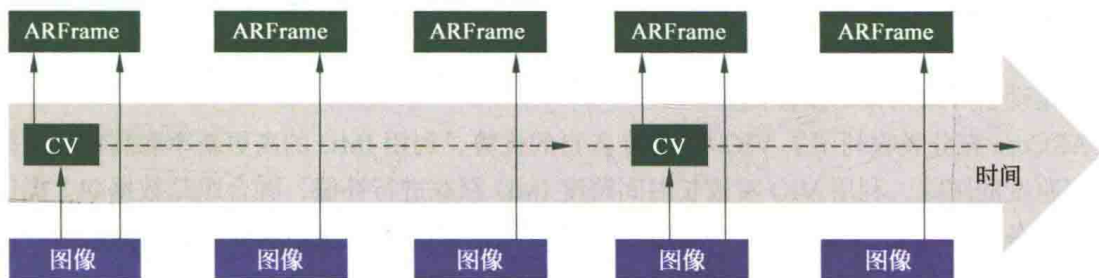


图 3-8 VIO 跟踪流程示意图

VIO 也存在误差,这种误差随着时间的积累也会变得很明显,表现出来就是放置在现实空间中的虚拟元素会出现一定的漂移。为抑制这种漂移,ARCore 使用锚点 (Anchor) 的方式绑定虚拟元素与现实空间环境,同时 ARCore 也会实时地对设备摄像头采集的图像进行匹配计算,如果发现当前采集的图像与之前某个时间点采集的图像匹配 (用户在同一位置以同一视角再次观察现实世界时),ARCore 就会对特征点的信息进行修正,从而优化跟踪 (回环检测)。

ARCore 融合了 VIO 与 IMU 跟踪各自的优点,提供了非常稳定的运动跟踪能力,也正是因为稳定的运动跟踪使利用 ARCore 制作的 AR 应用体验非常好。

3.2.4 ARCore 使用运动跟踪的注意事项

通过对 ARCore 运动跟踪原理的学习,现在可以很容易地理解前文所列 ARCore 的不足,因此,为了得到更好的跟踪质量,需要注意以下事项。

(1) 运动跟踪依赖于不间断输入的图像数据流与传感器数据流,某一种方式短暂地受到干扰不会对跟踪造成太大的影响,如用手偶尔遮挡摄像头图像采集不会导致跟踪失效,但如果中断时间过长,则跟踪就会变得很困难。

(2) VIO 跟踪精度依赖于采集图像的质量,低质量的图像 (如光照不足、纹理不丰富、模糊) 会影响特征点的提取,进而影响跟踪质量。

(3) 当 VIO 数据与 IMU 数据不一致时会导致跟踪问题,如视觉信息不变而运动传感器数据变化 (如在运动的电梯里),或者视觉信息变化而运动传感器数据不变 (如摄像头对准波光粼粼的湖面),这都会导致数据融合障碍,进而影响跟踪效果。

开发人员很容易理解以上内容,但这些信息,使用者在进行 AR 体验时可能并不清楚,因此,必须实时地给予引导和反馈,不然会使用户困惑。ARCore 为辅助开发人员了解 AR 运动跟踪状态提供了实时的状态监视,将运动跟踪状态分为暂停 (PAUSED)、停止 (STOPPED)、

跟踪 (TRACKING) 3 种, 分别指示运动跟踪状态的 3 种情况 (在整合进 AR Foundation 时, 状态由 ARSessionState 枚举描述)。为提升用户的使用体验, 应当在跟踪受限或者不可用时给出明确的原因和操作建议, 引导使用者提高运动跟踪的精度和稳定性。

3.3 设备可用性检查

从第 1 章可知, 并不是所有的 Android 手机都支持 ARCore, 因此, 通常在使用 ARCore 之前需要进行一次设备支持性检查。在 AR Foundation 框架中, 可使用 ARSession 类静态属性检查设备支持性, 典型代码如下:

```
// 第 3 章 /3-1
public class DeviceCheck
{
    [SerializeField]
    private ARSession mSession;

    IEnumerator Check() {
        if ((ARSession.state == ARSessionState.None) ||
            (ARSession.state == ARSessionState.CheckingAvailability))
        {
            yield return ARSession.CheckAvailability(); // 检查设备支持性
        }

        if (ARSession.state == ARSessionState.Unsupported)
        {
            // 设备不支持, 需要启用其他备用方案
        }
        else
        {
            // 设备可用, 启动会话
            mSession.enabled = true;
        }
    }
}
```

设备支持状态由 ARSessionState 枚举描述, 其枚举值如表 3-3 所示。

表 3-3 ARSessionState 枚举值

枚举值	描述
None	状态未知, 未安装 ARCore
Unsupported	当前设备不支持 ARCore
CheckingAvailability	系统正在进行设备支持性检查
NeedsInstall	设备硬件支持 ARCore, 但需要安装 ARCore 工具包

续表

枚举值	描 述
Installing	设备正在安装 ARCore
Ready	设备支持 ARCore 并且已做好使用准备
SessionInitialized	ARCore 会话正在初始化, 还未建立运动跟踪
SessionTracking	ARCore 会话运动跟踪中

在 ARCore 运动跟踪启动后, 如果运动跟踪状态发生变化, 则 `ARSession.state` 值也会发生变化, 此外还可以通过订阅 `ARSession.stateChanged` 事件处理运动跟踪变化情况。

3.4 AR 会话生命周期管理与跟踪质量

AR 会话整合运动传感器数据和计算机视觉处理数据跟踪用户设备姿态, 为得到更好的跟踪质量, AR 会话需要持续的运动传感器数据和视觉计算数据。在启动 AR 应用后, ARCore 需要一点时间来收集足够多的视觉特征点信息, 在这个过程中, ARCore 是不可用的。在 AR 应用运行过程中, 由于一些异常情况 (如摄像头被覆盖), ARCore 的跟踪状态也会发生变化, 可以在需要时进行必要的处理 (如显示 UI 信息)。

1. AR 会话生命周期

AR 会话的基本生命周期如图 3-9 所示, 在刚启动 AR 应用时, ARCore 还未收集到足够的特征点和运动传感器数据信息, 无法计算设备的姿态, 这时的跟踪状态是不可用状态。在经过几帧之后, 跟踪状态会变为设备初始化状态 (`SessionInitialized`), 这种状态表明设备姿态已可用但精度可能会有问题。

再经过一段时间后, 跟踪状态会变为正常状态 (`SessionTracking`), 这时说明 ARCore 已准备好了, 所有的功能都已经可用。



图 3-9 ARCore 跟踪开始后的状态变化

2. 跟踪受限

在 AR 应用运行过程中, 由于环境的变化或者其他异常情况, ARCore 的跟踪状态会发生变化, 如图 3-10 所示。



图 3-10 ARCore 跟踪状态会受到设备及环境的影响

在 ARCore 状态受限时，基于环境映射的功能将不可用，如平面检测、射线检测、二维图像检测等。在 AR 应用运行过程中，由于用户环境变化或者其他异常情况，ARCore 可能在任何时间进入跟踪受限状态，如当用户将摄像头对准一面白墙或者房间中的灯突然关闭，这时 ARCore 就会进入跟踪受限状态。

3. 中断恢复

在 AR 应用运行过程中，AR 会话也有可能被迫中断，如在使用 AR 应用的过程中突然来电话，这时 AR 应用将被切换到后台。当 AR 会话被中断后，虚拟元素与现实世界将失去关联。在 ARCore 中断结束后会自动尝试进行重定位（Relocalization），重定位如果成功，虚拟元素与现实世界的关联关系会恢复到中断前的状态，包括虚拟元素的姿态及虚拟元素与现实世界之间的相互关系，如果重定位失败，则虚拟元素与现实世界的原有关联关系被破坏。

在重定位过程中，AR 会话的运动跟踪状态保持为受限状态，重定位成功的前提条件是使用者必须返回 AR 会话中断前的环境中，如果使用者已经离开，则重定位永远也不会成功（环境无法匹配）。整个过程如图 3-11 所示。



图 3-11 ARCore 跟踪中断及重定位时的状态变化

提示

重定位是一件容易让使用者困惑的操作，特别是对不熟悉 AR 应用、没有 AR 应用使用经验的使用者而言，重定位会让他们感到迷茫，所以在进行重定位时，应当通过 UI 或者其他视觉信息告知使用者，并引导使用者完成重定位操作。

在 AR 应用运行中，可以通过 `ARSession.stateChanged` 事件获取跟踪状态变化，在跟踪受限时提示用户原因，引导用户进行下一步操作，典型代码如下：

```
// 第3章 / 3-2
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;

public class TrackingReason : MonoBehaviour
{
    private bool mSessionTracking;           // 运动跟踪状态标识

    // 注册事件
```



```

void OnEnable()
{
    ARSession.stateChanged += ARSessionOnStateChanged;
}
// 取消事件注册
void OnDisable()
{
    ARSession.stateChanged -= ARSessionOnStateChanged;
}

//AR 会话状态变更事件
void ARSessionOnStateChanged(ARSessionStateChangedEventArgs obj)
{
    mSessionTracking = obj.state == ARSessionState.SessionTracking ? true : false;
    if (mSessionTracking)
        return;
    switch (ARSession.notTrackingReason)
    {
        case NotTrackingReason.Initializing:
            Debug.Log("AR 正在初始化");
            break;
        case NotTrackingReason.Relocalizing:
            Debug.Log("AR 正在进行重定位");
            break;
        case NotTrackingReason.ExcessiveMotion:
            Debug.Log("设备移动太快");
            break;
        case NotTrackingReason.InsufficientLight:
            Debug.Log("环境昏暗, 光照不足");
            break;
        case NotTrackingReason.InsufficientFeatures:
            Debug.Log("环境无特性");
            break;
        case NotTrackingReason.Unsupported:
            Debug.Log("设备不支持跟踪受限原因");
            break;
        case NotTrackingReason.None:
            Debug.Log("运动跟踪未开始");
            break;
    }
}
}

```

在 AR 运动跟踪受限时, AR Foundation 会通过 NotTrackingReason 枚举值标识跟踪受限原因, NotTrackingReason 各枚举值如表 3-4 所示。

表 3-4 NotTrackingReason 枚举值

枚举值	描述
None	运动跟踪未开始, 状态未知
Initializing	正在进行跟踪初始化
Relocalizing	正在进行重定位
InsufficientLight	光照不足, 环境昏暗
InsufficientFeatures	环境中特征点不足
ExcessiveMotion	设备移动太快, 造成图像模糊
Unsupported	Provider 不支持跟踪受限原因
CameraUnavailable	设备摄像头不可用

3.5 ARCore 扩展包

AR Foundation 是一个上层封装框架, 由于要考虑不同底层的共通性, 不能做到对底层 SDK 所有功能特性都完全支持, 因此, ARCore 提供了一个扩展包 (ARCore Extensions), 用于实现其特定的功能和更精细的控制, 如对硬件摄像头、云锚点、会话录制 (Session Record) 的配置控制, 并支持 iOS 端使用云锚点与 Android 互通, 实现 AR 体验跨端共享。

ARCore 扩展包是一个可选包, 如果不需要上文提及的功能, 则可以不引入, ARCore 扩展包并不能直接在 Unity 包管理器 (Unity Package Manager, UPM) 中导入, 需要先下载^①或者从 Unity 包管理器中填写 ARCore 扩展包的 GitHub 地址进行导入。下面以使用 UPM 导入进行演示, 在 Unity 菜单栏中, 依次选择 Window → Package Manager, 打开 UPM 对话框, 单击该对话框左上角的“+”符号, 在下拉菜单中选择 Add package from git URL 选项, 如图 3-12 所示, 在弹出的网址输入框中填写 <https://github.com/google-ar/arcore-unity-extensions.git>。

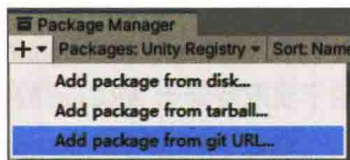


图 3-12 直接从 git URL 中导入 ARCore 扩展包

在导入 ARCore 扩展包后, 为了方便统一管理配置文件, 在工程窗口 (Project 窗口) 中新建一个文件夹, 命名为 Config, 然后在空白处右击, 在弹出菜单中依次选择 Create → XR → ARCore Extensions Config 创建 ARCoreExtensionsConfig 配置文件, 以同样的操作, 在弹出菜单中依次选择 Create → XR → Camera Config Filter 创建 ARCoreExtensionsCameraConfigFilter 配置文件。这两个配置文件, 一个用于配置云锚点, 另一个用于配置设备摄像头使用参数, 在后文使用时再详细阐述。

然后在层级窗口 (Hierarchy 窗口) 中右击, 在弹出菜单中依次选择 XR → ARCore Extensions 创建 ARCore Extensions 对象, 选择该对象, 在属性窗口 (Inspector 窗口) 中为 ARCore Extensions 组件各属性赋值 (将层级窗口中各相应对象和上文创建的两个配置文件设

^① 下载最新版本的网址为 <https://github.com/google-ar/arcore-unity-extensions/releases>。

置到相应的属性列中), 如图 3-13 所示。

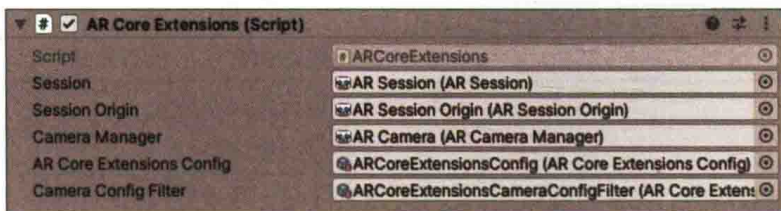


图 3-13 为 ARCore Extensions 组件各属性赋值

至此, 已经导入并完成了 ARCore 扩展包配置, 在具体使用时再详细阐述各功能配置的使用流程。

3.6 相机配置

移动设备基本有前置摄像头、后置摄像头、IMU 传感器, ARCore 利用这些传感器数据探索和理解现实环境。设备可能还有深度传感器, 能够检测目标距离、尺寸、范围等数据。这些传感器还有不同的性能技术参数, 如摄像头可以以 30 帧/秒或者 60 帧/秒运行, 所有这些构成了移动设备硬件环境的多样性。事实上, ARCore 可以充分利用这些传感器数据, 融合各类数据提供更好的使用体验。也可以通过 ARCore 扩展包进行相应配置以优化性能, 如可以使用相机图像元数据、帧缓冲和相机访问共享优化性能。

在 AR Foundation 中, 摄像头配置 (Camera Configurations) 描述了设备硬件传感器属性, 可以通过其子系统的 `XRCameraConfiguration` 类访问这些属性, 在 Android 平台, ARCore 扩展包在 `XRCameraConfiguration` 类基础之上提供了 `XRCameraConfigurationExtensions` 配置类, 用于获取特定于 ARCore 的属性, 可以使用这些属性进行更加细致的控制。

通过 `ARCameraManager.GetConfigurations()` 方法可以获取所有当前设备支持的相机配置, 通过 `CameraManager.currentConfiguration` 可以设置设备使用的相机参数, 典型代码如下^①:

```
// 第 3 章 / 3-3
using UnityEngine.XR.ARFoundation;
using Google.XR.ARCoreExtensions;

public ARCameraManager cameraManager; //ARCameraManager 组件

private void CameraConfigurations()
{
    using (NativeArray<XRCameraConfiguration> configurations = cameraManager.
        GetConfigurations(Allocator.Temp))
    {
        if (!configurations.IsCreated || (configurations.Length <= 0))
```

① 关于相机配置的更详细信息参见第 9 章。


```

{
    return;
}

// 迭代获取最大分辨率配置
var desiredConfig = configurations[0];
for (int i = 1; i < configurations.Length; ++i)
{
    // 选择最大帧率和分辨率的配置, 打开深度传感器
    if (configurations[i].GetFPSRange().y > desiredConfig.GetFPSRange().y &&
        configurations[i].GetTextureDimensions().x > desiredConfig.
GetTextureDimensions().x &&
        configurations[i].GetTextureDimensions().y > desiredConfig.
GetTextureDimensions().y &&
        configurations[i].CameraConfigDepthSensorUsage() ==
CameraConfigDepthSensorUsage.RequireAndUse)
    {
        desiredConfig = configurations[i];
    }
}

// 设置相机配置
if (desiredConfig != cameraManager.currentConfiguration)
{
    cameraManager.currentConfiguration = desiredConfig;
}
}

```

选择 3.5 节创建的 ARCoreExtensionsCameraConfigFilter 配置文件, 在属性窗口中展开其属性值, 如图 3-14 所示。

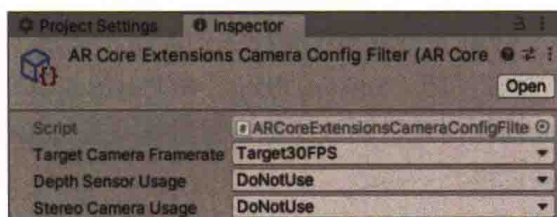


图 3-14 ARCore 扩展包相机配置界面

Target Camera Framerate 属性用于限制相机最高帧率, 在支持 60 帧/秒的设备上, ARCore 默认会使用高帧率, 这会对应用性能带来挑战, 也可以在下拉列表框中选择 30 帧/秒以限制最高帧率。

Depth Sensor Usage 属性用于设置是否使用深度传感器, 在配备有深度传感器的设备上, ARCore 默认会开启深度传感器, 这将提供更好的 AR 体验, 也可以选择 DoNotUse 禁用深度

传感器。

Stereo Camera Usage 属性用于设置是否使用双目传感器，在配备了双目传感器的设备上，ARCore 默认会开启该传感器，这将提高 SLAM 跟踪稳定性和质量，也可以选择 DoNotUse 禁用双目传感器。

除了在编辑时静态配置，也可以在应用运行时通过 ARCoreExtensions.OnChooseXRCameraConfiguration 事件动态地进行配置，典型代码如下：

```
// 第3章 / 3-4
public void Awake()
{
    ...
    arcoreExtensions.OnChooseXRCameraConfiguration = SelectCameraConfiguration;
    ...
}

// 执行动态相机配置
int SelectCameraConfiguration(List<XRCameraConfiguration> supportedConfigurations)
{
    int index = 0;
    ...
    // 自定义相机配置逻辑
    return index;
}
```

需要注意的是，使用动态配置时，必须确保在 ARCore 会话启动之前完成配置，通常是在 Awake() 方法中进行，这样才能在 AR 运行开始前生效。

3.7 会话录制与回放

如前文所述，AR 应用调试是一件非常烦琐且费时费力的工作，特别是对需要到现场进行调试的应用，如文旅、导航等与现地位置相关的应用，非常不友好，为解决这个问题，ARCore 引入了会话录制与回放功能 (Session Record & Playback)，利用该功能可以预先录制场景数据信息，在调试 AR 应用时可以回放这些场景数据并进行相应操作，因此可以对录制的会话进行重用，加速调试过程。

会话录制时会录制满足正常 SLAM 运动跟踪的所有信息，包括图像数据、IMU 数据及其他传感器数据 (如 ToF 传感器)，因此录制完的会话可以在其他平台回放，如图 3-15 所示，在回放时，AR 应用可以正常实现其他功能，如平面检测、点云重建等，与使用真机设备进行测试一样，从而达到一次录制多平台回放、会话重用的目的。

通过会话录制与回放，可以大幅度降低实地现场测试的需求，从而提高应用迭代速度。在默认情况下，出于计算性能考虑，ARCore 使用 640×480 像素分辨率的摄像头图像进行 SLAM 运动跟踪，而不是渲染到屏幕上的高分辨率图像，因此，默认的会话录制也会使用

640×480 像素分辨率, 通常这能满足回放的各项要求。如果确实希望录制高分辨率图像流, 则必须对摄像头进行配置, 此时 ARCore 进行运动跟踪依然会使用 640×480 像素分辨率图像, 同时, 还会采集配置文件中配置的高分辨率图像流, 该图像流仅用于播放(呈现效果), 并且会将高分辨率图像存储于视频主轨道, 而将用于 ARCore 运动跟踪的低分辨率图像存储于其他轨道, 需要注意的是, 同时捕获两个图像流会影响应用程序的性能, 并且不会提高会话回放的质量。

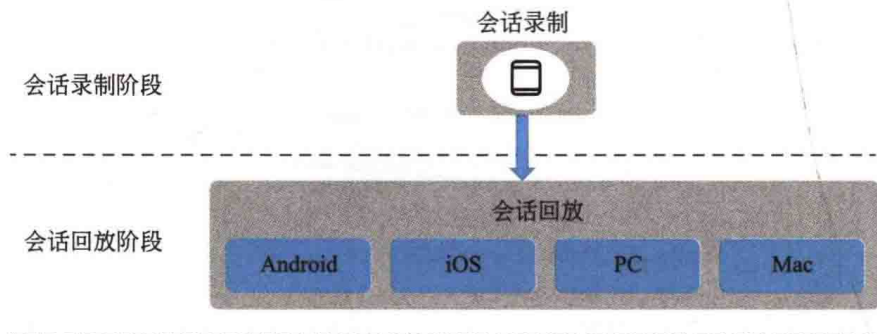


图 3-15 通过会话录制可以达到一次录制、多平台回放的目的

除此之外, 还可以采集其他传感器数据, 如设备的硬件深度传感器数据(如果有)。下面分成两种实现方式阐述会话录制与回放功能。

3.7.1 AR Foundation 录制与回放

AR Foundation 支持 ARCore 会话录制与回放^①, 该功能已内置到其子系统中, 因此使用起来非常方便, 在 Unity 中新建 SessionRecordAndPlayback.cs 脚本, 编写代码如下:

```
// 第3章 / 3-5
using System.IO;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARCore;

public class SessionRecordAndPlayback : MonoBehaviour
{
    private ARSession mSession;           // 会话组件
    private string mVideoFilePath;        // 文件存储路径
    void Awake()
    {
        mSession = GetComponent<ARSession>();
        mVideoFilePath = Path.Combine(Application.persistentDataPath,
            "ARCoreSession.mp4");
    }
}
```

① 该功能需要 AR Foundation 4.2 及以上版本。


```

    }
    // 开始录制会话
    public void StartRecord()
    {
        if (mSession.subsystem is ARCoreSessionSubsystem subsystem)
        {
            var session = subsystem.session;
            if (session == null)
                return;
            var recordingStatus = subsystem.recordingStatus;

            if (!recordingStatus.Recording())
            {
                using (var config = new ArRecordingConfig(session))
                {
                    config.SetMp4DatasetFilePath(session, mVideoFilePath);
                    config.SetRecordingRotation(session, GetRotation());
                    var status = subsystem.StartRecording(config);
                    Debug.Log($" 开始录制会话 {config.GetMp4DatasetFilePath
(session)} => {status}");
                }
            }
        }
    }
    // 结束会话录制
    public void StopRecord()
    {
        if (mSession.subsystem is ARCoreSessionSubsystem subsystem)
        {
            var session = subsystem.session;
            if (session == null)
                return;
            var recordingStatus = subsystem.recordingStatus;
            if (recordingStatus.Recording())
            {
                var status = subsystem.StopRecording();
                if (status == ArStatus.Success)
                {
                    Debug.Log(File.Exists(mVideoFilePath)
? $" 会话文件已保存到 {mVideoFilePath} ({GetFileSize(mVideo
FilePath)})"
: " 会话结束, 但无会话文件 ");
                }
            }
        }
    }
}

```

```

// 开始回放会话
public void StartPlayback()
{
    if (mSession.subsystem is ARCoreSessionSubsystem subsystem)
    {
        var session = subsystem.session;
        if (session == null)
            return;
        var playbackStatus = subsystem.playbackStatus;

        if (!playbackStatus.Playing() && File.Exists(mVideoFilePath))
        {
            var status = subsystem.StartPlayback(mVideoFilePath);
            DeBug.Log($" 开始回放会话 ({mVideoFilePath}) => {status}");
        }
    }
}

// 结束会话回放
public void StopPlayback()
{
    if (mSession.subsystem is ARCoreSessionSubsystem subsystem)
    {
        var session = subsystem.session;
        if (session == null)
            return;
        var playbackStatus = subsystem.playbackStatus;
        if (playbackStatus.Playing() || playbackStatus == ArPlaybackStatus.
Finished)
        {
            var status = subsystem.StopPlayback();
            if (status == ArStatus.Success)
                DeBug.Log($" 结束会话 => {status}");
        }
    }
}

#region Helper
// 根据手机旋转对录制的会话视频进行反向旋转
private static int GetRotation() => Screen.orientation switch
{
    ScreenOrientation.Portrait => 0,
    ScreenOrientation.LandscapeLeft => 90,
    ScreenOrientation.PortraitUpsideDown => 180,
    ScreenOrientation.LandscapeRight => 270,
    _ => 0
};

// 获取文件大小
private static string GetFileSize(string path)

```

```

{
    var ByteCount = new FileInfo(path).Length;
    const long kiloBytes = 1024;
    const long megaBytes = kiloBytes * kiloBytes;

    if (ByteCount / megaBytes > 0)
    {
        var size = (float)ByteCount / megaBytes;
        return $"{size:F2} MB";
    }

    if (ByteCount / kiloBytes > 0)
    {
        var size = (float)ByteCount / kiloBytes;
        return $"{size:F2} KB";
    }

    return $"{ByteCount} Bytes";
}
#endregion
}

```

在使用时, 将该脚本挂载到场景中的 AR Session 对象上, 使用按钮事件触发开始录制、结束录制、开始回放、结束回放 4 个相应的方法即可。

在真机设备运行时, 录制会话后会采集设备图像、IMU 等各传感器数据, 并将这些数据保存为 MPEG-4 格式视频文件, 在回放会话时, 应用会重放录制的视频 (IMU 等传感器数据也一并进行重放), 在回放会话时, ARCore 运动跟踪数据来源于录制的会话, 与真机设备实时数据无关, 真机设备可以保持不动, 但回放的会话就跟使用真机扫描环境一样, 同样也可以与场景进行交互, 实现如平面检测、虚拟物体放置等功能和操作, 与使用真机设备进行测试效果无异。

3.7.2 ARCore 扩展录制与回放

使用 AR Foundation 内置功能进行录制和回放, 使用简单, 但无法进行更多的操作, 如添加自定义数据、在其他平台回放会话等。使用 ARCore 扩展包也可以进行会话录制与回放, 并且提供了更多的操作灵活性, 使用 ARCore 扩展包进行会话录制与回放, 首先需要在场景中正确配置该扩展包 (详见 3.5 节), 然后利用该扩展包提供的 AR Recording Manager 和 AR Playback Manager 组件进行功能处理, 进行会话录制与回放的典型代码如下:

```

// 第 3 章 / 3-6
// 会话录制
// 通过 ARCoreRecordingConfig 可编程对象配置会话录制相关属性
ARCoreRecordingConfig recordingConfig = ScriptableObject.CreateInstance<ARCoreRecordingConfig>();

```



```
recordingConfig.Mp4DatasetUri = /* 存储会话录制后文件的路径 */;  
// 开始录制  
recordingManager.StartRecording(recordingConfig);  
// 结束录制  
recordingManager.StopRecording();  
  
// 会话回放  
session.enabled = false;  
playbackManager.SetPlaybackDatasetUri(uri);    // 设置会话文件路径  
// 开始回放  
session.enabled = true;  
  
// 结束会话回放实际上是将原会话文件路径置空  
session.enabled = false;  
playbackManager.SetPlaybackDatasetUri(null);  
session.enabled = true;
```

通过上述代码可以看到，会话回放实际上暂停了 AR 会话，然后使用录制好的会话进行 AR 会话重启。

ARCore 扩展包会话录制与回放的一大优势是在录制视频和传感器数据之外还可以同时录制自定义数据，这些录制的数据可以在回放的时候被提取出来，并且在相同的时刻被回放，如通过 `RecordTrackData()` 方法在 00:05:37 时刻录制了自定义数据，在会话回放时，该数据也会在 00:05:37 时刻被回放。

ARCore 不会自动添加任何开发人员自定义的数据，但允许开发者在特定的帧添加数据并在回放到该帧时取回数据，通过这种方式，增强了会话录制与回放功能的实用性。如可以在室外录制会话时同时录制 GPS 信息，在会话回放时这些 GPS 信息同样有效；或者可以通过录制会话，添加自定义信息并共享给其他人。

ARCore 扩展包在录制与回放开发人员自定义数据时借助了视频技术中轨道（track）的数据通道，一个轨道就是一个独立的数据通道，类似于图像处理软件中的图层，在视频播放时，多轨道数据通过时钟信号同步。录制自定义数据实际上就是添加一个独立的数据轨道，并将其设置到配置文件中，为了方便地提取到相关轨道，每个轨道都使用一个 GUID 值标识，典型代码如下：

```
// 第3章 / 3-7  
var track = new Track {  
    Id = Guid.Parse("de5ec7a4-09ec-4c48-b2c3-a98b66e71893")    // 生成的 GUID 值  
};  
  
List<Track> tracks = new List<Track>();  
tracks.Add(track);  
recordingConfig.Tracks = tracks;
```

元数据是用于描述轨道的数据，与图像帧和时间无关，通常用于描述轨道数据来源、位置、

用途等信息, 添加元数据的典型代码如下:

```
// 第 3 章 / 3-8
Byte[] metadata = System.Text.Encoding.Default.GetBytes("新的元数据");
track.Metadata = metadata;
```

默认的元数据类型为 `application/text`, 当然也可以是其他数据类型, 如逗号分隔值 (Comma Separated Values, CSV) 格式等, 用于存储更多的描述数据, 如果使用 CSV 格式, 则需要将 MIME 类型修改为 `text/csv`, 代码如下:

```
// 第 3 章 / 3-9
track.MimeType = "text/csv";
```

元数据与时间无关, 与时间相关的自定义数据在存储时与图像帧同步, 即这些数据与时间轴相关, 录制会话时添加自定义数据的时间与回放会话时提取该数据的时间保持一致, 添加自定义数据的典型代码如下:

```
// 第 3 章 / 3-10
public void onRecordCustomData()
{
    var customData = new Byte[] { 12, 34, 56, 78 };
    mRecordingManager.RecordTrackData(track.Id, customData);
}
```

因为自定义数据与时间轴、图像帧相关, 在提取这些数据时, 需要监视每帧会话中的数据轨道, 在轨道中有数据时将其提取出来, 所以提取自定义数据操作通常要放在 `update()` 这类每帧都会执行的方法中执行, 典型代码如下:

```
// 第 3 章 / 3-11
if (isPlayback)
{
    var trackDataList = mPlaybackManager.GetUpdatedTrackData(track.Id);
    foreach (TrackData trackData in trackDataList)
    {
        var data = trackData.Data;
        var customData = (int)(data[0]);
        Debug.Log($"元数据是 {track.Metadata.ToString()}; 自定义数据是: {customData}");
    }
}
```

一个完整的利用 ARCore 扩展包进行会话录制与回放的代码如下:

```
// 第 3 章 / 3-12
using System.IO;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using Google.XR.ARCoreExtensions;
```



```
using System.Collections;
using System.Collections.Generic;

public class SessionRecordAndPlayback : MonoBehaviour
{
    private ARSession mSession; // 会话组件
    private ARRecordingManager mRecordingManager; // ARRecordingManager 组件
    private ARPlaybackManager mPlaybackManager; // ARPlaybackManager 组件
    private string mVideoFilePath; // 会话文件存储路径
    private Track track; // 视频文件轨道, 用于存储自定义数据
    private bool isPlayback = false;
    void Start()
    {
        mSession = GetComponent<ARSession>();
        mRecordingManager = GetComponent<ARRecordingManager>();
        mPlaybackManager = GetComponent<ARPlaybackManager>();
        mVideoFilePath = Path.Combine(Application.persistentDataPath,
"ARCoreSession2.mp4");
        track = new Track
        {
            Id = System.Guid.NewGuid() // 该 GUID 应当保存起来
        };
    }
    private void Update()
    {
        if(isPlayback)
        {
            var trackDataList = mPlaybackManager.GetUpdatedTrackData(track.Id);
            foreach (TrackData trackData in trackDataList)
            {
                var data = trackData.Data;
                var customData = (int)(data[0]);
                DeBug.Log($"元数据是 {track.Metadata.ToString()} : 自定义数据是:
{customData}");
            }
        }
    }
    // 开始录制会话
    public void StartRecord()
    {
        ARCoreRecordingConfig recordingConfig = ScriptableObject.CreateInstance<AR
CoreRecordingConfig>();
        recordingConfig.Mp4DatasetUri = new System.Uri(mVideoFilePath);
        // 添加元数据
        Byte[] metadata = System.Text.Encoding.Default.GetBytes("新的元数据");
        track.Metadata = metadata;
```



```

        List<Track> tracks = new List<Track>();
        tracks.Add(track);
        recordingConfig.Tracks = tracks;

        mRecordingManager.StartRecording(recordingConfig);
        DeBug.Log(" 开始录制会话 ");
    }
    // 添加自定义数据
    public void onRecordCustomData()
    {
        var customData = new Byte[] { 12,34,56,78 };
        mRecordingManager.RecordTrackData(track.Id, customData);
    }
    // 结束录制会话
    public void StopRecord()
    {
        if (mRecordingManager.RecordingStatus == RecordingStatus.OK)
            mRecordingManager.StopRecording();
        DeBug.Log(File.Exists(mVideoFilePath)
            ? $" 会话文件已保存到 {mVideoFilePath} ({GetFileSize(mVideo
FilePath)})"
            : " 会话结束，但无会话文件 ");
    }
    // 开始回放会话
    public void StartPlayback()
    {
        DeBug.Log(" 开始回放会话 ");
        if (mPlaybackManager.PlaybackStatus == PlaybackStatus.None && File.
Exists(mVideoFilePath))
        {
            StartCoroutine(StartSessionPlayback());
        }
    }
    // 结束会话回放
    public void StopPlayback()
    {
        isPlayback = false;
        if (mPlaybackManager.PlaybackStatus == PlaybackStatus.OK ||
mPlaybackManager.PlaybackStatus == PlaybackStatus.FinishedSuccess)
        {
            StartCoroutine(StopSessionPlayback());
        }
    }

    #region Helper
    // 使用协程辅助延时

```

```

private IEnumerator StartSessionPlayback()
{
    mSession.enabled = false;
    while (true)
    {
        var result = mPlaybackManager.SetPlaybackDatasetUri(new System.
Uri(mVideoFilePath));
        if (result == PlaybackResult.ErrorPlaybackFailed || result ==
PlaybackResult.SessionNotReady)
        {
            yield return new WaitForSeconds(0.5f); // 等待 500ms
        }
        else
        {
            break;
        }
    }
    mSession.enabled = true;
    isPlayback = true;
    DeBug.Log("开始回放会话");
}
// 使用协程辅助延时
private IEnumerator StopSessionPlayback()
{
    mSession.enabled = false;
    yield return new WaitForSeconds(0.5f); // 等待 500ms
    while (true)
    {
        var result = mPlaybackManager.SetPlaybackDatasetUri((System.Uri)
null);
        if (result == PlaybackResult.ErrorPlaybackFailed || result ==
PlaybackResult.SessionNotReady)
        {
            yield return new WaitForSeconds(0.5f); // 等待 500ms
        }
        else
        {
            break;
        }
    }
    mSession.enabled = true;
    DeBug.Log("结束回放会话");
}
// 获取文件大小
private static string GetFileSize(string path)
{
    var ByteCount = new FileInfo(path).Length;
}

```

```

const long kiloBytes = 1024;
const long megaBytes = kiloBytes * kiloBytes;

if (ByteCount / megaBytes > 0)
{
    var size = (float)ByteCount / megaBytes;
    return $"{size:F2} MB";
}

if (ByteCount / kiloBytes > 0)
{
    var size = (float)ByteCount / kiloBytes;
    return $"{size:F2} KB";
}

return $"{ByteCount} Bytes";
}

#endregion
}

```

使用时, 将该脚本挂载到场景中的 AR Session 对象上, 并且将 AR Recording Manager 和 AR Playback Manager 组件也挂载到该对象上, 使用按钮事件触发开始录制、结束录制、添加自定义数据、开始回放、结束回放 5 个相应的方法。可以看到会话的录制与回放, 也可以正确录制和提取自定义数据^①。

使用会话录制与回放功能可以更方便地调试 AR 应用, 例如, 可以录制保存几个不同的场景会话, 利用这些场景会话, 就可以调试 AR 应用在不同场景中的表现而不用亲自到实际场景中去测试。

3.8 即时放置

通常, 用户在空间中放置 AR 虚拟对象时, 需要先扫描周围环境, 进行平面检测, 然后通过手势放置虚拟对象, 但这个过程并不非常直观, 有时也会让用户感到困惑, 通过即时放置 (Instant Placement) 功能允许用户无须移动设备, 在 ARCore 建立运动跟踪之前放置虚拟物体, 使用户更早地进入 AR 虚实融合氛围。在用户放置虚拟对象后, 随着环境探索的进行, 一旦 ARCore 建立运动跟踪, 在能够确定 AR 虚拟对象正确姿态时, 它就会更新虚拟对象的姿态和跟踪方法, 从而自动完成过渡, 如图 3-16 所示。

图 3-16 显示在 ARCore 中未检测到平面时, 可以在现实世界桌子上放置一个姿态预估的虚拟物体模型, 当 ARCore 正确检测到平面后, 就可以确定更精确的虚拟模型姿态, 并使用新的精确姿态进行对象跟踪。使用即时放置功能允许通过预先显示一个示意对象, 减少用户焦虑,

^① 为演示简单, 代码使用了简单的方式, 因此需要在同一个会话内完成会话录制与回放操作, 实际使用时应当存储数据轨道 GUID 值。

提升 AR 使用体验。

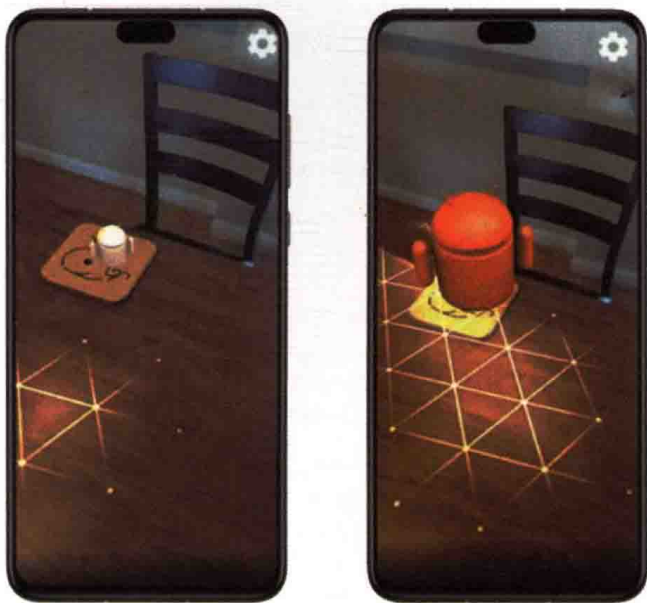


图 3-16 即时放置效果示意图

在 AR Foundation 中使用 ARCore 即时放置功能需要使用持续射线检测 (Persistent Raycast)，持续射线是指该射线每一帧都会进行射线检测^①，直到该射线被移除。在具体代码操作时，只需将持续射线添加到 AR Raycast Manager 组件便可以启用该持续射线并在每一帧进行更新，因此，实现即时放置功能的基本思路是设置好持续射线，预置一个放置位置，AR Raycast Manager 组件会持续地跟踪该射线的检测情况，在 ARCore 运动跟踪建立之前，使用预置的位置放置虚拟对象，而当检测到平面或者其他可跟踪对象时，该组件会自动进行姿态纠正，使用正确的姿态更新虚拟对象，代码如下：

```
// 第3章 /3-13
using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;

[RequireComponent(typeof(ARRaycastManager))]
public class InstancePlacement : MonoBehaviour
{
    private ARRaycastManager mRaycastManager;           // 射线检测组件
    private readonly float estimateDistance = 1.2f;     // 射线长度
    private bool placementPoseIsValid = false;          // 是否可放置指示标志
    void Awake()
    {
```

^① 射线检测更详细阐述可参见第4章。

```

        mRaycastManager = GetComponent<ARRaycastManager>();
    }
    void Update()
    {
        Touch touch;
        // 无操作时返回
        if (Input.touchCount < 1 || (touch = Input.GetTouch(0)).phase !=
        TouchPhase.Began)
        {
            return;
        }
        // 单击在 UI 上时返回
        if (EventSystem.current.IsPointerOverGameObject(touch.fingerId))
        {
            return;
        }
        // 已设置过示例对象时返回
        if (placementPoseIsValid)
            return;
        // 添加持续射线
        var raycast = mRaycastManager.AddRaycast(touch.position,
        estimateDistance);
        if (raycast != null)
        {
            placementPoseIsValid = true;
        }
    }
}

```

使用时，将 InstancePlacement 脚本挂载到场景中的 AR Session Origin 对象上，并同时在该对象上挂载 AR Plane Manager 组件和 AR Raycast Manager 组件，并为 AR Raycast Manager 组件的 Raycast Prefab 属性赋上需要放置的虚拟物体预制体。在 AR 应用启动后检测到平面之前，可以通过单击手势在场景中放置虚拟物体对象，而当应用检测到平面可放置点后，则会自动使用新的姿态更新虚拟物体。

通过以上方法可以实现即时放置功能，使用简单，但这种方式用两个问题：一是无法控制虚拟对象，如实现（如图 3-16 所示）预先展示示意物体，在检测到平面后放置真正的虚拟物体，也无法获取实例化后的虚拟对象；二是使用持续射线后，每一帧都进行射线检测，并且每一帧都会进行预制体的实例化操作，性能消耗极大，因此，改进思路是不通过 AR Raycast Manager 组件实例化虚拟物体，而是通过持续射线的 updated 事件进行示意模型与真正模型的转换操作，代码如下：

```

// 第 3 章 / 3-14
using UnityEngine;
using UnityEngine.EventSystems;

```

```
using System.Collections.Generic;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;

[RequireComponent(typeof(ARRaycastManager))]
public class InstancePlacement : MonoBehaviour
{
    public GameObject mPreObjectPrefab; // 在真实平面检测到之前用于示意的虚拟物体预制体
    public GameObject mObjectPrefab; // 真正需要放置的虚拟物体预制体

    private ARRaycastManager mRaycastManager; // 组件对象
    private readonly float estimateDistance = 1.2f; // 预先指定的放置示例物体距离
    private GameObject prePlacementObject; // 实例化后的指示物体
    private bool placementPoseIsValid = false; // 是否可放置指示器
    private ARRaycast mRaycast = null; // 构建的永久射线
    private Camera mCamera; // 场景渲染相机
    private static List<ARRaycastHit> Hits; // 碰撞结果

    void Awake()
    {
        mRaycastManager = GetComponent<ARRaycastManager>();
        Hits = new List<ARRaycastHit>();
        mCamera = Camera.main;
    }

    void Update()
    {
        Touch touch;
        // 无操作时返回
        if (Input.touchCount < 1 || (touch = Input.GetTouch(0)).phase !=
        TouchPhase.Began)
        {
            return;
        }
        // 单击在 UI 上时返回
        if (EventSystem.current.IsPointerOverGameObject(touch.fingerId))
        {
            return;
        }
        // 已设置过示例对象时返回
        if (placementPoseIsValid)
            return;
        // 添加持续射线, 实例化示例对象
        mRaycast = mRaycastManager.AddRaycast(touch.position, estimateDistance);
        if (mRaycast != null)
        {
            placementPoseIsValid = true;
            prePlacementObject = Instantiate(mPreObjectPrefab, mRaycast.
            transform.position, Quaternion.identity);
        }
    }
}
```



```

        mRaycast.updated += onRaycastUpdated;
    }
}
// 持续射线对象事件，每一帧都会触发
private void onRaycastUpdated(ARRaycastUpdatedEventArgs obj)
{
    Debug.Log("跟踪状态:" + mRaycast.trackingState);
    if (mRaycast.trackingState == TrackingState.Tracking)
    {
        Ray ray = new Ray(mCamera.transform.position, prePlacementObject.
transform.position);
        if (mRaycastManager.Raycast(ray, Hits, TrackableType.
PlaneWithinPolygon | TrackableType.PlaneWithinBounds))
        {
            var hitPose = Hits[0].pose;
            Instantiate(mObjectPrefab, hitPose.position, hitPose.rotation);
            mRaycastManager.RemoveRaycast(mRaycast);
            Destroy(prePlacementObject);
        }
    }
}
}
}

```

在上述代码中，使用了两个预制体，一个用于在未能找到合适放置点之前预先展示，另一个是真正需要放置的虚拟物体。通过持续射线 `updated` 事件再进行一次射线检测，当检测到平面后，实例化真正的虚拟物体，同时移除了持续射线，销毁示例物体^①，既提高了性能，也优化了视觉表现。

① 代码使用了射线与平面的碰撞检测，因此在该代码挂载的同一个场景对象上需要挂载 AR Plane Manager 组件。

功能技术篇



本篇对 ARCore 功能特性各方面进行了详细探索讨论，从平面检测、图像检测、人脸检测到云锚点、光影特效等，全方位地进行了技术剖析、讲解、演示，并对 AR 场景管理、图像信息处理、三维文字音频视频等实用功能进行了阐述和使用操作，提高读者实际运用能力。

功能技术篇包括以下章节。

第 4 章 平面检测与锚点管理

详细讲述了 ARCore 平面检测管理、可视化、个性化渲染、射线检测、特性点与点云相关功能技术点，并对锚点工作原理、操作使用、注意事项进行了阐述，带领读者熟悉可跟踪对象管理器的基本使用和操作方法。

第 5 章 图像检测跟踪

阐述了 ARCore 二维图像检测、识别、跟踪操作使用方法，并从实际应用出发，对静态、运行时添加 / 切换参考图像、参考图像库进行了详细使用演示，并介绍了图像跟踪优化的一般方法。

第 6 章 人脸检测跟踪

人脸特效在社交应用软件中使用广泛，本章详细阐述了人脸检测跟踪相关知识，对人脸姿态检测、人脸网格、人脸区域、多人脸检测进行了操作使用演示。

第 7 章 光影效果

光影是影响 AR 虚实融合的极其重要的因素，也是营造虚实融合沉浸体验的重要因素，本章讲述了在 AR 应用中实现光照估计、环境反射的基本方法，并详细阐述了在 AR 应用中实现阴影效果各类技术和技巧。

第 8 章 持久化存储与多人共享

持久化和多人体验共享是 AR 应用中的难点，本章通过 ARCore 提供的云锚点技术实现了数据持久化和多人共享功能，也演示了通过微软云实现类似功能的方法。

第 9 章 场景图像获取与场景深度

ARCore 需要捕获设备摄像头图像数据进行运动跟踪和 AR 场景背景渲染，本章阐述了捕获设备摄像头图像数据及 AR 场景图像数据的一般方法，通过一个实例演示图像数据处理的一般流程，并阐述了场景深度数据获取技术。

第 10 章 相机与手势操作

用户与 AR 场景或者虚拟对象交互是 AR 应用的重要功能组成部分，本章阐述了 AR 场景整体操作与场景中虚拟对象操作的方法和技巧。

第 11 章 三维文字与音视频

AR 场景中不仅有虚拟模型，也会有三维文字，AR 定位不仅包括视觉定位，也包括声源的三维定位，本章详细阐述了相关技术和应用，并阐述了在 AR 场景中播放视频的一般方法。

第 12 章 glTF 与 Model Viewer

本章阐述了在 Web 端使用 ARCore 能力的方法，通过 Web 端 AR，用户无须安装 App，能极大地方便应用的传播和推广，并简要介绍了 Web 端 AR 应用所使用的 glTF 模型格式文件相关知识。

第 4 章



平面检测与锚点管理

平面检测是很多 AR 应用的基础，无论是 ARCore、ARKit 还是 AREngine，都提供了平面检测功能。在平面检测算法中，ARCore 通过摄像头采集图像中的特征点构建三维空间点云，并依据点云分布情况进行平面划分和构建，将符合一定规律的三维空间点划归为平面。AR Foundation 也提供了锚点，用于更准确地锚定虚拟对象与现实环境的关系，实现更真实的虚实融合效果。本章主要学习有关平面检测、三维点云、锚点的知识。

4.1 平面检测引导

平面检测是很多 AR 应用的起点，而在技术上，平面检测需要场景视差信息，即要求使用者移动设备，这与当前用户的移动设备操作方式有很大不同，特别是对 AR 技术不熟悉的使用者，往往不知道该如何操作（这时场景中通常没有虚拟对象）。因此，适当地引导将有助于用户理解并执行期望的操作。

通常会以动画、文字、图片单独或者结合的方式对用户操作进行引导，相对而言，动画的形式更直观高效，本节使用动画的形式对平面检测、虚拟对象放置进行操作引导演示。

首先选择两个背景透明的视频文件，导入 Unity 中，在视频导入属性面板中勾选 Override for Android、Transcode、Keep Alpha 复选框，单击 Apply 按钮完成设置^①，如图 4-1 所示。

因为希望在屏幕空间中播放引导视频，我们的思路是通过 Video Player 组件播放视频^②，然后将视频帧渲染到 Raw Image 组件上，这中间需要借助一个渲染纹理进行视频帧的传递。

按照以上思路，首先在层级窗口（Hierarchy 窗口）中创建一个 Raw Image UI 对象，命名为 UX，设置好该对象的尺寸；在工程窗口（Project 窗口）中创建一个 Render Texture 纹理，命名为 UXRenderTexture，确保该纹理尺寸与 UX 对象尺寸一致；选择层级窗口中的 UX 对象，为其挂载 Video Player 组件，将其 Render Mode 属性设置为 Render Texture 值，然后将

① 如果 Android 选项卡中各属性均为灰色不可修改状态，则可先切换到 Default 选项卡，勾选 Transcode 复选框，应用后再切换到 Android 选项卡进行设置。

② Video Player 播放组件的更多详细内容可参见第 11 章。

UXRenderTexture 纹理赋给 Target Texture 属性、取消 Play On Awake 复选框、勾选 Loop 复选框；最后将 UXRenderTexture 纹理赋给 Raw Image 组件的 Texture 属性，即通过 UXRenderTexture 纹理接收并显示视频帧。Raw Image 组件与 Video Player 组件属性的设置如图 4-2 所示。

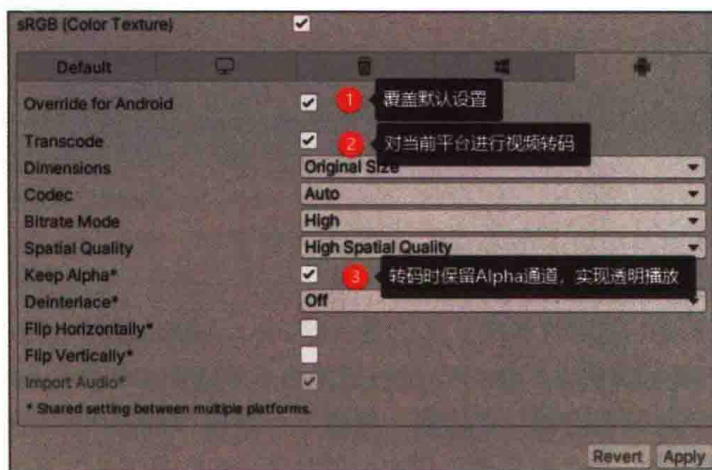


图 4-1 设置引导视频导入属性

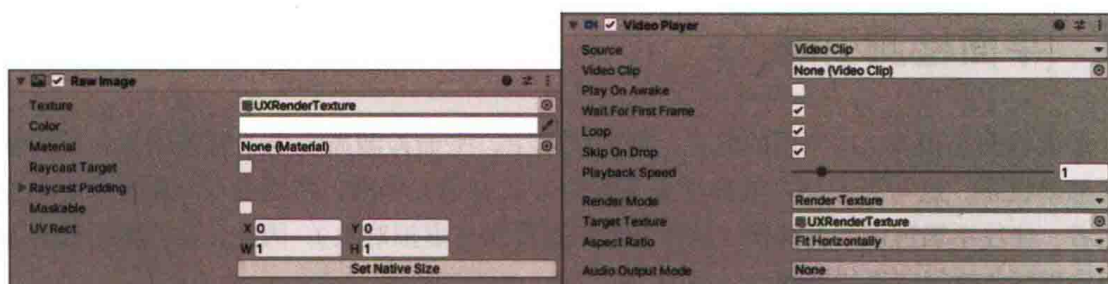


图 4-2 Raw Image 组件与 Video Player 组件属性设置

在工程窗口 Scripts 文件夹下，新建一个 C# 脚本，并命名为 UIManager，编写代码如下：

```
// 第 4 章 / 4-1
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Video;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;

[RequireComponent(typeof(ARPlaneManager))]
[RequireComponent(typeof(ARRaycastManager))]
public class UIManager : MonoBehaviour
{
    public VideoPlayer mVideoPlayer;           // 视频播放器
    public VideoClip mFindPlaneClip;           // 查找平面引导视频
    public VideoClip mPlaceObjectClip;         // 放置对象引导视频
}
```

```

public GameObject spawnPrefab;           // 需要放置的游戏对象预制体
public GameObject uiImage;               // 播放视频的 UI 对象

private bool isPlaneFound = false;       // 是否找到平面
private bool isPlaced = false;          // 是否放置对象
private static List<ARRaycastHit> mHits; // 碰撞检测结果

private ARPlaneManager mARPlaneManager;
private ARRaycastManager mARRaycastManager;

void Start()
{
    mHits = new List<ARRaycastHit>();
    mARPlaneManager = GetComponent<ARPlaneManager>();
    mARRaycastManager = GetComponent<ARRaycastManager>();
}

void Update()
{
    // 没有找到平面时, 播放查找平面引导视频
    if(!isPlaneFound)
    {
        if (mARPlaneManager.trackables.count > 0)
            isPlaneFound = true;
        if (!mVideoPlayer.isPlaying)
        {
            mVideoPlayer.clip = mFindPlaneClip;
            mVideoPlayer.Play();
        }
    }
    // 找到平面后, 并且没有放置虚拟对象时, 播放放置虚拟对象引导视频
    else if(!isPlaced)
    {
        if (mVideoPlayer.clip.name.ToLower().Equals(mFindPlaneClip.name.
ToLower()))
        {
            mVideoPlayer.clip = mPlaceObjectClip;
            mVideoPlayer.Play();
        }
    }
    // 找到平面, 放置虚拟对象后, 停止视频播放, 释放渲染纹理
    else
    {
        mVideoPlayer.Stop();
        var renderTexture = mVideoPlayer.targetTexture;
        renderTexture.DiscardContents();
        renderTexture.Release();
    }
}

```



```

        uiImage.SetActive(false);
    }
    // 放置虚拟对象逻辑
    if (Input.touchCount == 0 || isPlaced)
        return;
    var touch = Input.GetTouch(0);
    if (mARRaycastManager.Raycast(touch.position, mHits, TrackableType.
        PlaneWithinPolygon | TrackableType.PlaneWithinBounds))
    {
        var hitPose = mHits[0].pose;
        Instantiate(spawnPrefab, hitPose.position, hitPose.rotation);
        isPlaced = true;
    }
}
}

```

上述代码在检测到平面之前播放平面检测引导视频，在检测到平面后播放放置虚拟对象引导视频，并在虚拟对象放置后释放相应资源，关闭用于显示的 UI 对象。

在层级窗口中选择 AR Session Origin 对象，将该脚本挂载于此对象上，并设置好相关属性，运行效果如图 4-3 所示。

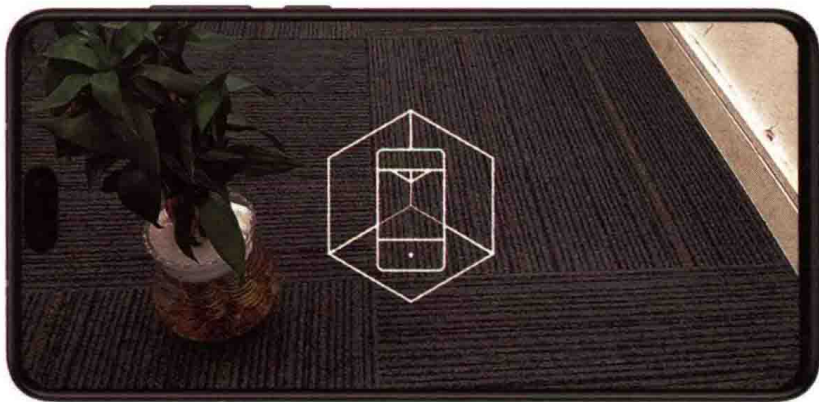


图 4-3 操作引导动画效果示意图

4.2 平面管理

技术层面而言，平面也属于可跟踪对象，在 AR Foundation 中使用 AR Plane Manager 组件管理平面检测、平面更新、平面移除等，该组件也是进行平面操作的常用组件。

4.2.1 平面检测

在 AR 中进行平面检测并不是 SLAM 运动跟踪工作的一部分，平面检测建立在 SLAM 运

动跟踪能力之上，检测平面的基本原理：

ARCore 对设备摄像头采集的图像数据进行分析处理，提取图像中的特征点（这些特征点往往都是图像中明暗、强弱、颜色变化较大的点），利用 VIO 和 IMU 跟踪这些特征点的三维空间信息，生成三维空间点云，再通过平面拟合算法将空间中位置相近或者符合一定规律的三维空间点构建成平面。由于 IMU 可以提供重力方向，因此可以将平面区分为水平或者垂直类型^①。

ARCore 检测到平面后，会提供平面的位置、方向和边界信息，AR Plane Manager 组件负责检测平面及管理检测到的平面，但它并不负责渲染平面。

在 AR Plane Manager 组件中，可以设置平面检测方式(Detection Mode)：水平平面(Horizontal)、垂直平面(Vertical)、水平平面 & 垂直平面(Everything)、不检测平面(Nothing)，如图 4-4 所示。

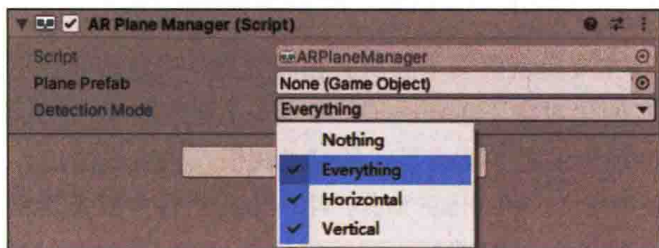


图 4-4 设置平面检测方式

当设置成检测平面后（水平、垂直、水平 & 垂直），AR Plane Manager 组件每帧都会进行平面检测计算，添加新检测到的平面、更新现有平面、移除过时平面^②。当一个新的平面被检测到时，AR Plane Manager 组件会实例化一个平面预制体（该预制体由 Plane Prefab 属性指定）表示该平面，如果此时 AR Plane Manager 组件的 Plane Prefab 属性没有赋值，AR Plane Manager 组件则会实例化一个空对象，并在这个空对象上挂载 AR Plane 组件，AR Plane 组件包含了有关该平面的所有相关数据信息。

AR Plane Manager 组件也提供了 planesChanged 事件，开发人员可以通过订阅该事件，在平面检测状态发生变化时进行自定义逻辑处理。

平面检测算法是一个计算密集型算法，而且每帧都会进行，会消耗较大性能，建议根据实际需要选择合适的检测方式、在不需要时关闭平面检测功能以优化应用性能。

4.2.2 可视化平面

在 AR Foundation 中，AR Plane Manager 组件并不负责平面的可视化渲染，而由其 Plane

① 平面检测建立在 SLAM 运动跟踪能力之上，SLAM 本身也会进行特征点的三角化工作，平面检测可以直接利用这些三维空间点云信息，不需要再独立进行处理。

② ARCore 检测到的平面与日常理解的平面有些许差异，如 ARCore 对地板进行平面检测时，可能会生成多个平面，这一个平面簇都是对地板的一个表达，随着环境探索的进行，一些平面会合并，但并不能保证只生成一个平面用于对应一个真实环境表面。

Prefab 属性指定的预制体负责。在第 1 章案例中,我们新建了一个 AR Default Plane 对象作为预制体,该预制体上挂载了如图 4-5 所示组件。

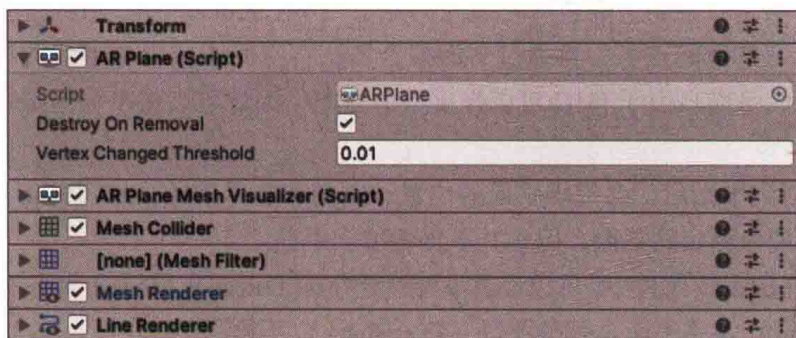


图 4-5 AR Default Plane 对象挂载的组件

其中 AR Plane 组件主要负责管理平面所有相关数据,也负责管理平面对象实例,如是否在移除平面时销毁该实例化对象 (Destroy On Removal 复选框),控制属性更新阈值 (Vertex Changed Threshold 属性用于设置平面边界点的变动阈值,只有平面边界点变动幅度超过设定值时才会触发平面更新事件); AR Plane Mesh Visualizer 组件主要负责根据平面边界点生成平面网格数据 (顶点、顶点索引、UV 坐标、法线向量),在生成平面网格后通过 Mesh Renderer 组件采用合适材质渲染平面; Mesh Collier 组件用于生成网格碰撞器,使检测到的平面可以与其他对象进行碰撞检测;默认平面预制体还有一个 Line Renderer 组件,它负责渲染平面可视化后的边界点连线。

所以使用默认平面预制体可视化 (渲染) 已检测到的平面效果如图 4-6 所示。



图 4-6 AR Default Plane 可视化已检测到的平面效果图

4.2.3 个性化渲染平面

平面默认的渲染效果显得有些生硬和突兀,通常我们需要更加友好的界面,这时就需要

对已检测到的平面定制个性化的可视方案，为达到更好的视觉效果，处理的思路如下：

(1) 不显示黑色边框。

(2) 重新制作渲染材质和 Shader 渲染着色器，材质纹理使用带 Alpha 通道半透明的 PNG 格式纹理，通过透明混合 Shader 渲染这个半透明纹理，将纹理透明区域镂空。

(3) 为达到更好的视觉过渡，编写一个渐隐的脚本，让平面边缘的纹理渐隐。

按照以上思路，可直接对 AR Default Plane 预制体进行改造。

(1) 删除 AR Default Plane 预制体上的 Line Renderer 组件。

(2) 制作一张带 Alpha 通道半透明 PNG 格式纹理，编写的 Shader 脚本如下：

```
// 第4章 /4-2
Shader "DavidWang/FeatheredPlaneShader"
{
    Properties
    {
        _MainTex("Texture", 2D) = "white" {}
        _TexTintColor("Texture Tint Color", Color) = (1,1,1,1)
        _PlaneColor("Plane Color", Color) = (1,1,1,1)
    }
    SubShader
    {
        Tags { "RenderType" = "Transparent" "Queue" = "Transparent" "RenderPipeline" =
"UniversalPipeline"}
        LOD 100
        Blend SrcAlpha OneMinusSrcAlpha
        ZWrite Off

        Pass
        {
            HLSLPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            #include
"Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
                float3 uv2 : TEXCOORD1;
            };

            struct v2f
            {
                float4 vertex : SV_POSITION;
```

```

float2 uv : TEXCOORD0;
float3 uv2 : TEXCOORD1;
};

TEXTURE2D(_MainTex);
SAMPLER(sampler_MainTex);

CBUFFER_START(UnityPerMaterial)
    float4 _MainTex_ST;
    float4 _TexTintColor;
    float4 _PlaneColor;
    float _ShortestUVMapping;
CBUFFER_END

v2f vert(appdata v)
{
    v2f o;
    o.vertex = TransformObjectToHClip(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    o.uv2 = v.uv2;
    return o;
}

float4 frag(v2f i) : SV_Target
{
    float4 col = SAMPLE_TEXTURE2D(_MainTex, sampler_MainTex, i.uv) * _
TexTintColor;
    col = lerp(_PlaneColor, col, col.a);
    col.a *= 1 - smoothstep(1, _ShortestUVMapping, i.uv2.x);
    return col;
}
ENDHLSL
}
}
}

```

新建一个材质文件，命名为 `FeaturePlane`，渲染 Shader 选择自定义的 Shader 脚本，Texture 属性即为纹理属性，将预先制作好的纹理赋给该属性。TextureTintColor 为纹理显示颜色，为了将半透明的十字星号显示出来，将其 Alpha 值设置为 220；PlaneColor 为平面渲染背景色，用于渲染平面底色，这里不设置背景色，将其 Alpha 值设置为 0，如图 4-7 所示。

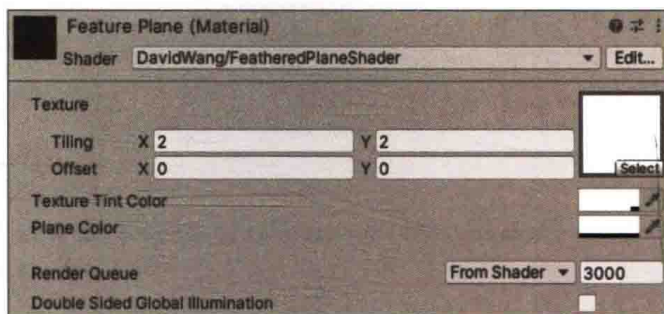


图 4-7 设置 Shader 参数

(3) 新建一个 C# 脚本文件, 命名为 ARFeatheredPlaneMeshVisualizer.cs, 编写代码如下:

```
// 第4章 / 4-3
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation;

[RequireComponent(typeof(ARPlaneMeshVisualizer), typeof(MeshRenderer),
typeof(ARPlane))]
public class ARFeatheredPlaneMeshVisualizer : MonoBehaviour
{
    [SerializeField]
    float mFeatheringWidth = 0.2f; // 平面边界渐隐区宽度

    static List<Vector3> mFeatheringUVs = new List<Vector3>(); // UV 坐标列表
    static List<Vector3> mVertices = new List<Vector3>(); // 顶点列表
    ARPlaneMeshVisualizer mPlaneMeshVisualizer; // 平面可视化器
    ARPlane mPlane; // 当前平面
    Material mFeatheredPlaneMaterial; // 平面使用的材质

    public float featheringWidth
    {
        get { return mFeatheringWidth; }
        set { mFeatheringWidth = value; }
    }

    void Awake()
    {
        mPlaneMeshVisualizer = GetComponent<ARPlaneMeshVisualizer>();
        mFeatheredPlaneMaterial = GetComponent<MeshRenderer>().material;
        mPlane = GetComponent<ARPlane>();
    }

    // 注册当前平面边界变化事件
    void OnEnable()
    {
        mPlane.boundaryChanged += ARPlane_boundaryUpdated;
    }
}
```



```

// 取消当前平面边界变化事件注册
void OnDisable()
{
    mPlane.boundaryChanged -= ARPlane_boundaryUpdated;
}
// 平面变化处理方法
void ARPlane_boundaryUpdated(ARPlaneBoundaryChangedEventArgs eventArgs)
{
    GenerateBoundaryUVs(mPlaneMeshVisualizer.mesh);
}

// 生成边界点的 UV
void GenerateBoundaryUVs(Mesh mesh)
{
    int vertexCount = mesh.vertexCount;

    mFeatheringUVs.Clear();
    if (mFeatheringUVs.Capacity < vertexCount) { mFeatheringUVs.Capacity =
vertexCount; }

    mesh.GetVertices(mVertices);

    Vector3 centerInPlaneSpace = mVertices[mVertices.Count - 1];
    Vector3 uv = new Vector3(0, 0, 0);
    float shortestUVMapping = float.MaxValue;

    for (int i = 0; i < vertexCount - 1; i++)
    {
        float vertexDist = Vector3.Distance(mVertices[i], centerInPlaneSpace);

        float uvMapping = vertexDist / Mathf.Max(vertexDist - featheringWidth,
0.001f);

        uv.x = uvMapping;

        if (shortestUVMapping > uvMapping) { shortestUVMapping = uvMapping; }

        mFeatheringUVs.Add(uv);
    }
    // 设置 Shader 中的参数
    mFeatheredPlaneMaterial.SetFloat("_ShortestUVMapping", shortestUVMapping);

    // 添加中心点 UV
    uv.Set(0, 0, 0);
    mFeatheringUVs.Add(uv);

    mesh.SetUVs(1, mFeatheringUVs);
    mesh.UploadMeshData(false);
}

```

```

}
}

```

该脚本主要实现 UV 坐标生成、边界变化事件处理、平面边界的平滑渐隐过渡等功能。将 ARFeatheredPlaneMeshVisualizer 挂载到 AR Default Plane 预制体上，完成之后的预制体组件脚本应该如图 4-8 所示。

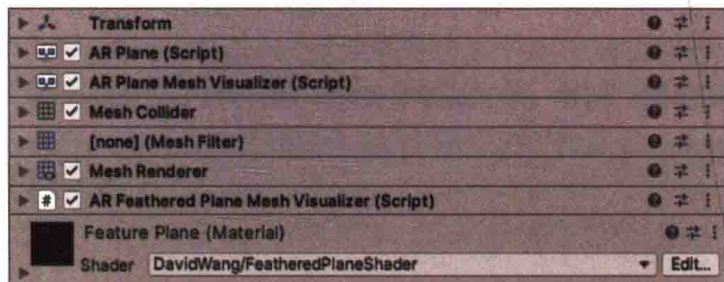


图 4-8 在 AR Default Plane 预制体上挂载自定义可视化脚本

编译运行，找一个富纹理表面进行平面检测，效果如图 4-9 所示，相比默认平面渲染效果，个性化后的渲染视觉效果要好很多，而且在平面边界处也有一个渐隐的平滑过渡。

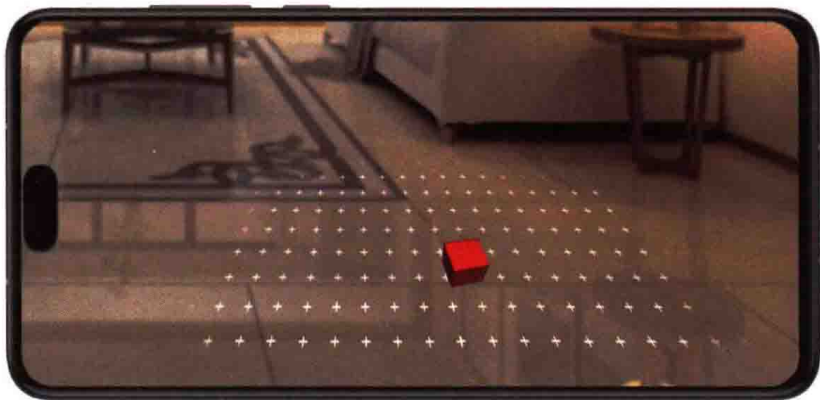


图 4-9 个性化渲染检测到平面效果图

提示

事件注册与取消注册一定是成双成对的，如代码清单 4-3 所示，在 Start() 方法中进行了注册，在 OnDisable() 方法中撤销了注册，如果事件注册没有在适当的时机撤销，则有可能会引发难以排查的错误。

4.2.4 开启与关闭平面检测功能

AR Foundation 使用 AR Plane Manager 组件管理平面检测、更新、移除任务，并为开发者提供了简单易用的属性和事件，使平面管理工作变得轻松简单。在实际应用开发中，经常

会遇到开启与关闭平面检测功能、显示与隐藏被检测到的平面等类似问题，利用 AR Plane Manager 组件可以方便地实现这些需求。

从前面可知，AR Plane Manager 组件负责管理平面检测相关工作，该组件有一个 enabled 属性，将其值设置为 true 即可开启平面检测，将其值设置为 false 则可关闭平面检测。因此，可以非常方便地使用代码控制平面检测功能的开启与关闭。在前文中也学习到，AR Plane Manager 组件并不负责对检测到的平面进行可视化渲染。因此，在关闭平面检测后还应该取消已检测到平面的显示。

为实现上述两项功能，新建一个 C# 脚本文件，命名为 PlaneDetection，编写代码如下：

```
// 第 4 章 / 4-4
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.UI;

[RequireComponent(typeof(ARPlaneManager))]
public class PlaneDetection : MonoBehaviour
{
    public Text txtPlaneToggle;           // 显示平面检测状态文字
    public Button BtnToggle;              // 控制平面检测开关按钮
    private ARPlaneManager mARPlaneManager;

    void Awake()
    {
        mARPlaneManager = GetComponent<ARPlaneManager>();
        BtnToggle.onClick.AddListener(TogglePlaneDetection);
    }

    // 开关平面检测功能及切换检测到的平面显隐
    public void TogglePlaneDetection()
    {
        mARPlaneManager.enabled = !mARPlaneManager.enabled;

        string planeDetectionMessage = "";
        if (mARPlaneManager.enabled)
        {
            planeDetectionMessage = "禁用平面检测 ";
            SetAllPlanesActive(true);
        }
        else
        {
            planeDetectionMessage = "启用平面检测 ";
            SetAllPlanesActive(false);
        }

        if (txtPlaneToggle != null)
            txtPlaneToggle.text = planeDetectionMessage;
    }
}
```



```
// 显示或者隐藏所有已检测到的平面对象
void SetAllPlanesActive(bool value)
{
    foreach (var plane in mARPlaneManager.trackables)
        plane.gameObject.SetActive(value);
}
}
```

在上述代码中, TogglePlaneDetection() 方法是切换平面检测功能的开关函数, 可使用按钮事件来控制该方法的调用, 每调用一次该方法则切换一次平面检测功能的开启与关闭; SetAllPlanesActive() 方法的主要目的是处理已检测到平面的显隐状态, 根据平面检测功能状态决定已检测到的平面是否显示^①。

在使用时, 只需将该脚本挂载在与 AR Plane Manager 组件相同的场景对象上, 以及设置相应的按钮及显示文字对象即可。

4.2.5 显示与隐藏已检测平面

在 4.2.4 节中, 已经实现了平面检测功能的开启与关闭, 并对已检测到的平面进行了处理, 4.2.4 节的代码可以实现预期目标。同时也可以看到, 即使关闭平面检测功能后, 已检测到的平面依然是存在的, 换言之, 开启与关闭平面检测功能并不会影响已检测到的平面。

在关闭平面检测功能后, AR Foundation 不再检测新的平面, 但原已检测到的平面依然存在, 这有时并不是所希望的, 例如在实际项目中, 希望隐藏已检测到的平面, 但还继续保留平面检测功能, 以便需要再次显示检测到的平面时能直接显示检测到的平面而不再去进行一遍平面检测工作。为实现这个功能, 新建一个 C# 脚本文件, 命名为 PlaneDisplay, 编写代码如下:

```
// 第4章 / 4-5
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.UI;

[RequireComponent(typeof(ARPlaneManager))]
public class PlaneDisplay : MonoBehaviour
{
    public Text txtPlaneToggle;           // 显示平面检测状态文字
    public Button BtnToggle;             // 控制平面检测开关按钮

    private ARPlaneManager mARPlaneManager;
    private bool isShow = true;          // 显示平面标志
```

^① 这里需要注意, 并没有使用 Destroy() 方法销毁已检测的平面, ARPlaneManager 组件负责平面检测功能管理, 不需要手动去销毁平面, 事实上, 手动销毁平面可能会引发错误。

```

private List<ARPlane> mPlanes;           // 所有已检测到的平面列表
void Start()
{
    mARPlaneManager = GetComponent<ARPlaneManager>();
    mPlanes = new List<ARPlane>();
    BtnToggle.onClick.AddListener(TogglePlaneDisplay);
    // 注册平面变化事件
    mARPlaneManager.planesChanged += OnPlaneChanged;
}
// 取消平面变化事件注册
void OnDisable() => mARPlaneManager.planesChanged -= OnPlaneChanged;

// 切换平面显示状态
public void TogglePlaneDisplay()
{
    isShow = !isShow;
    string planeDisplayMessage = "";
    if (isShow)
    {
        planeDisplayMessage = "隐藏平面 ";
    }
    else
    {
        planeDisplayMessage = "显示平面 ";
    }
    for (int i = mPlanes.Count - 1; i >= 0; i--)
    {
        if (mPlanes[i] == null || mPlanes[i].gameObject == null)
            mPlanes.Remove(mPlanes[i]);
        else
            mPlanes[i].gameObject.SetActive(isShow);
    }
    if (txtPlaneToggle != null)
        txtPlaneToggle.text = planeDisplayMessage;
}
// 平面变化事件处理方法
private void OnPlaneChanged(ARPlanesChangedEventArgs arg)
{
    for (int i = 0; i < arg.added.Count; i++)
    {
        mPlanes.Add(arg.added[i]);
        arg.added[i].gameObject.SetActive(isShow);
    }
}
}

```

在上述代码中，通过 OnPlaneChanged() 方法，把所有已检测到的平面保存到 mPlanes 列

表中,并处理了新检测到平面的显示状态;TogglePlaneDisplay()方法是切换平面显示状态的开关函数,可以使用按钮事件来控制该方法的调用,切换已检测到平面的显示状态。本示例演示了AR Plane Manager组件planesChanged事件的一般使用方法,也演示了存储检测到平面的一般处理方法。

需要注意的是,在代码中,把所有已检测到的平面都保存在mPlanes变量中,因此这个列表中保存的是所有历史已检测到的平面,并不会主动删除过期失效的平面(AR Plane Manager组件负责检测、更新、删除平面,但并不会影响已保存的平面),切换已失效平面的显示状态有可能会引发错误,读者可思考一下怎么删除已失效的平面^①。

与4.2.4节中脚本使用一样,在使用时,只需将该脚本挂载在与AR Plane Manager组件相同的场景对象上,设置相应的按钮及显示文字对象。

4.2.6 平面遮挡

当前主流移动终端(手机、平板),实现AR的方式是采用Pass-Through方案,即现实场景会以图像的形式作为AR应用的背景,由于缺乏现实场景的深度信息,无法实现虚实遮挡^②(虚拟对象会一直浮于现实场景之上),但同时,ARCore又能够检测区分出不同的平面,如桌面、地面,如果能够使用检测出来的平面进行虚实遮挡,则会进一步提高AR的真实感。

为实现以上目标,实现的思路是不渲染检测到的平面,但通过Shader脚本让检测到的平面参与深度测试,这样自然就可以实现虚实遮挡,因此,首先编写一个Shader脚本,代码如下:

```
// 第4章/4-6
Shader "DavidWang/PlaneOcclusion"
{
    SubShader
    {
        Tags { "RenderType" = "Opaque" "Queue" = "Geometry-1" "RenderPipeline" =
"UniversalPipeline" }
        ZWrite On
        ZTest LEqual
        ColorMask 0

        Pass
        {
            HLSLPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "Packages/com.unity.render-pipelines.universal/
ShaderLibrary/Core.hlsl"
```

① 利用planesChanged事件中的removed参数进行处理。

② 关于场景深度的更多探讨可参见第9章。


```

    struct appdata
    {
        float4 vertex : POSITION;
    };

    struct v2f
    {
        float4 vertex : SV_POSITION;
    };

    v2f vert (appdata v)
    {
        v2f o;
        o.vertex = TransformObjectToHClip(v.vertex);
        return o;
    }

    float4 frag (v2f i) : SV_Target
    {
        return float4(0.0, 0.0, 0.0, 0.0);
    }

    ENDHLSL
}
}
}

```

新建一个材质文件，命名为 `PlaneOcclusion`，选择该材质使用刚创建的 Shader，然后将 `PlaneOcclusion` 材质赋给平面预制体的 Mesh Renderer 材质属性即可。

编译运行，首先在地面上放置一个虚拟对象，然后移动到桌子后，可以看到，虚拟对象会被现实中的桌子遮挡，实现虚实遮挡效果。

4.3 射线检测

射线检测 (Hit Testing、Ray Casting) 是在三维空间中选择虚拟物体的最基本方法。在 AR Foundation 中，除了使用 Unity 引擎提供的 `Physics.Raycast()` 方法进行碰撞检测之外，还提供了 AR Raycast Manager 组件专门用于在 AR 中处理射线与平面、特征点的碰撞检测任务。

4.3.1 射线检测概念

在 AR 中，当检测并可视化一个平面后，如果需要在平面上放置虚拟物体，就会碰到一些问题，在平面上什么位置放置虚拟物体呢？要知道检测到的平面是三维的，而我们的手机屏幕却是二维的，如何在二维的平面上选择三维放置点呢？

解决这些问题的通常做法就是进行射线检测，Ray Casting 直译为射线投射，通常根据它的作用称为射线检测，射线检测是在三维数字世界里选择某个特定物体常用的一种技术，如在三维、VR 游戏中检查子弹命中敌人情况或者从地上捡起一支枪，都要用到射线检测。

射线检测的基本思想是在三维空间中从一个点沿一个方向发射出一条无限长的射线，在射线的方向上，一旦与添加了碰撞器的模型发生碰撞，就会产生一个碰撞检测对象。可以利用射线实现子弹击中目标的检测，也可以利用射线来检测发生碰撞的位置。

对于 AR 中放置点选择的问题，可以从屏幕上用户单击点，利用渲染相机的位置来构建一条射线，与场景中的平面进行碰撞检测，如果发生碰撞，则返回碰撞的位置，这样就可以在检测到的平面上放置虚拟物体了，射线检测原理如图 4-10 所示。

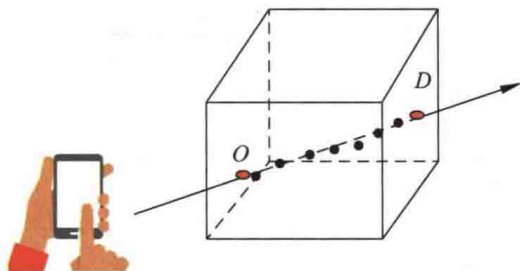


图 4-10 射线检测示意图

在图 4-10 中，对 AR 应用而言，用户操作的是其手机设备，射线检测的具体做法是检测用户单击屏幕的操作事件，以单击的位置为基准，连接该位置与渲染相机位置就可以两点构成一条直线。从渲染相机位置出发，通过单击点就可以构建一条射线^①，利用该射线与场景中的物体进行碰撞检测，如果发生碰撞，则返回碰撞对象，从而可以确定碰撞点位置坐标及旋转姿态信息。

其实在第 1 章的示例中，就已经使用了射线检测，相关代码如下：

```
// 第 4 章 / 4-7
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;

[RequireComponent(typeof(ARRaycastManager))]
public class AppController : MonoBehaviour
{
    public GameObject spawnPrefab;           // 需要放置的游戏对象预制体
    private static List<ARRaycastHit> Hits;   // 碰撞结果
    private ARRaycastManager mRaycastManager;
    private GameObject spawnedObject = null;  // 实例化后的游戏对象
}
```

^① 这里的描述并不严谨，实际上是从渲染相机所在位置出发，通过屏幕单击点在成像平面上的投影点构建射线，由于涉及相机成像的相关知识，读者只要认为屏幕单击点有对应的三维空间中的三维坐标即可。

```

private void Start()
{
    Hits = new List<ARRaycastHit>();
    mRaycastManager = GetComponent<ARRaycastManager>();
}

void Update()
{
    if (Input.touchCount == 0)
        return;
    var touch = Input.GetTouch(0);
    // 进行射线检测, 如果发生碰撞, 则进行虚拟对象操作
    if (mRaycastManager.Raycast(touch.position, Hits, TrackableType.
PlaneWithinPolygon | TrackableType.PlaneWithinBounds))
    {
        var hitPose = Hits[0].pose;
        // 如未放置游戏对象, 则实例化; 如果已放置游戏对象, 则将游戏对象移动到新的碰撞点位置
        if (spawnedObject == null)
        {
            spawnedObject = Instantiate(spawnPrefab, hitPose.position,
hitPose.rotation);
        }
        else
        {
            spawnedObject.transform.position = hitPose.position;
        }
    }
}
}

```

在上述代码中, 使用 `mRaycastManager.Raycast()` 方法做射线检测来确定放置虚拟物体的位置。相比 `Physics.Raycast()` 方法, AR Raycast Manager 组件专门用于在 AR 中处理射线与平面、特征点等的碰撞检测任务, 更适合在 AR 中对平面的射线检测。通常, 在 AR 中, 除了对放置位置无要求的自动放置 (如在检测到的平面上长草), 在精确放置虚拟物体时, 通常要做射线检测以便确定放置位置信息。

4.3.2 射线检测详细讲解

AR Raycast Manager 组件中的射线检测与 Unity 中 Physics 模块使用的射线检测相似, 但提供了独立的接口, 该组件只能用来检测与 AR 相关的平面、点云、二维图像、人脸 (只对应 AR 中的可跟踪对象), 不能用于检测场景中的三维物体。因为射线检测与被检测物体需要在同一个坐标空间中, 所以该组件需要挂载在场景中 AR Session Origin 对象上。

利用 AR Raycast Manager 组件进行射线检测可以使用以下两种方法之一, 其函数原型如表 4-1 所示。

表 4-1 AR Raycast Manager 组件射线检测方法

射线检测方法	描 述
<pre>public bool Raycast(Vector2 screenPoint, List<ARRaycastHit> hitResults, TrackableType trackableTypeMask = TrackableType.All)</pre>	参数 1 为屏幕坐标点；参数 2 为与射线发生碰撞的所有对象列表；参数 3 为 Trackable 类型掩码（只对一类或几类可跟踪对象进行射线检测）。该方法的返回值为 Bool 型，true 表示发生碰撞，false 表示未发生碰撞
<pre>public bool Raycast(Ray ray, List<ARRaycastHit> hitResults, TrackableType trackableTypeMask = TrackableType.All)</pre>	参数 1 为 Ray 类型的射线，包括位置与方向；参数 2 为与射线发生碰撞的所有对象列表；参数 3 为 Trackable 类型掩码，只对一类或几类可跟踪对象进行射线检测。该方法的返回值为 Bool 型，true 表示发生碰撞，false 表示未发生碰撞

trackableTypeMask 类型掩码用于过滤需要进行碰撞检测的对象类型，Trackable 类型的值可以是表 4-2 所示属性值中的一个，也可以是几个值的组合，如果是几种类型值组合，采用按位或方式进行组合，如 TrackableType.PlaneWithinPolygon | TrackableType.FeaturePoint。

表 4-2 trackableType 属性

trackableType 属性	描 述
All	这个值用于与所有可检测类型发生碰撞检测。如果选择该值，则发射的射线将与场景中的所有平面类型、多边形包围盒、带法线的特征点进行碰撞检测
FeaturePoint	与当前帧点云中所有的特征点进行碰撞检测
None	此值用于表示 trackableHit 返回值中没有碰撞对象，如果将此值传递给 Raycast() 方法，则不会得到任何碰撞结果
PlaneWithinPolygon	与已检测平面的凸边界多边形进行碰撞检测
PlaneWithinBounds	与当前帧中已检测平面的包围盒进行碰撞检测
PlaneWithinInfinity	与已检测到的平面进行碰撞检测，但这个检测不仅局限于包围盒或者多边形范围，也可以与已检测到平面的延展平面进行碰撞检测
FeaturePointWithSurfaceNormal	与带法线信息的特征点进行碰撞检测，这些特征点不一定能形成一个平面
Planes	射线与以上所有平面类型进行碰撞检测
Image	与二维图像进行碰撞检测
Face	与人脸进行碰撞检测

ARRaycastHit 类保存的是发生碰撞时检测到的碰撞体相关信息，其主要属性如表 4-3 所示。

表 4-3 ARRaycastHit 属性

ARRaycastHit 属性	描 述
Distance	float 类型，获取从射线源到命中点的距离
trackableId	发生碰撞的可跟踪对象 ID

续表

ARRaycastHit 属性	描 述
Pose	Pose 类型，获取射线检测击中的物体在 Unity 世界坐标空间中的姿态
hitType	Trackable 类型，获取命中的可跟踪对象类型，即表 4-2 所述的 8 种可跟踪类型之一
sessionRelativeDistance	在会话空间中从射线起点到碰撞点的距离
sessionRelativePose	在会话空间中碰撞点的姿态

在有了以上基础之后，我们就可以很容易地理解 4.3.1 节中射线检测代码的逻辑及含义，首先初始化一个 ARRaycastHit 类型的列表数组用于保存所有与射线检测发生碰撞的可跟踪对象，这个列表必须完成初始化，不可为 null 值，这也是为了避免垃圾回收机制（Garbage Collection, GC）反复分配内存，然后调用射线检测方法对指定可跟踪对象进行碰撞检测，如果发生碰撞，则使用 Hits[0]（这是第 1 个与射线发生碰撞符合条件的可跟踪对象，也是离射线源最近的一个，通常就是需要的结果）碰撞点所在的位置放置虚拟物体。

4.4 可视化放置点

在本书前面的放置操作中，加载虚拟物体的一般做法是先检测显示平面，然后通过屏幕单击手势进行射线检测，在平面碰撞点位置加载虚拟物体，这种方式不直观，无法在放置前获知准确的放置位置。本节讲述一种规避这两个问题的加载虚拟物体的方法，即直接在可放置虚拟物体的地方显示一个放置指示图标，单击屏幕任何地方都会在指示图标所在位置放置虚拟物体。

虽然现在不再需要显示已检测到的平面，但还需要平面检测功能（为了增强真实感，虚拟物体需要放置在平面上而不是悬浮在空中，这就需要平面检测功能检测到可放置虚拟物体的平面），因此仍然需要在场景中的 AR Session Origin 对象上挂载 AR Plane Manager 组件，但由于不需要渲染显示已检测识别的平面，所以将其 Plane Prefab 属性置空，如图 4-11 所示。

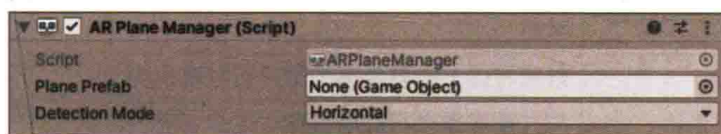


图 4-11 置空平面检测预制体属性

新建一个 C# 脚本，命名为 AppController，编写代码如下：

```
// 第 4 章 / 4-8
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;
```



```

[RequireComponent(typeof(ARRaycastManager))]
public class AppController : MonoBehaviour
{
    public GameObject mObjectPrefab;           // 需要放置的游戏对象预制体
    public GameObject mPlacementIndicator;      // 放置点指示图标预制体

    private ARRaycastManager mRaycastManager;
    private Pose placementPose;                // 放置点的姿态 Pose
    private GameObject placementIndicatorObject; // 实例化后的放置点指示图标对象
    private bool placementPoseIsValid = false; // 是否可放置指示器
    private static List<ARRaycastHit> Hits;     // 碰撞结果
    private Vector3 screenCenter;              // 屏幕中心点位置
    private Camera mCamera;                   // AR 主渲染相机

    void Start()
    {
        mRaycastManager = GetComponent<ARRaycastManager>();
        ARSessionOrigin mSession = GetComponent<ARSessionOrigin>();
        placementIndicatorObject = Instantiate(mPlacementIndicator, mSession.
trackablesParent);
        mCamera = Camera.main;
        screenCenter = mCamera.ViewportToScreenPoint(new Vector3(0.5f, 0.5f));
        Hits = new List<ARRaycastHit>();
    }

    void Update()
    {
        UpdatePlacementIndicator();
        // 在指示图标位置放置虚拟对象
        if (placementPoseIsValid && Input.touchCount > 0 && Input.GetTouch(0).
phase == TouchPhase.Began)
        {
            Instantiate(mObjectPrefab, placementPose.position, Quaternion.
identity);
        }
        // 进行射线检测，确定指示图标是否可见，更新指示图标姿态
        private void UpdatePlacementIndicator()
        {
            mRaycastManager.Raycast(screenCenter, Hits, TrackableType.
PlaneWithinPolygon | TrackableType.PlaneWithinBounds);

            placementPoseIsValid = Hits.Count > 0;
            if (placementPoseIsValid)
            {
                placementPose = Hits[0].pose;
            }
        }
    }
}

```



```

        var cameraBearing = new Vector3(mCamera.transform.forward.x, 0,
mCamera.transform.forward.z).normalized;
        placementPose.rotation = Quaternion.LookRotation(cameraBearing);
        placementIndicatorObject.transform.SetPositionAndRotation(placementP
ose.position, placementPose.rotation);
        placementIndicatorObject.SetActive(true);
    }
    else
    {
        placementIndicatorObject.SetActive(false);
    }
}
}

```

上述代码首先实例化一个指示图标，然后使用射线检测的方式检查与检测到平面的碰撞情况寻找可放置点（射线从设备屏幕的正中心位置发出，不再需要用手指单击屏幕），在检测到与平面碰撞后显示并实时更新指示图标姿态，当放置位置有效（指示图标显示）时单击屏幕后会在指示图标所在位置放置虚拟物体。

将该脚本挂载在场景中的 AR Session Origin 对象上，并为其属性赋上指示图标与虚拟物体预制体，编译运行，当检测到平面时会显示可放置指示图标，单击屏幕的任何地方都会会在指示图标显示位置放置虚拟物体，效果如图 4-12 所示。

通过这种方式，在放置虚拟物体之前，用户就可以看到放置位置，更直观，也更便于用户理解。

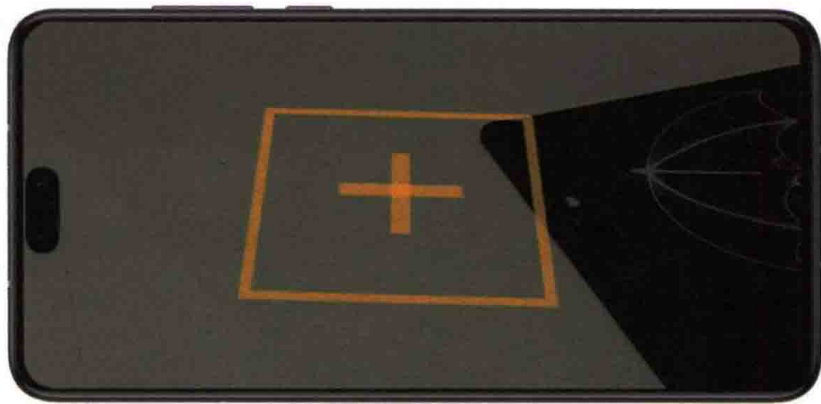


图 4-12 使用指示图标方式标识放置点位置

4.5 特征点与点云

ARCore 通过视觉 SLAM 进行运动跟踪，根据设备摄像头采集的图像信息估计相机的运动。基于特征点的视觉 SLAM 具有稳定及对光照、动态物体不敏感的优势，也是目前主流的视觉

SLAM 方案。特征点与点云对 AR 跟踪的稳定性至关重要，也是实现虚实融合的关键因素。

4.5.1 特征点

视觉 SLAM 的核心任务是根据输入图像估计相机运动，然而，图像本身是一个由亮度和色彩组成的二维矩阵，图像元素数量巨大，单像素粒度太细，不具备可区分性，因此，从图像矩阵层面进行分析会非常困难。主流的做法是从图像中选取比较有代表性的点，这些点在相机视角发生少量变化后会保持不变，便于在后续图像帧中找到相同的点，在这些点的基础上，讨论相机位姿估计问题，以及这些点的定位问题。在经典 SLAM 模型中，我们称这些点为路标（Land Mark），而在视觉 SLAM 中，路标则是指图像特征（Feature）。

图像特征是一组与计算任务相关的信息，特征点是图像中一些特别的地方，如图 4-13 所示，可以把图像中的角点、边缘和区块都当成图像中有代表性的地方。通过图 4-13 可以看到，指出某两幅图像中出现了同一个角点最简单；指出某两幅图像中出现同一条边缘则稍微困难一些，因为沿着该边缘前进，图像局部是相似的；指出某两幅图像中出现同一个区块则是最困难的。通过图示也可以看到，图像中的角点、边缘相比于像素区块而言更加“特别”，它们在不同图像之间的辨识度更高，所以，常见的特征点提取方式就是在不同图像间提取角点，确定它们的对应关系，在这种做法中，角点就是所谓的特征点。角点的提取算法有很多，例如 Harris 角点、FAST 角点、GFTT 角点等，但在大多数应用中，单纯的角点依然不能满足 SLAM 定位跟踪需求，例如，从远处看上去是角点的地方，当相机离近之后，可能就不显示为角点了，或者，当旋转相机时，角点的外观会发生变化，也就不容易在不同图像帧间进行匹配，为此，计算机视觉领域的研究者在长期的研究中设计了许多更加稳定的局部图像特征，如 SIFT、SURF、ORB 特征等。相比于朴素的角点，这些人工设计的特征点具有可重复性（Repeatability）、可区别性（Distinctiveness）、高效率（Efficiency）、本地性（Locality）等特点，可以保持图像特征具有旋转、尺度不变性。

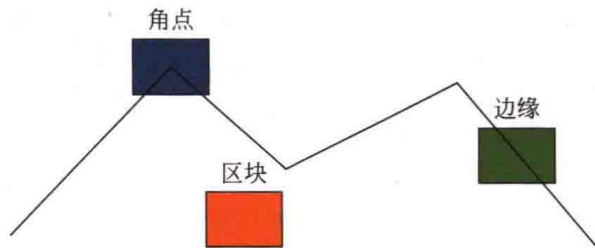


图 4-13 特征点类型示意图

所以更简单地说，特征点指 ARCore 通过 VIO 检测捕获的摄像头图像中的视觉差异点，这些视觉差异点是从图像中明暗、颜色、灰度差异比较大的点中挑选出来的，特征点的检测每一帧都会进行，提取图像特征是进行 AR 运动跟踪的重要环节，图像特征点示意图如图 4-14 所示。

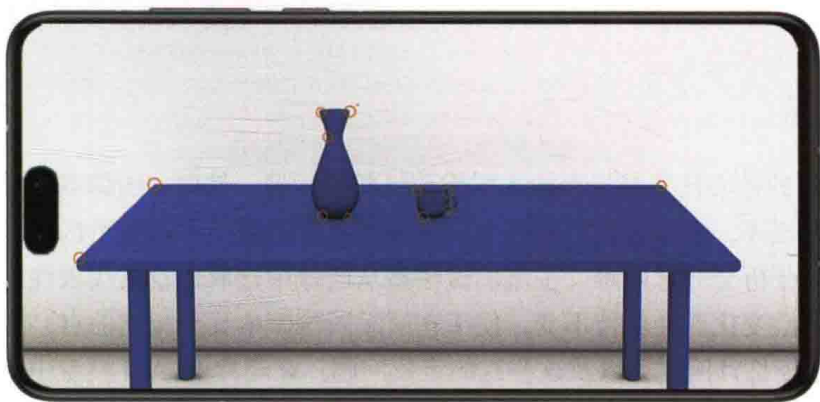


图 4-14 图像特征点提取示意图

4.5.2 点云

提取特征点的主要意义在于利用特征点可以在不同图像帧之间进行匹配, 由于特征点具有旋转和尺度不变性, 在不同图像帧之间进行的匹配也具有稳定性, 即可了解同一个特征点在不同的图像中的位置, 利用这个位置关系就可以反向估算出采集这两幅图像时的相机相对位置变化情况^①。在计算出相机位置变化关系后就可以把特征点从二维映射到三维, 这个过程称为三角化 (Triangulation), 三角化的过程其实就是估计出特征点的深度信息的过程, 三角化后的空间特征点称为三维空间点, 众多三维空间点就称为点云, 所以特征点与点云有直接关系, 只是特征点用于描述图像中的视觉差异点, 是在二维空间中, 而三维空间点是二维特征点的三维化, 是在三维空间中。

在 AR 应用启动后, AR Foundation 即开始实时跟踪这些点云信息, 在时间的推移中, 新的三维空间点会被添加、原有三维空间点信息会被更新、无效的三维空间点会被删除, 然后逐渐地就会形成一些稳定的三维空间点集合, 利用这个稳定的集合, 不仅可以跟踪到用户 (手机设备) 随着时间推移而相对于周围世界的姿态变化, 还可以大致了解用户周边的环境结构, 不仅如此, AR Foundation 也通过点云信息来检测平面。

点云是一个三维空间点的集合, 可以在帧与帧之间发生变化, 在一些平台上, 一个应用只生成维护一个点云, 而另一些平台则会将其三维空间点组织到不同空间区域的不同点云中。

在 AR Foundation 中, 三维空间点都可以在帧与帧之间被识别, 每个三维空间点都具有唯一的标识符。每个三维空间点都有一个 `Vector3` 类型的位置值 (positions)、一个 `ulong` 类型的 id 值 (identifiers)、一个 `float` 类型的置信值 (Confidence Value)。置信值表示 AR Foundation 对每个三维空间点的信心程度, 范围为 $[0,1]$, 值越大表示对这个空间点越确信。

在代码层面, 三维空间点的位置坐标可以通过 `ARPointCloud.position` 获取^②、id 标识符可

① 准确描述应该是位姿关系, 不仅包括位置关系, 还包括旋转关系。

② 需要注意的是, 这个坐标值为会话空间中的坐标值, 不是 Unity 空间中的坐标值。

以通过 `ARPointCloud.identifiers` 获取、置信值可以通过 `ARPointCloud.confidenceValues` 获取。

在 AR Foundation 中，点云由 AR Point Cloud Manager 组件负责管理，该组件负责点云的创建及三维空间点的创建、更新、移除。在使用中，空间点检测的启用禁用、空间点的显示隐藏与 AR Plane Manager 组件对平面处理方式一致。

点云数据可以辅助平面检测，更重要的是点云数据是环境重建的基础，这对 AR 理解环境非常关键。随着时间的推移，点云中的空间点数量会迅速增长，为了防止空间点过多而影响性能，点云中的总空间点数量必须控制在一定的数值范围内，ARCore 将空间点的最大数量限制在 61440 个。

AR 应用启动后即开始构建点云，为直观看到点云信息，可以将点云数据渲染出来，在 Unity 中层级窗口（Hierarchy 窗口）空白处右击，依次选择 `XR` → `AR Default Point Cloud` 创建一个点云对象，然后将 AR Default Point Cloud 对象从层级窗口拖动到工程窗口（Project 窗口）中的 Prefabs 文件夹下制作成一个预制体，并删除层级窗口中的该对象。

在层级窗口选择 AR Session Origin 对象，为其添加 AR Point Cloud Manager 组件，并将刚才制作好的 AR Default Point Cloud 预制体赋给 AR Point Cloud Manager 组件下的 Point Cloud Prefab 属性。

编译运行，找一个富纹理物体或表面，左右移动手机，这时 AR Foundation 会对特征点进行检测并渲染，效果如图 4-15 所示。



图 4-15 可视化点云效果图

特征点提取每帧都会进行，也会发生一些变化，三角化后的三维空间点也会随时间发生变化，置信值小的空间点数据会被移除，新的空间点数据会被加入。采用这种方式可视化点云只会渲染当前帧图像中置信度比较高的空间点，所以看到的渲染结果每帧都会有变化。

4.5.3 点云数据采集

AR Point Cloud Manager 组件管理所有点云数据，点云数据是进行 SLAM 运动跟踪的关键

数据,也是进行网格重建、实现虚实遮挡等高级功能的重要数据^①,我们也可以保存所有检测到的点云数据,可以将这些数据存储在字典或者列表中,或者存储到文件系统中。

下面演示将点云数据存储到文件系统中,新建一个C#脚本文件,命名为CloudPoints,并编写代码如下:

```
// 第4章 / 4-9
using System.Collections.Generic;
using System.IO;
using System.Text;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using System.Collections;
using UnityEngine.UI;

[DisallowMultipleComponent]
[RequireComponent(typeof(ARPointCloudManager))]
public class CloudPoints : MonoBehaviour
{
    public Button BtnStart;           // 开始采集点云 UI 按钮
    public Button BtnStopAndSave;     // 停止采集点云并保存 UI 按钮

    private string fileName;          // 保存点云的文件名
    private readonly string format = "MMddHHmmssy"; // 文件名日期格式
    private ARPointCloudManager arPointCloudManager; // 点云组件
    // 点云点集合
    private Dictionary<ulong, Points4> mPoints = new Dictionary<ulong, Points4>();
    private void Start()
    {
        fileName = System.DateTime.Now.ToString(format);
        arPointCloudManager = GetComponent<ARPointCloudManager>();
        BtnStart.onClick.AddListener(StartCloudSession);
        BtnStopAndSave.onClick.AddListener(StopCloudSession);
    }

    // 开始采集点云数据
    private void StartCloudSession()
    {
        mPoints.Clear();
        arPointCloudManager.pointCloudsChanged += OnPointCloudChanged;
    }

    // 停止点云数据采集,并保存点云数据
    private void StopCloudSession()
    {
        arPointCloudManager.pointCloudsChanged -= OnPointCloudChanged;
    }
}
```

^① ARCore 只提供稀疏点云,对场景重建而言是不充分的,但可以作为其他高级功能的数据来源。

```

StoreCloudPointsData();
}

// 格式化点云数据, 每组 4 个, 分别是 XYZ、置信度、分号 (;) 隔离、保存数据
private void StoreCloudPointsData()
{
    if (mPoints.Count > 0)
    {
        StringBuilder sb = new StringBuilder();
        foreach (var value in mPoints.Values)
        {
            sb.Append(value.posX);
            sb.Append(",");
            sb.Append(value.posY);
            sb.Append(",");
            sb.Append(value.posZ);
            sb.Append(",");
            sb.Append(value.confidence.ToString("F2"));
            sb.Append(";");
        }
        // 存储到文件系统
        StartCoroutine(SaveDataToDisk(sb.ToString()));
    }
    else
        Debug.Log("无点云数据可保存!");
}

// 保存数据协程
private IEnumerator SaveDataToDisk(string data)
{
    string filename = "CloudPoints" + fileName + ".txt";
    string path = Application.persistentDataPath;
    string filePath = Path.Combine(path, filename);
    File.WriteAllText(filePath, data);
    yield return 0;
    Debug.Log("点云数据保存成功!");
}

// 更新所有点云数据
void OnPointCloudChanged(ARPointCloudChangedEventArgs eventArgs)
{
    foreach (var pointCloud in eventArgs.added)
    {
        if (pointCloud.positions.HasValue)
        {
            for (int i = 0; i < pointCloud.positions.Value.Length; i++)
            {

```



```

        Points4 points4 = new Points4(pointCloud.positions.Value[i].
x, pointCloud.positions.Value[i].y, pointCloud.positions.Value[i].z, pointCloud.
confidenceValues.Value[i]);
        mPoints.Add(pointCloud.identifiers.Value[i], points4);
    }
}
}
foreach (var pointCloud in eventArgs.updated)
{
    if (pointCloud.positions.HasValue)
    {
        for (int i = 0; i < pointCloud.positions.Value.Length; i++)
        {
            Points4 points4 = new Points4(pointCloud.positions.Value[i].
x, pointCloud.positions.Value[i].y, pointCloud.positions.Value[i].z, pointCloud.
confidenceValues.Value[i]);
            mPoints[pointCloud.identifiers.Value[i]] = points4;
        }
    }
}
foreach (var pointCloud in eventArgs.removed)
{
    if (pointCloud.positions.HasValue)
    {
        for (int i = 0; i < pointCloud.positions.Value.Length; i++)
        {
            mPoints.Remove(pointCloud.identifiers.Value[i]);
        }
    }
}
}
// 自定义空间点结构体
class Points4
{
    public float posX;
    public float posY;
    public float posZ;
    public float confidence;
    public Points4(float x, float y, float z, float confidenceValue)
    {
        posX = x;
        posY = y;
        posZ = z;
        confidence = confidenceValue;
    }
}
}

```

上述代码主要利用 AR Point Cloud Manager 组件的 pointCloudsChanged 事件, 通过注册与取消注册实现对点云数据的采集, 最后将点云数据存储到文件系统中, 代码还演示了获取空间点位置信息、id 信息、置信度信息的一般方法。

在使用时, 只需将该脚本挂载在与 AR Point Cloud Manager 组件相同的场景对象上, 设置相应的按钮。

4.6 锚点

锚点 (Anchor) 的原意是指不让船舶漂移的固定锚, 这里用来指将虚拟物体固定在 AR 空间上的一种技术。由于 AR 运动跟踪使用的 IMU 传感器的特性, 误差会随着时间积累, 所以需要图像检测等方式对误差进行修正, 此时, 如果已存在于空间上的对象不同步进行校正, 则会出现偏差, 锚点的功能即是绑定虚拟物体与现实空间位置。被赋予锚点的对象将被视为固定在空间上的特定位置, 并自动进行位置校正, 因此锚点可以确保物体在空间中保持相同的位置和方向, 让虚拟物体在 AR 场景中看起来待在原地不动, 如图 4-16 所示。



图 4-16 连接到锚点上的虚拟对象如同固定在现实世界空间中一样

锚点的工作原理如下:

在 AR 应用中, 渲染相机和虚拟物体在现实世界空间中的位置会在帧与帧之间更新, 即虚拟物体在现实世界空间中的姿态每帧都会更新, 由于 IMU 传感器的误差积累, 虚拟物体会出现漂移现象, 为了解决这个问题, 需要使用一个参考点将虚拟对象固定在现实空间中, 如前文所述, 这个参考点姿态信息的偏差必须能用某种方式消除以确保参考点的姿态不会随着时间而发生变化。消除这个偏差的就是视觉校准技术, 通过视觉校准能让参考点保持相同的位置与方向, 这样, 连接到该参考点的虚拟对象也就不会出现漂移了, 这个参考点就是锚点。一个锚点上可以连接一个或多个虚拟对象, 锚点和连接到它上面的物体看起来会待在它们在现实世界中的放置位置, 随着锚点姿态在每帧中进行调整以适应现实世界空间更新, 锚点也将相应地更新物体的姿态, 确保这些物体能够保持它们的相对位置和方向, 即使在锚点姿态调整的情况下也能如此。有了锚点, 连接到锚点上的虚拟对象就像是固定在现实世界空间中

一样^①。

锚点是一种对资源消耗比较大的可跟踪对象，锚点的跟踪、更新、管理需要大量的计算开销，因此需要谨慎使用并在不需要的时候进行分离。

通常而言，在以下情况时有必要使用锚点：

(1) 保持两个或多个虚拟对象的相对位置。在这种情况下，可以将两个或多个虚拟物体挂载在同一个锚点下，这样，锚点会使用相同的矩阵更新挂载在其下的虚拟对象，因此可以保持它们之间的位置关系不受其他因素影响。

(2) 保证虚拟物体的独立性。在前文中，通过射线检测的方式在平面上放置了虚拟物体，通常情况下这没有问题，但如果后来因为某种原因 AR Plane Manager 被禁用、平面被销毁或者隐藏，就会影响到以平面为参考的虚拟物体位置的稳定性，导致虚拟物体漂移。在这种情况下，如果使用了锚点，将虚拟物体挂载在锚点下就能保持虚拟物体的独立性，不会受到平面状态的影响。

(3) 提高跟踪稳定性。使用锚点后，即通知跟踪系统需要独立跟踪该点从而提高挂载在该锚点上虚拟物体的稳定性。

(4) 保持跟踪对象与平面的相对位置稳定。使用 AttachAnchor() 方法可以将一个锚点与平面绑定起来，从而保持挂载在锚点下的虚拟物体与平面保持关系一致。如在一个垂直平面上使用 AttachAnchor() 方法建立一个锚点，在锚点更新时就会锁定 X、Z 分量值，从而保持虚拟物体与平面位置关系始终一致。

在 AR Foundation 中，锚点管理由 AR Anchor Manager 组件负责^②，其提供的主要方法和事件如表 4-4 所示。

表 4-4 AR Anchor Manager 组件提供的主要方法和事件

方 法	类型	描 述
AddAnchor(Pose)	方法	通过给定的 Pose 添加一个锚点，Pose 为世界空间中的位姿，返回一个新的 ARAnchor 对象
AttachAnchor(ARPlane, Pose)	方法	通过给定的 Pose 创建一个相对于已检测到平面的 ARAnchor 对象，其中 ARPlane 是一个已检测到的平面，Pose 为世界空间中的位姿
RemoveAnchor(ARAnchor)	方法	移除一个 ARAnchor 对象，如果移除成功，则返回值为 true，如果返回值为 false，则通常意味着这个 ARAnchor 对象已经不在跟踪状态
anchorsChanged	事件	在 ARAnchor 对象发生变化时触发的事件，如新的 ARAnchor 对象创建、对一个现存的 ARAnchor 对象进行更新、移除 ARAnchor 对象

① 从技术理论上而言，锚点实际上对其周围环境的特征进行了提取，然后在每一帧中进行特征匹配，这样就可以确保即使 SLAM 运动跟踪发生了漂移，也能将虚拟物体与现实空间锚定。

② AR Anchor Manager 组件有一个 Anchor Prefab 属性，该属性的使用有些特别的地方，通常不需要设置该属性，如果需要扩展锚点功能，则可先查阅官方资料。

AddAnchor(Pose) 方法与 AttachAnchor(ARPlane, Pose) 方法的区别在于, AddAnchor(Pose) 方法是创建一个锚点, 并要求跟踪系统跟踪空间中的这个特定位置, 如果会话发生变化, 则它可能会更新; AttachAnchor(ARPlane, Pose) 方法是 将一个锚点“附加”到一个平面上, 该锚点会保持与平面的相对距离。例如, 如果平面是水平的, 则锚点仅在平面的 Y 位置更改时上下移动, 其他情况下 Y 值不会发生变化, 这对于在平面上或平面附近“粘贴”虚拟对象很有用。

锚点操作是一个计算密集型操作, 添加一个 ARAnchor 对象需要一到两帧的时间, 在添加 ARAnchor 对象操作执行后到添加成功之前, 添加的 ARAnchor 对象处于 pending (挂起) 状态, 这个 pending 状态值可以通过 ARAnchor.pending 属性进行查询; 同样, 移除一个 ARAnchor 对象也需要一到两帧的时间, 如果尝试去移除一个还没有添加成功的 ARAnchor 对象, 则不会有任何效果。

锚点由 AR Anchor Manager 组件负责管理, 不需要使用 Destroy() 方法去销毁, 事实上这会引发错误。如果需要移除锚点, 则务必要通过 RemoveAnchor(ARAnchor) 方法移除 ARAnchor 对象。

下面演示锚点的使用方法, 新建一个 C# 脚本文件, 命名为 ARAnchorController, 编写代码如下:

```
// 第4章 /4-10
using UnityEngine;
using UnityEngine.XR.ARFoundation;

[RequireComponent(typeof(ARAnchorManager))]
public class ARAnchorController : MonoBehaviour
{
    public GameObject spawnPrefab; // 虚拟对象预制体

    private GameObject spawnedObject = null; // 实例化后的虚拟对象
    private ARAnchorManager mARAnchorManager;
    private readonly float distance = 0.5f; // 放置在设备前的距离
    private Camera mCamera; // 主渲染相机

    private void Start()
    {
        mARAnchorManager = transform.GetComponent<ARAnchorManager>();
        mCamera = Camera.main;
    }

    void Update()
    {
        if (Input.touchCount < 1)
            return;
        var touch = Input.GetTouch(0);
        if (spawnedObject == null && touch.phase == TouchPhase.Began)
        {
            // 方法一
        }
    }
}
```

```

        Vector3 mMenu = mCamera.transform.forward.normalized * distance;
        Pose mPose = new Pose(mCamera.transform.position + mMenu, mCamera.
transform.rotation);
        ARAnchor mReferencePoint = mARAnchorManager.AddAnchor(mPose);
        spawnedObject = Instantiate(spawnPrefab, mCamera.transform.position
+ mMenu, mCamera.transform.rotation, mReferencePoint.gameObject.transform);

        // 方法二
        Vector3 mMenu = mCamera.transform.forward.normalized * distance;
        Pose mPose = new Pose(mCamera.transform.position + mMenu, mCamera.
transform.rotation);
        spawnedObject = Instantiate(spawnPrefab, mCamera.transform.position
+ mMenu, mCamera.transform.rotation);
        spawnedObject.AddComponent<ARAnchor>();
    }
}
}

```

在上述代码中, 希望在用户设备正前方 0.5m 远的地方放置一个锚点, 并在这个锚点上连接一个虚拟对象。可以使用两种方法实现: 第一种方法更直观, 首先在指定的空间位置创建一个 ARAnchor 对象, 然后将虚拟对象连接到该锚点, 这样就可以锚定虚拟对象与现实环境之间的关系; 第二种方法更自然, 也是目前推荐的使用方式, 只需正常实例化虚拟对象, 然后在该虚拟对象上挂载 ARAnchor 组件。

使用锚点放置虚拟对象, 无须事先检测到平面或者特征点, 而且能保持虚实关系, 效果如图 4-17 所示。

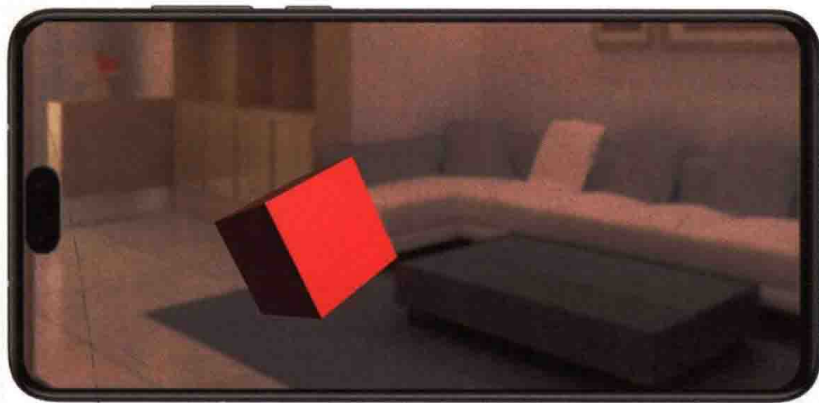


图 4-17 在空中锚定虚拟对象效果示意图

使用锚点的注意事项。

1. 尽可能复用锚点

在大多数情况下, 应当让多个相互靠近的物体使用同一个锚点, 而不是为每个物体创建

一个新锚点。如果物体需要保持与现实世界空间中的某个可跟踪对象或位置之间独特的空间关系,则需要为对象创建新锚点。因为锚点将独立调整姿态以响应 AR Foundation 在每一帧中对现实世界空间的估算,如果场景中的每个物体都有自己的锚点,则会带来很大的性能开销。另外,独立锚定的虚拟对象可以相对彼此平移或旋转,从而破坏虚拟物体间相对位置应保持不变的 AR 场景体验。

例如,假设 AR 应用可以让用户在房间内布置虚拟家具。当用户打开应用时,AR Foundation 会以平面形式开始跟踪房间中的桌面和地板。用户在桌面上放置一盏虚拟台灯,然后在地板上放置一把虚拟椅子,在此情况下,应将一个锚点连接到桌面平面,将另一个锚点连接到地板平面。如果用户向桌面添加另一盏虚拟台灯,此时可以重用已经连接到桌面平面的锚点。这样,两个台灯看起来都粘在桌面平面上,并保持它们之间的相对位置,椅子也会保持它相对于地板平面的位置。

2. 保持物体靠近锚点

锚定物体时,最好让需要连接的虚拟对象尽量靠近锚点,避免将物体放置在离锚点几米远的地方,以免由于 AR Foundation 更新现实世界空间坐标而产生意外的旋转运动。如果确实需要将物体放置在离现有锚点几米远的地方,则应该创建一个更靠近此位置的新锚点,并将物体连接到新锚点。

3. 分离未使用的锚点

为提升应用性能,通常需要将不再使用的锚点分离。因为每个可跟踪对象都会产生一定的 CPU 开销,AR Foundation 不会释放具有连接锚点的可跟踪对象,从而造成无谓的性能损失。

4. 锚点不可以手动更新

锚点由 AR Anchor Manager 组件负责自动更新,由于锚点采集了周边环境信息,此信息与现实环境空间位置绑定,开发人员无法通过重新设置锚点的 transform 变更其位置,所以如果希望变更锚点,则应当通过 RemoveAnchor() 方法(或者将 ARAnchor 组件禁用)移除锚点,然后创建新锚点。



对二维图像进行检测与跟踪是 AR 技术应用最早的领域之一，利用设备摄像头获取的图像数据，通过计算机图像算法对图像中的特定二维图像部分进行检测识别与姿态跟踪，并利用二维图像的姿态叠加虚拟物体对象，这种方法称为 Marker Based AR（基于标识的 AR），是 AR 早期最主要的形式，经过多年的发展，二维图像检测跟踪标识物已经从特定的 Marker 发展到普通图像。本章主要学习利用 AR Foundation 检测识别与跟踪二维图像的方法。

5.1 图像检测跟踪基本操作

在 AR 中，图像跟踪技术是指通过图像处理技术对设备摄像头中拍摄到的二维图像进行检测识别定位，并对其姿态进行跟踪的技术。图像跟踪技术的基础是图像识别，图像识别是指识别和检测出数字图像或视频中对象或特征的技术，图像识别技术是信息时代的一门重要技术，其产生的目的是为了计算机代替人类去处理大量的图形图像及真实物体信息，是众多其他技术的基础。

在 AR Foundation 中，图像跟踪系统依据参考图像库中的图像信息尝试在设备摄像头捕获的图像中检测匹配二维图像并跟踪之，在 AR Foundation 的图像跟踪处理中，一些特定的术语如表 5-1 所示。

表 5-1 图像跟踪术语表

术 语	描述说明
参考图像 (Reference Image)	识别二维图像的过程实际上是一个特征值比对的过程，AR Foundation 将从设备摄像头中获取的图像信息与参考图像库中的图像特征值信息进行比较，存储在参考图像库中的用于对比的图像叫作参考图像。一旦对比成功，真实环境中的图像将与参考图像库的参考图像建立对应关系，每个真实二维图像的姿态信息也一并被检测
参考图像库 (Reference Image Library)	参考图像库用来存储一系列的参考图像用于对比，每个图像跟踪程序都必须有一个参考图像库，但需要注意的是，参考图像库中存储的实际是参考图像的特征值信息而不是原始图像，这有助于提高对比速度和稳健性。参考图像库越大，图像对比就会越慢，建议参考图像库的图像数量不要超过 1000 张，通常应该控制在 100 张以内

续表

术 语	描述说明
跟踪组件提供方 (Provider)	AR Foundation 架构在底层 SDK 图像跟踪 API 之上，也就是说 AR Foundation 并不具体负责图像识别算法，它只提供一个接口，具体图像检测识别由算法提供方提供

ARCore 图像检测识别主要技术指标如表 5-2 所示。

表 5-2 ARCore 图像检测识别主要技术指标

序号	描 述
1	每个参考图像库可以存储最多 1000 张参考图像的特征点信息
2	ARCore 可以在环境中同时跟踪最多 20 张图像，但无法跟踪同一图像的多个实例
3	环境中的物理图像必须至少为 15cm × 15cm 且必须平坦（如不能起皱或卷绕在瓶子上）
4	在物理图像被跟踪后，ARCore 会提供对位置、方向和物理大小的估计值，随着 ARCore 收集的数据增多，这些估计值会持续优化
5	ARCore 可以跟踪移动中的图像
6	所有跟踪都在设备上完成，无须网络连接，支持在设备端或通过网络更新参考图像及参考图像库，无须重新安装应用

在 AR Foundation 中，使用二维图像检测跟踪时，基本操作分成两步：第一步是建立一个参考图像库；第二步是在场景中挂载 AR Tracked Image Manager 组件，并将一个需要实例化的预制体赋给其 Tracked Image Prefab 属性。

按照上述步骤，下面演示使用二维图像检测跟踪的基本流程。在 Unity 工程中，首先建立一个参考图像库，在工程窗口（Project 窗口）中的 ImageLib 文件夹下右击并依次选择 Create → XR → Reference Image Library 创建一个参考图像库，命名为 RefImageLib，如图 5-1 所示。

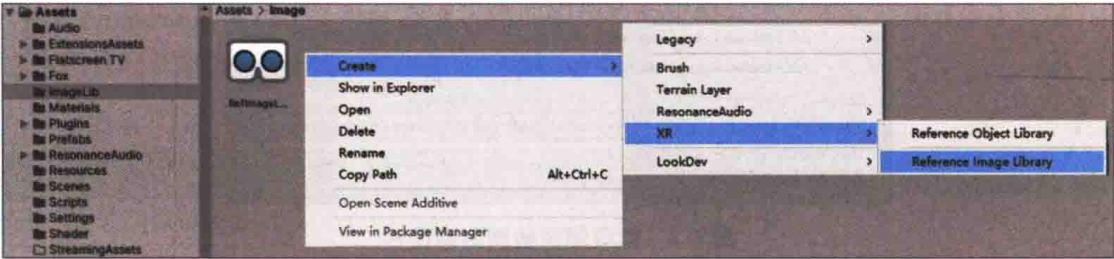


图 5-1 新建参考图像库

选择新建的 RefImageLib 参考图像库，在属性窗口（Inspector 窗口）中，单击 Add Image 按钮添加参考图像，将参考图像拖动到左侧图像框中^①，如图 5-2 所示。

^① 参考图像被添加到图像框中后，后台会进行参考图像的特征信息提取，实质上参考图像库中存储的是参考图像的特征信息。

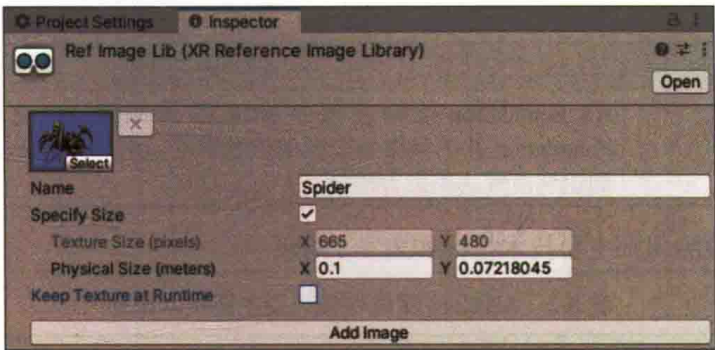


图 5-2 添加参考图像

在图 5-2 中，每个参考图像除了图像信息外还有若干其他属性，其具体含义如表 5-3 所示。

表 5-3 参考图像属性术语表

术 语	描述说明
Name	一个标识参考图像的名字，这个名字在进行图像对比时没有作用，但在比对匹配成功后可以通过参考图像名字获知是哪个参考图像，参考图像名字可以重复，因为在跟踪时，跟踪系统还会给每个参考图像一个 <code>referenceImage.guid</code> 值，利用这个 GUID 值唯一标识每个参考图像
Specify Size	这是个可选值，为加速图像检测识别过程，一些底层 SDK 要求提供一个二维待检测图像的真实物理尺寸，所以如果要设置，则这个值一定会是一个大于 0 的长和宽值对，当一个值发生变化时，Unity 会根据参考图像的长宽比例自动调整另一个值
Keep Texture at Runtime	一个默认的纹理，这个纹理可以用于修改预制体模型的外观

完成上述工作之后，在层级窗口中选择 AR Session Origin 游戏对象，并为其挂载 AR Tracked Image Manager 组件，将第一步制作的 RefImageLib 参考图像库拖动到其 Serialized Library 属性中，并设置相应的需要在检测到二维图像后实例化的预制体，如图 5-3 所示。



图 5-3 挂载图像检测跟踪组件

参考图像库也可以在运行时动态设置，不过 AR Tracked Image Manager 组件一旦启动图像检测跟踪，参考图像库就必须有设定值，不能为 null，即要求在 AR Tracked Image Manager 组件 Enable 之前设置参考图像库，后文会详细阐述。Max Number of Moving Images 属性指定了最大可跟踪的动态图像的数量，这里的动态图像是指被跟踪图像可以旋转、平移等，即被跟踪图像姿态可以发生变化，动态图像跟踪是一个非常消耗 CPU 性能的任务，过多的动态图像跟踪会导致应用性能下降。

编译运行，将设备摄像头对准需要检测的二维图像，跟踪效果如图 5-4 所示。仔细观察实例化后的虚拟蜘蛛对象，会发现模型在 Y 轴上旋转了 180° ，这是由于 ARCore 所采用的坐标系与 Unity 采用的坐标系不同造成的，为解决这个问题，在 ARCore 中实例化虚拟对象时需要将其绕自身 Y 轴再旋转 180° ，但在 AR Foundation 中，实例化虚拟对象是由 AR Tracked Image Manager 自动完成的，无法干预，所以目前使用这种方式实例化虚拟对象时，需要先把模型在 Y 轴上进行 180° 旋转再使用^①。

提示

如果将 Tracked Image Prefab 属性设置为一个播放视频的预制体，在检测到二维图像时，AR 应用就可以播放相应视频，并且 AR Foundation 会自动调整视频尺寸和姿态以适应二维图像大小与姿态，在检测到的二维图像上播放视频功能在一些场景中很有用，例如数字名片。视频播放功能可参见第 11 章。

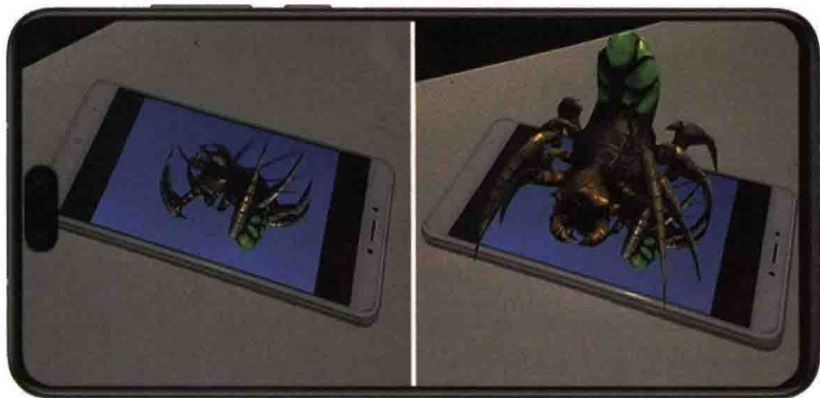


图 5-4 图像识别跟踪效果图

AR Foundation 使用了 3 种状态表示当前图像检测跟踪状态，各状态如表 5-4 所示，可以在应用运行时获取图像跟踪状态信息。

表 5-4 二维图像跟踪状态

术 语	描述说明
None	图像还未被跟踪，这也可能是第一次检测到图像时的状态
Limited	图像已经被跟踪，但当前属于跟踪受限状态。在以下两种情况时图像跟踪会进入受限状态： ①图像在当前帧中不可见；②当前图像被检测到，但并未跟踪，如跟踪图像数量超过了 <code>maxNumberOfMovingImages</code> 属性设定值
Tracking	图像正在跟踪中

^① 将模型在 Y 轴上旋转 180° 有两种办法：①如果模型是自己制作的，或者熟悉常见三维建模软件，可以在三维建模软件中调整模型的旋转再导出；②在 Unity 层级窗口中新创建一个空对象，将模型作为其子对象，在 Y 轴上旋转模型，然后连同空对象一起保存为新的预制体使用。

除获取图像跟踪状态信息外，也能获取当前图像检测跟踪的其他信息，典型代码如下：

```
// 第 5 章 / 5-1
using UnityEngine;
using UnityEngine.XR.ARFoundation;

[RequireComponent(typeof(ARTrackedImageManager))]
public class TrackedImageInfoManager : MonoBehaviour
{
    ARTrackedImageManager mTrackedImageManager;

    void Awake()
    {
        mTrackedImageManager = GetComponent<ARTrackedImageManager>();
    }
    // 注册图像检测事件
    void OnEnable()
    {
        mTrackedImageManager.trackedImagesChanged += OnTrackedImagesChanged;
    }
    // 取消图像检测事件注册
    void OnDisable()
    {
        mTrackedImageManager.trackedImagesChanged -= OnTrackedImagesChanged;
    }
    // 输出图像跟踪信息
    void UpdateInfo(ARTrackedImage trackedImage)
    {
        Debug.Log(string.Format(
            "参考图像名: {0}, 图像跟踪状态: {1}, GUID 值: {2}, 参考图像尺寸: {3} cm, 实际图像尺寸: {4} cm, 空间位置: {5}",
            trackedImage.referenceImage.name,
            trackedImage.trackingState,
            trackedImage.referenceImage.guid,
            trackedImage.referenceImage.size * 100f,
            trackedImage.size * 100f,
            trackedImage.transform.position.ToString()));
    }
    // 图像检测跟踪状态处理方法
    void OnTrackedImagesChanged(ARTrackedImagesChangedEventArgs eventArgs)
    {
        foreach (var trackedImage in eventArgs.added)
        {
            // 设置默认图像尺寸比例，检测到的图像单位是厘米，需要缩放
            trackedImage.transform.localScale = new Vector3(0.01f, 1f, 0.01f);
            UpdateInfo(trackedImage);
        }
    }
}
```



```
foreach (var trackedImage in eventArgs.updated)
    UpdateInfo(trackedImage);
}
```

通过分析上述代码输出,可以了解到图像检测跟踪状态的变化情况,当被检测图像移出视野后,图像跟踪状态并不是 None 状态,而是 Limited 状态,当该图像再次被检测到时,其跟踪状态会再次变为 Tracking,但 GUID 值会保持不变。

5.2 图像跟踪功能启用与禁用

在 AR Foundation 中,实例化出来的虚拟对象并不会随着被跟踪二维图像的消失而消失(在实例化虚拟对象后取走二维图像,虚拟对象并不会被销毁,而是保持在最后一次二维图像被检测到的空间位置),虚拟对象停留在原来的位置上,在某些应用场景下是合理的,但在另一些场景下也会变得很不合适,而且,图像跟踪是一个非常消耗性能的操作,在不使用图像跟踪时一般应当把图像跟踪功能关闭。参考平面检测功能的启用与禁用,也可以通过脚本代码来控制图像跟踪功能的启用与禁用及所跟踪对象的显示与隐藏,典型代码如下:

```
// 第5章/5-2
public Text mToggleImageDetectionText; // 显示当前显隐状态
private ARTrackedImageManager mARTrackedImageManager;
void Awake()
{
    mARTrackedImageManager = GetComponent<ARTrackedImageManager>();
}
// 启用与禁用图像跟踪
public void ToggleImageTracking()
{
    mARTrackedImageManager.enabled = !mARTrackedImageManager.enabled;

    string ImageDetectionMessage = "";
    if (mARTrackedImageManager.enabled)
    {
        ImageDetectionMessage = "禁用图像跟踪 ";
        SetAllImagesActive(true);
    }
    else
    {
        ImageDetectionMessage = "启用图像跟踪 ";
        SetAllImagesActive(false);
    }

    if (mToggleImageDetectionText != null)
        mToggleImageDetectionText.text = ImageDetectionMessage;
```



```
}  
  
// 显示或者隐藏所有已实例化的虚拟对象  
void SetAllImagesActive(bool value)  
{  
    foreach (var img in mARTrackedImageManager.trackables)  
        img.gameObject.SetActive(value);  
}
```

与平面检测功能启用与禁用类似, 可以使用一个按钮进行状态切换, 从而在运行时控制图像跟踪功能的启用与禁用。

5.3 多图像跟踪

在 AR Tracked Image Manager 组件中, Tracked Image Prefab 属性指定了需要实例化的虚拟对象。在默认情况下, AR Foundation 支持多图像跟踪, 通过在 RefImageLib 参考图像库中添加多张参考图像, 在这些参考图像被检测到时, 每个被检测到的参考图像都会生成一个虚拟对象, 如图 5-5 所示。



图 5-5 AR Foundation 默认支持多图像跟踪

但在 AR 应用运行时, 只能有一个 AR Tracked Image Manager 组件运行 (多个 AR Tracked Image Manager 组件会导致跟踪冲突), 这样就只能设置一个 Tracked Image Prefab 属性, 即不能实例化多种虚拟对象, 这将极大地限制图像跟踪的实际应用, 所以为了实例化多种虚拟对象, 只能动态地修改 Tracked Image Prefab 属性。经过测试, 会发现在 ARFoundation 中, AR Tracked Image Manager 组件在 trackedImagesChanged 事件触发之前就已经实例化了虚拟对象, 因此这样就无法通过在 trackedImagesChanged 事件中修改 Tracked Image Prefab 属性达到实时改变需要实例化的虚拟对象之目的。

鉴于此, 可采用的解决思路是: 在 AR Tracked Image Manager 组件 Tracked Image Prefab 中设置第 1 个需要实例化的虚拟对象预制体, 然后在 trackedImagesChanged 事件中捕捉到图

像 added 操作, 将 Tracked Image Prefab 属性更改为下一个需要实例化的虚拟对象预制体, 这样来达到动态调整虚拟对象的目的。如正常将 Tracked Image Prefab 设置为 Spider 预制体, 在检测到 Spider 图像后, 将 Tracked Image Prefab 修改为 Cat 预制体, 这样, 在检测到 Cat 图像后就会实例化 Cat 预制体了。新建一个 C# 脚本, 命名为 MultiImageTracking, 编写代码如下:

```
// 第5章 /5-3
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation;

public class MultiImageTracking : MonoBehaviour
{
    ARTrackedImageManager mImgTrackedmanager;
    public GameObject[] ObjectPrefabs; // 虚拟对象预制体数组

    private void Awake()
    {
        mImgTrackedmanager = GetComponent<ARTrackedImageManager>();
    }

    // 注册图像跟踪事件
    private void OnEnable()
    {
        mImgTrackedmanager.trackedImagesChanged += OnTrackedImagesChanged;
    }

    // 取消图像跟踪事件注册
    void OnDisable()
    {
        mImgTrackedmanager.trackedImagesChanged -= OnTrackedImagesChanged;
    }

    // 图像跟踪状态改变处理方法
    void OnTrackedImagesChanged(ARTrackedImagesChangedEventArgs eventArgs)
    {
        foreach (var trackedImage in eventArgs.added)
        {
            OnImagesChanged(trackedImage.referenceImage.name);
        }
        //foreach (var trackedImage in eventArgs.updated)
        //{
        //    OnImagesChanged(trackedImage.referenceImage.name);
        //}
    }

    // 动态设置 trackedImagePrefab 属性
    private void OnImagesChanged(string referenceImageName)
    {
    }
}
```

```

        if (referenceImageName == "Spider")
        {
            mImgTrackedmanager.trackedImagePrefab = ObjectPrefabs[1];
            Debug.Log("Tracked Name is .." + referenceImageName);
            Debug.Log("Prefab Name is .." + mImgTrackedmanager.
trackedImagePrefab.name);
        }
        if (referenceImageName == "Cat")
        {
            mImgTrackedmanager.trackedImagePrefab = ObjectPrefabs[0];
        }
    }
}

```

编译运行, 先扫描检测识别 Spider 图像, 待实例化 Spider 虚拟对象后再扫描检测 Cat 图像, 这时就会实例化 Cat 虚拟对象了, 效果如图 5-6 所示。



图 5-6 实例化多个虚拟对象示意图

但这种方式其实有一个很大的弊端, 即必须按顺序检测图像, 因为无法在用户检测图像之前预测用户可能会扫描检测的二维图像。为解决这个问题, 就不能使用 AR Tracked Image Manager 组件实例化对象, 而由自己负责虚拟对象的实例化。将 AR Tracked Image Manager 组件下的 Tracked Image Prefab 属性置空, 为 MultiImageTracking 脚本的 ObjectPrefabs 数组赋上相应的预制体对象, 并修改代码, 修改后的代码如下:

```

// 第 5 章 / 5-4
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation;

public class MultiImageTracking : MonoBehaviour
{
    ARTrackedImageManager mImgTrackedmanager;
    public GameObject[] ObjectPrefabs;           // 虚拟对象预制体数组
}

```



```

private void Awake()
{
    mImgTrackedmanager = GetComponent<ARTrackedImageManager>();
}
// 注册图像跟踪事件
private void OnEnable()
{
    mImgTrackedmanager.trackedImagesChanged += OnTrackedImagesChanged;
}
// 取消图像跟踪事件注册
void OnDisable()
{
    mImgTrackedmanager.trackedImagesChanged -= OnTrackedImagesChanged;
}
void OnTrackedImagesChanged(ARTrackedImagesChangedEventArgs eventArgs)
{
    foreach (var trackedImage in eventArgs.added)
    {
        OnImagesChanged(trackedImage);
    }
    //foreach (var trackedImage in eventArgs.updated)
    //{
    //    OnImagesChanged(trackedImage.referenceImage.name);
    //}
}
// 根据检测到的参考图像名确定需要实例化的虚拟对象
private void OnImagesChanged(ARTrackedImage referenceImage)
{
    if (referenceImage.referenceImage.name == "Spider")
    {
        Instantiate(ObjectPrefabs[0], referenceImage.transform);
    }
    if (referenceImage.referenceImage.name == "Cat")
    {
        Instantiate(ObjectPrefabs[1], referenceImage.transform);
    }
}
}

```

因为 Tracked Image Prefab 属性为空，AR Tracked Image Manager 组件不会实例化任何虚拟对象，所以需要自己负责虚拟对象的实例化。在上述代码中，会在 trackedImagesChanged 事件中捕捉到图像 added 操作，根据参考图像的名字在被检测图像的位置实例化虚拟对象，实现了目的，不再要求用户按顺序扫描图像，但现在还面临一个问题，即不可能使用 if-else 或者 switch-case 语句遍历所有可用的模型，因此应修改为动态加载模型的方式，编写代码如下：

```

// 第5章 /5-5
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation;

public class MultiImageTracking : MonoBehaviour
{
    ARTrackedImageManager mImgTrackedManager;
    private Dictionary<string, GameObject> mPrefabs = new Dictionary<string,
GameObject>(); // 虚拟对象预制体字典

    private void Awake()
    {
        mImgTrackedManager = GetComponent<ARTrackedImageManager>();
    }

    void Start()
    {
        mPrefabs.Add("Cat", Resources.Load("Cat") as GameObject);
        mPrefabs.Add("Spider", Resources.Load("Spider") as GameObject);
    }
    // 注册图像跟踪事件
    private void OnEnable()
    {
        mImgTrackedManager.trackedImagesChanged += OnTrackedImagesChanged;
    }
    // 取消图像跟踪事件注册
    void OnDisable()
    {
        mImgTrackedManager.trackedImagesChanged -= OnTrackedImagesChanged;
    }
    void OnTrackedImagesChanged(ARTrackedImagesChangedEventArgs eventArgs)
    {
        foreach (var trackedImage in eventArgs.added)
        {
            OnImagesChanged(trackedImage);
        }
        //foreach (var trackedImage in eventArgs.updated)
        //{
        //    OnImagesChanged(trackedImage.referenceImage.name);
        //}
    }
    // 根据检测到的参考图像名实例化虚拟对象
    private void OnImagesChanged(ARTrackedImage referenceImage)
    {
        Debug.Log("Image name:" + referenceImage.referenceImage.name);
    }
}

```



```

Instantiate(mPrefabs[referenceImage.referenceImage.name],
referenceImage.transform);
}
}

```

为了实现代码所描述的功能，还要完成两项工作：第一项工作是将虚拟对象预制体放置到 Resources 文件夹中方便动态加载；第二项工作是保证 mPrefabs 字典中的 key 值与 RefImageLib 参考图像库中的参考图像名一致。至此，已实现自由的多图像多模型功能。

但仔细思考，其实还有问题没有解决：①前文提到过，参考图像名在同一个参考图像库中是允许重名的，如果利用参考图像名为字典 key 值就只能有一个虚拟对象可以被使用，开发人员需要确保参考图像名的唯一性；②现在每个参考图像对应一个虚拟对象，这是在开发时就已经确定的，无法在运行时动态地切换虚拟对象。

解决第 1 个问题可以利用每个参考图像的 GUID 值，这个值是在添加参考图像时由算法生成的，可以确保唯一性，并且每个参考图像对应一个 GUID 值，在图像跟踪丢失后再次被跟踪时，这个 GUID 值也不会发生变化，可以利用这个值作为字典的 key 值。

解决第 2 个问题也可以利用被检测图像的 GUID 值，在检测到图像实例化后保存该虚拟对象，然后在需要的时候另外实例化其他虚拟对象并替换原虚拟对象。

以上两个问题解决方案的实现并不复杂，读者可自行实现。

5.4 运行时创建参考图像库

参考图像库可以在开发时创建并设置好，这种方式简单方便，但使用时不太灵活，有时无法满足应用需求，实际上，AR Foundation 支持在运行时动态创建参考图像库。从前面的学习中知道，AR Tracked Image Manager 组件在启动时其参考图像库必须不为 null 值，否则该组件不会启动，因此，在使用图像检测识别跟踪功能时，必须确保在 AR Tracked Image Manager 组件启用之前设置好参考图像库，所以在使用动态图像库时，一般是在需要的时候添加 AR Tracked Image Manager 组件，而不是在开发时预先添加该组件。动态添加 AR Tracked Image Manager 组件及参考图像库的典型代码如下（后文将实际演示使用）：

```

// 第5章 / 5-6
mTrackImageManager = gameObject.AddComponent<ARTrackedImageManager>();
XRReferenceImageLibrary runtimeImageLibrary;
// 创建参考图像库
mTrackImageManager.referenceLibrary = mTrackImageManager.CreateRuntimeLibrary(runtimeImageLibrary);
mTrackImageManager.requestedMaxNumberOfMovingImages = 2;
mTrackImageManager.trackedImagePrefab = mPrefabOnTrack;
mTrackImageManager.enabled = true;

```

在添加完 AR Tracked Image Manager 组件并设置好相应参考图像库及其他属性后，需要显式地设置 mTrackImageManager.enabled = true，启用该组件。

参考图像库可以是 `XRReferenceImageLibrary` 或者 `RuntimeReferenceImageLibrary` 类型, `XRReferenceImageLibrary` 可以在 Unity 编辑器中创建, 但不能在运行时修改, 不过在运行时 `XRReferenceImageLibrary` 会自动转换成 `RuntimeReferenceImageLibrary`。在使用时, 也可以直接从 `XRReferenceImageLibrary` 创建 `RuntimeReferenceImageLibrary`, 典型代码如下:

```
// 第 5 章 / 5-7
XRReferenceImageLibrary serializedLibrary = new XRReferenceImageLibrary();
RuntimeReferenceImageLibrary runtimeLibrary = mTrackImageManager.CreateRuntimeLibrary(serializedLibrary);
```

在从 `XRReferenceImageLibrary` 类型向 `RuntimeReferenceImageLibrary` 类型转换时, 参考图像的顺序并不确定, 即无法预先确定参考图像的次序, 但参考图像名称及其 GUID 值不会发生变化。

5.5 运行时切换参考图像库

在 AR 应用运行时, 也可以动态地切换参考图像库, 下面演示运行时动态切换参考图像库操作。首先制作好两个参考图像库 `ReferenceImageLibrary_1` 和 `ReferenceImageLibrary_2`, 然后使用两个按钮事件在运行时动态地切换所使用的参考图像库。新建一个 C# 脚本, 命名为 `ChangeImageLib`, 编写代码如下:

```
// 第 5 章 / 5-8
using System;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;
using UnityEngine.UI;

public class ChangeImageLib : MonoBehaviour
{
    [SerializeField]
    private Button BtnFirst, BtnSecond; // UI 按钮
    [SerializeField]
    private GameObject mPrefabOnTrack; // 虚拟对象预制体

    [SerializeField]
    XRReferenceImageLibrary[] mReferenceImageLibrary;
    private int currentSelectedLibrary = 0;
    private ARTrackedImageManager mTrackImageManager;
    void Start()
    {
        mTrackImageManager = gameObject.AddComponent<ARTrackedImageManager>();
        mTrackImageManager.referenceLibrary = mTrackImageManager.CreateRuntimeLibrary(mReferenceImageLibrary[0]);
```

```

mTrackImageManager.requestedMaxNumberOfMovingImages = 3;
mTrackImageManager.trackedImagePrefab = mPrefabOnTrack;
mTrackImageManager.enabled = true;
BtnFirst.onClick.AddListener(() => SetReferenceImageLibrary(0));
BtnSecond.onClick.AddListener(() => SetReferenceImageLibrary(1));
Debug.Log(" 初始化完成! ");
}
// 动态切换参考图像库
public void SetReferenceImageLibrary(int selectedLibrary = 0)
{
    mTrackImageManager.referenceLibrary = mTrackImageManager.CreateRuntimeLibrary(mReferenceImageLibrary[selectedLibrary]);
    Debug.Log(String.Format(" 切换参考图像库 {0} 成功! ", selectedLibrary));
}
}

```

在层级窗口中选择 AR Session Origin 对象，将该脚本挂载于此对象上，并设置好相关属性，将创建设置好的 ReferenceImageLibrary_1 和 ReferenceImageLibrary_2 参考图像库赋给 mReferenceImageLibrary 对象，如图 5-7 所示。

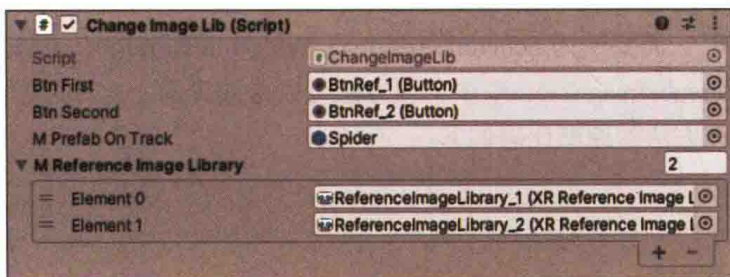


图 5-7 设置参考图像库及其他属性

在本演示中，在 ReferenceImageLibrary_1 和 ReferenceImageLibrary_2 参考图像库中分别添加不同的参考图像，通过切换参考图像库，AR 应用会及时做出反应，对新切换参考图像库中的参考图像进行检测识别，效果如图 5-8 所示。



图 5-8 AR 应用能及时对新切换参考图像库中的图像进行检测识别

5.6 运行时添加参考图像

ARCore 支持在运行时动态地将新的参考图像添加到参考图像库中, 这时 `RuntimeReferenceImageLibrary` 类型实际上就是一个 `MutableRuntimeReferenceImageLibrary` 类型, 在具体使用时, 也需要将 `RuntimeReferenceImageLibrary` 转换为 `MutableRuntimeReferenceImageLibrary` 类型再使用。如果需要在运行时创建新的参考图像库, 则可以使用无参的 `CreateRuntimeLibrary()` 方法, 然后将其转换为 `MutableRuntimeReferenceImageLibrary` 使用, 典型代码如下:

```
// 第 5 章 / 5-9
var library = trackedImageManager.CreateRuntimeLibrary();
if (library is MutableRuntimeReferenceImageLibrary mutableLibrary)
{
    // 添加图像到参考图像库
}
```

因为需要提取参考图像的特征值信息, 运行时动态添加参考图像是一个计算密集型任务, 这大概需要花费几十毫秒时间, 需要很多帧才能完成添加, 所以为了防止同步操作造成应用卡顿, 可以利用 Unity Job 系统异步处理这种操作。

添加参考图像时, 需要使用 `ScheduleAddImageWithValidationJob()` 方法将参考图像添加到 `MutableRuntimeReferenceImageLibrary` 中, 该方法可以是同步的, 也可以是异步的, 取决于开发者的使用方式, 该方法的原型代码如下:

```
// 第 5 章 / 5-10
public static AddReferenceImageJobState ScheduleAddImageWithValidationJob
(
    this MutableRuntimeReferenceImageLibrary library, // 参考图像库
    Texture2D texture, // 需要添加的参考图像
    string name, // 参考图像名称
    float? widthInMeters, // 待检测的图像物理尺寸宽 (单位为米)
    JobHandle inputDeps = default // 输入描述信息
);
```

可以通过该方法将多张参考图像添加到参考图像库, 即使当前的参考图像库正在跟踪图像及处于使用中也没关系。

动态添加参考图像对图像有一些特定要求, 第一要求参考图像可读写, 因为添加图像时需要提取图像特征值信息; 第二要求图像格式必须为应用平台支持的格式, 通常选择 RGB24 或者 RGBA32 格式。这个需要在图像的导入设置 (import setting) 中设置^①, 如图 5-9 所示^②。

① 在 Unity 工程窗口中选择作为参考图像的图像, 然后在属性窗口中会打开导入设置面板, 展开 Advanced 卷展栏, 进行高级设置。

② 如果作为动态添加的参考图像没有启用 Read/Write Enable 功能, 编译时将提示: The texture must be readable to be used as the source for a reference image, 如果所选定的格式平台不支持, 编译时将提示: The texture format ETC_RGB4 is not supported by the current image tracking subsystem。

经过测试，动态添加参考图像对图像编码格式也有要求，通常只支持 JPG、PNG 格式，但一些 JPG、PNG 类型图像的编码格式不支持，如 ETC_RGB4、Alpha8 等，需要仔细挑选作为参考图像的图像文件。



图 5-9 设置图片相应属性

下面演示在运行时动态添加参考图像，新建一个 C# 脚本，命名为 DynamicImageTracking，并编写代码如下：

```
// 第5章 /5-11
using Unity.Collections;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;
using UnityEngine.UI;
using Unity.Jobs;

public class DynamicImageTrackingAsync : MonoBehaviour
{
    // 用于异步图像操作的结构体
    struct DeallocateJob : IJob
    {
        [DeallocateOnJobCompletion]
        public NativeArray<Byte> data;
        public void Execute() { }
    }

    [SerializeField]
    private Button BtnAddImage; // UI 按钮

    [SerializeField]
    private GameObject mPrefabOnTrack; // 虚拟对象预制体
    private Vector2 scaleFactor = new Vector3(0.3f, 0.3f); // 参考图像尺寸
    private XRReferenceImageLibrary runtimeImageLibrary; // 运行时参考图像库
    private ARTrackedImageManager mTrackImageManager;

    [SerializeField]
```

```

private Texture2D mAddedImage;

void Start()
{
    mTrackImageManager = gameObject.AddComponent<ARTrackedImageManager>();
    // 创建参考图像库
    mTrackImageManager.referenceLibrary = mTrackImageManager.CreateRuntimeLibrary(runtimeImageLibrary);
    mTrackImageManager.requestedMaxNumberOfMovingImages = 2;
    mTrackImageManager.trackedImagePrefab = mPrefabOnTrack;
    mTrackImageManager.enabled = true;
    mTrackImageManager.trackedImagesChanged += OnTrackedImagesChanged;
    BtnAddImage.onClick.AddListener(() => AddImageJobAsync(mAddedImage));
}
// 取消图像检测事件注册
void OnDisable()
{
    mTrackImageManager.trackedImagesChanged -= OnTrackedImagesChanged;
}
// 添加参考图像同步
public void AddImageJob(Texture2D texture2D)
{
    try
    {
        MutableRuntimeReferenceImageLibrary mutableRuntimeReferenceImageLibrary
        = mTrackImageManager.referenceLibrary as MutableRuntimeReferenceImageLibrary;
        mutableRuntimeReferenceImageLibrary.ScheduleAddImageWithValidationJob(
            texture2D,          // 参考图像
            "Spider",          // 参考图像名
            scaleFactor.x);    // 参考图像尺寸宽
    }
    catch (System.Exception e)
    {
        Debug.Log(" 出现错误: "+e.Message);
    }
}

// 异步添加参考图像
void AddImageJobAsync(Texture2D refImage)
{
    Byte[] colorBuffer = refImage.GetRawTextureData();
    NativeArray<Byte> image = new NativeArray<Byte>(colorBuffer, Allocator.TempJob);
    if (mTrackImageManager.referenceLibrary is MutableRuntimeReferenceImageLibrary mutableLibrary)

```

```

{
    var referenceImage = new XRReferenceImage(
        SerializableGuid.empty,           //GUID 值
        SerializableGuid.empty,          // 默认纹理
        scaleFactor,                      // 物理尺寸设置
        "Spider",                         // 参考图像名称
        null);                            // 参考图像, 这里先不设置

    var jobState = mutableLibrary.ScheduleAddImageWithValidationJob(
        image,
        new Vector2Int(refImage.width, refImage.height),
        TextureFormat.ARGB32,            // 设置图像格式
        referenceImage);

    // 启动一个任务, 在任务结束后销毁相关资源
    new DeallocateJob { data = image }.Schedule(jobState.jobHandle);
}
else
{
    // 销毁图像资源
    image.Dispose();
}
}

// 图像检测事件
void OnTrackedImagesChanged(ARTrackedImagesChangedEventArgs eventArgs)
{
    foreach (ARTrackedImage trackedImage in eventArgs.added)
    {
        Debug.Log("检测到图像:" + trackedImage.name);
        trackedImage.transform.Rotate(Vector3.up, 180);
    }
    foreach (ARTrackedImage trackedImage in eventArgs.updated)
    {
        trackedImage.transform.Rotate(Vector3.up, 180);
    }
}
}
}

```

上述代码演示了同步和异步两种添加参考图像的方式, 在层级窗口中选择 AR Session Origin 对象, 将该脚本挂载在此对象上, 并设置好相关属性, 即可在运行时动态地添加参考图像。动态切换参考图像库与动态添加参考图像是非常实用的功能, 可以根据不同的应用场景切换到不同的参考图像库, 或者添加新的参考图像而无须重新编译或者中断应用。

5.7 脱卡

从前文中已经知道,在使用 ARCore 进行二维图像检测跟踪时,当检测识别到参考图像时,会实例化虚拟对象并叠加到物理图像之上,并能实时跟踪物理图像的移动、旋转,当物理图像移出视野范围(屏幕)后,实例化的虚拟对象也会跟随之移出视野(或者随二维图像移动到屏幕边缘外),这在很多应用场景下是合适的。

但有时,也希望当物理图像移出视野后,虚拟对象能停留在屏幕中心,并能与用户正常进行交互,例如在博物馆门口通过检测识别特定图像后实例化的虚拟导游,我们希望该虚拟导游能伴随整个游览过程,进行不间断的语音讲解。当物理图像消失后,实例化后的虚拟模型不随之消失,这个功能称为图像检测识别脱卡。

脱卡可以有很多种实现方式,例如当物理图像消失后,启用另一个渲染相机渲染实例化后的模型;例如通过当前渲染相机的前向向量在设备屏幕前的一定距离放置实例化模型等。

但在进行这个操作之前,需要定义物理图像消失事件,即什么情况可认为物理图像消失。AR Tracked Image Manager 组件的 `trackedImagesChanged` 事件会提供这个检测识别结果的状态变化,结合前文知识,只需检查跟踪图像的状态值,当跟踪状态从 `Tracking` 变更到 `Limited` 时,就可以认为物理图像消失事件已经发生,但 AR Foundation 指出,有多种未知情况可能会导致图像检测跟踪状态从 `Tracking` 变更到 `Limited`,所以以此作为判断依据并不严谨。

但考虑到二维图像检测识别后,通常会实例化虚拟对象,虚拟对象会追踪二维图像的位置,当物理图像移出视野后,虚拟对象也会移出视野,利用这个特性,可以通过虚拟对象的可见性判断物理图像是否移出视野。

典型代码如下:

```
// 第 5 章 / 5-12
using UnityEngine;

public class ObjectTrackLost : MonoBehaviour
{
    private bool isTrackLost = false;    // 可见与不可见标识
    private Camera mCamera;              // 主相机
    private float distance = 0.5f;       // 放置在主相机前方的距离
    void Start()
    {
        mCamera = Camera.main;
    }
    // 当虚拟对象可见时触发
    public void OnBecameVisible()
    {
        Debug.Log("跟踪正常");
        //isTrackLost = false;
    }
}
```

```
// 当虚拟对象不可见时触发
public void OnBecameInvisible()
{
    Debug.Log("跟踪丢失");
    isTrackLost = true;
}

void Update()
{
    if(isTrackLost)
    {
        // 将虚拟对象放置于主相机前指定的距离
        gameObject.transform.position = mCamera.transform.position +
mCamera.transform.forward.normalized * distance;
        gameObject.transform.rotation = mCamera.transform.rotation;
    }
}

// 当二维图像被跟踪时由外部调用，终止脱卡
public void SetVisible()
{
    isTrackLost = false;
}
}
```

在上述代码中，利用 `OnBecameInvisible()` 方法判断虚拟对象的可见性，然后在虚拟对象不可见（物理图像被移出）时，将虚拟对象放置在主相机前方指定的距离。该代码需要挂载于虚拟对象上，并且虚拟对象必须有 `Renderer` 组件（对象一定要可被渲染）。需要注意的是，不能再使用 `OnBecameVisible()` 方法恢复图像跟踪，而必须由外部代码在图像被再次跟踪时调用 `SetVisible()` 方法恢复跟踪。

5.8 图像跟踪优化

在二维图像被检测到之后，`ARCore` 会跟踪该图像的姿态（位置与方向），因此，可以实现虚拟对象与二维图像绑定的效果（虚拟元素的态度会随二维图像的态度发生变化），如在一张别墅的图片上加载一个虚拟的别墅模型，旋转、移动该别墅图片，虚拟别墅模型也会跟着旋转或者移动，就像虚拟模型粘贴在图片上一样。

图像检测跟踪效果与很多因素相关，为了更好地在应用中使用二维图像检测跟踪，提高用户体验，应当注意以下事项。

1. 有关参考图像

参考图像应当具有丰富纹理、高对比度、纹理不重复等特征，特征丰富的参考图像有利于 `ARCore` 进行图像检测，其他注意事项如表 5-5 所示。

表 5-5 参考图像一般注意事项

序号	描 述
1	参考图像支持 PNG 和 JPG 文件格式, 对于 JPG 文件, 为了获得最佳性能, 需避免过度压缩
2	参考图像特征提取仅基于高对比度的点, 所以彩色和黑白图像都会被检测到, 对物理图像颜色不敏感
3	参考图像的分辨率至少应为 300 × 300 像素
4	使用高分辨率的参考图像不会提升性能
5	避免使用特征稀疏、无纹理的参考图像
6	避免使用具有重复特征、重复纹理的参考图像

在参考图像的选择上, 肉眼很难分辨是否是高质量的参考图像, 因此 ARCore 提供了 ARCoreIMG 工具, 使用 ARCoreIMG 工具可以获取每个参考图像的得分情况, 取值范围为 [0,100], 建议参考图像的得分至少为 75 分。

ARCoreIMG 是一个命令行工具, 它不仅可以获取参考图像得分, 还可以从一组参考图像生成参考图像库文件, 此工具主要用于检查参考图像的质量。该工具支持 Windows、Linux 系统, 使用命令行操作。在 Windows 下检查参考图像质量的命令示例如下^①:

```
arcoreimg eval-img -input_image_path=4.jpg
```

ARCoreIMG 工具执行命令后会返回相关结果, 常见结果如表 5-6 所示。

表 5-6 ARCoreIMG 返回结果及原因

返回结果	原 因
返回数值 (如 55)	参考图像评估得分, 参考值为 [0, 100], 得分越高, 检测识别成功率越高, 建议选择分值为 75 分以上图像作为参考图像
input_image_path is not specified!	命令行中需要添加 input_image_path 参数
Unsupported image format.	格式不支持, 只支持 JPG、PNG 两种图像格式, 并且与具体图像编码格式有关, 有时即使为 JPG 格式, 也会报这个错误
Failed to get enough keypoints from target image.	图像纹理不丰富, 无法提取足够的特征值, 建议更换参考图像

参考图像的选择有一定的技巧, 包含高度重复特征的图像得分会很低, 因此建议采用细节丰富、纹理不重复的图像作为参考图像。

2. 有关参考图像库创建

ARCore 参考图像库会存储参考图像的特征值信息, 每张参考图像会占据大约 6KB 空间。在运行时添加参考图像, 将一张参考图像添加到参考图像库大约需要 30ms, 因此在运行时添

^① 官方给出的检查参考图像质量命令示例为 `arcoreimg eval-img dog.png`, 经过测试, 使用该命令会提示如下错误: `input_image_path is not specified`, 正确的命令应该加上 `input_image_path` 参数。

加参考图像需要使用异步操作。另外，不要在参考图像库中保存不使用的参考图像，因为这会对应用性能产生一定的影响。

3. 有关跟踪优化

在 ARCore 初次识别图像时，物理图像大约需要占据设备摄像头图像面积的 40% 或以上，需要及时提示用户将物理图像放置在摄像头取景范围内并保持合适大小。

一般而言，为了更好地优化二维图像检测跟踪，需要从参考图像的选择、参考图像库的设计、整体设计方面进行考虑。

(1) 尽量指定待检测图像的预期物理尺寸，此元数据可以提升识别跟踪性能，特别是对于较大的物理图像（长和宽超过 75cm），ARCore 会使用这些物理尺寸评估二维图像到用户设备的距离，不正确的物理尺寸会影响检测跟踪精度，从而影响加载的虚拟模型与二维图像的贴合度。

(2) 物理图像尽量展平，卷曲的图像，如包裹酒瓶的海报非常不利于 ARCore 检测或者导致检测出的位姿不正确。

(3) 确保需要检测的物理图像照明条件良好，光线昏暗或者反光（如玻璃橱窗里的海报）会影响 ARCore 的检测。

(4) 通常情况下，每个参考图像库的参考图像数量不应该超过 100 个，如果数量过多，则会影响检测准确性和检测速度。一般情况下，可以将大型的参考图像库拆分为小的参考图像库，然后根据 AR 应用运行时的条件动态切换参考图像库。

第 6 章



人脸检测跟踪

在计算机人工智能（Artificial Intelligence，AI）物体检测识别领域，研究得最多的是人脸检测识别，技术发展最成熟的也是人脸检测识别。目前人脸检测识别已经广泛应用于安防、机场及车站闸机、人流控制、安全支付等众多社会领域，也广泛应用于直播特效、美颜、Animoji 等娱乐领域，本章主要学习 ARCore 人脸检测跟踪相关知识。

6.1 人脸检测基础

ARCore 支持人脸检测跟踪，并且支持实时多人脸检测（最多 3 人），除此之外，还支持人脸区域划分，ARCore 人脸检测跟踪只使用设备摄像头采集的图像信息，不依赖其他传感器（如前置 ToF、结构光等），在支持 ARCore 的所有机型上都支持人脸检测跟踪。

利用 ARCore 提供的人脸检测跟踪能力，可以非常方便地实现诸如贴纸、挂载模型等特效，在很多领域都有应用价值。

6.1.1 人脸检测概念

人脸检测（Face Detection）是指利用计算机图像处理技术在数字图像或视频中自动定位人脸的过程，人脸检测不仅检测人脸在图像或视频中的位置，还应该检测出其大小与方向（姿态）。人脸检测是有关人脸图像分析应用的基础，包括人脸识别和验证、监控场合的人脸跟踪、面部表情分析、面部属性识别（性别、年龄、微笑、痛苦）、面部光照调整和变形、面部形状重建、图像视频检索等。近年来，随着深度学习技术的发展，人脸检测成功率和准确性大幅度提高，已经开始大规模实用，如机场及火车站人脸验票、人脸识别身份认证等。

人脸识别（Face Recognition）是指利用人脸检测技术确定两张人脸是否对应同一个人，人脸识别技术是人脸检测技术的拓展和应用，也是很多其他应用的基础。目前，ARCore 仅支持人脸检测，而不提供人脸识别功能。

人脸跟踪（Face Tracking）是指将人脸检测扩展到视频序列，跟踪同一张人脸在视频序列中的位置。理论上讲，任何出现在视频中的人脸都可以被跟踪，也就是说，在连续视频帧中

检测到的人脸可以被识别为同一个人。人脸跟踪不是人脸识别的一种形式，它是根据视频序列中人脸的位置和运动推断不同视频帧中的人脸是否为同一人的技术。

人脸检测属于模式识别的一类，但人脸检测成功率受到很多因素的影响。影响人脸检测成功率的因素主要有表 6-1 中所述情形。

表 6-1 影响人脸检测成功率的部分因素

术 语	描述说明
图像大小	人脸图像过小会影响检测效果，人脸图像过大会影响检测速度，图像大小反映在实际应用场景中就是人脸离摄像头的距离
图像分辨率	越低的图像分辨率越难检测，图像大小与图像分辨率直接影响摄像头识别距离。目前 4K 相机看清人脸的最远距离是 10m 左右，移动手机检测距离更小一些
光照环境	过亮或过暗的光照环境都会影响人脸检测效果
模糊程度	实际场景中主要是运动模糊，人脸相对于相机的移动经常会产生运动模糊
遮挡程度	五官无遮挡、脸部边缘清晰的图像有利于人脸检测。有遮挡的人脸会对人脸检测成功率造成影响
采集角度	人脸相对于相机角度不同也会影响人脸检测效果。正脸最有利于检测，偏离角度越大越不利于检测

随着人工智能技术的持续发展，在全球信息化、云计算、大数据的支持下，人脸检测识别技术也会越来越成熟，同时应用面会越来越广，可以预见，以人脸检测为基础的人脸识别将会呈现网络化、多识别融合、云互联的发展趋势。

6.1.2 人脸检测技术基础

人体头部是一个三维结构体，通常采用欧拉角来精确描述头部的姿态，本节描述的欧拉角源自于笛卡儿左手坐标系，并规定绕 Y 轴逆时针旋转角度为正，绕 Z 轴顺时针旋转角度为正，如图 6-1 所示。在人脸检测中，通常把绕 Y 轴旋转叫作 y 欧拉角，绕 Z 轴旋转叫作 r 欧拉角，平时做的摇头动作就是 y 欧拉角，而偏头的动作则是 r 欧拉角，如图 6-2 所示，绕 X 轴旋转在人脸检测中通常很少用到。

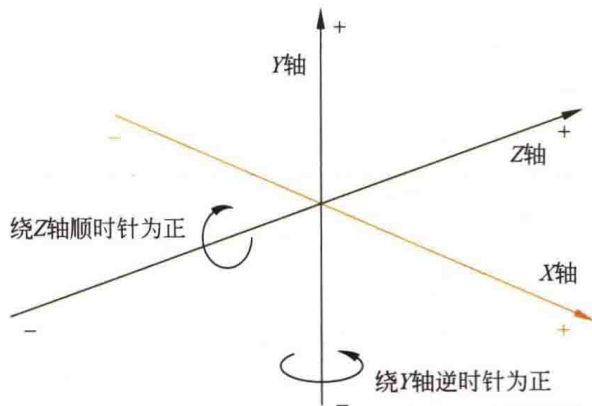


图 6-1 基于笛卡儿坐标系的欧拉角

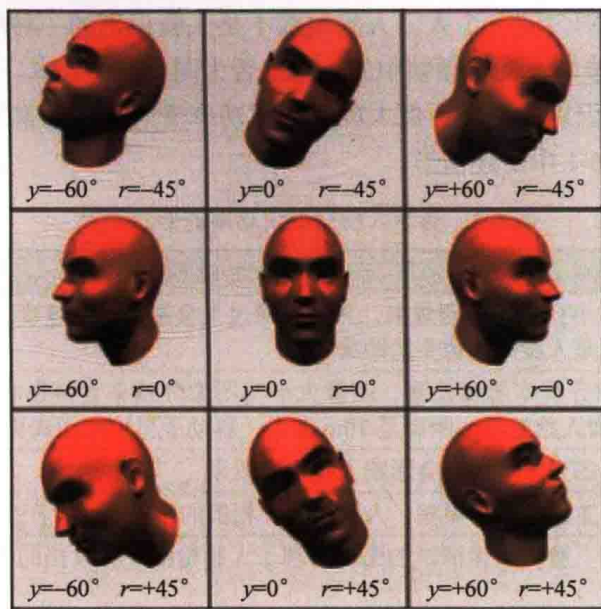


图 6-2 人脸检测中的欧拉角示意图

人脸检测技术的复杂性之一源于人体头部结构的三维性，而且相机捕捉到的人脸很多时候都不是正面，而是有一定角度且时时处于变化中。当然，人脸检测的有利条件是人脸有很多特征，如图 6-3 所示，可以利用这些特征做模式匹配，但需要注意的是，在很多人脸检测算法中，人脸特征并不是人脸轮廓检测的前提，换句话说，人脸检测是独立于人脸特征的，并且通常是先检测出人脸轮廓再进行特征检测，因为特征检测需要花费额外的时间，会对人脸检测效率产生影响。



图 6-3 人脸特征点示意图

人脸结构具有对称性，人脸特征会分布在 Y 轴两侧一定角度内，通常来讲，人脸特征分布情况符合表 6-2 所示规律。

表 6-2 人脸特征分布情况

y 欧拉角	人脸特征
小于 -36°	左眼、左嘴角、左耳、鼻底、左脸颊
$-36^\circ \sim -12^\circ$	左眼、左嘴角、鼻底、下嘴唇、左脸颊
$-12^\circ \sim 12^\circ$	左嘴角、右嘴角、上下嘴唇、鼻底
$12^\circ \sim 36^\circ$	右眼、右嘴角、鼻底、下嘴唇、右脸颊
大于 36°	右眼、右嘴角、右耳、鼻底、右脸颊

人脸检测不仅需要找出人脸轮廓,还需要检测出人脸姿态,包括人脸位置和面向方向。为了解决人脸姿态问题,一般的做法是制作一个三维人脸正面“标准模型”,这个模型需要非常精细,因为它将影响人脸姿态估计的精度。在有了这个三维标准模型之后,对人脸姿态检测的思路是在检测到人脸轮廓后对标准模型进行旋转,以期标准模型上的特征点与检测到的人脸特征点重合^①。从这个思路可以看到,对人脸姿态检测其实是个不断尝试的过程,选取特征点吻合得最好的标准模型姿态作为人脸姿态。

为快速检测人脸姿态,另一种常见做法是预先定义一个6关键点的三维脸部模型(左眼角、右眼角、鼻尖、左嘴角、右嘴角、下颌),如图6-4所示,然后使用计算机图像算法检测出视频帧中人脸对应的6个脸部关键点,并将这6个关键点与标准模型中对应的关键点进行拟合,解算出旋转向量,最后将旋转向量转换为欧拉角。

算法流程在理论上如此,但在实际应用中,可能并不能同时检测到人脸的6个关键点(由于遮挡、视觉方向、光照等原因),因此人脸姿态估计通常还需要进行信息推测以提高稳健性。

如前所述,虽然人脸结构是确定的,还有很多特征点可供校准,但由于姿态和表情的变化、不同人的外观差异、光照、遮挡等因素影响,准确地检测处于各种条件下的人脸仍然是较为困难的。幸运的是,随着深度神经网络技术的发展,在一般环境条件下,目前人脸检测准确率有了非常大的提高,甚至在某些条件下超过了人类。

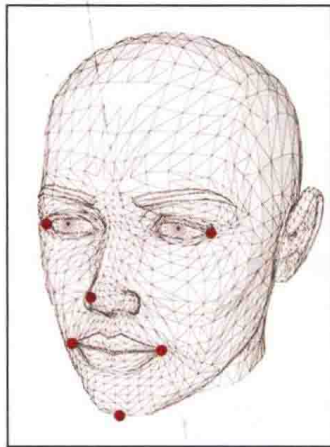


图 6-4 6 特征点解算人脸姿态示意图

6.2 人脸姿态与网格

在计算机视觉算法中,头部姿势估计是指推断头部朝向,结合 AR 位置信息构建人脸矩阵参数的过程,有利因素是人体头部运动范围有限,可以借此消除一些误差。除 6.1.2 节所述“标准模型”、6 特征点法以外,人脸姿态估计方法还有很多,如柔性模型法、非线性回归法、嵌入法等。

6.2.1 人脸姿态

人脸姿态估计主要是为了获得人脸朝向的角度信息,虽然算法本身很复杂,但在 AR Foundation 中,完全可以不用了解这些底层细节,AR Face Manager 组件已经进行了良好的封装,提供了非常简洁的使用界面。AR Face Manager 组件如图 6-5 所示。

^① 不同的人脸检测算法会有不同的实现方式。

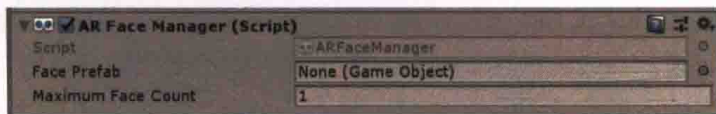


图 6-5 AR Face Manager 组件

AR Face Manager 组件通过底层 Provider (算法提供者) 提供的算法检测人脸, 但不同的底层对人脸检测的要求也不一样, 一般使用前置摄像头, ARCore 在使用前置摄像头时会禁用后置摄像头, 此时所有平面检测、运行跟踪、环境理解、图像跟踪等功能都会失效, 为了提高性能, 在使用人脸检测时最好禁用这些功能。

与其他可跟踪对象管理器一样, AR Face Manager 组件在检测到人脸时会使用人脸的姿态实例化一个预制体, 即图 6-5 中的 Face Prefab 属性指定的预制体, Face Prefab 属性可以置空, 但如果设置, 则在实例化预制体时, AR Face Manager 组件会负责为该预制体挂载一个 ARFace 组件, ARFace 组件包含了检测到的人脸相关信息。Maximum Face Count 属性用于设置在场景中同时检测的人脸数量, ARCore 支持多人脸检测, 但设置较小的值有助于性能优化。

由于当前并不是所有的设备都支持人脸检测功能, AR Face Manager 组件提供了一个 supported 属性, 用于检查用户设备是否支持人脸检测, 可以通过这个属性查看设备支持情况。AR Face Manager 组件还提供了 facesChanged 事件, 可以通过注册事件获取新检测、更新、移除人脸时的消息, 进行个性化处理。AR Face Manager 组件为每个检测到的人脸生成一个唯一的 TrackableId, 利用这个 TrackableId 值可以跟踪每张独立的人脸^①。

另外, 在人脸检测及后续过程中, AR Foundation 会自动进行环境光检测和估计, 不需要开发人员手动编写相关代码。

在 AR Foundation 中, AR Face Manager 组件会自动完成人脸姿态检测, 可以通过以下方法验证人脸姿态检测效果。首先制作一个交点在原点的三轴正交坐标系预制体, 如图 6-6 所示。

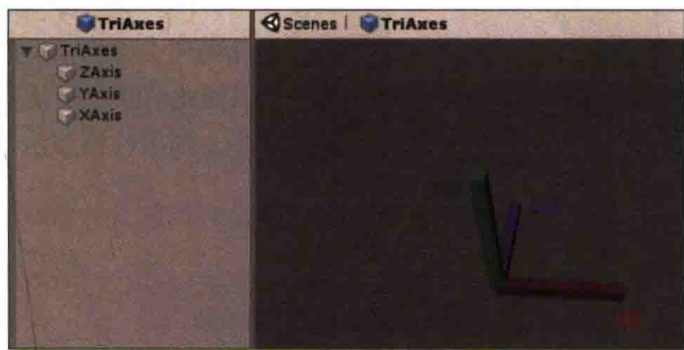


图 6-6 建立三轴正交坐标系预制体

在层级窗口 (Hierarchy 窗口) 中选择 AR Session Origin 对象, 为其挂载 AR Face Manager 组件, 将 Maximum Face Count 属性设置为 1, 并将制作好的三轴坐标系预制体赋给 Face

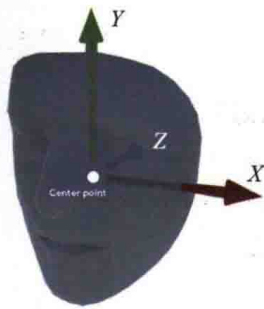
^① 在同一个会话中, 同一张人脸的 TrackableId 会保持不变, 即使该人脸检测后丢失, 被再次检测到时也一样。

Prefab 属性。由于人脸检测需要使用前置摄像头,在使用人脸检测功能时,场景中 AR Camera 对象挂载的 AR Camera Manager 组件上的 Facing Direction 属性需要选择为 User 值。

编译运行,将前置摄像头对准人脸,效果如图 6-7 (a) 所示。



(a)



(b)

图 6-7 人脸姿态检测示意图

从图 6-7 中可以看出, ARCore 检测到的人脸坐标系原点位于鼻尖靠后的位置,约在头颅中心,如图 6-7 (b) 所示,其 Y 轴向上, Z 轴向里, X 轴向左。对比图 6-6 和图 6-7 (a) 两张图 [或者对比图 6-7 (a) 和图 6-7 (b) 两张图], 可以看到, X 轴朝向相反,这是因为使用的是前置摄像头,这种反转 X 轴朝向的处理在后续人脸上挂载虚拟物体时可以保持虚拟物体正确的三轴朝向。

6.2.2 人脸网格

除了检测人脸姿态, AR Foundation 还为每张检测到的人脸提供了一个人脸网格,该网格可以覆盖检测到的人脸形状,这张网格数据也是由底层 SDK 提供的,包括顶点、索引、法线、纹理坐标等相关信息,如图 6-8 所示。ARCore 人脸网格包含 468 个顶点,对普通消费级应用来讲已经足够了。利用人脸网格数据,开发者只需通过网格点的坐标和特定区域的锚点就可以轻松地人脸附加一些特效,如面具、眼镜、虚拟帽子,或者对人脸网格进行扭曲以实现特定效果等。

在 AR Foundation 中,人脸网格相关数据信息由 ARFace 组件提供,而人脸网格的可视化则由 ARFaceMeshVisualizer 组件实现,该组件会根据 ARFace 组件提供的网格数据更新显示网格信息。在检测到人脸时显示人脸网格与显示检测到的平面非常类似,需要预先制作一个人脸预制件,区别是这个预制件中不需要三维模型,ARFace 组件会提供相应的网格数据,网格信息采集计算与网格可视化都由专门的组件负责实现。

下面演示人脸网格的基本使用方法,虽然 ARFace 组件提供了与人脸相关的所有数据,但其本身并不负责人脸网格渲染,所以需要使用纹理(直接渲染人脸网格数据也可以)对其进

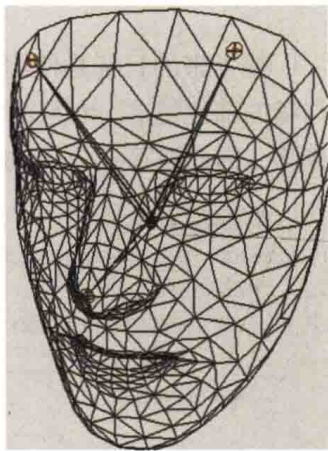


图 6-8 ARCore 人脸网格示意图

行渲染。在层级窗口 (Hierarchy 窗口) 中新建一个空对象, 命名为 FaceMesh, 并在其上挂载 ARFace 与 ARFaceMeshVisualizer 组件, 同时挂载渲染组件 Mesh Renderer 和 Mesh Filter, 如图 6-9 所示^①。

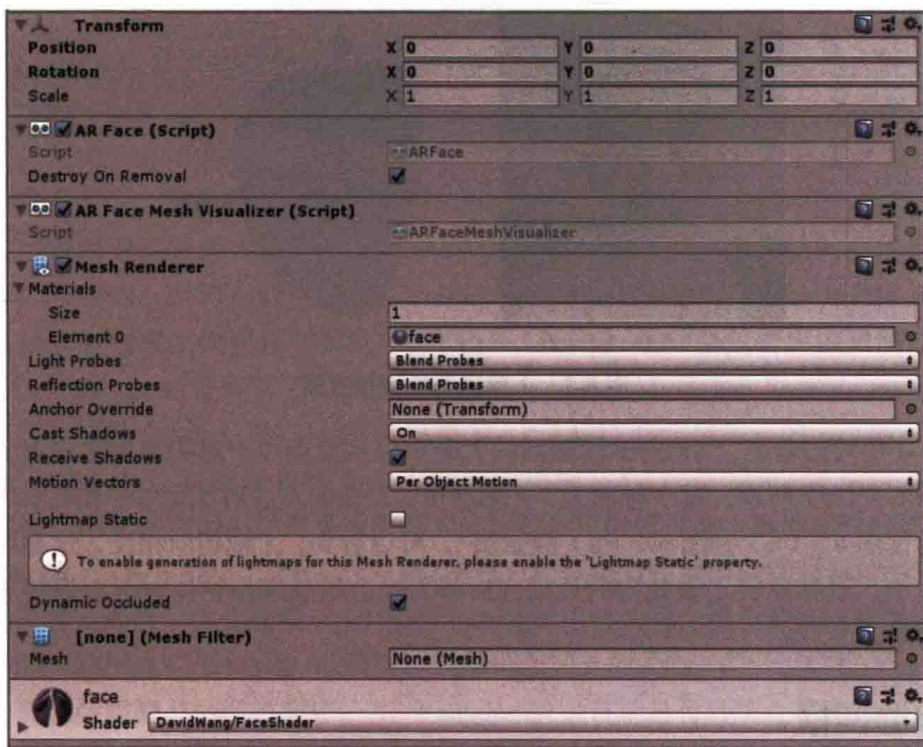


图 6-9 制作人脸网格预制体

接下来的目标是使用京剧脸谱纹理渲染网格, 为将眼睛与嘴部露出来, 需要提前将眼睛与嘴的部位镂空, 并将图像保存为带 Alpha 通道的 PNG 格式^②, 并且在渲染时需要将镂空部分剔除, 因此编写一个 Shader 脚本, 代码如下:

```
// 第 6 章 / 6-1
Shader "DavidWang/FaceShader" {
    Properties{
        _MainTex("Texture", 2D) = "white" {}
    }
    SubShader
    {
        Tags { "RenderType" = "Opaque" "RenderPipeline" = "UniversalPipeline" }

        Pass
```

① 事实上, 也可以在层级窗口中通过鼠标在空白处右击, 在弹出的菜单中依次选择 XR → AR Default Face 创建这个人脸对象, 由于使用的是 URP 管线, 所以显示可能不正确, 需要进行材质修正。

② 可以使用 PhotoShop 等图像处理软件进行预先处理。

```

{
    HLSLPROGRAM
    #pragma vertex vert
    #pragma fragment frag

    #include "Packages/com.unity.render-pipelines.universal/
ShaderLibrary/Core.hlsl"
    struct Attributes
    {
        float4 positionOS    : POSITION;
        float2 uv            : TEXCOORD0;
    };

    struct Varyings
    {
        float4 positionHCS   : SV_POSITION;
        float2 uv            : TEXCOORD0;
    };

    TEXTURE2D(_MainTex);
    SAMPLER(sampler_MainTex);

    CBUFFER_START(UnityPerMaterial)
    float4 _MainTex_ST;
    CBUFFER_END

    Varyings vert(Attributes IN)
    {
        Varyings OUT;
        OUT.positionHCS = TransformObjectToHClip(IN.positionOS.xyz);
        OUT.uv = TRANSFORM_TEX(IN.uv, _MainTex);
        return OUT;
    }

    half4 frag(Varyings IN) : SV_Target
    {
        half4 color = SAMPLE_TEXTURE2D(_MainTex, sampler_MainTex, IN.uv) ;
        clip(color.w - 0.1);
        return color;
    }
    ENDHLSL
}
}

```

本书采用 URP 渲染管线编写，为了兼容传统 Build-in 管线，这里同时提供传统管线的 Shader 脚本，代码如下：


```
// 第 6 章 / 6-2
Shader "DavidWang/FaceShader" {
    Properties{
        _MainTex("Texture", 2D) = "white" {}
    }
    SubShader{
        Tags{ "Queue" = "AlphaTest" "IgnoreProjector" = "True" "RenderType" =
"TransparentCutOut" }

        Cull Off
        CGPROGRAM
            #pragma surface surf Standard vertex:vert fullforwardshadows
            #pragma target 3.0

            sampler2D _MainTex;

            struct Input {
                float2 uv_MainTex;
                float y : TEXCOORD0;
            };

            void vert(inout appdata_full v,out Input o) {
                o.uv_MainTex = v.texcoord.xy;
                o.y = v.vertex.y;
            }

            void surf(Input IN, inout SurfaceOutputStandard o) {
                fixed4 c = tex2D(_MainTex, IN.uv_MainTex);
                o.Alpha = c.w;
                clip(o.Alpha - 0.1);
                o.Albedo = c.rgb;
            }
            ENDCG
        }
        FallBack "Diffuse"
    }
}
```

在 Shader 脚本代码中,剔除透明区域使用了 clip(x) 函数,该函数的功能是丢弃参数 x 值小于 0 的像素,即镂空纹理中特定区域。

在工程窗口 (Project 窗口) 中新建一个使用该 Shader 的材质,命名为 face,为该材质赋上京剧脸谱纹理,并将该材质应用到 Mesh Renderer 中,如图 6-9 所示,然后将场景中 FaceMesh 对象制作成预制体,并删除层级窗口中的 FaceMesh 对象,最后将 FaceMesh 预制体赋给层级窗口中 AR Session Origin 对象上 AR Face Manager 组件的 Face Prefab 属性。编译运行,

将前置摄像头对准人脸，效果如图 6-10 所示^①。

AR Face Manager 组件只有一个 Face Prefab 属性，因此只能实例化一类人脸网格渲染，如果要实现川剧中的变脸效果该如何处理呢？

前文提到，AR Face Manager 组件有一个 facesChanged 事件，因此可以通过注册这个事件实现期望的功能，思路如下：

实现变脸实际上就是要更换渲染的纹理，因此只需要在 AR Face Manager 组件检测到人脸的 added 事件，在其实例化人脸网格之前替换掉相应的纹理。为此，预先制作多张类似的纹理，并放置在 Resources 文件夹中以便运行时加载。新建一个 C# 脚本，命名为 ChangeFace，编写代码如下：



图 6-10 人脸面具效果图

```
// 第 6 章 / 6-3
using UnityEngine;
using UnityEngine.XR.ARFoundation;

[RequireComponent(typeof(ARFaceManager))]
public class ChangeFace : MonoBehaviour
{
    private GameObject mFacePrefab;
    private ARFaceManager mARFaceManager;
    private Material _material; // 材质对象
    private int mTextureIndex = 0; // 纹理索引号
    private int mTextureNumber = 3; // 纹理总数
    private Texture2D[] mTextures = new Texture2D[3]; // 纹理数组

    private void Awake()
    {
        mARFaceManager = GetComponent<ARFaceManager>();
    }
    // 加载纹理资源
    void Start()
    {
        mFacePrefab = mARFaceManager.facePrefab;
        _material = mFacePrefab.transform.GetComponent<MeshRenderer>().material;
        mTextures[0] = (Texture2D)Resources.Load("face0");
    }
}
```

① 人脸检测功能与平面检测、射线检测等功能不兼容，而且平面检测功能的优先级比人脸检测高，如果 AR Session Origin 对象上同时挂载了 AR Plane Manager 和 AR Face Manager 组件，则默认会执行平面检测功能，人脸检测功能将不可用。另外，由于人脸检测需要使用前置摄像头，在使用人脸检测功能时，场景中 AR Camera 对象上的 AR Camera Manager 组件上的 Facing Direction 属性需要选择为 User。

```

        mTextures[1] = (Texture2D)Resources.Load("face1");
        mTextures[2] = (Texture2D)Resources.Load("face2");
    }
    // 注册事件
    private void OnEnable()
    {
        mARFaceManager.facesChanged += OnFacesChanged;
    }
    // 取消事件注册
    void OnDisable()
    {
        mARFaceManager.facesChanged -= OnFacesChanged;
    }
    void OnFacesChanged(ARFacesChangedEventArgs eventArgs)
    {
        foreach (var trackedFace in eventArgs.added)
        {
            OnFaceAdded(trackedFace);
        }
        /*
        foreach (var trackedFace in eventArgs.updated)
        {
            OnFaceUpdated(trackedFace);
        }
        foreach (var trackedFace in eventArgs.removed)
        {
            OnFaceRemoved(trackedFace);
        }
        */
    }
    // 处理纹理替换
    private void OnFaceAdded(ARFace refFace)
    {
        mTextureIndex = (++mTextureIndex) % mTextureNumber;
        _material.mainTexture = mTextures[mTextureIndex];
        //Debug.Log("TextureIndex:" + TextureIndex);
    }
}

```

上述代码的主要逻辑是通过注册 `facesChanged` 事件，在 AR Face Manager 组件实例化人脸网格之前替换纹理。完成编码后，将该脚本也挂载在层级窗口中的 AR Session Origin 对象上。编译运行，将前置摄像头对准人脸，在检测到人脸后，当手在脸前拂过时，虚拟人脸网格纹理就会随之替换，类似于川剧中的变脸表演，效果如图 6-11 所示。

变脸效果演示了 `facesChanged` 事件中的 `added` 事件，`facesChanged` 事件还提供了 `updated`、`removed` 事件，利用这些事件，就能在检测到人脸时、检测到的人脸更新时、人脸丢失跟踪时

进行相应的逻辑处理,实现特定功能。在同一个会话中,ARCore会对每个检测到的人脸使用一个唯一的 TrackableId 进行标识,利用这个 TrackableId 可以判断当前人脸的跟踪情况,实现对特定人脸的跟踪功能。



图 6-11 变脸效果示意图

6.3 人脸区域与多人脸检测

除检测人脸姿态和提供检测到的人脸网络信息,ARCore 人脸检测还提供了人脸区域 (Region) 的概念与功能。

6.3.1 人脸区域

ARCore 在进行人脸检测时同时运行了两个实时深度神经网络模型,一个负责检测整张图像计算人脸位置,另一个借助通用三维网格模型通过回归方式检测并预测大致的面部结构。通过粗、细两个神经网络模型,可以有效地提高检测性能,降低资源消耗。

ARCore 检测到人脸后,提供一张由 468 个顶点组成的人脸网格,人脸网格包括顶点信息、中心点姿态信息、人脸区域信息。中心点是人脸网格顶点的坐标原点,即为人脸网格的局部坐标系原点。

除此之外,ARCore 还对网格的特定区域进行了标定,目前定义了 3 个人脸局部区域,即图 6-8 所示 3 个黄色小圆圈区域,这 3 个标定的区域坐标等信息可以在应用开发中直接使用。为了方便开发者,ARCore 在代码中定义了 ARCoreFaceRegion 枚举类型描述这 3 个标定区域,其枚举值如表 6-3 所示。

由于人脸的结构特性,在有这 3 个局部区域定位后,就可以方便地将虚拟物体挂载到脸上,ARCoreFaceRegion 枚举以后应该还会继续扩充,起码会包括眼睛、耳朵、嘴部、脸颊,更方便我们精确定位检测到的人脸模型的具体位置。

表 6-3 ARCoreFaceRegion 枚举

属 性	描 述
ForeheadLeft	定位人脸模型的左前额
ForeheadRight	定位人脸模型的右前额
NoseTip	定位人脸模型的鼻尖

下面以鼻尖区域 (NoseTip) 为例, 演示将盔甲面罩戴在人脸上, 具体思路如下:

(1) 制作一个盔甲面罩模型, 将眼睛部位镂空。

(2) 注册 AR Face Manager 组件的 facesChanged 事件, 在 added、updated 中将面罩挂在 NoseTip 位置。

为了将面罩挂在 NoseTip 位置, 需要在人脸区域中找到 NoseTip 位置, 新建一个 C# 脚本, 命名为 SingleFaceRegionManager, 编写代码如下:

```
// 第 6 章 / 6-4
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARCore;

[RequireComponent(typeof(ARFaceManager))]
[RequireComponent(typeof(ARSessionOrigin))]
public class SingleFaceRegionManager : MonoBehaviour
{
    [SerializeField]
    private GameObject mRegionPrefab; // 需要挂载的模型预制体
    private ARFaceManager mARFaceManager;
    private GameObject mInstantiatedPrefab; // 实例化后的预制体
    private ARSessionOrigin mSessionOrigin;
    private NativeArray<ARCoreFaceRegionData> mFaceRegions; // 人脸区域数组

    private void Awake()
    {
        mARFaceManager = GetComponent<ARFaceManager>();
        mSessionOrigin = GetComponent<ARSessionOrigin>();
        mInstantiatedPrefab = Instantiate(mRegionPrefab, mSessionOrigin.
trackablesParent);
    }
    // 注册事件
    private void OnEnable()
    {
        mARFaceManager.facesChanged += OnFacesChanged;
    }
    // 取消事件注册
    void OnDisable()
    {

```

```

        mARFaceManager.facesChanged -= OnFacesChanged;
    }
    void OnFacesChanged(ARFacesChangedEventArgs eventArgs)
    {
        foreach (var trackedFace in eventArgs.added)
        {
            OnFaceChanged(trackedFace);
        }

        foreach (var trackedFace in eventArgs.updated)
        {
            OnFaceChanged(trackedFace);
        }
        /*
        foreach (var trackedFace in eventArgs.removed)
        {
            OnFaceRemoved(trackedFace);
        }
        */
    }
    // 处理模型挂载位置
    private void OnFaceChanged(ARFace refFace)
    {
        var subsystem = (ARCoreFaceSubsystem)mARFaceManager.subsystem;
        subsystem.GetRegionPoses(refFace.trackableId, Allocator.Persistent, ref
mFaceRegions);
        for (int i = 0; i < mFaceRegions.Length; ++i)
        {
            var regionType = mFaceRegions[i].region;
            if (regionType == ARCoreFaceRegion.NoseTip)
            {
                mInstantiatedPrefab.transform.localPosition = mFaceRegions[i].
pose.position;
                mInstantiatedPrefab.transform.localRotation = mFaceRegions[i].
pose.rotation;
            }
        }
    }
    void OnDestroy()
    {
        if (mFaceRegions.IsCreated)
            mFaceRegions.Dispose();
    }
}

```

在上述代码中，为了更有效地利用模型而不是在每次检测到人脸 added、updated 事件时重新实例化盔甲面罩模型，可以只在 Awake() 方法中实例化一个盔甲，并在 added、updated

事件中实时地更新该实例化对象的姿态。

将该脚本挂载在层级窗口中的 AR Session Origin 对象上，并将盔甲面罩预制体赋给 mRegionPrefab 属性，同时将 AR Face Manager 组件中的 Face Prefab 属性置空。编译运行，将手机前置摄像头对准人脸，效果如图 6-12 所示。



图 6-12 在 ARCore 区域定位点挂载虚拟物体示意图

正如前文所述，ARCore 的区域标定只是方便我们以更精确的方式在特定位置挂载虚拟物体。事实上，在 ARCore 检测到人脸时会提供人体头部中心点姿态信息，加之人脸结构的固定性，即使没有人脸区域的概念，也可以通过操控模型偏移一定的距离而将模型挂载在人脸的指定位置，如可以通过调整模型位置只使用 AR Face Manager 组件将眼镜模型挂载在人脸，如图 6-13 和图 6-14 所示。

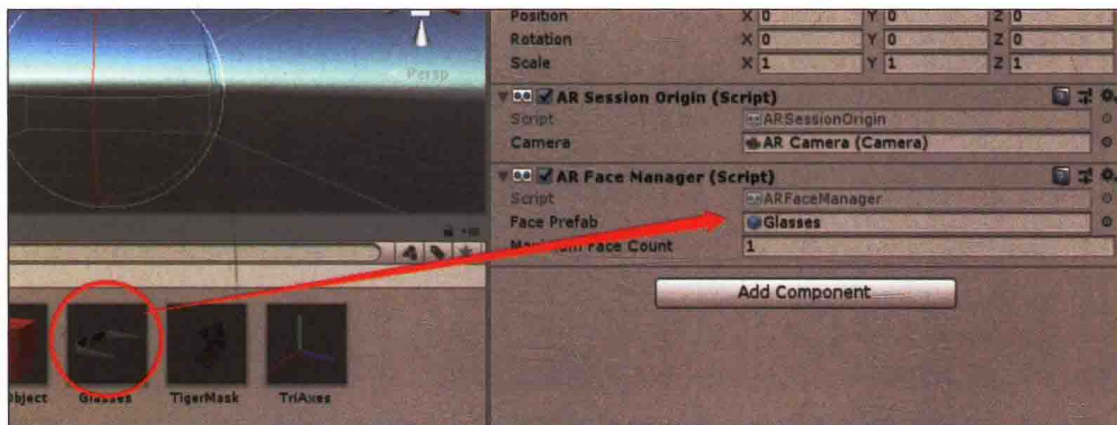


图 6-13 只使用 AR Face Manager 组件在特定位置挂载虚拟对象



图 6-14 只使用 AR Face Manager 组件在特定位置挂载虚拟对象示意图

但相比于特征区域，这种通过模型偏移实现模型挂载有一个问题，因为是通过相对中心点的偏移将虚拟物体挂载到特定位置，这个偏移量的个体差异会很大，如挂在小孩脸上的眼镜正合适时挂在成人脸上时眼镜就会偏下（小孩眼睛离中心点距离比成人小），因此在能使用区域位置特性时尽量使用区域，这样可以提高模型的定位精度。

6.3.2 多人脸检测

ARCore 支持多人脸检测^①，在多人脸检测中，除了注册 facesChanged 事件，也可以直接在 Update() 方法中进行处理，代码如下：

```
// 第6章 / 6-5
using UnityEngine.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;
using UnityEngine.XR.ARCore;

[RequireComponent(typeof(ARFaceManager))]
[RequireComponent(typeof(ARSessionOrigin))]
public class ARCoreFaceRegionManager : MonoBehaviour
```

① 由于人脸检测属于性能消耗较大的功能，检测跟踪太多人脸会导致应用性能下降。

```

{
    [SerializeField]
    private GameObject mRegionPrefab; // 模型预制体
    private ARFaceManager mFaceManager;
    private ARSessionOrigin mSessionOrigin;
    private NativeArray<ARCoreFaceRegionData> mFaceRegions; // 人脸区域数组
    private Dictionary<TrackableId, Dictionary<ARCoreFaceRegion, GameObject>>
    mInstantiatedPrefabs;

    void Start()
    {
        mFaceManager = GetComponent<ARFaceManager>();
        mSessionOrigin = GetComponent<ARSessionOrigin>();
        mInstantiatedPrefabs = new Dictionary<TrackableId, Dictionary<ARCoreFaceRegion,
        GameObject>>();
    }
    void Update()
    {
        var subsystem = (ARCoreFaceSubsystem)mFaceManager.subsystem;
        if (subsystem == null)
            return;
        // 通过 AR Face Manager 组件中的跟踪对象进行多人脸检测
        foreach (var face in mFaceManager.trackables)
        {
            Dictionary<ARCoreFaceRegion, GameObject> regionGos;
            if (!mInstantiatedPrefabs.TryGetValue(face.trackableId, out regionGos))
            {
                regionGos = new Dictionary<ARCoreFaceRegion, GameObject>();
                mInstantiatedPrefabs.Add(face.trackableId, regionGos);
            }
            subsystem.GetRegionPoses(face.trackableId, Allocator.Persistent, ref
            mFaceRegions);
            for (int i = 0; i < mFaceRegions.Length; ++i)
            {
                var regionType = mFaceRegions[i].region;
                if (regionType == ARCoreFaceRegion.NoseTip)
                {
                    GameObject go;
                    if (!regionGos.TryGetValue(regionType, out go))
                    {
                        go = Instantiate(mRegionPrefab, mSessionOrigin.
                        trackablesParent);

                        regionGos.Add(regionType, go);
                        //Debug.Log("Object Count :" + mInstantiatedPrefabs.
                        Count);
                    }
                    go.transform.localPosition = mFaceRegions[i].pose.position;
                }
            }
        }
    }
}

```



```

        go.transform.localRotation = mFaceRegions[i].pose.rotation;
    }
}

void OnDestroy()
{
    if (mFaceRegions.IsCreated)
        mFaceRegions.Dispose();
}
}

```

代码中演示了通过查询 AR Face Manager 组件中跟踪对象的方式实现了人脸检测相关功能，这与注册 facesChanged 事件相比更加直观，但是这种方式不如使用 facesChanged 事件方式优雅。

代码也是通过 NoseTip 位置挂载模型，如需使用 ForeheadLeft 或 ForeheadRight 位置挂载虚拟物体，处理方式完全一致。在上述代码中，我们在所有检测到的人脸 NoseTip 位置挂载了相同的虚拟物体模型，如果需要挂载不同的虚拟物体，则可以设置一个预制体数组用于保存所有待实例化的预制体，然后通过一个随机数发生器随机实例化虚拟物体。当前，无法做到在特定的人脸上挂载特定的虚拟物体，因为 ARCore 只提供人脸检测跟踪而不提供人脸识别功能。

需要注意的是，使用多人脸检测时，需要将 AR Face Manager 组件中的 Maximum Face Count 属性设置为需要检测的最大人脸数，这样才能启动多人脸检测功能，默认只检测单个人脸。

在人脸渲染的纹理或者挂载的模型通常由美术人员预先制作，由于人脸网格信息在运行时动态生成，没有直观的可参考对象，为帮助美术人员制作纹理和模型，ARCore 提供了一个人脸网格模型（canonical_face_mesh.fbx 模型文件）和一个纹理文件（canonical_face_mesh.psd 源文件）。

其中网格模型文件包含网格拓扑结构、UV 纹理坐标、区域特征标识等信息，通过三维制作软件（如 3ds Max、Blender 等）可以以此为参考制作特定的模型文件。在以 ARCore 提供的模型为参考制作特定模型时，需要保持模型的层次结构，不要修改顶点数据和区域名称等数据，因为这些数据都是预先规定好的，不正确的修改可能会导致运行时的问题，导出自定义模型时不需要包含 facemesh 模型，因为 ARCore 将在运行时动态创建人脸网格数据。

纹理文件可以协助美术人员创建所需纹理，以此为参考可以确保纹理与人脸面部特征吻合对齐，该纹理如图 6-15 所示，ARCore 提供的 PSD 源文件包括以下 4 个图层：

- （1）Mask 图层标识眼睛、鼻孔、嘴部所在位置，便于以此为参考镂空这些区域。
- （2）Lines 图层显示了人脸面部特征与纹理对齐的形式。
- （3）UVs 图层显示了人脸网格 468 个顶点的 UV 分布。

(4) Background 图层是一个中灰色的背景图，主要是为了方便在创作时使其他图层清晰可见。

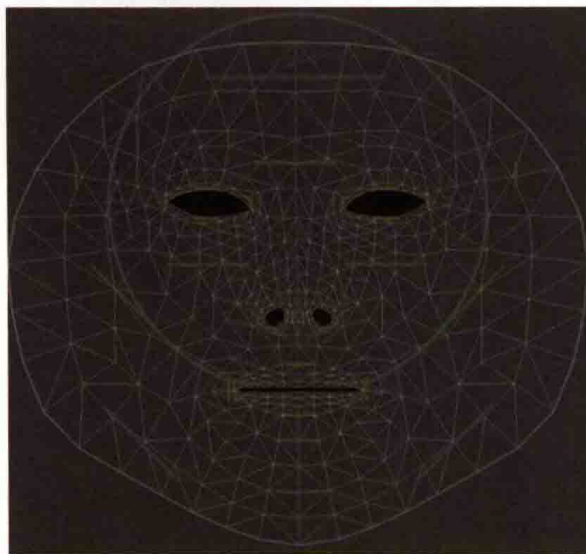


图 6-15 ARCore 提供的纹理源文件图

目前基于人脸检测的 AR 应用已经在很多场合被使用，如美妆、首饰佩戴、眼镜佩戴等，并在社交类应用中广泛使用，ARCore 提供的人脸检测跟踪功能可以极大地方便开发人员进行基于人脸的应用开发。

第7章



光影效果

光影是影响物体感观非常重要的部分，在真实环境中，人脑通过对光影的分析，可以迅速定位物体空间位置、光源位置、光源强弱、物体三维结构、物体之间的关系、周边环境等，光影还影响人们对物体表面材质属性的直观感受。在 AR 中，光影效果直接影响 AR 虚拟物体的真实感。本章主要学习在 AR 中实现光照估计、环境光反射、阴影生成等相关知识，提高 AR 场景中虚拟物体渲染的真实感。

7.1 光照基础

在现实世界中，光扮演了极其重要的角色，没有光万物将失去色彩，没有光世界将一片漆黑。在三维数字世界中亦是如此，三维数字世界本质上是一个使用数学精确描述的真实世界的副本，光照计算是影响这个数字世界可信度的极其重要的因素。

在图 7-1 中，图 (a) 是无光照条件下的球体（并非全黑是因为设置了环境光），这个球体看起来与一个二维圆形并无区别，图 (b) 是有光照条件下的球体，立体形象已经呈现，有高光、有阴影。这只是一个简单的示例，事实上视觉感知环境就是通过光与物体材质的交互而产生的。

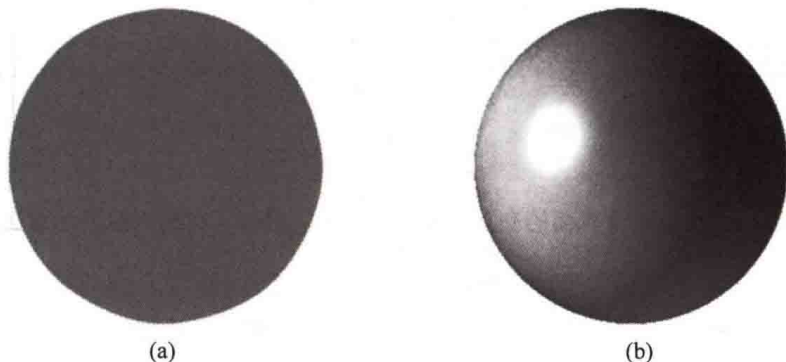


图 7-1 光照影响人脑对物体形状的判断

三维数字世界渲染的真实度与三维数字世界使用的光照模型有直接关系，越高级的光照模型对现实世界模拟得越好，场景看起来越真实，当然计算开销也越大，特别是对实时渲染

的应用来讲, 选择一个合适的折中方案就很关键。

7.1.1 光源

顾名思义, 光源即是光的来源, 常见的光源有阳光、月光、星光、灯光等。光的本质其实很复杂, 它是一种电磁辐射, 但却有波粒二相性 (此处不深入研究光学, 那将是一件非常复杂且枯燥的工作, 在计算机图形学中, 只需了解一些简单的光学属性并应用)。在实时渲染时, 通常把光源当成一个没有体积的点, 用 L 来表示由其发射光线的方向, 使用辐照度 (Irradiance) 来量化光照强度。对平行光而言, 它的辐照度可以通过计算在垂直于 L 的单位面积上单位时间内穿过的能量来衡量。在图形学中考虑光照, 只要想象光源会向空间中发射带有能量的光子, 然后这些光子会与物体表面发生作用 (反射、折射、透射和吸收), 最后的结果是我们看到物体的颜色和各種纹理。

7.1.2 光与材质的交互

当光照射到物体表面时, 一部分能量被物体表面吸收, 另一部分被反射, 如图 7-2 所示, 对于透明物体, 还有一部分光穿过透明体, 产生透射光。被物体吸收的光能转化为热量, 只有反射光和透射光能够进入人的眼睛, 产生视觉效果。反射和透射产生的光波决定了物体呈现的亮度和颜色, 即反射和透射光的强度决定了物体表面的亮度, 而它们含有的不同波长光的比例决定了物体表面的色彩, 所以物体表面光照颜色由入射光、物体材质, 以及材质和光的交互作用共同决定。

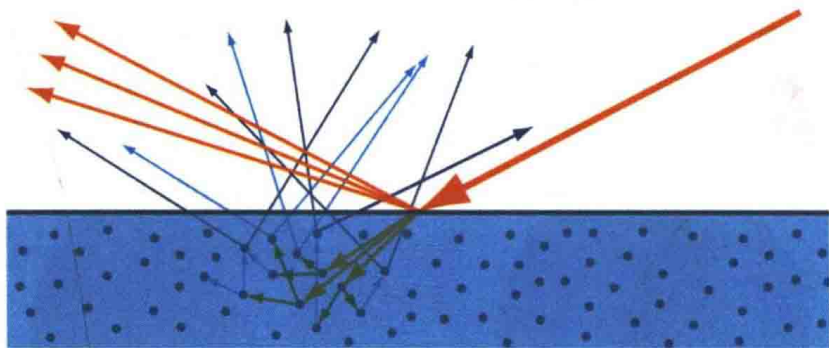


图 7-2 光与物体表面的交互, 反射、折射、散射、次表面散射示意图

物体材质可以认为是决定光如何与物体表面相互作用的属性。这些属性包括表面反射和吸收的光的颜色、材料的折射系数、表面光滑度、透明度等。通过指定材质属性, 可以模拟各种真实世界的表面视觉表现, 如木材、石头、玻璃、金属和水等。

在计算机图形学光照模型中, 光源可以发出不同强度的红、绿和蓝光, 因此可以模拟各种光色。当光从光源向外传播并与物体发生碰撞时, 其中一些光可能被吸收, 另一些光则可能被反射 (对于透明物体, 如玻璃, 有些光线透过介质, 但这里不考虑透明物体)。反射的光

沿着新的反射路径传播，并且可能会击中其他物体，其中一些光又被吸收和反射。光线在完全吸收之前可能会击中许多物体，最终，一些光线会进入我们的眼睛，如图 7-3 所示。

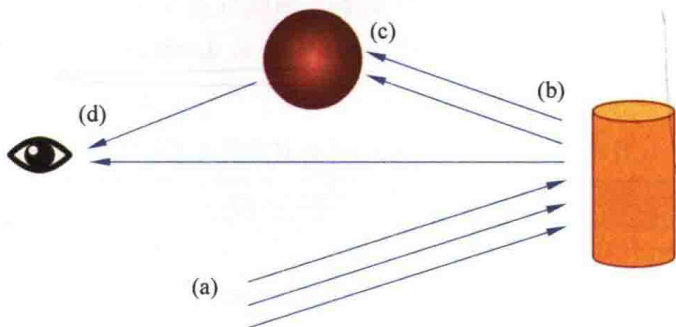


图 7-3 入射到眼睛中的光是光源与世界环境进行多次交互后综合的结果

根据三原色理论，眼睛的视网膜包含 3 种光感受器，每种光感受器对特定的红光、绿光和蓝光敏感（有些重叠），进入人眼的 RGB 光会根据光的强度将其相应的光感受器刺激到不同的强度。当光感受器受到刺激（或不刺激）时，神经脉冲从视神经向大脑发射脉冲信号，大脑综合所有光感受器的脉冲刺激产生图像（如果闭上眼睛，光感受器细胞就不会受到任何刺激，默认大脑会将其标记为黑色）。

在图 7-3 中，假设圆柱体表面材料反射 75% 的红光和 75% 的绿光，而球体反射 25% 的红光，吸收其余的红光，也假设光源发出的是纯净的白光。当光线照射到圆柱体时，所有的蓝光都被吸收，只有 75% 的红光和 75% 的绿光被反射出来（中等强度的黄光）。这束光随后被散射，其中一些进入眼睛，另一些射向球体。进入眼睛的部分主要将红色和绿色的锥细胞刺激到中等的程度，观察者会认为圆柱体表面是一种半明亮的黄色。其他光线向球体散射并击中球体，球体反射 25% 的红光并吸收其余部分，因此，稀释后的红光（中高强度红色）被进一步稀释和反射，所有射向球体的绿光都被吸收，剩余的红光随后进入眼睛，将红锥细胞刺激到一个较低的程度，因此观察者会认为球体是一种暗红色的球。这就是光与材质的交互过程。

在 AR 中，ARCore 会根据摄像头采集的环境图像自动评估环境中的光照信息并修正虚拟元素的光照，通常情况下都能达到比较理想的光照效果，但也可以在场景中手动添加灯光，调整灯光颜色和强度，从而更精准地控制虚拟物体的光照。

光与物体材质交互的过程是持续性的，即光线在环境中会持续地被反射、透射、吸收、衍射、折射、干涉，直到光线全部被吸收转化为热量，完整模拟这个过程目前还不现实（需要庞大的算力），因此，为了简化光与材质交互的计算，人们建立了一些数学模型来替代复杂的物理模型，这些模型被称为光照模型（Illumination Model）。

7.1.3 光照模型

光照模型，也称为明暗模型，用于计算物体某点的光强和颜色值，从数学角度而言，光照模型本质上是一个或者一组数学公式，使用这些公式来计算物体某点的光照效果。

根据光照模型所关注的重点和对光线交互的抽象层次的不同，光照模型又分为局部光照模型和全局光照模型：局部光照模型只关注光源所发射光线到物体的一次交互，常用于实时渲染；全局光照模型关注的光照交互层级更多，因此计算量更大，通常用于离线渲染。AR Foundation 支持的局部光照模型包括环境光 (Ambient)、Lambert、Blinn、Phong、PBR，详见表 7-1，它们的渲染效果如图 7-4 所示。

表 7-1 AR Foundation 支持的光照模型

光照模型	描 述
Ambient	所有物体表面使用相同的常量光照，常用于模拟环境光照效果
Lambert	该模型包含了环境光与漫反射两种光照效果
Phong	该模型包含了环境光、漫反射和高光反射 3 种光照效果
Blinn	该模型包含了环境光、漫反射和高光反射 3 种光照效果，改进自 Phong 模型，可以实现更柔和的高光效果，并且计算速度更快
PBR	该光照模型基于真实的光线与材质的交互物理规律，通常采用微面元理论，因此，表现出来的效果更真实，但需要注意的是，PBR 虽然是基于物理的光照模型，但并不是完全使用物理公式进行计算，也采用了部分经验公式，所以，并不是说 PBR 光照模型就完全可信，当前主流的渲染引擎基本支持 PBR



图 7-4 不同光照模型对物体光照计算的效果图

7.1.4 三维渲染

计算机图形学中，三维渲染又称为着色 (Shading)，是指对三维模型进行纹理与光照处理并光栅化成像素的过程。图 7-5 直观地展示了三维渲染过程。在三维渲染中，顶点着色器 (Shader) 负责对顶点进行空间变换和光照计算 (Transformation&Lighting, T&L)，然后输入片元着色器中进行逐像素照明计算和阴影处理，三维渲染的最后一步称为光栅化，即生成每像素的颜色信息，随后这些像素被输到帧缓冲中由显示器进行显示^①。



图 7-5 AR 渲染物体的过程

^① 这里讨论的渲染过程是指利用 DirectX 或 OpenGL 技术在设备的 GPU 上进行标准的实时渲染过程，目前已有一些渲染方式采用另外的渲染架构，但那不在我们的讨论范围之内。

7.2 光照估计

AR 与 VR 在光照计算上最大的不同在于 VR 场景世界是纯数字世界，有一套完整的数学模型，而 AR 则是将计算机生成的虚拟物体或关于真实物体的非几何信息叠加到真实世界的场景之上实现对真实世界的增强，融合了真实世界与数字世界。就光照计算而言，VR 中的光照类型、属性完全由开发人员决定，光照效果是一致的，而 AR 中则不得不考虑真实世界的光照信息与虚拟的三维光照信息的一致性，举个例子，如果在 AR 应用中设置了一个模拟太阳的高亮度方向光，而用户是在光照比较昏暗的时间或者地点使用这个 AR 应用，如果不考虑光照一致性，则渲染出来的虚拟物体的光照与真实世界其他物体的光照反差将会非常明显，由于人眼对光照信息的高度敏感，这种渲染可以说是失败的，沉浸感全无。在 AR 中，由于用户与真实世界的联系并未被切断，光照的交互方式也要求更自然，如果真实世界的物体阴影向左而渲染出来的虚拟物体阴影却向右，这也是让人难以接受的，所以在 AR 中，需要尽力将虚拟光照与真实光照保持一致，这样才能提高虚拟物体渲染的可信度和真实感。

在真实世界中，当用户移动时，光照方向和强度有可能会剧烈地发生变化（如用户从室外走到室内），这种变化也会导致 AR 应用虚拟物体渲染光照不一致的问题。

7.2.1 光照一致性

光照一致性的目标是使虚拟物体的光照情况与真实场景中的光照情况保持一致，虚实物体有着一致的明暗、阴影、高光效果，以增强虚拟物体的真实感。解决光照一致性问题的关键是获取真实场景中的准确光照信息，准确的光照信息能够实现更加逼真的虚拟物体渲染效果。光照一致性包含的技术性问题很多，完全的解决方案需要场景精确的几何模型和光照模型，以及场景中物体的光学属性描述，这样才可能绘制出真实场景与虚拟物体的光照交互，包括真实场景中的光源对虚拟物体产生的明暗、阴影和反射及虚拟物体对真实物体的明暗、阴影和反射的影响。实现光照一致性是增强现实技术中的一个涉及众多因素的技术挑战，光照模型的研究是解决光照一致性问题的重要手段，其中一个关键的环节是要获取现实环境中真实光照的分布信息。目前对真实环境光照的估计方法主要包括借助辅助标志物的方法、借助辅助拍摄设备的方法、基于图像分析的方法等，ARCore 使用基于图像分析的方法。

7.2.2 光照估计实例

AR Foundation 支持 ARCore 和 ARKit 光照估计功能，在 AR 应用中使用光照估计，首先要打开 AR Foundation 光照估计功能，在层级（Hierarchy）窗口中，依次选择 AR Session Origin → AR Camera，然后在属性（Inspector）窗口的 AR Camera Manager 组件中，选择 Light Estimation Mode 为相应值（非 None 值）即可打开光照估计功能，如图 7-6 所示。



图 7-6 开启 AR Foundation 光照估计功能

从图 7-6 可以看到，AR Foundation 支持很多种类的光照估计类型，但实际上这些值与特定平台相关，具体如表 7-2 所示。

表 7-2 AR Foundation 支持的光照估计值与平台详情

光照估计选项	平 台	前后摄像头	描 述
Ambient Intensity	ARKit	后	环境光强度估计
Ambient Color	ARCore/ARKit	后	环境光颜色估计
Ambient Spherical Harmonics	ARCore/ARKit	前	环境光球谐分布估计
Main Light Direction	ARCore	后	场景主光源方向
Main Light Intensity	ARCore	后	场景主光源强度

在使用光照估计时，这些值可以多选，但它们只在特定的平台被支持，如果平台不支持，选择的光照估计值无效，不会产生输出。在 Android 平台上，光照估计会对环境中的主光源强度（Intensity）、颜色（Color）、方向（Direction）进行评估计算，而在 iOS 平台上，只对环境中的光照强度（AmbientIntensity）、颜色（AmbientColor）进行估计。

在代码层面，AR Foundation 使用 `ARLightEstimationData` 结构体描述光照估计值，但其值与 AR Camera Manager 组件中的选项值略有出入，容易产生混淆，大致对应关系如表 7-3 所示。

表 7-3 `ARLightEstimationData` 结构体与平台光照估计选项的对应关系

光照估计选项	结 构 体 项	描 述
Ambient Intensity	<code>averageBrightness / averageIntensity-InLumens</code>	环境光强度估计，结构体值使用两种表示法
Ambient Color	<code>averageColorTemperature / colorCorrection / mainLightColor</code>	环境光颜色估计，ARKit 支持 <code>average - ColorTemperature</code> ，ARCore 支持后两种
Ambient Spherical Harmonics	<code>ambientSphericalHarmonics</code>	环境光球谐分布估计
Main Light Direction	<code>mainLightDirection</code>	场景主光源方向
Main Light Intensity	<code>averageMainLightBrightness / mainLightIntensityLumens</code>	场景主光源强度，结构体值使用两种表示法

下面进行使用演示，在 Unity 工程中新建一个 C# 脚本，命名为 `LightEstimation`，编写代

码如下:

```
// 第7章 /7-1
using UnityEngine;
using UnityEngine.XR.ARFoundation;

public class LightEstimation : MonoBehaviour
{
    [SerializeField]
    private ARCameraManager mCameraManager;
    private Light mLight;

    void Awake()
    {
        mLight = GetComponent<Light>();
    }
    // 注册事件
    void OnEnable()
    {
        if (mCameraManager != null)
            mCameraManager.frameReceived += FrameChanged;
    }
    // 取消事件注册
    void OnDisable()
    {
        if (mCameraManager != null)
            mCameraManager.frameReceived -= FrameChanged;
    }
    // 光照估计每一帧都会进行, 因此是一个资源消耗大的功能
    void FrameChanged(ARCameraFrameEventArgs args)
    {
        if (args.lightEstimation.averageBrightness.HasValue)
        {
            mLight.intensity = args.lightEstimation.averageBrightness.Value;
            Debug.Log("averageBrightness:" + args.lightEstimation.
averageBrightness.Value);
        }

        if (args.lightEstimation.mainLightColor.HasValue)
        {
            mLight.color = args.lightEstimation.mainLightColor.Value;
            Debug.Log("mainLightColor:" + args.lightEstimation.mainLightColor.
Value.ToString());
        }

        if (args.lightEstimation.mainLightDirection.HasValue)
        {

```



```
mLight.transform.rotation = Quaternion.LookRotation(args.lightEstimation.  
mainLightDirection.Value);  
    Debug.Log("mainLightDirection:" + args.lightEstimation.  
mainLightDirection.Value.ToString());  
    }  
}  
}
```

使用时, 将该脚本挂载在场景中的 Directional Light 对象上, 并将 AR Session Origin → AR Camera 赋给脚本的 mCameraManager 属性, 编译运行, 即可看到相应输出。通过观察输出, 可以看到, 在 Android 平台, 使用 ARCore 时, averageBrightness 项没有值。

在实际应用中, 颜色校正 (colorCorrection) 其实是比较微弱的; 主光源光照强度值 (averageMainLightBrightness) 大概率在 0.4 ~ 0.5 浮动, 因此有时为了让效果更明显, 会通过倍乘一个系数来对光照强度、颜色值进行增强, 例如乘以 2 扩大影响范围。

ARCore 与 ARKit 的光照估计技术都是建立在图像分析方法基础之上, 使虚拟物体能够根据环境光照信息改变自身光照情况, 增强虚拟物体在现实世界中的真实感。基于图像的光照估计算法需要对摄像头获取的每一帧图像的每像素进行数学运算, 计算量非常大, 因此, 在不需要光照估计功能时应当及时关闭该功能。关闭光照估计的方法是选择场景 AR Session Origin → AR Camera 对象, 将其 AR Camera Manager 组件中的 Light Estimation Mode 属性设为 None 值。

需要说明的是, 默认光照估计功能是关闭的, 而且虚拟场景中的光照不会自动进行调整, 但通过上述代码, 可以使用代码的方式根据真实光照估计情况控制虚拟场景中的主光源方向、强度、颜色等, 从而达到光照一致性效果。图 7-7 是在同一环境光照条件下, 通过更换不同的背景模拟环境光改变, 虚拟物体光照也发生了比较明显的变化 (为强化效果, 此处将 ColorCorrection 参数乘以系数 5)。



图 7-7 环境光变化影响虚拟物体渲染效果图

7.3 环境反射

在 AR 中实现环境光反射是一项非常高级的功能，也是增强 AR 虚拟物体可信度的一个重要组成部分，虚拟物体反射其周边环境，能极大地增强其真实感，但因为 AR 应用环境信息的采集往往都不完整，需要利用人工智能技术推算并补充不完整的环境信息，因此，AR 中的环境反射不能做到非常精准。本节在进行环境反射操作学习之前，先讲述一些基础知识。

7.3.1 立方体贴图

立方体贴图（Cubemap）通常用于环境映射，Unity 中的 Skybox（天空盒）就是立方体贴图。立方体贴图通常被用来作为具有反射属性物体的反射源，它是一个由 6 个独立的正方形纹理组成的纹理集合，包含 6 个二维纹理，每个二维纹理为立方体的一个面，6 个纹理共同组成一个有贴图的立方体，如图 7-8 所示。

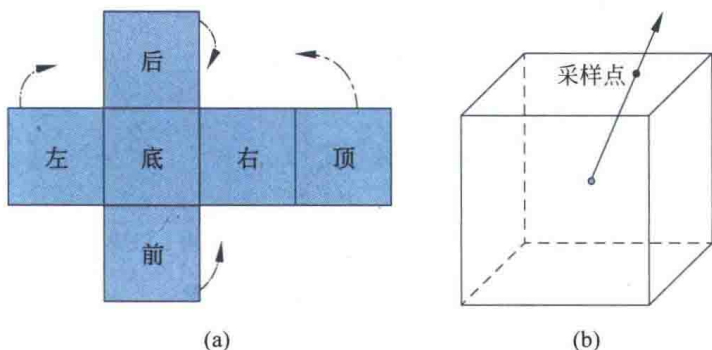


图 7-8 立方体贴图展开与采样示意图

在图 7-8 (a) 中，沿着虚线箭头方向可以将这 6 个面封闭成一个立方体，形成一个纹理面向内的贴图集合，这也是立方体贴图名字的由来。立方体贴图最大的特点是构成了一个 720° 全封闭的空间，因此如果组成立方体贴图的 6 张纹理选择连续无缝贴图就可以实现 720° 无死角的纹理采样，形成完美的天空盒效果。

与二维纹理采样使用 UV 坐标不同，立方体贴图需要一个三维查找向量进行采样，定义这个查找向量为原点位于立方体中心点的三维向量，如图 7-8 (b) 图所示，三维查找向量与立方体相交处的纹理就是需要采样的纹理。在 GLSL、HLSL、Cg 中都定义了立方体贴图采样函数，可以非常方便地进行立方体贴图采样操作。

立方体贴图因其 720° 封闭的特性常常用来模拟某点的周边环境，实现反射、折射效果，如根据赛车位置实时更新立方体贴图可以模拟赛车车身对周边环境的反射效果。在 AR 中，也是利用同样的原理实现虚拟物体对真实环境的反射。

立方体贴图需要 6 个无缝的纹理，使用静态的纹理可以非常好地模拟全向场景，但静态纹理不能反映动态的物体变化，如赛车车身对周围环境的反射，如果使用静态纹理将不能反

射路上行走的人群和闪烁的霓虹灯,这时就需要使用实时动态生成的立方体贴图,这种方式能非常真实地模拟赛车对环境的反射,但性能开销比较大,通常需要进行优化。

7.3.2 纹理采样过滤

纹理采样过滤 (Texture Sampling Filter) 是指对采样纹理的过滤算法,这里只针对 AR Foundation 中对反射的 3 种采样过滤方式 (Point、Bilinear、Trilinear) 进行阐述。在虚拟物体渲染时,很多时候物体的尺寸与提供的纹理并不能一一对应,如在放大、缩小模型时,纹理就会发生变形,为了达到更好的视觉效果,需要对采样的纹理进行处理。

为了方便阐述,从一维的角度进行说明,假设现有一张 256×256 的纹理,其在 X 轴上就有 256 像素,若在使用中模型的某一顶点的 UV 坐标中的 U 值为 0.13852676,则其对应的像素为 $256 \times 0.13852676 = 35.46$,如图 7-9 所示。在纹理中,小数是没有定义的,因此,35.46 并不能直接对应某一像素的值。

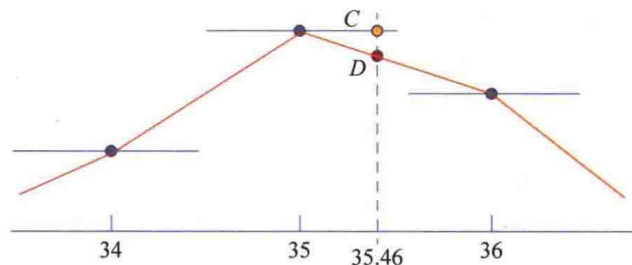


图 7-9 非整数纹理采样示意图

采样过滤就是为了解决这类纹理不能直接映射的问题。Point 过滤处理的方式是采用四舍五入法取最近的值,因此其值为 C ,与 35 对应的像素值一致。Bilinear 过滤的处理方式是在最近的两个值之间作插值,即在 35 与 36 所对应的像素值间根据小数值做插值,其值为 D 。可以看到,Point 过滤比 Bilinear 过滤简单得多,但是 Bilinear 过滤比 Point 过滤结果更平滑,其效果如图 7-10 所示,(a) 图为原图,(b) 图为使用 Point 过滤且放大后的图像,可以看到明显的块状像素,(c) 图为使用 Bilinear 过滤且放大后的图像,效果要平滑得多。

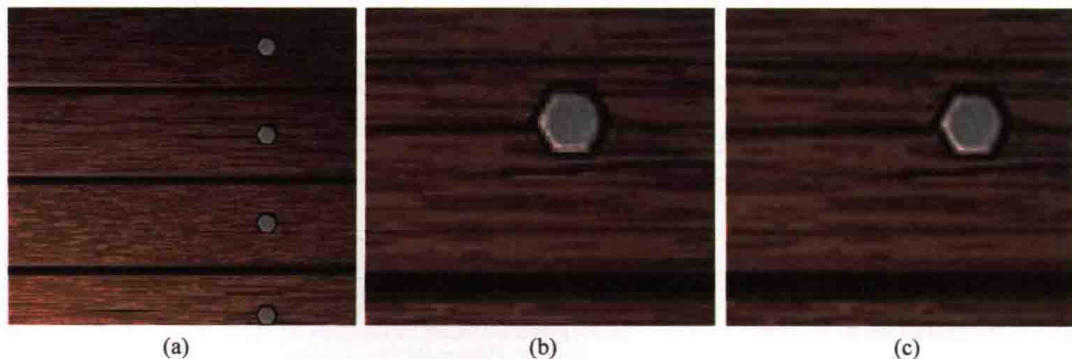


图 7-10 纹理采样 Point 过滤与 Bilinear 过滤效果图

Trilinear 过滤解决的是在不同 LoD (Levels of Detail) 间过滤纹理的问题, 原理如图 7-11 所示, 其实际上是进行了两次插值计算, 先进行一次 Bilinear 过滤计算出 C_b 与 C_a 的值, 然后在不同的 LoD 间进行一次 Bilinear 过滤, 因此结果更加平滑。

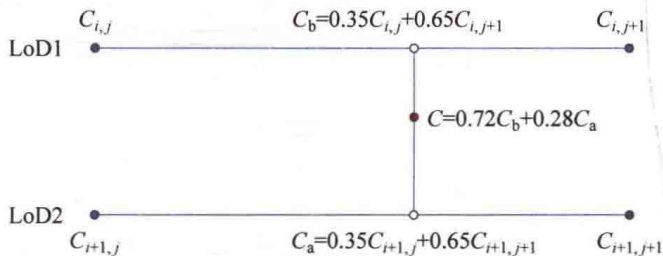


图 7-11 纹理采样 Trilinear 过滤示意图

在 AR Foundation 中使用环境光反射时, 为了达到更好的效果, 通常会选择 Bilinear 或者 Trilinear 纹理过滤方式, 如图 7-12 所示。

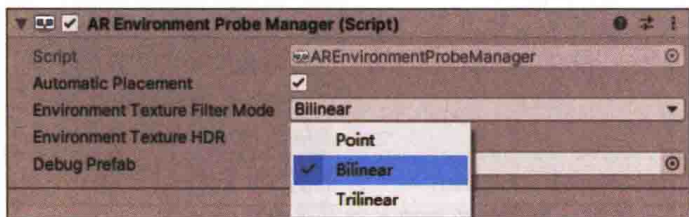


图 7-12 AR 中环境反射纹理采样过滤模式选择面板

7.3.3 AR Environment Probe Manager 组件

在 AR Foundation 中, 由 AR Environment Probe Manager 组件负责管理环境反射相关任务, AR Environment Probe Manager 组件有 3 个属性, 分别是 Automatic Placement (自动放置)、Environment Texture Filter Mode (环境纹理过滤模式)、DeBug Prefab (调试预制体), 如图 7-12 所示。

1. Automatic Placement

AR Foundation 既允许自动放置反射探头 (Probe) 也允许手动放置反射探头, 或者两者同时使用。在自动放置模式下, 由应用程序自动选择合适的位置放置反射探头。在手动放置模式下, 由开发人员在指定位置放置反射探头。自动放置由底层 SDK 提供的算法自主选择在何处及如何放置反射探头, 以获得比较高质量的环境信息, 自动放置位置的确定依赖于在实际环境中检测到的关键特征点信息。手动放置主要是为了获得对特定虚拟对象的最精确环境信息, 绑定反射探头与虚拟对象位置可以提高反射渲染的质量, 因此, 手动将反射探头放置在重要的虚拟对象中或其附近会产生为该对象生成的最准确的环境反射信息。通常而言, 自动放置可以提供对真实环境比较好的宏观环境信息, 而手动放置能提供在某一个点上对周围环

境更准确的环境映射从而提升反射的质量，但 ARCore 只支持自动放置，在 Android 平台使用环境反射时，需要勾选自动放置复选框。

2. Environment Texture Filter Mode

环境纹理过滤模式主要可选为 Point、Bilinear、Trilinear，为达到比较好的视觉效果，一般选择 Bilinear 或者 Trilinear。

3. Debug Prefab

该属性用于在调试时可视化反射探头，在发布时应置为空。

AR Environment Probe Manager 组件还提供了如表 7-4 所示方法和事件，用于管理环境反射相关任务。

表 7-4 AR Environment Probe Manager 组件常用方法与事件

方 法	描 述
AddEnvironmentProbe(Pose pose, Vector3 scale, Vector3 size)	添加一个 AREnvironmentProbe，参数 pose 指定了放置环境探头的位置和方向，参数 scale 指定了相对于 AR 会话空间的比例，参数 size 指定了反射探头可探测的环境范围
GetEnvironmentProbe(TrackableId trackableId)	根据 TrackableId 获取一个 AREnvironmentProbe
RemoveEnvironmentProbe(AREnvironmentProbe probe)	移除一个 AREnvironmentProbe，如果移除成功，则返回值为 true，否则返回值为 false
environmentProbesChanged	在反射探头发生变化时触发，如一个新的环境探头创建、对一个现存的环境探头进行更新、移除一个环境探头

7.3.4 使用环境反射

AR 环境反射是一个高级功能，需要学习并掌握相关知识才能运用自如，但明白原理后，在 AR Foundation 中使用时却非常简单，基本步骤如下：

- (1) 在场景中 AR Session Origin 对象上挂载 AR Environment Probe Manager 组件并进行相应设置。
- (2) 确保需要反射的虚拟对象带有反射材质并能反射 Probe 探头。
- (3) 使用自动方式设置反射探头捕获环境信息。

按照上述步骤，首先在 AR Session Origin 对象上挂载 AR Environment Probe Manager 组件，勾选自动放置反射探头复选框，并将纹理过滤模式选择为 Bilinear，如图 7-12 所示。

然后选择需要进行环境反射的虚拟物体（或者预制体），设置其 Mesh Renderer 组件中的 Probes → Light Probes 属性为 Blend Probes，如图 7-13(a)所示；材质使用 URP 中的 Lit 着色器，并勾选上 Advanced Options 栏的 Specular Highlights 和 Environment Reflections 复选框，如图 7-13(b)所示。

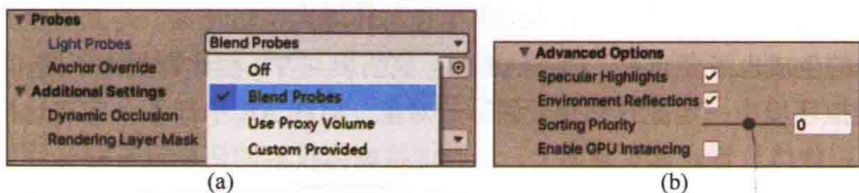


图 7-13 设置虚拟物体探针和反射材质属性

无须编写任何代码，编译运行，在放置虚拟物体后，可以旋转一下手机，扩大 AR 应用对环境信息的感知，效果如图 7-14 所示，虚拟物体反射了周边环境。



图 7-14 使用环境反射效果示意图

7.4 内置实时阴影

阴影在现实生活中扮演着非常重要的角色，通过阴影能直观地感受到光源位置、光源强弱、物体离地面高度、物体轮廓等，阴影还影响大脑对空间环境的判断，是构建立体空间信息的重要参考因素，如图 7-15 所示。



图 7-15 光照与阴影影响人脑对空间环境的认知理解

阴影的产生与光源密切相关,阴影的产生也与环境光密切相关。与真实世界一样,在数字世界中阴影的生成也需要光源,但在AR中生成阴影与VR相比有很大不同,VR是纯数字世界,在其虚拟环境中一定能找到接受阴影的对象,但在AR中却不一定有这样的对象,如将一个AR虚拟物体放置在真实桌面上,这时虚拟物体投射的阴影没有接受物体(桌面并不是数字世界中的对象,无法直接接受来自虚拟物体的阴影),因此就不能生成阴影。为使虚拟物体产生阴影,实现的思路是在虚拟物体下方放置一个接受阴影的对象,这个对象需要接受阴影但又不能有任何材质表现,即除了阴影部分,其他地方需要透明,这样才不会遮挡现实世界中的物体。

7.4.1 ShadowMap 技术原理

在Unity内置阴影实现中,实时阴影混合了多种阴影生成算法,根据要求可以是Shadowmap、Screen Space Shadowmap、Cascaded Shadow等,Shadowmap按技术又可以分为Standard Shadow Mapping、PCF、PSM、LISPSM、VSM、CSM/PSSM等。之所以设计这么多阴影生成算法,最主要的目的是平衡需求与复杂度。阴影生成的理论本身并不复杂,有光线照射的地方就是阳面(没有阴影),没有被光照射的地方就是阴面(有阴影),但现实环境光照非常复杂,阴影千差万别,有在太阳光直射下棱角特别分明的硬阴影,有在只有环境光照下界线特别不分明的超软阴影,也有介于这两者之间的阴影,目前没有一种统一的算法可以满足各种阴影生成要求,因此发展了各类阴影算法处理在特定情况下阴影的生成问题。

利用ShadowMap技术生成阴影需要进行两次渲染:第一次从光源的视角渲染一张RenderTexture,将深度值写入纹理中,这张深度纹理称为深度图;第二次从正常的相机视角渲染场景,在渲染场景时需要将当前像素到光源的距离与第一次渲染后的深度图中对应的深度信息作比较,如果距离比深度图中取出的值大就说明该点处于阴影中,如图7-16所示。

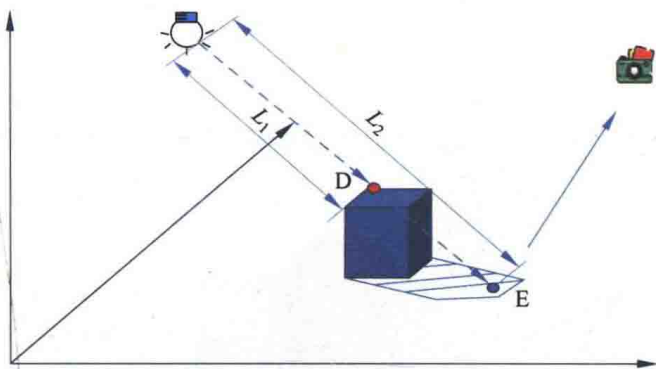


图 7-16 ShadowMap 技术原理示意图

在第一次渲染中,从光源视角出发,渲染的是光源位置可见的像素,将这些深度信息记录到深度图中,如D点对应的深度信息是 L_1 ;第二次从相机视角渲染场景,这次渲染的是相机位置可见的像素,如E点,计算E点到灯光的距离 L_2 ,将 L_2 与深度图中对应的深度值作比较,

从图 7-16 可以看到,在第一次渲染时,与 E 点对应的深度信息是 L_1 ,并且 $L_1 < L_2$,这说明从光源的视角看,E 点被其他点挡住了,因此 E 点不能被灯光照射到,即 E 点处于阴影中^①。

7.4.2 使用实时阴影

在使用阴影之前,我们还得阐述一下 Unity 中阴影质量的控制逻辑:首先在 URP 管线渲染设置中有全局阴影生成控制属性,这些设置具有全局效果,优先于光源对象中的阴影设置;其次光源对象也具有阴影生成的控制属性,负责该光源阴影生成的相关逻辑;再次投射阴影的对象上是否有投射阴影属性控制开关,控制该对象是否投射阴影。下面我们来分别了解一下各控制项的具体内容。

1. 阴影全局控制

在工程窗口 (Project 窗口) 中选择 URP 渲染管线配置文件 ARCorePipeline 对象^②,在属性窗口 (Inspector 窗口) 中将会展示出该配置文件的相关属性,如图 7-17 所示。

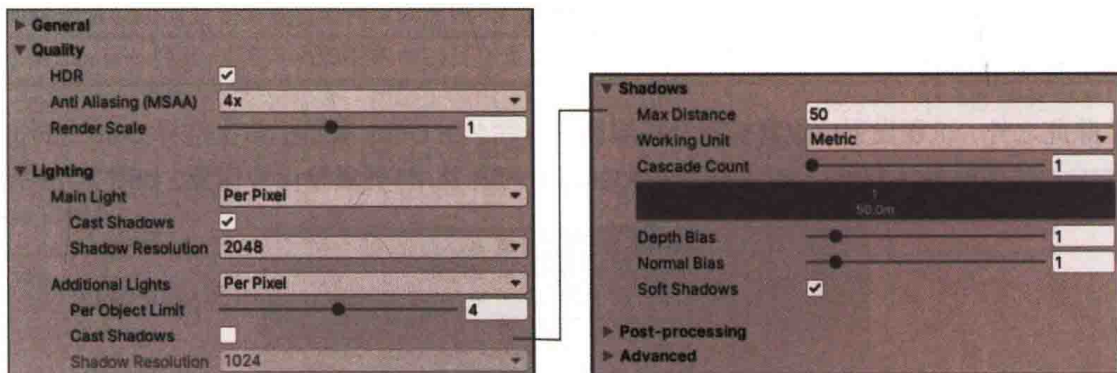


图 7-17 URP 阴影生成全局控制属性面板

在该属性面板中,与阴影生成和阴影质量相关的各属性含义如表 7-5 所示。

表 7-5 URP 阴影生成的质量控制属性

属 性		描 述
Anti Aliasing(MSAA)		抗锯齿,对阴影边缘质量有影响,可选择 Disable、2x、4x、8x,数据越高,抗锯齿效果越好,但性能消耗越大
Lighting	Main Light	设置主光源渲染模式,可选择 Disable、Per Pixel,即关闭光源作用或者使用逐像素光照
	Cast Shadows	是否投射阴影
	Shadow Resolution	生成 Shadowmap 的纹理尺寸,越大阴影质量越好,但性能消耗也越大

① 特别需要注意:距离比较一定要在同一坐标空间中才有意义,即示例中的 L_1 与 L_2 的大小对比一定要在同一坐标空间中对比。

② ARCorePipeline 为自定义的 URP 渲染配置文件,详见第 1 章。

续表

属 性		描 述
Lighting	Additional Lights	附属光源渲染模式，可选择 Disable、Per Vertex、Per Pixel，即关闭光源作用、逐顶点光照、逐像素光照
	Cast Shadows	是否投射阴影
	Shadow Resolution	生成 Shadowmap 的纹理尺寸，越大阴影质量越好，但性能消耗也越大
Shadows	Max Distance	阴影最大有效范围，即在此距离外，不生成阴影，用于节约资源
	Working Unit	工作单位，可选为 Metric、Percent，选择 Metric 时，单位为米
	Cascade Count	层叠阴影，取值范围 [1, 4]，更高的层叠阴影会带来更好的阴影边缘柔和效果，层叠阴影对性能消耗非常大
	Depth Bias	偏移量，范围为 [0, 10]，默认值为 1，描述阴影与模型的位置关系，值越大阴影与模型偏移越大，与 Normal Bias 结合使用可防止自阴影 (Shadow acne) 和阴影偏离 (peter-panning)
	Normal Bias	法线偏移量，范围为 [0, 10]，默认值为 1，该属性实际上用于调整表面坡度偏移量，坡度越大偏移量应该越大，但过大的偏移量又会导致阴影偏离 (peter-panning)，因此固定的偏移量需要法向偏移量来修正
	Soft Shadow	是否使用软阴影

除此之外，工程设置中也有控制全局阴影的属性，在 Unity 菜单栏依次选择 Edit → Project Settings，打开 Project Settings 对话框，选择 Quality 选项卡，打开质量设置面板，如图 7-18 所示。

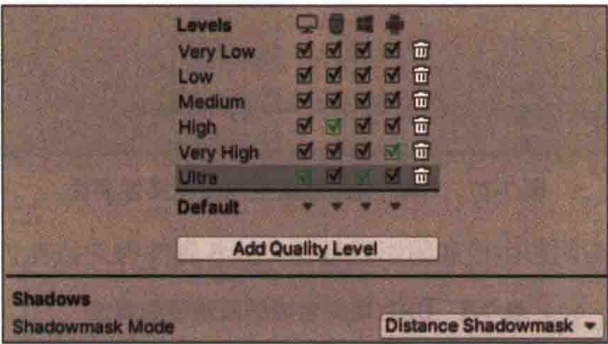


图 7-18 Unity 工程质量控制面板

在该面板中，可通过 Android 图标下的选框选择工程整体质量控制，整体质量可分为 Very Low、Low、Medium、High、Very High、Ultra，共 6 级，级别越高，阴影生成质量越好，但性能消耗也越大，在实际项目中，需要通过多次测试寻找一个平衡点。

阴影遮罩模式 (Shadowmask Mode) 只在光源设置为 Mixed (混合) 模式时才有用，可选值有 Shadowmask、Distance Shadowmask。Shadowmask 表示所有静态物体都使用烘焙阴影，Distance Shadowmask 与表 7-5 中的 Max Distance 配合使用，在 Max Distance 范围内使用实时阴影，在范围外使用烘焙阴影，因此，Distance Shadowmask 性能消耗比 Shadowmask 高。

2. 光源阴影控制

每个光源都可以独立设置其阴影生成模式和效果,但如果与全局设置冲突,则以全局设置为准。在层级窗口选择 Directional Light 对象,然后在属性窗口可以看到光源的阴影生成相关属性,如图 7-19 所示。

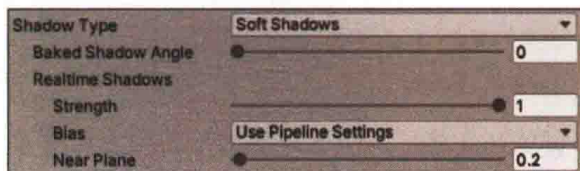


图 7-19 光源阴影生成相关属性

在 Directional Light 组件属性中,与阴影相关的属性的详细信息如表 7-6 所示。

表 7-6 光源光经常用阴影参数

属 性	描述
Shadow Type	阴影类型, 可选值为 No Shadow、Hard Shadow、Soft Shadow
Baked Shadow Angle	渲染阴影的角度, 只在光源设置为 Mixed、Baked 时才有效
Strength	阴影强度, 值越大阴影越浓, 值越小阴影越淡, 范围为 [0, 1], 默认值为 1
Bias	偏移量, 可独立设置偏移量, 也可以使用全局设置
Near Plane	近平面, 产生阴影的最小距离, 范围为 [0.1, 10], 调整该值可以调整阴影裁剪近平面

3. 对象阴影控制

在 URP 管线中, 每个虚拟对象都可以独立设置是否投射阴影, 但默认接受阴影, 因此可以独立控制每个对象对阴影的影响。

注意

阴影生成及阴影质量受上述所有因素共同影响, 而且需要强调的是, 阴影生成, 特别是高质量软阴影的生成是一项重度资源消耗型功能, 在实际项目中, 阴影生成及其质量控制要遵循适度原则, 在移动 AR 受限的资源条件下寻找一个合适的折中点、实现效果和性能的平衡非常关键。

如前所述, 在 AR 应用中生成阴影, 一方面需要有光源, 另一方面还需要有一个接受并显示阴影的载体。本节中将采用 Unity 提供的阴影解决方案生成 AR 实时阴影, 光源采用 Directional Light 类型, 使用一个 Quad 做阴影接受和显示载体。

首先, 制作一个接受阴影且透明的阴影接受对象。在层级窗口中新建一个空对象 (Create Empty), 命名为 ARPlane, 然后在其下新建一个 Quad 子对象, 在属性窗口中将其 Transform 组件 Rotation X 属性值修改为 90, 以使其水平放置, 将其 Scale 值修改为 (0.2,0.2,0.2)。

然后在工程窗口中新建一个材质, 命名为 ARShadow, 将使用的 Shader 设置为 AR → Shader Graphs → TransparentShadowReceiver^①, 并将 ARShadow 材质赋给 Quad 对象。

最后将 ARPlane 对象制作成预制体, 删除层级窗口中的 ARPlane 对象, 接受阴影的平面制作完成。

在工程窗口中新建一个 C# 脚本, 命名为 AppController, 编写代码如下:

```
// 第7章 / 7-2
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;

[RequireComponent(typeof(ARRaycastManager))]
public class AppController : MonoBehaviour
{
    public GameObject spawnPrefab;           // 虚拟物体预制体
    public GameObject ARPlane;               // 阴影接受平面
    static List<ARRaycastHit> Hits;          // 碰撞结果
    private ARRaycastManager mRaycastManager; // 组件对象
    private GameObject spawnedObject = null; // 实例化的虚拟物体对象
    private float mARCoreAngle = 180f;      // ARCore 需要在 Y 轴上旋转虚拟物体 180°
    private void Start()
    {
        Hits = new List<ARRaycastHit>();
        mRaycastManager = GetComponent<ARRaycastManager>();
    }

    void Update()
    {
        if (Input.touchCount == 0)
            return;
        var touch = Input.GetTouch(0);
        if (mRaycastManager.Raycast(touch.position, Hits, TrackableType.
            PlaneWithinPolygon | TrackableType.PlaneWithinBounds))
        {
            var hitPose = Hits[0].pose;
            if (spawnedObject == null)
            {
                spawnedObject = Instantiate(spawnPrefab, hitPose.position,
                hitPose.rotation);
                spawnedObject.transform.Rotate(Vector3.up, mARCoreAngle);
                var p = Instantiate(ARPlane, hitPose.position, hitPose.rotation);
                // 将接受阴影的平面作为虚拟物体子对象
            }
        }
    }
}
```

① 这里直接使用了 Unity 提供的 Shader Graph 阴影生成 Shader, 该 Shader Graph 文件及使用说明可参见本书配套代码文件。

```
p.transform.parent = spawnedObject.transform;  
}  
else  
{  
    spawnedObject.transform.position = hitPose.position;  
}  
}  
}
```

该脚本的主要功能是在检测到的平面上放置虚拟对象，在放置虚拟对象时，同时实例化了一个接受阴影的平面，并将该平面设置为虚拟对象的子对象，以便将其与虚拟对象绑定。将该脚本挂载在场景中的 AR Session Origin 对象上，并将虚拟物体与前文制作好的 ARPlane 预制体赋给相应属性。编译运行，寻找一个平面，在放置虚拟对象后，效果如图 7-20 所示。



图 7-20 硬阴影生成效果图

除硬阴影，Unity 还提供了一个柔和的 AR 软阴影 Shader，使用软阴影与使用硬阴影操作完全一样，只需将 ARShadow 材质的 Shader 选择为 Shader Graphs → BlurredShadowPlane，使用软阴影生成的阴影效果如图 7-21 所示。

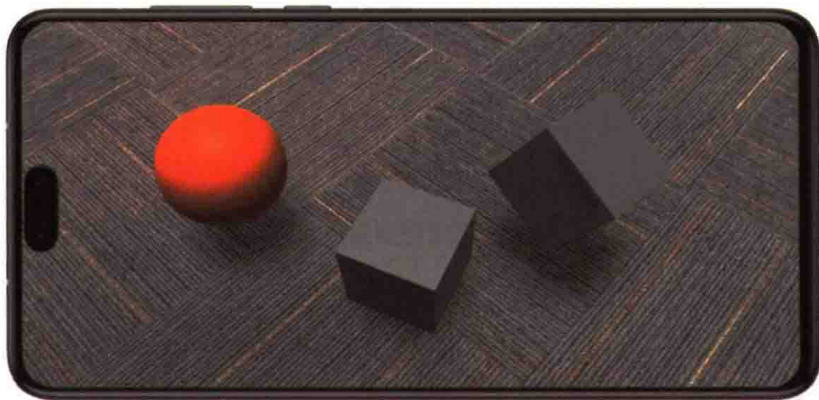


图 7-21 软阴影生成效果图

再次强调,使用阴影,特别是高质量的软阴影,可能会对性能造成非常大的影响,阴影使用应遵循适度原则。

7.5 Planar 阴影

Unity 内置的 Shadowmap 阴影生成方式具有普适性的特点,适用范围广,并且是通过物理和数学的方式生成阴影,因此阴影能够投射到自身及不规则非平滑的物体表面,与场景复杂度没有关系,但通过前面的学习已了解到,这种生成阴影的方法在带来较好效果的同时,性能开销也是比较大的,特别是在使用高分辨率高质量实时软阴影时,非常有可能成为性能瓶颈。

在某些情况下,可能并不需要那么高质量、高通用性的阴影,或者由于性能制约不能使用那么高质量的阴影,因此可能会寻求一个“适用于某些特定场合”的“看起来正确”的实时阴影以降低性能消耗。本节将学习一种平面投影阴影 (Planar Projected Shadow),以适应一些对性能要求非常高,以降低阴影质量换取性能的应用场景。

7.5.1 数学原理

在光源产生阴影时,阴影区域其实就是物体在投影平面上的投影区域,因此为了简化阴影渲染,可以直接将物体的顶点投射到投影平面上,并在这些投影区域中使用特定的颜色着色即可,如图 7-22 所示。

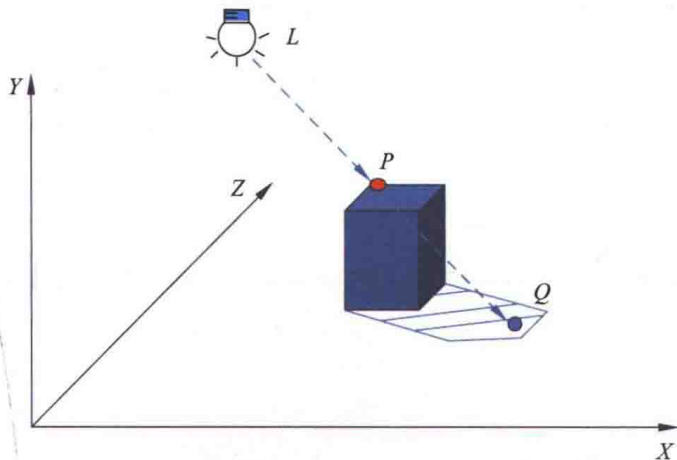


图 7-22 平面投影阴影产生原理图

因此,阴影计算转化为求物体在平面上投影的问题,求投影可以使用解析几何方法,也可以使用平面几何方法。在图 7-22 中,阴影投影计算可以转化为数学模型:已知空间内一个方向向量 $L(L_x, L_y, L_z)$ 和一个点 $P(P_x, P_y, P_z)$,求点 P 沿着 L 方向在指定平面上的投影位置 $Q(Q_x, Q_y, Q_z)$ 。

下面使用平面几何的计算方式进行推导,为了简化问题,只在二维空间内进行推导,三维空间可以类推。在二维空间中,图 7-22 所描述问题可以抽象成图 7-23 所示数学问题,即求点 P 沿 L 方向在直线 $y=h$ 上的投影 Q 。

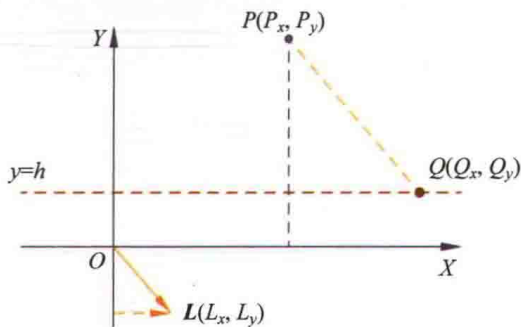


图 7-23 阴影数学计算原理图

观察图 7-23, 根据相似三角形定理, 可以得到下面公式:

$$-\frac{L_y}{P_y - h} = \frac{L_x}{Q_x - P_x} \quad (7-1)$$

因此

$$Q_x = P_x - \frac{L_x(P_y - h)}{L_y} \quad (7-2)$$

$$Q_y = h \quad (7-3)$$

坐标 $Q(Q_x, Q_y)$ 即为要求的投影点坐标, 推广到三维, 坐标 $Q(Q_x, Q_y, Q_z)$ 计算公式如下:

$$Q_x = P_x - \frac{L_x(P_y - h)}{L_y} \quad (7-4)$$

$$Q_y = h \quad (7-5)$$

$$Q_z = P_z - \frac{L_z(P_y - h)}{L_z} \quad (7-6)$$

公式 (7-4)、公式 (7-5)、公式 (7-6) 即为三维空间中的阴影投影 X 、 Y 、 Z 坐标公式。

7.5.2 代码实现

在有了计算公式以后, 就可以通过 Shader 对阴影进行渲染^①, 代码如下:

```
// 第7章 / 7-3
Shader "Davidwang/PlanarShadow"
{
    Properties{
        _ShadowColor("Shadow Color", Color) = (0.35,0.4,0.45,1.0)
```

^① 为与原有渲染流程统一, 使用平面投影阴影时, 只需要在原来着色器 Shader 中多写一个 pass 进行阴影渲染, 其余部分不用修改。

```

    }
    SubShader{
        Tags
        {
            "RenderPipeline" = "UniversalPipeline" "RenderType" = "Transparent" "Queue"
= "Transparent-1"
        }
        // 阴影 pass
        Pass
        {
            Name "PanarShadow"
            Tags { "LightMode" = "UniversalForward" }
            // 使用模板测试以保证 alpha 显示正确
            Stencil
            {
                Ref 0
                Comp equal
                Pass incrWrap
                Fail keep
                ZFail keep
            }
            // 透明混合模式
            Blend SrcAlpha OneMinusSrcAlpha
            // 关闭深度写入
            ZWrite off
            // 深度稍微偏移防止阴影与地面穿插
            Offset -1 , 0

            HLSLPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma exclude_renderers d3d11_9x

            #include "Packages/com.unity.render-pipelines.universal/
ShaderLibrary/Lighting.hlsl"
            struct appdata
            {
                float4 vertex : POSITION;
            };

            struct v2f
            {
                float4 vertex : SV_POSITION;
                float4 color : COLOR;
            };

            CBUFFER_START(UnityPerMaterial)

```



```

float4 _LightDir = float4(0.2,-0.8,0.3,1);
float4 _ShadowColor;
float _ShadowFalloff = 1.34f;
CBUFFER_END

float3 ShadowProjectPos(float4 vertPos)
{
    float3 shadowPos;
    // 得到顶点的世界空间坐标
    float3 worldPos = mul(unity_ObjectToWorld , vertPos).xyz;
    // 灯光方向
    float3 lightDir = normalize(_LightDir.xyz);
    // 阴影的世界空间坐标 (低于地面的部分不做改变)
    shadowPos.y = min(worldPos.y , _LightDir.w);
    shadowPos.xz = worldPos.xz - lightDir.xz * max(0 , worldPos.y -
_LightDir.w) / lightDir.y;
    return shadowPos;
}

v2f vert(appdata v)
{
    v2f o = (v2f)0;
    // 得到阴影的世界空间坐标
    float3 shadowPos = ShadowProjectPos(v.vertex);
    // 转换到裁切空间
    o.vertex = TransformObjectToHClip(shadowPos);
    // 得到中心点世界坐标
    float3 center = float3(unity_ObjectToWorld[0].w , _LightDir.w ,
unity_ObjectToWorld[2].w);
    // 计算阴影衰减
    float falloff = 1 - saturate(distance(shadowPos , center) * _
ShadowFalloff);
    // 阴影颜色
    o.color = _ShadowColor;
    o.color.a *= falloff;
    return o;
}

float4 frag(v2f i) : SV_Target
{
    return i.color;
}

ENDHLSL
}

FallBack "Hidden/Universal Render Pipeline/FallbackError"
}

```

在上述 Shader 脚本中, `_LightDir.xyz` 是光源方向, `_LightDir.w` 是接受阴影的平面高度, `_ShadowColor` 为阴影颜色。 `_LightDir` 可以在运行中通过 C# 脚本将值传进来, 在运行中动态传值是因为不能提前知道灯光方向和检测到的平面 (接受阴影) 高度信息^①。使用模板和混合是为了营造阴影衰减的效果, 如图 7-24 所示。



图 7-24 阴影衰减示意图

计算中心点世界坐标代码如下:

```
float3 center = float3( unity_ObjectToWorld[0].w , _LightPos.w , unity_ObjectToWorld[2].w);
```

该公式中 `unity_ObjectToWorld` 是 Unity 内置矩阵, 这个矩阵各行的第 4 个分量分别对应物体 Transform 属性中的 X、Y、Z 值。

```
float falloff = 1-saturate(distance(shadowPos , center) * _ShadowFalloff);
```

该语句所做工作是计算衰减因子, 以便于后续进行阴影混合。AR 中平面投影阴影使用时需要动态地传递灯光与平面相关信息, 新建一个 C# 脚本, 命名为 `PlanarShadowManager`, 编写代码如下:

```
// 第 7 章 / 7-4
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;

[RequireComponent(typeof(ARRaycastManager))]
public class PlanarShadowManager : MonoBehaviour
{
    public GameObject spawnPrefab; // 虚拟对象预制体
```

^① AR 会话空间坐标系在运行时才能建立。

```

public GameObject ARPlane;           // 阴影接受平面
public Light mLight;                 // 主光源

static List<ARRaycastHit> Hits;      // 射线检测碰撞结果
private ARRaycastManager mRaycastManager; // 射线检测组件
private GameObject spawnedObject = null; // 实例化后的虚拟对象
private float mARCoreAngle = 180f;    // 纠正 ARCore 与 Unity 左右手坐标系差异
private List<Material> mMatList = new List<Material>(); // 虚拟对象材质对象
private float mPlaneHeight = 0.0f;    // 阴影平面高度

private void Start()
{
    Hits = new List<ARRaycastHit>();
    mRaycastManager = GetComponent<ARRaycastManager>();
}

void Update()
{
    if (Input.touchCount == 0)
        return;
    var touch = Input.GetTouch(0);
    // 射线与检测到的平面碰撞检测
    if (mRaycastManager.Raycast(touch.position, Hits, TrackableType.
PlaneWithinPolygon | TrackableType.PlaneWithinBounds))
    {
        var hitPose = Hits[0].pose;
        if (spawnedObject == null)
        {
            spawnedObject = Instantiate(spawnPrefab, hitPose.position,
hitPose.rotation);
            spawnedObject.transform.Rotate(Vector3.up, mARCoreAngle);
            spawnedObject.transform.Translate(0, 0.02f, 0);
            var p = Instantiate(ARPlane, hitPose.position, hitPose.
rotation);
            mPlaneHeight = hitPose.position.y;
            p.transform.parent = spawnedObject.transform;

            GameObject spider = GameObject.FindGameObjectWithTag("spider");
            SkinnedMeshRenderer[] renderlist = spider.GetComponentsInChildre
n<SkinnedMeshRenderer>();
            // 获取虚拟对象所有使用的材质
            foreach (var render in renderlist)
            {
                if (render == null)
                    continue;
                mMatList.Add(render.material);
            }
        }
    }
}

```



```

        if (spawnedObject != null)
            UpdateShader();
    }
    else
    {
        spawnedObject.transform.position = hitPose.position;
        spawnedObject.transform.Translate(0, 0.02f, 0);
        // 加个偏移, 防止阴影闪烁
        mPlaneHeight = hitPose.position.y + 0.02f;
        // 更新主光源
        if (spawnedObject != null)
            UpdateShader();
    }
}

// 更新主光源和 Shader 中的光源
private void UpdateShader()
{
    Vector4 projdir = new Vector4(mLight.transform.forward.x, mLight.
transform.forward.y, mLight.transform.forward.z, mPlaneHeight);
    foreach (var mat in mMatList)
    {
        if (mat == null)
            continue;
        // 设置主光源
        mat.SetVector("_LightDir", projdir);
    }
}
}

```

该脚本代码比较简单, 在实例化虚拟物体和修改虚拟物体位姿时会将光源和阴影平面相关信息传递到 Shader 着色器中, 使用平面投影渲染的阴影如图 7-25 所示。



图 7-25 平面投影阴影效果图

从图 7-25 可以看到,平面投影阴影的渲染效果整体还是不错的,更关键的是性能比使用内置实时软阴影提升巨大,这也是当前很多移动端游戏使用平面投影阴影代替 Shadowmap 阴影的主要原因。

从平面投影阴影数学原理可以看到,该阴影只会投影到指定的平整平面上,不能投影到凹凸不平的平面。平面投影阴影不会将阴影投射到其他虚拟对象上,同时该阴影也不会自行截断,即会穿插到其他虚拟物体或墙体里面去,但在 AR 中,这些问题都不是很严重的问题,为了提高移动端 AR 应用的性能使用此阴影生成方式是一个不错的选择。

7.6 伪阴影

在前述章节中,实现的阴影都是实时阴影,阴影会根据虚拟物体的形状、位置、光源的变化而变化,实时阴影在带来更好适应性的同时也会消耗大量计算资源,特别是在移动设备上,这会挤占其他功能的可用资源,严重时会造成应用卡顿。在 VR 中,可以将光照效果烘焙进场景中以达到提高性能的目的,然而由于 AR 场景源于真实环境,无法预先烘焙场景^①,但如果 AR 应用中的虚拟物体是刚体,不发生形变,并且不脱离 AR 平面时,则可以采用预先制作阴影的方法实现阴影效果,利用这种方式制作的阴影不会消耗计算资源,非常高效。

所谓预先制作阴影,就是在虚拟物体下预先放置一个平面,平面所使用的渲染纹理是与虚拟物体匹配的阴影纹理,以此来模拟阴影。因为这个阴影使用图像的方法实现,所以最大的优势是不浪费计算资源,其次是阴影可以根据需要自由处理成硬阴影、软阴影、超软阴影、斑点阴影等类型,自主性强。缺点是阴影一旦设定后在运行时不能依据环境的变化而产生变化,不能对环境进行适配,由于是预先制作阴影,为了更好的效果,虚拟物体不能脱离 AR 平面,如果虚拟物体允许悬空则不适用该方法。

下面演示伪阴影的使用方法,为了防止 Unity 内置阴影产生干扰,首先关闭场景中 Directional Light 光源阴影,即在层级窗口中选择 Directional Light 对象,在属性窗口中将 Light 组件下的 Shadow Type 属性选择为 No Shadows 值。

然后制作一张与虚拟物体相匹配的阴影效果图,通常这张图应该是带 Alpha 通道的 PNG 格式阴影纹理图,如图 7-26 所示。

新建一个材质,命名为 FakeShadow,材质使用 Universal Render Pipeline → Simple Lit 着色器,并将其 Surface Type 属性设置为 Transparent 值以实现透明效果,Base map 纹理使用制作好的阴影纹理。

在层级窗口中新建一个空对象,命名为 ARPlane,并将虚拟物体模型作为子对象放置在其下。在 ARPlane 下创建一个 Quad 子对象,将其 X 轴旋转 90° 以使其平铺,将上文制作的 FakeShadow 材质赋给它,并调整 Quad 对象与模型对象的相对关系,使阴影与虚拟物体匹配,如图 7-27 所示。

^① 在复杂 AR 静态场景中,包括物体自身阴影,也可以采用光照烘焙方法提高性能。

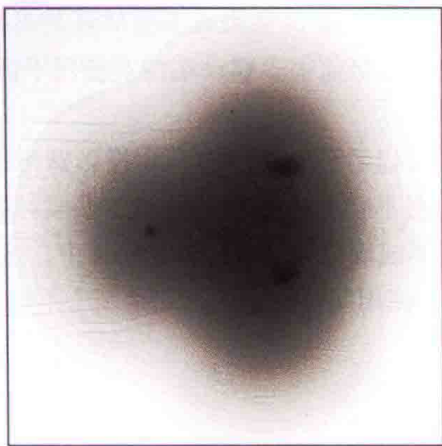


图 7-26 制作一张带 Alpha 通道的 PNG 格式阴影纹理图



图 7-27 制作带阴影的预制体

制作完成后将 ARPlane 对象制作成预制体，并删除层级窗口中的 ARPlane 对象。编译运行，寻找一个平面，在放置虚拟对象后，效果如图 7-28 所示。



图 7-28 伪阴影实测效果图

通过预先制作阴影的方法，避免了在运行时实时计算阴影，没有阴影计算性能开销，并且可以预先制作阴影贴图，实现所需的阴影类型，通过图 7-28 可以看到，这个阴影效果在模拟一般普通环境下的阴影时非常有效。

第 8 章



持久化存储与多人共享

到现在为止，前面章节中的所有 AR 应用案例都无法持久化地存储运行数据，在应用启动后扫描检测到的平面、加载的虚拟物体、环境特征、设备姿态等都会应用关闭后丢失。在很多应用背景下，这种模式是可以被接受的，每次打开应用都会是一个崭新的应用，不受前一次操作的影响，但对一些需要连续间断性进行的应用或者多人共享的应用来讲这种模式就有很大的问题，这时更希望应用能保存当前状态，当再次进入时能直接从当前状态进行下一步操作而不是从头再来。除此之外，希望能与其他人共享 AR 体验，实现互动。本章中将主要学习 AR 应用数据持久化存储和跨设备多人共享 AR 体验相关知识。

8.1 云锚点

锚点是 AR、MR 技术中最重要的概念之一，任何需要锚定到现实空间、二维图像、三维物体、人体、人脸的虚拟对象都需要通过特定的锚点连接^①，持久化存储、共享 AR/MR 体验也必须通过锚点实现。

从第 4 章中已经知道，锚点（Anchor）是指将虚拟物体固定在 AR 空间上的一种技术，通过锚点可以确保物体在空间中保持相同的位置和方向。云锚点（Cloud Anchor），顾名思义，是指存储在云端服务器上的锚点，借助于云锚点，可以让同一环境中的多台 ARKit 设备 / ARCore 设备同步共享 AR 体验。每一台设备都可以将云锚点添加到其应用中，通过读取、渲染连接到云锚点上的三维对象，就可以在同样的空间位置看到相同的虚拟内容，同时，每一台设备也都可以创建云锚点并托管到云端，其他设备也可以同步这些锚点从而实现 AR 体验共享。

1. 锚点托管

锚点信息与其周围真实环境密切关联，在生成云锚点时，设备会将锚点信息及其周边的真实环境特征信息（包括特征点和描述符）从设备本地端发送到云端服务器上，这些数据

^① 在 AR Foundation 中，很多锚点不需要开发人员手工创建，而是由系统统一管理。

上传后会被处理成稀疏的点云图,并存储在云端,这个过程称为锚点托管。具体而言,在托管锚点时,ARCore 会将环境视觉特征数据从用户设备发送到谷歌云平台 (Google Cloud Platform, GCP),上传到服务器后,这些数据会被处理成稀疏的点云图,类似于 ARCore 点云。由于点云的稀疏性,其他人员无法根据稀疏点云确定用户的地理位置或者重建与用户物理环境相关的图像,有利于安全和保护隐私。

2. 锚点解析

当有设备向服务器请求云锚点时,该设备会将其摄像头图像中的真实环境特征信息发送到云端服务器,服务器则会尝试将这些图像特征信息与存储在云端的稀疏点云图进行匹配,如果匹配成功,则返回该位置的云锚点信息,利用锚点信息,请求设备即可恢复连接到该锚点上的三维对象,由于锚点信息与其周围的真实环境信息相互关联,因此恢复的锚点也会位于真实环境中相同的位置,并保持一样的方向,这个过程称为锚点解析。

当一个用户向云服务器发起解析请求时,如果解析成功,则 AR 应用会利用服务器返回的相关信息恢复云锚点及其连接的虚拟物体在场景中的姿态,因此每个用户在他们的设备上都能看到场景中的虚拟对象,达到锚点信息及连接在其上虚拟对象共享的目的。

云锚点托管及解析原理示意图如图 8-1 所示。

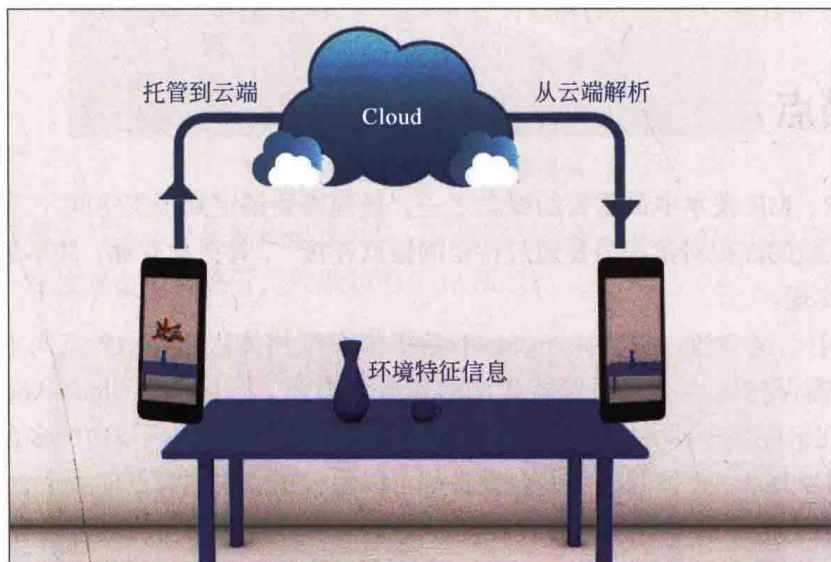


图 8-1 云锚点工作原理示意图

8.2 ARCore 云锚点使用实例

利用云锚点可以持久化存储 AR 应用进程数据,云锚点本质上是将用户场景对象的状态信息转换为可存储可传输的形式(序列化)保存到文件系统或者服务器中,当用户再次加载这些对象状态信息后即可恢复应用进程。云锚点不仅包含某一空间位姿信息,还包含场景特

征信息,通过锚点解析,可以允许两个不同坐标系统之间的匹配(托管和解析锚点时两次 AR 空间坐标系不同),恢复 AR 应用进程,共享 AR 体验。

如图 8-2 所示,设备 1 的 AR 应用在进行环境扫描、平面检测、虚拟物体放置等相关操作后,可以将其当前应用场景中的对象状态信息、环境特征点信息、设备姿态信息序列化保存到文件系统或数据库系统中。稍后,设备 1 或者设备 3 可以加载这些信息以恢复设备 1 之前的应用进程,还原虚拟物体^①。设备 1 也可以将对象状态信息、环境特征点信息、设备姿态信息序列化后通过网络传输给设备 2,设备 2 接收并加载这些信息后也可以恢复设备 1 之前的应用进程,从而达到共享体验的目的。

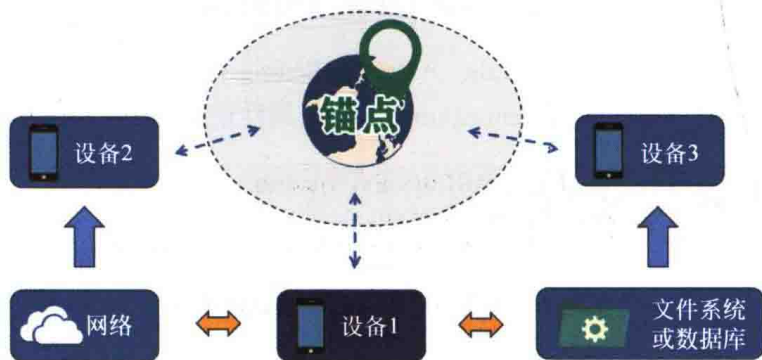


图 8-2 通过云锚点共享 AR 场景示意图

不管是将场景特征信息、设备姿态信息存储到文件系统还是通过网络传输到服务器,都需要对数据进行序列化,在读取数据后需要进行反序列化还原对象信息,如图 8-3 所示。序列化是一个将结构对象转换成字节流的过程,序列化使对象信息更加紧凑,更具可读性,同时降低错误发生概率,有利于通过网络传输或者在文件及数据库中持久化保存。在 ARCore 改进后的云锚点托管/解析中,这个过程已经被隐藏,开发人员无须关注传输处理过程。

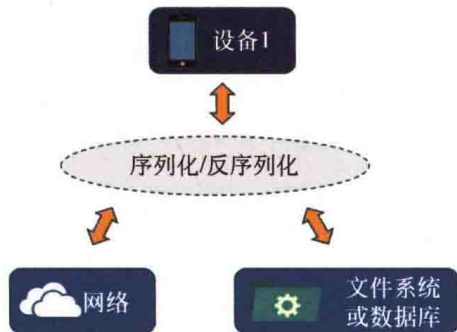


图 8-3 序列化与反序列化示意图

^① 目前 ARCore 不支持直接通过文件系统或者网络将云锚点发送给其他设备,由其他设备自行进行锚点解析的操作,但从原理上讲,这是可行的,事实上,ARKit 提供了 ARWorldMap 对象,可以完成设备间的云锚点解析。

使用 ARCore 云锚点需要 ARCore 扩展包的支持^①，在工程窗口 Config 文件夹中选择 ARCoreExtensionsConfig 配置文件，然后在属性窗口中将 Cloud Anchor Mode 属性设置为 Enabled 值，如图 8-4 所示。

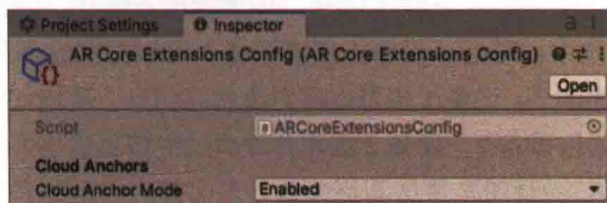


图 8-4 开启 ARCore 云锚点服务

在 Unity 菜单栏中，依次选择 Edit → Project Setting 打开工程设置窗口，选择 XR → ARCore Extensions，将 Android Authentication Strategy 属性设置为 Keyless 值，如图 8-5 所示。

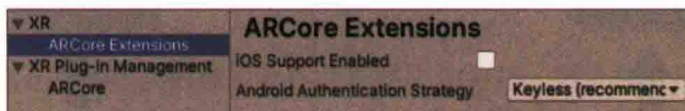


图 8-5 设置 ARCore 云锚点认证策略

Android Authentication Strategy 属性用于设置 ARCore 云锚点的存储认证策略，有 3 个可选项，即 Do Not Use、Keyless、API Key。默认为 Do Not Use，即不使用 ARCore 云锚点服务，禁用了云锚点功能，防止应用调用相应的库造成无谓的性能损失。

相比于上一个云锚点版本，新的 ARCore 云锚点版本大幅度简化了开发人员操作，也无须开发人员进行 GCP 注册申请，只需将 Android Authentication Strategy 属性设置为 Keyless，后续完全不需要进行其他处理，使用 Keyless 值可以将云锚点在 GCP 上托管 1 ~ 365 天。

API Key 值主要用于 iOS 端使用 ARCore 云锚点服务，相比于 Keyless 值，使用 API Key 需要开发人员预先在 GCP 上申请托管账号，获取 API Key 字符串，而且云锚点最多只能托管一天。

经过上述操作，已经设置好了使用 ARCore 云锚点的相应配置，新创建一个 C# 脚本，命名为 ARCloudAnchorManager，编写代码如下：

```
// 第 8 章 / 8-1
using System.Threading.Tasks;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;
using Google.XR.ARCoreExtensions;
using UnityEngine.UI;
```

① 关于 ARCore 扩展包导入参见第 3 章。

```

[RequireComponent(typeof(ARRaycastManager))]
[RequireComponent(typeof(ARAnchorManager))]
public class ARCloudAnchorManager : MonoBehaviour
{
    public GameObject mObjectPrefab;           // 需要放置的游戏对象预制体
    public Button BtnHost;                     // 存储云锚点
    public Button BtnResolve;                  // 解析云锚点

    private GameObject mInstantedObject = null; // 实例化后的虚拟对象
    private List<ARRaycastHit> Hits;           // 射线检测结果
    private ARRaycastManager mRaycastManager;
    private ARAnchorManager mAnchorManager;
    private string cloudAnchorID = "云锚点ID"; // 云锚点ID
    private Camera mCamera;                   // AR主渲染相机

    void Awake()
    {
        mRaycastManager = GetComponent<ARRaycastManager>();
        mAnchorManager = GetComponent<ARAnchorManager>();
        Hits = new List<ARRaycastHit>();
        BtnHost.onClick.AddListener(HostCloudAnchor);
        BtnResolve.onClick.AddListener(ResloveCloudAnchor);
        mCamera = Camera.main;
    }

    void Update()
    {
        if (Input.touchCount < 1)
            return;
        var touch = Input.GetTouch(0);
        if (mInstantedObject == null && mRaycastManager.Raycast(touch.position,
            Hits, TrackableType.PlaneWithinPolygon | TrackableType.PlaneWithinBounds))
        {
            var hitPose = Hits[0].pose;
            mInstantedObject = Instantiate(mObjectPrefab, hitPose.position,
                hitPose.rotation);
            mInstantedObject.AddComponent<ARAnchor>();
        }
    }

    // 托管云锚点
    async private void HostCloudAnchor()
    {
        // 异步处理特征点云采集及托管操作
        await HostCloudAnchorAsync();
    }

    // 解析云锚点
    async private void ResloveCloudAnchor()

```

```

{
    if (string.IsNullOrEmpty(cloudAnchorID))
        return;
    // 异步解析
    await ResloveCloudAnchorAsync();
}
// 异步处理云锚点解析操作
async private Task ResloveCloudAnchorAsync()
{
    var cloudAnchor = mAnchorManager.ResolveCloudAnchorId(cloudAnchorID);
    if (cloudAnchor != null)
    {
        while (true)
        {
            if (cloudAnchor.cloudAnchorState == CloudAnchorState.
TaskInProgress)
            {
                await Task.Delay(330);
            }
            else if (cloudAnchor.cloudAnchorState == CloudAnchorState.Success)
            {
                if (mInstantedObject == null)
                {
                    mInstantedObject = Instantiate(mObjectPrefab,
cloudAnchor.pose.position, cloudAnchor.pose.rotation);
                }
                else
                {
                    mInstantedObject.transform.position = cloudAnchor.pose.
position;
                    mInstantedObject.transform.rotation = cloudAnchor.pose.
rotation;
                }
                break;
            }
            else
            {
                Debug.Log("云锚点解析失败");
                break;
            }
        }
    }
}
// 异步处理云锚点托管操作
async private Task HostCloudAnchorAsync()
{
    if (mInstantedObject == null)

```



```

        return;
    while (true)
    {
        FeatureMapQuality quality = mAnchorManager.EstimateFeatureMapQuality
        ForHosting(new Pose(mCamera.transform.position, mCamera.transform.rotation));
        if (quality != FeatureMapQuality.Sufficient)
        {
            await Task.Delay(330);
            Debug.Log(" 环境特征采集进度: " + quality.ToString());
        }
        else
            break;
    }
    var cloudAnchor = mAnchorManager.HostCloudAnchor(mInstantableObject.
    GetComponent<ARAnchor>(), 7);
    if (cloudAnchor != null)
    {
        while (true)
        {
            if (cloudAnchor.cloudAnchorState == CloudAnchorState.
TaskInProgress)
            {
                await Task.Delay(330);
            }
            else if (cloudAnchor.cloudAnchorState == CloudAnchorState.
Success)
            {
                var id = cloudAnchor.cloudAnchorId;
                Debug.Log(" 云锚点存储成功, 云锚点 ID: " + id);
                break;
            }
            else
            {
                Debug.Log(" 云锚点存储失败 ");
                break;
            }
        }
    }
}
}
}

```

在上述代码中, 引入了 Google.XR.ARCoreExtensions 命名空间, 在该命名空间中, ARCore 扩展了 ARAnchorManager 组件功能, 更方便我们使用。

在采集云锚点时, 需要设置一个参考对象, 该参考对象必须挂载 ARAnchor 组件, 然后调用 EstimateFeatureMapQualityForHosting() 方法评估采集的环境特征质量情况, 该方法原型如下:

```
FeatureMapQuality quality = ARAnchorManager.EstimateFeatureMapQualityForHosting(pose)
```

通过方法原型可以看到,该方法要求传入一个 Pose 参数,这个 Pose 即为采集图像信息时设备摄像头位姿,通过这个参数也能看出,该方法只能就某一个设备摄像头位姿采集的环境图像特征信息质量进行评估^①,实际应用时,需要确保图像采集摄像头在不同位姿都有比较好的质量。该方法返回 FeatureMapQuality 枚举值,该枚举用于标示采集图像的质量,其具体枚举值如表 8-1 所示。

表 8-1 FeatureMapQuality 枚举值

枚举值	描 述
Insufficient	采集图像的摄像头在某个位姿采集的环境特征比较少,不足以重定位和解析云锚点
Sufficient	云锚点可以托管和被解析,但可能重定位精度会有所降低
Good	环境特征信息非常好,能准确地进行云锚点解析和重定位

在采集到足够的环境特征信息后,直接使用 HostCloudAnchor() 方法即可将云锚点信息托管到 GCP 中,该方法的一个参数为挂载在参考对象上的 ARAnchor 组件,另一个参数为云锚点需要在云服务器上存储的时间。取值范围为 [1,365]。云锚点托管无须开发人员处理序列化、网络通信等相关工作,并且为异步方法,不会阻塞主线程,执行后返回 ARCloudAnchor 类型对象,其 CloudAnchorState 成员属性标识了云锚点托管执行结果,CloudAnchorState 成员也是一个枚举,其枚举值如表 8-2 所示。

表 8-2 CloudAnchorState 枚举值

枚举值	描 述
None	状态不确定
TaskInProgress	任务正在执行中
Success	托管成功
ErrorInternal	发生内部错误,任务失败
ErrorNotAuthorized	未授权,任务失败
ErrorResourceExhausted	服务器资源耗尽,任务失败
ErrorHostingDatasetProcessingFailed	托管处理时出错,任务失败
ErrorResolvingCloudIdNotFound	解析云锚点时无云锚点 ID,任务失败
ErrorResolvingPackageTooOld	解析云锚点版本太旧 (ARCore v1.24 之前的版本解析新服务接口),任务失败
ErrorResolvingPackageTooNew	解析云锚点版本太新 (ARCore v1.24 之后的版本解析旧服务接口),任务失败
ErrorHostingServiceUnavailable	托管服务器不可用,通常是因为网络异常

^① 亦即说明,单独使用该方法只能保证某一个摄像头位姿采集的图像质量已达到云锚点在该方向重定位的可靠性,如果重定位时采集图像信息的摄像头位姿与该特定位姿差距较大,则无法完成云锚点的定位恢复,所以为了从各个方向都能较好地恢复云锚点,则需要确保采集图像时设备摄像头在不同位姿都有比较好的质量。本节代码为了演示原理,只确保单个位姿质量达标即终止了环境特征采集。

云锚点托管成功后会返回一个云锚点 ID (Cloud Anchor ID)，这是一个 GUID 值，唯一标识托管的云锚点，在需要解析时，需要将该 ID 提交到 GCP 由服务器进行解析。

使用 ARCore 云锚点解析时非常简单，只需将需要解析的云锚点 ID 由 `ResolveCloudAnchorId()` 方法提交解析请求，当解析成功后，可直接返回云锚点在当前 ARCore 世界空间中的位姿 (Pose)，利用这个位姿即可恢复原 AR 场景，实现 AR 场景的持久化存储。

由上述代码可以看到，新版本的 ARCore 云锚点大幅度简化了开发人员操作，使用方法简单便捷，也不用处理复杂的 GCP 申请、认证、坐标系转换等操作，实用性进一步提高。

在使用时，首先在层级窗口中添加 ARCore 扩展对象并设置好相应配置值^①，然后在层级窗口中选择 AR Session Origin 对象，将该脚本挂载于此对象上，并设置好相关属性即可使用 ARCore 云锚点功能。

8.3 ARCore 云锚点使用注意事项

ARCore 云锚点托管 / 解析会增加内存、CPU、电池消耗，其间会将数据上传到 GCP，涉及网络通信、锚点的托管与解析，过程相对比较复杂，在使用云锚点的过程中的注意事项如表 8-3 所示。

表 8-3 云锚点使用注意事项

类 型	描 述
数据存储和访问限制	因为云锚点使用 GCP，而 GCP 对用户数据有存储和访问要求，所以包括以下几点： (1) 托管锚点时上传至云端的原始视觉特征信息在过期后舍弃。 (2) 锚点会根据存储的稀疏点云图在服务器端解析，生成稀疏的点云图可用于云锚点解析请求，托管的视觉特征信息不会发送至用户设备。 (3) 无法根据稀疏点云图确定用户的地理位置或者重建任何图像和用户的物理环境。 (4) 任何时候都不会存储请求中用于解析锚点的视觉特征描述信息
一般原则	(1) 避免在发光面上托管或解析云锚点。为了获得最佳结果，避免使用反光面或不具有视觉特征的平面，例如空白的光滑白墙，因为 ARCore 很难提取这些环境的特征点信息，特征点信息不足将导致锚点无法托管和解析。 (2) 确保云锚点采集与解析时光线充足。 (3) 云锚点托管和解析请求时光照条件应当一致。同一环境下不同光照条件会影响 ARCore 对特征点的提取，进而影响锚点的托管与解析
托管锚点	在托管云锚点之前，为了便于 ARCore 更好地理解周边环境，需要注意以下事项： (1) 从不同的角度观察锚点，采集不同角度的视觉特征信息。 (2) 围绕锚点至少移动几秒，确保采集到足够多的视觉性信息。 (3) 确保采集设备 (手机) 离锚点不要太远

^① 具体详情参见第 3 章。

续表

类 型	描 述
解析锚点	(1) 确保设备靠近托管锚点的位置, 如果请求锚点的设备离云锚点的位置过远, 则云锚点可能无法正确解析。 (2) 避免在相似度高的环境中采集和解析锚点, 如果用户手机摄像头对着的位置与托管云锚点的位置不同, 但看上去视觉特征相同, 云锚点可能无法正确解析, 因为这些相似度很高的周边环境会让 ARCore 提取相同的特征点, 这可能会得到错误的解析结果
性能提示	(1) 启用云锚点后, 内存使用将增加。 (2) 由于使用网络且内存需求增加, 所以电池消耗也会增加

Google GCP 为 ARCore 云锚点使用带宽提供如表 8-4 所示配额, 同一设备 IP 或项目将受到表中所示限制的影响。

表 8-4 带宽限制

配额类型	最大值	持续时间	适用对象
锚点数量	无限制	不适用	项目
锚点托管请求	30	分钟	用户 IP 和项目
锚点解析请求	300	分钟	用户 IP 和项目

云锚点托管和解析需要向 GCP 发送数据, 并且有一定的时延 (取决于网络), 因为托管和解析需要连接到互联网以向 GCP 发送和接收云锚点信息, ARCore 检测到的环境特征点越多, 解析就越容易成功, 但是要发送和接收的数据量也将会大幅增加, 从而造成网络通信需求和延时增加。

ARCore 云锚点并没有直接提供修改、获取云锚点 ID、删除云锚点的代码接口, 但提供了 oauth2l 工具进行服务器端云锚点的管理, 具体可参阅官方文档。

8.4 Azure 云服务概述

计算机与互联网已成为当今社会最重要的基础设施之一, 信息服务设备也已经成为商业运营最重要的必要设施, 拥有强大处理能力、高可靠性的云服务应运而生, 云服务提供全天候的硬件资源与软件应用, 降低了信息服务运作成本, 并且随着 5G 通信、小型设备、可穿戴设备的发展, 云服务的功能特性也得到极大的加强, 使用者可以在任何地方随时调用资源, 用完释放之后又可供再分配, 避免了资源浪费, 降低了对小型和可穿戴设备的硬件要求。

AR 应用持久化存储除了使用 ARCore 云锚点服务, 也可以使用微软公司的 Azure 云服务, Azure 云服务是微软公司基于其云计算操作系统的云服务技术, 它使用微软全球数据中心的存储、计算能力和网络基础服务, 主要目标是为开发者提供一个平台, 帮助简化开发可运行在云服务器、数据中心、边缘设备、Web 和 PC 计算机上的应用程序。Azure 以云技术为核心, 提供了软件 + 服务 + 计算的服务能力, 能够将处于边缘的开发者个人能力, 同微软全球数据中心网络托管的服务 (例如存储、计算和网络基础设施服务) 紧密结合起来。Azure 云服务功

能强大,本节只对 ARCore 使用其空间定位点功能进行阐述。

8.5 Azure 空间定位点使用实例

在 Azure 云中,云锚点被称为空间锚点(Spatial Anchor),Azure 云服务空间定位点^①(Azure Spatial Anchor)也能稳健地实现支持 ARCore 安卓设备之间的 AR 体验共享和持久化存储,当使用多个云锚点时,利用 Azure 云服务,还可以实现路径导航,这对很多室内应用都有非常重要的价值。Azure 云服务对云锚点提供了非常好的支持,本节将阐述使用 Azure 云服务实现云锚点的详细步骤。

使用 Azure 云服务中的云锚点,首先需要在 Azure 云服务中创建空间定位点资源(假定读者已注册个人 Azure 账号):登录 Azure 门户^②,在界面左栏中选择“创建资源”,并在随后打开的右侧页面中检索“空间定位点”,如图 8-6 所示。



图 8-6 创建资源、检索“空间定位点”

在检索结果中选择“空间定位点”服务,然后选择“创建”,填写资源名称、资源组、位置等信息,中国区建议选择东南亚区(Southeast Asia),如图 8-7 所示,最后单击“创建”开始创建云锚点资源。

当资源创建完成后,在部署完成界面中选择“转到资源”即可查看云锚点资源信息,将“账号域”和“账号 ID”信息保存下来以备使用,如图 8-8 所示。

在左侧栏中选择“设置”→“访问密钥”,将主密钥也保存下来备用,如图 8-9 所示。

^① 在 Azure 中称为空间定位点,本书根据其作用亦将其称为云锚点。

^② 网址为 <https://portal.azure.com/>。



图 8-7 设置云锚点资源信息

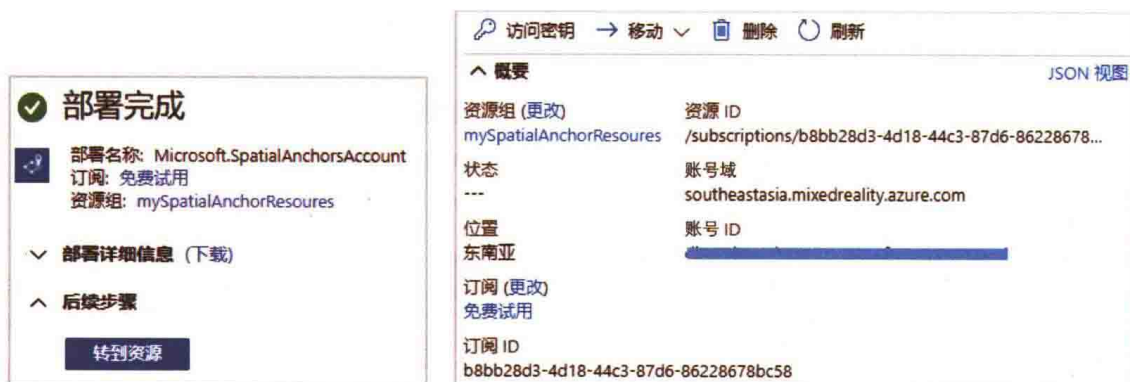


图 8-8 查看记录账号域、账号 ID 信息



图 8-9 查看记录云锚点密钥信息

至此, Azure 云锚点空间创建完成, 云端的设置工作也完成了。使用 Azure 云锚点服务时, 需要在工程中添加 Azure 空间锚点 SDK (Azure Spatial Anchors SDK), 由于 Azure 空间锚点 SDK 工具包目前并非 Unity 官方提供的包, 因此首先需要在清单文件 (manifest.json) 中加入包源地址并且在 Unity 中进行注册。在 Windows 文件管理器中打开 Unity 工程文件路径, 使用 VS 2019 打开项目清单 Packages/manifest.json 文件, 将 Azure 空间锚点工具包 (Azure Spatial

Anchors 工具包)所使用的包名和版本号添加到依赖项(dependencies)中,代码如下:

```
// 第8章/8-2
"dependencies": { //manifest.json 文件原依赖项声明行
  "com.microsoft.azure.spatial-anchors-sdk.android": "file:MixedReality/com.
microsoft.azure.spatial-anchors-sdk.android-2.11.0.tgz",
  "com.microsoft.azure.spatial-anchors-sdk.core": "file:MixedReality/com.microsoft.
azure.spatial-anchors-sdk.core-2.11.0.tgz",
  ...
}
```

完成所有配置后,保存文件,返回 Unity 软件,Unity 将会自动加载 Azure 空间锚点工具包^①,在菜单中依次选择 Window → Package Manager,打开包管理器窗口,可以在该管理器面板中进行工具包的升级、卸载管理。

在 ARCore 中使用 Azure 云锚点的基本步骤流程如图 8-10 所示。

用户 A 开启云会话,创建云锚点,关闭云会话;

用户 B 开启云会话,根据云锚点 ID 进行检索、重定位、解析云锚点,根据需要决定是否删除云锚点,关闭云会话。

用户 A 和用户 B 可以是同一台设备,也可以是不同设备,同时设备本身、设备之间也可以通过文件系统或者网络通信传递云锚点信息。

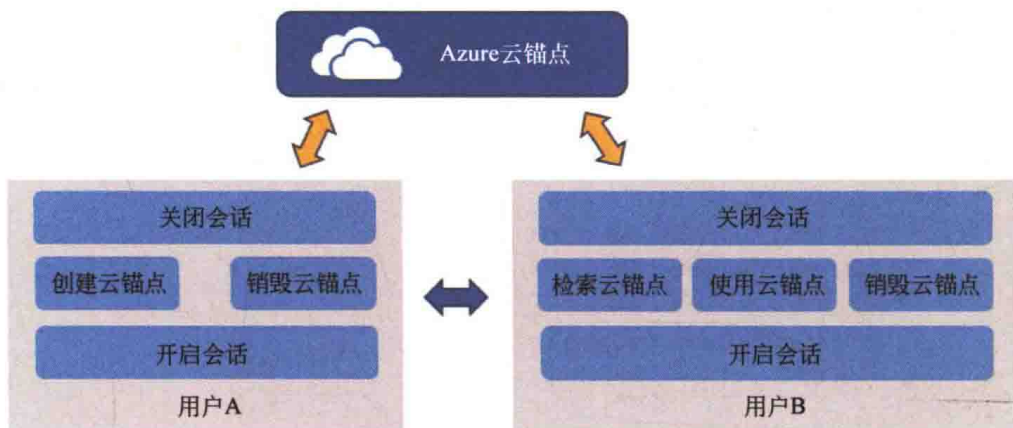


图 8-10 Azure 云锚点基本使用流程

使用 Azure 云锚点时需要用到 SpatialAnchorManager 组件,该组件由 Azure 空间锚点 SDK 提供,使用时需要将该组件挂载到场景对象上。在工程窗口中新建一个 C# 脚本文件,命名为 AzureSpatialAnchorController,编写代码如下:

```
// 第8章/8-3
using System;
using System.Collections.Generic;
```

^① 除可以直接编辑 manifest.json 清单文件,也可以使用混合现实特性工具(Mixed Reality Feature Tool, MRFT)添加云锚点 SDK,使用 MRFT 添加工具包更直观,具体使用方法可查阅相关资料。

```

using System.IO;
using System.Threading.Tasks;
using UnityEngine;
using Microsoft.Azure.SpatialAnchors;
using Microsoft.Azure.SpatialAnchors.Unity;
#if Windows_UWP
using Windows.Storage;
#endif
public class AzureSpatialAnchorController : MonoBehaviour
{
    [HideInInspector]
    public string currentAzureAnchorID = ""; // 云锚点 ID
    private SpatialAnchorManager cloudManager;
    private CloudSpatialAnchor currentCloudAnchor;
    private AnchorLocateCriteria anchorLocateCriteria;
    private CloudSpatialAnchorWatcher currentWatcher;

    private readonly Queue<Action> dispatchQueue = new Queue<Action>();
    void Start()
    {
        cloudManager = GetComponent<SpatialAnchorManager>();
        cloudManager.AnchorLocated += CloudManager_AnchorLocated;
        anchorLocateCriteria = new AnchorLocateCriteria();
    }

    void Update()
    {
        lock (dispatchQueue)
        {
            if (dispatchQueue.Count > 0)
            {
                dispatchQueue.Dequeue();
            }
        }
    }

    void OnDestroy()
    {
        if (cloudManager != null && cloudManager.Session != null)
        {
            cloudManager.DestroySession();
        }

        if (currentWatcher != null)
        {
            currentWatcher.Stop();
            currentWatcher = null;
        }
    }
}

```



```

}
#region Azure 云锚点处理方法
public async void StartAzureSession()
{
    if (cloudManager.Session == null)
    {
        await cloudManager.CreateSessionAsync();
    }
    await cloudManager.StartSessionAsync();
    Debug.Log(" 开启云锚 Session 成功! ");
}

public async void StopAzureSession()
{
    cloudManager.StopSession();
    await cloudManager.ResetSessionAsync();
    Debug.Log(" 关闭云锚 Session 成功! ");
}

public async void CreateAzureAnchor(GameObject theObject)
{
    theObject.CreateNativeAnchor();
    CloudSpatialAnchor localCloudAnchor = new CloudSpatialAnchor();
    localCloudAnchor.LocalAnchor = theObject.FindNativeAnchor().
GetPointer();
    if (localCloudAnchor.LocalAnchor == IntPtr.Zero)
    {
        Debug.Log(" 无法创建本地锚点 ");
        return;
    }
    localCloudAnchor.Expiration = DateTimeOffset.Now.AddDays(7);
    while (!cloudManager.IsReadyForCreate)
    {
        await Task.Delay(330);
        float createProgress = cloudManager.SessionStatus.
RecommendedForCreateProgress;
        QueueOnUpdate(new Action(() => Debug.Log($" 请缓慢移动以采集更多环境信息,
当前进度: {createProgress:0%}")));
    }

    bool success;
    try
    {
        await cloudManager.CreateAnchorAsync(localCloudAnchor);
        currentCloudAnchor = localCloudAnchor;
        localCloudAnchor = null;
        success = currentCloudAnchor != null;
    }
}

```



```

        if (success)
        {
            Debug.Log($"云锚点 ID: '{currentCloudAnchor.Identifier}' 创建成功");
            currentAzureAnchorID = currentCloudAnchor.Identifier;
        }
        else
        {
            Debug.Log($"创建云锚点 ID: '{currentAzureAnchorID}' 失败");
        }
    }
    catch (Exception ex)
    {
        Debug.Log(ex.ToString());
    }
}

public void RemoveLocalAnchor(GameObject theObject)
{
    theObject.DeleteNativeAnchor();
    if (theObject.FindNativeAnchor() == null)
    {
        Debug.Log("本地锚点移除成功");
    }
    else
    {
        Debug.Log("本地锚点移除失败");
    }
}

public void FindAzureAnchor(string id = "")
{
    if (id != "")
    {
        currentAzureAnchorID = id;
    }
    List<string> anchorsToFind = new List<string>();
    if (currentAzureAnchorID != "")
    {
        anchorsToFind.Add(currentAzureAnchorID);
    }
    else
    {
        Debug.Log("无需要查找的云锚点 ID");
        return;
    }
}

```

```

        anchorLocateCriteria.Identifiers = anchorsToFind.ToArray();
        if ((cloudManager != null) && (cloudManager.Session != null))
        {
            currentWatcher = cloudManager.Session.CreateWatcher(anchorLocateCriteria);

            DeBug.Log(" 开始查找云锚点 ");
        }
        else
        {
            currentWatcher = null;
        }
    }

    public async void DeleteAzureAnchor()
    {
        await cloudManager.DeleteAnchorAsync(currentCloudAnchor);
        currentCloudAnchor = null;
        DeBug.Log(" 云锚点移除成功 ");
    }

    public void SaveAzureAnchorIdToDisk()
    {
        string filename = "SavedAzureAnchorID.txt";
        string path = Application.persistentDataPath;
#if Windows_UWP
        StorageFolder storageFolder = ApplicationData.Current.LocalFolder;
        path = storageFolder.Path.Replace('\\', '/') + "/";
#endif
        string filePath = Path.Combine(path, filename);
        File.WriteAllText(filePath, currentAzureAnchorID);
        DeBug.Log($" 保存文件成功! ");
    }

    public void GetAzureAnchorIdFromDisk()
    {
        string filename = "SavedAzureAnchorID.txt";
        string path = Application.persistentDataPath;
#if Windows_UWP
        StorageFolder storageFolder = ApplicationData.Current.LocalFolder;
        path = storageFolder.Path.Replace('\\', '/') + "/";
#endif
        string filePath = Path.Combine(path, filename);
        currentAzureAnchorID = File.ReadAllText(filePath);
    }

    public void ShareAzureAnchorIdToNetwork()
    {

```

```

        // 通过网络传输云锚点
    }

    public void GetAzureAnchorIdFromNetwork()
    {
        // 通过网络接收云锚点
    }
}

#endregion

private void CloudManager_AnchorLocated(object sender, AnchorLocatedEventArgs
args)
{
    if (args.Status == LocateAnchorStatus.Located || args.Status ==
LocateAnchorStatus.AlreadyTracked)
    {
        currentCloudAnchor = args.Anchor;

        QueueOnUpdate(() =>
        {
            Debug.Log($" 云锚点定位成功 ");
            gameObject.CreateNativeAnchor();
            if (currentCloudAnchor != null)
            {
                // 利用锚点信息变换游戏对象位姿
                gameObject.GetComponent<UnityEngine.XR.WSA.WorldAnchor>().Se
tNativeSpatialAnchorPtr(currentCloudAnchor.LocalAnchor);
            }
        });
    }
    else
    {
        QueueOnUpdate(new Action(() => Debug.Log($" 锚点 ID: '{args.Identifier}'
定位失败, 定位状态为 '{args.Status}'"));
    }
}

private void QueueOnUpdate(Action updateAction)
{
    lock (dispatchQueue)
    {
        dispatchQueue.Enqueue(updateAction);
    }
}
}

```

在上述代码中, 首先在 Start() 方法中获取 SpatialAnchorManager 组件, 然后将其 AnchorLocated 事件设置到云锚点重定位处理方法。

在代码中,使用 `StartAzureSession()` 方法开启 Azure 云会话,使用 `StopAzureSession()` 方法关闭 Azure 云会话,通过 `CreateAzureAnchor()` 方法创建云锚点,通过 `FindAzureAnchor()` 方法在 Azure 中根据指定的云锚点 ID 检索锚点(检索结果通过重定位事件处理方法 `CloudManager_AnchorLocated()` 处理),在不需要云锚点时,使用 `DeleteAzureAnchor()` 方法销毁云锚点。

在具体操作时,通过 `AzureSpatialAnchorController` 脚本就可以使用 Azure 云锚点服务,基本流程如下:

在 Unity 层级(Hierarchy)窗口中新建一个 Cube 对象,命名为 `AnchorParent`,将 `SpatialAnchorManager` 组件挂载到该对象上,将 Azure 云服务器上创建的空间定位点资源所在域、账号 ID、主密钥对应的设置到该组件对象的 `Account Domain`、`Account Id`、`Account Key` 属性中。将 `AzureSpatialAnchorController` 脚本也挂载到该对象上,然后在场景中通过按钮事件驱动云会话开启、锚点创建等方法就可以使用 Azure 云锚点了。

Azure 空间锚点 SDK 使用时也需要正确配置工程,并需要在场景中挂载 `AR Anchor Manager` 组件,Azure 空间锚点 SDK 完成了绝大部分云锚点的处理工作(如坐标变换、空间特征点提取、锚点信息格式生成与解析等),提供给开发人员的使用界面非常简洁,大大降低了使用难度^①。Azure 云锚点的使用注意事项与 `ARCore` 云锚点一致。

利用 Azure 云锚点可以支持 `HoloLens` 设备、支持使用 `ARKit` 的 iOS 设备、支持使用 `ARCore` 的安卓设备,因此可以实现不同硬件设备之间的 AR 体验共享,轻松实现第三方视角、投屏展示等实际开发需求。

^① Azure 空间锚点 SDK 通信使用了 `OKHttp3` 轻量级网络通信框架,需要在 Unity 工程中进行 `Gradle` 配置,具体可参阅相关文档。

第 9 章



场景图像获取与场景深度

对主流移动终端（手机、平板），实现 AR 的方式是采用 Pass-Through 方案（通过设备摄像头采集真实场景图像，然后渲染为 AR 场景背景）。在 AR Foundation 中，设备摄像头图像^①的获取与背景渲染由底层的 SDK Provider 提供，AR Foundation 提供了一个公共接口界面（AR Camera Background）负责管理背景渲染，开发者也可以自定义摄像头图像的获取与背景渲染。本章将讲述从 GPU、CPU 中获取场景原始图像及虚实融合图像的方法，并以边缘检测实例简要展示图像获取的后续使用流程。

ARCore 还实现了利用单目相机估算场景深度的功能，利用这个功能可以实现虚实遮挡、物理仿真等效果，能更进一步地提高虚实融合的真实感，本章也将讨论利用 ARCore 获取场景深度的方法及一般应用。

9.1 获取 GPU 图像

截屏是移动手机用户经常使用的一项功能，也是一项特别方便用户保存、分享屏幕信息的方式。移动设备（包括 iOS 设备和 Android 设备）都具备方便高效的截屏快捷键。

但在 AR 应用中，使用者可能有不同于直接截屏的需求，如剔除屏幕上的 UI 元素，只保留摄像头图像和渲染的虚拟物体图像；开发人员有时也需要直接获取摄像头中的数据进行后续处理，即获取纯粹的不包含 UI 元素与虚拟物体的摄像头硬件采集的视频图像原始数据，如使用自定义计算机视觉算法。

在 AR Foundation 应用开发中，可以通过 GPU 获取摄像头、屏幕图像数据，也可以通过 CPU 获取摄像头原始图像数据，本节主要学习通过 GPU 获取摄像头原始图像及屏幕图像的方法。

9.1.1 获取摄像头原始图像

AR Foundation 本身其实并不负责摄像头图像信息获取及其在屏幕上的渲染，而是根据不

^① 本章所述摄像机 / 摄像头指移动设备硬件摄像头，以与渲染场景所使用的渲染相机相区别。

同的平台选择不同的底层 Provider（提供者）来提供摄像头图像，这样做的原因是不同的平台在获取摄像头数据时的视频编码及显示编码格式并不相同，没办法提供一个兼容解决方案。当然，这么做也提高了不同平台获取摄像头图像进行渲染的灵活性，可以兼容更多的算法及实现，甚至开发者也可以提供个性化的实现方案。

在 AR Camera 游戏对象上挂载的 ARCameraBackground 组件中，有一个 Use Custom Material 属性选项，如图 9-1 所示。

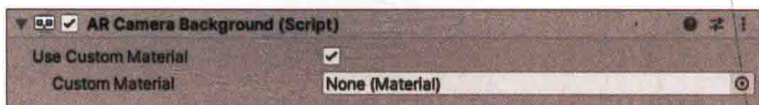


图 9-1 AR Camera Background 组件

在不勾选这个选项时，AR Foundation 会根据所编译目标平台自动选择底层摄像头图像获取及渲染 Provider，如在 Android 平台，ARCore 会提供这个功能实现；在 iOS 平台，则由 ARKit 提供。如果勾选这个选项，则需要开发者自定义一个获取摄像头图像的算法和渲染背景的材质，然后 AR Foundation 会使用这个材质将图像信息渲染到设备屏幕。

也就是说，AR Foundation 利用第三方提供的摄像头图像和材质进行 AR 场景背景渲染，因此，也可以通过 AR Camera Background 组件获取摄像头的原始图像信息，不管图像是来自于 ARCore、ARKit，还是用户自定义，只要获取这个材质并将其渲染到 Render Texture，就可以获取摄像头的原始图像信息。基于以上思想，核心代码如下^①。

```
// 第9章 / 9-1
public void CaptureGPUImageWithoutObject()
{
    if (mRenderTexture == null)
    {
        RenderTextureDescriptor renderTextureDesc = new
            RenderTextureDescriptor(Screen.width, Screen.height, RenderTextureFormat.BGRA32);
        mRenderTexture = new RenderTexture(renderTextureDesc);
    }

    var commandBuffer = new CommandBuffer();
    commandBuffer.name = "Capture GPU Image Without Object";
    var texture = !mArCameraBackground.material.HasProperty("_MainTex") ? null :
        mArCameraBackground.material.GetTexture("_MainTex");
    Graphics.SetRenderTarget(mRenderTexture.colorBuffer, mRenderTexture.depthBuffer);
    commandBuffer.ClearRenderTarget(true, false, Color.clear);
    commandBuffer.Blit(texture, BuiltinRenderTextureType.CurrentActive,
        mArCameraBackground.material);
    Graphics.ExecuteCommandBuffer(commandBuffer);
}
```

① 完整的代码稍后将详细阐述。


```

        if (mLastCameraTexture == null)
            mLastCameraTexture = new Texture2D(mRenderTexture.width, mRenderTexture.
height, TextureFormat.RGB24, true);
        mLastCameraTexture.ReadPixels(new Rect(0, 0, mRenderTexture.width,
mRenderTexture.height), 0, 0);

        var Bytes = mLastCameraTexture.EncodeToPNG();
        var path = Application.persistentDataPath + "/texture01.png";
        System.IO.File.WriteAllBytes(path, Bytes);
    }

```

在上述代码中, 使用 `commandBuffer` 渲染命令进行后渲染, `commandBuffer.Blit()` 方法将摄像头原始图像数据使用特定的材质渲染后存储到目标纹理区, 这种方法在屏幕后处理特效中使用得非常多, 该方法有多个重载, 典型原型如下:

```

public static void Blit(Texture source, Rendering.RenderTargetIdentifier dest,
Material mat, int pass);

```

在本示例中, 源纹理即为摄像头原始图像, 在得到这个背景纹理后, 就可以采用 RGB24 的格式将其保存下来。使用这种方法可以直接获取底层 Provider 摄像头图像数据, 所以获取的图像不包含任何虚拟物体、UI 等 AR 元素, 是纯设备硬件摄像头的图像数据。

`CommandBuffer` 为开发者定制渲染管线提供了强大支持, 允许开发者自定义渲染命令, 用于拓展渲染管线的渲染效果, 更多详细信息可查阅 Unity 官方资料。

9.1.2 获取屏幕显示图像

在某些时候希望获取用户设备屏幕图像, 即包括虚拟对象、UI 等信息的屏幕截图。获取屏幕显示图像的方法是直接获取屏幕输出图像信息, 即获取场景中渲染相机画面信息 (此处的渲染相机为 AR 场景中的虚拟相机, 与 9.1.1 节所讲的摄像头不是同一概念), 因此, 实现的思路是将渲染相机图像信息输出到 `Render Texture`, 这样就可以转换成图片格式进行存储了, 核心代码如下^①:

```

// 第9章/9-2
private void CaptureGPUImageWithObject()
{
    if (mRenderTexture == null)
    {
        RenderTextureDescriptor renderTextureDesc = new RenderTextureDescriptor
(Screen.width, Screen.height, RenderTextureFormat.BGRA32);
        mRenderTexture = RenderTexture.GetTemporary(renderTextureDesc);
    }
}

```

① 完整的代码稍后将详细阐述。

```

mCamera.targetTexture = mRenderTexture;
mCamera.Render();
//RenderTexture currentActiveRT = RenderTexture.active;
RenderTexture.active = mRenderTexture;
if (mLastCameraTexture == null)
    mLastCameraTexture = new Texture2D(mRenderTexture.width, mRenderTexture.
height, TextureFormat.RGB24, true);
mLastCameraTexture.ReadPixels(new Rect(0, 0, mRenderTexture.width, mRender
Texture.height), 0, 0);

mLastCameraTexture.Apply();
mCamera.targetTexture = null;
RenderTexture.active = null;

Byte[] Bytes = mLastCameraTexture.EncodeToPNG();
var path = Application.persistentDataPath + "/texture02.png";
System.IO.File.WriteAllBytes(path, Bytes);
}

```

在上述代码中，在将相机图像渲染到 Render Texture 后，一定要设置 `mCamera.targetTexture = null`；这样才能恢复将渲染相机的图像信息输出到屏幕缓冲区，不然屏幕将无图像输出。从技术角度而言，通过改变渲染相机图像输出流方向达到将渲染相机图像输出到特定纹理缓冲区的目的，因此在操作结束后需要将输出方向再改回默认的方向（默认为屏幕缓冲区）。

下面通过完整示例，演示这两种获取图像信息的方法，具体实现操作如下：首先新建一个 C# 脚本，命名为 `GPUImageManager`，编写代码如下：

```

// 第9章/9-3
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.XR.ARFoundation;
using UnityEngine.Rendering;

public class GPUImageManager : MonoBehaviour
{
    public ARCameraBackground mArCameraBackground; // 组件对象
    public Camera mCamera; // 组件对象
    public Button BtnCapture; // UI 按钮
    public Button BtnCapture2; // UI 按钮

    private RenderTexture mRenderTexture; // Render 纹理
    private Texture2D mLastCameraTexture; // 二维纹理
    void Start()
    {

```



```

        BtnCapture.transform.GetComponent<Button>().onClick.AddListener(CaptureGPUImageWithoutObject);
        BtnCapture2.transform.GetComponent<Button>().onClick.AddListener(CaptureGPUImageWithObject);
    }
    // 获取 GPU 摄像头原始图像
    public void CaptureGPUImageWithoutObject()
    {
        if (mRenderTexture == null)
        {
            RenderTextureDescriptor renderTextureDesc = new RenderTextureDescriptor(Screen.width, Screen.height, RenderTextureFormat.BGRA32);
            mRenderTexture = new RenderTexture(renderTextureDesc);
        }

        var commandBuffer = new CommandBuffer();
        commandBuffer.name = "Capture GPU Image Without Object";
        var texture = !mArCameraBackground.material.HasProperty("_MainTex") ? null : mArCameraBackground.material.GetTexture("_MainTex");
        Graphics.SetRenderTarget(mRenderTexture.colorBuffer, mRenderTexture.depthBuffer);
        commandBuffer.ClearRenderTarget(true, false, Color.clear);
        commandBuffer.Blit(texture, BuiltinRenderTextureType.CurrentActive, mArCameraBackground.material);
        Graphics.ExecuteCommandBuffer(commandBuffer);

        if (mLastCameraTexture == null)
            mLastCameraTexture = new Texture2D(mRenderTexture.width, mRenderTexture.height, TextureFormat.RGB24, true);
        mLastCameraTexture.ReadPixels(new Rect(0, 0, mRenderTexture.width, mRenderTexture.height), 0, 0);

        var Bytes = mLastCameraTexture.EncodeToPNG();
        var path = Application.persistentDataPath + "/texture01.png";
        System.IO.File.WriteAllBytes(path, Bytes);
    }
    // 获取 GPU 渲染相机虚实混合图像
    private void CaptureGPUImageWithObject()
    {
        if (mRenderTexture == null)
        {
            RenderTextureDescriptor renderTextureDesc = new RenderTextureDescriptor(Screen.width, Screen.height, RenderTextureFormat.BGRA32);
            mRenderTexture = RenderTexture.GetTemporary(renderTextureDesc);
        }

        mCamera.targetTexture = mRenderTexture;
    }

```



```
mCamera.Render();  
//RenderTexture currentActiveRT = RenderTexture.active;  
RenderTexture.active = mRenderTexture;  
if (mLastCameraTexture == null)  
    mLastCameraTexture = new Texture2D(mRenderTexture.width, mRenderTexture.  
height, TextureFormat.RGB24, true);  
mLastCameraTexture.ReadPixels(new Rect(0, 0, mRenderTexture.width,  
mRenderTexture.height), 0, 0);  
  
mLastCameraTexture.Apply();  
mCamera.targetTexture = null;  
RenderTexture.active = null;  
  
Byte[] Bytes = mLastCameraTexture.EncodeToPNG();  
var path = Application.persistentDataPath + "/texture02.png";  
System.IO.File.WriteAllBytes(path, Bytes);  
}  
}
```

关键代码的功能及作用在前文中已经分析过，不再赘述。将 GPUImageManager 脚本挂载到场景中任意对象上，本节使用按钮触发，因此需要在层级（Hierarchy）窗口中新建两个 Button UI，然后正确地对 mArCameraBackground、mCamera、BtnCapture、BtnCapture2 进行赋值。

编译运行，在检测到的平面上放置一个立方体，然后通过不同的按钮分别获取摄像头原始图像信息与屏幕显示图像信息，效果如图 9-2 所示。



图 9-2 获取摄像头原始图像与屏幕图像效果图

提示

本示例获取并保存的图像并没有刷新到手机设备的相册中，只保存在应用安装目录的 Data 文件夹下，因此要得到保存的图像文件，需要再将其复制出来。

9.2 获取 CPU 图像

9.1 节中, 已经实现从设备摄像头或屏幕显示中获取图像信息, 在处理类似截屏这类任务时非常合适, 但在获取实时连续图像视频流时, 由于干预了渲染管线命令队列, 会有较大的性能问题, 主要原因在于获取的图像数据信息全部来自于 GPU 显存, 而从 GPU 显存将数据回读到 CPU 内存是一项复杂的操作。就获取设备摄像头图像视频数据而言, 这种方式也是从 CPU 到 GPU 再到 CPU 绕了一圈, 增加了性能消耗。本节将学习从 CPU 中读取设备摄像头图像信息 (这种需求在进行深度开发时还是比较普遍的, 如对设备摄像头图像进行计算机视觉处理, 包括手势检测、物体识别等), 然后以边缘检测为例, 对获取的图像流信息进行实时二次处理演示。

9.2.1 AR 摄像头图像数据流

在计算机系统中, CPU 与 GPU 有明确的任务分工^①, 在 AR 应用中, CPU 主要负责逻辑运算及流程控制, GPU 主要负责图像渲染处理, 它们都有各自独立的存储器, 分别称为内存与显存。从设备摄像头中获取图像数据的数据流图如图 9-3 所示。

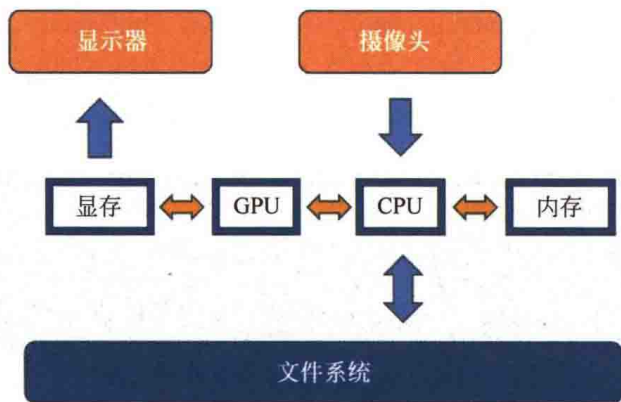


图 9-3 计算机系统数据流图

从图 9-3 可以清楚地看到, AR 场景中的所有对象数据最终都将交由 GPU 进行渲染并输出到显示缓冲区, 包括摄像头获取的图像信息。另一方面也可以看出, 实时获取设备摄像头图像信息的最佳方式是直接从 CPU 中读取, 这样可以避免摄像头图像由 CPU 采集获取传输到 GPU, 然后回读到 CPU 的性能问题, 这种方案是一种理论上的最佳方案, 但在 AR Foundation 中, AR 场景背景由第三方底层 Provider 提供, 即从设备摄像头采集图像数据到进行渲染显示并不由 AR Foundation 具体实现, 而且第三方底层实现的方式可能会多种多样, 也没有强制对上开放, 这样导致无法直接从 AR Foundation 中采集 CPU 图像数据, 所以 AR Foundation 只能

^① CPU 利用 GPU 加速计算、GPGPU 也处在快速发展中, 这里主要还是指传统的 CPU 和 GPU 分工。

退而求其次，即从 GPU 回读图像数据到 CPU，然后向开发者提供 CPU 图像数据。由于这个原因，在 AR Foundation 中，获取 CPU 图像数据流其实也并不是直接采集设备摄像头图像数据，这个性能开销依然很大^①。

9.2.2 从 CPU 中获取摄像头图像

AR Camera Manager 组件提供了 TryAcquireLatestCpuImage() 方法，使用这种方法可以直接从 CPU 获取设备摄像头的图像数据信息，该方法的原型如下：

```
public bool TryAcquireLatestCpuImage(out XRCpuImage cpuImage);
```

参数 XRCpuImage 是一个本地化的结构体，通常需要在用完及时释放（使用 Dispose() 方法释放），因为在很多平台上，XRCpuImage 数量是有限制的，如果长时间不释放，则会导致无法获取新的图像数据信息。在 AR Foundation 中，使用这个 XRCpuImage 结构体时，可以设置获取的摄像头图像模式、直接获取摄像头视频图像通道信息，支持同步转换到灰度图或彩色图、异步转换到灰度图或彩色图，以下对这 4 个特性分别进行阐述。

1. 设置摄像头图像模式

在采集 CPU 图像数据时，摄像头的分辨率和帧率可由开发者设定，当然这些设定值必须被摄像头支持，所以一个比较合适的做法是列举出所有摄像头支持的模式，由用户根据其需求选择最合适的分辨率与帧率。ARCameraManager 组件提供了一个 GetConfigurations() 方法，利用该方法可以获取摄像头所支持的所有模式，其原型如下：

```
public NativeArray<XRCameraConfiguration> GetConfigurations(Allocator allocator);
```

在默认情况下，AR Foundation 会选择分辨率最低的模式，因为在该模式下可以覆盖尽可能多的用户设备并降低硬件资源需求。

下面演示如何遍历摄像头所支持的模式和更改摄像头数据采集模式。新建一个 C# 脚本，命名为 CpuImageConfiguration，该脚本主要完成对当前摄像头所支持模式的遍历，并将所有的支持模式存储到一个下拉列表框中，当用户选择其他模式时，更改摄像头采集数据的模式。同时，为了更清晰直观地看到变更效果，直接显示了当前使用的摄像头数据采集模式名称，编写代码如下：

```
// 第9章 / 9-4
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.XR.ARFoundation;
```

^① 即虽然在 AR Foundation 中称之为直接从 CPU 中获取摄像头图像数据，但实质上图像数据是 AR Foundation 预先从 GPU 中回读的数据，所以在实际项目中，理想的方法应当是自定义摄像头图像采集和 AR 背景渲染，从自定义的摄像头图像中读取图像数据，而不应当使用 AR Foundation 提供的 CPU 图像数据，这可以大幅度提高性能。


```

public class CpuImageConfiguration : MonoBehaviour
{
    public Dropdown mDropdown;           // 下拉列表框
    public Text mText;                   // 当前摄像头显示信息
    private ARCameraManager mCameraManager;
    private List<string> mConfigurationNames; // 配置文件名
    void Awake()
    {
        mCameraManager = GetComponent<ARCameraManager>();
        mDropdown.ClearOptions();
        mDropdown.onValueChanged.AddListener(delegate { OnDropdownValueChanged(mDropdown); });
        mConfigurationNames = new List<string>();
    }
    // 下拉列表项中当前选中值发生变更时触发
    void OnDropdownValueChanged(Dropdown dropdown)
    {
        if ((mCameraManager == null) || (mCameraManager.subsystem == null) || !mCameraManager.subsystem.running)
        {
            return;
        }
        var configurationIndex = dropdown.value;
        using (var configurations = mCameraManager.GetConfigurations(Unity.Collections.Allocator.Temp))
        {
            if (configurationIndex >= configurations.Length)
            {
                return;
            }
            var configuration = configurations[configurationIndex];
            mCameraManager.currentConfiguration = configuration;
            mText.text = configuration.resolution.ToString() + "@" + configuration.framerate.ToString();
        }
    }
    // 遍历摄像头所支持的数据采集模式并将其存储到 Dropdown 列表框中, 选中当前使用的模式
    void PopulateDropdown()
    {
        if ((mCameraManager == null) || (mCameraManager.subsystem == null) || !mCameraManager.subsystem.running)
        {
            return;
        }
        using (var configurations = mCameraManager.GetConfigurations(Unity.Collections.Allocator.Temp))
        {
            if (!configurations.IsCreated || (configurations.Length <= 0))

```

```

    {
        return;
    }
    foreach (var config in configurations)
    {
        mConfigurationNames.Add(config.resolution.ToString()+"@"+config.
        framerate.ToString());
    }
    mDropdown.AddOptions(mConfigurationNames);

    var currentConfig = mCameraManager.currentConfiguration;
    for (int i = 0; i < configurations.Length; ++i)
    {
        if (currentConfig == configurations[i])
        {
            mDropdown.value = i;
            mText.text = currentConfig.Value.resolution.ToString()+"@" +
            currentConfig.Value.framerate.ToString();
        }
    }
}

void Update()
{
    if (mConfigurationNames.Count == 0)
        PopulateDropdown();
}
}

```

将 CpuImageConfiguration 脚本挂载到层级窗口 AR Session Origin 游戏对象下的 AR Camera 对象上, 并设置好 UI, 编译运行, 通过选择下拉列表中的不同摄像头模式选项, 可以看到摄像头采集数据的模式也在发生变化, 如图 9-4 所示。



图 9-4 通过配置变更摄像头采集数据模式示意图

2. 获取图像通道信息数据

在视频编码中绝大多数编码格式采用 YUV 格式, 其中 Y 表示明亮度, 用于描述像素明暗程度, UV 则表示色度, 用于描述色彩及饱和度, 在 AR Foundation 中使用 Plane 这个术语来描述视频通道 (不要与平面检测中的 Plane 混淆, 这两者没有任何关系)。获取特定平台的 YUV 通道信息数据可以直接使用 `XRCpuImage.GetPlane()` 方法, 该方法返回各通道的视频原始 YUV 编码信息, 示例代码如下:

```
// 第 9 章 / 9-5
XRCpuImage image;
if (!cameraManager.TryAcquireLatestCpuImage(out XRCpuImage image))
    return;

// 遍历视频各通道信息
for (int planeIndex = 0; planeIndex < image.planeCount; ++planeIndex)
{
    // 获取指定通道图像数据信息, 包括尺寸、宽、每像素所占字节数
    var plane = image.GetPlane(planeIndex);
    Debug.LogFormat("Plane {0}:\n\tsize: {1}\n\trowStride: {2}\n\tpixelStride: {3}",
        planeIndex, plane.data.Length, plane.rowStride, plane.pixelStride);

    // 对图像进行进一步处理的自定义算法
    MyComputerVisionAlgorithm(plane.data);
}

// 为了防止内存泄漏, 及时释放资源
image.Dispose();
```

提示

由于历史原因, 黑白与彩色显示器曾长期共存, 视频采用 YUV 编码格式主要是为了解决黑白与彩色显示器的兼容问题。

使用 `XRCpuImage.Plane` 可以通过 `NativeArray<Byte>` 直接访问系统本地的内存缓冲区, 因此可以直接获取本地内存视图, 而且在用完后不需要显式释放 `NativeArray`, 在 `XRCpuImage` 对象被释放后, `NativeArray` 一并被销毁。另外, 视频图像缓冲区是只读区, 应用程序不能直接向该区域写数据。

3. 同步转换到灰度图或彩色图

从摄像头视频流中获取灰度或者彩色图像需要对 YUV 编码的视频流图像格式进行转换, `XRCpuImage` 类提供了同步与异步两种转换方式, 转换方法的原型如下:

```
public void Convert(XRCpuImage.ConversionParams conversionParams, IntPtr
    destinationBuffer, int bufferLength);
```


其中 `conversionParams` 为控制转换的参数, `destinationBuffer` 为接受转换后图像数据的目标缓冲区。如果设定转换格式 `TextureFormat.Alpha8` 或者 `TextureFormat.R8`, 则将视频图像转换成灰度图像, 其他格式则转换为彩色图像, 转换为彩色图像比转换为灰度图像需要转换的数据量更大, 因此计算更密集, 转换也会更慢。

提示

通过 `TextureFormat.Alpha8` 或者 `TextureFormat.R8` 参数可以提取原始图像 Alpha 通道或 R 通道颜色分量值。Unity 本身并没有灰度纹理的概念 (只能渲染成灰度的纹理)。以 `Alpha8` 或者 `R8` 格式存储的纹理可以渲染成灰度图, 操作方式是在 Shader 中提取以 `R8` 格式存储的纹理, 然后生成一个各颜色分量均相同的 RGB 值, 渲染输出即是灰度图。如果只是提取了 Alpha 通道或 R 通道颜色分量直接显示, 则只会显示该分量的颜色值。

另外, 通过 CPU 获取的图像数据是摄像头原始未经过处理的图像数据, 因此图像方向可能会与手机显示屏方向不一致, 如旋转了 90° 等。

`XRCpuImage.ConversionParams` 结构体定义如下:

```
// 第9章 / 9-6
public struct ConversionParams
{
    public RectInt inputRect;
    public Vector2Int outputDimensions;
    public TextureFormat outputFormat;
    public Transformation transformation;
}
```

该结构体共有 4 个参数, 各参数定义及意义如表 9-1 所示。

表 9-1 `XRCpuImage.ConversionParams` 结构体参数表

属 性	描述说明
<code>inputRect</code>	需要转换的图像区域, 可以是视频流图像原始尺寸, 也可以是比原始图像小的一个区域。转换区域必须与原始图像完全匹配, 即转换区域必须是原始图像有效区域的全部或一部分。在明确了解所使用图像区域时, 对图像进行部分区域转换可加快转换速度
<code>outputDimensions</code>	输出图像的尺寸。 <code>XRCpuImage</code> 支持降采样转换, 允许指定比原图像更小的图像尺寸, 如可以将输出图像尺寸定义为 $(inputRect.width / 2, inputRect.height / 2)$, 从而获取原图像一半的分辨率, 这样可以降低转换彩色图像的时间。输出图像尺寸只能小于或者等于原始图像尺寸, 即不支持升级采样
<code>outputFormat</code>	输出图像格式, 目前支持 <code>TextureFormat.RGB24</code> 、 <code>TextureFormat.RGBA24</code> 、 <code>TextureFormat.ARGB32</code> 、 <code>TextureFormat.BGRA32</code> 、 <code>TextureFormat.Alpha8</code> 、 <code>TextureFormat.R8</code> 共 6 种格式, 可以在转换前通过 <code>XRCpuImage.FormatSupported</code> 检查受支持的图像格式
<code>transformation</code>	对原始图像进行变换, 目前支持 X 轴、Y 轴镜像

在转换前需要分配转换后的图像存储空间, 因此需要知道所需空间的大小, 可以通过 `GetConvertedDataSize()` 方法获取所需空间的大小, 该方法的原型如下:

```
public int GetConvertedDataSize(Vector2Int dimensions, TextureFormat format);
```

另一个重载原型如下:

```
public int GetConvertedDataSize(XRCpuImage.ConversionParams conversionParams);
```

通过 `Texture2D.LoadRawTextureData()` 得到转换后的数据格式, 使其与 `Texture2D` 格式兼容以便于操作, 示例代码如下:

```
// 第 9 章 / 9-7
using System;
using Unity.Collections;
using Unity.Collections.LowLevel.Unsafe;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;

public class CameraImageExample : MonoBehaviour
{
    Texture2D mTexture;
    // 注册事件
    void OnEnable()
    {
        cameraManager.cameraFrameReceived += OnCameraFrameReceived;
    }
    // 取消事件注册
    void OnDisable()
    {
        cameraManager.cameraFrameReceived -= OnCameraFrameReceived;
    }
    // 在获取图像帧时执行图像转换
    unsafe void OnCameraFrameReceived(ARCameraFrameEventArgs eventArgs)
    {
        if (!cameraManager.TryAcquireLatestCpuImage(out XRCpuImage image))
            return;

        var conversionParams = new XRCpuImage.ConversionParams
        {
            // 设置获取全幅图像
            inputRect = new RectInt(0, 0, image.width, image.height),
            // 降采集为原图像一半
            outputDimensions = new Vector2Int(image.width / 2, image.height / 2),
            // 设置使用 RGBA 格式
            outputFormat = TextureFormat.RGBA32,
            // 执行 Y 轴镜像操作
        }
    }
}
```



```

        transformation = XRCpuImage.Transformation.MirrorY
    };

    // 计算目标图像所需空间大小
    int size = image.GetConvertedDataSize(conversionParams);
    // 申请分配空间
    var buffer = new NativeArray<Byte>(size, Allocator.Temp);
    // 执行图像转换
    image.Convert(conversionParams, new IntPtr(buffer.GetUnsafePtr()), buffer.
Length);
    // 图像已转换并已存储到指定储存区域, 因此需要释放 image 对象以防止内存泄漏
    image.Dispose();
    // 这里可以执行对图像的进一步处理, 如进行计算机视觉处理等
    // 将其存入纹理以便显示
    mTexture = new Texture2D(
        conversionParams.outputDimensions.x,
        conversionParams.outputDimensions.y,
        conversionParams.outputFormat,
        false);
    // 转换成与 Texture2D 兼容的格式
    mTexture.LoadRawTextureData(buffer);
    mTexture.Apply();

    // 使用完后释放缓冲区
    buffer.Dispose();
}
}

```

4. 异步转换到灰度图或彩色图

同步转换会阻塞 CPU 后续流程, 严重时会造成应用卡顿, 因此在对实时视频流图像处理要求不高时可以采用异步转换的方式, 异步转换方法为 `XRCpuImage.ConvertAsync()`。

因为是异步转换, 所以对主流程不会造成很大影响, 可以获取大量异步转换图像而不影响主流程。通常, 异步转换会在当前帧的下一帧被处理, 异步转换可以采用队列的方式对所有要求异步转换的请求进行依次处理。`XRCpuImage.ConvertAsync()` 方法会返回 `XRCpuImage.AsyncConversion` 对象, 通过这个对象可以查询转换进程, 了解转换状态, 以便进行下一步操作, 典型的使用代码如下:

```

// 第9章 / 9-8
XRCpuImage.AsyncConversion conversion = image.ConvertAsync(...);
while (!conversion.status.IsDone())
    yield return null;

```

使用 `status` 标识可以查询转换是否完成, 如果 `status` 为 `XRCpuImage.AsyncConversion-Status.Ready`, 则可以通过调用 `GetData<T>()` 方法获取 `NativeArray<T>` 型图像数据。`GetData<T>()`

方法实际上是返回一个本地内存布局的一个视图，在 `XRCpuImage.AsyncConversion` 对象调用 `Dispose()` 方法前，这个视图一直存在，但在 `XRCpuImage.AsyncConversion` 对象调用 `Dispose()` 方法后，该视图也一并被销毁。另外，无须显式调用 `NativeArray<T>` 的 `Dispose()` 方法，在 `XRCpuImage.AsyncConversion` 对象被释放后该内存块也将被释放，所以尝试在 `XRCpuImage.AsyncConversion` 对象被释放后读取该内存数据会出现运行时错误。

注意

`XRCpuImage.AsyncConversion` 对象必须及时显式调用 `Dispose()` 方法进行释放，使用后不及时调用 `Dispose()` 方法会导致内存泄漏。虽然在 `XRCameraSubsystem` 对象被销毁后，所有的异步转换资源也将被释放，但还是强烈建议及时调用 `Dispose()` 方法释放 `XRCpuImage.AsyncConversion` 对象。

`XRCpuImage.AsyncConversion` 对象所属资源与 `XRCpuImage` 对象没有关系，`XRCpuImage` 有可能会在异步转换完成前被销毁，但这不会影响异步转换数据信息。

异步转换的典型操作代码如下：

```
// 第 9 章 / 9-9
Texture2D mTexture;    // 存储转换后的 Texture2D 图像信息

public void GetImageAsync()
{
    // 获取摄像头 image 对象信息
    if (cameraManager.TryAcquireLatestCpuImage(out XRCpuImage image))
    {
        // 启动协程进行异步转换
        StartCoroutine(ProcessImage(image));

        // 在异步转换完成前销毁 image
        image.Dispose();
    }
}

IEnumerator ProcessImage(XRCpuImage image)
{
    // 创建异步转换请求
    var request = image.ConvertAsync(new XRCpuImage.ConversionParams
    {
        // 设置获取全幅图像
        inputRect = new RectInt(0, 0, image.width, image.height),
        // 降采集为原图像一半
        outputDimensions = new Vector2Int(image.width / 2, image.height / 2),
        // 设置使用 RGB 格式
        outputFormat = TextureFormat.RGB24,
```

```

        // 执行 y 轴镜像操作
        transformation = XRCpuImage.Transformation.MirrorY
    ));

    // 等待转换完成
    while (!request.status.IsDone())
        yield return null;

    // 检测转换状态
    if (request.status != XRCpuImage.AsyncConversionStatus.Ready)
    {
        // 转换过程出现错误
        DeBug.LogErrorFormat("Request failed with status {0}", request.status);

        // 销毁转换请求
        request.Dispose();
        yield break;
    }

    // 获取转换后的图像数据
    var rawData = request.GetData<Byte>();

    // 这里可以执行对图像的进一步处理，如进行计算机视觉处理等
    // 将其存入纹理以便显示
    if (mTexture == null)
    {
        mTexture = new Texture2D(
            request.conversionParams.outputDimensions.x,
            request.conversionParams.outputDimensions.y,
            request.conversionParams.outputFormat,
            false);
    }

    // 转换成与 Texture2D 兼容的格式
    mTexture.LoadRawTextureData(rawData);
    mTexture.Apply();

    // 销毁转换请求以便释放与请求相关联的资源，包括原始数据
    request.Dispose();
}

```

除了使用上面的方法，也可以通过代理（delegate）进行回调的方式实现异步转换，而不是直接返回 `XRCpuImage.AsyncConversion` 对象，操作示例代码如下：

```

// 第9章 / 9-10
public void GetImageAsync()
{

```



```

// 获取摄像头 image 信息
if (cameraManager.TryAcquireLatestCpuImage(out XRCpuImage image))
{
    // 如果成功, 则启动协同等待图像信息处理, 然后应用到 Texture2D
    image.ConvertAsync(new XRCpuImage.ConversionParams
    {
        // 设置获取全幅图像
        inputRect = new RectInt(0, 0, image.width, image.height),
        // 降采集为原图像一半
        outputDimensions = new Vector2Int(image.width / 2, image.height / 2),
        // 设置使用 RGB 格式
        outputFormat = TextureFormat.RGB24,
        // 执行 Y 轴镜像操作
        transformation = CameraImageTransformation.MirrorY
    }, ProcessImage);

    // 在异步转换完成前销毁 image
    image.Dispose();
}

// 转换完成后的回调方法
void ProcessImage(XRCpuImage.AsyncConversionStatus status, XRCpuImage.
ConversionParams conversionParams, NativeArray<Byte> data)
{
    if (status != XRCpuImage.AsyncConversionStatus.Ready)
    {
        Debug.LogErrorFormat("Async request failed with status {0}", status);
        return;
    }

    // 自定义图像处理方法
    DoSomethingWithImageData(data);

    // 在异步转换回调执行前, 相关资源已被释放, 这里无须再进行资源处理
}

```

在使用这种方法时 `NativeArray<Byte>` 是与 request 请求关联的本地内存布局视图, 不需要显式调用 `Dispose()` 方法释放, 该视图只在代理 (delegate) 执行期间有效, 一旦代理方法返回, 则其也会被销毁, 如果需要在代理执行周期外使用这些数据, 则需要使用 `NativeArray<T>.CopyTo()` 和 `NativeArray<T>.CopyFrom()` 将数据及时备份出来。

9.3 边缘检测原理

边缘检测是图像处理和计算机视觉中的基本问题, 边缘检测的目的是标识数字图像中变

化明显的点。数字图像中像素的显著变化通常反映了属性的重要事件和变化,包括深度上的不连续、表面方向不连续、物质属性变化和场景照明变化。利用边缘检测可以大幅度减少需要处理的数据量,并且可以剔除不相关的信息,保留图像重要的结构属性,因而通常作为图像处理的预处理。在进行图像处理时,通常是先进行边缘检测,然后进行边缘处理,边缘检测采用的算法是采用卷积对图像进行处理。

9.3.1 卷积

卷积 (Convolution) 本质上来讲只是一种数学运算,与加减乘除没有区别,卷积是一种积分运算,可以看作加权求和。卷积把一个点的像素值用它周围点的像素值的加权平均代替,通常用于消除噪声、增强特征。卷积的具体方法是在图像处理中准备一个模板(这个模板就是卷积核 (Kernel)),对图像上的每像素,让模板的原点和该点重合,然后模板上的点和图像上对应的点相乘,最后将各点的积累加,得到该点的卷积值,用该卷积值代替原图像的对应像素值,如图 9-5 所示。移动模板到下一像素,以此类推,对图像上的每像素都进行类似处理,最后得到的一张使用卷积处理后的图。

卷积核通常是一个正方形网格结构(如 2×2 、 3×3),该网格区域内的每个方格都有一个权重值。当对图像中的某像素进行卷积操作时,会把卷积核的中心放置于该像素上,如图 9-6 所示,翻转核之后再依次计算核中每个元素和其覆盖的图像像素值的乘积,最后将各乘积累加,得到的结果就是该像素的新像素值,直到所有像素都处理完。

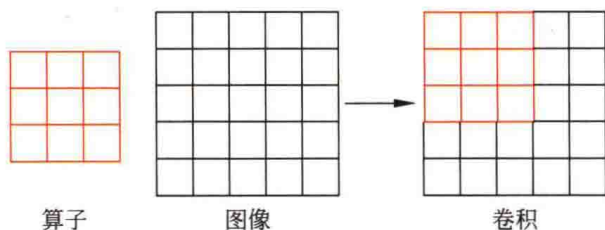


图 9-5 使用卷积核对图像进行处理的示意图

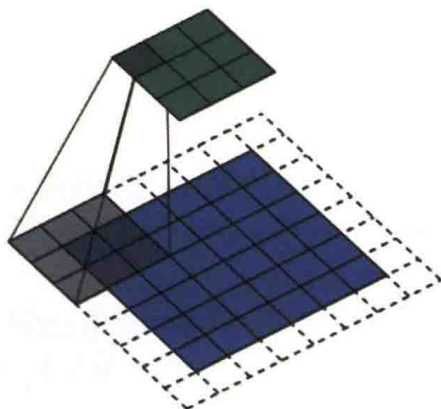


图 9-6 使用卷积核对图像中的每像素依次进行处理就得到该图像的卷积结果

卷积听起来很难,但在图形处理中其实很简单,卷积可以实现很多常见的图像处理效果,例如图像模糊、边缘检测等。

9.3.2 Sobel 算子

卷积的神奇之处在于选择的卷积核,用于边缘检测的卷积核也叫边缘检测算子,先后有好几种边缘检测算子被提出来。

1. Roberts 算子

Roberts 算子采用对角线方向相邻两像素之差近似梯度幅值检测边缘,该算子如图 9-7 所示。Roberts 算子检测水平和垂直边缘的效果好于斜向边缘,定位精度高,但对噪声敏感。

+1	0
0	-1

G_x

0	+1
-1	0

G_y

图 9-7 Roberts 算子

2. Prewitt 算子

Prewitt 算子利用像素上下左右相邻点灰度差在边缘处达到极值的特点检测边缘,该算子如图 9-8 所示, Prewitt 算子对噪声具有平滑作用,但是定位精度不够高。

-1	0	+1
-1	0	+1
-1	0	+1

G_x

+1	+1	+1
0	0	0
-1	-1	-1

G_y

图 9-8 Prewitt 算子

3. Sobel 算子

Sobel 算子主要用作边缘检测,它是一个离散的一阶差分算子,用来计算图像亮度函数的一阶梯度近似值,该算子如图 9-9 所示。在图像的任何一点使用此算子,将会产生该点对应的梯度向量或是法向量。与 Prewitt 算子相比, Sobel 算子对像素位置的影响做了加权,可以降低边缘模糊程度,因此效果更好。

Sobel 算子包含两组 3×3 的矩阵,分别为横向及纵向,将其与图像作平面卷积,即可分别得出横向及纵向的亮度差分近似值。如果以 A 代表原始图像, G_x 及 G_y 分别代表经横向及纵向边缘检测的图像灰度值,则其公式如图 9-10 所示。

-1	0	+1
-2	0	+2
-1	0	+1

G_x

+1	+2	+1
0	0	0
-1	-2	-1

G_y

图 9-9 Sobel 算子

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +2 \end{bmatrix} \times A \quad \text{和} \quad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \times A$$

图 9-10 Sobel 算子对图像进行处理的计算公式

具体计算公式如下：

$$\begin{aligned} G_x &= -1 \times f(x-1, y-1) + 0 \times f(x, y-1) + 1 \times f(x+1, y-1) + (-2) \times f(x-1, y) + 0 \times f(x, y) + 2 \times \\ &\quad f(x+1, y) + (-1) \times f(x-1, y+1) + 0 \times f(x, y+1) + 1 \times f(x+1, y+1) \\ &= [f(x+1, y-1) + 2 \times f(x+1, y) + f(x+1, y+1)] - [f(x-1, y-1) + 2 \times f(x-1, y) + f(x-1, y+1)] \\ G_y &= 1 \times f(x-1, y-1) + 2 \times f(x, y-1) + 1 \times f(x+1, y-1) + 0 \times f(x-1, y) + 0 \times f(x, y) + 0 \times \\ &\quad f(x+1, y) + (-1) \times f(x-1, y+1) + (-2) \times f(x, y+1) + (-1) \times f(x+1, y+1) \\ &= [f(x-1, y-1) + 2f(x, y-1) + f(x+1, y-1)] - [f(x-1, y+1) + 2 \times f(x, y+1) + f(x+1, y+1)] \end{aligned}$$

其中 $f(a, b)$ 表示原始图像 (a, b) 点的灰度值，得到每像素的横向及纵向灰度值后，可以通过以下公式来计算该点灰度的大小：

$$G = \sqrt{G_x^2 + G_y^2}$$

通常，为了提高效率使用不开平方的近似值：

$$G = |G_x| + |G_y|$$

如果梯度值 G 大于某一阈值，则认为该点 (x, y) 为边缘点。Sobel 算子也根据像素上下左右相邻点灰度加权差在边缘处达到极值这一原理来检测边缘，对噪声具有平滑作用，能提供较为精确的边缘方向信息，但边缘定位精度不够高，当对精度要求不是很高时，是一种较为常用的边缘检测方法。

Sobel 算子的计算速度比 Roberts 算子慢，但其较大的卷积核在很大程度上平滑了输入图像，使算子对噪声的敏感性降低。与 Roberts 算子相比，通常也会为相似的边缘产生更高的输出值，与 Roberts 算子一样，操作时输出值很容易溢出，仅支持小整数像素值（例如 8 位整数图像）图像类型。当发生溢出时，标准做法是简单地将溢出的输出像素设置为最大允许值（为避免此问题，可以通过支持范围更大像素值的图像类型）。

在前文 CPU 图像获取的基础上，下面演示使用 Sobel 算子进行边缘检测的流程，这是一个结合 ARCore 进行计算机图像处理的通用流程。

9.4 CPU 图像边缘检测实例

在前面的理论基础上，以同步方式从 CPU 中获取摄像头图像信息，对获取的图像信息进行边缘检测，并将检测结果直接输到 RawImage 上查看。

首先在层级窗口中新建一个 RawImage UI 对象，并将其位置固定在屏幕上方。新建一个 C# 脚本，命名为 CPUcameraImage，编写代码如下：

```
// 第 9 章 / 9-11
using System;
using Unity.Collections.LowLevel.Unsafe;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;
using UnityEngine.UI;

public class CPUcameraImage : MonoBehaviour
{
    [SerializeField]
    private RawImage mRawImage;           //RawImage UI 对象
    private ARCameraManager mARCameraManager;
    private Texture2D mTexture;           // 转换后的 Texture2D 对象
    private static int mTotalSize;        // 需要转换的图像大小
    private static Byte[] mFinalImage;    // 图像字节数组

    void OnEnable()
    {
        mARCameraManager = GetComponent<ARCameraManager>();
        mARCameraManager.frameReceived += OnCameraFrameReceived;
    }
    // 取消事件注册
    void OnDisable()
    {
        mARCameraManager.frameReceived -= OnCameraFrameReceived;
    }
    unsafe void OnCameraFrameReceived(ARCameraFrameEventArgs eventArgs)
    {
        XRCpuImage image;
        if (!mARCameraManager.TryGetLatestImage(out image))
            return;

        var format = TextureFormat.R8;
        var conversionParams = new XRCpuImage.ConversionParams(image, format,
            XRCpuImage.Transformation.None);
        mTotalSize = image.GetConvertedDataSize(conversionParams);
    }
}
```

```

    if (mTexture == null || mTexture.width != image.width || mTexture.height
        != image.height)
    {
        mTexture = new Texture2D(image.width, image.height, format, false);
        mFinalImage = new Byte[mTotalSize];
    }
    var rawTextureData = mTexture.GetRawTextureData<Byte>();
    try
    {
        image.Convert(conversionParams, new IntPtr(rawTextureData.
GetUnsafePtr()), rawTextureData.Length);
        Sobel(mFinalImage, mTexture.GetRawTextureData(), image.width, image.
height);
        mTexture.LoadRawTextureData(mFinalImage);
        mTexture.Apply();
    }
    finally
    {
        image.Dispose();
    }
    mRawImage.texture = mTexture;
}

```

```

unsafe private static void Sobel(Byte[] outputImage, Byte[] mImageBuffer, int
width, int height)
{
    // 边缘检测的阈值
    int threshold = 128 * 128;
    for (int j = 1; j < height - 1; j++)
    {
        for (int i = 1; i < width - 1; i++)
        {
            // 将处理中心移动到指定位置
            int offset = (j * width) + i;

            // 获取 9 个采样点的像素值
            int a00 = mImageBuffer[offset - width - 1];
            int a01 = mImageBuffer[offset - width];
            int a02 = mImageBuffer[offset - width + 1];
            int a10 = mImageBuffer[offset - 1];
            int a12 = mImageBuffer[offset + 1];
            int a20 = mImageBuffer[offset + width - 1];
            int a21 = mImageBuffer[offset + width];
            int a22 = mImageBuffer[offset + width + 1];

            int xSum = -a00 - (2 * a10) - a20 + a02 + (2 * a12) + a22;
            int ySum = a00 + (2 * a01) + a02 - a20 - (2 * a21) - a22;

```

```

        if ((xSum * xSum) + (ySum * ySum) > threshold)
        {
            outputImage[(j * width) + i] = 0xFF; // 是边缘
        }
        else
        {
            outputImage[(j * width) + i] = 0x1F; // 不是边缘
        }
    }
}
}
}

```

在上述代码中, 采用了同步获取图像的方式, 将转换后的图像直接输入 Sobel() 方法进行边缘检测操作。在使用 Sobel 算子对图像所有像素进行卷积处理时, 设置一个阈值(threshold)对输出像素进行过滤, 达到只渲染边缘的目的。

将 CPUcameraImage 脚本挂载到层级窗口中 AR Session Origin 对象下的 AR Camera 对象上, 并设置好 mRawImage 属性。由于图像转换使用了非托管堆资源, 即在非托管堆中进行, 所以相应代码也标识了 unsafe, 编译 unsafe 代码需要开启 Unity 中的 unsafe 选项, 在 Unity 编辑器中, 依次选择 Edit → Project Settings, 打开工程设置面板, 依次选择 Player → Android 平台 → Other Settings, 勾选 Allow 'unsafe' Code 后的复选框, 如图 9-11 所示。

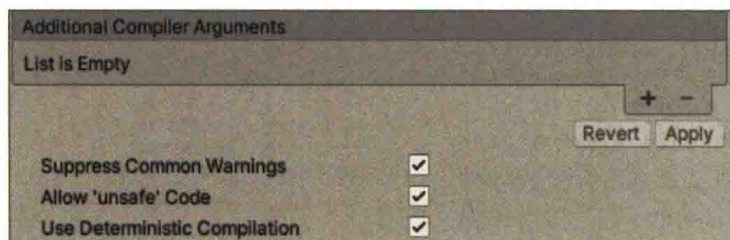


图 9-11 勾选 Allow 'unsafe' Code 后的复选框, 允许编译 unsafe 代码

使用 Sobel 算子做边缘检测的效果如图 9-12 所示, 从结果来看, 边缘检测效果还是非常不错的, 杂点也很少, 但需要注意的是, 由于使用 TextureFormat.R8 格式编码获取灰度图, Sobel 运算后的结果存储成了 Texture2D 格式, 从而导致最终的图像纹理 G、B 通道没有数值(默认为 0), 所以实现时看到的图像纹理呈红色^①, 并且由于是直接获取摄像头原始图像数据, 图像可能会出现 90° 的旋转, 在进行实际开发时需要进行旋转校正。

① 可以将最终的 Texture2D 纹理保存为文件, 通过 PhotoShop 等图像处理软件打开查看各通道数值。



图 9-12 Sobel 算子对 CPU 获取的摄像头图像进行边缘检测效果

9.5 Depth API 概述

在移动 AR 应用中，设备采集摄像头图像，并以此为 AR 场景背景进行渲染。通过 SLAM 系统进行环境感知与运动跟踪，将虚拟对象依据几何一致性原则嵌入实景空间中，形成虚实融合的增强现实环境，再输出到显示系统中呈现给使用者。目前，AR 虚拟物体与真实场景背景在光照、几何一致性方面已取得了非常大的进步，融入真实场景背景的几何、光照属性使虚拟物体看起来更真实。

当将虚拟物体叠加到真实场景中时，虚拟物体与真实场景间存在一定的空间位置关系，即遮挡与被遮挡关系，由于移动 AR 应用无法获取真实环境的深度信息，所以虚拟物体无法与真实场景进行深度测试，即无法实现遮挡与被遮挡，以致虚拟物体一直呈现在真实场景上方，这让虚拟物体看起来像飘在空中，如图 9-13 所示。

正确实现虚拟物体与真实环境的遮挡关系，需要基于对真实环境三维结构的了解，感知真实世界的三维结构、重建真实世界的数字三维模型，然后基于深度信息实现正确的遮挡，但真实世界是一个非常复杂的三维环境，精确快速地感知周围环境，建立一个足够好的真实世界三维模型非常困难，特别是在不使用其他传感器的情况下（如结构光、ToF、双目等）。



图 9-13 虚拟物体无法与真实环境实现正确遮挡

随着移动设备处理性能的提高、深度神经网络技术的普及应用，ARCore 实现了只利用单目相机估计场景深度的功能（Depth API），相比于 ARKit 利用 LiDAR 硬件实现场景深度感知，ARCore 无须额外的硬件支持，只使用单目相机，拓宽了应用范围^①。

场景深度帮助设备了解场景中真实对象的大小和形状。利用 Depth API 开放的场景深度信息，可以实现更加自然、无缝的虚实体验，并且，Depth API 提供的深度信息是逐像素的，因此，可以实现比较精细的效果，如精确地控制特效的范围，营造与现实融合度非常高的虚实效果。

从技术上说，Depth API 采用了运动恢复深度算法（Depth from Motion Algorithm），深度图中的每像素值都与场景中的对应点到摄像头的距离相关。该算法通过从不同角度获取当前场景的多个图像，进行视差计算，估计场景中每个图像点到摄像头的距离。在进行视差计算时，算法也会使用深度神经网络进行深度处理，因此，即使用户的移动幅度很小，也能比较准确地估计出场景深度。该算法还会利用设备可能拥有的其他传感器硬件，如 ToF，算法会自动融合来自所有可用数据源的信息进行综合计算。通过这种方式可以增强深度图像数据的可靠性和精度，并能在设备不移动时进行深度估算，还可以在具有很少或没有特征的曲面（如白墙）上提供更好的深度，甚至可以剔除场景中动态的人或者物体。

使用 Depth API 时，当用户移动设备后，深度数据即可用，该算法可以在 0 ~ 8 米内获得稳健、准确的深度估计。通常，在 0.5 ~ 5 米范围内的数据精度更高、更可信。有效的深度数据只有在用户开始移动设备后才可用（这就是运动恢复深度名字的由来），因为深度是从运动中获取的，并且随着用户设备的移动，深度估计值会越来越好。使用 Depth API，可以获取与摄像头每帧图像匹配的深度图像，因此可以实现实时的动态深度效果，如遮挡、物理仿真、边界处理，获取的深度图像具有与摄像头图像一致的时间戳和尺寸，深度图中每像素代表了场景点到设备摄像头的距离^②（单位为毫米）。真实场景与深度图像示意图如图 9-14 所示。

① ARCore 实现 Depth API 不依赖于特定硬件，但由于算力需求较大，对设备型号有要求，只有其官方列表中标识为 Supports Depth API 的设备才能使用，另外，如果设备本身有额外的传感器，如 ToF 相机、结构光、激光雷达等，这时 ARCore 也会利用这些传感器，得到的场景深度数据会更精准。

② 其实是在场景前向轴上的投影，后文会详细阐述。

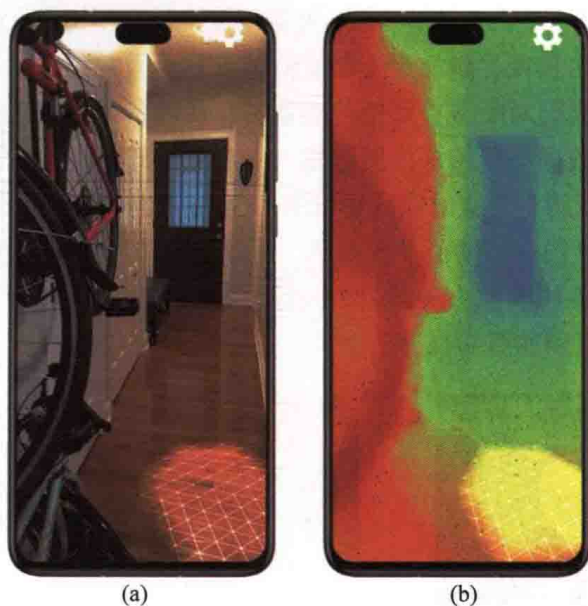


图 9-14 真实场景与深度图像示意图

利用 Depth API 可以获取当前场景的深度值,但这个深度值实际上并不是设备摄像头到场景表面点的距离,而是场景点到摄像头前向轴上的投影,如图 9-15 所示。在图中,C 点为摄像头光心,A 点代表场景中的空间点,而 a 点则是场景空间点在深度图上的投影点,从 Depth API 中获取的深度值不是 CA,而是 CA 在摄像头前向轴上的投影 CB,前向轴并不是世界空间的 Z 轴,而是穿过摄像头光心与成像平面中心点的轴,理解其意义有助于更好地使用场景深度值。

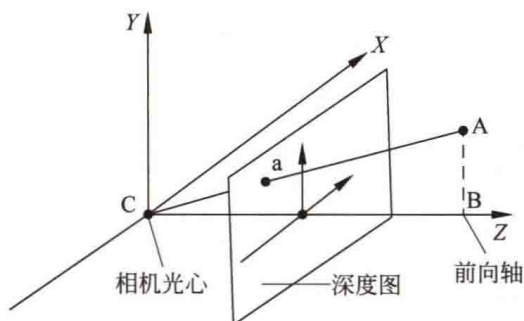


图 9-15 场景深度值示意图

事实上,ARCore 提供的 Depth API 支持两种格式的场景深度图模式,即原始深度图(Raw Depth API)和全深度图(Full Depth API)。

原始深度图数据直接由 ARCore 通过采集环境特征点、深度神经网络估算得到,通常认为这些计算出来的值更准确,但原始深度图由于环境纹理的原因,并不完整(如无纹理区域没有深度数据),因此并不能提供逐像素的场景深度值,即与摄像头采集的图像并不能一一对应,但原始深度图提供了一个额外的深度置信图(Confidence Image),深度置信图用于标识深

度数据的可靠程度,其每个值的范围为 $[0,1]$,0表示深度值最不可靠,1表示深度值非常可靠,中间值代表深度值的置信程度,越靠近1深度值越可靠。通常而言,场景纹理丰富的地方获取的深度值的置信度更高,对配备有其他传感器的移动设备,通常离摄像头近的深度值的置信度更高,深度置信图数据与原始深度图数据一一对应,即深度置信图也是不完整的。需要强调的是,只有原始深度图有对应的深度置信图,全深度图没有对应的深度置信图。

全深度图由原始深度图进行插值平滑得到,全深度图与原始深度图有同样的格式和尺寸,但全深度图与摄像头采集的图像一一对应,即每个图像像素都有一个对应的场景深度值。全深度图与原始深度图的区别如图9-16所示。

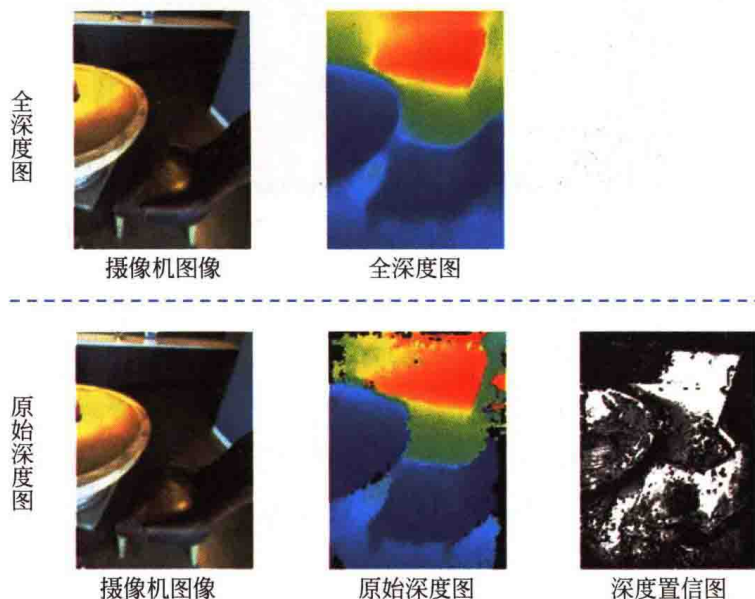


图 9-16 全深度图与原始深度图区别

9.6 Depth API 实例

由于 AR Foundation 的封装^①, Depth API 功能已被集成到 AR Occlusion Manager 组件中,通过该组件,可以获取全深度图、原始深度图、深度置信图等数据信息,还可以直接获取 Texture2D 类型的全深度纹理和原始深度纹理等,非常方便实用。

由于 Depth API 特性并不是所有支持 ARCore 的手机都支持,因此,在使用 Depth API 之前,应当进行支持性检查,检查代码如下:

```
// 第 9 章 / 9-12
if (occlusionManager.descriptor?.environmentDepthImageSupported == Supported.
Supported)
```

① 使用 Depth API 需要 AR Foundation 4.2 以上插件包支持。

```
{
    // 支持 Depth API
    //TODO
}
```

在确保设备支持 Depth API 之后，获取深度图及深度置信图的代码如下：

```
// 第9章 /9-13
// 设置获取全深度图或者原始深度图，设置为 true 时为全深度图，设置为 false 时为原始深度图
occlusionManager.environmentDepthTemporalSmoothingRequested = true;

// 获取深度图，可以是全深度图，也可以是原始深度图
if (occlusionManager.TryAcquireEnvironmentDepthCpuImage(out XRCpuImage image))
{
    using (image)
    {
        // 获取深度图
    }
}

// 获取深度置信图
if (occlusionManager && occlusionManager.TryAcquireEnvironmentDepthConfidenceCpuImage(out XRCpuImage image))
{
    using (image)
    {
        // 获取深度置信图
    }
}
```

在了解了以上基础知识之后，下面通过一个实例演示获取设备屏幕中心点对应的场景深度值的方法。首先，在场景对象 AR Session Origin → AR Camera 游戏对象上挂载 AR Occlusion Manager 组件，将其 Environment Depth Mode 属性设置为 Fastest，勾选 Temporal Smoothing 属性后的复选框，如图 9-17 所示。

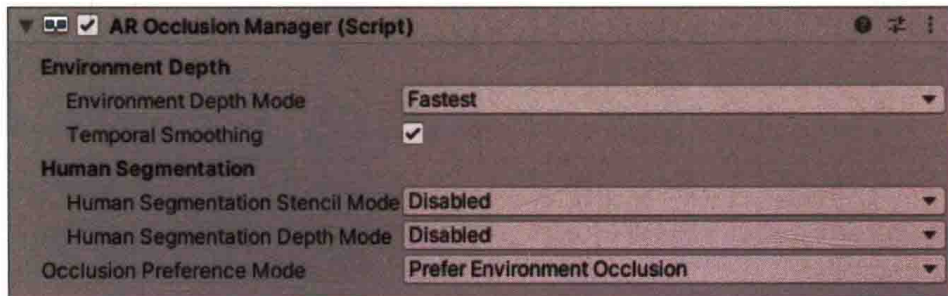


图 9-17 在 AR Camera 游戏对象上挂载 AR Occlusion Manager 组件

新建一个 C# 脚本，命名为 DepthMapDataController，编写代码如下：

```
// 第 9 章 / 9-14
using System;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;

public class DepthMapDataController : MonoBehaviour
{
    public AROcclusionManager mAROcclusionManager;
    public ARCameraManager mARCameraManager;
    private bool mUseSmoothDepth = true; // 是否使用平滑后的数据
    private short[] mDepthArray = new short[0]; // Short 型的深度值数组
    private Byte[] mDepthBuffer = new Byte[0]; // 字节型的深度值数组
    private Byte[] mConfidenceArray = new Byte[0]; // 置信度数据
    private int mDepthHeight = 0; // 深度图高度
    private int mDepthWidth = 0; // 深度图宽度
    private const float MillimeterToMeter = 0.001f; // 将毫米转换成米系数
    private readonly Vector2 mScreenPosition = new Vector2(0.5f, 0.5f); // 屏幕中心点位置
    private float mDepthValue = -1.0f; // 指定点的深度值
    private bool isDepthAPISupported = false; // 是否支持 DepthAPI

    void Start()
    {
        if (mAROcclusionManager.descriptor?.environmentDepthImageSupported ==
            Supported.Supported)
        {
            isDepthAPISupported = true;
            mARCameraManager.frameReceived += OnCameraFrameReceived;
            mAROcclusionManager.environmentDepthTemporalSmoothingRequested =
                mUseSmoothDepth; // 使用平滑后的深度数据
        }
    }

    void Update()
    {
        if (isDepthAPISupported)
        {
            mDepthValue = GetDepthFromUV(mScreenPosition);
            if (mDepthValue != 0)
            {
                Debug.Log(mDepthValue.ToString());
            }
        }
    }

    private void OnDestroy()
    {
        if (isDepthAPISupported)
            mARCameraManager.frameReceived -= OnCameraFrameReceived;
    }
}
```



```

}
private void OnCameraFrameReceived(ARCameraFrameEventArgs eventArgs)
{
    UpdateEnvironmentDepthImage();
}
// 获取每一帧的深度图
private void UpdateEnvironmentDepthImage()
{
    if (mAROcclusionManager && mAROcclusionManager.TryAcquireEnvironmentDepthCpuImage(out XRCpuImage image))
    {
        using (image)
        {
            mDepthWidth = image.width;
            mDepthHeight = image.height;

            int numPixels = image.width * image.height;
            int numBytes = numPixels * image.GetPlane(0).pixelStride;

            if (mDepthBuffer.Length != numBytes)
            {
                mDepthBuffer = new Byte[numBytes];
            }

            image.GetPlane(0).data.CopyTo(mDepthBuffer);

            // 将字节数据转换为 Short 数组
            int bufferLength = mDepthWidth * mDepthHeight;
            if (mDepthArray.Length != bufferLength)
            {
                mDepthArray = new short[bufferLength];
            }

            Buffer.BlockCopy(mDepthBuffer, 0, mDepthArray, 0, mDepthBuffer.Length);
        }
    }
}
// 获取每一帧的深度置信图
private void UpdateEnvironmentDepthConfidenceImage()
{
    if (mAROcclusionManager && mAROcclusionManager.TryAcquireEnvironmentDepthConfidenceCpuImage(out XRCpuImage image))
    {
        using (image)
        {
            mDepthWidth = image.width;

```

```

        mDepthHeight = image.height;

        int numPixels = image.width * image.height;
        int numBytes = numPixels * image.GetPlane(0).pixelStride;

        if (mConfidenceArray.Length != numBytes)
        {
            mConfidenceArray = new Byte[numBytes];
        }
        image.GetPlane(0).data.CopyTo(mConfidenceArray);
    }
}

// 获取深度图纹理
private void UpdateDepthTexture(ref Texture2D depthTexture)
{
    depthTexture = mAROcclusionManager.environmentDepthTexture;
}

// 获取深度置信图纹理
private void UpdateDepthConfidenceTexture(ref Texture2D depthConfidenceTexture)
{
    depthConfidenceTexture = mAROcclusionManager.environmentDepthConfidenceTexture;
}

// 获取指定点的深度值, 单位为米
private float GetDepthFromUV(Vector2 uv)
{
    if (mDepthArray.Length <= 0)
    {
        return 0.0f;
    }

    int depthX = (int)(uv.x * (mDepthWidth - 1));
    int depthY = (int)(uv.y * (mDepthHeight - 1));
    if (depthX >= mDepthWidth || depthX < 0 || depthY >= mDepthHeight ||
        depthY < 0)
    {
        return 0.0f;
    }

    var depthIndex = (depthY * mDepthWidth) + depthX;
    var depthInShort = mDepthArray[depthIndex];
    var depthInMeters = depthInShort * MillimeterToMeter;
    return depthInMeters;
}
}

```

上述代码实现了获取设备屏幕中心点对应的场景深度值的功能。在代码中,首先检查设备是否支持 Depth API,如果支持,则通过 AR Camera Manager 组件获取每一帧的数据信息,使用 UpdateEnvironmentDepthImage() 方法获取每一帧的场景深度值,同时保存到 mDepthBuffer 字节数组和 mDepthArray Short 类型数组中。GetDepthFromUV() 方法获取归一化的 UV 坐标对应的场景深度值,并将单位从毫米转换为米。

代码通过 mUseSmoothDepth 变量控制获取全深度图或者原始深度图,在获取原始深度图时,可以通过 UpdateEnvironmentDepthConfidenceImage() 方法获取对应的深度置信图。同时,代码还演示了直接获取深度图纹理和深度置信度纹理的方法^①。

将 DepthMapDataController 脚本挂载到场景中的任何对象上,并正确地对 mAROcclusionManager 和 mARCameraManager 属性进行赋值后,在支持 Depth API 的设备上,即可在控制台看到场景深度数值输出。

经过测试,通过 Depth API 获取的场景深度信息还是比较准确的,能满足日常绝大部分应用需求。当然,如果希望通过 Depth API 进行场景三维重建或者物体三维重建等对精度要求更高的工作,建议使用配备深度硬件传感器的设备(如 ToF、LiDAR、结构光),Depth API 支持融合多传感器数据,可以获得更高精度的场景深度信息。

9.7 场景深度应用场景

通过 ARCore Depth API 获取当前摄像头场景深度信息,在此基础上就可以实现环境遮挡、物理仿真、虚拟光照等各种应用,在提升 AR 沉浸体验的同时,也可以实现很多特效和创新的交互方式,典型的应用场景包括以下类型。

1. 环境遮挡

环境遮挡是提升虚实融合真实感的重要方面,当一个虚拟物体能正确地与真实环境产生遮挡与被遮挡关系时,这将极大地增强虚拟物体的置信度,因此,该应用场景也是场景深度的一个重要应用领域。

2. 物理仿真

使用物理仿真后,当一个虚拟的小球从半空掉落到地板时,小球将从地板弹起并产生与真实小球一样的物理行为。物理仿真使用数学的方式模拟物体的行为,可以复现与真实世界同样的摩擦、黏滞、弹性、碰撞等现象,在使用物理引擎后,不需要进行虚拟物体行为编码,物理引擎会依据牛顿运动定律、弹簧定律等物理定律模拟仿真虚拟物体受力,实现虚拟物体与真实环境的自然交互。

^① 只有设置获取原始深度图时才能获取深度置信图。

3. 景深

由于人眼的特性,聚焦处的物体会在人眼中清晰成像,而其他部分则会虚化,这有助于突出关注点。利用场景深度信息,就可以实现景深效果,模糊虚化非关注点的场景。景深效果的实现,符合人眼观察世界的方式,可以提升用户体验。

4. 虚实交互

在获取场景深度后,虚实交互变得更加自然。获取场景深度后,事实上是获取了当前场景的结构信息,有了这些结构信息,虽然不知道语义,但远比仅检测到平面要精准得多,碰撞检测、物理仿真更加真实可靠。技术上,可以获取场景的法线信息,利用这些法线信息,物体放置、碰撞、渲染则更符合真实场景,交互也更自然。

5. 碰撞检测

场景深度信息有助于改善射线检测结果。通常,平面碰撞检测仅适用于具有纹理的地板、地面,利用场景深度信息,射线检测更为具体详细,可以适用于非平面和低纹理区域,也可以确定碰撞点的正常深度和方向。

6. 特效

没有场景结构信息,所有的特效都无法与真实场景结合,也只能浮于场景之上,有明显的分层感觉。通过获取场景深度信息,特效可以与真实场景结合,如雪花落在地板、沙发、桌面、植物表面;实现场景雾效,营造真实场景渐隐的效果;甚至可以利用虚拟灯光照亮真实物体表面。这些特效更真实,可以有效提升 AR 沉浸体验。

第 10 章



相机与手势操作

虚实交互是 AR 应用生命力的重要基础，基于屏幕的手势操作是用户与 AR 场景交互的基本方式，也是目前最成熟、最符合大众使用习惯的方式。在实际应用项目中，有时需要对整个 AR 场景进行调整，如调整大小尺寸、角度等，这时就需要对 AR 虚拟相机进行操作，以达到快速对整个数字场景进行调整的目的。本章主要学习屏幕手势操作和虚拟相机操作相关知识，熟悉屏幕操作及利用相机平移、缩放、旋转 AR 场景的一般方法。

10.1 AR 场景操作

为更弹性地渲染虚拟场景，AR Session Origin 对象允许整体缩放 AR 虚拟场景及其相对渲染摄像机（Camera，以下简称相机）的偏移。在使用缩放或偏移功能时，相机必须作为 AR Session Origin 对象的子对象（默认就是子对象），由于相机位姿由 SLAM 运动跟踪驱动并且是 AR Session Origin 对象的子对象，因此设置 AR Session Origin 对象的缩放与偏移值，相机和检测到的可跟踪对象将随之缩放与偏移，从而实现 AR 场景整体缩放与偏移的目的。

通过移动、旋转和缩放 AR Session Origin 对象而不是操作具体的虚拟对象，一方面可以对 AR 场景产生整体影响，另一方面可以避免场景生成后操作虚拟对象带来的问题（AR 场景的特点是虚实融合，移动虚拟对象会破坏沉浸体验，复杂场景通常在创建后不应当移动，如地形），并且虚拟对象缩放会对其他系统（例如物理、粒子效果和导航网格）产生负面影响。在需要将 AR 场景整体进行平移时，如将场景从地面移动到桌面，或者将整个 AR 场景缩放到某个运行时才能确定的尺寸（如检测到的桌面平面），AR Session Origin 对象的平移、旋转、缩放功能非常有用。

10.1.1 场景操作方法

将偏移应用于 AR Session Origin 对象，可以直接设置其 Transform 组件的 Position 值，这里设置的偏移会影响所有虚拟物体的渲染。除此之外，AR Foundation 还提供了 MakeContentAppearAt() 方法，该方法有 3 个重载，利用它可以非常方便地将场景放置到指定

位置。MakeContentAppearAt() 方法不会修改场景中具体虚拟对象的位置及大小，它直接更新 AR Session Origin 对象的 Transform 组件的相应值，使场景处于给定的位置和方向，这对于在运行时无法移动具体虚拟对象时将 AR 场景放置到指定平面上非常有用，例如，在数字场景中包括地形或导航网格时，则无法 / 不应当手动移动或旋转具体虚拟对象。

将缩放应用于 AR Session Origin 对象，可以直接设置其 Transform 组件的 Scale 值，更大的 Scale 值会使 AR 场景显示得更小，例如，10 倍的 Scale 值将使虚拟场景显示缩小到原大小的 1/10，而 0.1 倍的 Scale 值则会将虚拟场景放大 10 倍。在需要整体缩放数字场景时可以考虑调整该值。

将旋转应用于 AR Session Origin 对象，可以直接设置其 Transform 组件的 Rotation 值，一般而言，AR 场景不应当对 X、Z 轴做旋转，而只对 Y 轴做旋转。

10.1.2 场景操作实例

新建一个场景，为了便于将 AR 场景放置到指定位置，创建一个空对象，命名为 Contents，然后将所有需要整体操作的对象放置到该对象下，如灯光、运行时实例化的对象等。为了方便操作及观看演示效果，新建两个 UI Slider 控件，一个命名为 ScaleSlider，用于缩放场景，另一个命名为 OrientationSlider，用于旋转场景。新建一个 C# 脚本，命名为 AppController，编写代码如下：

```
// 第 10 章 /10-1
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;
using UnityEngine.UI;

[RequireComponent(typeof(ARRaycastManager))]
public class AppController : MonoBehaviour
{
    public GameObject spawnPrefab;           // 需要放置的虚拟物体预制体
    public ARSessionOrigin mARSessionOrigin; // Session Origin 组件
    public Transform mContent;               // AR 场景父对象
    public Slider mRotationSlider;           // 旋转控制滑动条
    public Slider mScaleSlider;              // 缩放控制滑动条

    private static List<ARRaycastHit> Hits; // 射线检测结果
    private ARRaycastManager mRaycastManager;
    private GameObject spawnedObject = null; // 实例化后的虚拟对象
    private readonly float MinAngle = 0f;    // 最小旋转角度
    private readonly float MaxAngle = 360f;  // 最大旋转角度
    private readonly float MinScale = 0.1f;  // 最小缩放倍率
    private readonly float MaxScale = 10f;   // 最大缩放倍率
}
```



```

private void Start()
{
    Hits = new List<ARRaycastHit>();
    mRaycastManager = GetComponent<ARRaycastManager>();
    mRotationSlider.onValueChanged.AddListener(OnRotate);
    mScaleSlider.onValueChanged.AddListener(OnScale);
}
// 单击放置虚拟物体
void Update()
{
    if (Input.touchCount == 0)
        return;
    var touch = Input.GetTouch(0);
    if (mRaycastManager.Raycast(touch.position, Hits, TrackableType.
PlaneWithinPolygon | TrackableType.PlaneWithinBounds))
    {
        var hitPose = Hits[0].pose;
        if (spawnedObject == null)
        {
            spawnedObject = Instantiate(spawnPrefab, hitPose.position,
hitPose.rotation);
            mARSessionOrigin.MakeContentAppearAt(mContent, hitPose.position,
hitPose.rotation);
            spawnedObject.transform.parent = mContent;
        }
    }
}
// 缩放 AR 场景
public void OnScale(float value)
{
    if (mARSessionOrigin != null)
    {
        var scale = value * (MaxScale - MinScale) + MinScale;
        mARSessionOrigin.transform.localScale = Vector3.one * scale;
    }
}
// 旋转和平移 AR 场景
public void OnRotate(float value)
{
    if (mARSessionOrigin != null)
    {
        var angle = value * (MaxAngle - MinAngle) + MinAngle;
        mARSessionOrigin.MakeContentAppearAt(mContent, mContent.transform.
position, Quaternion.AngleAxis(angle, Vector3.up));
    }
}
}

```

在上述代码中,当检测到用户操作时,在指定位置实例化一个预制体,随后将实例化对象设置为 Contents 对象的子对象。在 OnScale() 方法中,通过调整 AR Session Origin 对象 Transform 的 LocalScale 属性值来达到缩放场景的目的,在 OnRotate() 方法中,则直接通过 MakeContentAppearAt() 方法调整场景的偏移和旋转。

将该脚本挂载在场景中 AR Session Origin 对象上,并设置好相应属性,编译运行,在检测到的平面上放置立方体,通过拖动滑动条滑块,可以看到整个场景都将随之缩放或者旋转,如图 10-1 所示。

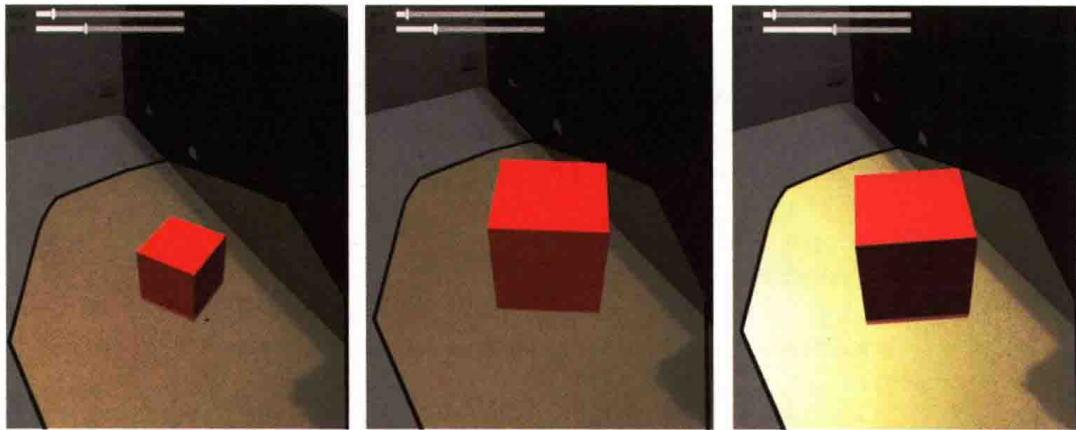


图 10-1 操作 AR 场景缩放旋转效果图

10.2 手势操作交互

AR Foundation 提供了特征点与平面检测功能,利用检测到的特征点和平面,就可以在其上放置虚拟对象了。SLAM 算法能实时跟踪这些点和平面,当用户移动时,连接到这些特征点或者平面上的虚拟对象会进行姿态更新,从而保持它们与真实环境的相对位置关系,就像真实存在于环境中一样。除了通过算法对虚拟物体进行自动姿态更新外,很多时候也希望与 AR 场景中的虚拟物体进行交互,如旋转、缩放、平移虚拟物体等。

智能移动设备的手势操作是使用者接受并已习惯的操作方式,在移动端 AR 应用中,对虚拟物体的操作也基本通过手势操作完成,本节将学习手势检测基础知识。需要注意的是本节中描述的手势检测是指用户在手机屏幕上的手指操作检测,不是指利用计算机视觉处理技术对使用者手部运动的检测。

10.2.1 手势检测

在本节之前,其实已经使用了手势检测,如通过单击屏幕在检测到的平面上放置虚拟对象,这是最基础最简单的手势,下面学习一些手势检测基础知识和通过手势控制虚拟对象行为的一般操作。

1. 手势检测定义

手势检测是指通过检测用户在手机屏幕上的手指触控运动来判断用户操作意图的技术，如单击、双击、缩放、滑动等，常见的手势操作如图 10-2 所示。



图 10-2 常见的手势操作

2. Unity 中的手势检测

Unity 提供了对触控设备底层 API 的访问权限和高级手势检测功能，可以满足不同手势定制需求。底层 API 访问能够获取手指单击的原始位置、压力值、速度信息，高级手势检测功能则借助手势识别器来识别预设手势（包括单击、双击、长按、缩放、平移等）。Unity 通过 Input 类管理设备输入及相关操作（包括手势触控输入），Unity 常见的与手势检测相关的方法属性如表 10-1 所示。

表 10-1 Input 类常见手势检测相关的方法属性

方法属性名	描 述
Input.GetTouch(int index)	返回特定触控状态对象，该方法返回 Touch 结构体
Input.touchCount	接触触摸屏的手指数量，只读属性
Input.touches	返回上一帧中所有触控状态对象 List，每个 List 值表示一个手指触控状态
touchPressureSupported	Bool 值，检测用户手机设备是否支持触控压力特性
touchSupported	Bool 值，检测用户手机设备是否支持触控输入

1) Touch 结构体

Touch 结构体用于描述手指触控屏幕的状态。Unity 可以跟踪触摸屏上与触控有关的各类数据，包括触控阶段（开始、移动、结束）、触控位置及该触控是一个单一的触点还是多个触点。此外，Unity 还可以检测到帧更新之间触控的连续性，因此可以检测到一致的 fingerId 并用来确定手指如何移动。Touch 结构体被 Unity 用来存储与单个触控实例相关的数据，并由 Input.GetTouch() 方法查询和返回。在 Unity 内部，每次帧更新都需要对 GetTouch 进行一次调用，以便从设备获得最新的触控信息，还可以通过手指 fingerId 来识别帧间的相同触控，Touch 结

构体更多的属性如表 10-2 所示。

表 10-2 Touch 结构体

结构体属性	描 述
altitudeAngle	触控时的角度, 0 值表示触头与屏幕表面平行, $\pi/2$ 表示垂直
azimuthAngle	手写笔的角度, 0 值表示手写笔沿着设备的 X 轴指向
deltaPosition	上次触控变化后的位置增量, delta 值
deltaTime	自上一次记录的触点值变化以来已流逝的时间, delta 值
fingerId	触控的唯一索引值
maximumPossiblePressure	屏幕触摸的最大压力值, 如果 Input.touchPressureSupported 返回值为 false, 则此属性值保持为 1.0F
phase	描述触控阶段
position	以像素坐标表示的触控位置
pressure	当前施加在屏幕上的压力, 1.0F 被认为是平均触点的压力, 如果 Input.touchPressureSupported 返回值为 false, 则此属性的值保持为 1.0F
radius	一个触点半径的估计值, 加上 radiusVariance 值得到最大触控尺寸, 减去 radiusVariance 值得到最小触控尺寸
radiusVariance	该值决定了触控半径的准确性, 把该值加到半径中, 可得到最大的触控尺寸, 减去它则可得到最小的触控尺寸
rawPosition	触控原始位置
tapCount	触点的数量
type	指示触控是直接、间接 (或远程) 还是手写笔类型的值

2) TouchPhase 枚举

TouchPhase 是一个枚举类型, 用于描述手指触控阶段类型。Unity 可以跟踪触控的连续状态, 包括按压开始、保持、移动、结束等, 触控阶段指示了触控在最新帧中所处的阶段, TouchPhase 枚举值如表 10-3 所示。

表 10-3 TouchPhase 枚举值

枚 举 值	描 述
Began	手指接触到屏幕, 开始跟踪
Moved	手指在屏幕上移动
Stationary	手指触摸着屏幕, 保持接触, 但没有移动
Ended	手指从屏幕上抬起, 跟踪结束
Canceled	系统取消了对触摸的跟踪

10.2.2 手势操作控制

本节将运用手势检测基础知识, 结合 Unity 的 Input 类配合射线检测功能实现对虚拟物体

的操控,为更好地学习原理,不使用任何手势插件。

在单物体场景中对物体操控时,甚至可以不用射线检测来查找需要操作的虚拟对象,因为场景中只有一个物体,只需要在实例化该对象时将其保存到全局变量中,在进行手势操作时只针对这个全局变量进行操作。

新建一个 C# 脚本,命名为 `GestureManager`,编写代码如下:

```
// 第 10 章 / 10-2
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;

[RequireComponent(typeof(ARRaycastManager))]
public class GestureManager : MonoBehaviour
{
    public GameObject spawnPrefab;           // 需要放置物体的预制体
    private static List<ARRaycastHit> Hits;   // 射线检测碰撞结果
    private ARRaycastManager mRaycastManager;
    private GameObject spawnedObject = null;  // 实例化后的虚拟物体
    private readonly float mRotateSpeed = -20f; // 旋转速率
    private readonly float mZoomSpeed = 0.01f;  // 缩放速率

    private void Start()
    {
        Hits = new List<ARRaycastHit>();
        mRaycastManager = GetComponent<ARRaycastManager>();
    }

    void Update()
    {
        switch (Input.touchCount)
        {
            case 1:
                if (spawnedObject == null)
                    onSpawnObject();
                else
                    onRotate();
                break;
            case 2: onZoom(); break;
            default: return;
        }
    }

    // 放置虚拟物体
    private void onSpawnObject()
    {
        var touch = Input.GetTouch(0);
```

```

        if (mRaycastManager.Raycast(touch.position, Hits, TrackableType.
PlaneWithinPolygon | TrackableType.PlaneWithinBounds))
        {
            var hitPose = Hits[0].pose;
            spawnedObject = Instantiate(spawnPrefab, hitPose.position, hitPose.
rotation);
        }
    }
    // 缩放虚拟物体
    private void onZoom()
    {
        if (spawnedObject != null)
        {
            Touch touchZero = Input.GetTouch(0);
            Touch touchOne = Input.GetTouch(1);
            Vector2 touchZeroPrevPos = touchZero.position - touchZero.deltaPosition;
            Vector2 touchOnePrevPos = touchOne.position - touchOne.deltaPosition;
            float prevTouchDeltaMag = (touchZeroPrevPos - touchOnePrevPos).magnitude;
            float touchDeltaMag = (touchZero.position - touchOne.position).magnitude;
            float deltaMagnitudeDiff = touchDeltaMag - prevTouchDeltaMag;
            float pinchAmount = deltaMagnitudeDiff * mZoomSpeed * Time.deltaTime;
            spawnedObject.transform.localScale += new Vector3(pinchAmount,
pinchAmount, pinchAmount);
        }
    }
    // 旋转虚拟物体
    private void onRotate()
    {
        Touch touch;
        touch = Input.GetTouch(0);
        if (touch.phase == TouchPhase.Moved)
        {
            spawnedObject.transform.Rotate(Vector3.up * mRotateSpeed * Time.
deltaTime * touch.deltaPosition.x, Space.Self);
        }
    }
}

```

在代码中，onSpawnObject() 方法负责在用户单击检测到的平面时放置虚拟物体，这里使用了射线检测来确定射线与平面的碰撞位置，用于设置虚拟物体的姿态（位置与方向）。

onRotate() 方法用于检测用户触控操作，使用 touch.deltaPosition.x 值来控制旋转，touch.deltaPosition 存储了两次手势检测之间手指在屏幕上移动的位置数据，这里只使用了 X 轴的 delta 值。

```
Vector3.up * mRotateSpeed * Time.deltaTime * touch.deltaPosition.x
```


该语句用于获取虚拟物体绕其本身 Y 轴的旋转量, 通过这个旋转量就可以控制虚拟物体的旋转。

onZoom() 方法负责缩放虚拟物体, 缩放使用双指, 首先获取每个手指的当前位置与前一次检测时的位置, 然后分别计算当前两个手指之间的距离与前一次两手指之间的距离, 这么做的目的是获取两次距离之差, 这个差值即是手指放大或缩小的度量值, 如果当前两指之间的距离大于之前两指之间的距离, 说明用户在使用放大手势, 反之, 用户就是在使用缩小手势。

```
deltaMagnitudeDiff * mZoomSpeed * Time.deltaTime
```

该语句用于计算缩放量, 通过这个缩放量就可以缩放物体了。

在使用时, 将 GestureManager 脚本挂载于 AR Session Origin 对象上并设置好需要放置的虚拟对象即可。编译运行, 在检测到平面后单击放置虚拟物体, 可以使用单指左右滑动旋转虚拟物体, 或者使用双指捏合缩放虚拟物体。

示例演示了在单物体场景中操控对象的一般方法, 对多物体场景, 手势操控时首先需要明确操作对象, 即通过射线检测选择被操作对象, 然后进行操作。通常为了更好的操作体验、告知用户将要操作的对象, 需要将选中的对象通过颜色、大小变化、包围盒等方式可视化并展示出差异。

10.3 XR Interaction Toolkit

直接使用手势检测可以非常灵活地进行手势操作控制, 但需要开发人员自行处理细节, 而且就 AR 应用而言, 很多操作具有共通性, 重复发明轮子可能并不是最明智的选择^①。事实上, Unity 已经推出了 XR 交互工具包 (XR Interaction Toolkit, XRIT), 从工具包名字可以看出, XRIT 的目标已经不仅是 AR 交互, 也囊括了 VR 交互操作。借助 XRIT, 开发人员无须从头开始编写交互代码, 可以快速构建交互操作及交互视觉表现, 同时支持对交互的扩展, 兼容 AR Foundation 支持的所有 AR/MR 平台。

如前所述, XRIT 支持 VR、AR/MR 交互操作, 由于 VR 交互操作比较复杂, XRIT 引入了很多专业概念和术语, 但 AR/MR 操作相对简单, 这些概念和术语对我们意义不大, 下面主要讲述利用 XRIT 进行 AR/MR 的交互操作, 不详细阐述与 AR/MR 操作无关的概念和术语^②。具体而言, XRIT 支持对 AR 对象放置、选择、平移、缩放、旋转操作, 同时支持操作时的视觉外观呈现。

目前 XRIT 已经更新到 v2.0 版本, 可以通过 Unity 包管理器 (Unity Package Manager) 进行导入, 在 Unity 菜单中, 依次选择 Window → Package Manager 打开包管理器, 选择 XR Interaction Toolkit 进行安装, 安装该工具包时, 会自动安装其所依赖的 XR Plugin Management

① 在实现更细致或者更个性化的手势操作时, 仍然需要直接对手势进行处理。

② 读者可查阅官方文档, 网址为 <https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.0/manual/locomotion.html>。

插件库, 如图 10-3 所示。

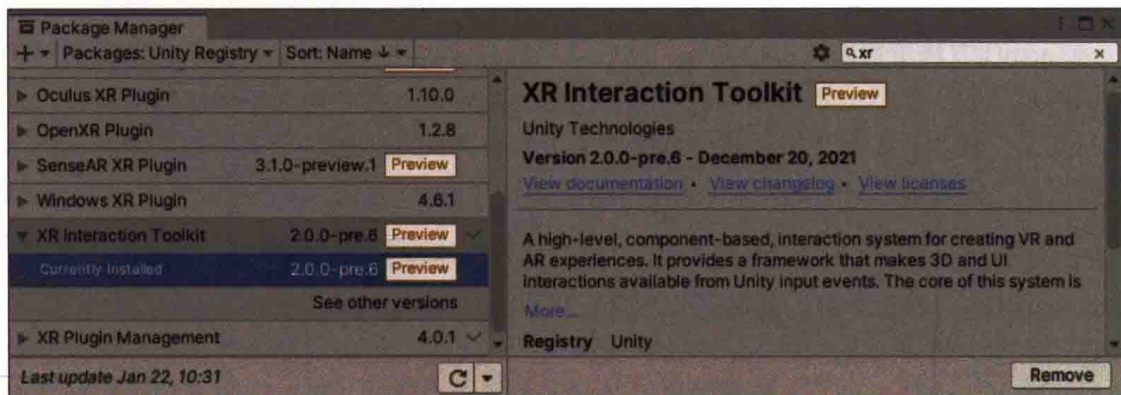


图 10-3 安装 XR Interaction Toolkit 工具包

安装导入该工具包后, 可能会与 Unity 默认使用的输入处理方式冲突, 在 Unity 菜单中, 依次选择 Edit → Project Settings → Player, 打开 Android 平台 → Other Settings, 将 Active Input Handling 属性设置为 Input System Package(new) 或者 Both 值^①。

为应用新的输入处理工具, 需要对场景进行两处设置:

(1) 在层级窗口中选择 AR Session Origin → ARCamera 对象, 为其挂载 AR Gesture Interactor 组件, 并将其 AR Session Origin 属性设置为层级窗口中的 AR Session Origin 对象, 其他属性可以根据需求进行设置。

(2) 在层级窗口中选择 AR Session 对象, 为其挂载 XR Interaction Manager 组件。

完成这两处设置, 现在 XRIT 已经能完成功能初始化, 为使用做好了准备。

10.3.1 对象放置

XRIT 提供了 AR Placement Interactable 组件用于放置 AR 对象, 在层级窗口中选择 AR Session 对象, 为其挂载该组件, 挂载后, 该组件会自动引用 XR Interaction Manager 组件和 AR Session Origin 组件。一般使用时, 只需将需要放置的虚拟对象预制体设置到 Placement Prefab 属性。

需要注意的是, AR Placement Interactable 组件在使用时仍然需要 AR Plane Manager 组件进行平面检测, 也需要 AR Raycast Manager 组件进行射线检测, 所以该放置组件正常工作的前提是场景中必须挂载这两个组件。

AR Placement Interactable 组件各属性及意义如表 10-4 所示。

^① 导入 XRIT 工具包后可能会要求 Unity 重启以应用新的输入处理工具。

表 10-4 AR Placement Interactable 组件各属性意义

属 性 名	描 述
Interaction Manager (交互管理器)	XR Interaction Manager 组件引用
Interaction Layer Mask (交互层掩码)	射线检测的碰撞层掩码
Colliders (碰撞器)	可以与射线检测进行碰撞检测的对象组
Custom Reticle (自定义指示器)	显示射线检测与其他对象交互点, 目前仅用于 VR
Select Mode (选择模式)	可交互对象选择类型, 支持单选和多选
XR Origin	用于 AR Foundation 5.0 及以后版本的 XR Origin 组件引用
AR Session Origin	用于 AR Foundation 5.0 之前版本的 AR Session Origin 组件引用
Placement Prefab (放置对象预制体)	用于放置的虚拟对象预制体
Fallback Layer Mask (回退层掩码)	当进行射线检测时, 如果没有与 AR 检测到的平面发生碰撞, 则提供给 Unity 默认的碰撞层掩码

从上述属性可以看到, AR Placement Interactable 组件使用界面比较简洁, 在进行交互操作时, 该组件还提供了全局性的交互事件, 这些交互事件被分成了 6 类, 对交互各阶段进行了划分, 方便开发人员使用, 这些事件如表 10-5 所示。

表 10-5 AR Placement Interactable 组件提供的全局性事件

类 型	事 件 名	描 述
对象操作	ObjectPlaced	虚拟物体放置后触发
First/Last Hover (第一次 / 最后一次聚焦)	First Hover Entered	第一次聚焦虚拟对象时触发 (全局)
	Last Hover Exited	最后一次聚焦虚拟对象退出时触发 (全局)
Hover (聚焦)	Hover Entered	聚焦虚拟对象时触发 (全局)
	Hover Exited	聚焦虚拟对象退出时触发 (全局)
First/Last Select (第一次 / 最后一次选择)	First Select Entered	第一次选择虚拟对象时触发 (全局)
	Last Select Exited	最后一次选择虚拟对象后取消选择时触发 (全局)
Select (选择)	Select Entered	选择虚拟对象时触发 (全局)
	Select Exited	选择虚拟对象后取消选择时触发 (全局)
Activate (激活)	Activated	当 XRIT 架构激活该组件时触发 (非 Unity 游戏对象中的激活)
	Deactivated	当 XRIT 架构取消激活该组件时触发 (非 Unity 游戏对象中的取消激活)

利用 AR Placement Interactable 组件事件可以介入交互操作的各阶段, 方便灵活地控制各阶段的交互状态, 但需要注意的是, 该组件提供的事件都是全局交互事件, 与虚拟对象本身的事件有所区别。

利用 AR Placement Interactable 组件放置虚拟对象简单方便, 但相比于自己通过代码控制放置过程灵活性会受到一定影响, 例如不能控制放置虚拟对象的数量, 默认情况下只要用户

单击检测到的平面就会生成虚拟对象。为解决该问题,可以通过 Object Placed 事件,在放置一个虚拟对象后将 Placement Prefab 属性设置为 null,但这种方式会形成一个循环引用^①,不够优雅,为此,可以直接继承 ARBaseGestureInteractable 类,添加相应的功能控制逻辑,代码如下:

```
// 第10章/10-3
using System;
using System.Collections.Generic;
using UnityEngine.Events;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;
using UnityEngine;
using UnityEngine.XR.Interaction.Toolkit.AR;

// 定义事件
[Serializable]
public class ARObjPlacementExtensionEvent : UnityEvent<ARObjPlacementExtensionEventArgs> { }
// 定义事件参数
public class ARObjPlacementExtensionEventArgs
{
    public ARPlacementInteractableExtension placementExtension { get; set; }
    public GameObject placementObject { get; set; }
}

public class ARPlacementInteractableExtension : ARBaseGestureInteractable
{
    public GameObject placementPrefab; // 需要放置的对象预制体
    [SerializeField]
    private int MaxPlaceObjectNum = 2; // 最大可放置的虚拟对象数量
    public LayerMask fallbackLayerMask; // 回退层掩码
    public ARObjPlacementExtensionEvent objectPlaced; // 放置对象后事件

    private readonly ARObjPlacementExtensionEventArgs mEventArgs = new ARObjPlacementExtensionEventArgs();
    private static readonly List<ARRaycastHit> mHits = new List<ARRaycastHit>();
    private int mPlacedObjectNum = 0; // 当前已放置的虚拟对象数量

    // 触发订阅事件
    protected virtual void OnObjectPlaced(ARObjPlacementExtensionEventArgs args)
    {
        objectPlaced?.Invoke(args);
    }
}
```

① 事件代码所在脚本需要引用 AR Placement Interactable 组件, AR Placement Interactable 组件 Object Placed 事件也需要引用事件代码脚本。

```

// 重写操作判断可否操作方法
protected override bool CanStartManipulationForGesture(TapGesture gesture)
{
    return gesture.targetObject == null;
}

// 重写操作结束方法
protected override void OnEndManipulation(TapGesture gesture)
{
    base.OnEndManipulation(gesture);

    if (gesture.isCanceled || arSessionOrigin == null || mPlacedObjectNum >=
MaxPlaceObjectNum || gesture.targetObject != null )
        return;

    // 进行射线检测, 确定放置位置
    if (GestureTransformationUtility.Raycast(gesture.startPosition, mHits,
arSessionOrigin, TrackableType.PlaneWithinPolygon, fallbackLayerMask))
    {
        var hit = mHits[0];
        var camera = arSessionOrigin != null ? arSessionOrigin.camera :
Camera.main;
        if (camera == null)
            return;
        // 进行平面正反面检查, 排除击中背面的平面
        if (Vector3.Dot(camera.transform.position - hit.pose.position, hit.
pose.rotation * Vector3.up) < 0)
            return;
        mPlacedObjectNum++;
        var placementObject = Instantiate(placementPrefab, hit.pose.
position, hit.pose.rotation);
        var anchor = new GameObject("PlacementAnchor").transform;
        anchor.position = hit.pose.position;
        anchor.rotation = hit.pose.rotation;
        placementObject.transform.parent = anchor;
        var trackablesParent = arSessionOrigin != null ? arSessionOrigin.
trackablesParent : null;
        if (trackablesParent != null)
            anchor.parent = trackablesParent;

        mEventArgs.placementExtension = this;
        mEventArgs.placementObject = placementObject;
        OnObjectPlaced(mEventArgs);
    }
}
}

```

事实上, ARPlacementInteractableExtension 脚本实现了 ARPlacementInteractable 组件功能, 可以取代后者, 通过添加 MaxPlaceObjectNum 属性, 就能够更灵活地控制生成虚拟对象的数量, 因为继承自 ARBaseGestureInteractable 类, 其他属性事件也都可以正常使用。本示例也演示了当原组件功能不能满足要求或者需要添加额外功能时, 可以通过直接继承 ARBaseGestureInteractable 类进行扩展。

10.3.2 对象选择

选择对象是进行交互操作的前提, 选择对象一方面是让用户知道将要操作的对象, 另一方面也是让系统知道操作的目标, 选择是针对对象本身的, 因此, 需要将操作组件挂载于被操作对象上。使用 XRIT 时, 直接将 AR Selection Interactable 组件挂载于对象上即可 (被操作对象必须挂载碰撞器)。

AR Selection Interactable 组件属性事件和 AR Placement Interactable 组件属性事件基本一致, 因为它们都继承自 ARBaseGestureInteractable 类, 但 AR Selection Interactable 组件包含一个 Selection Visualization (选择视觉) 属性, 这个属性用于设置操作对象被选择时的视觉外观表现, 如呈现一个对象包围盒、在操作对象下方呈现一个圆形指示图标等。Selection Visualization 属性是一个 GameObject 类型, 接受一个游戏对象, 因此可以做出任何想要的视觉效果, 下面以显示对象包围盒为例进行演示。

在 Unity 场景中创建一个 3D 球体, 命名为 sphere, 在 sphere 对象下创建一个立方体子对象 (cube), 命名为 BoundingBox, 并根据 sphere 对象调整立方体的位置和大小, 使它们重合。为渲染线框, 新建一个 Shader 文件, 编写代码如下:

```
// 第10章/10-4
shader "DavidWang/Wireframe" {
    properties{
        _Color("Color",Color) = (1.0,1.0,1.0,1.0)
        _Width("Width",Range(0,1)) = 0.2
    }

    SubShader
    {
        Tags {"Queue" = "Transparent""IgnoreProjector" = "True"}
        Cull Back
        Pass
        {
            HLSLPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/
Core.hlsl"
```



```

struct a2v {
    half4 uv : TEXCOORD0;
    half4 vertex : POSITION;
};

struct v2f {
    half4 pos : SV_POSITION;
    half4 uv : TEXCOORD0;
};

CBUFFER_START(UnityPerMaterial)
    float4 _Color;
    float _Width;
CBUFFER_END

v2f vert(a2v v)
{
    v2f o;
    o.uv = v.uv;
    o.pos = TransformObjectToHClip(v.vertex.xyz);
    return o;
}

float4 frag(v2f i) : COLOR
{
    float LowX = step(_Width, i.uv.x);
    float LowY = step(_Width, i.uv.y);
    float HighX = step(i.uv.x, 1.0 - _Width);
    float HighY = step(i.uv.y, 1.0 - _Width);
    float num = LowX * LowY * HighX * HighY;
    clip((1 - num) - 0.1f);
    return _Color;
}

ENDHLSL
}
}
}

```

利用这个 Shader 创建一个材质，并应用到 BoundingBox 对象上，最后将 sphere 对象保存为预制体，再打开并编辑该预制体，为其挂载 AR Selection Interactable 组件^①，并将 BoundingBox 对象赋给其 Selection Visualization 属性。

将制作好的预制体赋给 AR 放置组件中的 Placement Prefab 属性，运行后的效果如图 10-4 所示。

① 不要在场景中为 sphere 对象挂载选择组件，因为该组件挂载时会自动搜索并引用场景中已有的 XR Interaction Manager 组件和 AR Session Origin 组件，预制体引用场景中其他组件会导致运行时出错。

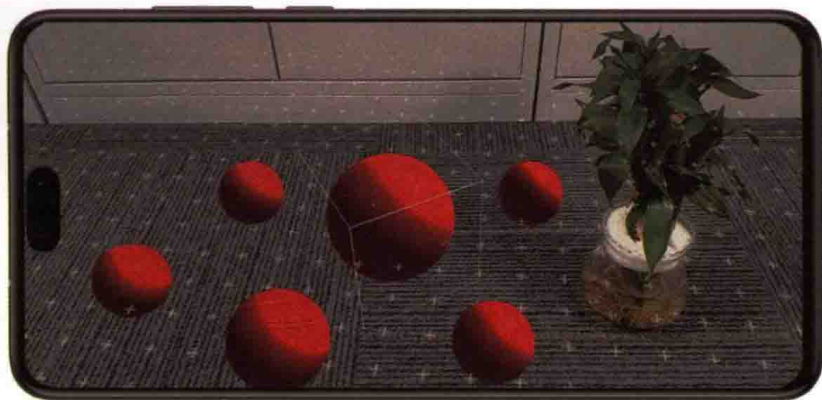


图 10-4 使用包围盒线框渲染展示对象选择状态

挂载 AR Selection Interactable 组件的对象, 在放置后该对象即处于选中状态, 单击可切换选中状态。

10.3.3 对象操作

选中对象后, 就可以进行对象操作了, XRIT 支持单选和多选两种模式, 通过设置 AR Selection Interactable 组件的 Select Mode 属性可以切换选择模式。

1. 对象平移

XRIT 工具包平移使用 AR Translation Interactable 组件, 该组件也继承自 ARBaseGestureInteractable 类 (缩放、旋转组件也继承自该类), 因此, 它们都有基本相同的属性和事件。

平移组件 Object Gesture Translation Mode (平移模式) 属性用于设置平移模式: Horizontal 值用于设置只能在水平方向平移、Vertical 值用于设置只能在垂直方向平移、Any 值用于设置允许水平和垂直平移。Max Translation Distance 属性用于设置最大平移距离, 防止用户将虚拟对象移动得过远。

平移对象时, 需要手指按压住对象进行拖曳。

2. 对象缩放

XRIT 工具包缩放使用 AR Scale Interactable 组件, 该组件 Min Scale 属性用于设置最小缩放值; Max Scale 属性用于设置最大缩放值; Elastic Ratio Limit 属性用于设置缩放对象时使用的弹性比率; Sensitivity 属性用于设置对缩放的敏感度, 防止缩放震荡, 降低缩放时的性能消耗; Elasticity 属性用于设置回弹系数, 当放大时大于最大缩放值后和缩小时小于最小缩放值后, 对象会回弹, 可以实现更好的视觉效果。

缩放时使用双指捏合手势进行操作, 与普通应用缩放图片手势一致。当选择对象后, 可以在屏幕任何位置使用双指捏合手势进行缩放操作。

3. 对象旋转

XRIT 工具包旋转使用 AR Rotation Interactable 组件，该组件 Rotation Rate Degrees Drag 属性用于设置使用拖曳手势时的旋转速率；Rotation Rate Degrees Twist 属性用于设置使用 Twist Gesture（扭转手势）时的旋转速率。

从组件属性可以看出，旋转操作可使用两种手势，一种是单指拖曳手势，另一种是双指扭转手势，当选择对象后，可以在屏幕任何位置使用单指拖曳手势进行旋转操作，也可以在屏幕任何位置使用双指扭转手势进行旋转操作，扭转手势需要双指向同一方向做顺时针或者逆时针滑动。

第 11 章



三维文字与音视频

人类对世界的感知主要来源于视觉信息，长期以来形成的先验知识使我们能在毫秒级的时间内形成周围环境的三维地图，除此之外，听觉也起着非常重要的作用，在真实世界中，不仅利用视觉信息，也利用听觉信息来定位跟踪三维物体。为了达到更好的沉浸式体验效果，在 AR 应用中也应当真实还原现实世界的三维音效，即 AR 应用不仅包括虚拟物体的位置定位，还应该包括声音的三维定位。本章主要学习在 AR 应用中使用三维文字、音频和播放视频的相关知识。

11.1 三维文字

文字是虚拟场景中非常重要的组成部分，常用于对象描述、说明、操作指示，在 AR 应用中，可以使用 3 种方案渲染文字，即 UI Text、3D Text Mesh、Text Mesh Pro。UI Text 借助于 Unity Canvas（画布）渲染文字；3D Text Mesh 则使用独立网格渲染文字；Text Mesh Pro 使用有向距离场（Signed Distance Field, SDF）技术提高文字渲染质量，可同时处理 UI 和三维文字渲染工作。从本质上讲，在 AR 应用中渲染文字需要使用文字纹理，文字纹理图与其他所有纹理图一样，也会随距离变化出现采样过滤而导致纹理模糊的问题，对中文汉字而言，文字纹理图文件大小与其容纳的文字数量相关，文字数量越多纹理体积越大。Text Mesh Pro 使用的有向距离场技术类似于图形中的向量图，可以在放大或缩小时保证不模糊，因此渲染的文字质量更高。

11.1.1 文字单位换算

在 Unity 坐标系中，1 单位为 1m，导入的对象默认为 100% 缩放值，因此，Unity 中的 1 单位在 AR 设备中也是 1m。3D Text Mesh 默认字体大小为 13，对应 Unity 中的 1 单位（高为 1m）；UI Text 默认字体大小为 14，但由于使用 Unity Canvas 的缘故，其相当于 Unity 中的 10 单位（高为 10m）；Text Mesh Pro 渲染三维文字时，默认大小为 36，其相当于 Unity 中的 2.5 单位（高为 2.5m），渲染 UI 文字时，默认大小也是 36，但相当于 Unity 中的 25 单位（高为 25m）。

25m)。

基于以上比例的文字尺寸与 Unity 坐标单位的换算关系很不直观, 因此, 人们使用点 (Point) 的概念来描述三维空间中的文字大小, 将三维空间中的 1m 定义为 2835 点, 从而就很容易将文字字体大小与 Unity 中的缩放值 (Scale) 关联起来。如 3D Text Mesh 中默认字体大小为 13, 因此, $13/2835=0.005$ 则描述了普通文字展示与三维文字展示呈现相同效果时的缩放关系, 由此可以总结 UI Text、3D Text Mesh 和 Text Mesh Pro 中各文字对象的缩放值, 如表 11-1 所示。

表 11-1 3 种文字渲染方式的缩放值

渲染方式		默认文字大小	Scale 值
3D Text Mesh		13	0.005
UI Text		14	0.0005
Text Mesh Pro	渲染三维文字	36	0.005
	渲染 UI 文字	36	0.0005

通过以上比例, 就能获得与常规文字 1:1 的显示效果, 也方便我们操作场景中的文字对象, 获得连续一致的设计和使用体验, 但由于文字显示大小受字体类型及导入方式影响, 相同大小的文字最终呈现的效果也会略有差异。

由于 Text Mesh Pro 能提供更好的文字渲染质量, 而且相同情况下所需要的字体纹理更小, 建议使用这种方式渲染 UI 与三维文字。事实上, 当创建 Unity 工程时, 默认会引入 Text Mesh Pro 插件。

11.1.2 中文字体制作

Unity 默认不带中文字体, 因此无法显示中文字符, 在需要显示汉字时, 可以从互联网上下载 Unity 使用的中文字体, 也可以自己创建 Unity 中文字体。如前所述, 在 Unity 中渲染文字实际是使用文字纹理贴图, 创建 Unity 中文字体即是创建一张包含汉字的纹理贴图, 但由于汉字非常之多, 如果都渲染到一张贴图中会导致这张贴图非常大 (运行时占用大量内存), 因此, 通常会根据需要创建一张包含常见汉字的贴图以节约资源。

创建 Unity 中文字体需要使用字符文件和常规字体文件, 字符文件是纹理贴图所需要包含汉字的文本文件, 常规字体文件 (Windows 系统所有字体文件存放在系统安装盘 Fonts 目录下, 如 C:\Windows\Fonts) 用于渲染汉字字符。下面以使用微软雅黑字体为例, 创建一个包含 7000 个汉字的 Unity 中文字体。

Text Mesh Pro 自带 Unity 字体创建器 (Text Mesh Pro 插件可以使用 Unity 包管理器安装, 从 Unity 菜单中依次选择 Window → Package Manager 打开包管理器进行安装), 在安装 Text Mesh Pro 插件后, 可以通过 Unity 菜单栏 Window → TextMeshPro → Font Asset Creator 打开字体创建面板, 如图 11-1 所示。

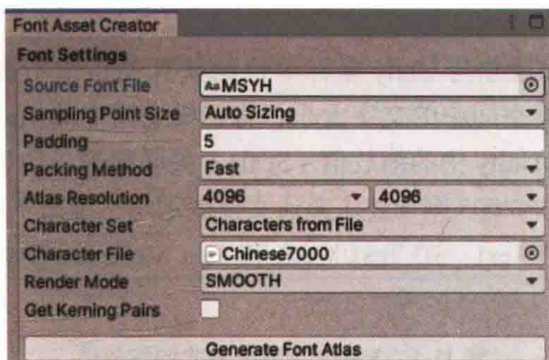


图 11-1 字体创建面板界面

创建面板中 Source Font File 属性为使用的常规字体、Atlas Resolution 属性为生成字体纹理贴图的分辨率、Character File 属性为需要生成的汉字字符文件，设置完成后单击 Generate Font Atlas 按钮生成字体文件，然后保存生成的字体文件即可在 Unity 中使用。

11.2 三维声频

在前面章节中，学习了如何定位追踪用户（实际上是移动设备）的位置与方向，然后将虚拟场景叠加到真实环境中营造虚实融合的效果，在用户移动时，通过 VIO 和 IMU 更新用户的位置与方向信息、更新投影矩阵，就可以把虚拟物体固定在空间的某点上，从而达到以假乱真的视觉体验。

三维空间音效处理的目的是还原声音在真实世界中的传播特性，进一步提高 AR 应用虚实融合的真实度。事实上，三维音效在电影、电视、电子游戏中被广泛应用，但在 AR 中三维空间声音的处理有其特别之处，类似于电影中采用的技术并不能很好地解决 AR 中三维音效的问题。

在电影院中，观众的位置是固定的，因此可以通过在影院的四周加装上音响设备，通过设计不同位置音响设备上声音的大小和延迟，就能给观众营造逼真的三维声音效果。经过大量的研究与努力，人们根据人耳的结构与声音的传播特性也设计出了很多新技术，可以只用两个音响或者耳机就能模拟出三维音效，这种技术叫双耳声（Binaural Sound），它的技术原理如图 11-2 所示。

在图 11-2 中，从声源发出来的声音会直接传播到左耳和右耳，但因为左耳离声源近，所以声音会先到达左耳再到达右耳，由于在传播过程中的衰减，左耳听到的声音要比右耳大，这是直接的声音信号，大脑会接收到两只耳朵传过来的信号。同时，从声源发出的声音也会被周围的物体反射，这些反射与直接信号相比有一定的延迟并且音量更小，这些是间接的声音信号。大脑会采集到直接信号与所有的间接信号并比较从左耳与右耳采集的信号，经过分析计算达到定位声源的效果。在了解大脑的工作模式后，就可以通过算法控制两个音响或者耳机的音量与延迟来模拟三维声源的效果，让大脑产生出虚拟的三维声音场景。

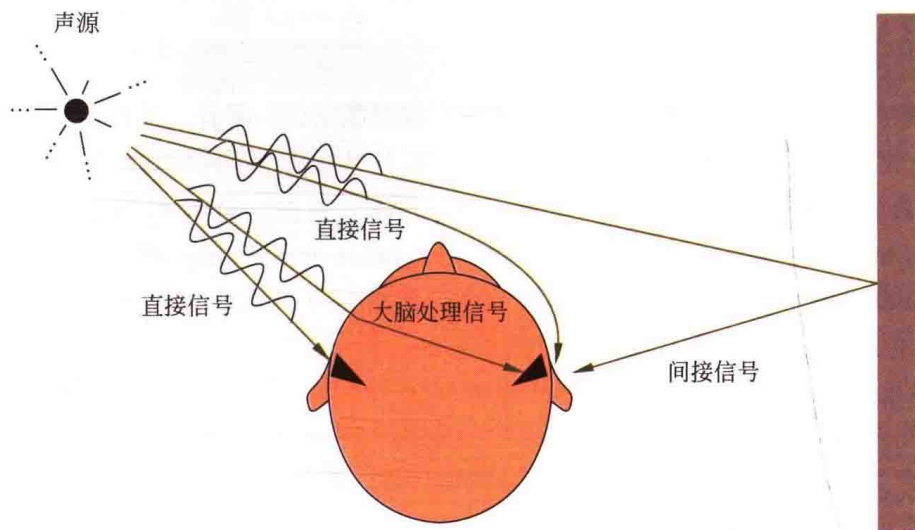


图 11-2 大脑通过双耳对来自声源的直接信号与间接信号进行分析，可以计算出声源位置

11.2.1 三维声场原理

三维声场，也称为三维声频、虚拟三维声频、双耳声频等，它是根据人耳对声音信号的感知特性，使用信号处理的方法对到达两耳的声音信号进行模拟、重建的复杂空间声场。通俗地说就是通过数学的方法对声音信号到达耳朵的音量大小与延迟进行数字建模，模拟真实声音在空间中的传播特性，让人产生听到三维声频的感觉。

不仅用户周边环境对声音传播有影响，当人耳在接收到声源发出的声音时，人的耳廓、耳道、头盖骨、肩部等对声波的折射、绕射和衍射及鼓膜接收的信息都会被大脑捕获，大脑通过计算、分析、经验来对声音的方位、距离进行判断。与大脑工作原理类似，在计算机中通过信号处理的数学方法，构建头部相关传输函数（Head Related Transfer Functions, HRTF），通过多组滤波器来模拟人耳接收到声源的“位置信息”，其原理如图 11-3 所示。

目前三维声场重建技术已经比较成熟，人们不仅知道如何录制三维声频，还知道如何播放这些三维声频，让大脑产生逼真的三维声场信息，实现与真实环境相同的声场效果，然而，目前大多数三维声场重建技术假设用户处于静止状态（或者说与用户位置无关），而在 AR 应用中，情况却有很大不同，AR 应用的用户是随时移动的，这意味着用户听到的三维声音也需要调整，这一特殊情况导致目前的三维声场重建技术在 AR 应用时失效。

为解决这个问题，在安卓端，谷歌公司为 AR 和 VR 开发了一个三维声频 API，称为共振声频（Resonance Audio, RA），下面探讨在 AR 应用中引入和使用 RA 声频的相关内容。

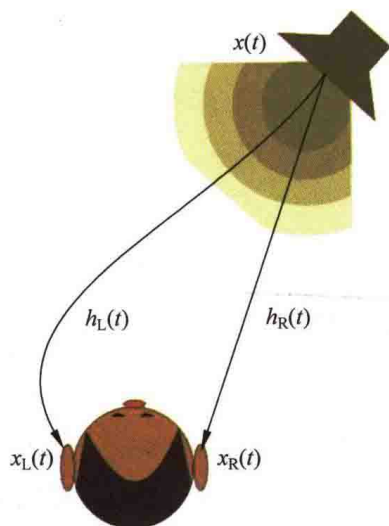


图 11-3 通过信号处理的数学方法可以模拟三维音效

11.2.2 RA 音频使用

RA 音频技术其实是一整套关于 AR/VR 中三维声源定位、混音、处理、多音轨合成的高性能音频处理技术，其更详细的技术细节超出了本书的范围，不再详述。在 AR 应用中应用 RA 音频，首先需要下载 Resonance Audio Unity SDK。目前，SDK 版本是 V1.2.1，其官方下载网址为 <https://github.com/resonance-audio/resonance-audio-unity-sdk/releases>，只需下载 ResonanceAudioForUnity_1.2.1.unitypackage 文件，如图 11-4 所示。

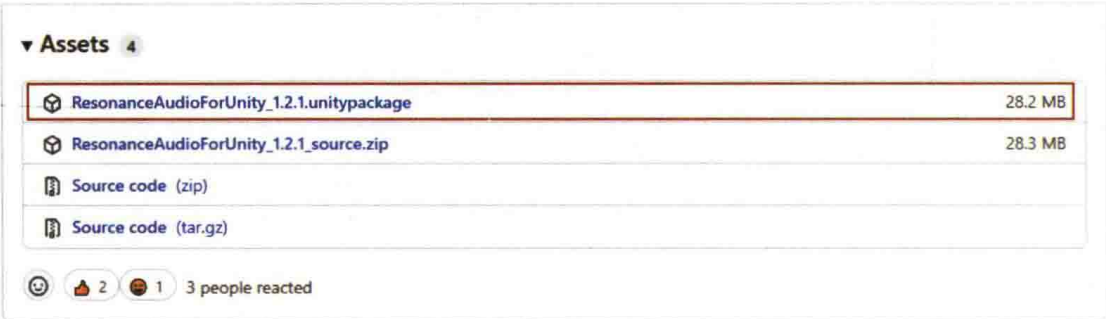


图 11-4 下载共振音频 Unity SDK

下载完成后，导入下载的 Resonance Audio SDK，导入后应该如图 11-5 所示。

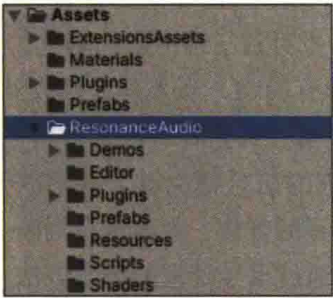


图 11-5 导入共振音频 Unity SDK

11.2.3 RA 组件

在工程窗口（Project 窗口）中打开导入的 ResonanceAudio → Prefabs 文件夹，可以看到 4 个预制件，这 4 个预制件提供了音效处理最基本和常用的对象，详细描述见表 11-2。

表 11-2 共振音频提供的常用预制件

组 件 名	描 述
ResonanceAudioSource	通过引入额外的可选参数，如方向性模式、遮挡和采样质量，增强可控的音频源特性。在同一个对象上需要挂载 Unity 内置组件 AudioSource
ResonanceAudioSoundfield	通过在用户周围的虚拟声场球体上编码声波来呈现完整的 360° 空间音频

续表

组 件 名	描 述
ResonanceAudioRoom	通过引入动态的早期反射和后期混响来模拟特定空间的房间效果。利用 Unity 对象的 Transform 组件应用房间效果。当用户设备(或者音频监听器)在房间模型的指定范围内时,启用相应的房间效果
ResonanceAudioReverbProbe	提供了更精细的空间建模和更细微的混响效果的高级选项

现实生活中,声源在不同的环境中进入人脑后会呈现完全不同的听觉感受,例如,同一首曲子在空旷的原野上、在地下室里、在大演播厅里、在浴室里、在小房间里会给人带来完全不同的感觉。RA 音频使用一个叫作 ResonanceAudioRoom 的对象来模拟不同的环境。

可以把 ResonanceAudioRoom 对象想象成一个长方体房间,用来模拟声源周围的环境。这个房间共有 6 面墙,上下左右前后各一面,每面墙都可以设置其音效材质,RA 音频共提供了 23 种不同的材质,如图 11-6 所示,通过这 23 种材质的组合,可以非常灵活地定义任何想要的房间效果。ResonanceAudioRoom 对象不仅可以用来描述一个房间,其也可以用来描述一个室外空间或者半开放空间,在使用 transparent 选项时就意味着声音会沿着那个方向穿过虚拟墙,如果 6 个方向全部设置为 transparent 值,就是一个完全空旷的空间,这个组合对我们来讲也很重要。

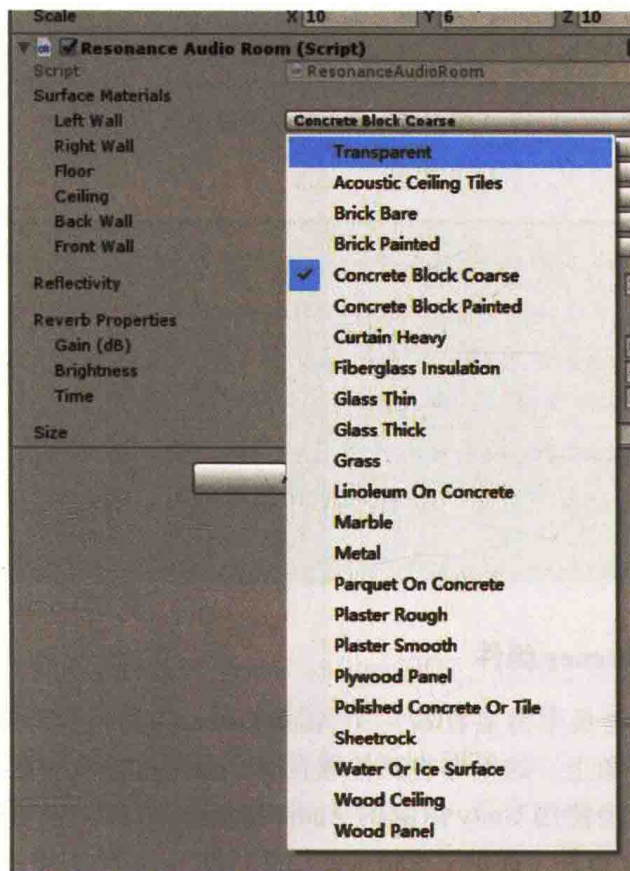


图 11-6 Resonance Audio Room 组件提供了大量的环境模拟选项

11.2.4 使用 RA 音频

通过前面的学习,现在已经了解了足够多的知识来使用 RA 音频,三维空间音效算法会消耗很多计算资源,也非常复杂,但 RA 音频提供了简洁的使用界面,使用非常简单。

1. 配置工程

在使用 RA 音频之前,需要对 Unity 进行配置,让 Unity 使用 RA 音频引擎而不是自带声音引擎来处理音频。在 Unity 菜单中依次选择 Edit → Project Settings → Audio, 打开音频管理器设置面板。将 Spatializer Plugin 属性设置为 Resonance Audio 值、将 Ambisonic Decoder Plugin 属性也设置为 Resonance Audio 值,如图 11-7 所示。

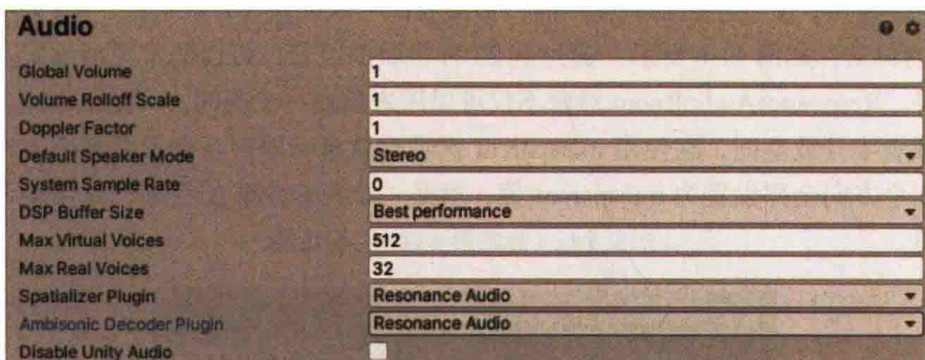


图 11-7 配置应用音频引擎

提示

Ambisonics 是一种录制和播放声音的技术,是专门用来模拟原始三维声场效果的声音系统,它通过由四个指向不同方向的心形话筒组成“四面体阵列”实现三维全覆盖的 360°沉浸式全景环绕声音录制,与普通环绕声不同,播放效果更类似于 Dolby Atmos, Ambisonics 除了水平环绕声音,还包括水平与垂直方向的声源。

RA 音频采用 Ambisonics 技术处理声波与环境的互动,在此项技术中 Ambisonic order 是影响模拟声波的重要因素,当 Ambisonic order 增加时,模拟的结果就会越清晰准确,反之则模糊。

2. 添加 AudioListener 组件

为了接收音频,场景中需要挂载一个 AudioListener 组件,在 AR 应用中,通常把它挂载在场景 Camera 对象上,这里亦将其挂载在 AR Camera 游戏对象上。对这个组件没有特殊的要求,可以直接使用 Unity 内置的 Audio Listener 组件,也可以使用 RA 音频中的 ResonanceAudioListener 组件(提供了更精细的控制特性),如图 11-8 所示。这里直接使用内置的 Audio Listener 组件。



图 11-8 可以直接添加 ResonanceAudioListener 组件，也可以使用 Unity 内置的 AudioListener 组件

3. 添加 ResonanceAudioRoom 预制体

本节演示一只狐狸发出声音，体验三维空间音效。在层级窗口（Hierarchy 窗口）中新建一个空对象，命名为 Fox，将狐狸模型放置到该空对象下，调整好位置与旋转关系，然后将工程窗口中 ResonanceAudio → Prefabs 文件夹下的 ResonanceAudioRoom 预制体拖到层级窗口中的 Fox 对象下，并对 ResonanceAudioRoom 组件房间音效材质进行调整，如图 11-9 所示。



图 11-9 在 Fox 对象下创建 ResonanceAudioRoom 对象并调整其组件参数

4. 添加 AudioSource 组件

还需要在 Fox 游戏对象上挂载一个 AudioSource 组件，这样才能将音频绑定到 Fox 对象上模拟声音从 Fox 对象发出的效果。选中层级窗口中的 Fox 对象，可以直接使用 Unity 内置的 AudioSource 组件，也可以使用 RA 音频中的 ResonanceAudioSource 组件（能提供更精细的控制特性），这里直接使用内置的 AudioSource 组件，并为 AudioClip 选择一个音频，如图 11-10 所示。



图 11-10 可以直接添加 ResonanceAudioSource 组件, 也可以使用 Unity 内置的 AudioSource 组件

除此之外, 还需要对其他属性进行简单设置, 保持选中层级窗口中的 Fox 对象, 在属性窗口 (Inspector 窗口) 中进行的设置如下:

- (1) 将工程窗口中 ResonanceAudio → Resources 文件夹下 ResonanceAudioMixer 对象中的 Master 混音器赋给 AudioSource 组件的 Output 属性。
- (2) 勾选 Spatialize 复选框, 勾选 Spatialize Post Effect 复选框。
- (3) 将 Spatial Blend 属性滑动条滑到三维状态 (设置为 1)。
- (4) 如果需要, 则可以勾选 Play On Awake 和 Loop 复选框, 以便在场景加载后循环播放音频, 如图 11-11 所示。

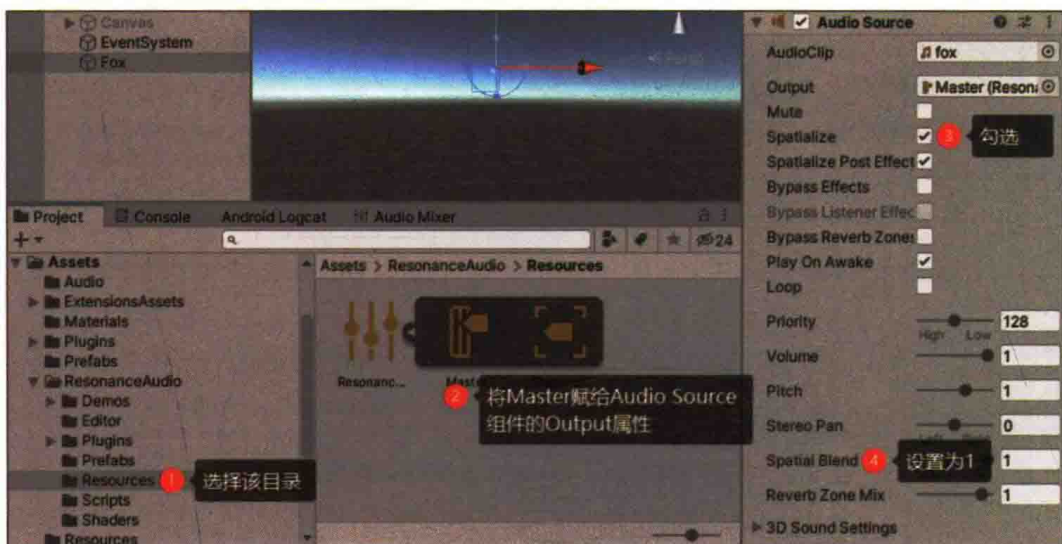


图 11-11 设置 AudioSource 的其他参数

完成后将 Fox 对象拖动到工程窗口中 Prefabs 文件夹下并保存成新的 Fox 预制体, 最后删除层级窗口中的 Fox 对象。

编译运行 AR 应用, 使用耳机 (注意耳机上的左右耳塞勿戴反, 一般会标有 L 和 R 字样) 或者双通道音响体验三维音效, 在检测到的平面放置狐狸对象后^①, 移动手机或者旋转手机朝

① 可以使用射线检测的方法, 在检测到平面的地方实例化 Fox 预制体即可, 这个过程读者应该很熟悉, 不再赘述。

向, 体验在 AR 中三维空间音效效果。

提示

在设置 Audio Source 源组件属性参数时, 应当先勾选 Spatialize 复选框, 再拖动 Spatial Blend 滑块到 3D 值(1), 如果在未勾选 Spatialize 复选框时先拖动 Spatial Blend 滑块到 3D 值, 则 Unity 会使用默认的内置空间音效器 (Panning Spatializer) 而不使用 RA 音频引擎。

经过多次迭代, Unity 内置 AudioSource 组件已经从简单的声音播放组件, 发展成一个功能强大、支持二维和三维空间声音播放的音频播放组件, 除支持混音, 还支持三维空间音效的最大及最小距离、声音衰减等功能, 其主要属性如表 11-3 所示。

表 11-3 音频源组件主要属性

属 性 名	描 述
音频剪辑 (AudioClip)	指定播放的音频文件
音频输出 (Output)	设置音频输出目标, 为空时输出到音频监听器 (AudioListener), 也可以设置到具体的音频混音器 (AudioMixer)
静音 (Mute)	使用静音不会卸载音频数据, 只是停止音频输出, 取消静音可以恢复当前音频播放
空间音效开关 (Spatialize)	设置是否使用空间音效
空间音效后处理效果 (Spatialize Post Effects)	设置空间音效是在滤波器使用前还是使用后起作用
音源滤波开关 (Bypass Effects)	作用在当前音频源的音频滤波器开关
监听器滤波开关 (Bypass Listener Effects)	作用在当前监听器的音频滤波器开关
回音混响开关 (Bypass Reverb Zones)	执行回音混响效果开关
加载即播放开关 (Play on Awake)	对象加载后立即播放音频
循环播放开关 (Loop)	是否循环播放音频
播放优先级 (Priority)	本音频源在整个场景中所有音频源中的播放优先级
音量 (Volume)	音量设置
音调 (Pitch)	播放音频时速度的变化量, 默认值为 1, 表示以正常速度播放。当 <1 时, 慢速播放; 当 >1 时, 快速播放, 速度越快, 音调越高
声道占比 (Stereo Pan)	左右声道占比, 默认为 0, 表示左右均值, 当 <1 时, 左声道占比高; 当 >1 时, 右声道占比高
空间混合 (Spatial Blend)	指定音频源是二维音源 (0)、三维音源 (1)、二者插值的复合音源
混响区混音 (Reverb ZoneMix)	音频源混合到混响区中的音量, 0 ~ 1 的范围为线性区 (与音量属性类似), 1 ~ 1.1 是增强范围区, 最高将混响信号增强 10dB。在模拟从近场到远处声音的过渡时, 与基于距离的衰减曲线结合使用非常有用

在使用三维空间音效时,AudioSource 组件还专门设置了 3D Sound Settings(三维声音设置) 卷展栏,在该卷展栏下,Volume Rolloff(音量衰减)用于设置声音的衰减模式;Min Distance(最小距离)为最小声音变化距离,在该距离内,声音将保持最大设定值,不再发生变化;Max Distance(最大距离)为最大声音变化距离,在该距离之外,声音将衰减到最小设定值,不再变化,其他三维音效参数的意义如表 11-4 所示。

表 11-4 AudioSource 组件三维音效主要属性

属 性 名	描 述
多普勒级别 (Doppler Level)	决定多少多普勒效应将被应用到这个音频信号源(如果设置为 0,则无效果)
扩散 (Spread)	设置三维立体声或多声道声源在扬声器空间的传播角度
音量衰减模式 (Volume Rolloff)	有 3 种衰减模式:Logarithmic Rolloff(对数衰减),接近声频源时声音响亮,远离声频源时声音成对数下降;Linear Rolloff(线性衰减),声频源音量成线性衰减;Custom Rolloff(自定义衰减),根据设置的衰减曲线调整声频源音量衰减行为
最小距离 (Min Distance)	最小距离,在最小距离内声音会保持最响亮
最在距离 (Max Distance)	最大距离,声音停止衰减距离,超过这一距离,音量保持为衰减曲线末音量,不再衰减

11.2.5 音频效果设置

11.2.4 节已经利用 RA 音频实现了三维音效,这只是 RA 音频最简单的应用,RA 音频提供了远比例丰富的功能技术特性,特别是模拟房间的参数,可以模拟很多生活中的真实环境特性,本节主要对房间(Room)参数、API 进行更深入且详细的探究。

如图 11-12 所示,ResonanceAudioRoom 组件属性面板中主要包括 Surface Materials、Reflectivity、Reverb Properties、Size 四大主要功能属性,合理利用这 4 个属性,可以模拟不同环境下的声源音效表现,营造出非常逼真的虚拟声场效果。

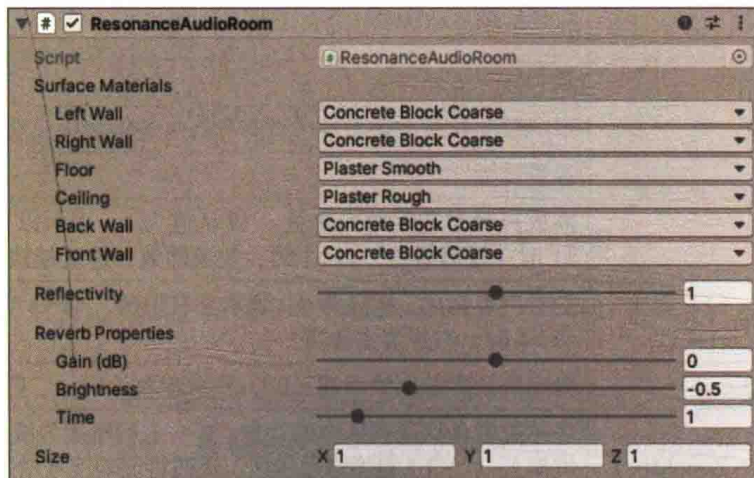


图 11-12 ResonanceAudioRoom 属性面板

该面板的详细信息如下。

1. Surface Materials (表面材质)

利用该属性可以对房间 6 个声学表面赋予不同的声学材质, RA 音频共提供了 23 种不同的材质, 可以直接选择需要的声学材质。每种声学材质定义了在不同的频率下声波吸收或反射率等声学特性, 例如, Heavy Curtain (厚窗帘) 吸收高频部分, 因此可以模拟一个更干燥、温暖房间的声音特性; Polished Concrete (抛光混凝土) 在所有频率上都会反射更多的声音能量, 从而产生更明亮和更有回声的房间特性。

2. Reflectivity (反射)

这个参数用来对 ResonanceAudioRoom 组件中声源的早期反射强度进行调节, 这样用户就能对他们周围房间的大小和形状有一个大致的印象。例如可以通过减少这个参数的值来模拟受限小空间的声音, 增加这个值用来模拟空旷空间的声音。

3. Reverb Properties (混响属性)

这些参数影响 ResonanceAudioRoom 组件的后期混响, 它主要包括以下 3 个参数。

1) Reverb Gain

这个参数用来调整房间的混响增益 (Reverb Gain), 这个混响增益不是直接来自声源的声波。可以用这个参数来调整混合声音的“厚重”或“单薄”。例如, 如果想让房间混响变弱, 可以通过调整 Reverb Gain 使声音听起来更“单薄”。

2) Reverb Brightness

混响亮度 (Reverb Brightness) 用来调节混响中低频或高频的数量。ResonanceAudioRoom 组件可以提供不同频率的不同混响衰减率, 房间的墙面材质对不同的频率的音频吸收和反射率是不一样的, 就像在真实房间中一样。可以使用这个参数来调整房间声音的充盈程度。例如, 降低混响亮度会给人一个房间更饱满的印象, 并模拟一个包含许多物体或人的房间声音效果。

3) Reverb Time

混响时间 (Reverb Time) 用来调节混响的时间长度, 即混响的衰减时间。该值是根据 ResonanceAudioRoom 组件指定的表面材质和房间尺寸计算的混响时间的乘数, 可以使用这个参数来对模拟房间的大小进行声学调整。

4. Room Size (房间尺寸)

使用 X、Y 和 Z 参数设置房间的尺寸 (以米为单位)。房间尺寸会影响房间声音效果, 这个尺寸同时也设置了声音的边界, 当用户进入声场范围内时触发房间声音效果, 或者平稳地从一个 RA 房间切换到另一个中。

11.2.6 RA 音频 API

除可以通过 Unity 属性面板在开发时调整 RA 音频音效, 也可以在应用运行时通过脚本调整音频音效, Resonance Audio SDK 提供了丰富的音效 API, 其主要 API 类如表 11-5 所示。

表 11-5 共振音频 (Resonance Audio) 主要 API 类

类 名	描 述
ResonanceAudio	这是 RA 音频的主要类, 它负责 RA 音频与操作系统及硬件的交互
ResonanceAudioAcousticMesh	这个类用于处理网格, 保存网格过滤器 (Mesh Filter) 或者地形 (terrain) 的网格信息, 以及分配给三角形的表面材质, 利用这些环境网格可以增强音效的真实感
ResonanceAudioListener	对 Unity 内置 AudioListener 的增强, 提供更多高级的三维声场的调节参数
ResonanceAudioMaterialMapper	Resonance Audio Material Mapper 可编程对象, 负责处理 GUID 到表面材质的映射 (这些材质定义了表面的声学特性)
ResonanceAudioMaterialMapperUpdater	Resonance Audio Material Mapper 更新器
ResonanceAudioReverbProbe	混响探针, 作为声音的采样点, 这个采样点会向其周围的环境发射射线, 以此来感知周围环境的声学特性并利用这些信息来计算混响音效
ResonanceAudioRoom	模拟房间, 利用这个模拟房间来模拟真实的音效环境
ResonanceAudioRoomManager	模拟房间的管理类, 管理模拟房间的三维声场
SurfaceMaterialDictionary	一个可序列化的字典类, 它负责从表面材质到 GUID 的映射
ResonanceAudioSource	对 Unity 内置 AudioSource 的增强, 提供更多高级的三维声场的调节参数

此外, RA 音频技术还提供了对声音的混响烘焙 (Reverb Baking), 与对光照进行烘焙一样, 通过设置探针 (Probe), 可以将声音特性直接烘焙到环境中, 省去实时的音频处理, 降低处理器的压力, 这对移动设备有限的资源来讲是非常有利的。

11.2.7 运行时启用和禁用三维音效

不管是使用 Unity 内置的 AudioSource 音频源组件还是 RA 音频 ResonanceAudioSource 音频源组件设置和播放声音, 在 AR 应用运行时, 都可以方便地通过设置音频源组件参数实现音频的二维、立体声、三维空间音效的切换, 以 AudioSource 音频源组件为例, 典型代码如下:

```
// 第 11 章 / 11-1
using UnityEngine;
[RequireComponent(typeof(AudioSource))]
public class ToggleAudio : MonoBehaviour
{
```

```

private AudioSource mSourceObject;
private bool mIsSpatialized = false;

public void Start()
{
    mSourceObject = gameObject.GetComponent<AudioSource>();
    mSourceObject.spatialBlend = 1;
    mSourceObject.spatialize = true;
}

public void ToogleAudioSpatialization()
{
    if (mIsSpatialized)
    {
        mIsSpatialized = false;
        mSourceObject.spatialBlend = 0;
        mSourceObject.spatialize = false;
    }
    else
    {
        mIsSpatialized = true;
        mSourceObject.spatialBlend = 1;
        mSourceObject.spatialize = true;
    }
}
}

```

代码清单 11-1 所示脚本应与 AudioSource 组件挂载于同一个场景对象上, 通过按钮事件切换音频的二维、立体声、三维空间音效, 其中 spatialBlend 属性用于控制音效的二维声与立体声(Stereo)混合, 当 spatialize 属性被设置为 true 值时, 声音音量将根据设置的衰减曲线衰减, 而当 spatialize 属性被设置为 false 值时, 则音量只会在最大值与无音量之间切换; spatialize 属性用于控制音效的三维空间化, 实现三维空间音效。

11.2.8 使用回响营造空间感

在实际生活中, 不仅能听出声音的远近和方向, 还能通过声音了解周边环境, 这就是回响, 如在开阔空间、密闭小屋、大厅等各种不同环境中听同一种声音感受会非常不一样。通过前述章节, 已经能够实现三维空间音效, 能正确地反映出声音的远近和方向, 也能通过设置房间属性反映周边环境。本节将阐述在 Unity 中实现回响的一般方法, 由于影响回响的因素非常多, 在开始之前, 先了解回响的基本原理和术语。

声波在空气中传播时, 除了透过空气进入人耳外, 还会经由墙壁或是环境中的介质反射后进入耳内, 混合成我们听到的声音, 这就产生了一个复杂的反射组合。声波的混合分为以下 3 个部分:

(1) 直达音：从声源直接传入耳朵的声波称为直达音 (Direct Sound)，直达音随着传播距离的增加能量会逐渐减弱，这也是离我们越远的声源声音会越小的原因。

(2) 早期反射音：声波在传播时遇到介质会不断反射，如果仅经过 1 次反射后就进入人耳则称其为早期反射音 (Early Reflections)，早期反射音会帮助我们了解所处空间的大小与形状，如图 11-13 所示。

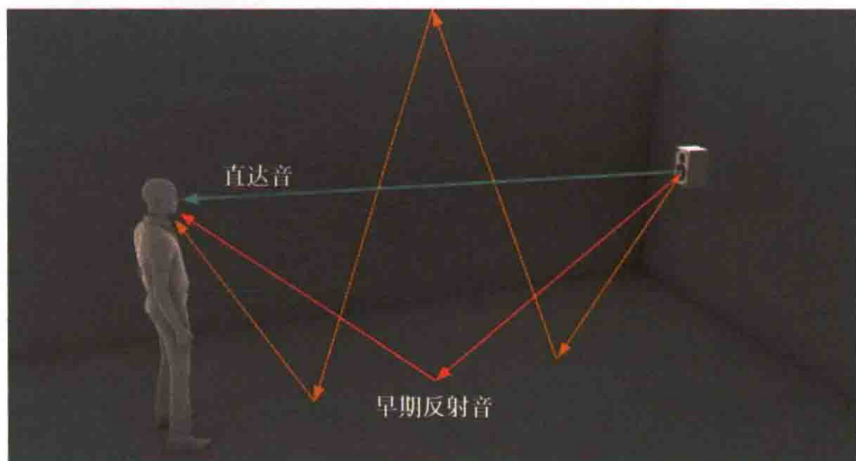


图 11-13 早期反射音有助于我们了解所处空间的大小和形状

后期混响：声波会在环境中不断地反射，反射音随着时间的不断增加形成更多也更密集的回响音，会密集到人耳无法分辨，这种现象称为后期混响 (Late Reverb)。

直达音、早期反射音、后期混响混合成我们最终听到的声音，人脑会根据经验将听到的声音与空间环境关联起来，从而形成空间环境感知。在使用计算机进行模拟时，将输入的纯净音频称为干音，而将经过回响混音后的音频称为湿音，最终这两种音频都会同时输出到混音器中混合成最终播放出来的声音。

使用 RA 音频 ResonanceAudioRoom 组件 Reverb Properties 相关属性，可以简单地调整回响，也可以利用 Unity 内置音频处理功能实现更加复杂的回响模拟，在工程窗口中，双击 ResonanceAudio → Resources 文件夹下 ResonanceAudioMixer 文件将打开 Unity 混音器面板，在该面板中 Group 栏下的 Master 混音器组上右击，在弹出的菜单中选择 Add child group (添加子组) 新添加一个组，命名为 RoomEffectGroup，如图 11-14 所示。

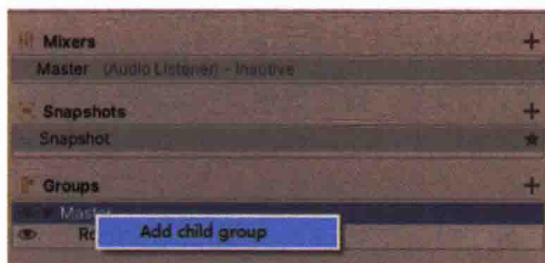


图 11-14 添加新的音频组

选择 RoomEffectGroup 组，在右侧展开的面板中，单击其最下方的 Add 按钮，在弹出的菜单中选择 SFX Reverb 回响特效组件，如图 11-15 所示。

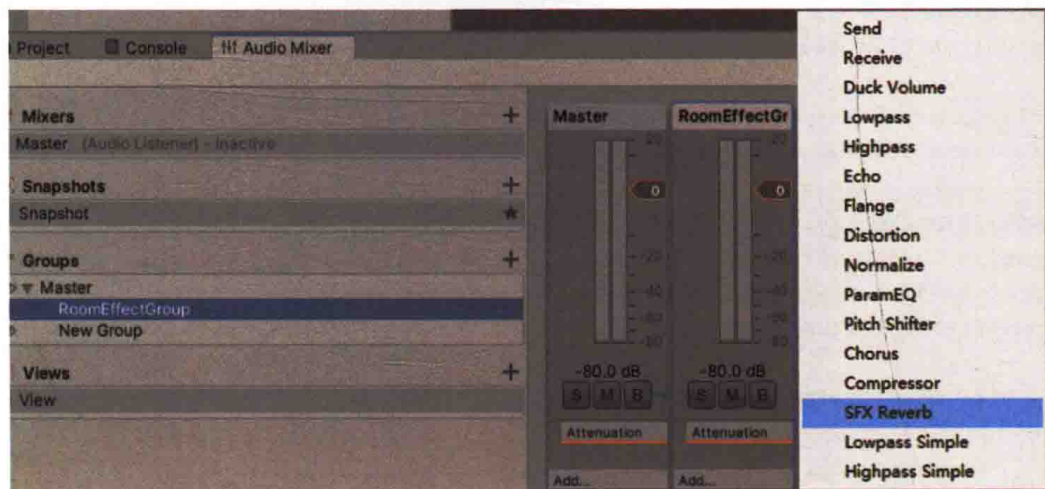


图 11-15 添加 SFX Reverb 音效组件

在属性窗口打开的 SFX Reverb 属性面板中可以调整各干湿声频的参数，其中 Dry Level 参数用于设置输入干声频的音量，因为最后回响混音后的湿音会输出到 Master 混音器，该值可以设置为最小值（-10000mB）；Room 参数用于设置低频声音在房间中的传播效果，该值可以设置为最大值（0.0mB）；Decay Time 参数用于设置声音反射时延，单位为秒，通过该值能感知到空间大小；Reflections 和 Reflect Delay 参数用于设置早期反射音信息；Reverb 和 Reverb Delay 参数用于设置后期混响信息。属性窗口中的每个参数都与真实声音传播的一个特性相关联，在需要时，可以查阅相关声频处理资料了解更多详细信息，如图 11-16 所示的参数设置仅供参考。

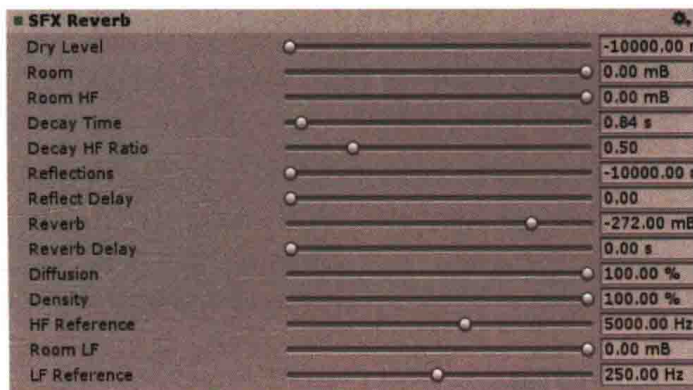


图 11-16 SFX Reverb 参数设置参考值

在完成音效回响效果设置后，可以直接将 RoomEffectGroup 组赋给 AudioSource 组件的 Output 属性，这样就可以体验到带回响的空间混音效果，也可以通过代码动态地使用回响效

果, 典型使用代码如下:

```
// 第 11 章 / 11-2
using UnityEngine;
using UnityEngine.Audio;

[RequireComponent(typeof(AudioSource))]
public class ToogleAudio : MonoBehaviour
{
    public AudioManager RoomEffectGroup;
    public AudioManager MasterGroup;
    private AudioSource mSourceObject;
    private bool mIsSpatialized = false;

    public void Start()
    {
        mSourceObject = gameObject.GetComponent<AudioSource>();
        mSourceObject.spatialBlend = 1;
        mSourceObject.spatialize = true;
    }

    public void SwapSpatialization()
    {
        if (mIsSpatialized)
        {
            mIsSpatialized = false;
            mSourceObject.spatialBlend = 0;
            mSourceObject.spatialize = false;
            mSourceObject.outputAudioMixerGroup = MasterGroup;
        }
        else
        {
            mIsSpatialized = true;
            mSourceObject.spatialBlend = 1;
            mSourceObject.spatialize = true;
            mSourceObject.outputAudioMixerGroup = RoomEffectGroup;
        }
    }
}
```

11.2.9 三维音效设计原则

AR 应用程序可以实现沉浸感很强的虚实融合, 但相比真实物体对象, 虚拟对象的操作缺乏触感这种很重要的操作确认手段, 通过触感我们能感知到身体部位与物体对象的接触程度、物体对象的质地等, 而这些在虚拟对象操作时均无法提供, 因此, 为了更好的使用体验, AR

应用应当通过其他途径弥补操作触觉确认缺失，在所有方法中，音效是一个非常不错的选择，可以通过合理使用音效提示用户各操作阶段发生的事情，让使用者得到听觉上的反馈，强化对操作的确认。除此之外，还可以利用声音通知使用者某些事件的发生，通过视觉与听觉的结合，提高 AR 应用使用者的满意度。

在 AR 应用中使用声音的一般原则如表 11-6 所示。

表 11-6 AR 应用中声音使用的基本原则

类 型	描 述
通知与事件	在应用状态变化（如任务完成、收到信息）时应当通过声音提醒用户，在 AR 应用中，一些虚拟对象有时会超出用户视野范围，合理使用三维音效可以帮助用户确认对象位置；对于伴随用户移动的虚拟对象，如一只虚拟宠物狗，沙沙的三维脚步声将会帮助用户跟踪这些对象的位置；事件发生时可根据事件类型制定不同的声音方案，如事件成功，可使用积极的声音，事件失败则使用比较消极的声音，对于不能区分成功与失败的事件则应当使用中性声音
触发时机	声音使用应符合大众审美，并且保持一致性，每类声音用于表达某种意义，不可混合乱用；将最重要的声音音量调高，同时暂时性地降低其他声音音量；使用声音延时，防止在同一时间触发过多声音
手势交互	以按钮操作为例，按钮事件符合传统 PC 按钮基本机制，在按钮按下时触发短促、低频的声音，在按钮释放时触发积极、高频的声音以传达不同操作阶段
语音命令交互	如果在 AR 应用中使用了语音命令，语音命令通常应当提供视觉反馈，但可以通过声音强化语音命令识别或者执行结果，使用中性或者积极的声音表达执行确认，使用积极的声音表示执行成功，使用消极的声音表示语音识别或者执行失败
声音依赖	不要过分依赖声音，用户可能由于隐私、环境或者听力问题而无法感知声音

具有空间特性的对象所发出的声音也应当具有空间效果(三维)，包括 UX 元素、视觉提示、虚拟物体发出的声音。三维化 UX 元素声音可以强化使用者的空间感知，三维音效有助于用户了解视野之外发生事件的空间位置和距离。

由于三维音效需要消耗计算资源，对不具有空间特性的对象使用三维音效则不会对提升用户体验产生任何有利影响，如旁白、简单事件提示音等，同时应注意，在 AR 应用中，应当确保任何时候都不能有多于两个的三维空间音效同时播放，这可能会导致性能问题。

声音随距离的衰减在 AR 应用中应用得非常广，衰减类型和曲线可以调节声音衰减效果，通常而言，线性衰减能适应绝大部分声音衰减需求，也可以通过调整衰减曲线营造不同的衰减效果。信息提示类声音更应关注声音的有效性而非距离，无须使用衰减。

11.3 三维视频

视频播放是一种非常方便高效的展示手段，在 AR 应用中播放视频也是一种常见的需求，如在一个展厅中放置的虚拟电视上播放宣传视频，或者在游戏中为营造氛围而设置的虚拟电视视频播放，本节将学习如何在 AR 场景中播放视频。

11.3.1 VideoPlayer 组件

VideoPlayer 组件是 Unity 提供的一个跨平台视频播放组件, 这个组件在播放视频时会调用运行时系统本地的视频解码器, 即其本身并不负责视频的解码, 因此开发人员需要确保在特定平台上视频编码格式能被支持。VideoPlayer 组件不仅能播放本地视频, 也能够通过 HTTP/HTTPS 协议播放远端服务器视频, 而且支持将视频播放到相机平面 (Camera Plane, 包括近平面和远平面)、作为渲染纹理 (Render Texture)、作为材质纹理、作为其他组件的 Texture 属性纹理, 因此, 可以方便地将视频播放到相机平面、三维物体、UI 界面上, 功能非常强大。VideoPlayer 组件界面如图 11-17 所示。



图 11-17 Video Player 组件界面

VideoPlayer 组件的主要属性及意义如表 11-7 所示。

表 11-7 VideoPlayer 组件的主要属性及意义

属 性	描 述
Source (视频源)	视频源, 其下拉菜单包括两个选项: Video Clip 与 URL。选择 Video Clip 时, 可以将项目内的视频直接赋给 Video Clip 属性。当选择 URL 时, 可以使用视频的路径定位视频 (如使用 http:// 或者 file://), 既可以在编辑状态时为其设置固定的地址信息, 也可以在运行时通过脚本代码动态地修改这个路径达到动态控制视频加载的目的
Play on Awake (加载即播放)	是否在场景加载后就播放视频, 如果不勾选该项, 则应当在运行时通过脚本控制播放视频
Wait for First Frame (等待第一帧)	该选项主要用于同步视频播放与场景进度, 特别是在游戏场景中, 如果勾选, Unity 则会在游戏开始前等待视频源加载显示, 如果取消勾选, 则可能会丢弃前几帧以使视频播放与游戏的其余部分保持同步, 在 AR 中建议不勾选, 不勾选时视频会跳帧播放
Loop (循环)	循环播放
Skip on Drop (跳帧同步)	是否允许跳帧以保持与场景同步, 建议勾选

续表

属 性	描 述
Playback Speed (播放速度)	视频播放速度倍率, 如设置为 2, 视频则以 2 倍速播放, 范围为 [0,10]
Render Mode (渲染模式)	定义视频的渲染模式。其下拉菜单中包括 4 个选项: Camera Far Plane (相机远平面)、Camera Near Plane (相机近平面)、Render Texture (渲染纹理)、Material Override (材质复写)、API only (仅 API)。各渲染模式功能如下。 (1) Camera Far Plane: 将视频渲染到相机的远平面, 其 Alpha 值可设置视频的透明度。 (2) Camera Near Plane: 将视频渲染到相机的近平面, 其 Alpha 值可设置视频的透明度。 (3) Render Texture: 将视频渲染为渲染纹理, 通过设置 Target Texture 定义 Render Texture 组件渲染到其图像的 Render Texture。 (4) Material Override: 将视频渲染到物体材质上。 (5) API only: 仅能通过脚本代码将视频渲染到 VideoPlayer.texture 属性中
Audio Output Mode (音轨输出模式)	定义音轨输出, 其下拉菜单包括 4 个选项: None (无)、Audio Source (音频源)、Direct (直接)、API only (仅 API)。选择 None 时不播放音轨; 选择 Audio Source 时可以自定义音频输出; 选择 Direct 时, 会直接将视频中的音轨输出; 选择 API only 时, 只能使用脚本代码控制音轨的设置
Mute (静音)	是否静音
Volume (音量)	音频音量

除此之外, VideoPlayer 组件还有很多供脚本调用的属性、方法及事件, 常用的主要包括表 11-8 所示方法事件, 其他方法在使用时读者可以查阅相关文档说明。

表 11-8 VideoPlayer 组件方法事件

名 称	类型	描 述
EnableAudioTrack	方法	启用或禁用音轨, 需要在视频播放前设置
Play	方法	播放视频
Pause	方法	暂停视频播放
Stop	方法	停止视频播放, 与暂停视频相比, 停止视频播放后, 再次播放视频会从头开始
isPlaying	属性	视频是否正在播放
isLooping	属性	视频是否循环播放
isPrepared	属性	视频是否已准备好 (如加载是否完成可供播放)
errorReceived	事件	发生错误时回调
frameDropped	事件	丢帧时回调
frameReady	事件	新帧准备好时回调, 这个回调会非常频繁, 需要谨慎使用
loopPointReached	事件	视频播放结束时回调
prepareCompleted	事件	视频资源准备好时回调
started	事件	视频开始播放后回调

注意

当设置 Source 属性时，广域网中使用 http:// 定位视频源在某些情况下可能无法获取视频文件，这时应该使用 https:// 定位。HTTPS 是一种通过计算机网络进行安全通信的传输协议，经由 HTTP 进行通信，利用 SSL/TLS 建立安全信道，加密数据包，在某些平台必须使用该协议才能获取视频源。

VideoPlayer 组件还有很多其他方法、属性、事件，利用这些方法属性可以开发出功能强大的视频播放器。如前所述，VideoPlayer 会调用运行时本地系统中的原理解码器，因此为确保在特定平台视频播放正常，需要提前了解运行系统的解码能力，通常，不同的操作系统对视频的原理解码支持并不相同，表 11-9 为当前主流系统对视频格式的原理解码支持列表。

表 11-9 主要平台原生可以解码的视频格式

平 台	支持的原生视频格式
Windows	.asf、.avi、.dv、.mv4、.mov、.mp4、.mpg、.mpeg、.ogv、.vp8、.webm、.wmv
Mac OS X	.dv、.m4v、.mov、.mp4、.mpg、.mpeg、.ogv、.vp8、.webm、.wmv
Linux	.ogv、.vp8、.webm
Android	.3gp、.mp4、.webm、.ts、.mkv
iPhone	.m4v、.mp4、.mov

表 11-9 所列原生视频格式并不一定全面完整，在具体平台上使用 VideoPlayer 时应当详细了解该平台所支持的原生视频解码格式，防止出现无法播放的情形。

注意

特别需要提醒的是：即使视频后缀格式相同也不能确保一定能解码，因为同一种格式会有若干种编码方式，在具体使用过程中需要查询特定平台的解码格式。

除可以在 Unity 属性面板中进行参数属性设置，也可以使用脚本代码的方式控制视频加载、播放、暂停等操作，典型的操作代码如下：

```
// 第 11 章 / 11-3
using UnityEngine;
using UnityEngine.Video;
[RequireComponent(typeof(VideoPlayer))]
public class VideoController : MonoBehaviour
{
    private VideoPlayer mVideoPlayer;
    void Start()
    {
        mVideoPlayer = gameObject.GetComponent<VideoPlayer>();
    }
}
```



```

public void Play()
{
    mVideoPlayer.Play();
}
public void Pause()
{
    mVideoPlayer.Pause();
}
public void Stop()
{
    mVideoPlayer.Stop();
}
}

```

11.3.2 三维视频播放实现

由于 AR 虚实融合场景的特殊性，使用 UI 及相机平面播放视频模式不太适合 AR 应用，以虚拟物体作为视频播放承载物是最好的选择，视频渲染模式使用 Material Override 非常有利于在模型特定区域播放视频，因此，首先制作一个预制体，在模型的特定区域采用 Material Override 模式播放视频。

新建一个工程，并在层级窗口中新建一个空对象，命名为 TV，将电视机模型文件拖放到该空对象下作为其子对象并调整好大小尺寸，然后新建一个 Quad，用这个 Quad 作为视频播放的载体，将该 Quad 调整到电视机屏幕位置，处理好尺寸与位置关系。因为采用 Material Override 模式播放视频，所以需要新建一个材质，命名为 videos，该材质选用 Universal Render Pipeline → Unlit → Texture 着色器，但不赋任何纹理、不设置任何参数（因为后面将使用视频帧作为纹理渲染到材质上），并将该材质赋给 Quad 对象。最后为 Quad 对象挂载 VideoPlayer 组件，并设置好视频源等属性，如图 11-18 所示。



图 11-18 实现三维视频播放的步骤

将 TV 制作成预制体，同时删除层级窗口场景中的 TV 对象。为将虚拟电视机放置到真实环境中，采取与以往一样的射线检测放置方式，新建一个 C# 脚本，命名为 AppController，编写代码如下：

```
// 第 11 章 / 11-4
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;
using UnityEngine.UI;

[RequireComponent(typeof(ARRaycastManager))]
public class AppController : MonoBehaviour
{
    public GameObject spawnPrefab;           //TV 预制体
    public int MaxTVnumber = 1;              // 可放置的数量
    public Button BtnPlay;                   //UI 按钮
    public Button BtnPause;
    public Button BtnStop;

    static List<ARRaycastHit> Hits;
    private ARRaycastManager mRaycastManager;
    private GameObject spawnedObject = null;
    private ARPlaneManager mARPlaneManager;
    private int mCurrentTVnumber = 0;
    private UnityEngine.Video.VideoPlayer mVideoPlayer ;

    private void Start()
    {
        Hits = new List<ARRaycastHit>();
        mRaycastManager = GetComponent<ARRaycastManager>();
        mARPlaneManager = GetComponent<ARPlaneManager>();
        // 事件监听注册
        BtnPlay.transform.GetComponent<Button>().onClick.AddListener(() =>
        mVideoPlayer.Play());
        BtnPause.transform.GetComponent<Button>().onClick.AddListener(() =>
        mVideoPlayer.Pause());
        BtnStop.transform.GetComponent<Button>().onClick.AddListener(() =>
        mVideoPlayer.Stop());
    }

    void Update()
    {
        if (Input.touchCount == 0)
```

```

        return;
    var touch = Input.GetTouch(0);
    // 射线检测放置对象
    if (mRaycastManager.Raycast(touch.position, Hits, TrackableType.
PlaneWithinPolygon | TrackableType.PlaneWithinBounds) )
    {
        var hitPose = Hits[0].pose;
        if (spawnedObject == null && mCurrentTVnumber < MaxTVnumber)
        {
            spawnedObject = Instantiate(spawnPrefab, hitPose.position,
hitPose.rotation);
            mVideoPlayer = spawnedObject.gameObject.transform.Find("Quad").
gameObject.GetComponent<UnityEngine.Video.VideoPlayer>();
            TrunOffPlaneDetect();
            mCurrentTVnumber++;
        }
        else
        {
            spawnedObject.transform.position = hitPose.position;
        }
    }
}
// 关闭平面检测功能
void TrunOffPlaneDetect()
{
    mARPlaneManager.enabled = false;
    foreach (var plane in mARPlaneManager.trackables)
        plane.gameObject.SetActive(false);
}
}

```

在上述代码中，首先定义了 3 个按钮（分别用于控制视频播放、暂停、停止）及最大可放置虚拟电视机的数量（视频解码也是一项计算密集型任务，过多地虚拟电视播放视频会严重影响应用程序性能），然后使用射线检测的方式放置虚拟电视模型，并在放置模型后停止平面检测和已检测到平面的显示。

将 AppController 脚本挂载到层级窗口中的 AR Session Origin 对象上，然后在场景中新建 3 个 UI 按钮，并设置好脚本的相应属性（将之前制作的 TV 预制体设置到 Spawn Prefab 属性，将 3 个按钮设置到对应的按钮属性）。

编译运行，在检测到平面后单击加载虚拟电视机，这时电视机上会播放设定的视频，可以通过按钮控制视频的播放、暂停与停止，效果如图 11-19 所示。



图 11-19 三维视频播放效果图

11.3.3 视频音效空间化

在直接播放视频时, 如果不勾选禁音 (Mute) 属性, 则音频可以同步播放, 但播放的音频并不具备空间定位能力, 即声音不会呈现空间特性, 在 AR 中实现视频音轨三维空间化功能需要与前文所述的空间音频组件配合使用。

在 11.3.2 节中, 通过挂载和设置 VideoPlayer 组件, 已经实现了 AR 视频播放。为实现三维空间音效, 打开 TV 预制体^①, 选择 Quad 子对象, 为其挂载 AudioSource 组件, 并按照 11.2 节所述正确设置 Output、Spatialize、Spatial Blend 属性, 但其 AudioClip 属性留空不设置音频源。

将 VideoPlayer 组件的 Audio Output Mode 属性设置为 Audio Source 值、勾选 Track 0[2 ch] 复选框, 并将其 Audio Source 属性设置为 Quad 对象上挂载的 AudioSource 组件 (确保 Audio Source 组件与 VideoPlayer 组件挂载于同一个场景对象上), 如图 11-20 所示。



图 11-20 设置视频播放中的音轨从 AudioSource 输出

通过以上设置, 在使用 VideoPlayer 组件播放视频时, 其音轨将从视频文件中剥离并输出到 AudioSource 组件进行混音处理, 因此自然就可以利用 AudioSource 组件的所有音效功能, 如混音、回响、距离衰减等, 从而实现三维空间音效。

^① 在工程窗口中选择 TV 预制体, 然后在属性窗口中单击 Open Prefab 即可以对预制体进行再编辑。

第 12 章



glTF 与 Model Viewer

ARCore 因其稳定的运动跟踪和环境理解能力在推出后就迅速在移动 App 端获得应用,并成为 Android 平台上 AR 开发引擎标杆,但移动端 App 应用需要用户独立安装,限制了 AR 体验的传播便捷性,因此,谷歌公司开发了 Model Viewer 功能模块,通过 Chrome 等浏览器^①,在 Web 端也可以体验到 AR 功能,极大地便利了 AR 普及传播。本章主要学习在 Web 端构建 AR 体验的方法及使用的三维模型格式。

12.1 glTF 概述

信息时代,我们每天都在与各式各样的数据格式打交道(.txt、.doc、.pdf、.jpg 等),数据格式是一种规范,通过预先定义好的规则进行读写,确保数据能被正确地存储和解析。对三维模型而言,由于结构比较复杂、应用领域各异、功能属性侧重点不同,格式更加繁多,如常见的.obj、.fbx、.stl 等。通常三维模型都由三维建模软件创建,如 3ds Max、Blender 等,这些建模软件可以编辑三维模型文件,如顶点、面片、相机、动画、顶点材质等,建模软件通常以自己的方式来存储这些可编辑的三维场景数据,例如 Blender 将它的场景数据存储为.blend 文件,LightWave3D 存储为.lws 文件,3ds Max 存储为.max 文件,这些与场景编辑相关的模型格式通常称为源文件,源文件包含很多与编辑三维模型相关的信息,它们与导出发布文件格式的关系类似于 PSD 格式与 JPG 格式之间的关系。

三维模型源文件通常只有创建该模型的软件才能解析,而发布后的文件则可以供遵循格式标准的软件读取,通常这类发布后的三维数据格式不包含场景相关数据,由读取软件进行几何数据预处理和渲染。图形语言传输格式(Graphics Language Transmission Format, glTF)的目标是为三维模型数据格式提供统一的标准,方便读取和渲染,它由三维图形标准组织 Khronos 定义,已成为 Web 端三维模型通用格式标准。

glTF 格式是一种中间格式,它使用 JSON 描述场景结构,方便应用程序处理,其三维模型几何数据不需要应用程序解码或者预处理就可以被大多数图形 API 使用,它是三维模型创

^① 目前只在 Chrome、Safari、Edge、Samsung Internet 浏览器上获得支持。

建软件与渲染应用软件之间的桥梁。glTF 正在成为 Web 端三维模型的通用格式，目前绝大多数三维建模软件支持该格式的导出版。

glTF 支持纹理、动画，支持模型几何数据分离 (GLB 格式)，方便 Web 端引用和传输。glTF 格式的核心是一个 JSON 文件，该文件描述了三维场景的全部内容，包括场景结构、渲染、动画和层级关系，该文件通过节点 (nodes) 的层级关系 (hierarchy) 定义场景图 (Scene Graph)，其结构如图 12-1 所示。

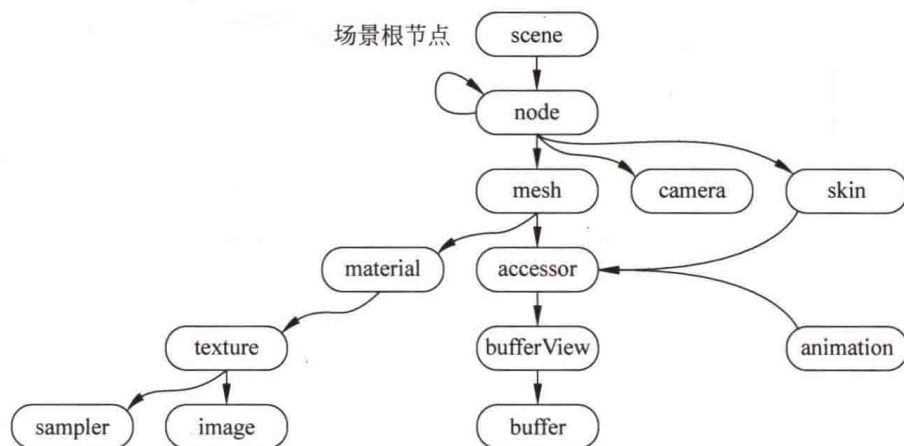


图 12-1 glTF 格式描述文件层级结构图

场景中的三维模型几何结构由节点下的 mesh 定义，而 material 定义了对对象的外观、animation 定义三维模型变换动画（如旋转、翻转等）、skin 定义了三维模型蒙皮变形、camera 描述了渲染器 (renderer) 的视图配置 (View Configuration) 等，各节点内容如表 12-1 所示。

表 12-1 glTF JSON 文件各节点含义

节点名	描 述
scene	glTF 整个场景描述的入口，它指向场景中的一个或者多个节点，每个节点又可以有子节点 (children node)
node	场景层级结构中的一个节点，它可以包含一个变换 (Transformation，如旋转或翻转等)，可以指向其他子节点。通常，一个节点指向该节点的 mesh 或 camera 对象，也可以指向 skin 对象
camera	定义场景渲染时的视图配置
mesh	描述一个节点中的几何对象，它指向 accessor 和 material 对象，其中 accessor 对象用于获得具体的几何数据，material 则定义了渲染对象外观表现
skin	定义 Vertex Skinning 需要的参数，使 mesh 可以基于模型的位姿发生形变，所有数据源于 accessor 对象
animation	描述节点随时间变化的动画信息
accessor	是任意数据 (Arbitrary Data) 的抽象，mesh、skin、animation 对象都需要用到它提供的几何数据 (Geometry Data)、Skinning Parameters 或基于时间的动画值 (Time-dependant Animation Values)。在图 12-1 中，mesh、skin、animation 都指向 accessor。accessor 会引用 bufferView，一个 bufferView 是 buffer 的一部分，buffer 包含了实际的原始三维模型的几何数据
material	定义渲染对象时的外观参数，通常引用 texture 对象，将其作为纹理应用到渲染中
texture	通过一个 sampler (采样器) 和一个 image 来定义，sampler 定义纹理图像 image 如何采样

一个带有材质信息的最简单的 glTF 格式文件如下：

```
// 第 12 章 /12-1
{
  "scenes" : [
    {
      "nodes" : [ 0 ]
    }
  ],

  "nodes" : [
    {
      "mesh" : 0
    }
  ],

  "meshes" : [
    {
      "primitives" : [ {
        "attributes" : {
          "POSITION" : 1
        },
        "indices" : 0,
        "material" : 0
      } ]
    }
  ],

  "buffers" : [
    {
      "uri" : "data:application/octet-stream;base64,AAABAAAA/AAAAAAAAA=",
      "ByteLength" : 44
    }
  ],

  "bufferViews" : [
    {
      "buffer" : 0,
      "ByteOffset" : 0,
      "ByteLength" : 6,
      "target" : 34963
    },
    {
      "buffer" : 0,
      "ByteOffset" : 8,
      "ByteLength" : 36,
      "target" : 34962
    }
  ]
}
```

```

],
"accessors" : [
  {
    "bufferView" : 0,
    "ByteOffset" : 0,
    "componentType" : 5123,
    "count" : 3,
    "type" : "SCALAR",
    "max" : [ 2 ],
    "min" : [ 0 ]
  },
  {
    "bufferView" : 1,
    "ByteOffset" : 0,
    "componentType" : 5126,
    "count" : 3,
    "type" : "VEC3",
    "max" : [ 1.0, 1.0, 0.0 ],
    "min" : [ 0.0, 0.0, 0.0 ]
  }
],
"materials" : [
  {
    "pbrMetallicRoughness": {
      "baseColorFactor": [ 1.000, 0.766, 0.336, 1.0 ],
      "metallicFactor": 0.5,
      "roughnessFactor": 0.1
    }
  }
],
"asset" : {
  "version" : "2.0"
}
}

```

通过前文的描述，现在应该能基本理解 glTF JSON 文件格式的含义了，通常而言，这个文件由三维建模软件导出 glTF 模型时自动生成，但在使用时，可能会需要对该文件进行修改。一般三维模型的几何、纹理数据不会包含在 JSON 文件中，而是存储在专门的文件或者文件夹中，JSON 文件仅包含指向这些文件的路径或链接，这使模型或者纹理数据可以以非常紧凑的二进制形式存储，更有效地通过 Web 传输。鉴于本书目的，更详细的 glTF 格式的相关内容可查阅官方文档。

12.2 Model Viewer

为更好地传播及共享 AR 体验，谷歌公司在 2019 年引入了 Model Viewer (模型展示器，

为避免混淆, 后文直接使用英文) 功能, 并在 Chrome v83 及以上版本浏览器中深度集成了 Model Viewer, 可以在浏览器中利用 ARCore 运动跟踪和平面检测能力体验 AR 功能。Model Viewer 实际上是一个浏览器功能, 它接受 Web 端 JavaScript 指令, 在需要的时候调用设备本地 ARCore 提供 AR 运动跟踪、平面检测服务, 起一个中介和桥梁作用, 使 JavaScript 端代码能顺利调用设备本地端 API, 从而实现 AR 功能, 因此, 使用 Model Viewer 实现 AR 功能的前提是设备硬件支持 ARCore 并安装了 ARCore 服务应用程序, 然后使用 Chrome 等支持该功能的浏览器进行三维模型 AR 展示。

通过 Model Viewer 进行三维模型展示和 AR 体验时, 其模型只能使用 glTF 格式。

12.2.1 Model Viewer 使用实例

在技术层面, 谷歌公司创造了一个 HTML 页面标签 `<model-viewer></model-viewer>`, 与其他内容标签一样, 该标签既支持属性, 也支持内嵌其他标签内容。在页面加载后, `model-viewer.min.js` 脚本^①会解析该标签, 然后通过浏览器查询设备支持情况, 如果设备支持 AR 功能, 则通过浏览器启动设备本地端 ARCore, 实现 AR 功能, 如果设备不支持 AR 功能, 则构建一个在线三维模型渲染界面, 提供三维模型浏览功能。在实现时, 还需要 Google LLC 工具的支持, 这样才能通过浏览器安全地调用设备本地端 ARCore, 执行相应功能, 所以使用该功能时, 首先需要更新 Google LLC 工具^②, 并同时更新 ARCore 到最新版本, 如图 12-2(a) 所示, 如果没有更新到最新版, 在使用 Model Viewer 尝试进行 AR 体验时会弹出如图 12-2(b) 所示提示框。



图 12-2 使用 Model Viewer 体验 AR 功能时需要安装 Google LLC 工具和 ARCore

一个简单的使用 Model Viewer 功能的 HTML 页面脚本如下:

```
// 第 12 章 /12-2
<!doctype html>
<html lang="cn">
  <head>
    <title>model viewer 使用演示 </title>
```

① 下载网址为 <https://unpkg.com/@google/model-viewer/dist/model-viewer.min.js>。

② 由于一些原因, 更新 Google LLC 工具需要科学上网。


```
<meta charset="utf-8">
<meta name="description" content="model viewer 使用演示 ">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link type="text/css" href="images/styles.css" rel="stylesheet"/>
<script src="js/focus-visible.js" defer></script>
</head>
<body>
  <!-- model-viewer 模块 -->
  <model-viewer bounds="tight" src="model/Robot.glb" ar ar-modes="webxr scene-
viewer quick-look" camera-controls environment-image="neutral" poster="images/
poseter.png" shadow-intensity="1" autoplay>
    <div class="progress-bar hide" slot="progress-bar">
      <div class="update-bar"></div>
    </div>
    <button slot="ar-button" id="ar-button">
      在你的空间内查看
    </button>
    <div id="ar-prompt">
      
    </div>
  </model-viewer>
  <!-- 用户控制脚本 -->
  <script src="js/script.js"></script>
  <!-- 加载 model viewer 模块 -->
  <script type="module" src="js/model-viewer.min.js"></script>
</body>
</html>
```

从上述 HTML 脚本可以看到，使用 Model Viewer 功能时需要在页面中引入 model-viewer.min.js 和 focus-visible.js 脚本文件，其中前者用于在浏览器中解析 <model-viewer> 标签，而后者是可选脚本，主要用于针对用户输入进行一些适配性工作，上述 HTML 脚本 <model-viewer> 标签中各属性如表 12-2 所示。

表 12-2 <model-viewer> 标签各属性含义

属 性 名	描 述
bounds	计算模型包围盒的方法，该方法会影响模型阴影位置，可选为 Legacy 和 Tight 两种模式之一，Legacy 模式计算相对保守，但有时会导致包围盒过大，Tight 模式计算更加准确，使用该模式时会调用相机参数，确保模型始终处于视野中，默认为 Tight
src	三维模型 URL，可以是绝对路径，也可以是相对路径，只支持 glTF 和 GLB 两种格式
ar	启用 AR 功能
ar-modes	实现 AR 方式的优先级，支持 webxr、scene-viewer、quick-look 3 种模式，webxr 用于在浏览器中体验 AR 功能，scene-viewer 则会调起设备本地端的 Scene Viewer app 应用，quick-look 则是调用 iOS 端的 Quick Look app。模式间通过空格符相隔，其排列顺序决定了 Model Viewer 尝试 AR 体验的技术方式
Camera-controls	支持通过鼠标 / 手势控制渲染相机，默认属性

续表

属 性 名	描 述
environment-image	环境贴图，控制模型的环境光反射，与 skybox-image 属性的区别是，skybox-image 会渲染环境背景，而 environment-image 只会用于模型的环境反射而不会渲染背景。可不设置、设置为 neutral、设置为 .hdr/.jpg 图片 URL，不设置时使用默认环境光，设置为 neutral 时，环境光反射更加均匀，设置为图片 URL 时，则使用该图片进行环境反射
poster	在模型下载完成前显示一张海报图，防止出现空白
shadow-intensity	控制模型阴影强度，取值范围为 [0,1]
autoplay	在模型有动画时，自动播放动画

需要说明的是，<model-viewer> 标签的属性远不止表 12-2 所示类型，后文会进行详细讲解。除了属性，<model-viewer> 标签也支持内嵌其他标签内容，因此，使用起来非常方便，开发者可以在 Model Viewer 模块的任何位置放置任何需要的其他标签内容。本节演示示例中加入了 progress-bar <div> 标签，用于显示三维模型加载进度；ar-button <button> 标签，用于用户交互，启动 AR 功能；ar-prompt <div> 标签，用于引导用户进行运动跟踪和平面检测操作。

示例页面在支持 Model Viewer 功能的手机设备上打开后的效果如图 12-3（a）所示，当用户单击“在你的空间内查看”按钮后，浏览器即会弹出要求使用摄像头授权申请，确认后即会启动 ARCore 服务，在检测到的平面上自动放置虚拟物体对象，如图 12-3（b）所示。不管是在页面端以三维形式浏览模型还是在 AR 空间中呈现模型，都可以使用屏幕手势进行操作，使用单指选择模型后进行滑动可以平移、在模型区域之外使用单指滑动可以旋转、使用双指扭转（双指向同一方向顺时针或者逆时针旋转）也可以进行旋转、使用双指捏合可以缩放。单击窗口“×”符号返回。

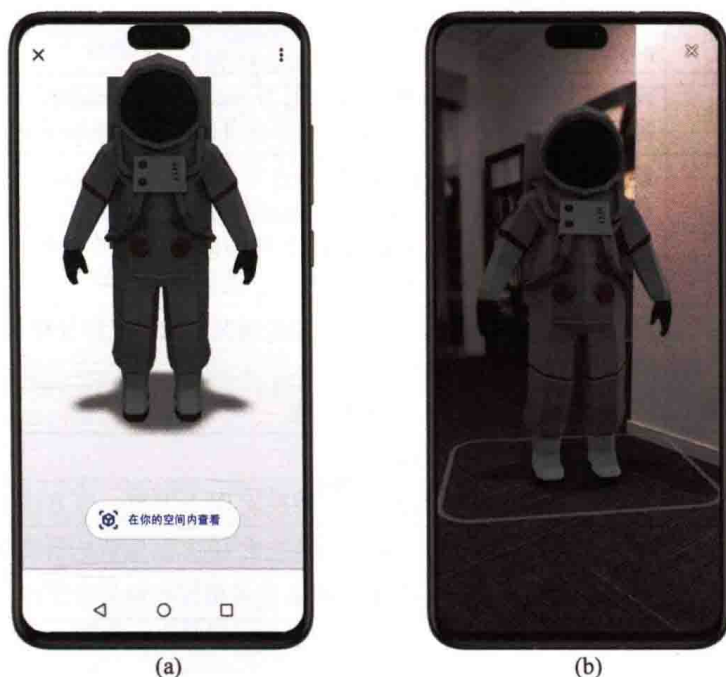


图 12-3 使用 Model Viewer 体验 AR 功能效果图

<model-viewer> 标签已在当前主流的 4 个浏览器的最新版本中得到支持: Chrome、Safari、Edge、Samsung Internet。通过 Web 实现 AR 的方式, 极大地强化了 AR 传播的便捷性, 在电商零售、产品展示、儿童教育等领域有着广泛的应用前景。

<model-viewer> 标签支持 CSS 样式属性设置、内嵌其他标签内容, 使用起来非常灵活, 可以无缝接入原有网页中, 实现无缝集成的 AR 体验。

12.2.2 Model Viewer 详解

在 Web 页面中, Model Viewer 以 <model-viewer> 标签形式引入功能, 自动检测设备的 AR 支持情况, 在设备不支持 AR 功能时使用三维形式浏览模型, 在设备支持 AR 功能时, 可引导用户开启 AR 体验。

为全面控制 <model-viewer> 标签的行为表现, <model-viewer> 标签将相关属性、方法、事件分成 7 大类, 下面分别进行阐述。

1. 加载类

加载类属性、事件、方法与模型加载相关, 主要用于控制模型加载时的行为表现, 如模型 URL、海报展示、模型加载方式、加载背景颜色等, 主要属性、方法、事件如表 12-3 所示。

表 12-3 <model-viewer> 标签加载类属性、方法、事件含义

名 称	类 型	描 述
src	属性	指向三维模型的 URL, 只能使用 glTF 或者 GLB 格式模型
alt	属性	为 <model-viewer> 标签提供文本描述, 主要用于屏幕阅读软件
poster	属性	在模型下载完成前显示一张海报图, 如果将海报图设置为透明, 则需要配合 poster-color 属性使用
loading	属性	设置三维模型加载方式, 可选值为 auto、lazy、eager, auto 等同于 lazy, 在需要的时候才下载, 而 eager 则会在页面打开后进行下载
reveal	属性	设置三维模型展示方式, 可选值为 auto、interaction、manual, auto 值在模型下载完成后马上进行渲染, interaction 值在用户与海报图进行交互时 (如单击) 才进行渲染, manual 值不渲染模型, 即使模型已经下载完成, 只能通过代码调用 dismissPoster() 方法才进行模型渲染
loaded	属性	只读属性, 用于指示三维模型加载情况, 如果三维模型加载已完成, 则返回值为 true, 否则返回值为 false
showPoster()	方法	展示海报图, 隐藏三维模型渲染
getDimensions()	方法	返回三维模型包围盒尺寸
toBlob()	方法	将三维模型数据当前视角快照转换成 Blob 对象, 该方法返回一个 promise
toDataURL()	方法	将三维模型数据当前视角快照转换成 URL 图像数据格式
error	事件	当三维模型下载错误或者 webGL 出现错误时触发该事件

2. AR 类

AR 类属性、事件、方法与 AR 体验相关，主要属性、方法、事件如表 12-4 所示。

表 12-4 <model-viewer> 标签 AR 类属性、方法、事件含义

名 称	类 型	描 述
ar	属性	在支持 AR 功能的设备上开启 AR 功能
ar-modes	属性	实现 AR 方式的优先级，支持 webxr、scene-viewer、quick-look 3 种模式，webxr 在浏览器中体验 AR 功能，scene-viewer 则会调起设备本地端的 Scene Viewer App 应用，quick-look 则会调用 iOS 端的 Quick Look App。模式值通过空格符相隔，其排列顺序决定 Model Viewer 尝试 AR 体验的技术方式
ar-scale	属性	是否允许三维模型缩放，如家具放置类应用，应当避免用户缩放模型。可选值为 auto、fixed，默认为 auto，允许缩放，当设置为 fixed 时，则不允许缩放模型
ar-placement	属性	ARCore 支持两种平面检测，即水平和垂直，使用该属性设置平面检测和虚拟物体放置方式，可选值为 floor、wall
ios-src	属性	主要用于兼容 ARKit AR Quick Look，用于设置 USDZ 模型文件 URL
xr-environment	属性	开启环境光估计，当使用该属性时，ARCore 会实时进行环境光估计，environment-image 之类的环境反射设置将失效
ar-status css 属性	属性	只读属性，用于指示当前 AR 状态，可能值为 not-presenting、session-started、object-placed、failed
ar-tracking	css 属性	只读属性，用于指示当前 AR 运动跟踪状态，可能值为 tracking、not-tracking
canActivateAR	css 属性	只读属性，设备支持 AR 功能时，该值的返回值为 true，否则返回值为 false
activateAR()	方法	激活 AR 功能，通常通过用户交互调用
ar-status	事件	当 AR 状态变化时触发
ar-tracking	事件	当 AR 运动跟踪状态变化时触发
quick-look-button-tapped	事件	在 iOS 端，用户单击 AR 按钮时触发

3. 相机与展示类

相机展示类属性、事件、方法与相机渲染及模型展示相关，主要属性、方法、事件如表 12-5 所示。

表 12-5 <model-viewer> 标签相机展示类属性、方法、事件含义

名 称	类型	描 述
camera-controls	属性	支持通过鼠标 / 手势控制渲染相机，默认属性
touch-action	属性	支持通过鼠标 / 手势平移三维物体，默认为 pan-y

续表

名 称	类型	描 述
disable-zoom	属性	相机缩放控制，默认不添加该属性
orbit-sensitivity	属性	调整 θ 和 ϕ 角速度，通常用于查看模型的内部结构
auto-rotate	属性	三维模型是否自动旋转，默认不添加该属性
auto-rotate-delay	属性	三维模型自动旋转延时，即多长时间后开始自动旋转，单位为毫秒，默认值为 3000
rotation-per-second	属性	每秒旋转的弧度或者角度，如 30deg、0.5rad
interaction-policy	属性	交互策略，可选值为 always-allow、allow-when-focused，默认值为 always-allow
interaction-prompt	属性	交互引导提示，可选值为 auto、when-focused、none，默认值为 auto
camera-orbit	属性	设置相机的起始或后续位置，通过设置相机轨道，可以实现动画效果
camera-target	属性	设置相机关注点，即设置相机指向的位置
field-of-view	属性	设置视口，取值范围为 [0,180]
bounds	属性	计算模型包围盒的方法，该方法会影响模型阴影位置，可选为 Legacy 和 Tight 两种模式之一，Legacy 模式计算相对保守，但有时会导致包围盒过大，Tight 模式计算更加准确，使用该模式时会调用相机参数，确保模型始终处于视野中，默认为 Tight
resetInteractionPrompt()	方法	再次显示交互引导
getCameraOrbit()	方法	获取实时的相机轨道位置信息
getCameraTarget()	方法	获取实时的相机指向位置信息
getFieldOfView()	方法	获取相机视口信息
camera-change	事件	当相机位置、视口发生变化时触发

4. 标注类

标注类属性、方法与模型标注相关，主要属性、方法、事件如表 12-6 所示。

表 12-6 <model-viewer> 标签标注类各属性含义

名 称	类 型	描 述
--min-hotspot-opacity	css 属性	设置隐藏热点标注的不透明度，取值范围为 [0,1]，默认值为 0.25
--max-hotspot-opacity	css 属性	设置显示热点标注的不透明度，取值范围为 [0,1]，默认值为 1
updateHotspot()	方法	更新热点标注
positionAndNormalFromPoint()	方法	获取用户单击屏幕时与模型进行射线检测碰撞点在世界坐标系下的位置、法线和 UV 坐标信息
hotspot-*	子元素	<model-viewer> 标签子元素，每个以该值开头的元素标识一个热点

标注是在三维模型展示时使用得非常多的功能，主要对展示物体的特定部件、零件进行注解，非常适合对物体组成原理、组成要素的说明。<model-viewer> 标签支持静态标注、动态标注、二维标注、三维标注等标注类型，可以满足绝大部分使用需求。一个典型的标注 HTML 脚本如下：

```
// 第 12 章 /12-3
<style>
  button{
    display: block;
    width: 20px;
    height: 20px;
    border-radius: 10px;
    border: none;
    background-color: blue;
    box-sizing: border-box;
  }

  button[slot="hotspot-hand"]{
    --min-hotspot-opacity: 0;
    background-color: red;
  }

  button[slot="hotspot-foot"]:not([data-visible]) {
    background-color: transparent;
    border: 3px solid blue;
  }

  #annotation{
    background-color: #888888;
    position: absolute;
    transform: translate(10px, 10px);
    border-radius: 10px;
    padding: 10px;
  }
  /* 在模型加载时隐藏标注 */
  :not(:defined) > * {
    display: none;
  }
</style>

<model-viewer id="hotspot-demo" ar ar-modes="webxr" camera-controls src="model/
Astronaut.glb" style="--poster-color:#ffff00;" alt="宇航服">
<!-- 三维 标注点，通过 position、normal 指定位置和方向 -->
<button slot="hotspot-visor" data-position="0 1.75 0.35" data-normal="0 0 1"></button>
<button slot="hotspot-hand" data-position="-0.54 0.93 0.1" data-normal="-0.73
0.05 0.69">
<!-- 常规注释，二维文字 -->
```



```
<div id="annotation"> 二维提示文字 </div>
</button>
<!-- 三维标注点, 通过 position、normal 指定位置和方向 -->
<button slot="hotspot-foot" data-position="0.16 0.1 0.17" data-normal="-0.07
0.97 0.23" data-visibility-attribute="visible"></button>
<div id="annotation"> 世界空间中的三维标注 </div>
</model-viewer>
```

使用该标注实现的效果如图 12-4 所示。

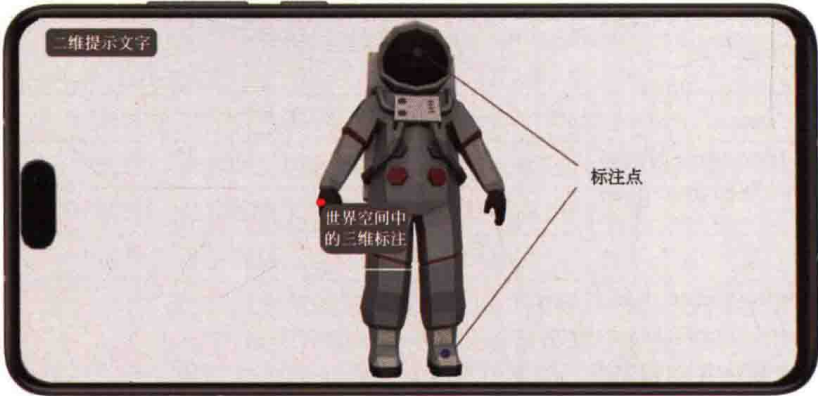


图 12-4 使用 Model Viewer 进行空间标注效果图

5. 光照与环境反射类

光照与环境反射类属性、事件与模型渲染光照和环境反射相关，主要属性、方法、事件如表 12-7 所示。

表 12-7 <model-viewer> 标签光照与环境反射类各属性含义

名 称	类型	描 述
skybox-image	属性	用于设置天空盒，在三维浏览模式下有效，支持 png、jpg、hdr 格式
environment-image	属性	环境贴图，控制模型的环境光反射，与 skybox-image 属性的区别是，skybox-image 会渲染环境背景，而 environment-image 只会用于模型的环境反射而不会渲染背景。可不设置、设置为 neutral、设置为 .hdr/.jpg 图片 URL，不设置时使用默认环境光，设置为 neutral 时，环境光反射更加均匀，设置为图片 URL 时，则使用该图片进行环境反射
exposure	属性	控制曝光度，取值范围为 [1, ∞]
shadow-intensity	属性	控制模型阴影强度，取值范围为 [0,1]
shadow-softness	属性	控制模型阴影软度，取值范围为 [0,1]，其中 0 为硬边阴影，而 1 为模糊阴影，该值越大，阴影渲染越快
environment-change	事件	skybox-image 或者 environment-image 属性值发生变化时触发

6. 动画类

glTF 格式支持模型动画，利用 `<model-viewer>` 标签可以灵活地控制动画播放行为，其主要属性、方法、事件如表 12-8 所示。

表 12-8 `<model-viewer>` 标签动画类属性、方法、事件含义

名 称	类型	描 述
animation-name	属性	指定需要播放的动画名，动画名是 glTF 模型中预先设置好的动画名称之一，如果没有设置该值，在使用 play() 方法播放动画时则会选择模型中第 1 个动画进行播放
animation-crossfade-duration	属性	当模型动画切换时， <code><model-viewer></code> 标签会自动进行动画融合插值，该属性指定两段动画过渡时长，单位为毫秒，默认值为 300
autoplay	属性	在三维模型有动画时，自动播放动画
availableAnimations	属性	获取已加载三维模型的动画数，返回一个 string 类型数组
currentTime	属性	获取当前正在播放动画的播放进度，如果当前没有动画播放，则返回 0
timeScale	属性	该属性用于慢放、快进动画播放，其值小于 1 时为慢速播放，大于 1 时为快进播放
duration	属性	只读属性，获取当前正在播放动画的总时长，如果当前没有动画播放，则返回 0
paused	属性	只读属性，获取当前动画是否已暂停
play()	方法	播放动画，可以设置该方法参数 {repetitions, pingpong}，参数 repetitions 指定动画重复次数，将参数 pingpong 设置为 true 值时在动画正向播放完后会倒向播放
pause()	方法	暂停动画播放，如果需要重置当前动画，则需要将其 currentTime 属性设置为 0
play	事件	在动画开始播放时触发
pause	事件	在动画暂停播放时触发
loop	事件	当动画重复播放时触发，每重播一次触发一次
finished	事件	当动画播放结束时触发

7. 场景相关类

场景类属性、事件、方法与模型渲染场景相关，其主要属性、方法、事件如表 12-9 所示。

表 12-9 `<model-viewer>` 标签场景相关类各属性含义

名 称	类型	描 述
variant-name	属性	通过三维模型变量设置模型属性，如给模型替换材质、纹理等
orientation	属性	通过 roll、pitch、yaw 欧拉角设置场景方向，如 0deg 0deg 0deg
scale	属性	设置场景缩放值，可独立控制各轴方向的缩放比例，如 0.5 1 1.5

续表

名 称	类型	描 述
availableVariants	属性	用于获取当前场景所有的三维模型
exportScene()	方法	将场景中的三维模型输出, 该方法返回一个 promise
materialFromPoint()	方法	获取屏幕单击点的模型材质
scene-graph-ready	事件	模型加载完成, 进行渲染时触发

通过上述内容可以看到, Model Viewer 功能强大灵活, 利用 <model-viewer> 标签可以自由地控制模型三维、AR 展示各阶段、各方面, 更详细的关于 Model Viewer 功能属性方法内容可查阅官方文档^①。

需要注意的是, Model Viewer 功能需要通过 HTTPS 协议进行通信, 另外, 如果该标签包含在 iframe 中, 则 iframe 必须允许源站使用 xr-spatial-tracking 策略。另外, Model Viewer 功能只能通过 Web 页面提供服务, 为扩展其功能, 谷歌公司又开发了 Scene Viewer 功能, Scene Viewer 功能允许通过 Web 页面和 App 端快速接入 AR 功能, 但由于该功能需要在使用时连接 arvr.google.com 服务器, 由于一些原因, 国内并不能方便地使用, 本书不再详细阐述。

12.2.3 使用 Model Viewer 的注意事项

Model Viewer 功能非常强大, 可以渲染出真实感极强的 AR 效果, 如 PBR 材质、实时软阴影、环境反射、模型动画等, 但开发人员应当明白, 营造这些出色效果的背后是强大的计算资源支持, 如果过多过滥地使用材质、纹理、模型则会导致性能的快速恶化, 应用会出现卡顿、假死等, 特别需要提醒的是, Web 端的渲染性能与设备本地端相差非常大, 大约只有本地端渲染能力的 1/10 到 1/5, 因此, 在使用 Model Viewer 时需要进行一些限制与优化。

1. 优化模型和透明度

通过 glTF 模型格式, Model Viewer 支持多模型展示, 但通常而言, 每个模型渲染都需要一个 Drawcall, 过多的 Drawcall 会导致 GPU 与 CPU 过载, 因此, 建议将模型数目 (Meshes) 控制在 10 个以内。另外, 为优化性能, 可以将共享相同材质的模型对象合并, 尽量使用合并各种小纹理的大纹理集, 而不是分开使用小纹理, 因为纹理不同会导致模型网格无法合并。

Model Viewer 使用投影阴影 (Project Shadow), 而阴影生成是一项非常消耗计算资源的操作, 为减少内存使用、提高性能, 通常要求所有渲染的模型多边形数保持在 10 万面以下。

透明度混合是逐像素进行的计算密集型操作, 会大大地增加片元着色器 (Fragment Shader) 的消耗, 因此, 若希望通过减少应用程序着色器压力而提高性能, 则应谨慎使用透明度。

^① 文档网址为 <https://modelviewer.dev/docs/>。

2. 控制纹理数量

Model Viewer 支持包括漫反射纹理、金属纹理、法线纹理、自发光纹理、OA 遮挡纹理等 PBR 渲染纹理，大量纹理的处理不仅要消耗内存资源，还会消耗计算资源。为提升性能，纹理尺寸大小通常设置为 1024×1024 像素或更小，最好不要超过 2048×2048 像素。如果有多个小纹理，则可以使用纹理坐标将它们合并为一个较大的纹理。同时，建议每个模型文件的纹理数量尽量控制在 4 个以内。

3. 在各种设备上测试

支持 ARCore 的设备多种多样，硬件性能也各不相同，为更好地兼容所有设备，在开发时应当先从一个保守的设备配置开始，以确保 Model Viewer 在低硬件资源的设备上得到支持，Model Viewer 也会自动检测设备并根据设备性能最佳化使用体验。

高级篇



本篇为在移动设备上进行 AR 应用开发高级主题篇，主要阐述了 Android 与 Unity 交互通信、AR 应用设计、性能优化等相关主题，提升开发人员在 AR 应用开发时的实际应用能力和整体把握能力。

高级篇包括以下章节：

第 13 章 Android 与 Unity 交互通信

ARCore 运行于 Android 平台，在实际项目中，使用 Unity 开发 ARCore 应用时，通常涉及 Android 与 Unity 交互通信等相关问题，本章从实际出发，详细阐述了在 AR 应用开发过程中 Android 与 Unity 之间的相互代码调用、数据通信、功能控制相关知识，着力提升读者的实际开发能力。

第 14 章 设计原则

本章讨论了 AR 应用与普通应用的区别，也指出了在 AR 应用设计开发时应该注意的事项，提出了在 AR 应用设计开发中应该遵循的基本准则，阐述了提升虚拟对象渲染真实感的方法技巧，着力提升 AR 应用的用户体验。

第 15 章 性能优化

AR 是计算密集型应用，其运行的载体为轻便移动设备，软硬件资源非常有限，本章主要对 AR 开发时的性能问题排查及优化技术进行学习，着力提升 AR 应用的性能。

第 13 章



Android 与 Unity 交互通信

ARCore 只运行于 Android 平台，而 Unity 引擎被设计用于跨平台开发，为适应跨平台需求，Unity 引擎架构设计要复杂灵活得多，基于 Unity 引擎开发的 AR 应用运行于独立的 VM(Virtual Machine, 虚拟机) 中^①，这给 AR 应用与 Android 原生系统代码的交互带来了困难。在实际应用中，基于 ARCore 的 AR 应用与底层的 Android 平台之间经常有交互需求，本章主要学习 Android 平台与 Unity 引擎的交互通信。

13.1 Android 与 Unity 通信原理

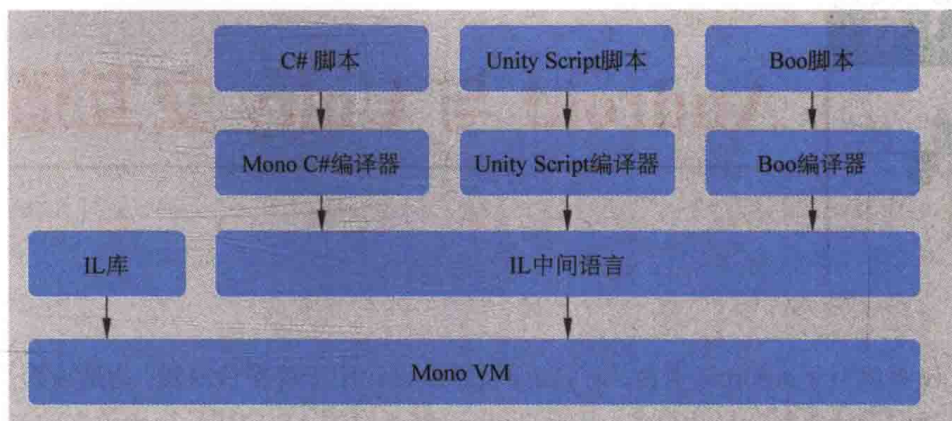
Unity 引擎最大的优势和特点是一次制作、多端部署，极大地减轻了多平台游戏的开发和维护成本，而 Unity 引擎实现强大跨平台能力的基础是 Mono / IL2CPP，Mono / IL2CPP 是 Unity 引擎跨平台的核心和根本。

在 2001 年，电信标准组织 ECMA 制定了一个与特定语言无关的跨体系结构的运行环境 CLI (Common Language Infrastructure, 公共语言基础) 标准规范，只要使用规范定义的高级语言进行开发、应用程序符合 CLI 规范即可确保在不同的计算机体系结构上实现跨平台运行。在此基础上，微软公司根据标准实现了 .NET Framework 公共语言运行时 (Common Language Runtime, CLR)，因此 CLR 是 CLI 的一个实现，.NET Framework 是一个运行于 CLR 基础上的框架，它支持 C#、VB.NET、C++、Python 等语言，但由于 .NET Framework 与 Windows 的深厚渊源，.NET Framework 本身并不能跨平台。

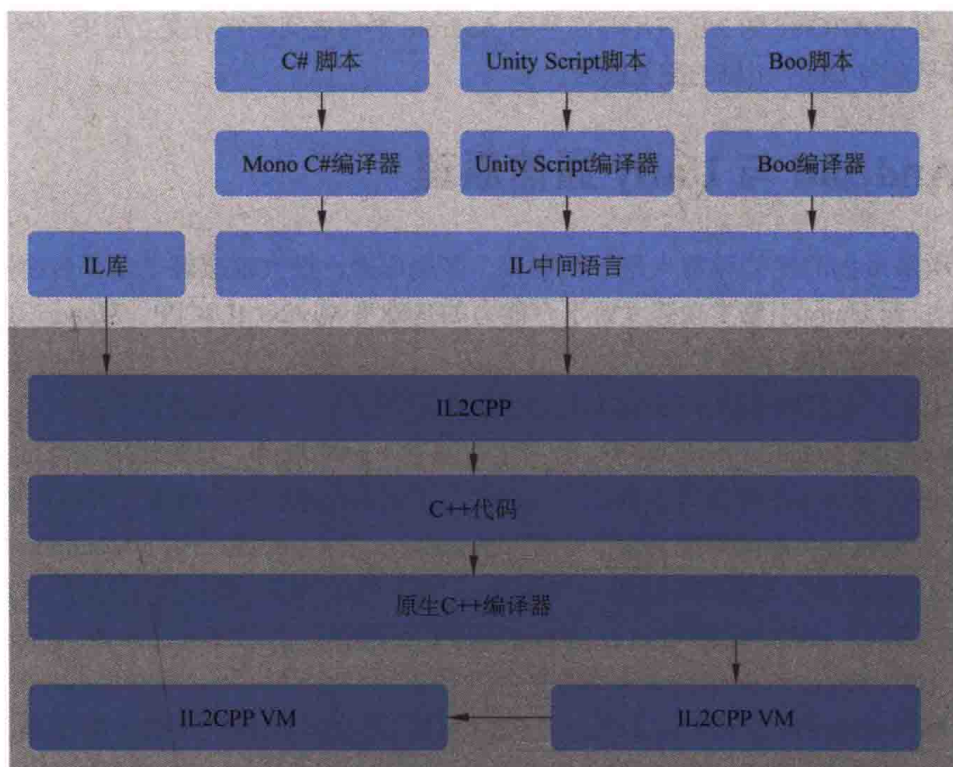
Mono 则是 Xamarin 公司主导的另一个 CLI 实现，它通过内置 C# 语言编译器、CLR 运行时和各类基础类库，可以使应用程序运行在 Windows、Linux、FreeBSD、Android、iOS 等各种平台上，因此，通过 Mono 能使用 C# 语言编写 Android 或 iOS 应用程序。在 CLI 规范中，高级语言并不是直接被编译成机器字节码，而是编译成中间语言 (Intermediate Language, IL)，这是一种介于高级言与机器字节码之间的与特定底层硬件无关的语言，在真正需要执行

^① 采用 IL2CPP 后端编译的应用，运行时仍然需要虚拟机支持。

的时候, IL 会被加载到 Mono VM 中^①, 由 VM 动态地编译成机器码再执行 (Just In Time, JIT 编译), 其执行过程如图 13-1 (a) 图所示。



(a)



(b)

图 13-1 Mono 编译运行代码与 IL2CPP 编译运行代码流程示意图

通过 Mono, Unity 引擎能够将应用部署到各类不同的系统平台上, 实现跨平台运行, 但由于 IL 在 Mono VM 中运行, 而 Mono VM 与系统平台紧密相关, 无法实现跨平台, 所以

^① Unity 支持 C#、Unity Script、Boo 三种脚本开发语言, 所有高级语言都会被编译成 IL 中间语言。

Unity 需要维护各种平台数量众多的 Mono VM，并且 JIT 编译方式也降低了应用程序的运行速度，因此，Unity 引擎引入了 IL2CPP 后端编译方式，通过将 IL 重新编译成 C++ 代码，再由各类平台的 C++ 编译器直接编译成原生汇编代码（Ahead Of Time，AOT 编译），从而提升代码执行效率，并可利用各平台的 C++ 编译器执行编译期间的优化，但由于高级动态语言的特性，内存管理和回收仍然需要统一维护，即需要 IL2CPP VM 负责 GC、线程等服务性工作，但此时 IL2CPP VM 不再负责 IL 加载和动态解析，维护 VM 的工作量大为降低。

虽然 IL2CPP 最后采用 AOT 方式直接编译成各平台原生机器码，但其继承了 IL 中间语言的机制，对上层语言几乎没有影响^①，但它也需要借助 VM 进行内存管理等工作，如图 13-1(b) 所示。Android 原生应用采用 Java 语言编写，Java 语言也是先编译为 Bytecode IL 中间语言，然后依赖 Android 上的 Dalvik /Art 虚拟机解释执行，所以 Unity 引擎与 Android 原生代码交互通信实际上是两个 VM 之间的通信，如图 13-2 所示。虽然 Unity 代码与 Android 原生代码在不同的 VM 中执行，但它们都处于同一个进程中，因此可以共享数据。

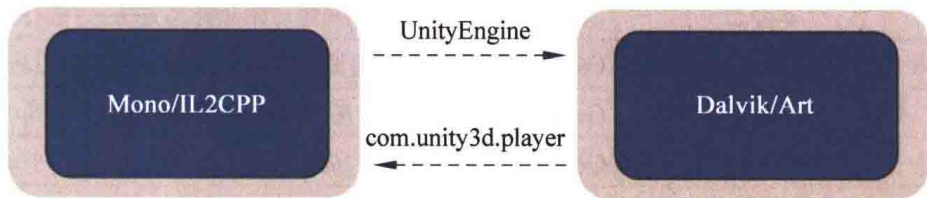


图 13-2 Unity 与 Android 交互通信示意图

在图 13-2 中，Unity 引擎通过 UnityEngine 提供的 API 调用 Android 方法，Android 则借助于 com.unity3d.player 包提供的 API 调用 Unity 方法。通过这种方式，Unity 引擎可以直接调用 Android 类及对象的方法，而 Android 则只能调用 Unity 中指定 GameObject 所挂载的脚本的方法，或者通过动态代理的方式调用 Unity 引擎中的方法。

在执行层面，UnityEngine 封装了 AndroidJavaObject、AndroidJavaClass、AndroidJavaProxy 类，通过这几个类就可以获取 Android 端静态类或者动态对象，从而可以执行其相应方法；Android 则通过 Unity 应用的 MainActivity 与 C# 代码通信^②。

Activity 是 Android 应用中最基本和最重要的组件之一，其作用类似于 Web 中的 page、winform 中的 form，因此，只有活跃的 Activity 才能响应输入，所以，Android 代码首先要通过 com.unity3d.player.UnityPlayer 包下的 currentActivity 获取活跃 Activity，利用其与 Unity 代码通信。

① 因为 C++ 是静态语言，采用 AOT 方式编译，JIT 动态语言的一些特性将不再可用。

② 本章讲述的 Unity 与 Android 的交互通信实际上是指 C# 代码与 Java 代码、JAR 包、AAR 包、SO 包的相互调用，但遵循习惯描述为 Unity 引擎与 Android 操作系统软件之间的通信。

13.2 Unity 直接调用 Java 代码

Unity 2018 之后的版本统一使用 Gradle 进行 Android 端的编译、构建和打包，而 Android Studio 也使用 Gradle 进行编译、构建和打包，即它们都使用同一种编译构建工具，也就是说 Java 代码与 C# 代码都可以在 Unity 中被正确地编译到 Android 端，这就为在 Unity 中直接使用 Java 与 C# 语言打下了基础。

而且为了方便映射 Java 数据结构，在 UnityEngine 类中还内置了若干封装好的类，其中最重要的类有 AndroidJavaClass、AndroidJavaObject、AndroidJavaProxy，这些类是进行 Java 端与 C# 端相互调用的基础。AndroidJavaClass 是 Java.lang.Class 类在 Unity 中的表达，主要用于类结构反射、获取类静态属性或者调用类静态方法，其公共方法如表 13-1 所示。

表 13-1 AndroidJavaClass 公共方法列表

公共方法	调用对象 / 属性类型	泛型方法	描 述
Call	非静态	Call<T>	调用对象非静态方法
CallStatic	静态	CallStatic<T>	调用类静态方法
Get	非静态	Get<T>	获取对象非静态属性
GetStatic	静态	GetStatic<T>	获取类静态属性
Set	非静态	Set<T>	设置对象的非静态属性
SetStatic	静态	SetStatic<T>	设置类静态属性
GetRawClass	—	GetRawClass<T>	获取原生 Java 类型的指针，用于 JNI
GetRawObject	—	GetRawObject<T>	获取原生 Java 对象的指针，用于 JNI
Dispose	—	—	IDisposable 接口回调

AndroidJavaObject 是 Java.lang.Object 类在 Unity 中的表达，而 Java.lang.Object 类是 Java 中所有类的基类，因此其能接受所有的 Java 对象，其公共方法如表 13-2 所示。

表 13-2 AndroidJavaObject 公共方法列表

公共方法	调用对象 / 属性类型	泛型方法	描 述
Call	非静态	Call<T>	调用对象非静态方法
CallStatic	静态	CallStatic<T>	调用类静态方法
Get	非静态	Get<T>	获取对象非静态属性
GetStatic	静态	GetStatic<T>	获取类静态属性
Set	非静态	Set<T>	设置对象的非静态属性
SetStatic	静态	SetStatic<T>	设置类静态属性
GetRawClass	—	GetRawClass<T>	获取原生 Java 类型的指针，用于 JNI

续表

公共方法	调用对象 / 属性类型	泛型方法	描 述
GetRawObject	—	GetRawObject<T>	获取原生 Java 对象的指针, 用于 JNI
Dispose	—	—	IDisposable 接口回调

通过表 13-1 和表 13-2 可以看到, 这两个类的公共方法完全一样, 这就为开发者使用这两个类提供了完全一致的使用外观。

在使用中, 通过 `AndroidJavaClass` 类的 `forName()` 方法、`.class` 属性生成相应类的对象, 由于 `Class` 类方法常用于反射, 所以一般用于调用对应类的静态属性或者方法; `AndroidJavaObject` 表示对象, 通过其 `getClass()` 方法可以获取该对象的类型, 所以一般用于调用对象的实例方法或者属性。例如有一个类 `com.davidwang.util`, 其有一个静态方法 `StaticMethod()`, 一个实例方法 `InstanceMethod()`, 则 `new AndroidJavaClass("com.davidwang.util").callStatic("StaticMethod")` 等同于调用 `util.StaticMethod()`, 而 `new AndroidJavaObject("com.davidwang.util").call("InstanceMethod")` 等同于 `new Util().InstanceMethod()`。

下面通过实际例子演示这两个类的基本用法, 由于交互通信涉及两端, 所以先使用 `Android Studio` 创建 `Unity2Java` 类, `Java` 代码如下:

```
// 第 13 章 / 13-1
package com.example.davidwang;

public class Unity2Java {
    public static void StaticPrint(String str){
        System.out.println(str);
    }
    public static int StaticAdd(int a,int b)
    {
        return a+b;
    }

    public void DynamicPrint(String str){
        System.out.println(str);
    }
    public int DynamicAdd(int a,int b)
    {
        return a+b;
    }
}
```

找到该类所在的 `.Java` 文件, 并将该文件复制到 `Unity` 工程 `Assets/Plugins/Android` 目录或其子目录下, 然后在 `Unity` 工程窗口 (`Project` 窗口) 中选中该 `Java` 文件, 在属性窗口 (`Inspector` 窗口) 中查看其导入设置, 确保 `Android` 平台被选择, 如图 13-3 所示。

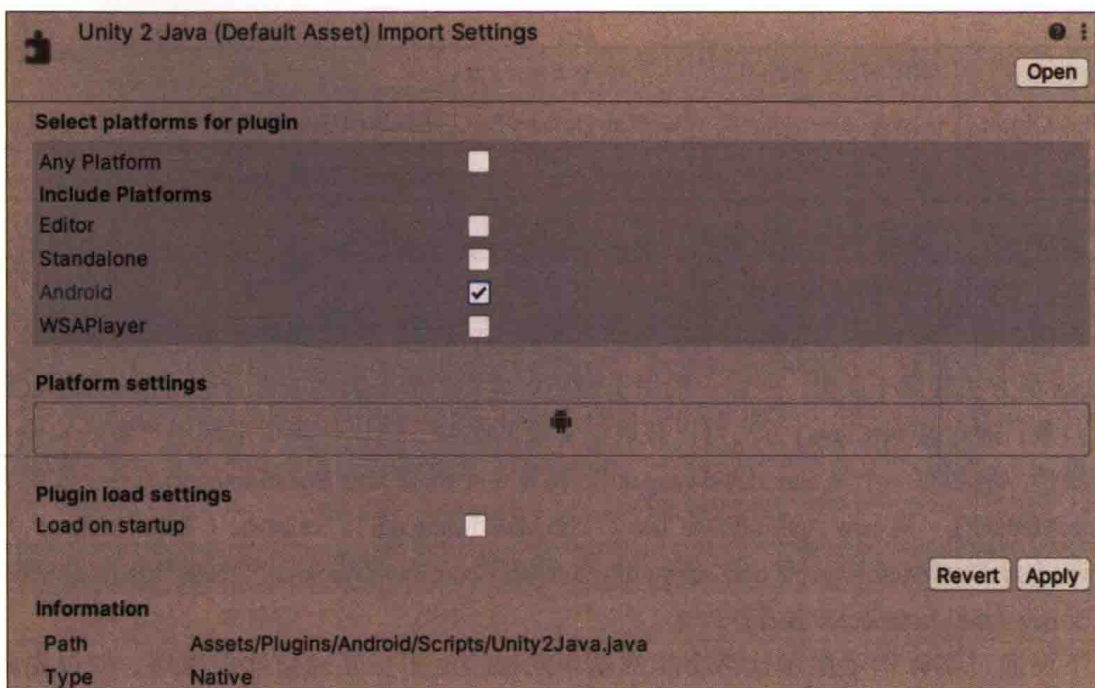


图 13-3 在 Java 文件导入设置中勾选 Android 复选框

为在 Unity 端调用 Java 代码，通过 Unity 创建 Android2Unity.cs 脚本文件，代码如下：

```
// 第 13 章 / 13-2
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class Android2Unity : MonoBehaviour
{
    void Start()
    {
        using (AndroidJavaClass Unity2JavaClass = new AndroidJavaClass("com.
example.davidwang.Unity2Java"))
        {
            Unity2JavaClass.CallStatic("StaticPrint", "Hello World from Android
static method ");
            int result1 = Unity2JavaClass.CallStatic<int>("StaticAdd", 1, 1);
            Debug.Log(" 结果 1: " + result1);

            Unity2JavaClass.Call("DynamicPrint", "Hello World from Android
dynamic method ");
            int result2 = Unity2JavaClass.Call<int>("DynamicAdd", 1, 2);
            Debug.Log(" 结果 2: " + result2);
        }
    }
}
```

```

using (AndroidJavaObject Unity2JavaObject = new AndroidJavaObject("com.
example.davidwang.Unity2Java"))
{
    Unity2JavaObject.CallStatic("StaticPrint", "Hello World from Android
static method ");
    int result3 = Unity2JavaObject.CallStatic<int>("StaticAdd", 1, 3);
    DeBug.Log(" 结果 3: " + result3);

    Unity2JavaObject.Call("DynamicPrint", "Hello World from Android
dynamic method ");
    int result4 = Unity2JavaObject.Call<int>("DynamicAdd", 1, 4);
    DeBug.Log(" 结果 4: " + result4);
}
}
}

```

在 Unity 中，将 Android2Unity 脚本挂载到场景中的任意对象上，连接手机，打包运行^①，输出结果如下：

行号	类型	输出信息
1	System.out	Hello World from Android static method
2	Unity	结果 1: 2
3	Unity	
4	Unity	结果 2: 0
5	System.out	Hello World from Android static method
6	Unity	结果 3: 4
7	System.out	Hello World from Android dynamic method
8	Unity	结果 4: 5

上述代码首先演示了 Java 端无返回值、有返回值方法的调用，通过泛型方法定义返回值类型获取 Java 端方法执行结果，由于数据类型的不同，Java 端与 C# 端交互支持的类型有 string、int、float、bool、AndroidJavaObject 共 5 类（也可以返回这些数据类型的数组）；其次演示了静态方法和实例方法的调用，类静态方法使用带 static 后缀的方法访问，而对象实例方法则使用不带 static 后缀的方法调用；再次演示了 AndroidJavaClass 和 AndroidJavaObject 类使用上的区别，通过输出结果，可以看到 AndroidJavaClass 调用对象实例方法既不报错，也不执行；AndroidJavaObject 类调用类静态方法可以正常执行^②，虽然在使用时都使用了 new 关键字，但 AndroidJavaClass 类不会生成实例对象，而 AndroidJavaObject 类会实例化对象。

类与实例的属性获取 / 设置与上述方法使用基本一致，通过这种方式，就可以直接在 C# 代码中调用 Java 端的原生类，代码如下^③：

① Java 代码编译后运行于 Dalvik / Art 虚拟机，其控制台输出不能输出到 Unity Debug 窗口，所以只能通过真机运行，并通过 Logcat 查看输出。

② Java 语言支持实例对象调用类静态方法或者获取类静态属性，这与 C# 语言不同。

③ 为了简化排版，后续 Java 代码与 C# 代码将放置于同一个代码片段中，并使用注释进行说明。

```

// 第13章/13-3
//Java 端代码
package com.example.davidwang;
import android.app.Activity;
import android.widget.Toast;

public class Unity2Java {
    public boolean ShowToast(Activity activity, String str){
        Toast.makeText(activity,str,Toast.LENGTH_SHORT).show();
        return true;
    }
}

//C# 端代码
// 获取设备 UUID
private string GetAndroidID()
{
    string androidID = "NONE";
    #if UNITY_ANDROID && !UNITY_EDITOR
        using (AndroidJavaObject contentResolver = new AndroidJavaClass("com.
unity3d.player.UnityPlayer").GetStatic<AndroidJavaObject>("currentActivity").Cal
l<AndroidJavaObject>("getContentResolver"))
        {
            using (AndroidJavaClass secure = new AndroidJavaClass("android.provider.
Settings$Secure"))
            {
                androidID = secure.CallStatic<string>("getString", contentResolver,
"android_id");
            }
        }
    #endif
    return androidID;
}

// 调用 Android 端 Toast
private void ShowToast()
{
    if (Application.platform == RuntimePlatform.Android)
        using (AndroidJavaObject Unity2JavaObject = new AndroidJavaObject("com.
example.davidwang.Unity2Java"))
        {
            using (AndroidJavaObject activity = new AndroidJavaClass("com.
unity3d.player.UnityPlayer").GetStatic<AndroidJavaObject>("currentActivity"))
            {

```



```

        bool isSuccess = Unity2JavaObject.Call<bool>("ShowToast",
activity, "From Unity");
        Debug.Log("ShowToast Status : " + isSuccess);
    }
}
}

```

在代码 13-3 中, 演示了两种直接调用 Android 端原生类的方式, 第一种是通过直接获取 Android 端的原生类, 调用其静态方法; 第二种是通过调用自定义的 Java 类间接调用 Android 原生类方法。同时, 由于 C# 代码运行平台不确定, 为确保代码兼容多平台, 也使用了两种判断代码执行平台的方法, 第一种使用预编译指令区分平台, 第二种通过 Application 类直接判断当前运行平台, 这也是在多平台开发中经常使用的技巧。

除此之外, 代码 13-3 还演示了获取当前活动 Activity 的方法, 即通过 com.unity3d.player.UnityPlayer 类获取当前 Activity, Android 很多类需要传递活动的 Activity 或者 Context 上下文对象, 通过这种方式获取当前 Activity 是一种常用方法。

提示

在开发中, 通常是在 Android Studio 中进行 Java 代码编写, 然后将代码复制到 Assets/Plugins/Android 目录中供 Unity 使用, 如果改动了 Android Studio 中的 Java 代码, 则需要再次复盖 Unity 中的对应文件, 而且同时存在两份一样的文件, 文件内容同步维护会是非常大的问题, 这时可以通过创建文件/文件夹链接确保只有一份文件。文件夹链接类似于 Windows 操作系统中的快捷方式, 但可供第三方软件使用。

Windows 下创建方式: `mklink /D <LINK_PATH><SOURCE_PATH>`

Linux 下创建方式: `ln -s <SOURCE_PATH><LINK_PATH>`

其中参数 LINK_PATH 为链接的路径, 这个路径无真实文件夹或者文件; SOURCE_PATH 为源文件/文件夹路径, 即真实的文件路径。通过创建文件/文件夹链接, 就可以只在 Assets/Plugins/Android 目录下保留一份代码文件, Android Studio 通过链接连接到相应的文件或文件夹, 从而可以保持文件同步, 无须进行复制代码文件的操作。

使用 AndroidJavaClass 和 AndroidJavaObject 类直接调用 Java 代码或 Android 原生类非常方便, 但只能单向由 C# 调用 Java 代码, Java 没办法反向调用 C# 代码, 实现反向调用则必须使用 AndroidJavaProxy 类, 正如其名, 这是个代理类, 负责在 Java 和 C# 代码之间桥接, 后续还会详细介绍该类。

如前文所述, Java 端与 C# 端交互支持的类型有 string、int、float、bool、AndroidJavaObject 共 5 类, AndroidJavaObject 可以表示所有对象类型, 因此, 除 string、int、float、bool 4 种基本类型, 其余对象都可由 AndroidJavaObject 表达。C# 端调用 Java 端的方法千差万别, 但 Java 端调用 C# 端只有以上 5 类 (如果包括数组, 则共 10 类), 因此可以编写一个通用的框架, 因为结构稍微有点复杂, 涉及 C# 端 AndroidCallbackManager.cs、AndoridCallbackInterface.cs 两个脚本文件,

Java 端 `CallUnityInterface.Java`、`Java2Unity.Java` 两个代码文件，其中，`AndroidCallbackManager.cs` 文件的代码如下：

```
// 第 13 章 / 13-4
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class AndroidCallbackManager : MonoBehaviour
{
    void Start()
    {
        using (AndroidJavaObject activity = new AndroidJavaClass("com.unity3d.
player.UnityPlayer").GetStatic<AndroidJavaObject>("currentActivity"))
        {
            using (AndroidJavaObject appController = new AndroidJavaObject("com.
example.davidwang.Java2Unity"))
            {
                AndoridCallbackInterface callback = new AndoridCallbackInterface
("com.example.davidwang.CallUnityInterface");
                callback.stringCallBack = StringProcess;
                callback.intCallBack = IntProcess;
                callback.floatArrayCallBack = FloatArrayProcess;
                appController.Call("Init", activity, callback);
                Debug.Log("In Start");
            }
        }
    }

    public void StringProcess(string str)
    {
        Debug.Log("string callback :"+str);
    }
    public void IntProcess(int value)
    {
        Debug.Log("int callback : " + value);
    }
    public void FloatArrayProcess(float[] arr)
    {
        foreach (var value in arr)
            Debug.Log("float in arr:" + value);
    }
}
```

在代码 13-4 中，`AndroidCallbackManager` 类是 Unity 端的使用类，是调用入口，其首先获取当前 Activity，实例化 Java 端的 `Java2Unity` 类，并且同时实例化了一个 C# 端的 `AndoridCallbackInterface` 类，然后设置 `AndoridCallbackInterface` 类的回调方法之后调用了 Java 端的初始化方法。

AndoridCallbackInterface.cs 文件的代码如下:

```
// 第 13 章 /13-5
using System;
using UnityEngine;

public class AndoridCallbackInterface : AndroidJavaProxy
{
    public Action<AndroidJavaObject> JavaObjectCallBack;
    public Action<bool> boolCallBack;
    public Action<string> stringCallBack;
    public Action<int> intCallBack;
    public Action<float> floatCallBack;
    public Action<float[]> floatArrayCallBack;
    // 构造方法
    public AndoridCallbackInterface(string interfaceName) : base(interfaceName)
    {
    }
    public void JavaObjectCallBack(AndroidJavaObject _data)
    {
        if (JavaObjectCallBack != null)
            JavaObjectCallBack(_data);
    }
    public void BoolCallBack(bool _data)
    {
        if (boolCallBack != null)
            boolCallBack(_data);
    }
    public void StringCallBack(string _data)
    {
        if (stringCallBack != null)
            stringCallBack(_data);
    }
    public void IntCallBack(int _data)
    {
        if (intCallBack != null)
            intCallBack(_data);
    }
    public void FloatCallBack(float _data)
    {
        if (floatCallBack != null)
            floatCallBack(_data);
    }
    public void FloatArrayCallBack(float[] _data)
    {
        if (floatArrayCallBack != null)
            floatArrayCallBack(_data);
    }
}
```


在代码 13-5 中, AndoridCallbackInterface 类继承自 AndroidJavaProxy 类, 定义了回调方法, 如前文所述, Java 端与 C# 端交互支持的类型共有 5 种, 这里为了通用将这 5 种回调方法都进行了演示 (还包括一个 Float 数组的演示方法)。该类构造方法很重要, 这里需要通过 AndroidJavaProxy 类将 C# 端实例与 Java 端的 CallUnityInterface 接口对应起来^①, 这样 Java 端就可以调用 C# 端实例的方法了。

CallUnityInterface.Java 文件的代码如下:

```
// 第 13 章 /13-6
package com.example.davidwang;

public interface CallUnityInterface {
    public void JavaObjectCallBack(Object _data);
    public void BoolCallBack(boolean _data);
    public void StringCallBack(String _data);
    public void IntCallBack(int _data);
    public void FloatCallBack(float _data);
    public void FloatArrayCallBack(float[] _data);
}
```

在代码 13-6 中, CallUnityInterface 接口方法签名与 AndoridCallbackInterface 类中对应方法签名需要完全一致, 这样才能确保正确地相互调用。

CallUnityInterface.Java 文件的代码如下:

```
// 第 13 章 /13-7
package com.example.davidwang;

import android.content.Context;
public class Java2Unity {
    private Context context = null;           // 上下文对象
    private CallUnityInterface callback = null; // 缓存回调
    public void Init(Context context , CallUnityInterface callback)
    {
        this.context = context;
        this.callback = callback;
        try {
            Java.lang.Thread.sleep(5000);
            Run();
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
```

^① 可以理解为建立了一个从 Java 端到 C# 端 AndoridCallbackInterface 类实例的指针, 这样 Java 端就可以通过这个指针访问 C# 端的实例了。

```

}
private void Run()
{
    int i = 100;
    String str = "From Java";
    float[] floatArr = {1.01f,1.02f,1.03f};
    this.callback.IntCallBack(i);
    this.callback.StringCallBack(str);
    this.callback.FloatArrayCallBack(floatArr);
}
}

```

在代码 13-7 中, Java2Unity 类首先通过 Init() 方法获取当前上下文对象^①和回调实例, 通过回调接口即可调用 C# 端的方法, 这里通过延时 5s 触发回调方法。

在这个通用框架中, 这 4 个类的相互关系如图 13-4 所示。

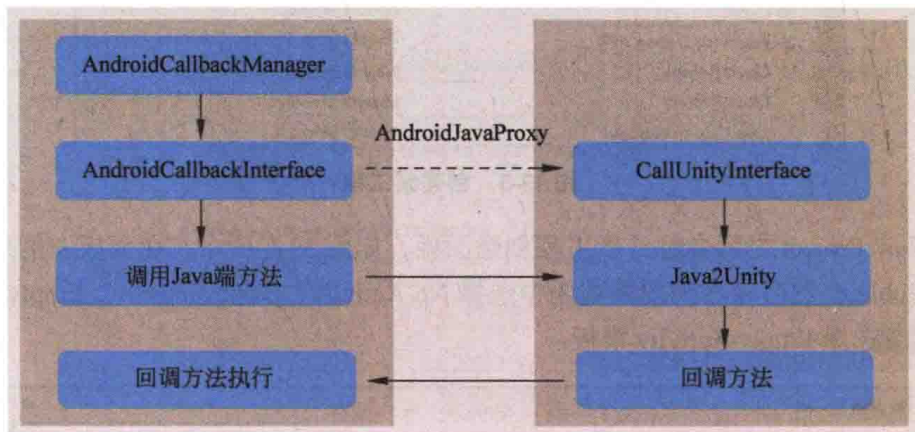


图 13-4 Java 端与 C# 端相互调用关系示意图

在 Unity 中, 将 AndroidCallbackManager 脚本挂载到场景中的任意对象上, 连接手机, 打包运行^②, 输出结果如下:

行号	类型	输出信息
1	Unity	int callback :100
2	Unity	string callback :From Java
3	Unity	float in arr:1.01
4	Unity	float in arr:1.02
5	Unity	float in arr:1.03
6	Unity	In Start

通过结果看到, Java 端正确地向 C# 端回调了相应方法并且参数传递无误, 而且也可以看

① 在本示例中, 上下文对象并没有使用, 但很多时间需要上下文对象或者当前 Activity。

② Java 代码编译后运行于 Dalvik / Art 虚拟机, 其控制台输出不能输出到 Unity Debug 窗口, 所以只能通过真机运行, 并通过 Logcat 查看输出。

到, 这种调用是同步的, 会阻塞当前线程。

13.3 模块调用

在实际使用中, 将 Android 端代码编译成 AAR 包供 Unity 端调用更有利于代码管理、人员分工、降低开发工具之间的耦合, 而且目前有大量第三方库通过 AAR 包的形式提供, 本节主要演示通过 Android Studio 生成 AAR 包及在 Unity 中调用 AAR 包中功能的流程及方法。

在 Android Studio 中生成 AAR 包的流程如下:

创建新 Android Studio 工程。在 Android Studio 中, 依次选择 File → New → New Project, 如图 13-5 所示。

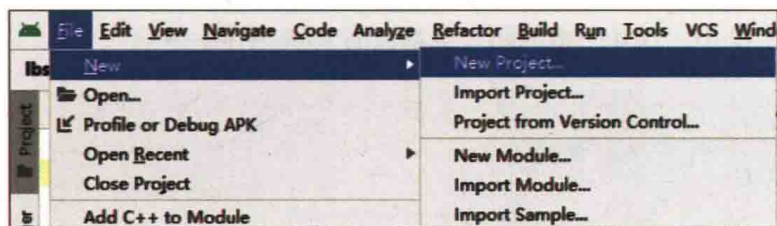


图 13-5 创建新工程

选择 New Project 之后会打开新工程创建面板, 如图 13-6 所示, 在面板左侧列表中选择 Phone and Tablet, 然后在右侧图像列表中选择 No Activity、Basic Activity、Empty Activity 模板之一, 本节选择 Empty Activity 模板。

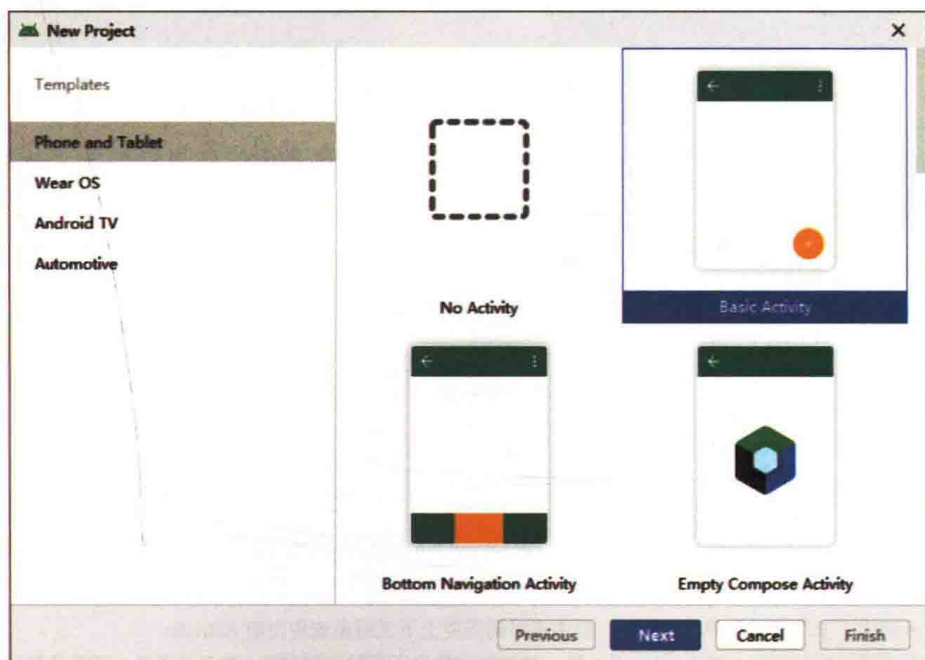


图 13-6 新工程创建面板

选择好所使用的工程模板后,单击 Next 按钮进入工程信息填写面板,如图 13-7 所示,其中 Package name 即为所使用的包名,需要认真填写;Minimum SDK 建议选择 Android 8.0 以上(ARCore 不支持 Android 7.0 以下版本);其余项可根据需要填写,单击 Finish 按钮完成工程创建。

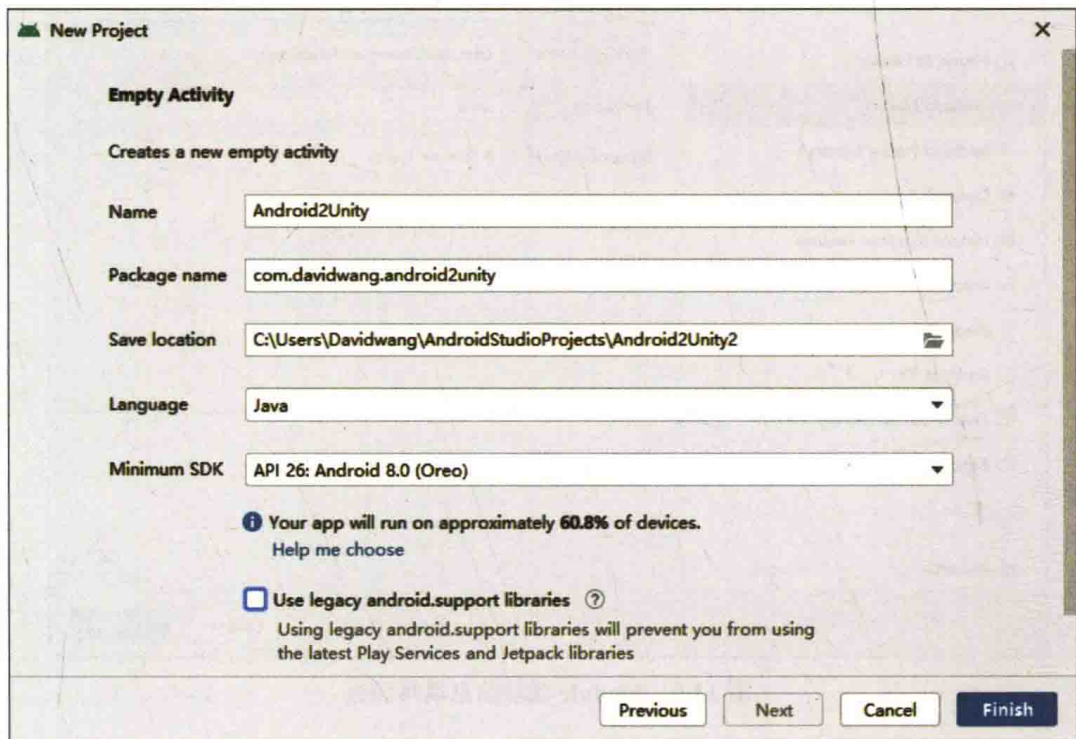


图 13-7 填写工程信息

工程创建完成后,因为希望以模块的形式导出 AAR 包,所以首先创建应用模块。在新创建的工程左侧 Project 列表面板 app 目录上右击,在弹出的级联菜单中依次选择 New → Module,如图 13-8 所示。

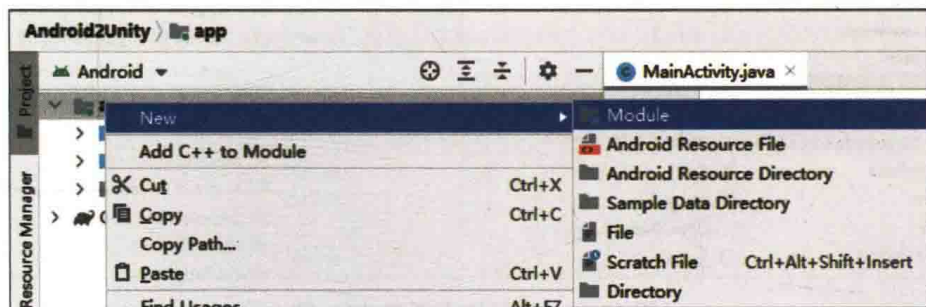


图 13-8 新建 Module 模块

在打开的新建模块面板中,因为希望创建 Android 类库,在其左侧列表中选择 Android

Library, 然后在右侧面板中填写 Module name (模块名) 与其他信息, 如图 13-9 所示, 最后单击 Finish 按钮完成 android2unity 模块的创建^①。

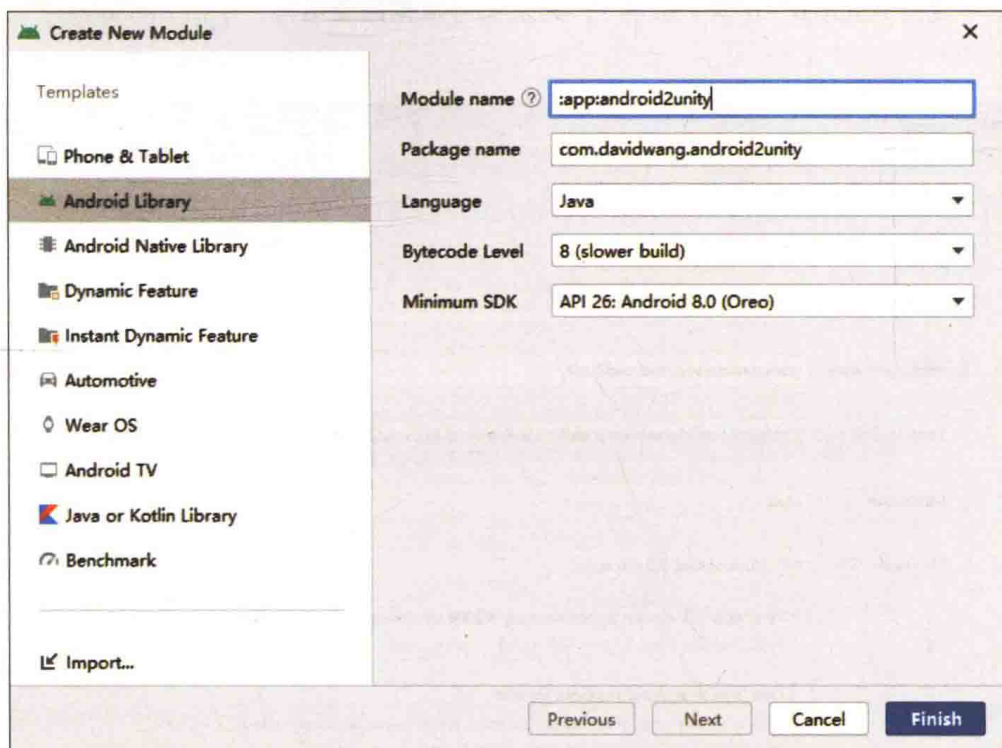


图 13-9 Module 模块信息填写面板

Android Studio 会自动生成模块的层级结构, 依次选择 android2unity 模块下的 Java → com.davidwang.android2unity 包名, 并在其上右击打开级联菜单, 如图 13-10 所示。

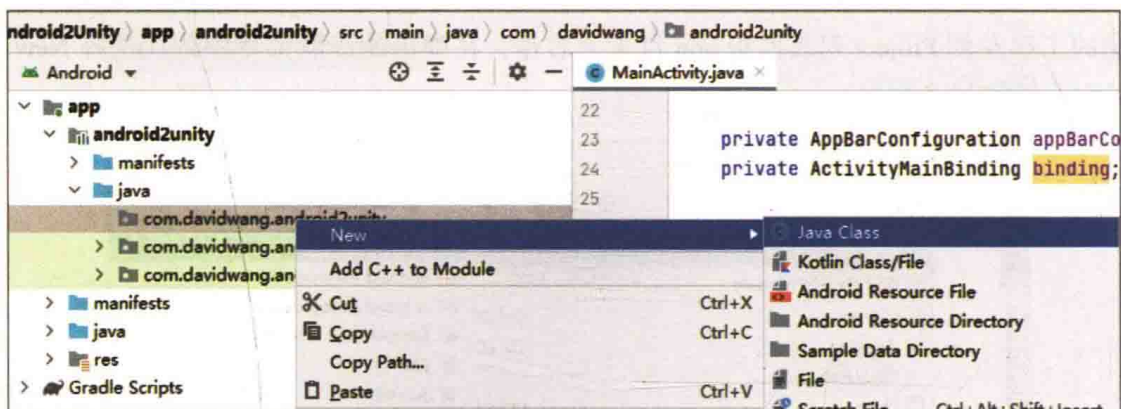


图 13-10 在模块下新建 Java 类

在弹出的级联菜单中依次选择 New → Java Class 创建新类, 本节类名命名为 example, 并

① 在本示例中, 工程名与模块名重名, 在实际使用中建议根据模块功能命名并避免与工程重名。

编写代码如下:

```
// 第 13 章 /13-8
//Java 端代码
package com.davidwang.android2unity;

import android.app.Activity;
import android.widget.Toast;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class example {
    private Activity _unityActivity;
    private String _unityObject;
    private String _unityMethod;
    // 初始化, 注册 Unity 对象与回调方法
    public void Init(String unityObject,String unityMethod){
        try {
            Class<?> classtype = Class.forName("com.unity3d.player.UnityPlayer");
            Activity activity = (Activity) classtype.getDeclaredField("currentAc
tivity").get(classtype);
            _unityActivity = activity;
        }
        catch (ClassNotFoundException e) { }
        catch (IllegalAccessException e) { }
        catch (NoSuchFieldException e) { }
        _unityObject = unityObject;
        _unityMethod = unityMethod;
    }
    // 通过 UnitySendMessage 方法传播消息
    private boolean callUnity(String args){
        try {
            Class<?> classtype = Class.forName("com.unity3d.player.UnityPlayer");
            Method method =classtype.getMethod("UnitySendMessage", String.
class,String.class,String.class);
            method.invoke(classtype,_unityObject,_unityMethod,args);
            return true;
        }
        catch (ClassNotFoundException e) {}
        catch (NoSuchMethodException e) {}
        catch (IllegalAccessException e) {}
        catch (InvocationTargetException e) {}
        return false;
    }
    // 调用 Android 端本地方法
    public boolean showToast(String content){
        Toast.makeText(_unityActivity,content,Toast.LENGTH_SHORT).show();
        callUnity("Call from android");
    }
}
```



```

        return true;
    }
}

```

在上述代码中, 采用 `UnitySendMessage()` 消息传播方式回调 Unity 端的方法, 由于 `UnitySendMessage()` 方法需要明确场景中游戏对象、游戏对象脚本中方法名作为参数, 为了方便 Unity 中的调用, 这里通过 `Init()` 方法由 Unity 端注入这两个参数, 从而不需要一一绑定游戏对象与脚本方法名。同时, 为避免引入 Unity 引擎的 `Classes.jar` 包, 这里采用了反射的方式获取当前 `Activity`。

至此, Android 端的模块已完成, 在左侧 Project 列表中选中 `android2unity` 模块, 然后在 Android Studio 菜单中依次选择 `Build` → `Rebuild Project` 开始构建模块 AAR 包, 如图 13-11 所示, 构建完成后, 可以在 `android2unity` 模块下的 `build` → `outputs` → `aar` 目录中找到生成的 AAR 包。

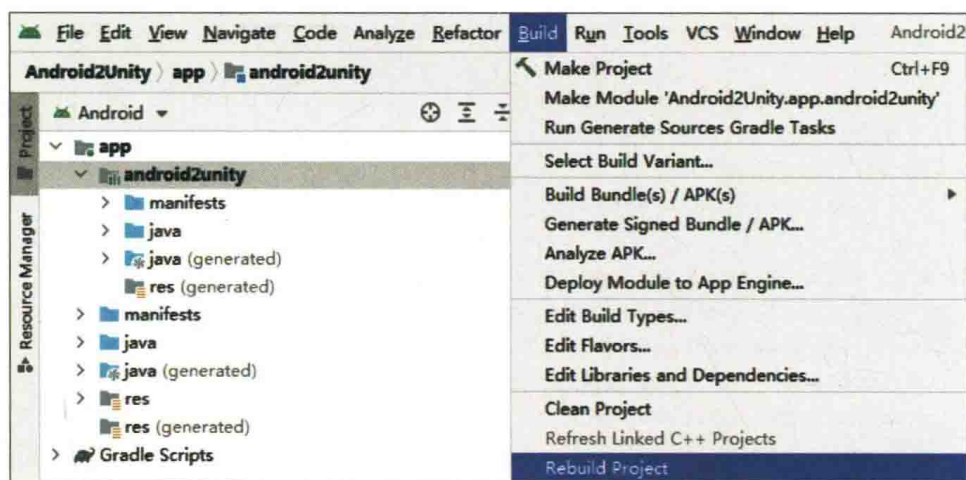


图 13-11 构建模块 AAR 包

将该 AAR 包文件复制到 Unity 工程 `Assets/Plugins/Android` 目录或其子目录下, 完成 AAR 包的导入。为测试其功能, 在 Unity 中新建 `Android2Unity.cs` 脚本文件, 代码如下:

```

// 第 13 章 / 13-9
// C# 端代码
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Android2Unity : MonoBehaviour
{
    void Start()
    {
        using(AndroidJavaObject jo = new AndroidJavaObject("com.davidwang.
        android2unity.example"))
        {

```

```

        jo.Call("Init",gameObject.name, "AndroidCallback");
        bool success = jo.Call<bool>("showToast", "Content from unity");
        if (true == success)
        {
            Debug.Log("Method executed");
        }
    }
}

//Android 端回调的方法
public void AndroidCallback(string str)
{
    Debug.Log(str);
}
}

```

该脚本逻辑比较简单，首先通过实例化 Android 端的类，并通过其 Init() 方法注入回调游戏对象名及脚本方法名。将该脚本文件挂载到场景中的任意对象上，连接手机，打包运行，可以看到 Unity 端的方法参数可以正确传递到 Android 端，Android 端也可以正确回调 Unity 端的脚本方法。

由于 UnitySendMessage() 方法需要遍历场景中的游戏对象，效率并不很高，特别是在频繁调用或者场景比较复杂的情况下可能会导致性能问题，在实际开发中，建议通过 AndroidJavaProxy 类交互。通过构建模块 AAR 包的形式，有利于代码管理，并且可以方便地整合第三方类库，提供一个整体的模块方案，简化 Unity 端的调用形式。

13.4 继承 UnityPlayerActivity 类

在一些深度交互场合，例如 Activity 切换、程序启动预处理等，这时可能会需要继承 Application 和 UnityPlayerActivity 类，本节将演示该功能。

本示例需要引入 UnityPlayer 类，该类位于 Unity 提供的 Classes.jar 包中^①，所以需要先引入该 JAR 包，首先直接复制 Classes.jar 包，然后在 Android Studio 工程 app 目录（本示例使用模块，所以将其放置到 android2unity 模块的 libs 目录下）的 libs 文件夹上右击，在弹出的菜单中选择 Paste 粘贴 Classes.jar 包，然后打开同级目录的 build.gradle 文件，在其 dependencies 配置中引入该 JAR 包，如图 13-12 所示。

由于只是引入其进行编译，保证编译正确进行，打包的时候并不需要 classes.jar 包，所以使用 compileOnly 指令确保该 JAR 包只用于编译^②。

① 该包的文件位置与 Unity 安装路径和版本有关，如笔者使用 Unity 2020.3.15，Unity 安装在 C 盘默认目录，其路径为 C:\Program Files\Unity\Hub\Editor\2020.3.15f1c1\Editor\Data\PlaybackEngines\AndroidPlayer\Variations\il2cpp\Release\Classes。

② 修改完 build.gradle 文件后不要忘记单击 Sync Now 按钮以使配置起作用。

```

43 dependencies {
44     implementation 'androidx.appcompat:appcompat:1.2.0'
45     implementation 'com.google.android.material:material:1.3.0'
46     testImplementation 'junit:junit:4.+'
47     androidTestImplementation 'androidx.test.ext:junit:1.1.2'
48     androidTestImplementation 'androidx.test.espresso:espresso-core:3.3.0'
49     //使用compileOnly指定该jar只是开发时引用,编译时不导出
50     compileOnly files('libs/classes.jar')
51 }

```

图 13-12 引入 classes.jar 包

在 Unity 2019 以后的版本中, UnityPlayerActivity 类不再位于 Classes.jar 包, 而是以独立的 UnityPlayerActivity.Java 文件存在^①, 将该文件复制到模块代码文件夹下。

继承 Application 类的示例代码如下:

```

// 第 13 章 /13-10
//Java 端代码
package com.davidwang.android2unity;
import android.app.Application;

public class CustomApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        System.out.println("自定义 Application");
    }
}

```

继承 UnityPlayerActivity 类的示例代码如下:

```

// 第 13 章 /13-11
//Java 端代码
package com.davidwang.android2unity;
import android.os.Bundle;

public class CustomActivity extends UnityPlayerActivity {
    @Override
    protected void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        System.out.println("自定义 Activity");
    }
}

```

正常导出 AAR 包, 将该 AAR 包文件复制到 Unity 工程 Assets/Plugins/Android 目录下。

^① 该 Java 文件位置与 Unity 安装路径和版本有关, 如笔者使用 Unity 2020.3.15, Unity 安装在 C 盘默认目录, 其路径为 C:\Program Files\ Unity\ Hub\ Editor\ 2020.3.15\Editor\ Data\ PlaybackEngines\ AndroidPlayer\ Source\ com\unity3d\ player。

由于继承了 Application 和 UnityPlayerActivity 类, 实际上是修改了应用程序的入口, 所以需要在 Unity 工程 Assets/Plugins/Android 目录下新建 AndroidManifest.xml 文件, 文件内容如下:

```
// 第 13 章 /13-12
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.davidwang.android2unity">
<application
    android:name="com.davidwang.android2unity.CustomApplication"
    android:label="@string/app_name">
<activity android:name="com.davidwang.android2unity.CustomActivity"
    android:label="@string/app_name"
    android:configChanges="fontScale|keyboard|keyboardHidden|locale|mn
c|mcc|navigation|orientation|screenLayout|screenSize|smallestScreenSize|uiMo
de|touchscreen">
<intent-filter>
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
<meta-data android:name="android.app.lib_name" android:value="unity" />
<meta-data android:name="unityplayer.ForwardNativeEventsToDalvik" android:value=
"false" />
</activity>
</application>
</manifest>
```

在 AndroidManifest.xml 文件中第一行定义了应用程序包名, 所以 Unity 工程中的包名必须与该包名一致, 然后将应用程序入口 (application 节) 设置为自定义的 CustomApplication, 将启动 Activity 设置为自定义的 CustomActivity, 并设置了相应的 intent-filter 过滤器。

因为直接继承了 Unity 的主 Activity, 所以不需要在 Unity 场景中做任何处理, 打包到真机运行, 可以看到正确的自定义内容输出。

提示

在导入该 AAR 包后, unity 打包时会引发配置错误, 原因是 AAR 包中 BuildConfig.class 类配置与 Unity 项目配置有冲突。将导入的 AAR 包名后缀改为 rar, 然后利用 rar 解压工具直接打开 (不是解压出来), 可以看到 classes.jar 包, 双击后在另一个解压界面中打开, 然后单击项目包名依次展开, 可以看到在工程/模块包名下有一个 BuildConfig.class 文件, 右击此文件后在弹出的菜单中选择删除, 关闭新打开的解压界面, 回到原解压界面时会弹出提示, 选择更新压缩文件, 完成包内文件的删除, 最后将 rar 后缀改为 aar 即可。

如果打包为 JAR 文件, 则处理方法类似, 但其包内没有 classes.jar 包, 可以直接单击模块包名依次展开, 删除 BuildConfig.class 文件, 后续处理方法与前文一致。

13.5 AndroidJavaProxy 类详细讲解

在前文中,已经使用了 AndroidJavaProxy 代理接口,本节将详细地介绍 AndroidJavaProxy 代理的用法。正如其名,AndroidJavaProxy 是一个代理,它在 Android 端代码与 Unity 端代码交互中起一个桥接作用。其一般用法为在 Java 代码中定义接口(Interface),建立代码调用外观,然后在 Unity 端用 C# 实现 Java 代码定义的接口,在使用时,在 C# 代码中实例化实现接口的类并将该实例对象传递到 Java 端,Java 端根据情况执行接口方法,回调 C# 中的实现逻辑。

下面通过一个实例进行演示。首先需要在 Java 端定义一个接口,ProxyExample.Java 文件的代码如下:

```
// 第13章/13-13
//Java 端代码
package com.davidwang.android2unity;

public interface ProxyExample
{
    void OnFired(String msg);
}
```

然后,在 C# 端,通过 AndroidJavaProxy 代理实现该接口,在使用时将实现接口的类的实例传递到 Java 端,代码如下:

```
// 第13章/13-14
//C# 端代码
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class UnityProxyExample : MonoBehaviour
{
    void Start()
    {
        using (AndroidJavaObject jo = new AndroidJavaObject("com.davidwang.
        android2unity.ProxyUseCase"))
        {
            var mCallbackListener = new SDKCallbackListener();
            // 调用 JAR 中的方法,并把 new 的值传进去
            jo.Call("Init", mCallbackListener);
        }
    }

    class SDKCallbackListener : AndroidJavaProxy
    {
        // 引用 JAR 中接口
```



```

    public SDKCallbackListener() : base("com.davidwang.android2unity.
ProxyExample") { }

    // 实现接口中的方法
    public void OnFired(string str)
    {
        Debug.Log("OnFired:" + str);
    }
}

```

在代码 13-14 中, C# 端实现了 Java 接口, 接口中的方法必须为 Public, 方法签名与 Java 端完全一致, 然后通过 Java 端的 Init() 方法将实现接口的实例对象传递到 Java 端, Java 端使用方法的代码如下:

```

// 第 13 章 / 13-15
//Java 端代码, ProxyUseCase.Java
package com.davidwang.android2unity;

public class ProxyUseCase {
    private ProxyExample proxyExample;
    public void Init(ProxyExample proxyExample){
        this.proxyExample = proxyExample;
        try {
            Java.lang.Thread.sleep(5000);
            EventCallback();
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
    public void EventCallback(){
        proxyExample.OnFired("some event fired in Android");
    }
}

```

在代码 13-15 中, EventCallback() 方法本应该由 Java 端的其他事件触发, 为了演示简单, 这里采用了延时触发方法。Java 端通过调用其接口方法, 实现了对 C# 端实例对象方法的调用。它们之间的关系示意图如图 13-13 所示。

在 C# 端, 通过实现 Java 端的接口, 可以生成本地端的实例对象, 这个实例对象在 Mono/IL2CPP 虚拟机内, 然后通过调用 Java 端的方法将本地端的实例对象引用传递到 Java 端, 因为实例对象实现了 Java 端的接口, 所以 Java 端可以通过接口调用实例对象的方法, 本质上是通过引用调用 Mono/IL2CPP 虚拟机端的本地实例对象, 所以 AndroidJavaProxy 代理实际上是在 C# 端与 Java 端建立了一个对象引用指针, 通过这个指针可以指向实例对象, 虽然这个实

例对象存在于另一个虚拟机中。

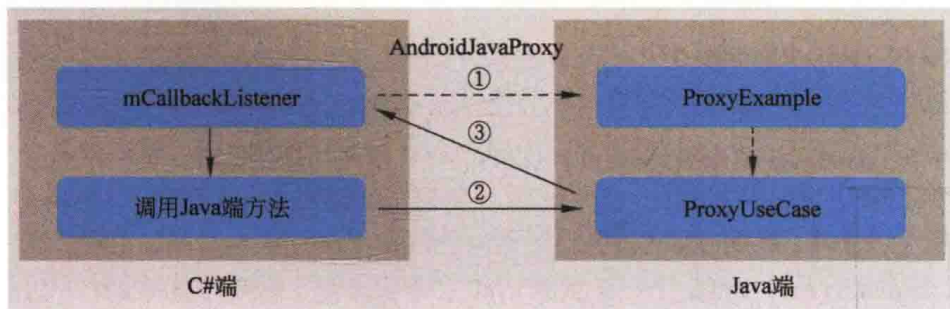


图 13-13 AndroidJavaProxy 原理示意图

本节，采用另一种打包方式，将代码打包成 JAR 包。JAR 包只包含编译后的 class 文件与清单文件，不包含图片布局等资源文件；AAR 包则包含所有相关的资源，class 文件、清单文件、res 资源文件等，所以如果只有类库文件时建议生成 JAR 包；如果还包含布局文件、图片、字体等资源文件，则建议使用 AAR 包。

在 Android Studio 中，类库的建立方法与 13.3 节所述一致，不再赘述。单击展开 app 目录下的 android2unity 目录，双击鼠标打开该目录下的 build.gradle 文件，如图 13-14 所示。

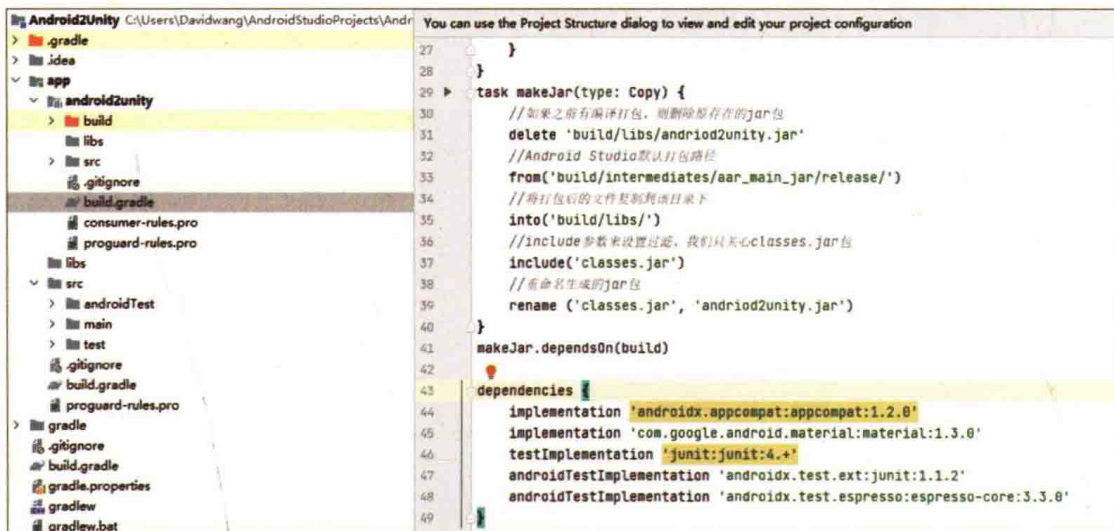


图 13-14 打开类库目录下的 build.gradle 文件

如图 13-14 所示，在打开的 build.gradle 文件的 dependencies 指令前添加一个 task^①，task 任务指令代码如下：

```
// 第 13 章 /13-16
task makeJar(type: Copy) {
    // 如果之前有编译打包，则删除原存在的 JAR 包
```

① Task 任务要作为一级指令，即不包含在其他编译指令之中。

```

delete 'build/libs/andriod2unity.jar'
//Android Studio 默认打包路径
from('build/intermediates/aar_main_jar/release/')
// 将打包后的文件复制到该目录下
into('build/libs/')
//include 参数设置过滤, 只关心 classes.jar 包
include('classes.jar')
// 重命名生成的 JAR 包
rename ('classes.jar', 'andriod2unity.jar')
}

makeJar.dependsOn(build)

```

编译 JAR 包时, 单击 task 任务左侧的绿色三角形图标展开下拉菜单, 选择 Run Android2Unity[makeJ... 开始编译打包, 如图 13-15 所示。

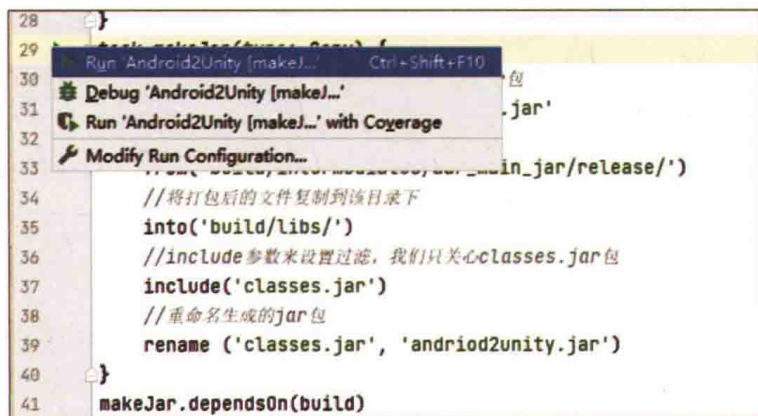


图 13-15 编译生成 JAR 包

编译打包成功后, 即可在 android2unity 类库模块下的 build/libs 目录下找到生成的 android2unity.jar 包^①。与使用 AAR 包一样, 将该 JAR 文件复制到 Unity 工程 Assets/Plugins/Android 目录或其子目录下, 然后在 Unity 端将 UnityProxyExample 脚本挂载到场景中的任意对象上, 连接手机, 打包运行, 即可看到正确的调用输出。

13.6 订阅 Android 广播

使用 AndroidJavaProxy 代理接口, 可以很方便地实现对 Android 端事件的订阅, 灵活地进行 Android 端与 Unity 端的代码交互。通过 AndroidJavaProxy 代理, 也可以在安卓端使用 BroadcastReceiver 接收广播, 然后利用自定义的方法传递给 C# 端, 但这种方式不太通用, 希

^① 根据 Android Studio 版本不同, 有的版本不会生成 libs 目录, 这时可以将 JAR 包生成到 outputs 目录中, 即代码清单 13-16 中将 JAR 包生成的路径修改到 build/outputs 目录下。通过 TASK 任务的方式还可同时生成 AAR 包, 该包区分 Debug、Release 版本, 位于 build/outputs/aar 目录下。

望能直接在 C# 端注册 Android 广播，下面演示一个通用的广播接收器。

其原理依然是利用 `AndroidJavaProxy` 接口进行桥接，但通过 Unity 端注入 `action` (Android 端广播动作名称) 参数，即由 C# 端控制需要注册的广播，Android 端接收到广播之后，进行相应的数据处理 (方便将这些数据传递到 C# 端)，然后调用接口方法，触发 C# 端广播处理方法。首先看一下调用逻辑，代码如下：

```
// 第 13 章 /13-17
//C# 端代码
using System.Collections.Generic;
using UnityEngine;

public class BroadcastExample : MonoBehaviour, UnityBroadcastHelper.IBroadcastListener
{
    UnityBroadcastHelper helper;
    void Start()
    {
        helper = UnityBroadcastHelper.Register( new string[] { "some_action_
name" }, this);
    }
    public void OnReceive(string action, Dictionary<string, string> dictionary)
    {
        // 广播处理逻辑
    }
    void OnDisable()
    {
        helper.Stop();
    }
}
```

通过代码 13-17 可以看到广播注册界面非常简洁，脚本实现了 `UnityBroadcastHelper.IBroadcastListener` 接口，在 `Start()` 方法中初始化了一个 `UnityBroadcastHelper` 对象，然后通过 `OnReceive()` 方法等待 Android 端回调并进行相应处理即可，使用完后，通过 `OnDisable()` 方法停止广播监听。

C# 端 `UnityBroadcastHelper` 类的代码如下：

```
// 第 13 章 /13-18
//C# 端代码
using System.Collections.Generic;
using UnityEngine;

public class UnityBroadcastHelper
{
    //Java 端对象
    private readonly AndroidJavaObject JavaObject;
    // 构造函数，生成唯一的 Java 端对象实例
    private UnityBroadcastHelper(string[] actions, IBroadcastListener listener)
    {

```



```

        ListenerAdapter adapter = new ListenerAdapter(listener, this);
        JavaObject = new AndroidJavaObject("com.example.davidwang.UnityBroadcastHelper",
actions, adapter);
    }

    // 注册广播
    public static UnityBroadcastHelper Register(string[] actions, IBroadcastListener
listener)
    {
        return new UnityBroadcastHelper(actions, listener);
    }

    // 取消广播订阅
    public void Stop()
    {
        JavaObject.Call("stop");
    }

    // 广播订阅者接口
    public interface IBroadcastListener
    {
        void OnReceive(string action, Dictionary<string, string> dictionary);
    }

    // AndroidJavaProxy 实现桥接, 处理数据并触发订阅方法
    private class ListenerAdapter : AndroidJavaProxy
    {
        readonly IBroadcastListener listener;
        readonly UnityBroadcastHelper helper;
        public ListenerAdapter(IBroadcastListener listener, UnityBroadcastHelper
helper) : base("com.example.davidwang.UnityBroadcastHelper$BroadcastListener")
        {
            this.listener = listener;
            this.helper = helper;
        }

        void onReceive(string action)
        {
            AndroidJavaObject JavaObject = helper.JavaObject;
            if (!JavaObject.Call<bool>("hasKeyValue"))
            {
                return;
            }

            string[] keys = JavaObject.Call<string[]>("getKeys");
            string[] values = JavaObject.Call<string[]>("getValues");
            JavaObject.Call("pop");
            Dictionary<string, string> dictionary = new Dictionary<string, string>();
            int n = keys.Length;
            for (int i = 0; i < n; i++)
            {
                dictionary[keys[i]] = values[i];
            }
        }
    }

```

```

        }
        listener.OnReceive(action, dictionary);
    }
}
}

```

UnityBroadcastHelper 类是接口的辅助类，其内部定义了 IBroadcastListener 接口及桥接类 ListenerAdapter，并定义了注册与取消注册方法，从而简化调用端的使用。

Java 端的代码如下：

```

// 第 13 章 /13-19
//Java 端代码
package com.example.davidwang;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;
import Java.util.LinkedList;
import Java.util.Queue;
// 引入 UnityPlayer 类
import com.unity3d.player.UnityPlayer;

public class UnityBroadcastHelper {
    private static final String TAG = "UnityBroadcastHelper";
    // 接口定义
    public interface BroadcastListener {
        void onReceive(String action);
    }
    private final BroadcastListener listener;
    private Queue<String[]> keysQueue = new LinkedList<>();
    private Queue<String[]> valuesQueue = new LinkedList<>();
    // 构造函数，注入 Unity 端广播订阅者并注册广播
    public UnityBroadcastHelper(String[] actions, BroadcastListener listener) {
        this.listener = listener;
        IntentFilter intentFilter = new IntentFilter();
        for (String action : actions) {
            intentFilter.addAction(action);
        }
        Context context = UnityPlayer.currentActivity;
        if (context == null) {
            return;
        }
        context.registerReceiver(broadcastReceiver, intentFilter);
    }
}

```

```

public boolean hasKeyValue() {
    return !keysQueue.isEmpty();
}

// 返回键
public String[] getKeys() {
    return keysQueue.peek();
}

// 返回值
public String[] getValues() {
    return valuesQueue.peek();
}

public void pop() {
    keysQueue.poll();
    valuesQueue.poll();
}

// 取消广播注册
public void stop() {
    Context context = UnityPlayer.currentActivity;
    if (context == null) {
        return;
    }
    context.unregisterReceiver(broadcastReceiver);
}

// 广播处理并回调 Unity 中的 OnReceive() 方法
private BroadcastReceiver broadcastReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        MyLog.d(TAG, "UnityBroadcastHelper: action: " + action);
        Bundle bundle = intent.getExtras();
        if (bundle == null) {
            bundle = new Bundle();
        }
        int n = bundle.size();
        String[] keys = new String[n];
        String[] values = new String[n];
        int i = 0;
        for (String key : bundle.keySet()) {
            keys[i] = key;
            Object value = bundle.get(key);
            values[i] = value != null ? value.toString() : null;
            i++;
        }
        keysQueue.offer(keys);
        valuesQueue.offer(values);
        listener.onReceive(action);
    }
}

```



```
};  
}
```

UnityBroadcastHelper 类中的 BroadcastListener 接口即为需要桥接的接口，在该类的构造函数中传入了 C# 端定义的 action 及桥接实例对象并进行了广播注册操作，在 stop() 方法中取消了广播注册，UnityBroadcastHelper 通过持有 BroadcastReceiver，将广播注册到 Context 上下文中。为在两端进行数据传递，将 Extra 的 Bundle 数据映射为 C# 中的 Dictionary，传递给 OnReceive() 方法，方便 C# 代码使用。为了演示简单，这里将所有类型的数据都映射成 string 类型，在实际使用时可根据需要进行处理，如通过 AndroidJavaObject 进行复杂类型数据传递。

提示

本示例也需要引入 UnityPlayer 类，该类位于 Unity 提供的 Classes.jar 包，引入 JAR 包流程和处理与 13.5 节一样。

通过该通用类，不仅可以注册自定义广播，还可以注册系统广播（可能需要权限申请），以及一次性注册多个广播，界面简洁，使用简单。

13.7 调用 so 库文件

so 库文件是 Linux 系统下的程序函数库，即编译好的可以供其他程序使用的代码和数据，so 库文件也称为共享库文件，由 C 或 C++ 代码编译生成，由于 C 和 C++ 代码执行效率高，通常计算密集型算法通过 C 或者 C++ 编译能获得可观的性能提升。Android 系统源于 Linux 系统，因此 Android 系统自然也支持 so 库文件，但需要注意的是，Android 系统运行的硬件平台通常是 ARM 或者 ARM64 架构芯片，因此需要使用编译到这两个平台的 so 库文件。本节学习在 Unity 中调用 so 库文件的方法。

编写 C 或者 C++ 头文件，代码如下^①：

```
// 第 13 章 / 13-20  
//NativeCodeExample.h 文件  
#ifndef __NativeCodeExample_H__  
#define __NativeCodeExample_H__  
  
#if 0  
// 导出 dll 声明  
#define EXPORT_DLL __declspec(dllexport)  
#else  
#define EXPORT_DLL  
#endif
```

^① 通常 C 或者 C++ 代码由第三方或者其他工程师编写。

```
extern "C" {
    EXPORT_DLL int AddFunc(int _a, int _b);
    EXPORT_DLL char* GetCharArr();
}
#endif
```

编写 C 或者 C++ 代码，文件如下：

```
// 第 13 章 /13-21
//NativeCodeExample.cpp 文件
#include "NativeCodeExample.h"

extern "C" {
    int AddFunc(int _a, int _b)
    {
        return _a + _b;
    }

    char* GetCharArr() {
        return new char[11]{ 'H','e','l','l','o','W','o','r','l','d' };
    }
}
```

将这两个源文件放置到 Android Studio 工程里，通过编写相应编译指令，利用 NDK 即可生成 ARM 和 ARM64 使用的 so 文件^①（分别放置在 armeabi-v7a、arm64-v8a 等文件夹下，文件夹名对应平台架构，根据 Unity 中的设置，App 应用运行时会根据运行时的 CPU 架构选择相应的库文件，依据配置指令的不同，生成的 so 文件平台也不同）。将生成的 so 文件（连同文件夹，如 armeabi-v7a、arm64-v8a）一并复制到 Unity 工程 Assets/Plugins/Android 目录下，so 库文件导入完成。

在 Unity 工程中新建脚本 UnitysoUseCase.cs 文件，代码如下：

```
// 第 13 章 /13-22
//C# 文件
using System;
using UnityEngine;
using System.Runtime.InteropServices;

public class UnitysoUseCase : MonoBehaviour
{
    [DllImport("NativeCodeExample")]
    public static extern int AddFunc(int x, int y);
    [DllImport("NativeCodeExample")]
    static public extern IntPtr GetCharArr();
}
```

① 因为 so 库文件通常由第三方提供，本节不具体讲述 so 库文件的生成过程。


```

void Start()
{
    int sum = AddFunc(100, 11);
    string str = Marshal.PtrToStringAnsi(GetCharArr());
    Debug.Log("Integer from native code:" + sum);
    Debug.Log("String from native code:" + str);
}
}

```

然后将 UnitysoUseCase 脚本挂载到场景中的任意对象上，连接手机，打包运行，即可看到正确的调用输出。

Unity 端 C# 代码调用 so 库文件中方法时首先通过 DllImport 属性指定方法所在程序集，程序集既没有 lib 前缀，也没有 .so 后缀，然后声明方法，因为方法原自于外部库文件，所以所有的方法都需要使用 extern 修饰符，另外需要注意的是，so 库方法在原生非托管堆中执行，需要通过 Marshal 类在托管与非托管代码间进行转换。Marshal 类可以在两个不同的实体（两个线程或者进程、托管和非托管之间）间进行方法调用和参数传递的转换，能将指针、引用、委托等在托管与非托管代码间进行合适的转换，并提供了很多方法用于分配非托管内存、复制非托管内存块、将托管类型转换为非托管类型，还提供了在与非托管代码交互时使用的其他杂项方法。

13.8 Android 与 Unity 交互注意事项

如前文所述，Android 与 Unity 交互通信实际上是两个虚拟机之间的通信，这是一个资源消耗非常大的操作，特别是在涉及上下文切换的场景，性能开销更大，而且涉及两端执行，资源管理难度大，所以，在设计交互通信功能时，要充分考虑到这点并注意以下事项：

(1) 理解调用原理。C# 端调用 Java 端代码的调用时序图如图 13-16 所示。

从图 13-16 可以看出，整个流程可分为几个阶段：第一，通过 GetMethodId() 找到目标方法对应的实际内存地址；第二，创建调用入参，同时处理 AndroidJavaObject、AndroidJavaProxy 等特殊类型的参数；第三，通过方法实际内存地址调用目标方法，获取结果。

整个流程使用了反射、遍历等操作，是一个漫长的操作过程，因此频繁地调用会带来性能上的问题。从图中也可以看到，在第一次实际调用之后，由于已经获取了目标方法的实际内存地址，后续调用时可以跨过这个阶段，因此后续调用相对开销要稍小一些。

(2) 使用 using() 语句确保资源及时释放。Android 与 Unity 交互通信在两个虚拟机中运行，对实例对象的管理很难做到用完即释放，所以最合适的方式是将实例对象的使用通过 using() 语句块包含起来，这能确保使用完后资源被及时释放，方便 GC 回收内存，本章所有示例都遵循了该原则。

(3) 使用单例模式确保只生成一个实例对象。因为对象的生成与对象的调用在不同端，为最小化资源开销，通常通过单例模式确保只生成一个实例对象，示例代码如下：

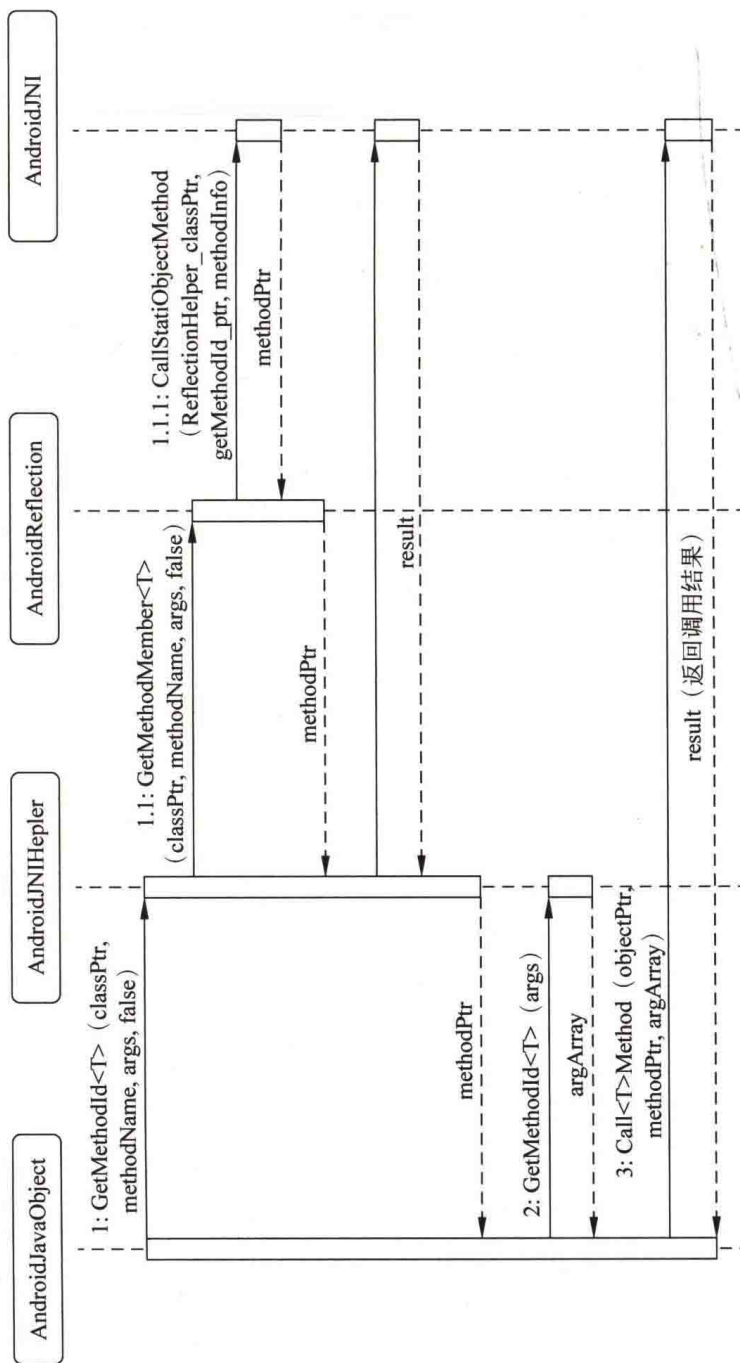


图 13-16 C# 端调用 Java 代码的整体时序图

```

// 第13章 /13-23
//Java 端代码
package com.example.davidwang;

public class Unity2Java {
    private static Unity2Java _instance;
    public static Unity2Java instance()
    {
        if(_instance == null )
            _instance = new Unity2Java();
        return _instance;
    }
    public void DynamicPrint(String str){
        System.out.println(str);
    }
    public int DynamicAdd(int a,int b)
    {
        return a+b;
    }
}

//C# 端代码
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Android2Unity : MonoBehaviour
{
    void Start()
    {
        using (AndroidJavaClass Unity2JavaClass = new AndroidJavaClass
("com.example.davidwang.Unity2Java"))
        {
            using(AndroidJavaObject Unity2JavaObject =
Unity2JavaClass.CallStatic<AndroidJavaObject>("instance"))
            {
                Unity2JavaObject.Call("DynamicPrint", "Hello World from Android
dynamic method ");
                int result = Unity2JavaObject.Call<int>("DynamicAdd", 1, 1);
                DeBug.Log(" 结果: " + result);
            }
        }
    }
}

```

(4) 关注对象生命周期。由于代码调用双方分属不同的虚拟机，它们有完全不同的生命周期，因此，在交互时需要充分关注对象的生命周期情况，错误的调用时机可能会导致应用

崩溃。

Android 与 Unity 交互通信涉及不同的工具链,还可能涉及与其他语言编写的库通信(如 .so 库),对知识面宽度要求较高,本章演示了各种常见的交互方法,示例经过精心设计,基本覆盖了 Android 与 Unity 交互通信的常见方式,通过本章的学习,相信读者已经对 Android 与 Unity 交互通信有了较深入的理解,并能解决大部分交互相关问题,包括通过说明文档引入第三方库。



AR 是一种全新的应用形式，不同于以往任何一种通过矩形框（电视机、智能手机、屏幕）来消费的视觉内容，AR 是一个完全没有形状束缚的媒介，环境显示区域完全由用户自行控制，这是一种新奇的体验方式，但也给传统操作习惯带来了极大的挑战。本章通过 AR 技术特性、AR 应用设计原则探讨提升 AR 应用用户体验的相关话题。

14.1 移动 AR 带来的挑战

在 AR 眼镜成熟并且价格降到普通消费者可承受之前，基于移动手机和平板计算机的 AR 技术仍会是 AR 应用的主流，也是在元宇宙发展浪潮的初级阶段最有希望率先普及的技术。得益于 Apple ARKit、Google ARCore、华为 AREngine 等 SLAM 引擎，开发 AR 应用的技术门槛也越来越低，这对 AR 应用的普及无疑至关重要，但也要看到 AR 应用在带来更强烈视觉刺激的同时，对普通用户长期以来形成的移动设备操作习惯也形成一种挑战。因为 AR 应用与普通应用无论在操作方式、视觉体验还是功能特性方面都有很大的区别，目前用户形成的某些移动设备操作习惯无法适应 AR 应用，另外还需要扩展很多特有的操作方式，例如用户在空间手部动作的识别、凌空对虚拟物体的操控等，这些都是传统移动设备操作中没有的，并且更关键的是目前智能设备操作系统并不是为 AR 应用而开发的，因此从底层架构上就没有彻底地支持 AR，这对开发人员来讲构成了一个比较大的挑战，所以，在设计 AR 应用时就应当充分考虑这方面的需求，用一种合适的方式引导用户采用全新操作手段去探索 AR 世界，而不应在这方面让用户感到困惑。不仅如此，对习惯于传统应用开发的设计人员而言，这些挑战还包括以下几个方面。

14.1.1 三维化思维

相比于传统 PC、手机应用，AR 应用最大的区别是显示方式的变革。传统应用所有的图像、文字、视频等信息均排布在一个二维屏幕内，人与机器存在天然的隔离，WIMP（Window、Icon、Menu、Pointer）是其典型的操作方式，而 AR 应用则完全打破了传统屏幕信息显示模

式，显示内容彻底摆脱了屏幕的限制，并与真实的现实世界融为一体，人机交互也向着更自然、更直观的三维形式演进，因此，设计 AR 应用首先需要将思维从传统的二维模式转换到三维模式，在设计之初就应当充分考虑应用的交互空间（包括空间大小、平面类型、遮挡关系）、输入方式（语音、手势、控制器）、信息呈现方式（三维模型、二维界面、UX 元素）、音效（空间音效、界面反馈音效、背景音效）、融合方式（叠加、遮挡、跟随）等。

三维思维应该贯穿整个应用开发周期，而不仅局限于界面设计，因为世界本身就是三维的，AR 应用是将人也置于应用之中，而不再将其当成旁观者，这就需要在 AR 应用开发中跳出传统应用设计的二维思维模式，以一种全新的视角重新审视 AR 应用开发设计。

14.1.2 用户必须移动

由于 AR 应用的技术特性，要求用户要么站起来移动、要么举着手机来回扫描以获得更好的体验。这是与传统移动设备操作最大的不同，在 AR 之前，从来不会有应用要求用户站起来操作，这种操作在带来用户新奇感的同时也会降低用户使用操作时间，由于生物力学原因，没有人愿意为了操作某个 AR 应用而长时间站立。或许在使用初期大家还能忍受为这种新奇而带来的身体不适，但作为一个 AR 应用而言，很难要求用户在长时间使用与长时间移动中取得平衡，而这在使用手机作为 AR 应用硬件载体时表现得尤为突出。

因此，就目前而言，这也是应用开发者在开发 AR 应用时首先要考虑的情况，如果开发的应用持续时间较长那就应该采取合适的方法来降低因为要求用户移动而带来的用户流失风险。

14.1.3 要求手持设备

就目前 AR 主要使用的手机和平板计算机设备而言，运行 AR 应用后要求用户一直手持设备而不能将设备放置在一旁，如图 14-1 所示，这也是对长时间持续使用 AR 应用的一个考验。用户至少需要使用一只手来握持手机或平板以使用 AR，这是对用户的一种束缚，这意味着很多看起来很美好的东西其实并不适合做成 AR。

要求手持设备带的另一个问题是会加速用户的疲劳感，在用手机等移动设备看电影时，通常会找一个支架或者将移动设备放置在一个让自己感觉很舒适的地方，而这对 AR 应用并不适用，AR 应用启动后如果将移动设备长期放置在一个地方，则 AR 的所有优势将全部转化为劣势。

另一方面，要求用户手持设备至少会占用用户的一只手（除非借助于专门的支架设备），导致用户在进行其他操作时的便捷度大大降低。例如，修理师傅在使用一个演示发动机构造的 AR 应用修理发动机时，由于要用一只手一直握持设备，从而导致其维修的灵活性与效率下降。



图 14-1 用户操作 AR 应用示意图

14.1.4 近脸使用

移动手机 AR 应用终归还是手机应用，需要将手机放在脸前。如果是短时的体验，这并没有不妥，但从长远来看，将手机时刻保持在脸前可能会让人感到厌烦，特别是当用户还处在移动中，如在街上行走。

其他非 AR 应用也要求用户将手机放在脸前，但是非 AR 应用不要求用户必须移动或者手持。考虑这个因素，如移动实景导航类应用就需要格外关注，将手机时刻保持在脸前不仅会导致用户疲劳，还会增加现实环境中的安全风险。同时将手机放在用户脸前也意味着用户使用 AR 的 FoV (Field of View, 视场角) 非常有限，这会降低 AR 的整体沉浸感。

14.1.5 操作手势

AR 应用中，用户往往不再满足于屏幕上的手势操作而希望采用空间上的手势操作，在二维的屏幕上操作三维空间中的对象，这本身就违背了人的操作本能，这不仅会导致沉浸体验的降低，也会带来操作上的困难。不过就目前来讲，移动端还没有非常成熟的空间手势操作解决方案，虽然也有一些适用于移动设备的手势操作解决方案出现，但受限于移动终端资源和性能，并未大规模普及，可穿戴肢体动作捕捉设备及灵巧控制器的出现可能会对操作方式中面临的困难起到一定的缓解。

提示

屏幕手势操作就是目前常用的在手机屏幕上单击、滑动、双击等操作方式，这是用户已经习惯的操作方式，也有非常成熟的手势操作识别解决方案。空间手势操作是指用户在空中而非在手机屏幕上的操作，也叫虚拟三维手势操作。当用户手部在摄像头前时，算法可以捕获识别用户手掌和手指的变化，并且通过采用计算机视觉分析算法，过滤掉除手掌和手指外的其他轮廓，然后将此转换为手势识别模型，并进一步转换成操作指令。

14.1.6 启动应用

所有移动 App 应用都面临这个问题，如果不打开相应的 App，就不能进行相应操作，但对 AR 应用来讲，这个问题变得更加严重。对用户而言，最自然的方式是在需要的时候弹出、显示、提示相应的信息，而用户不用去关注其细节，使用以手机为主的移动 AR 显然还做不到，但这个问题归根结底还是没有专门针对 AR 的硬件平台和操作系统，现行的移动操作系统都源自 PC 操作系统，已经不能适应 AR、MR 技术的发展需求。

必须打开 App 与人的本性形成了相对的冲突，人在做某件事情的时候通常是功利性的，在使用手机端普通 App 应用时，弹出的提示、新闻的阅读通过轻轻一点就能返回用户满意的结果，而 AR 则不然。如果使用 AR 像使用摄像头拍照那样，需要用户启动一个 App 但却不总是能给用户以满足，那启动关闭 AR 应用将是致命的，因为对于基于智能手机的 AR，让用户决定将放在口袋里的手机取出并使用 AR 体验，这本身就是一个挑战，而且目前来看，AR 还不是一种必需品。

14.1.7 色彩运用

由于屏幕显示的一致性，在传统应用配色上，只要符合一定的设计准则，即可提供视觉效果良好的用户界面，而 AR 应用则有很大的不同，其应用背景是现实环境，AR 应用中显示的内容直接叠加到现实环境色彩之上，因此存在一个色彩融合的问题。测试结果显示，高饱和度的颜色在适应环境的过程中辨识度更高，纯饱和度的颜色由于只包含单一波长，因此在 AR 应用中所呈现的视觉效果更明亮；深色颜色和现实场景融合度更高，甚至能够融为一体，白色与环境区分度最高，但大面积使用白色，用户在长时间使用时会产生视觉疲劳，也会对眼睛视力造成一定的损害。

通常在 AR 应用中，高饱和度的蓝色、绿色、紫色、橙黄色或白色效果更好，不建议在应用中过多地使用红色，红色具有警示效果，容易让人产生误解。另外，在使用 AR 应用时，呈现效果在强背景光的情况下很不理想，即使高饱和度的色彩也很难在高强度光线下完美呈现，如果 AR 应用设计为室外环境使用，应当考虑此因素影响。

14.2 移动 AR 设计准则

在 14.1 节中，我们了解了当前移动 AR 在使用上对用户操作习惯形成的巨大挑战，对用户而言，如果一个应用不是必不可少但却要求用户付出额外的操作或者让用户感到不适，则用户很可能就会放弃该应用而选择替代方案，所以在设计 AR 应用时有些准则需要遵循以使开发的应用更有生命力。

14.2.1 有用或有趣

正如前文所述,说服一个人取出手机并启动 AR 应用本身就是非常大的挑战。要触发此类行为,则该应用必须具备有用或有趣的内容来吸引用户。将一个三维模型放在 AR 平面上确实很酷绚,但这并不能每次都能达到触发用户启用 AR 应用的开启线,所以从长远来看,用户真正愿意用的 AR 必须有用或者有趣,如在教学中引入 AR 能强化学生对知识的直观体验,能加深学生对事物的理解;或者如 Pokemon Go 能做得非常有趣,让人眼前一亮。

所以 AR 应用开发创意是击败上述 AR 对用户操作带来挑战的有效手段,一个真正解决用户痛点的 AR 应用会破解所有不利因素的影响,因此,对 AR 开发设计人员来讲,这是首先要遵循的准则,否则可能花费很长时间做出来的应用无人问津。

14.2.2 虚实融合必须有意义

AR 字面意思也是增强现实,但只有将虚拟元素与真实环境融合时,增强现实才有意义,并且这种混合场景在单独的现实或单独的虚拟中都不能全部体现,也就是将虚拟对象附加到现实环境中后具有附加价值。如果附加上的虚拟对象对整体来讲没有附加价值,则这个附加就是失败的。例如将一个虚拟电视挂在墙上,透过智能手机的屏幕看播放的电影绝对没有附加价值,因为通过这种方式看电影不仅没有增加观影体验而且还会因为手机视场过小而感觉很不舒适,结果还不如直接看电视或者在手机上播放二维电影更有意义。

但如果开发一个食品检测 AR,能够标注一款食品的卡路里或者各成分含量,在扫描到一个面包后就能在屏幕上显示出对应的卡路里,这符合有意义准则,因为无论是现实还是虚拟都不能提供一眼就可以看到的食品卡路里值。

14.2.3 移动限制

移动 AR 通过设备摄像头将采集的现实物理环境与生成的虚拟对象进行合理组合以提供沉浸式交互体验,如前所述,移动 AR 要求用户一直握持设备,因此设计 AR 应用时对用户移动和设备移动要有良好的考量,尽量不要让用户全时移动并减少大量不必要的设备移动操作。

在交互方面,目前用户习惯于屏幕操作,这就要求在进行 AR 开发设计时,要对现实与虚拟对象的交互有很好的把握。交互设计要可视化、直观化、便于一只手操作甚至不用手去实现。移动 AR 设备是用户进行 AR 体验的窗口,还必须充分考虑用户使用移动 AR 的愉悦性,设计的 AR 应用要能适应不同的屏幕大小和方向。

14.2.4 心理预期与维度转换

大屏幕移动手机普及后,用户已形成特定的使用习惯及使用预期,二维应用操作时,其良好的交互模式可以带来非常好的应用体验,但 AR 应用与传统应用在操作方式上有非常大

的差异, 因此在设计时需要引导用户去探索新的操作模式, 鼓励用户在物理空间中移动, 并对这种移动给予适当的奖励回报, 让用户获得更深入、更丰富的体验。

操作习惯的转换固然很难, 但对开发者而言, 其有利的一点是现实世界本身就是三维的, 以三维的方式观察物体这种交互方式也更加自然, 良好的引导可以让用户学习和适应这种操作方式。如在图 14-2 中, 当虚拟的鸟飞离手机屏幕时, 如果能提供一个箭头指向, 就可以引导用户通过移动手机设备来继续追踪这只飞鸟的位置, 这无疑会提高用户的体验效果。

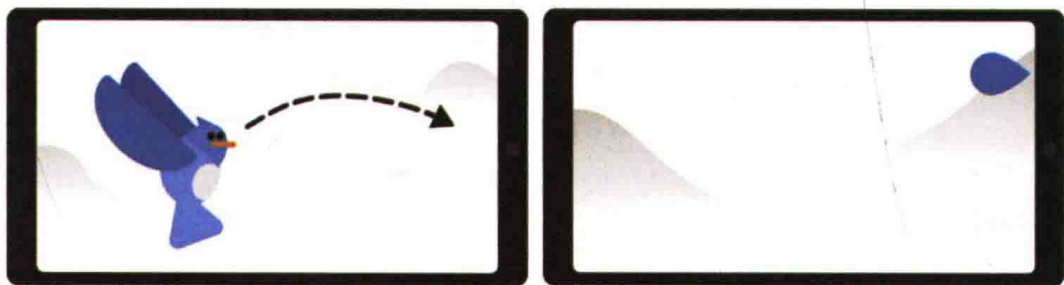


图 14-2 为虚拟对象提供适当的视觉指引

14.2.5 环境影响

AR 建立在现实环境基础之上, 如果虚拟的对象不考虑现实环境, 则将直接降低用户的使用体验。每个 AR 应用都需要拥有一个相应的物理空间与运动范围, 过于狭小紧凑的设计往往会让人感到不适。AR 有检测不同平面大小和不同平面高度的能力, 这非常符合三维的现实世界, 也为应用开发人员准确地缩放虚拟对象提供了机会。

一些 AR 应用对环境敏感, 另一些则不是, 如果对环境敏感的应用没有充分地考虑环境影响, 则可能导致应用开发失败, 如图 14-3 所示, 虚拟对象穿透墙壁会带来非常不真实的感觉, 在应用开发时, 应尽量避免此类问题。



图 14-3 虚拟对象穿透墙壁会带来非常不真实的感觉

14.2.6 视觉效果

AR 应用结合了现实物体与虚拟对象, 如果虚拟对象呈现的视觉效果不能很好地与现实统一, 就会极大的影响综合效果, 例如一只虚拟的宠物小狗放置在草地上, 小狗的影子与真实物体的影子相反, 这将会营造非常怪异的景象。

先进的屏幕显示技术及虚拟照明技术的发展有助于解决一些问题, 可以使虚拟对象看起来更加真实, 同时三维的 UI 设计也能增强 AR 的应用体验。三维 UI 的操作 (如状态选择和功能选择) 对用户的交互体验非常重要, 良好的 UI 设计可以有效地帮助用户对虚拟物体进行操作, 如在扫描平面时, 可视化的效果能让用户实时地了解到平面检测的进展情况并了解哪些地方可以放置虚拟对象。

AR 应用需要综合考虑光照、阴影、实物遮挡等因素, 动画、光影等视觉效果的完善对用户的整体体验提升非常重要。

14.2.7 UI 设计

UI 可以分为两类, 一类是固定在用户设备显示平面的 UI, 这类 UI 不随应用内容的变化而改变, 另一类是与对象相关的标签 UI, 这类 UI 有自己的空间三维坐标。这里主要指第一类 UI 设计, 这类 UI 是用户与虚拟世界交互的中间媒介, 也是目前用户非常适应的一种操作方式, 但要非常小心, AR 应用 UI 设计需要尽量少而精, 由于 AR 应用的特殊性, 面积大的 UI 应尽量采用半透明的方式, 如一个弹出的物品选择界面, 半透明效果可以让用户透过 UI 看到真实的场景, 营造一体化沉浸体验, 而不应让人感觉 UI 与 AR 分离。同时, 过多过密的 UI 会打破用户的沉浸体验, 导致非常糟糕的后果, 如图 14-4 所示。

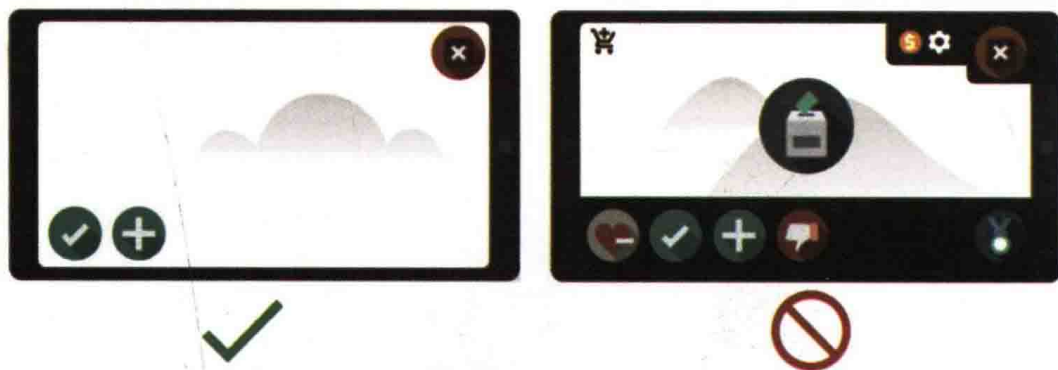


图 14-4 UI 元素过多过密会极大地影响 AR 应用沉浸感

14.2.8 沉浸式交互

在 AR 应用中, 所谓沉浸式就是让用户相信虚拟的对象就是存在于现实环境中的对象而不仅是一张图标。这是 AR 应用与 VR 应用在沉浸式方面很大的不同, VR 天生就是沉浸式的,

AR 由于现实世界的参与让沉浸变得困难。

由于上述原因，移动 AR 在设计交互时，包括对象交互、浏览、信息显示、视觉引导方面都与传统的应用设计有很大的不同。如在信息显示时根据对象的远近来决定显示信息的多少，这就会让用户感到很符合实际；通过考虑将虚拟对象放置在固定位置或动态缩放，可以优化可读性、可用性和用户体验；由于真实世界的三维特性，对虚拟物体的直接操作比对通过 UI 图标操作更符合 AR 应用，如图 14-5 所示。

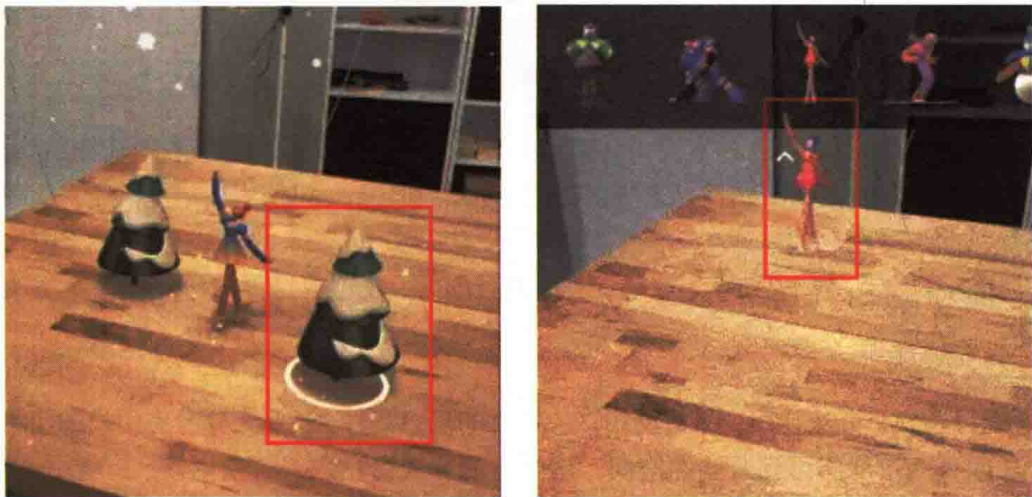


图 14-5 对虚拟物体的直接操作比对通过 UI 图标操作更符合 AR 应用

14.3 移动 AR 设计指南

在 AR 应用开发中遵循一些设计原则，可以应对移动 AR 带来的挑战，并能为用户操作提供方便和提升应用的体验效果。总体来讲，AR 设计应当遵循表 14-1 所示原则。

表 14-1 移动 AR 应用设计指南

体验要素	设计指南
操作引导	循序渐进地引导用户进行互动
	图文、动画、声频结合引导，能让用户更快上手操作，可通过声频、振动烘托气氛，增强代入感
	告知用户手机应有的朝向和移动方式，引导图表意清楚，明白易懂
模型加载	避免加载时间过长，减少用户对加载时长的感知
	大模型加载提供加载进度图示，减少用户焦虑
交互	增加趣味性、实用性，提高用户互动参与度
	避免强制性地用户在用户环境中添加 AR 信息
	综合使用声频、振动可以提升沉浸感
	操作手势应简单统一，符合用户使用习惯

续表

体验要素	设计指南
状态反馈	及时对用户的操作给予反馈，减少无关的杂乱信息干扰
模型真实感	充分运用光照、阴影、PBR 渲染、环境光反射、景深、屏幕噪声等技术手段提高模型真实感，提高模型与环境的融合度
异常引导	在运行中出现错误时，通过视觉、文字等信息告知用户，并允许用户在合适的时机重置应用

14.3.1 环境

VR 中的环境由开发者在开发应用时确定，开发人员能完全控制使用者所能体验到的各类环境，这也为用户体验的一致性打下了良好的基础，但 AR 应用使用的环境却千差万别，AR 应用必须足够“聪明”才能适应不同的环境，不仅如此，更重要的是 AR 的设计开发人员需要比其他应用开发人员更多地考虑这种差异，并且形成一套适应这种差异的设计开发方法。

1. 真实环境

AR 设计者要想办法让用户了解使用该应用时的理想条件，还要充分考虑用户的使用环境，从一个桌面到一个房间再到开阔的空间，用合适的方式让用户了解使用该 AR 应用可能需要的空间大小，如图 14-6 所示。尽量预测用户使用 AR 应用时可能带来的一些挑战，包括需要移动身体或者移动手机等。

特别要关注在公共场所使用 AR 应用而带来的更多挑战，包括大场景中的跟踪和景深遮挡的问题，还包括在用户使用 AR 应用时带来的潜在人身安全问题或者由于用户使用 AR 而导致出现一些影响到他人正常活动的问题。

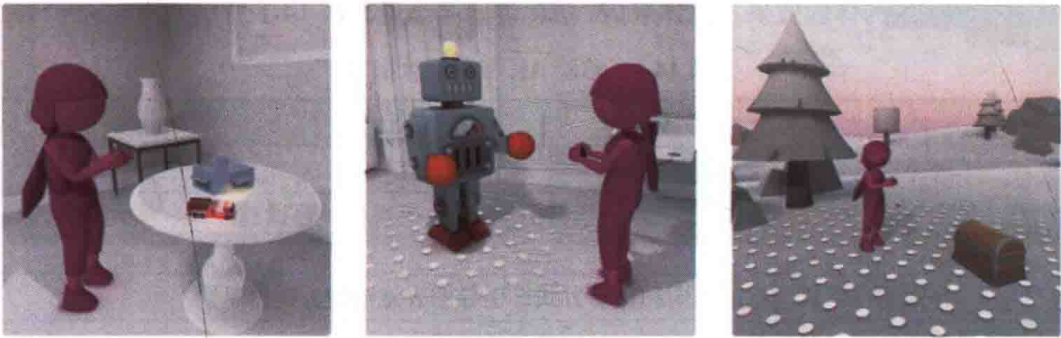


图 14-6 预估 AR 应用可能需要的空间

2. 增强环境

增强环境由真实的环境与叠加在其上的虚拟内容组成，AR 能根据用户的移动来计算用户的相对位置，还会检测捕获摄像头图像中的特征点信息并以此来计算其位置变化，再结合

IMU 惯性测量结果估测用户设备随时间推移而相对于周围环境的姿态（位置和方向）。通过将渲染三维内容的虚拟相机姿态与 AR 设备摄像头的姿态对齐，用户就能够从正确的透视角度看到虚拟内容。渲染的虚拟图像可以叠加到从设备摄像头获取的图像上，让虚拟内容看起来就像现实世界的一部分。

从前面章节已经知道，ARCore 可以持续地改进对现实环境的理解，它可以对水平、垂直、有角度的表面特征点进行归类和识别，并将这些特征点转换成平面供应用程序使用，如图 14-7 所示。

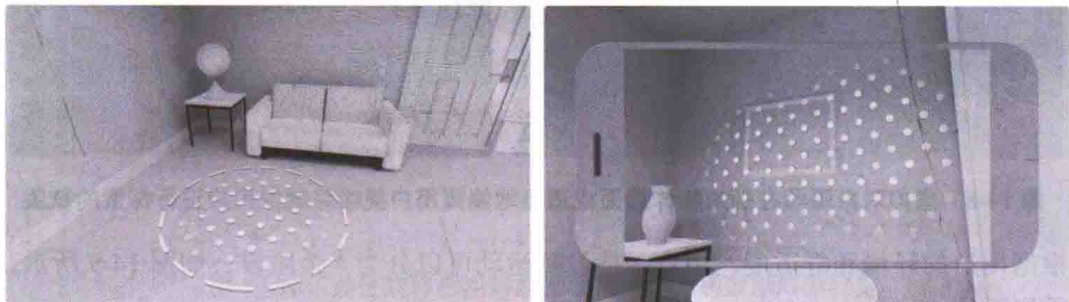


图 14-7 ARCore 检测环境并进行平面识别示意图

目前影响准确识别平面的因素包括以下几类：

- (1) 无纹理的平面，如白色办公桌、白墙。
- (2) 低亮度环境。
- (3) 极其明亮的环境。
- (4) 透明反光表面，如玻璃。
- (5) 动态的或移动的表面，如草叶或水中的涟漪。

当用户在使用 AR 时遇到上述环境限制时，需要设计友好的提醒告知用户变换环境或者操作方式。

14.3.2 用户细节

在智能设备长期使用中，用户已经习惯了当前的移动设备二维交互方式，往往倾向于保持静止，但 AR 应用是一种全新的应用形式，在 AR 中，保持静止就不能很好地发挥 AR 的优势，因此，在设计时应当关注此种差异并遵循适当的设计原则，引导用户熟悉新的交互操作形式。

1. 用户运动

由于 AR 要求用户移动操作的特殊性，与传统的设备操作习惯形成相对冲突，这时开发人员就应该通过合适的设计引导用户体验并熟悉这种新的操作模式，例如将虚拟对象放置在别的物体后面或者略微偏离视线中央可触发用户进一步探索的欲望，如图 14-8 所示。另外，通过合适的图标来提示用户进行移动以便完成后续操作也是不错的选择。

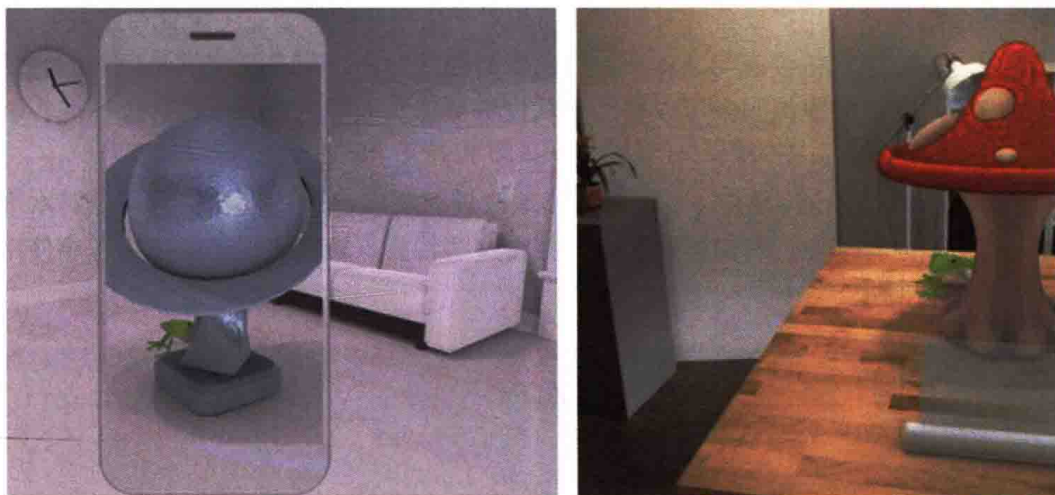


图 14-8 虚拟对象隐藏在别的物体后面或适当地偏离用户视线可触发用户进行探索的欲望

根据用户的环境和舒适度，用户手持设备运动可以分为 4 个阶段，如图 14-9 所示，这 4 个阶段如下：

- (1) 坐着，双手固定。
- (2) 坐着，双手移动。
- (3) 站着，双手固定。
- (4) 全方位的动作。

从第一阶段到第四阶段，对运动的要求越来越高，从部分肢体运动到全身运动，幅度也越来越大。对开发设计者而言，不管是要求用户的运动处于哪个阶段，都需要把握处理好以下几个问题：

- (1) 设计舒适，确保不会让用户处于不舒服的状态或位置，避免大幅度或突然的动作。
- (2) 当需要用户从一个动作转换到另一个动作时，提供明确的方向或指引。
- (3) 让用户了解触发体验所需要的特定动作。
- (4) 对移动范围给予明确的指示，引导用户对位置、姿态或手势进行必要的调整。
- (5) 降低不必要的运动要求，并循序渐进地引导用户进行动作。

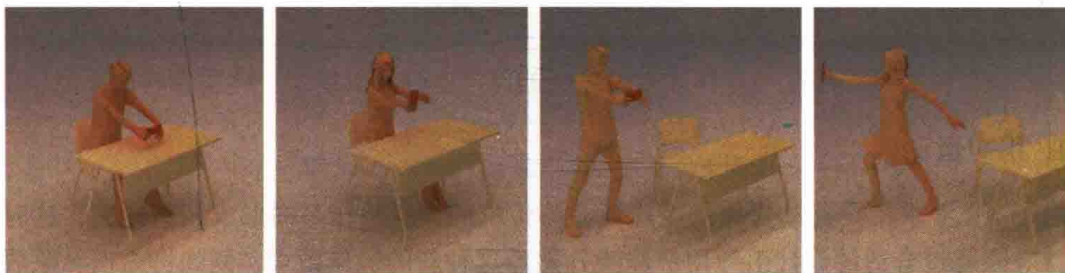


图 14-9 用户手持设备运动阶段示意图

在某些情况下，用户可能无法四处移动，此时设计者应提供其他备选方案。如可以提示

用户使用手势操作虚拟对象，通过旋转、平移、缩放等对虚拟物体进行操作来模拟需要用户运动才能达到的效果，如图 14-10 所示，当然这可能会破坏沉浸体验。

在用户疲劳时或者处于无法运动的状态时，通过旋转、平移、缩放操作可以大大地方便用户，开发人员应当考虑到类似需求并提供解决方案。

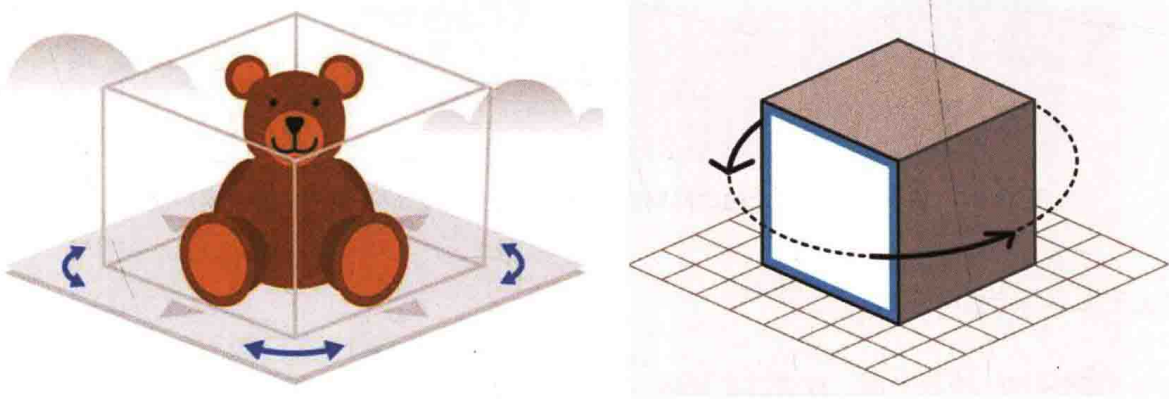


图 14-10 通过旋转虚拟对象来模拟用户围绕虚拟对象观察

2. 用户舒适度

AR 应用要求用户一直处在运动状态，并且要求至少用一只手握持移动设备，在长时间使用后可能会带来身体上的疲惫，所以在设计应用时需要时刻关注用户的身体状况。

- (1) 在应用设计的所有阶段考虑用户的身体舒适度。
- (2) 注意应用体验的时长，考虑用户可能需要的休息时间。
- (3) 让用户可以暂停或保存当前应用进度，在继续操作时使他们能够轻松地恢复应用进程，即使他们变更了物理位置。

3. 提升应用体验

AR 应用设计时应充分考虑用户的使用限制并符合一定的规则，尽量提前预估并减少用户使用 AR 应用时带来的不适。

预估用户实际空间的限制：室内和室外、实际的物理尺寸、可能的障碍物（包括家具、物品或人）。虽然仅仅通过应用无法知道用户的实际位置，但尽量提供建议或反馈以减少用户在使用 AR 应用时的不适感，并在设计时注意以下几个方面：

- (1) 不要让用户后退、进行快速大范围的身体动作。
- (2) 让用户清楚地了解 AR 体验所需的空间大小。
- (3) 提醒用户注意周围环境。
- (4) 避免将大物体直接放在用户面前，因为这样会强迫他们后退从而引发安全风险，如图 14-11 所示。



图 14-11 将一个尺寸过大的虚拟对象放置在用户前面会惊吓用户并强迫用户后退

14.3.3 虚拟内容

扫描环境、检测平面、放置虚拟物体这是 AR 应用常用的流程，但因为 AR 应用是一种新的应用形式，用户还不习惯不清楚其操作方式，因此提供合适、恰当的引导至关重要。

1. 平面识别

平面识别包括发现平面和检测平面两部分，在进行虚拟物体操作时还可能涉及多平面间的操作。

1) 发现平面

AR 应用启动时会进行初始化操作，在此过程中 AR 应用程序会评估周围环境并检测平面，但这时由于屏幕上并无可供操作的对象，用户对进一步的操作存在一定的困惑，为了减少可能的混淆，在设计 AR 时应为用户提供有关如何扫描环境的明确指引，如图 14-12 所示。通过清晰的视觉提示，来引导用户正确移动手机，加快平面检测过程，如提供动画来帮助用户了解所需进行的移动，例如顺时针或圆周运动。

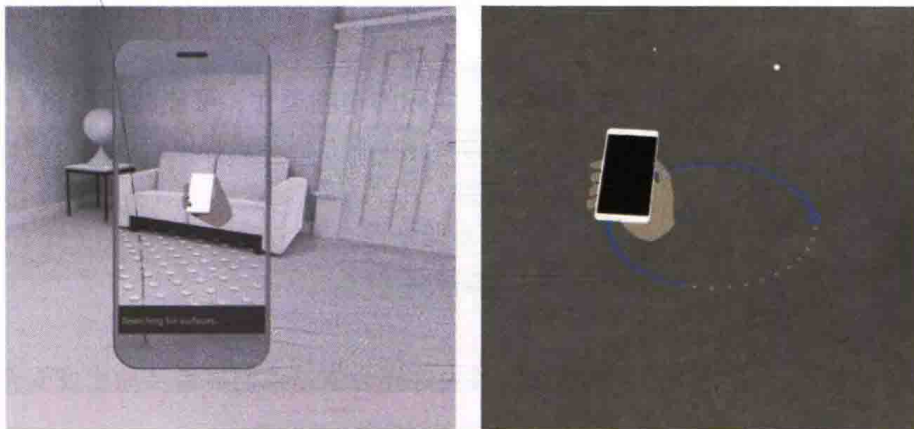


图 14-12 提供文字或动画引导用户扫描环境

2) 检测平面

在应用检测平面时, 提供清晰明确实时的平面检测反馈, 这个反馈可以是文字提示类, 也可以是可视化的视觉信息, 如图 14-13 所示, 在用户扫描环境时, 实时地反馈可以缓解用户的焦虑。

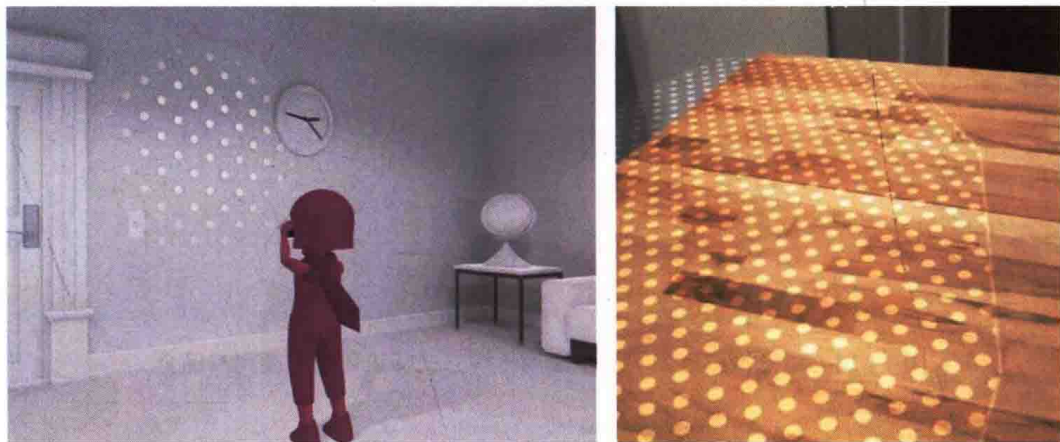


图 14-13 对 AR 检测发现的平面进行可视化显示更符合人类发现事物的规律

当用户检测到平面时, 应引导用户进行下一步操作, 建立用户信心并降低用户操作不适感, 如通过以下方式来引导用户进行后续操作。

(1) 设计无缝的过渡: 当用户快速移动时, 系统可能会无法跟踪, 在发现平面和检测平面之间设计平滑的过渡。

(2) 标识已检测到平面: 在未检测与已检测的平面之间做出区分, 考虑通过视觉上的可视化信息来标识已检测到的平面。

(3) 以视觉一致性为目标: 为了保持视觉的一致性, 每种状态的视觉信息也应具备大众的审美属性。

(4) 使用渐进式表达: 应当及时和准确地传达系统当前的状态变化, 通过视觉高亮或文本显示可以更好地表达出平面检测成功的信息。

3) 多个平面

ARCore 可以检测多个平面, 在设计 AR 应用时, 应当通过明确的颜色或者图标来标识不同的平面, 通过不同颜色或者图标来可视化平面可以协助用户在不同的平面上放置虚拟对象, 如图 14-14 所示。

为更好地提供给用户明确的平面检测指示, 应遵循以下原则:

(1) 高亮显示那些已检测并准备好放置物体的平面。

(2) 在不同平面之间构建视觉差异, 通常以不同颜色来标识不同平面, 以避免在后续放置虚拟物体时发生混淆。

(3) 仅在视觉上高亮显示用户正在浏览或指向的平面, 但需要注意的是, 不要一次高亮多个平面, 这会让用户失去焦点。

(4) 在发现平面的过程中缺乏视觉或者文字提示会导致用户失去耐心, 通常需要将可用的平面与不可用的平面明确区分并以可视化的方式告知用户。



图 14-14 对已检测到的不同平面用不同方式标识可以方便用户区分

2. 物体放置范围

确定最佳放置范围有助于确保用户将虚拟对象放置在舒适的观察距离内, 在该范围内放置虚拟对象能优化用户的使用体验, 避免将虚拟对象放置在屏幕边缘, 除非有意引导用户移动。

1) 场景分区

手机屏幕有限的视场会对用户感知深度、尺度和距离带来挑战, 这可能会影响用户的使用体验及与物体交互的能力, 尤其是, 对深度的感知会由于物体位置关系而发生变化。如将虚拟物体放置得离用户太近, 会让人感到惊讶甚至惊恐, 此外, 将大物体放置在离用户过近的位置时, 可能会强迫他们后退, 甚至撞到周围的物体, 引发安全风险, 而过远过小的虚拟物体则非常不方便操作。

为了帮助用户更好地了解周围环境, 可通过将屏幕划分为 3 个区域设置舒适的观察范围: 下区、中区、上区, 如图 14-15 所示。



图 14-15 对手机屏幕进行区域划分找出最佳位置

下区：离用户太近，如果虚拟对象没有遵照期望，放置得离用户过近，用户很难看到完整的对象，从而迫使用户后退。

上区：离用户太远，如果虚拟对象被放在上区，用户会很难理解“物体缩小与往远放置物体（近大远小）”之间的关系，导致理解与操作困难。

中区：这是用户最舒适的观察范围，也是最佳的交互区域。

注意，3 个区域的划分相对于手机视角，与用户手持手机的姿态没有关系。

2) 最大放置距离

在 AR 设计及操作时，应该引导用户在场景中的最佳位置放置物体，帮助用户避免将物体放置在场景内不舒适的区域中，也不宜将虚拟对象放置得过远，如图 14-16 所示，过近或过远的区域都不建议使用。



图 14-16 设计时应确保虚拟对象不要放置得过远或过近

使用最远放置距离有助于确保将对象放置在舒适的观察距离上，也可以保证用户在连续拖动时，保持物体的真实比例。

3) 目标位置

目标位置是指最终放置物体的位置，在放置虚拟物体时应提供最终位置指引，如图 14-17 所示。



图 14-17 对虚拟对象最终放置提供可视化指引

在用户放置物体时通过可视化标示引导用户，阴影可以帮助指明目标位置，并让用户更容易了解物体将被放置在已检测平面的精确位置。

3. 虚拟对象放置

如果不在真实环境中放置虚拟对象，AR 就没有意义，只有将虚与实结合起来才能带给用户耳目一新的体验。在真实场景中放置虚拟对象可以是自动的（由程序控制放置），也可以是手动的（由用户选择放置）。

1) 自动放置

自动放置是由应用程序控制在场景中放置物体，如一旦检测到平面，程序就自动在检测到的平面上种花草，随着检测的进行，种草的过程也一直在动态地进行，并且这个过程不需要用户的参与，检测完成时所有检测到的平面上就都种上了植物，如图 14-18 所示。

通常来讲，自动放置适用于以下情况：

- (1) 虚拟环境需要覆盖整个现实空间，例如一个魔法界面或者游戏地形。
- (2) 非常少或者完全不需要交互。
- (3) 不需要精确地控制虚拟物体的放置位置。
- (4) AR 应用模式需要在启动应用时就自动开始放置虚拟对象。



图 14-18 自动在已检测的平面上种植植被

2) 手动放置

手动放置是由用户控制的放置行为，用户可以在场景中实施虚拟对象放置和其他操作，这可以包括锚定一个游戏空间或者设置一个位置来开启 AR 体验。

(1) 单击放置对象。允许用户通过单击场景中已检测到的平面位置，或者通过单击选中的虚拟物体图标来放置虚拟物体，通过单击选中的虚拟物体图标放置物体如图 14-19 所示。

单击已检测的平面放置虚拟物体的方式通常对用户来讲是非常自然的，以下情况下更适合单击放置对象：

- A. 在放置之前，虚拟对象不需要进行明显的调整或转换（缩放 / 旋转）。
- B. 通过单击快速放置。

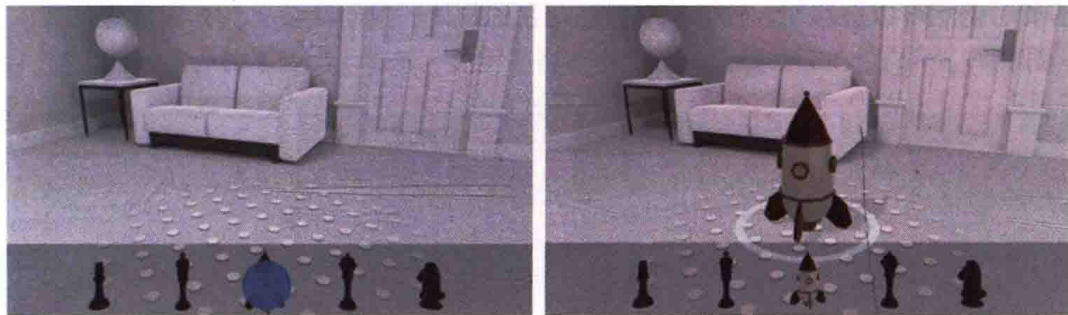


图 14-19 通过单击选中的图标在已检测到的平面上放置虚拟对象

当需要同时将多个不同虚拟对象放置在场景中时，单击已检测到的平面放置多个虚拟对象可能并不适用，这时可以弹出菜单由用户选择放置的对象，而不是同时将多个相同的虚拟对象放置在平面上，如图 14-19 底部显示的可供选择的虚拟物体菜单列表。

(2) 拖动放置对象。这是一种精度很高且完全由用户控制的放置物体操作，该操作允许用户将虚拟物体从库中拖动到场景中，如图 14-20 所示。

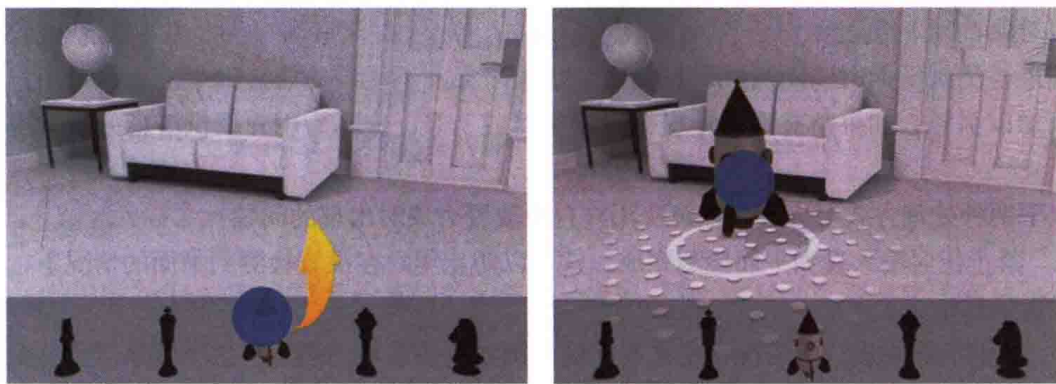


图 14-20 拖放放置方式可以精确地控制虚拟物体的放置位置

通常用户可能不太熟悉这种拖放操作，这时应该给用户视觉或者文字提示，提供拖动行为的明确指引和说明，当用户事先并不了解放置操作手势时，拖动行为就无法很好地工作，显示放置位置的定点标识图标就是不错的做法。

拖动操作非常适合以下情形：

- A. 虚拟对象需要进行显著调整或转换。
- B. 需要高精度地放置。
- C. 放置过程是体验的一部分。

(3) 锚定放置。锚定对象与拖动放置物体不同，通常锚定的对象不需要经常性地旋转、平移、缩放，或者锚定的对象包含很多其他对象，需要整体操作。被锚定的对象通常会固定在场景中的一个位置上，除非必要一般不会移动。锚定对象通常在放置一些需要固定的对象时有用，例如游戏地形、象棋棋盘，如图 14-21 所示，当然，锚定对象也不是不可以移动，在

用户需要时仍然可以被移动。在场景中静态存在的物体一般不需要采用锚定放置的方式,如沙发、灯具等。

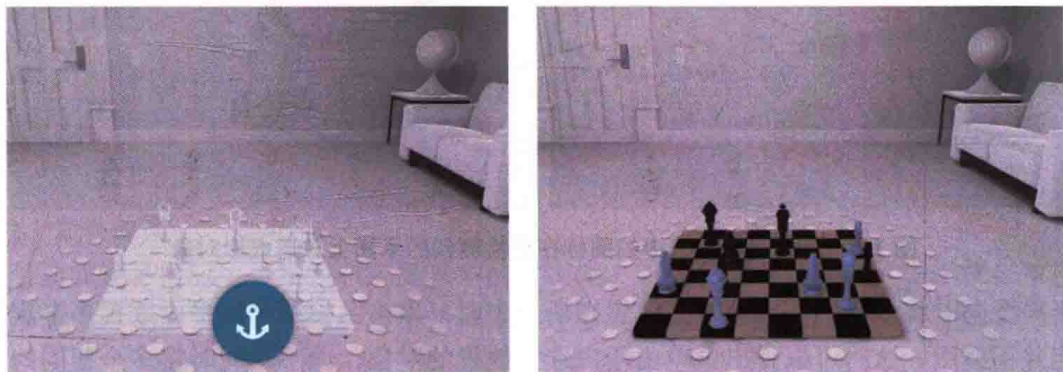


图 14-21 将虚拟物体锚定放置到指定位置

(4) 自由放置。自由放置允许用户自由地对虚拟物体进行放置操作,但在未检测到平面时放置物体通常会造成混乱,如果虚拟对象出现在未检测到平面的环境中,会造成幻象,因为虚拟对象看起来像是悬浮的,这将破坏用户的 AR 体验,并阻碍用户与虚拟对象进一步交互,因此在未检测到平面时,应当让用户知道该虚拟对象并不能准确放置到平面上,对用户的行为应当加以引导,如果是有意想让虚拟对象悬浮或上升,应当为用户提供清晰的视觉提示和引导。

以下两种处理方式可以较好地解决用户自由放置对象时出现的问题:

A. 禁止任何输入直到平面检测完成,这可以防止用户在没有检测到平面的情况下,将物体放置在场景中。

B. 提供不能放置对象的视觉或文本反馈。例如使用悬停动画、半透明虚拟对象、振动或文本传达出在当前位置不能放置虚拟物体的信息,并引导用户进行下一步操作。

14.3.4 交互

模型的交互程度,需根据模型自身属性/产品的类型定义,并非都需要涉及所有可交互类型,在进行与核心体验无关的交互时,可予以禁止或增加操作难度。如科普类模型固定放置在平面后,需要便捷地旋转以查看模型细节,但 Y 轴移动查看的需求不大,部分场景可考虑禁止沿 Y 轴移动的操作。

手势设计优先使用通用的方式,若没有通用的方式,则尽可能使用简单和符合用户直觉的方式进行设计。若违反该原则,则可能造成用户的理解和记忆障碍,给用户操作造成困难。

1. 选择

选择是交互的最基本操作,除了环境类的虚拟对象,应当允许用户辨别、选择虚拟物体,以及与虚拟物体进行交互。

在用户选择虚拟对象时应当创建视觉指引，高亮或者用明显的颜色、图标标识那些可以与用户交互的对象，如图 14-22 所示。尤其是在有多个可供选择物体的情况下，明确指示反馈显得非常重要，同时还应保持虚拟物体原本的视觉完整性，不要让视觉提示信息凌驾于虚拟物体之上。

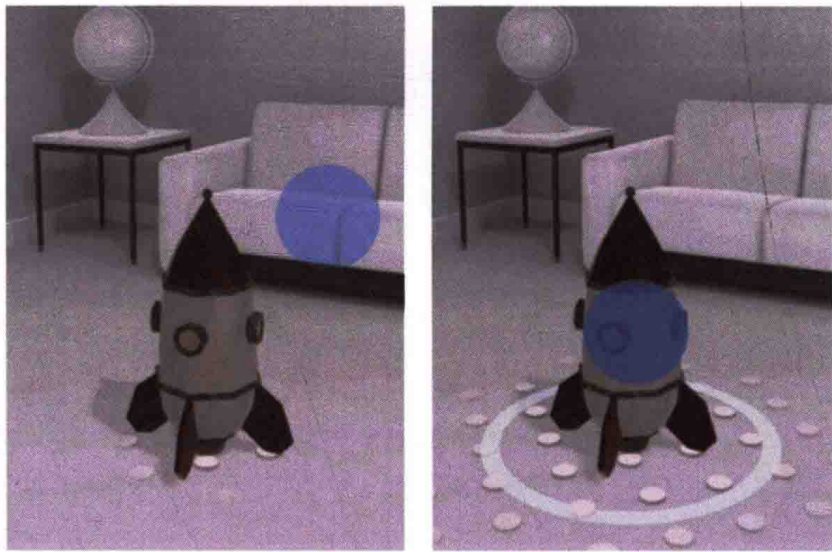


图 14-22 对可选择的虚拟对象提供一个清晰的视觉提示

2. 平移

AR 应用应当允许用户沿着检测到的平面移动虚拟对象，或从一个平面移动到另一个平面。

1) 单平面移动

单平面移动指只在一个检测到的平面内移动虚拟物体，在移动物体前用户应当先选择它，用手指沿屏幕拖动或实际地移动手机从而移动虚拟物体，这种方式相对比较简单，如图 14-23 所示。

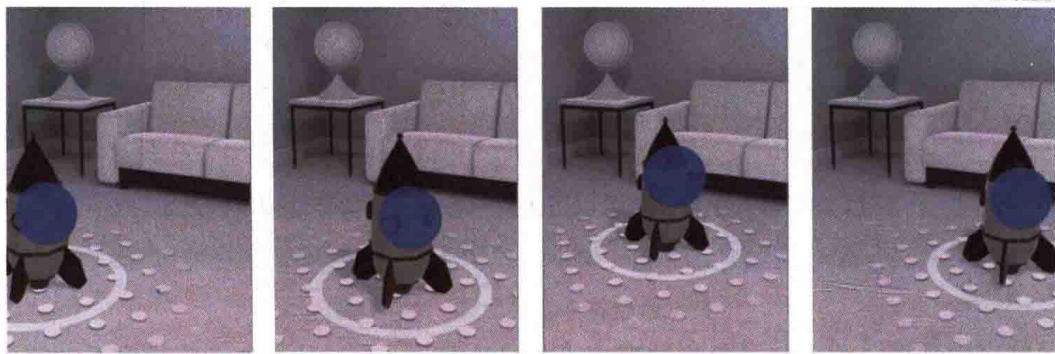


图 14-23 在选择虚拟物体后可实施移动

2) 多平面移动

多平面移动是指将虚拟对象从 AR 检测到的一个平面移动到另一个平面,多平面移动比单平面移动需要考虑的因素更多一些,在移动过程中应当避免突然旋转或缩放,要有视觉上的连续性,不然极易给用户带来不适。对于多平面间的移动需要注意以下几点:

(1) 在视觉上区分多个平面,明确标示出不同的平面。

(2) 避免突然变化,如旋转或者缩放。

(3) 在用户松开手指前,在平面上显示虚拟物体放置点以提示用户放置的平面,如图 14-24 所示。

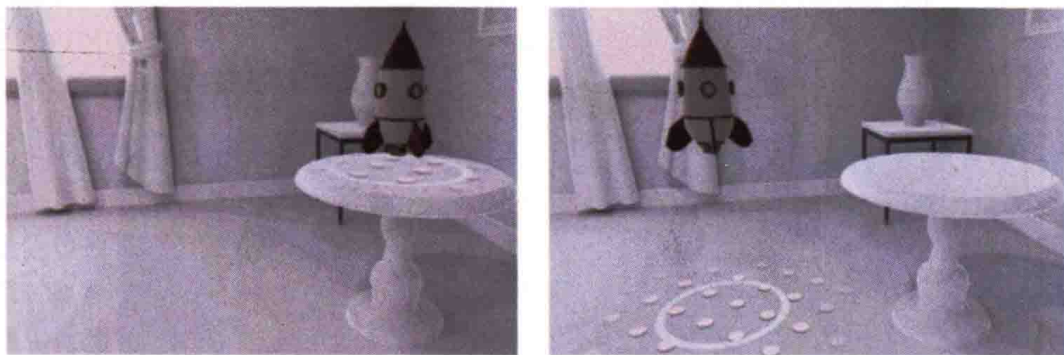


图 14-24 在平面间移动物体时提前告知用户放置平面及放置点

3) 平移限制

AR 应用应当对用户的平移操作进行适当限制,限制最大移动范围。添加最大平移限制主要是为了防止用户将场景中的物体平移得太远,以至于无法查看或操作,如图 14-25 所示,在虚拟物体移动得过远时用红色图标标识当前位置不能使用。



图 14-25 添加平移限制可以防止用户将虚拟对象移动得过远过小

3. 旋转

旋转可以让用户将虚拟对象旋转到其所期望的方向,旋转分为自动旋转和手动旋转,旋转也应该对用户的操作给出明确的提示。

手动旋转:手动旋转可使用单指进行旋转,也可使用双指进行旋转。使用单指进行旋转时,

通常通过屏幕手势 delta.x 值提供旋转量值;通过双指手势进行旋转时,为避免与缩放手势冲突,要求双指同时顺时针或者逆时针旋转,如图 14-26 所示。

自动旋转:尽量避免自动旋转,除非这是体验中有意设计的一部分,长时间自动旋转可能会令用户感到不安,如果物体的方向被锁定为朝向用户,则当手动更改物体方向时应限制自动旋转。

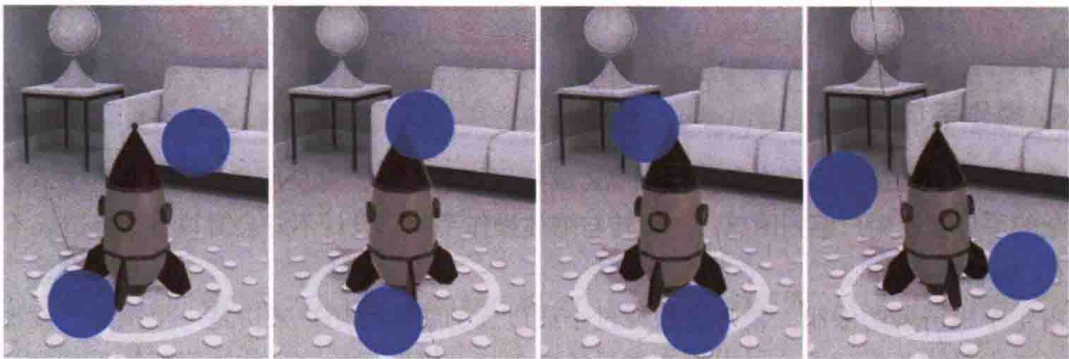


图 14-26 双指同时逆时针旋转操作物体旋转

4. 缩放

缩放是指放大或缩小虚拟物体的显示尺寸,在屏幕上的操作手势应尽量与当前操作二维缩放的手势保持一致,这符合用户使用习惯,方便用户平滑过渡。

1) 缩放

缩放常用捏合手势操作,如图 14-27 所示。

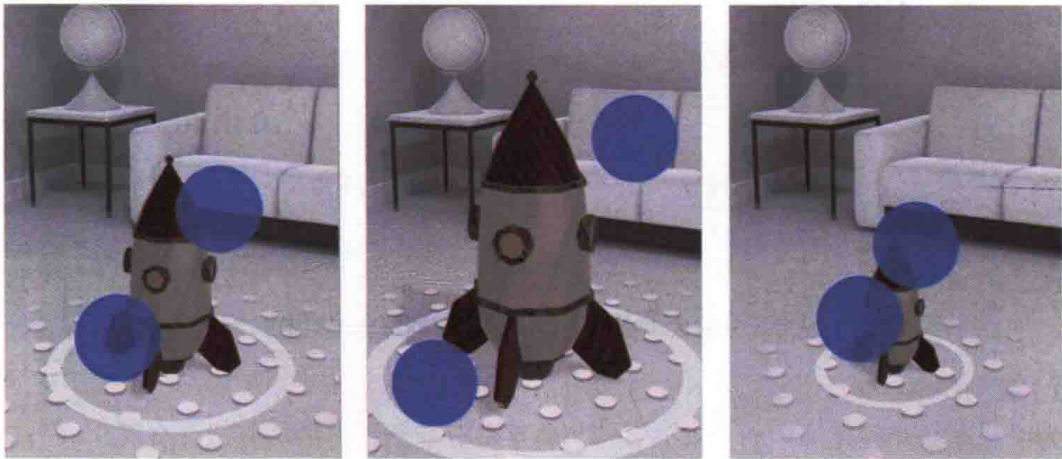


图 14-27 使用捏合手势缩放对象

2) 约束

与平移一样,也应当添加最小和最大缩放限制,防止用户将虚拟对象放得过大或缩得过小。允许较小的缩放比例用于精确的组合场景,考虑添加回弹效果来指示最大和最小尺寸。

另外,如果物体已根据需达到实际比例,则应当给用户以提示,如达到 100% 比例时添加一定的吸附效果。

3) 可玩性

有时候,也可以不必拘泥于约束,夸张的缩放可能会带来意外惊喜,如放置在场景中的大型虚拟角色可以增加惊喜元素,并让人感觉更加有趣。另外,声音也可用于与缩放同步,以增强真实性,如当物体放大时,同时增大音量,配合模型缩放的音量调整会让用户感觉更加真实。

5. 操作手势

在手机屏幕上操作虚拟对象是目前 AR 成熟可用的技术,也是用户习惯的操作方式,但有时复杂的手势会让用户感到困惑,如旋转与缩放操作手势,设计不好会对执行手势造成不便。

1) 接近性

由于手机屏幕的限制,精准地操控过小或过远的物体对用户来讲是一个挑战。设计开发时应当考虑触控目标的大小,以便实现最佳的交互。当应用检测到物体附近的手势时,应当假设用户正在与它进行交互,即使可能并没有完全选中该目标。另外,尽管目标物体尺寸比较小,但也应当提供合理的触控尺寸,在达到一定范围后,触控尺寸不应随着目标的缩小而缩小,如图 14-28 所示。



图 14-28 触控尺寸的大小在某些情况下不应随着虚拟对象的缩放而缩放

2) 采用标准手势

为手势和交互创建统一的标准体系,当将手势分配给特定的交互或任务时,应当避免使用类似的手势来完成不同类型的任务,如通过双指捏合手势缩放物体时,应当避免使用此手势来旋转物体。

3) 融合多种两指手势

双指手势通常用于旋转或缩放对象,下面这些触控手势应为两指手势的一部分。

- (1) 使用食指加拇指旋转。
- (2) 使用拇指加食指，用拇指作为中心，旋转食指。
- (3) 分别独立使用两个拇指。

14.3.5 视觉设计

视觉设计是一款正式应用软件必须重视的用户体验，一款软件能否有用户黏度，除了功能外，视觉设计也占很大一部分原因。在 AR 应用中，尽可能多地使用整个屏幕来查看及探索物理世界和虚拟对象，避免过多过杂的 UI 控制图标和提示信息，因为这将使屏幕混乱不堪从而降低沉浸体验。

1. UI

UI 设计应以沉浸体验为目标，目的是在视觉上融合虚拟与现实，既方便用户操作又与场景充分融合。创建一个视觉上透明的 UI 可以帮助构建无缝的沉浸式体验，切记不可满屏的 UI，这会极大地降低 AR 应用的真实感从而破坏沉浸式体验。

1) 界面风格

在 UI 设计时应当尽量避免让用户在场景和屏幕之间来回切换，这会分散用户注意力并减少沉浸感，可以考虑减少屏幕上 UI 元素的数量，或尽量将这些控件放在场景中，如图 14-29 所示，应使用如左图所示的简洁菜单而不要采用铺满全屏的设计。

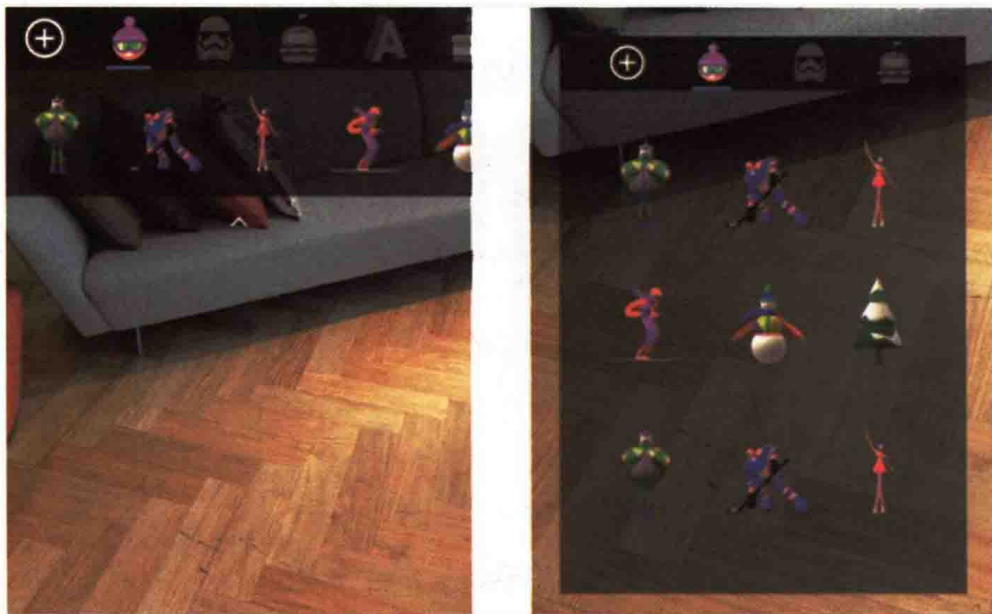


图 14-29 使用简单的菜单更适合 AR

2) 删除

由于移动手机二维软件操作传统，将物体拖动到垃圾桶图标上进行删除更符合当前用户

的操作习惯,如图 14-30 所示。当然这种操作也有弊端,如物体太大时不宜采用这种方式。在删除物体时最好提供被删除物体消失的动画,在增强趣味性的同时也增强用户的视觉感受。



图 14-30 通过拖放的方式来删除虚拟对象

3) 重置

应当允许用户进行重置,在一定的情况下重新构建体验,这包括以下几点:

- (1) 当系统无响应时。
- (2) 体验是渐进式并且任务完成后。

重置是一种破坏性的操作,应当先征询用户意见,如图 14-31 所示。



图 14-31 在特定的情况下允许用户重置应用

4) 权限申请

明确应用需要某些权限的原因,如告诉用户需要访问其设备摄像头或 GPS 位置信息的原因。一般需要在应用使用某一权限时才提出请求,而不是在应用启动时要求授权,不然用户可能

会犹豫是否同意访问。

5) 错误处理

应用出现错误在某些情况下不可避免，特别是对于 AR 这类使用环境不可预测的应用。在出现错误时应当积极帮助用户从错误中恢复，使用视觉、动画和文本的组合，告知用户当前发生的问题，并为系统错误和用户错误传达明确清晰的解决操作措施。

标明出现的问题，用语要通俗易懂，要避免专业术语，并给出进行下步操作的明确步骤，错误提醒的部分示例如表 14-2 所示。

表 14-2 错误提示示例

错误类型	描 述
黑暗的环境	环境太暗无法完成扫描，请尝试打开灯或移动到光线充足的区域
缺少纹理	当前图像纹理太少，请尝试扫描纹理信息丰富的表面
用户移动设备太快	设备移动太快，请不要快速移动设备
用户遮挡传感器或摄像头	手机摄像头传感器被阻挡，请不要遮挡摄像头

2. 视觉效果

AR 应用界面设计在视觉上应该要有足够的代入感，但同样要让用户感觉可控，界面效果充分融合虚与实。

1) 界面

界面是用户打开应用后最直观的视觉感受，虽然 AR 应用要求界面简洁，但杂乱放置、风格不一致的图标会在用户心中留下糟糕的第一印象。AR 应用界面设计时，既要设计身临其境的沉浸感，又要考虑用户独立的控制感，如图 14-32 所示。



图 14-32 简洁一致的界面

在设计用户界面时应时刻关注以下几个问题。

- (1) 覆盖全屏：除非用户自己明确选择，否则应避免这种情况发生。
- (2) 二维元素覆盖：避免连续的二维元素覆盖，因为这会极大地破坏沉浸感。
- (3) 连续性体验：避免频繁地让用户反复进出场景，确保用户在场景中即可执行主要和次要任务，如允许用户选择、自定义、更改或共享物体而无须离开 AR 场景。

2) 初始化

在启动 AR 应用时，屏幕从二维到 AR 三维之间的转换，应当采用一些视觉技术清晰地指明系统状态，如在即将发生转换时，将手机调暗或使用模糊屏幕等效果，提供完善的引导流程，避免突然变化。

允许用户快速开启 AR 应用，并引导用户在首次运行中按流程执行关键的任务，这将有助于执行相关任务并建立黏性。在添加流程引导提示时，需要注意以下几点：

- (1) 任务完成后即时解除提示。
- (2) 如果用户重复相同的操作，则应当提供提示或重新开始重要的视觉引导。

依靠视觉引导，而不是仅仅依赖于文本，使用视觉引导、动作和动画的组合来指导用户，如在操作中用户很容易理解滑动手势，可以在屏幕上通过图标向他们展示，而不是仅仅通过纯文本指令性地进行提示，如图 14-33 所示。



图 14-33 提供多种方式的组合来引导用户操作

3) 用户习惯

尽量利用用户熟悉的 UI 形式、约定及操作方式，与标准 UX 交互形式和模式保持一致，同时不要破坏体验的沉浸感，这会加快用户适应操作并减少对说明或详细引导的需求。

4) 横屏与竖屏

尽量提供对竖屏和横屏模式的支持，除非非常特别的应用，都应该同时支持竖屏与横屏操作，如果无法做到这一点，就需要友好地提示用户。支持这两种模式可以创造更加身临其

境的体验，并提高用户的舒适度，如图 14-34 所示。

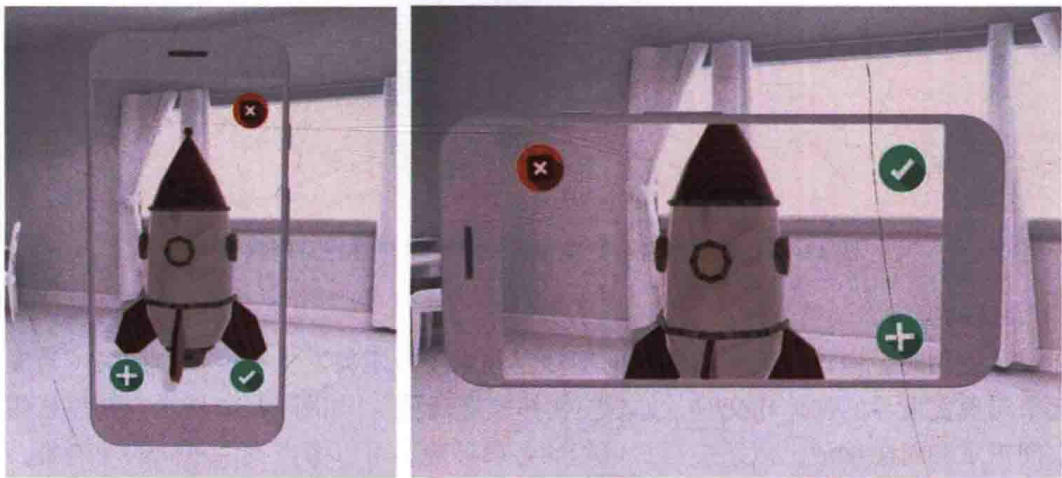


图 14-34 提供平滑的横屏与竖屏转换

在横竖屏适应及转换中还应该注意以下几点。

- (1) 相机和按钮位置：对于每种模式，注意相机的位置及其他按钮的放置要尽量符合一般原则，并且不要影响设备的深度感知、空间感知和平面感知。
- (2) 关键目标位置：不要移动关键目标，并允许对关键目标的旋转操作。
- (3) 布局：适当的情况下，更改次要目标的布局。
- (4) 单一模式支持：如果只支持一种模式，则应当向用户说明原因。

5) 声频

使用声频可以鼓励用户参与并增强体验。声频可以帮助构建身临其境的 360° 环绕沉浸体验，但需要注意的是要确保声音可增强这种体验，而不是分散注意力。在发生碰撞时使用声音效果或振动是确认虚拟对象与物理表面或其他虚拟对象接触的好方法，在沉浸式游戏中，背景音乐可以帮助用户融入虚拟世界。在三维虚拟物体或 360° 环境中增加声频，应当注意以下几个方面：

- (1) 避免同时播放多个声音。
- (2) 为声音添加衰减效果。
- (3) 当用户没有操控物体时，可以让声频淡出或停止。
- (4) 允许用户手动关闭所选物体的声频。

6) 视觉探索

当需要用户移动时，可以使用视觉或声频提示来鼓励用户进行屏幕外空间的探索。有时候用户很难找到位于屏幕外的物体，可以使用视觉提示来引导用户，鼓励用户探索周边更大范围内的 AR 世界。例如，飞鸟飞离屏幕时应该提供一个箭头指引，让用户移动手机以便追踪其去向，并将其带回场景，如图 14-35 所示。

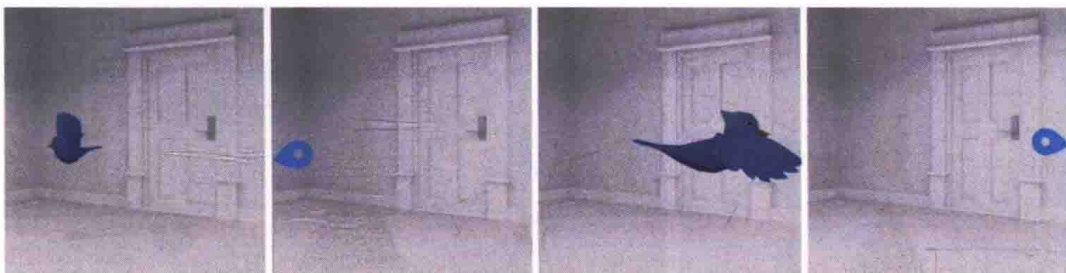


图 14-35 使用声音或视觉提示来鼓励用户进行空间探索

7) 深度冲突

在设计应用时,应该始终考虑用户的实际空间大小,避免发生深度上的冲突(当虚拟物体看起来与现实世界的物体相交时,如虚拟物体穿透墙壁),如图 14-36 所示。并注意建立合理的空间需求和对象缩放,以适应用户可能的各种环境,可以考虑提供不同的功能集,以便在不同的环境中使用。

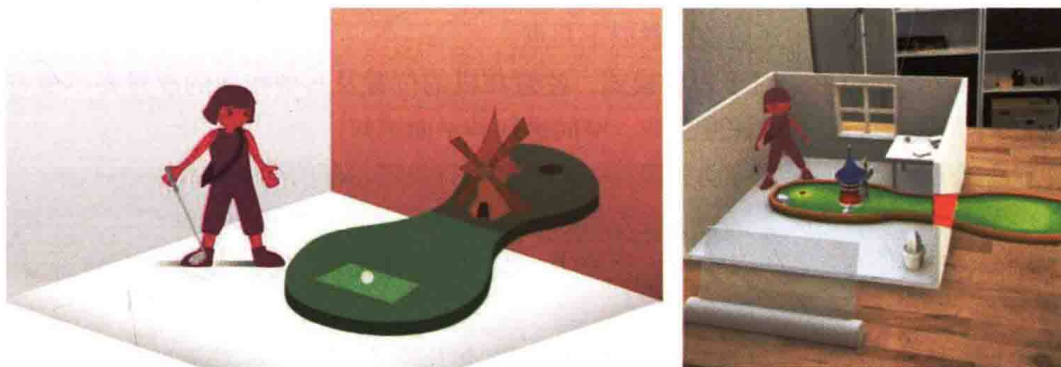


图 14-36 深度冲突会破坏用户体验的沉浸感

8) 穿透物体

在用户使用中,有可能会因为离虚拟物体太近而产生穿透进入虚拟物体内部的情况,这会破坏虚拟物体的真实性并打破沉浸体验,当这种情况发生时,应当让用户知道这种操作方式不正确或距离过近,通常在摄像头进入物体内部时可采用模糊的方式提示用户。

14.3.6 真实感

通过利用阴影、光照、遮挡、反射、碰撞、物理作用、PBR 渲染(Physically Based Rendering, 基于物理的渲染)等技术手段来提高虚拟物体的真实感,可以更好地将虚拟物体融入真实世界中,提高虚拟物体的可信度,营造更加自然真实的虚实环境。

1. 建模

在构建模型时,模型的尺寸应与真实的物体尺寸保持一致,如一把椅子的尺寸应与真实的椅子尺寸相仿,一致的尺寸更有利于在 AR 中体现真实感。在建模时,所有的模型应当在

相同的坐标系下构建,建议全部使用右手坐标系,即 Y 轴向上、 X 轴向右、 Z 轴向外。模型原点应当构建在物体中心下部平面上,如图 14-37 所示。另外需要注意的是,在 AR 中,模型应当完整,所有面都应当有材质与纹理,以避免部分面出现白模的现象。

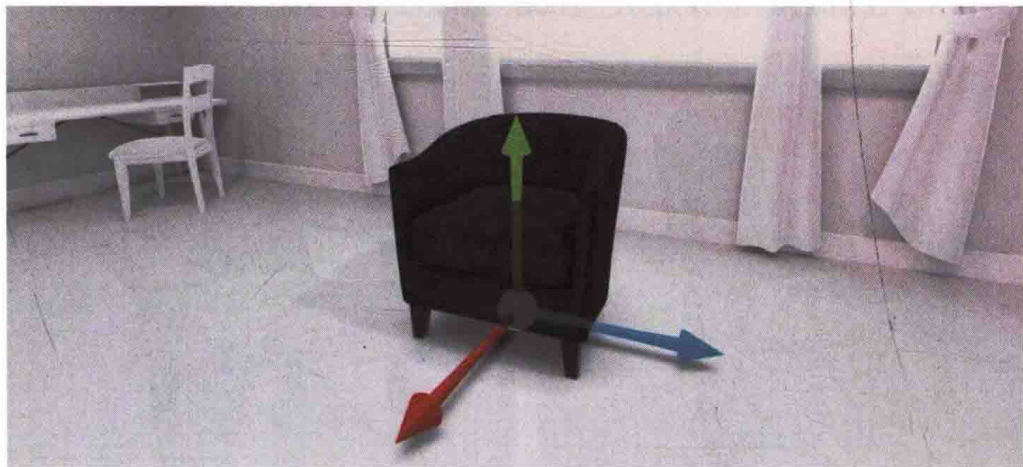


图 14-37 模型原点位置及坐标系示意图

2. 纹理

纹理是表现物体质感的一个重要因素,为加快载入速度,纹理尺寸不应过大,建议分辨率控制在 2K 以内。带一点噪声的纹理在 AR 中看起来会更真实,重复与单色纹理会让人感觉虚假,带凸凹、裂纹、富有变化、不重复的纹理会让虚拟物体看起来更富有细节和更可信。

1) PBR 材质

当前在模型及渲染中使用 PBR 材质可以让虚拟物体更真实,PBR 可以给物体添加更多真实的细节,但 PBR 要达到理想效果通常需要很多纹理。使用物理的方式来处理这些纹理可以让渲染更自然可信,如图 14-38 所示,这些纹理共同作用定义了物体的外观,可以强化 AR 中的视觉表现。



图 14-38 采用 PBR 渲染可以有效提升真实感

2) 法线贴图

法线贴图可以在像素级层面上模拟光照,可以给虚拟模型添加更多细节,而无须增加模型顶点及面数。法线贴图是理想的制作照片级模型渲染的手段,可以添加足够细节的外观表现,如图 14-39 所示,左图使用了法线贴图,可以看到细节纹理更丰富。

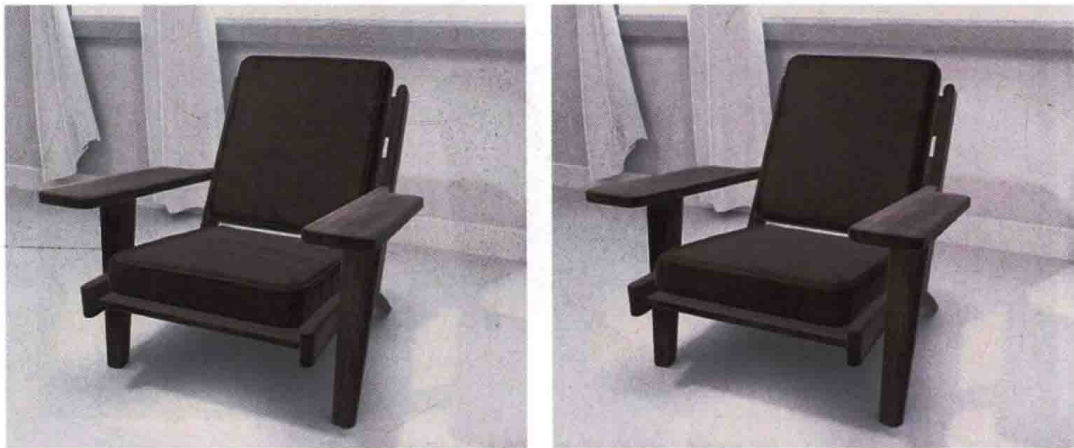


图 14-39 法线贴图能非常好地表现纹理细节

3) 环境光遮罩贴图

环境光遮罩贴图是一种控制模型表面阴影的技术方法,使用环境光遮罩贴图,来自真实世界的光照与阴影会在模型表面形成更真实的阴影效果,更富有层次和景深外观表现。

3. 深度

透视需要深度信息,为营造这种近大远小的透视效果,需要在设计时利用视觉技巧让用户形成深度感知以增强虚拟对象与场景的融合和真实感,如图 14-40 所示,在远处的青蛙要比在近处的青蛙小,这有助于帮助用户建立景深。



图 14-40 深度信息有助于建立自然的透视

通常用户可能难以在增强现实体验中感知深度和距离,综合运用阴影、遮挡、透视、纹理、常见物体的比例,以及放置参考物体来可视化深度信息,可帮助建立符合人体视觉的透视效果。如青蛙从远处跳跃到近处时其比例、尺寸大小的变化,通过这种可视化方式表明空间深度和层次。开发人员可以采用阴影平面、遮挡、纹理、透视制造近大远小及物体之间相互遮挡的景深效果。

4. 光照

光照是影响物体真实感的一个重要因素,当用户真实环境光照条件较差时,可以采用虚拟灯光照明以为场景中的对象创建深度和真实感,也就是在昏暗光照条件下可以对虚拟物体补光,如图 14-41 所示,但过度的虚拟光照会让虚拟对象与真实环境物体形成较大反差,进而破坏沉浸感。

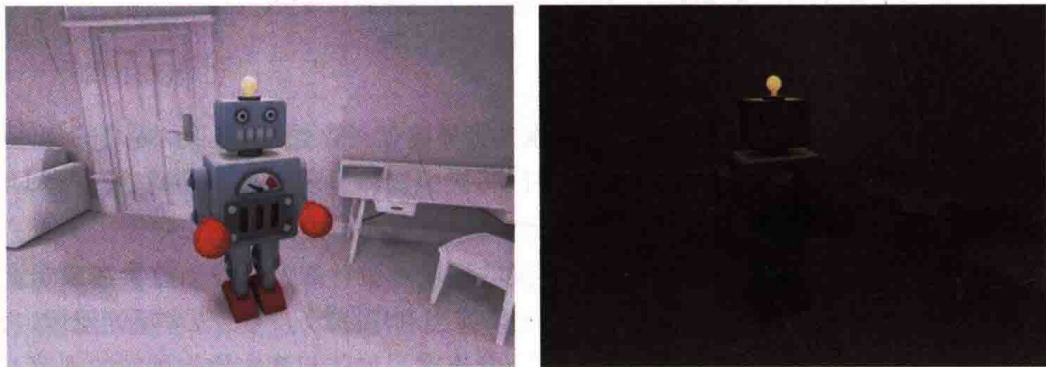


图 14-41 适度地进行补光可以营造更好的真实感

使用光照估计融合虚拟物体与真实环境,可以比较有效地解决虚拟物体与真实环境光照不一致的问题,防止在昏暗的环境中虚拟物体渲染得太亮或者在明亮的环境中虚拟物体渲染过暗的问题,如图 14-42 所示。



图 14-42 真实环境光照与虚拟光照不一致会破坏真实感

5. 阴影

在 AR 中,需要一个阴影平面来接受阴影渲染,阴影平面通常位于模型下方,该平面只负责渲染阴影,本身没有纹理。使用阴影平面渲染阴影是强化三维立体效果最简单有效的方式,阴影可以实时计算也可以是预先烘焙,模型有阴影后立体感觉会更强烈并且可以有效地避免

虚拟物体的飘浮感,如图 14-43 所示,左图添加阴影效果后模型的真实感大幅提升。



图 14-43 阴影的正确使用能营造三维立体感和增强可信度

6. 真实感

在 AR 体验中,应当想办法将虚拟对象融入真实环境中,充分营造真实、逼真的物体形象。使用阴影、光照、遮挡、反射、碰撞等技术手段将虚拟物体呈现于真实环境中,可大大地增强虚拟物体的可信度,提高体验效果。

在 AR 场景中,虚拟物体的表现应当与真实的环境一致,如台灯应当放置在桌子上而不是悬浮在空中,而且也不应该出现漂移。虚拟物体应当利用阴影、光照、环境光遮罩、碰撞、物理作用、环境光反射等模拟真实物体的表现,营造虚拟物体与真实物体相同的观感,如在桌子上的球滚动下落到地板上时,应当充分利用阴影、物理仿真来模拟空间与反弹效果,让虚拟物体看起来更真实。

真实感的另一方面是虚拟对象与真实环境的交互设计,良好的交互设计可以让虚拟对象看起来像真的存在于真实世界中一样,从而提升沉浸体验,在 AR 体验中,可以通过虚拟对象对阴影、光照、环境遮挡、物理和反射变化的反应来模拟物体的存在感,当虚拟物体能对现实世界环境变化做出反应时将会大大地提高虚拟物体的可信度,如图 14-44 所示,虚拟狮子对真实环境中灯光的实时反应可以显著提升其真实感。



图 14-44 虚拟对象对真实环境做出适当反应能有效地提升其真实感

14.3.7 三维音效

人脑除了依赖视觉信息感知周边环境外,还利用双耳对声音进行三维定位,在 AR 应用中,也可以充分利用三维空间音效营造更加沉浸式的虚实体验,扩大使用者的感知范围,提供传统应用无法提供的独特体验。三维音效可使人机交互变得更自然和自信,使虚拟元素的空间立体感更饱满,提供关于目标对象的其他额外信息,吸引使用者注意力,提供更强烈的感知冲击力。使用声频还可以鼓励用户参与并增强体验。

声频可以帮助构建身临其境的 360° 环绕沉浸体验,但需要注意的是要确保声音可增强这种体验,而不是分散注意力。在发生碰撞时使用声音效果或振动是确认虚拟对象与物理表面或其他虚拟对象接触的好方法,在沉浸式游戏中,背景音乐可以帮助使用者融入虚拟世界。

1. 三维音效所扮的角色

在 AR 应用中,三维音效也扮演着重要的角色。

1) 提高置信度

在现实生活中,通过融合各类感知器官对环境的感知信息整合形成最终环境印象,如通过视觉获取物体表面的凸凹光照信息,通过抚摸获取其粗糙不平的触觉信息,通过双耳获取声音在粗糙平面的混音信息,最终形成完整的对物体表面的印象。当从不同感知器官获取的信息不匹配时,如看起来凸凹不平的表面摸起来却很光滑,大脑的先验信息会受到质疑,使我们对物体的属性产生怀疑,这也说明大脑获取的各类感知信息会相互印证,印证通过后会确认物体的真实性。在 AR 应用中,如果来自视觉与来自听觉的感知信息能够相互印证,这会帮助人们确信眼前一切的真实性,提高虚实融合的置信度。

2) 扩大感知范围

通常人们只会关注其视野范围内的对象,但也会通过声音感知不在视野范围内的事件,这对于全场景感知的 AR 应用非常重要,如在游戏中,可以通过三维音效提醒玩家其他方向的敌人正在靠近,这是只凭视觉无法完成的任务。

3) 提供旁白

当使用者关注某个对象时,可以通过旁白提供该对象的更详细信息,这对某些文旅类、观展类应用非常有帮助,会大大地提高信息的使用效率。

2. 声频设计

与基于矩形框屏幕的应用相比,在 AR 应用中声频的设计与使用完全不一样。在 AR 应用中,声频设计的最一般原则是满足用户期望,成功的声频运用可以大大提高虚拟对象的可信度和真实感。

1) 空间音效

在 AR 应用中,很多时候需要使用空间音效(三维音效),如果一个人形机器人正在移动,则该机器人应当发出特定的声音,如果声音缺失就会造成大脑感知信息的不一致,从而产生

对机器人真实性的质疑,而且不同于传统屏幕应用,在AR中,使用者可以从不同距离、不同角度、不同方向对机器人进行观察,因此音效也应当是空间性的,这就意味着需要对机器人发声部位进行认真设计。

2) 使用多声源

由于声音的空间特性,所有发声的部位都应当设置声源,以确保声音的可信性,但过多的声源又会造成混乱。

3) 与视觉同步

声音效果与视觉效果应当紧密同步,每个动画和基于物理的运动都应当关注声音与视觉的同步。

3. 混音设计

在AR应用中,开发人员无法预测使用者观察虚拟对象的距离、视角,但仍然需要考虑各种可能性,以确保在各种情况下都能有令人信服的用户体验。

1) 衰减设置

对声源进行不同距离、角度的测试,力争音效的衰减效果能满足大部分使用场景,设置最大音效范围和衰减速率非常重要,如一只蜜蜂的声音衰减要比一头大象的脚步声衰减快得多。在某些场景中,声音的开始与结束不应突兀地出现或者消失(如旁白),一个渐变的过程更符合自然。

2) 层次感

对于多声源的对象,声音应当有层次感,如深沉的低频声源衰减得更慢,而尖细的声源衰减得更快,因而从远处只能听到主要的声源声音,而随着距离的接近,各类声源会加入进来从而丰富整个声场,给人丰富的声音层次感体验。

3) 聚束混音

聚束混音(Spotlight Mixing)是对空间声音计算的一种动态混合技术,当场景中的声源逐渐增多时,声音的混合会出现问题,聚束混音就是为了解决这类问题,它考虑了用户的姿势、眼睛注视或控制方向(实际上是移动设备姿态),以确定用户在任何给定时刻最感兴趣的内容并实时动态地调整音量,类似于声频聚光灯,聚光灯内的声源比它外面声源的音量大。

设计良好的音效可以满足用户的期望并扩大使用者的感知范围,增强虚拟元素的可信度,但若设计不好,如声音太小则达不到效果,太大则会让人反感,保持某种平衡非常重要,通常而言,音量保持在70%左右的水平能满足大部分使用场景。

第 15 章



性能优化

性能优化是一个非常宽泛的主题，但性能优化又是必须探讨的主题，特别是对手机、平板这种移动式、资源有限的设备，性能优化起着举足轻重的作用，甚至可能会决定应用的成败，但性能优化又是一个庞大的主题，要想深入理解性能优化需要对计算机图形渲染管线、CPU/GPU 协作方式、内存管理、计算机体系架构、代码优化有很好的掌握，这样才能在实践中逐步加深这个理解从而更好地优化性能。本章将对利用 ARCore 开发 AR 应用的性能优化内容与技巧进行阐述。

15.1 性能优化基础

性能优化是一个非常综合、涉及技术众多的大主题，从计算机出现以来，人们就一直在积极追求降低内存使用、提高单位指令性能、降低功耗、提高代码效率。虽然随着硬件技术特别是 CPU 性能按照摩尔定律飞速发展，PC 上一般应用代码的优化显得不再那么苛刻，但是对计算密集型应用而言，如游戏、实时三维仿真、AR/VR/MR，优化仍然是一个在设计、架构、开发、代码编写各阶段都需要重点关注的事项。相对于 PC，移动设备在硬件性能上还有比较大的差距，而且其 CPU、GPU 等设计架构与 PC 完全不同，能使用的计算资源、存储资源、带宽都非常有限，功耗要求也更严格，因此，AR 应用的优化显得尤为重要。

AR 应用与游戏相似，是属于对 CPU、GPU、内存重度依赖的计算密集型应用，因此在开发 AR 应用时，需要特别关注性能优化，否则可能会出现卡顿、反应慢、掉帧、虚拟场景漂移等问题，导致 AR 体验变差甚至完全无法正常使用。

15.1.1 影响性能的主要因素

从 CPU 架构来讲，移动设备 ARM 架构与 PC 常见的 x86、MIPS 架构有很大的不同。从处理器资源、内存资源、带宽资源到散热，移动设备相对来讲都受到更多的限制，因此移动设备的应用都要尽可能地使用更少的硬件及带宽资源。

从 GPU 架构来讲，目前移动设备使用最多的 GPU 包括 PowerVR、Adreno、Mali、Tegra，

它们之间的技术也不太相同，有的采用 Tiled-based Deferred Rendering（基于瓦片的延时渲染），有的采用 Early-Z，同时，在减少 Overdraw（多次绘制）方面也存在很大的差异。

从移动设备多样性来讲，移动设备相比 PC 更加复杂多样，各类 CPU、GPU、分辨率，各式各样的附加设施（深度传感器、黑白摄像头、彩色摄像头、结构光、ToF 等），复杂的操作系统版本。正是这些差异，针对不同芯片、不同操作系统的移动设备性能优化要更加复杂，也带来了更大的挑战。

对 AR 应用而言，需要综合利用 CPU、GPU 并让应用在预期的分辨率下保持一定的帧速，这个帧速至少要在每秒 30 帧以上才能让人感觉到流畅不卡顿，其中 CPU 主要负责场景加载、物理计算、特征点提取、运动跟踪、光照估计等工作，而 GPU 主要负责虚拟物体渲染、更新、特效处理等工作。具体来讲，影响 AR 应用性能的主要原因如表 15-1 所示。

表 15-1 影响 AR 应用性能的主要原因

类 型	描 述
CPU	过多的 drawcall 复杂的脚本计算或者特效
GPU	过于复杂的模型、过多的顶点、过多的片元 过多的逐顶点计算、过多的逐片元计算 复杂的 Shader、显示特效
带宽	大尺寸、高精度、未压缩的纹理 高精度的帧缓存
设备	高分辨率的显示 高分辨率的摄像头

了解了主要制约因素，就可以有针对性地进行优化，对照优化措施如表 15-2 所示。

表 15-2 性能优化的主要措施

类 型	描 述
CPU	减少 drawcall，采用批处理技术 优化脚本计算或者尽量少使用特效，特别是全屏特效
GPU	优化模型、减少模型顶点数、减少模型片元数 使用 LoD（Level of Detail）技术 使用遮挡剔除（Occlusion Culling）技术 控制透明混合、减少实时光照 控制特效使用、精减 Shader 计算
带宽	减少纹理尺寸及精度 合理缓存
设备	利用分辨率缩放 对摄像头获取数据进行压缩

表 15-2 所列是通用的和一般的优化方案，但对具体应用需要具体分析，在优化之前需要找准性能瓶颈点，针对瓶颈点的优化才能取得事半功倍的效果，才能有效地提高帧速。由于

是在独立的 CPU 芯片上处理脚本计算、在 GPU 芯片上处理渲染，总的耗时不是 CPU 花费时间加 GPU 花费时间，而是两者中的较长者，这个认识很重要，这意味着如果 CPU 负荷重、处理任务重，光优化 Shader 着色器根本不会提高帧速率，如果 GPU 负荷重，则光优化脚本和 CPU 特效也根本无济于事，而且，AR 应用的不同部分在不同的情况下表现也不同，这意味着 AR 应用有时可能完全是由于脚本复杂而导致帧速低，而有时又是因为加载的模型复杂或者过多而减速，因此，要优化 AR 应用，首先需要知道所有的瓶颈在哪里，然后才能有针对性地进行优化，并且要对不同的目标机型进行特定的优化。

15.1.2 AR 应用常用调试方法

在 Unity 中使用 ARCore 开发 AR 应用时，通常需要把程序编译后下载到真机上运行试错，需要将开发计算机直接连接到真机进行联机排错，相对来讲没有调试其他应用方便，这是个耗时费力的工作，对存疑的地方按下面的方法进行处理能加快调试过程，方便查找问题原因、对结果进行分析，以便更有效地进行故障排除^①。

1. 控制台

将代码运行的中间结果输出到控制台是最常用的调试手段，即使在真机上运行程序也可以实时不间断地查看所有中间结果，这对理解程序内部执行或者查找出错点有很大的帮助，通常这也就是在真机上调试应用的最便捷直观的方式。

2. 写日志

有时可能不太方便真机直接连接开发计算机进行调试（如装在用户机上试运行的应用），这时写日志反而就成了最方便的方式，可以将原本输出到控制台上的信息保存到日志中，再通过网络通信将日志发回服务器，以便及时地了解应用在用户机上的试运行情况。除了将日志记录成文本文档格式，甚至还可以直接将运行情况写入服务器数据库中，更方便查询统计。

3. 弹出

除了将应用运行情况发送到控制台进行调试，也可以在必要的时候在真机上弹出运行情况报告，这种方式也可以查看应用的实时运行情况，但这种方式不宜弹出过频，应以弹出重要的关键信息为主，不然可能很快就会耗尽设备应用资源。通过在代码的关键位置弹出信息，可以帮助分析代码的运行流程，以便确定代码的关键部分是否正在运行及如何运行。

4. 代码断点

Unity 配合 VS 可以对 AR 应用脚本代码设置断点进行调试，设置断点调试方法可以精确

^① 目前 AR 应用调试还有很多不方便的地方，IDE 没有集成 AR 调试功能，模拟器使用复杂、问题多，真机调试耗时长、不方便等，但这些都是发展中的问题，AR 应用是全新的应用类型，各类 IDE、工具、新的调试方法随着时间的推移也会慢慢成熟。

跟踪脚本代码的执行过程，获取执行过程中的所有中间变量值。这是一种对程序员非常友好的调试方式，只是过程稍烦琐，而且调试周期长。

15.1.3 AR 应用性能优化的一般原则

对三维游戏开发应用的优化策略与技巧完全适用于 AR 应用开发，如静态批处理、动态批处理、LoD、光照烘焙等优化技术全部都可以应用到 AR 开发中，但 AR 应用运行在移动端设备上，这是比 PC 或者专用游戏机更苛刻和复杂的运行环境，而且边端设备的各硬件性能与 PC 或专用游戏机相比还有很大的差距，因此，对 AR 应用的优化比 PC 游戏要求更高。对移动设备性能进行优化也是一个广泛而庞大的主题，此处只取其中的几个代表性方面进行一般性阐述。

在 AR 应用开发中，应当充分认识边端设备软硬件的局限性，了解图形渲染管线，使用一些替代性策略来缓解计算压力，如将一些物理数学计算采用动画或预烘焙的形式模拟，将实时光照效果使用纹理的形式模拟等，对资源受限设备需要谨慎使用的技术及优化方法如表 15-3 所示。

表 15-3 资源有限设备需要谨慎使用的技术及优化方法

谨慎使用	替代或优化技术
全屏特效，如发光和景深	在对象上混合 Sprite 代替发光效果
动态地逐像素光照	只在主要角色上使用真正的凹凸贴图（Bumpmapping）；尽可能多地将对比度和细节烘焙进漫反射纹理贴图，将来自凹凸贴图的光照信息烘焙进纹理贴图
全局光照	关闭全局光，采用 Lightmaps 贴图，而不是直接使用光照计算；在角色上使用 Lightprobes，而不是真正的动态灯光照明；在角色上采用唯一的动态逐像素照明
实时阴影	使用伪阴影
光线追踪	使用纹理烘焙代替雾效；采用淡入淡出代替雾效
高密度粒子特效	使用 UV 纹理动画代替密集粒子特效
高精度模型	降低模型顶点数与面数；使用纹理来模拟细节
复杂特效	使用纹理 UV 动画替代，降低着色器的复杂度

总体而言，由于移动设备硬件性能限制，为获得更好的渲染效果和性能，在 AR 应用开发中，总渲染的三角形数以不超过 50 万个为宜，针对不同场景复杂度，具体如表 15-4 所示。

表 15-4 不同场景复杂度最大对象和三角形数量

	简单场景	中等场景	复杂场景
渲染对象数	1 ~ 3	4 ~ 10	> 10
每个对象三角形数	< 100 000	< 30 000	< 10 000
每个对象材质数	1 ~ 2	1 ~ 2	1 ~ 2

就图形渲染格式而言,相比 STEP、STL、BIM 等工业高精度模型,FBX、OBJ、GLB/glTF 2.0 等格式对渲染更友好,可以进行更多的优化,如顶点数量、顶点属性、法线等,可以在不明显影响渲染效果的情况下,将几万的顶点数优化到几千,对性能提升起到非常大的帮助。

性能优化不是最后一道工序而应该贯穿于 AR 应用开发的整个过程,并且优化也不仅是程序员的工作,它也是美工、策划的工作任务之一,如可以烘焙灯光时,美工应该制作烘焙内容而不是采用实时光照计算。

15.2 AR 应用性能调试工具

为帮助开发人员了解和分析 AR 应用运行时的性能情况,ARCore 提供了一个内建运行日志功能和性能分析面板,Unity 提供了内建性能分析工具(Unity Profiler)和外部性能分析工具(Unity Performance Reporting, UPR),利用这些性能调试分析工具可以从各个层面查看 AR 应用的性能情况。

15.2.1 ARCore 性能调试功能

ARCore 运行在 Android 操作系统平台上,也支持 ADB 命令(Android Debug Bridge),通过 ADB 工具可以将命令发送到 ARCore,支持运行日志记录、性能分析面板功能。Android SDK 提供了 ADB 工具,该工具也可以独立安装。在第 1 章中介绍并安装了 Android Studio 2020,使用该 IDE 安装 Android SDK 后,为了方便使用,可以将 ADB 工具路径添加到开发计算机环境变量中^①,如图 15-1 所示。

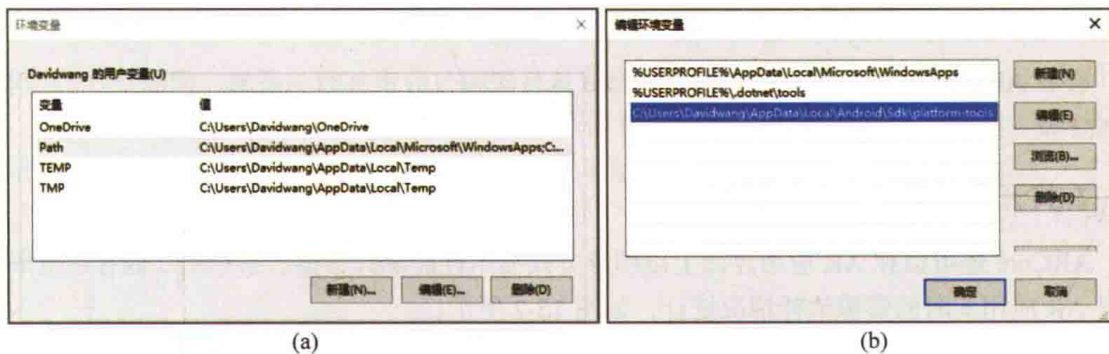


图 15-1 将 ADB 工具路径添加到系统环境变量中

完成环境变量添加后,将开发计算机通过 USB 连接到移动设备,打开命令行工具,通过输入 adb devices 命令进行测试,如果输出结果如下,则说明 ADB 工具功能正常(b17e1954 是设备串号,不同设备会不同)。

^① 以笔者计算机为例,默认 ADB 工具在 Android SDK 目录下,所在路径为 C:\Users\Davidwang\AppData\Local\Android\Sdk\platform-tools。

```
// 第 15 章 /15-1
C:\Users\Davidwang>adb devices
List of devices attached
b17e1954          device
```

1. 打开及关闭 ARCore 运行日志

当查看 ARCore 运行时行为和进行应用性能排查时,可以通过开启 ARCore 运行日志,将所有 ARCore 内部 API 调用信息输出,进行行为分析,在不需要时,可以关闭 ARCore 运行日志功能,以提高性能。

开启 ARCore 运行日志需要 Google Play Services for AR 版本为 V1.23 或以上,可以通过 ADB 命令检查版本信息,确保版本符合要求。

需要注意的是,使用 ADB 工具开启或者关闭 ARCore 运行日志需要 AR 应用处在运行状态 (AR 应用已启动并且为活跃状态),其基本命令如下:

```
// 第 15 章 /15-1
// 检查 ARCore 版本信息
adb shell pm dump com.google.ar.core | findstr /i "packages: versionName"

// 打开 ARCore 运行日志
adb shell am broadcast -a com.google.ar.core.ENABLE_API_CALL_LOGGING

// 关闭 ARCore 运行日志
adb shell am broadcast -a com.google.ar.core.DISABLE_API_CALL_LOGGING

// 查看日志输出
adb logcat -s ARCore-API
```

查看 ARCore 运行日志可以洞察 ARCore 运行流程,分析其行为逻辑,但需要比较专业的 SLAM 及三维渲染方面的知识储备。

2. 打开及关闭性能分析面板

ARCore 还可以在 AR 应用界面上以图形方式显示性能统计数据,如 CPU、内存、帧率等,提供 AR 应用实时的资源消耗情况统计,如图 15-2 所示。

同样,使用 ADB 工具开启或者关闭 ARCore 性能分析面板需要 AR 应用处在运行状态,使用 ADB 工具开启及关闭 ARCore 性能统计面板的命令如下:

```
// 第 15 章 /15-2
// 开启性能统计
adb shell am broadcast -a com.google.ar.core.ENABLE_PERFORMANCE_OVERLAY

// 关闭性能统计
adb shell am broadcast -a com.google.ar.core.DISABLE_PERFORMANCE_OVERLAY
```


ARCore 性能统计面板展示的数据非常直观,包括会话启动时长、内存消耗、Java 虚拟机占用的内存、帧率等,其中 Process CPU Usage 字段表示 AR 应用进程使用的 CPU 算力占设备总可用 CPU 算力的百分比,此值包括 AR 应用程序工作负载、ARCore 工作负载和应用程序使用的任何其他库负载总和。例如,如果 AR 应用程序进程在四核设备上运行时使单个 CPU 内核完全饱和,则该字段将显示 25% 的 CPU 使用率。

性能统计面板也提供了 ARCore 独立功能的 CPU 使用情况明细,这些独立功能包括运动跟踪、平面检测、深度 API、屏幕空间跟踪 (Instant Placement 和图像跟踪)。

通过 ARCore 性能分析面板,能直观地看到当前 AR 应用的各种资源消耗情况,非常方便进行性能问题排查。

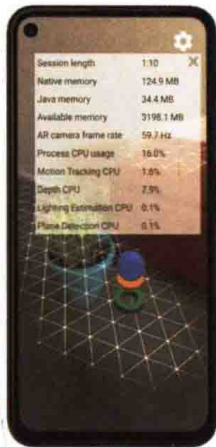


图 15-2 ARCore 性能统计数据面板示意图

注意

ARCore 在 AR 应用上弹出性能分析面板需要授权,因此需要预先在移动设备上开启该权限,通常是在 Android 系统:设置→应用管理→权限管理中找到 ARCore 或者 Google Play Services for AR 打开其悬浮面板功能特性进行授权,不同制造商的设备设置会略有不同。

15.2.2 Unity 分析器

除了程序逻辑运行正确,性能表现也是 AR 应用需要时刻重点关注的事项,Unity 强大的性能分析工具 Profiler 在性能分析调试中非常有帮助(下文中 Unity Profiler 译为 Unity 分析器)。

Unity 分析器可以提供应用性能表现的详细信息。当 AR 应用存在性能问题时,如低帧率或者高内存占用,性能分析工具可以帮助发现问题并协助解决问题。Unity 分析器是一个非常强大的性能剖析工具,不仅有利于分析性能瓶颈,也提供了窥视 Unity 内部各部分工作情况的一个机会,使用 Unity 分析器的步骤如下。

1. 打开分析器窗口

在 Unity 菜单中依次选择 Window → Analysis → Profiler,打开 Unity 分析器窗口,将窗口拖到 Game 选项卡旁边,以使其停靠在 Game 选项卡右侧(当然,可以把它放在任何地方)。

2. 设置远程调试

使用 Ctrl+Shift+B 组合键打开 Build Settings 对话框,勾选 Development Build、Autoconnect Profiler、Deep Profiling Support 复选框,启用自动连接和调试分析功能,如图 15-3 所示。

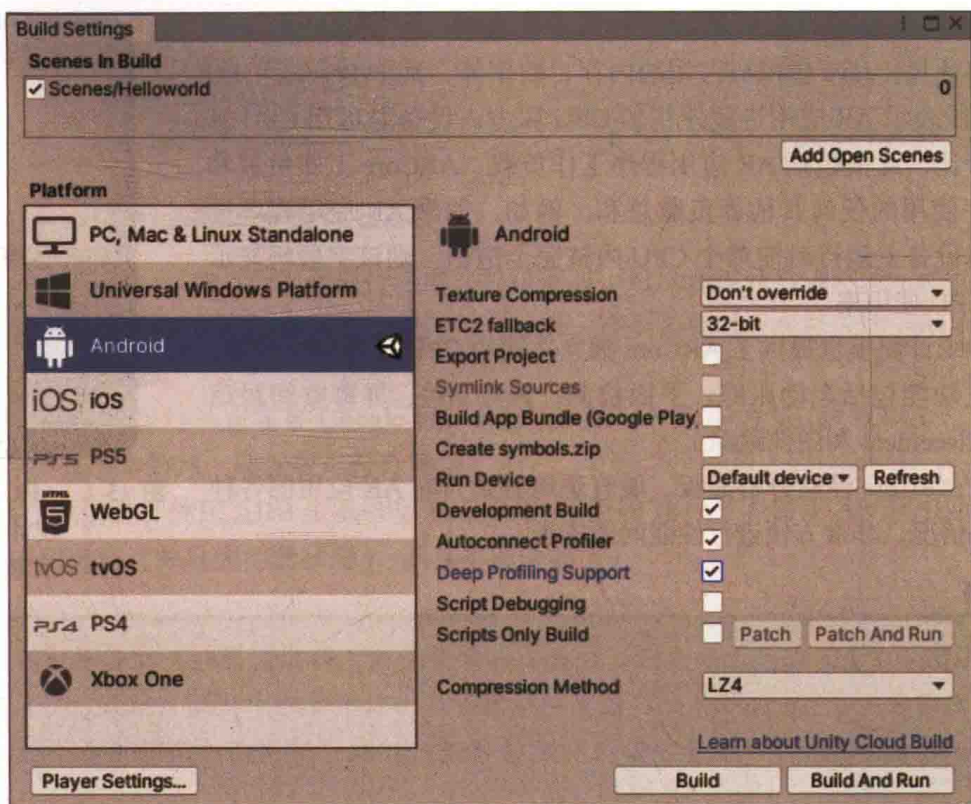


图 15-3 设置项目远程调试

3. 连接调试设备

将手机通过 USB 连接到开发计算机，选择 Build Settings 对话框中的 Build And Run 按钮，将应用下载到真机上启动运行，在 Unity 分析器 Editor 下拉菜单中选择真机设备，如图 15-4 所示，然后按下开始 (▶符号) 按钮开始捕获应用运行数据。



图 15-4 选择需要远程调试的设备

连接上真机后就可以在 Unity 分析器上看到分析器窗口的全貌了，如图 15-5 所示。

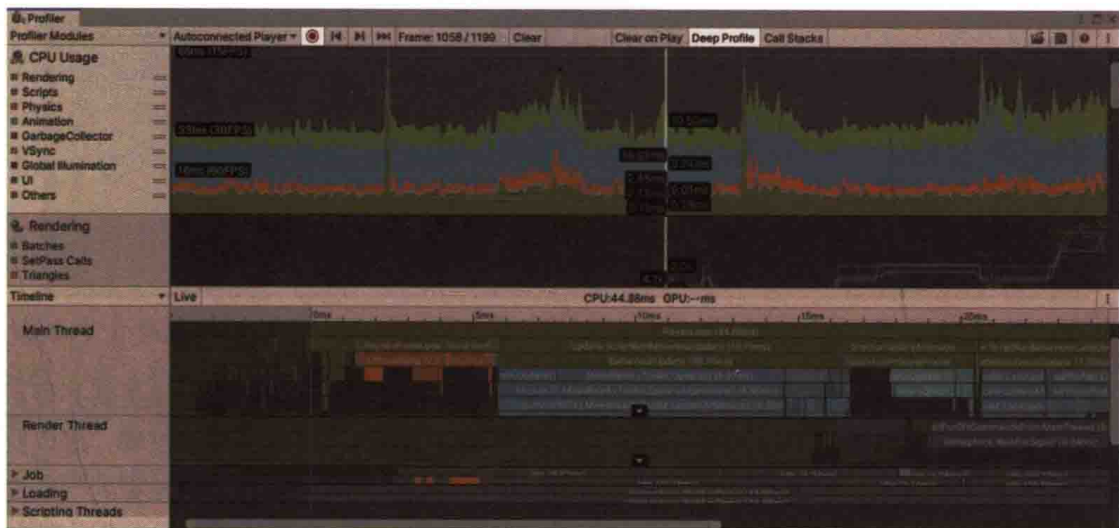


图 15-5 Unity 分析器窗口

如果在测试的时候 Unity 分析器连接不上手机设备，则在确保防火墙没有屏蔽连接的端口并且手机已打开开发者选项中调试功能的情况下，可能需要打开内部分析器（Internal Profiler），具体操作是在 Unity 菜单中，依次选择 Editor → Project Settings → Player，在属性窗口中依次选择 Android → Other Settings → Optimization，勾选 Enable Internal Profiler 属性后的复选框，如图 15-6 所示，正常情况下应该就能连接上手机设备了。

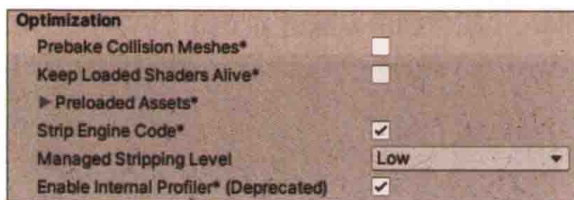


图 15-6 打开 Enable Internal Profiler 开关

Unity 分析器通过网络在开发计算机与 ARCore 设备间进行通信，因此需要开发计算机与真机设备连接到同一个局域网（使用 WiFi 连接时），另外，为确保连接正常，需要确保它们均开启开发者选项。

Unity 分析器可以提供应用程序不同部分运行情况的深度信息。使用分析器可以了解到性能优化的不同方面，例如应用如何使用内存、每个任务组消耗了多少 CPU 时间、物理运算执行频率等。最重要的是可以利用这些数据找到引起性能问题的原因，并且测试解决方案的有效性。

在 Unity 分析器窗口左侧，可以看到一系列子分析器（sub Profilers），每个子分析器显示应用程序运行时某一方面的信息，分别为 CPU 使用情况、GPU 使用情况、渲染、内存使用情况、声音、物理和网络等，如图 15-7 所示。

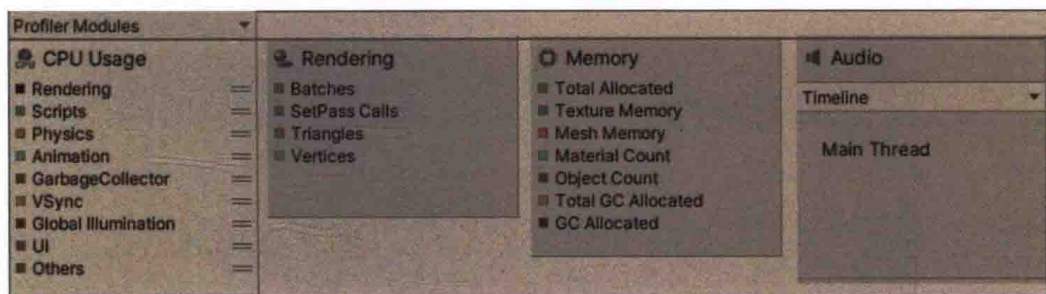


图 15-7 Unity 分析器中的各类子分析器

在开始录制性能数据后，窗口上部的每个子分析器都会随着时间推移不断地更新并显示数据。性能数据随着时间发生变化，是一个动态的连续变化过程，通过观察这个过程，可以获取很多仅仅凭一帧性能数据分析无法提供的信息。通过观察曲线变化，可以清楚地看到哪些性能问题是持续性的，哪些问题仅仅在某一帧中出现，还有哪些性能问题是随着时间逐渐显现的。

选择某一个子分析器模块时，窗口下半部会显示当前所选择分析器的当前帧的详细信息，此处显示的数据依赖于所选择的子分析器。例如，当选中内存分析器时，该区域显示应用运行时内存和总内存占用等与内存相关数据；如果选中渲染分析器，则会显示被渲染的对象数量或者渲染操作执行次数等数据。

这些子分析器会提供应用运行时详尽的性能数据，但是我们并不总是需要使用所有的这些子分析器模块，事实上，通常在分析应用性能时只需观察一个或者两个子分析器。例如，当游戏应用运行得比较慢时，可以先查看 CPU 分析器，如图 15-8 所示。



图 15-8 CPU 使用分析器

CPU 分析器提供了设备 CPU 使用情况总览，可以观察到应用各部分占用 CPU 时间的情况。该分析器可以只显示特定应用部分对 CPU 的占用，通过单击 CPU 分析器窗口左侧的任务组颜色图标可以打开或关闭对应任务组的数据显示。针对各任务组的 CPU 占用情况，可以通过查看对应子分析器获取更详尽的信息，例如发现渲染占用了很长时间，就可以查看渲染分析器模块去获取更多的详细信息。

也可以关闭一些不关心的子分析器，通过单击 Unity 分析器子分析器模块右上角的“x”符号按钮关闭该模块，在需要时也可以随时通过单击分析器窗口左上角的 Add Profiler 按钮添加想要的子分析器模块。添加删除子分析器的操作不会清除从设备获取的性能数据，仅仅是显示或者隐藏相应分析模块而已。

Unity 分析器窗口的顶部包含一组控制按钮，可以使用这些按钮控制性能分析的开始、停止和浏览收集的数据，如图 15-4 所示。一个典型的分析流程如下：开始分析应用进程，当应用出现性能问题时，暂停分析，然后通过时间线控制，逐帧地找到显示出性能问题的帧，该帧的详细信息会在下半部窗口中显示。

15.2.3 帧调试器

Unity 分析器是一个强大的性能分析工具，除此之外，Unity 还提供了用于单帧分析的帧调试器（Frame Debug），帧调试器允许将正在运行的应用在特定的帧冻结回放，并查看用于渲染该帧的单次绘制调用。除列出 Drawcall 外，调试器还允许逐个地遍历它们，一个 Drawcall 一个 Drawcall 地渲染，这样就可以非常详细地看到场景如何一步一步绘制出来。

使用帧调试器的流程与使用 Unity 分析器的流程基本一致，具体流程如下。

1. 打开 Debug 窗口

在 Unity 菜单中，依次选择 Window → Analysis → Frame Debug，打开帧调试器窗口，将该窗口拖到 Game 选项卡旁边，使其停靠在 Game 选项卡右侧。

2. 选择调试设备

当 AR 应用真机设备上运行后，由于在构建应用时勾选了 Autoconnect Profiler 复选框，应用会自动连接到帧调试器，按下帧调试器左上角的 Enable 按钮（单击 Enable 按钮时会暂停应用）捕获当前帧数据，这时帧调试器窗口中将会加载应用程序渲染该帧时的相关信息，如图 15-9 所示。

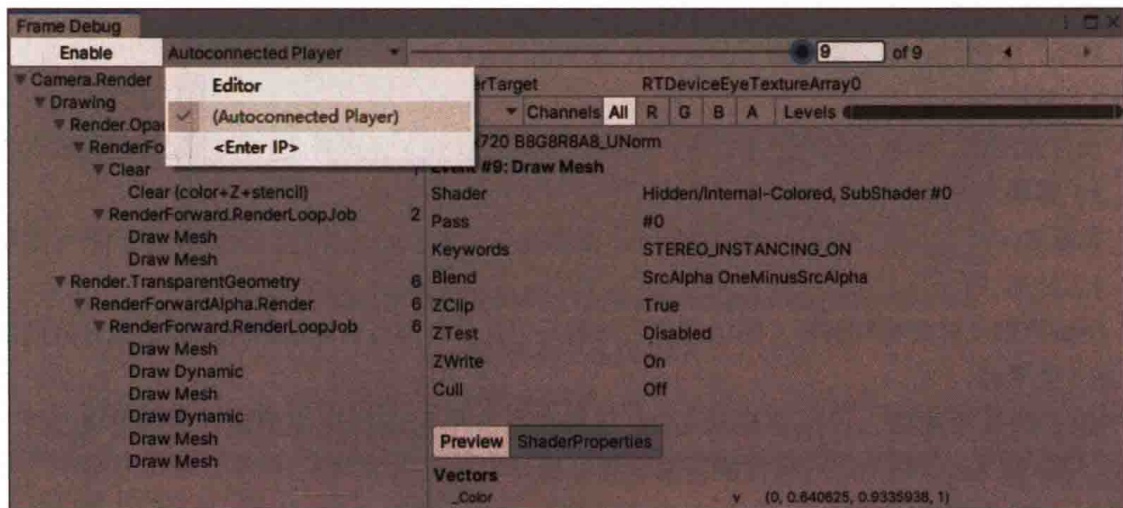


图 15-9 帧调试器窗口

帧调试器窗口左侧主列表以树形层次结构的形式显示了 Drawcalls 序列 (及其他事件, 如 framebuffer clear), 该层次结构标识了调用的起源。列表右侧的面板提供了关于 Drawcall 的更多详细信息, 例如用于渲染的几何网格细节和着色器。

单击左侧列表中的一个项目将在右侧显示该项目的详细 Drawcall 信息, 包括着色器中的参数信息 (对每个属性都会显示值, 以及它在哪个着色器阶段 “顶点、片元、几何、hull、domain” 中被使用)。

工具栏中的左右箭头按钮表示向前和向后移动一帧, 也可以使用键盘方向按键达到同样的效果。此外, 窗口顶部的滑块允许在 Drawcall 调用中快速 “擦除”, 从而快速找到感兴趣的点。

帧调试器右侧窗口顶部是一个工具栏, 分别显示红、绿、蓝和 Alpha 通道信息, 以方便查看应用的当前状态。类似地, 可以使用这些通道按钮右侧的 Levels 滑块, 根据亮度级别隔离视图的区域, 但这个功能只有在渲染到渲染纹理时才能启用。

15.2.4 UPR 分析器

1. UPR 分析器的优势

Unity UPR (Unity Performance Reporting, Unity 性能报告) 是 Unity 官方提供的专业在线性能优化工具, 旨在帮助开发者诊断与优化项目开发过程中存在的性能问题, 提供零侵入、方便易用的测试流程。UPR 基于 Unity 性能分析工具 (Unity Profiler), 直接收集运行在目标硬件上的测试包所产生的分析数据, 相较于 Unity 性能分析器, UPR 以更直观的方式, 提供更多的比较维度和追踪角度, 展示更多的内容, 方便测试和开发人员更快地定位性能瓶颈, 同时, 还具有以下优势。

1) 全面

提供全面的工具链, 收集包括运行时 Unity 性能分析工具数据、静态资源、内存、GPU、Lua 等各个维度的数据, 更全面地反映项目性能状况和瓶颈。

2) 易用

全自助测试, 甚至无须开发人员介入, 测试人员可以独立完成各个阶段的配置与测试过程。

3) 零侵入

无须 ROOT 或集成 SDK, 测试过程简单无依赖。无须上传 APK 包, 有利于保护项目隐私。

4) 极速

测试数据实时传输和解析, 测试完成后只需等待几分钟即可获取全面详细的性能分析报告。

5) 全平台

通过 UPR 桌面版, 可以实现对 Unity 支持的所有平台进行性能测试, 通过移动版 UPR App, 可以对 AR 应用的运行数据进行采集和分析。

6) 自动化支持

UPR 桌面端的 CLI 工具, 可以将 UPR 集成到自动化开发和测试流程中。

2. UPR 分析器的使用步骤

UPR 通过开发构建 (Development Build) 版本的应用程序内置 Unity 性能分析器采集应用运行时数据, 并将此类数据发送到其分析网站进行解析和处理, 生成各维度、各阶段的性能报告, 因此, 在 AR 应用中使用 UPR 的一般步骤如下。

1) 使用准备

通过访问 UPR 网站 (<https://upr.unity.cn>) 注册或登录 Unity 账号, 单击“现在开始使用”按钮创建 UPR 测试项目, 正确填写项目信息, 其中“项目包名”必须与待测项目的 APK 包名保持一致; “Unity 版本号”必须与待测项目所使用的 Unity 版本保持一致。

2) 下载并安装 UPR App

下载并安装 UPR App (下载网址为 <https://upr.unity.cn/download>), 该 App 负责采集 AR 应用性能数据并发送到 UPR 网站, 由于在使用时该 App 会将控制按钮悬浮于 AR 应用界面之上, 因此需要先开启其悬浮框权限^①。

3) 构建并安装 AR 应用

使用 Development Build 方式构建 AR 应用并安装到真机设备, AR 应用无须也不能提前运行, UPR App 使用时会自动调起 AR 应用。

4) 创建测试

在 UPR 网站单击第一步中创建的 UPR 测试项目开始测试“>”符号, 通过 UPR App 应用扫描弹出的二维码获取待测试项目信息, 单击“开始测试”后会调起需要进行测试的 AR 应用, AR 应用启动后, UPR 会在应用界面上显示控制按钮用于开始、暂停、停止测试等功能, 如图 15-10 所示。



图 15-10 使用 UPR

在测试结束后, 返回 UPR 官方网站即可查看各项性能分析报告。UPR 还提供了 Unity 资源 (Asset) 检测、Asset Bundle 冗余检测、静态代码检测及各类实用工具集, 详细内容可查阅官方文档。

^① 以小米手机为例, 依次打开设置→应用设置→授权管理→应用权限管理→UPR, 开启“显示悬浮窗”权限。

15.3 Unity 分析器使用

Unity 分析器 (Unity Profiler) 是非常强大的性能分析工具, 它可对 CPU 使用、GPU 使用、渲染、内存使用、UI、网络通信、音频、视频、物理仿真、全局光照进行实时查看, 非常有助于发现性能瓶颈, 为分析性能问题提供有力支持。Unity 分析器包含很多子分析器, 通过每个子分析器针对某一方面情况进行更深入的分析。

15.3.1 CPU 使用情况分析器

CPU 使用情况分析器 (CPU Usage Profiler, 下文简称为 CPU 分析器) 是对应用运行时设备上 CPU 使用情况进行实时统计的分析器。CPU 分析器以分组的形式对各项任务建立逻辑组, 即左侧列表中的 Rendering、Scripts、Physics、Animation 组等, 如图 15-11 所示, 其中 Others 组记录了不属于渲染的脚本、物理、垃圾回收及 VSync 垂直同步的总数, 还包括动画、AI、音频、粒子、网络、加载和 PlayerLoop 数据 (其他分析器与此一致)。CPU 分析器采用分组的形式对属于该组的信息进行独立统计计算, 更直观地显示出各任务组对 CPU 占用比, 可以方便地勾选或者不勾选某个 / 某些任务组, 以简化分析曲线。



图 15-11 CPU 分析器可以选择左侧的任务组显示特定任务组的统计曲线

在分析器窗口左侧选择 CPU 分析器后, 下方窗格将显示详细的分层时间数据, 数据显示模式可以是时间线 (Timeline)、层次结构 (Hierarchy)、原始层次结构 (Raw Hierarchy), 通过左侧下拉菜单选择, 如图 15-12 所示。



图 15-12 CPU 分析器详细信息可以通过选择 Timeline、Hierarchy、Raw Hierarchy 方式显示

当以层次结构或原始层次结构显示时, 各列属性含义如表 15-5 所示, 在下方窗格右侧的下拉菜单中, 还可以选择无细节 (No Details)、显示细节 (Show Related Objects)、显示 Drawcall (Show Calls) 以便进行更加深入的细节检查。

表 15-5 CPU 分析器属性列含义

列名称	描 述
Self 列	指在特定函数中花费的时间量，不包括调用子函数所花费的时间
Time ms 和 Self ms 列	显示相同的信息，以毫秒为单位
Calls 列	指 Drawcall 调用次数
GC Alloc 列	显示当前帧占用的将要在后面被垃圾回收器回收的存储空间，此值最好保持为 0

CPU 分析器还可以进行物理标记、性能问题检测告警、内存性能分析、时间轴高亮细节等功能，是进行性能分析时使用得最多的分析器。

15.3.2 渲染情况分析器

渲染情况分析器（Rendering Profiler）主要对图形渲染情况进行统计计算，包括 Drawcalls、动态批处理、静态批处理、纹理、阴影、顶点数、三角面数等，使用界面如图 15-13 所示。

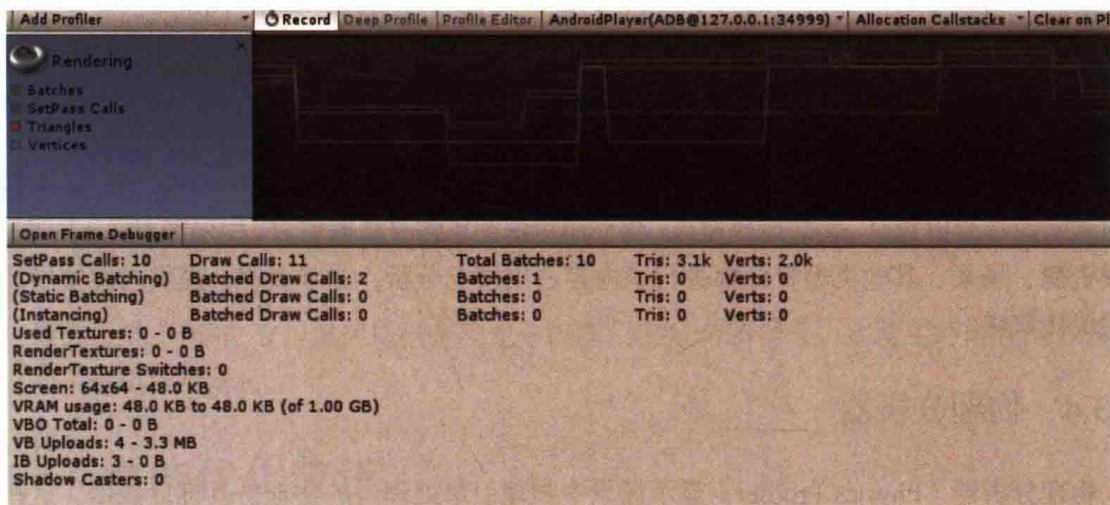


图 15-13 渲染分析器窗口

渲染分析器中一些统计信息与 Unity 编辑器 Stats 渲染统计信息窗口中显示的统计信息非常接近。在使用渲染分析器进行分析时，在某个特定时间点，也可以通过单击 Open Frame Debugger 按钮打开帧调试器，以便对某一帧进行更加深入的分析。

15.3.3 内存使用情况分析器

内存使用情况分析器（Memory Profiler）提供了两种对应用内存使用情况的查看模式，即简单模式（Simple）和细节模式（Details），可以通过其下方窗格左上角的下拉列表进行选择，如图 15-14 所示。

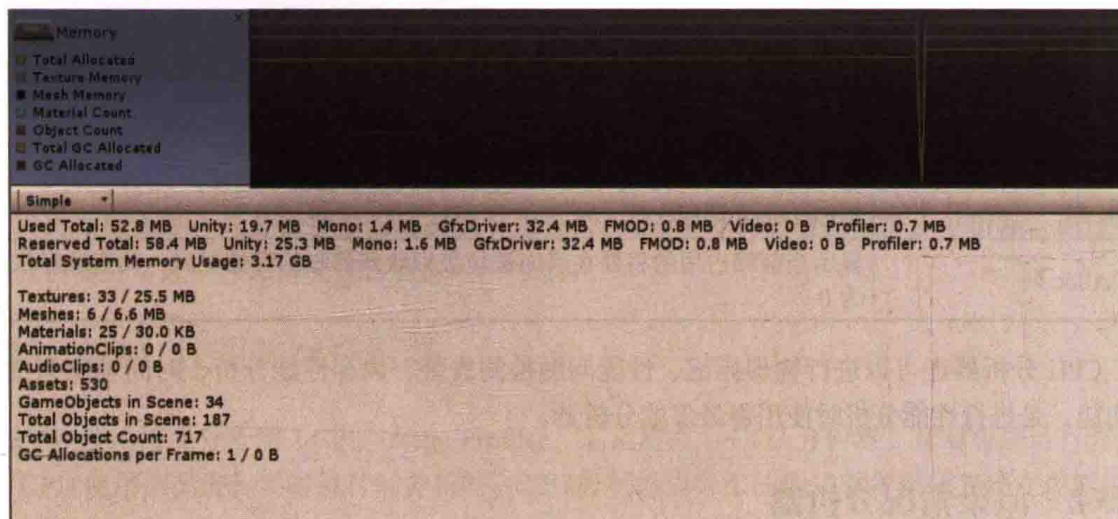


图 15-14 以 Simple 方式查看内存分析器

简单查看模式只简单显示应用程序在真实设备上每帧所占用的内存情况，包括纹理、网格、动画、材质等对内存占用的大小。从图 15-14 中也可以看到，Unity 保留了一个预分配的内存池，以避免频繁地向操作系统申请内存。内存使用情况包括为 Unity 代码分配的内存量、Mono 托管代码内存量（主要是垃圾回收器）、GfxDriver 驱动程序（纹理、渲染目标、着色器）、FMOD 音频驱动程序的内存量、分析器内存量等。

详细查看模式允许获取当前设备使用内存情况的快照。可以通过单击 Take Sample 按钮捕获详细的内存使用情况，在获取快照后，将使用树形图更新分析器窗口，树形图对 Assets 资源、内置资源、场景、其他类型资源都进行了内存占用情况分析，非常直观，提供内存使用情况深度统计数据。

15.3.4 物理分析器

物理分析器（Physics Profiler）显示场景中物理引擎处理过的物理仿真统计数据，这些数据有助于诊断和解决与场景中物理仿真相关的性能问题，使用界面如图 15-15 所示。

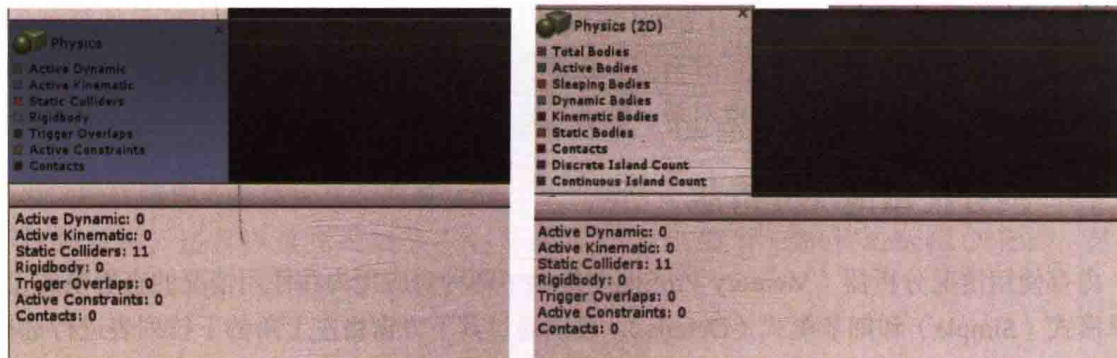


图 15-15 物理分析器窗口

物理引擎有非常多的专业术语，使用物理分析器时主要的术语含义如表 15-6 所示。

表 15-6 物理分析器术语含义

术 语	描 述
Active Dynamic（活跃动态体）	活动状态的非运动学刚体组件
Active Kinematic（活跃运动学刚体）	活动状态的运动学刚体组件，带关节的运动学刚体组件可能会在同一帧出现多次
Static Colliders（静态碰撞体）	没有刚体组件的静态碰撞体
Rigidbody（刚体）	由物理引擎处理的活动状态刚体组件
Trigger Overlaps（触发器重叠）	重叠的触发器
Active Constraints（激活的约束）	由物理引擎处理的原始约束数量
Contacts（接触）	场景中所有碰撞体的接触对总数

Unity 中，物理模拟运行在与主逻辑更新循环不同的固定频率更新周期上，并且只能通过每次调用 `Time.fixedDeltaTime` 来步进时间，这类似于 `Update` 和 `FixedUpdate` 之间的差异。在使用物理分析器时，如果当前有大量的复杂逻辑或图形帧更新导致物理模拟需要很长时间，物理分析器则需要多次调用物理引擎从而有可能导致物理模拟暂停。

15.3.5 音视频分析器

音频视频分析器（Audio Profiler, Video Profiler）是对应用中声频与视频使用情况进行统计计算的分析器。声频分析器窗口面板中有监视声频系统的重要性能数据，如总负载和语音计数，还展示了有关声频系统各部分的详细信息。视频分析器与声频分析器差不多，使用起来也很直观。

除此之外，Unity 分析器还包括全局光分析器、GPU 分析器等，在使用时可查阅 Unity 官方文档了解更详细情况。

15.4 性能优化流程

对 AR 应用而言，引起掉帧、卡顿、漂移问题的原因可能有上百个，模型过大过多、纹理精度高、特效频繁、网络传输数据量过大等都有可能导致用户使用体验的下降。解决性能问题首先要找到引起性能问题的最主要原因，通常遵循以下流程：

（1）收集所有性能数据。彻底了解应用程序的性能，收集来自多个渠道的性能数据，包括 ARCore 性能分析数据、Unity 和 VS 性能调试数据、UPR 性能分析数据、测试人员测试体验反馈、使用用户对 AR 应用的反馈、AR 应用运行异常报告等。

（2）查找并确定引起性能问题的主要原因。利用上一阶段获取的数据信息，结合应用的性能目标与使用预期，找出影响性能问题的瓶颈。

（3）对主要性能问题进行针对性修改完善。针对查找到的问题，有针对性地进行优化。

（4）对照性能表现反馈查看修改效果。对比修改后的性能数据与原始数据，以确定修改

是否有效,直到达到预期效果。

(5) 修改完善下一个性能问题。按照上述流程继续迭代对下一个主要性能问题进行优化。性能提升是一个循环过程,需要不断迭代完善直至满足要求,如图 15-16 所示。

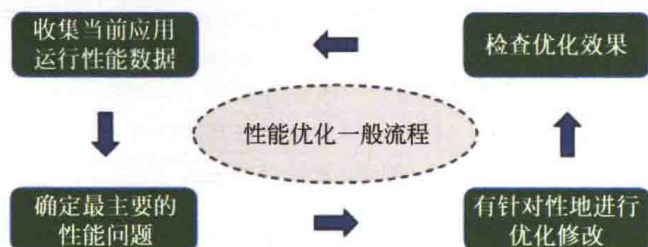


图 15-16 性能优化一般流程

对 AR 应用而言,最少化资源使用有助于提高用户对应用的好感和提升用户的体验,具体而言:

(1) 降低 AR 应用的启动加载时间,这不仅有助于提高用户体验,还能防止用户对应用假死的误判。

(2) 降低内存使用,这可以减少垃圾回收系统在后台频繁地清理内存垃圾,提高响应速度。

(3) 降低频繁的外存储器读写,这可以提高响应速度。

(4) 降低电池消耗,对非必须的能源消耗大的特性在不使用时及时关闭,如平面检测、环境理解、锚点等功能。

在 AR 应用开发完成后,即使在测试机上没有出现性能问题,也应当进行一次性能优化流程,一方面进一步提升性能,另一方面防止非正常情况下性能的恶化。

AR 应用不同于一般桌面应用,用户对桌面应用的耐受性比较高,如执行一个复杂统计,用户有足够的耐心等待 3s,而 AR 应用卡顿 3s 则是完全不可接受的。对 AR 应用来讲,帧率要在 30 帧/秒以上才可以接受,低于这个值就会让人感觉有延时、卡顿、操作反应迟钝。

AR 应用与游戏一样,对性能要求高,特别是对单帧执行时间有严格限制,按最低 30 帧/秒计算,每帧的执行时间必须限制在 33.3ms 内,若超时必将导致掉帧。

帧率 (Frame Rate) 是 AR 应用流畅度的重要衡量指标,也是衡量应用性能的基本指标,一帧类似于动画中的一个画面,一帧图像就是 AR 应用绘制到屏幕上的一个静止画面,绘制一帧到屏幕上也称为渲染一帧。帧率通常用 FPS (Frame Per Second) 衡量,提高帧率就要降低每帧的渲染时间,即渲染每帧画面所需要的毫秒数,这个毫秒数有助于指导我们优化性能表现。

渲染一帧图像,AR 应用需要执行很多不同的任务。简单地说,Unity 必须更新应用的状态,获取应用的快照并且把快照绘制到屏幕上。有些任务需要每帧都执行,包括读取用户输入、执行脚本、运行光照计算等,还有许多操作需要每帧执行多次,如物理运算。当所有这些任务都执行得足够快时,AR 应用才能有稳定且可接受的帧率,当这些任务中的一个或者多个执行得不够快时,渲染一帧将花费超过预期的时间,帧率就会因此下降。

因此知道哪些任务执行时间过长,对解决性能瓶颈问题就显得非常关键。一旦知道了哪

些任务降低了帧率,就可以有针对性地去优化应用的这一部分,而如何获取这些任务执行时的性能表现则是性能分析器的工作。

AR 应用非常类似于移动端游戏应用,因此,移动游戏应用中的那些优化策略和技巧完全适用于 AR 应用,下面对 AR 应用性能分析与性能优化各流程进行详细阐述。

15.4.1 收集运行数据

收集 AR 应用运行时性能数据主要通过汇总 ARCore 性能分析数据、Unity 和 VS 性能调试数据、UPR 性能分析数据、测试人员测试体验反馈、使用用户对 AR 应用的反馈、AR 应用运行异常报告工具数据等。ARCore 性能分析面板是最直观获取性能数据的方式,操作简单、数据呈现直观;在需要深入进行性能问题定位和排查时,Unity Profiler、UPR、Frame Debug 工具是强大的分析利器,如图 15-17 所示,通过 Unity Profiler、Frame Debug 收集 AR 应用实时运行数据的步骤如下:

(1) 在目标设备上生成 Development Build (开发构建) 应用并运行,实时采集 AR 应用运行时数据。

(2) 对采集的运行数据进行观察,特别是那些性能消耗过高的节点,寻找导致帧速降低的关键点,录制运行时数据以便后续分析。

(3) 在录制性能问题的样本数据后,单击 Profiler 窗口上部的任意位置暂停应用,并且选择一帧。

(4) 在分析器上部窗格中选择展示有性能问题的应用帧(这可能是帧速突然降低帧,也可能是持续性的低速帧)。通过键盘的左右箭头按键或者窗格控制栏的前后按钮移动选择目标帧,直到选到需要进行分析的目标帧。

(5) 对选定的帧启动帧调试器进行更加深入的数据采集。

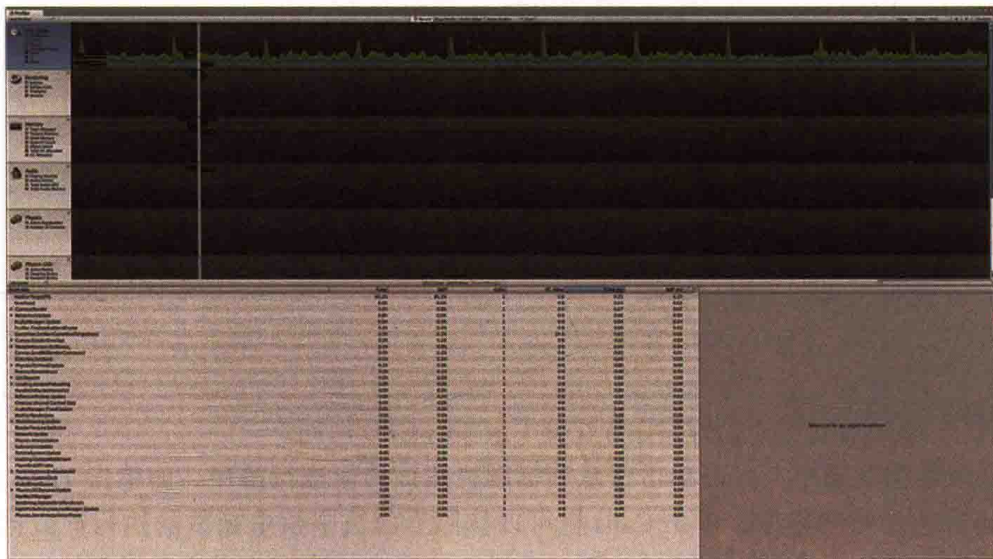


图 15-17 使用 Unity 分析器分析各模块各任务的执行情况

15.4.2 分析运行数据

采集 AR 应用运行时数据是基础,对这些数据进行分析并找到引起性能问题的原因才是修复问题的关键。这里以 CPU 分析器为例,讲解如何分析采集到的数据,其他分析器的分析方法与此类似。

在出现性能问题时,CPU 分析器通常是使用得最多的分析器。在 CPU 分析器窗口上部,可以很清晰地看到为完成一帧画面各任务组花费的 CPU 时间,如图 15-18 所示。



图 15-18 打开 CPU 使用分析器窗口分析各任务组执行数据

对各任务组,分析器以不同颜色进行了标识分类,可以选择一个或几个任务组进行查看。不同的颜色分别代表了在渲染、脚本执行、物理运算等方面花费的时间,如图 15-18 所示,分析器左侧显示了哪种颜色代表哪类任务。

在如图 15-19 所示的截图中,窗口底部显示了这帧中所有 CPU 运算耗时共计 85.95ms。



图 15-19 CPU 分析器窗口显示了总执行时间

对照颜色查看各任务组,发现大部分时间消耗在渲染上,由此可以知道,是渲染性能问题造成了掉帧,那么渲染优化就成了当前最主要的优化方向。

CPU 分析器还提供了不同的显示模式,可以是时间线 (Timeline)、层次结构 (Hierarchy)、原始层次结构 (Raw Hierarchy)。在发现是渲染问题导致掉帧后,可以选择使用层次结构模式去挖掘更深入的信息,通过分析器左下窗口的下拉菜单中选择层次结构后可以查看 CPU 任务的详细信息,查看在这帧中是哪些任务花费了最多的 CPU 时间。

在层级结构视图中,可以单击任意列标题栏按该列值进行排序。如单击 Time ms 栏可以按照函数花费时间排序,单击 Calls 栏可以按照当前选中帧中函数的执行次数排序。在图 15-20 所示截图中,按照时间排序,可以看到 Camera.Render 任务花费了最多的 CPU 时间。

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms	
▼ Camera.Render	94.6%	3.0%	1	0 B	81.38	2.64	
▼ Drawing	77.2%	0.0%	1	0 B	66.40	0.03	
▼ Render.OpaqueGeometry	75.4%	0.1%	1	0 B	64.87	0.15	
▼ RenderForwardOpaque.Render	64.9%	1.9%	1	0 B	55.79	1.63	
▼ Shadows.RenderShadowMap	49.9%	0.6%	1	0 B	42.95	0.56	
▼ Shadows.RenderJob	39.6%	0.0%	1	0 B	34.05	0.00	
▶ Shadows.RenderJobDir	39.6%	3.4%	1	0 B	34.05	11.58	
WaitingForJob	9.6%	9.6%	1	0 B	8.31	8.31	
RenderTexture.SetActive	0.0%	0.0%	1	0 B	0.00	0.00	
JobAlloc.Grow	0.0%	0.0%	1	0 B	0.00	0.00	
▶ RenderForward.RenderLoop	12.8%	7.5%	1	0 B	11.07	6.51	

图 15-20 通过层次视图查看性能消耗情况

在层级结构视图中，如果左侧行标题名字的左侧有箭头，则可以单击展开，进一步查看这个函数调用了哪些其他函数，并且这些函数是怎样影响性能的。在这个例子中，在 Camera.Render 任务中消耗 CPU 时间最多的是 Shadows.RenderJob 函数，即使现在对这个函数还不太了解，也已经对影响性能的问题有了大致的印象，知道了性能问题与渲染相关，并且最耗时的任务是处理阴影。

切换到时间线（Timeline）模式，如图 15-21 所示，时间线视图展示了两个重要的事项，即 CPU 任务执行顺序和各线程负责的任务。

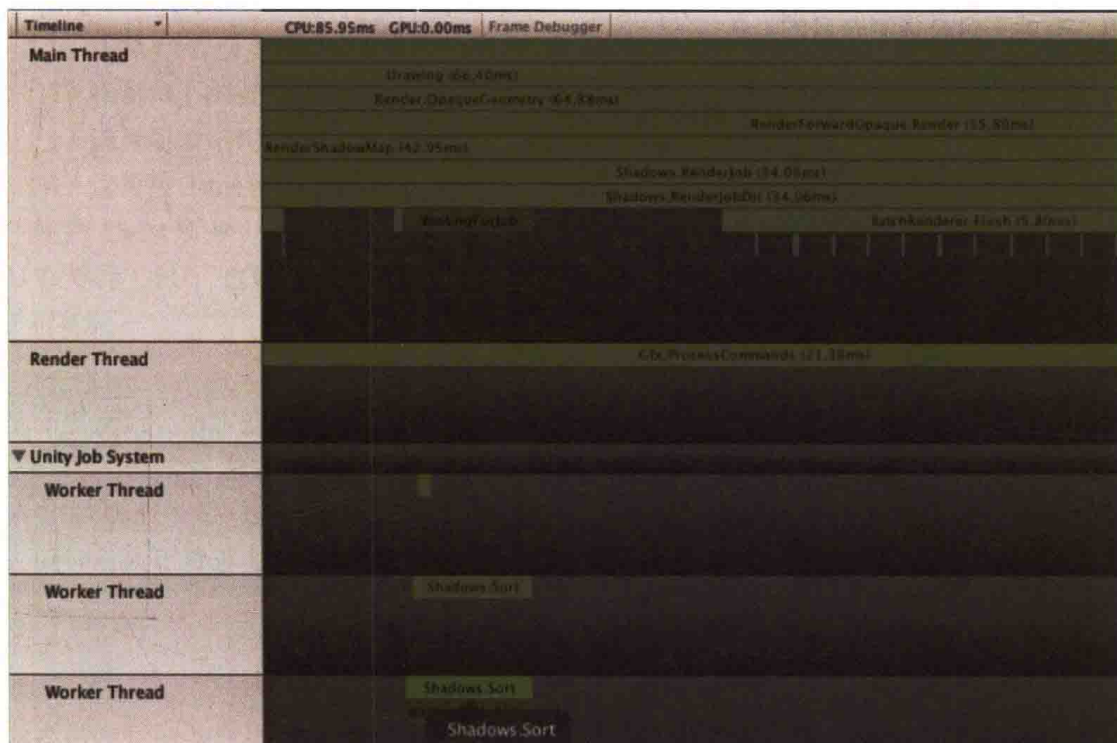


图 15-21 各线程执行情况

通过查看时间线任务执行图，可以找到执行最慢的线程，这也是下一步需要优化的线程。在图 15-21 中，看到 Shadows.RenderJob 调用的函数发生在主线程，主线程的一个任务

WaitingForJob 指示出主线程正在等待工作者线程完成任务, 因此, 可以推断出和阴影相关的渲染操作, 在主线程和工作线程同步上花费了大量时间。

提示

线程允许不同的任务同时执行, 当一个线程执行一个任务时, 另外的线程可以执行另一个完全不同的任务。和 Unity 渲染过程相关的线程有 3 种: 主线程、渲染线程和工作者线程 (Worker Threads), 多线程就意味着需要同步, 很多时候性能问题就出在同步上。

15.4.3 确定问题原因

找到问题是解决问题的第一步, 为查找引起性能问题的原因, 首先, 需要排除垂直同步的影响, 垂直同步 (VSync) 用于同步应用的帧率和屏幕的刷新率, 打开垂直同步会影响 AR 应用的帧率, 在分析器窗口中可以看到影响。垂直同步的影响可能看起来像性能问题, 会影响我们判断排错, 所以在继续查找问题之前应该先关闭垂直同步, 在 Unity 菜单中, 依次选择 Edit → Project Settings, 打开工程设置对话框, 切换到 Quality 选项卡, 在 Other 栏中将 V Sync Count 属性设置为 Don't Sync, 如图 15-22 所示, 但垂直同步不是在所有的平台都可以关闭, 有的平台是强制开启的。

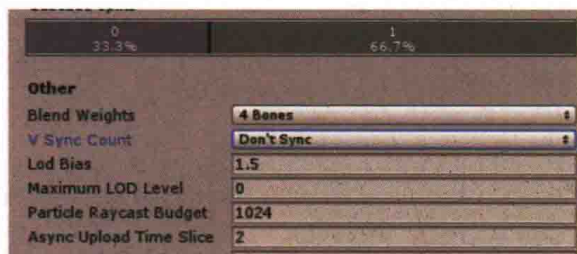


图 15-22 关闭垂直同步

渲染是常见的引起性能问题的原因, 当尝试修复一个渲染性能问题之前, 最重要的是确认 AR 应用是受限于 CPU 还是受限于 GPU, 不同的问题需要用不同的方法去解决。简单地说, CPU 负责决定什么东西需要被渲染, 而 GPU 负责渲染它。当渲染性能问题是因为 CPU 花费了太长时间去渲染一帧时, 是 CPU 受限, 当渲染性能问题是因为 GPU 花费了太长时间去渲染一帧时, 是 GPU 受限。

1. GPU 受限

识别是否 GPU 受限的最简单方法是使用 GPU 分析器, 但遗憾的是并非所有的设备和驱动都支持 GPU 分析器, 因此需要先检查 GPU 分析器在目标设备上是否可用, 单击打开 GPU 使用情况分析器, 如果目标设备不支持, 则可以看到右侧显示信息 “不支持 GPU 分析 (GPU profiling is not supported)” 字样, 如图 15-23 所示。

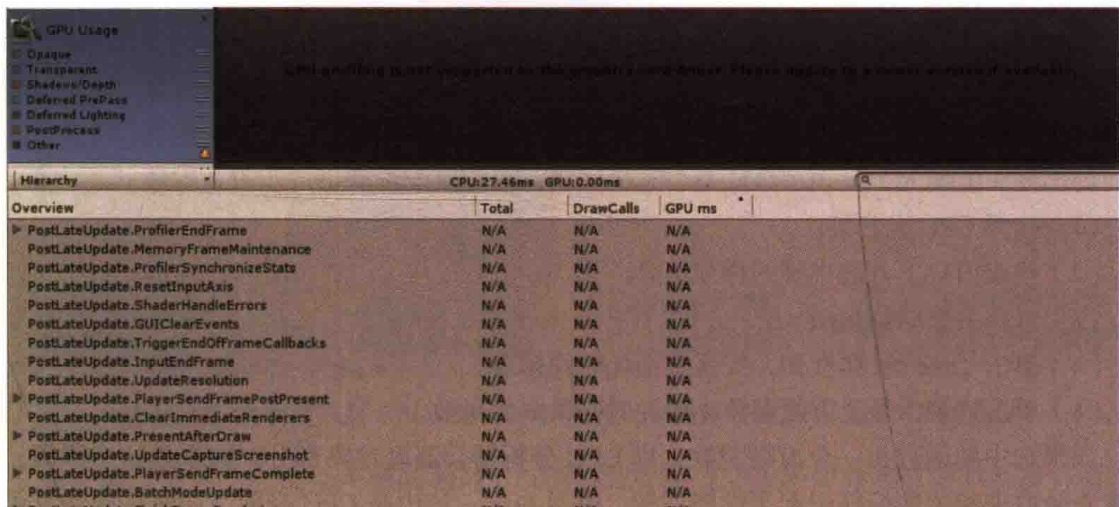


图 15-23 设备不支持 GPU 分析界面

如果设备支持 GPU 分析，则只需查看 GPU 分析器窗口区域下方中间部分的 CPU 时间和 GPU 时间，如果 GPU 时间大于 CPU 时间，就可以确定 AR 应用是 GPU 受限。如果 GPU 分析器不可用，则仍然有办法确认 AR 应用是否是 GPU 受限，打开 CPU 分析器，如果看到 CPU 在等待 GPU 完成任务，即意味着 GPU 受限，步骤如下：

- (1) 选择 CPU 分析器。
- (2) 在窗口下部查看选择帧的详细信息。
- (3) 选择层级结构视图。
- (4) 选择按 Time ms 属性列排序。

如图 15-24 所示，函数 Gfx.WaitForPresent 在 CPU 分析器中花费的时间最长，这表明 CPU 在等待 GPU，也就是 GPU 受限。如果是 GPU 受限，下一步就应该主要对 GPU 图形渲染进行优化。



图 15-24 当设备不支持 GPU 分析时通过 CPU 分析器判断是否是 GPU 受限

2. CPU 受限

如果 AR 应用渲染不是受限于 GPU, 就需要检查 CPU 相关的渲染问题。选择 CPU 分析器, 在分析器窗口的上部, 检查代表渲染任务组颜色的数据, 在有性能问题的帧中, 如果大部分时间都消耗在渲染上, 则表示是渲染引起了问题, 按照以下步骤进一步查找性能问题信息:

- (1) 选择 CPU 分析器。
- (2) 检查窗口下方选中帧的详细信息。
- (3) 选择层级结构视图。
- (4) 单击 Time ms 属性列, 按函数消耗时间排序。
- (5) 单击列表中最上方的函数 (消耗时间最多的函数)。

如果选中的函数是一个渲染函数, 则 CPU 分析器会高亮渲染部分。如果是这个原因, 则意味着是渲染相关的操作引起了性能问题, 并且在这一帧中是 CPU 受限, 同时需要注意函数名和函数是在哪个线程执行, 这些信息在尝试修复问题时十分有用。如果是 CPU 受限, 下一步就应该主要对 CPU 图形渲染进行优化。

3. 其他引起性能问题的主要原因

通常垃圾回收、物理计算、脚本运行也是引起性能问题的主要原因, 可以通过 CPU 分析器、内存分析器等多分析器联合进行分析。如果函数 GC.Collect() 出现在最上方, 并且花费了过多的 CPU 时间, 则可以确定垃圾回收是应用性能的问题所在; 如果在分析器上方高亮物理运算, 则说明 AR 应用的性能问题与物理引擎运算相关; 如果在分析器上方高亮显示的是用户脚本函数, 则说明 AR 应用的性能问题与用户脚本相关。

尽管讨论了图形渲染、垃圾回收、物理计算、用户脚本 4 种最常见的引起性能问题的原因, 但 AR 应用在运行时可能会遇到各种各样的性能问题, 但万变不离其宗, 遵循上述解决问题的方法, 先收集数据, 使用 CPU 分析器检查应用运行信息, 找到引起问题的原因, 一旦知道了引起问题的函数名字, 就可以在 Unity Manual、Unity Forums、Unity Answers 中查找函数的相关信息, 最终解决问题。

15.5 渲染优化

对 AR 应用而言, 绝大部分情况下, 发生卡顿、掉帧、假死等情况都源于场景渲染问题, 源于系统无法按时完成场景渲染任务。为达到流畅的用户体验, ARCore 要求应用帧率达到 30 帧/秒以上, 即每一帧所有工作完成时间不大于 33.3ms, 这对计算实时性提出了很高要求。

15.5.1 渲染流程

在 AR 应用中, 渲染性能受很多因素影响, 并且高度依赖 CPU 与 GPU 的协作, 优化渲染性能最重要的是通过调试实验, 精确分析性能检测结果, 有针对性地解决问题。深入理解

Unity 渲染事件流有助于研究和分析,在渲染过程中,CPU 与 GPU 各自都有其独立的任务周期,如图 15-25 所示。

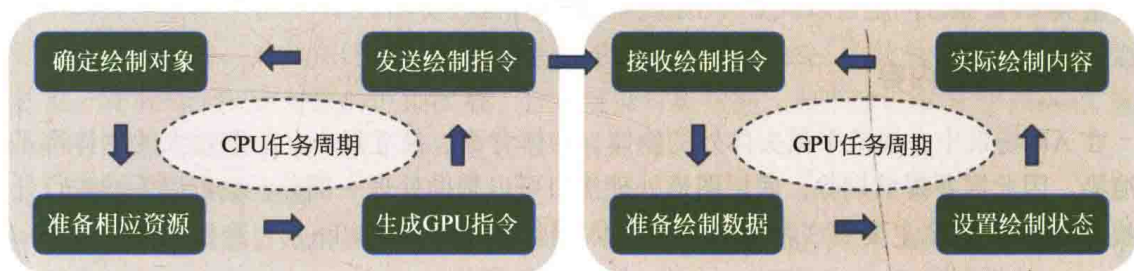


图 15-25 左侧循环是 CPU 渲染工作流程图、右侧循环是 GPU 渲染工作流程图

首先需要清楚,在渲染一帧画面时 CPU 和 GPU 需要分工协作并完成各自任务,它们中的任何一方花费时间超过预期都会造成渲染延迟,因此渲染性能出现问题有两类基本的原因:第一类问题是 CPU 计算能力弱,如果 CPU 处理数据时间过长(往往是因为复杂的计算或大量的数据准备),打断了平滑的数据流时 CPU 成为渲染瓶颈;第二类问题是 GPU 渲染管线引起的,当渲染管线中一步或者多步花费时间过长(往往是渲染管线需要处理的数据太多,如模型顶点多、纹理尺寸大等),打断了平滑的数据流时 GPU 成为渲染瓶颈。简而言之,如果 CPU 与 GPU 负载分布不平衡,就会导致其中一个过载而另一个空载,一方等待另一方。在给定硬件条件下,对渲染的优化就是寻求一个平衡点,使 CPU 和 GPU 都处于忙碌但又不过载的状态。如果 CPU 与 GPU 硬件性能确实不能满足要求,则只能通过降低效果、精简模型等手段,以性能换质量。

15.5.2 CPU 瓶颈

在渲染一帧画面时, CPU 主要完成三类工作:决定绘制内容、为 GPU 准备命令、将命令发送到 GPU。这三类工作包含很多独立的任务,这些任务可以通过多线程协同完成。

Unity 渲染过程与三类线程相关:主线程(Main Thread)、渲染线程(Render Thread)和工作者线程(Worker Threads)。主线程用于完成 AR 应用主要 CPU 任务,包括一些渲染任务;渲染线程专门用于将命令发送给 GPU;每个工作者线程执行一个单独的任务,如剔除或网格蒙皮。哪些任务在哪个线程执行取决于 AR 应用运行的硬件和应用设置,通常 CPU 核心数量越多,生成的工作者线程数也可以越多。

由于多线程渲染非常复杂并且依赖硬件,在尝试改进性能时,首先必须了解是哪些任务导致了 CPU 瓶颈,如果 AR 应用运行缓慢是因为在一个线程上的剔除操作花费了太长的时间,则尝试在另一个线程上降低网格蒙皮的时间不会有任何帮助。

Player Settings 设置面板中的 Graphics Jobs 选项(在 Player Settings → Player → Other Settings 选项卡下)决定了 Unity 是否使用工作者线程去执行一些原本需要在主线程或者渲染线程中执行的任务,在支持这个功能的平台上启用该选项能够提供可观的性能提升。如果希

望使用这个功能, 则应该分别对开启和关闭此功能进行性能分析, 观察该功能对性能的影响, 看一看是否有助于性能的提升。

解决 CPU 瓶颈问题可以从 CPU 的三类工作入手逐个分析。

1. 决定绘制内容

在 AR 场景中, 视锥平截头体外的物体、物体背面、尺寸过小、距离过大的物体都不需要渲染, 因此需要提前剔除, 底层图形处理接口可以帮助处理一部分, 也可以手动进行处理以提高性能。剔除主要有 3 种形式: 视锥体剔除 (Frustum Culling)、遮挡剔除 (Occlusion Culling)、远近剔除 (LayerCullDistances), 如表 15-7 所示。

表 15-7 剔除方式及处理方法

剔除方式	常用技巧
Frustum Culling	使用 Camera 的 Frustum Matrix 属性剔除不需要显示的物体, 简单地说就是渲染相机看不到的物体不需要显示, Unity 默认开启
Occusion Culling	将被遮挡的物体剔除, Unity 里需设置遮挡标识 (Occlude Flag)
LayerCullingDistance	将超过一定距离的物体剔除, 使用该方法时需要提前设置需要剔除物体所属的层 (Layer), 如设置为 grass, 则剔除方法如下: <pre>float[] distances = new float[32]; distances[LayerMask.NameToLayer("grass")] = 50; camera.layerCullSpherical = true; camera.layerCullDistances = distances;</pre> 当物体与相机距离超过 50m 后, 物体将被剔除不再渲染, 该方法一般用于处理数量多且体积小的虚拟物体

2. 为 GPU 准备命令

为 GPU 准备命令包括将渲染所需数据加载到显存、设置渲染状态。在准备渲染时, CPU 需要将所有渲染所需数据从硬盘加载到内存中, 模型顶点、索引、纹理等数据又被从内存加载到显存中, 这些数据量在复杂场景中会非常大。在将渲染数据加载到显存后, CPU 还需要设置 GPU 渲染状态, 如网格的顶点 / 片元着色器、光照模型、抗锯齿模式等, 不仅如此, 还需要设置 GPU 的管线状态, 如 RasterizerState、BlendState、DepthStencilState、InputLayout 和其他数据, 这个过程需要 CPU 与 GPU 同步, 也是一个耗时的过程。在这个阶段优化的重点就是减少 Drawcall, 能一次性渲染的内容不要分阶段、分区域多次渲染, 能合并的模型、纹理、材质尽量合并。

3. 将命令发送到 GPU

将命令发送到 GPU 消耗时间过长是引起 CPU 性能问题的最常见原因。在大多数平台上, 将命令发送到 GPU 任务由渲染线程执行, 个别平台由工作者线程执行 (如 PS4), 在将命令发送到 GPU 时, 最耗时的操作是 SetPass call。如果 CPU 瓶颈是由发送 GPU 命令引起, 则降低

SetPass call 的数量通常是最好的改善性能的办法。在 Unity 渲染分析器中，可以直观地看到有多少 SetPass call 和 Batches 被发送。

不同硬件处理 SetPass call 的能力差异很大，在高端 PC 上可以发送的 SetPass call 数量要远远大于移动平台。通常在优化时为了降低 SetPass call 和 Batches 的数量，需要减少渲染对象数量、降低每个渲染对象的渲染次数、合并渲染对象数据，如果读者学习过 DirectX 或者 OpenGL，对图形渲染管线有所了解则应该很容易理解。

1) 减少渲染对象数量

减少渲染对象数量是最简单的降低 Batches 和 SetPass call 的方法，减少渲染对象数量的常见方法如表 15-8 所示。

表 15-8 减少渲染对象数量的常见方法

序号	常用技巧
1	降低场景中可见对象数量。如渲染很多僵尸时，可以尝试减少僵尸数量，如果看起效果不错，就是一个简单且高效的优化方法
2	降低渲染相机裁剪平面的远平面（Far Plane）来减小绘制范围。这个属性表示距离相机多远后的物体将不再被渲染，并通常使用雾效来掩盖远处物体从不渲染到渲染的突变
3	如果需要基于距离的更细粒度控制物体渲染，则可以使用相机的 Layer Cull Distances 属性，它可以给不同的层设置单独的裁剪距离，当场景中有很多前景装饰细节时，这种方法很有用
4	可以使用遮挡剔除功能关闭被遮挡物体的渲染。如场景中有一个很大的建筑，可以使用遮挡剔除功能，关闭该建筑后面被遮挡物体的渲染。需要注意的是 Unity 的遮挡剔除功能并不是所有场景都适用，它会导致额外的 CPU 消耗，并且相关设置比较复杂，但在某些场景中可以极大地改善性能
5	手动关闭物体渲染。可以手动关闭用户看不见的物体的渲染，如场景包含一些过场的物体，在它们出现之前或者移出之后，应该手动关闭它们的渲染。手动剔除往往比 Unity 动态地遮挡剔除有效得多

2) 降低每个渲染对象的渲染次数

阴影和环境反射可以大幅提高 AR 应用的真实感，但是这些操作也非常昂贵，使用这些功能可能导致物体被渲染多次，从而对性能造成影响，通常来讲，这些功能对性能的影响程度依赖于场景选择的渲染路径，降低每个渲染对象的渲染次数的常见方法如表 15-9 所示。

表 15-9 降低每个渲染对象的渲染次数的常见方法

序号	常用技巧
1	深入理解 Unity 中的动态光照，尽量少使用动态光照，包括点光和聚光
2	使用烘焙技术去预计算场景的光照
3	尽量少使用反射探头
4	Shader 中尽量不使用多 pass 渲染，并使用 URP 渲染管线

提示

渲染路径即是在绘制场景时渲染计算的执行顺序。不同渲染路径最大的不同是处理实时光照、阴影和环境反射的方法。通常来讲，如果 AR 应用运行在比较高端的设备上，并且应用了较多实时光照、阴影和反射时延迟渲染是比较好的选择。前向渲染适用于目前的移动设备，它只支持较少的逐像素光照。

3) 合并物体

合并小的渲染物体可以大大地减少 Drawcall，一个 Batch 可能包含多个物体的数据，如果要合并物体，则这些物体必须共享相同材质的相同实例、相同的渲染设置（纹理、Shader、Shader 参数等）。合并渲染物体的常见方法如表 15-10 所示。

表 15-10 合并渲染物体的常见方法

序号	常用技巧
1	静态批处理（Batching），允许 Unity 合并相邻的不移动的物体。如一堆相似的静态石头就可以合并
2	动态批处理，不论物体是运动的还是静止的都可以进行动态合并，但对能够使用这种技术合并的物体有一些要求及限制
3	合并 Unity 的 UI 元素
4	GPU Instancing，允许大量一样的物体十分高效地合并处理
5	纹理图集，把大量小纹理合并为一张大的纹理图，通常在二维渲染和 UI 系统中使用，但也可以在三维渲染中使用
6	手动合并共享相同材质和纹理的网格，不论是在 Unity 编辑器中还是在运行时使用代码手动合并
7	在脚本中，谨慎使用 <code>Renderer.material</code> ，因为这会复制材质并且返回一个新副本的引用，导致渲染器（Renderer）不再持有相同的材质引用，从而破坏批处理。如果需要访问一个在合并中的物体的材质，则应该使用 <code>Renderer.sharedMaterial</code>

15.5.3 GPU 瓶颈

GPU 性能瓶颈中最常见的问题是填充率、显存带宽、顶点处理。

1. 填充率

填充率是指 GPU 每秒可以在屏幕上渲染的像素数量。如果 AR 应用受填充率限制，即应用每帧尝试绘制的像素数量超过了 GPU 的处理能力就会出现卡顿，解决填充率问题最有效的方式是降低显示分辨率，其他填充率问题的常见优化方法如表 15-11 所示。

表 15-11 填充率优化常见方法

序号	常见引起填充率问题的原因及优化技巧
1	片元着色器处理像素绘制，GPU 需要对每个需要绘制的像素进行计算，如果片元着色器代码效率低，就很容易发生性能问题，复杂的片元着色器是很常见的引起填充率问题的原因
2	重绘制（Overdraw）是指相同的像素被绘制的次数，重绘制过高也容易引起填充率问题。最常见的引起重绘制的原因是透明材质、未优化的粒子、重叠的 UI 元素，优化它们可以改善重绘制问题
3	屏幕后处理技术也会极大地影响填充率，尤其是在联合使用多种屏幕后处理技术时。如果在使用屏幕后处理时遇到了填充率问题，则应该尝试不同的解决方案或者使用更加优化的屏幕后处理版本。如使用 Bloom（Optimized）替换 Bloom，如果在优化屏幕后处理效果后，仍然有填充率问题，则应当考虑关闭屏幕后处理，尤其是在移动设备上

2. 显存带宽

显存带宽是指 GPU 读写显存的速度。如果 AR 应用受限于显存带宽，通常意味着使用的纹理过多或者纹理尺寸过大，以至于 GPU 无法快速处理，这时需要降低纹理的内存占用，降低纹理使用的常见方法如表 15-12 所示。

表 15-12 降低纹理使用的常见方法

技 术	常用技巧
纹理压缩技术	纹理压缩技术可以同时降低纹理在磁盘和内存中的大小。如果是显存带宽的问题，则使用纹理压缩减小纹理在内存中的大小可以帮助改善性能。Unity 内置了很多可用的纹理压缩格式和设置，通常来讲，纹理压缩格式只要可用就应该尽可能地使用，尽管如此，通过实践找到针对每个纹理最合适的设置是最好的
MipMap	多级渐远纹理是指 Unity 对远近不同的物体使用不同的纹理分辨率版本的技术，如果场景中包含距离相机很远的物体或者物体与相机距离变化较大，则可以通过多级渐远纹理来缓解显存带宽问题，但 MipMap 也会增加 33% 显存占用

3. 顶点处理

顶点处理是指 GPU 处理渲染网格中的每个顶点的工作。顶点处理主要由需要处理的顶点数量及在每个顶点上的操作两部分组成。减少渲染的顶点，简化在顶点上的操作可以降低顶点处理压力，降低顶点处理的常见方法如表 15-13 所示。

表 15-13 降低顶点处理的常见方法

技 术	常见优化技巧
减少顶点数	最直接的降低顶点数量的方法就是在三维建模软件中创建模型时使用更少数量的顶点，另外，场景中渲染的物体越少越有利于减少需要渲染的顶点数量
法线贴图	法线贴图技术可以用来模拟更高几何复杂度的网格而不用增加模型的顶点数，尽管使用这种技术有一些 GPU 消耗，但在多数情况下可以获得性能提升
关闭切线操作	如果模型没有使用法线贴图技术，则在模型的导入设置中可以关闭顶点的切线，这将降低在顶点上的操作复杂度

续表

技 术	常见优化技巧
LoD	LoD (Level of Detail) 是一种当物体远离相机时降低物体网格复杂度的技术。LoD 可以有效地降低 GPU 需要渲染的顶点数量, 并且不影响视觉表现
优化顶点 Shader	顶点着色器用于处理绘制的每个顶点, 优化顶点着色器可以降低在顶点上的操作, 有助于性能提升
优化 Shader	特别是自定义的 Shader, 应该尽可能地优化

15.6 代码优化

代码优化问题就像 C 语言一样古老, 从编程语言诞生之日起, 对代码进行优化就一直伴随着编程语言的发展。执行效率更高一直是语言发展和程序开发人员追求的目标, 有关代码优化的方法、实践、指导原则也非常多, 本节重点学习与 AR 开发密切相关的垃圾回收、对象池及部分极大影响性能却容易被忽视的代码细节。

15.6.1 内存管理

在 AR 应用运行时, 应用会从内存中读取数据, 也会往内存中写入数据, 在运行一段时间后, 某些写入内存中的数据可能会过期并不再被使用 (如从硬盘中加载到内存中的一个狐狸模型在狐狸被子弹击中并不再显示后), 这些不再被使用的数据就像垃圾一样, 而存储这些数据的内存应该被释放以便重新利用, 如果垃圾数据不被清理将会一直占用内存从而导致内存泄漏, 进而造成应用卡顿或崩溃。

通常把存储了垃圾数据的内存叫作垃圾, 把重新使这些存储垃圾数据的内存变得可用的过程叫作垃圾回收 (Garbage Collection, GC), 垃圾回收是 Unity 内存管理的一部分。在应用运行过程中, 如果垃圾回收执行得太频繁或者垃圾太多, 就会导致应用卡顿甚至假死。

Unity 自动进行内存管理, 其会替开发人员做很多 (如堆栈管理、分配、垃圾回收) 工作, 在应用开发中开发人员不用操心这些细节。在本质上, Unity 中的自动内存管理大致如下:

(1) Unity 在栈 (Stack) 内存和堆 (Heap) 内存两种内存池中存取数据, 栈用于存储短期和小块的数据, 堆用于存储长期和大块的数据。

(2) 当一个变量创建时, Unity 在栈或堆上申请一块内存。

(3) 只要变量在作用域内 (仍然能够被脚本代码访问), 分配给它的内存就处于使用中状态, 称这块内存已经被分配。一个变量被分配到栈内存时称为栈上对象, 被分配到堆内存时称为堆上对象。

(4) 当变量离开作用域后, 其所占的内存即成为垃圾, 就可以被释放回其申请的内存池。当内存返回内存池时, 称为内存释放。当栈上对象不在作用域内时, 栈上的内存会立刻被释放, 当堆上对象不在作用域时, 在堆上的内存并不会马上被释放, 并且此时内存状态仍然是已分配状态。

(5) 垃圾回收器周期性地清理堆分配的内存，识别并释放无用的堆内存。

Unity 在栈上分配释放内存和在堆上分配释放内存存在很大的区别：栈上分配和释放内存很快并且很简单，分配和释放内存总是按照预期的顺序和预期的大小执行。栈上的数据均是简单数据元素集合（在这里是一些内存块），元素按照严格的顺序添加或者移除，当一个变量存储在栈上时，内存简单地在栈的“末尾”被分配，当栈上的变量不在作用域时，存储它的内存马上被返回栈中以便重用。

在堆上分配内存要复杂得多，堆会存储各种大小和类型的数据，堆上内存的分配和释放并不总是有预期的顺序，并且需要不同大小的内存块。当一个堆对象被创建时 Unity 会执行以下步骤：

(1) 首先检查堆上是否有足够的空闲内存，如果堆上空闲内存足够，则为变量分配内存。

(2) 如果堆上内存不足，则 Unity 会触发垃圾回收器并尝试释放堆上无用的内存。如果垃圾回收后空闲内存足够，则为变量分配内存。

(3) 如果执行垃圾回收操作后，堆上空闲内存仍然不足，Unity 则会向操作系统申请增加堆内存容量。如果申请到足够内存空间，则为变量分配内存。

(4) 如果申请失败，应用则会出现内存不足错误。

因此，堆上分配内存可能会很慢，尤其是在需要执行垃圾回收和扩展堆内存时。

提示

Unity 中所有值类型存储在栈上，引用类型则存储在堆上。在 C# 中，值类型包括基本数据类型、结构体、枚举；引用类型包括 Class、Interface、Delegate、Dynamic、Object、String 等。

15.6.2 垃圾回收

Unity 在执行垃圾回收时，垃圾回收器会检查堆上的每个对象，查找所有当前对象的引用，确认堆上对象是否还在作用域，将不在作用域的对象标记为待删并随后删除这些对象，然后将这些对象所占内存返还到堆中。

下列情况发生时将触发垃圾回收：

(1) 当需要在堆上分配内存且空闲内存不足时。

(2) 周期性地回收垃圾，自动触发（频率由平台决定）。

(3) 手动强制执行垃圾回收，如手动调用 `System.GC.Collect()` 方法。

垃圾回收是一项工作密集型的操作，堆上的对象越多它需要做的工作就越多、对象的引用越多它需要做的工作就越多，并且垃圾回收还可能在不恰当的时间执行和导致内存空间碎片化，严重时会导致应用卡顿或假死。减少垃圾生成和降低垃圾回收频率的几个技术如下。

1. 字符串使用

在 C# 语言中, 字符串是引用类型, 创建和丢弃字符串会产生垃圾, 而且由于字符串被广泛使用, 这些垃圾可能会积少成多、快速累积, 特别是在 Update() 方法中使用字符串时, 不恰当的方法可以迅速耗尽内存资源。

C# 语言中的字符串值不可变, 这意味着字符串值在被创建后不能被改变, 每次操作字符串 (例如连接两个字符串) 时会创建一个新的字符串保存结果, 然后丢弃旧的字符串, 从而产生垃圾, 因此, 在使用字符串时, 应当遵循一些简单的规则以使字符串在使用时所产生的垃圾最少化, 如表 15-14 所示。

表 15-14 优化字符串使用的常见方法

序 号	常用技巧
1	减少不必要的字符串创建。如果使用同样的字符串多于一次, 则应该只创建一次, 然后缓存它
2	减少不必要的字符串操作。如果需要频繁地更新一个文本组件的值, 并且其中包含一个连接字符串操作, 则应该考虑把它分成两个独立的文本组件
3	使用 StringBuilder 类。在需要频繁操作字符串时, 应该使用 StringBuilder 类, 该类设计用于做动态字符串处理, 并且不产生堆内存频繁分配问题, 在连接复杂字符串时, 这将减少很多垃圾的产生
4	移除 Debug.Log()。在 AR 应用的 Release 版本中 Debug.Log() 虽然不会输出任何东西, 但它仍然会被执行。调用一次 Debug.Log() 至少创建和释放一次字符串, 所以如果应用包含了很多次 Debug.Log() 调用, 则会产生大量垃圾

例 15-1 在 Update() 方法中合并字符串, 设计意图是在 Text 组件上显示当前时间, 但会产生很多垃圾, 代码如下:

```
// 第 15 章 / 15-3
public Text titleText;
private float timer;
void Update()
{
    timer += Time.deltaTime;
    titleText.text = "当前时间: " + timer.ToString();
}
```

优化方案是将显示当前时间的字符串分成两部分, 一个部分显示“当前时间:”字样, 另一部分显示时间, 这样不用字符串合并也能达到想要的效果, 代码如下:

```
// 第 15 章 / 15-4
public Text HeaderText;
public Text ValueText;
private float timer;
void Start()
{
```

```

    HeaderText.text = " 当前时间: ";
}
void Update()
{
    ValueText.text = timer.toString();
}

```

2. 共用集合

创建新集合对象时在堆上分配内存是一个比较复杂的操作，在代码中可以共用一些集合对象，缓存集合引用，使用 `Clear()` 方法清空集合内容复用以替代耗时的集合创建操作并减少垃圾。

例 15-2 频繁地创建集合，每次使用 `New` 方法都会在堆上分配内存，代码如下：

```

// 第 15 章 /15-5
void Update()
{
    List mList = new List();
    Pop(mList);
}

```

优化方案是缓存公用集合，只在集合创建或者在底层集合必须调整大小的时候才分配堆内存，代码如下：

```

// 第 15 章 /15-6
private List mList = new List();
void Update()
{
    mList.Clear();
    Pop(mList);
}

```

3. 降低堆内存分配频度

在 Unity 中，最糟糕的设计是在那些频繁调用的函数中分配堆内存，如在 `Update()` 和 `LateUpdate()` 方法中，这些方法每帧调用一次，所以如果在这里产生了垃圾，垃圾将会快速累积。良好的设计应该考虑在 `Start()` 或 `Awake()` 方法中缓存引用，或者确保引起堆分配的代码只在需要的时候才运行。

例 15-3 频繁调用会生成大量垃圾，代码如下：

```

// 第 15 章 /15-7
void Update()
{
    ShowPostion(" 当前位置: "+transform.position.x);
}

```

优化方案是只在 `transform.position.x` 值发生改变时才调用生成垃圾的代码, 并对字符串的使用进行处理, 代码如下:

```
// 第 15 章 /15-8
private float previousPositionX;
private float PositionX;
void Update()
{
    PositionX = transform.position.x;
    if (PositionX != previousPositionX)
    {
        ShowPostion(PositionX);
        previousPositionX = PositionX;
    }
}
```

4. 缓存

堆内存对象重复创建会造成堆内存重复分配, 产生不必要的垃圾, 改进方案是保存结果的引用并复用它们, 通常这种方法称作缓存。

例 15-4 方法被重复调用, 因为方法中会创建数组, 方法每次被调用时都会造成堆内存分配, 代码如下:

```
// 第 15 章 /15-9
void OnTriggerEnter(Collider other)
{
    Renderer[] allRenderers = FindObjectsOfType<Renderer>();
    ExampleFunction(allRenderers);
}
```

优化方案是缓存结果, 缓存数组并复用可以保证不生成更多垃圾, 代码如下:

```
// 第 15 章 /15-10
private Renderer[] allRenderers;
void Start()
{
    allRenderers = FindObjectsOfType<Renderer>();
}

void OnTriggerEnter(Collider other)
{
    ExampleFunction(allRenderers);
}
```

5. 装拆箱

C# 中所有的数据类型都是从基类 `System.Object` 继承而来, 所以值类型和引用类型的值可

以通过显式（或隐式）操作相互转换，该过程也就是装箱（Boxing）和拆箱（UnBoxing）。装箱操作，通常发生在将值类型的变量（如 int 或者 float）传递给需要 object 参数的函数时，如 Object.Equals()，而拆箱通常发生在将引用类型赋给值类型变量。

例 15-5 函数 String.Format() 接收一个字符串和一个 object 参数。当传递参数为一个字符串和一个 int 时，int 会被装箱，代码如下：

```
// 第 15 章 /15-11
void ShowPrice()
{
    int cost = 5;
    string displayString = String.Format(" 商品价格: {0} 元 ", cost);
}
```

另一个非常典型的装箱是返回 null 值类型，如开发人员希望函数调用时为值类型返回 null，尤其是在该函数操作可能失败的情况时。当一个值类型变量被装箱时，Unity 会在堆上创建一个临时的 System.Object 用于包装值类型变量，当这个临时对象被创建和销毁时会产生垃圾。

装箱拆箱会产生垃圾，而且十分常见，即使在代码中没有直接的装箱操作，使用的插件或者其他间接调用的函数也可能在幕后进行了装箱操作。装箱拆箱在频繁、大量使用时可能会导致极大的性能问题，如使用粒子系统在粒子操作时使用装箱拆箱操作，由于粒子数量大、更新频繁，短时间就会消耗掉大量内存空间。避免装箱操作最好的方式是尽可能地少使用导致装箱操作的函数及避免使用直接的装箱操作。

6. 材质实例化

在每次触发 Renderer.material(s) 赋值时，Unity 会自动实例化新的材质，由开发人员负责销毁原材质，因此，在很多时候这都会引发内存泄漏问题，代码如下：

```
// 第 15 章 /15-12
private void Update()
{
    var cube = GameObject.CreatePrimitive(PrimitiveType.Cube);
    cube.GetComponent<Renderer>().material.color = Color.red;
    ...
    Destroy(cube);
}
```

一个改进的方法称为材质实例化（Material Instance）技术，它会跟踪实例化材质的生命周期，并在不需要时自动销毁实例化的材质，代码如下：

```
// 第 15 章 /15-13
private void Update()
{
    var cube = GameObject.CreatePrimitive(PrimitiveType.Cube);
```

```
cube.EnsureComponent<MaterialInstance>().Material.color = Color.red;  
...  
Destroy(cube);  
}
```

15.6.3 对象池

对象池 (Object Pool) 顾名思义就是一个包含一定数量已经创建好对象 (Object) 的集合。对象池技术在对性能要求较高的应用开发中使用得非常广泛,尤其在内存管理方面,可以通过重复使用对象池中的对象来提高性能和内存使用。如在游戏开发中,通常构建一个子弹对象池,通过重用而不是临时分配和释放子弹对象的方式提高内存利用效率,在发射子弹时从子弹池中取一个未用的子弹,在子弹与其他物体碰撞或者达到一定距离消失后将该子弹回收到子弹池中,通过这种方式可更快速地创建子弹,更重要的是确保使用这些子弹不会导致内存垃圾。

1. 使用对象池的好处

复用池中对象没有分配内存和创建堆中对象的开销,没有释放内存和销毁堆中对象的开销,从而可以减少垃圾回收器的负担,避免内存抖动,也不必重复初始化对象状态,对于比较耗时的构造函数 (Constructor) 和析构函数 (Finalize) 来讲非常合适,使用对象池可以避免实例化和销毁对象带来的常见性能和内存垃圾问题。

在 Unity 中,实例化预制体 (Prefab) 时,需要将预制体内容加载到内存中,然后将所需的纹理和网格上载到 GPU 显存,从硬盘或者网络中将预制体加载到内存是一个非常耗时的操作,如果将预制体预先加载到对象池中,则可以避免频繁地进行加载和卸载。

对象池技术在以下情况使用能有效地提高性能并减少垃圾产生:

- (1) 需要频繁地创建和销毁对象。
- (2) 性能响应要求高的场合。
- (3) 数量受限的资源,例如数据库连接。
- (4) 创建成本高昂的对象,比较常见的线程池、字节数组池等。

2. 使用对象池的不足

合理地使用对象池能有效地提升性能和内存使用,但并不意味着任何时机任何场合都适合使用对象池,创建对象池会占用内存,减少可用于其他目的的堆内存量。对象池大小设置不合理也会带来问题:如果分配的对象池过小,或如果需要继续在池上分配内存,则可能会更频繁地触发垃圾回收,不仅如此,还会导致每次回收操作都变得更加缓慢(因为回收所用的时间会随着活动对象的数量增加而增加);如果分配的池太大或者在一段时间内不需要对象池所包含的对象时仍保持它们的状态,则应用性能也将受到影响。此外,许多类型的对象不适合放在对象池中,如应用中包含的持续时间很长的魔法效果、需要渲染大量敌人而这些敌

人随着游戏的进行只能逐渐被杀死，在此类情况下对象池的性能开销会超过收益。

对象池使用不当也会出现如下问题：

(1) 并发环境中，多个线程可能需要同时存取池中对象，因此需要在堆数据结构上进行同步或者因为锁竞争而产生阻塞，这种开销要比创建销毁对象的开销高数百倍。

(2) 由于池中对象的有限数量，势必造成可伸缩性瓶颈。

(3) 很难正确合理地设定对象池大小。

(4) 设计和使用对象池容易出错，设计时出现状态不同步、使用时出现忘记归还或者重复归还、归还后仍使用对象等问题。

15.6.4 常见影响性能的代码优化策略

在开发 AR 应用时，AR Foundation 和 ARCore XR Plugin 工具包提供了很多方便快捷的方法、接口、工具，使用它们可以快速构建应用架构、实现应用功能，但在代码开发中，性能与便捷性很多时候是一对矛盾体，高的使用便捷性往往带来性能的降低，特别是在不了解底层实现时不恰当的调用会带来严重的性能问题。

1. 引用缓存

`GetComponent<T>()` 之类的方法非常方便获取对象的引用，`Camera.main` 属性也常用于获取渲染相机，但实际上它们在底层都使用了搜索（遍历），如 `GetComponent<T>()` 使用了类型搜索，而 `Camera.main` 使用了 `FindGameObjectsWithTag()` 搜索，而且使用 `GetComponent(string)` 字符串方法会比使用 `GetComponent<T>()` 泛型方法性能更差、开销更大。遍历是 $O(n)$ 级的复杂度，因此，为降低开销，通常应当在 `Awake()` 或者 `Start()` 方法中缓存引用，代码如下：

```
// 第 15 章 / 15-14
private Camera cam;
private CustomComponent comp;
void Start()
{
    cam = Camera.main;
    comp = GetComponent<CustomComponent>();
}
void Update()
{
    // 使用缓存引用
    this.transform.position = cam.transform.position + cam.transform.forward * 10.0f;
    // 使用搜索
    this.transform.position = Camera.main.transform.position + Camera.main.
transform.forward * 10.0f;
    // 使用缓存引用
    comp.DoSomethingAwesome();
}
```



```
// 使用搜索
GetComponent<CustomComponent>().DoSomethingAwesome();
}
```

在 Unity 中，此类方法还非常多，虽然使用很方便，但其性能开销极大，大部分此类方法 API 涉及在整个场景中搜索匹配对象，为避免频繁使用，通常应该缓存引用，常用的此类方法如表 15-15 所示。

表 15-15 常见性能开销大的方法

序 号	方 法 名
1	GameObject.SendMessage()
2	GameObject.BroadcastMessage()
3	UnityEngine.Object.Find()
4	UnityEngine.Object.FindWithTag()
5	UnityEngine.Object.FindObjectOfType()
6	UnityEngine.Object.FindObjectsOfType()
7	UnityEngine.Object.FindGameObjectsWithTag()
8	UnityEngine.Object.FindGameObjectsWithTag()

2. 避免使用 LINQ

尽管 LINQ 非常简洁且易于读写，但与手写算法相比，它所需的计算资源要多得多，尤其是在内存分配上表现更明显，所以如非必要不使用 LINQ，LINQ 的代码如下：

```
// 第 15 章 / 15-15
List<int> data = new List<int>();
data.Any(x => x > 10);
var result = from x in data
where x > 10 select x;
```

3. 空回调方法

在开发 AR 应用时，应该精心慎重地编写每秒执行很多次的任何函数、方法，如 Update()，如果在这些函数或者方法中发生了高开销的操作，则影响会非常巨大。在 Unity 中新建脚本文件时，默认会生成 Start() 和 Update() 方法，空的 Update() 方法看似没有妨碍，但实际上这些 Update() 方法每帧都会被调用，Unity 会在 UnityEngine 代码与应用程序代码块之间进行托管 / 非托管切换，上下文的切换会产生相当高的开销，即使没有需要执行的操作。如果应用中有数百个对象（包含空 Update() 方法），就很容易造成性能问题。与 Update() 方法类似，其他执行频率高的回调方法也有一样的问题，如 FixedUpdate()、LateUpdate()、OnPostRender()、OnPreRender()、OnRenderImage() 等。

4. 杂项

结构体 (Struct) 为值类型, 将其直接传递给函数时, 其内容将复制到新建的实例。这种复制增加了 CPU 开销及栈上内存。对于简单结构体, 这种影响通常可以忽略, 因此是可接受的, 但是, 对于每帧重复调用、采用复杂结构体的函数, 应当将函数定义修改为按引用传递。

限制物理模拟仿真迭代次数、避免使用网格碰撞体 (Mesh Collider)、禁用空闲动画、降低算法复杂度等方法都有利于提高应用性能。

15.7 ARCore 优化设置

AR 应用的一般优化原则与移动设备优化一致, 使用 Unity 开发 AR 应用时, 应当充分利用 Unity 提供的优化技术, 降低性能消耗, 提高用户体验。

15.7.1 UI/UX 优化

利用 Unity 进行 AR 应用开发时, 在 UI/UX 中使用图集 (Atlas) 能有效降低 Drawcall。在界面中默认一张图片一个 Drawcall, 但同一张图片多次渲染时仍然为一个 Drawcall, 因此将多张小图合在一起形成图集可以减少 Drawcall 的数量。另外, 由于影响 Drawcall 数量的是 Batch (批处理数), 而 Batch 以图集为单位进行处理, 所以在处理图集时, 通常做法是将常用图片放在一个公共图集, 将独立界面图片放在另一个图集, 一个 AR 应用 UI/UX 图集数建议为 3 ~ 4 个。

UI/UX 层级的深度也对 Drawcall 有很大影响, 在使用中, 应当尽量减少 UI/UX 层级的深度, 在层级 (Hierarchy) 窗口中 UGUI 节点的深度表现的就是 UI/UX 层级的深度, 深度越深, 不处在同一层级的 UI/UX 就越多, Drawcall 数就会越大。

15.7.2 常用优化设置

在使用 Unity 开发 AR 应用时, 其通常会提供一些默认的设置, 但这些默认设置是针对普通应用的普适性设置, 并不能提供对 AR 应用的特殊优化, 加之 ARCore 使用的移动设备属于边端设备, 在性能上比 PC、PlayStation 等设备有更多的约束, 为了提高性能, 通常需要进行一些针对性的设置。在 Unity 菜单中, 依次选择 Edit → Project Settings 打开 Project Settings 窗口, 选择 Player 项, 在 Android 平台下的 Other Settings 栏中进行以下设置。

1. 开启 Graphics Jobs

在单线程设备中, 图形渲染工作也由主线程承担, 而在多线程设备中, 则允许使用其他线程分担渲染任务, 从而可以并行计算, 提高效率。Graphics Jobs (图形任务) 属性决定 Unity 进行图形渲染可利用的线程, 当启用该功能后, Unity 可以利用渲染线程、工作者线程

进行渲染任务,从而带来可观的性能改进。

2. 开启 Multithreaded Rendering

ARCore v1.7 及后续版本都支持多线程渲染,与 Graphics Jobs 属性类似,开启 Multithreaded Rendering 功能可以带来可观的性能提升。

3. 开启 Use Incremental GC

Unity 2019 及后续版本新增了增量垃圾回收 (Incremental Garbage Collection),虽然使用的仍然是原 GC 收集器,但将原本全量式垃圾回收拆分成多块任务渐进式执行,这在垃圾较多或者应用规模较大时能分多阶段执行一次完整的垃圾回收,从而降低卡顿感。

4. 使用 ARM64 打包

在 Target Architectures 属性中选择 ARM64 (Scripting Backend 需要选择 IL2CPP),使用 ARM64 打包应用可以提高应用的运行速度,降低包体体积。事实上 ARCore 将在 2022 年底终止对 32 位应用的支持。

5. 关闭深度要求

在没有深度传感器的设备上,关闭深度要求,即在 XR Plug-in Management → ARCore 栏,将 Depth 属性选择为 Optional 值,防止应用使用深度传感器和进行深度计算,节约计算性能。

6. 使用低图像渲染质量

高质量图像渲染设置在带来更高渲染表现的同时也会消耗更多的资源,这往往是导致性能问题的重要因素,牺牲微小的质量以求性能也是一种折中的办法,因此,为了保证适当帧率,通常建议将 Unity 默认的 Unity Quality Settings (Unity 质量设置)修改为 Very Low 值。具体操作为,在 Unity 菜单中依次选择 Edit → Project Settings → Quality,单击 Android 目标列下的 Default 属性下拉箭头,选择 Very Low。

7. 其他

在使用 ARCore 开发 AR 应用时,计算密集型功能有平面检测、图像检测、锚点、阴影生成、图像捕获处理、三维音频、三维视频,因此在使用这些功能时,需要注意使用度,或者在不需要时及时关闭这些功能特性以提高性能、节约资源。

15.7.3 AR 应用开发的一般注意事项

移动设备是边端设备,受限于硬件处理能力、功耗,为使渲染帧率达到 30 帧/秒,一般

建议场景中模型多边形数量不应当超过 50 万个。在设计、开发 AR 应用时，开发人员需要时刻关注性能问题，通常而言，开发与使用 AR 应用应当注意以下方面。

- (1) 使用 URP 渲染管线而不是 Unity 原 Build-in 管线。
- (2) 使用平面投射阴影代替 Unity 提供的 Shadowmap 阴影。
- (3) 及时关闭计算密集型功能。
- (4) 不添加不使用的可跟踪对象管理器。

参考文献

- [1] 汪祥春. AR 开发权威指南: AR Foundation[M]. 北京: 人民邮电出版社, 2020.
- [2] AR Foundation 文档 [EB/OL]. <https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@4.2/manual/>, 2022.
- [3] Subsystem 文档 [EB/OL]. <https://docs.unity3d.com/Packages/com.unity.xr.arsubsystems@4.2/manual/>, 2022.
- [4] ARCore 开发文档 [EB/OL]. <https://developers.google.cn/ar/develop>, 2022.
- [5] Augmented Reality Design Guidelines [EB/OL]. <https://designguidelines.withgoogle.com/ar-design/>, 2018.
- [6] Unity. Practical guide to optimization for mobiles [EB/OL]. <https://docs.unity3d.com/Manual/MobileOptimizationPracticalGuide.html>, 2018.

AR Foundation 增强现实开发实战 (ARCore版)

好评如潮

在元宇宙发展推动下, AR技术将更深入渗透到工业、教育、培训、军事、娱乐、游戏等产业中, 相关龙头企业将迅速拉开与竞争对手的差距, 同时, 越来越多的企业会进行布局, 先发红利窗口开始关闭。《AR Foundation增强现实开发实战 (ARCore版)》一书是技术人员投身AR行业的优秀参考资料, 是非常好的ARCore开发入门手册。

—— **欧阳正杰** 中国社会科学院经济学博士、美国特许金融分析师 (CFA) 持证者、
联合信用投资咨询有限公司投资总监

AR+元宇宙时代, 行业期待更多优秀人才的加入, 一起加速元宇宙建设和落地。《AR Foundation增强现实开发实战 (ARCore版)》系统阐述了AR应用开发, 从技术原理、理论体系到实际应用, 结构严谨, 通俗易懂, 有助于开发者迅速掌握AR开发相关技术。

—— **陶柏帆** 北京奇领空间数字科技有限公司CEO、中电万维信息技术有限公司首席AR行业技术专家

ARKit、ARCore与华为AR Engine作为AR三剑客, 越来越多地被不同行业、不同领域的开发者所使用, 将不断影响和改变人们的生活方式。《AR Foundation增强现实开发实战 (ARCore版)》一书深入浅出地讲述了AR技术原理, 理论与实践相结合, 从不同维度讲述了AR技术在不同应用场景下的使用方式与代码实现方法, 本书不仅适合软件开发人员及相关从业人员使用, 同样也适用于对AR技术感兴趣的爱好者。

—— **杨祥勇** 华为HDE、AR开发技术专家

VR/AR的产品应用日益深入, 元宇宙 (Metaverse) 的概念方兴未艾; 未来的10年, 人们必将从二维移动互联网, 跃迁到下一代以三维全息和多模态自然交互为核心特征的三维空间计算新时代。新的计算时代, 将革新人们工作/生产/生活的方式, 更高效、更自然、更沉浸、更有趣; 新的计算平台, 也必然会给广大AR应用开发者带来新的发展契机和提供展示聪明才智/创意的舞台。《AR Foundation增强现实开发实战 (ARCore版)》是汪老师又一力作, 恰逢其时, 必将给广大优秀AR开发者或团队提供新的营养工具和系统权威见解, 是一本不可或缺的AR应用开发参考书。

—— **Jon.Yang** AR增强现实领域知名产品专家、AR行业老兵

“元宇宙”可谓是当今最火爆的名词之一, 背后的增强现实技术经过多年发展已日渐成熟, 在交通出行、娱乐休闲等很多方面都扮演着重要角色。本书全面深入地阐述了AR的基础知识与工程实现方式, 让读者能够快速理解相关概念并搭建自己的AR项目, 深入浅出, 内容丰富, 结构合理, 脉络清晰, 非常适合希望了解并从事AR相关工作的读者阅读。

—— **张铎** 在读博士, 华为HDE、JBHI等权威期刊审稿人

资源下载·扩展阅读

清华大学出版社



书 圈

官 方 微 信 号

上架指导: 计算机/增强现实

ISBN 978-7-302-61218-6



9 787302 612186 >

定价: 109.00元

[General Information]

楞埜暄=410

SS鑒?15168266