班 级 2103053

学 号 21009200502

西安电子科技大学

本科毕业设计论文

题 目 基于虚幻引擎的跨平台游戏开发技术研究

学 院 计算机科学与技术学院

专 业 软件工程

学生姓名 马鋆宇

导师姓名 高悦

摘要

跨平台开发正成为游戏行业的一大课题。跨平台游戏指在多个平台（如个人电脑、手机、游戏主机）上具备相同内容和游玩体验的游戏。虚幻引擎作为成熟商业引擎，提供了成熟基建，便于开发者在此基础上开发跨平台部署的产品。本文从三个跨平台游戏开发存在的子课题出发，设计与跨平台渲染管线、新一代输入处理工具和资产及碰撞烘焙技术相关的模块，并对其进行测试和检验。最后总结模块设计和实现中的不足，探索更高效，成本更低的跨平台游戏开发策略。

关键词：虚幻引擎

ABSTRACT

Keywords: Unreal Engine

目录

第一章 绪论

1.1 研究背景及研究意义

近年来移动设备性能快速进步，新一代设备在核心性能上已经大幅缩小同高性能 PC & Console 的差距。同时，Nintendo Switch 的不断热销和 Linux 游戏生态的改进，催生了输入方式、屏幕尺寸、硬件组合多样化的掌上游戏设备生态。2020 年以来，开发以“一次开发，多端部署，管线共用，内容一致”为鲜明特征的跨平台游戏成为行业主流趋势之一。

虚幻引擎的渲染管线、关卡流送等基建在行业中十分突出，在近年来落地许多前沿成果。2020 年对前向渲染管线提供现代化支持；2021 年率先交付软动态全局光照技术，并提供原生“一 Actor 一文件”流送能力；2023 年为全资产提供虚拟几何体支持，等等。

经历 3-4 年的研发周期后，2024 年行业出现了一批高品质、涵盖多种主流品类的跨平台游戏。跨平台游戏开发需要大量前期投入和试错成本，近期基于虚幻引擎开发并上线的跨平台游戏几乎全部为数百人团队、5 亿成本之上的大型游戏。它们的开发过程中积累了许多宝贵经验，贡献了许多可供复现和整合的思路。在这个时间点，小型开发者通过借鉴这些经验和思路，开发出更高品质跨平台游戏已经成为可能。

据此开展的一系列研究，面向跨平台渲染管线、新一代输入处理工具和资产及碰撞烘焙技术三个跨平台游戏开发堵点，综合集成并改进技术方案，并量化评估其改进成果和现实意义。

1.2 国内外研究现状

现代渲染管线的研究经历了约 30 年历程。在介绍具体的电子游戏工业界研究之前，有必要先简述相关渲染技术的重要研究节点。

前向渲染（即 Forward Rendering）概念由来已久，与早期图形管线的发展紧密相关。Nvidia 在 1999 年推出了首款现代 GPU，标志着前向渲染过渡到可编程管线。

传统延迟渲染（Deferred Rendering）可追溯到 1990 年。Saito 和 Takahashi 最早提出了延迟着色（即 Deferred Shading）的概念。[1]2004 年，Rich Geldreich 和 Matt Pritchard 在当年的游戏开发者大会（Game Developer Conference，后称 GDC）上正式提出了延迟渲染概念：先执行深度测试，再进行着色计算，将本来在三维空间进行光照计算放到了二维空间进行处理。延迟渲染的概念中，先将所有物体都先绘制到屏幕空间的缓冲（即 G-buffer，Geometric Buffer，几何缓冲区，对应显示硬件的特定区域）中。之后，逐个按光源对物体进行遍历，完成着色。计算被深度测试丢弃的片元（像素）的着色在传统前向渲染管线中会产生不必要的开销，延迟渲染将能够避免这一点。

Lauritzen, Andrew 在 2010 年对延迟渲染管线进行了进一步展望，引入了 Tiled-Based 延迟渲染等概念。

延迟渲染管线表现优异，但由于 G-buffer 对显存容量和带宽要求较高，它对移动端支持不佳。随着 iPhone 和 App Store 的推出，21 世纪 10 年代移动端游戏市场规模快速发展（Chen Zou），移动游戏开发成为一个课题。

2012 年，Harada 等人的研究介绍了拓展前向渲染管线（即 Forward+，有时也成为“现代前向渲染管线”；本文中如无特别指出，均用“前向渲染管线”或“Forward+”简称“拓展前向渲染管线”），它是对传统前向渲染管线的拓展，可视为一种通过光照剔除和仅存储对像素有贡献的光源来渲染许多光源的方法。具言之，Forward+拓展了一个光照剔除 Pass，用以响应对高效多光源渲染的需求。作为前向渲染流程的拓展，它还可避免渲染庞大的 G-buffer，十分节省内存和显存。Takahiro Harada, Jay McKee, 和 Jason C.Yang 的研究对 Forward+渲染管线和传统延迟渲染管线的性能进行了量化比较。

虚拟几何体、软动态全局光照和 MegaLight 等技术可追溯至 5-7 年内。Chris Wyman 等人在研究中提出了 ReGIR 采样算法，对光源的重要性进行采样，大大

[1] Saito 和 Takahashi, 《ComprehensibleRenderingof 3-D Shapes》.

拓展了场景中动态灯光数量上限；Daqi Lin，Cem Yuksel 和 Chris Wyman 在后来的工作又将原基于屏幕空间的算法扩展到体积介质的多维路径空间，实现了低噪声、交互式的体积路径追踪。

Brian Karis，Rune Stubbe，Graham Wihlidal 等人提出了在虚拟纹理系统基础上创新的填充方案，在光栅化流程之前剔除高精度资产等三角面，允许用户将超高精度的资产整合到场景中；这一研究为工业界的“虚拟多边形”技术奠定了基础。

Daniel Wright、Krzysztof Narkowicz、Patrick Kelly 等人的研究将动态间接光照工程化，并直接推动了软动态全局光照在游戏引擎中的实现。这些工作解放了预烘焙光照的许多局限。

这些技术为下面工业界的实践奠定了基础。

国内工业界在跨平台游戏技术研究方面走在前列。2024 年，Shangli Liang 和 Yang Shi 介绍了一种利用虚幻引擎延迟管线，涵盖资产生产、运行时处理等多方面内容，以实现高效利用和跨平台适配的资产生产、合并、剔除策略和技术。报告内容中提及，《三角洲行动》团队采用 LOD 链管理细节层次，将碰撞网格与物体网格分离，在高性能平台和低性能平台保证一致的游戏性内容。同时，开发了专门的外部工具 Jade Hub 用以指定资产平台和设置资产层级，在导入环节对资产进行验证，通过 JSON 文件传递资产路径、材质模板等关键信息。

Tieyi Zhang 介绍了一种通过定制虚幻引擎的默认延迟管线来实现“奇异博士”传送门的技术。开放式传送门通常会使渲染工作量翻倍，这给性能带来了挑战。报告中提供 CPU 和 GPU 优化的高级概述，以及用于将传送门调整为其他英雄技能的许多策略。 最终，《漫威争锋》玩家将体验到逼真、可自由放置和交互式的传送门。这些传送门在保持相对较高帧率的同时，还能让玩家看到对面的情况。该解决方案可以在另一侧显示实时场景，允许角色、子弹和其他物体无缝通过，而不会妨碍玩家的主视图。

Shaoyong "Abel" Zhang 从移动游戏视觉特效艺术家的角度，对 14 种着色器进行数学分析。最终，他提出了一种在设计之初就能保证移动端兼容性，保证表现并提高性能的策略。

国外工业界在跨平台游戏开发方面的研究在今 3-5 年同样取得许多成果。Kosuke Tanaka 和 Takashi Komada 介绍了一种对中型开发者来说稳健和具备高品

质效果的风格化视觉效果实现路径。这一报告对本文所着重对小型开发者尤为关键，它介绍了在不修改引擎的前提下，完全利用虚幻引擎的 Forward+管线进行定制化光照开发的技术，以及利用延迟管线制作全局光照的许多实践。

以上是国内外业界通过对虚幻引擎渲染管线的发掘和改进进行的跨平台游戏开发研究实践。新一批跨平台游戏开发技术将拓展中小型游戏开发者的工程能力，允许其制作更大规模、更高品质、具更强竞争实力的产品；工业界正在尝试落地包括但不限以上列出的新一代技术。当前技术门槛和试错成本较高，我们可以观察到行业正处于大型团队初步交付、中型团队落地开发、小型开发者借鉴跟进的阶段。已初步具备进行相应研究和探索的条件。

1.3 研究内容及章节安排

本研究立足跨平台游戏开发技术。主体上，基于小团队视角，复现、探索和集成跨平台渲染管线在 Toon Shading，定制化室内光源（阴影）、SSAO 和全局光照方面的应用、新一代输入处理工具及资产及碰撞烘焙技术。之后通过 Unreal Insight 等性能测试工具，对整体解决方案进行定量评估。

本文的章节安排如下：

第一章，绪论。介绍了研究背景与意义，然后对跨平台游戏开发的基本原理和国内外研究现状进行介绍，最后陈述本文研究内容和计划。

第二章，跨平台游戏开发原理。在绪论基础上，介绍与本文工作相关的模块、算法、三种渲染管线和全局光照等技术的原理与特征，并列举一些工业界实践中的具体成果。

第三章，基于虚幻引擎的工程设计。本章节详述工程中使用的虚幻引擎功能以及三个模块的具体设计。

第四章，测试、评估和讨论。首先介绍性能评估技术和流程，并展示性能评估结果和（不涉及性能部分模块的）实际效果展示。

第五章，结论。总结全文，分析研究取得的成果和存在的不足之处，并讨论未来可能的工作方向。

第二章 基本概念与技术

2.1 跨平台渲染管线背景信息

2.1.1 GPU 着色器

GPU 着色器是 GPU 中完全可编程的模块。GPU 渲染管线是更宏观的渲染管线之一部分。图形渲染管线即在给定虚拟相机、三维物体、光源、照明模式和纹理等条件下，生成对应二维图像的过程，分为应用程序阶段、几何阶段和光栅化阶段。

1987 年，Cook 最先提出了可编程着色的思想。1999 年 Nvidia 发布了首个包含定点处理功能的图形芯片，并将其命名为 GPU。该芯片可以处理顶点着色器、几何着色器、CLipping、屏幕映射这些几何阶段的运算，亦能处理光栅化阶段中包括像素着色器的多种算法。GPU 提供了对顶点着色器（Vertex Shader)、几何着色器（Geometry Shader）和像素着色器(Pixel Shader)的可编程支持。现代着色阶段中，三套着色器共享同一套编程模型，使用类如 HLSL 和 GLSL 一类的着色器语言，下文中会使用它们。

顶点着色器在三个着色器中位于最靠前的阶段。它可以在现代 CPU 或 GPU 中执行。2001 年，DirectX 8 中首先引入这一概念；如今，现代 OpenGL、Vulkan 和 Metal 仍在使用。顶点着色器可对创建、修改或变换顶点。这里的“修改”有专门强调的价值：多边形顶点的参数包括位置、颜色、法线、纹理坐标，等等。顶点着色器会输出顶点空间变换后（一般是齐次裁剪空间）的坐标。顶点着色器输出的数据可在光栅化过程后发送给像素着色器，也可发送给几何着色器，或二者兼有。

几何着色器位于顶点着色器和像素着色器之间，接收图元输入。它只能输出点、折线或三角形条。后文使用的部分虚幻引擎基建大量设计使用了几何着色器，但本文中不会直接使用它，而是通过已有基建进行几何变换。

像素着色器在现代图形 API 中也称为片元着色器（Fragment Shader），对每个顶点执行着色方程，输出顶点的颜色。它的输入实际上是顶点着色器的输出。严谨地讲，前文的“发送”实际上是像素着色器内部流程的一部分。光栅化如 Clipping、屏幕映射、三角形设定的许多步骤在像素着色器内部完成。

在像素着色器之后（有些时候是像素着色器的最后），进行合并操作。合并阶段中，进行模板缓冲（即 Stencil-Buffer）和 Z 缓冲（也就是 Z-Buffer）。虚幻引

擎渲染管线中将这部分数据暴露给用户，允许其对顶点数据进行涉及 RGBA（也就是颜色和 Alpha 值）的加减乘和其它位运算，下文中我们将利用这一点。

虚幻引擎通过深度缓存算法实现透明排序。首先介绍这个算法。

在渲染开始前，将深度缓存中的所有像素值初始化为一个极大值（表示无限远）或一个特定值（取决于深度范围定义）。同时，初始化颜色缓冲为背景色。

然后逐个渲染场景中的图元（如三角形）。对于每个图元覆盖的像素，首先计算该像素对应的三维空间点的深度值（Z 值）。这个深度值将会与深度缓冲中对应像素位置存储的深度值进行比较。 如果新计算的深度值比缓存中的值更小，表示该顶点更靠近观察者；更新深度缓存中该像素的深度值为新的、更近的深度值。当然，也要更新颜色缓存中该像素的颜色为当前图元的颜色。如果新计算的深度值不小于缓存中的值，则表示该像素已被更近的物体遮挡，丢弃当前图元的这个像素信息，不更新任何缓存。

当所有图元都处理完毕后，颜色缓存中存储的就已经是最终可见表面的图像。这一算法显然无法对半透明物体进行排序。后文中会提及延迟渲染管线和前向渲染管线的分别，但在这里已经可以首先论述：半透明物体的数据存在 G-buffer 等之中，属于延迟渲染管线。因此，在具体的工程实践中，笔者尽力使用前向渲染管线功能来达成跨平台支持，非常谨慎地规划所有与半透明相关的功能。

虚幻引擎首先对所有物体进行渲染，填充数据到 G-buffer 和深度缓冲之中。另外，虚幻引擎引入了一个单独通道（半透明）来处理半透明物体的排序。如前文所述，它还为用户提供了一个单独的缓冲（称为 Custom Stencil），允许用户从模板缓冲中获取数据，或用自定义数据覆盖之。

2.1.2 BRDF 模型

双向反射分布函数（Bidirectional Reflectance Distribution Function，下文称 BRDF）模型在绝大多数图形算法中都得到采用。它用于描述光反射现象，即入射光线经过某个表面反射后在出射方向上的分布模式。BRDF 中的几个变量使用一些辐射度量学单位。下面的公式中会用到：

辐射通量（Radiant Flux，又译作光通量，辐射功率）描述的是在单位时间穿过截面的光能，或每单位时间的辐射能量，通常用 Φ 来表示，单位是 W，瓦特。

$$\Phi = \frac{dQ}{dt}$$

其中的 Q 表示辐射能(Radiant energy)，单位是 J，焦耳。

对一个点（比如说点光源）来说，辐射强度表示每单位立体角的辐射通量，用符号 I 表示，单位 $W \cdot sr - 1W \cdot sr^{-1}$ ：

$$I = d\Phi d\omega I = \frac{d\Phi}{d\omega}$$

辐射率（Radiance，又译作光亮度，用符号 L 表示）,表示物体表面沿某一方向的明亮程度，它等于每单位投影面积和单位立体角上的辐射通量，单位是瓦特每球面度每平方米。辐射率的微分形式：

$$L = d2\Phi dA\cos\theta d\omega L = \frac{d^2\Phi}{dA\cos\theta d\omega}$$

其中：Φ是辐射通量，单位瓦特（W）；Ω 是立体角，单位球面度（sr）。

辐照度（Irradiance，又译作辉度，辐射照度，用符号 E 表示），指入射表面的辐射通量，即单位时间内到达单位面积的辐射通量，或到达单位面积的辐射通量，也就是辐射通量对于面积的密度。用符号 E 表示，单位为瓦特每平方米。辐照度可以写成辐射率（Radiance）在入射光所形成的半球上的积分：

$$d\Phi dA = E = \int \Omega L(\omega)\cos\theta d\omega \frac{d\Phi}{dA} = E\int_{\Omega} \quad = \quad L(\omega)\cos\theta d$$

其中，Ω 是入射光所形成的半球。L(ω)是沿 ω 方向的光亮度。

BRDF 从出射角度出发，描述入射方向和出射方向能量的关系。对反射表面进行积分后，以出射辐射率的微分与入射辐照度的微分作比，便得到 BRDF 方程的微分形式：

$$f(l,v) = dLo(v)dE(l)f(l,v) = \frac{dL_o(v)}{dE(l)}$$

这些描述是为了推导着色方程。在工程中，更多使用对 BRDF 着色方程的某种拟合。下面列出原始着色方程为：

$$L_o(v) = \sum_{k=1}^{n} f(l_k, v) \otimes E_{L_k} cos\theta_{i_k}$$

其中 k 是每个光源的索引。

虚幻引擎采用基于物理的 BRDF 模型。由 Cook 和 Torrance 提出，Cook-Torrance BDDF 模型是最早的基于物理的 BRDF 模型。它将菲涅尔反射引入到着色模型之中，定义为：

$$f(l, v) = \frac{F(l, h)G(l, v, h)D(h)}{4(n \cdot h)(n \cdot v)}$$

其中， F 为菲涅尔反射函数(Fresnel 函数)；G 为阴影遮罩函数（Geometry Factor，几何因子），即未被 shadow 或 mask 的比例；D 为法线分布函数(NDF)。

2.2 三种着色管线

基于光栅化的渲染技术因其计算效率高，在对帧率有严苛要求的交互式应用（如电子游戏、虚拟现实和实时模拟）中占据主导地位[1, 3]。二十年来，图形硬件性能快速提升，应用场景对视觉真实感的需求也在日益增长。光栅化着色管线在这一过程中经历了显著演进。本节中，重点阐述三种具有代表性的渲染管线技术：前向着色（Forward Shading，也称 Forward Rendering，下同）、延迟渲染（Deferred Rendering）以及拓展前向渲染（Forward+ Rendering），对其核心原理、发展脉络及关键研究突破剖析之。

这里的着色管线也称为“渲染管线”，它们是运行在 GPU 渲染管线上具体的渲染技术。特别指出，前文提及的渲染管线是一个抽象的概念流程，与此有别，见图 2.2。为避免与前文混淆，在标题中使用“着色管线”替代之。后文中如无特别声明，笔者都使用更加通行的说法“xx 渲染管线”指代着色管线。

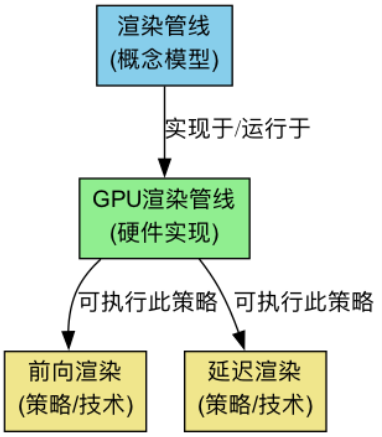

图 2.2 渲染概念之间的关系

2.2.1 前向渲染管线

前向渲染是最为经典且直观的渲染范式。在此管线中，每个几何图元（通常是三角形）独立地经过顶点处理、几何处理（可选）、光栅化和片段处理等阶段，得到最终图片。核心的光照计算发生在像素着色器中，针对每个通过深度测试和模板测试的像素（潜在的片元），根据其材质属性、表面法线、视角方向以及场景中的相关光源信息，直接计算最终颜色值。

前向渲染的早期形态与固定功能图形硬件紧密相关。此时，光照计算模型相对简单且在硬件中以硬编码存储。20 世纪 90 年代中后期可编程着色器的引入允许用户在着色器中实现复杂的材质和光照模型。

前向渲染管线结构简单，对 MSAA 和透明物体的支持也更好。前向渲染管线中，光照计算复杂度通常与场景中的“对象数量 × 光源数量”成正比（即 $O(\text{对象} \times \text{光源})$）。当场景中存在大量动态光源时，渲染性能会急剧下降。此外，对于复杂场景，大量像素可能被后续渲染的物体遮挡，（这一现象称作过度绘制，Overdraw），在这些最终不可见的像素上浪费了大量的片段着色计算。传统前向渲染管线对许多现代图形特性的支持也十分有限。

2.2.2 延迟渲染管线

延迟渲染管线将渲染过程解耦为两个主要阶段：几何 Pass（Geometry Pass）和光照 Pass（Lighting Pass）。在几何 Pass 中，场景中的不透明物体的几何信息（如世界坐标位置、法线、漫反射颜色、高光属性等）首先被渲染（“写入”）到一组屏幕空间的中间缓冲区中，这组缓冲区统称为几何缓冲区（Geometry Buffer, G-buffer）。在光照阶段，通过读取 G-buffer 中存储的逐像素信息，对屏幕上每一个可见的像素执行一次光照计算。这样，光照计算的复杂度与屏幕像素数量和光源影响范围相关，而与场景的几何复杂度（对象数量）基本解耦，显著提高了处理大量光源的效率，并从根本上避免了对被遮挡像素进行光照计算的浪费。用户还可以添加自定义 Pass，为游戏增加多种风格化表现。延迟渲染管线也提供对半透明物体的支持。

延迟着色的概念雏形可以追溯到 Deering 等人 (1988) 以及 Saito 和 Takahashi (1990) 的研究工作，后者明确提出了 G-buffer 的概念。该技术在早期并未普及，直到图形硬件具备了足够的特性支持，特别是 Shader Model 3.0 于 2004 年引入的多渲染目标（即 Multiple Render Targets, MRT）能力，才使得延迟渲染得以高效实现并逐渐成为主流。诸如《怪物史莱克》(2001) 和《杀戮地带 2》(2007) 等游戏的早期实践和演示推动了延迟渲染管线在业界的采纳。2010 年前后，出现了基于瓦片的延迟渲染（Tile-Based Deferred Rendering，下文称 TBDR）由于 G-buffer 的读写操作十分频繁且会随着着色器和自定义 Pass 规模膨胀而大幅上升，延迟渲染会带来较高的内存带宽消耗，这让许多面向延迟渲染管线开发的图形特性难以适配到低性能平台上。

2.2.3 拓展前向渲染管线（Forward+）

拓展前向渲染是一种旨在结合前向渲染和延迟渲染优点的混合渲染技术。它本质上仍是前向渲染流程，保留了对透明度和 MSAA 的良好支持。其核心创新在于引入了一个高效的光照剔除（Light Culling）预处理阶段。在该阶段，通常利用计算着色器（Compute Shader），将屏幕空间划分为二维的“平铺”（Tiles）或者进一步考虑深度信息划分为三维的“聚类”（Clusters）。然后，针对每个 Tile 或 Cluster，快速构建一个仅包含对其产生影响的光源列表。在后续的（传统）前向

渲染着色阶段，每个片段着色器仅需从其所属 Tile/Cluster 的光源列表中读取光源信息并进行计算，从而将需要处理的光源数量大幅减少，使其能够高效应对大量光源的场景。

该技术的相关思想（如平铺着色）大约在 2011 年由 Olsson 和 Assarsson 等人提出。Harada 等人 (2012) 正式提出了 Forward+ 的概念并进行了详细阐述。虽然它增加了光照剔除阶段的实现复杂度，但相较于需要为透明度和 MSAA 添加复杂变通方案的延迟渲染系统，其整体架构可能更为统一和简洁，且通常内存带宽压力更小。由于其在性能、效果兼容性和实现复杂度之间的良好平衡，Forward+及其变体（如 Clustered Forward）已成为现代跨平台游戏和高性能渲染引擎的重要选择。

2017 年，虚幻引擎添加对 Forward+的支持。2023 年，虚幻引擎移动端 Forward+管线迎来重大改进。

第三章 基于虚幻引擎的功能设计

虚幻引擎发布于 1998 年，最早用于支持 Epic Games 的游戏《虚幻竞技场》，逐渐发展成为商业引擎。如今，虚幻引擎已成为跨平台游戏开发的主流选择。当前最新版本为虚幻引擎 5.5。

本项目主要使用虚幻引擎渲染管线、增强输入系统和碰撞系统方面的基建，在其基础上作定制化开发。3.1 节分析跨平台游戏开发面临的技术问题并划分切片，3.2 至 3.4 节分别介绍弹性美术管线、输入与控制系统和碰撞生成等外围模块的设计和实现细节。

3.1 跨平台技术框架概述

3.1.1 目标平台差异性分析

在利用虚幻引擎进行跨平台游戏开发时，深入理解目标平台间的固有差异性是构建稳健且高效技术框架的基石。本段涵盖 GPU、内存子系统特性、屏幕规格以及用户输入方式等多个维度，它们直接影响用户体验和开发策略 [Source 1]。

1. 图形处理性能与特性支持存在显著差异

不同平台间，GPU 计算能力具量级上的差距。以浮点运算性能（FLOPS）为例，高端 PC 平台的 GPU（如 Nvidia RTX 4090）可达到数十 TFLOPS [Source 3]，主流游戏主机（下文称 Console，如 PlayStation 5 和 Xbox Series X）则在 10-12 TFLOPS 这一量级 [Source 1, 3]。作为对照，移动平台的 GPU 性能跨度极大，从几 GFLOPS 到数 TFLOPS 不等，即使是高端移动 GPU（如高通 Adreno X1-85 或苹果 M 系列 GPU）其峰值性能也与主机和 PC 存在显著差距；尽管其发展迅速，已能超越部分老旧主机 [Source 1, 3]。

巨大的性能鸿沟直接决定了不同平台上图形特性的支持程度和实现复杂度。高端 PC 和 Console 藉由强大的 GPU 算力和更大的显存带宽（PC 可达 1 TB/s 以上，Console 约 400-560 GB/s） [Source 1, 5]，能够支持如实时光线追踪、高分辨率纹理、复杂 Pass 结构和着色器模型以及高密度几何体渲染等尖端图形技术。移动平台 GPU 性能和内存带宽（通常在几十至数百 GB/s 范围） [Source 1, 5] 相对

有限，往往需要依赖 TBDR [Source 1] 和纹理压缩技术（如 ASTC）来缓解带宽压力 [Source 1]。移动游戏往往在特性支持上有所取舍，一些方案包括简化光照模型、降低多边形规模或采用屏幕空间技术的近似实现。

操作系统（Windows, macOS, Android, iOS）及其对应的图形 API（DirectX, Metal, Vulkan, OpenGL ES）间在功能集、驱动效率和底层硬件访问权限上亦有区别 [Source 1]。在虚幻引擎提供了渲染抽象层来屏蔽部分底层差异的情况下，开发者仍需考虑降级图形特性，在构筑图形方案时考虑共用能力 [Source 1]。

2. 屏幕尺寸、分辨率与输入方式的多样性

移动设备市场存在大量不同的屏幕尺寸、分辨率（从低于 HD 到超过 QHD）和像素密度（DPI 范围从 300 到 500+ PPI） [Source 1, 7, 33]，高度碎片化。Console 通常面向客厅电视或显示器，目标分辨率较为固定。 [Source 1, 7, 35]。PC 平台介于两者之间，1080p（1920x1080）仍是主流，但 1440p 和 4K 分辨率的普及率快速增长，且支持包括超宽屏在内的各种宽高比 [Source 1, 7, 39]。

平台间输入方式多样。如简单的位移操作，在键盘上对应 4 个按键的布尔值，在控制器中对应一个二维浮点数。也就是说，无法简单地通过替换映射来达成跨平台适配。

3.1.2 技术切片划分

基于上述情况，功能设计分为三个模块：弹性风格化美术管线、输入处理中间件和碰撞管理功能。在这里，笔者简述模块设计思路；下文将介绍其具体实现。

美术管线在这里指代图形特性的选型、制作和部署。承接前文，所有具体实践服务于三个目标：共用基座——核心流程一致；表现可伸缩——部分特性具备高端化能力；不修改引擎——小型团队跨平台开发的客观需求[hifi]。虚幻引擎也使用“PC 端渲染管线”和“移动端渲染管线”的提法。作为基建，这两条管线中是延迟管线和拓展前线管线之具体特性按不同方式组合而成的。下文中使用“某某特性从属于某某渲染管线”这样的提法时，如无特别指出，一般不指代虚幻引擎面向用户提供的“渲染管线”概念，而是更加通用的“某某特性具备延迟管线（前向管线）的特征”。管线部分大致划分如下：

1. 一般核心特性。二分 Toon Shading 和自定义灯光在拓展前向管线中完成。非天光阴影和 SSAO 正常产生投影使用自行开发的排线和网点效果。描边效果在延迟管线中完成，仅添加 1 个 Pass。

2. 后处理特性。高光溢出效果和 Bloom 基于虚幻基建作简单调整。在 G-Buffer 中，存储了自定义 Stencil 用以遮挡剔除。

3. 全局光照。在虚幻引擎的基础上实现；使用 Lightmass 重要性体积，只在想要的区域烘焙 Lightmass Map。预设项目是恒定侧边视角，在高端平台上启用该机制。

输入处理系统。虚幻引擎现有的输入基建为 UEnhancedInput，较为完备地考虑了通用场景。本文中，基于 Gameplay Ability System（后称 GAS）拓展了输入处理能力，将输入操作直接暴露给游戏逻辑，简化了跨平台输入修饰工作。材质函数包含以下几个区别于普通材质节点的节点：

虚幻引擎的输入基建如图 3.2。

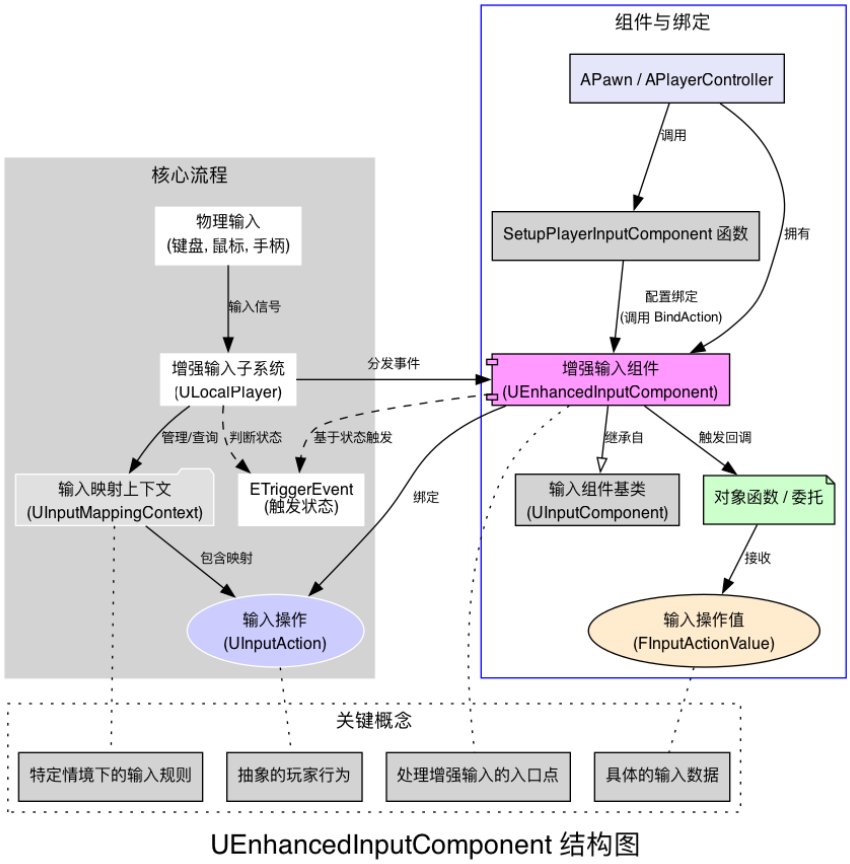

图 3.2 虚幻引擎输入基建图示

碰撞管理功能。不同平台所使用的资产精度有所分别。为实现碰撞与几何的解耦，使用 Pocedural Mesh 构建了通用碰撞机制。

（3）预设项目背景

虽然本研究不涉及交付完整、可玩的游戏本身，但有必要预设一些项目背景。只有在本段限定的游戏类型之内，下文中所描述的特性才能正确工作。

本研究的意义在于探索小型团队可复制的跨平台游戏开发技术。类《银河战士》是小型团队近年来投入开发的主流品类之一。下文的技术选型也以该类型游戏开发为背景，打造卡通风格化画面表现。

3.2风格化弹性美术管线实现

3.2.1 美术管线核心特性设计

整体来说，在虚幻引擎渲染管线的基础上进行定制。使用的工具以材质蓝图（Material Blueprint）为主。一部分逻辑使用 BP 编写，也有许多特性使用 HLSL 这类着色器语言。虚幻引擎提供了 Custom 节点，由此可将自定义着色器封装到整体管线中。

主光照构建策略。基本的 Toon Shading 在拓展前向管线中完成。Toon Shading 的核心目的是明亮、锐利和快速；作为最核心的基本表现，也需要做到完全共用。主光照的目的为使物体产生自阴影（亮暗面）。亮面与暗面之间有明确分界线，且暗部会被指定一个特定颜色。最终，决定使用唯一天光照亮整个场景。所有物体并不直接响应天光，而是基于其方向生成明暗二分。另外，暴露了暗部颜色和过渡级数给材质实例，支持人工干涉最终色彩表现。材质组件面板见图 3.3。所需要的地形大小和表现的细节，可以设置大小不一的尺寸用于各种需要。每个地形组建的高度数据存储在单独的纹理中。因此，纹理的大小必须是顶点数量的二次方。两个相邻地形组件的共同顶点行是重叠的，分别存储在各组件中。

场景光照构建策略。项目中同时存在 3D 模型和 2D Sprite，需要分别制作符合资产特性的灯光。Unreal 自带灯光在延迟管线和前向管线上表现差异较大，且容易生成伪影和噪点。因此，本研究中笔者在前向管线上自定义 3D 灯光。虚幻

引擎拥有较为成熟的贴花技术，由于风格化场景不追求物理精确的光照强度衰减，使用其作为自定义光源非常合适。

灯光被封装为一个单独蓝图 Actor，以世界位置为中心，向一定半径内的物体表面投射光照。为了改善表现，制作了多种类似现实中异形镜头光圈的 Mask 进行投射。可以通过搭配这些灯光，丰富场景视觉效果。另外，制作了剔除 Box，可将光照在场景中“截断”：既可以让用户灵活定制光照表现，亦具防止光照渗透到物体背面之用。

2D 灯光需要解决默认情况下 3D 灯光照亮 Sprite 边缘，“纸片感”强烈的问题。为解决这个问题，在延迟管线中自定义光照性质。按距离的平方而非距离衰减，过渡会更自然。2D 灯光响应 Sprite 法线。理论来说，人工绘制法线效果最优；但从工程角度，这不现实。最终，在 Sprite 材质中增加特性：使用边缘检测算法自动生成边缘法线。

表 3.1 虚幻引擎可接受的地形组件表

总体大小（顶点数）	四边形数/分段	分段数/组件	地形组件尺寸	地形组件总数
8129 x 8129	127	4 (2x2)	254x254	1024 (32x32)
4033 x 4033	63	4 (2x2)	126x126	1024 (32x32)
2017 x 2017	63	4 (2x2)	126x126	256 (16x16)
1009 x 1009	63	4 (2x2)	126x126	64 (8x8)
1009 x 1009	63	1	63x63	256 (16x16)
505 x 505	63	4 (2x2)	126x126	16 (4x4)
505 x 505	63	1	63x63	64 (8x8)
253 x 253	63	4 (2x2)	126x126	4 (2x2)
253 x 253	63	1	63x63	16 (4x4)
127 x 127	63	4 (2x2)	126x126	1
127 x 127	63	1	63x63	4 (2x2)

3.2.2 周边特性

接下来介绍风格化高光/阴影、环境光遮蔽等技术相关细节。

具言之，在前向管线中构建半色调网点纹理，再将其投射至场景中。Bloom 效果由半色调网点填充，阴影和环境光遮蔽由排线填充。项目中没有使用图片纹理，而是完全由 GPU 驱动，程序化生成纹理以节省内存。使用 HLSL 编写着色器生成两类纹理：网点和排线。效果图见图 3.4。基于有向距离场生成纹理。通过期望纹理坐标、点的颜色、背景颜色、点的半径就可以绘制出一个点。制作排线时，将 Y 轴的 UV 赋 0。网点和填充线的尺寸、密度、颜色均暴露给材质实例调节。见图 3.5。

后续，使用三平面投射算法将其混合到几何体表面。Bloom 效果混合半色调网点，阴影混合排线。在有些场景中，可能需要根据实际表现对场景进行暗部调节。项目中使用与上文类似的技术制作了贴花阴影，可满足相关需求。Bloom 区域在延迟管线中生成，存储在 G-Buffer 中。这一数据仅用于后续基于屏幕空间的纹理混合。在强度控制上，笔者使用虚幻引擎自带的 Bloom 算法。额外添加的网点混合已经能够带来相对完备的视觉表现。

在类《密特罗德》类型的电子游戏中，常常要处理物体遮挡关系判定。传统的判定方式是射线检测等；在这里提出使用深度通道进行编码的检测方式。用户可以访问虚幻引擎渲染管线中的 Z-Buffer，或用自定义深度（Custom Stencil）按 Actor 覆盖之。自定义深度使用 8 位二进制编码。之后，添加一个自定义 Mask，对像素的深度数据按位运算。简单对比最终结果，即可得知物体遮挡关系。见图 3.6。

3.3 输入与控制系统开发

3.3.1 控制层基础设施

从操作系统和硬件驱动接收信号这一步骤由虚幻引擎基建 UEnhancedInput 实现。本项目中，相关模块的开发情况如下：图 3.7 描述了一个角色控制器的上下游资源。

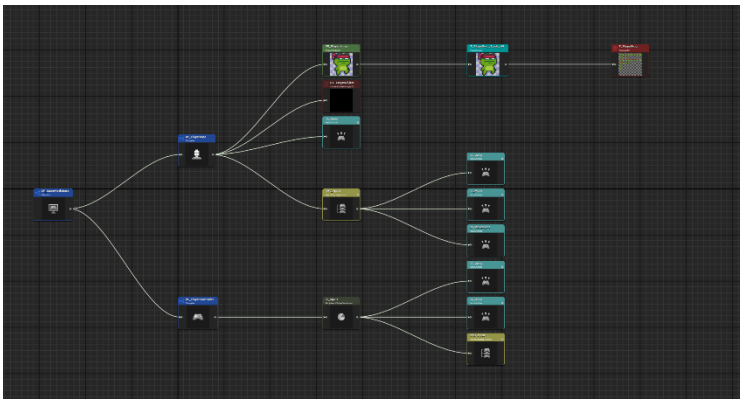

图 3.7 控制层基础设施及其资源

从 Character 类继承，创建了自定义 Character 类 GAS_Character。GAS_Character 类的所有基本功能（包括移动和输入）都绑定到 GameplayAbility，供下面的集成调用。添加的 C++类如：

```
public:
    AGAS_PaperCharacter();

    /** ability system */
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Abilities, meta = (AllowPrivateAccess = "true"))
    class UAbilitySystemComponent* AbilitySystem;
    UAbilitySystemComponent* GetAbilitySystemComponent() const
    {
        return AbilitySystem;
    };
    /** ability list */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Abilities)
    TArray<TSubclassOf<class UGameplayAbility>> AbilityList;

    /** BeginPlay, PossessedBy override */
    virtual void BeginPlay() override;
```

```
    virtual void PossessedBy(AController* NewController) override;

};
```

在 GAS_Character 类的基础上，得到蓝图类 PlayerBase。从头设计其中的组件。使用组合的设计思想，纳入一系列与输入控制、碰撞和响应的组件。

3.3.2 GAS 集成

将输入系统同 GAS 集成在工程上有一系列好处。通过添加一个抽象中间层，项目得以将输入处理逻辑和游戏角色逻辑解耦，并能方便地通过输入直接触发游戏逻辑（或通过输入数据本身进行游戏逻辑判定）。项目引入了第三方中间件 NinjaInput 作为 GAS 集成的基础。笔者的工作主要集中于将该中间件集成到项目的角色控制系统中，之后根据项目需求进行配置：将输入操作映射到移动、交互等具体能力，以验证跨平台输入支持方面的可行性。中间件及虚幻引擎相关基建的组件和具体绑定模式见表 3.1。表格中大部分组件都没有开头的字母 U，是因为虚幻引擎用“U+组件名”的方式命名组件所属的 C++类。本项目中大多数逻辑在 BP 中完成，添加的 C++文件仅仅起到定义作用。中间件自身的工作流程见图 3.8。

表 3.1 中间件及相关虚幻引擎基建的组件和关系

组件	所属系统	主要职责	与 Enhanced Input 交互	与 GAS 交互
UEnhancedInputComponent	EnhancedInput	绑定 UInputAction 到回调函数。	接收来自 UEnhancedInputLocalPlayerSubsystem 的触发事件。	(间接) 通常其回调函数会调用 NinjaInput 组件。

UInputAction	EnhancedInput	代表一个抽象的输入动作（如跳跃、射击）。	由 UEnhancedInputComponent 触发。	(间接) 作为信息传递给 NinjaInput。
UNinjaInputManagerComponent	NinjaInput	接收 Enhanced Input 事件，管理 UNinjaInputHandler，协调输入处理流程，管理输入缓冲（如果 UNinjaInputBufferComponent 存在）。	通常由 UEnhancedInputComponent 的回调函数调用，接收 UInputAction 和 ETriggerEvent。	获取 UAbilitySystemComponent，并将处理委托给具体的 UNinjaInputHandler。
UNinjaInputHandler (及其子类)	NinjaInput	定义如何处理特定的 UInputAction 和 ETriggerEvent 组合，决定如何与 GAS 交互。	由 UNinjaInputManagerComponent 调用，基于 UInputAction 和 ETriggerEvent 进行判断和执行。	(核心交互点) 调用 UAbilitySystemComponent 的方法来激活/取消技能 (通过 InputID, Class, Tag 等)。
UNinjaInputBufferComponent	NinjaInput	(可选) 实现输入缓冲逻辑，允许在特定时间窗口内“记住”输入，以便稍后执行。	由 UNinjaInputManagerComponent 或 UNinjaInputHandler 查询和管理。	(间接) 影响何时向 GAS 发送激活/取消指令。
AnimNotifyState_InputBuffer	NinjaInput	(可选) 在动画蒙太奇中定义输入缓冲的有效时间窗口。	无	(间接) 控制 UNinjaInputBufferComponent 的启用/禁用状态。

UAbilitySystemComponent	GAS	管理技能、效果、属性，处理技能的激活、取消和执行。	无	接收来自 UNinjaInputHandler 的激活/取消请求，执行相应的技能逻辑。
UGameplayAbility	GAS	定义具体的技能逻辑。	无	由 UAbilitySystemComponent 实例化和管理，其激活/取消由 UAbilitySystemComponent 控制。

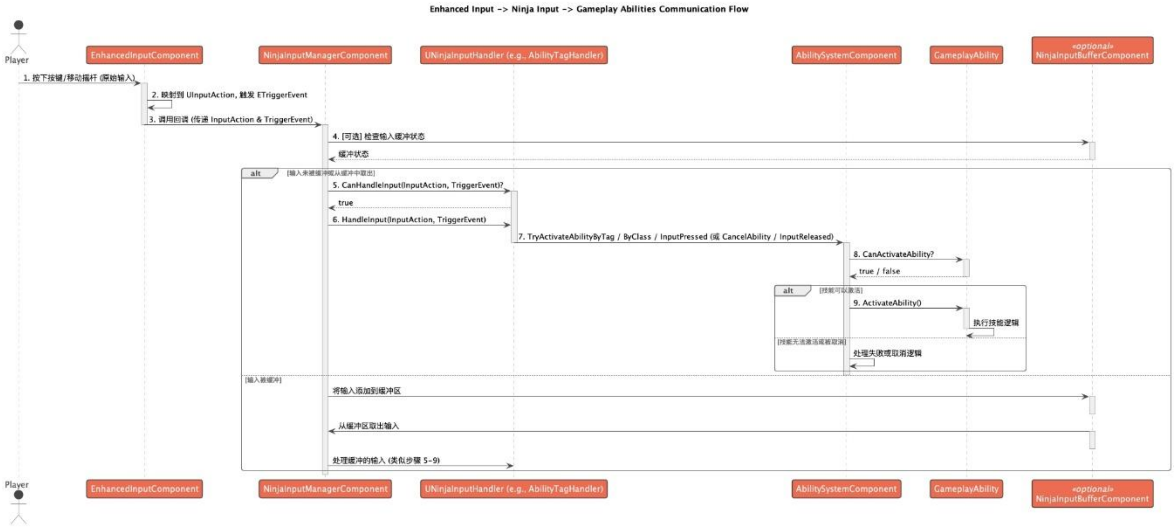

图 3.8 中间件 Ninja Input 的工作流程

具体而言，笔者访问 UNinjaInputHandler，并在其中拿到来自 UEnhancedInput 组件的输入信息 UInputAction。中间件允许开发者在 InputHandler 中注入自己的逻辑。在本项目中，由于 GameplayAbilityComponent 在 GAS_Character 类中已经激活，笔者可直接通过蓝图将其绑定到自行设计的 GameplayTags 上。等效 C++伪代码为：

```
void UInputHandler_SampleActivation::HandleTriggeredEvent_Implementation(UNinjaInputManagerComponent* Manager, const FInputActionValue& Value, const UInputAction* InputAction, float ElapsedTime) const { UAbilitySystemComponent* ASC = Manager->GetAbilitySystemComponent(); ASC->TryActivateAbilitiesByTag(AbilityTags, true); }
```

GameplayTags 包括基本移动输入（Move 和 Jump），在项目设置中预置即可，此处不再赘言。之后，在 InputHandler 组件中即可将 Gameplay Event 发送（对应上图中的 Activate Ability）到 BP_PlayerBase（也是 Character）类中，只需要在对应接口中实现具体逻辑即可。也就是说，InputHandler 向上接管抽象输入，自身提供对输入本身逻辑的处理空间，向下允许具体蓝图通过接口实现逻辑。只需要在 InputHandler 中制作对不同输入逻辑的支持，即可实现跨平台适配。

3.4外围工程模块

3.4.1 碰撞生成系统

碰撞生成系统的核心设计目的是：在几何体和碰撞解耦合的背景下，基于空间点的采样生成经过这些点的碰撞盒。虚幻引擎提供了 DynamicMesh 基建，赋予开发者在运行时建立、修改和烘焙碰撞的能力。本项目中将利用它。具体而言，碰撞系统包含两个模块 BP_FrameManager 和 BP_CombinedBoundary。模块设计见图 3.9。

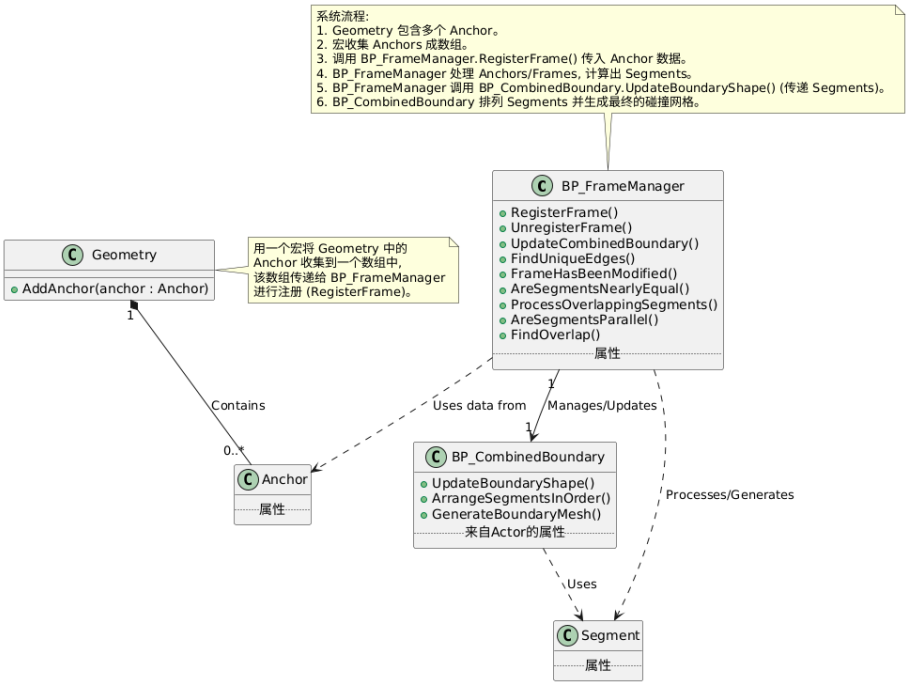

图 3.9 碰撞系统模块设计

系统流程:

1. Geometry 包含多个 Anchor。

2. 宏收集 Anchors 成数组。

3. 调用 BP_FrameManager.RegisterFrame() 传入 Anchor 数据。

4. BP_FrameManager 处理 Anchors/Frames, 计算出 Segments。

5. BP_FrameManager 调用 BP_CombinedBoundary.UpdateBoundaryShape() (传递 Segments)。

6. BP_CombinedBoundary 排列 Segments 并生成最终的碰撞网格。

笔者选用样条线作为生成 Dynamic Mesh 的几何数据。最终，这一系统产生简化和去重后的样条线，供相应函数生成和更新，见图 3.10。

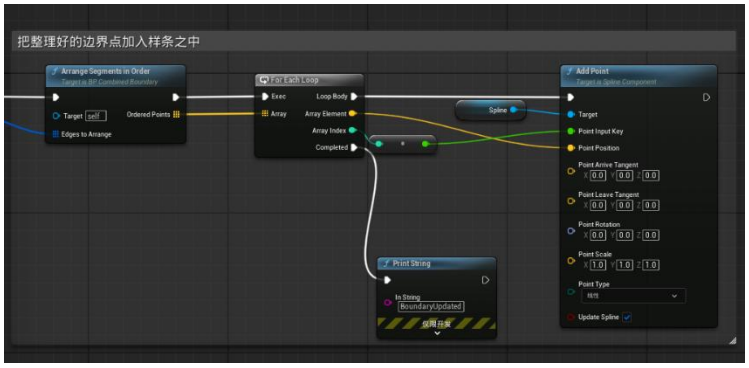

图 3.10 以样条线格式存储边界形状

除了图片描述的两个模块之外，还封装了 BP_FrameGenComponent。任何 Actor 添加该组件后，都具备输出锚点和向 BP_FrameManager 注册的能力。万事俱备之后，在 GenerateBoundaryMesh()中烘焙为 StaticMesh。虚幻引擎的碰撞概念抽象为 Object Channel 和 Trace Channel。碰撞盒自身占据一个 Object Channel 类型，其它类（如 Character）会被 Block；碰撞盒之间通过单独的 TraceTV 通道互相检测，用以在布尔运算之前对边界预先剔除。

第四章 测试、评估和讨论

对于较重要的功能，制作了测试场景来对照效果和量化性能。对于无法量化的部分（如视觉效果），侧重于模拟不同设备场景，比照表现差异；而对于可量化的部分（性能），使用原生方案作为对照，量化比照性能数据。

4.1 图形性能测试

本节对第三章所述的渲染管线进行性能测试。测试分为基准性能测试和跨平台性能对比。基准性能测试将最终解决方案和 Unreal 默认管线性能进行对比，跨平台性能对比展示在高性能（PC&Console）/低性能平台（Mobile，Nintendo Switch）上的性能和表现差异。各项测试数据均在 Unreal 独立线程中使用 Unreal Insight 抓取，并经过相关性检验。图 4.1 展示了测试对象和宏观指标，下文将展示这一过程。

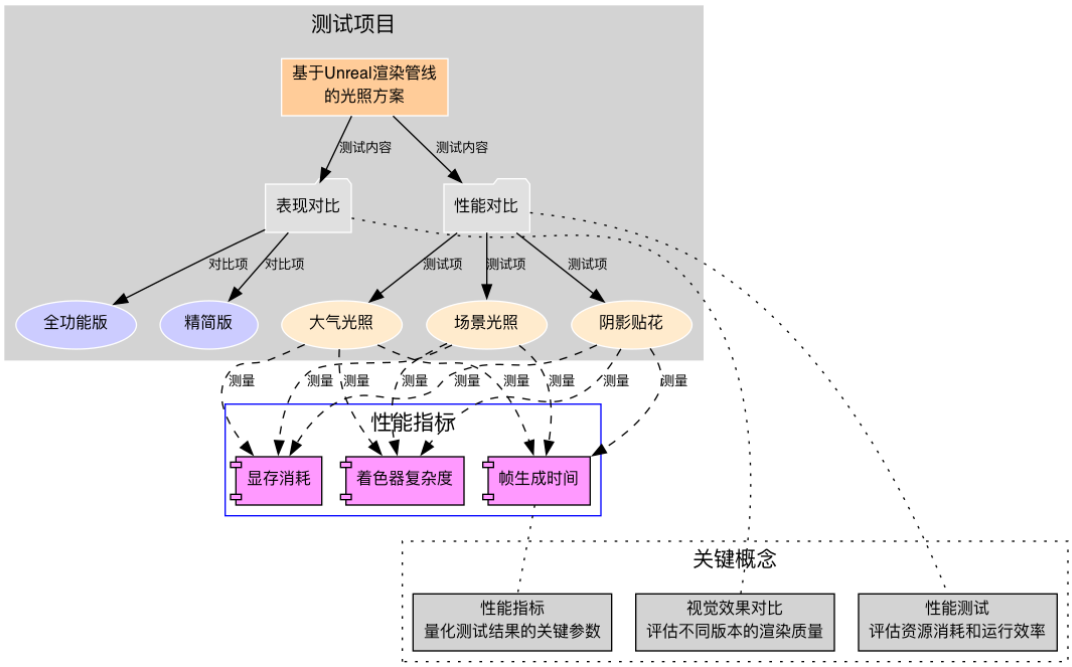

图 4.1 测试对象和宏观指标

4.1.1 基准性能测试

测试场景搭建。在搭载 AMD Ryzen 9 5900HX 和 NVIDIA GeForce RTX 3080 Laptop GPU 的 Windows 计算机上进行测试。为模拟实际游戏运行效果，导入外部资产搭建了测试场景。测试场景自身参数见图 4.2。

对照组为同资产在虚幻引擎默认 PC 渲染管线下的性能参数。抓取了一段时间内性能参数（包括 Frame Time、Draw Thread Time、Draw Call 规模、GPU Base Pass 耗时、Lighting 耗时），详细导出数据见附录 1。对附录中的数据进行一致性检验。

（过程待补充）

剔除离散数据之后，整理的性能对比结果见表 4.2。

（待补充）

着色器复杂度对比见图 4.3.1、图 4.3.2。

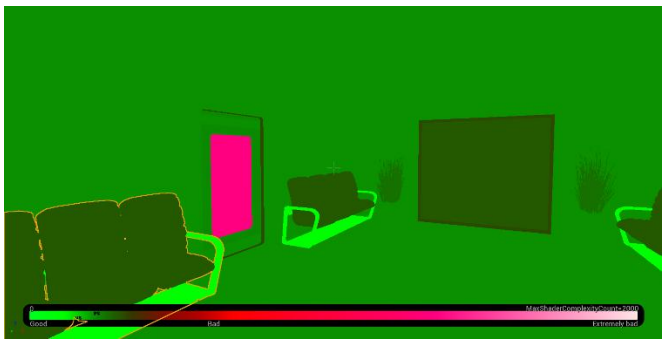

图 4.3.1 项目方案的着色器复杂度

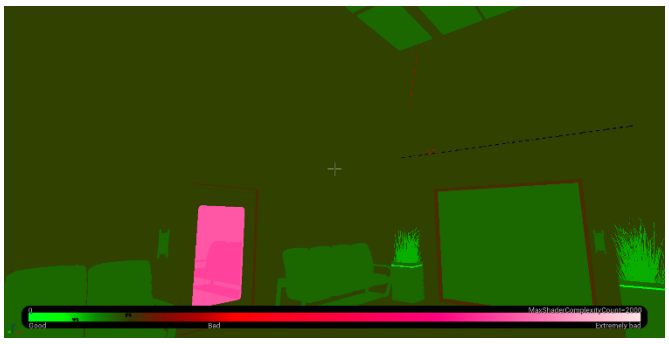

图 4.3.2 虚幻引擎默认 PC 渲染管线下的着色器复杂度

综上，可认为本项目的美术管线消耗的 GPU 和 CPU 资源普遍少于虚幻引擎默认 PC 渲染管线。

4.1.2 跨平台性能对比

在高端与低端平台的弹性部署上，笔者没有做任何复杂的事情。有些高端特性（如 GI）原生不支持移动端，模拟时会直接降级；资产流送参照虚幻引擎默认解决方案处理。唯一值得提及的是，由于资产本身没有任何阻止向下兼容的特性，直接将高端平台的低规格内容用于低端平台即可。降级细节见表 4.3：

（在后续步骤完成后会相应更新。这里的某些参数（顶点）只是当前状态，完成后再来排版）

资产/设置类型	PC平台代表性资产/设置	移动平台代表性资产/设置	量化指标	指标对比值 (PC -> Mobile)
模型 - LOD	LOD 0/LOD 1/LOD2	LOD 2 / LOD 3	顶点数	场景总体：24k -> 15k verts
纹理 - 分辨率	2K / 4K	512 / 1K	分辨率 (像素)	2048x2048 -> 1024x1024
纹理 - 压缩格式	BCn (DXT)	ASTC / ETC2	压缩格式 / 质量/大小权衡	ASTC 6x6 vs DXT1
材质 - 复杂度	全部项目内特性+GI	全部项目内特性，室内烘焙光照	Shader指令数	252 -> 198
材质 - 特性	Clear Coat, Subsurface 等	Base Color, Normal, Roughness	支持的Shading Model	高级 -> 基础

项目能够在移动预览（独立线程模式）下正常加载和运行（见图 4.4），足以论证解决方案的跨平台兼容性。不过，在测试时遇到了 Android 包体连接到 Unreal Insight 上的困难，因此采用了基于 Unreal Editor 内置工具的模拟分析、引擎特性分析、内容差异对比以及引用公开数据相结合的方法。模拟数据（如 ProfileGPU 在移动预览模式下的结果）不能完全代表真实设备的绝对性能，但可以有效揭示相对差异和渲染瓶颈。笔者将尽力结合现实中硬件性能差异和 Epic Games 的相关技术分享，弥补逻辑上的不足。

图 4.4 测试场景在移动预览（独立线程模式）下的运行效果

(数据待清洗，简单平均后结果如下。完成后再来排版)

性能指标 (Stat Unit, Stat GPU, ProfileGPU)	PC预览 (SM5/SM6) - 模拟值	移动预览 (ES3.1) - 模拟值	单位	数据来源/获取命令
Frame Time (总帧时)	18.5 ms	11.0 ms	ms	Stat Unit
Game Thread Time	8.2 ms	7.8 ms	ms	Stat Unit
Draw Thread Time	6.5 ms	4.0 ms	ms	Stat Unit
GPU Time (总)	18.5 ms	11.0 ms	ms	Stat Unit
Draw Calls	1650	720	count	Stat GPU
Primitives Drawn (Triangles)	2,800,000	1,500,000	count	Stat GPU
ProfileGPU: Base Pass	7.5 ms	4.5 ms	ms	ProfileGPU
ProfileGPU: Shadow Depths	4.2 ms	1.8 ms	ms	ProfileGPU
ProfileGPU: Lighting	3.8 ms	1.5 ms	ms	ProfileGPU
ProfileGPU: PostProcessing	2.0 ms	0.7 ms	ms	ProfileGPU
ProfileGPU: HZB (Occlusion Culling)	0.5 ms	0.8 ms	ms	ProfileGPU

综上所述，这一美术管线总体达成了设计目的：在高低端平台上做到弹性表现的同时，最大化做到共用流程和资产。

4.2外围模块测试

4.2.1 输入模块测试

设计测试场景来比照不同输入设备（键鼠、游戏手柄、虚拟摇杆）的兼容性和输入延迟。构建这个场景非常简单：虚幻引擎提供了全局计时器，只需要利用它打出输入时间即可。源数据见附录 2，经过清理的数据见表 4.4。

（数据待排版）

此外，还打出了输入处理栈。可见到同一输入操作先后由键盘和游戏手柄出发，证实了该模块对热插拔的兼容性。

4.3 不足之处和后续展望

测试也反映出现有方案的缺陷。如：

1. 未能在 Android 设备上原生验证，仅借用虚拟化技术转义运行。

2. 对最新引擎特性的追赶和涉猎不足。虚幻引擎推出的一系列新技术（MegaLight、新材质系统、移动端 VG）等都在不断对齐移动端对先进图形特性支持的上限，尽管还没有完全进入正式版，但已经对笔者的研究内容构成挑战。本文中部分结论的时效性可能有限。

3. 高端平台表现没有拉开明显差异。这一方面是时间、成本等因素让工业化项目更易追求表现上限的原因，另一方面也是笔者缺乏为项目从头制作美术资产能力所致。

4. 本研究使用的测试场景是为了突出特定的跨平台渲染特性对比而设计的。虽然涵盖了多种光照、材质和模型复杂度，但其规模和复杂度可能未能完全代表一个完整商业游戏项目的平均或峰值负载水平。据此，接下来可以从三个方面改进：

1. 首要工作是克服 Android 打包和部署的技术障碍。这可能涉及：简化测试项目以定位具体问题、尝试使用不同版本的引擎或编译/打包工具链、更深入地学习 Android NDK/SDK 配置知识等。成功部署后，应在多种具有代表性的 Android 设备（覆盖低、中、高端不同性能层级和主流 GPU 供应商上，使用 Unreal 自带的工具（如 Unreal Insights）和第三方工具（RenderDoc）进行全面的性能分析。获取并对比真实的帧率、CPU/GPU 线程耗时、Draw Call/Triangle Count、内存占用峰值与均值、功耗以及温度数据，以验证和补充当前基于模拟的发现。”

2. 拓展测试场景和复杂度，进一步评估在更高负载下的跨平台性能表现和瓶颈。使用或构建更接近商业游戏规模和复杂度的场景进行测试，包含更多动态元素、更复杂的 AI 行为、更丰富的粒子效果，等等。

3. 拓展项目规模，纳入更多新特性。积极尝试并验证虚幻引擎最新特性，构建规模更大、更具时效性的研究项目。

第五章 总结

本

致谢

参考文献