



南京大學

研究生毕业论文 (申请工程硕士学位)

论 文 题 目 全动飞行模拟视景系统中

数据交换子系统的设计与实现

作 者 姓 名 陈氢

学 科、专 业 名 称 工程硕士（软件工程领域）

研 究 方 向 软件工程

指 导 教 师 冯桂焕副教授

2023 年 4 月 23 日

学 号：**MF21320019**

论文答辩日期：xxxx 年 xx 月 xx 日

指导教师： (签字)

Design and Implementation of Data Exchange Subsystem in Full Flight Simulator Visual System

by

Chen Qing

Supervised by

Professor Feng Guihuan

A dissertation submitted to
the graduate school of Nanjing University
in partial fulfilment of the requirements for the degree of
MASTER OF ENGINEERING
in
Software Engineering



Software Institute
Nanjing University

April 23, 2023

学位论文原创性声明

任何收存和保管本论文的单位和个人，未经作者本人授权，不得将本论文转借他人并复印、抄录、拍照或以任何方式传播，否则，引起有碍作者著作权益的问题，将可能承担法律责任。

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含其他个人或集体已经发表或撰写的作品成果。本文所引用的重要文献，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：_____

日期：_____

南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目：全动飞行模拟视景系统中数据交换子系统
的设计与实现

工程硕士（软件工程领域）专业 2021 级硕士生姓名：陈氢
指导教师（姓名、职称）：冯桂焕副教授

摘 要

飞行模拟不只是普罗大众概念里一类体验飞行环游的游戏，其同时肩负着飞行训练这一更严肃使命。依托全动飞行模拟机进行日常训练是每一位飞行员的重要科目。达到飞行训练要求的模拟机称为全动飞行模拟机。目前我国的相关设备基本依赖进口，近年来国际局势剧烈变动，加之中国商飞 C919 客机取得中国民航局颁发的合格证，相应全动飞行模拟机的自主研发必须快马加鞭。

本项目立足于全动飞行模拟机中承担视觉效果的视景系统。仿真机是一台模拟机的核心，视景系统的运作需要仿真机的指令驱动。然而结合文档和网络抓包发现仿真机是一个相当底层的设备，其数据只经过以太网协议封装。这与基于游戏引擎开发的视景系统全然不处于同一层级，它们无法直接交流。为此本文设计并实现了视景系统中的数据交换子系统，其主要充当网络协议栈和翻译的角色。一方面它绕过操作系统的网络协议栈直接对仿真机侧的数据进行解封和封装，另一方面它负责仿真机自拟的数字表示形式与常规认知的数字间的转换。通过以上两点搭建仿真机与视景系统沟通的桥梁。

此过程中使用到 ProtoBuffer 数据交换协议来提升序列化的效率；引入了仿真机指令与自定义指令的映射机制屏蔽仿真机间的差别。此外实现了视景系统中基本的飞机飞行与计算反馈信息的逻辑，构成了完成数据流动的闭环，对数据交换结果也有了形象的观察。目前该视景系统可以在 CAE 仿真机操控下，以至少 60 帧率实现地景数据库中各机场附近环绕飞行，标志着迈出了动起来的坚实第一步。以此为根基，后续对天气、灯光、植被等多种系统进行研究与引入，尽早达成训练用模拟机的验收标准。

关键词：视景系统；数据交换；自研引擎；飞行模拟

南京大学研究生毕业论文英文摘要首页用纸

THESIS: Design and Implementation of Data Exchange Subsystem
in Full Flight Simulator Visual System

SPECIALIZATION: Software Engineering

POSTGRADUATE: Chen Qing

MENTOR: Professor Feng Guihuan

Abstract

Flight simulation in most people's concept may be a kind of game to experience flight. However, it's a more significant function for flight simulation to assist flight training. Daily training by professional simulator is an important subject for every pilot. Simulators for training have strict requirements such as identical cockpit, six-degree freedom acceleration device, and similar landform of worldwide airports, which is totally different from games. These simulators are named full flight simulator.

This project is based on the visual system who undertakes visual effects of full flight simulator. Visual system exchanges data with simulator when running. It receives data from simulator to render surroundings, and feedback flight infomation on real time. This paper focuses on data exchange in visual system and application of basic flight data. First of all, in order to achieve real-time rendering with stable frame rate, efficient and controllable data exchange is needed. This system uses Protocol Buffers to improve data exchange efficiency. It, meanwhile, is able to send data packets at a fixed rate to ensure smooth visual effects. Then domestic existing visual systems will not compatible with other companies' equipment voluntarily, because all of them are binded with imported simulator. As a new product, this visual system not only serves the coming domestic simulator, but also adapts data exchange protocols of exsiting introduced devices to expand market influence. Last but not least, on the basis of completing data exchange, the system achieves flight in Earth-Centered Earth-Fixed coordinate system with essential flight data. It can also interact with the terrain system to provide feedback for the simulator.

At present, under the control of CAE simulator, the system realizes the circling flight around airports in the landscape database at least 60 frame rate. The flying plane

in the visual system marks a crucial first step. The next step is to study and introduce weather, lighting, vegetation and other systems, so as to up to standard of D-Level simulator as soon as possible.

keywords: Visual System, Data Exchange, Self-developed Game Engine, Flight Simulation

目 录

目 录	iv
插图清单	vii
附表清单	x
第一章 引言	1
1.1 项目背景	1
1.2 国内外相关技术的发展概况	2
1.2.1 飞行模拟机	2
1.2.2 视景系统	4
1.3 本文主要工作	5
1.4 本文组织形式	6
第二章 相关技术概述	7
2.1 WinPcap 架构	7
2.2 ProtoBuffer 协议	8
2.3 Tbuspp 中间件	9
2.4 Nagle 算法	9
2.5 CrossEngine 游戏引擎	10
2.6 WireShark 抓包工具	11
2.7 帧缓冲机制	11
2.8 插值算法	11
2.9 航空坐标系及坐标系转换	11
2.10 本章小结	14
第三章 数据交换子系统的需求分析与概要设计	15
3.1 视景系统整体概述	15
3.2 数据交换子系统需求分析	16
3.2.1 涉众分析	16
3.2.2 功能性需求	17
3.2.3 非功能性需求	19

3.2.4 数据交换子系统用例设计	20
3.3 数据交换子系统概要设计	26
3.3.1 系统逻辑视图	28
3.3.2 系统开发视图	29
3.3.3 系统进程视图	29
3.3.4 系统部署视图	30
3.4 数据交换子系统详细设计	31
3.4.1 仿真机侧数据交互模块	31
3.4.2 数据转换模块	33
3.4.3 图像生成器侧数据交互模块	34
3.4.4 数据同步模块	36
3.4.5 飞行数据处理模块	36
3.5 本章小结	37
第四章 数据交换子系统的详细设计与实现	39
4.1 仿真机侧数据交换模块	39
4.1.1 顺序图	39
4.1.2 仿真机建立连接	40
4.1.3 数据接收与反馈	42
4.1.4 多仿真机通用化	44
4.2 协议转换模块	46
4.2.1 顺序图	46
4.2.2 ProtoBuf 文件生成	47
4.2.3 协议转换算法	48
4.2.4 转换方法注册	49
4.3 游戏引擎侧数据交换模块	50
4.3.1 顺序图	50
4.3.2 Tbuspp 连接建立	50
4.3.3 Tbuspp 数据发送	53
4.3.4 稳定发送频率	53
4.4 飞行控制模块	54
4.4.1 顺序图	54
4.4.2 飞行位置与姿态计算	55

4.4.3 可操控摄像机	58
4.4.4 检测地形信息	60
4.5 本章小结	62
第五章 系统测试	63
5.1 开发 PC 环境测试	63
5.1.1 测试环境	63
5.1.2 功能测试	64
5.2 FFS 环境测试	69
5.2.1 测试环境	69
5.2.2 功能测试	70
5.2.3 性能测试	72
5.3 本章小结	72
第六章 总结与展望	73
6.1 项目总结	73
6.2 项目展望	74
致 谢	75
致 谢	77
参考文献	78

插图清单

1-1 D 级模拟机外部	3
1-2 D 级模拟机内部	3
2-1 WinPcap 结构	7
2-2 CrossEngine 编辑器	10
2-3 飞机自身坐标	12
2-4 三类坐标系	13
2-5 坐标系变换	14
3-1 视景系统整体架构	16
3-2 仿真机与图像生成器层级差异	18
3-3 数据交换子系统用例图	21
3-4 CAE 仿真机数据帧结构	21
3-5 数据交换子系统架构图	28
3-6 逻辑视图	29
3-7 开发视图	30
3-8 进程视图	31
3-9 部署视图	31
3-10 仿真机侧数据交互流程图	32
3-11 仿真机侧数据交互核心类图	33
3-12 数据转换流程图	34
3-13 数据转换核心类图	35
3-14 视景系统侧数据交互流程图	35
3-15 视景系统侧数据交互核心类图	36
3-16 飞行数据处理流程图	37
3-17 飞行数据处理核心类图	38
4-1 仿真机侧数据交换顺序图	39

4-2 仿真机环境初始化代码	40
4-3 链路层连接建立代码	41
4-4 创建工作线程代码	41
4-5 原始数据包接收代码	42
4-6 原始数据包解析代码	43
4-7 模拟数据处理代码	44
4-8 反馈消息逻辑	44
4-9 反馈消息实现代码	45
4-10 模拟数据下反馈	45
4-11 转换类模板代码	46
4-12 协议转换顺序图	47
4-13 ProtoBuffer 结构	48
4-14 数据转换算法	49
4-15 转换执行模板类	49
4-16 转换执行类注册代码	50
4-17 游戏引擎侧数据交换顺序图	51
4-18 引擎侧环境初始化代码	51
4-19 Tbuspp 配置文件	52
4-20 Tbuspp 初始化代码	52
4-21 引擎侧发送信息代码	53
4-22 Tbuspp 发送信息代码	53
4-23 限制频率代码	54
4-24 Windows 中禁用 Nagle	54
4-25 飞行控制顺序图	55
4-26 LLA 转换为 ECEF 坐标代码	56
4-27 ECEF 转换为 LLA 坐标代码	57
4-28 LLA 与 ENU 坐标系	58
4-29 摄像机切换代码	59
4-30 第一人称摄像机代码	59
4-31 第三人称环绕摄像机代码	60
4-32 自由摄像机代码	60
4-33 地形检测逻辑代码	61

4-34 射线发射代码	61
5-1 30HZ 频率限制效果	65
5-2 GIS 绘制路径	66
5-3 飞机第一视角	66
5-4 飞机环绕视角	68
5-5 自由视角	68
5-6 仿真机连接	70
5-7 服役中 FFS 运行效果	71
5-8 本视景系统运行舱内效果	71

附表清单

1-1 视景系统部分标准对比	3
2-1 ProtoBuffer 类型列表	8
2-2 引擎 Runtime 比较	11
3-1 涉众分析列表	17
3-2 系统功能性需求列表	19
3-3 系统非功能性需求列表	20
3-4 建立链路层连接用例描述表	22
3-5 收取数据帧用例描述表	22
3-6 发送数据帧用例描述表	23
3-7 读取模拟数据用例描述表	24
3-8 仿真机指令转自定义指令用例描述表	24
3-9 自定义指令转仿真机指令用例描述表	25
3-10 收发 Tbuspp 消息用例描述表	25
3-11 飞行控制用例描述表	26
3-12 检测地形信息用例描述表	26
3-13 切换摄像机视角用例描述表	27
4-1 不同仿真机字段比较	46
5-1 开发环境物理配置	64
5-2 开发环境软件配置情况	64
5-3 反馈信息对比	69
5-4 FFS 环境物理配置	70
5-5 帧率测试结果表	72

第一章 引言

1.1 项目背景

1920 年我国第一条民用航线开航，一百年后飞机早已是寻常百姓家的出行选择。《2021 年全国民用运输机场生产统计公报》[1] 显示，2021 年我国境内民用机场已达 248 个，在地域广袤的中西部，机场几乎是城市标配。同时《2022 年度全球民航航班运行报告》[2] 显示，新冠疫情背景下 2022 年度国内航线实际执行客运航班量仍达 239 万架次，且相比 2019 年疫情前刚恢复五成以上。航空较陆路有更复杂的安全因素，民用航班的普及离不开航空安全的精进。据《国际航协 2021 年全球航空运输安全报告》[3] 统计，2021 年客机所属涡轮螺旋桨飞机每百万次飞行发生 1.77 起损毁，5 年内平均值为 1.22。这一方面得益于成熟的飞机制造，另一方面全动飞行模拟机（Full Flight Simulator 以下简称 FFS）为飞行员提供身临其境的训练环境功不可没。

FFS 是由模拟座舱、视景系统、声音系统、运动系统、网络系统和仿真机构成的飞行训练工具 [4]。汽车驾驶员可以驾驶符合驾照类别的任意汽车，而民航飞行员想获取某型号飞机的驾驶资格，必须在对应型号 FFS 上进行日常训练。因此新机型若想步入市场，配套 FFS 不可或缺。我国航空市场广阔，且国产大飞机 C919 已蓄势待发，FFS 需求旺盛，但目前该领域仍主要依靠行业巨擘加拿大 CAE 公司。进口产品购置成本高，二次开发困难，且可能面临技术封锁。为降低成本与风险，FFS 的国产化迫在眉睫。

本文中基于自研游戏引擎开发的视景系统是最终实现 FFS 国产化的重要组成部分。一台 FFS 的核心是仿真机，它相当于整个 FFS 的后台，负责根据飞行员在模拟座舱中的输入操作计算生成各类指令。各个系统运作均由来自仿真机的控制。视景系统负责根据仿真机指令中的位置信息从地景数据库中加载周边地形地貌，并结合时间，天气等综合信息实时渲染座舱前方的仿真景象；飞行中产生的如碰撞等异常行为也由视景系统反馈给仿真机。业界将视景系统中负责逻辑计算和渲染的该部分统称为图像生成器（Image Generator）。由此可见开发视景系统离不开仿真机的支持，本视景系统便是基于国内占有量最高的

CAE 仿真机开发。

经过对多种仿真机抓包分析后，发现问题在于仿真机是一个非常底层的设备，从网络模型角度看其最高层是数据链路层，指令数据用以太网协议封装后便进行发送，接收数据也只能仅被以太网协议封装过，否则无法正确解读。因此无法通过工作于传输层以上的传统 socket 连接模式与本视景系统中的图像生成器交流。为了使位于不同网络体系结构层级的仿真机与图像生成器沟通，本文为视景系统设计了数据交换子系统。一方面连接仿真机，收发仅用以太网协议封装的数据帧，转换为自拟指令类型后，通过传统网络体系层层封装再与图像生成器交互。这不仅解决了图像生成器与仿真机的交流问题，还屏蔽了不同厂商仿真机间数据组织结构的差异，保留了搭载于多种仿真机上的可能。

其次在真实训练环境下，模拟座舱前方是一个球幕，需要多个图像生成器同时工作进行融合投影才能产生良好的视觉效果。但出于网络波动和 TCP 仅支持单播的特点，数据到达各图像生成器的时间存在差异，导致最终的融合投影产生画面撕裂。为解决该问题，在图像生成器侧设计了网络帧缓冲机制和多种插值方法，使得多台设备逻辑线程可以做到几乎同时使用相同的数据进行计算。

目前本视景系统以适配国内仍大量服役的进口仿真机为短期目标，同时也为搭载于孕育中的国产仿真机做准备，期待国产 FFS 能够早日走进各大飞行训练基地。

1.2 国内外相关技术的发展概况

1.2.1 飞行模拟机

世界上第一架飞机于 1903 年由莱特兄弟试飞成功，最初的飞行模拟机则诞生于 1910 年，仅具备三个维度的手动旋转功能 [5]。1917 年，法国的 Lender 和 Heidelbergof 发明了燃油驱动旋转的版本 [?]。而现代飞行模拟机的雏形则是发明于 1930 纯电气驱动的 Linker Trainer，它在 1937 年被美国航空公司引入。之后在二战和电子计算机的刺激下，飞行模拟机逐渐发展出更复杂体系和更真实的感官效果 [6]。

现代飞行模拟机的种类很多，从大的方面来看，基本可以分为试验用飞行模拟机和训练用飞行模拟机两大类。试验用飞行模拟机主要用于新型飞机研制或旧机型改进。训练用飞行模拟机早已拥有完备的标准。国际民航组织 ICAO

于 1995 年发布《飞行模拟鉴定标准手册》并持续更新 [7]，所有现代商业飞行模拟机均需要按照标准设计，通过鉴定后才可以用于飞行员培训。我国于 2005 年由中国民用航空总局正式颁发的 CCAR-60《飞行模拟设备的鉴定和使用规则》作为国内航空公司飞行员模拟训练设备的鉴定标准，并于 2019 年重新修订 [8]。《规则》中说明我国训练用飞行模拟机分为 A、B、C、D 四个等级，其中 D 级为最高标准，即要达成《规则》文件中的全部最高标准，才可以作为全程飞行训练的模拟机。C、B、A 三个等级则是在响应时间，功能完整性方面逐步放宽要求，可以用于一些专项训练。表 1-1 提供了《规则》中视景系统部分标准的对比。图 1-1 与图 1-2 提供了 D 级模拟机外部和内部样貌。

表 1-1: 视景系统部分标准对比

视景系统要求	A	B	C	D
视景系统不应具有导致不真实特性的光学不连续性和人工痕迹。	√	√	√	√
模拟机应当在每个驾驶员座位上提供连续最小水平 90°、垂直 40° 的准直视场。			√	√
视景系统应当提供着陆期间判断下降率（深度感觉）所必须的目视提示。		√	√	√
提供黄昏和黎明视景，保证环境光强度减弱的色彩表征。			√	√
模拟机应当能在起飞、进近和着陆期间表现雷暴附近的轻度、中度和重度降水的特殊天气。				√
模拟机应当表现全部机场灯光的真实颜色和方向性。				√



图 1-1: D 级模拟机外部



图 1-2: D 级模拟机内部

近年来我国用于飞行员训练的 D 级模拟机均来自 CAE、Flight Safety 等国外模拟机制造商。2020 年 8 月，北京蓝天航空科技有限公司研发的新舟 60 飞机对应的全动模拟机通过了 D 级飞行模拟机认证，开始打破国外模拟机制造商对于 D 级飞行模拟机研制的垄断，迈出了国产模拟机的坚实一步 [9]。但新舟 60 本已是几近停飞的老旧型号飞机，市场存量很小，更复杂的市场主流型号客机以及 C919 客机模拟机的自主研发仍在进程中。

1.2.2 视景系统

最早的飞行模拟机并不具备视景系统，驾驶员仅能在地面上体验旋转。在上世纪 30 年代，一个位于机身前循环播放的画轴被视为第一个视景系统。60 年代闭路电视的发展让视景系统有了新形态，让相机扫过带有场景的皮带再将画面投影至飞行员眼前 [10]，此设备可以模拟简单光照，但仍是二维视觉效果。计算机图像生成技术则将视景系统拉入三维时代，发展至今成为根据飞机位置调取场景数据库，并结合天气设置完成渲染的现代视景系统 [11]。

目前在视景系统方面，能够参与并通过 D 级飞行模拟机鉴定的视景系统主要为 CAE 公司的 Tropos 视景系统和 Flight Safety 公司的 VITAL 视景系统 [12]，它们都被使用在各公司自研的飞行模拟机上。RSI 公司则专注于视景系统开发，其研制的 Epic Visual System 视景系统已经超过 D 级标准，对 4K 分辨率图像实时渲染帧率已能够达到 120HZ[13]。而国内关于视景系统的研究开发还不能通过最高标准的验收，在当前国际背景下，研制出能够搭载于 D 级飞行模拟机上并通过鉴定的视景系统，对打破垄断突破技术封锁有重要意义。

无论是服务于娱乐休闲还是专业训练，如今一款视景系统的开发必然要基于基本的图形引擎，综合功能更强大的游戏引擎当然是更好的选择。在游戏引擎出现之前，需要数学、图形、物理等各个领域的专家齐聚一堂花费大量时间精力才能完成一个简单的游戏 [14]。游戏引擎则是集合图像渲染引擎、物理引擎、网络引擎、动画引擎、脚本引擎、人工智能引擎等于一身，将功能封装为组件供开发者直接调用，大大降低学习成本，缩短开发周期 [15]。目前最主流的商业游戏引擎莫过于 EPIC 公司的 Unreal Engine 以及 Unity Technologies 公司的 Unity3D。它们在技术上集成了各类游戏开发所需引擎，可以实现极高的画面质量，且支持 PC、移动端、游戏机等设备，达成多平台兼容，在业界运用程度高范围广 [?]。

杜等人基于 OGRE 面向对象图形引擎实现视景渲染 [?]，其主要研究了大

地形的渲染算法；董等人依托于视景仿真软件 Mantis，设计了针对某军用型号飞机的视景系统 [16]。本文中的民航视景系统则是基于腾讯自研游戏引擎 CrossEngine 开发。随着游戏市场越来越成熟，游戏产品已进入拼品质的时期，而且逐渐向全平台游戏发展。从业界来看，国外知名游戏厂商基本都有内部自研游戏引擎，且经过几代产品的迭代打磨，在业内已经具有相当的影响力，例如 EA 公司的 Frostbite[17]。国内网易游戏的 NeoX 和 Messiah 引擎也已为公司创造了巨大的价值。近年来受国际关系的影响，使用第三方商业引擎成为了一个潜在的风险，CrossEngine 便在此背景下诞生。使用自研引擎开发视景系统可以更加自由的调整渲染风格，从更底层角度提升渲染效率，助力达成 D 级模拟机的验收标准。

1.3 本文主要工作

本项目目标是开发能够搭载于 D 级全动飞行模拟机上的视景系统，本文主要描述该视景系统中数据交换子系统的设计与实现。该子系统旨在为只使用数据链路层协议的仿真机与使用更高层网络协议的图像生成器间搭建双向沟通的桥梁，是视景系统运作的基础。本文工作主要涉及以下几点：

- (1) 在项目开始前使用抓包工具对进口模拟机的输入输出数据包进行分析，发现其中只有数据帧的头部信息，说明仿真机是一个相当底层的设备，与基于游戏引擎开发的图像生成器并不在同一层面上。为解读数据帧中的具体信息，又结合有限的文档信息进行分析，部分确认了其数据组织结构与数字表示方法。
- (2) 基于上一步中的认知，对视景系统中的数据交换子系统进行了需求分析和设计。主要的功能要求是按照仿真机的既定规则收发数据帧，将其解析为本视景系统使用的指令集后再与图像生成器交流。其次为确保多台图像生成器最终投影画面的一致性，需要数据同步的功能。非功能需求则主要是运行帧率、稳定性和通用性。结合以上需求对系统架构进行了设计并完成技术选型。
- (3) 视景系统中的数据交换子系统实现部分分为四个模块，仿真机侧数据交互模块负责与仿真机这一底层网络设备的交互；数据转换模块负责仿真机指令与自定义指令间的转换，屏蔽仿真机的差异；图像生成器侧数据交互模块负责图像生成器这一高层应用的交互；数据同步模块负责多台图像生成

器同时运行时数据层面的同步。

- (4) 测试部分第一对于仿真机侧收包的频率稳定性做了测试，第二对于各种指令数据转换结果作为视景系统中的一部分，对数据交换系统的功能测试离不开最终的飞行画面。测试主要在开发 PC 和飞行训练基地 FFS 两种环境下进行。经测试飞机可以在模拟数据或仿真机的驱动下按正确路径飞行，且连续运行下帧率稳定。

1.4 本文组织形式

本文围绕视景系统中数据交换子系统的设计与实现展开论述，共分为六章，每一章的内容编排如下所述：

第一章引言。本章首先阐述了飞行模拟视景系统以及数据交换子系统的背景与项目意义，明确了本文的工作价值。之后对于国内外飞行模拟机、视景系统的技术发展历程做了概述，确定了全动飞行模拟机逐步国产化的宏观目标。

第二章相关技术概述。本章对于数据交换子系统中相关的软件和算法技术做了介绍。软件部分主要介绍了 WinPcap 架构、Tbuspp 中间件、CrossEngine 游戏引擎和 WireShark 抓包工具；算法部分则介绍了 ProtoBuffer 协议的编码方式，Nagle 算法，帧缓冲机制和几种插值算法。

第三章系统需求分析与设计。本章首先阐述了视景系统的整体架构，确认数据交换子系统在整个系统中的位置。之后子系统进行了需求分析，并确定了系统用例。在此基础上，结合逻辑视图、开发视图、进程视图和部署视图对子系统的概要设计进行说明。确认了系统的四个模块并结合顺序图和类图详细说明。

第四章系统实现。本章在需求分析的基础上，为仿真机侧数据交互模块，数据转换模块，图像生成器侧数据交互模块和数据同步模块结合顺序图、类图和时序图阐述了各自的详细设计，并通过解释关键代码说明了四个模块各自的实现细节。

第五章系统测试。本章进行了视景系统在开发 PC 和飞行训练基地 FFS 两种环境下的测试。首先各自说明了软硬件配置情况，之后在两种环境下各进行了部分功能测试和性能测试。结合各种图表说明了测试结果。

第六章总结与展望。本章对本文中的重点工作进行简要总结，并对目前本视景系统面临的问题及未来的发展方向进行了分析。

第二章 相关技术概述

2.1 WinPcap 架构

WinPcap (windows packet capture) 是由意大利人 Loris Degioanni 在 2000 年提出并实现的一个架构，目的在于为 Windows 平台应用程序提供访问网络底层的能力 [18]。传统 socket 通信中，两台主机之间通信，socket 接收到的内容都是已经过网络协议栈处理的通信内容，并不会含有如数据帧头，IP 头，TCP/UDP 等内容。WinPcap 功能在于独立于操作系统的网络协议栈而发送和接收数据帧，非常适合做网络协议分析、网络监控等工作 [19]。

抓包系统必须绕过操作系统的协议栈来访问在网络上传输的数据帧，这就要求一部分运行在操作系统核心内部，直接与网卡驱动交互。Winpcap 是针对 Win 平台上的抓包和网络分析的一个架构。它包括一个内核态的包过滤器，一个底层的动态链接库 packet.dll 和一个用户态的程序接口库 wpcap.dll[20]。本系统中用其收取和发送数据链路层上的数据帧，帮助视景系统与仿真机交流。

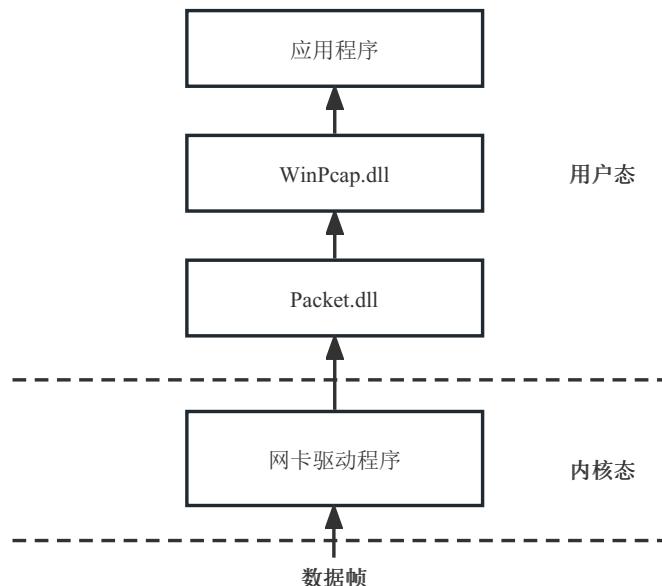


图 2-1: WinPcap 结构

2.2 ProtoBuffer 协议

Protocol Buffer 是 Google 提供的一种数据序列化协议，可用于通讯协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式 [21]。它属于一种二进制协议，相比较文本协议如 XML，JSON 等在体积和封解包速度方面有巨大的优势 [22]，适合诸如本系统的实时渲染应用。

ProtoBuffer 序列化后消息紧凑得益于巧妙设计的编码方式。首先其使用了 Varint 这种紧凑的表示数字的方法。它用一个或多个字节来表示一个数字，值越小的数字使用越少的字节数。这能减少用来表示数字的字节数。Varint 中的每个字节的最高位有特殊的含义，如果该位为 1，表示后续的字节也是该数字的一部分，如果该位为 0 则结束，其他的 7 位都用来表示数字，因此小于 128 的数字都可以用一个字节表示，而不是统一为 4 个字节。从统计的角度来说，一般不会所有的消息中的数字都是大数，大多情况下采用 Varint 可以用更少的字节数来表示数字信息。在进一步的优化中使用到了 ZigZag 编码，即用无符号数交错表示正数与负数，减少了负数的编码长度。

其次 ProtoBuffer 对 Key 的定义为 field_number+wire_type，field_number 表示该属性在结构中的编号，3 位的 wire_type 则指明了该属性的类型。Proto-Buffer 中共有 6 种 wire_type，如表 2-1 所示。

表 2-1: ProtoBuffer 类型列表

ID	名称	包含类型
0	VARINT	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	I64	fixed64, sfixed64, double
2	LEN	fixed64, sfixed64, double
3	SGROUP	group start (deprecated)
4	EGROUP	group end (deprecated)
5	I32	fixed32, sfixed32, float

解包过程中 XML 需要从文件中读取出字符串，再转换为 XML 文档对象结构模型。之后再从 XML 文档对象结构模型中读取指定节点的字符串，最后再将这个字符串转换成指定类型的变量。其中的计算消耗无疑非常巨大。

ProtoBuffer 在解包时只需要简单地将一个二进制序列，按照指定的格式读取到对应的结构体中就可以了。当然这也说明其必须要事先编写结构体文件给到接收方，否则无法正确解包。

2.3 Tbuspp 中间件

Tbuspp 是腾讯为完整解决游戏后台复杂与低延迟通讯需求而建立的服务网格中间件。随着分布式架构越来越复杂和服务越拆越细，开发人员迫切的希望有一个统一的控制面维护和管理各项服务。边车模式有效分离了系统控制和业务逻辑，使开发人员专注业务逻辑 [23]。服务网格是用于处理服务间通信的基础设施层，服务可以插入其中的代理网格，代理作为边车注入到每个服务部署中 [24]。服务间的调用通过代理实现，封装了其复杂性。

Tbuspp 的目标是构建功能完备的面向消息通讯中间件，能够完整解决全球同服部署模式下，游戏后台复杂与低延迟通讯需求，并能尽量降低开发与运维成本，做到简单易用。其与 Envoy 等流行的开源服务网格组件有两点本质区别，第一是提供专用 API 供应用服务调用，与 Envoy 采用透明方式劫持应用服务的流量存在显著差异。第二是基于 SHM 消息队列与应用服务交换消息，这点是针对游戏服务特殊的业务背景：游戏服务一般是有状态服务，希望在对服务快速重启更新的同时，保持游戏世界状态的连续性，因此往往需要将核心运行状态保存在 SHM 中，同时也基于 SHM 与边车交换消息，以便服务重启期间不间断消息收发。本系统中使用 Tbuspp 作为虚拟仿真机与游戏引擎间进行 TCP 消息沟通的插件。

2.4 Nagle 算法

在使用一些协议通讯时，如果每次只发送一个字节的有用信息，却要附带几十个字节的头部信息，这笔开销会增加拥塞情况的出现。John Nagle 就提出了一种通过减少需要通过网络发送包的数量来提高 TCP/IP 传输的效率 [25]，即 Nagle 算法。Nagle 算法核心是避免发送小的数据包，要求一个 TCP 连接上最多只能有一个未被确认的小分组，在该分组的确认到达之前不能发送其他的小分组。TCP 会搜集这些小的分组，然后在之前小分组的确认到达后将刚才搜集的小分组合并发送出去。

但 Nagle 算法也有弊端，对于实时性要求很高的交互上，我们不能使用 Nagle 算法 [26]。比如在网络游戏中，每一帧玩家的操作信息或状态改变并不会产生很大的包体，却需要及时的将状态同步到服务器中。此时若 Nagle 算法启用，部分信息将被延迟给到服务端，严重影响用户体验。因此特别是在一些对时延要求较高的交互式操作环境中，必须禁用 Nagle 算法，让所有的小分组必须尽快发送出去。

2.5 CrossEngine 游戏引擎

游戏引擎是打造出优秀游戏的核心因素之一，CrossEngine 是腾讯为降低商业风险提升核心能力而自研的跨平台游戏引擎。其基础架构参考了 ECS 框架，这是一种主要用于游戏引擎的软件开发架构，大体上由实体 Entity、组件 Component 和系统 System 三部分构成 [?]。其中实体是对场景内需要逻辑控制的物体的抽象；组件代表被挂载于实体上的数据，一个实体可以搭载若干组件，相当于该实体被赋予了一系列属性；系统在此架构中承担了全部的逻辑代码，可以访问实体中的组件。此架构突出了组合大于继承的理念，可以更加灵活的表示现实甚至想象出的概念。引擎中如内存分配、数学运算、资源管理等核心基本由 C++ 编写，引擎编辑器主要由 C# 开发。图 2-2 展示了目前 CrossEngine 的编辑器界面。

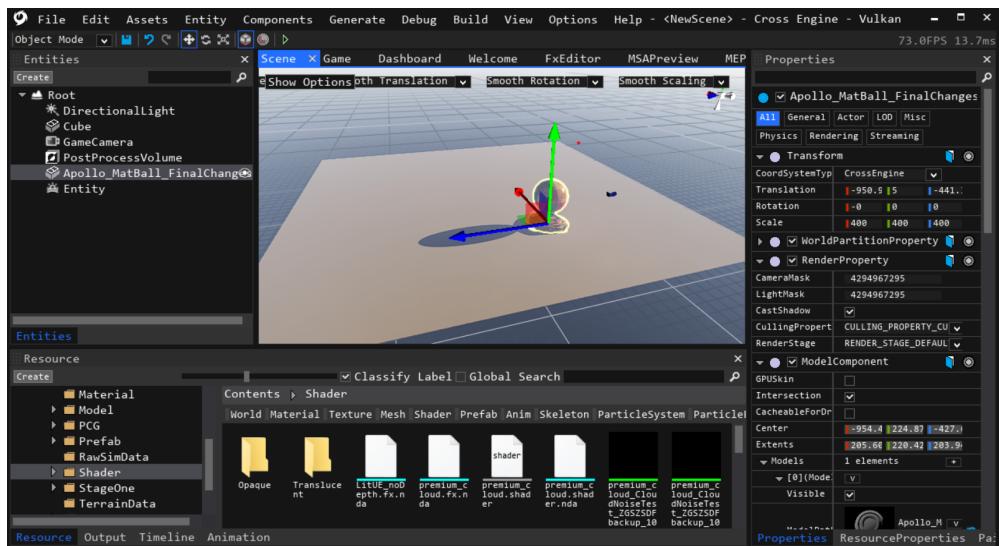


图 2-2: CrossEngine 编辑器

目前 CrossEngine 引擎已具备一定程度的生产实践能力。渲染方面目前已支持 DX12、Vulkan、GLES3 渲染后端，HDR-LinearSpace 工作流内置基于物理的渲染；功能系统方面具备基于 PhysX 的物理引擎，骨骼动画系统、粒子系统、脚本系统也基本完善。编辑器部分则是对标商业引擎主流设计，尽可能降低学习成本。CrossEngine 从最初的架构设计到三方基础库的选型都为 Runtime 尺寸做了许多工作，如表 2-2 所示在该方面对比商业引擎有一定优势。在后续开发中也将坚持控制 Runtime 在较小的尺寸，以带来更多应用可能。

表 2-2：引擎 Runtime 比较

游戏引擎	Runtime 体积
CrossEngine	2.2m
UnrealEngine4	40m+
Unity	23m

2.6 Wireshark 抓包工具

2.7 帧缓冲机制

2.8 插值算法

2.9 航空坐标系及坐标系转换

在模拟飞行中，涉及到位置和旋转的数据都是以某个坐标系为基础，常用坐标系有飞机自身坐标系、经纬高 LLA 坐标系、地心地固 ECEF 坐标系和北东天 ENU 坐标系。其中只有 LLA 坐标系是以经度纬度海拔确认位置的坐标系，其余均为笛卡尔坐标系。

- (1) 飞机自身坐标系一般以驾驶舱位置为原点 O，Z 轴指向飞机下方，X 轴指向飞机左侧，Y 轴垂直于 xOz 平面指向飞机前方构成右手坐标系，如图 2-3 所示。其作用为确定飞机上如起落架、各类灯的相对位置 [27]。
- (2) LLA 坐标系是以经度纬度海拔来确定位置的球面坐标系。对地球而言经度的定义为本初子午线为 0 经度，向东增加。纬度的定义为椭球表面的法线

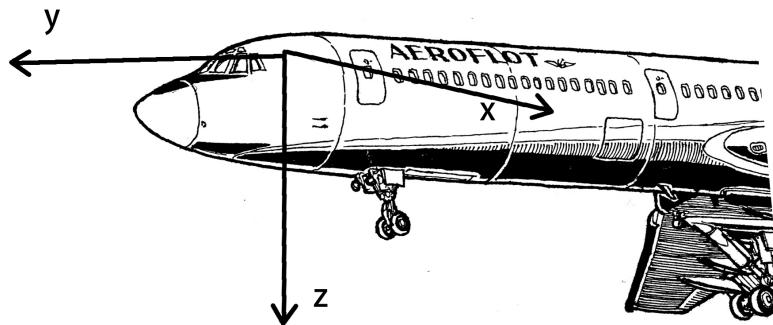


图 2-3: 飞机自身坐标

与赤道面的夹角角度。海拔则是沿椭球表面法线方向距离平均海准面的距离。地球的形状则是以 WGS-84 为准，地球为长半轴 6378137.0 米，扁率 1/298.257223563 的椭球 [28]。在 FFS 中，仿真机给出的飞机位置信息便是经纬高的形式。

- (3) 地心地固坐标系 ECEF 是一种以地心为原点的地固坐标系。原点 $O(0,0,0)$ 为地球质心，Z 轴与地轴平行指向北极点，X 轴指向本初子午线与赤道的交点，Y 轴垂直于 xOz 平面构成右手坐标系 [29]。在视景系统中需要以该坐标系作为世界坐标系，即所有物体的坐标最终都要转换到该坐标系下。
- (4) 东北天 ENU 坐标系是以物体所在球面位置为原点，X 轴指向东方，Y 轴指向北方，Z 轴垂直于 xOy 面指向天空构成的右手坐标系 [30]。在 FFS 中，每一帧下飞机的初始姿态便是在该坐标系下，且飞机的旋转是以偏航 Yaw，俯仰 Pitch，翻滚 Roll 的顺序完成。

以上是航空常用 4 种坐标系，在实际需求中必然涉及飞机的旋转以及坐标在不同坐标系下的变换。向量的旋转和坐标的变换可以通过旋转矩阵完成 [31]。最直观的旋转操作是按照固定坐标轴依次旋转，在三维空间中，按 z 轴旋转 α 角度可以表示为如下形式。旋转矩阵中的列向量为旋转后坐标系的坐标轴在旋转前的坐标系中的坐标，自然互相正交且为单位向量，所以旋转矩阵为正交矩阵。旋转的逆变换可以直接使用转置矩阵作为逆矩阵。

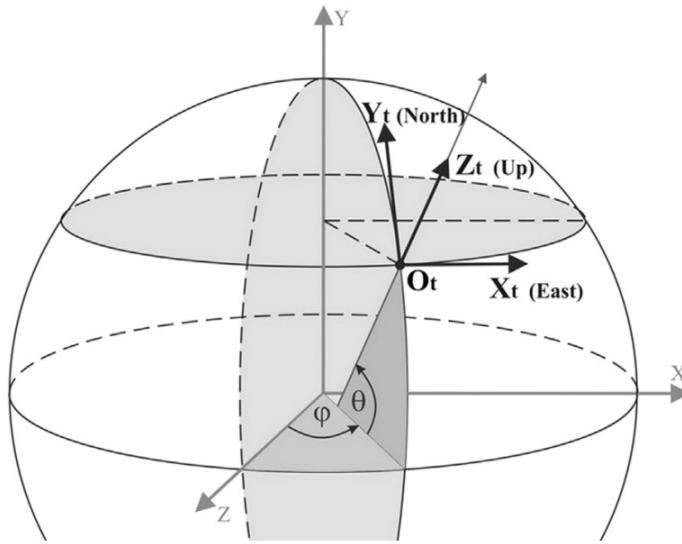


图 2-4: 三类坐标系

$$\begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

坐标系的转换也可视为旋转，特别要注意，上述公式中矩阵的意义是将如图 2-5 中 p 点在旋转后的坐标系（红色）中的坐标，转换为旋转前的坐标系（黑色）中的坐标。此时将旋转矩阵中的列向量视作旋转后坐标轴在旋转前坐标轴上的投影，或者叫做方向余弦。

旋转在游戏引擎中都是通过四元数进行。上文中提到的按照坐标系轴依次旋转的方式，旋转轴顺序的变换会影响最终结果，且旋转时不幸让某些坐标轴重合了就会发生万向节死锁，导致丢失一个方向上的旋转能力 [32]，始终得不到最终结果。四元数可以简单理解为一个任意旋转轴加旋转角度，对人类而言直观性会下降，但能够减少矩阵计算的复杂度，还可以方便的进行插值操作 [33]。

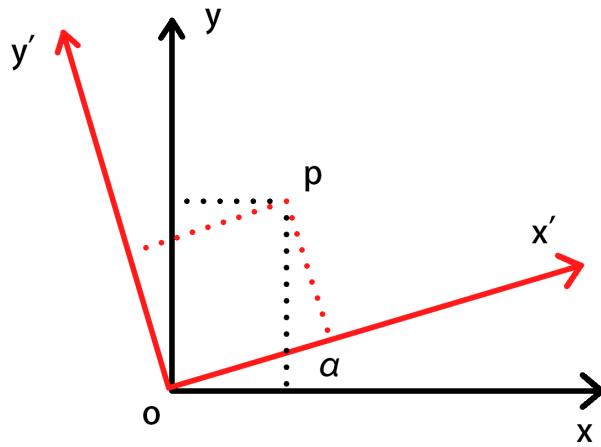


图 2-5: 坐标系变换

2.10 本章小结

本章介绍了视景系统中数据交换子系统开发中涉及的主要技术。。首先介绍了在数据交换过程中使用到的技术，包括负责绕过 OS 的网络协议栈直接与网卡驱动交流的 WinPcap，具有高序列化与反序列化效率的 ProtoBuffer 协议，有可能造成小数据包发送延迟的 Nagle 算法，以及主要服务于游戏的低延迟通信插件 Tbuspp。其次，介绍了开发本视景系统使用的自研游戏引擎 CrossEngine 和适合于编写游戏逻辑的嵌入式脚本语言 Lua。最后介绍了模拟飞行中用到的 4 种航空坐标系，以及计算机中实现旋转和坐标系转换的方式。

第三章 数据交换子系统的需求分析与概要设计

3.1 视景系统整体概述

视景系统是 FFS 中负责连续生成模拟座舱前方飞行画面的系统。飞行员在模拟座舱中的操作将作为仿真机的输入，经过模拟计算后输出如位置、姿态、灯光等一系列指令，图像生成器根据指令要求执行逻辑后便可渲染连贯的飞行画面，提供模拟飞行的视觉效果。同时图像生成器也要以指令的形式反馈给仿真机如碰撞等信息。仿真机可以据此通知如声音、运动系统做出相应反馈。

由上述可知仿真机相当于 FFS 的服务端，视景系统类似客户端，但此案例的服务端与客户端在网络体系结构中处于不同层级，需要请一个翻译才能正常进行数据交换，本系统中将该翻译角色称为虚拟仿真机。视景系统整体架构如图 3-1 所示。仿真机作为一个底层网络设备，其只能接收或发送仅用以太网协议封装的数据包，且使用该公司自定义的数据交换协议；虚拟仿真机作为翻译需要对双方的消息解封、转换再封装，最后发送。同时在没有仿真机的开发环境下也可以读取模拟数据实现流程；虚拟仿真机与客户端之间使用 ProtoBuffer 作为数据交换协议，TCP 通信用任务由 Tbuspp 插件承担。图像生成器需要根据收到的指令先完成数据同步工作，再执行逻辑，计算反馈信息。图中虚线框中的部分包含了数据由仿真机到图像生成器中被使用，生成的反馈信息再最终回到仿真机的全过程，此部分便是视景系统中的数据交换子系统。

图像生成器中还含有许多重要的功能组成部分。首先引擎核心为其提供了渲染、物理、脚本、数学计算等运行核心组件，许多逻辑依赖其中的方法实现，逻辑执行结束后也需要渲染介入。机场资产则是场景的数据库，地形、建筑、纹理等资源皆存于其中，需要根据飞机所处位置去查找并加载这些资源，这对于数据的组织形式，查找策略都有较高的要求。投影同步插件负责在 FFS 上运行时控制多台投影仪投影相同的一帧以减少画面撕裂，另外由于模拟座舱前方是一个球幕，需要校准球幕投影时发生的畸变。

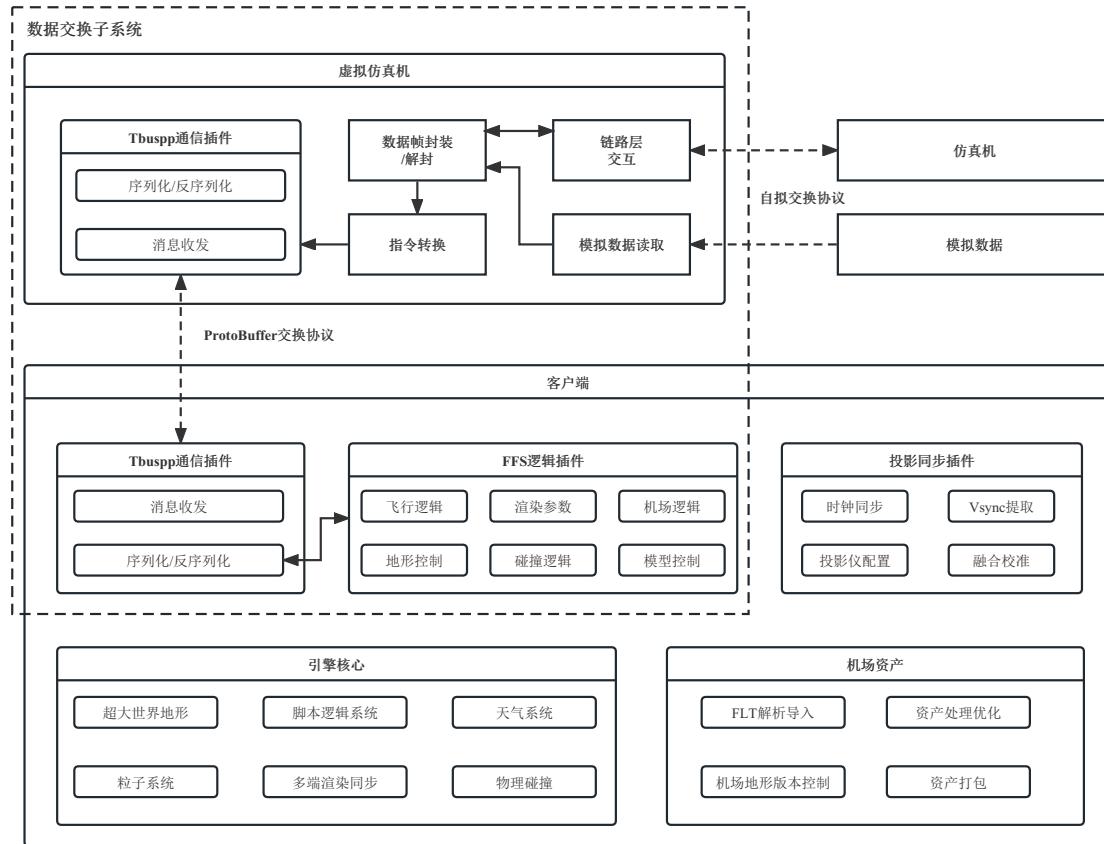


图 3-1: 视景系统整体架构

3.2 数据交换子系统需求分析

系统设计的第一步是需求分析，其主要目的是明晰用户非形式化的需求，最终产生完整的需求规格说明。本节首先分析了系统的重点涉众，阐述了涉众对系统的期望。随后从系统的角度解释软件，定义了系统的功能性需求和非功能性需求。最后在明确需求的基础上，将各需求拆解为用例，使用用例图和用例描述表详细描述各个用例。

3.2.1 涉众分析

本系统作为视景系统中负责数据交换的子系统，仅包含虚拟仿真机和图像生成器两个角色。虚拟仿真机作为桥梁希望与仿真机和图像生成器双向交流，而图像生成器希望以稳定的数据驱动逻辑运行。详细的涉众分析如表 3-1 所示。

表 3-1: 涉众分析列表

涉众名称	涉众期望
虚拟仿真机	作为仿真机与图像生成器间的桥梁，虚拟仿真机希望能正确解释并转换双发发送的指令，能按照仿真机的运行频率及时读取数据并及时发送给图像生成器。同时作为视景系统的一部分，能够屏蔽不同仿真机数据组织结构的差异。
图像生成器	图像生成器作为执行逻辑和渲染画面的角色，希望从虚拟仿真机处得到数据驱动逻辑执行。另由于多台图像生成器需要同时工作，希望能同时使用基本一致的数据进行运算，最终的融合投影才不会产生撕裂

3.2.2 功能性需求

图像生成器依据仿真机指令进行飞行画面渲染并反馈飞行数据要求仿真机与图像生成器之间进行双向数据交换。如图 3-2 所示，仿真机作为底层设备其输出与输入均为只用以太网协议封装的数据帧，想要与其沟通必须先结合相关设计文档并进行抓包确认数据帧的数据组织结构和数字表示方法。掌握规则后便要据此转换指令数据，并按照传统网络体系重新层层封装为便于图像生成器使用的数据段。因此数据交换子系统中需要一个桥梁角色与建立起仿真机与视景系统的交流，即虚拟仿真机。其同时承担着在没有仿真机的开发条件下读取模拟数据驱动视景运转的责任。在信息的传输过程中避免不了存在网络波动；且虚拟仿真机与图像生成器间用无法广播的 TCP 协议通信，对于单一图像生成器而言数据到达频率可能不稳定，对于多个图像生成器而言同一数据到达时刻也不相同。因此需要进行数据同步抵消差异。另外从单一图像生成器角度，逻辑帧的开始时间受渲染帧约束，不能保证绝对稳定，如果按照仿真机的原始数据飞行可能会产生不自然的颤动，需要一定的数据平滑策略减少影响。在本视景系统中数据的流动可用下面这一完整流程描述：

- (1) 仿真机根据飞行员的输入计算飞行状态，产生一条条指令数据，将这些指令数据仅通过以太网协议封装后以数据链路层数据帧的形式输出。
- (2) 虚拟仿真机接收数据链路层数据帧，去除以太网协议的头尾内容，完成解封。识别指令代号后，根据对应数据组织结构直接使用结构体反序列化指令内容。

- (3) 将指令中特殊数字表示方式下的数据通过算法转换为常规认知下的数字形式。如将经度 0000007f ffffffff 转换为 179.99 后，再赋值给 ProtoBuffer 结构化数据（也称作 message）中的对应字段，生成我们自定义的指令集。
- (4) 将该结构化数据通过 ProtoBuffer 协议进行序列化，并通过 TCP 协议发送给图像生成器，此过程中的协议封装则全权交由操作系统的内核网络协议栈完成。
- (5) 图像生成器接收到数据后，同样由内核网络协议栈解封，再通过 ProtoBuffer 协议进行反序列化得到指令结构化数据，经过数据同步和平滑后供给逻辑线程使用。
- (6) 逻辑线程的反馈信息则通过完全相反的过程，最终逆向发送到仿真机。

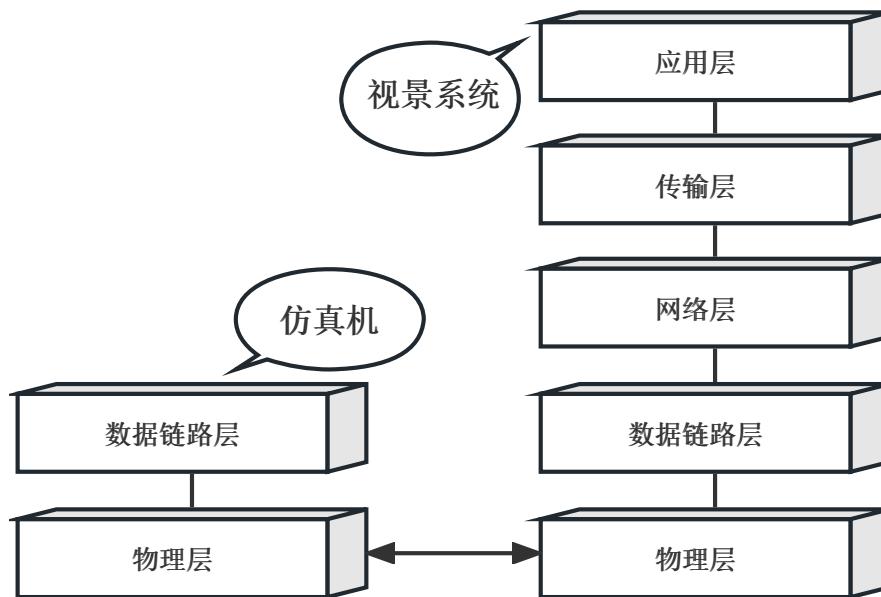


图 3-2: 仿真机与图像生成器层级差异

整个指令数据传输的过程中需要保证虚拟仿真机与图像生成器依照仿真机的工作频率运行，即数据到达便要处理并发送，这个过程中有一些缓存机制需要禁用。另外为观察整套数据传输、同步、平滑的效果，需要简单的飞机飞行逻辑观察飞行中画面的效果。数据交换子系统的功能性需求列表如表 3-2 所示。

表 3-2: 系统功能性需求列表

ID	需求名称	需求描述
R1	与仿真机交互	虚拟仿真机可以绕过所在操作系统的网络协议栈，直接读取或生成流经网卡的原始数据帧，需要亲自实现解封和封装数据帧的过程。此过程中需要按照仿真机的频率执行逻辑，确保指令数据的实时性。
R2	指令数据转换	虚拟仿真机可以将仿真机使用的指令与图像生成器中使用的指令进行映射，同时需要进行数字表示方法的转换，方便双方对于数据的使用。
R3	与图像生成器交互	虚拟仿真机可以与图像生成器进行自定义指令的交互，其中需要使用高效的数据交换协议对指令内容序列化和反序列化。
R4	数据同步与平滑	图像生成器不可以立即使用到达的数据指令，存储几帧内容与其他机器进行同步处理。图像生成器可以对存储的内容进行平滑，减少逻辑帧的不稳定带来的抖动。
R5	飞行控制	图像生成器接收到来自仿真机关于飞行的指令后，可以按照经纬度海拔数据将飞机置于正确位置，按照俯仰、翻滚、偏航欧拉角确定飞机的飞行姿态。

3.2.3 非功能性需求

帧率是动态画面视觉体验的重要因素，对于电视与电影这类视频行业来讲 24Hz 以上的帧率便能达到良好的观看体验 [34]。但对于需要飞行员实施操控的飞行模拟而言，60Hz 以上的帧率才不会让飞行员产生操作延迟的感觉，因此本视景系统初期要求在训练基地的 FFS 设备上能达到 60Hz 的帧率，且长时间运行帧率不会明显下降。目前国内的进口模拟机来自 CAE、波音等多家公司，虽然各厂商的仿真机都是数据链路层设备，但他们的指令有不同的数据组织结构和数字表示方法，数据交换子系统需要屏蔽这些差异，方便日后的二次开发以适配不同仿真机。系统的非功能性需求列表如表 3-3 所示。

表 3-3: 系统非功能性需求列表

ID	需求名称	需求描述
R1	运行帧率	飞行画面在 60Hz 以上才不会产生明显操作延迟感，因此要求初期在真实 FFS 设备上能够达到最低 60Hz 的渲染帧率，也意味着数据交换子系统能够以这个频率处理数据。且日后经游戏引擎角度的不断优化能够达到 100Hz 以上。
R2	可靠性	一节飞行训练课约为 50 分钟，要求视景系统在连续运行 50 分钟期间不出现明显的画面撕裂、抖动和帧率下降趋势。
R3	可扩展性	由于国内现存各厂商的仿真机均使用自定义数据组织结构和数字表示方法，数据交换子系统应体现仿真机侧无关性，将不同厂商的指令映射为我们自定义的指令集，方便经过二次开发后在各类仿真机上搭载。

3.2.4 数据交换子系统用例设计

经过需求分析后最终确定将数据交换子系统划分为十个用例，系统用例图如图 3-3 所示。其中的角色分为虚拟仿真机，客户端和开发人员。虚拟仿真机与仿真机建立数据链路层连接，并通过仅用以太网协议封装的数据帧交互。虚拟仿真机中的指令数据经过解封、转换、包装等一系列动作后成为一组自定义指令，序列化后通过 Tbuspp 发送给客户端；客户端接收数据后，根据反序列化后得到的指令内容执行逻辑，逻辑执行的结果最终用于加载对应资源和渲染画面等过程。飞行中产生的反馈信息则由客户端逆向发送最终以数据帧形式给到仿真机。作为开发人员在测试过程中可以切换不同的视角观察飞机和场景的状态，及时对问题做出调整。

具体而言，虚拟仿真机的用例包括建立链路层连接、收取数据帧、发送数据帧、读取模拟数据、仿真机指令转自定义指令、自定义指令转仿真机指令和收发 TCP 消息。客户端的用例包括收发 TCP 消息、飞行控制、检测地形信

息。开发人员的用例包括切换摄像机视角。总共十个用例。

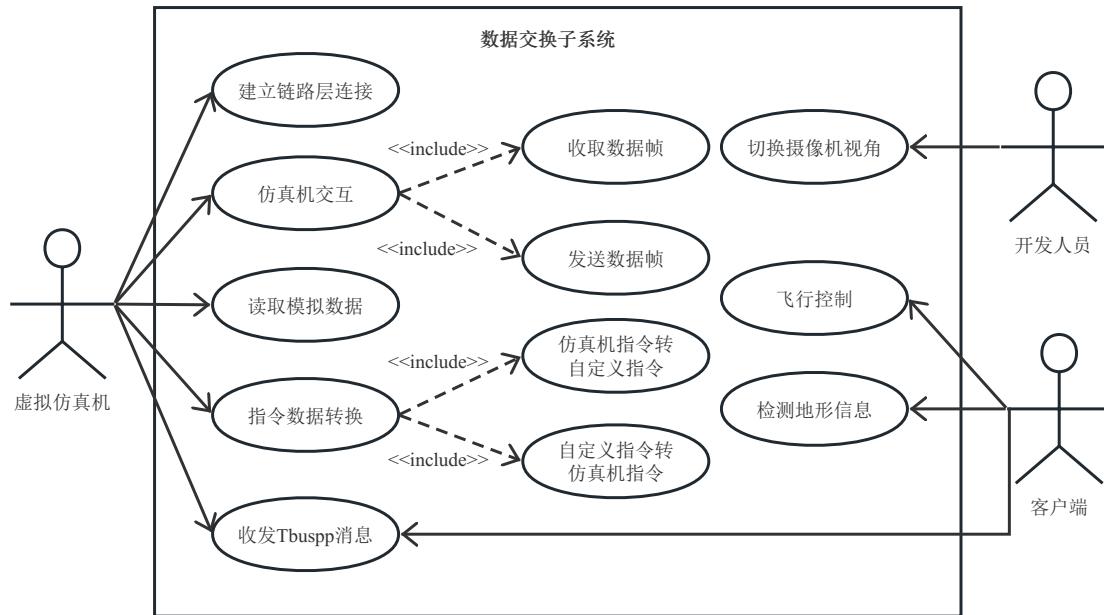


图 3-3: 数据交换子系统用例图

下面将对用例图中提到的系统用例通过用例描述表的形式进行详细解释。

建立链路层连接，是虚拟仿真机与仿真机建立沟通路径的方式。如图 3-4 中的 CAE 仿真机数据帧结构所示，由于仿真机是底层设备，其产生的数据帧不含有 IP 头、TCP 头等信息，所以不能直接交由虚拟仿真机运行环境中的网络协议栈处理，必须由虚拟仿真机亲自侦听网卡上的原始数据帧，并亲自解封或封装。用例的具体情况如表 3-4 所示。

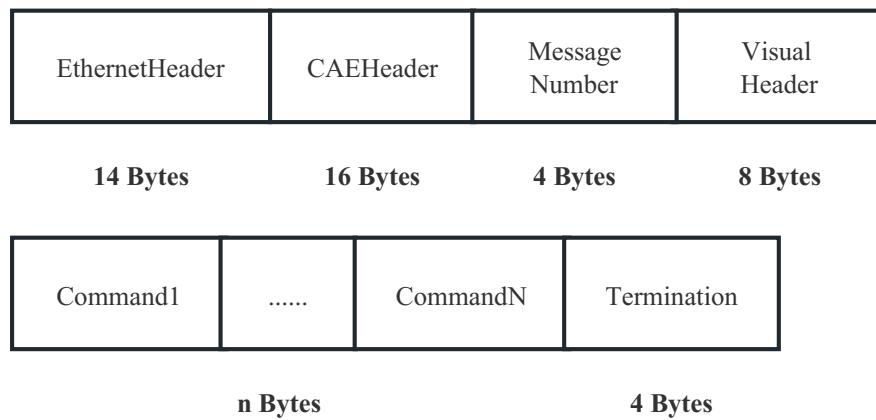


图 3-4: CAE 仿真机数据帧结构

表 3-4: 建立链路层连接用例描述表

ID	UC1
参与者	虚拟仿真机
触发条件	视景系统开始运行。
前置条件	1. 虚拟仿真机处于接收仿真机消息模式。 2. 仿真机处于开启状态。
后置条件	能够与仿真机进行数据交流。
优先级	高
正常流程	1. 查找运行环境下的网络适配器列表。 2. 选择其中一个网络适配器。 3. 获取混杂模式数据包捕获句柄。 4. 开启及时转发模式。
扩展流程	网络适配器不能正常运作，打印错误提示。

收取数据帧，是虚拟仿真机从网卡上读取仿真机发送的数据帧的过程。此过程不依靠网络协议栈自动解封，而是自己实现解封过程。收取具体情况如表 3-5 所示。

表 3-5: 收取数据帧用例描述表

ID	UC2
参与者	虚拟仿真机
触发条件	仿真机向视景系统发送数据帧。
前置条件	虚拟仿真机侦听了正确的网卡。
后置条件	数据帧被分为一个个指令段。
优先级	高
正常流程	1. 去除数据帧头部信息。 2. 读取指令长度并按该长度截取。 3. 将数据段加入指令集合。 4. 回到流程 2 循环至全部数据读取结束。
扩展流程	无

发送数据帧，是虚拟仿真机发送信息到仿真机的过程，此过程同样不依靠

网络协议栈自动封装，而是自己实现封装过程。具体情况如表 3-6 所示。

表 3-6: 发送数据帧用例描述表

ID	UC3
参与者	虚拟仿真机
触发条件	反馈指令到达。
前置条件	虚拟仿真机侦听了正确的网卡。
后置条件	仿真机收到视景系统反馈并通知其它系统协同。
优先级	高
正常流程	<ol style="list-style-type: none"> 1. 反馈指令数据段到达。 2. 将多条指令按规则粘合在一个数据包中。 3. 为数据包添加以太网头尾信息。 4. 将数据帧加入发送队列中。
扩展流程	无。

国内的 FFS 全部位于航空公司的训练基地内，其价格昂贵且巨大，无法搬运至开发环境中。在日常开发中，虚拟模拟机需要读取文件中的模拟数据来驱动视景系统运作。具体情况如表 3-7 所示。

仿真机指令转自定义指令，是虚拟仿真机将难以理解的仿真机指令转换为可供视景系统使用的子定义指令的过程。具体情况如表 3-8 所示。

自定义指令转仿真机指令，是反馈信息时虚拟仿真机将自定义指令翻译为仿真机使用的仿真机指令的过程。具体情况如表 3-9 所示。

视景系统与虚拟仿真机利用 Tbuspp 插件进行沟通，双方都需要对 Tbuspp 消息进行收发。具体情况如表 3-10 所示。

视景系统中的飞机需要按照收到的飞行状态在地心地固坐标系下的场景进行运动，主要使用飞机的位置和旋转信息，根据飞行画面可以证明数据交换系统的功能是否正常。具体情况如表 3-10 所示。

飞机在运动过程中会实时监测三个起落架竖直下方的地形信息，并将他们作为反馈信息最终给到仿真机，用于进近时与地面碰撞情况的判断。具体情况如表 3-11 所示。

摄像机的视野决定了最终的画面渲染范围，为了开发人员观察飞机运动情况和机场环境，多种视角摄像机必不可少。三种视角切换可操控摄像机可以对

表 3-7: 读取模拟数据用例描述表

ID	UC4
参与者	虚拟仿真机
触发条件	使用模拟数据驱动视景系统运作。
前置条件	虚拟仿真机处于读取模拟数据模式。
后置条件	文本信息被转化为数据帧信息。
优先级	高
正常流程	<ol style="list-style-type: none"> 1. 指定文件路径。 2. 按行读取文本。 3. 将 16 进制字符两两一组转换为一个字节。 4. 按用例二中的流程进行。
扩展流程	文件不存在或格式有错误则产生警告。

表 3-8: 仿真机指令转自定义指令用例描述表

ID	UC5
参与者	虚拟仿真机
触发条件	存在待转换的仿真机指令。
前置条件	定义了对应仿真机指令的自定义指令。
后置条件	构建了一个自定义指令。
优先级	高
正常流程	<ol style="list-style-type: none"> 1. 获取仿真机指令中的指令代号。 2. 根据指令代号查找对应的转换器。 3. 使用转换器完成转换。
扩展流程	对于暂时使用不到的仿真机指令直接丢弃。

飞机以及其所处场景进行全方位的细致观察。具体情况如表 3-12 所示。

表 3-9: 自定义指令转仿真机指令用例描述表

ID	UC6
参与者	虚拟仿真机
触发条件	存在待转换的自定义指令。
前置条件	无
后置条件	构建了一个仿真机指令。
优先级	高
正常流程	1. 获取自定义指令的指令代号。 2. 根据指令代号查找对应的转换器。 3. 使用转换器完成转换。
扩展流程	无

表 3-10: 收发 Tbuspp 消息用例描述表

ID	UC7
参与者	虚拟仿真机 & 客户端
触发条件	有自定义指令待发送。
前置条件	双方使用 Tbuspp 成功建立连接。
后置条件	无
优先级	高
正常流程	数据发送过程: 1. 用 ProtoBuffer 协议序列化自定义指令结构。 2. 将消息写入 Tbuspp 发送队列。 数据接收过程: 1. 从 Tbuspp 接收队列读取数据。 2. 使用 ProtoBuffer 协议反序列化消息得到自定义指令结构。
扩展流程	无

表 3-11: 飞行控制用例描述表

ID	UC8
参与者	客户端
触发条件	客户端收到飞机飞行相关指令数据。
前置条件	存在飞机实体及对应位置场景资源。
后置条件	连续更新飞机位置和姿态获得动态视景效果。
优先级	高
正常流程	<ol style="list-style-type: none"> 1. 获取经纬度海拔和欧拉角信息。 2. 将 LLA 坐标转换为 ECEF 坐标。 3. 将飞机模型置于场景中该坐标位置。 4. 根据欧拉角调整飞机姿态。
扩展流程	无

表 3-12: 检测地形信息用例描述表

ID	UC9
参与者	客户端
触发条件	飞机处于飞行状态。
前置条件	场景数据库中存在飞机所处位置的地形。
后置条件	无
优先级	中
正常流程	<ol style="list-style-type: none"> 1. 飞机位置更新。 2. 发射射线获取飞机竖直下发的地形高度和飞机离地高度。 3. 将信息组合成自定义指令。
扩展流程	无

3.3 数据交换子系统概要设计

数据交换子系统的架构设计如图 3-5 所示，整个系统分为三个部分，仿真机与虚拟仿真机间通过数据帧进行交互，虚拟仿真机与游戏引擎通过 TCP 协议

表 3-13: 切换摄像机视角用例描述表

ID	UC10
参与者	开发人员
触发条件	开发人员按键切换摄像机视角
前置条件	无
后置条件	相机可以循环切换。
优先级	中
正常流程	1. 初始为飞机第一视角摄像机。 2. 按键切换至第三人称环绕视角，将以飞机为中心环绕观察。 3. 再次切换为自由摄像机，可使用输入设备在场景中漫游观察。
扩展流程	无

交互。仿真机的作用是对飞行员的操作输入进行仿真计算，得出飞机状态，这部分由 FFS 厂商设计开发。虚拟仿真机则将不同的仿真机数据协议翻译为统一的供视景系统使用的指令结构体，以此屏蔽仿真机的差异。游戏引擎的只需按照收到的指令数据渲染飞行画面，并提供简单的反馈信息。

在数据交换子系统中虚拟仿真机的通信服务完全使用 C++ 语言开发，游戏引擎中核心功能如通用坐标系转换、射线探测等由 C++ 语言开发。模拟飞行视景系统的专属逻辑如飞行控制则使用 Lua 脚本。在开发过程中飞行逻辑需要不断调整优化，且日后仍有如灯光、起落架等大量逻辑加入，Lua 作为轻量小巧的解释性语言，非常适合作为游戏开发中不断嵌入的脚本语言。

仿真机侧数据交换使用到 WinPcap 作为与仿真机数据交换的工具，WinPcap 不依靠主机的诸如 TCP/IP 协议去收发数据包，为应用程序提供访问网络底层的能力。数据链路层的原始数据帧收发都需要通过 WinPcap 实现。由于不同厂商的数据帧结构不尽相同，因此数据帧解析和生成会对应不同的实现。

协议转换模块主要负责指令结构体与 ProtoBuffer 协议结构体之间的转换。谷歌公司的 ProtoBuffer 数据序列化协议有非常高的生成和解析速度，作为实时渲染的数据交换协议十分适合。

引擎侧数据交换模块则主要负责虚拟仿真机与游戏引擎间的通信。通信内容是由 ProtoBuffer 封装好的指令数据，通过腾讯通信组件 Tbuspp 实现通讯。

给到游戏引擎的数据发送频率应当与渲染帧率相一致，对该线程的运行做出限制。

飞行控制模块负责实现飞机的基础飞行功能用于检测数据交换的正确性，其中主要涉及到地理坐标系的转换算法。同时在基础可操控组件模板下编写了实现了三种可操控摄像机负责观察飞机和场景。

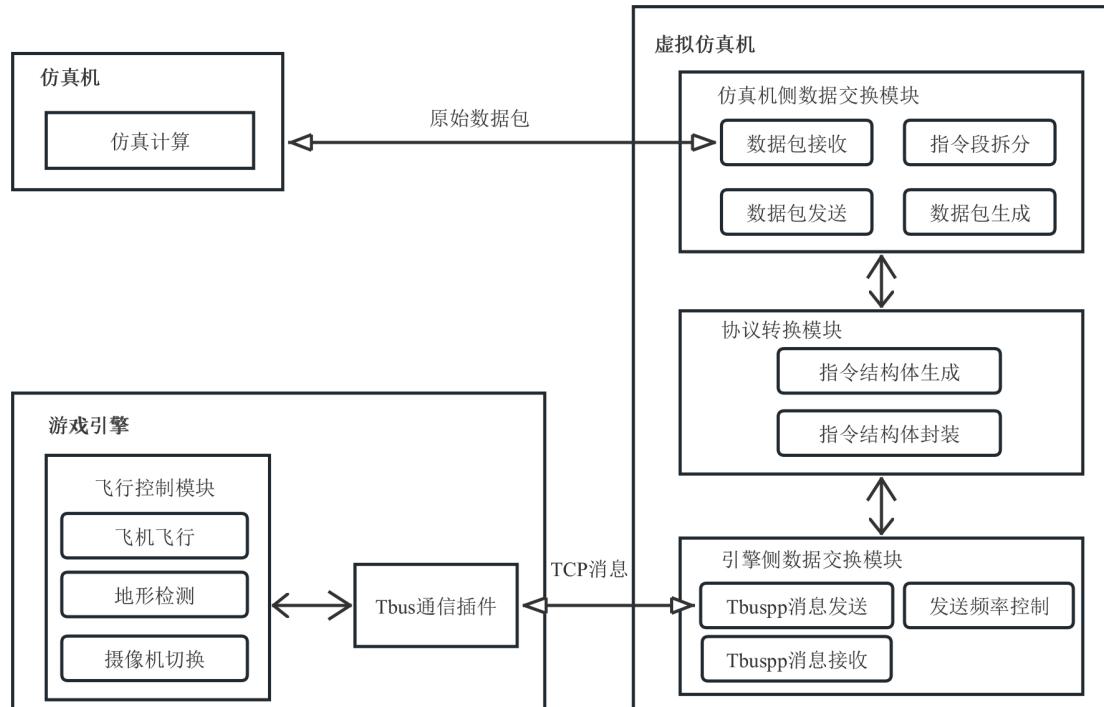


图 3-5: 数据交换子系统架构图

3.3.1 系统逻辑视图

逻辑视图以用户为中心，展示了系统的架构和功能模块。系统通过逻辑层次划分，把各个功能抽象成类，实现功能的封装。如图 3-6 描述了系统的逻辑视图。SimNodeCommunicate 代表仿真机与虚拟仿真机间沟通的模块，包含 EstablishConnection 功能，负责建立维持底层网络的连接，在此基础上实现原始数据包的收发；SendDataFrame 和 ReceviceDataFrame 负责维护各自的消息队列，并真正执行原始数据包的发送和接收任务，前者将数据包交给数据转换模块，后者则将收到的数据反馈给仿真机。DataConverter 表示数据转换模块。来自仿真机的消息会经历原始数据包到中间指令结构体再到 ProtoBuffer 协议数据段的变换过程，反馈消息则与之相反。DataAnalysis

负责原始数据包重新解释为指令结构体或将结构体添加头部信息封装为原始数据包，DataEncapsulate 负责指令结构体与 ProtoBuffer 结构间的转换。GameEngineCommunication 表示虚拟仿真机与游戏引擎间沟通的模块，使用到了 Tbuspp 这一插件，EstablishConnection 同样负责建立和维持 Tbus 节点的连接，在此基础上收发 TCP 数据段。SendDataSegement 和 ReceviceDataSegement 负责维护各自的消息队列，并真正执行数据段的发送和接收任务，前者将数据段交给游戏逻辑使用，后者将反馈信息给到数据转换模块逆向转换。GameLogic 模块负责飞行模拟的运行逻辑，其中 FlightControl 模块负责控制飞机位置和旋转姿态，也是最基本的飞行模拟要求。Camera 模块负责确定飞行画面渲染的范围，第一人称视角只会跟随飞机运动，环绕视角可以在飞机为中心的球面上移动，自由视角则可以在场景中漫游。DetectTerrain 负责检测飞机低空飞行时其下方的地形信息，此信息会被反馈给仿真机处理。

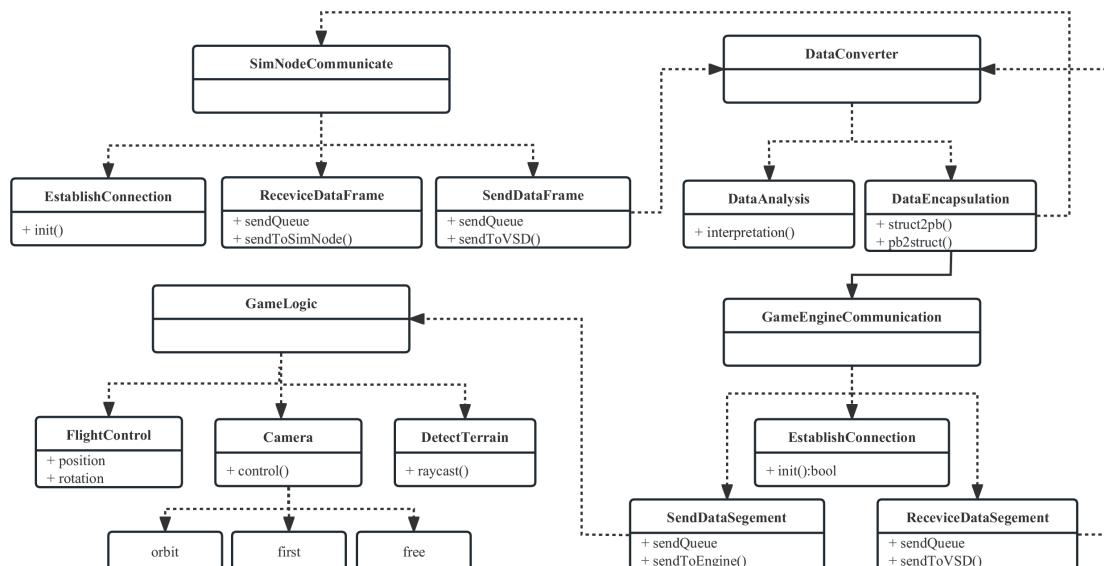


图 3-6: 逻辑视图

3.3.2 系统开发视图

3.3.3 系统进程视图

进程视图是用系统工程师的视角对系统进行阐述。可以详细阐述系统的动态运行过程，重点描述系统运行时的行为。图 3-8 是本系统的进程视图，在虚拟仿真机中，主线程启动后会开启一个发送线程和一个接收线程，负责接收来

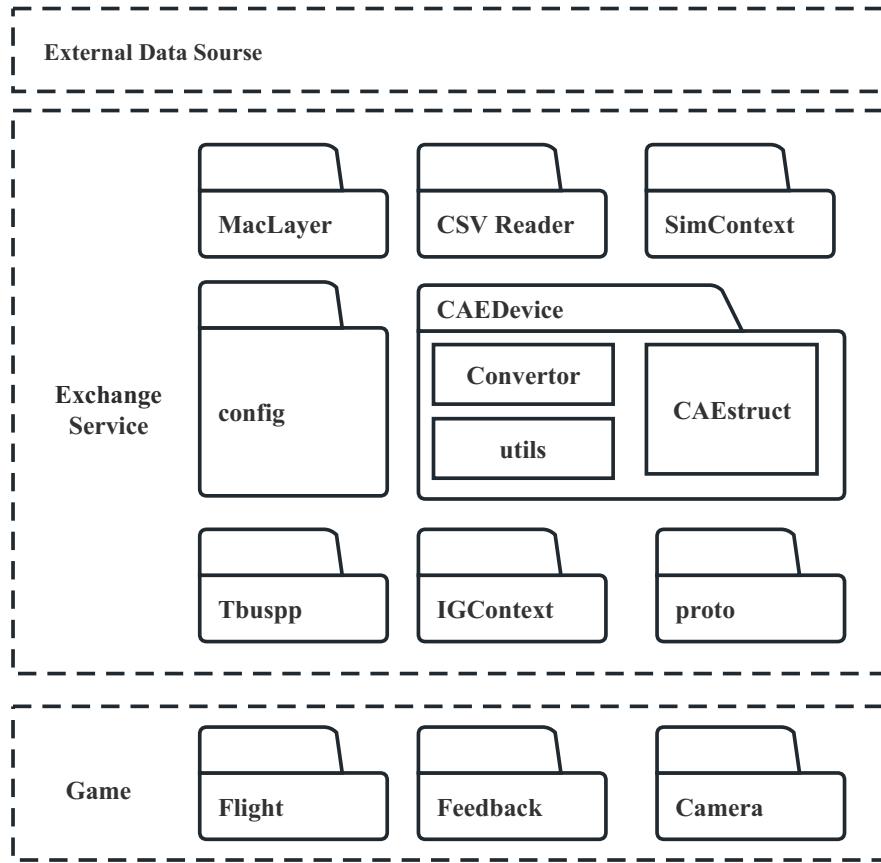


图 3-7: 开发视图

自仿真机的数据处理后发送给游戏引擎，或者接收来自游戏引擎的反馈给到仿真机。在游戏引擎中，逻辑线程负责接收到数据后完成对场景的布置，同时与发送线程同步，避免积累太多数据在内存中。逻辑线程同时需要计算碰撞情况并给出反馈信息。渲染线程专门负责对逻辑线程中生成的场景进行渲染，一般来讲逻辑线程比渲染线程快一帧。

3.3.4 系统部署视图

部署视图是用运维人员的视角对系统进行阐释。它着重于解释说明系统的整体物理结构，包括各组件之间的连接方式等等，又称为物理视图。图 3-9 是本系统的部署视图。飞行员在机舱中操作被仿真机记录并进行仿真计算，数据的解析和转换在虚拟仿真机内完成，游戏引擎收到 TCP 消息后完成场景的生成和渲染。

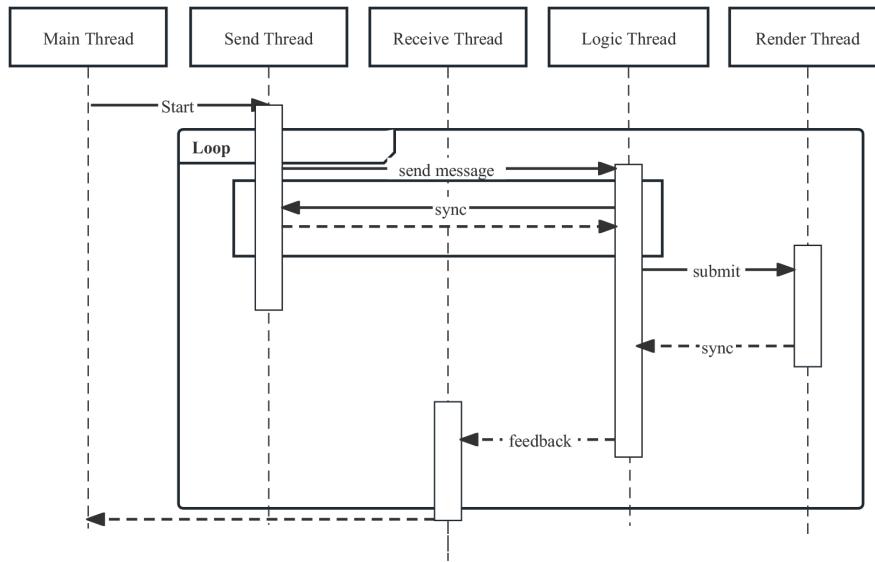


图 3-8: 进程视图

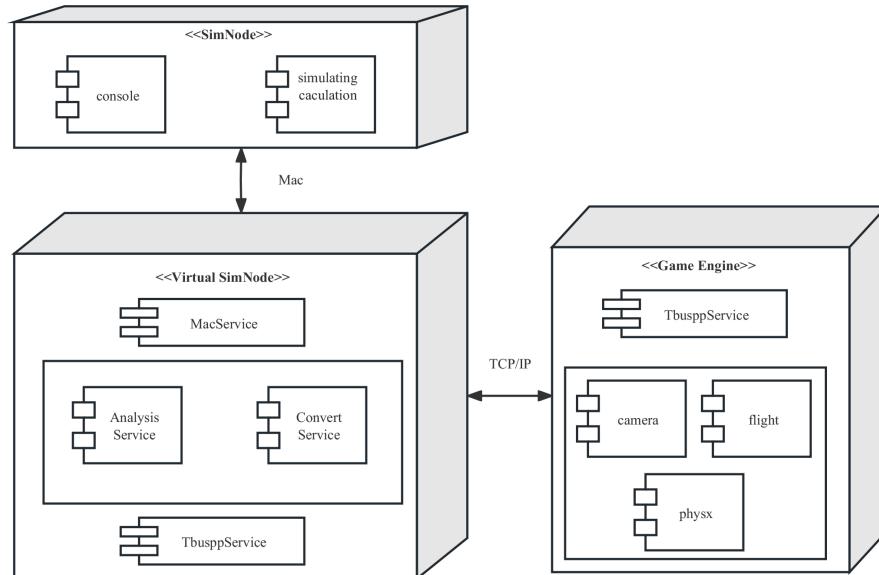


图 3-9: 部署视图

3.4 数据交换子系统详细设计

3.4.1 仿真机侧数据交互模块

仿真机侧数据交换模块负责虚拟仿真机与仿真机之间的交流。他们之间通过数据链路层的原始数据包进行沟通，即除去真实数据外，仅含有 Mac 地址等

信息。因此不能依靠 TCP/IP 协议收发而是直接访问底层网络。该模块会将到来的数据帧拆为指令结构体供后续使用，或者将指令结构体包装为数据帧发送给仿真机。

本模块的主要执行流程如图 3-10 所示。流程分为接收消息和发送反馈消息两部分。在接收到消息后，需要先读取数据帧中的指令代号信息，如果代号已知，则调取对应的转换对象，将其中信息重新解释为对应的指令结构体。由于指令代号有接近百种，项目初期只用到部分指令，对于未知的指令进行抛弃。发送反馈消息过程中，则需要对指令结构体添加数据帧头部信息，并通过底层网络直接发送。

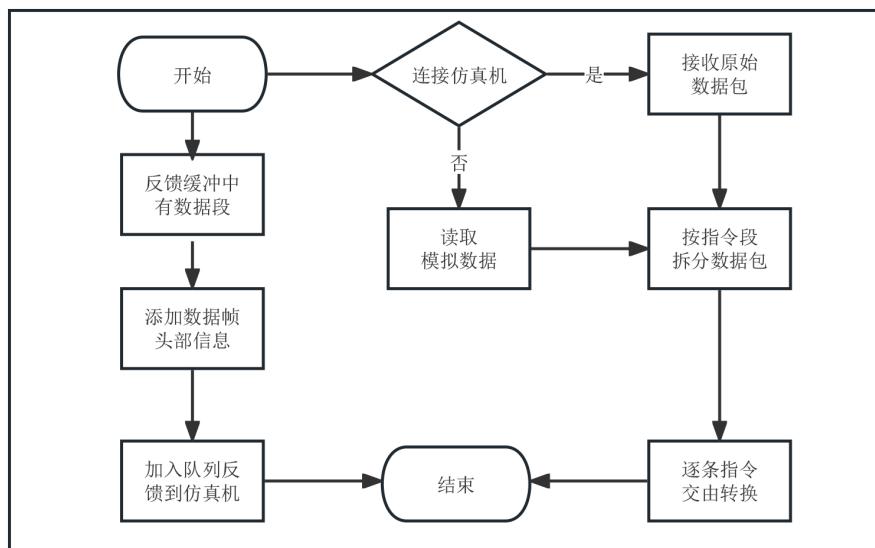


图 3-10: 仿真机侧数据交互流程图

本模块的核心类图如图 3-11 所示。其中的核心是类 `SimulationDeviceContext`，它表示在虚拟仿真机中与仿真机交流的环境，其中包含控制信息收发的线程和待发送信息队列。`ISimulatorDeviceInterface` 则是与仿真机交流的接口，其中包含了对于数据帧中指令代号的读取方法 `GetSimOperateCode`，以及发送消息给仿真机 `SendToSimulationDevice` 方法。`FullFightSimulatorDeviceBase` 中包含对该接口的实现。`CAEGenericSimulatorDeviceBase` 代表 CAE 公司 FFS 设备，其中含有解读并转换该设备原始数据包的具体算法。当需要接入其他公司设备时，只需要重新设计对应方法即可。`MacLinkCommunication` 类负责建立数据链路层连接并与真实仿真机进行原始数据包的交流。

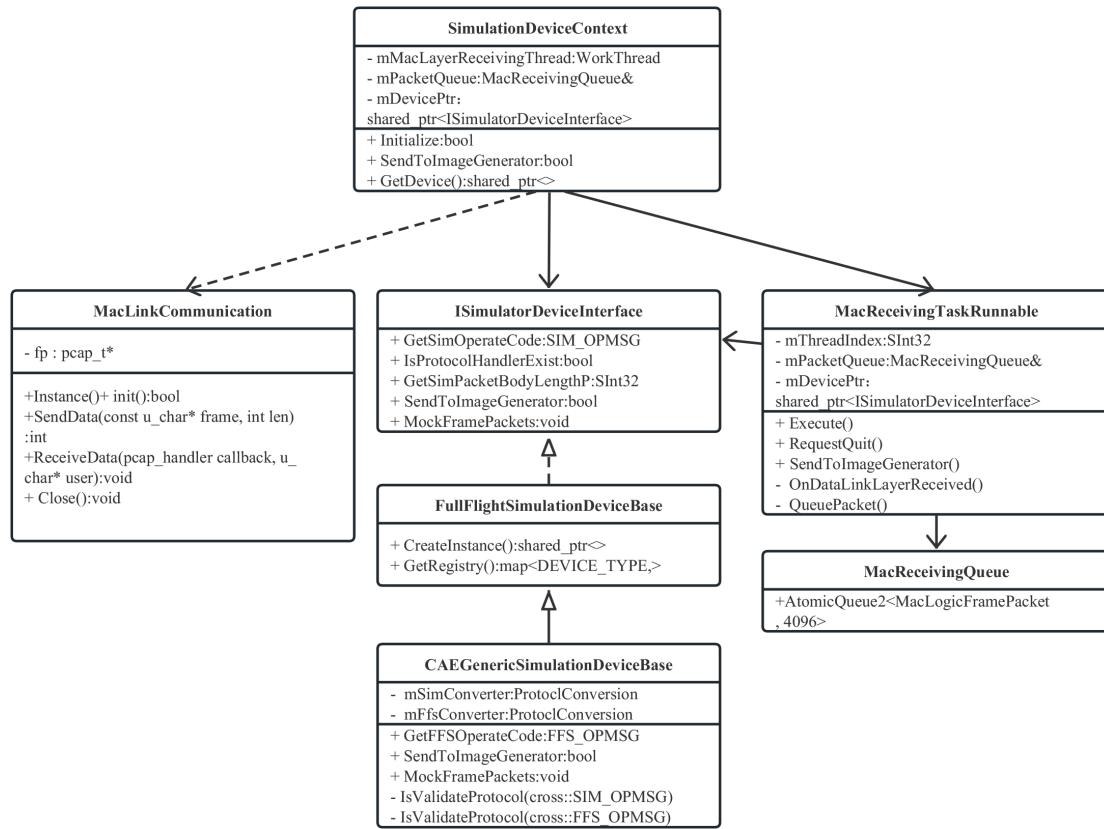


图 3-11: 仿真机侧数据交互核心类图

3.4.2 数据转换模块

协议转换模块负责指令结构体与 ProtoBuffer 协议结构间的封装与解封。从仿真机收到的消息中对于浮点数的表达方式是由该厂商自行设计的，对于不同精度要求的数据表示方式也存在差异。因此这两种结构之间的转换需要严格依照设计文档中的说明设计转换算法，而不是简单的赋值。ProtoBuffer 协议结构为给到游戏引擎的数据结构，指令结构体则是准备反馈给仿真机的结构。

本模块的主要执行流程如图 3-12 所示。在封装的过程中，需要先确定该指令对应的 ProtoBuffer 结构，经过数据转换算法后得到该结构，之后添加部分头部信息便可将其加入发送队列。解封时根据头部信息确定指令类型，再使用逆向的数据转换算法生成指令结构体即可准备发送给仿真机。

本模块的核心类图如图 3-13 所示。其中核心为 IPacketConversion 接口，其中声明了指令结构体与 ProtoBuffer 协议结构体互相转换的两种方法 Convert。ProtocolConversion 是一个模板类，成员属性 T 表示一个指令结构体，成

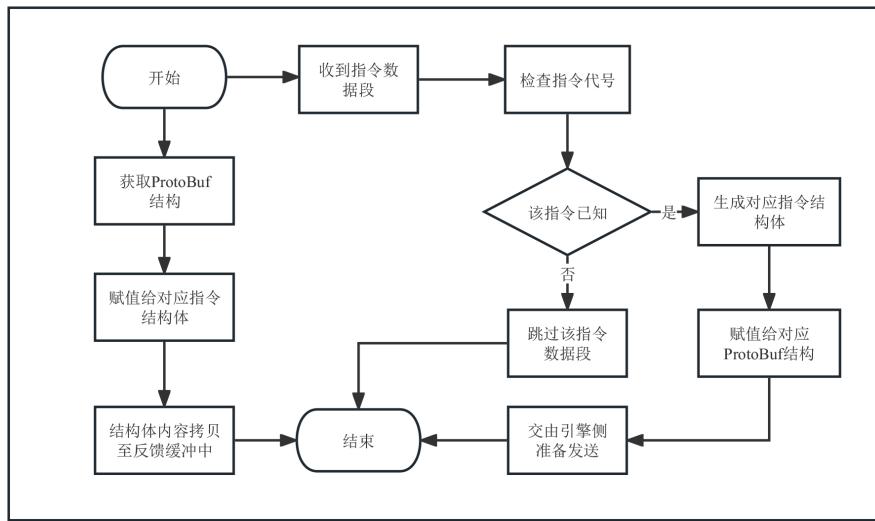


图 3-12: 数据转换流程图

员属性 F 表示一个 `ProtoBuffer` 协议结构体，T 和 F 组成互相转换的一对。该模板实例化的对象调用 `Convert` 方法便可以调用到对应结构体的转换方法。`SD_PACKET_21H` 是一个指令结构体的实例类型，包含了经纬度欧拉角等飞机飞行信息，其中也含有该指令类型转换的具体方法，发送给仿真机的队列 `MacLogicFramePacket` 由添加了数据帧头的指令结构体构成。`GeodeticCSUpdateStruct` 为 `SD_PACKET_21H` 指令结构体对应的 `ProtoBuffer` 协议结构体，发送给游戏引擎的消息队列 `FfsLogicFramePacket` 由各种 `ProtoBuffer` 协议结构体组成。

3.4.3 图像生成器侧数据交互模块

游戏引擎侧数据交换模块负责虚拟仿真机与游戏引擎之间的交流。其利用 `Tbuspp` 插件使用 `TCP` 消息沟通，数据交换协议则是 `ProtoBuffer`。再发送给游戏引擎的过程中，需要将发送频率的固定，因此该线程会以固定的频率被唤醒。

本模块的主要执行流程如图 3-14 所示。当发送队列中有消息时，若可以进行发送则直接发送，否则要等待新的发送时机。反馈消息接收时则不需考虑频率，直接接收消息并给到协议转换模块。

本模块的核心类图如图 3-15 所示。其中的核心是类 `ImageGeneratorContext`，它表示在虚拟仿真机中与游戏引擎交流的环境，其中包含控制信息收发的线程和信息队列。`IImageGeneratorDeviceInterface` 则是与游戏引擎交流的接口，其中包含了对于 `ProtoBuffer` 结构中指令代号的读取方法 `GetOperateCode`，

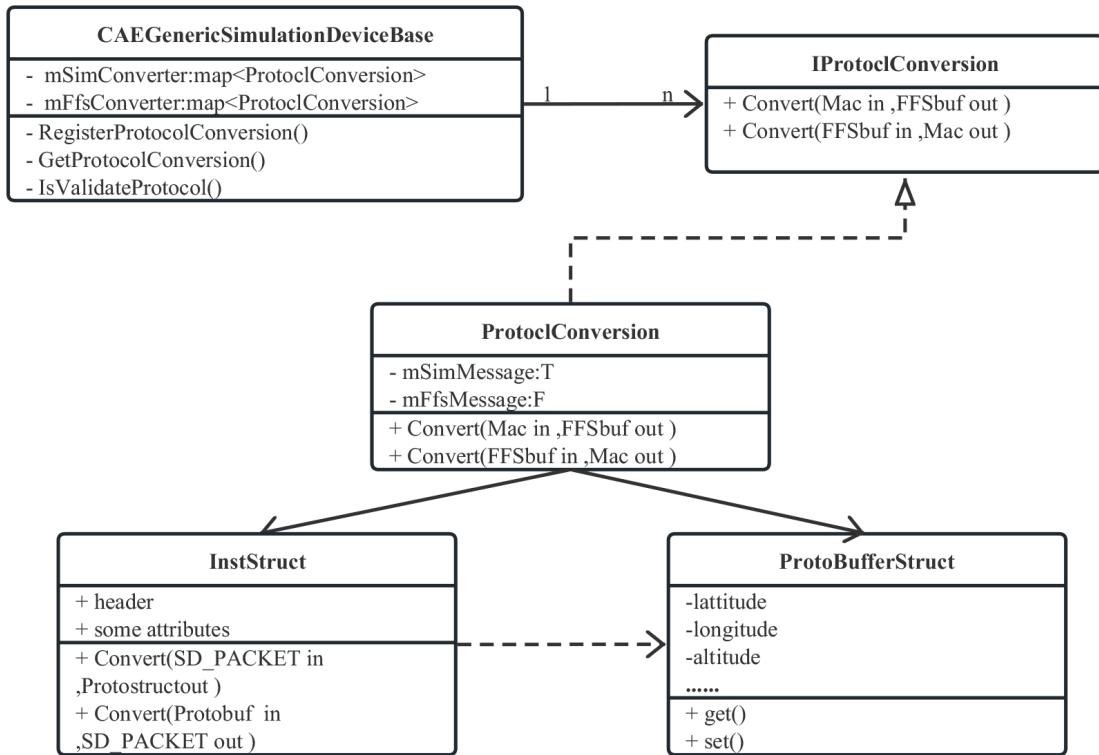


图 3-13: 数据转换核心类图

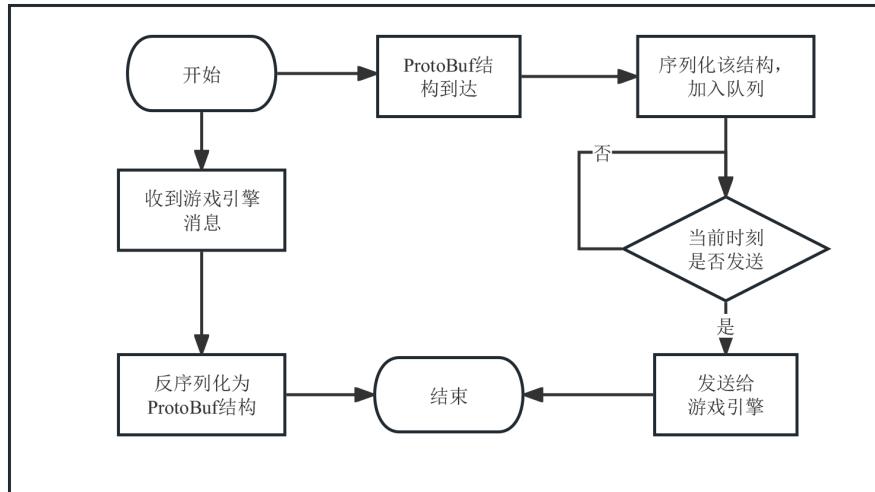


图 3-14: 视景系统侧数据交互流程图

以及发送消息给游戏引擎 `SendToImageGenerator` 方法。`FullFightSimulatorDeviceBase` 是对该接口的实现，其依赖控制发送频率的类 `FrameSynchronization`。`CAEGenericSimulatorDeviceBase` 代表 CAE 公司 FFS 设备，其中含有与该公司

设备进行沟通的具体方法。

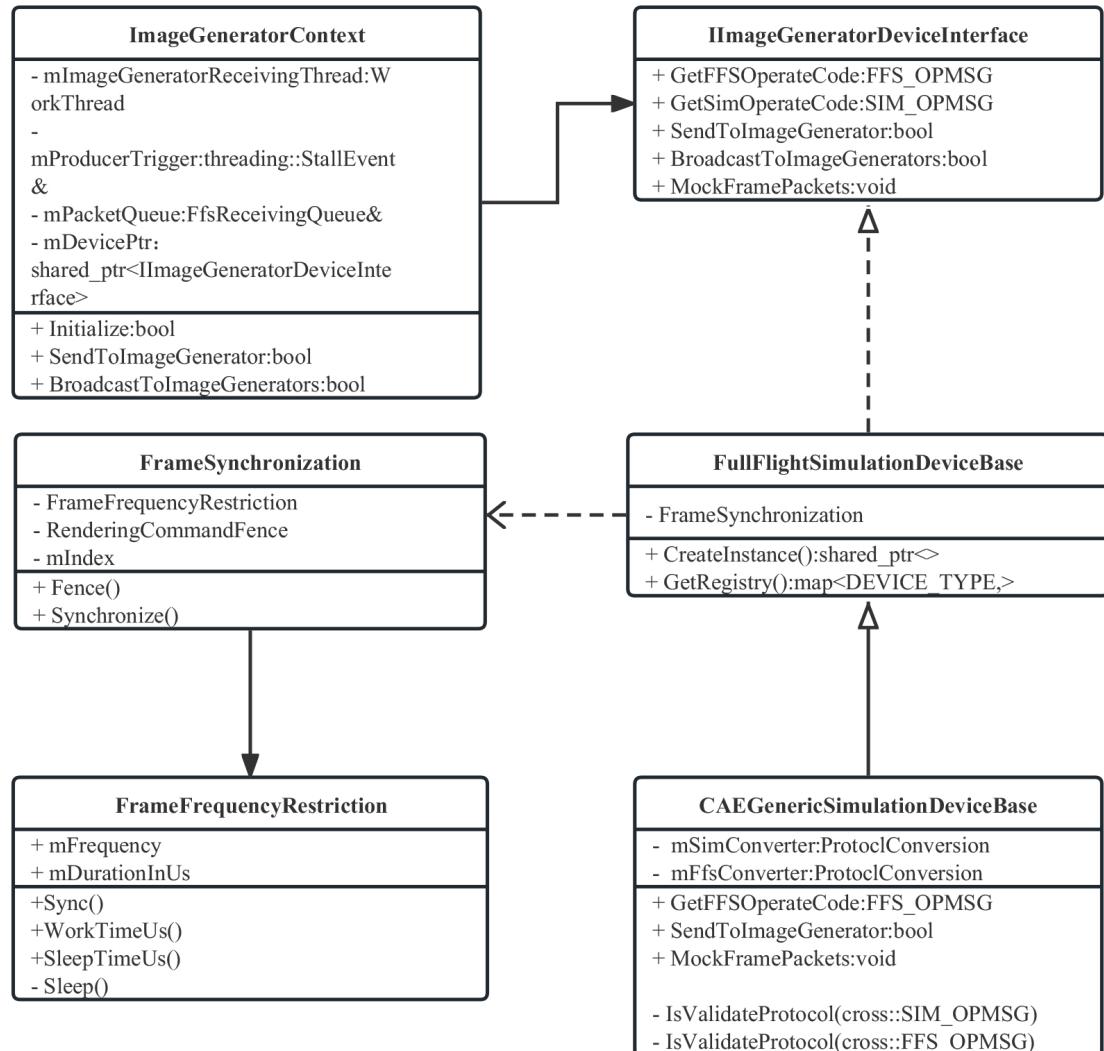


图 3-15: 视景系统侧数据交互核心类图

3.4.4 数据同步模块

3.4.5 飞行数据处理模块

基础飞行控制模块负责让视景系统中的飞机运动起来并能被开发人员细致观察。实际过程中，游戏引擎会根据接收的数据实时改变飞机位置和飞行姿态，并加载周边场景信息。飞机会随时对当前竖直下方的地形信息进行探测，并作为反馈信息。飞行过程中开发人员可以进行三种视角的观察。

本模块的主要执行流程如图 3-16 所示。对于不用于飞机位置的指令直接跳过。对于位置和姿态信息则需要一定的转换，因为其在指令中的定义与场景是不同的坐标系。数据经过转换后便可以直接给到飞机实体和摄像机，改变其在场景中的位置，生成连贯的飞行画面。

本模块的核心类图如图 3-17 所示。其中用于飞机位置控制的类为 TransformSystemG，这是位于游戏引擎核心中的一个类，负责控制场景中实体的位置和旋转。其中列出了与球面世界相关的函数，主要为 WGS84，ECFC 和东北天三种坐标系的相互转换方法。ControllableUnit 类可以给场景中的实体赋予被用户输入操作的能力，ControllableCamera 继承自该类，其中含有 RotationScale 和 TranslateScale 属性调整移动和旋转的速度。TestEnvControllableCamera 则是为此视景系统专门定制的可控摄像机类，它可以在被控制的基础上实现三种控制模式间切换的功能。视景系统中的世界 World 需要依赖以上两个组件，实现飞机的飞行控制和摄像机的观察。

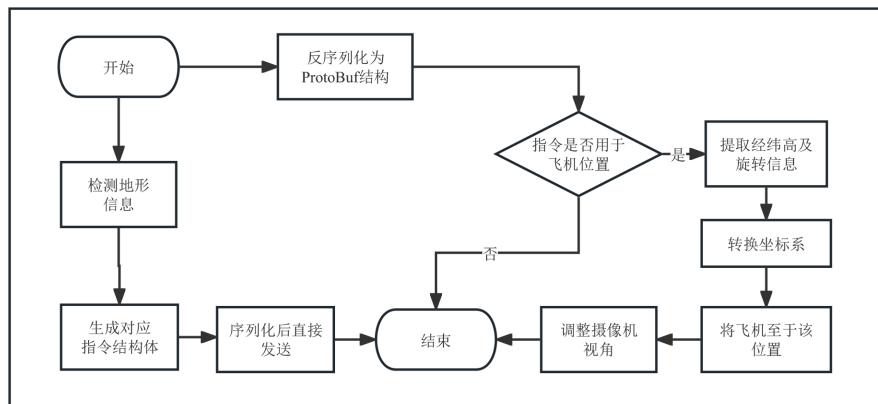


图 3-16: 飞行数据处理流程图

3.5 本章小结

按照软件开发的一般步骤，本章详细介绍了视景系统中数据交换子系统的需求分析与设计过程。首先，对于整个视景系统的工作流程给出了整体说明。其后，对数据交换子系统进行了需求分析，明确功能性需求和非功能性需求，并以用例图和用例描述表的形式给出规格说明。其次，本章给出了数据交换子系统的架构设计。最后，本章将子系统的主要功能划分为仿真机侧数据交换模

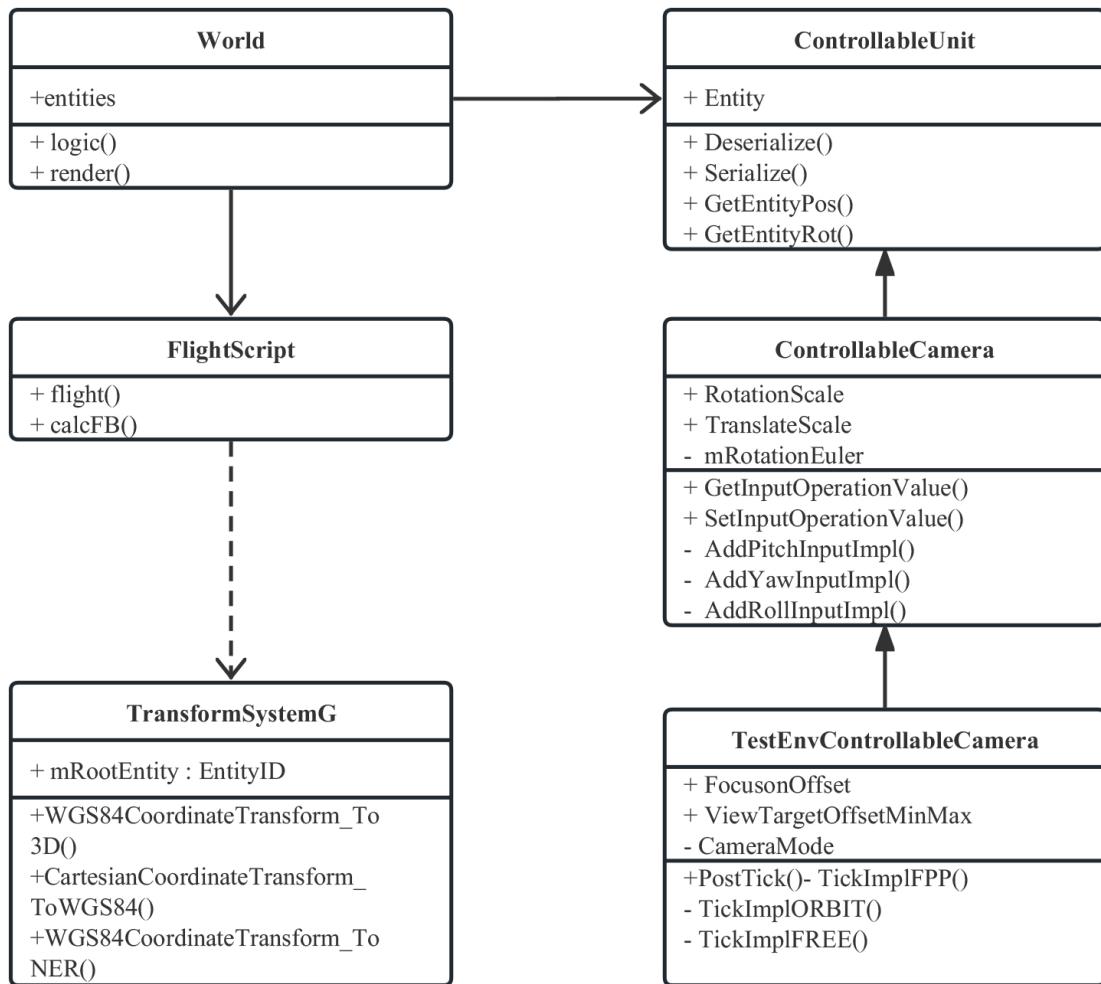


图 3-17: 飞行数据处理核心类图

块、协议转换模块、游戏引擎侧数据交换模块和基础飞行控制模块，以流程图和核心类图描述了对各个模块的详细设计。

第四章 数据交换子系统的详细设计与实现

4.1 仿真机侧数据交换模块

4.1.1 顺序图

图 4-1 是仿真机侧数据交换模块的顺序图，描述了虚拟仿真机与仿真机沟通过程中各类的交互过程。首先交流环境类 SimDeviceContext 以 mac 地址作为参数尝试初始化即与仿真机建立连接，MacLink 类中利用 WinpPcap 实现连接建立，并判断是否建立成功。成功建立连接后，便开始数据的收发。交流环境通过 send 方法最终交由代表 CAE 公司设备的类执行发送，CAEDevice 则需要调用其专属转换类中的 convert 方法实现原始数据包与指令结构体之间的转换。

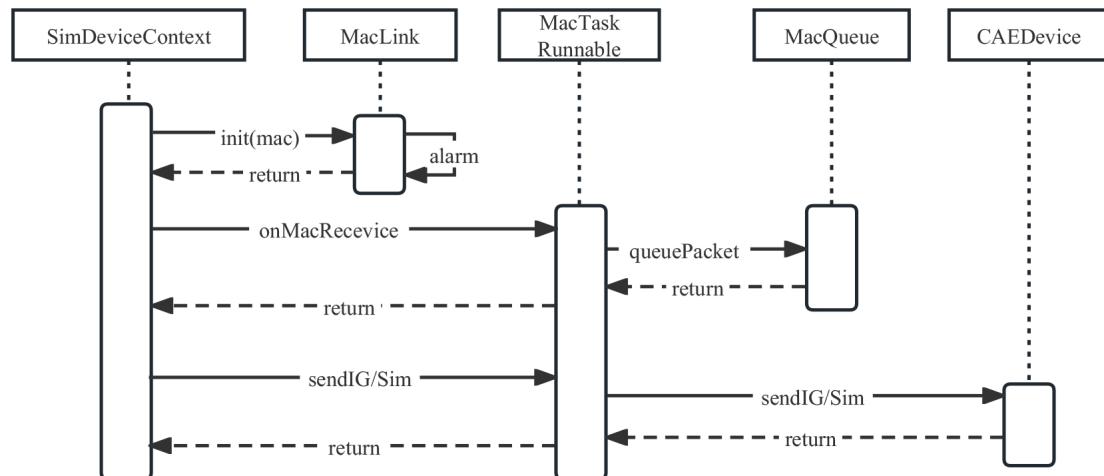


图 4-1: 仿真机侧数据交换顺序图

4.1.2 仿真机建立连接

虚拟仿真机与仿真机间通过数据链路层数据帧直接交换信息，首先要建立连接。类 `SimulationDeviceContext` 表示在虚拟仿真机中与仿真机交流的环境，`Initialize` 函数表示了该环境的初始化流程。代码中用到了条件编译，即根据编译时的参数来编译不同部分的代码。条件 `MOCKSIM` 表示是否使用模拟数据。如果编译时带有该参数，那么需要去读取配置文件中的模拟数据文件路径；否则进一步与真实仿真机建立真实链路层连接。两种方式都需要启动新的线程负责消息的收发工作。

```
bool SimulationDeviceContext::Initialize(const DeserializeNode& inRootNode)
{
    MacReceivingTaskRunnable* producerRunnable = nullptr;
    #ifdef MOCKSIM
    {
        LOG_INFO("Initialize Mock SIM!!!");
        auto mockDocPath = inRootNode["MockSimDocument"].AsString();
        producerRunnable = new MockMacReceivingTaskRunnable
            (0, mDevicePtr, mPacketQueue, mockDocPath);
    }
    #else
    {
        LOG_INFO("Initialize SIM!!!");
        MacLinkCommunication::Instance().Init();
        producerRunnable = new MacReceivingTaskRunnable(0, mDevicePtr, mPacketQueue);
    }
    #endif
    mMacLayerReceivingThread = WorkerThread{producerRunnable,
        threading::RunnableThread::CreateRunnableThread(producerRunnable)};
    return true;
}
```

图 4-2: 仿真机环境初始化代码

真实链路层连接的建立则是使用到 `WinPcap`。在 `Init` 方法中，首先通过 `pcap.findalldevs` 方法获得该机器中所有网卡的信息，用户需要根据打印的信息选择需要使用的网卡。之后使用 `pcap_open_live` 获得该网卡数据包捕获描述字，成功后即表明完成初始化 link layer 句柄，虚拟仿真机与仿真机可以通过用户选择的网卡进行交流。最后通过 `pcap_freealldevs` 释放网卡数据列表。

```

bool MacLinkCommunication::Init() {
    pcap_if_t* alldevs;
    pcap_if_t* d;
    /* the device list */
    if (pcap.findalldevs(&alldevs, errbuf) == -1)
    {
        fprintf(stderr, "Error in pcap.findalldevs: %s\n", errbuf);
        return false;
    }
    .....
    printf("Enter the interface number (1-%d):", i);
    scanf_s("%d", &inum);

    /* Jump to the selected adapter */
    for (d = alldevs, i = 0; i < inum - 1; d = d->next, i++);
    .....
    /* Open the adapter */
    if ((adhandle = pcap_open_live(d->name, // name of the device
        65536, // lenght of the packet to capture.
        1, // promiscuous mode (nonzero means promiscuous)
        1000, // read timeout
        errbuf // error buffer
    )) == NULL)
        return false;

    pcap_freealldevs(alldevs);
    return true;
}

```

图 4-3: 链路层连接建立代码

WorkerThread 是一个由线程定义与线程实例组成的结构体。上文中提到，根据条件编译结果，会得到与仿真机或模拟数据交流的两种线程定义，CreateRunnableThread 负责根据编译情况创建工作线程，图 4-4 工作线程 mMacLayerThread，作为与仿真机交流环境中的线程。

```

struct WorkerThread
{
    class MacReceivingTaskRunnable* mTaskRunnable = nullptr;
    class threading::RunnableThread* mRunnableThread = nullptr;
};

WorkerThread mMacLayerReceivingThread = WorkerThread
{producerRunnable, threading::RunnableThread::CreateRunnableThread(producerRunnable)};

```

图 4-4: 创建工作线程代码

4.1.3 数据接收与反馈

如图 4-5 中第一段代码所示，原始数据包接收线程启动后，会在收到结束请求前循环执行 `ReceiveData` 方法收取数据。该方法中含有回调函数 `OnDataLinkReceived`，用于进一步处理接收到的原始数据包，其实现如第二段代码所示。该回调函数中会将原始数据包中的多条指令拆开，并将每段指令依次存入集合 `framePackets` 中。最后调用 `QueuePacket` 函数将该集合加入仿真机消息接收队列中。

```
void MacReceivingTaskRunnable :: Execute( SInt32 inQueueIndex , ThreadID currentThreadID )
{
    // do data link layer net receiving logic
    do
    {
        MacLinkCommunication :: Instance () . ReceiveData
            ( MacReceivingTaskRunnable :: OnDataLinkLayerReceived , reinterpret_cast < u_char * > ( t
        } while ( ! mQuitRequested );
    }

void MacReceivingTaskRunnable :: OnDataLinkLayerReceived
                                ( const struct pcap_pkthdr* header , const u_char* pkt_data )
{
    static thread_local std :: vector < MacLogicFramePacket > framePackets ;
    MacLinkCommunication :: Instance () . OnDataLinkLayerReceived ( header , pkt_data , framePackets );
    // Flood fill in all eth packets
    auto runnable = reinterpret_cast < MacReceivingTaskRunnable * > ( param );
    std :: for_each ( framePackets . begin () , framePackets . end () , [ & ] ( auto & packet ) {
        runnable -> QueuePacket ( std :: move ( packet ) , runnable -> GetThreadID () );
    });
    framePackets . clear ();
    // Flood fill finished
    runnable -> OnRecvEthernPacketsFinish ();
}
```

图 4-5: 原始数据包接收代码

如图 4-6 所示。原始数据包的解析逻辑位于方法 `OnDataLinkLayerReceived` 中。由于原始数据包是数据链路层消息，其带有数据帧头、厂商自定义信息头等数据头部信息，拆分指令段前指针 `cursor` 需要先通过 `ParseMacLinkLayer-Header` 方法跳过该部分内容，进入正式数据部分。该部分中含有多条指令，每条指令头部有其长度信息，指针需要按照读取到的长度将数据部分进行拆分，并将指令内容复制到 `MacEncodingPacket` 对象中，加入上述仿真机消息接收队列中。虚拟仿真机从仿真机接收消息的过程就此结束。

在真实的飞行训练中虚拟仿真机自然要与仿真机进行直接沟通，但在没有仿真机的开发环境中，虚拟仿真机也需要能够读取模拟数据来驱动视景系统运

行。在编译项目时将条件设置为使用模拟数据，便可以开启该流程。

图 4-7 是处理模拟数据的实现。设备指针 mDevicePtr 调用 MockFramePacket 方法读取 csv 文件，ConvertCsv 函数将 16 进制字符两两一组转换为字节内容，最后将读取到的内容转换为 MacEncodingPacket 类型加入仿真机消息接收队列，此部分与连接仿真机时的过程相同。读取 csv 文件使用到了第三方库 rapidcsv。NextCurrentFrameMockData 方法中指针 mCurrentCursor 记录了当前读取到的 csv 文件行数，每次被调用都会从当前位置读取下一行数据内容。其中，mDocument 表示 csv 文件对象，GetRow 方法可以将 csv 中的一行内容读取到一个 vector 对象中。随后会根据读取到的指令代号调用相应的转换方法 ConvertCsv，把 vector 对象转换为指令结构体。

```
void MacLinkCommunication::OnDataLinkLayerReceived
(const struct pcap_pkthdr* header, const u_char* pkt_data, std::vector<MacLogicFramePacket>& outPackets)
{
    // Parse mac header + cae header + cae external header
    SInt32 cursorOffset = 0;
    SInt8 const* cursor = reinterpret_cast<SInt8 const*>(pkt_data);
    cursorOffset += cross::IProtocolConversion::PraseMacLinkLayerHeader(cursor, macDataLinkHeader);
    cursor += cursorOffset;

    // Parse packets
    SInt32 leftBytesToRead = macDataLinkHeader.byte_to_send_count;
    auto currentFramePacketPtr = &outPackets.emplace_back();
    while (leftBytesToRead > 0)
    {
        if (currentFramePacketPtr->is_full())
            currentFramePacketPtr = &outPackets.emplace_back();
        auto curPacketLength = (*reinterpret_cast<SInt16 const*>(cursor + 2)) * sizeof(SInt32) + sizeof(SInt32);
        SInt8* packetBuffer = new SInt8[curPacketLength];
        MacEncodingPacket packet(packetBuffer, curPacketLength, frameMsgNumber);
        memcpy(reinterpret_cast<SInt8*>(packet.Buffer.get()), cursor, curPacketLength);
        currentFramePacketPtr->push_back(std::move(packet));

        leftBytesToRead -= curPacketLength;
        cursor += curPacketLength;
    }
}
```

图 4-6: 原始数据包解析代码

第三章需求部分提到，视景系统会给仿真机反馈信息，该消息同样需经过虚拟仿真机，将 TCP 消息最终转换为原始数据包进行发送。在仿真机交流环境中 SendToSimulationDevice 方法负责反馈信息，即将指令结构体转换为原始数据包后发送，其调用上文中的工作线程下的同名方法，由该线程负责执行工作。最终的实现方法则在具体的设备中，因为不同厂商的仿真机使用的数据组织结构大有不同，此处列出了为适配 CAE 公司仿真机而编写的方法。

从视景系统中来的反馈信息类型为 FfsLogicFramePacket，需要将其最终封装为数据帧发送。首先需要获取其中的指令代号，调用对应指令的转换方法，

```

void CAEGenericSimulatorDeviceBase::MockFramePackets
{
    std::vector<cross::MacLogicFramePacket>& outPackets, cross::MockImageGeneratorTaskRunnable* mockRunnable)
{
    auto csvRowData = mockRunnable->NextCurrentFrameMockData();
    auto messageNumber = mockRunnable->NextMessageNumber();
    u_char* pkt_data;
    ConvertCsv(pkt_data, csvRowData);
}

std::vector<double> MockImageGeneratorTaskRunnable::NextCurrentFrameMockData() const
{
    if (mDocument.GetRowCount() <= 1)
        return std::vector<double>();
    if (mCurrentCursor >= mDocument.GetRowCount())
    {
        auto rowData = mDocument.GetRow<double>(mCurrentCursor - 1);
        return rowData;
    }
    return mDocument.GetRow<double>(mCurrentCursor++);
}

```

图 4-7: 模拟数据处理代码

```

bool SimulationDeviceContext::SendToSimulationDevice(cross::FfsLogicFramePacket const& inFfsPacket)
{
    return mMackLayerReceivingThread.mTaskRunnable->SendToSimulationDevice(inFfsPacket, cross::ThreadID::MainThread);
}

bool MacReceivingTaskRunnable::SendToSimulationDevice(cross::FfsLogicFramePacket const& inFfsPacket, ThreadID currentThreadID)
{
    return mDevicePtr->SendToSimulationDevice(inFfsPacket);
}

bool CAEGenericSimulatorDeviceBase::SendToSimulationDevice(cross::FfsLogicFramePacket const& inFfsPacket)
{
    .....
}

```

图 4-8: 反馈消息逻辑

若暂时无法理解该指令便产生警告。其次需要为该数据帧添加仿真机规定的头部和尾部信息，最后通过 MacLinkCommunication 单例中的 SendData 完成数据链路层信息的发送。

在使用模拟数据的情况下，如图 4-10 对于信息反馈只要解析其中的指令代号并打印日志即可。

4.1.4 多仿真机通用化

CAEGenericSimulatorDeviceBase 中的构造函数里进行了注册 convert 不同厂商的仿真机厂商对于数据包中字段组成和属性编排有明显差异。如表 4-1 中对比了 CAE 公司和波音公司模拟机对于飞机移动这一指令数据包中的情况。通过对比可知，首先他们的头部信息存在差别；其次虽然这一数据包中都含有

```

bool CAEGenericSimulatorDeviceBase::SendToSimulationDevice(cross::FfsLogicFramePacket const& inFfsPacket)
{
    .....
    for (int index = 0; index < inFfsPacket.Count; ++index)
    {
        FfsEncodingPacket ffsProtocol = inFfsPacket.Protocols[index];
        // Grab protocol converter
        auto ffsOpCode = ffsProtocol.message();
        if (!IsValidProtocol(ffsOpCode))
        {
            .....
            LOG_WARN("invalidate protocol {}H found in SEND TO SIM DEVICE, ignore it", command_str);
            return false;
        }
    }

    // Fill mac data link layer head
    IProtocolConversion::AssembleMacLinkLayerHeader(totalPacketLength, messageNumber, mac_header);
    memcpy(generator_pack, &mac_header, headerLen);
    // Fill mac body then
    memcpy(generator_pack + headerLen, &generator_body, totalPacketLength);
    // Fill mac tail finally
    memcpy(generator_pack + headerLen + totalPacketLength, &tail, 4);

    MacLinkCommunication::Instance().SendData(reinterpret_cast<u_char const*>(generator_pack), headerLen + totalPacketLength + 4);
    return true;
}

```

图 4-9: 反馈消息实现代码

```

bool MockMacReceivingTaskRunnable::SendToSimulationDevice(cross::FfsLogicFramePacket const& inFfsPacket, ThreadID currentThreadID)
{
    for (int index = 0; index < inFfsPacket.Count; ++index)
    {
        FfsEncodingPacket ffsProtocol = inFfsPacket.Protocols[index];
        auto descriptor_command = ce::net::FullFlightSimulatorPacket_EnumFFSCommand_descriptor();
        auto command_str = descriptor_command->FindValueByNumber(ffsProtocol.message())->name().c_str();
        LOG_INFO("MOCK Simulation device received a packet {}, msg number {}", command_str, ffsProtocol.msg_number());
    }
    return true;
}

```

图 4-10: 模拟数据下反馈

经纬度、海拔高度和欧拉角信息，但他们的排列顺序明显不同；最后，对于同一个属性如 Altitude 长度不同，存在精度上的差异。这些差异最终需要重新解释为统一格式以实现通用化。如图 4-11 中的代码所示，SD_PACKET_21H 便是一个统一后用于表示飞机飞行数据的指令结构体。其中包含了基本的飞行用属性。适配新的仿真机只需要重新编写对数据包的解析函数。

表 4-1: 不同仿真机字段比较

CAE 字段	长度	Boeing 字段	长度
OpCode	16bit	Packet ID	16bit
CS number	32bit	Entity ID	16bit
Latitude MSW	32bit	Roll	32bit
Latitude LSW	32bit	Pitch	32bit
Longitude MSW	32bit	Yaw	32bit
Longitude LSW	32bit	Latitude MSW	32bit
Altitude	32bit	Latitude LSW	32bit
Roll	32bit	Longitude MSW	32bit
Pitch	32bit	Longitude LSW	32bit
Yaw	32bit	Altitude	64bit
etc.		etc.	

```

typedef struct SD_PACKET_21H
{
    Sd_Header_Common header;
    UInt32 latitude_msw;
    UInt32 latitude_lsw;
    UInt32 longitude_msw;
    UInt32 longitude_lsw;
    SInt32 altitude;
    SInt32 roll;
    SInt32 pitch;
    SInt32 yaw;
} sd_packet_21h;

static constexpr SInt32 sd_Packet21H_size = sizeof(sd_packet_21h);
static constexpr cross::SIM_OPMSG sd_Packet21H_code = { 0x21 };

```

图 4-11: 转换类模板代码

4.2 协议转换模块

4.2.1 顺序图

图 4-12 是协议转换模块的顺序图。描述了由指令结构体与 ProtoBuffer 协议结构间的转换中各类的交互过程。正向过程中，转换类调用指令结构类中的

convert 方法开启转换过程，过程中会将指令中对应的属性赋值给 ProtoBuffer 协议结构，当此结构完整后便可以加入发送给游戏引擎的队列中。逆过程中，则是调用 ProtoBuffer 协议结构中的 convert 方法，将属性组合成反馈指令结构体，加入给到仿真机的队列中。

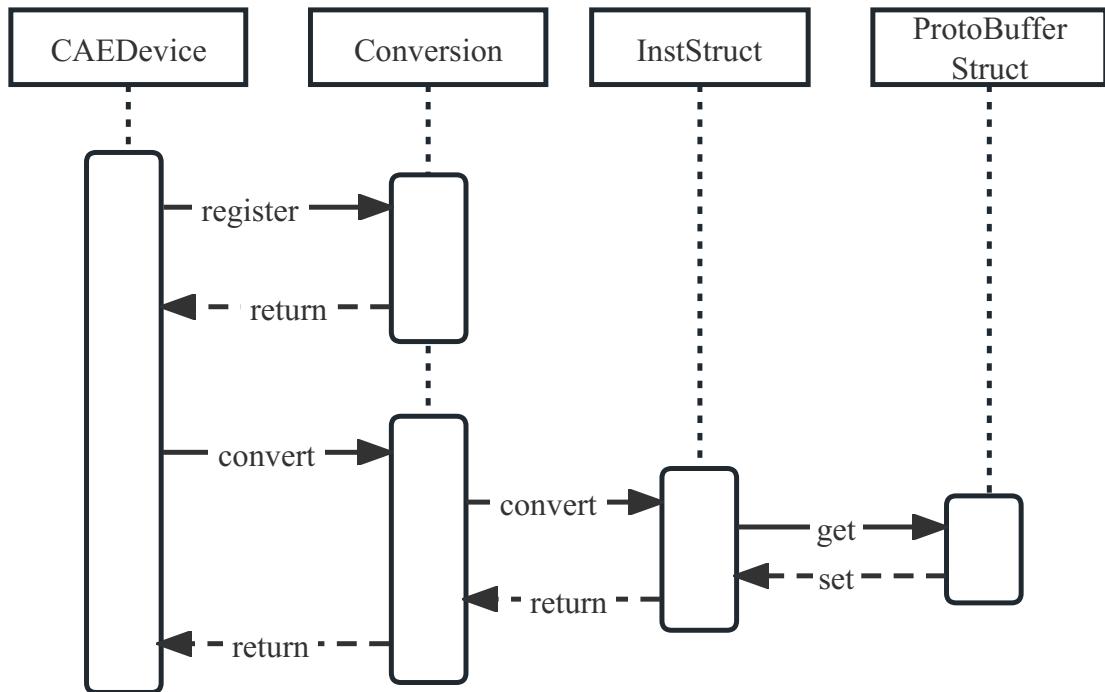


图 4-12: 协议转换顺序图

4.2.2 ProtoBuf 文件生成

使用 ProtoBuffer 首先需要编写一个 proto 文件定义程序中需要处理的结构化数据，在 ProtoBuffer 的术语中，结构化数据被称为 Message。如图 4-13 中的代码，定义了一个名为 GeodeticCSUpdate 的结构化数据，其中的成员为飞机飞行所需属性，且每一个成员均被赋予唯一的编号，在编码时使用该编号表示该成员。

写好 proto 文件之后就可以用 ProtoBuffer 编译器将该文件编译成目标语言。编译为 C++ 后会得到.pb.h 和.pb.cc 文件，分别为该类的头文件和实现文件，提供了一系列的 get/set 函数用来修改和读取结构化数据中的数据成员。当需要将该结构化数据序列化时，类中已经提供相应的方法来把复杂的数据变成字节序列。对想要读取数据的程序来说，也只需要使用类中的相应反序列化方

```
// This command is used to position a coordinate system (CS) in world coordinates
// CAE EMB190 Aircraft device OPCODE -- 0x21
message GeodeticCSUpdateStruct {
    required double latitude = 1;
    required double longitude = 2;
    required double altitude = 3;
    required double roll = 4;
    required double pitch = 5;
    required double yaw = 6;
    required double delta_latt = 7;
    required double delta_long = 8;
    required double delta_alt = 9;
    required double delta_roll = 10;
    required double delta_pitch = 11;
    required double delta_yaw = 12;
}
```

图 4-13: ProtoBuffer 结构

法来将这个字节序列重新转换为结构化数据。

4.2.3 协议转换算法

仿真机侧给到的属性并不是可以直接读取的浮点数，需要做出进一步的转换。欧拉角占 32 位，其数值单位为 $360/2^{32}$ ，即一份是一个非常小的角度，可表示范围是 -180° 到 179.999° 。对于海拔这类高度长度而言，单位统一为 0.5mm，即一份为半个毫米。对于此类转换只需要简单的乘法做单位转换。

经纬度这种两字节属性的转换则稍微复杂。其中第一个字节为高位，且前 24 位表示符号，剩余 8 位表示高位数字，后一个字节为低位，即单位是 $360/2^{40}$ 。且由于补码原因，需要对低位数字进行判断。低位数一定是一个正数，若为负数，则需要按照补码规则转换为正数再与高位相加。

fffff80 00000000 represents $-180.00 degrees$

00000040 00000000 represents $90.00 degrees$

```
#define REV2_DEG 3.27418092638254e-10
void DecodeUInt.ToDouble(double& outV, UInt32 msw, UInt32 lsw)
{
    SInt32 s_msw = static_cast<SInt32>(msw);
    SInt32 s_lsw = static_cast<SInt32>(lsw);

    double v1 = (double)s_msw * POW2_32;

    double v2 = (double)s_lsw;
    if (v2 < 0) v2 += POW2_32;
    double v3 = v1 + v2;
    outV = v3 * REV2_DEG;
}
```

图 4-14: 数据转换算法

4.2.4 转换方法注册

系统在需要转换的时候总要根据指令代号来确定对应的转换方法，便要求在系统运作前完成对于这些转换执行者的注册。如图 4-15 所示，类 ProtocolConversion 是转换执行者，这是一个模板类。T 代表一个指令结构体，如上文中的 SD_PACKET_21H；F 代表一个 ProtoBuffer 结构如 GeodeticCSUpdate。如果指令代号为 21H，则使用对应的转换执行对象里的 Convert 方法即可实现转换。

```
template<typename T, typename F>
class ProtocolConversion : public IProtocolConversion
{
public:
    virtual bool Convert(FullFlightSimulatorDeviceBase* inDevice, MacEncodingPacket const& inPacket, SInt8* outFfsBuffer, SInt32& outLength) override
    {
        // Grab actual sim struct pointer and convert
        T* simPtr = reinterpret_cast<T*>(inPacket.Buffer.get());
        T::Convert(simPtr, mFfsMessage);

        // Serialize packet body into buffer array
        outLength = static_cast<int>(mFfsMessage.ByteSizeLong());
        mFfsMessage.SerializeToArray(outFfsBuffer, outLength);

        return true;
    }
};
```

图 4-15: 转换执行模板类

对于一系列 ProtocolConversion 对象的注册代码如图 4-16 所示。转换执行者分为两组，一组为 mSimConverter，根据仿真机侧的指令代号找到转换执行者；另一组为 mFfsConverter，可以根据引擎侧的指令代号找到转换执行者。

RegisterProtocolConversion 函数负责将执行者加入两个 map 中，key 为指令代码，value 为转换执行者的指针。此函数会在整个环境的构造函数中被多次调用，完成所有注册。

```
std::unordered_map<cross::SIM_OPMSG, PROTO_FFS_CONVERTER> mSimConverter;
std::unordered_map<cross::FFS_OPMSG, PROTO_SIM_CONVERTER> mFfsConverter;

template<class T, class = std::enable_if_t<cross::TIsInheritTemplateOf<T, cross::ProtocolConversion>::value>>
void RegisterProtocolConversion(cross::SIM_OPMSG inSimMsg, cross::FFS_OPMSG inFfsMsg)
{
    auto covPtr = std::make_shared<T>();
    auto iCovPtr = TYPE_CAST_SHARD_PTR(ProtocolConversion, covPtr);

    mSimConverter[inSimMsg] = std::make_pair(iCovPtr, inFfsMsg);
    mFfsConverter[inFfsMsg] = std::make_pair(iCovPtr, inSimMsg);
}
```

图 4-16: 转换执行类注册代码

4.3 游戏引擎侧数据交换模块

4.3.1 顺序图

图 4-17 是游戏引擎侧数据交换模块的顺序图。描述了虚拟仿真机与游戏引擎间沟通时各类的交互过程。首先交流环境 IGContext 先通过 Tbuspp 的配置文件建立连接，Tbuspp 插件类会判断连接是否成功。连接建立后，交流环境通过 send 方法通知 FFSDevice 已有数据准备好发送，此时 FrameSync 类会介入判断在规定发送频率下此时是否可以发送。到达发送时间后，会调用 IGDevice 中的 send 方法执行具体发送动作。

4.3.2 Tbuspp 连接建立

虚拟仿真机与游戏引擎之间通过 Tbuspp 插件进行信息交流，第一步是要先建立连接。类 ImageGeneratorContext 表示在虚拟仿真机中与游戏引擎交流的环境，Initialize 函数表示了该环境的初始化流程。代码中同样用到了条件编译。条件 MOCKIG 表示不与游戏引擎建立真实连接，而是直接读取配置文件中的模拟数据文件路径，模仿视景系统给到的反馈，暂时该功能并未投入使用；否则进一步与游戏引擎建立连接。两种方式都需要启动新的线程负责消息的收发工作。

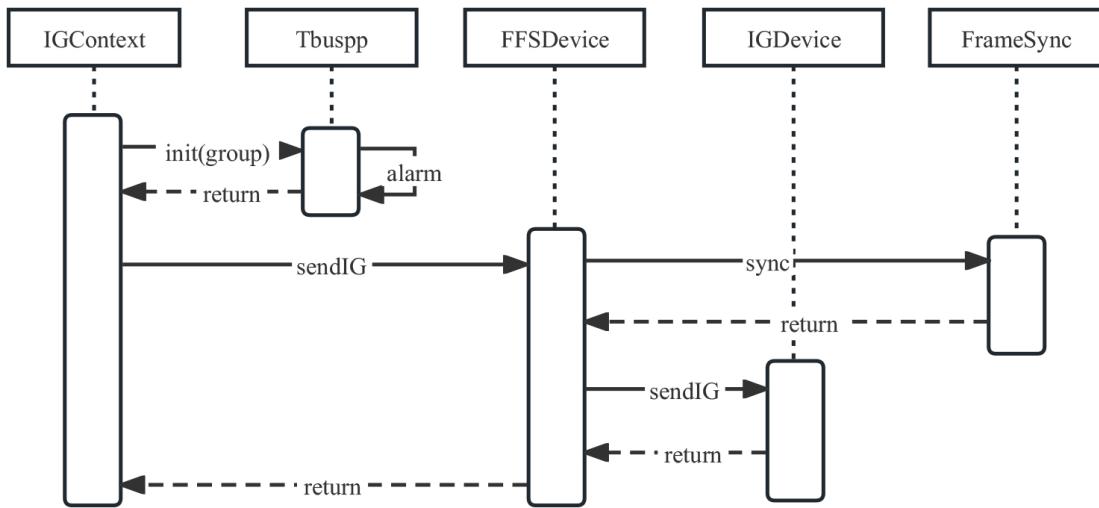


图 4-17: 游戏引擎侧数据交换顺序图

```

bool ImageGeneratorContext::Initialize(const DeserializerNode& inRootNode)
{
    ImageGeneratorTaskRunnable* producerRunnable = nullptr;
    #ifdef MOCKIG
    LOG_INFO("Initialize Mock IG!!!");
    {
        auto mockDocPath = GetExecutablePath().GetCString() + inRootNode["MockIGDocument"].AsString();
        producerRunnable = new MockImageGeneratorTaskRunnable(0, mDevicePtr, mPacketQueue, mockDocPath, mockSavePath);
    }
    #else
    LOG_INFO("Initialize IG!!!");
    {
        producerRunnable = new ImageGeneratorTaskRunnable(0, mDevicePtr, mPacketQueue);
    }
    #endif
    ImageGeneratorReceivingThread = WorkerThread{ producerRunnable, threading::RunnableThread::CreateRunnableThread(producerRunnable) };
    return true;
}
  
```

图 4-18: 引擎侧环境初始化代码

Tbuspp 连接的建立首先需要编写一个简单的配置文件，其中需要给出每个节点的 url 和 busid。如图 4-19 中所示，VSD 节点表示虚拟仿真机，另外还包含有两个视景节点，ImageGeneratorGroup 则代表所有的视景节点，当需要广播消息时可以给到该节点。此处的 url 均为本地地址，实际运行中，虚拟仿真机与视景系统运行在不同机器上，需根据实际情况进行配置。

在 Tbuspp 初始化方法中，node 参数表示已经过反序列化的配置文件内容，首先通过 LoadDomainConfig 读取所有的节点；其次通过 SetLocalClient 设置本地的服务节点，在虚拟仿真及侧该节点为 VSD。之后使用 tbuspp_open 方法开启连接，并检查连接状态是否正常。随后获取消息的接收和发送队列 in_queue 和 out_queue 并对他们进行一次清空操作，做好收发消息的准备。

```
{  
    "VSD": {  
        "agent_url": "127.0.0.1:8050",  
        "busid": "1.0.0.1"  
    },  
    "ImageGeneratorGroup": {  
        "agent_url": "127.0.0.1:8050",  
        "busid": "2.0.0.0"  
    },  
    "ImageGenerator0": {  
        "agent_url": "127.0.0.1:8050",  
        "busid": "2.0.0.1"  
    },  
    "ImageGenerator1": {  
        "agent_url": "127.0.0.1:8050",  
        "busid": "2.0.0.2"  
    },  
}
```

图 4-19: Tbuspp 配置文件

```
bool TbusDomainContext::Initialize(const DeserializeNode& node)  
{  
    LoadDomainConfig(node);  
    SetLocalClient();  
    .....  
    int err;  
    mClient.ep_ = tbuspp_open(&conf, 100, &err);  
    if (mClient.ep_ == nullptr)  
    {  
        LOG_ERROR("Connect tbuspp2 agent failed: {}", tbuspp_error_string(err));  
        return false;  
    }  
  
    mClient.in_queue_ = tbuspp_get_input_queue(mClient.ep_);  
    mClient.out_queue_ = tbuspp_get_output_queue(mClient.ep_);  
  
    Assert(mClient.in_queue_);  
    Assert(mClient.out_queue_);  
  
    // clean tbuspp2 queue  
    tbuspp_queue_clear(mClient.in_queue_);  
    tbuspp_queue_clear(mClient.out_queue_);  
  
    return true;  
}
```

图 4-20: Tbuspp 初始化代码

4.3.3 Tbuspp 数据发送

`SendToImageGenerator` 方法负责发送信息至游戏引擎。其参数包括需要发送的数据和接收数据的视景系统节点名称。使用 `SerializeToArray` 方法对待发送数据进行序列化后，便交由 `TbusDomainContext` 单例处理发送操作。其中先根据节点名称确定发送的目标节点，之后通过 `tbuspp_queue_write` 方法将发送内容写入本地服务节点的发送队列 `out_queue` 中，由 `Tbuspp` 完成发送。

```
bool CAEGenericSimulatorDeviceBase::SendToImageGenerator(cross::MacEncodingPacket const& inSimPacket, cross::FFS_TARGET inTarget)
{
    static thread_local SInt8 generator_pack[1024];

    cross::FfsEncodingPacket ffsPacket;
    if (Convert(inSimPacket, ffsPacket) == false)
        return false;
    SInt32 pack_size = static_cast<SInt32>(ffsPacket.ByteSizeLong());
    ffsPacket.SerializeToArray(generator_pack, pack_size);

    std::string dst_bsid = cross::TbusDomainContext::Instance().GetDstBsid(inTarget);
    cross::TbusDomainContext::Instance().QueuePacket(dst_bsid, reinterpret_cast<SInt32*>(generator_pack), pack_size);
    return true;
}
```

图 4-21: 引擎侧发送信息代码

```
bool TbusDomainContext::QueuePacket(const std::string &busid, SInt32 const* inData, UInt32 inSize)
{
    if (mClient.out_queue_ == nullptr)
        return false;

    tbuspp_id_t dest = tbuspp_bsid_aton(busid.c_str());
    return tbuspp_queue_write(mClient.out_queue_, dest, inData, inSize, nullptr);
}
```

图 4-22: Tbuspp 发送信息代码

4.3.4 稳定发送频率

在游戏中逻辑线程负责数据的产生，渲染线程负责根据数据渲染画面，一般比逻辑线程慢许多。因为逻辑线程跑的太快基本没有意义，还会耗光内存。因为逻辑线程不断的产生数据传递给渲染线程，如果渲染线程消费数据远远慢于产生数据，就会有越来越多的数据存于内存中。因此需要对逻辑线程做出一定限制，不能让其一直保持工作状态。

在本系统中对于仿真机数据的处理和传输可以认为是逻辑线程的工作，而游戏引擎部分基本为渲染线程的工作。图 4-23 是对于逻辑线程进行限制的核心方法 `Sync`。`high_resolution_clock` 是 C++11 中的新特性，提供的拥有最小计

次周期的时钟。在本系统中以微妙作为基本周期，FrameDurationUs 表示每次发送的间隔周期，如果要求为 60Hz 的发送频率则该值为 $10^6/60$ 微秒。若距离上一次发送的时间仍小于间隔周期，则使用 Sleep 方法将线程挂起剩余的时间长度。

```
void FrameFrequencyRestriction::Sync()
{
    a = std::chrono::high_resolution_clock::now();
    auto restrictionWorkTime = WorkTimeUs();
    constexpr double divisor = 1000.0 * 1000.0;

    if (restrictionWorkTime.count() < FrameDurationUs())
    {
        std::chrono::duration<double, std::micro> delta_us(FrameDurationUs() - restrictionWorkTime.count());
        auto delta_us_duration = std::chrono::duration_cast<std::chrono::microseconds>(delta_us);

        Sleep(delta_us_duration.count() / divisor);
    }

    b = std::chrono::high_resolution_clock::now();
}
```

图 4-23: 限制频率代码

在第二章的介绍中提到 Nagle 算法可以减少 TCP 包的个数，更高效的利用网络带宽。但同时也会带来一些延时问题，在实时交互应用中尤其重要。对于部署虚拟仿真机和游戏引擎的 Windows 系统机器都需要在注册表中将 TcpAckFrequency 字段和 TcpNoDelay 字段值修改为 1，以禁用 Nagle 算法。

 TcpAckFreque...	REG_DWORD	0x00000001 (1)
 TcpNoDelay	REG_DWORD	0x00000001 (1)

图 4-24: Windows 中禁用 Nagle

4.4 飞行控制模块

4.4.1 顺序图

图 4-25 是飞行控制模块的顺序图，描述了游戏引擎执行飞行逻辑时各类大致的交互过程。场景类 World 每帧调用 tick 函数执行该帧中的操作，逻辑脚本 FlightScript 调用引擎核心中负责变换和物理的 System 中的算法，完成飞机的移

动旋转，以及对于下方地形的检测。Camera 类也会被 World 每帧调用，执行不同视角逻辑下的移动旋转，变焦等动作。

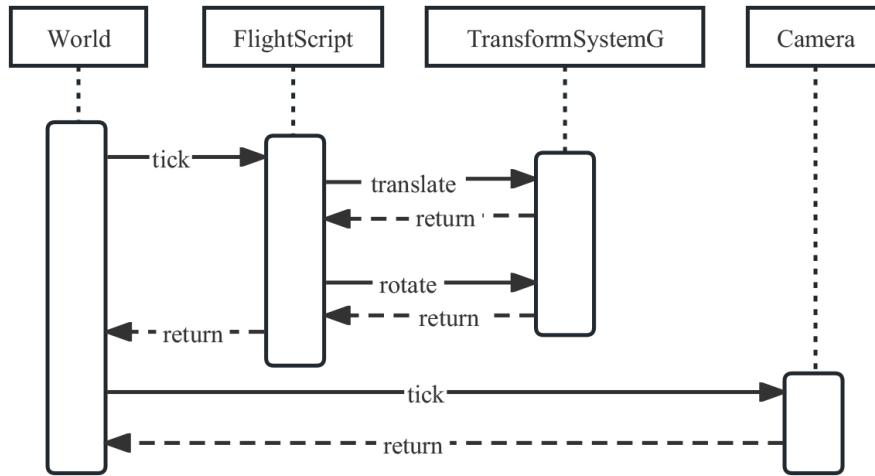


图 4-25: 飞行控制顺序图

4.4.2 飞行位置与姿态计算

由经度 longitude，纬度 latitude 和高度 altitude 组成的 LLA 坐标系，可以说是最为广泛运用的一个地球坐标系，它给出一点的大地纬度、大地经度和大地高程直观地告诉我们该点在地球中的位置，故又被称作经纬高坐标系。仿真机给出的飞机位置便是 LLA 坐标形式。地心地固坐标系 ECEF 是以地心为原点的笛卡尔坐标系，在游戏引擎中自然使用该坐标系更为便捷。在两种坐标系下地球都是默认为两级略扁的规则椭球形，赤道长半轴为 6378137.0 米，两极短半轴为 6356752.314245 米，扁率为 1/298.257223563。B. R. Bowring 在 1985 年便提出了两种坐标间的转换方法 [35]。如图 4-26 和图 4-27 所示。其中 a 为长半轴，b 为短半轴，e 为椭球的偏心率，N 为椭球的曲率半径。

$$e^2 = \frac{a^2 - b^2}{a^2}$$

$$N = \frac{a}{\sqrt{1 - e^2 \sin^2(lat)}}$$

```
Double3 TransformSystemG::WGS84CoordinateTransformRadian_To3D(double longitude, double latitude, double altitude)
{
    constexpr double a = 6378137.0; // unit meter
    constexpr double b = 6356752.314245; // unit meter
    constexpr double a2 = a * a;
    constexpr double b2 = b * b;
    constexpr double c2 = (a2 - b2) / a2;
    double v = a / sqrt(1 - c2 * sin(latitude) * sin(latitude));

    double x = (v + altitude) * cos(latitude) * sin(longitude);
    double y = ((1 - c2) * v + altitude) * sin(latitude);
    // reverse the z component to translate the 3D coordinate from right-handed to left-handed
    double z = -(v + altitude) * cos(latitude) * cos(longitude);

    return Double3(x, y, z);
}
```

图 4-26: LLA 转换为 ECEF 坐标代码

```

Double3 TransformSystemG::CartesianCoordinateTransform_ToWGS84(const double x, const double y, const double z)
{
    constexpr double a = 6378137.0;      // unit meter
    constexpr double b = 6356752.314245; // unit meter
    constexpr double a2 = a * a;
    constexpr double b2 = b * b;
    constexpr double e2 = (a2 - b2) / a2;
    constexpr double c2 = (a2 - b2) / b2;
    double p = sqrt(z * z + x * x);
    double R = sqrt(p * p + y * y);
    double tanBeta = (b * y) / (a * p) * (1 + c2 * b / R);
    double beta = atan(tanBeta);
    double sinBeta = sin(beta);
    double cosBeta = cos(beta);
    double tanLat = (y + c2 * b * sinBeta * sinBeta * sinBeta) / (p - e2 * a * cosBeta * cosBeta * cosBeta);
    double tanLon = x / (-z);
    double lat = atan(tanLat);
    double lon = atan(tanLon);
    lon = z > 0 ? (x > 0 ? lon + PI : lon - PI) : lon;
    double v = a / sqrt(1 - e2 * sin(lat) * sin(lat));
    double alt = p * cos(lat) + y * sin(lat) - a2 / v;

    return Double3(lat, lon, alt);
}

```

图 4-27: ECEF 转换为 LLA 坐标代码

除了 LLA 与 ECEF 之间的转换，我们还需要建立一个东北天坐标系 ENU，这是一个在物体所在位置建立的坐标系，三个轴分别指向东方，上方和北方。飞机的欧拉角便是以 ENU 坐标系为初始状态进行旋转。我们依旧可以仅从 LLA 坐标得到 ENU 坐标系。如图 ?? 所示，我们可以根据经度和纬度做简单的三角函数运算得出该点法线向量 normal ，即 ENU 坐标系中的 up 方向。同时法线与 y 轴所构成的平面一定与东方向垂直。在三维向量下，两个不平行的向量进行叉乘可以得到垂直于该平面的一个向量。根据该几何意义，将 normal 与 y 轴单位向量 $(0,1,0)$ 叉乘即可得到东方向。同理将东方向与 normal 叉乘，即可得到北方向，ENU 坐标系就此构建完毕。飞机的姿态需要使用欧拉角与 ENU 坐标系形成的旋转矩阵相乘，才能正确得到游戏引擎世界坐标系 ECEF 下的姿态。

$$\text{normal}.x = \cos(\text{lat}) * \sin(\text{lon})$$

$$\text{normal}.y = \sin(\text{lat})$$

$$\text{normal}.z = -\cos(\text{lat}) * \cos(\text{lon})$$

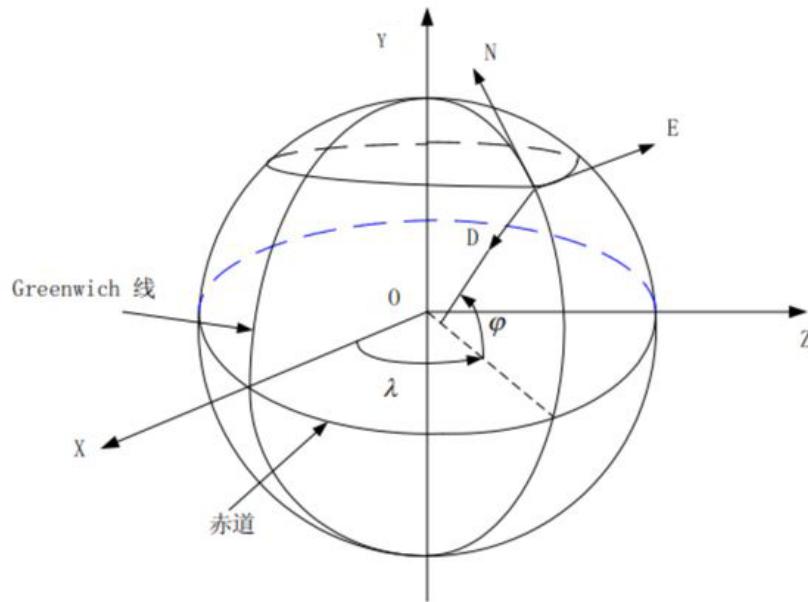


图 4-28: LLA 与 ENU 坐标系

4.4.3 可操控摄像机

游戏引擎渲染的画面是由摄像机决定的，要相对场景进行观察，摄像机是必不可少的。对于飞行训练来讲，一台在驾驶舱位置的第一视角摄像机即可满足训练要求。但是对于开发人员来讲，需要观察机身上如起落架、灯光等部件是否按照代码逻辑正常运作；同时也要观察机场中各要素是否正确渲染。除了第一视角的摄像机外还实现了可以环绕机身的摄像机以及可以脱离飞机自行移动的摄像机。如代码 4-29 所示，摄像机的模式由枚举类型 mode 控制，开发人员可以通过按键 X 进行模式切换。第一视角下不接受任何输入；环绕模式下只接受鼠标输入，根据鼠标的移动进行环绕；自由模式下则另外接受键盘输入可以自由移动。

第一人称摄像机拥有最简单的逻辑，位置和旋转与飞机保持一致即可。为避免飞机的原点并不在模型驾驶舱位置的问题，添加了 FocusonOffset 变量，表示摄像机与模型原点的相对位置，当然这个向量也要旋转至世界坐标中再使用。系统中关于旋转的信息都是以 TRSQuaternionAType 类定义的四元数表示。

环绕相机的操作效果是让相机在物体固定距离的球面上运动且始终看向对焦的物体。实际的实现思路则与之相反，可以让相机初始位于驾驶舱位置，通过鼠标移动确定观察角度后，再沿反方向后移固定距离，即可实现以飞机为焦

```

void TestEnvControllableCamera::PostTick(float deltaTime, GameWorld* inWorld)
{
    if (user->WasJustPressed(input::CEKeys::X))
    {
        mode = static_cast<CameraMode>((static_cast<int>(mode) + 1) % 3);
    }
    if (mode == CameraMode::FPP)
    {
        TickImplFPP(inWorld);
    }else{
        Mouse Input...
        if (mode == CameraMode::ORBIT) {
            TickImplORBIT(yaw_delta, pitch_delta, forward_delta, inWorld);
        }else{
            KeyBoard Input...
            TickImplFREE(rotationDelta, translationDelta, inWorld);
        }
    }
}

```

图 4-29: 摄像机切换代码

```

void TestEnvControllableCamera::TickImplFPP(GameWorld* inWorld)
{
    TRSQuaternionAType focusRotation = GetEntityRotT(inWorld, Focuson);

    TRSVector3AType focusPosition = GetEntityPosT(inWorld, Focuson) + focusRotation.Double3Rotate(Double3(FocusonOffset));

    TRSVector3AType finalPosition=focusPosition + focusRotation.Double3Rotate(Double3(FocusonOffset));
    TRSQuaternionAType finalRotation=focusRotation;

    sys->SetWorldTranslationT(comp.Write(), finalPosition);
    sys->SetWorldRotationT(comp.Write(), finalRotation);
}

```

图 4-30: 第一人称摄像机代码

点的环绕相机。在具体代码中，首先根据鼠标移动信息改变摄像机的俯仰和偏航角度，因为视野始终要与地平线平行，翻滚角不能去改变。新的欧拉角也是定义在当前位置的东北天坐标系下，需要与该坐标系对应的旋转再次相乘才能获得世界坐标系下的摄像机真实旋转姿态。确定好了旋转姿态后，向摄像机朝向的反方向移动固定距离即可。

自由摄像机相比环绕摄像机中旋转的逻辑是相同的，唯一的区别是相机不再会跟随某一物体移动。在具体代码中引入 `translationDelta` 参数表示这一帧下的移动距离，最终的摄像机位置为初始位置进行旋转后，在其朝向加上移动距离。

```

void TestEnvControllableCamera::TickImplORBIT(float horiontalDeltaAxis, float verticalDeltaAxis, float offsetDelta, GameWorld* inWorld) {
    AddPitchInputImpl(verticalDeltaAxis);
    AddYawInputImpl(horiontalDeltaAxis);
    mViewTargetOffset -= offsetDelta * TranslateScale.z;

    // controllable unit handle position
    TRSQuaternionAType focusRotation = GetEntityRotT(inWorld, Focuson);
    Double3 focusPosition = GetEntityPosT(inWorld, Focuson) + focusRotation.Double3Rotate(Double3(FocusonOffset));

    TRSQuaternionAType cameraRotation = TRSQuaternionAType::EulerToQuaternion64(mRotationEuler);

    Double3 EUNNormal;
    TransformSystemG::CartesianCoordinateTransform_ToWGS84(focusPosition.x, focusPosition.y, focusPosition.z, EUNNormal);
    TRSQuaternionAType worldRotation = FFSWGS84SystemG::CreateSphereSpaceWorldRotation(EUNNormal);
    TRSQuaternionAType finalRotation = cameraRotation * worldRotation;

    mViewTargetOffset = std::clamp(mViewTargetOffset, ViewTargetOffsetMinMax.x, ViewTargetOffsetMinMax.y);
    Double3 finalPosition = focusPosition + finalRotation.Double3Rotate(Double3(0, 0, -mViewTargetOffset));

    sys->SetWorldTranslationT(comp.Write(), finalPosition);
    sys->SetWorldRotationT(comp.Write(), finalRotation);
}

```

图 4-31: 第三人称环绕摄像机代码

```

void TestEnvControllableCamera::TickImplFREE(Float3 rotationDelta, Double3 translationDelta, GameWorld* inWorld)
{
    AddPitchInputImpl(rotationDelta.x);
    AddYawInputImpl(rotationDelta.y);

    TRSQuaternionAType cameraRotation = TRSQuaternionAType::EulerToQuaternion64(mRotationEuler);
    TRSVector3AType cameraPosition = GetEntityPosT(inWorld, mSelf);

    Double3 EUNNormal;
    TransformSystemG::CartesianCoordinateTransform_ToWGS84(cameraPosition.x, cameraPosition.y, cameraPosition.z, EUNNormal);
    TRSQuaternionAType worldRotation = FFSWGS84SystemG::CreateSphereSpaceWorldRotation(EUNNormal);

    TRSQuaternionAType finalRotation = cameraRotation * worldRotation;
    TRSVector3AType finalPosition = cameraPosition + finalRotation.Double3Rotate(translationDelta * TranslateScale.x);

    sys->SetWorldTranslationT(comp.Write(), finalPosition);
    sys->SetWorldRotationT(comp.Write(), finalRotation);
}

```

图 4-32: 自由摄像机代码

4.4.4 检测地形信息

游戏引擎在飞机飞行的过程中需要对沿法线下方的地形信息做检测，检测内容包括地形高度、离地距离等等。目的是将这些信息给反馈给仿真机做进一步的判断。离地距离较大时，飞机只需作为一个整体进行检测。当离地距离较小时即飞机起飞或降落的过程中，需要检测每一个起落架下方的地形信息。如图 4-33 代码所示，检测地形信息需要通过发射射线与地形进行碰撞，射线包含起点与方向两个基本属性。射线的方向为当前飞机所在经纬度，地球表面法线方向，使用上一节中提到的函数，便可通过经纬度得到法线方向。起落架的位置则是通过相对飞机原点位置给出，需要先将偏移量转换到世界坐标中。得到起点与方向，便可以完成射线发射，如果射线碰撞到地形，则将信息反馈。

方法 RayCast 负责射线的发射，其参数包括射线的起点，方向，最大距离，与场景中的哪些物体进行碰撞，最大碰撞次数，以及碰撞后的最终返回结

```

terrainInfo(script::Arguments const& args)
{
    Double3 up = {};
    Double3 position = TransformSystemG::WGS84CoordinateTransform_To3D(longitude, latitude, 0, up);
    Double3 dir = -up;
    NEUMatrix = CreateMatrixFromUpPosition(up, position);
    .....
    auto tPos = Double3::Transform(localTrans, aircraftTransMatrix);
    tPos = Double3::Transform(tPos, airportInvNEUMatrix);
    hit = phySys->RayCast(Float3(tPos), Float3(dir), 1e6, CollisionChannelBit::Terrain, HitFlag::Default, 1, &hitResult);

    if (hit)
    {
        Double3 hitPos = Double3::Transform(Double3(hitResult.position), airportNEUMatrix);
        hog = NEUPos.y;
        hat = NEUPlane.y - hog;
        .....
    }
}

```

图 4-33: 地形检测逻辑代码

果。在一个场景中可能存在数以百计的物体，如果遍历物体是非常耗时的工作。碰撞检测除了使用八叉树等技术对空间进行划分外，另一重要方法是对场景中的物体进行分类。本系统中的射线检测目标仅限地形，filterData 便是对于碰撞物体的限制。碰撞检测则是使用到了 Physx 物理引擎，这是一套由 AGEIA 设计的负责物理运算部分的处理器，类似专门负责处理图形的 GPU。NVIDIA 将其收购后，利用 CUDA 技术可使用 GPU 加速物理运算。该物理引擎最终的碰撞结果中会将如碰撞位置，距离，地形法线甚至纹理坐标等信息反馈。

```

UInt32 PhysicsSystemG::RayCast(const Float3& origin, const Float3& unitDir, float maxDistance, CollisionChannelBit collisionChannel,
                                HitFlag flag /*= HitFlag::Default*/, UInt32 maxHit/* = 1*/, PhysicsRayCastResult* outResults /*= nullptr*/)
{
    .....
    physx::PxQueryFilterData filterData;
    filterData.data.word1 = EncodeCollisionWord(false, CollisionTypeValue{}, collisionChannel);
    physx::PxRaycastBuffer hit(touchBuffer.get(), (maxHit == 1 ? 0 : maxHit));
    bool result = mPxScene->raycast(CrossVec3ToPxVec3(origin), CrossVec3ToPxVec3(unitDir), maxDistance, hit, pxFlag,
                                      filterData, nullptr, /*(maxHit == 1 && mRayCastCache.shape ? &mRayCastCache : nullptr)*/ nullptr);

    if (outResults)
    {
        if (maxHit == 1)
        {
            outResults->flags = PxHitFlagToCrossHitFlag(hit.block.flags);
            outResults->position = PxVec3ToCrossVec3(hit.block.position);
            outResults->uv = Float2A{ hit.block.u, hit.block.v };
            outResults->normal = PxVec3ToCrossVec3(hit.block.normal);
            outResults->distance = hit.block.distance;
            .....
        }
    }
}

```

图 4-34: 射线发射代码

4.5 本章小结

本章在需求分析及总体设计的基础上，对系统包含的四个模块核心功能的实现做了具体阐述。对于虚拟仿真机与仿真机和游戏引擎的数据交换模块，都介绍了建立连接的过程和收发信息的过程。仿真机侧额外介绍了为应对不同厂商模拟机而做的设计，游戏引擎侧则额外介绍了稳定信息收发帧率的实现方法。在协议转换模块中介绍了使用 ProtoBuffer 协议的流程方法，对较为特殊的字段转换算法做了详细说明。在最后的飞行控制模块中，介绍了让飞机飞行的实现，三种观察方式摄像机的实现，以及飞行中对于地形信息检测的实现。

第五章 系统测试

系统测试是为了在用户开始使用软件前，尽可能地发现软件中潜在的错误和不合理之处，确保最后将高质量的软件交付给用户。为了验证系统的功能和性能是否与需求分析时的规格说明一致，本章节在开发环境和真实 FFS 环境中分别做综合性的功能测试和性能测试。

5.1 开发 PC 环境测试

由于国内的 FFS 全部位于航空公司的训练基地内，不具备移动的可能性，平日开发过程中的测试均使用虚拟仿真机读取模拟数据，并在开发 PC 上完成视景渲染。受限于硬件性能，在开发环境下的测试以功能测试为主，确保功能正常运作即可，对于画面帧率和分辨率之类不做严格要求。

5.1.1 测试环境

在开发环境下，虚拟仿真机由笔记本电脑担任，模拟数据也位于该电脑上，在 Windows 系统上运行程序，便可以通过 Tbuspp 与视景建立连接，不断地读取并发送数据。对于运行视景的机器有较高的硬件要求，GPU 部分要使用较高型号才能让开发人员流畅的观察场景。同时由于搭载了大型机场场景数据库，对于硬盘容量也有较高要求。在测试环境的具体物理配置信息如表 5-1 所示。

为了顺利运行系统，服务器上需要配置系统运行的软件环境，详细信息如表 5-2 所示。开发 PC 测试环境下的软件要求与真实 FFS 下的测试相同，后不再赘述。两方都需要 C++ 与 Python 的运行环境，且双方通过 Tbuspp 沟通。在虚拟仿真机中，需要与仿真机进行直接的链路层信息交流，需使用 WinPcap 软件。测试环境部署配置完成后应当能保证系统不会因为环境问题出错。

表 5-1: 开发环境物理配置

设备	配置项	详情
视景运行机	CPU	AMD Ryzen 7 4700G 8-Core Processor
	GPU	NVIDIA GeForce RTX 3060
	内存	32GB
	硬盘	8TB
虚拟仿真机	系统	Windows 10 专业版 22H2
	CPU	AMD Ryzen 7 5800H with Radeon Graphics
	GPU	AMD Radeon(TM) Graphics
	内存	16GB
	硬盘	512GB
	系统	Windows 10 专业版 22H2

表 5-2: 开发环境软件配置情况

配置项	详情
C++ 运行环境	VS_C++_MSVC
Python 运行环境	Python 3.10
底层网络访问	WinpCap v4.1.3
服务网格	Tbuspp 0.6.0

5.1.2 功能测试

对于本系统的功能测试而言，视景中的飞机能够按照模拟数据中的路径飞行，即可说明从数据读取到协议转换再到最后的信息发送功能都是通过测试的。

图 5-1 是对频率限制工具 FrameFrequencyRestriction 测试的效果，测试过程中将其频率设置为 30Hz，运行 csv 文件读取部分程序，可以看到从读取第 18 行数据到读取第 48 行数据中正好间隔 1 秒钟。多种频率均能通过测试。

1679564190	17
1679564191	18
1679564191	19
1679564191	20
1679564191	21
1679564191	22

•••••

1679564191	44
1679564191	45
1679564191	46
1679564191	47
1679564192	48
1679564192	49

图 5-1: 30HZ 频率限制效果

我们在测试中使用到的模拟数据是飞行员在训练基地 FFS 上操作绕深圳宝安机场飞行一周后得到的数据。该数据的采样频率为 60Hz，共 6 万余组数据，每组数据中包含仿真机输出的原始经纬高和欧拉角信息。首先测试经过虚拟仿真机发送给游戏引擎的数据是否正确，采用的方法是在 GIS 软件中绘制这组输出数据，查看绘制路径与飞行员的飞行路径是否相同。图 5-2 中的轨迹是由上述原始数据经过虚拟仿真机处理最终输出的 6 万个点构成，可以看到轨迹起始位置精确的位于宝安机场的跑道上，起飞后环绕深圳市区飞行，最终截止于羊台山森林公园。与飞行员核对后确认本路径准确无误，说明由仿真机到游戏引擎中间一系列数据操作均可以通过测试。

在确认虚拟仿真机输出无误后，便可使用此数据进一步测试游戏引擎中飞行控制逻辑。图 5-3 是飞机第一视角下的飞行画面，可以看到根据模拟数据，飞机准确出现在宝安机场跑道上，且后续飞行中的爬升转向时的视角偏转完全符合实际情况，最终飞机按照轨迹终止于场景中的森林公园上空。此测试过程同样表明第一视角摄像机可以正确跟随飞机位置并做出相同的姿态。

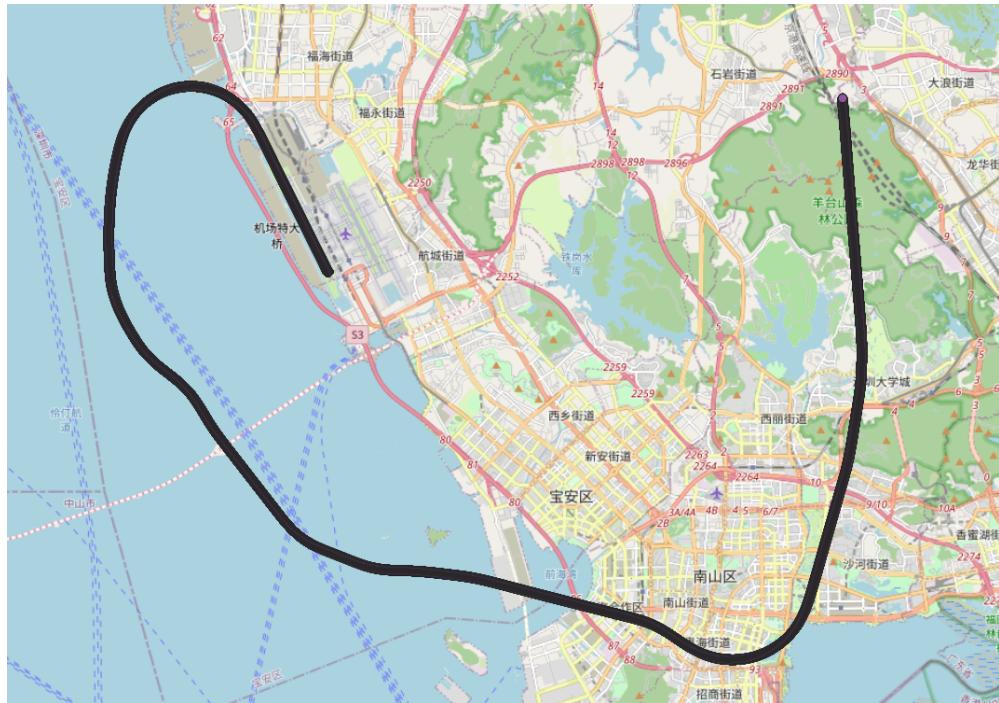


图 5-2: GIS 绘制路径



图 5-3: 飞机第一视角

当摄像机位于第一视角下，按下 X 键即可切换为环绕视角。如图 5-4 所示，此时摄像机不在位于驾驶舱位置，而是在一定距离外始终看向驾驶舱。鼠标左右方向的滑动可以改变 Yaw 角使摄像机水平方向的环绕，上下的滑动则改

变 Pitch 角是竖直方向的环绕。经测试操作感受与主流游戏基本一致。



图 5-4: 飞机环绕视角

当摄像机位于环绕视角下，按下 X 键即可切换为自由视角。如图 5-5 所示，此时摄像机会在当前位置脱离飞机，并可以去到场景中的任意位置。键盘的 WASD 键为前后左右移动，EQ 键控制上下移动，这些移动方向都根据摄像机自身坐标系改变。Shift 键则可以加速移动。经测试，操作感受与游戏引擎中的场景漫游基本一致。



图 5-5: 自由视角

飞机在飞行过程中会产生一些反馈信息，沿原路径最终送回仿真机。比如飞机在起飞和降落过程中会持续检测三个起落架的离地高度，测试时便选用此

信息，最终检查虚拟仿真机输出的原始数据包中此信息的值与训练基地 FFS 的反馈是否类似。表 5-3 对比了同一帧下 CrossEngine 和 FFS 关于起落架离地高度的反馈信息。由于地形资源存在些许差异，双方在此数据上并不会完全一致，在合理范围内即可。

表 5-3: 反馈信息对比

起落架编号	CrossEngine	FFS
1	3.822m	3.392m
2	3.787m	3.368m
3	3.787m	3.368m
...
1	40.031m	39.894m
2	39.914m	39.785m
3	39.912m	39.785m

5.2 FFS 环境测试

真实 FFS 环境测试去到南方航空珠海基地，这里是亚洲最大，机型最全的模拟飞行训练基地，共拥有 28 台民航局运输司认证的最高 D 等级 FFS，每年有超过 7000 名飞行员在此训练，是名副其实的中国民航飞行员培养摇篮。在真实 FFS 环境中，主要对于与仿真机连接、飞机飞行等功能测试；同时对于帧率的稳定性进行测试。

5.2.1 测试环境

在真实 FFS 环境下虚拟仿真机与视景运行机同样位于两台电脑上，此时虚拟仿真机不再读取模拟数据，而是与仿真机进行交流，即可以接受飞行员的操作。在硬件配置方面，基本为最新且性能最强的配件，保障视景流畅运行。在测试环境的具体物理配置信息如表 5-4 所示。

表 5-4: FFS 环境物理配置

设备	配置项	详情
视景运行机	CPU	Intel® Core™ i9-12900K Processor
	GPU	NVIDIA RTX A6000
	内存	64GB
	硬盘	8TB
虚拟仿真机	系统	Windows 10 专业版 21H2
	CPU	Intel® Core™ i9-12900K Processor
	GPU	Intel® UHD Graphics 770
	内存	32GB
	硬盘	2TB
	系统	Windows 10 专业版 21H2

5.2.2 功能测试

仿真机与虚拟仿真机通过网线连接。测试中启动与仿真机连接程序后，先选择要使用的网卡。连接建立成功后便可以开始正常接收数据。测试情况如图 5-6 所示。虚拟仿真机已成功读取到网卡上来自仿真机的原始数据包。

```
1: Local Loopback Adapter {2A6DCC36-6D3C-11ED-95F4-806E6F6E6963}
2: Realtek PCIe 2.5GbE Family Controller {DA6079E1-61E6-433A-B1B8-C3D43C1C67BF}
3: MediaTek Wi-Fi 6 MT7921 Wireless LAN Card {4528662D-7368-4B67-8640-E626E12
|6227C}
Enter the interface number (1-3):
```

• • • • •

```
|0324 10:12:14.430931 39136 server.go:407] recv msg:<rpc_chan:0xc00013e070,remo
te_addr=50-EB-F6-BB-C7-4C,local_addr=36-6F-24-2A-10-A9>cmd:3 seqno:91 agent_h
b_req:{agent_id:1 hb_seq:90}
|0324 10:12:14.430931 39136 server.go:652] handleRpc:req=cmd:3 seqno:91 agent
_hb_req:{agent_id:1 hb_seq:90},res=cmd:1003 seqno:91 agent_hb_res:{hb_seq:90}
|0324 10:12:14.432933 39136 server.go:447] send msg:<rpc_chan:0xc00013e070,
remote_addr=50-EB-F6-BB-C7-4C,local_addr=36-6F-24-2A-10-A9>cmd:1003 seqno:91
agent_hb_res:{hb_seq:90}
```

图 5-6: 仿真机连接

对于数据交换相关用例的测试同样是与服役中的 FFS 使用同一条预设路径飞行进行对比。本次测试选择的机场为广州白云机场，路径为跑道上的起飞过程。在服役 FFS 中运行的视频效果如图 5-7 所示，飞机即将在白云机场 02L 号跑道上完成起飞。图 5-7 则展示了本视景系统运行该路径的情况，系统可以根据仿真机的数据正确将飞机初始位置置于机场 02L 号跑道，且在后续起飞过程中视景画面与上述服役 FFS 视频中完全一致。目前画面仍存在球幕上的畸变问题以及颜色不准确的问题，但对于数据交换系统而言已经实现了核心功能。



图 5-7: 服役中 FFS 运行效果



图 5-8: 本视景系统运行舱内效果

5.2.3 性能测试

性能测试的部分主要测试连续运行下视景系统的帧率稳定性。帧率检测使用第三方的监控软件，连续飞行一个小时，记录平均帧率、最高帧率、最低帧率和 1% Low 帧的情况。其中最高帧率与最低帧率并不是某一时刻的帧率，而是极短时间内的平均帧率。1% Low 是选取了帧生成时间最长的 1% 的帧计算的平均帧率，这些帧不一定是连续的，所以一般会低于最低帧率，表示整段测试时间内某些时刻的剧烈帧率波动。

测试结果如表 5-5 所示，视景系统限制帧率上限为 60FPS，平均帧率十分接近这一数值，说明总体来讲基本能达到长时间运行下的帧率要求。最高帧率 60.8FPS 说明对于帧率的限制较为成功。1%Low 距离平均帧率有较大的距离，说明在某些复杂场景下会产生比较剧烈的瞬时帧率波动。

表 5-5: 帧率测试结果表

项目	数值
运行时间	3612.015s
总帧数	212748 Frame
平均帧率	58.9 FPS
最低帧率	48.6 FPS
最高帧率	60.8 FPS
1% Low	28.1 FPS

5.3 本章小结

本章首先介绍了系统测试，通过系统测试可以有效保证系统的质量，同时可以验证系统的可用性。随后将测试分为了开发环境和真实 FFS 环境，分别作了部分功能测试，并由于硬件性能原因只在 FFS 环境中做了性能测试。通过此测试可以证明系统满足需求分析中的功能需求，并在 FFS 环境中达成非功能需求。

第六章 总结与展望

6.1 项目总结

在国际局势紧张的大背景下，掌握技术主权愈发关键，也是我国正在紧锣密鼓进行中的任务，这两年在各行业也涌现了惊喜的突破。在关系到万家百姓的航空领域，我们已经拿出了 C919 这一成果，相信不久之后便能迎来国产飞机的第一批旅客。而在相关的全动飞行模拟机方面仍有许多空白。本文便介绍了一个目标为搭载至 D 级模拟机上的视景系统的初期工作。以让飞机模型正确飞行为目标，让游戏引擎与仿真机成功交流是本文的主要工作。

为了实现游戏引擎与仿真机的交流，系统中引入了虚拟仿真机这一角色。其负责让讲着数据链路层语言的仿真机与讲着网络层语言的游戏引擎顺利沟通，其核心流程是读取网卡的数据帧，解读其中的指令代码，将指令中的信息重新整合为结构体，最终按规定数据交换协议封装发送。当然也包括游戏引擎产生反馈数据的逆过程。在实施渲染工程中格外强调效率，因此数据交换协议采用了封装效率较高的 ProtoBuffer，同时禁止了有碍于小数据包发送效率的 Nagle 算法。此外为了帧率的稳定设计了数据发送时机的限制器，可以按照 60Hz 的帧率发送数据，避免飞行画面产生顿挫效果。同时为适配不同协议的各色仿真机，模拟仿真机设计了接口层，将不同种类仿真机的信息整合统一为指令结构体，为今后的适配留下空间。

文章行文介绍了视景系统的项目背景介绍了飞行模拟机、视景系统在国内外的技术发展历程，看出我国在该领域有许多空白。接着一章介绍了系统中涉及到的相关技术和专业概念帮助理解。在第三章中描述了系统需求分析，使用用例图和用例规格描述了九个用例。给出了数据交换系统的整体架构图，结合核心类图等阐述了每个模块的详细设计。在第四章的系统实现中。给出了各个模块的核心代码。最后对于整个系统在 PC 以及真实 FFS 上进行部署测试，测试了视景系统的实际运行效果。

6.2 项目展望

虽然该视景系统已经取得一定的效果，但显然距离终极目标仍有距离，需要在后续的工作中安排完成。

- (1) 在多种仿真机适配方面虽然在系统结构上留下了空间，但实际操作比较困难。因为难以与仿真机厂商达成合作，只能依赖现有文档去做逆向工程探索仿真机的私有协议结构与内容，这会是极其繁琐的工作。期待能够与未来的国产 FFS 厂商通力合作，不再经历如此费力的工作。
- (2) 从整个视景系统来看，距离达到 D 级模拟机的标准任重道远，作为新产品必将经过严格的审查，各类灯光、天气、突发事件的模拟必须准确到位。实现几百条的细节要求仍需要整个团队相当时间的通力合作。
- (3) 此视景系统是基于腾讯自研游戏引擎搭建，这不仅迈出国产视景系统的一大步，也是又一自研引擎成长的过程，当前综合渲染效果及稳定性肯定不及成熟的商业游戏引擎。随着该游戏引擎的逐步完善，整个视景系统的体验必然会越来越好。

致 谢

在南京大学的两年研究生生涯如白驹过隙，很快又要踏上新的人生旅途。这段时间里一个不得不提的话题便是新冠疫情，疫情初期是我准备研究生考试的日子，在我即将毕业时，新冠疫情似乎终于成为一段历史。疫情给我的学习生涯带来了些许不便，但同样也赋予了这一段时光特别的含义。时光飞逝但不代表回忆寡淡，在此我想感谢每一位为生活带来温暖的人。

首先要感谢我的父母。自从开启住校的学习生活，我与父母便是聚少离多，进入研究生阶段更是如此。但虽然相聚千里，每一通电话里都能感受到就在背后的支持。父母对我的信任也是这两年我能够自信面对各种人生第一次的重要原由，真正觉得自己可以独立面对社会。疫情管制全面放开后，最年轻力壮的我竟然成为家里症状最重的患者，在难熬的一个月里我又觉得自己始终是他们的孩子，有父母照顾的感觉真幸福。父母给了我一个幸福的原生家庭，我也一定要让这份幸福延续下去。

其次感谢同在南京租房的两位室友。由于没有校内宿舍，我拥有了一段租房上学的新鲜经历。虽说是微信群里开盲盒聚在一起的室友，却开启了一段珍贵的友谊。清早一起卖力蹬车进学校开启充满活力的一天，约定的每周聚餐令人心神放松。遇到阳关明媚的假期满城找公园散步，疫情居家期间三人手忙脚乱的做饭让我这个独生子女也体验了一段仿佛有亲兄弟的日子。总之三个人和谐的交流沟通除去了许多生活的烦恼，也让我们各自对未来的发展有了更清晰的认识。收获了如此愉快的租房生活，我对二位室友表示真诚的感谢。

同时感谢实习期间的同事们。这一段接近半年的实习是我第一次正儿八经的线下实习经历。记得第一次踏入办公楼大门时的紧张与不知所措，在被大家主动约饭和称兄道弟的交谈中渐渐消散。游戏引擎的开发是一个有些难度的领域，入职前我也只是有一个模糊的认识。导师给了我充足的时间去做了解，并在恰当的时机给予实际的项目问题，让我在解决问题的过程中对于游戏引擎的整体架构有了进一步的认识，更是增长了自身解决问题的自信心。这些可爱可敬的同事让我体验到了劳逸结合的工作氛围，也让我认识到今后的工作生涯需要找到持续学习与享受生活平衡点，才能更好的应对可能出现的风险。这段实

习经历让我在知识和心理上都获得了成长。

最后感谢人机交互实验室的导师和同学。在徐歆驰学长的带领下，我们小组实现了 VR 绘制的部分功能，在实验中体会到了 VR 在游戏之外的重要意义。这也增强了我对于游戏引擎的理解，为我的职业选择指明了道路。邵天成和张蓉蓉两位同学不仅是实验中鼎力相助的好搭档，更是成为了生活中分享轶事和烦恼的好伙伴。冯桂焕老师在实验室中不仅给予技术问题上的引导，更是努力让我们探索软件工程的本源，带领我们从软件角度思考社会问题。最后的毕业设计阶段面对形形色色的项目背景也总能迅速提出高屋建瓴的建议。遇到如此可爱的实验室大家庭是我的幸运。

致 谢

感谢在实验室度过的两年时光，老师无论在学术还是人生的指导上都对我起到了很大的帮助；师兄师姐小伙伴们们的鼓励支持和陪伴是我坚持下去的动力。

参考文献

- [1] 中国民用航空局. 2021 年全国民用运输机场生产统计公报 [EB/OL]. 2022.
https://www.mot.gov.cn/tongjishuju/minhang/202204/t20220408_3649981.html.
- [2] 飞常准. 2022 年度全球民航航班运行报告 [EB/OL]. 2023.
<https://data.variflight.com/reports>.
- [3] 国际航空运输协会. 国际航协 2021 年全球航空运输安全报告 [EB/OL].
2022.
<https://www.iata.org/contentassets/8a147274808e4700bf3aee94e31b00f2/2022-03-02-01-cn.pdf>.
- [4] 陈又军. 现代飞行模拟机技术发展概述 [J]. 中国民航飞行学院学报, 2011(2): 25–27.
- [5] HAWARD D. The Sanders Teacher[C] // Flight Vol 2 No.50. 1910 : 1006 – 1007.
- [6] PAGE R L. Brief history of flight simulation[J]. SimTecT 2000 proceedings, 2000 : 11 – 17.
- [7] ICAO. 飞行模拟机鉴定标准手册 [R]. [S.l.]: 国际民航组织, 1995.
- [8] 中国民用航空局. 飞机飞行模拟机鉴定使用标准 [R]. [S.l.]: 中国民用航空局飞行标准司, 2019.
- [9] 戈文一. 面向 D 级飞行模拟机视景系统的高真实感三维地形构建关键技术研究 [D]. [S.l.]: 四川大学, 2021.
- [10] HELLINGS E E. A Visual System for Flight Simulators[J]. British Communications and Electronics, 1960, 15(5) : 334 – 337.
- [11] SPOONER A M. Collimated Displays for Flight Simulation[J]. Optical Engineering, 1976, 15(3) : 215 – 219.

- [12] CAE. CAE 7000XR Series Level D Full-flight Simulator[EB/OL]. 2018.
[https://www.cae.com/civil-aviation/aviation-simulation-equipment/training-equipment/full-flight-simulators/cae7000xr/.](https://www.cae.com/civil-aviation/aviation-simulation-equipment/training-equipment/full-flight-simulators/cae7000xr/)
- [13] 刘长发, 李慧涌. 高等级飞行模拟机视景客观测试方法研究 [J]. 系统仿真学报, 2016, 28(7): 1609.
- [14] 王龙. 游戏引擎研究与分析 [J]. 软件导刊, 2018(5-7).
- [15] 黄河. 游戏引擎环境下 4K 虚拟演播室系统设计探讨 [J/OL]. 数字技术与应用, 2022(176-178).
<http://dx.doi.org/10.19695/j.cnki.cn12-1369.2022.11.54>.
- [16] 董鸿鹏. 飞行模拟器视景系统的设计与实现 [J]. 计算机应用, 2018(228-231).
- [17] GREGORY J. Game engine architecture[M]. [S.l.]: CRC Press, 2018.
- [18] DEGIOANNI L. Development of an architecture for packet capture and network traffic analysis[J]. Graduation Thesis, Politecnico Di Torino (Turin, Italy, 2000).
- [19] LU X, SUN W, LI H. Design and research based on WinPcap network protocol analysis system[C] // 2010 International Conference on Computer, Mechatronics, Control and Electronic Engineering : Vol 1. 2010 : 486 – 488.
- [20] DEGIOANNI L, BALDI M, RISSO F, et al. Profiling and optimization of software-based network-analysis applications[C] // Proceedings. 15th Symposium on Computer Architecture and High Performance Computing. 2003 : 226 – 234.
- [21] FENG J, LI J. Google protocol buffers research and application in online game[C] // IEEE conference anthology. 2013 : 1 – 4.
- [22] SUMARAY A, MAKKI S K. A comparison of data serialization formats for optimal efficiency on a mobile platform[C] // Proceedings of the 6th international conference on ubiquitous information management and communication. 2012 : 1 – 6.
- [23] SHIEH A, KANDULA S, SIRER E G. Sidecar: building programmable datacenter networks without programmable switches[C] // Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks. 2010 : 1 – 6.

- [24] LI W, LEMIEUX Y, GAO J, et al. Service mesh: Challenges, state of the art, and future research opportunities[C] // 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE). 2019 : 122 – 1225.
- [25] MINSHALL G, SAITO Y, MOGUL J C, et al. Application performance pitfalls and TCP’s Nagle algorithm[J]. ACM SIGMETRICS Performance Evaluation Review, 2000, 27(4) : 36 – 44.
- [26] HANIF M K, AAMIR S M, TALIB R, et al. Analysis of network traffic congestion control over tcp protocol[J]. IJCSNS, 2017, 17(7) : 21.
- [27] MAUREL C. CAE Tropos™ Interface Control Document[R]. [S.l.] : CAE Tropos™, 2003.
- [28] 魏子卿. 2000 中国大地坐标系及其与 WGS84 的比较 [J]. 大地测量与地球动力学, 2008, 28(5) : 1 – 5.
- [29] ZHOU Y, LEUNG H, BLANCHETTE M. Sensor alignment with earth-centered earth-fixed (ECEF) coordinate system[J]. IEEE Transactions on Aerospace and Electronic systems, 1999, 35(2) : 410 – 418.
- [30] DIMITRIJEVIĆ A M, RANČIĆ D D. Ellipsoidal Clipmaps—A planet-sized terrain rendering algorithm[J]. Computers & Graphics, 2015, 52 : 43 – 61.
- [31] STEVE MARSCHNER P S. Fundamentals of Computer Graphics fifth edition[M]. [S.l.] : CRC Press, 2022.
- [32] HEMINGWAY E G, O’ REILLY O M. Perspectives on Euler angle singularities, gimbal lock, and the orthogonality of applied forces and applied moments[J]. Multibody System Dynamics, 2018, 44 : 31 – 56.
- [33] SHOEMAKE K. Animating rotation with quaternion curves[C] // Proceedings of the 12th annual conference on Computer graphics and interactive techniques. 1985 : 245 – 254.
- [34] 柏龄. 从《阿凡达：水之道》浅析 3D 技术及高帧率技术的发展与应用 [J]. 现代电影技术, 2022, 11(50-53).

- [35] BOWRING B R. THE ACCURACY OF GEODETIC LATITUDE AND HEIGHT EQUATIONS[J]. Survey Review, 1985, 28: 202–206.