



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

学 科、专 业 名 称

研 究 方 向

指 导 教 师

2023 年 5 月 2 日

学 号：

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

Supervised by

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 Institute
Nanjing University

May 2, 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：_____

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
专业 _____ 级硕士生姓名：
指导教师（姓名、职称）：

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。

本项目立足于自主研发的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。然而进口仿真机并不对外开放接口，为了基于其开发视景系统，本文首先通过网络流量分析的方法结合文档验证了仿真机网络最高层为数据链路层，并解读了仿真机与视景系统沟通的二十余种指令信息。为了将视景系统接入该仿真机，在数据交换子系统中设计并实现了充当仿真机网络协议栈、多种指令转化以及与视景系统的图像生成器沟通等功能。过程中使用到 ProtoBuffer 协议来提升交换效率，设计了自定义指令集屏蔽仿真机间的差异。

在初步测试中，数据交换子系统可以正确处理并传递已解读的指令。然而也发现在需要连接多台图像生成器融合投影的情况下，图像的拼接部分有撕裂，抖动的现象。经过分析排查后，引入了网络帧缓存和插值机制，实现多图像生成器的数据同步与平滑，改善了投影效果。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、植被等多种效果进行研究与引入，尽早达成训练用模拟机的验收标准。

关键词：飞行模拟；视景系统；数据交换；数据同步

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:

POSTGRADUATE:

MENTOR:

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. The simulator that meets the requirements of flight training is called Full Flight Simulator(FFS). Currently, the FFS equipment of China depends on imports. FFS independent research and development must be accelerated because the international situation has changed dramatically and the COMAC C919 has received a certificate from the Civil Aviation Administration of China.

This project is based on the self-developed visual system who undertakes visual effects of FFS. The visual system needs to be driven by the instructions of imitator, which is the core of a simulator. However, the imported imitators do not open the interface. In order to develop the visual system based on it, through the method of network traffic analysis and documents in this paper, it is found that the top layer of the imitator network is the data link layer. At the same time, we verify more than 20 kinds of instruction that the imitator communicates with the visual system. In the data exchange subsystem, functions such as acting as the network protocol stack of the imitator, converting various instructions and communicating with the image generator of the visual system are designed and implemented, for the sake of connecting the visual system to the imitator. The ProtoBuffer protocol is used to improve the data exchange efficiency, and a user-defined instruction set is designed to shield the discrepancies between the imitators.

In preliminary tests, the data exchange subsystem is able to correctly process and deliver the verified instructions. However, it is also found that the splicing part of the

image has screen tearing and jitter when multiple image generators are connected. To improve projection effect, the network frame cache and interpolation mechanism are introduced to realize data synchronization and smoothing in multiple image generators. At present, under the control of CAE imitator, 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 effects, so as to up to standard of training simulator as soon as possible.

keywords: Flight Simulation, Visual System, Data Exchange, Data Synchronism

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 Wireshark 网络分析工具	7
2.2 WinPcap 架构	7
2.3 ProtoBuffer 协议	8
2.4 Tbuspp 中间件	9
2.5 Nagle 算法	10
2.6 CrossEngine 游戏引擎	11
2.7 航空坐标系及坐标系转换	12
2.8 V-Sync 垂直同步	14
2.9 本章小结	15
第三章 数据交换子系统的需求分析与概要设计	16
3.1 全动飞行模拟机整体概述	16
3.2 仿真机数据帧分析	17
3.2.1 数据帧的获取	17
3.2.2 数据帧的解读	19
3.3 基础数据交换需求分析	21
3.3.1 涉众分析	22

3.3.2 功能性需求	23
3.3.3 非功能性需求	24
3.3.4 用例设计	26
3.4 基础数据交换概要设计	31
3.4.1 系统逻辑视图	33
3.4.2 系统开发视图	33
3.4.3 系统进程视图	35
3.4.4 系统部署视图	36
3.5 本章小结	36
第四章 数据交换子系统的详细设计与实现	37
4.1 仿真机侧数据交换模块	37
4.1.1 流程图	37
4.1.2 核心类图	38
4.1.3 顺序图	39
4.1.4 关键代码	40
4.2 指令转换模块	43
4.2.1 流程图	43
4.2.2 核心类图	44
4.2.3 顺序图	45
4.2.4 关键代码	45
4.3 图像生成器侧数据交换模块	49
4.3.1 流程图	49
4.3.2 核心类图	49
4.3.3 顺序图	51
4.3.4 关键代码	52
4.4 本章小结	54
第五章 系统测试与优化	55
5.1 初步运行测试	55
5.1.1 测试环境	55
5.1.2 功能测试	56
5.2 数据同步机制	60
5.2.1 问题分析	60

5.2.2 网络帧缓冲.....	62
5.3 数据平滑机制.....	64
5.3.1 问题分析	64
5.3.2 插值平滑	65
5.4 二次测试.....	68
5.4.1 对照测试	68
5.4.2 性能测试	69
5.5 本章小结.....	70
第六章 总结与展望.....	71
6.1 项目总结.....	71
6.2 项目展望.....	72
参考文献	73

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 Wireshark 抓包界面	7
2-2 WinPcap 结构	8
2-3 CrossEngine 编辑器	11
2-4 飞机自身坐标	12
2-5 三类坐标系	13
2-6 画面撕裂效果	14
2-7 垂直同步示意图	15
3-1 模拟机运转方式	16
3-2 CAE 仿真机数据帧结构	17
3-3 数据帧中的指令结构	18
3-4 捕获的数据帧	18
3-5 仿真机与视景系统层级差异	19
3-6 21H 指令结构	20
3-7 经纬度数字转换算法	20
3-8 GIS 绘制路径	21
3-9 数据交换子系统边界	26
3-10 数据交换子系统用例图	27
3-11 数据交换子系统架构图	32
3-12 逻辑视图	34
3-13 开发视图	34
3-14 进程视图	35
3-15 部署视图	36
4-1 仿真机侧数据交互流程图	38

4-2 仿真机侧数据交互核心类图	39
4-3 仿真机侧数据交换顺序图	40
4-4 接收数据帧代码	41
4-5 发送数据帧代码	42
4-6 数据转换流程图	43
4-7 指令转换核心类图	44
4-8 指令转换顺序图	45
4-9 自定义指令结构	46
4-10 指令转换代码	47
4-11 纬度示意图	48
4-12 LLA 转 ECEF 坐标算法	48
4-13 图像生成器侧数据交互流程图	50
4-14 图像生成器侧数据交互核心类图	50
4-15 图像生成器侧数据交换顺序图	51
4-16 自定义指令封装代码	52
4-17 Tbuspp 收发消息代码	53
4-18 通用结构解封代码	54
5-1 测试环境设备布置	55
5-2 飞行控制指令测试结果	58
5-3 时间变换指令测试结果	58
5-4 灯光指令测试结果	59
5-5 天气指令测试结果	59
5-6 画面撕裂情况	60
5-7 更新周期的丢失	61
5-8 虚拟仿真机接收间隔	62
5-9 网络帧缓冲示意图	63
5-10 网络帧缓冲机制代码	64
5-11 画面抖动示意	65
5-12 投影仪刷新不同步	65
5-13 利用缓冲数据插值	66
5-14 网络帧缓冲机制代码	67
5-15 原视景系统运行效果	68

5-16 本视景系统运行效果	69
----------------------	----

附表清单

1-1 视景系统部分标准对比	3
2-1 ProtoBuffer 类型列表	9
2-2 引擎 Runtime 比较	12
3-1 仿真机部分指令列表	22
3-2 涉众分析列表	23
3-3 系统功能性需求列表	24
3-4 不同仿真机字段比较	25
3-5 系统非功能性需求列表	25
3-6 建立链路层连接用例描述表	28
3-7 发送数据帧用例描述表	28
3-8 发送数据帧用例描述表	29
3-9 使用模拟数据用例描述表	29
3-10 指令格式转换用例描述表	30
3-11 自定义指令封装用例描述表	30
3-12 收发 Tbuspp 消息用例描述表	31
5-1 软件配置表	56
5-2 硬件配置表	57
5-3 控制指令 21H 测试用例	57
5-4 反馈信息对比	60
5-5 帧率测试结果表	69

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指令控制。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。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。

但作为拥有行业主导地位的进口设备，CAE 仿真机本就不打算搭载除本公司以外的视景系统，它们之间通过怎样的接口通信毫无说明。为了将自研视景系统成功接入该仿真机，需要先理解仿真机的指令，再实现与方便视景系统使用的自定义指令间的转换方法，才能让双方成功交流，这便是数据交换子系统要承担的任务。此外有了该子系统帮助翻译，一定程度上屏蔽了仿真机侧的差异，为视景系统的通用性留下余地。

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 [6]。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。之后在二战爆发和电子计算机兴起的刺激下，飞行模拟机逐渐发展出更复杂体系和更真实的感官效果 [7]。最终发展为现在这种以飞行员在座舱中的操作为输入，仿真机根据输入计算各种状态，再以指令的形式驱动视景、声音、运动等系统工作的结构。

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 [8]，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 [9]。《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 [10]。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，驾驶员仅能坐在木制机舱中体验旋转。在上世纪 30 年代，一个位于机门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，通过相机扫过带有场景的皮带再将画面投影至飞行员眼前作为视景 [11]，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，并结合多种环境设置完成渲染的现代视景系统 [12]。

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 [13]，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[14]。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，现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 [15]。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 [16]。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 [17]。

杜等人基于 OGRE 面向对象图形引擎实现视景渲染 [18]，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 [19]。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。从业界来看，国内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[20]，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) 在项目开始阶段使用网络抓包工具对进口模拟机的输入输出数据包进行分析，发现其中只有以太网协议头部信息，说明仿真机的网络协议栈相当简单，与基于游戏引擎开发的图像生成器并不在同一层面上。为解读数据帧中指令的具体信息，又结合有限的文档信息进行分析，确认了部分指令的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。其主要的功能要求是按照仿真机的既定规则收发数据帧，将其解析为自定义指令后再与图像生成器交流。明确需求的基础上，设计了系统用例，结合逻辑、开发、时序和部署视图确定了系统架构。在实现部分完成了确认的 21 条仿真机控制指令和 5 条视景系统反馈指令的转换，保障如飞行、天气变化、碰撞检测等基本功能的实现。
- (3) 开发完成后，去到飞行训练基地对系统进行测试，测试环境下实现模拟座舱前方球幕的投影需要多个图像生成器融合投影，过程中发现在投影拼接部分会出现肉眼可观察的图像撕裂与跳帧现象。问题在于虚拟仿真机接收数据频率的不稳定，以及数据不能同时到达多台图像生成器。本文设计的网络帧缓冲机制可以同时缓解以上两种问题。
- (4) 加入帧缓冲机制后再次测试，发现画面之间仍有肉眼可察的抖动现象。排查后发现是多台投影仪刷新时间点不同步导致的问题。本系统中利用了实际刷新时间和理论时间的差值对数据进行了插值平滑，减少了图像间的抖动现象。
- (5) 基于基础功能并引入两种机制后最终进行测试，飞机可以在仿真机的驱动已基本稳定 60 帧的条件下飞行，且运行过程中没有出现明显的图像撕裂和

抖动问题。

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做了介绍。软件部分主要介绍了 WireShark 网络分析工具，WinPcap 架构、Tbuspp 中间件、CrossEngine 游戏引擎；算法部分则介绍了 ProtoBuffer 协议的编码方式，Nagle 算法，和垂直同步机制。

第三章基础数据交换的需求分析与概要设计。本章首先阐述了视景系统的运行方式，对进口仿真机的通信机制进行了研究。在此基础上对数据交换子系统进行了需求分析，并确定了系统用例。随后结合逻辑视图、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。

第四章基础数据交换的详细设计与实现。本章在概要设计的基础上，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三个模块各自的实现细节。

第五章数据同步与平滑机制的引入。本章开篇对数据交换系统按照指令代号进行了测试，期间发现存在多投影仪投影帧不一致、图像抖动问题。经问题排查与分析后加入了网络帧缓冲机制，帮助多个图像生成器中的数据同步；引入插值平滑机制抑制抖动问题。

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 Wireshark 网络分析工具

Wireshark 是开源网络包分析工具。主要作用是能够在网卡接口处捕获数据包，并显示数据包中的具体协议信息 [21]。其适用于 Windows 和 UNIX 系统，且支持多协议的网络数据包解析。在实践过程中，网络管理员用其检测网络波动问题，网络安全工程师用它来检查安全相关问题，协议开发者则使用它来测试新的通讯协议 [22]。本文中使用 Wireshark 对进口 FFS 中仿真机发送的数据包进行捕获，为的是研究其使用的通信协议，进而将自主研发的视景系统接入该仿真机中。

数据包捕获界面如图 2-1 所示，上方是捕获到的所有数据帧列表；左下方是所选数据帧的信息分析，包括其使用的网络协议、长度信息等；右下方则是这个数据帧的具体内容，此处以十六进制的形式展现。

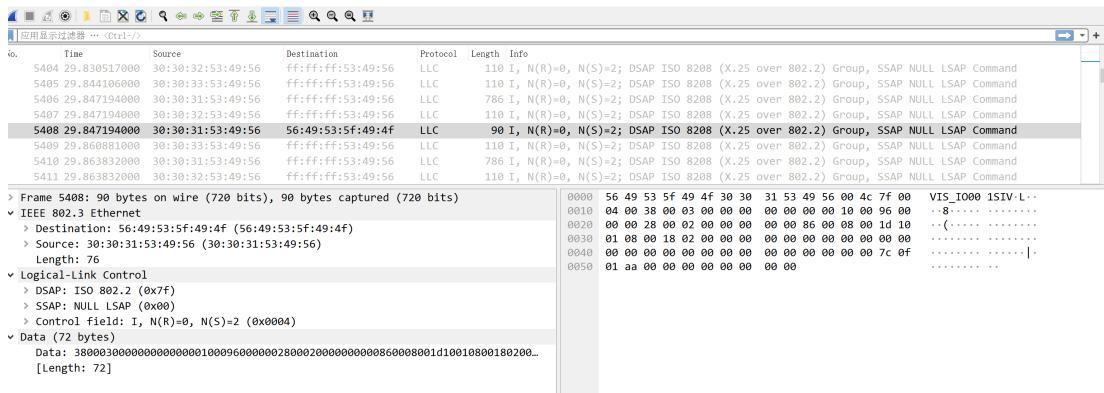


图 2-1: Wireshark 抓包界面

2.2 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 [23]。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 [24]。

抓包系统必须绕过操作系统的网络协议栈以获取网络上传输的数据帧，要求必须有一部分运行在操作系统核心，直接与网卡设备交互。Winpcap 作为 Win 平台上的抓包和网络分析架构，它包括一个操作系统内核的包过滤器，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[25]。本系统中用其收取和发送来自仿真机的数据帧，协助仿真机与视景系统交流。

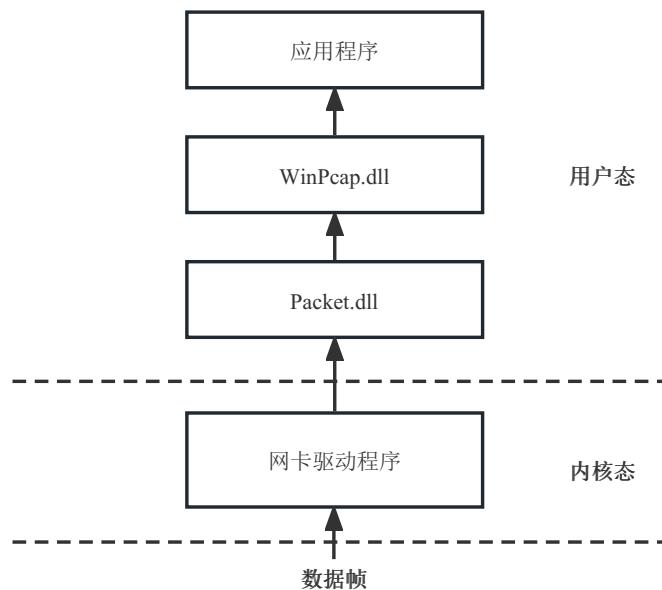


图 2-2: WinPcap 结构

2.3 ProtoBuffer 协议

Protocol Buffer 是 Google 提供的一种数据序列化协议，是可用于通讯协议、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 [26]。它属于一种二进制协议，相比较文本协议如 XML，JSON 等在体积和解包速度方面有巨大的优势 [27]，适合诸如本系统的中高频数据交换的场景。

ProtoBuffer 序列化后体积小巧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，通常情况下消息中非常大的数字占少数，因此采用 Varint 经常可以减少字节数来缩小体积。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。ProtoBuffer 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

反序列化过程中 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。之后再从该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 在反序列化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，否则无法正确反序列化。

2.4 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 [28]。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 [29]。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 作为虚拟仿真机与图像生成器间进行 TCP 协议沟通的插件。

2.5 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 就提出了一种减少需要通过网络发送包的数量来提高 TCP/IP 传输的效率 [30] 的方法，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 [31]。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。

2.6 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 [32]。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中甚至想象出的概念。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 2-3 展示了目前 CrossEngine 的编辑器界面。

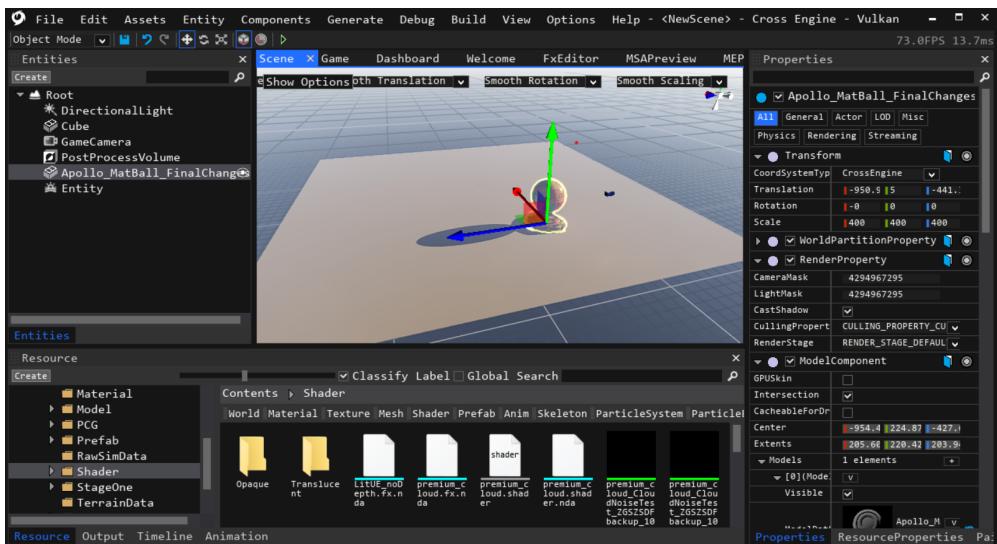


图 2-3: CrossEngine 编辑器

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

2.7 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 坐标系和东北天 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 2-4 所示。其作用为确定飞机上如起落架、各类灯的相对位置 [33]。

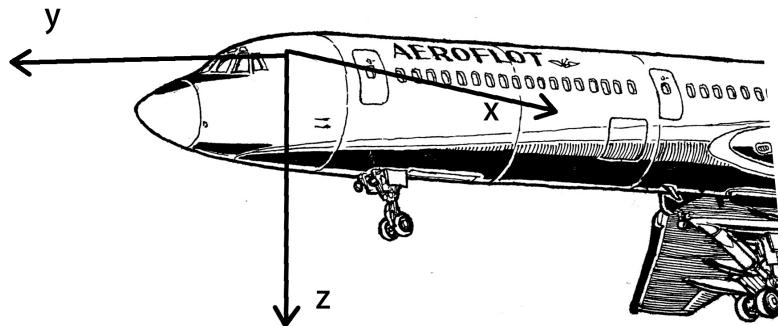


图 2-4: 飞机自身坐标

- (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平面的距离。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 [34]。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，x 轴指向本初子午线与赤道的交点，y 轴与地轴平行指向北极点，z 轴垂直于 xOy 平面构成右手坐标系 [35]。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 [36]。在 FFS 中，表示飞机姿态的欧拉角定义在该坐标下，且飞机的旋转是以偏航 Yaw，俯仰 Pitch，翻滚 Roll 的顺序完成。

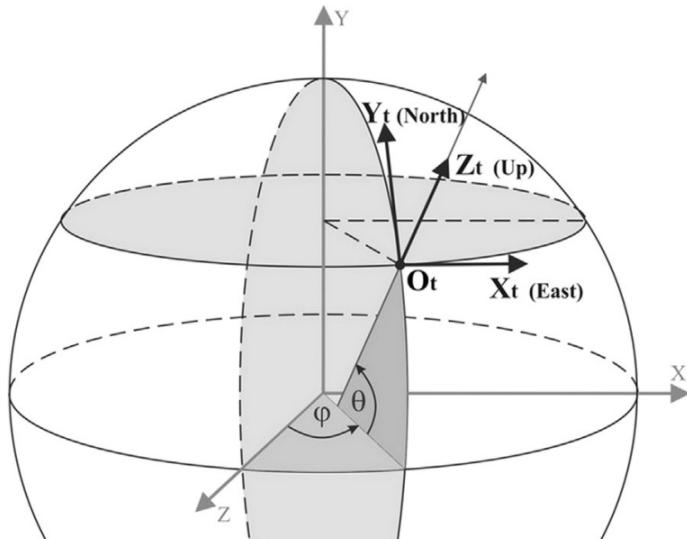


图 2-5: 三类坐标系

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不同坐标系的变换。向量的旋转和坐标的变换可以通过旋转矩阵表示 [37]。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投影，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。

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

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点在旋转后的坐标系中的坐标，转换为旋转前的坐标系中的坐标。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。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 [38]。使用欧拉角进行旋转时，中间的旋转取极端值不幸让某些坐标轴重合就会发生万向节死锁，导致丢失一个方向上继续旋转的能力 [39]。但在本项目中，仅就飞机的姿态而言是由仿真机给出的一个欧拉角直接确定，并不存在旋转叠加的问题，也就不会产生死锁。

2.8 V-Sync 垂直同步

显示设备刷新一帧图像时，并不是一次性刷新整个图像，而是从上到下一行一行将图像绘制出来。这个过程被称为逐行扫描，是目前显示设备最主要的成像方式。如果显示设备的刷新率为 60Hz，意味着一秒钟可以通过该方式成像 60 次。图像信息存放在显卡的缓冲区中，显卡会将渲染完毕的图像写入后缓冲区，与此同时前缓冲区中的图像会发送给显示设备。后缓冲区中新鲜图像写入完成后，为减少拷贝过程，会直接将前缓冲区和后缓冲区的名字对调 [40]。问题在于如果前缓冲区中的图像刚部分发送给显示设备，两个缓冲区便进行交换，之后发送的便是下一帧的图像，在显示设备上产生画面撕裂。图 2-6 展示了产生画面撕裂的效果。

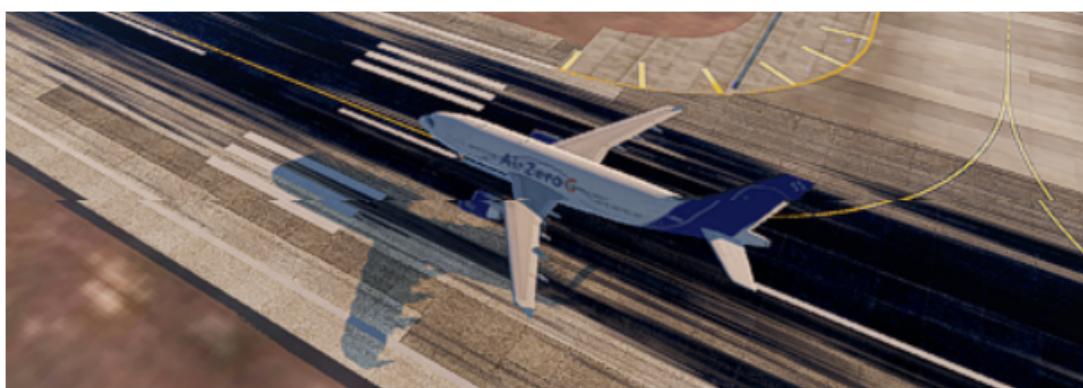


图 2-6: 画面撕裂效果

V-Sync 垂直同步就是为了解决画面撕裂的问题。显示设备有自己的固定刷新率，在一次逐行扫描结束到下一次开始前会有小段间隙，此时硬件会发出垂直同步脉冲保证前后缓冲可以在最佳的时间点交换 [41]，相当于限制了显卡的绘制频率。当然这只能在显卡绘制频率大于显示设备刷新率时才有效果，否则依旧会有卡顿跳帧问题。如图 2-7 所示本系统中使用垂直同步来同步投影仪与逻辑帧和渲染帧的运行，避免画面撕裂。

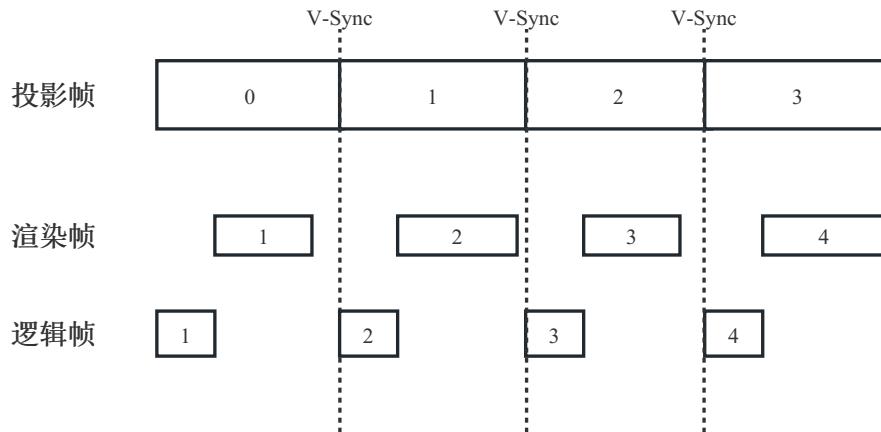


图 2-7: 垂直同步示意图

2.9 本章小结

本章介绍了视景系统中数据交換子系统开发中涉及的主要工具和技术。在软件方面介绍了实现网络包捕获的 Wireshark 工具；能够绕过操作系统网络协议栈的 WinPcap 架构；负责 TCP 协议通信的 Tbuspp；以及开发本视景系统的自研游戏引擎 CrossEngine。在技术方面介绍了具有高序列化与反序列化效率的 ProtoBuffer 协议；会造成小数据包发送延迟的 Nagle 算法；飞行中用到的 4 种航空坐标系；以及 V-Sync 垂直同步技术。它们共同保证了视景系统中飞行画面的流畅与稳定。

第三章 数据交换子系统的需求分析与概要设计

3.1 全动飞行模拟机整体概述

现代的飞行模拟机由模拟座舱、仿真机、视景系统、声音系统、运动系统等构成。其运作方式如图 3-1 所示。模拟座舱是一比一还原的对应型号飞机驾驶舱，飞行员通过操作各种操作杆与按钮驾驶飞机。这些操作会作为核心组件仿真机的输入，经过仿真计算后得到一系列的状态，比如当前飞机所处的位置，飞行的姿态，以及环境声音等等。教练员也可以增加如雨雪天气之类的设定，实现不同场景下的训练。这些状态会以指令的形式给到各个系统，视景系统根据指令完成场景搭建和画面渲染，声音系统据此产生各类音效，运动系统据此调整模拟座舱的姿态并产生体感加速度。

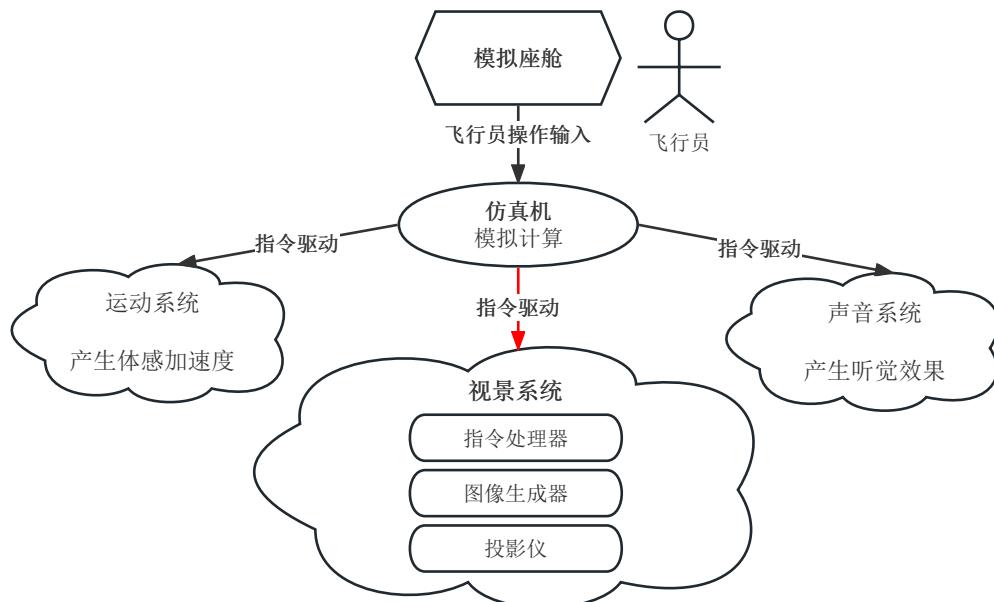


图 3-1: 模拟机运转方式

由上述可知仿真机相当于整个 FFS 的服务端，所有系统都需要与其连接，

听从指令才能正常工作。视景系统中指令处理器负责指令收发与解析，本文将该部分集成在虚拟仿真机中；图像生成器负责场景逻辑与渲染；投影仪负责成像。因此，想要基于现有仿真机开发视景系统，首先要理解该仿真机输出的指令，包括其中的数据组织结构和数字表示方法等。

3.2 仿真机数据帧分析

3.2.1 数据帧的获取

对于仿真机与视景系统间行为的分析，最直接的想法是获取处理指令的源代码。经过一番探索，在其中发现了名为 Visual Interface 进程。但该程序是编译型语言最终得出的二进制指令，并不知道其使用怎样的编译器编译而来；而且此方式通常用于分析主干代码，并不适合获取本文需要的指令细节，这是一条时间成本与预期结果都无法估算的路径。幸运的是，相关资料中同时拥有一份关于该仿真机的文档，其中含有对于仿真机与视景系统交互指令的详细说明，唯一的问题是该文档编写于 20 多年前，其时效性有待验证。

有了以上思路后，关于仿真机指令的分析方法就从获取源码变为了验证文档。对于两方通讯方式的研究，网络流量分析也是重要的方法。为了验证文档内容的正确性，本文使用网络抓包工具 Wireshark 截取了运行时仿真机与视景系统交流的原始数据帧，验证数据帧的组织结构与数字表示方法是否与文档中的说明相符。图 3-2 为文档中定义的数据帧结构，也就是说在理想情况下可以对捕获的数据帧按照该结构进行拆分。

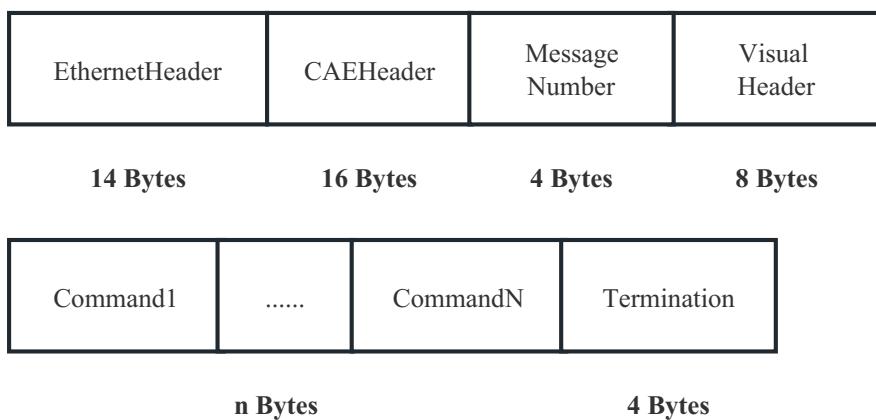


图 3-2: CAE 仿真机数据帧结构

在上述结构中本文最需要关注的是指令 Command 的具体结构。文档中关于指令结构的定义如图 3-3 所示。指令的数据部分前有 8 字节的头部信息，目前来说比较重要的头部信息一是 Opcode，它是每一种指令的代号，是指令的身份证。本文可以通过代号得知指令 Data 部分的组织结构，以此来进行反序列化。第二是 Size，它表明了这条指令的总长度，此信息可以帮助拆分数据帧中的多条指令。

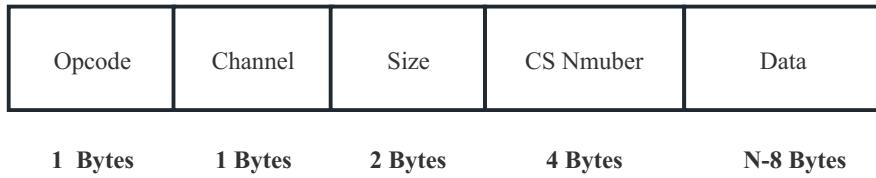


图 3-3: 数据帧中的指令结构

图 3-4 展示了通过 Wireshark 捕获到的流经视景系统网卡的一个数据帧。最左侧一列是对字节的计数，为十六进制，该数据帧共有 90 字节。依照文档中仿真机数据帧结构来看，第一行前 14 字节为以太网协议头，分别表示源 mac 地址，目标 mac 地址和除去该协议头后的长度，004c 正好表示 76 字节。跳过所有头部信息来到指令部分，第三行的 86 表示这之后是代号为 86 的指令数据段。以此来看，文档中对于数据帧结构的描述是正确的。

```

0000  56 49 53 5f 49 4f 30 30 31 53 49 56 00 4c 7f 00

0010  04 00 38 00 03 00 00 00 00 00 00 00 10 00 96 00

0020  00 00 28 00 02 00 00 00 00 00 00 00 86 00 08 00 1d 10

0030  01 08 00 18 02 00 00 00 00 00 00 00 00 00 00 00 00 00

0040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 7c 0f

0050  01 aa 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

图 3-4: 捕获的数据帧

3.2.2 数据帧的解读

通过对比文档中的数据帧结构说明和截取到的真实数据帧，从该数据帧头部仅有 Ethernet Header 可以得出仿真机网络协议栈非常简单。从网络模型角度看其最高层是数据链路层，数据内容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，否则仿真机无法正确理解反馈信息。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交流。图 3-5 形象的解释了仿真机与视景系统的网络层级差异。

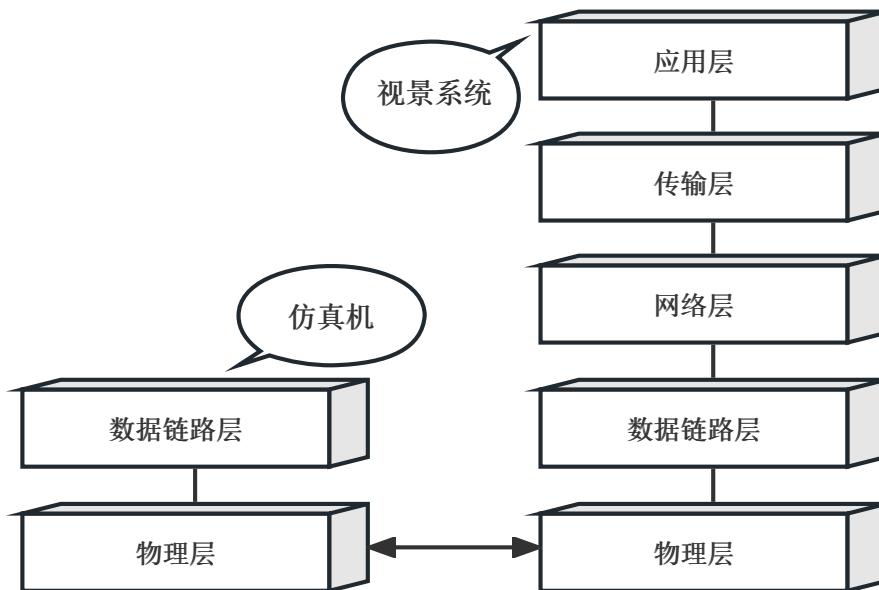


图 3-5: 仿真机与视景系统层级差异

当剥去数据帧头部的以太网协议头和 CAE 自定义的部分信息后便余下该数据帧中的所有指令数据，一个数据帧可能包含多条指令。此处以指令代号为 21H (H 表示十六进制) 的指令为例，此指令在飞行模拟中十分关键，其中以经纬度海拔和欧拉角的形式给出了飞机的位置和姿态信息，是仿真机每帧运行都要发出的指令。文档中关于该指令数据结构的描述如图 3-6 中的结构体所示。其种 Int32 仅代表对应字段占据了 32bit，并不表示实际数据类型。由于仿真机侧的数据序列化并没有使用任何数据交换协议，如果文档内容正确，则可以直接使用该结构体完成对应数据段的反序列化。

为了验证文档的时效性，本文需要对该指令中数据的合理性进行检查。在这步工作前需要先进行数字表示方式的转换。仿真机发送指令中的字段并不是可以直接读取的浮点数，需要做出进一步的转换。角度信息一般占 32 位，

```

typedef struct PACKET_21H
{
    Header_Common header;
    UInt32 latitude_msw;
    UInt32 latitude_lsw ;
    UInt32 longitude_msw;
    UInt32 longitude_lsw ;
    SInt32 altitude ;
    SInt32 roll ;
    SInt32 pitch ;
    SInt32 yaw;
}

```

图 3-6: 21H 指令结构

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° 。对于海拔这类高度或长度而言, 单位统一为 0.5mm, 即一份为半个毫米。此类数据只需要通过乘法做单位转换。

经纬度这种八字节属性的转换则稍微复杂。其中前四个字节为高位, 且前 24 位表示符号, 剩余 8 位表示高位数字, 后四个字节为低位, 整个字段的单位是 $360/2^{40}$ 。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。图 3-7 中给出了转换经纬度数字表示形式时使用的算法。

<i>fffffff80</i>	<i>00000000</i>	<i>represents</i>	<i>- 180.00degrees</i>
<i>00000040</i>	<i>00000000</i>	<i>represents</i>	<i>90.00degrees</i>

```

#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 3-7: 经纬度数字转换算法

完成数字形式转换后便可以进行正式的合理性检查, 为本文对飞行中截取到的六万余条 21H 指令进行了转换, 将其位置在 GIS 系统中进行标注, 最终形

成了如图 3-8 所示的飞行轨迹。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，说明文档对于指令 21H 的描述完全正确，该指令验证完成。

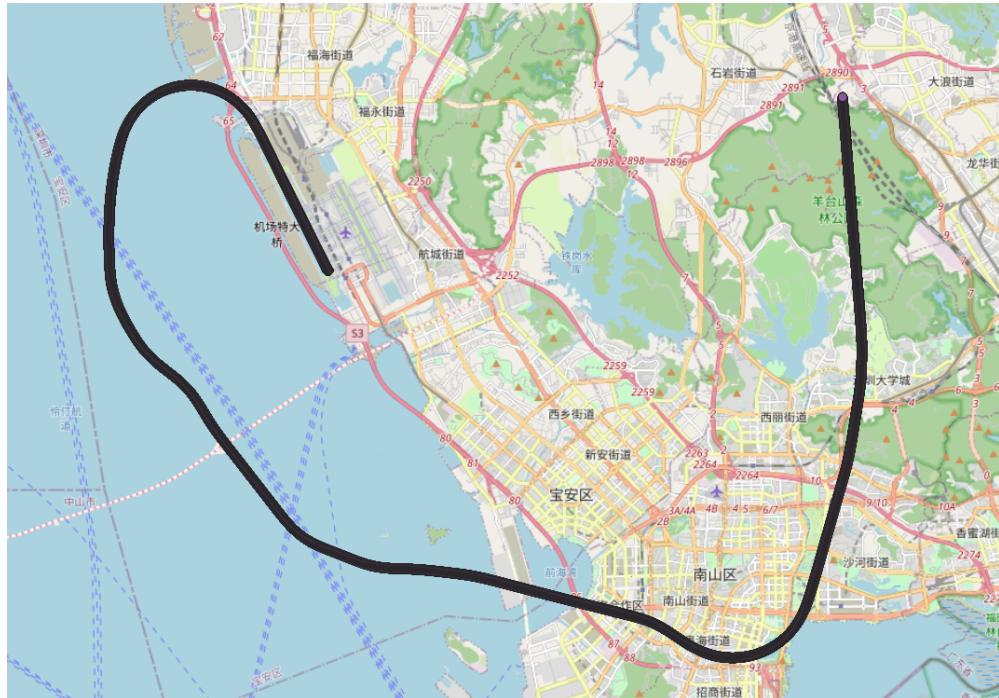


图 3-8: GIS 绘制路径

以上是对于最重要的飞行指令的验证过程，而对于设置灯光、天气等指令的验证可以通过手动操作变化，检查捕获到的数据帧中的变化进行验证。文档中给出的 CAE 仿真机指令共有近 70 条，由于文档老旧，其中有些指令已经停用，同时也有文档中并不存在的指令。好在目前阶段通过结合文档和飞行员经验，已有 21 条仿真机控制指令和 5 条视景系统反馈指令通过验证，其中囊括了飞机地理位置、机身灯光、雨雪天气、能见度等指令，也包含了如碰撞提示等反馈指令。这些指令能保障视景系统中如飞行、天气变化、碰撞检测等基本功能的实现。表 3-1 列举了已成功验证的部分仿真机指令代号及功能。

3.3 基础数据交换需求分析

确定了仿真机与原视景系统间的指令沟通方式后，便可以开展数据交换子系统的设计与实现工作。系统设计的第一步是需求分析，其主要目的是明晰非

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中的涉众，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。最后在理解需求的基础上，设计了多个系统用例，并使用 4+1 视图确定了系统的整体结构。

表 3-1: 仿真机部分指令列表

代号	指令用途
21H	负责控制飞机的位置与飞行姿态，是仿真机每帧都要更新的最基本指令，其通过经纬度海拔确定位置，通过该位置东北天坐标系下的欧拉角确定姿态。
42H	负责控制机身主要位置灯光开关和照明角度，指令中包含灯光的状态和相对机身水平竖直方向的两个角度。
43H	负责控制机场灯光，指令中包括跑道滑行灯、PAPI 灯、四组不同距离的环境灯等灯光的状态。
44H	负责部分天气效果的控制，指令中包含下雨、下雪及其程度的描述，云层的效果等信息。
45H	负责能见度的控制，指令中包含雾的浓度，可视角度等信息，在盲降训练时会被使用。
47H	负责所处时间段的控制，指令中包含白昼、黑夜、黎明、黄昏四种时间段信息，且根据不同的日期会有不同的太阳高度和月相信息。反映到视景系统中会产生不同的天光。
86H	负责一些通用信息的反馈，包括视景中使用的地球坐标系类型，当前加载的机场编号，是否发生严重碰撞等等。
82H	负责反馈飞机上某点的离地高度，一般是在降落时起落架相关信息的计算。

3.3.1 涉众分析

本系统作为视景系统中负责数据交换的子系统，参与者主要有二，一是每帧产生各种指令数据的仿真机，二是负责根据收到的指令执行逻辑并完成渲染的图像生成器。仿真机和图像生成器都希望有一座桥梁协助进行双向交流，以

实现最终飞行画面的渲染。详细的涉众分析如表 3-2 所示。

表 3-2: 涉众分析列表

涉众名称	涉众期望
仿真机	仿真机作为驱动图像生成器工作的数据源头，希望自己每一帧产生的指令数据能被图像生成器及时接收并正确解读。同时希望收到图像生成器的反馈信息以联动其他系统作出反应。
图像生成器	图像生成器作为执行逻辑和渲染画面的角色，希望从仿真机处取得指令数据，使用其中的数据完成逻辑计算。同时需要将产生的反馈数据发送给到仿真机。另外本视景系统中的图像生成器希望有搭载于不同仿真机上的能力。

3.3.2 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。上一节中对仿真机的分析里提到，仿真机网络协议栈简单，其输出与输入均为只用以太网协议封装的数据帧，这种数据帧不能交由一般的网络协议栈处理，需要自己实现针对该数据帧的解封和封装过程，转换为自定义指令后的数据则可以通过 TCP 协议栈发送给图像生成器。本文将处理仿真机指令数据帧的部分称为虚拟仿真机。另外图像生成器需要在没有仿真机的开发条件下使用模拟数据帧驱动逻辑运转。

在本视景系统中具体得到数据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，直接使用对应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便于图像生成器使用的数字形式。如将经度 0000007f ffffffff 转换为 179.99 后，结合纬度和海拔信息转换为笛卡尔坐标系位置，再赋值给自定义指令中的对应字段，生成本系统自定义的指令集。
- (4) 将该自定义指令通过数据交换协议进行序列化，并通过 TCP 协议发送给图

像生成器，此过程中的网络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，再使用同样的交换协议反序列化得到指令结构化数据，供给逻辑线程使用。
- (6) 图像生成器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。

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。数据交换子系统的功能性需求列表如表 3-3 所示。

表 3-3: 系统功能性需求列表

ID	需求名称	需求描述
R1	仿真机与虚拟仿真机交互	由于仿真机数据帧的特性，虚拟仿真机需要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发送频率读取网卡数据，确保指令数据的实时性。
R2	指令数据转换	仿真机使用的指令需要与图像生成器使用的指令进行映射，同时需要进行数字表示方法的转换，方便双方对于指令数据的使用。
R3	图像生成器与虚拟仿真机交互	图像生成器可以与虚拟仿真机以自定义指令的格式进行交互，图像生成器需要正确识别指令类型。

3.3.3 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 [42]。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，且运行时不出现可观察到的抖动现象。目前国内的进口模拟机来自 CAE 等外国公司，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。表 3-4 展示了 CAE 公司与 Flight Safety 公司仿真机关于飞机地

理位置的指令结构对比，它们在组织结构上存在明显差异，且其中的数字解释方法不尽相同。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 3-5 所示。

表 3-4: 不同仿真机字段比较

CAE 字段	长度	FS 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	64bit
Altitude	32bit	Longitude	64bit
Roll	32bit	Altitude	32bit
Pitch	32bit	
Yaw	32bit		
.....			

表 3-5: 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	一次飞行训练课持续 50 分钟，要求视景系统在连续运行 30 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本系统自定义的指令集，方便经过二次开发后在各类仿真机上搭载。

3.3.4 用例设计

图 3-9 展示了数据交换子系统的边界，确定了其在视景系统中的所处位置。首先本文并不关心仿真机中的指令数据如何产生，只需要接收或发送仅用以太网协议封装过的数据帧。数据经过虚拟仿真机一系列处理后通过 TCP 协议发送给图像生成器，图像生成器解析出指令数据后给到逻辑部分使用，从此走出子系统边界。当然逻辑执行完后还需要将结果交给资产管理和引擎核心部分实现场景加载和渲染。系统的功能集中在整个虚拟仿真机和图像生成器中的通信部分。

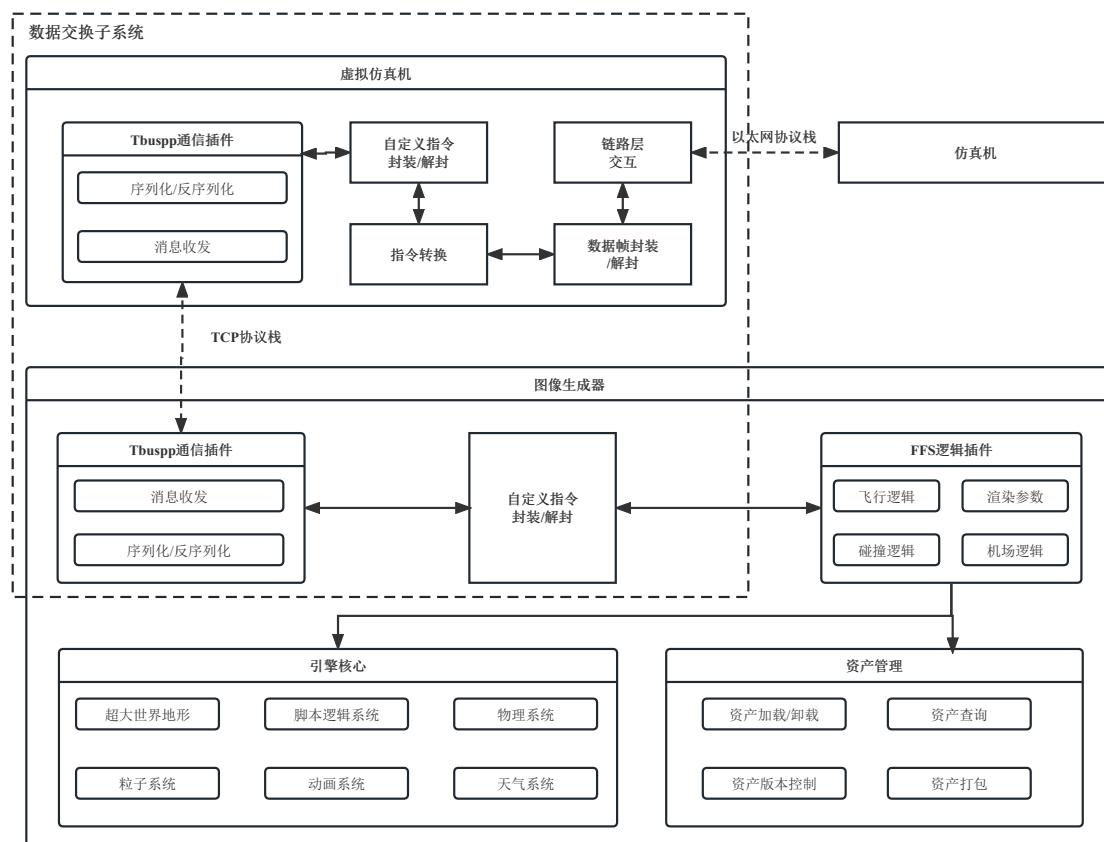


图 3-9: 数据交换子系统边界

经过需求分析后最终确定将数据交换子系统划分为七个用例，系统用例图如图 3-10 所示。其中的角色分为仿真机和图像生成器。仿真机与虚拟仿真机建立数据链路层连接，并使用各自的指令交互。仿真机中的指令数据经过解封、转换、封装等一系列动作后变为一个自定义指令，序列化后通过 Tbuspp 发送给图像生成器；图像生成器接收数据后，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图像生成器逆向发送最终以仿真机指令的形式给到仿真机。

具体而言，仿真机的用例包括建立链路层连接、发送数据帧、收取数据帧、以及指令格式转换。图像生成器的用例包括收发 Tbuspp 消息、自定义指令的封装与解封、指令格式转换和使用模拟数据。

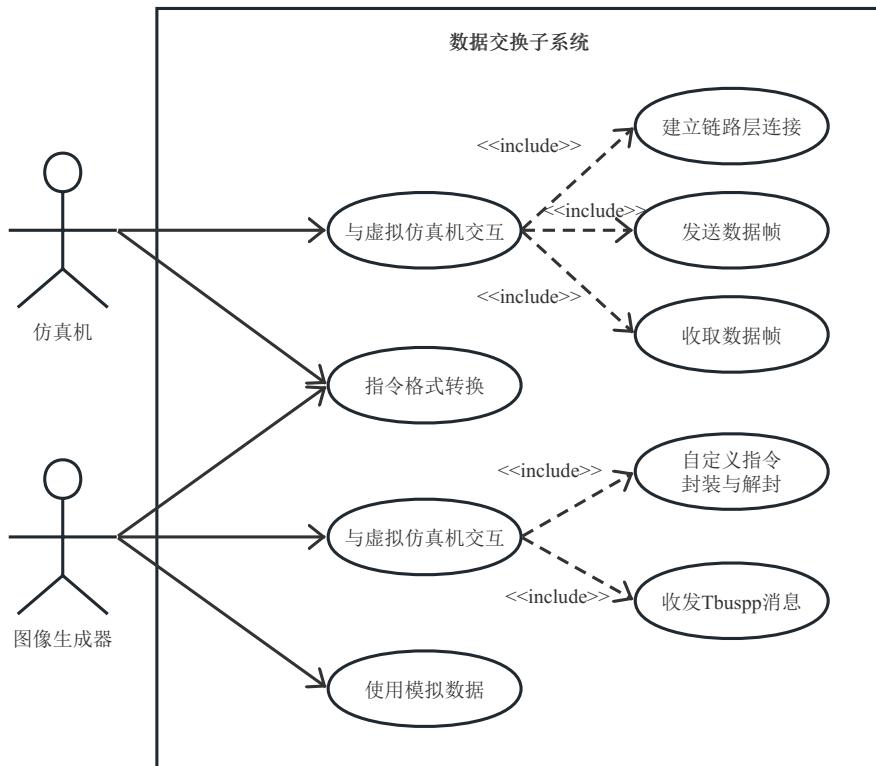


图 3-10: 数据交换子系统用例图

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，是仿真机与虚拟仿真机建立沟通路径的方式。由于仿真机简单的网路协议栈，其产生的数据帧不含有 IP 头、TCP 头等信息，与仿真机相关的数据不能直接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，并实现解封或封装。另需额外注意侦听数据时要使用及时转发模式，否则网卡的缓存机制会降低侦听的频率，产生较大的延迟，最终影响到图像的生成。用例的具体情况如表 3-6 所示。

表 3-6: 建立链路层连接用例描述表

ID	UC1
参与者	仿真机
触发条件	视景系统开始运行。
前置条件	虚拟仿真机处于接收仿真机消息模式。
后置条件	能够与虚拟仿真机进行数据交流。
优先级	高
正常流程	<ol style="list-style-type: none"> 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。

发送数据帧，是仿真机将数据帧发送到虚拟仿真机的过程。虚拟仿真机侦听到数据到来后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 3-7 所示。

表 3-7: 发送数据帧用例描述表

ID	UC2
参与者	仿真机
触发条件	仿真机向虚拟仿真机发送数据帧。
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数据段。
优先级	高
正常流程	<ol style="list-style-type: none"> 1. 仿真机发送数据帧。 2. 虚拟仿真机去除数据帧头部信息。 3. 读取指令长度信息并按该长度截取。 4. 将数据段加入仿真机指令集合。 5. 回到流程 3 循环至全部数据读取结束。
扩展流程	无

收取数据帧，是仿真机收取反馈信息的过程，此过程虚拟仿真机同样不依

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 3-8 所示。

表 3-8: 发送数据帧用例描述表

ID	UC3
参与者	仿真机
触发条件	仿真机反馈指令到达。
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	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. 将数据帧发送给仿真机。
扩展流程	无。

国内的 FFS 全部位于航空公司的训练基地内，其价格昂贵且庞大，是及其珍贵的训练资源，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 3-9 所示。

表 3-9: 使用模拟数据用例描述表

ID	UC4
参与者	图像生成器
触发条件	使用模拟数据驱动图像生成器运作。
前置条件	虚拟仿真机处于读取模拟数据模式。
后置条件	文本信息被转化为仿真机数据帧信息。
优先级	高
正常流程	1. 指定文件路径。 2. 虚拟仿真机按行读取文本。 3. 将字符两两一组转换为一个字节。 4. 按用例 2 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。

指令格式转换用例，描述了是仿真机指令与自定义指令相互转换的过程，转换后的指令格式更适合另一方使用。由于其流程类似，将两方的转换过程合并为同一个用例。具体情况如表 3-10 所示。

表 3-10: 指令格式转换用例描述表

ID	UC5
参与者	仿真机 & 图像生成器
触发条件	存在待转换的指令。
前置条件	注册了该指令的转换器。
后置条件	构建出自定义指令/仿真机指令。
优先级	高
正常流程	1. 获取指令结构中的指令代号。 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不存在转换器的指令直接丢弃。

ProtoBuffer 作为二进制数据交换协议，面对多种指令同样需要一个代号来决定使用何种结构反序列化指令字段。因此需要为自定义指令额外封装指令代号等信息，构成一个通用结构，整个过程需要两次序列化。作为接收方则同样需要两次反序列化。具体情况如表 3-11 所示。

表 3-11: 自定义指令封装用例描述表

ID	UC6
参与者	图像生成器
触发条件	图像生成器需要发送或接收自定义指令。
后置条件	自定义指令被封装并序列化。
优先级	高
正常流程	1. 将自定义指令序列化。 2. 为序列化后的包添加代号等信息，构成通用结构。 3. 将通用结构再次序列化。

图像生成器与虚拟仿真机利用 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 3-12 所示。

表 3-12: 收发 Tbuspp 消息用例描述表

ID	UC7
参与者	图像生成器
触发条件	有通用结构待发送。
前置条件	Tbuspp 连接成功建立。
后置条件	无
优先级	高
正常流程	<p>数据发送过程：</p> <ol style="list-style-type: none"> 用 ProtoBuffer 协议序列化通用结构。 将消息写入 Tbuspp 发送队列。 <p>数据接收过程：</p> <ol style="list-style-type: none"> 从 Tbuspp 接收队列读取消息。 使用通用结构反序列化该消息。
扩展流程	Tbuspp 连接失败产生警告。

3.4 基础数据交换概要设计

数据交换子系统的架构设计如图 3-11 所示，整个系统分为三个部分，数据交换的功能集中于虚拟仿真机与图像生成器中。仿真机与虚拟仿真机间通过数据链路层协议进行交互，虚拟仿真机与图像生成器通过 TCP 协议交互。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。虚拟仿真机则将仿真机指令翻译为图像生成器方便使用的自定义指令，使用自定义指令同时屏蔽了不同仿真机的差异。图像生成器需要按照收到的指令数据进行逻辑演算并渲染飞行画面，并提供飞行中的反馈信息。

功能主要分为三个模块，分别是仿真机侧数据交换模块，指令转换模块和图像生成器侧数据交换模块。仿真机侧数据交换模块负责处理与仿真机进行交流的全部任务。其中使用到 WinPcap 作为直接访问网卡的工具，数据帧的收取和发送都通过 WinPcap 实现。为处理仿真机的特殊数据帧，需要自己实现数据帧的解封和封装逻辑。指令转换模块是一个承上启下的模块，其负责仿真机指

令与自定义指令间的转换。其中涉及到数字表示方式的变化和如经纬海拔与笛卡尔坐标系转换的算法。此模块不仅提供翻译功能，作为视景系统的一部分，其屏蔽了仿真机侧指令的差异。图像生成器侧数据交换模块负责处理与图像生成器进行交流的全部任务，此处的通信则使用常规的 TCP 协议，不再需要亲自实现网络协议栈。此处的数据交换协议使用二进制协议 ProtoBuffer，其拥有非常高的序列化反序列化速度，以及相当小的序列化体积，适合交换频率高的场景。图像生成器中也有一个类似的模块负责与虚拟仿真机交流。

数据交换子系统完全使用 C++ 语言开发，图像生成器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 脚本嵌入，方便修改。

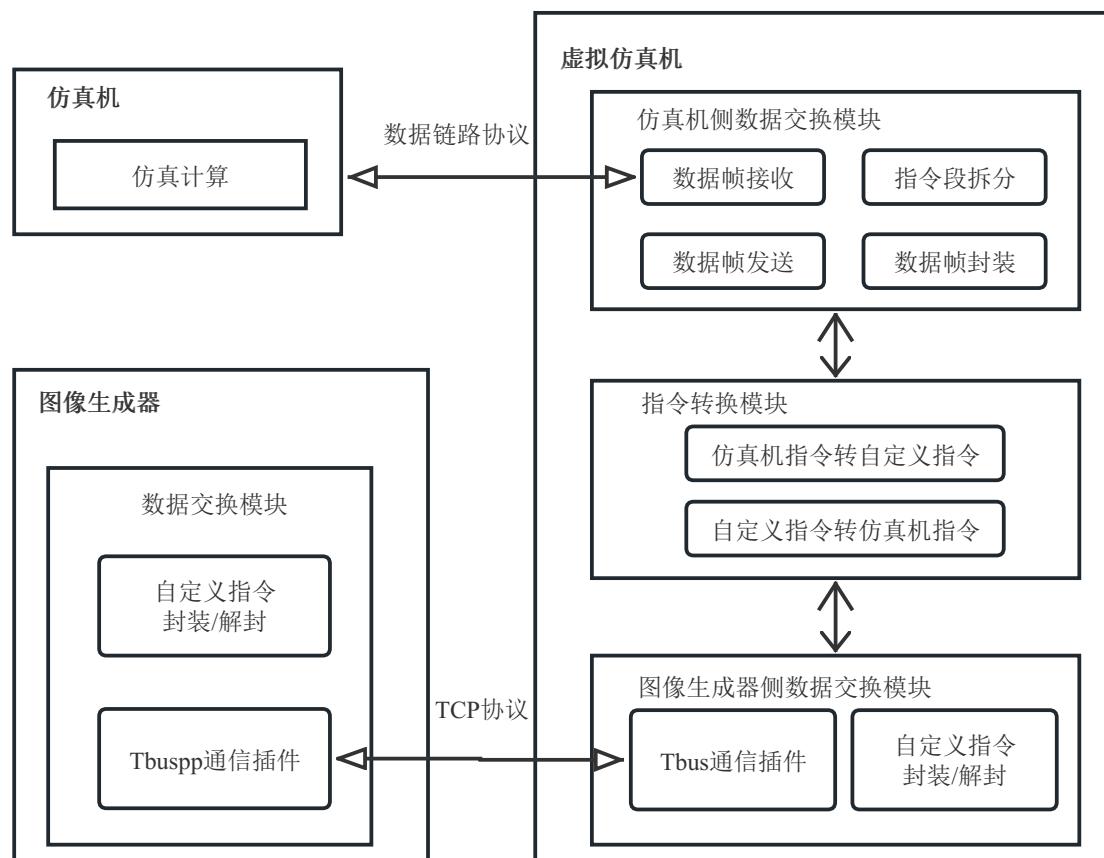


图 3-11: 数据交换子系统架构图

3.4.1 系统逻辑视图

逻辑视图以用户为中心，以类图的形式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 3-12 为数据交换子系统的逻辑视图。SimNodeContext 代表仿真机与虚拟仿真机间沟通的环境，其中依赖 MacLinkLayer 类，负责建立维持底层网络的连接，兼顾数据帧收发过程的实现；MacReceive 和 MacSend 分别表示虚拟仿真机接收和发送数据帧的线程，MacReceive 线程会持续运行，当 MacLinkLayer 收到数据帧后，该线程会立即对其进行处理，再交给 Convertor 模块转换。MacSend 则会在收到 Convertor 的数据后进行封装，再交给 MacLinkLayer 执行发送。

Convertor 表示指令转换模块。仿真机和图像生成器都无法直接使用对方生成的指令信息，因此需要该模块对双方的指令按照规则转换。IProtocolConversion 是一个模板类，Convertor 通过实例化该模板，表示一对指令间的转换。在初始化时需要完成对于所有 Convertor 的注册。Utils 类中则是一些特殊的数字表示方式转换算法。

ImageGeneratorContext 表示虚拟仿真机与图像生成器间沟通的模块，该模块的通信使用 Tbuspp 插件实现 TCP 通信，FFSDevice 是对图像生成器的抽象，在搭配不同仿真机的情况下会使用不同的 Convertor 集合进行指令转换。IGReceive 和 IGSend 分别表示图像生成器接收和发送消息的线程。当收到来自 Tbuspp 的消息时，会将其交给 Handler 首先解读出指令代号，再使用对应结构反序列化。当有序列化完毕的消息发送时则由 IGSend 写入 Tbuspp 的发送队列。

3.4.2 系统开发视图

开发视图注重描述软件开发过程中实际模块的组织，反映了系统工程的具体实施过程。本系统的开发视图如图 3-13 所示，虚拟仿真机接收来自仿真机这一服务端的信息，其中处理过程分为三层。第一层负责与仿真机交流，使用交流环境到线程实例到具体设备再到链路层连接的结构进行数据帧收发的处理。中间层负责对双方指令的转换，其中使用到策略模式实现繁多指令间的转化。第三层负责图像生成器的交流，同样使用环境到线程实例到设备再到具体连接的结构进行开发，图像生成器中另外含有使用数据的逻辑部分。

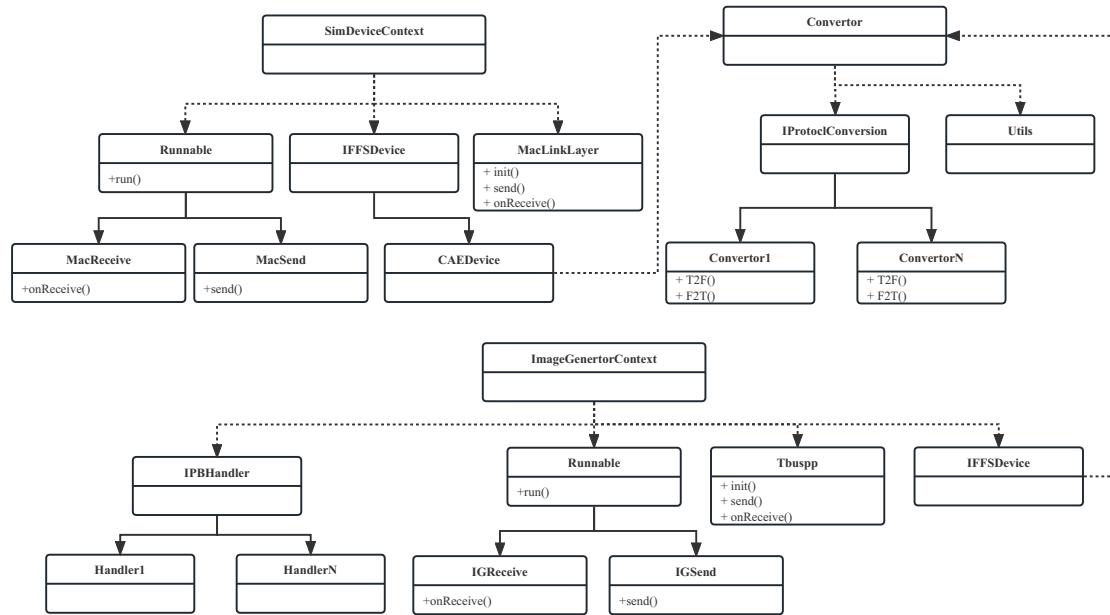


图 3-12: 逻辑视图

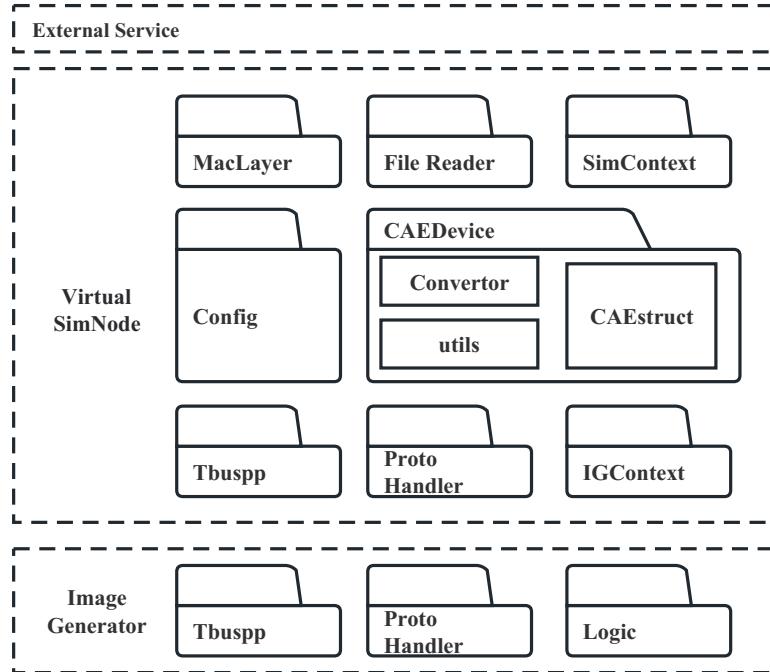


图 3-13: 开发视图

3.4.3 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3-14是本系统的进程视图，主进程启动时会先对各种交流环境初始化，对指令转换类进行注册，一切妥当后便可以接收来自仿真机的数据帧。收到消息后虚拟仿真机会立刻请求指令转换，生成自定义指令，发送给图像生成器进程。图像生成器收到信息后立即对消息解封并反序列化，交付对应的逻辑处理。逻辑进程完成计算后可能会产生一些反馈指令，图像生成器将指令封装后反馈给虚拟仿真机，再请求转换为仿真机指令后便可发送给仿真机。当然这之中逻辑线程的计算结果会提交给渲染进程完成画面生成，一般来讲逻辑进程比渲染进程快一帧。

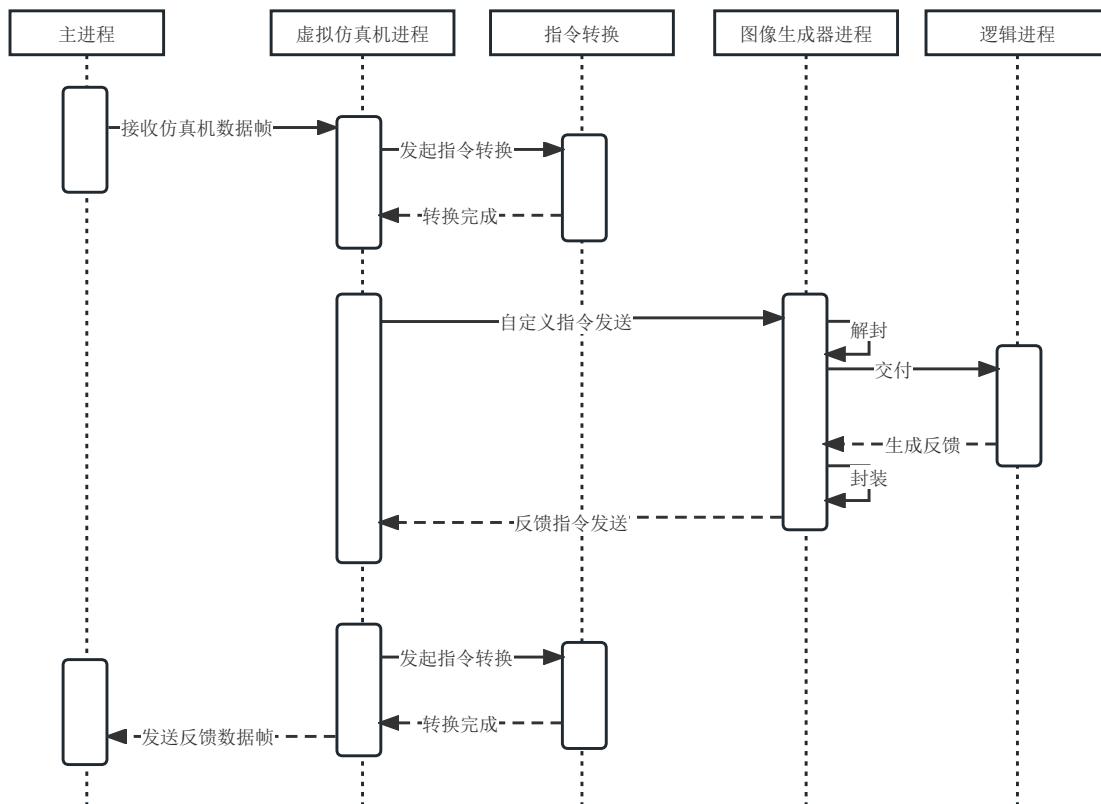


图 3-14: 进程视图

3.4.4 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和各组件之间的连接方式等，又称为物理视图。图 3-15 是本系统的部署视图。飞行员在模拟座舱中操作被仿真机记录并进行仿真计算产生指令，通过以太网协议封装后发送给虚拟仿真机。指令的解析和转换在运行虚拟仿真机的机器内完成，通过 TCP 协议发送给图像生成器。图像生成器收到 TCP 消息后，根据解析的数据执行逻辑，完成场景的生成和渲染。其产生的反馈信息会沿原路径回到仿真机。

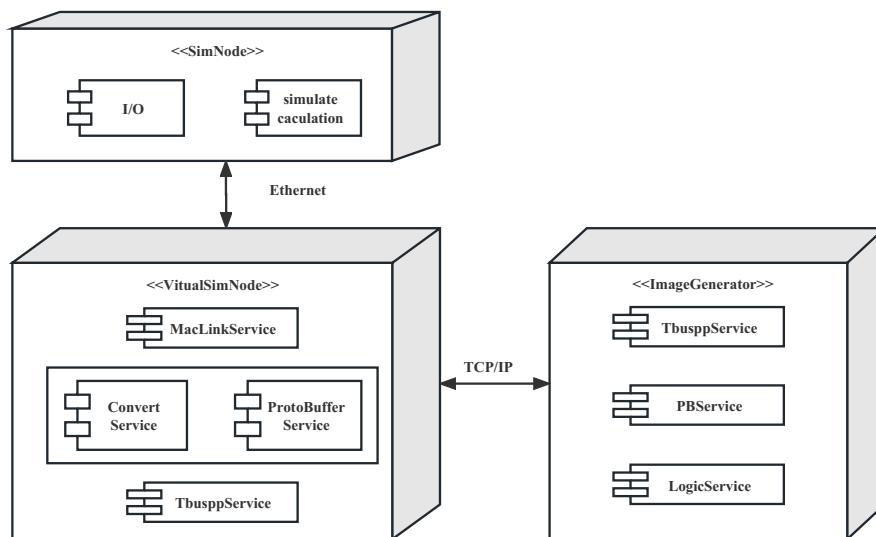


图 3-15: 部署视图

3.5 本章小结

本章开篇对于视景系统的工作方法做出说明。根据说明可知本项目需要将视景系统接入一个不清楚接口的仿真机设备，于是通过结合文档和分析网络流量的方法解读出仿真机的交流协议。明确了接入方法后，便可以按照软件开发的一般步骤进行。首先，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，其次，通过用例图和用例描述表的形式给出场景说明。最后，本章以系统开发 4+1 视图的形式给出了数据交换子系统的架构设计。

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

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。仿真机侧发送出的数据帧仅被以太网协议封装过，即除去数据内容外，仅含有以太网协议头。此种数据帧如果交给虚拟仿真机的网络协议栈处理将不能得到正确内容。因此在实现中虚拟仿真机需要绕过网络协议栈直接接手数据帧自行处理。该模块会将收到的数据帧解封，并拆解为多条仿真机指令供后续使用，或者将多个仿真机反馈指令粘合，并自行添加以太网协议内容封装为数据帧发送给仿真机。

4.1.1 流程图

本模块的主要执行流程如图 4-1 所示。流程分为接收消息和发送反馈消息两部分。虚拟仿真机在接收到原始数据帧后首先取得头部中的指令部分总长度信息，跳过头部后，根据每条指令的长度信息将粘合在一起的多条指令进行拆分，交付给转换模块。当总长度读取完毕后，数据帧的接收流程便结束。此过程中可能遇到暂时没有验证的仿真机指令，将其暂时丢弃即可。

反馈消息流程中，该模块会收到来自指令转换模块的仿真机指令，其需要将这些指令粘合并为它们添加总长度、以太网协议头等信息构建成一个数据帧，最后将其写入发送队列结束流程。

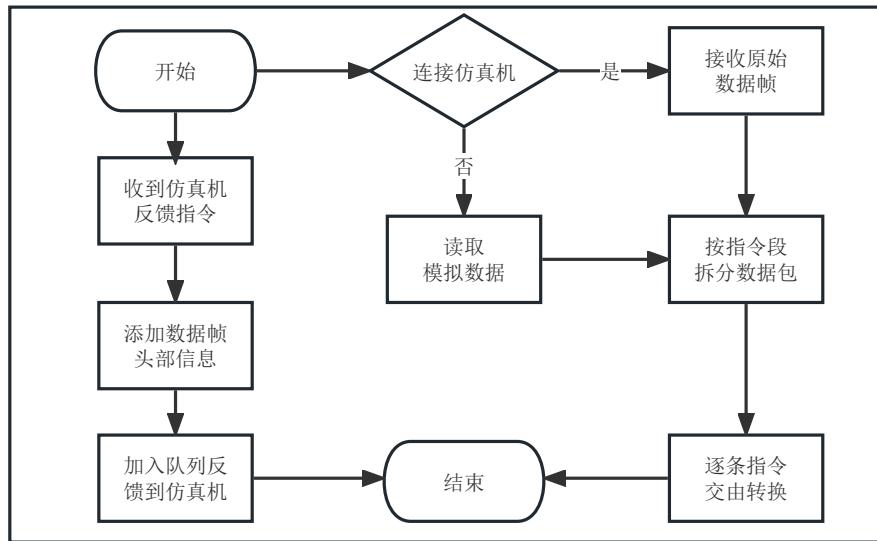


图 4-1: 仿真机侧数据交互流程图

4.1.2 核心类图

本模块的核心类图如图 4-2 所示。其中的核心是类 `SimulationDeviceContext`，它表示仿真机与虚拟仿真机的交流环境。`MacLinkCommunication` 类负责初始化与某一网卡设备的侦听关系，并负责数据帧的实际收取和发送。`MacReceivingRunnable` 是处理信息收发的线程实例，收取到的数据帧会加入接收数据帧的队列 `MacReceivingQueue`，线程实例从中取数据并交由指令转换模块。当需要反馈时，线程实例会调用设备实例中的发送方法，最终通过 `MacLink` 实现发送。`ISimulatorDeviceInterface` 代表仿真机设备类的接口，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。`CAEGenericSimulatorDeviceBase` 是该接口的一个实现，其代表 CAE 公司的仿真机设备，其中含有解读并转换该设备指令的具体算法。当需要接入其他公司设备时，只需要重新实现 `SimulatorDevice` 接口。

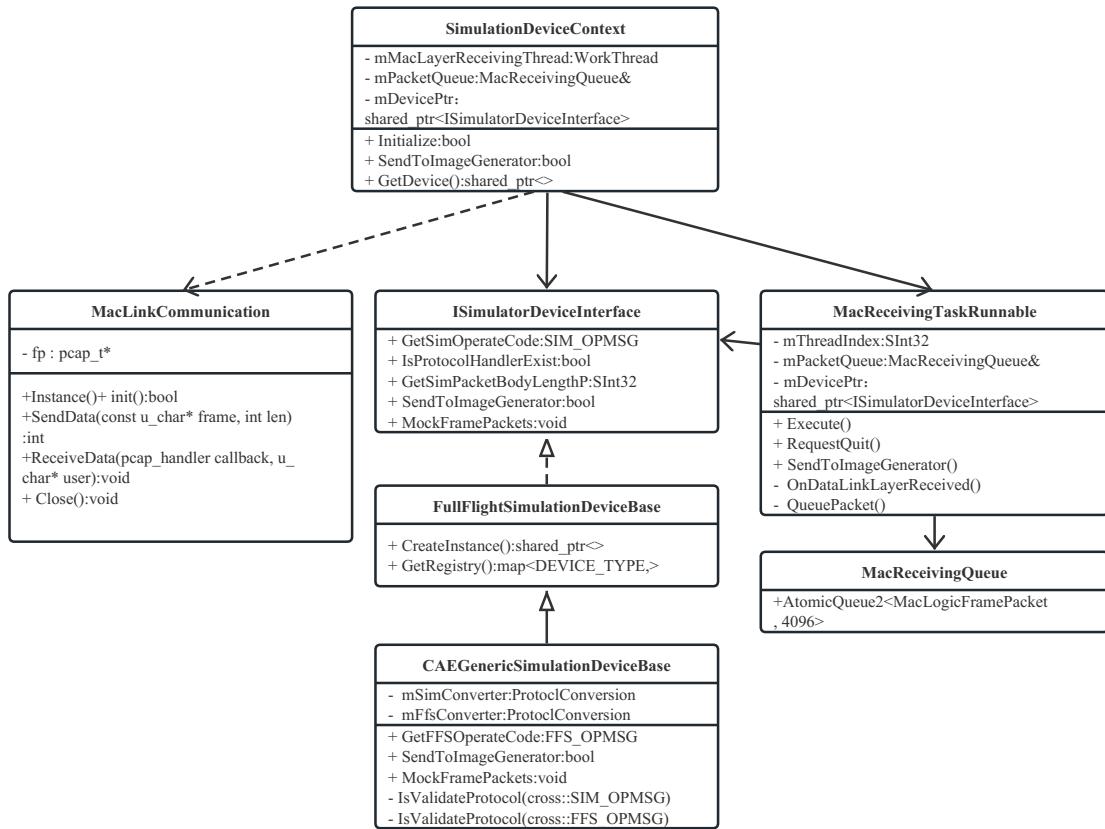


图 4-2: 仿真机侧数据交互核心类图

4.1.3 顺序图

图 4-3 是仿真机侧数据交换模块的顺序图，描述了仿真机与虚拟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 MacLink，即开始侦听对应网卡设备，MacLink 类中利用 WinPcap 实现侦听，并判断是否侦听成功。成功建立连接后，便可以进行数据的收发。交流环境线程 MacTaskRunnable 运行后会循环执行 OnMacReceived 方法，当有数据到达时会对其进行解封处理并加入 MacQueue 队列等待指令转换。当有消息需要反馈时，交流环境调用线程实例的 send 方法，线程实例调用具体设备实例的 send 方法，而设备实例最终通过 MacLink 对象实现发送。

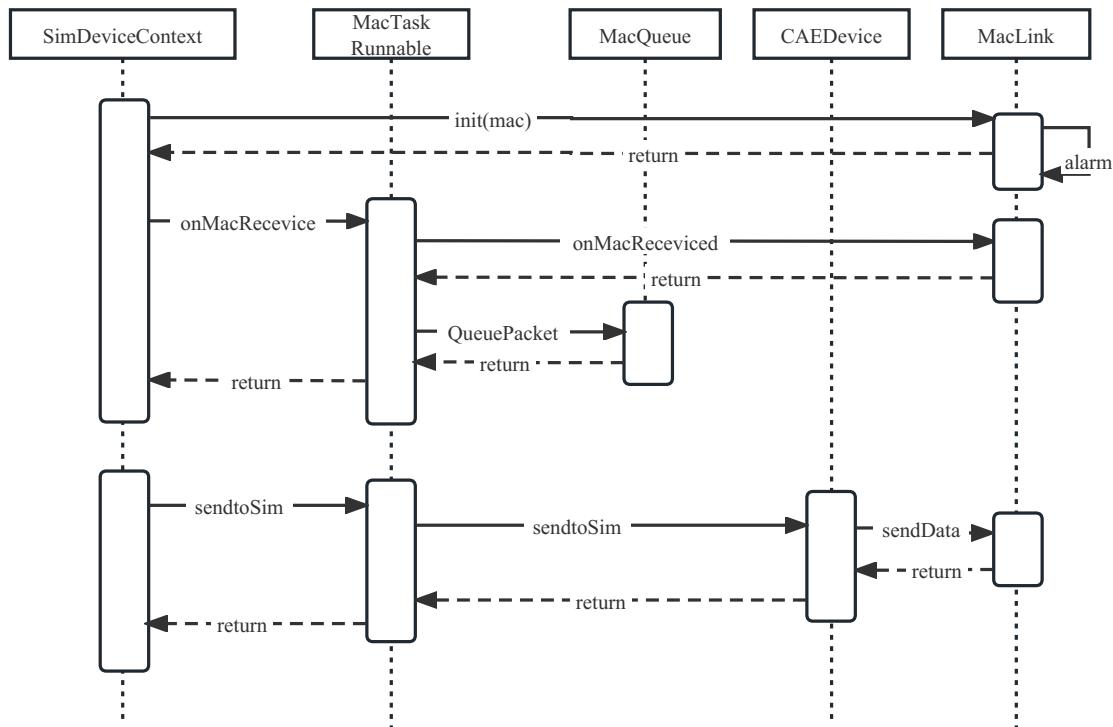


图 4-3: 仿真机侧数据交换顺序图

4.1.4 关键代码

图 4-4 中是虚拟仿真机接收数据帧时的代码。当有数据包达到时，OnDataLinkLayerReceviced 会作为回调函数被调用。参数 pkt_data 是收取到的数据流指针，类型为 `u_char*` 表示要按字节读取数据流中的信息。outPackets 是该方法的输出结果，一个数据帧中可能含有多条仿真机指令，这些指令所在的数据段被拆分后放入集合 outPackets 中。

临时变量 cursor 指向数据流中当前正被处理的字节，其初始指向位置与 `pkt_data` 相同。首先应该跳过数据帧的头部，如第三章第一节中提到的数据帧组织结构，数据帧头部含有以太网协议头、CAE 头等信息，通过 `ParseMacLinkLayerHeader` 方法保留这部分头部信息，同时跳过对应的长度，使指针 cursor 来到数据帧中的指令部分。

数据帧头部信息中含有该帧除去头部的长度，使用临时变量 `leftBytesToRead` 保存。当其仍大于零时，说明还有仿真机指令没有读取。每条指令的头部特定位置也有单一指令的长度，是以 4 字节为单位，`curPacketLength` 计算了该指令以字节为单位的长度。有了这些信息，便能够以 cursor 为起点，

curPacketLength 为长度拷贝指令段到一个 packet 中。这样就完成了一个仿真机指令的拆分，当读取完整个数据帧后，虚拟仿真机便完成了接收数据帧的过程。

```
/* Callback function invoked by libpcap for every incoming packet */
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);

    SimToIGLinkLayerHeader macDataLinkHeader;
    cursorOffset += ParseMacLinkLayerHeader(cursor, macDataLinkHeader);
    cursor += cursorOffset;

    // Parse packets
    SInt32 leftBytesToRead = macDataLinkHeader.byte_to_send_count;
    auto currentFramePacketPtr = &outPackets.emplace_back();

    UInt32 frameMsgNumber = macDataLinkHeader.vis_msg_number;
    currentFramePacketPtr->MessageNumber = frameMsgNumber;
    while (leftBytesToRead > 0)
    {
        if (currentFramePacketPtr->is_full())
        {
            currentFramePacketPtr = &outPackets.emplace_back();
            currentFramePacketPtr->MessageNumber = frameMsgNumber;
        }

        auto curPacketLength = (*(reinterpret_cast<SInt16 const*>(cursor + 2)))
            * sizeof(SInt32) + sizeof(SInt32) * 2;

        SInt8* packetBuffer = new SInt8[curPacketLength];
        MacEncodingPacket packet(packetBuffer, curPacketLength);

        memcpy(reinterpret_cast<SInt8*>(packet.Buffer.get()), cursor, curPacketLength);
        currentFramePacketPtr->push_back(std::move(packet));

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

图 4-4: 接收数据帧代码

来自图像生成器的反馈信息最终也会以数据帧的形式给到仿真机。该方法的参数就是一组待发送仿真机指令的引用，每个指令中含有自己的指令代号 opCode 和指令长度 packetLength。首先需要将全部的仿真机指令粘合在一起，存储在 generator_body 中。期间需要对指令长度进行累积计算，totalPacketLength 作为最后整个数据帧头部的长度信息。

完成仿真机指令的粘和后，便需要根据规则构建一个可供仿真机使用的数据帧。首先是添加以太网协议头、CAE 头等头部信息。其次将粘合后的指令集 generator_body 添加在头部信息后。最后还需要添加一个 4 字节的全为 0 的尾部，一个数据帧便构建完成。此数据帧不需要任何数据交换协议的变化，直接以原始二进制的形式通过 WinPcap 最终发出。

```
bool CAEGenericSimulatorDeviceBase::SendToSimulationDevice
    ( vector<cross :: MacEncodingPacket> const& inPackets)
{
    static thread_local SInt8 generator_body[1024];
    static thread_local SInt8 generator_pack [1514];
    static thread_local IGToSimLinkLayerHeader mac_header;

    SInt32 totalPacketLength = 0;
    for ( int index = 0; index < inPackets . size (); ++index)
    {
        MacEncodingPacket macProtocol = inPackets . at (index );
        // checkOpcode
        auto OpCode = macProtocol.opcode();
        if ( ! IsValidateProtocol (OpCode))
        {
            LOG_WARN("invalidate protocol {}H found , ignore it ", OpCode);
            return false ;
        }
        // Make sim packet actual body with ffs array
        paste( this , macProtocol ,&generator_body[ totalPacketLength ] , packetLength);
        totalPacketLength += packetLength;
    }

    // Fill mac data link layer head
    constexpr SInt32 headerLen = sizeof(IGToSimLinkLayerHeader);
    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
    UInt32 constexpr const tail = 0x00000000;
    memcpy(generator_pack + headerLen + totalPacketLength , &tail , 4);

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

图 4-5: 发送数据帧代码

4.2 指令转换模块

指令转换模块负责仿真机指令与自定义指令之间的转换。经过转换后的指令才能被对方理解并使用。首先仿真机指令中对于浮点数的表达方式是由该厂商自行设计的，对于不同用途不同精度要求的数据其数字表示方式均存在差异。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。其次仿真机指令中最重要的无疑是关于飞机位置和姿态的指令，但由仿真机给出的位置是由经纬度高度组成的数据，而图像生成器中使用的坐标系是笛卡尔坐标系，必须经过坐标系的转换才能使用该信息。

4.2.1 流程图

本模块的主要执行流程如图 4-6 所示。当收到来自仿真机的指令段时，数据转换模块先根据指令代号确定指令类型，用对应的仿真机指令结构体实现反序列化，再根据指令代号查找出对应的转换器，该转换器中的转换方法可将该指令转换为对应的自定义指令。当收到自定义指令时，同样根据指令类型查找相应的转换器，使用该转换器中的方法完成转换。

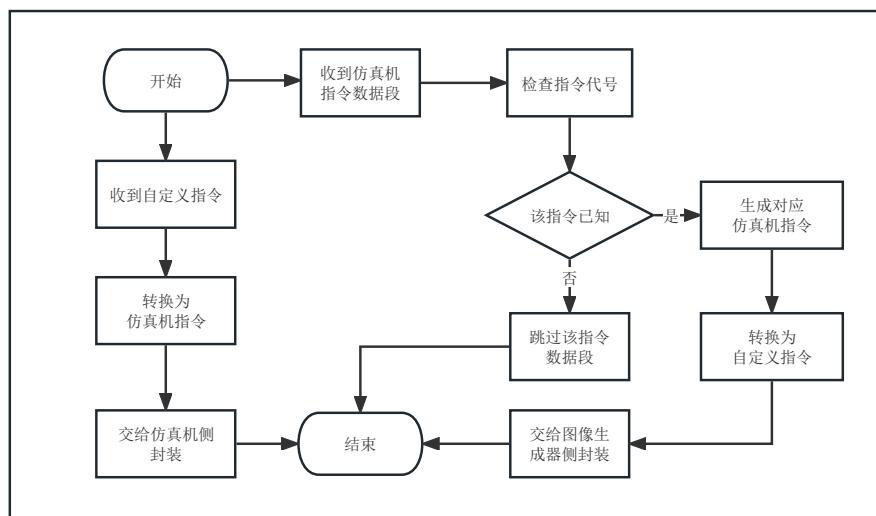


图 4-6: 数据转换流程图

4.2.2 核心类图

本模块的核心类图如图 4-7 所示。其中核心为 `IProtocolConversion` 接口，该接口中声明了仿真机指令与自定义指令间转换的方法 `Convert`。需要注意的是在接口的实现中只需要实现其中一个方向的转换，因为一种指令只可能由仿真机给图像生成器或由图像生成器反馈给仿真机。在 CAE 仿真机类中使用 `map` 存储这些转换器，表示这些转换器是专用于 CAE 仿真机指令的转换，如果需要接入其他仿真机则需要实现对应的转换器。`ProtocolConversion` 是一个模板类，成员属性 `T` 表示一个仿真机指令，成员属性 `F` 表示一个自定义指令，`T` 和 `F` 组成互相转换的一对。`IGCommand` 是自定义指令类，`SimCommand` 是仿真机指令类，真正的 `Convert` 实现在仿真机指令类中。

当调用实例化模板类中的 `Convert` 方法时，该方法会调用对应指令的 `Convert` 方法。这样设计的原因是当增加仿真机指令类型时不用考虑编写对应的转换器，为协同开发带来便利。

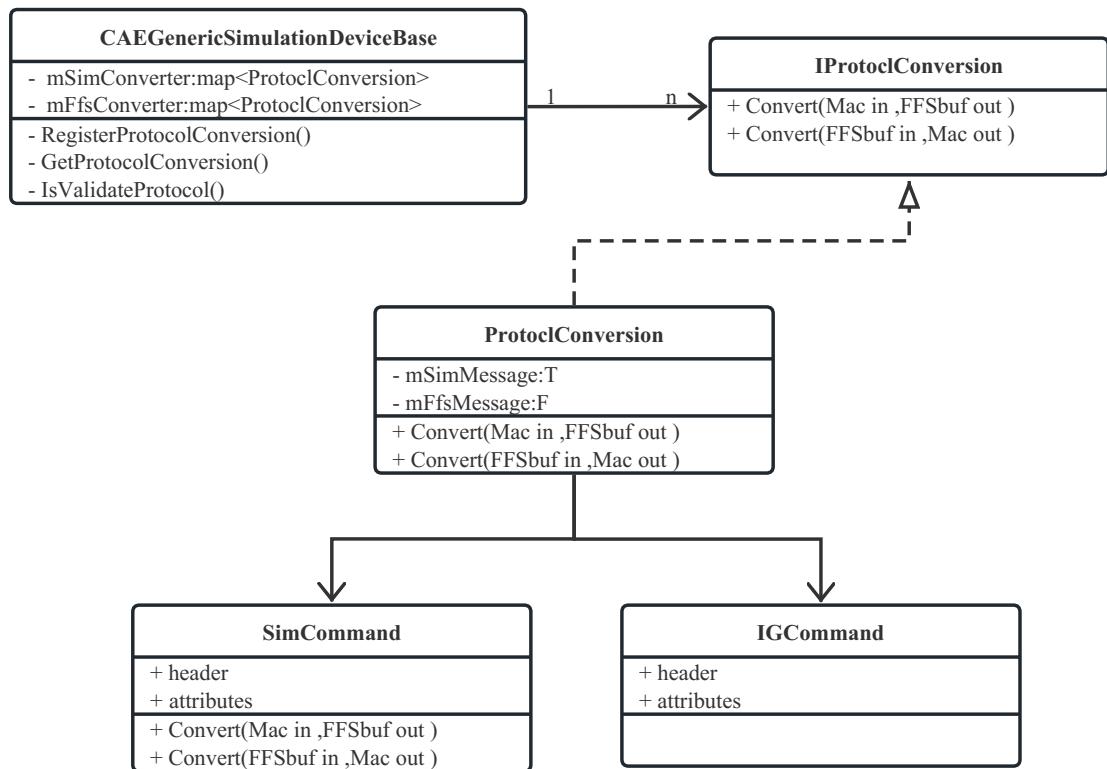


图 4-7: 指令转换核心类图

4.2.3 顺序图

图 4-8 是协议转换模块的顺序图。描述了仿真机指令与自定义指令的转换中各类的交互过程。在系统启动后，CAEDevice 的构造函数会完成所有转换器的注册，即将指令代号与指令转换器作为键值对加入 map 中。在指令转换前，需要先通过指令代号到 map 中获取对应的转换器。调用实例化转换器中的 Convert 方法时，该方法会调用到仿真机指令中的 Convert 实现方法。

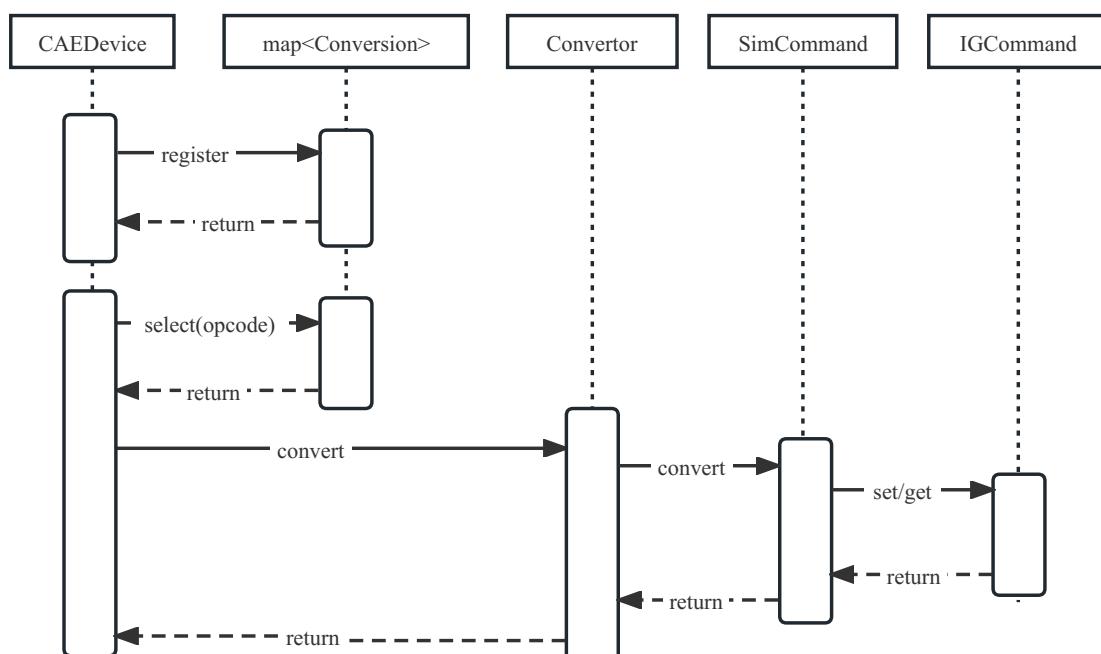


图 4-8: 指令转换顺序图

4.2.4 关键代码

指令转换模块负责仿真机指令与自定义指令间的转换。关于仿真机指令的结构前文已有详细介绍，自定义指令的结构如图 4-9 所示。这是 proto 文件中定义的结构化数据，因为本文使用到二进制交换协议 ProtoBuffer，发送方与接收方都必须清楚结构才能正常序列化与反序列化。最上层的 FFSFrame 结构是虚拟仿真机与图像生成器交流时使用的通用结构，其头部信息中包含发送方、接收方、发送时间等。最后带有一个可重复的字段 packets，代表多个自定义指令。

中间的 FFSPacket 则表示单一的自定义指令，第一个枚举类型代表指令代

号，第二个 bytes 类型的字段 content 则是已经经过序列化的该指令具体内容。接收方可以根据 opcode 信息实现 content 的反序列化。最后则是一个自定义指令的具体案例，AircraftGcsStatus 是用于更新飞机位置和姿态的自定义指令，其中含有当前帧飞机在笛卡尔坐标系中的位置以及旋转姿态。

完成 proto 文件的编写后，就可以用对应的 ProtoBuffer 编译器将该文件编译成目标语言。如果编译为 C++ 后会得到.pb.h 和.pb.cc 文件，分别表示自定义指令的头文件和实现文件，其中提供了一系列的 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对象变成字节序列。对想要读取数据的一方来说，也只需要使用类中的反序列化方法来将这个字节序列重新转换为结构化数据。

```
message FFSFrame {
    required Version          version      = 1;
    required EnumFFSTarget   source       = 2;
    required EnumFFSTarget   destination = 3;
    required uint32           msg_number  = 4;
    required uint64           send_time_us = 5;
    repeated FullFlightSimulatorPacket packets     = 6;
}

message FullFlightSimulatorPacket {
    enum EnumFFSCommand {
        UnknownCommand      = 0;
        AircraftGcsStatus = 1;
        .....
    }
    required EnumFFSCommand opcode = 1 [ default = UnknownCommand ];
    optional bytes          content = 2;
}

message AircraftGcsStatus {
    required double x      = 1;
    required double y      = 2;
    required double z      = 3;
    required double roll    = 4;
    required double pitch   = 5;
    required double yaw    = 6;
}
```

图 4-9: 自定义指令结构

图 4-10是一对指令转换的实现。该方法将仿真机中负责飞机位置和姿态调整的 21H 号指令，转换为自定义指令 AircraftGcsStatus。该对于仿真机指令中的浮点数据一般都需要进行数字表示方式的转换后才能赋值给自定义指令。该指令中比较特殊的一点是，仿真机是通过经度纬度和海拔给出的飞机位置，而

图像生成器中需要使用地心地固坐标系这种笛卡尔系确定位置。其中涉及到坐标系间的转换。而飞机的旋转是在该位置的东北天坐标系下定义，该坐标系三个轴分别是与大地水准面相切的东方，北方以及地表的法线方向。

```
bool SD_PACKET_21H::Convert(SD_PACKET_21H const* inSimPtr, ce::net::AircraftGcsStatus & outFfs)
{
    double latitude = 0.;
    cross :: DecodeUInt.ToDouble(latitude, inSimPtr->latitude_msw, inSimPtr->latitude_lsw );
    double longitude = 0.;

    cross :: DecodeUInt.ToDouble(longitude, inSimPtr->longitude_msw, inSimPtr->longitude_lsw);
    double altitude = 0.;

    altitude = DecodeSInt.ToDouble(inSimPtr->altitude, HMM_M);

    Double3 ecef_pos = WGS84LLA_To_ECEF(longitude, latitude, altitude );

    outFfs.set_x(ecef_pos.x);
    outFfs.set_y(ecef_pos.y);
    outFfs.set_z(ecef_pos.z);

    outFfs.set_yaw(DecodeSInt.ToDouble(inSimPtr->yaw, REV1_DEG));
    outFfs.set_pitch(DecodeSInt.ToDouble(inSimPtr->pitch, REV1_DEG));
    outFfs.set_roll(DecodeSInt.ToDouble(inSimPtr->roll, REV1_DEG) * -1.0);

    return true;
}
```

图 4-10: 指令转换代码

由大地经度，大地纬度和海拔高度组成的 LLA 坐标系，是广泛应用的一个地球坐标系。这里需要说明大地纬度的定义并不是某点同地心连线后与赤道平面的夹角，而是该点地表法线与赤道平面的夹角。如图 4-11 所示由于地球是一个两级略扁的椭球体，这两种定义并不相同。海拔高度的定义也是沿法线的方向到地表的距离。地心地固坐标系 ECEF 是以地心为原点，x 轴指向赤道与本初子午线的交点，y 轴指向北地极，z 轴由手性决定的笛卡尔坐标系。

图 4-12 中的代码给出了经纬度海拔坐标系 LLA 转换为地心地固坐标系 ECEF 的算法。在两种坐标系下地球都是默认为两级略扁的规则椭球体，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 [43]。代码中 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。最后对 z 轴调整手性是为了配合图像生成器中使用的左手系。

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

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

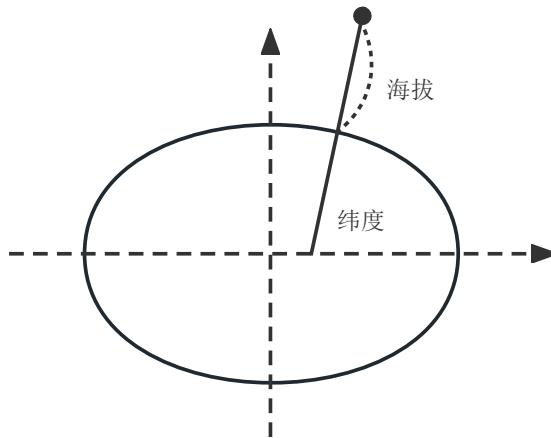


图 4-11: 纬度示意图

```

Double3 WGS84LLA_To_ECEF(double longitude, double latitude, double altitude )
{
    longitude = MathUtils :: Clamp(longitude, -180.0, 180.0);
    latitude = MathUtils :: Clamp(latitude , -90.0, 90.0);

    double lon = MathUtils :: ConvertToRadians(longitude );
    double lat = MathUtils :: ConvertToRadians( latitude );

    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;
    double v = a / sqrt(1 - e2 * sin( lat ) * sin( lat ));

    double x = (v + altitude ) * cos( lat ) * sin(lon);
    double y = ((1 - e2) * v + altitude ) * sin( lat );
    // reverse the z component to translate from right-handed to left-handed
    double z = -(v + altitude ) * cos( lat ) * cos(lon);

    return Double3(x, y, z);
}

```

图 4-12: LLA 转 ECEF 坐标算法

飞机的旋转姿态定义在东北天坐标系下，最终需要将该旋转由东北天坐标系复原到世界坐标 ECEF 下。为此要得到东北天坐标系的三个坐标轴在世界坐标系中的投影。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 ECEF 坐标中的 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。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。飞机的姿态需要使用欧拉角与东北天坐标系形成的旋转矩阵相乘，才能正确得到图像生成器世界坐标系下的姿态。

$$\begin{aligned}\text{normal}.x &= \cos(\text{lat}) * \sin(\text{lon}) \\ \text{normal}.y &= \sin(\text{lat}) \\ \text{normal}.z &= -\cos(\text{lat}) * \cos(\text{lon})\end{aligned}$$

4.3 图像生成器侧数据交换模块

图像生成器数据交换模块负责图像生成器与虚拟仿真机之间的交流。其利用 Tbuspp 插件使用 TCP 协议沟通，数据交换协议则是 ProtoBuffer。此过程中为了对自定义指令添加头部信息构建通用结构，需要对两次序列化，解封过程同样需要两次反序列化。

4.3.1 流程图

本模块的主要执行流程如图 4-13 所示。当收到来自指令转换模块的自定义指令时首先将其进行一次序列化。为了保证接收方了解怎样反序列化，还需要为序列添加指令代号，相当于生成了一个虚拟仿真机与图像生成器间交流的通用指令结构。将此结构再一次序列化后可以加入到 Tbuspp 发送队列中进行发送。而当从 Tbuspp 的接收队列中读取了反馈信息后，需要使用通用结构第一次反序列化得到指令代号，根据指令代再使用对应自定义指令结构对剩余部分反序列化。便可以将得到的自定义指令交给指令转换模块。

4.3.2 核心类图

本模块的核心类图如图 4-14 所示。其中的核心是类 ImageGeneratorContext，它表示图像生成器与虚拟仿真机的交流环境，其中主要的成员是代表信息收发进程的 ImageGeneratorTaskRunnable。该类中的 mAssembler 成员负责在需要发送自定义指令时为其指令添加代号等头部信息，生成通用结构 FFSPacket。多个 FFSPacket 可以粘合为一个 FFSFrame 一起发送。

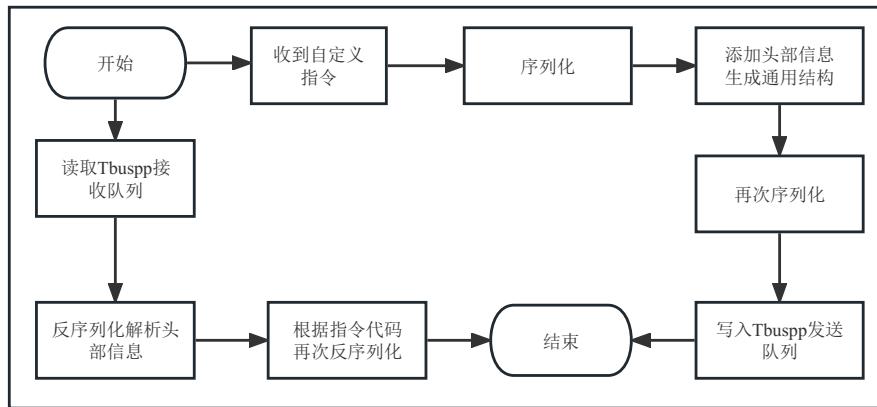


图 4-13: 图像生成器侧数据交互流程图

MsgHandler 则是在图像生成器侧信息接收部分的专属类。不同于虚拟仿真机接收到自定义指令后不区分类别的将其交给指令转换器，图像生成器需要根据不同的指令调用不同的逻辑执行。比如收到控制飞行的指令后，就需要将其中的数据给到飞机实体。具体方法由实现类中的 OnExecutedImpl 实现。

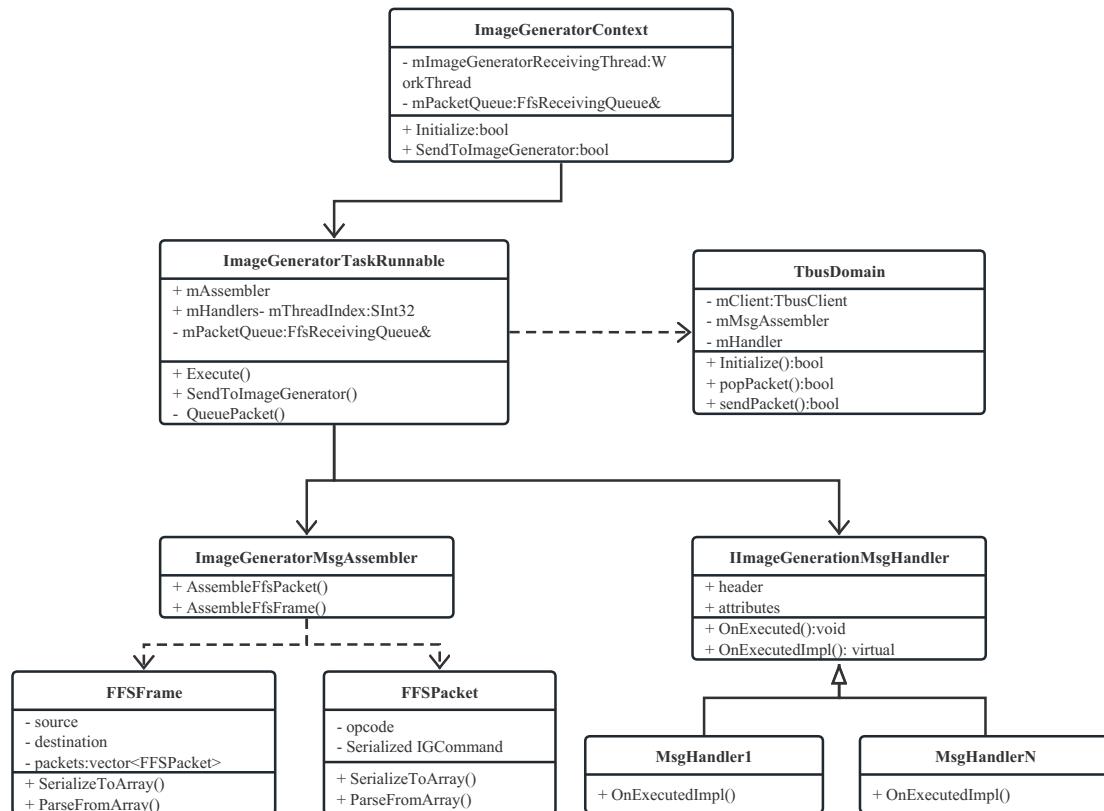


图 4-14: 图像生成器侧数据交互核心类图

4.3.3 顺序图

图 4-15 是图像生成器侧数据交换模块的顺序图。描述了虚拟仿真机与图像生成器间沟通时各类的交互过程。在系统初始化时，IGContext 会首先通过 Tbuspp 的配置文件初始化交流环境。需要发送信息时，交流环境会通过 send 方法将自定义指令交给进程实例，该进程实例会调用 Assembler 中的封装流程获得指令通用结构，最后调用 Tbus 中的发包方法完成发送。接收数据时，进程实例会首先通过 popPacket 读取 Tbus 的接收队列，随后将信息交给 Assembler 类进行解封反序列化等一系列操作，虚拟仿真机中流程到这一步便结束，图像生成器中则还需要将指令分发给对应的 handler 进行处理。

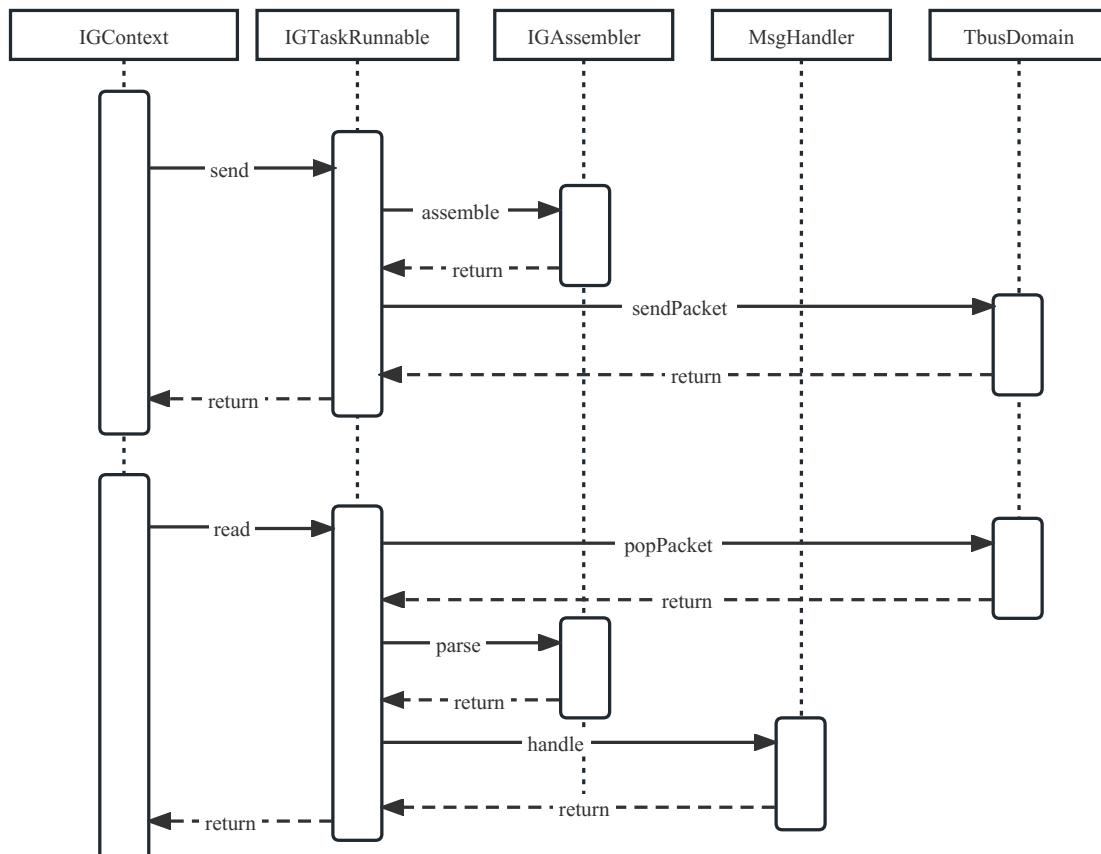


图 4-15: 图像生成器侧数据交换顺序图

4.3.4 关键代码

图 4-16 中的代码展示了将一组自定义指令封装为一个通用结构 FFSFrame 并发送的过程。首先为 outFFSFrame 添加虚拟仿真机的版本信息 version，方便对开发过程中的不同版本进行对比。之后需要将全部的自定义指令逐一进行序列化并添加于 outFFSFrame 的对应字段中。全部添加完成后使用 AssembleFrameHeader 方法为其添加发送方、接受方、发送时间等信息再次序列化。进入消息发送阶段，每一个 Tbuspp 的终端都有唯一的 ID，发送前需要查找接收方的 ID 后通过 QueuePacket 方法将该序列写入对应的发送队列。

```
bool SendToFFSTarget(const vector<FFSPacket>& inPackets, cross :: FFS_TARGET inTarget)
{
    SInt8 packetBuffer[1024];
    cross :: FFSFrame outFFSFrame;

    auto* version = new ce :: net :: Version();
    version->set_major(VsdVersion::Instance().GetMajor());
    version->set_minor(VsdVersion::Instance().GetMinor());
    outFFSFrame.set_allocated_version(version);

    for (int index = 0; index < inPackets.size(); ++index)
    {
        inPackets[index].SerializeToArray(packetBuffer, outLength);
        outFFSFrame.add_packets()>CopyFrom(packetBuffer);
    }

    AssembleFrameHeader(FFS_TARGET::SimulationDevice,
                        inTarget, inSimFrame.MessageNumber, outFFSFrame);

    auto packetLength = static_cast<int>(outFFSFrame.ByteSizeLong());
    outFFSFrame.SerializeToArray(packetBuffer, packetLength);

    std :: string dst_busid = cross :: TbusDomainContext::Instance().GetDstBusid(inTarget);
    TbusDomainContext::Instance().QueuePacket(
        dst_busid, reinterpret_cast<SInt32*>(packetBuffer), outLength);
    return true;
}
```

图 4-16: 自定义指令封装代码

图 4-17 中的代码展示了 Tbuspp 在发送和收取信息时的一些细节。发送函数 QueuePacket 中，参数 busid 表示数据的接受方，inData 是将要发送数据的指针，inSize 是该数据的长度。首先如果发送队列指针为空则说明未正确初始化。之后根据 busid 获得接收方实例，同时配置使用一些默认的发送控制参数 param。在写入队列时，需要保证线程安全，写入前先使用 mutex 为该区域加

锁，随后向 `out_queue` 中写入从 `inData` 开始长度为 `inSize` 字节的内容。

接收消息时，首先定义信息缓冲区 `buffer`，长度为 1024 字节，因为经最初的抓包经验可知不会存在更大的包。通过 `tbuspp_queue_peek` 方法获取到消息的指针 `msg`，和消息的长度 `outSize`。随后将从 `msg` 开始，长度为 `outSize` 的信息拷贝到 `buffer` 完成消息读取。最后将接收队列中的该消息移出队列。

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

    tbuspp_id_t dest = tbuspp_busid_aton(busid.c_str());

    tbuspp_msg_param_t param;
    tbuspp_init_msg_param(&param);

    std::unique_lock<std::mutex> locker(mTbusppDataMutex);
    int result = tbuspp_queue_write(mClient.out_queue_, dest, inData, inSize, &param);

    return result;
}

bool TbusDomainContext::PopPacket(SInt32 const*& outData, UInt32& outSize)
{
    static char buffer[1024] = { 0 };
    if (mClient.in_queue_ == nullptr)
        return false;

    tbuspp_msg_desc_t desc = { 0 };
    auto msg = tbuspp_queue_peek(mClient.in_queue_, &outSize, &desc);

    Assert(outSize <= 1024);
    memcpy(buffer, msg, outSize);
    outData = reinterpret_cast<SInt32 const*>(buffer);

    tbuspp_queue_pop(mClient.in_queue_);
    return outSize > 0;
}
```

图 4-17: Tbuspp 收发消息代码

虚拟仿真机与图像生成器之间通过通用指令结构进行沟通，在接收到通用结构后需要进一步解封获取自定义指令。图 4-18 中给出了图像生成器接收到消息后进行解封并交付使用的过程。首先依旧是从 `tbuspp` 接收队列中读取消息，反序列化后将其中的自定义指令依次加入 `circular_queue` 中，使用自定义指令的逻辑线程会从该队列中拿取指令。

`OnHandleTbusppPacket` 是对自定义指令进一步反序列化并交付使用的过

程。当队列中存在自定义指令时，会根据指令的 opCode 从 handlerMap 中查找对应的处理者，该处理者会调用 OnExecuted 方法会将指令数据给到 mGameWorld 中需要它的实体。

```
int IGClient :: Read()
{
    const char* msg;
    uint32_t size;
    tbuspp_msg_desc_t desc;
    msg = tbuspp_queue_peek(in_queue_, &size, &desc);

    FFSFrame packets;
    packets.ParseFromArray(msg, size);
    for(packet in packets){
        circular_queue_.push(packet);
    }

    tbuspp_queue_pop(in_queue_);
    return 0;
}

void OnHandleTbusppPacket(const TbusHandler& handler)
{
    FFSPacket packet;
    while (circular_queue_.try_pop(packet))
    {
        auto opCode = packet.get_opCode();
        if (mFfsPacketHandlerMap.find(opCode) == mFfsPacketHandlerMap.end())
        {
            auto sourceStr = targetDescriptor->FindValueByNumber(source)->name();
            LOG_ERROR("No handler found with opcode {} from {}", opCode, sourceStr);
            return ;
        }
        auto& handler = mFfsPacketHandlerMap[opCode];
        handler->OnExecuted(data, size, source, destination, mGameWorld, engineFrameCount);
    }
    return 0;
}
```

图 4-18: 通用结构解封代码

4.4 本章小结

本章在需求分析及概要设计的基础上，对系统包含的仿真机侧数据交换模块、指令转换模块和图像生成器侧数据交换模块的详细设计与实现做了阐述。详细设计通过流程图、核心类图和顺序图说明了运行过程中各个类之间的交互逻辑，实现则是通过解读关键代码的方式进行说明。

第五章 系统测试与优化

5.1 初步运行测试

经过第四章中设计与实现的内容后，本文得到了一个可以连接仿真机和图像生成器并转换双方指令的数据交换子系统。为测试系统在真实 FFS 环境下的表现，需要去到飞行训练基地接入专业设备进行初步的功能测试。

5.1.1 测试环境

为使用真实仿真机设备，项目组去到了南方航空珠海训练基地。此处是亚洲最大，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。

该测试环境中各设备布置方式如图 5-1 所示。虚拟仿真机需要配置两张网卡，一张通过交换机连接仿真机，另一张通过交换机连接三台图像生成器。三台图像生成器各自连接一台投影仪才能实现对模拟座舱前方球幕的融合投影。

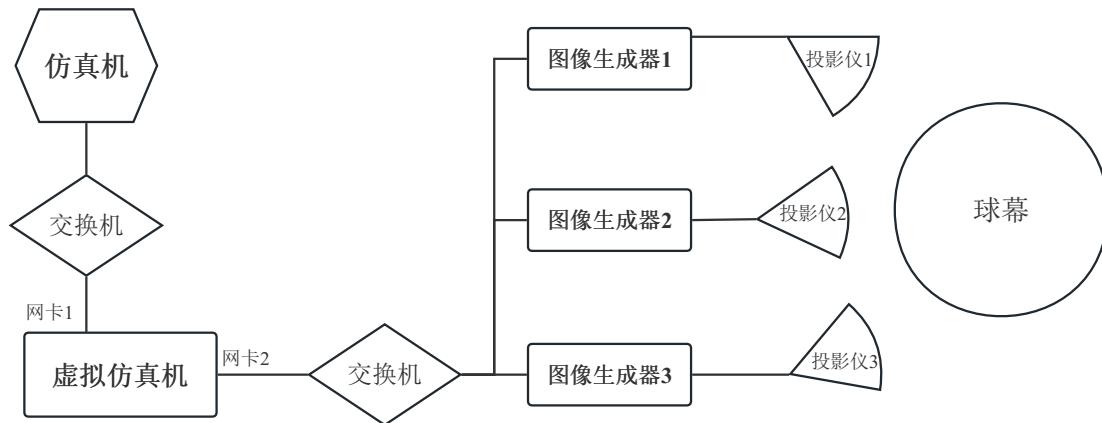


图 5-1: 测试环境设备布置

此环境下虚拟仿真机运行于一台机器上，三台图像生成器运行在配置完全相同的三台机器上，它们通过网线连接交换机。为了顺利运行视景系统，运行

设备上需要配置系统运行的软件环境，详细信息如表 5-1 所示。虚拟仿真机和图像生成器双方都需要 C++ 与 Python 的运行环境，且双方需要通过 Tbuspp 插件沟通。在虚拟仿真机中，需要使用 WinPcap 软件侦听流经网卡的数据包。测试环境部署配置完成后应当能保证系统不会因为软件环境问题无法运行。

表 5-1: 软件配置表

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

在硬件配置方面，运行图像生成器的机器基本配备最新且性能最强的配件，保障渲染帧率能跟上仿真机产生数据的频率。测试环境下的具体硬件配置信息如表 5-2 所示。测试环境中的网络完全隔绝外部网络，网络环境非常良好，网络时延在 1ms 内，网络抖动几乎为 0。也意味着如果测试中出现问题，网络环境并不是一个首先需要考虑的因素。

5.1.2 功能测试

对于本数据交换子系统的功能测试依照指令代号进行。第三章中提到目前已经验证了 21 种仿真机指令，我们让飞行员和教练员在模拟座舱中进行操作依次产生这些指令，如果投影画面产生了对应指令代号的变化，则说明数据交换子系统对于该类指令的接收、转换、发送、分配使用的过程顺利，可以通过测试。表 5-3 给出了对于仿真机指令代号为 21H 指令的测试用例，该指令负责控制飞机的位置和姿态，其他种类指令的测试用例与此表内容类似，不再赘述。

表 5-2: 硬件配置表

设备	配置项	详情
图像生成器	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

表 5-3: 控制指令 21H 测试用例

ID	TC1
测试名称	仿真机 21H 指令控制测试。
待测功能	图像生成器能接收并使用转换后的 21H 指令。
测试步骤	<ol style="list-style-type: none"> 1. 飞行员通过操纵杆让飞机运动。 2. 仿真机生成 21H 指令。 3. 虚拟仿真机转换其为自定义指令。 4. 图像生成器收到指令分配给对应处理逻辑。
预期结果	投影画面中的飞机按飞行员的操作运动。

图 5-2展示了投影画面中的飞机根据飞行控制指令沿跑道起飞的过程。证明数据交换子系统对该指令的处理流程通过测试。

图 5-3展示了投影画面对时间变换指令的响应效果。模拟场景中精确到一天中的每一分钟都可以由时间变换指令控制，图中是将时间调整到黄昏时分的环境效果。指令被正确应用，数据交换子系统对时间变换指令的处理流程通过测试。



图 5-2: 飞行控制指令测试结果

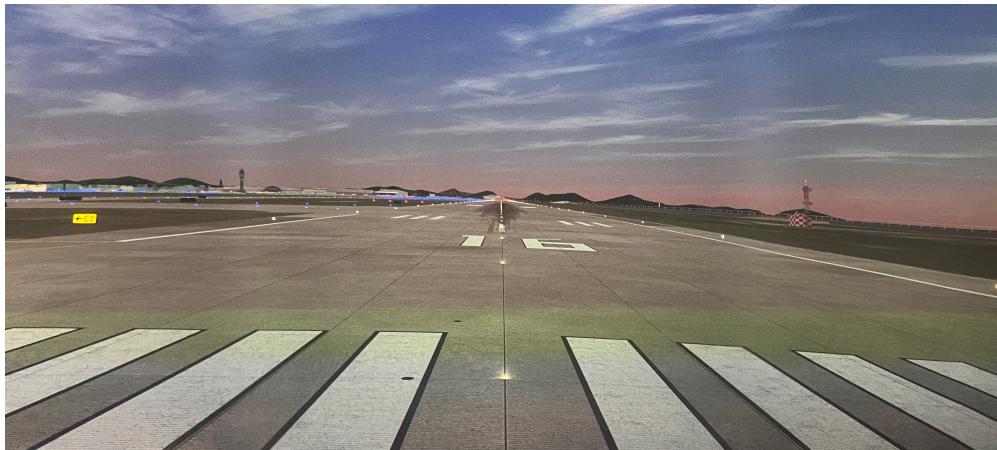


图 5-3: 时间变换指令测试结果

图 5-4展示了投影画面对灯光控制指令的响应效果。飞机身上诸如滑行灯、机翼灯、防撞灯等多种灯光的开关控制均由灯光指令控制。图中是飞机在夜间开启滑行灯后的视景效果。指令可以正确应用，数据交换子系统对灯光指令的处理流程通过测试。

图 5-5展示了投影画面对天气控制指令的响应效果。飞行过程中的雨雪天气及等级都由天气指令控制，图中展示了视景中大雪天气的效果，视野中不仅有雪花的加入，能见度也大大降低。指令被正确应用，数据交换子系统对天气指令的处理流程通过测试。

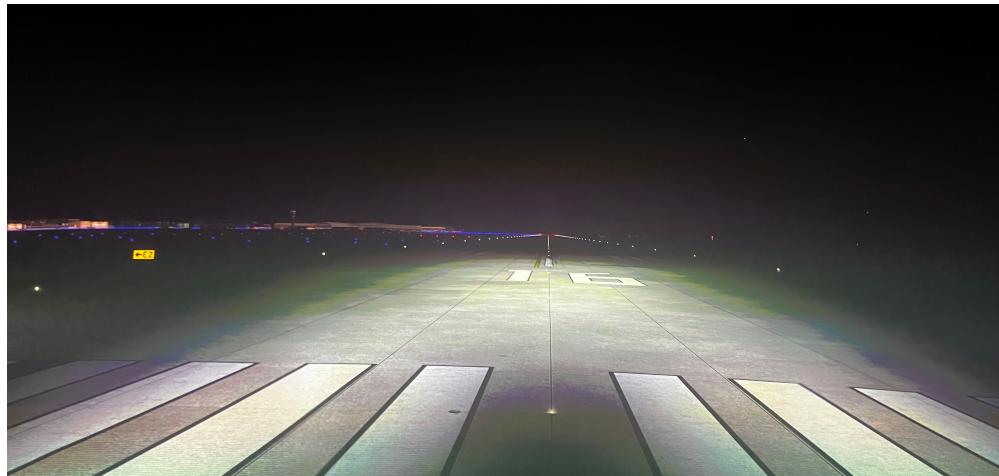


图 5-4: 灯光指令测试结果



图 5-5: 天气指令测试结果

当然已通过验证的仿真机指令远不止这些，此处不再逐一列举测试情况。飞行过程中会产生一些需送回仿真机的反馈指令，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。测试本文对比了图像生成器构建的自定义指令与从仿真机侧捕获到的仿真机指令中的内容。表 5-4展示了两方数据的对比结果，仿真机对于高度使用的单位是 0.5mm，经换算后与自定义指令一致。说明数据交换子系统可以正确处理该反馈指令。

表 5-4: 反馈信息对比

起落架编号	自定义指令	仿真机指令
1	3.822m	7644
2	3.787m	7574
3	3.787m	7574
...
1	40.031m	80062
2	39.914m	79828
3	39.912m	79824

5.2 数据同步机制

5.2.1 问题分析

本次测试中同样发现了一些问题。最具代表性的问题如图 5-6 红框中所示，发现在飞行的过程中融合投影的图像中存在撕裂现象。此处说的图像撕裂并非传统的同一个显示设备的图像上下两部分的撕裂，因为采用了投影仪垂直同步也不会产生这种撕裂。此时的撕裂是在两台投影仪的交界处存在左右两部分撕裂，即两台设备各自渲染的部分图像并不能完美拼接在一起。且就整体画面而言，偶尔会出现肉眼可观察的顿挫现象，即存在跳帧的问题。

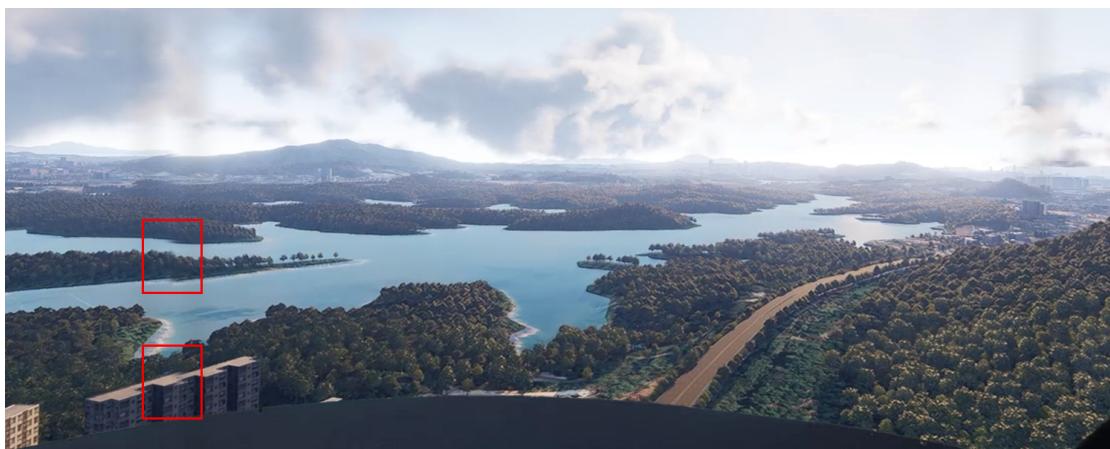


图 5-6: 画面撕裂情况

对于此现象首先考虑是否投影图像本身就无法拼接，即投影仪校准或多台图像生成器各自渲染时的投影矩阵存在问题，但比较明确的是在飞机静止时并不会出现如上的画面撕裂问题，这证明并非图像本身无法拼接。由此便仅有一种可能性，便是多台投影仪的投影图像并非同一帧的内容，两侧的图像间产生了差距所以不能拼接。经过截帧查看数据后发现它们的计算数据确实存在不同。图 5-7 展示了投影图像不能及时更新的原因。图像生成器的逻辑帧需要收到投影仪 V-Sync 信号后才能执行，但在本场景中逻辑帧的执行要依赖于从仿真机发送来的指令内容，如果此时指令并未送达便会错过这一更新周期，导致投影图像不会改变。如果其他的图像生成器即时获得了指令，则会正常更新图像，便会领先一帧的内容，产生图像撕裂。

如果一台图像生成器多次错过更新周期则会导致跳帧，即缓存中的某些帧未经投影直接被覆盖，产生视觉上的顿挫感。

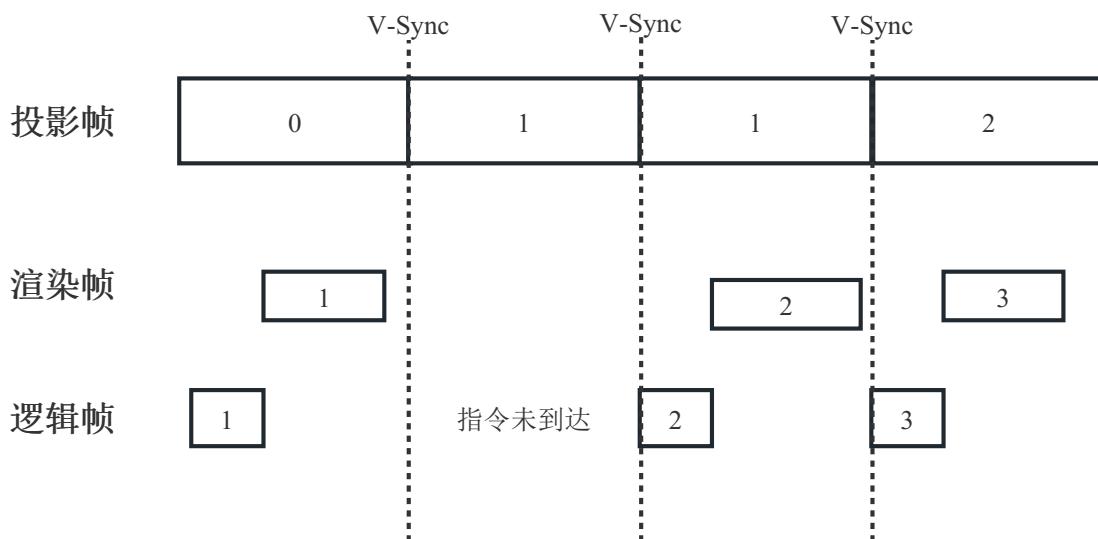


图 5-7: 更新周期的丢失

接下来就需要排查指令不能及时到达图像生成器的原因。前面提到机房的网络环境十分优秀，网络波动基本可以排除。结合整个指令流动过程，可能使指令到达频率不稳定的地方有二，一是发送给多台图像生成器时存在时间上的差距，二是虚拟仿真机接收仿真机指令的频率不稳定。其中，第一点的原因比较明确，因为虚拟仿真机与图像生成器间采用 TCP 通信，TCP 不支持广播所以无法同时向多个图像生成器发送相同的指令数据。轮询发送的结果是最后一台机器收到数据的时间大约慢 5 毫秒，对于 60Hz 的刷新率来讲该延迟已占据一

帧中三分之一的时间，极有可能令某些图像生成器错过更新周期。

对于第二点的检查则监控了虚拟仿真机接收数据帧的间隔时间，图 5-8 展示了 3600 条日志记录，即理论上 60Hz 频率下 1 分钟的统计结果。从中发现在仿真机以 60Hz 的频率产生数据帧的情况下，虚拟仿真机侧从网卡接收数据的时间间隔成钟形分布，总体而言平均间隔时间接近 16.67 毫秒。但同时看出即使 WinPcap 开启了立即转发模式，虚拟仿真机每分钟也会产生约 10 次 3 毫秒以上的接收延迟。当然这个延迟对于多台图像生成器而言是统一的，可能会同时错过这一帧的更新进而产生顿挫。

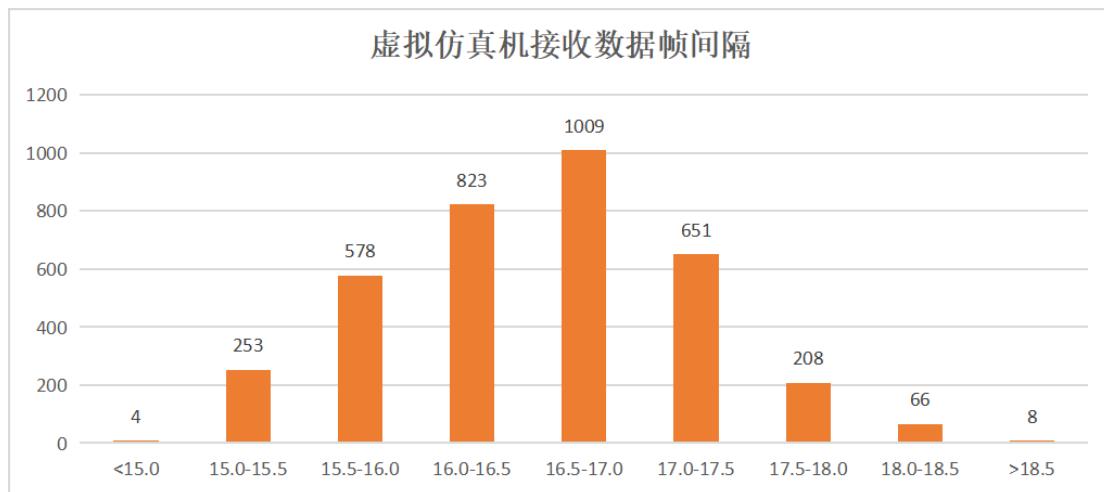


图 5-8: 虚拟仿真机接收间隔

5.2.2 网络帧缓冲

从以上分析可知，造成问题的原因是在图像生成器需要使用指令数据时，接收队列中可能并没有准备好的数据供使用。如果在图像生成器侧准备一个对于指令的缓冲，当队列中的指令达到一定数目时再开始运转，就可以有效缓解这些问题。当然引入缓冲机制后一定会造成操作响应上的延迟，根据官方标准来讲，真实的飞机响应飞行员的操作本就存在延迟，模拟飞行中，要求视景系统可以在这个延迟的基础上于 120 毫秒内产生响应。而在 60Hz 工作频率下，本系统对指令的转换，缓冲加渲染的时间相较而言绰绰有余，增加几帧的缓冲并不影响达到标准。

如图 5-9 所示，如果网络帧缓冲的数值被设为 2，那么图像生成器需要等待当前频率下第三个周期的时刻再开始执行第一帧逻辑。此时的指令队列中一定已存在两个以上的指令内容，此时再使用指令 1 去执行第一帧的逻辑就不会出现指令未到达而错过更新周期的问题。此方法可以同时屏蔽来自 TCP 逐个发送的时间差和虚拟仿真机接收仿真机指令时的频率不稳定。

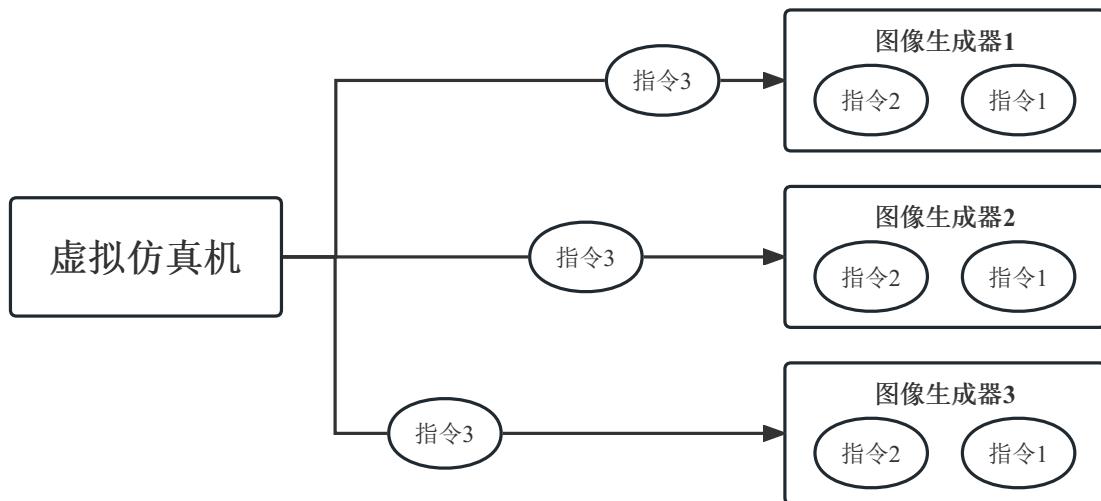


图 5-9: 网络帧缓冲示意图

图 5-10 是网络帧缓冲机制的关键代码，Tbuspp 中接收到的通用结构 packet 会被加入 circue_queue_ 队列中，该结构的头部含有数据发送方信息 source，代码中首先按照发送方将该通用结构分开存放到 received_packets_ 中。之后需要读取 Tbuspp 相关设置中的帧缓冲信息 cachedFrameCount 以决定使用几帧网络帧缓冲。随后便需要对 received_packets_ 中的指令数据进行处理，目前本系统仅处理发送方为仿真机的数据。如果该发送方对应的帧缓存数量还没有超过设定值，那么便不能交由逻辑处理，继续等待更多指令的到来。超过阈值则可以交给对应的 handler 去做处理，并将其从缓冲中擦除。

```

int FetchMsgWithBuffer(const TbusHandler& handler)
{
    static bool is_first_fetch = true;
    ce :: net :: CrossEnginePacket packet;
    while (circular_queue_.try_pop(packet))
    {
        received_packets_[packet.source()].emplace_back(packet);
        received_count += 1;
    }

    UInt32 cachedFrameCount = TbusppSetting::GetTbusppSetting().cachedFrameCount;
    UInt32 latestMessageNumber = received_packets_
        [ce :: net :: EnumFFSTarget::SimulationDevice].back().msg_number();
    for (auto& [source, packets] : received_packets_)
    {
        if (is_first_fetch && source == ce::net::EnumFFSTarget::SimulationDevice)
        {
            if (packets.size() <= cachedFrameCount)
            {
                cached_count += packets.size();
                continue;
            }
            else
            {
                is_first_fetch = false;
            }
            for (auto it = packets.begin(); it != packets.end())
            {
                auto packet = *it;
                handler(reinterpret_cast<const void*>(&packet), latestMessageNumber);
                it = packets.erase(it);
            }
        }
        return 0;
    }
}

```

图 5-10: 网络帧缓冲机制代码

5.3 数据平滑机制

5.3.1 问题分析

加入数据帧缓冲机制后，图像撕裂和顿挫现象基本消失，三台图像生成器可以及时获得指令数据并最终完成投影，但同时又发现在飞机高速运动时，图像的拼接部分出现了持续的抖动现象。图 5-11对该现象做了示意，在图像交界处会在运动方向产生波纹状的持续抖动，在大屏幕上尤其明显。

在引入了帧缓冲机制后，多台投影仪一定会投影同一帧的画面，理论上应该不会再有拼接部分的问题。但是之前忽略了一个问题，虽然多台投影仪拥有相同的刷新率，但是它们每一帧开始的时间可能并不相同。即如图 5-12所示，由于投影仪的刷新时间不同步，会最终影响逻辑帧的开始时间不同。相较于之前图像的撕裂，此问题会导致在一帧以内时间的图像不一致性，且会连续在一

致与不一致的状态间转换，由此在图像交叠处产生抖动效果。其实此问题也会加剧上一节中提及的图像撕裂问题，在加入帧缓冲后以抖动的形式出现。

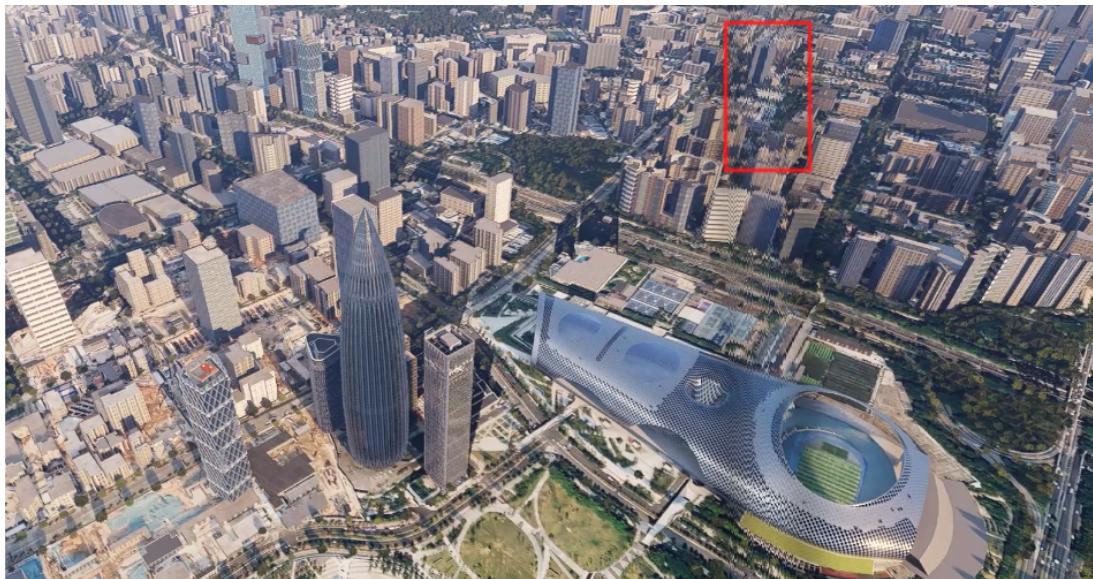


图 5-11: 画面抖动示意



图 5-12: 投影仪刷新不同步

5.3.2 插值平滑

想令多台投影仪同步其实有硬件层面的解决方法，但是需要定制显卡并将多张显卡用专用同步线进行连接，市面上流通的显卡设备并不支持此项功能。此解决方案的购买成本和后期维护成本都比较高。所以暂时先从软件层面进行解决。前文提到，抖动是由于两侧的图像刷新时间不一致导致的。肉眼能够察觉说明飞机在快速运动的情况下，位置变化剧烈，每一帧图像的变动比较大。受到网络帧缓冲机制的启发，如果利用缓冲的两个数据帧进行插值，就可以减少两侧图像的相对变动。

如图 5-13 所示，在不采用插值的情况下，当图像 1 已经变为使用数据 2 的渲染结果后，图像 2 仍然有一段时间是数据 1 的结果，如果这段时间足够长就会产生抖动。在使用插值后，图像 1 变为插值后数据 2.2 的结果时，图像 2 则为数据 1.5 的结果。相当于把数据 1 与数据 2 间的差距为 1 的差距，分为前一帧 0.3 与后一帧 0.7 的差距，这样能够有效缩小抖动的幅度。同时也不会产生明显的图像撕裂。

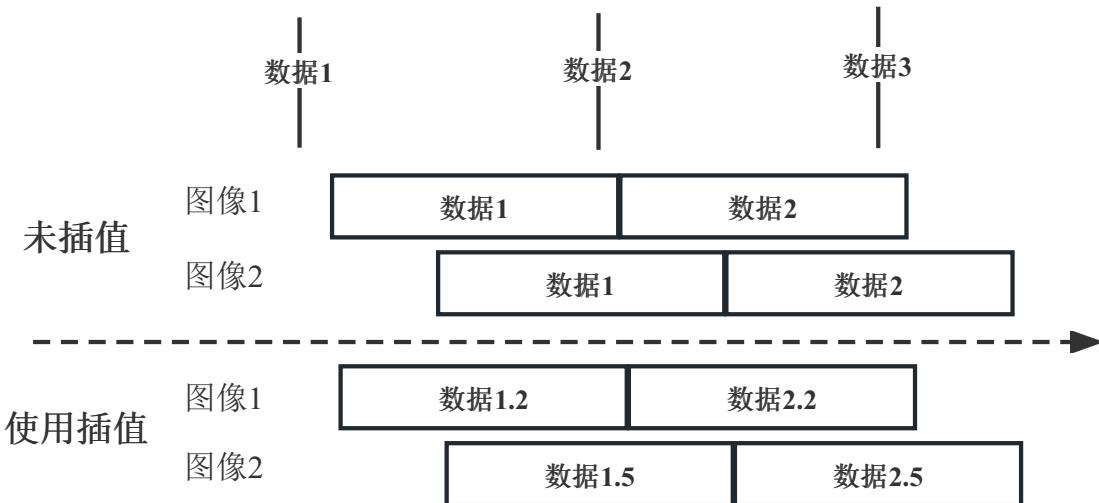


图 5-13: 利用缓冲数据插值

插值的一般表示方法如下方公式表示，根据起点 *from*，终点 *to* 和一个比例 *t* 得到一个位于两者之间的位置。在本场景中起点与终点非常明确，它们就是位于缓冲中前后两个指令中的数据。如何获得每个图像生成器的帧开始时间在两个数据中的相对位置是需要重点解决的问题。

$$\text{Lerp}(\textit{from}, \textit{to}, t) = \textit{from} + (\textit{to} - \textit{from}) * t$$

本系统中有一个十分重要的前提条件，我们认为仿真机产生指令数据的频率是稳定的。如果仿真机工作在 60Hz 的频率下，那么前后两次发送指令的间隔应该为 16.67 毫秒。即使这些指令最终到达图像生成器的频率并不十分稳定，但是由于网络帧缓存机制的存在，依旧可以认定其时间间隔为 16.67 毫秒。如果为每条指令添加一个发送时刻信息，就可以通过频率和缓冲帧数计算它应当被处理的时刻，这个时刻对所有图像生成器都是统一的。每个图像生成器的帧开始时间是真正处理该指令的时刻，使用两个时刻的差值除以 16.67 毫

秒后便是 60Hz 的情况下帧开始时间位于两个指令数据中的位置。

```

void OnBeginFrame(FrameParam* frameParam)
{
    mMessageNumber = frameParam->GetFrameCount();
    UInt64 clockerTime = SyncClocker::GetInstance()->GetClockerTimeStampUs();
    clockerTimeLastFrame = clockerTime;

    int networkFrameRate = tbusSys->GetNetworkFrameRate(tbusComp.Read());
    if (networkFrameRate)
    {
        mNetworkFps = 1000.0 / networkFrameRate; // Unit ms
        UInt32 cachedFrameCount = TbusppSetting::GetTbusppSetting().cachedFrameCount;
        UInt64 theoHandleTime=packet.send_time_us() / 1000.0+cachedFrameCount*mNetworkFps;
        mFrameDelta = (clockerTime - theoHandleTime)/ mNetworkFps;
    }
}

void TickAircraftPosition ( float deltaTime, GameWorld* inWorld)
{
    float frameRemainedDelta = ffsSys->GetContext()->GetFrameRemainedDelta();
    // lerp position
    Double3 firstPositionToLerp = Double3(mX, mY, mZ);
    Double3 secondPositionToLerp = ffsSys->GetContext()->GetAircraftPoistionToLerp ();
    if (!secondPositionToLerp.IsNearlyZero ())
    {
        Double3 lerpedPosition = LerpStable( firstPositionToLerp ,
                                              secondPositionToLerp, networkFrameRemainedDelta);
    }
    // lerp rotation
    Double3 firstRotationToLerp = Double3(mHeading, mPitch, mRoll);
    Double3 secondRotationToLerp = ffsSys->GetContext()->GetAircraftRotationToLerp ();
    if (!secondRotationToLerp.IsNearlyZero ())
    {
        Double3 lerpedRotation = LerpAngle<AngleType::Degree>(firstRotationToLerp,
                                                               secondRotationToLerp, networkFrameRemainedDelta);
    }
}

```

图 5-14: 网络帧缓冲机制代码

图 5-14 中的代码展示了插值的部分过程。OnBeginFrame 是图像生成器中每一帧开始的函数，首先要获取当前时间 clockerTime，这是一个微妙级别的时钟时间。其次需要获取当前工作的频率，用此频率计算每一帧的时间；之后需要从设置中获得网络帧缓冲的数量，使用此指令的发送时间加上帧缓冲数量乘以每帧的时间，就能得到该指令理论上要被处理的时间。使用此方式计算，多个图像生成器中的该值是完全一致的。最后使用当前时间减去理论时间就可以得到每个图像生成器相对的偏移。

有了时间偏移后计算插值就十分容易了，需要注意的是，插值仅适用于飞机位置和姿态这种每帧都会有的连续数据。对于灯光的开关，天气的变化并不

需要插值，因为它们是表示状态或等级的离散数据，且对应指令并不会每帧都存在。

5.4 二次测试

经过以上的优化后，本文对系统再次进行了测试。本次测试中还做出了用光纤转 DP 的方式代替 30 米 HDMI 线连接投影仪，用 3090 显卡代替散热较差会产生降频的 A6000 显卡等变动。这些改变都能提升系统的稳定性。

5.4.1 对照测试

功能测试部分对比了本视景系统与原视景系统在相同操作下的投影图像情况。测试选择广州白云机场，分别进行了昼夜更替，天气变换，灯光控制等指令的测试，并对比了使用同一条预设飞行路径的飞行过程。原视景系统的运行效果如图 5-15 所示，飞机即将晴朗的白天在白云机场 02L 号跑道上完成起飞。图 5-16 则展示了本视景系统的运行情况，可以根据仿真机指令控制产生类似的环境效果，且后续飞行的路线与原视景系统完全一致。测试过程中，融合投影中已经不存在长时间图像撕裂问题；交界处的抖动明显得到抑制，日后可能考虑加入硬件同步机制。目前仍存在球幕上颜色不准确的问题，但对于数据交换子系统而言已经实现了核心功能。

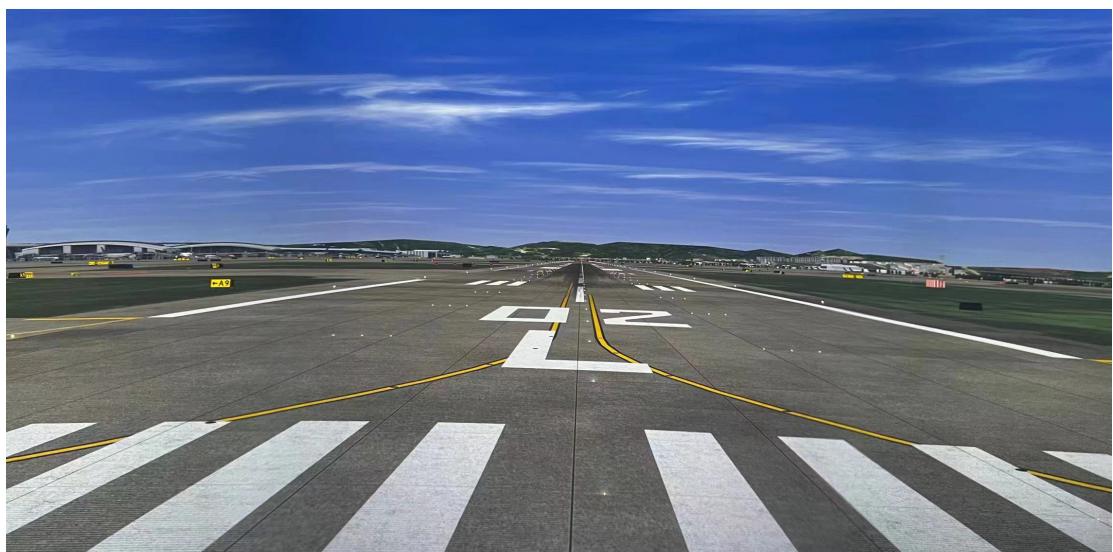


图 5-15：原视景系统运行效果



图 5-16: 本视景系统运行效果

5.4.2 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 30 分钟，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。

表 5-5: 帧率测试结果表

项目	本视景系统	原视景系统
运行时间	1802.015s	1806.228s
总帧数	105418 Frame	108735 Frame
平均帧率	58.5 FPS	60.2 FPS
最低帧率	48.6 FPS	59.2 FPS
最高帧率	60.8 FPS	60.8 FPS
1% Low	38.1 FPS	56.7 FPS

测试结果如表 5-5 所示。本视景系统为了达成 60 帧率的基本要求，在数据传输、图像渲染等多方面上做出改善和取舍，根据结果也可知本视景系统在当前设备的支持下比较吃力的基本达到要求。而原视景系统是在锁定 60 帧率后的运行结果，整个过程非常稳定。1%Low 两方有非常大的差距，说明在某些复杂场景下本视景系统会产生比较剧烈的瞬时帧率波动。这主要是渲染时间较长导致的问题，说明系统中使用的场景和游戏引擎依旧有很大的优化空间。

5.5 本章小结

本章首先对于数据交换子系统处理已核实指令的能力做了测试，根据投影图像结果可知，该系统可以正确转换并传输双方的指令。使用多图像生成器融合投影时先后发现图像存在撕裂和抖动的现象，经过排查分析，引入了网络帧缓冲和插值机制缓解了问题。最终将本视景系统与原视景系统进行了对比，在功能方面两方基本一致，但帧率稳定性仍需提升。

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，突破技术封锁愈发关键，这两年在国家紧锣密鼓的推进下各行业纷纷涌现出惊喜突破。在关系到万家百姓出行的航空领域，我们已经拿出 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。而在飞行训练相关的全动飞行模拟机领域仍有许多空白，能够达到训练标准的视景系统便是其中之一。本文中描述了自主研发的全动飞行模拟机视景系统中的数据交换部分的设计与实现过程，能够让视景系统接入未开放接口的仿真机是主要工作。

在项目开始阶段，本文通过结合文档和网络流量分析的方法，明确了仿真机与外界的交流方式。仿真机中的网络协议栈最高层为数据链路层，其发出或接收的数据帧仅被以太网协议封装。为了让其与位于应用层的视景系统沟通，系统中引入了虚拟仿真机这一角色。其首先充当仿真机侧的网络协议栈，负责接收并解封仿真机的数据帧。其次因为双方都有自己的数字表示方法和数据组织结构，它需要对双方的数据内容进行转换。最后仿真机也负责与视景系统中的图像生成器通过 TCP 消息交流，将仿真机的指令送达或接收反馈指令。在此过程中，为了尽可能提高交流效率，采用了开启立即转发模式、使用 ProtoBuffer 数据交换协议、禁用 Nagle 算法等措施。

系统实现完成后进行了初步的测试，在将已被验证的多种指令给到视景系统后，其均能产生正确的对应效果。但使用多个图像生成器进行融合投影时，发现存在两侧图像撕裂的问题。经排查原因在于多个图像生成器的数据没能同步，由此引入网络帧缓冲机制解决该问题。之后又发现飞机在低空较高速飞行时，图像交界处存在抖动现象，经排查发现问题是个别投影仪的刷新时间没能同步。最终采用对缓冲中的数据插值的方法在软件层面缓解了该问题。

6.2 项目展望

虽然该视景系统已经达成一定的效果，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。因为难以与国外 FFS 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。实现上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。

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] LENDER P H. A New Form of Flight Simulator[J]. British Patent, 1917, 27.
- [7] PAGE R L. Brief history of flight simulation[J]. SimTecT 2000 proceedings, 2000 : 11 – 17.
- [8] ICAO. 飞行模拟机鉴定标准手册 [R]. [S.l.]: 国际民航组织, 1995.
- [9] 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.
- [10] 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.
- [11] HELLINGS E E. A Visual System for Flight Simulators[J]. British Communications and Electronics, 1960, 15(5) : 334 – 337.

- [12] SPOONER A M. Collimated Displays for Flight Simulation[J]. Optical Engineering, 1976, 15(3): 215–219.
- [13] 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/>.
- [14] 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.
- [15] 王龙. 游戏引擎研究与分析 [J]. 软件导刊, 2018(5-7).
- [16] 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>.
- [17] 张宇. 基于虚幻 4 的场景植被系统的设计与实现 [D]. [S.l.]: 南京大学, 2017.
- [18] 杜日旭. 通用飞机飞行模拟器视景系统的研究与开发 [D]. [S.l.]: 沈阳航空航天大学, 2014.
- [19] 董鸿鹏, 王春财, 张波. 飞行模拟器视景系统的设计与实现 [J]. 计算机应用, 2018, 38(A01): 228–231.
- [20] GREGORY J. Game engine architecture[M]. [S.l.]: CRC Press, 2018.
- [21] 徐克付罗青林 □ □. Wireshark 环境下的网络协议解析与验证方法 [J/OL]. 计算机工程与设计, 2011, 32(770-773).
<http://dx.doi.org/10.16208/j.issn1000-7024.2011.03.068>.
- [22] OREBAUGH A, RAMIREZ G, BEALE J. Wireshark & Ethereal network protocol analyzer toolkit[M]. [S.l.]: Elsevier, 2006.
- [23] DEGIOANNI L. Development of an architecture for packet capture and network traffic analysis[J]. Graduation Thesis, Politecnico Di Torino (Turin, Italy, 2000).
- [24] 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.

- [25] 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.
- [26] FENG J, LI J. Google protocol buffers research and application in online game[C] // IEEE conference anthology. 2013 : 1–4.
- [27] 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.
- [28] 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.
- [29] 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.
- [30] 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.
- [31] 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.
- [32] 李响. 基于 Unity3D 的 ECS 框架的设计与实现 [D]. [S.l.] : 上海交通大学, 2019.
- [33] MAUREL C. CAE TroposTM Interface Control Document[R]. [S.l.] : CAE TroposTM, 2003.
- [34] 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.

- [35] 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.
- [36] DIMITRIJEVIĆ A M, RANČIĆ D D. Ellipsoidal Clipmaps—A planet-sized terrain rendering algorithm[J]. Computers & Graphics, 2015, 52 : 43–61.
- [37] STEVE MARSCHNER P S. Fundamentals of Computer Graphics fifth edition[M]. [S.l.] : CRC Press, 2022.
- [38] SHOEMAKE K. Animating rotation with quaternion curves[C] // Proceedings of the 12th annual conference on Computer graphics and interactive techniques. 1985 : 245–254.
- [39] 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.
- [40] VEILLEUX I, SPENCER J A, BISS D P, et al. In vivo cell tracking with video rate multimodality laser scanning microscopy[J]. IEEE Journal of selected topics in quantum electronics, 2008, 14(1) : 10–18.
- [41] KUBIAK I, PRZYBYSZ A. Fourier and Chirp-Z Transforms in the Estimation Values Process of Horizontal and Vertical Synchronization Frequencies of Graphic Displays[J]. Applied Sciences, 2022, 12(10) : 5281.
- [42] 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).
- [43] BOWRING B R. THE ACCURACY OF GEODETIC LATITUDE AND HEIGHT EQUATIONS[J]. Survey Review, 1985, 28 : 202–206.