

通用体感模拟系统设计与实现

摘要

体感模拟系统已有 40 多年成熟发展历史，多以硬件驱动形式被应用于军用和民航领域。随着虚拟现实和现实增强技术的出现及普及，商业和个人用户也开始对体感模拟系统产生迫切需求，但目前以软件驱动的廉价体感模拟系统的研发和成果则还很少。

本文提出了一种具有通用性的体感模拟系统的软件设计及实现方法。本文首先分析了传统成熟的基于硬件驱动的体感模拟系统的实现原理，给出一种使用软件驱动运动平台的方案。接下来，本文基于模块化和松耦合架构思想，设计并实现了功能可配置、可替换、可扩展的体感模拟系统软件，使其具备较强的运动平台兼容能力和个性化定制能力，充分发挥出软件驱动的优势。最后，为了表明该通用系统具备解决特定需求的能力，本文测试了将其与 Microsoft Flight Simulator X 软件结合实现个人级飞行体感模拟的效果。

关键词：体感模拟，洗出算法，飞行模拟

装
订
线

The Design and Implementation of a Generic Motion Simulation Software

ABSTRACT

In the last four decades, hardware-controlled motion simulators have been maturely developed and are widely used in military and civil aviation fields. However, cheap software-controlled motion simulators targeting at commercial and personal use lack sufficient attention, which becomes an active demand nowadays under the development and popularization of the virtual reality and augmented reality technology.

This paper introduced the architecture design and implementation of a generic motion simulation software to control motion platforms. First, the theory and approach of common hardware-controlled motion simulators is given, along with adaptions to make to be implemented with softwares. Then the design and implementation of our motion simulation software is presented, based on modular programming and loose coupling architecture to achieve configurable, replaceable and extensible functionalities, which is difficult when using hardware controllers. Finally, to show that our generic software is adaptable enough to satisfy a specific demand, we tested the performance of personal flight motion simulations by connecting our software with Microsoft Flight Simulator X.

Key words: motion cueing, washout algorithm, flight simulation

装
订
线

目 录

1 绪论	1
1.1 课题来源及研究目的和意义	1
1.2 发展历史及研究现状	2
1.2.1 体感模拟器	2
1.2.2 运动平台	2
1.2.3 体感模拟算法	3
1.3 文章结构	4
2 基础理论	5
2.1 综述	5
2.2 Stewart 六自由度并联运动平台	5
2.2.1 类型	5
2.2.2 串并联机构	6
2.2.3 自由度	7
2.2.4 坐标系	7
2.2.5 平台位置姿态	8
2.2.6 运动学反解	10
2.3 人体运动感知理论	10
2.3.1 本体感受器	11
2.3.2 前庭系统	11
2.3.3 视觉系统	13
2.4 体感模拟理论	13
2.4.1 模拟位移	13
2.4.2 模拟突发线加速度	13
2.4.3 模拟持续线加速度	13
2.4.4 模拟旋转	14
2.4.5 模拟角加速度	14
2.5 经典洗出算法	14
2.5.1 算法流程	14
2.5.2 体坐标系	15
2.5.3 缩放	15
2.5.4 滤波器	17
2.5.5 运动平台倾斜协调	17
2.5.6 运动平台位移运动	19
2.5.7 运动平台旋转运动	21
2.6 可扩展软件开发	23
2.6.1 模块化	23
2.6.2 内聚性和耦合性	23
2.6.3 S.O.L.I.D. 原则	24
2.6.4 IO 模型	24
3 软件需求分析及概要设计	26
3.1 需求综述	26
3.1.1 功能性需求	26
3.1.2 运行环境	26

装
订
线

3.2 总体设计.....	27
3.2.1 数据流图.....	27
3.2.2 架构.....	27
3.2.3 功能性需求与模块关系.....	29
4 体感模拟算法的软件实现研究.....	31
4.1 模拟滤波器的数字实现.....	31
4.2 积分的数字实现.....	33
4.3 算法复杂度分析.....	34
5 服务端基础框架详细实现.....	35
5.1 服务端技术和组件选择.....	35
5.2 服务端配置界面技术和组件选择.....	35
5.3 事件驱动系统实现.....	36
5.3.1 模块数据流.....	36
5.3.2 事件列表.....	36
5.3.3 具体实现.....	37
5.4 插件系统实现.....	38
5.4.1 插件形式.....	38
5.4.2 插件发现.....	39
5.4.3 插件逻辑层加载.....	40
5.4.4 插件逻辑层基类.....	40
5.4.5 插件界面层加载.....	41
5.4.6 插件界面层与框架界面层通信.....	43
5.4.7 插件界面层与逻辑层通信.....	46
5.4.8 插件选择实现.....	46
6 插件模块详细实现.....	48
6.1 FSX 模块	48
6.1.1 SimConnect 和 FSUIPC 机制	48
6.1.2 飞行姿态实时采样实现	48
6.1.3 FSX 信号插件之间通讯实现	51
6.2 标准飞行姿态输入模块	62
6.2.1 位置姿态存储	62
6.2.2 位置姿态信号输入	63
6.3 经典洗出算法模块	64
6.3.1 流程综述	64
6.3.2 滤波器参数配置	64
6.3.3 滤波器计算	65
6.3.4 算法实现	65
6.4 Stewart 运动平台可视化仿真输出模块	66
6.4.1 WebGL 技术和 Three.js 框架	66
6.4.2 几何形状构建	67
6.4.3 位置姿态计算	69
6.4.4 模块通讯机制	70
6.4.5 效果展示	71
7 测试及优化	72
7.1 FSX 模块	72
7.1.1 采样结果	72
7.1.2 采样效率	75

7.2 经典洗出算法.....	76
7.2.1 体感模拟结果.....	76
7.2.2 计算效率.....	80
7.3 服务端效果.....	81
7.4 Stewart 运动平台可视化仿真效果.....	82
8 结论和展望.....	84
8.1 结论.....	84
8.2 展望.....	84
参考文献.....	85
谢 辞.....	88

装
订
线

1 绪论

1.1 课题来源及研究目的和意义

体感模拟（Motion Simulation）也称为运动模拟、动感模拟。体感模拟通过在有限的物理空间中模拟出不同的加速度和方向，欺骗用户的大脑提供运动的感觉，如位置变化、速度变化、加速度变化等^[1]。目前，随着虚拟现实和增强现实技术的普及应用，商业和个人用户通过低廉的价格即可实现视觉和听觉的模拟，在此基础上对低廉成本的体感模拟有了迫切需求。体感模拟系统自上世纪 70 年代以来，已在军用和民航领域取得了非常丰富的成果和应用，但由于其成本一般超过 \$100,000 美元^[2]，个人或商业用户难以承受。

现有体感模拟器的高成本主要来自于：

- (1) 根据领域特点和需求进行专门定制
- (2) 全套训练或研究目的的驾驶控制仿真硬件
- (3) 高精度、高性能和高承载能力的运动平台
- (4) 维护费用及人员训练费用

21 世纪以来，一些个人飞行爱好者和企业开始探索面向 DIY 爱好者的低端体感模拟器。DIY 爱好者自行选择合适的硬件进行组装、调试，并使用商业体感模拟软件实现体感模拟。这样一套低端体感模拟器由于不包含专有硬件或软件且精度要求较低，成本已降低到 \$10,000 美元以下^[3]，但需要用户有非常强的动手能力，仍属于小众产品，无法适应当下的需求。另外，这些体感模拟软件也不具有调整体感模拟算法的能力和扩展能力，体感模拟效果将完全取决于软件本身，功能也受到限制^[4, 5]。

本文提出，对于个人级体感模拟器，体感模拟软件之于这个模拟器，应当类似于操作系统之于个人电脑，即体感模拟软件控制硬件而不限于特定硬件、为应用软件提供基础运行环境而不限于特定应用。其中，体感模拟器中的硬件主要为运动平台，应用软件主要为针对不同需求开发的各种组件。

基于此，本文设计了一种通用的体感模拟系统软件，以功能模块可配置、可替换、可扩展为核心理念，意图做到以下目的：

A. 提供开放性的运动平台硬件支持

通过开放的接口，允许不同运动平台厂商适配本课题提出的既定接口（即提供“驱动程序”或复用现有“驱动程序”），消除专有硬件软件的壁垒，引入了行业竞争，理论上能使用户能以较低的成本直接获取整套运动平台硬件方案。

B. 实现与其他系统的灵活集成

对于开发者而言，允许自由开发模块并组装，实现与飞行模拟器、虚拟现实头盔等其他系统的集成，满足用户各种需求。对于研究人员而言，允许其复用其他组件，自己专注于其中一个组件（如体感模拟算法）的改进和创新，同时能轻易地将这些模块直到硬件平台串联起来进行集成测试，并能进行实验对照。

C. 给出将基于硬件的体感模拟算法在软件上实现的一种可行方案

由于历史原因，现有体感模拟算法大多基于模拟信号设计，直接通过硬件实现，在软件上通过数值仿真软件进行模拟分析。本文基于先前论文给出的建议^[6]，研究了软件实现体感模拟算法的原理，并给出了高性能运行的实现，为后续此类软件体感模拟的课题提供一种参考，同时允许体感模拟算法研究人员继续使用熟悉的流程和工具、基于模拟信号进行算法设计和研究。

D. 给出此类模拟软件的一种详细架构设计及实现方案

为后续此类研究尤其是接口标准的制定提供一种参考。

1.2 发展历史及研究现状

1.2.1 体感模拟器

体感模拟器（Motion Simulator）最早多为飞行模拟器，用于进行飞行员培训^[7]。1910 年第一个体感模拟器 Sanders Teacher 诞生^[8]，在有风时飞行员能进行三自由度（3-Degree Of Freedom, 3-DOF）飞行控制模拟。1929 年，Link Trainer 被成功研发，它实现了运动系统，允许飞行员使用控制杆操作并相应地使用电力驱动平台进行三自由度旋转^[9]。1958 年研发的 Comet IV 则升级到使用液压驱动并实现了真正意义上的体感模拟，随后还逐渐实现了更高自由度的体感模拟器^[7]。1960 年代起，在阿波罗计划背景下，美国 NASA 机构进行了大量关于体感模拟的研究^[7]，发布了几个具有重要意义的体感模拟算法和模型。之后，诸如汽车驾驶模拟器、摩托车驾驶模拟器、坦克驾驶模拟器、舰船体感模拟器、列车驾驶模拟器等相继被开发出来^[10]。

装
订
线



图 1.1 Link Trainer 飞行模拟器



图 1.2 Comet 4 飞行模拟器

1.2.2 运动平台

运动平台（Motion Platform, Motion Base）是现代体感模拟器的基础，甚至很多时候“体感模拟器”一词就是指代运动平台^[11]。运动平台一般由一个固定在地面上的固定部分（称为固定平台、定平台、Fixed Platform）和一个能进行一定距离移动或角度旋转的部分（称为负载平台、动平台、Payload Platform）组成^[11]。运动平台分类标准一般是自由度，即平台分别在三个方向旋转、三个方向平移的能力。运动平台的自由度最高是六自由度，但近年来也有八自由度概念被提出，主要强调提供了额外两个方向的扩展位移^[11]。

Gough-Stewart 平台（简称为 Stewart 平台，也被称为 Hexapod 平台）是一种常见的六自由度平台，由 D Stewart 于 1965 年发表的著名文章“A Platform With Six Degrees of Freedom”^[12]中提出。Stewart 平台由上、下两个平台、六个可伸缩的支腿和连接它们的铰链构成，是一种并联机

构。由于其结构紧凑、承载能力强、运动学反解容易等优点，已成为高端体感模拟器的标准选择^[13]。



图 1.3 一个典型的 Stewart 运动平台



图 1.4 CableRobot 六自由度运动平台

尽管运动平台已有很长的发展历史，但新的运动平台仍不断被发明创造，如 2016 年德国研究人员开发了一种基于绳索的新型六自由度运动平台 CableRobot^[14]，由于其摆脱了固定底座的限制，可运动范围极大，在体感模拟上表现出了很大的潜力。

1.2.3 体感模拟算法

体感模拟算法（Motion Cueing Algorithm, MCA）指的是能将载具或载具模型的运动转换为运动平台的姿态同时保持人体运动感觉的算法^[15]，是体感模拟器的核心算法，直接决定了体感模拟的逼真程度。

运动平台的运动范围是极其有限的，载具的运动范围大大超过运动平台的运动范围，因此从物理角度来说运动平台不可能完全模拟载具的运动。体感模拟算法利用人体感知缺陷，在运动平台的物理限制之下，尽可能模拟出人体感知运动所需的因素，“欺骗”人体使得人体拥有相似的运动感觉^[16]。

目前主流的体感模拟算法为洗出算法（Washout Algorithm），主要有经典洗出算法、自适应洗出算法、最优洗出算法等。

A. 经典洗出算法

经典洗出算法（Classical Washout Algorithm）又被称为线性仿真算法（Linear Cueing Algorithm）^[17, 18]，是最早出现且目前应用最广泛的体感模拟算法。该算法公式简单，参数较少，经过校准后能够拥有很好的体感模拟效果。

经典洗出算法以线加速度和角速度作为输入，其中突变的线加速度经过二重积分后被用作为运动平台的位移来复现加速度；固定的线速度由于容易使平台移动超出物理限制，因此使用倾斜调整而不是位移来复现加速度；固定的角速度由于不易被人体察觉因此被丢弃；突变的角速度经过积分后被用作为运动平台的旋转角度来复现。

B. 自适应洗出算法

自适应洗出算法（Adaptive Washout Algorithm）^[19]是 NASA Langley 研究中心在经典洗出算法上进行改进提出的，主要为滤波器参数引入了自适应变化。自适应洗出算法在平台处于原点时相比经典洗出算法具有更高逼真程度。根据研究^[6]，自适应洗出算法相比经典洗出算法能取得

更高的逼真度。但由于自适应洗出算法引入了更复杂的公式和更多的参数，调参难度更高且计算时间更长。

C. 最优洗出算法

最优洗出算法（Optimal Washout Algorithm）^[20]主要利用人体运动感知上的研究，将人体前庭系统的模型引入洗出算法中，旨在最小化该模型下的运动感觉差距。最优洗出算法的效果直接取决于前庭模型的准确程度，调参比经典洗出算法困难。

表 1.1 比较了上述三种体感模拟算法的复杂度。其中，运行时间是在 Perkin Elmer 3250 计算机上得出的。

表 1.1 三种洗出算法的比较^[6]

算法	解微分方程数量	运行时间/迭代 (ms)	自由参数数量	参数透明度
经典洗出算法	13	1.0	21	非常好
最优洗出算法	24	1.3	38	一般
自适应洗出算法	38	1.8	64	一般

1.3 文章结构

本文共分为八章，按如下结构组织全文：

第一章阐述了本文的背景、目的、意义、相关领域发展历史、领域研究现状及本文结构；

第二章阐述了本课题基础理论和技术，包括运动平台模型、体感模拟理论、体感模拟算法模型、模块化开发原理；

第三章给出了本文所要实现的软件的需求分析和概要设计；

第四章研究并实现了将体感模拟算法运用在软件中进行高效计算的一种方法；

第五、六章给出了软件的详细设计和模块关键部分的代码实现；

第七章展示了本文实现的软件并进行了效果测试和优化分析；

第八章总结了本项课题的成果、意义，并对未来工作进行展望。

2 基础理论

2.1 综述

体感模拟器是一种在有限的空间内为人员提供近似于载具运动感觉的设备，包含一个能操纵人员在有限空间移动或旋转的运动平台。由于体感模拟器空间有限，而被模拟的载具则会在无限虚拟空间内运动，因此从物理上体感模拟器无法将被模拟载具的运动直接应用于控制运动平台运动。这种将载具模型的运动姿态转换为运动平台姿态（包括位移和旋转角度）、同时保持人体运动感觉的算法被称为体感模拟算法，是体感模拟器的核心算法，直接决定了体感模拟的逼真程度。

2.2 Stewart 六自由度并联运动平台

运动平台按机构可分为串联平台和并联平台，按自由度可分为六自由度平台、五自由度平台、四自由度平台、三自由度平台等。本文软件所适配的 Stewart 平台是一种在体感模拟器中最广泛使用的六自由度并联运动平台，也称为 Hexapod 平台，最早由德国研究人员 D Stewart 于 1965 年在英国杂志 IMECE 上发表^[12]。早期工业机器人多使用串联形式。串联机器人具备较大运动空间，而并联机器人空间有限，因此 Stewart 平台并未受到广泛应用。后来随着体感模拟器成为研究热点，Stewart 平台由于其在受限空间内良好的效果和更高的承载能力^[10]，逐渐成为主流的体感模拟器运动平台。

2.2.1 类型

Stewart 平台由上下两个平台和并行连接平台的多个支撑杆组成，多个支撑杆都可以独立的进行伸缩运动。上平台是动平台，下平台是定平台，如图 2.1 所示。

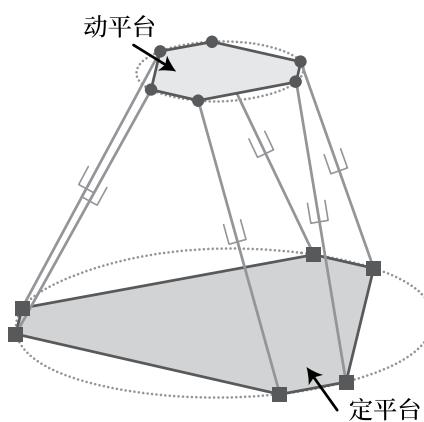


图 2.1 Stewart 平台模型

根据上下平台连接的支撑杆数目、支撑杆的运动链结构可以进一步将 Stewart 划分出不同类型^[21]，如 6-6-SPS 类型指的是下平台连接 6 个支撑杆、上平台连接 6 个支撑杆、支撑杆运动链结

构为球铰-移动副-球铰 (Spherical-Prismatic-Spherical); 6-6-UPS 类型支撑杆运动链结构则为万向铰-移动副-球铰 (Universal-Prismatic-Spherical)。

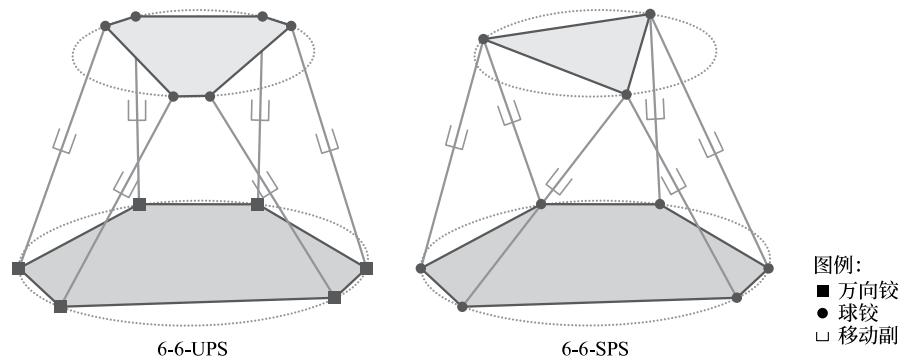


图 2.2 两种常见的 Stewart 平台结构模型

装

由于不同类型的 Stewart 平台结构参数不同，因此不加说明的情况下本文统一使用如图 2.2 所示的 6-6-UPS 结构的 Stewart 平台。

2.2.2 串并联机构

串联机构 (Serial Manipulator) 是工业机器人最广泛采用的结构，其末梢和基座之间使用多个运动链串联连接而成；并联机构 (Parallel Manipulator) 则是体感模拟器中最广泛采用的结构，其末梢和基座之间使用多个运动链并联连接而成。图 2.3、图 2.4 分别展示了一种典型的串联机构和一种典型的并联机构。

订

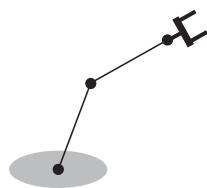


图 2.3 一种串联机构

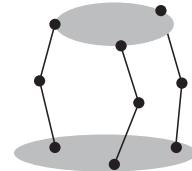


图 2.4 一种并联的机构

在体感模拟器中，使用并联机构有如下优势：

- (1) 体感模拟器的动平台需要具有较高承载能力以便承载显示器、控制器、驾驶员等等。并联机构的末梢（平台）由多个运动链共同支撑，相比串联机构拥有结构稳定、刚度大承载能力强的优点；

- (2) 体感模拟器要求运动精度越高越好。串联机构的运动误差是各个运动链叠加而成的，而并联机构不存在误差放大问题，具有更高运动精度；

- (3) 体感模拟器需要进行实时的运动学反解，即已知平台姿态求出运动链的目标长度以

便控制运动平台达到所需姿态。串联机构运动反解困难、正解容易，而并联机构则运动反解容易、正解困难，从而具有更高的计算速度。

2.2.3 自由度

确定运动时所必须给定的独立运动参数的数目被称作为自由度（Degree Of Freedom，简称 DOF）。例如在一个导轨上运动的汽车具有一自由度（1-DOF），因为汽车位置可由导轨上运动的距离确定；一个自由刚体在三维空间上的位置和朝向具有六自由度（6-DOF），可由三个方向上位移长度和三个方向上旋转角度确定。显然，若一个自由刚体拥有高于六自由度，其位置和朝向也能唯一确定，但由于六自由度已足够确定其位置和朝向，因此其中部分自由度是冗余的。对于体感模拟器来说，运动平台能进行的运动种类越多、其复现运动的能力就越强，因此体感模拟器最适合使用六自由度或更高自由度的运动平台。

根据 Chebychev - Grubler - Kutzbach 准则，可使用以下公式计算系统自由度^[22]：

$$F = 6(N - 1) - \sum_{i=1}^j 6 - f_i \quad (2.1)$$

其中， F 是计算出的系统自由度；

N 是刚体个数；

j 是运动副个数；

f_i 是每一个运动副的自由度。

对于一个 6-6-UPS Stewart 平台，其动平台处是球铰，具有三自由度；中间连接处是移动副，具有一自由度；定平台处是万向铰，具有二自由度^[23]。因此其自由度为：

$$F_{Stewart} = 6(14 - 1) - 6 \times 3 - 6 \times 5 - 6 \times 4 = 6 \quad (2.2)$$

2.2.4 坐标系

为了后续描述方便，本文为常见的 6-6-UPS Stewart 平台建立以定平台中心为原点的空间坐标系 O_{xyz} ，如图 2.5 所示。

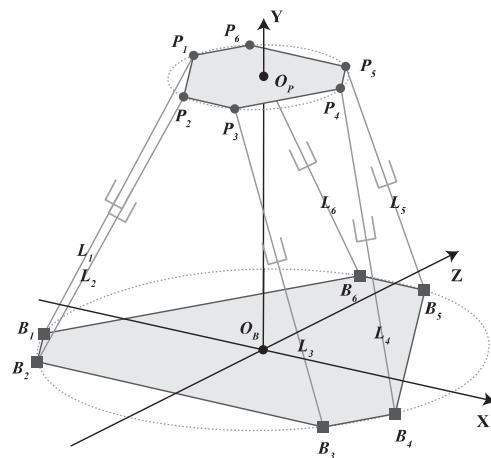


图 2.5 6-6 UPS Stewart 平台坐标系示意图

其中， $\overrightarrow{O_B}$ 是定平台中心点，也是坐标系原点；
 $\overrightarrow{O_p}$ 是动平台中心点；
 $\overrightarrow{B_i}$ ($i \in 1..6$) 是定平台六个铰点；
 $\overrightarrow{P_i}$ ($i \in 1..6$) 是动平台六个铰点；
 $L_i = |\overrightarrow{P_i} - \overrightarrow{B_i}|$ ($i \in 1..6$) 是六个支撑杆的长度。

2.2.5 平台位置姿态

A. 结构参数

图 2.5 所示的 Stewart 结构中，动平台和定平台都是由一个正三角形和一个圆截取而成的六边形，可由圆的半径 r 和短边圆心角的一半 α 来描述，如图 2.6 所示。

装
订
线

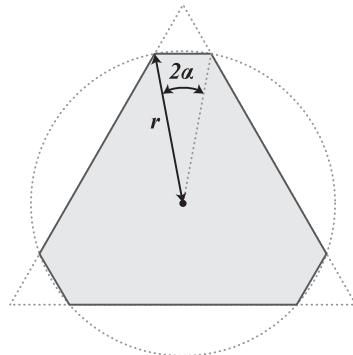


图 2.6 动平台和定平台平面图形

定义动平台和定平台的参数 r, α 分别为 r_p, α_p 、 r_B, α_B ，则初始情况下，在 O_{xz} 坐标系顶视图下其模型如图 2.7 所示。

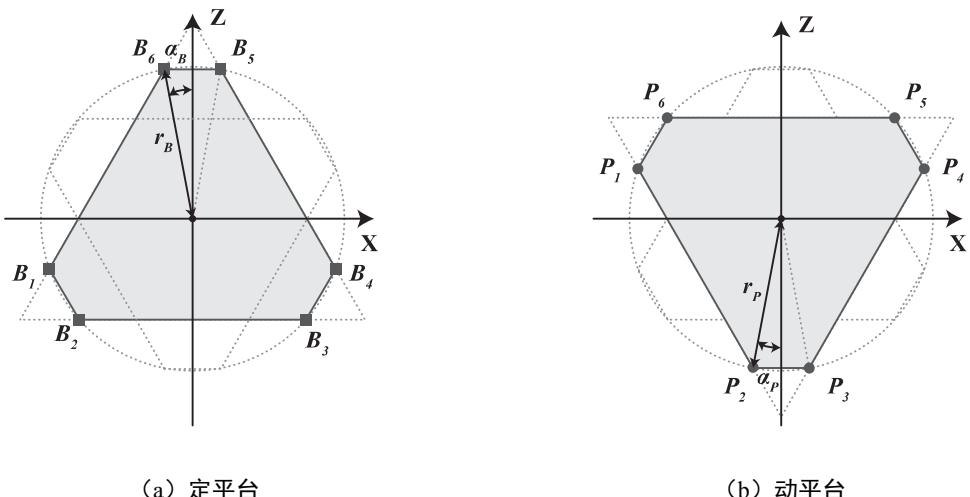


图 2.7 动平台和定平台模型

确定了 $r_P, \alpha_P, r_B, \alpha_B$ 和高度 $L = O_B O_P$, 则 Stewart 平台初始状态可被完全确定:

$$\vec{\theta}_B = [210^\circ - \alpha_B \quad 210^\circ + \alpha_B \quad 330^\circ - \alpha_B \quad 330^\circ + \alpha_B \quad 90^\circ - \alpha_B \quad 90^\circ + \alpha_B]$$

$$\vec{\theta}_P = [150^\circ + \alpha_P \quad 270^\circ - \alpha_P \quad 270^\circ + \alpha_P \quad 30^\circ - \alpha_P \quad 30^\circ + \alpha_P \quad 150^\circ - \alpha_P]$$

$$\vec{B}_i = \begin{bmatrix} r_B \cdot \cos(\theta_B^{<i>}) \\ 0 \\ r_B \cdot \sin(\theta_B^{<i>}) \end{bmatrix}, \vec{P}_i = \begin{bmatrix} r_P \cdot \cos(\theta_P^{<i>}) \\ L \\ r_P \cdot \sin(\theta_P^{<i>}) \end{bmatrix}, i \in 1..6 \quad (2.3)$$

将 $r_P, \alpha_P, r_B, \alpha_B, L$ 称为平台结构参数。在确定了结构参数基础上，在任意时刻，若确定了动平台的姿态（即确定了动平台中心 $\overrightarrow{O_P}$ 的坐标和动平台 $\overrightarrow{P_{1..6}}$ 所在平面的法向量的方向），则可以确定此刻 Stewart 平台的状态，包括动平台六个铰点坐标、支撑杆长度、支撑杆端点坐标等。

B. 位置

本文使用空间向量 \vec{t} 描述任意时刻动平台的位置偏移:

$$\vec{t} = [x, y, z]^T \quad (2.4)$$

其中， x, y, z 分别为在 X, Y, Z 轴方向上的偏移。

定义初始情况下动平台位置偏移 $\overrightarrow{t_{initial}}$ 为:

$$\overrightarrow{t_{initial}} = [0, 0, 0]^T \quad (2.5)$$

则任意时刻动平台中心 $\overrightarrow{O_P}$ 为:

$$\overrightarrow{O_P} = \overrightarrow{O_{P_{initial}}} + \vec{t} \quad (2.6)$$

其中， $\overrightarrow{O_{P_{initial}}}$ 为初始情况下动平台中心。

C. 姿态

本文使用欧拉角向量 \vec{r} 描述任意时刻动平台的姿态旋转角:

$$\vec{r} = [\alpha, \beta, \gamma]^T \quad (2.7)$$

其中， α, β, γ 分别为在 X, Y, Z 轴方向上旋转的角度。

定义初始情况下动平台姿态旋转角 $\overrightarrow{r_{initial}}$ 为:

$$\overrightarrow{r_{initial}} = [0, 0, 0]^T \quad (2.8)$$

则任意时刻动平台所在平面的单位法向量为:

$$\overrightarrow{V_P} = R_X \cdot R_Y \cdot R_Z \cdot \overrightarrow{V_{P_{initial}}} \quad (2.9)$$

其中， $\overrightarrow{V_{P_{initial}}}$ 为初始情况下动平台所在平面的单位法向量:

$$\overrightarrow{V_{P_{initial}}} = [0, 1, 0]^T$$

R_X 为绕 X 轴旋转的旋转矩阵:

$$R_X = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha & \sin \alpha \\ 0 & -\sin \alpha & \cos \alpha \end{bmatrix}$$

R_Y 为绕 Y 轴旋转的旋转矩阵:

$$R_Y = \begin{bmatrix} \cos \beta & 0 & -\sin \beta \\ 0 & 1 & 0 \\ \sin \beta & 0 & \cos \beta \end{bmatrix}$$

R_Z 为绕 Z 轴旋转的旋转矩阵:

$$R_Z = \begin{bmatrix} \cos \gamma & \sin \gamma & 0 \\ -\sin \gamma & \cos \gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2.2.6 运动学反解

从动平台的位置偏移 t 和姿态旋转 r 求解各个支撑杆伸缩量 ΔL_i 称为运动学反解 (Inverse Kinematics)，反之称为运动学正解 (Forward Kinematics)。体感模拟器需要根据载具的姿态调整平台的姿态，调整平台姿态的途径是调整支撑杆伸缩量，因此体感模拟器进行的是运动学反解。

方便起见，以动平台初始情况下的中心点建立辅助坐标系 O'_{xyz} :

$$\overrightarrow{O_{offset}} = \overrightarrow{O_{initial}} - \overrightarrow{O_B} = \overrightarrow{O_{initial}} \quad (2.10)$$

$$\overrightarrow{O'_{xyz}} = \overrightarrow{O_{xyz}} + \overrightarrow{O_{offset}} \quad (2.11)$$

则在辅助坐标系 O'_{xyz} 下，动平台中心点初始坐标 $O'_{P_{initial}}$ 为:

$$\overrightarrow{O'_{P_{initial}}} = \overrightarrow{O_{initial}} - \overrightarrow{O_{offset}} = [0, 0, 0]^T \quad (2.12)$$

根据空间变换公式，空间上任意坐标 V 以坐标轴原点为旋转中心旋转 r 并位移 t 后的坐标 V' 为:

$$\overrightarrow{V'} = R_X \cdot R_Y \cdot R_Z \cdot \overrightarrow{V} + \vec{t} \quad (2.13)$$

因此，对于任意时刻，若给定的动平台位置偏移 \vec{t} 和姿态旋转角 \vec{r} ，则动平台各个铰点坐标在辅助坐标系下的坐标 $\overrightarrow{P'''}_{1..6}$ 为:

$$\overrightarrow{P'''_i} = R_X \cdot R_Y \cdot R_Z \cdot \overrightarrow{P'_i} + \vec{t}, i \in 1..6 \quad (2.14)$$

其中， $\overrightarrow{P'_i}$ 为辅助坐标系下的动平台各个铰点坐标 $\overrightarrow{P_{1..6}}$:

$$\overrightarrow{P'_i} = \overrightarrow{P_i} - \overrightarrow{O_{offset}}, i \in 1..6$$

综上，原坐标系 O_{xyz} 下任意时刻动平台各个铰点坐标 $\overrightarrow{P''}_{1..6}$ 为:

$$\overrightarrow{P''_i} = \overrightarrow{P'''_i} + \overrightarrow{O_{offset}} = R_X \cdot R_Y \cdot R_Z \cdot (\overrightarrow{P'_i} - \overrightarrow{O_{offset}}) + \vec{t} + \overrightarrow{O_{offset}}, \quad i \in 1..6 \quad (2.15)$$

由图 2.5 可知，任意时刻支撑杆的长度为:

$$\overrightarrow{L_i} = |\overrightarrow{P'_i} - \overrightarrow{B_i}|, i \in 1..6 \quad (2.16)$$

则在已知初始情况时各个支撑杆长度 $L_{i0}(i \in 1..6)$ 的情况下，各个支撑杆伸缩量 ΔL_i 为:

$$\Delta L_i = L_{i0} - |\overrightarrow{P''_i} - \overrightarrow{B_i}|, i \in 1..6 \quad (2.17)$$

2.3 人体运动感知理论

人体对运动的感知原理是体感模拟器复现体感效果的关键。正是由于人体的运动感知系统无法实现精确和准确的感知，才允许体感模拟器在不违背物理原理的条件下，在有限的空间内复现被模拟载具在无限空间内的运动。

大脑接收并处理不同感觉系统的信号形成人体和周围环境的感知，例如视觉、听觉、平衡和接触等。这些信号由感受器接受外界刺激产生。外感受器（Exteroceptors）分布在皮肤、视器、听器等处，接受压力、温度、光波、声波等人体外部的刺激；内感受器（Interoception）分布在血管等处，接受渗透压、疼痛等人体内部的刺激；本体感受器（Proprioceptors）分布在肌、关节、内耳等处，接受运动和平衡变化产生的刺激^[24]。

人体维持平衡并感知空间定位主要依靠的是本体感受器、前庭系统和视觉系统^[25]。

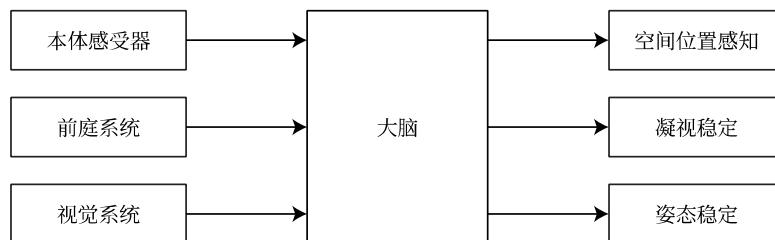


图 2.8 人体运动感知方式

装
订
线

2.3.1 本体感受器

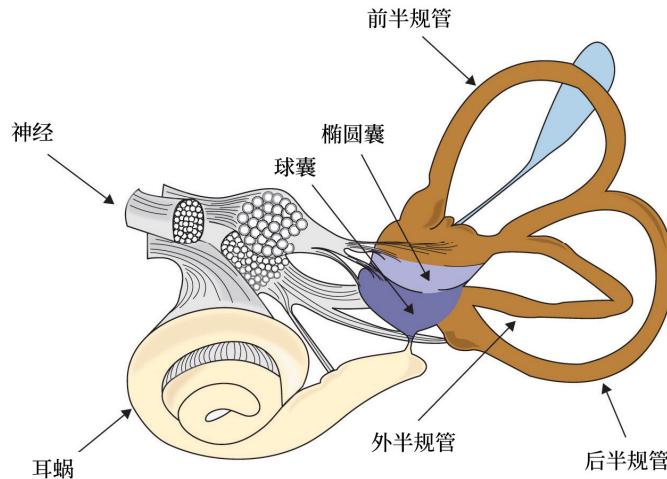
本体感受器（Proprioceptors）主要用于感知人体的空间位置。当空间移动速度超过阈值后，本体感受器将无法感知到移动^[26]，此时人体依靠视觉系统（如看到窗外物体移动）或受力变化（如感受到椅背推力）感知位置变化。

体感模拟器会利用这个缺陷进行平台位置洗出。洗出指的是将平台慢慢回复到原点以便为下一次体感模拟留下足够的运动空间。当运动速度超过阈值后，本体感受器感觉不到运动，此时体感模拟器进行洗出动作，而人体则以为仍然在进行连续运动。

2.3.2 前庭系统

前庭系统（Vestibular System）感知人体在空间上的移动和旋转，从而对身体姿态进行平衡，是主要的运动信号来源。前庭系统位于内耳内，拥有三个相互垂直的装有一定流体的半规管、球囊和椭圆囊，如图 2.9 所示。球囊和椭圆囊合起来称为耳石。半规管可以感知角速度，耳石可以感知线加速度。匀速运动不会刺激前庭系统。

当头部具有加速度时，半规管中的液体会发生流动，带动毛囊移动，从而刺激神经末梢使得大脑感知到角速度和线加速度。但当一个方向的加速度持续 10~20 秒后，毛囊将恢复原位，导致大脑感知成加速度停止^[27]。

图 2.9 前庭系统结构^[28]

根据研究^[29], 耳石对线加速度的感知是比力 (Specific Force)。比力的定义为:

$$\vec{f} = \vec{a} - \vec{g} \quad (2.18)$$

其中, \vec{f} 为比力向量 (m/s^2);

\vec{a} 为人体的绝对线加速度向量 (m/s^2);

\vec{g} 为重力加速度向量 (m/s^2)。

前庭系统缺陷有:

- (1) 无法分辨出比力是由哪些力组成的;
- (2) 只有超过阈值的加速度或速度才会被前庭系统感知到。

根据 Meiry、Young 的研究^[30]和 Peters 的研究^[31], 前庭系统感觉阈值如表 2.1 所示。

表 2.1 前庭系统感觉阈值^[30, 31]

类型	阈值
前后加速度 (m/s^2)	0.17
左右加速度 (m/s^2)	0.17
上下加速度 (m/s^2)	0.28
俯仰加速度 (deg/s^2)	0.5
横滚加速度 (deg/s^2)	0.5
偏航加速度 (deg/s^2)	0.14
俯仰速度 (deg/s)	2.6
横滚速度 (deg/s)	3.2
偏航速度 (deg/s)	1.1

与本体感受器类似, 体感模拟器可以利用前庭系统缺陷使得洗出动作不被人体感知到。

2.3.3 视觉系统

视觉（Visual）是人感受运动的重要来源。大脑通过眼睛看到周围景物的大小、方向判断视觉人体的绝对空间位置和方向。体感模拟提供的模拟视觉信号必须和提供的加速度及角速度同步，否则由于大脑无法协调这些感受器输入的信号，人会感到眩晕和强烈的不适^[1]。

2.4 体感模拟理论

运动平台的运动范围受到物理限制，因此大部分时候需要基于上节描述的人体运动感知理论，利用人体运动感知的缺陷，对载具的运动进行体感上的模拟，尽可能“欺骗”大脑使之产生相近的运动感觉。本节具体描述了各种运动的模拟方式。

2.4.1 模拟位移

由于人体前庭系统和本体感受器无法感知绝对位置和速度，只能感知相对速度变化（即加速度），因此大脑完全依靠视觉系统提供运动依据。例如，在以匀速运动的车上，除视觉系统以外，感觉与静止状态是一样的。

基于此，利用视觉的仿真即可模拟位移。

2.4.2 模拟突发线加速度

突发线加速度可直接复现。但运动平台本身能做到的最大加速度是受限的，这决定了突发线加速度的逼真程度。

另外由于运动平台空间是有限的，因此在突发线加速度发生后，平台需要以人无法感知到的加速度回到原点以便有足够的空间进行下一次加速度模拟。

2.4.3 模拟持续线加速度

平台空间是有限的，所以持续的线加速度无法直接模拟，但由于人无法感知加速度的组成和精确的加速度大小，因此可以依靠倾斜平台，将一部分重力加速度转换为前后或左右方向的加速度，给人以持续的加速度感觉。另外，倾斜后椅背给予人的压力也会增加或减少，这进一步增强了人体的运动感觉。这种方法称为倾斜协调（Tilt Coordination）。

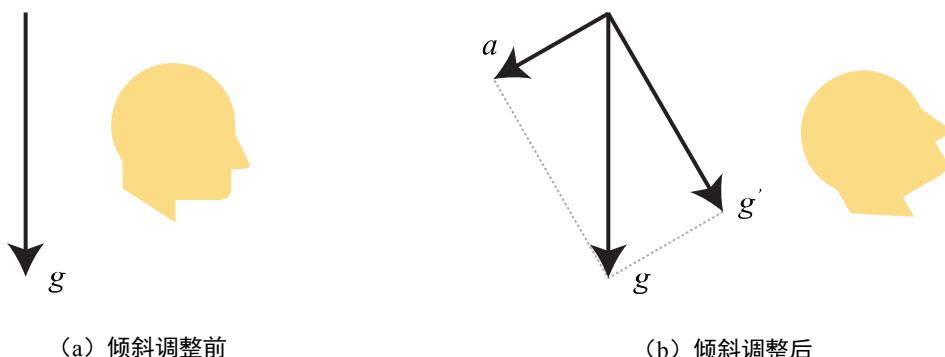


图 2.10 倾斜协调示例

如图 2.10 所示，倾斜前人体有垂直方向上大小为 g 的重力加速度，在倾斜后人体受到的重力加速度没变，但感受到的是垂直方向上大小为 $g' = g \cdot \cos \theta$ 的加速度和前后方向上大小为 $a = g \cdot \sin \theta$ 的加速度。只要倾斜角度不太大，则垂直方向上加速度的减小不明显，人无法感知到这个变化。

例如，对于一个持续 10 秒、前后方向、大小为 1m/s^2 的加速度，其 10 秒后的位移为 $S = 50\text{m}$ ，一般运动平台无法做到这么大的位移，因此利用倾斜协调模拟。若倾斜 5.8° ，则倾斜后人体垂直方向上加速度为 $g' = g \cdot \cos(5.8^\circ) = 0.995g$ ，差别非常小，无法被感知；人体前后方向上有 $a' = g \cdot \sin(5.8^\circ) = 1\text{m/s}^2$ 的加速度，可以被感知，提供了持续的线加速度感觉。

倾斜协调具有一定局限性：

- (1) 只能模拟前后和左右方向上的加速度，无法模拟垂直方向上的加速度；
- (2) 倾斜速度和加速度不能超过表 2.1 中的旋转阈值，否则倾斜会被察觉；
- (3) 只能模拟较小的加速度，垂直方向上加速度的明显改变会导致倾斜被察觉。

装
订
线

2.4.4 模拟旋转

目前旋转还无法模拟，只能在视觉上给予旋转的感觉。

2.4.5 模拟角加速度

目前还无法欺骗人体产生角加速度感觉，只能直接复现突发角加速度。复现结束后，为了能进行下一次角加速度复现，需要在不超过表 2.1 所给出的阈值的条件下进行洗出操作让平台回复原位。

需要注意的是，由于平台倾角已被用于进行线加速度的模拟，因此在一些情况下模拟角加速度会与模拟线加速度产生歧义，造成仿真效果降低。

2.5 经典洗出算法

基于前节的体感模拟理论，研究人员提出了各种体感模拟算法，即能将载具或载具模型的运动转换为运动平台的姿态同时保持人体运动感觉的算法。经典洗出算法是体感模拟器中应用最广泛也是最早提出的算法之一。经典洗出算法具有公式简单、参数少、调参方便、模拟效果好、计算快等优点。

2.5.1 算法流程

经典洗出算法流程如图 2.11 所示。

其中，输入 \vec{f}_A 是体坐标系下比力向量；

输入 $\vec{\omega}_A$ 是体坐标系下角速度向量；

常数 \vec{g} 是重力加速度向量；

输出 \vec{s} 是运动平台的位置偏移向量，即式 (2.4) 定义的向量；

输出 $\vec{\theta}$ 是运动平台的姿态旋转角向量，即式 (2.7) 定义的向量；

s 是拉普拉斯算子 (Laplace Operator)，所有输入输出都在 s 域 (s-domain) 下。

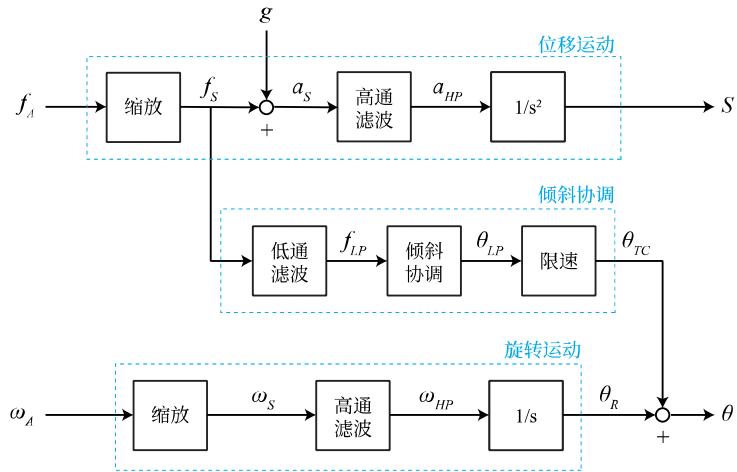


图 2.11 经典洗出算法流程

装
订
线

大多数情况下，运动模拟算法的原始输入信号是体坐标系下的绝对线加速度 \vec{a}_A 。根据比力计算式 (2.18)，输入比力 \vec{f}_A 可由以下计算得出：

$$\vec{f}_A = \vec{a}_A - \vec{g}_A \quad (2.19)$$

其中， \vec{g}_A 是体坐标系下的重力加速度向量。

2.5.2 体坐标系

规定载具前后方向为 x 轴，左右方向为 y 轴，垂直方向为 z 轴，且向前、向左、向上为正方向；绕 x 轴旋转的欧拉角为 α ，绕 y 轴旋转的欧拉角为 β ，绕 z 轴旋转的欧拉角为 γ ，且向右翻滚、向下俯冲、向左偏航为正方向，如图 2.12 所示。

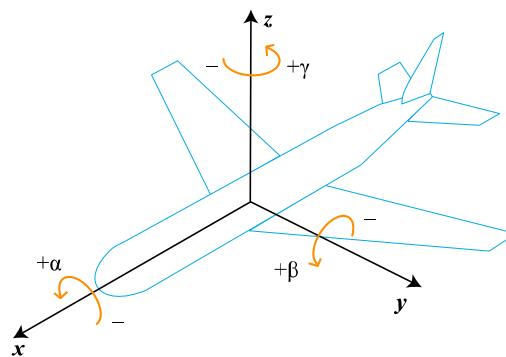


图 2.12 载具坐标系

2.5.3 缩放

运动平台的硬件限制包括位移和角度、速度和加速度等，因此输入的比力 f_A 和旋转角速度 ω_A 需要进行缩放以避免超出运动平台硬件限制^[18]。另外，一般还希望能对较小幅度的输入信号

进行适当增强以便有运动感觉^[10]。

对于一个需要缩放的输入信号 x , 缩放运算可以表示为:

$$y = \begin{cases} y_{max} & x > x_{max} \\ f(x) & 0 \leq x \leq x_{max} \\ -f(-x) & -x_{max} \leq x < 0 \\ -y_{max} & x < -x_{max} \end{cases} \quad (2.20)$$

其中, y 是缩放后的信号;

$f(x)$ 是一个缩放函数, 对于 $0 \leq x \leq x_{max}$ 满足 $0 \leq f(x) \leq y_{max}$;

x_{max} 为输入信号最大值, 与输入信号有关, 需事先测定;

y_{max} 为缩放后信号最大值, 由运动平台物理限制决定。

由式 (2.20) 可见, 缩放运算是一个奇函数, 将范围 $[-x_{max}, x_{max}]$ 的输入信号使用缩放函数 $f(x)$ 缩放到 $[-y_{max}, y_{max}]$, 将范围以外的输入信号 $|x| > x_{max}$ 限定在 $|y| = y_{max}$ 。

缩放函数一般使用线性函数和三次函数。

A. 线性缩放

线性缩放使用一次函数进行缩放:

$$f(x) = c_1x + c_0 \quad (2.21)$$

其系数为:

$$\begin{cases} c_1 = \frac{y_{max}}{x_{max}} \\ c_0 = 0 \end{cases}$$

即:

$$f(x) = \frac{y_{max}}{x_{max}} \cdot x \quad (2.22)$$

线性缩放效果如图 2.13 所示。

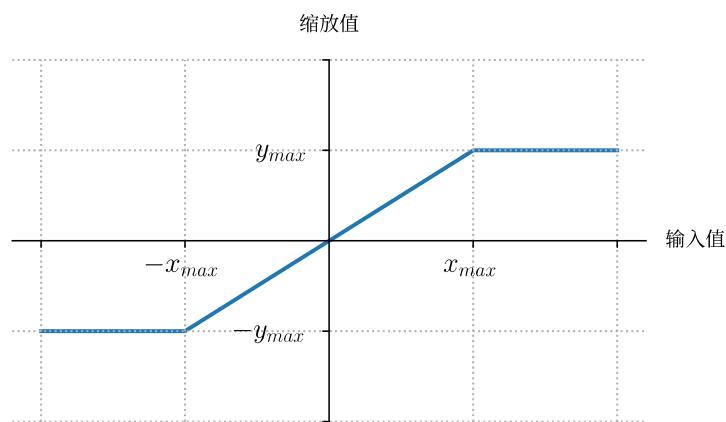


图 2.13 使用线性缩放函数进行缩放 ($x_{max} = 10, y_{max} = 6$)

B. 三次缩放

三次缩放使用三次函数进行缩放：

$$f(x) = c_3x^3 + c_2x^2 + c_1x + c_0 \quad (2.23)$$

其系数为：

$$\begin{cases} c_3 = x_{max}^{-3}(k_0x_{max} - 2y_{max} + k_1x_{max}) \\ c_2 = x_{max}^{-2}(-2k_0x_{max} + 3y_{max} - k_1x_{max}) \\ c_1 = k_0 \\ c_0 = 0 \end{cases}$$

其中： k_0, k_1 分别是 $x = 0, x = x_{max}$ 时的斜率，一般取 $k_0 = 1, k_1 = 0.1$ 。

三次缩放效果如图 2.14 所示，由图可见使用三次函数缩放后的信号过渡比较平滑。

装
订
线

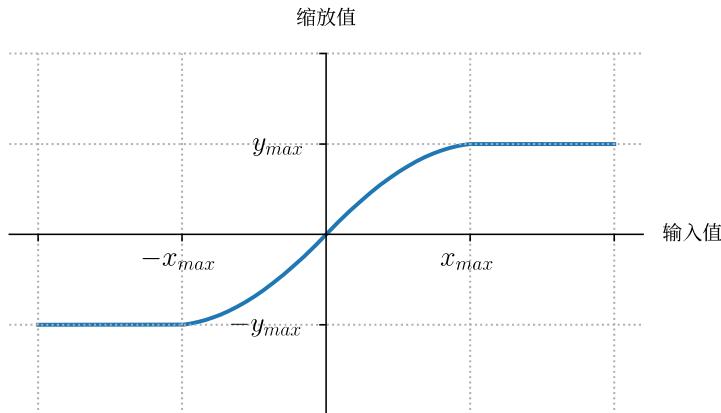


图 2.14 使用三次缩放函数进行缩放 ($x_{max} = 10, y_{max} = 6$)

2.5.4 滤波器

洗出算法中一般包括高通滤波 (High Pass Filter) 和低通滤波 (Low Pass Filter)。广义上高通滤波指的是允许高频信号通过、减弱低频信号的滤波器，低通滤波指的是允许低频信号通过、减弱高频信号的滤波器。

在体感模拟器中，高频信号在时域上的含义是突发信号，低频信号的含义是持续信号。即高通滤波减弱持续信号留下突发信号，相反地，低通滤波减弱突发信号留下持续信号。

根据本文前一节描述的体感模拟理论，由于运动平台的物理限制，为了最大化体感模拟效果，持续的线加速度和突发的线加速度需要采用不同的模拟策略：持续的线加速度非常容易使得运动平台超出范围，因此要依靠倾斜模拟，而突发的线加速度不容易使运动平台超出范围，因此可直接用于运动平台而不需要模拟。通过使用高通滤波器和低通滤波器，就可以对突发线加速度和持续线加速度使用上不同策略。

需要注意的是，体感模拟器中滤波器习惯使用模拟滤波器，即滤波器输入的是连续的模拟信号，描述滤波器所用的传递函数 (Transfer Function) 是 s 域上的函数。

2.5.5 运动平台倾斜协调

被模拟载具前后 (x) 和左右 (y) 的持续线加速度利用倾斜协调进行模拟。对于输入的缩放后的比力向量 \vec{f}_S , 其 x, y 方向上的分量使用二阶低通滤波器获得 f_{LP}^x, f_{LP}^y 。该滤波器传递函数为:

$$\frac{f_{LP}^{x,y}}{f_S^{x,y}} = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \quad (2.24)$$

其中, ω_n 为滤波器的截止频率;

ζ 为阻尼比。

根据研究^[6], 该滤波器建议参数为:

表 2.2 加速度低通滤波器建议参数^[6]

滤波器	ω_n	ζ
x 方向	5.0	1.0
y 方向	8.0	1.0

装

订

线

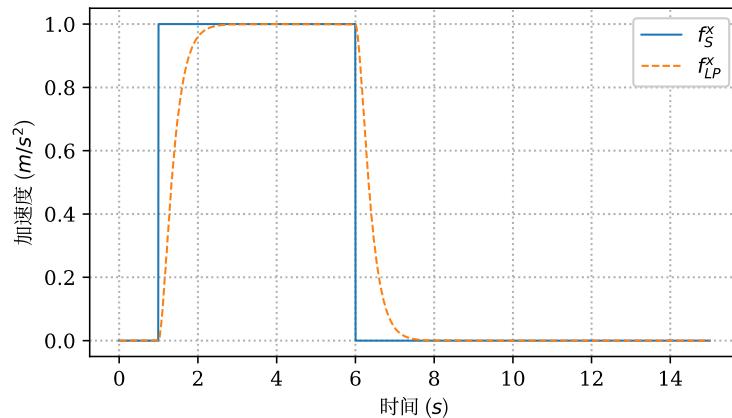


图 2.15 二阶低通滤波器输入输出

现计算低通滤波后的信号与倾斜角度的关系。根据图 2.10, 前后倾斜角度为 θ 时可以由重力加速度在体坐标系前后方向产生 $x = g \sin \theta$ 大小的分量, 因此为了模拟产生大小为 x 的前后方向分量, 需要倾斜的角度 θ 为:

$$\theta = \sin^{-1} \frac{x}{g} \quad (2.25)$$

左右方向倾斜角度计算同理。

根据坐标系修正正负后可有如下公式:

$$\theta_{LP} = \begin{bmatrix} \sin^{-1} \frac{f_{LP}^y}{g} \\ -\sin^{-1} \frac{f_{LP}^x}{g} \\ 0 \end{bmatrix} \quad (2.26)$$

最后，为了使得倾斜协调时倾斜过程不被人体察觉，需要限制倾斜协调的角速度在表 2.1 所列的阈值之下，即满足：

$$\dot{\theta}_{TC}^x \leq 3.2, \dot{\theta}_{TC}^y \leq 2.6 \quad (2.27)$$

2.5.6 运动平台位移运动

被模拟载具前后 (x)、左右 (y) 和垂直 (z) 方向的突发线加速度直接在运动平台上复现。由于输入是缩放后的比力向量 \vec{f}_S ，因此需要根据式 (2.18) 得到绝对线加速度 \vec{a}_S ：

$$\vec{a}_S = \vec{f}_S + \vec{g} \quad (2.28)$$

对于 \vec{a}_S ，需要使用高通滤波器提取出其中的高频加速度信号 \vec{a}_{HP} ，该信号经过二次积分后即可得到运动平台目标位移 \vec{s} 。该滤波器在经典洗出算法的论文^[17]中使用的是二阶高通滤波器：

$$\frac{\vec{a}_{HP}}{\vec{a}_S} = \frac{s^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \quad (2.29)$$

其中， ω_n 为滤波器的截止频率；

ζ 为阻尼比。

根据研究^[6]，该滤波器建议参数如表 2.3 所示。

表 2.3 加速度二阶高通通滤波器建议参数^[6]

滤波器	ω_n	ζ
x 方向	2.5	1.0
y 方向	4.0	1.0
z 方向	4.0	1.0

使用二阶高通滤波器不会使二次积分后的平台位移 \vec{s} 回归原点，但由于加速度经过缩放，且真实的载具很少会有持续加速情况，因此大多数情况下二阶高通滤波器是够用的^[17, 18]。若要使之回归原点，可以再加一个一阶滤波器，从而成为三阶高通滤波器：

$$\frac{\vec{a}_{HP}}{\vec{a}_S} = \frac{s^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \cdot \frac{s}{s + \omega_b} \quad (2.30)$$

其中， ω_n 为二阶滤波器的截止频率；

ω_b 为一阶滤波器的截止频率；

ζ 为二阶滤波器的阻尼比。

该额外一阶滤波器截止频率 ω_b 的建议取值^[32]为 $\omega_b = \omega_n / 10$ 。

现采用与本文 2.5.5 节（见 17 页）一样的场景测试二阶滤波器和三阶滤波器的效果。

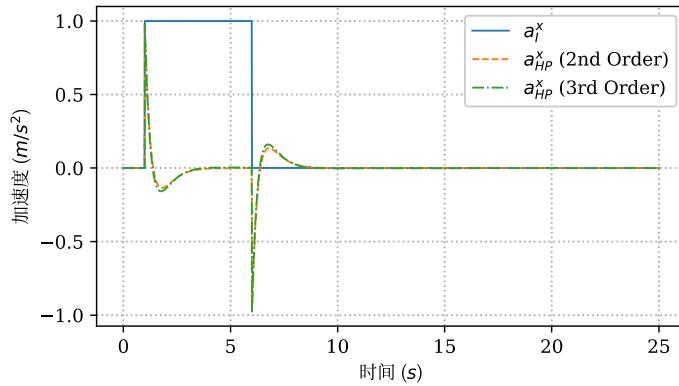


图 2.16 二阶和三阶高通滤波器对脉冲信号输出比较

如图 2.16 所示，二阶和三阶高通滤波器的加速度输出非常相近。

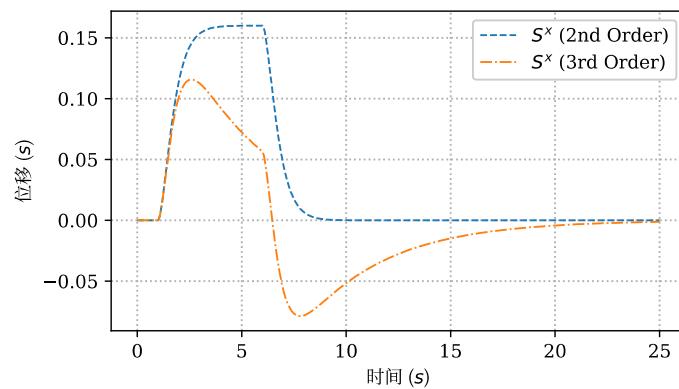


图 2.17 二阶和三阶高通滤波器对脉冲信号输出位移比较

如图 2.17 所示，对于脉冲加速度输入，二阶和三阶高通滤波器的位移输出都能回到原点。

三阶高通滤波器优势在于对于持续的加速度，输出的位移也能回归原点。现测试输入加速度信号为阶跃信号的情况，曲线如图 2.18 所示。

如图 2.19 所示，在持续加速的情况下三阶滤波器的位移输出能缓慢回到原点，二阶滤波器无法回到原点。

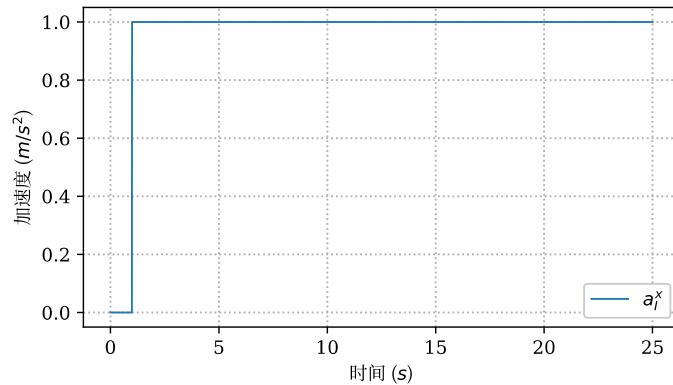


图 2.18 阶跃加速度输入情况

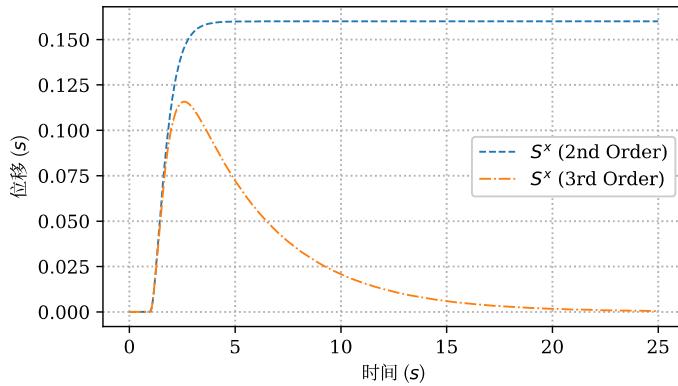


图 2.19 二阶和三阶高通滤波器对阶跃信号输出位移比较

2.5.7 运动平台旋转运动

被模拟载具三个方向的低频旋转无法模拟也无法复现，因此丢弃不模拟，高频旋转则直接在运动平台上复现。对于输入的缩放后的角速度 $\overrightarrow{\omega_s}$ ，使用高通滤波器提取出其中的高频部分，高频角速度信号 $\overrightarrow{\omega_{HP}}$ 经过一次积分后即可得到旋转角度 $\overrightarrow{\theta_R}$ 。该旋转角度与 2.5.5 节中倾斜协调输出的角度 $\overrightarrow{\theta_{TC}}$ 相加后即可得出平台最终旋转角度 $\vec{\theta}$ ：

$$\vec{\theta} = \overrightarrow{\theta_R} + \overrightarrow{\theta_{TC}} \quad (2.31)$$

滤波器在经典洗出算法的论文^[17]中使用的是一阶高通滤波器：

$$\frac{\overrightarrow{\omega_{HP}}}{\overrightarrow{\omega_s}} = \frac{s}{s + \omega_n} \quad (2.32)$$

其中， ω_n 为滤波器的截止频率。

表 2.4 角速度一阶高通滤波器建议参数^[6]

滤波器	ω_n
所有方向	1.0

根据研究^[6], 该滤波器建议参数如表 2.4 所示。

一阶高通滤波器在一次积分后得出的角度不具有回到原点的特性。若载具在某个方向上可能进行任意方向大范围旋转且不旋转回去，则需要将滤波器换为二阶高通滤波器以保证洗出角度为零。一般将绕 z 轴的旋转即偏航角 γ 升为二阶高通滤波器：

$$\frac{\overrightarrow{\omega_{HP}^{\gamma}}}{\overrightarrow{\omega_s^{\gamma}}} = \frac{s^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \quad (2.33)$$

其中, ω_n 为滤波器的截止频率;

ζ 为滤波器的阻尼比, 取 $\zeta = 1$ 。

设输入信号为阶跃角速度信号, 现比较一次和二次滤波器之间输出的角速度和输出积分后角度的差异。

装
订
线

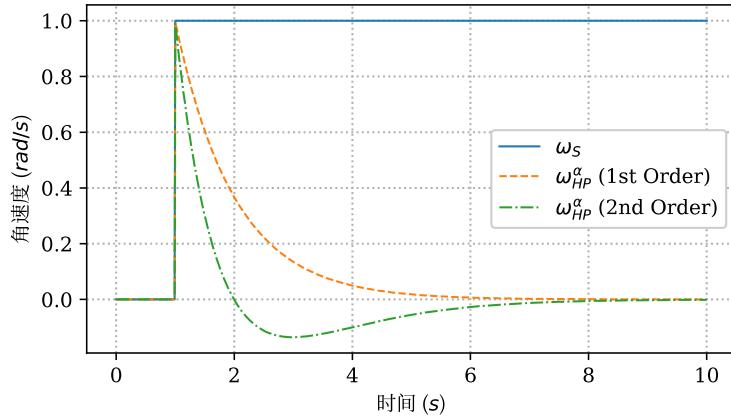


图 2.20 一阶和二阶高通滤波器对阶跃信号输出比较

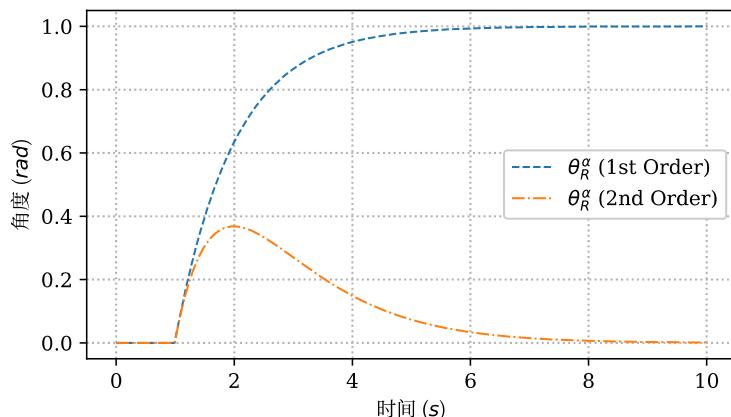


图 2.21 一阶和二阶高通滤波器对阶跃信号输出角度比较

由图 2.20 可见, 对于阶跃信号一阶和二阶滤波器输出的角速度都能在最后回到零。

由图 2.21 可见, 对于阶跃信号一阶滤波器不具备使角度缓慢回到原点的特性, 而二阶滤波

器则能使积分后的角度缓慢回到原点。

2.6 可扩展软件开发

本课题需要开发一个具有高度可扩展性的软件，这与通常用于完成一个特定目标的软件不同，需要具备精心设计的架构和接口以匹配当前的需求并能以较小成本适应未来的变化。达成可扩展性的途径之一是由独立性高的模块组成软件功能。

2.6.1 模块化

模块（Module）是数据说明、可执行语句等程度对象的集合。模块化是将程序划分成若干个模块，每个模块具有一个确定的子功能，这些模块组合而成可以完成指定的功能。在软件系统模块化时，最重要的原理是模块独立程度，其定性衡量标准包括内聚和耦合^[33]。

2.6.2 内聚性和耦合性

内聚（Cohesion）是模块独立程度的衡量标准之一，它衡量模块内部功能之间结合紧密程度^[33]。内聚有以下分类：

- (1) 偶然内聚（Coincidental Cohesion）：模块中各个单元只是刚好放在一起；
- (2) 逻辑内聚（Logical Cohesion）：模块中各个单元逻辑上同类，通过参数确定模块完成的功能；
- (3) 时间内聚（Temporal Cohesion）：模块中各个单元必须在同一段时间内执行；
- (4) 过程内聚（Procedural Cohesion）：模块中各个单元必须按特定次序执行；
- (5) 通信内聚（Communicational Cohesion）：模块中各个单元处理相同的输入数据，或产生相同的输出数据；
- (6) 顺序内聚（Sequential Cohesion）：模块中各个单元的输出和输出彼此相关，像流水线一样执行；
- (7) 功能内聚（Functional Cohesion）：模块中各个单元完成单一明确的功能。

功能内聚是最高程度的内聚，偶然内聚是最低程度的内聚。内聚程度越高、模块独立性越强。

耦合（Coupling）是模块独立程度的另一个衡量标准，它衡量不同模块之间互相依赖程度^[33]。耦合有以下分类：

- (1) 内容耦合（Content Coupling）：一个模块直接使用另一个模块的数据、有一部分代码重叠或通过不正常入口进入另一个模块内部；
- (2) 公共耦合（Common Coupling）：一组模块间都访问同一个全局数据；
- (3) 外部耦合（External Coupling）：一组模块共用数据格式或协议；
- (4) 控制耦合（Control Coupling）：一个模块调用另一个模块时，传递的是控制变量，被调用模块根据控制变量执行某一功能；
- (5) 标记耦合（Stamp Coupling）：几个模块间通过参数传递共享的记录；
- (6) 数据耦合（Data Coupling）：模块间通过参数交换数据；
- (7) 无耦合（No Coupling）：模块完全不和其他模块交换数据。

无耦合是最低程度的耦合，内容耦合是最高等级的耦合。耦合程度越低、模块独立性越强。一般原则是尽可能采取数据耦合，少使用控制耦合，完全不使用内容耦合^[33]。

2.6.3 S.O.L.I.D. 原则

S.O.L.I.D. 是面向对象设计中前五大基本原则的首字母缩写^[34]。这些原则结合在一起能够帮助实现高内聚、低耦合的模块设计并方便程序员开发出易于维护和扩展的软件。

- (1) 单一职责原则 (S - Single Responsibility): 模块应当有且仅有一种单一功能；
- (2) 开放封闭原则 (O - Open/Closed): 模块应当对扩展开放、对修改封闭，即允许在不改变原有源代码的情况下变更行为；
- (3) 里氏替换原则 (L - Liskov Substitution): 对象模块应当可以在不改变程序正确性的前提下被它的子类所替换，即子类应当扩展父类的功能并避免改变父类原有的功能；
- (4) 接口隔离原则 (I - Interface Segregation): 对象的用户应当只关心它感兴趣的方法，即避免庞大臃肿的接口、实现小而具体的接口；
- (5) 依赖倒置原则 (D - Dependency Inversion): 高层模块不应依赖于低层模块的实现，而是依赖于其抽象接口。

2.6.4 IO 模型

在进行模块间通信时，通信的两端往往需要等待另一端的数据，并需要处理多个客户端的连接。以服务端为例，一般有多种等待数据方式，且有多种处理多个客户端连接方式，它们组合起来被称为 IO 模型 (Input/Output Model)。

A. 同步单线程

指采用单线程 (Single Thread) 方式处理连接，且对每一个连接采用同步 (Synchronous) 方式进行 IO。

该模型在单一线程中处理连接，且使用同步方式进行 IO，即发送或接收等 IO 操作会阻塞当前线程，只有 IO 结束后线程才会继续下一步操作。

由于采用了同步的方式，因此该模型只能处理完一个连接后再处理下一个连接，是一种单客户端 IO 模型，也是最简单的 IO 模型。

B. 同步多线程

指采用多线程 (Multi Thread) 方式处理多个连接，且对每一个连接采用同步方式进行 IO。

当有新客户端连接时，该模型启动一个新线程处理该连接。在这个线程中，使用同步方式进行 IO。

同步多线程是大多数编程语言中使用最广泛的一种多客户端 IO 模型，具有开发简便、直观的优点。但由于每个连接都需要开辟一个新线程处理，而每个线程需要消耗显著的系统资源，因此无法应用在并发连接数较大的场景，如无法解决 C10k 问题^[??]。

C. 异步单线程

指采用单线程 (Single Thread) 方式处理多个连接，且对每一个连接采用异步 (Asynchronous) 方式进行 IO。

由于采用异步方式进行 IO，因此处理连接的线程不会被 IO 操作阻塞，从而可以同时处理多个连接。

异步单线程是一种最简单的异步 IO 模型，诸如 libuv, libev, Node.js 等都采用了该模型。由于异步代码不按照代码书写顺序执行，因此相比同步代码来说具有更高的开发难度。该模型由于不需要为每一个连接开辟一个新线程处理，因此在具有相同连接数的情况下资源消耗相比同步多线程更少。但同样由于单线程处理，该模型支持的并发连接数取决于单核处理性能。异步单线程能在少量连接保持活跃的条件下解决 C10k 问题^[35]，但无法处理大部分连接都保持活跃的场景。

D. 异步多线程

指采用多线程的方式处理多个连接，且对每一个连接采用异步方式进行 IO。

该模型相比异步单线程的区别是它采用线程池技术利用多线程处理各个连接，其中每一个线程上都异步地处理多个连接的 IO。

与同步多线程不同的是，它不为每一个连接创建一个新线程，而是保持在一个固定的最大线程数。该技术一般使用不超过计算机逻辑 CPU 个数的线程数以保证每个 CPU 资源利用最大化。

由于采用了多线程技术，因此该模型将异步单线程的连接处理能力扩展到多个核心上。该模型能完整利用多核计算机的计算资源，且相比同步多线程有更高的处理能力，是一种 C10k 问题的解决方案，但其编程复杂度也是最高的^[36]。

装
订
线

3 软件需求分析及概要设计

3.1 需求综述

3.1.1 功能性需求

A. Microsoft Flight Simulator X 体感模拟

经过配置后，用户能在使用 Microsoft Flight Simulator X（简称 FSX）飞行模拟软件时，实时地根据当前飞行状态在一个运动平台上产生对应的体感效果。若飞行暂停或停止，则不再产生体感效果。

B. 预制飞行动作体感模拟

用户能直接选择进行飞机起飞、降落、左偏航、右偏航的体感模拟，选择某一项后能在运动平台上产生对应的体感效果。

C. 模拟六自由度运动平台

用户能看到一个模拟的六自由度运动平台实时根据运动状态调整姿态和位置。该模拟平台需要能复现真实平台的情况，并支持调节上下平台半径、短边圆心角、上下平台间距等平台参数，供用户测试各种参数情况下的模拟情况。

D. 内置体感模拟算法半自动调参

内置基于经典洗出算法的体感模拟算法。用户能根据硬件情况、模拟载具运行情况等调整模拟算法的参数，以便达到更好的体感模拟效果。

E. 开发并使用输入插件

允许开发者将其他数据源接入体感模拟系统从而实现体感模拟。数据源需至少支持三个方向线性加速度、三个方向角速度的实时反馈。用户能自行选择数据源。

F. 开发并使用体感模拟算法

允许开发者实现其他类型的体感模拟算法并提供参数配置界面。允许用户安装并使用不同的体感模拟算法进行体感模拟。

G. 开发并使用运动平台驱动程序

允许开发者将运动姿态情况与一种真实硬件型号连接起来，实现运动平台驱动。允许用户安装并使用不同的运动平台驱动。

H. 开发并使用输出插件

允许开发者获取当前运动平台应有的运动姿态、载具的姿态等数据并做其他用处，如实现与视觉系统的连接。允许用户安装并使用普通的输出插件。

3.1.2 运行环境

表 3.1 描述了保证软件正常运行所需的最低限度的环境要求。

表 3.1 正常运行所需最低环境

组件	内存	CPU	操作系统	网络延迟	网络带宽
FSX 插件	4GB+	2+	Windows 7+	< 10 ms	≥ 1 Mbps
硬件驱动组件	1GB+	2+	和硬件驱动程序一致	< 10 ms	≥ 1 Mbps
其他所有组件	4GB+	2+	Windows 7+ Ubuntu 14.04+	< 10 ms	≥ 5 Mbps

3.2 总体设计

3.2.1 数据流图

本课题软件的数据流图如图 3.1 和图 3.2 所示。

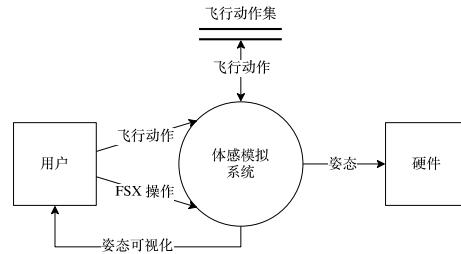


图 3.1 顶层数据流图

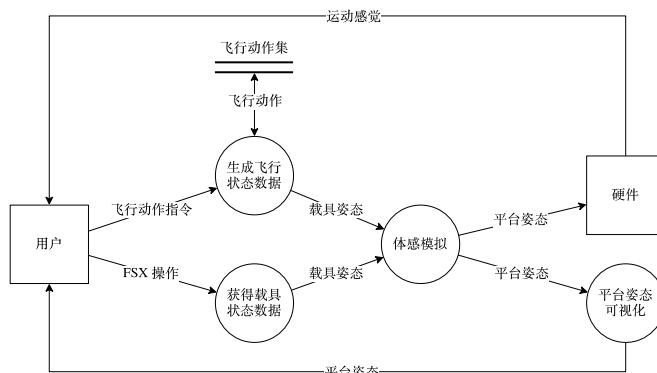


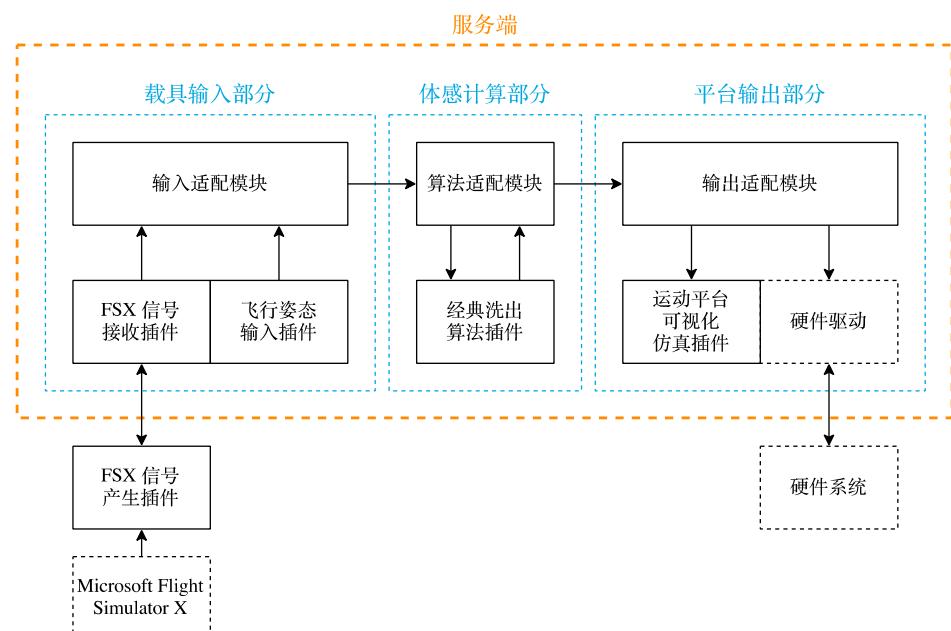
图 3.2 第 0 层数据流图

3.2.2 架构

本课题软件由三部分组成：载具输入部分、体感计算部分、平台输出部分。载具输入部分产生载具状态，体感计算部分根据载具状态使用算法计算出平台状态，平台输出部分根据平台状态控制具体硬件、虚拟硬件或传递给其他软件如视觉模拟系统。数据从载具输入部分产生或输入，流动到体感计算部分进行计算，最后计算结果流动到平台输出部分进行输出，如图 3.3 所示。其中，每个部分都是可替换、可配置、可扩展的，且载具输入和体感计算部分只有单个、平台输出部分可能有多个。



为了在工程上能够实现这样的设计，架构中利用设计模式中的适配器模式设计模块，设计出的总体架构如图 3.4 所示。为了便于从整体上理解，此处展示的是高层模块划分，即模块的功能具体将由一个或多个类共同配合完成，而不是一个模块对应一个类。



装
订
线

以下具体解释每个模块的功能。

A. 输入适配模块

该模块负责发现、管理和根据用户的配置对接具体的一个输入模块。对接包括接受输入模块输入的实时数据、接受模块定义的配置界面、传递用户对模块的配置等。

B. FSX 信号产生插件

该模块实时从一个当前系统里正在运行的 Microsoft Flight Simulator X 实例中读取飞机位置和姿态数据作为载具输入状态。

C. FSX 信号接收插件

由于 FSX 运行在 Windows 系统上。为了允许让服务端不限于该操作系统，“FSX 信号产生插件”不属于服务端的组成部分。服务端仅包含“FSX 信号接收插件”，即一个与“FSX 信号产生插件”相匹配的远程模块，它和“FSX 信号产生插件”远程通信实时获取到数据。另外，有关“FSX 信号产生插件”的配置也由该模块在服务端统一提供界面并管理。通过这种方式，用户

在服务端一处即可完成各种配置。

D. 飞行姿态输入插件

该模块内置了起飞、降落、左右偏航四种飞机状态下的位置和姿态的变化关系，用户可在在一个状态变化完毕后通过界面切换到另一个状态。这些状态将实时地作为载具输入状态。另外该模块还提供了相应的配置功能，允许用户新定义其他状态数据或修改现有飞机状态数据。

E. 算法适配模块

该模块负责发现、管理和根据用户的配置对接一个具体的算法模块。对接包括将载具输入实时传递给算法模块计算、实时从算法模块取得计算后的平台状态数据、接受模块定义的配置界面、传递用户对模块的配置等。

F. 经典洗出算法插件

该模块提供了基于经典洗出算法的体感模拟，实时地根据载具状态计算出平台状态数据。允许通过配置调节参数。

G. 输出适配模块

该模块负责发现、管理和根据用户的配置对接一个具体的输出模块。对接包括将平台状态数据传递给输出模块、接受模块定义的配置界面、传递用户对模块的配置等。

H. 运动平台可视化仿真插件

该模块根据平台姿态情况，实时地三维仿真呈现一个 Stewart 六自由度运动平台的位置和姿态。用户可以从任意角度观察这个模拟运动平台。利用该模块，用户不需要硬件也能观察体感模拟情况，分析输入、计算等步骤的运行情况。用户还可以将该仿真运动平台与真实平台运行状态进行比较，为参数调节给出参考。

I. 硬件驱动

该模块是一组假想模块，是用于驱动具体的运动平台硬件所需的对接模块，其职责类似于操作系统中的驱动程序，需要开发者或硬件提供商开发。由于本课题没有合适的运动平台硬件，因此这里不提供硬件驱动的样例模块。

该模块的几种常见场景：

- (1) 若具体硬件是一个通过串口通信、直接接受平台位置和姿态数据的一个高级六自由度运动平台硬件，则驱动模块应当实现和该硬件进行串口通信，传递平台姿态和位置数据；
- (2) 在 (1) 基础上进一步地，若与该硬件的通信只能在 Windows 操作系统中进行，则需要拆分驱动模块为两部分，即一个远程部分和一个控制部分，用于保证服务端可以跨平台运行；
- (3) 若具体硬件是一组用户自己组装的传动杆 (Actuator)，传动杆只通过串口接受位置和速度的控制，组成了一个低级六自由度运动平台硬件，则驱动模块应当实现位置反解并和这些传动杆进行串口通信，传递反解后传动杆位置数据。

3.2.3 功能性需求与模块关系

根据功能性需求，各个需求与模块之间的关系如表 3.2 所示。

表 3.2 功能需求与模块关系

功能 \ 模块	输入适配	FSX 信号产生	FSX 信号接收	飞行姿态 输入	算法适配	经典洗出	输出适配	运动平台	可视化仿真	硬件驱动
开发使用输出插件	-	-	-	-	-	-	✓	-	-	-
开发使用运动平台驱动	-	-	-	-	-	-	✓	-	-	✓
开发使用体感模拟算法	-	-	-	-	-	-	-	-	-	-
开发使用输入插件	✓	-	-	-	-	-	-	-	-	-
体感算法及调参	-	-	-	-	✓	-	-	-	-	-
模拟 6DOF 运动平台	-	-	-	-	✓	-	-	✓	-	-
飞行动作体感模拟	✓	-	-	✓	✓	✓	✓	✓	-	-
FSX 体感模拟	✓	✓	✓	-	✓	✓	✓	✓	-	-

4 体感模拟算法的软件实现研究

现有体感模拟算法的研究大多建立在连续量系统上，通过采用 Laplace 变换求解微分方程计算。而基于软件实现的体感模拟器则是一个采样系统，输入的体感数据是离散量。体感模拟算法的软件实现，就是需要建立起连续算法等效的离散算法。本课题对此进行了研究实现。

体感模拟算法一般包括连续量系统上的模拟滤波器和积分组成，以下分别阐述其在离散量系统上实现的算法。

4.1 模拟滤波器的数字实现

对离散数据的处理实际上是数字处理领域。双线性变换（Bilinear Transform）是一种标准的将连续滤波器转换为等效的数字滤波器的方法。

双线性变换公式为：

$$H(z) = H(s) \Big|_{s=2f_s \frac{z-1}{z+1}} \quad (4.1)$$

其中， f_s 是频率。

利用双线性变换，可以将模拟滤波器传递函数转换为数字滤波器传递函数：

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \dots + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \dots + a_M z^{-M}} \quad (4.2)$$

接下来可直接利用数字滤波器差分计算公式进行计算：

$$\begin{aligned} y_m &= b_0 x_m + b_1 x_{m-1} + \dots + b_P x_{m-P} \\ &\quad - a_1 y_{m-1} - a_2 y_{m-2} - \dots - a_Q y_{m-Q} \end{aligned} \quad (4.3)$$

其中， x 是输入信号；

y 是滤波器输出。

以本文采用的 x 方向上的二阶低通滤波器为例，式 (2.24) 填入表 2.2 中参数可得模拟滤波器传递函数：

$$H(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} = \frac{25}{s^2 + 10s + 25} \quad (4.4)$$

以 100Hz 作为频率 f_s ，根据式 (4.1) 带入可得：

$$H(z) = \frac{25}{\left(200 \frac{z-1}{z+1}\right)^2 + 10 \cdot 200 \frac{z-1}{z+1} + 25} \quad (4.5)$$

整理得：

$$H(z) = \frac{z^2 + 2z + 1}{1681z^2 - 3198z + 1521} \quad (4.6)$$

根据式 (4.2) 对系数进行规格化可得数字滤波器传递函数：

$$\begin{aligned}
 H(z) &= \frac{0.000596z^2 + 0.001190z + 0.000596}{z^2 - 1.902439z + 0.904819} \\
 &= \frac{0.000596 + 0.001190z^{-1} + 0.000596z^{-2}}{1 - 1.902439z^{-1} + 0.904819z^{-2}}
 \end{aligned} \tag{4.7}$$

因此根据式 (4.3), 该滤波器差分公式为:

$$\begin{aligned}
 y_m &= 0.000596x_m + 0.001190x_{m-1} + 0.000596x_{m-2} \\
 &\quad - (-1.902439)y_{m-1} - 0.904819y_{m-2}
 \end{aligned} \tag{4.8}$$

根据式 (4.3), 代码 4.1 实现了滤波器计算类, 其初始化参数为数字滤波器系数参数, 提供的接口是传入一个新的值作为输入信号并计算输出响应。

```

class RealtimeFilter():
    def __init__(self, b, a):
        assert(len(b) == len(a))
        self.n = len(b) # n = order + 1
        self.b = b
        self.a = a
        self.reset()

    def reset(self):
        self.input = np.zeros(self.n, dtype=np.float)
        self.output = np.zeros(self.n, dtype=np.float)

    def apply(self, v):
        self.input[self.n - 1] = v
        self.output[self.n - 1] = 0
        output = 0
        for i in range(0, self.n):
            output = output + \
                self.b[i] * self.input[self.n - 1 - i] - \
                self.a[i] * self.output[self.n - 1 - i]
        self.output[self.n - 1] = output
        for i in range(0, self.n - 1):
            self.input[i] = self.input[i+1]
            self.output[i] = self.output[i+1]
        return output

```

代码 4.1 数字滤波器计算类的实现

为了方便用户调整参数, 应当直接以模拟滤波器参数作为参数, 需要计算得出数字滤波器参数, 则需要软件实现双线性变换计算。利用 Python 科学计算库 `scipy` 可简便而高效地完成该计算, 如代码 4.2 所示。其中, 输入参数 `b_analog`, `a_analog` 分别是模拟滤波器传递函数中分子和分母的系数; 输入参数 `fs` 是频率; 输出 `b_digital`, `a_digital` 分别是双线性变换后传递函数中分子和分母的系数。

```
b_digital, a_digital = scipy.signal.bilinear(b_analog, a_analog, fs)
```

代码 4.2 模拟滤波器转数字滤波器示例代码

仍以式(4.4)所示传递函数为例,将其通过双线性变换转换为数字滤波器的代码如代码4.3所示,运行结果如代码4.4所示,该运行结果和式(4.7)中计算得出的系数是一致的。

```
b_analog = [0, 0, 25]
a_analog = [1, 10, 25]
b_digital, a_digital = signal.bilinear(b_analog, a_analog, 100)
```

代码4.3 将式(4.7)所示滤波器转换为数字滤波器的代码

```
b_digital: [ 0.00059488  0.00118977  0.00059488]
a_digital: [ 1.          -1.90243902  0.90481856]
```

代码4.4 数字滤波器输出结果

基于代码4.2,分别实现一阶到三阶高通和低通滤波器的工厂函数,以模拟滤波器为参数构造数字滤波器计算类,如代码4.5所示。

```
def build_1st_filter(omega, lp=True, freq=FREQ):
    b = [1, 0] if not lp else [0, omega]
    a = [1, omega]
    return RealtimeFilter(*signal.bilinear(b, a, fs=freq))

def build_2nd_filter(omega, zeta, lp=True, freq=FREQ):
    b = [1, 0, 0] if not lp else [0, 0, omega ** 2]
    a = [1, 2 * zeta * omega, omega ** 2]
    return RealtimeFilter(*signal.bilinear(b, a, fs=freq))

def build_3rd_filter(omega, zeta, omega_1, lp=True, freq=FREQ):
    b = [1, 0, 0, 0] if not lp else [0, 0, 0, omega ** 3]
    a = [1, 2 * zeta * omega + omega_1, omega ** 2 + omega_1 * 2 * zeta * omega, omega ** 2 * omega_1]
    return RealtimeFilter(*signal.bilinear(b, a, fs=freq))
```

代码4.5 数字滤波器计算类的工厂函数实现

4.2 积分的数字实现

体感模拟算法一般需要进行积分以便将加速度转成位移、将角速度转成角度。根据定积分近似计算公式,可以将函数利用矩形切割近似计算:

$$\int_I f(t)dt \approx \sum_{i=0}^{n-1} (x_{i+1} - x_i)f(c_i) \quad (4.9)$$

对于式(4.8)中得到的加速度二阶低通滤波器的差分函数,应用近似计算公式可得离散情况下的速度表达式:

$$v_t = \sum_{t=0}^{t-1} \frac{1}{f_s} y_t = \sum_{t=0}^{t-1} 0.01 y_t \quad (4.10)$$

再一次积分可得离散情况下的位移表达式：

$$S_t = \sum_{t=0}^{t-1} \frac{1}{f_s} v_t = \sum_{t=0}^{t-1} 0.01 v_t \quad (4.11)$$

4.3 算法复杂度分析

对体感模拟器来说，需要确保算法能在很短的实现内完成计算，这样才能实现运动平滑顺畅，因此算法复杂度分析尤为重要。

滤波器计算包括模拟滤波器转数字滤波器、数字滤波器计算两个步骤。由于参数是在体感模拟前事先确定的，因此模拟滤波器转数字滤波器这一步骤对系统性能没有影响，这里不作考虑。对于数字滤波器计算步骤，容易发现其对每个需要计算的信号时间和空间复杂度都为 $O(1)$ ，效率极高。

对于积分计算步骤，若以式 (4.10) 给出的方法直接计算，则时间复杂度和空间复杂度都为 $O(N)$ (N 为当前已采样点的个数)，随着采样点越来越多，消耗的资源也将越来越多，是不能接受的。注意到该积分表达式等价于递推关系：

$$v_t = \begin{cases} v_{t-1} + \frac{1}{f_s} y_{t-1}, & t > 0 \\ 0, & t = 0 \end{cases} \quad (4.12)$$

利用该递推关系可计算出相同结果，但其时间和空间复杂度也都为 $O(1)$ ，效率大大提高。利用该递推关系对式 (4.11) 的实现如代码 4.6 所示。

```
output_v[i] = output_v[i-1] + 0.01 * output[i-1];
output_s[i] = output_s[i-1] + 0.01 * output_v[i-1];
```

代码 4.6 式 (4.11) 的等价 C 语言示例代码

5 服务端基础框架详细实现

5.1 服务端技术和组件选择

本课题服务端包括载具输入部分模块、体感计算部分模块、平台输出部分模块。本课题服务端设计具有以下特性：

- (1) 跨平台运行；
- (2) 支持以 Web 形式呈现的界面用于模块配置；
- (3) 易于进行模块替换、扩展；
- (4) 高效率；

以下分别列举本课题服务端基于的组件和选择该组件的因素。

A. Python 3.5+ (编程语言)

- (1) 作为一门跨平台编程语言，能够满足服务端跨平台的要求；
- (2) 从 3.5 版本开始，支持异步及其同步化书写的语法，可以以直观代码编写异步的高性能程序；
- (3) 拥有众多科学计算库，能非常方便将模拟滤波器转换为数字滤波器；
- (4) 作为脚本语言，为其编写插件时无需编译，具有极强扩展性。

B. Sanic (Web 框架)

- (1) 基于 Python 3.5 异步特性实现；
- (2) 效率是同类框架中最高的；
- (3) 功能全面。

C. MongoDB (数据库)

- (1) 文档型数据库，无需事先定义记录结构；
- (2) 支持异步操作；
- (3) 可持久化；
- (4) 高效率。

5.2 服务端配置界面技术和组件选择

A. Web 技术

本课题利用 Web 技术实现配置界面。

- (1) 相比其他界面技术，Web 技术动态性高，为轻松实现插件化界面提供了有力支持；
- (2) 能轻松实现美观的用户界面；
- (3) 配置界面多为表单类型，现有 Web 组件够用，无需客户端界面那样复杂的组件；
- (4) 允许软件服务端运行在不具有图形化界面的操作系统中，如主流服务器采用的服务器版 Linux 操作系统。

B. Webpack (前端打包工具)

- (1) 可将插件所用资源打包在一处目录下，减少了插件分发的难度；

(2) 只需引入一个打包出来的 JavaScript 文件即可自动完成其他资源的加载;

(3) 允许开发者以较低难度进行模块化前端开发。

C. Vue (前端 MVVM 渲染框架)

(1) 配置界面多为表单界面, 因此使用 MVVM 框架可以大大减轻开发难度;

(2) 自带友好的模板引擎, 代码可读性较高;

(3) 支持组件功能, 允许开发者将各个界面控件分离实现并实现界面控件复用。

5.3 事件驱动系统实现

5.3.1 模块数据流

为了实现插件和模块松耦合, 本课题服务端利用观察者设计模式 (Observer Design Pattern) 的思想, 采用事件驱动的架构进行不同类别模块间的通信, 包括插件与适配模块之间、适配模块与适配模块之间等。关键模块间数据流图 5.1 如所示, 数据流动的顺序是从①到⑥。

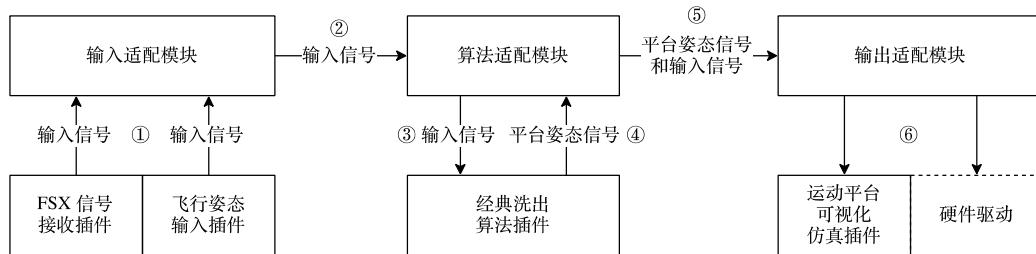


图 5.1 关键模块数据流图

5.3.2 事件列表

除了模块间事件, 框架也提供了事件。本课题服务端定义的所有事件如表 5.1 所示。

表 5.1 服务端事件列表

名称	描述	发出者	接受者	DFD 图编号
hexi.pipeline.start	体感模拟开始	框架	任意	/
hexi.pipeline.stop	体感模拟结束	框架	任意	/
hexi.pipeline.input.	输入信号, 内容包括被模拟载具加速度等信息, 在模拟开始后以毫秒级触发	输入插件	输入适配	①
raw_data	速度等信息, 在模拟开始后以毫秒级触发			
hexi.pipeline.input.data	经过统一处理的输入信号	输入适配	算法插件	②③
hexi.pipeline.mca.raw_data	平台位置姿态信号, 是输入信号经过体感计算模块计算后产生的信号, 内容包括输入信号数据和计算后的平台位置姿态数据	算法插件	算法适配	④

续表 5.1

名称	描述	发出者	接受者	DFD 图编号
hexi.pipeline.mca.data	经过统一处理的平台位置姿态信号	算法适配	输出插件	⑤⑥
hexi.start	软件启动	框架	任意	/
hexi.stop	软件关闭	框架	任意	/
hexi.plugin.activate	一个插件被激活	框架	任意	/
hexi.plugin.deactivate	一个插件被取消激活	框架	任意	/
hexi.plugin.load	一个插件被加载	框架	任意	/

5.3.3 具体实现

事件驱动的核心是事件调度模块（Event Dispatcher），即其他模块向该模块订阅事件，或要求向已订阅事件的模块广播一个事件。本课题采用异步架构，因此相应地实现了异步事件调度模块。在该调度模块中，广播消息操作是异步进行的：订阅者将同时收到广播消息，在所有订阅者都处理完消息后，广播消息操作才结束。由于异步特性，多个订阅者可以同时进行异步 IO 操作，提高了处理效率。

```

_subscribers = {}

async def publish(key, value):
    coroutines = [subscriber({'key': key, 'value': value})
                 for subscriber, key_set in _subscribers.items()
                 if key in key_set]
    await asyncio.gather(*coroutines)

def subscribe(callback, keys):
    assert type(keys) in (set, list, tuple)
    _subscribers[callback] = keys

def unsubscribe(callback):
    if callback in _subscribers:
        del _subscribers[callback]

```

代码 5.1 异步事件调度器的实现

以 hexi.pipeline.input.data 为例。若不关心事件在何时被处理完毕，可使用代码 5.2 广播事件；若需要在事件处理完毕后异步地执行操作，可使用代码 5.3 广播事件；若需要同步地等待事件处理完毕进行后续操作，可使用代码 5.4 广播事件；可使用代码 5.5 订阅事件。

```
asyncio.ensure_future(event.publish('hexi.pipeline.input.data', param))
```

代码 5.2 利用异步事件调度器异步广播事件、不关心广播结果

```
def on_task_done(future):
    # HANDLE EVENT PUBLISHED
    pass

task = asyncio.ensure_future(event.publish('hexi.pipeline.input.data', param))
task.add_done_callback(on_task_done)
```

代码 5.3 利用异步事件调度器异步广播事件、关心广播结果

```
loop = asyncio.get_event_loop()
loop.run_until_complete(event.publish('hexi.pipeline.input.data', param))
```

代码 5.4 利用异步事件调度器同步广播事件

```
async def on_input_data(e):
    # HANDLE EVENT ARRIVAL
    pass

event.subscribe(on_input_data, ['hexi.pipeline.input.data'])
```

代码 5.5 利用异步事件调度器订阅事件

5.4 插件系统实现

可替换、可扩展的模块即插件。服务端的插件系统是本课题软件实现部分需要重点解决的问题之一，现有体感模拟软件正是由于其架构上不够开放完备、无法支撑插件系统，因而不具有本课题的研究意义。为了实现可配置，插件系统还需支持界面。本课题插件系统实现了：

- (1) 逻辑层插件机制。例如对于一个输入插件，插件产生的输入信号可以正常流动到用户配置的一个计算组件并最终流动到用户配置的输出组件。
- (2) 界面层插件机制。例如对于(1)中的输入插件，它还可以提供配置界面，用户在界面上能够对其进行配置，实现模块的可配置性。
- (3) 插件界面统一。各个独立插件提供的界面组合起来具有连贯性、统一性，即对于用户来说，能以同样的入口对不同插件进行配置，而不是通过散落在各处的入口进行配置。
- (4) 开箱即用。用户能以极少的操作安装一个插件或删除一个插件，而不需要经过复杂的流程（如代码编译）。

5.4.1 插件形式

规定插件以服务端 `plugins` 目录下一个目录为单位，目录中包括插件描述文件、逻辑层代码、界面层代码。

当用户需要安装新插件时，只需将新插件目录放置在 `plugins` 下即可生效。若要删除插件，则将插件在 `plugins` 目录下的目录删除即可。上述“安装新插件”和“删除插件”步骤可通过界面进行友好实现。

A. 插件描述文件

规定插件描述文件描述了关于插件的所有元信息，包括插件唯一标识符 (Id)、插件名称

(Name)、插件类别 (Category)、插件版本 (Version)、插件描述 (Description)、逻辑层代码入口 (Module) 等。唯一标识符必须和插件目录名称一致。

规定描述文件必须以 .plugin 作为扩展名，名称任意。一个插件只能有一个描述文件。代码 5.6 是一个描述文件的样例。

```
[Core]
Id = input_fsx
Category = input
Name = FSX 输入插件
Module = plugin

[Documentation]
Author = Built-in
Version = 0.1
Description = 对 Microsoft Flight Simulator X 中的飞机进行体感模拟
```

代码 5.6 插件描述文件样例

装
订
线

B. 插件逻辑层代码

规定插件逻辑层代码使用 Python 语言编写，是一个普通的 Python 模块。逻辑层的入口文件由描述文件给出。由于 Python 语言限制，插件目录名和插件唯一标识符只能使用字母、数字和下划线。

C. 插件界面层代码

规定插件界面层代码位于.ui_built 目录下，且入口文件为 .ui_built/main.js。该入口文件将被服务端按照插件唯一标识符映射到一个可访问的 Web 路径下从而被加载。一般，插件开发者使用课题给出的 Webpack 配置将界面层中以相对路径给出的源代码及相关资源编译、打包、处理输出到界面层所需的.ui_built 目录下。

5.4.2 插件发现

插件发现指找出当前所有已安装插件并获取它们的描述信息。该步骤需遍历插件目录 plugins 下所有子文件夹，找到插件描述文件 *.plugin 并对其进行解析。本课题基于 Python 的 yapsy 插件库实现上述逻辑，核心逻辑如代码 5.7 所示。

```
def loadPlugins():
    pm.setPluginPlaces(['./plugins'])
    pm.setPluginInfoExtension('plugin')
    pm.setCategoriesFilter(pm.pluginsFilter)
    pm.collectPlugins()
    for plugin in pm.getAllPlugins():
        try:
            id = plugin.details.get('Core', 'Id')
        except configparser.NoOptionError:
            _logger.error('Plugin `{}` is ignored because of missing valid `Id`'
                         'property.'.format(plugin.name))
            try:
                category = plugin.details.get('Core', 'Category')
```

```

assert(category in pluginsByCategory.keys())
except configparser.NoOptionError:
    _logger.error('Plugin `{0}` is ignored because of missing valid `Category`'
property.'.format(plugin.name))

pluginsByCategory[category].append(plugin)
pluginsById[id] = plugin

```

代码 5.7 利用 yapsy 发现并加载插件

该代码执行完毕后，pluginsById 变量即包含了以插件 ID 作为键、插件对象作为值的键值对，且各个插件按照描述文件中给出的类别 Category 进行分类，方便后续快速查询某一类插件。

5.4.3 插件逻辑层加载

利用 Python 动态语言特性，插件包含的代码可以被动态地执行，且可以与服务端其他模块进行互操作。因此，只需根据描述文件，载入其所指定的入口模块即可。根据 yapsy 要求，入口模块应当遵循以下接口，如图 5.2 所示。

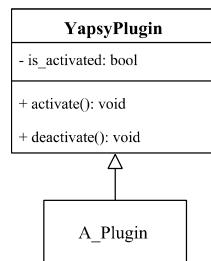


图 5.2 yapsy 插件类图

(1) activate(): 插件被用户激活时被调用。可以开始与其他模块进行通信。

(2) deactivate(): 插件被用户禁用时被调用。应当终止与其他模块通信。

在 yapsy 中，pm.collectPlugins() 中已包含逻辑层的加载功能。

5.4.4 插件逻辑层基类

输入插件职责是产生输入信号，算法插件职责是根据输入信号产生平台姿态信号，输出插件的职责是接受平台姿态信号再进行进一步处理。基于此，在图 5.2 所示基类基础上，再分别为它们封装实现基类方法，类图如图 5.3 所示。其中，BasePlugin 基类是本课题定义的所有插件的基类，它封装了对当前插件配置的加载和保存，提供当前插件路由对象，并提供插件描述文件内容。InputPlugin 类是所有输入插件的基类，封装了事件分发函数；MCAPPlugin 类是所有算法插件的基类，封装了事件分发函数和事件接收函数；OutputPlugin 类是所有输出插件的基类，封装了事件接收函数。这些对事件接收和分发的封装隐藏了架构上事件驱动的具体细节，提供了更高阶、抽象的接口形式。

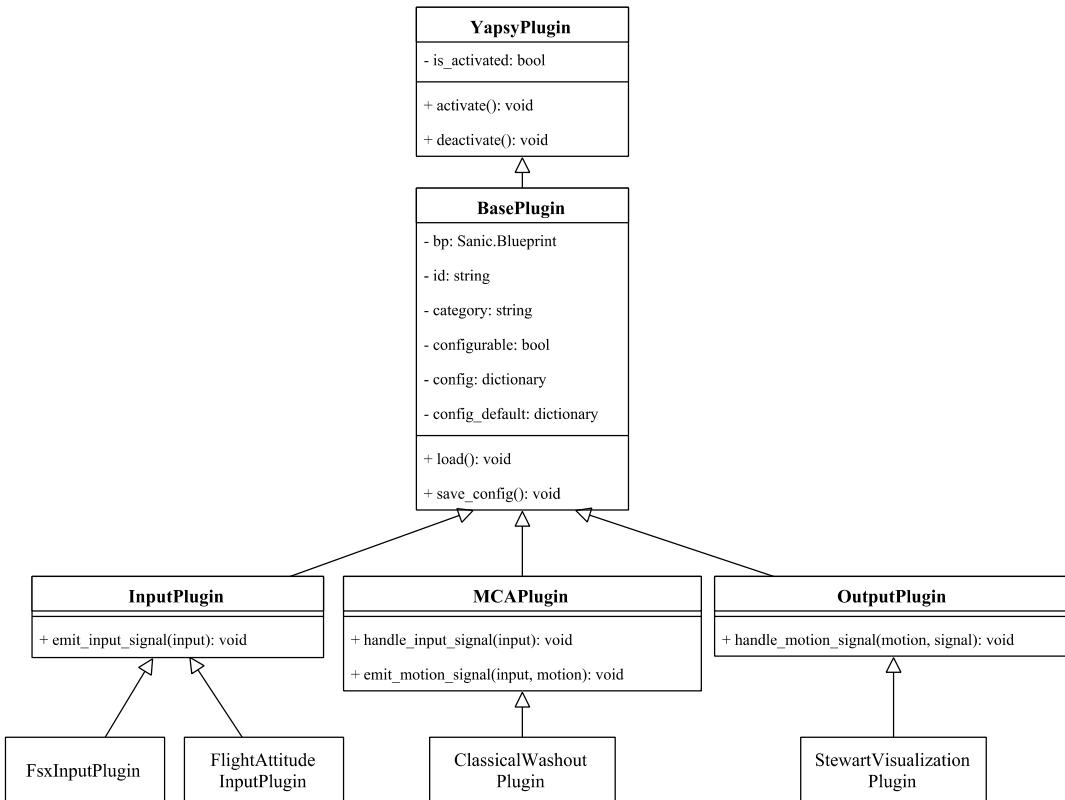


图 5.3 插件类图

装
订
线

5.4.5 插件界面层加载

对于前端来说，若要成功加载一段 JavaScript 代码文件包括两个任务：

- (1) 允许代码文件通过特定 URL 被访问到；
- (2) 已知 URL 的前提下，在 HTML 文件中通过 `<script>` 标签引入该代码文件，或通过 JavaScript 动态地引入。

同理，上述两个任务也适用于在前端加载其他资源，如样式、图片、字体等。但由于采用了 Webpack 对界面层代码进行打包处理，因此样式、图片、字体等其他资源已被打包在了 JavaScript 代码内，或已包含加载该资源所需代码，因此只需确保：

- (1) JavaScript 代码能通过 URL 被访问且在 HTML 中引入；
- (2) 对于没有包括在 JavaScript 内的资源，如图片、字体等二进制内容，允许其通过 URL 被访问到；

即可实现完整的界面层加载。又由于 Webpack 打包出来的 JavaScript 代码和代码以外的资源文件在同一个目录下，因此实现上只需确保：

- (1) 资源目录下各个文件可被访问；
- (2) 在 HTML 中引入 JavaScript URL。

接下来描述上述两个步骤的具体实现。

A. 资源目录下文件可访问

该步骤暗含的要求是不同插件的资源文件 URL 能被区分开来，不会互相冲突。当然，插件的资源文件 URL 也不能和软件框架自身涉及到的 URL 相冲突，因此定义如表 5.2 所示的静态路由规则。

表 5.2 本课题定义的静态路由规则

URL 路由	文件系统路径	描述
/core/static	./hexi/.ui_builtin	框架资源目录
/plugins/[plugin_id]/static	./plugins/[plugin_id]/.ui_builtin	插件资源目录

在该路由规则下，只需各个插件具备独立的标识符，即 plugin_id，即可确保资源文件访问不会冲突，实现如代码 5.8 所示。

```
bp = Blueprint('core', url_prefix='/core')
bp.static('/static', 'hexi/.ui_builtin')
app.blueprint(bp)

bp = Blueprint('plugin-{0}'.format(id), url_prefix='/plugins/{0}'.format(id))
bp.static('/static', os.path.join(os.path.dirname(plugin.path), '.ui_builtin'))
app.blueprint(bp)
```

代码 5.8 为框架和插件创建不同路由

B. 在 HTML 中引入 JavaScript URL

在本课题中，由于插件列表预先未知，取决于用户安装数量，因此需要动态的 HTML 内容。一般情况下，可使用模板引擎实现。但在本课题中，使用模板引擎代价过高，其阻塞式的渲染会影响软件其余部分逻辑的顺畅运行，因此不采用模板引擎，而采用静态 HTML 内容结合动态脚本实现动态加载。其中动态加载分为同步方式和异步方式，由于界面层需要在所有插件加载完毕后进行操作，因此采用同步加载方式较为便利。

a. 静态 HTML 内容

静态 HTML 内容中相关代码如代码 5.9 所示，其中第 1、2 行会加载框架，第三行会加载所有插件，loadPlugins.js 的内容由服务端动态生成。

```
<script src="/core/static/core_dll.js"></script>
<script src="/core/static/main.js"></script>
<script src="/core/plugin/loadPlugins.js"></script>
```

代码 5.9 静态 HTML 关键代码

b. 动态 loadPlugins.js 内容

loadPlugins.js 需要包含同步加载当前所有插件界面部分的代码，利用 document.write() 函数写入<script>标签实现。其内容由代码 5.10 动态地根据当前所有插件计算生成。假设当前已安装

插件的 ID 有 input_flight_attitude 和 input_fsx，则生成的 loadPlugins.js 内容如代码 5.11 所示。

```
bp = Blueprint('plugin', url_prefix='/core/plugin')

@bp.route('/loadPlugins.js')
async def get_plugins (request):
    pluginIdList = [plugin.details.get('Core', 'Id')
        for plugin in pm.getAllPlugins()]
    responseText = 'var EXTERNAL_PLUGINS = {0};\n'.format(json.dumps(pluginIdList));
    for id in pluginIdList:
        responseText += ('try{{document.write(\'<script'
    src="/plugins/{0}/static/main.js"></script>\');}}catch(e){{}}
    responseText += ('});}catch(e){{}'
    return text(responseText, content_type='application/javascript')
```

代码 5.10 生成 loadPlugin 的代码

```
var EXTERNAL_PLUGINS = ["input_flight_attitude", "input_fsx"];
try{document.write('<script'
src="/plugins/input_flight_attitude/static/main.js"></script>');}catch(e){}
try{document.write('<script'
src="/plugins/input_fsx/static/main.js"></script>');}catch(e{})
```

代码 5.11 loadPlugin 内容样例

5.4.6 插件界面层与框架界面层通信

插件加载完毕后，需要能在一个框架级别的界面上统一提供插件配置入口。例如，插件可能需要在侧边栏添加入口，也可能需要在“插件列表”界面添加入口等。该特性利用适配器设计模式实现，要求插件界面入口实现如表 5.3 所示的接口。其中，Route 和 Menu 的类结构如图 5.4 所示。

所有项都包含自己的唯一名称（Base.name）、父项唯一名称（Base.parent）及排序权值（Base.order），以扁平的方式存储支持优先级的树形结构数据，如图 5.5 所示。

表 5.3 插件界面层接口约定

接口名	签名	描述
registerRoutes	void (Route[] routes)	插件在此函数中注册路由，路由中可指定界面的父界面
registerSidebarMenus	void (Menu[] menus)	插件在此函数中注册侧边栏入口，可指定侧边栏项的父项

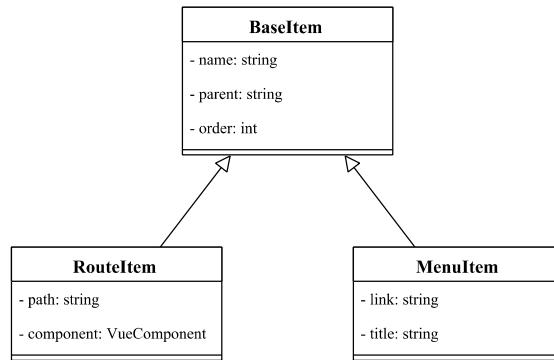


图 5.4 路由项和菜单项类图

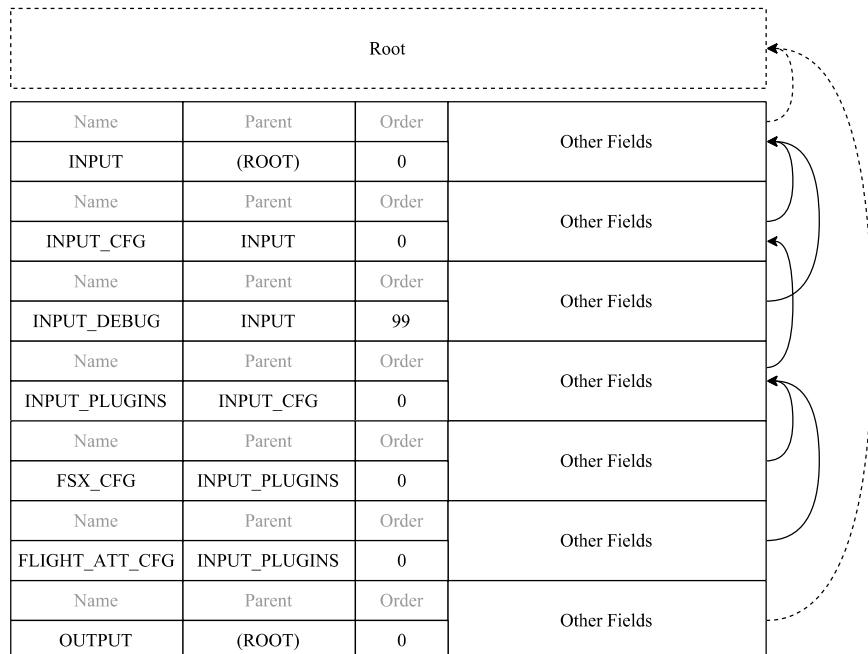
装
订
线

图 5.5 扁平存储的树形结构

对于插件来说，由于加载顺序不同，依赖的父级可能尚未加载，因此插件自身以扁平结构弱定义父级、在所有插件加载完毕后统一将其转换成树形结构是更合理的方案，故每个项自身需定义名称和父级名称。另外，路由和侧边栏有优先级顺序，早加载的插件可能希望在结构末尾增加入口，因此每个项还可选地支持指定排序权值。

对于框架来说，最终希望以树形结构进行递归式的处理，如图 5.6 所示。代码 5.12 利用函数式编程思想，简洁地将图 5.4 描述的扁平结构转换成了树形结构。

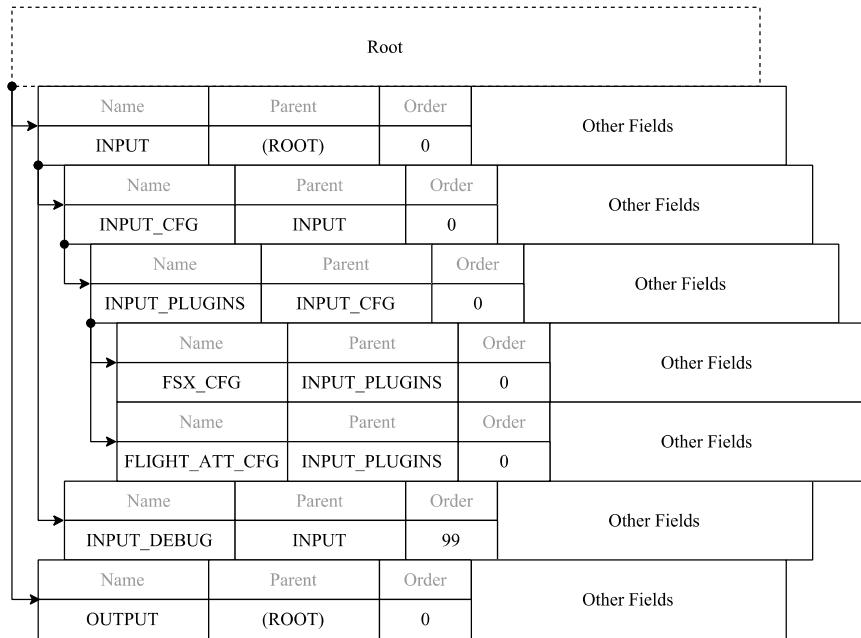


图 5.6 树形结构

```

import _ from 'lodash';

export default function buildTreeFromPlain(list, idField = 'name', orderField =
'order', parentField = 'parent') {
  const sortedList = _(list)
    .map(item => {
      item[orderField] = item[orderField] || 0;
      return item;
    })
    .sortBy([orderField])
    .map(item => {
      delete item[orderField];
      return item;
    })
    .value();

  const mapping = _.keyBy(sortedList, idField);
  const root = [];

  sortedList.forEach(item => {
    const parentId = item[parentField];
    if (parentId) {
      const parent = mapping[parentId];
      if (parent) {
        parent.children = parent.children || [];
        parent.children.push(item);
      }
    } else {
      root.push(item);
    }
  });

  sortedList.forEach(item => {
    delete item[parentField];
  });
}

```

装订线

```

    return root;
}

```

代码 5.12 将扁平结构转换为树形结构的核心代码

5.4.7 插件界面层与逻辑层通信

界面层与逻辑层采用 Ajax + JSON 方式通信。逻辑层根据需求定义服务端路由，界面层利用 Ajax 方式异步地访问路由从而实现数据交互。JSON^[39]全称为 JavaScript Object Notation，是一种跨平台、跨应用的通用序列化格式，常见于 Web 应用的通信。JSON 无需事先定义数据结构模式（Schema），非常灵活，另外它是文本格式的，易于人阅读理解和调试，但相应地也有数据不紧密、序列化和反序列化时间较长、内存占用较大、无法流式处理等缺陷。对于插件界面层与逻辑层来说，由于其仅在用户需要改变参数时才会发生通信，频率极低、内容较少，因此使用 JSON 通信既能拥有其优势又不会被其劣势影响到。

5.4.8 插件选择实现

用户在使用软件过程中，可以随时改用其他输入插件、算法插件和输出插件，这是通过启用和禁用插件实现的。对于输入插件和算法插件，只能有一个插件处于启用状态，否则输入信号会互相冲突；对于输出插件，可以有多个插件处于启用状态，允许同时向多个目标输出体感信号。

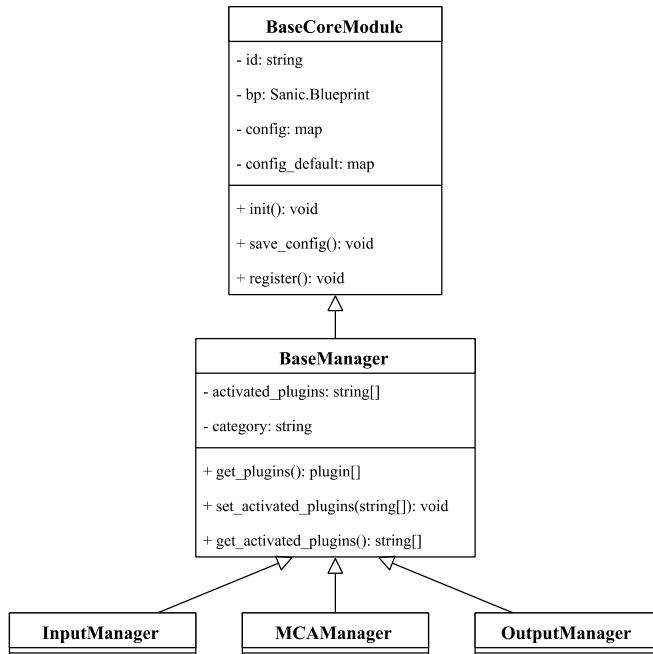


图 5.7 适配模块类图

在逻辑层，插件的启用和禁用是由适配模块调用插件成员函数 activate() 和 deactivate() 实现

的；界面层上，则需要适配模块提供插件信息列表，并响应用户单选或多选的操作。本课题对于单选或多选插件采用相同实现，因此设计了三个适配模块的基类 BaseManager 提供插件选择的功能，类图如图 5.7 所示。值得注意的是，BaseManager 继承自 BaseCoreModule。BaseCoreModule 职责与 BasePlugin 类似，是一个实现了统一配置管理的基类，只是 BasePlugin 面向外部插件，而 BaseCoreModule 面向内部模块。

装
订
线

6 插件模块详细实现

6.1 FSX 模块

Microsoft Flight Simulator 简称 MSFS，是一个面向家庭娱乐的飞行模拟器软件系列，在 Windows 操作系统上运行，最早于 1980 年推出。MSFS 是当前历史最悠久、最有名、功能最全面的家庭级飞行模拟器^[38]。Microsoft Flight Simulator X 简称 FSX，是该软件的最新版本，于 2006 年推出。MSFS 模拟真实世界中飞行所遇到的各种元素，如空气动力、气象、地理环境、飞行操控系统等，为用户呈现视觉和听觉。MSFS 可通过键盘鼠标或手柄控制。

本课题实现与 FSX 的集成，用户安装相应插件并完成配置后，即可实现对 FSX 的实时体感模拟，其中 FSX 负责进行视觉和听觉模拟。

由于 FSX 只能运行在 Windows 操作系统中，因此为了保持本课题软件的跨平台性，FSX 模块分为两部分，一部分与 FSX 运行在同一个操作系统中，从 FSX 实时获取数据，即“FSX 信号产生插件”；另一部分与本课题软件运行在一起，实时接收“FSX 信号产生插件”获取到的数据并进行体感模拟，即“FSX 信号接收模块”。

6.1.1 SimConnect 和 FSUIPC 机制

“FSX 信号产生插件”需要以一定周期从 FSX 软件获取实时飞机姿态数据。FSX 提供了 SimConnect 插件机制，允许以 API 方式访问 FSX 数据；除此以外 FSX 的第三方扩展层 FSUIPC 也提供了类似的插件机制。由于 SimConnect 出现时间较晚，大多数 MSFS 及 FSX 插件都是基于 FSUIPC 实现的。早起版本的 MSFS 不支持插件和编程接口，因此 FSUIPC 利用跨进程内存读写和函数钩子等底层技术实现对 MSFS 的控制；最新的 FSX 上 FSUIPC 利用 SimConnect 上层接口实现对 MSFS 的控制。

6.1.2 飞行姿态实时采样实现

由于 FSUIPC 资料比较多，发展成熟，因此本课题基于 FSUIPC 机制从 FSX 软件实时获取数据。当 FSX 启动时，FSUIPC 会跟着 FSX 一起加载，插件则需要通过共享内存方式访问 FSUIPC 存放在相应内存地址处的数据。

本课题所需的数据和其地址偏移、数据宽度如表 6.1 所示。体感模拟只需用到其中的线加速度和角速度共 6 个数据，其他数据仅在界面上显示，不参与计算，供用户查看状态。

需要特别注意的是，FSX 坐标系与本文规定的坐标系是不一致的。在 FSX 坐标系中，X 是左右、Y 是垂直、Z 是前后，但本文理论部分坐标系则为 X 是前后、Y 是左右、Z 是垂直。本文约定，在 FSX 数据处理成为“输入信号”事件前，其 X、Y、Z 代表 FSX 坐标系；在处理成为“输入信号”后，其 X、Y、Z 代表本文约定的坐标系。

表 6.1 本课题飞行数据采样源

偏移地址	数据宽度	数据类型	描述
0x0264	2	Int16	是否暂停状态, 1=暂停, 0=运行中
0x02BC	4	Int32	指示空速 (knots * 128)
0x6010	8	Double	飞机纬度 (deg), >0=北, <0=南
0x6018	8	Double	飞机经度 (deg), >0=东, <0=西
0x3098	8	Double	体坐标系下飞左右 (X) 方向速度 (ft/sec)
0x30A0	8	Double	体坐标系下飞机垂直 (Y) 方向速度 (ft/sec)
0x3090	8	Double	体坐标系下飞机前后 (Z) 方向速度 (ft/sec)
0x3060	8	Double	体坐标系下飞机左右 (X) 方向加速度 (ft/sec^2)
0x3068	8	Double	体坐标系下飞机垂直 (Y) 方向加速度 (ft/sec^2)
0x3070	8	Double	体坐标系下飞机前后 (Z) 方向加速度 (ft/sec^2)
0x30A8	8	Double	体坐标系下倾斜角速度 (rads/sec)
0x30B0	8	Double	体坐标系下横滚角速度 (rads/sec)
0x30B8	8	Double	体坐标系下偏航角速度 (rads/sec)

装
订
线

FSUIPC 提供了 SDK 可以简化与 FSUIPC 的通信, 本课题“FSX 信号产生插件”基于 FSUIPC C# SDK 开发, 采用轮询方式不断尝试连接到 FSUIPC, 如代码 6.1 所示。一旦连接成功则启动新线程, 以 20Hz 频率读取飞行数据并以事件形式广播给其他模块处理, 如代码 6.3、代码 6.4 所示。其中, 根据 SDK 要求, 读取飞行数据前首先需要声明偏移地址, 由代码 6.2 实现。

```

try
{
    FSUIPCConnection.Open();
    FsxiConnected?.Invoke(this, EventArgs.Empty);
    FsxiValueBagUpdated?.Invoke(this, EventArgs.Empty);
    updateThread = new Thread(new ParameterizedThreadStart(ThreadFunc));
    updateThread.Start(this);
    return true;
}
catch (FSUIPCException)
{
    return false;
}
catch (Exception e)
{
    log.Error(e);
    return false;
}

```

代码 6.1 轮询连接 FSUIPC

```

private Offset<Int16> ipcPaused = new Offset<Int16>(0x0264);
private Offset<Int32> ipcTrueAirSpeed = new Offset<Int32>(0x02BC);
private Offset<Double> ipcLat = new Offset<Double>(0x6010);

```

```

private Offset<Double> ipcLng = new Offset<Double>(0x6018);
private Offset<Double> ipcXVelocity = new Offset<Double>(0x3098);
private Offset<Double> ipcYVelocity = new Offset<Double>(0x30A0);
private Offset<Double> ipcZVelocity = new Offset<Double>(0x3090);
private Offset<Double> ipcXAcceleration = new Offset<Double>(0x3060);
private Offset<Double> ipcYAcceleration = new Offset<Double>(0x3068);
private Offset<Double> ipcZAcceleration = new Offset<Double>(0x3070);
private Offset<Double> ipcPitchVelocity = new Offset<Double>(0x30A8);
private Offset<Double> ipcRollVelocity = new Offset<Double>(0x30B0);
private Offset<Double> ipcYawVelocity = new Offset<Double>(0x30B8);

```

代码 6.2 声明偏移地址

```

private static void ThreadFunc(Object obj)
{
    FsxCController controller = (FsxCController)obj;
    DateTime beginTime, endTime;
    double elapsedMs;
    while (true)
    {
        beginTime = DateTime.Now;
        controller.threadTick();
        endTime = DateTime.Now;
        elapsedMs = (endTime - beginTime).TotalMilliseconds;
        if (elapsedMs < UpdateInterval && elapsedMs >= 0)
        {
            Thread.Sleep(UpdateInterval - (int)elapsedMs);
        }
    }
}

```

代码 6.3 固定频率算法

```

private void threadTick()
{
    if (!ValueBag.Connected)
    {
        return;
    }
    try
    {
        FSUIPCConnection.Process();
        ValueBag.Paused = ipcPaused.Value == 1;
        ValueBag.TrueAirSpeed = (double)ipcTrueAirSpeed.Value / 128d;
        ValueBag.Lat = ipcLat.Value;
        ValueBag.Lng = ipcLng.Value;
        ValueBag.XVelocity = ipcXVelocity.Value;
        ValueBag.YVelocity = ipcYVelocity.Value;
        ValueBag.ZVelocity = ipcZVelocity.Value;
        ValueBag.XAcceleration = ipcXAcceleration.Value;
        ValueBag.YAcceleration = ipcYAcceleration.Value;
        ValueBag.ZAcceleration = ipcZAcceleration.Value;
        ValueBag.PitchVelocity = ipcPitchVelocity.Value * 180d / Math.PI;
        ValueBag.RollVelocity = ipcRollVelocity.Value * 180d / Math.PI;
        ValueBag.YawVelocity = ipcYawVelocity.Value * 180d / Math.PI;
        FsxiValueBagUpdated?.Invoke(this, EventArgs.Empty);
    }
}

```

```
        catch (FSUIPCEException ex) when (ex.FSUIPCErrorCode ==  
FSUIPCError.FSUIPC_ERR_SENDMSG)  
    {  
        Disconnect();  
    }  
    catch (Exception)  
    {  
        // In case of bad data conversion  
    }  
}
```

代码 6.4 读取飞行数据

6.1.3 FSX 信号插件之间通讯实现

FSX 体感模拟具有以下特性或要求：

- (1) 载具状态数据不具有前后依赖性，因此载具姿态的新数据可覆盖旧数据；
- (2) 用户不会时刻运行 FSX，因此“信号接收插件”需要支持“信号产生插件”随时连接或断开；
- (3) 用户希望在一处控制所有配置，因此“信号产生插件”需要不经过配置就能运行；
- (4) 出于逼真度考虑，要求 IO 通信延迟尽可能低；
- (5) 出于高采样频率和延迟考虑，要求通信中花在序列化上的时间尽可能少。

基于此，本课题设计了基于 TCP 控制、UDP 传输数据、异步高性能的通讯模式：

- (1) “信号产生插件”使用 UDP 将姿态数据传递给“信号接收插件”，从而降低延迟。其中，“信号接收插件”是 UDP 服务端，“信号产生插件”是 UDP 客户端；
- (2) “信号接收插件”使用 TCP 控制“信号产生插件”的 UDP 数据流，包括告知 UDP 通信地址和端口等。其中“信号接收插件”是 TCP 服务端，“信号产生插件”是 TCP 客户端；
- (3) “信号产生插件”和“信号接收插件”都异步的发送和接收数据，从而能在 IO 等待时处理其他任务；
- (4) “信号产生插件”和“信号接收插件”都进行高效率的二进制序列化和反序列化；
- (5) “信号产生插件”和“信号接收插件”互相支持任意时刻断线重连，并能处理对方无法连接的情况。

A. 流程

- (1) “信号接收插件”根据用户配置的 FSX 地址，尝试基于 TCP 连接与 FSX 一同运行的“信号产生插件”。由于“信号接收插件”是一个服务端模块，因此用户能统一在服务端上对其进行配置，是满足要求的。
- (2) “信号产生插件”服务端收到连接后，记下并保持该连接的源地址，即“信号接收插件”的 IP 地址。
- (3) “信号接收插件”将用户的其他配置、“信号接收插件”接收数据的 UDP 端口号、UDP 验证码通过 TCP 发送给“信号产生插件”。UDP 验证码是一个 4 字节随机数，用于进行简单的数据时效性验证。
- (4) “信号产生插件”将配置、UDP 端口号、验证码等信息记录下来。

(5) 当 FSX 产生数据时，“信号产生插件”根据当前所有记下了的连接进行广播，即将数据带着 UDP 验证码发送到记录下的 UDP 端口。

(6) 当“信号产生插件”检测到“信号接收插件”的 TCP 连接断开时，将记录下的 UDP 端口号等信息清除。后续 FSX 数据不再发送给该客户端。

(7) 当“信号接收插件”检测到“信号产生插件”的 TCP 连接断开时，不断尝试重新进行连接。

上述流程中关键部分的时序图如图 6.1 所示。

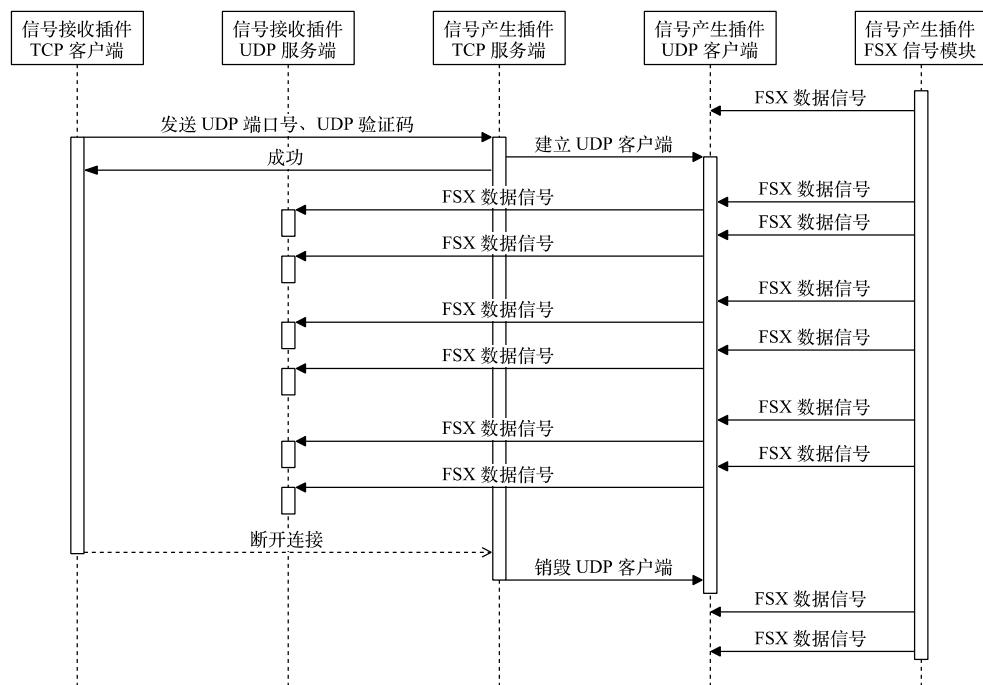


图 6.1 插件通信关键模块时序图

B. TCP 保持活动连接

由流程可见，TCP 连接用于传递发送 UDP 数据包所需的配置信息，并指示了 TCP 发送端（即 UDP 接收端）是否还在运行，对于不在运行的情况不会发送 UDP 数据以节约带宽资源。换言之，TCP 连接仅在初始状态下有数据，建立连接后，只要用户没有修改配置，TCP 连接就没有数据通信了。

由于每个 TCP 连接都要维护一个状态，因此大部分网关或防火墙会丢弃长时间（如 30 分钟）没有数据通信的 TCP 连接（称为不活动连接）。对于被丢弃的 TCP 连接，其实际状态是已经断开，但双方均不知道连接已断开，只有在尝试发送数据后才能知道已断开。但对于“信号产生插件”来说，TCP 连接后续都没有通讯了，因此实际上完全无法知道连接已被断开，此时连接的一段真正掉线后，另一端是无法感知到的。

为了解决这个问题，引入心跳（Heartbeat）机制。心跳机制是指连接通路双方定期向对方（如 1 分钟一次）发送不需要包含有意义内容的数据包，这些数据包可以保持连接活动，并且在连接被丢弃时能及时检测出来。

心跳机制时序图如图 6.2 所示。

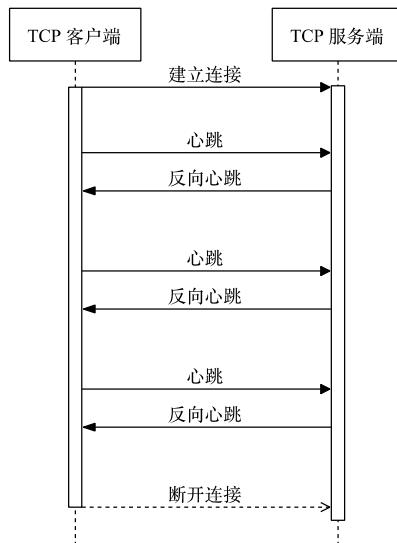


图 6.2 心跳机制时序图

C. TCP 粘包处理

TCP 是一个连接的基于字节流的传输协议。流的含义是没有数据分块。对于一端来说，若其发送了一次 8 字节数据，那么另一端可能一次收到了 8 字节数据，也可能一次收到了 3 字节，剩余 5 字节在第二次收到，还有可能会和下一次发送的数据合在一起被收到。总之数据接收端无法确切地知道发送端每次发送了多少长度的数据。

为了正确解析每一条指令，需要给指令插入长度前缀来进行分割，如图 6.3 所示。长度前缀是一个长度固定为 4 字节、字节顺序是小端序的整数，代表单条指令有多长。接下来相应长度的数据就是指令内容。对于接收端来说，可先收满 4 字节，解析出长度，再继续收满相应字节，从而构成一条完整指令。

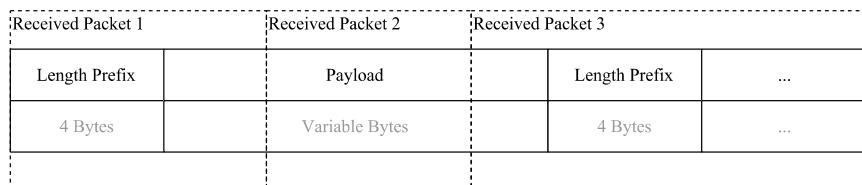


图 6.3 长度前缀示意图

D. UDP 数据包丢包和乱序处理

UDP 是一个无状态、不可靠的协议，“信号接收插件”接收到的数据包可能是乱序的，也可能会有丢包情况。但由于 FSX 姿态数据不具有顺序上的前后依赖性，当收到最新姿态数据后，旧姿态数据已没有意义，因此无需处理丢包的情况，且乱序数据包可以被丢弃。

这里使用递增的计数器识别乱序情况并丢弃。在发送端“信号产生插件”发送数据包时，

附带一个递增的序列号，代表包的发送顺序，则接收端“信号接收插件”只需记录当前已收到的最大序列号，若收到不超过当前最大序列号的数据包，代表是一个过期数据包，新数据包已被接收过，因此该数据包直接被丢弃；若收到超过当前最大序列号的数据包，则代表是一个新数据包，可以被使用。

如图 6.4 所示情况，接收端依次收到了数据包 0、3、2、4，且数据包 1 丢失，则这个过程中实际采用的是数据包 0、3、4 作为输入信号。

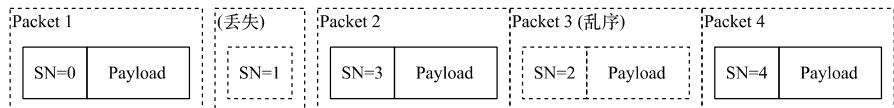


图 6.4 UDP 乱序和掉包示意图

E. 序列化协议

通讯中采用 Protocol Buffer^[39]实现二进制序列化。Protocol Buffer 是 Google 公司设计的一种跨平台、跨应用的通用序列化格式，常见于对性能要求较高的应用，如游戏客户端与游戏服务器之间^[40]。Protocol Buffer 序列化后数据排列紧密，序列化和反序列化时间较短，但其要求序列化反序列化双方事先已协商出一份数据结构。

代码 6.5 展示了本课题中“信号接收插件”对“信号产生插件”控制请求的数据结构，代码 6.6 展示了响应请求的数据结构，代码 6.7 展示了“信号产生插件”向“信号接收插件”传递的 UDP 数据结构。

```

message TcpRequestMessage {
    enum MsgType {
        MSG_TYPE_SET_CONFIG = 0;
        MSG_TYPE_PING = 1;
        MSG_TYPE_TEST_CONNECTION = 2;
    }

    message SetConfigBody {
        int32 udpPort = 1;
        int32 udpToken = 2;
    }

    message PingBody {
        int32 timeStamp = 1;
    }

    message TestConnBody {
        int32 magicToken = 1;
    }

    MsgType msgType = 1;
    oneof msgBody {
        SetConfigBody setConfigBody = 2;
        PingBody pingBody = 3;
        TestConnBody testConnBody = 4;
    }
}

```

装
订
线

代码 6.5 控制请求数据结构的代码定义

```
message TcpResponseMessage {
    bool success = 1;
    int32 timeStamp = 2;
}
```

代码 6.6 控制响应数据结构的代码定义

```
message UdpResponseMessage {
    enum MsgType {
        MSG_TYPE_TEST_CONNECTION_CALLBACK = 0;
        MSG_TYPE_TRANSMISSION_DATA = 1;
    }

    message TestConnCallbackBody {
        int32 magicToken = 1;
    }

    message TransmissionDataBody {
        double xAcceleration = 1;
        double yAcceleration = 2;
        double zAcceleration = 3;
        double pitchVelocity = 4;
        double rollVelocity = 5;
        double yawVelocity = 6;
    }

    MsgType msgType = 1;
    int32 serialNumber = 2;
    int32 token = 3;
    oneof msgBody {
        TestConnCallbackBody testConnCallbackBody = 4;
        TransmissionDataBody transmissionDataBody = 5;
    }
}
```

代码 6.7 飞行姿态数据结构的代码定义

F. 信号产生插件异步 TCP 服务端实现

异步 TCP 服务端的主要任务是接收请求、处理并反馈处理成功的响应。其中对于每个已连接的客户端，应当维护一份配置列表。根据代码 6.5 规定的结构，请求有以下三种类型：

- (1) 更新关于“信号接收插件”的配置，包括端口号等，对应于数据结构中 msgType 字段为 MSG_TYPE_SET_CONFIG 的情况；
- (2) 维持心跳，对应于数据结构中 msgType 字段为 MSG_TYPE_PING 的情况；
- (3) 测试连接通路，对应于数据结构中 msgType 字段为 MSG_TYPE_TEST_CONNECTION 的情况。该功能在本文中未实现。

代码 6.8 展示了具体实现的核心逻辑。由于运用了 `async / await` 特性，能以同步的风格编写了异步的代码，既保持了代码的可读性又具有较高的性能。

装订线

```
while (true)
{
    // Read size
    await networkStream.ReadAsync(bufferReqSize, 0, 4, token);
    Int32 reqSize = BitConverter.ToInt32(bufferReqSize, 0);

    // Read body
    bufferReqBody = new byte[reqSize];
    await networkStream.ReadAsync(bufferReqBody, 0, reqSize);
    TcpRequestMessage request = TcpRequestMessage
        .Parser
        .ParseFrom(bufferReqBody);

    bool responseSuccess = false;
    switch (request.MsgType)
    {
        case TcpRequestMessage.Types.MsgType.Ping:
            responseSuccess = true;
            break;
        case TcpRequestMessage.Types.MsgType.SetConfig:
            client.SetTransmissionTarget(
                request.SetConfigBody.UdpPort,
                request.SetConfigBody.UdpToken);
            responseSuccess = true;
            break;
        case TcpRequestMessage.Types.MsgType.TestConnection:
            responseSuccess = true;
            break;
    }

    TcpResponseMessage response = new TcpResponseMessage
    {
        Success = responseSuccess,
        TimeStamp = Utils.GetTimeStamp(DateTime.UtcNow),
    };

    // Write size and body
    bufferResBody = response.ToArray();
    bufferResSize = BitConverter.GetBytes(bufferResBody.Length);
    await networkStream.WriteAsync(bufferResSize, 0, 4, token);
    await networkStream.WriteAsync(bufferResBody, 0, bufferResBody.Length, token);
}
```

代码 6.8 异步 TCP 服务端关键实现

由于以多线程异步运行，因此共享的数据结构（如已连接的客户端列表）需要上锁，如代码 6.9 所示。此处用的是异步读写锁，可以异步地等待。

```
private async void HandleTcpClientConnectionAsync(TcpClient tcpClient,
    CancellationToken token)
{
    HexiUdpClient client = new
    HexiUdpClient(((IPEndPoint)tcpClient.Client.RemoteEndPoint).Address.MapToIPv4().ToString());
    using (await ValueBag.HexiClientsLock.WriterLockAsync())
    {
```

```

        ValueBag.HexiClients.Add(client);
    }
    try
    {
        while (true)
        {
            // .....
        }
    }
    catch (OperationCanceledException)
    {
        throw;
    }
    finally
    {
        if (tcpClient.Connected)
        {
            tcpClient.Close();
        }
        client.Close();
        using (await ValueBag.HexiClientsLock.WriterLockAsync())
        {
            ValueBag.HexiClients.Remove(client);
        }
    }
}

```

代码 6.9 利用异步读写锁解决多线程竞争条件

装订线

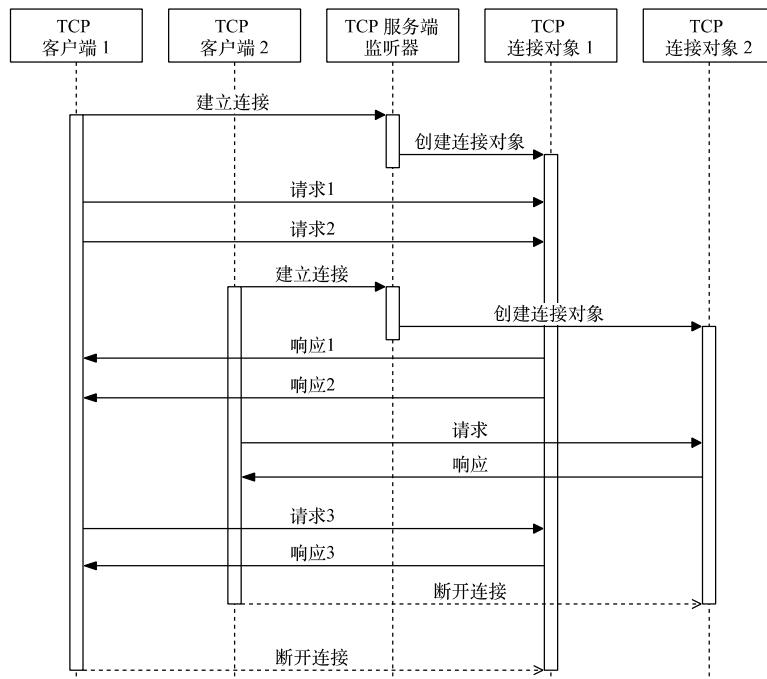


图 6.5 异步 TCP 服务端时序图

时序图 6.5 描述了异步 TCP 服务端的时序关系。时序图 6.5 中“连接对象”即代码 6.8 中的 TcpClient。注意到 TCP 客户端和 TCP 服务端均是异步实现的，因此客户端不会等待收到上一个

请求响应后再发送下一个请求，服务端也不会等响应发送完毕后再处理下一个请求。这减少了IO等待从而提高了效率。

G. 信号产生插件异步 UDP 客户端实现

UDP 客户端根据 TCP 连接确定的配置发送数据并递增计数器，核心逻辑如代码 6.10 所示。

对于 UDP 来说，不存在流的概念，因此无需写入长度前缀。由于飞行姿态实时采样是同步进行的，代码 6.10 还特别实现了用于同步环境的异步函数 SendMessage，其任务会在线程池上异步执行，不阻塞当前线程代码。

```
public Task<int> SendAsync(byte[] bytes, int n)
{
    if (Valid)
    {
        return client.SendAsync(bytes, n);
    }
    return Task.FromResult<int>(0);
}

public async Task SendMessageAsync(UdpResponseMessage message)
{
    // No need to add Length prefix since packet size < MTU
    SerialNumber = SerialNumber + 1;
    message.Token = Token;
    message.SerialNumber = SerialNumber;
    var body = message.ToByteArray();
    await SendAsync(body, body.Length);
}

public void SendMessage(UdpResponseMessage message)
{
    Task.Run(() => SendMessageAsync(message));
}
```

装
订
线

代码 6.10 异步 UDP 客户端关键实现

H. 信号接收插件异步、自动重连 TCP 客户端实现

自动重连包括建立初始连接时的重复尝试和连接断开后的重复尝试。又由于插件可能随时被用户禁用和重新启用，因此 TCP 客户端包括重连机制需要支持取消和重新开始。本课题基于有限状态机实现自动重连，状态转换图如图 6.6 所示。

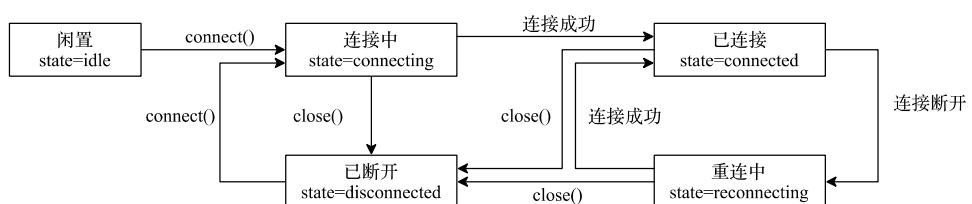


图 6.6 TCP 客户端状态转换图

由状态转换图可见，初始连接和断线重连情况下的“重试并连接”操作可以归一到了同一个处理逻辑下，如代码 6.11 所示。为了能在插件被禁用时停止相关操作，connect、work、heartbeat 等异步或协程函数的执行控制器都被存储了下来，以便在需要的时候能中断其执行，如代码 6.12 所示。

```

async def connect_async(self):
    while True and (self.state in ['connecting', 'reconnecting']):
        try:
            future = asyncio.open_connection(self.host, self.port)
            reader, writer = await asyncio.wait_for(future, timeout=3)
            _logger.info('Telemetry connected')
            self.reader = reader
            self.writer = writer
            self.state = 'connected'
            self.work_future = asyncio.ensure_future(self.work_async())
            self.work_future.add_done_callback(self.on_work_done)
            self.heartbeat_future = asyncio.ensure_future(self.heartbeat_async())
            self.heartbeat_future.add_done_callback(self.on_heartbeat_done)
            self.ee.emit('tcp_connected')
        except (OSError, asyncio.TimeoutError):
            # server not connected, retry
            await asyncio.sleep(self.retry_sec)

def connect(self):
    assert(self.state in ['idle', 'disconnected'])
    assert(self.connect_future == None)
    self.state = 'connecting'
    self.connect_future = asyncio.ensure_future(self.connect_async())
    self.connect_future.add_done_callback(self.on_connect_done)
    return self.connect_future

def on_connect_done(self, future):
    self.connect_future = None

```

代码 6.11 建立连接的实现

```

def disconnect(self):
    assert(self.state in ['connecting', 'connected', 'reconnecting'])
    self.state = 'disconnected'
    if self.connect_future != None:
        self.connect_future.cancel()
    if self.reconnect_future != None:
        self.reconnect_future.cancel()
    if self.work_future != None:
        self.work_future.cancel()
    if self.heartbeat_future != None:
        self.heartbeat_future.cancel()
    if self.writer != None:
        self.writer.close()

```

代码 6.12 取消所有连接和重试的实现

成功建立连接后，work 和 heartbeat 协程同时异步地运行。本课题中，由于支持重连的异步 TCP 客户端逻辑比较复杂，因而是作为一个独立的模块，它不关心什么时候要发送什么数据，只向

外提供发数据的接口，由调用者决定发送时机，所以 work 协程中不会发送内容，而仅仅是异步地尝试读取响应并产生相应事件，如代码 6.13 所示。其中，由于需要处理粘包情况，因此是以流形式读取，且根据长度前缀决定后续需要读取的数据长度。对应地，发送消息的接口实现如代码 6.14 所示。

```
async def work_async(self):
    try:
        while True:
            size_buffer = await self.reader.readexactly(4)
            size = int.from_bytes(size_buffer, byteorder='little')
            body_buffer = await self.reader.readexactly(size)
            msg = fsx_pb2.TcpResponseMessage()
            msg.ParseFromString(body_buffer)
            self.ee.emit('tcp_received_message', msg)
    except (asyncio.IncompleteReadError, ConnectionResetError, ConnectionAbortedError):
        pass
```

代码 6.13 接收消息协程实现

装

```
def write_message(self, msg):
    data = msg.SerializeToString()
    data = len(data).to_bytes(4, byteorder = 'little') + data
    self.writer.write(data)
```

代码 6.14 发送消息接口实现

订

线

heartbeat 模块需要主动地定期发送数据包以保持连接，如代码 6.15 所示。

```
async def heartbeat_async(self):
    while True:
        await asyncio.sleep(10)
        msg = fsx_pb2.TcpRequestMessage()
        msg.msgType = fsx_pb2.TcpRequestMessage.MSG_TYPE_PING
        msg.pingBody.timeStamp = int(time.time())
        self.write_message(msg)
```

代码 6.15 心跳实现

最后是断线重连逻辑。当连接中断时，代码 6.13 中的 readexactly 异步函数会产生一个异常，该异常是 asyncio.IncompleteReadError、ConnectionResetError、ConnectionAbortedError 中的某一个，从而打断了无限尝试读取数据流的循环，造成 work_async 函数结束，进而触发 on_work_done 回调函数。需要注意的是，work_async 函数除了可能在断线后结束，还可能在禁用插件后结束，因此 on_work_done 中需要判明是被动断线了还是主动关闭了，在后者的情况下不能重新连接，如代码 6.16 所示；在前者情况下需要切换状态并重新连接，如代码 6.17 所示。

```
def on_work_done(self, future):
    _logger.info('Telemetry connection lost')
```

```

self.work_future = None
if self.heartbeat_future != None:
    self.heartbeat_future.cancel()
self.reader = None
self.writer = None
if self.state != 'disconnected':
    self.reconnect()

```

代码 6.16 work 协程结束的处理逻辑

```

async def reconnect_async(self):
    await self.connect_async()

def reconnect(self):
    assert(self.state == 'connected')
    assert(self.reconnect_future == None)
    _logger.info('Telemetry reconnecting')
    self.state = 'reconnecting'
    self.reconnect_future = asyncio.ensure_future(self.reconnect_async())
    self.reconnect_future.add_done_callback(self.on_reconnect_done)
    return self.reconnect_future

```

代码 6.17 重连的实现

该模块时序图如图 6.7 所示。时序图展示了插件被随时启用、禁用且 FSX 被随时打开关闭情况下的时序情况。

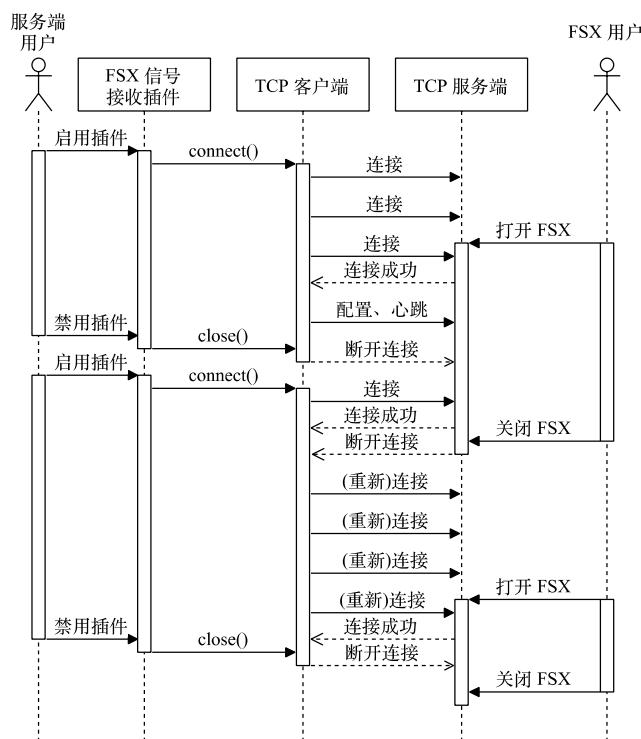
装
订
线

图 6.7 TCP 客户端时序图

I. 信号接收插件异步 UDP 服务端实现

UDP 服务端实现比较简单，只需判明 UDP 验证码（代码中称为 token）是否一致、是否是旧数据包需要丢弃即可，如代码 6.18 所示。代码中，对于每个收到或丢弃的消息都会以不同事件传递给外界，从而可以实现错误消息数量统计等功能。

```
class UDPServer(asyncio.DatagramProtocol):

    def __init__(self, manager, token):
        super().__init__()
        self.manager = manager
        self.token = token
        self.sn = 0

    def datagram_received(self, data, addr):
        try:
            # Note: there are no Length prefix in UDP packets
            msg = fsx_pb2.UdpResponseMessage()
            msg.ParseFromString(data)
            if msg.token != self.token:
                _logger.warn('A message is discarded because of incorrect token')
                print(msg.token, self.token)
                self.manager.ee.emit('udp_discarded_message')
                return
            if msg.serialNumber <= self.sn:
                _logger.warn('A message is discarded because of received newer message')
                self.manager.ee.emit('udp_discarded_message')
                return
            self.sn = msg.serialNumber
            self.manager.ee.emit('udp_received_message', msg)
        except Exception as e:
            _logger.warn(e)
            self.manager.ee.emit('udp_discarded_message')

    def connection_lost(self, exc):
        self.manager.ee.emit('udp_closed')
```

代码 6.18 UDP 服务端核心实现

6.2 标准飞行姿态输入模块

该模块内置了起飞、降落、左偏航、右偏航四种状态下飞机的位置姿态，用户不需要安装 Microsoft Flight Simulator X 即可对这些运动进行体感模拟。另外，该模块还允许用户重定义或创建新的状态。

6.2.1 位置姿态存储

为了便于用户修改或创建，该模块利用 JSON 文件存储不同状态下的姿态变化。姿态变化是一个连续的过程，需要存储一批连续的位置姿态数据，模块将以 20Hz 频率对其进行复现。每个状态存储在一个文件中，其格式由代码 6.19 所示的 JSON Schema 描述。

其中，attitudes 字段每一项是一个长度为 6 的数组，数组中每一项含义如表 6.2 所示；fromState 字段描述了改状态可以由哪些状态转换而来，若不指定则代表只可以由初始状态转换

而来。代码 6.20 是一个符合格式的状态文件样例。

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "id": { "type": "string" },
    "text": { "type": "string" },
    "order": { "type": "integer" },
    "fromState": {
      "type": "array",
      "items": { "type": "string" }
    },
    "attitudes": {
      "type": "array",
      "items": {
        "type": "array",
        "minItems": 6,
        "maxItems": 6,
        "items": { "type": "number" }
      }
    }
  },
  "required": [ "id", "text", "attitudes" ]
}
```

代码 6.19 飞行姿态输入模块状态文件格式描述

表 6.2 飞行姿态输入模块状态文件 attitudes 字段各项描述

下标	数据类型	描述
0	Double	X 方向加速度 (m/s^2)
1	Double	Y 方向加速度 (m/s^2)
2	Double	Z 方向加速度 (m/s^2)
3	Double	α 方向角速度 (rad/s)
4	Double	β 方向角速度 (rad/s)
5	Double	γ 方向角速度 (rad/s)

```
{
  "id": "take_off",
  "text": "起飞",
  "attitudes": [
    [0, 0, 0, 0, 0, 0],
    [0.5, 0, 0, 0, 0, 0],
    [1, 0, 0, 0, 0, 0],
    [2, 0, 0, 0, 0, 0],
    .....
  ]
}
```

代码 6.20 飞行姿态输入模块状态文件样例

6.2.2 位置姿态信号输入

代码 6.21 以 20Hz 为频率异步地由状态文件内容产生输入信号。

```
SLEEP_INTERVAL = 1 / 20

async def send_signal_async(self, state_id):
    state = self.states[state_id]
    for attitude in state['attitudes']:
        self.emit_input_signal(attitude)
    await asyncio.sleep(SLEEP_INTERVAL)
```

代码 6.21 飞行姿态输入模块产生输入信号的实现

6.3 经典洗出算法模块

6.3.1 流程综述

- (1) 置初始姿态向量为 0;
- (2) 当有输入信号时, 计算得出新姿态向量;
- (3) 基于新姿态向量, 调用接口产生运动平台姿态向量。

6.3.2 滤波器参数配置

为了便于用户修改滤波器参数, 滤波器采用配置文件描述参数。每个滤波器配置具有如表 6.3 所示的配置项。本模块所有滤波器初始参数配置如表 6.4 所示。

表 6.3 滤波器配置项描述

配置项名称	类型	是否可选	描述
order	int	否	滤波器阶数, 取值 1 ~ 3
lp	bool	否	是否是低通滤波器, 取 False 代表是高通滤波器
zeta	float	是	阻尼比, 仅 2、3 阶滤波器需要, 即滤波器公式中参数 ζ
omega	float	否	截止频率, 即滤波器公式中参数 ω_n
omega_1	float	是	额外截止频率, 仅 3 阶滤波器需要, 即参数 ω_b

表 6.4 所有滤波器初始参数配置

配置项路径	描述	order	lp	zeta	omega	omega_1
tilt.x	倾斜协调 X 方向滤波器	2	True	1.0	5.0	/
tilt.y	倾斜协调 Y 方向滤波器	2	True	1.0	8.0	/
movement.x	位移运动 Z 方向滤波器	3	False	1.0	2.5	0.25
movement.y	位移运动 Y 方向滤波器	3	False	1.0	4.0	0.4
movement.z	位移运动 Z 方向滤波器	3	False	1.0	4.0	0.4
rotate.alpha	旋转运动 α 方向滤波器	1	False	/	1.0	/
rotate.beta	旋转运动 β 方向滤波器	1	False	/	1.0	/
rotate.gamma	旋转运动 γ 方向滤波器	2	False	1.0	1.0	/

代码 6.22 能够根据表 6.3 所示的配置项构造代码 4.1 所示的滤波器计算类。

```
def build_filter(*, order:int=1, lp:bool=True,
                 omega:float=0.0, zeta:float=1.0, omega_1:float=0.0, freq=FREQ):
    assert order in [1, 2, 3]
    if order == 1:
        return build_1st_filter(omega, lp, freq)
    elif order == 2:
        return build_2nd_filter(omega, zeta, lp, freq)
    else:
        return build_3rd_filter(omega, zeta, omega_1, lp, freq)

def rebuild_filters(self):
    self.filters = {}
    for kind, dimensions in self.config['filter'].items():
        self.filters[kind] = {}
        for d, filter_config in dimensions.items():
            self.filters[kind][d] = dfilter.build_filter(**filter_config)
```

代码 6.22 根据配置构造滤波器计算类

6.3.3 滤波器计算

构造出代码 4.1 所示的滤波器计算类后，可通过代码 6.23 所示函数分别计算用于位移运动、倾斜调整、旋转运动的滤波器。这些函数接收一组向量并输出结构一致的滤波后的向量。

装
订
线

```
def apply_movement_filter(self, a_i):
    filters = self.filters['movement']
    return numpy.array([
        filters['x'].apply(a_i[0][0]),
        filters['y'].apply(a_i[1][0]),
        filters['z'].apply(a_i[2][0]),
    ]).T

def apply_tilt_filter(self, f_s):
    filters = self.filters['tilt']
    return numpy.array([
        filters['x'].apply(f_s[0][0]),
        filters['y'].apply(f_s[1][0]),
        0,
    ]).T

def apply_rotate_filter(self, omega_s):
    filters = self.filters['rotate']
    return numpy.array([
        filters['alpha'].apply(omega_s[0][0]),
        filters['beta'].apply(omega_s[1][0]),
        filters['gamma'].apply(omega_s[2][0]),
    ]).T
```

代码 6.23 利用滤波器计算类进行计算

6.3.4 算法实现

根据图 2.11 所示流程，代码 6.24 实现了体感模拟核心算法。

```
def handle_input_signal(self, data):
    global ig_disp_1, ig_disp_2, ig_rot_1, ps, po
    g_a = VECTOR_G
    a_a = numpy.array([data[0:3]]).T
    omega_a = numpy.array([data[3:6]]).T

    f_a = a_a - g_a
    f_s = self.apply_movement_scaling(f_a)
    a_s = f_s + VECTOR_G
    a_hp = self.apply_movement_filter(a_s)
    ig_disp_1 = ig_disp_1 + delta_time * a_hp
    ig_disp_2 = ig_disp_2 + delta_time * ig_disp_1
    ps = numpy.copy(ig_disp_2)

    f_lp = self.apply_tilt_filter(f_s)
    theta_lp = numpy.array([
        math.asin(self.apply_tilt_scaling(f_lp[1][0]) / scipy.constants.g),
        -math.asin(self.apply_tilt_scaling(f_lp[0][0]) / scipy.constants.g),
        0
    ]).T
    theta_tc = theta_lp

    omega_s = self.apply_rotate_scaling(omega_a)
    omega_hp = self.apply_rotate_filter(omega_s)
    ig_rot_1 = ig_rot_1 + delta_time * omega_hp

    po = ig_rot_1 + theta_tc

    self.emit_mca_signal(data, [
        ps[0][0],
        ps[1][0],
        ps[2][0],
        po[0][0],
        po[1][0],
        po[2][0],
    ])
```

装
订
线

代码 6.24 经典洗出算法的实现

6.4 Stewart 运动平台可视化仿真输出模块

本课题利用 WebGL 技术实现了 Stewart 运动平台的三维可视化仿真。基于该三维仿真，用户可以在不需要硬件的情况下查看运动平台的实时姿态，调整、验证各项参数并及时看到效果，显著降低输入插件和体感模拟插件的开发难度和调试难度。

6.4.1 WebGL 技术和 Three.js 框架

WebGL 是一项利用 JavaScript API 渲染 3D 或 2D 图形的技术。其中 3D 图像基于 OpenGL ES 2.0 接口，使用 GLSL 作为着色器。WebGL 在 HTML 5 标准中被引入，被主流浏览器原生支持，无需插件即可渲染。

WebGL 提供的是底层渲染接口，一般开发者还需借助 3D 引擎或框架才能真正实现三维渲染。本课题使用的是 Three.js 框架。Three.js 是一个流行的跨浏览器 JavaScript 3D 框架，对灯光、

场景、特效、材质、动画等各种三维渲染框架常见功能有良好的支持，能基于 WebGL 实现 GPU 加速的高效三维渲染。

6.4.2 几何形状构建

在 Three.js 中渲染物体有以下步骤：

- (1) 建立物体几何形状 (Geometry)，包括各个顶点坐标和各个表面信息；
- (2) 设定表面材质 (Material)；
- (3) 由两者共同建立抽象物体 (Object)，将物体加入场景；
- (4) 调用函数渲染场景。

由于材质决定的是物体颜色，对三维仿真而言不重要，因此只阐述几何形状的构建过程。

A. 平台

代码 6.25 根据式 (2.3) 在给定 r, α 参数的情况下构造以 $(0, 0)$ 为中心的平台二维平面图形路径。其中，路径最后一个点和第一个点重合，形成封闭图形。由于上下平台是镜像关系，因此函数还接收 reverse 选项，允许构建旋转 180 度后的图形。

```
getPlatePoints(radius, alpha) {
  const shapePoints = [];
  const baseDeg = this.options.reverse ? Math.PI : 0;
  for (let i = 0; i < 3; ++i) {
    const orient = 2 / 3 * Math.PI * i + baseDeg;
    shapePoints.push(new THREE.Vector2(
      radius * Math.cos(orient - alpha),
      radius * Math.sin(orient - alpha)
    ));
    shapePoints.push(new THREE.Vector2(
      radius * Math.cos(orient + alpha),
      radius * Math.sin(orient + alpha)
    ));
  }
  shapePoints.push(shapePoints[0]);
  return shapePoints;
}
```

代码 6.25 根据参数计算平台几何图形

利用 Three.js 中的 ExtrudeGeometry，可将该二维封闭图形扩展到三维空间几何图形并自动建立相应的表面，如代码 6.26 所示。由于 Three.js 是右手坐标系，而 ExtrudeGeometry 不改变原路径最低两维，扩展的是前后方向的 Z 轴，因此建立三维几何图形后需要对其进行旋转，以朝向正上方。另外，Stewart 平台中的定平台一般是镂空的，本课题也对此进行仿真，因此还需改用较小的半径作为内部镂空路径。

```
const shape = new THREE.Shape(this.getPlatePoints(
  this.options.radius,
  this.options.alpha
));
if (this.options.innerRadius > 0) {
```

```

const inner = new THREE.Path(this.getPlatePoints(
  this.options.innerRadius,
  this.options.alpha
));
shape.holes.push(inner);
}
const geometry = new THREE.ExtrudeGeometry(shape, {
  amount: this.options.thickness,
  bevelEnabled: false,
});
geometry.lookAt(new THREE.Vector3(0, 1, 0));

```

代码 6.26 利用 getPlatePoints 和 ExtrudeGeometry 建立平台三维几何图形

支撑杆一般会和平台内的连接点相连接，而非与平台六个角直接连接，因此还需额外定义六个支撑杆连接处的坐标。代码 6.27 利用定义的连接点所在圆半径“jointRadius”确定连接点坐标。

```

this.jointPoints = this
  .getPlatePoints(this.options.jointRadius, this.options.alpha)
  .slice(0, 6)
  .map(v2 => new THREE.Vector3(v2.x, this.options.thickness / 2, v2.y));

```

代码 6.27 建立支撑杆连接点坐标

装
订
线

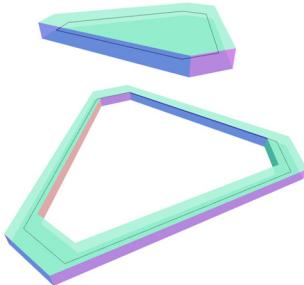


图 6.8 平台渲染效果

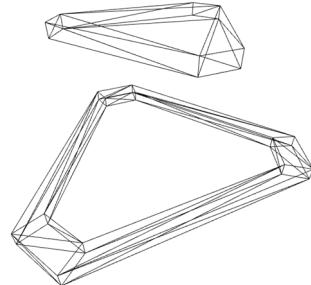


图 6.9 平台线框图

图 6.8、图 6.9 展示了某个角度下渲染出的上下平台效果。六个连接点坐标所连图形在图 6.8 中以黑色多边形呈现。

B. 支撑杆

本课题实现的三维仿真中，使用的是连接定平台、动平台的棱柱体代替了常用的圆柱体支撑杆，从而在不影响真实性的条件下提高渲染速度。在给出了棱柱体两端三维坐标的情况下，棱柱体可直接使用 Three.js 自带的 TubeGeometry 构建，如代码 6.28 所示。TubeGeometry 以三维空间路径为基础（不限于直线），构建指定棱数、指定半径的三维空间图形。最后，以定平台六个连接点空间坐标和动平台六个连接点空间坐标为支撑杆两端坐标，即可构建出六个支撑杆，如代码 6.29 所示。

```
this.geometry = new THREE.TubeGeometry(
  new THREE.LineCurve3(this.options.p1, this.options.p2),
  1, this.options.radius, 4
);
```

代码 6.28 构建棱柱体几何图形

```
this.legs = [];
for (let i = 0; i < 6; ++i) {
  const leg = new Leg({
    p1: jointsBase[i],
    p2: jointsTop[(i + 3) % 6],
    radius: 5,
  });
  leg.addTo(this.group);
  this.legs.push(leg);
}
```

代码 6.29 基于连接点坐标和棱柱体几何图形对支撑杆进行仿真

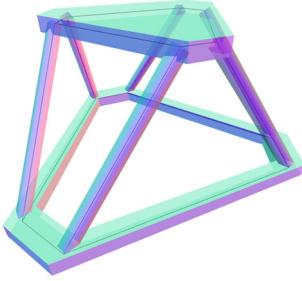
装
订
线

图 6.10 平台和支撑杆半透明渲染效果

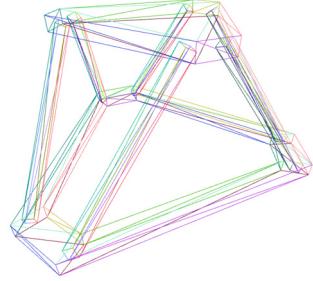


图 6.11 平台和支撑杆线框图

图 6.10、图 6.11 展示了平台和支撑杆共同渲染出的整体效果。为了便于观察连接点位置，图 6.10 中表面是半透明的，并且将连接点和支撑杆路径用黑色路径标出。

6.4.3 位置姿态计算

输出模块接受的信号是运动平台的位置和姿态，即式 (2.4)、(2.7)，它们决定了连接点坐标，可通过式 (2.15) 计算得出。由于 Three.js 内置了强大的数学计算功能，因此连接点坐标可直接根据给出的位置偏移和姿态方向计算得出，无需再自行实现旋转矩阵的计算等步骤，如代码 6.30 所示。

```
// 设定平台位置和姿态
this.object.rotation.x = this.attitude.roll;
this.object.rotation.y = this.attitude.yaw;
this.object.rotation.z = this.attitude.pitch;
this.object.position.x = this.attitude.surge;
this.object.position.y = this.attitude.heave + this.options.yOffset || 0;
this.object.position.z = this.attitude.sway;
this.object.updateMatrix();
```

```
// 根据平台位置姿态更新对应的连接点坐标
const matrix = this.object.matrix;
return this.jointPoints.map(v3 => v3.clone().applyMatrix4(matrix));
```

代码 6.30 位置姿态计算

6.4.4 模块通讯机制

三维仿真需要实时地根据当前体感模拟信号做出姿态仿真，理论上可基于异步 UDP 二进制数据流构建出最优方案，但由于 Web 技术限制，尚不支持建立 UDP 数据信道，因此采用了目前技术条件下最优方案 WebSocket + JSON 来实现。

WebSocket 是浏览器端支持的一种可靠的长连接数据协议，基于 TCP 实现，可从 HTTP 协议升级协商。与 HTTP 协议不同的是，WebSocket 可以建立起浏览器与服务端双向长连接；与 TCP 面向流不同的是，WebSocket 数据以帧为单位，开发较为简单。

利用 Sanic 的 WebSocket 功能，可在模块服务端定义 WebSocket 通信接口，如代码 6.31 所示。其中，用户可能打开多个浏览器标签页访问仿真平台，平台姿态信号到仿真平台是一种广播消息，因此模块需要维护当前已连接的 WebSocket 客户端列表，并在收到平台姿态信号时向所有已连接的客户端广播消息，如代码 6.32 所示。

```
@self_bp.websocket('/api/signal')
async def signal_feed(request, ws):
    try:
        self.connected_clients.add(ws)
        while True:
            await ws.recv()
    finally:
        self.connected_clients.remove(ws)
```

代码 6.31 服务端 WebSocket 路由的实现

```
def handle_motion_signal(self, input_signal, motion_signal):
    data_to_send = json.dumps(motion_signal)
    for client in self.connected_clients:
        asyncio.ensure_future(client.send(data_to_send))
```

代码 6.32 服务端 WebSocket 广播信号的实现

界面部分，需要建立起与服务端的 WebSocket 连接，并在收到平台姿态信号的时候修改并渲染运动平台外观，如代码 6.33 所示。

```
const ws = new
WebSocket(`ws://${location.host}/plugins/output_stewart_visualize/api/signal`);
ws.addEventListener('message', ev => {
    try {
        const data = JSON.parse(ev.data);
        this.controlOptions.attitude.surge = data[0] * 50;
        this.controlOptions.attitude.sway = data[1] * 50;
```

```
this.controlOptions.attitude.heave = data[2] * 50;
this.controlOptions.attitude.roll = THREE.Math.radToDeg(data[3]);
this.controlOptions.attitude.pitch = THREE.Math.radToDeg(data[4]);
this.controlOptions.attitude.sway = THREE.Math.radToDeg(data[5]);
this._updateAttitude();
} catch (e) {
}
});
```

代码 6.33 浏览器端接收 WebSocket 实时信号的实现

6.4.5 效果展示

图 6.12 展示了初始姿态仿真平台整体界面，用户可以通过鼠标拖拽进行视图旋转。图 6.13 展示了不同视角和不同姿态下的效果。

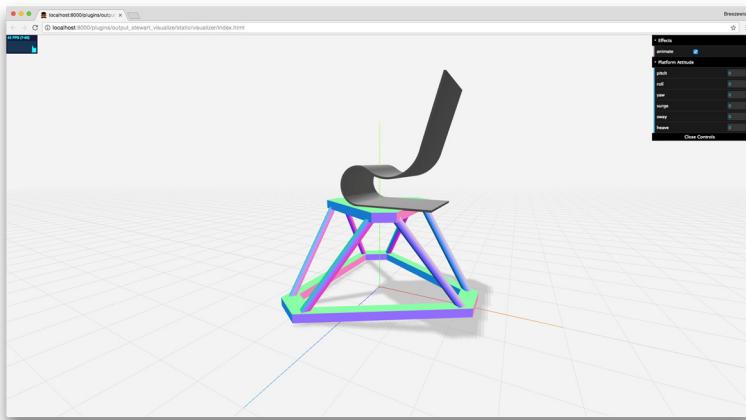


图 6.12 初始状态下可视化仿真效果

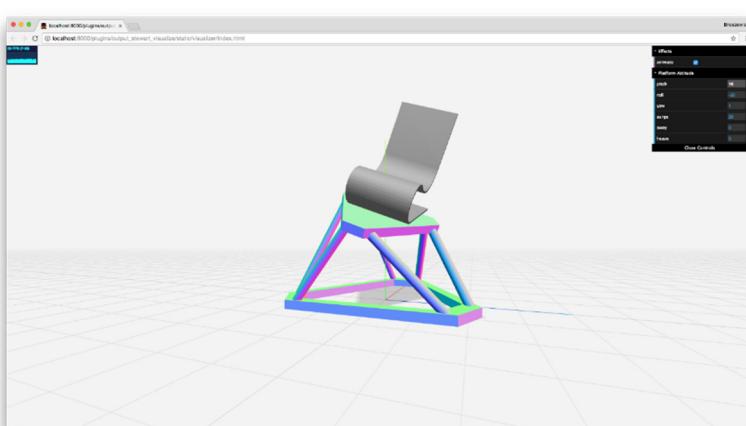


图 6.13 改变观察角度和平台位置姿态后可视化仿真效果

装
订
线

7 测试及优化

7.1 FSX 模块

7.1.1 采样结果

在 FSX 内以 AirCreation Trike Ultralight 为机型，在 KEDW 机场分别进行跑道起飞、左偏航、右偏航、降落到土地的测试，采样得出加速度和角速度变化如图 7.1 到图 7.4 所示。

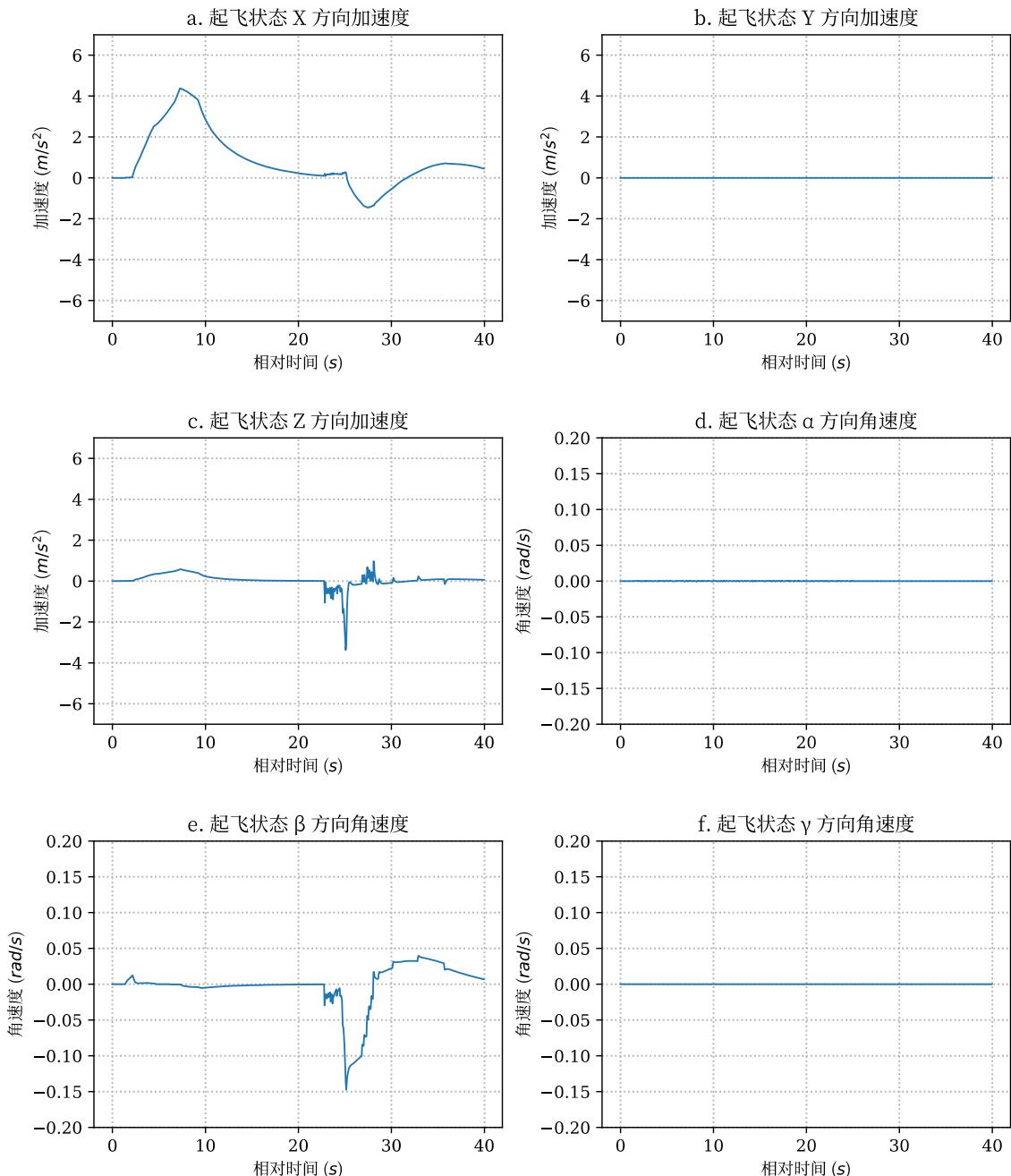


图 7.1 起飞状态加速度和角速度变化

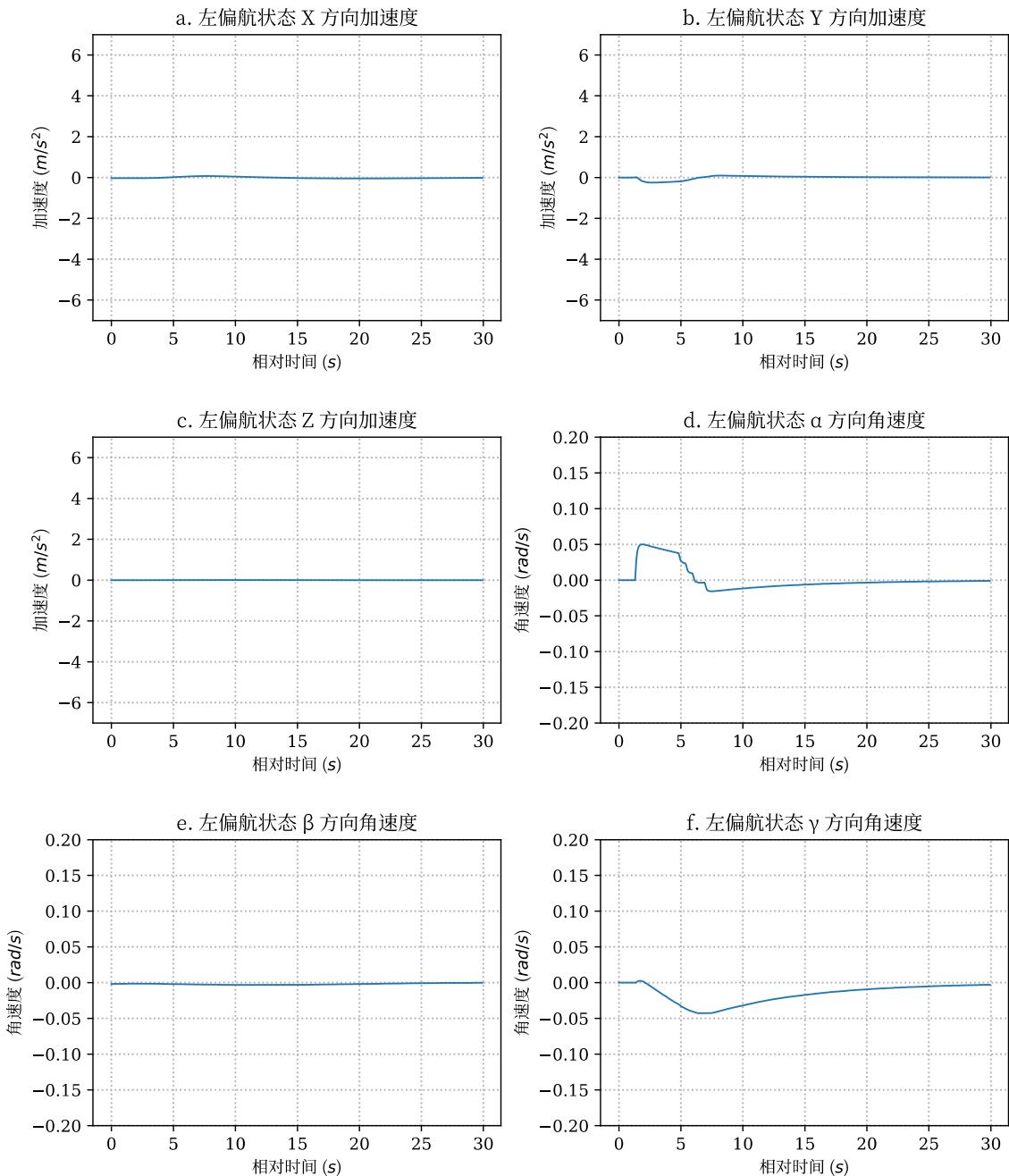
装
订
线

图 7.2 左偏航状态加速度和角速度变化

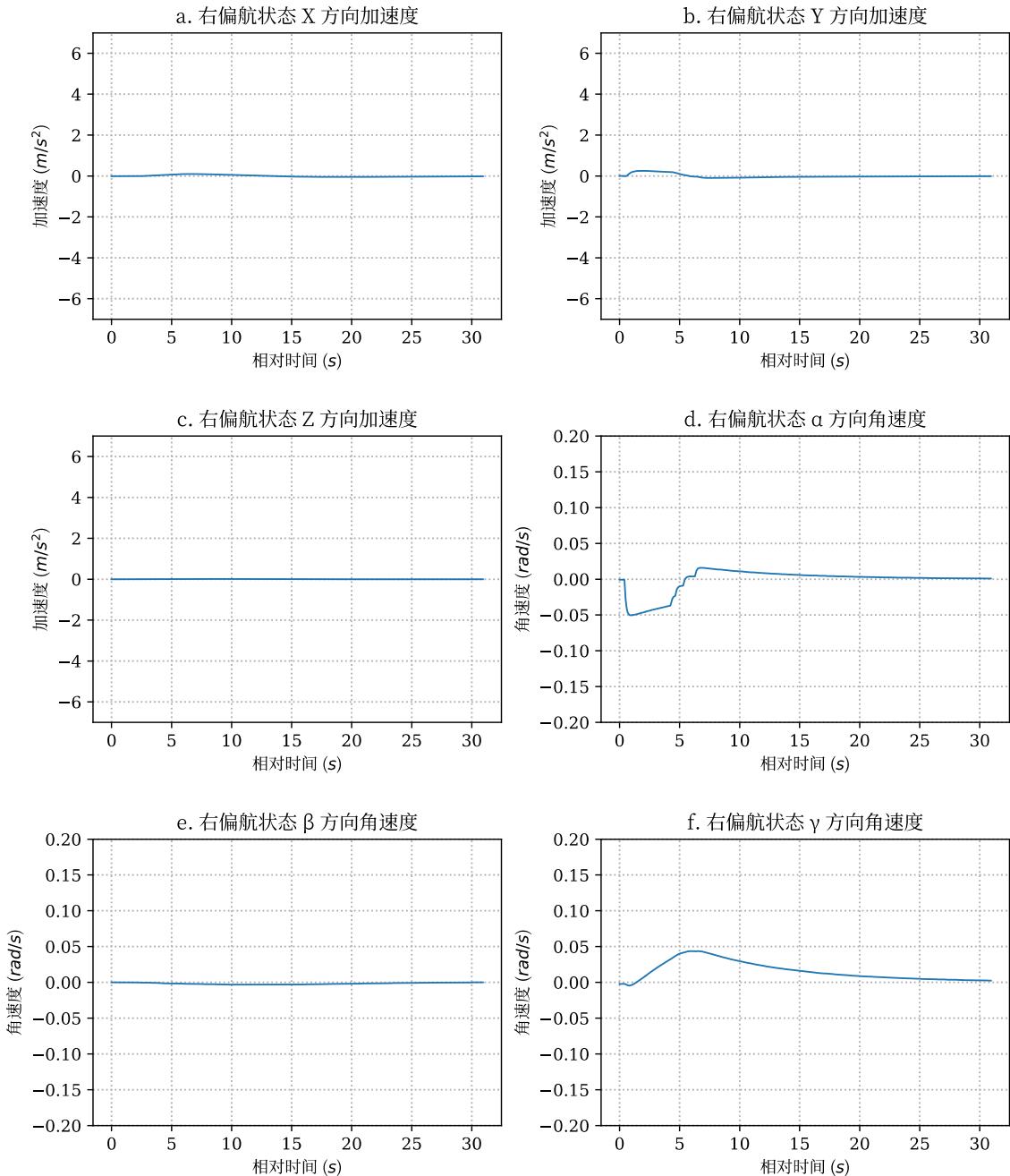
装
订
线

图 7.3 右偏航状态加速度和角速度变化

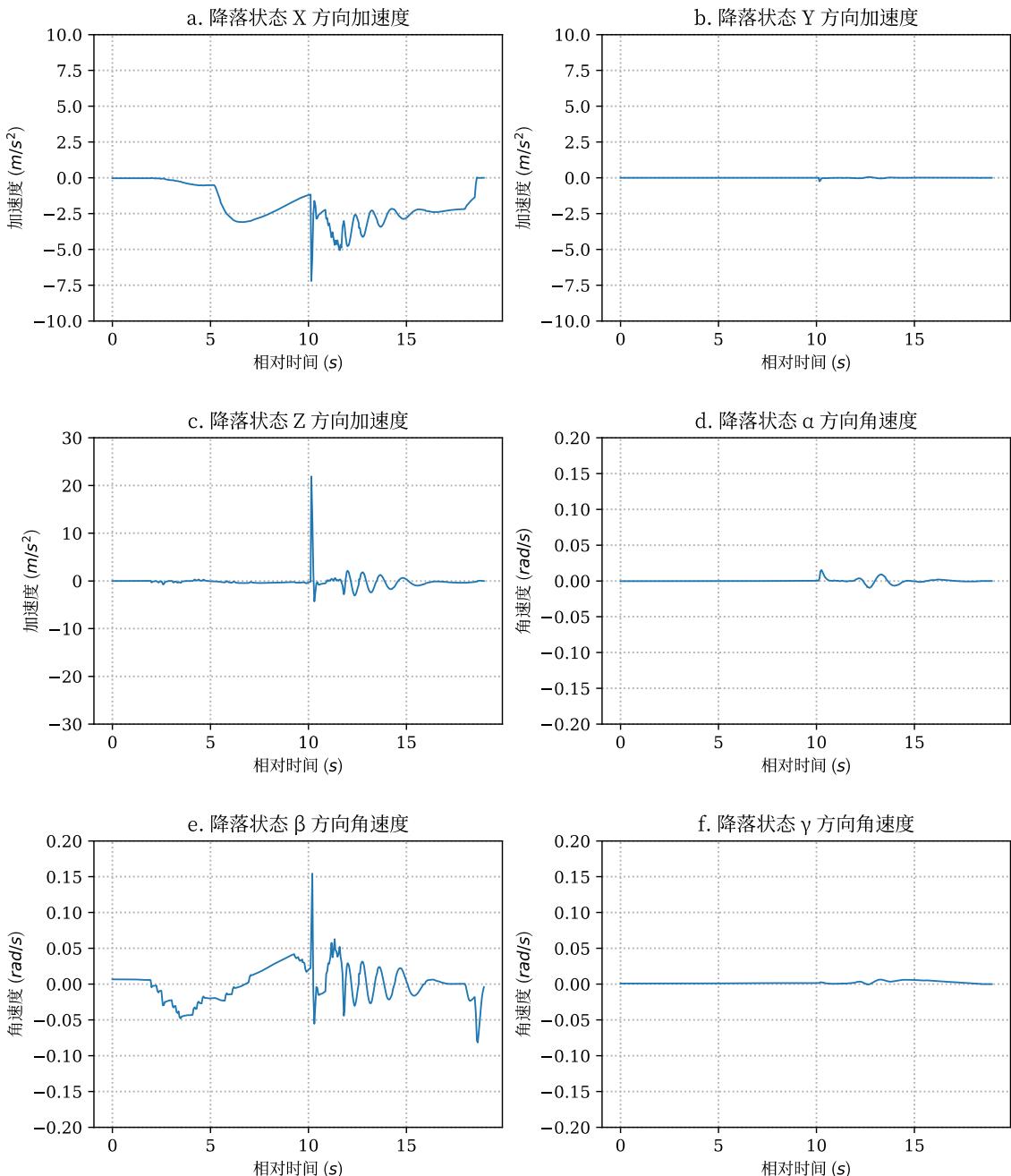


图 7.4 降落状态加速度和角速度变化

图 7.4 中由于降落在土地上、有显著颠簸，因此各项数值都出现了震荡情况。

7.1.2 采样效率

测试环境硬件：

- (1) CPU: 1.7GHz Intel Core i7 (2 逻辑核心);
- (2) 内存: 4GB DDR3;
- (3) 显卡: Intel HD Graphics 5000 512 MB。

测试环境软件：

- (1) 操作系统：Windows 7 x64；
- (2) 系统运行于 Parallels 11 虚拟机下；
- (3) FSX：10.0.60905.0。

在上述环境下，测得该模块相关各项性能指标最大如表 7.1 所示，其中 FSX 前台无论暂停还是运行都在进行图形渲染并占满一个核心。由表格可见，影响 FSX 插件采样效率的是 FSX 自身而非本课题插件的实现；FSX 在 CPU、GPU 资源短缺的情况下无法预留足够的 CPU 资源处理采样请求。

表 7.1 FSX 插件相关性能指标

项	条件	性能
FSX 渲染帧率	FSX 前台运行	20.0 FPS
FSX 插件最大采样频率	FSX 前台运行	35 ± 5 ops/s
FSX 插件最大采样频率	FSX 前台暂停	35 ± 5 ops/s
FSX 插件最大采样频率	FSX 后台暂停	80 ± 5 ops/s
FSX 插件正常 CPU 占用	FSX 前台运行	$7 \pm 5\%$
FSX 插件正常 CPU 占用	FSX 后台暂停	$8 \pm 5\%$
FSX 插件正常内存占用	FSX 运行	21MB

由于体感模拟采用 20Hz，因此目前情况下采样效率仍然满足需求。在升级硬件后可提高采样频率，实现更好的体感模拟效果。

7.2 经典洗出算法

7.2.1 体感模拟结果

以 7.1.1 中跑道起飞、左偏航、右偏航、降落到土地为输入信号进行体感模拟，得出运动平台位置姿态变化如图 7.5 到图 7.8 所示。

以起飞为例，测试飞机在跑道上从静止开始加速（图 7.5 中 0~25 秒），直到保持匀速运动后（图 7.5 中 25 秒以后）进行拉升离开地面。在这个过程中，突发的初始加速度由直接位移来复现，如图 7.5 (a) 所示。加速度继续保持时，改由倾斜协调进行复现，如图 7.5 (e) 所示，与此同时平台洗出回到原点。

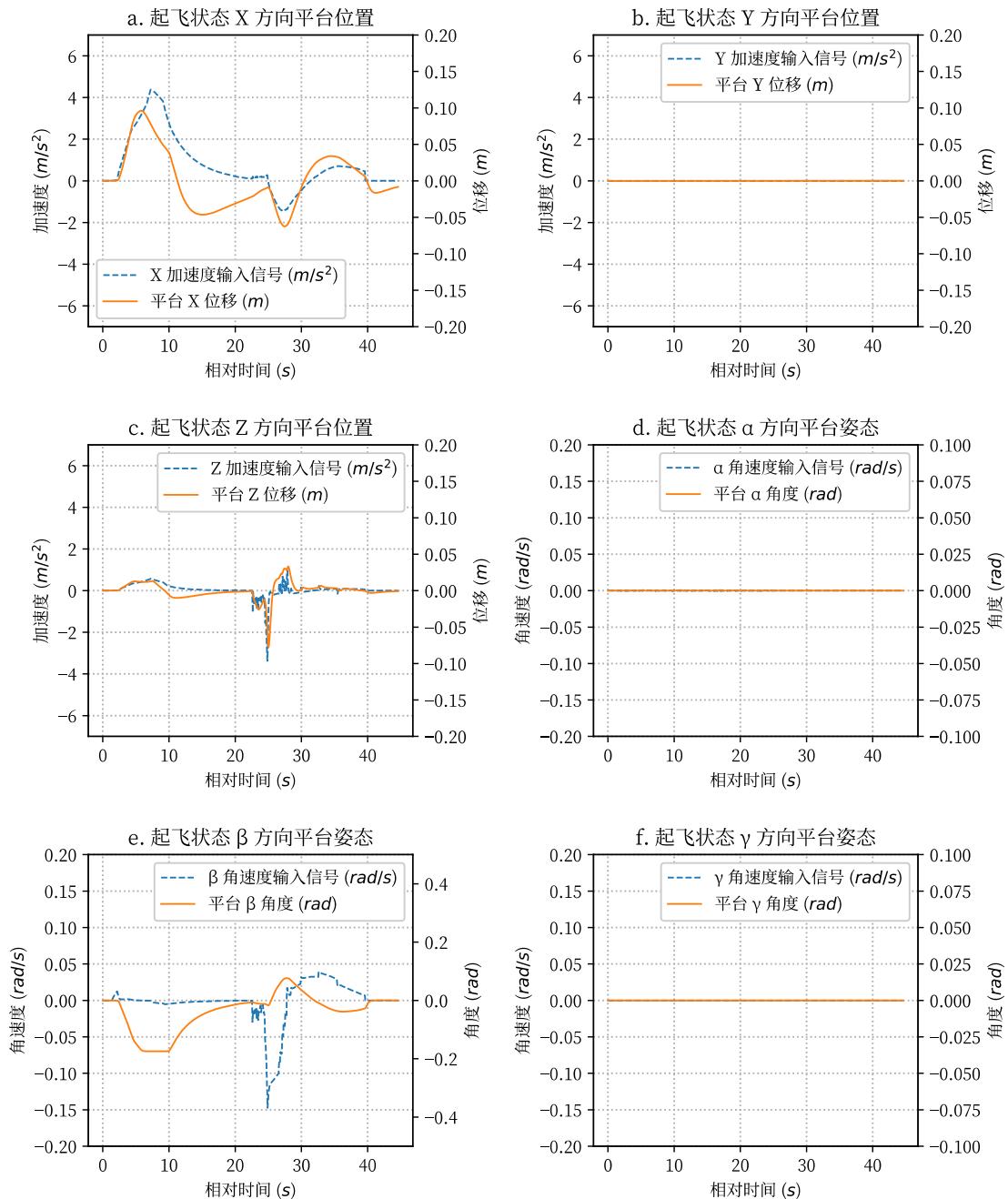


图 7.5 起飞状态运动平台位置姿态变化

装订线

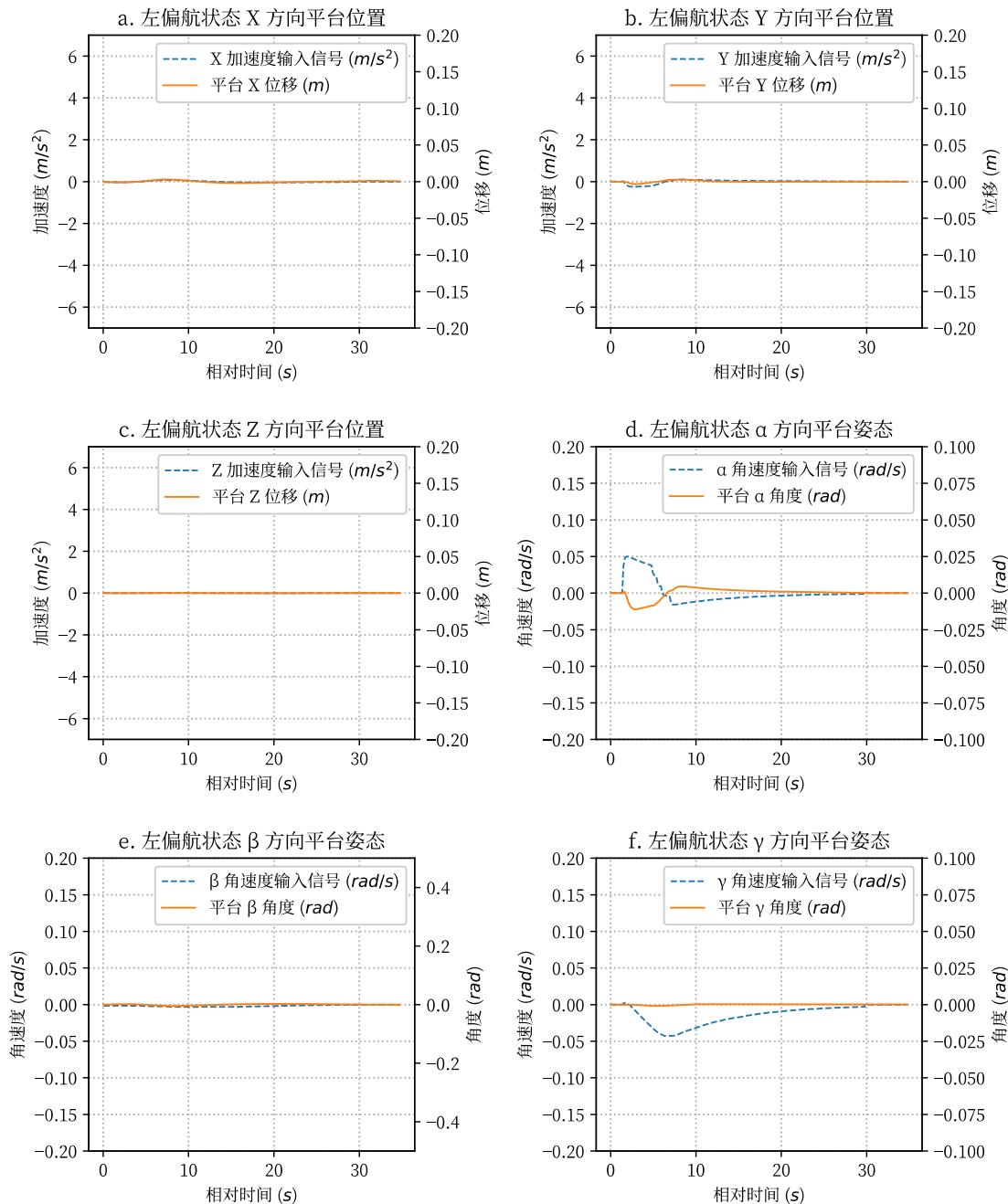


图 7.6 左偏航状态运动平台位置姿态变化

装订线

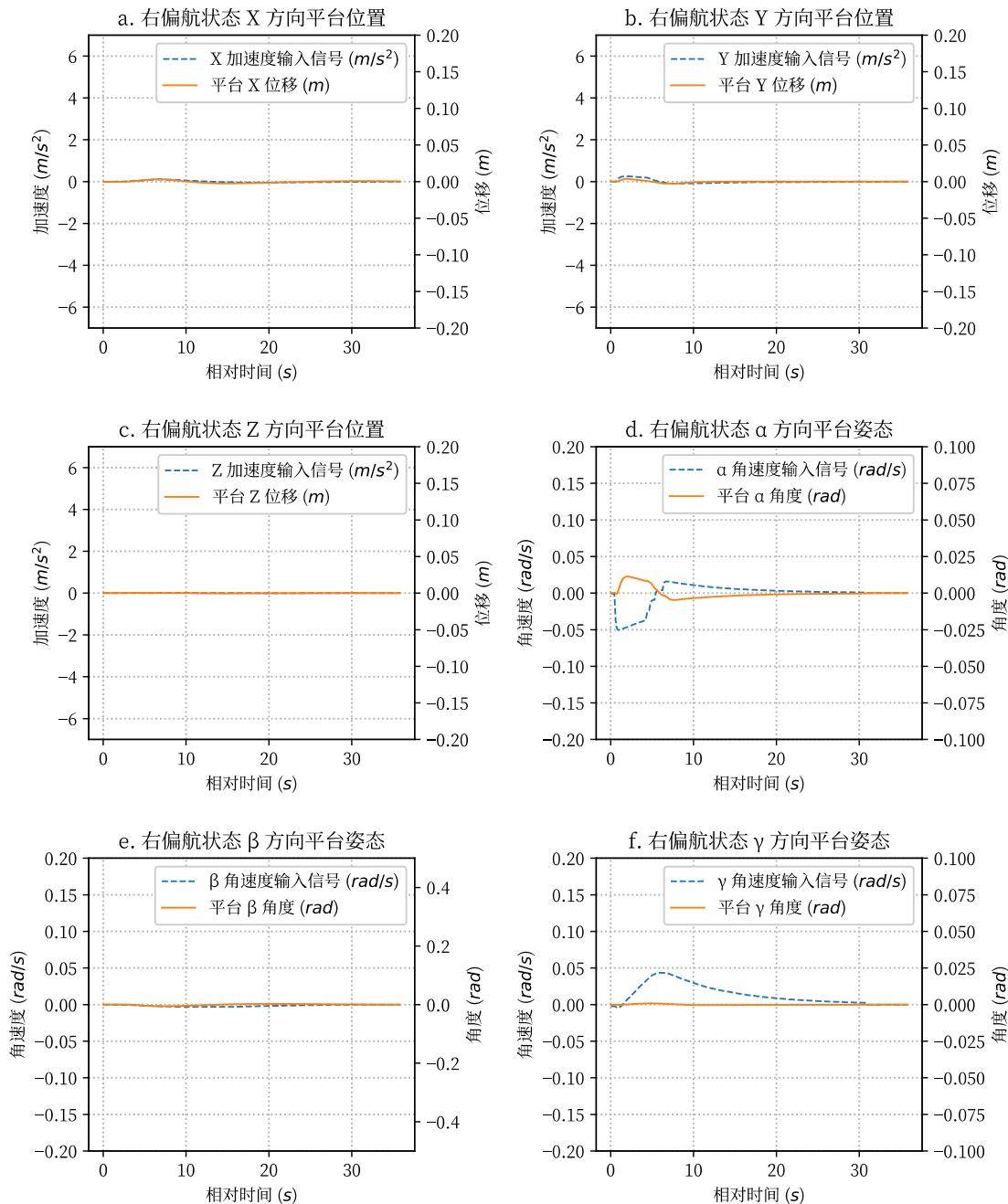


图 7.7 右偏航状态运动平台位置姿态变化

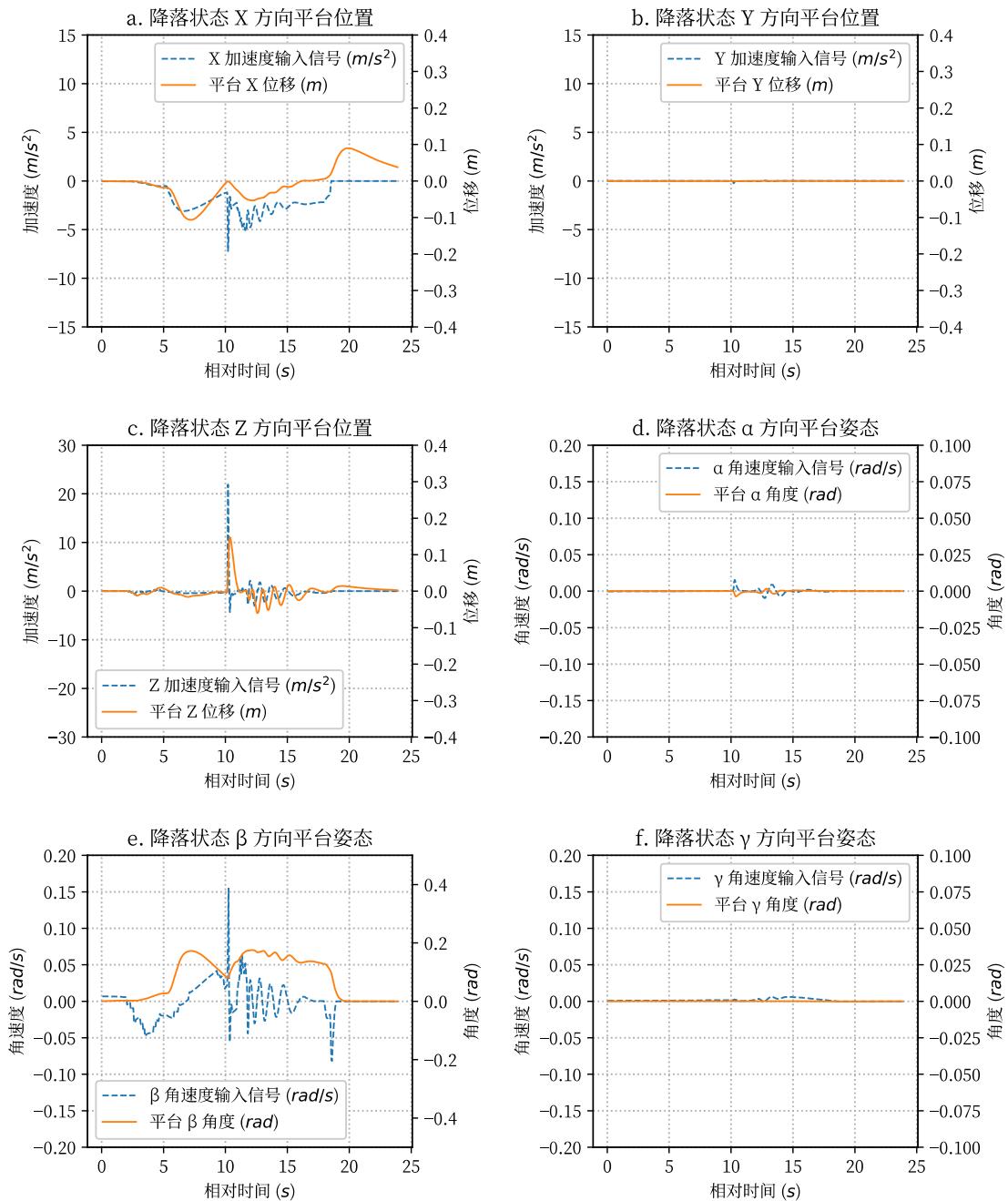
装
订
线

图 7.8 降落状态运动平台位置姿态变化

7.2.2 计算效率

测试环境硬件:

- (1) CPU: 1.7GHz Intel Core i7 (4 逻辑核心);
- (2) 内存: 8GB DDR3。

测试环境软件:

- (1) 操作系统: Mac OS X Yosemite;

(2) Python: 3.6.1。

在上述环境下，测得该模块相关各项性能指标最大如表 7.2 所示。由数据可见，经典洗出算法模块计算效率极高，能支持超过 1000Hz 的信号输入处理，本课题采用的 20Hz 绰绰有余。

表 7.2 经典洗出算法模块相关性能指标

项	性能
算法最大运行效率	6153 ± 15 cycles/s
算法正常 CPU 占用	5 ± 2 %
滤波器最大构造效率	1600 ± 50 per/s

7.3 服务端效果

装
订
线

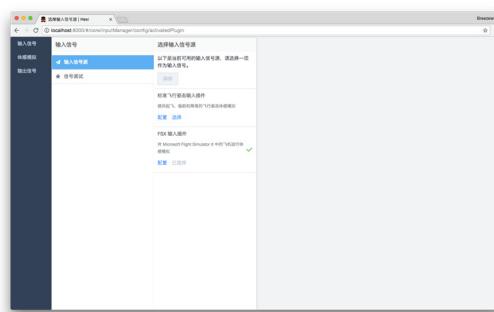


图 7.9 选择输入源

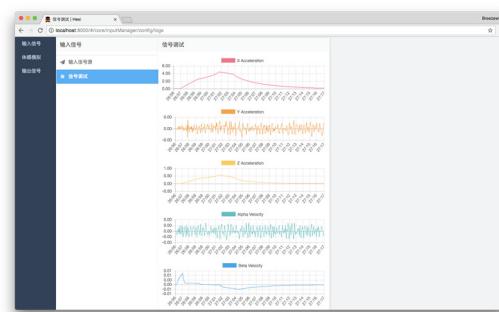


图 7.10 实时显示输入信号



图 7.11 FSX 输入插件配置

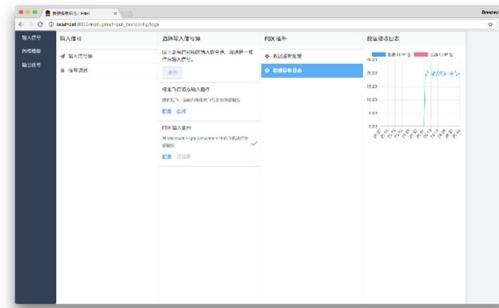


图 7.12 实时显示 FSX 输入插件数据包接收情况



图 7.13 标准飞行姿态模块选择姿态

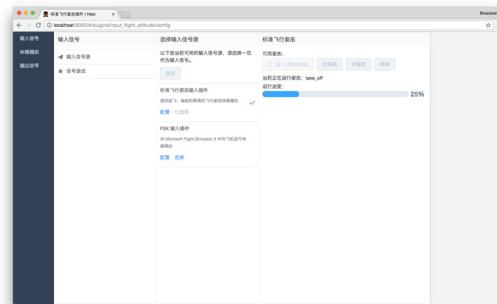


图 7.14 标准飞行姿态模块正在复现姿态



图 7.15 经典洗出算法参数配置



图 7.16 经典洗出算法自动缩放调节

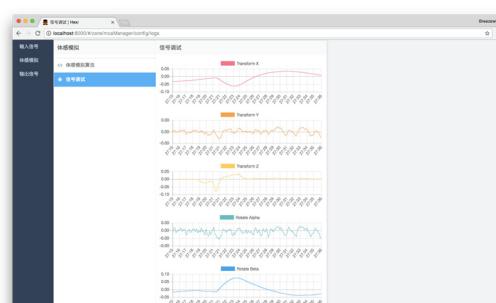


图 7.17 实时显示平台姿态信号

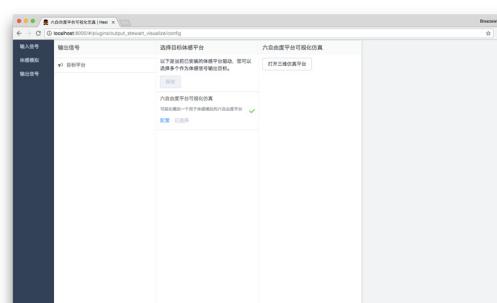


图 7.18 选择平台信号输出目标

7.4 Stewart 运动平台可视化仿真效果

使用标准飞行姿态输入模块模拟图 7.1 所示的起飞状态，姿态输入模块界面如图 7.19 所示，此刻输入信号如图 7.20 所示，经体感模拟算法得到输出信号如图 7.21 所示，可视化仿真呈现运动平台图 7.22 所示。

装
订
线

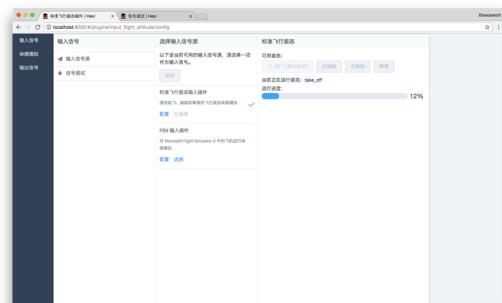


图 7.19 姿态输入模块状态

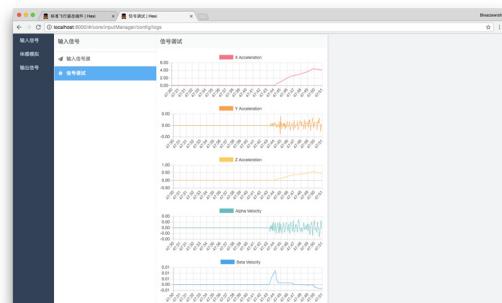


图 7.20 输入信号

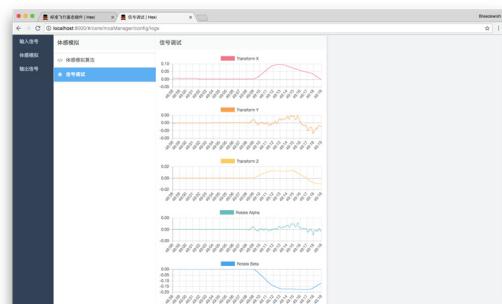


图 7.21 输出信号

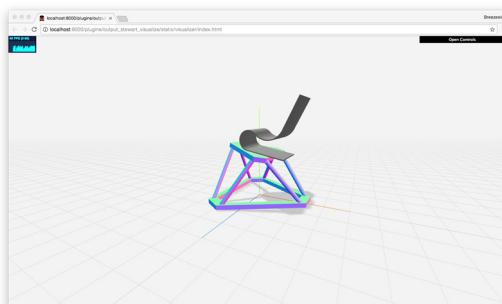


图 7.22 输出信号可视化仿真

装
订
线

8 结论和展望

8.1 结论

随着虚拟现实设备的广泛使用，个人级体感模拟需求日益增大，这是相对古老的体感模拟技术在当代的一个新需求。为了满足该需求，本文提出，体感模拟器应当控制硬件而不限于特定硬件、为需求提供基础运行环境和硬件的抽象而不限于特定需求。

本文基于现有的体感模拟理论，首先研究并实现了体感模拟算法之一“经典洗出算法”核心逻辑在计算机上的高效算法。该算法既能高效运行，又允许研究人员基于传统的模拟滤波器理论进行调参。接下来，本文详细设计并实现了一个具有极高可配置性、可扩展性的通用性体感模拟系统。该系统完全实现了可扩展插件系统，不局限于特定硬件或客户需求，并具备调试功能。最后，本文为该系统设计并实现了四个插件，分别为 Microsoft Flight Simulator X 模块：允许用户通过 FSX 飞行模拟器实时地进行体感模拟；标准飞行姿态模块：允许用户简单地进行不同飞行状态下的体感模拟；经典洗出算法模块：允许使用经典洗出算法进行体感模拟的计算；Stewart 运动平台仿真模块：允许用户在没有硬件的情况下观察体感模拟效果。

测试表明，本文设计的系统总体性能高，可满足远超需求的体感模拟精度、插件系统行之有效，可有效分隔不同功能。另外，本文实现的上述插件也能达到预期目标，可互相协作，完整串联起流程，高效地实现实时体感模拟。

8.2 展望

在未来需要进一步完善的工作：

(1) 本课题由于缺少硬件环境支持，仅能实现对输出信号进行软件仿真。在具备硬件条件以后，可考虑实现几种常见的六自由度平台硬件驱动杆（如 SCN5、SCN6）的输出模块，从而真正实现体感模拟。

(2) 本课题研究的是被最广泛应用的经典洗出算法，未来可进一步加入其它体感模拟算法，提高体感模拟逼真度。

(3) 本课题架构上假设了输出到一个具有六自由度的运动平台。鉴于目前不少个人 DIY 用户使用的是三自由度运动平台，有必要为此提供支持。具体来说，需要为各种插件提供约束条件的支持，限制只有其他插件或环境满足某些条件后才能使用该插件。

(4) 本课题目前需要对目录进行复制粘贴来添加插件，删除目录来移除插件，未来可实现用户界面简化操作。

参考文献

- [1] DENNE P. Motion platforms or motion seats[J]. Published September, 2004.
- [2] WEINBERG G, HARSHAM B. Developing a low-cost driving simulator for the evaluation of in-vehicle technologies: Proceedings of the 1st International Conference on Automotive User Interfaces and Interactive Vehicular Applications, 2009[C]. ACM.
- [3] 6DOF Costs[EB/OL]. (2010-10-03)[2017-05-10].
<https://www.xsimulator.net/community/threads/6dof-costs.2711/>.
- [4] SimTools Game Engine[EB/OL]. (2013-10-21)[2017-05-10]. <http://www.xsimulator.net/simtools-game-engine/>.
- [5] X-Sim Math Setup[EB/OL]. [2017-05-10]. http://www.x-sim.de/manual/math_setup.html.
- [6] NAHON M A, REID L D. Simulator motion-drive algorithms-A designer's perspective[J]. Journal of Guidance, Control, and Dynamics, 1990,13(2):356-362.
- [7] PAGE R L. Brief history of flight simulation[J]. SimTecT 2000 Proceedings, 2000:11-17.
- [8] HAWARD D M. The sanders teacher[J]. Flight, 1910,2(50):1006-1007.
- [9] KELLY L L. The Pilot Maker.[J]. 1970.
- [10] 肖慧琼. 六自由度平台体感算法研究[D]. 北京交通大学, 2014.
- [11] FISCHER M, SEHAMMER H, PALMKVIST G. Motion cueing for 3-, 6-and 8-degrees-of-freedom motion systems[J]. DSC Europe, 2010.
- [12] STEWART D. A platform with six degrees of freedom[J]. Proceedings of the institution of mechanical engineers, 1965,180(1):371-386.
- [13] ALLEN L. Evolution of flight simulation[M]//Flight Simulation and Technologies. 1993:3545.
- [14] MIERMEISTER P, LÄCHELE M, BOSS R, et al. The CableRobot simulator large scale motion platform based on cable robot technology: Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on, 2016[C]. IEEE.
- [15] Motion Cueing Algorithms[EB/OL]. (2017-03-24)[2017-05-10].
<http://www.kyb.tuebingen.mpg.de/research/dep/bu/motion-perception-and-simulation/motion-cueing-algorithms.html>.
- [16] von der HEYDE M, RIECKE B E. How to cheat in motion simulation – comparing the engineering and fun ride approach to motion cueing[J]. Max-Planck-Institut für biologische Kybernetik, 2001.
- [17] REID L D, NAHON M A. Flight simulation motion-base drive algorithms: Part 1. Developing and testing equations[R].University of Toronto, 1985.
- [18] REID L D, NAHON M A. Flight simulation motion-base drive algorithms: Part 2. Selecting the system parameters[R].University of Toronto, 1986.
- [19] PARRISH R V, DIEUDONNE J E, MARTIN JR D J. Coordinated adaptive washout for motion simulators[J]. Journal of aircraft, 1975,12(1):44-50.

- [20] SIVAN R, ISH-SHALOM J, HUANG J. An optimal control approach to the design of moving flight simulators[J]. IEEE Transactions on Systems, Man, and Cybernetics, 1982,12(6):818-827.
- [21] SZUFNAROWSKI F. Stewart platform with fixed rotary actuators: a low cost design study[J]. Advances in Medical Robotics, 2013.
- [22] WALDRON K J. The constraint analysis of mechanisms[J]. Journal of Mechanisms, 1966,1(2):101-114.
- [23] MASORY O, WANG J. Workspace evaluation of Stewart platforms[J]. Advanced robotics, 1994,9(4):443-461.
- [24] KANDEL E R, SCHWARTZ J H, JESSELL T M, et al. Principles of neural science[M]. McGraw-hill New York, 2000.
- [25] MEIRY J L. The vestibular system and human dynamic space orientation.[D]. Massachusetts Institute of Technology, 1965.
- [26] MACDONALD S P. How Human Physiology Processes and Responds to Motion[EB/OL]. (2011-02-15)[2017-05-10]. <http://ezinearticles.com/?How-Human-Physiology-Processes-and-Responds-to-Motion&id=5931085>.
- [27] LEIGH R J, ROBINSON D A, ZEE D S. A Hypothetical Explanation for Periodic Alternating Nystagmus: Instability in the Optokinetic-Vestibular System[J]. Annals of the New York Academy of Sciences, 1981,374(1):619-635.
- [28] STANGOR C, WALINGA J. Introduction to psychology[M]. Flatworld Knowledge, 2010.
- [29] ORMSBY C C, YOUNG L R. Perception of static orientation in a constant gravitoinertial environment.[J]. Aviation, space, and environmental medicine, 1976.
- [30] YOUNG L R, MEIRY J L. A revised dynamic otolith model.[J]. Aerospace medicine, 1968,39(6):606.
- [31] PETERS R A. Dynamics of the vestibular system and their relation to motion perception, spatial disorientation, and illusions[J]. 1969.
- [32] 叶正茂, 张辉, 张晋, 等. 飞行模拟器的洗出算法研究[J]. 机床与液压, 2006(6):177-180.
- [33] 覃征. 软件工程与管理[M]. 清华大学出版社有限公司, 2005.
- [34] METZ S. Practical Object-Oriented Design in Ruby: An Agile Primer[M]. Pearson Education, 2012.
- [35] KEGEL D. The C10K problem[Z]. 2006.
- [36] HU J C, PYARALI I, SCHMIDT D C. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks: Global Telecommunications Conference, 1997. GLOBECOM'97., IEEE, 1997[C]. IEEE.
- [37] BRAY T. The javascript object notation (json) data interchange format[J]. 2014.
- [38] WIKIPEDIA. Microsoft Flight Simulator[EB/OL]. (2017-05-25)[2017-05-26]. https://en.wikipedia.org/w/index.php?title=Microsoft_Flight_Simulator&oldid=782244983.
- [39] AUTHORS P. protobuf-c - C bindings for Google's Protocol Buffers - Google Project Hosting[Z].

- [40] FENG J, LI J. Google protocol buffers research and application in online game: Conference Anthology, IEEE, 2013[C]. IEEE.

装
订
线

谢 辞

毕业论文（设计）是本科阶段学得各项能力的综合体现。本次毕业设计让我对此深有体会。在研究、设计、实现课题的过程中，软件学院老师们先前教给我的知识和学习知识的能力给了我非常大的帮助，让我最终能系统地构建出超越自己本科阶段所有课程项目水平的软件。

首先，我衷心感谢这次毕业设计的指导老师尹长青老师。在我迷茫于选题时，尹老师从不同角度给出了专业的建议，这才有了本文的研究设计内容。尹老师对我个人能力和潜力有非常到位的评估，甚至比我自己都更了解。坦诚地说，在刚定下课题的时候我完全没有头绪，非常担心由于并非自己擅长的领域、能力不足而无法很好地完成项目，另外当时也对该课题之于自己专业能力的发挥和提升程度有诸多疑问。但随着研究深入，我逐渐看清了问题完整的面貌，发现了其有趣、有挑战性、有价值的部分，结合自己的知识和学习能力，最终较好地完成了该课题。研究过程中，尹老师给的建议也令我大有收获，令我留下了深刻印象。

其次，我要感谢在研究上给予了我很多帮助的机械学院的熊德浩同学。体感模拟方向现有论文大多由机械工程方向研究人员撰写，论文中存在大量对于机械知识的认知假设，这些一笔带过的各种专业知识对于完全不熟悉机械领域的我而言是很大的挑战，经常出现术语理解有偏差、抓不住关键问题、概念不了解等问题。熊德浩同学利用自己机械专业的知识为我解答了很多疑惑，并为关键问题指出了后来被证明是正确的探索方向，帮助我避免了很多由于跨领域不熟悉而带来的弯路，节约了大量宝贵的时间。正是在他的帮助下，我才得以在相对较短的时间里成功学习、理解并验证了体感技术的理论，为课题其余部分例如插件系统和高性能通信的设计和实现留出了充足的时间。

再次，我要感谢过去四年中传授我各项专业知识、学习方法并给予充分实践机会的老师们。刘岩老师的课程令我有机会完整地设计实践了基于事件驱动的系统架构，该架构在本课题中起到了极其关键的作用；侯捷老师的课程让我对各种软件设计模式有了深入的学习和了解，使我能以非常专业的角度进行软件架构设计，这些技术思想令我受益良多，已渗透到项目的方方面面；黄杰老师的课程让我对软件工程有了专业而又系统的了解，使我得以在本文中运用相关知识和理论撰写出专业的内容；饶卫雄老师、杜庆峰老师、朱宏明老师、夏波涌老师等软件学院教师开设的课程也都为我完成该课题提供了必要的知识和学习方法。本课题综合运用了 C#、Python、JavaScript 等多种编程语言，结合了 Web、3D 渲染、WinForm、异步多线程、TCP、UDP 等多种技术，使用最恰当的工具或技术实现了各项功能。正是由于这些老师的传道受业解惑，我才得以触类旁通，积累出这样综合性的能力，以很高的质量完成本课题。

最后，感谢一直默默帮助并支持我发展兴趣的父母，没有你们一如既往对个人兴趣的支持我一定不会取得现在的成就；感谢在四年间出现在我身边的同学和朋友们，尤其是室友们和各种课程项目一同组队的同学们，你们都在我生活中留下了深刻的印记，带给了我美好的记忆并让我有积极乐观的态度面对各种挑战。