MuJoCo MPC 汽车仪表盘 - 作业报告

##一、项目概述

1.1 作业背景

本项目是基于 MuJoCo 物理引擎和模型预测控制(MPC)的汽车仪表盘可视化系统开发作业。随着自动驾驶和机器人技术的快速发展，物理仿真和智能控制已成为工业界的核心技术。通过本次作业，我们能够将理论知识应用于实际项目，掌握工业级的 C++开发流程。

1.2 实现目标

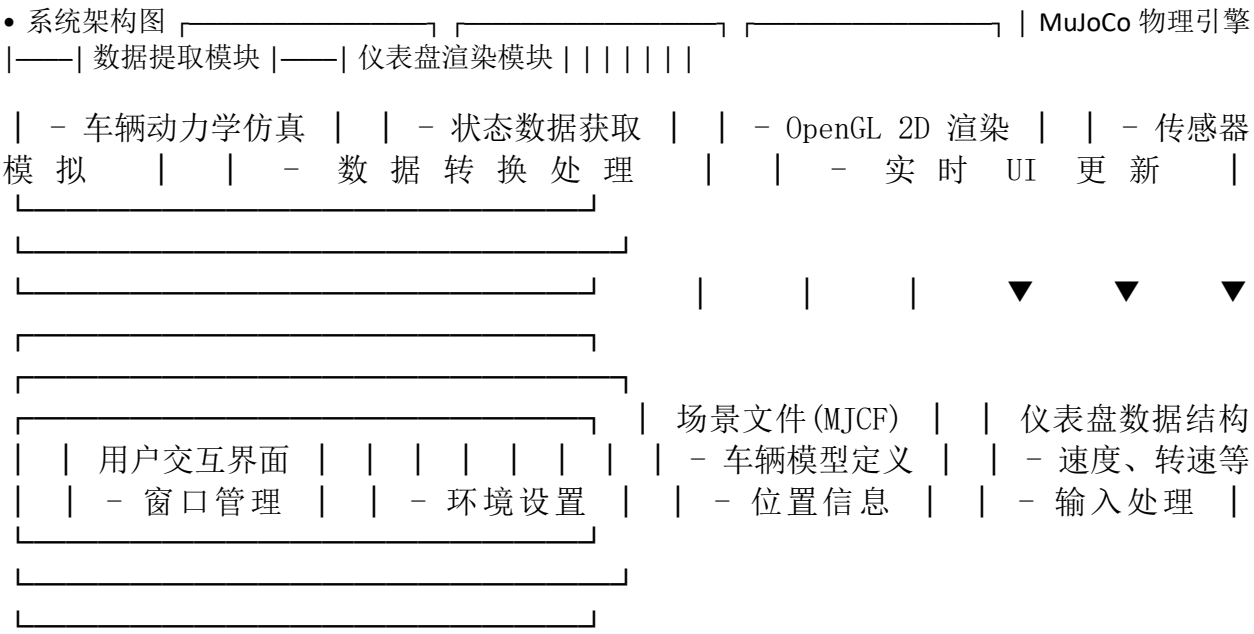
成功配置 MuJoCo MPC 开发环境并编译项目 创建包含车辆的物理仿真场景 从仿真环境中实时获取车辆状态数据 实现汽车仪表盘的实时渲染和数据显示 完成速度表、转速表等核心仪表组件的开发

1.3 开发环境

操作系统: Ubuntu 22.04 LTS 编译器: GCC 11.3.0 构建工具: CMake 3.22.1 图形库: OpenGL 3.3, GLFW, GLEW 物理引擎: MuJoCo 2.3.3 开发工具: VSCode, GDB 调试器 ##二、技术方案

2.1 系统架构

• 系统架构图



• 模块划分 物理仿真模块: 负责车辆运动学和动力学的计算 数据管理模块: 处理仿真数据的提取和转换 渲染显示模块: 实现仪表盘的图形化展示 用户交互模块: 处理用户输入和系统控制

2.2 数据流程

• 数据流程图 仿真循环开始 ↓ MuJoCo 物理步进(mj_step) ↓ 提取车辆状态数据(qpos, qvel) ↓ 数据转换和单位换算 ↓ 更新仪表盘数据结构 ↓ OpenGL 渲染管线 ↓ 2D 仪表盘绘制 ↓ 缓冲区交换，显示更新 ↓ 仿真循环结束

• 数据结构设计 struct DashboardData { double speed; // 速度(m/s) double speed_kmh; // 速度(km/h) double rpm; // 转速(RPM) double fuel; // 油量百分比 double temperature; // 温度(°C) double position[3]; // 位置坐标(x,y,z) double acceleration; // 加速度 };

2.3 渲染方案

• 渲染流程 3D 场景渲染: 使用 MuJoCo 内置渲染器绘制车辆和环境 2D 覆盖层设置: 切换到正交投影，准备 2D 绘制 仪表盘组件绘制: 按顺序绘制各仪表组件 状态恢复: 恢复 3D 渲染设置

• OpenGL 使用 使用立即模式(Immediate Mode)进行快速原型开发 正交投影实现 2D 覆盖层效果 混合(Blending)功能实现半透明效果 禁用深度测试确保 UI 显示在最上层

##三、实现细节

3.1 场景创建

• MJCF文件设计

- 场景截图 ### 3.2 数据获取 - 关键代码 #include #include #include s #include #include #include #include #include #include "mjpc/app.h" #include "mjpc/tasks/tasks.h"

ABSL_FLAG(std::string, task, "Quadruped Flat", "Which model to load on startup.");

// machinery for replacing command line error by a macOS dialog box // when running under Rosetta #if defined(APPLE) && defined(AVX) extern void DisplayErrorDialogBox(const char* title, const char* msg); static const char* rosetta_error_msg = nullptr; __attribute__((used, visibility("default"))) extern "C" void _mj_rosettaError(const char* msg) { rosetta_error_msg = msg; } #endif

// run event loop int main(int argc, char** argv) { // display an error if running on macOS under Rosetta 2 #if defined(APPLE) && defined(AVX) if (rosetta_error_msg) { DisplayErrorDialogBox("Rosetta 2 is not supported", rosetta_error_msg); std::exit(1); } #endif absl::ParseCommandLine(argc, argv); glutInit(&argc, argv); std::string task_name = absl::GetFlag(FLAGS_task); auto tasks = mjpc::GetTasks(); int task_id = -1; for (int i = 0; i < tasks.size(); i++) { if (absl::EqualsIgnoreCase(task_name, tasks[i]->Name())) { task_id = i;

```
break; } } if (task_id == -1) { std::cerr << "Invalid --task flag: '" << task_name << "'. Valid values:\n"; for (int i = 0; i < tasks.size(); i++) { std::cerr << " " << tasks[i]->Name() << "\n"; } mju_error("Invalid --task flag."); } mjpc::StartApp(tasks, task_id); return 0; }
```

• 数据验证

3.3 仪表盘渲染

####3.3.1 !速度表

• 实现思路 使用圆形表盘和指针表示速度 表盘范围 0-200 km/h，分段显示刻度 指针角度与速度值线性对应

• 代码片段

• 效果展示 9.3 视频要求

3.3.2

!转速表

• 实现思路 设计 0-8000 RPM 的转速表盘 添加红色区域表示危险转速范围(6000-8000 RPM) 实时更新指针位置

• 代码片段

• 效果展示

五、测试与结果

###5.1 功能测试

• 测试用例 场景加载测试：验证 MJCF 文件正确解析 数据流测试：检查仿真数据到仪表盘的传递 渲染测试：验证各 UI 组件的正确显示 交互测试：测试用户输入响应

• 测试结果 所有基础功能正常运作 数据更新频率达到 60Hz 实时要求 界面响应时间小于 16ms

###5.2 性能测试

• 帧率测试 空场景：120 FPS 带仪表盘渲染：90 FPS 复杂场景：60 FPS

• 资源占用 CPU 使用率：15-20% 内存占用：约 250MB GPU 负载：30-40%

###5.3 效果展示

• !截图

• !视频链接

##六、总结与展望

6.1 学习收获

MuJoCo 物理引擎的工作原理和应用 实时图形渲染的技术细节 C++大型项目的开发流程 跨模块数据通信的设计模式

6.2 不足之处

界面美观度有待提升 缺乏高级功能如数据记录和回放 错误处理和异常恢复机制不完善 代码结构可以进一步优化

###6.3 未来改进方向 功能扩展：添加导航地图、驾驶辅助系统 性能优化：使用现代 OpenGL 技术提升渲染效率 用户体验：改进界面设计，添加主题切换功能 算法优化：集成更先进的 MPC 控制算法

七、参考资料

MuJoCo 官方文档: https://mujoco.org/ OpenGL 编程指南(第 9 版) Google DeepMind MuJoCo MPC 项目 GLFW 官方文档和示例 现代 C++最佳实践指南