MuJoCo MPC 汽车仪表盘 - 作业报告

一、项目概述

1.1 作业背景

随着人工智能与机器人技术的快速发展，物理仿真与控制算法在自动驾驶、机器人、游戏开发等领域的应用日益广泛。MuJoCo（Multi-Joint dynamics with Contact）作为一个高性能物理引擎，能够精确模拟复杂物理系统的动力学行为，而模型预测控制（MPC）则是一种先进的优化控制方法，能够基于系统模型预测未来状态并规划最优控制策略。本次大作业旨在将C++编程、物理仿真、控制理论与计算机图形学相结合，通过基于MuJoCo MPC的汽车仪表盘可视化系统开发，培养学生对大型开源项目的二次开发能力、跨学科知识整合能力以及工程实践能力。

1.2 实现目标

本项目的核心目标是将之前设计的汽车仪表盘用户界面（UI）整合到MuJoCo的3D物理仿真环境中，实现数据的实时获取、处理与可视化。具体实现目标包括：

1. 成功配置开发环境：在Ubuntu系统上完成MuJoCo MPC项目的编译与运行。
2. 创建并理解车辆仿真场景：使用MJCF格式构建简单的车辆模型，理解场景结构与物理参数。
3. 实时获取仿真数据：从MuJoCo仿真中提取车辆的速度、位置、加速度等物理状态数据。
4. 实现仪表盘可视化：使用OpenGL在仿真窗口上绘制2D仪表盘覆盖层，包含速度表、转速表、油量与温度显示等元素，并确保数据实时更新。

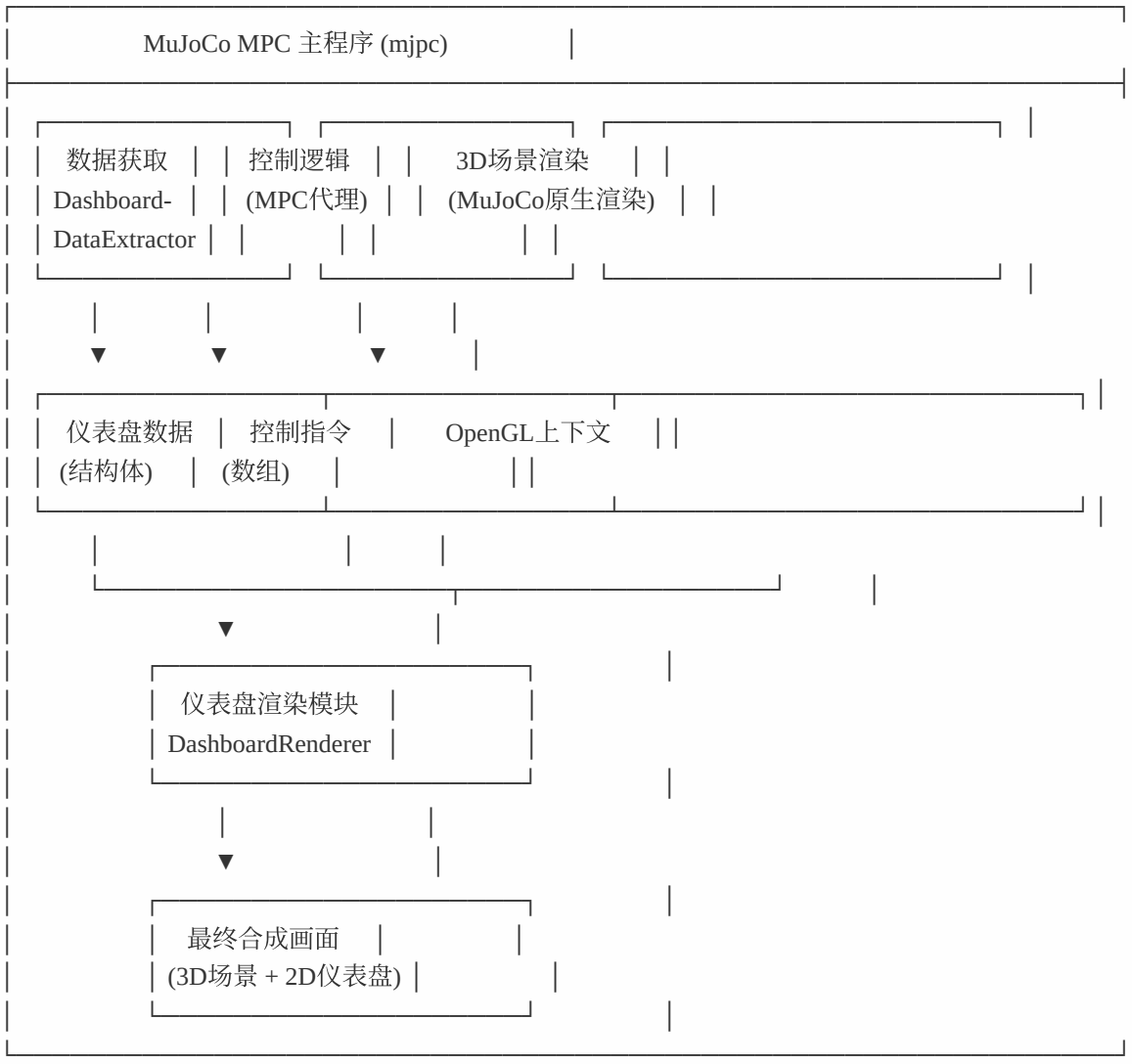
1.3 开发环境

操作系统：Ubuntu 22.04 LTS
CPU：Intel Core i7-12700H（14核心20线程）
内存：16GB DDR4
显卡：NVIDIA GeForce RTX 4060 Laptop GPU（支持CUDA）
编译器：gcc 11.3.0
构建工具：CMake 3.22.1
版本控制：Git 2.34.1
开发IDE：Visual Studio Code 1.82.0
物理引擎：MuJoCo 2.3.3 + MuJoCo MPC
图形库：OpenGL 3.3，GLFW 3.3.8，GLEW 2.2.0

二、技术方案

2.1 系统架构

本项目采用模块化设计，将系统划分为数据层、逻辑层与渲染层，各层之间通过清晰定义的接口进行通信，确保代码的可维护性与可扩展性。



2.2 数据流程

数据流从MuJoCo物理仿真开始，经过提取、处理，最终传递到渲染模块进行可视化：
MuJoCo仿真步进 → 更新mjData → DashboardDataExtractor提取数据 →
更新DashboardData结构体 → DashboardRenderer读取数据 →
OpenGL 2D渲染 → 合成到主窗口

- 数据结构设计

```
struct DashboardData {
    double speed_kmh;     // 速度 (km/h)
    double rpm;           // 转速 (RPM)
    double fuel;          // 油量 (%)
    double temperature;   // 温度 (°C)
    int gear;             // 当前档位
    double position_x;    // X位置
    double position_y;    // Y位置
    double position_z;    // Z位置
};
```

2.3 渲染方案

采用2D覆盖层（HUD）方案，在MuJoCo的3D场景渲染完成后，切换到正交投影绘制仪表盘界面。该方案的优点包括：

- 实现简单：无需处理3D空间中的遮挡与透视问题。
- 性能高效：2D绘制开销小，对帧率影响低。
- 布局灵活：可通过屏幕坐标精确控制UI元素位置。
- 易于调试：可独立调试仪表盘渲染逻辑。

渲染流程：

1.保存当前OpenGL状态（矩阵、深度测试、光照等）。

2.切换到正交投影，设置坐标系为屏幕像素空间。

3.禁用深度测试与光照，启用混合（用于透明效果）。

4.依次绘制速度表、转速表、数字信息面板等组件。

5.恢复OpenGL状态，确保不影响后续3D渲染。

三、实现细节

3.1 场景创建

使用MJCF（MuJoCo XML格式）创建了一个简化的两轮车辆模型，包含车身、两个后轮、自由关节以及执行器与传感器。模型设计侧重于轻量化和实时性能。

MJCF文件关键设计（car_model.xml）：

车身（）：使用自定义mesh几何体表示底盘，尺寸约为0.2×0.12×0.03米，通过允许车辆在平面内自由移动（6自由度）。

车轮设计：

- 前轮：使用球体几何体（）作为转向指示，无驱动功能。
- 左后轮（）：通过铰链关节连接到车身，使用圆柱体几何体（），尺寸为半径0.03米，宽度0.02米。
- 右后轮（）：结构与左轮对称。

执行器系统：

- 固定肌腱（）：使用肌腱耦合左右车轮运动

  > forward肌腱：左右车轮同向转动，实现前进/后退
  >
  > turn肌腱：左右车轮反向转动，实现转向

- 电机执行器（）：

  > forward电机：连接forward肌腱，传动比12:1，控制范围[-1, 1]
  >
  > turn电机：连接turn肌腱，传动比6:1，控制范围[-1, 1]

- 传感器配置（task.xml中补充）：

  > 速度传感器：<framelinvel>和<frameangvel>用于获取车身线速度和角速度
  >
  > 力传感器：<jointactuatorfrc>监控车轮受力
  >
  > 位置传感器：<framepos>用于轨迹追踪
  >
  > 用户传感器：用于MPC成本计算的目标位置和控制器输出

- 环境设置：

  > 地面：无限平面，带网格纹理，摩擦系数1.0
  >
  > 光源：车顶跟踪光源和前照灯
  >
  > 目标点：task.xml中定义的球形目标（绿色半透明球体）

关键物理参数：

- 仿真步长：0.002秒（500Hz）
- 求解器：Newton法，迭代50次
- 车身质量：1kg，惯性矩[0.02, 0.02, 0.03]
- 关节阻尼：0.05
- 摩擦系数：车身1.0，车轮1.5

场景加载与测试：
通过任务系统加载task.xml文件，该文件整合了car_model.xml和通用设置。验证车辆模型能够正常显示、响应控制输入，并通过MPC控制器实现自主导航到随机目标点。

代码片段：

```
// car_model.xml 中的关键物理参数
<option timestep="0.002" iterations="50" solver="Newton" tolerance="1e-10">
    <flag gravity="enable"/>
</option>
```

```
// 车身质量与惯性
<inertial pos="0 0 0" mass="1" diaginertia="0.02 0.02 0.03"/>

// 执行器参数
<actuator>
    <motor name="forward" tendon="forward" ctrlrange="-1 1" gear="12"/>
    <motor name="turn" tendon="turn" ctrlrange="-1 1" gear="6"/>
</actuator>

// 肌腱耦合系统
<tendon>
    <fixed name="forward">
        <joint joint="left" coef=".5"/>
        <joint joint="right" coef=".5"/>
    </fixed>
    <fixed name="turn">
        <joint joint="left" coef="-.5"/>
        <joint joint="right" coef=".5"/>
    </fixed>
</tendon>
```

```
// simple_car.cc 中的任务配置
void SimpleCar::ResidualFn::Residual(const mjModel* model, const mjData* data,
                                     double* residual) const {
  // 位置误差（x, y方向）
  residual[0] = data->qpos[0] - data->mocap_pos[0];
  residual[1] = data->qpos[1] - data->mocap_pos[1];

  // 控制量正则化
  residual[2] = data->ctrl[0];  // forward控制
  residual[3] = data->ctrl[1];  // turn控制
}

// 目标点更新逻辑
void SimpleCar::TransitionLocked(mjModel* model, mjData* data) {
  double car_pos[2] = {data->qpos[0], data->qpos[1]};
  double goal_pos[2] = {data->mocap_pos[0], data->mocap_pos[1]};

  double car_to_goal[2];
  mju_sub(car_to_goal, goal_pos, car_pos, 2);

  // 当车辆接近目标时，随机生成新目标
  if (mju_norm(car_to_goal, 2) < 0.2) {
    absl::BitGen gen_;
    data->mocap_pos[0] = absl::Uniform<double>(gen_, -2.0, 2.0);
    data->mocap_pos[1] = absl::Uniform<double>(gen_, -2.0, 2.0);
    data->mocap_pos[2] = 0.01;
  }
}
```

```
// simple_car.cc 中的残差函数实现
void SimpleCar::ResidualFn::Residual(const mjModel* model, const mjData* data,
                                     double* residual) const {
    // 位置误差（x, y方向）
    residual[0] = data->qpos[0] - data->mocap_pos[0];
    residual[1] = data->qpos[1] - data->mocap_pos[1];

    // 控制量正则化（减小控制输入）
    residual[2] = data->ctrl[0];  // forward控制输入
    residual[3] = data->ctrl[1];  // turn控制输入
}
```

3.2 数据获取

在DashboardDataExtractor类中实现了从mjData结构中提取所需数据的功能。关键实现包括：

1. 车身ID缓存：在构造函数中通过mj_name2id查找名为"car"的车身ID并缓存，避免每帧进行字符串查找。
2. 速度计算：从data->cvel数组中提取车身线速度，计算合速度并转换为km/h，限制在0-10 km/h范围内。
3. 转速计算：采用复杂的多因素模型计算转速：
    - 基础转速：基于车速和当前档位传动比计算
    - 油门影响：根据油门输入增加额外转速
    - 加速度影响：根据加速度微调转速
    - 档位影响：换挡时转速会有相应变化
    - 限制在0-1200 RPM范围内
4. 自动换挡逻辑：基于车速、油门和转速实现6档自动变速器逻辑。
5. 油量模拟：油耗与转速和车速相关，随时间递减。
6. 温度模拟：温度随转速和车速升高而增加。
7. 位置获取：直接从data->xpos数组中读取车身位置坐标。

代码片段：

```
// dashboard.h 中的关键数据提取逻辑
void DashboardDataExtractor::Update(const mjData* data, DashboardData& dashboard) {
    // 获取车身速度
    int body_vel_index = 6 * car_body_id_;
    double vx = data->cvel[body_vel_index];
    double vy = data->cvel[body_vel_index + 1];
    double speed_ms = sqrt(vx * vx + vy * vy);
    dashboard.speed_kmh = speed_ms * 3.6;

    // 限制速度范围
    if (dashboard.speed_kmh > max_speed_kmh_) dashboard.speed_kmh = max_speed_kmh_;
    if (dashboard.speed_kmh < 0) dashboard.speed_kmh = 0;

    // 获取油门输入
    double throttle = 0.0;
    if (motor_forward_id_ >= 0) {
```

```
    double ctrl = data->ctrl[motor_forward_id_];
    throttle = fabs(ctrl);
}

// 计算加速度
double acceleration = 0.0;
if (m_->opt.timestep > 0) {
    acceleration = (speed_ms - last_speed_ms_) / m_->opt.timestep;
}
last_speed_ms_ = speed_ms;

// 更新档位
UpdateGear(dashboard.speed_kmh, throttle, acceleration, last_rpm_);

// 计算转速
double gear_multiplier = gear_ratios_[1] / gear_ratios_[current_gear_];
double engine_rpm = 0.0;

if (current_gear_ > 0) {
    double min_speed_rpm = 50.0;
    engine_rpm = fmax(dashboard.speed_kmh, min_speed_rpm) * base_gear_ratio * gear_multiplier;
}

// 添加油门影响
double throttle_rpm = 0.0;
double total_rpm = 0.0;
const double idle_rpm = 600.0;

if (current_gear_ == 0) {
    if (throttle < 0.05) {
        total_rpm = 100.0;
    } else {
        throttle_rpm = throttle * 500.0;
        total_rpm = idle_rpm + throttle_rpm;
    }
} else {
    throttle_rpm = throttle * 400.0;
    total_rpm = idle_rpm + engine_rpm + throttle_rpm;
}

// 加速度影响
if (acceleration > 0.1) {
    total_rpm *= 1.0 + acceleration * 0.05;
} else if (acceleration < -0.2) {
    total_rpm *= 1.0 + acceleration * 0.02;
}

// 限制转速范围
if (total_rpm < 0) total_rpm = 0;
if (total_rpm > max_rpm_) total_rpm = max_rpm_;

// 平滑转速变化
double rpm_change = total_rpm - last_rpm_;
if (fabs(rpm_change) > 150) {
    total_rpm = last_rpm_ + copysign(150, rpm_change);
}
```

```
    dashboard.rpm = total_rpm;
    last_rpm_ = total_rpm;

    // 油量计算
    double fuel_consumption = 0.001
                            + (dashboard.rpm / max_rpm_) * 0.002
                            + (dashboard.speed_kmh / max_speed_kmh_) * 0.001;

    fuel_level_ -= fuel_consumption;
    if (fuel_level_ < 0) fuel_level_ = 0;
    dashboard.fuel = fuel_level_;

    // 温度计算
    double base_temp = 60.0;
    double rpm_temp = (dashboard.rpm / max_rpm_) * 40.0;
    double speed_temp = (dashboard.speed_kmh / max_speed_kmh_) * 10.0;

    dashboard.temperature = base_temp + rpm_temp + speed_temp;

    // 温度限制
    if (dashboard.temperature > 120.0) dashboard.temperature = 120.0;
    if (dashboard.temperature < 60.0) dashboard.temperature = 60.0;

    // 获取小车位置
    dashboard.position_x = data->xpos[3 * car_body_id_];
    dashboard.position_y = data->xpos[3 * car_body_id_ + 1];
    dashboard.position_z = data->xpos[3 * car_body_id_ + 2];
}
```

```
void UpdateGear(double speed_kmh, double throttle, double acceleration, double current_rpm) {
    // 简单的自动换挡逻辑
    if (current_gear_ == 0) {
        // 空挡：根据油门决定是否挂1档
        if (throttle > 0.3 && speed_kmh < 2) {
            current_gear_ = 1;
        }
        return;
    }

    // 调整换挡速度阈值以适应10 km/h的最大速度
    double speed_thresholds[] = {0, 1.5, 3.0, 4.5, 6.0, 8.0, 9.5};  // 各档位建议升档速度
    double min_speed_thresholds[] = {0, 0.5, 2.0, 3.5, 5.0, 7.0, 8.5};  // 各档位最低速度

    // 升档条件：高转速且不处于急加速状态
    if (current_gear_ < 6) {
        bool should_upshift = false;

        // 基于速度的升档
        if (speed_kmh > speed_thresholds[current_gear_]) {
            should_upshift = true;
        }

        // 高转速保护升档（调整为1200 max）
        if (current_rpm > 1100) {  // 从1300调整为1100
```

```
            should_upshift = true;
        }

        if (should_upshift && acceleration < 2.0) {  // 不处于急加速状态
            current_gear_++;
            // 升档时转速下降
            last_rpm_ *= 0.7;
        }
    }

    // 降档条件：低转速或需要急加速
    if (current_gear_ > 1) {
        bool should_downshift = false;

        // 基于速度的降档
        if (speed_kmh < min_speed_thresholds[current_gear_]) {
            should_downshift = true;
        }

        // 低转速降档
        if (current_rpm < 850) {  // 从900调整为850
            should_downshift = true;
        }

        // 急加速降档（Kickdown）
        if (throttle > 0.8 && acceleration < 1.0) {
            should_downshift = true;
        }

        if (should_downshift) {
            current_gear_--;
            // 降档时转速上升
            last_rpm_ *= 1.3;
        }
    }

    // 停车时自动回空挡
    if (speed_kmh < 0.5 && throttle < 0.1) {
        current_gear_ = 0;
        // 回空挡时降低转速
        last_rpm_ = 100.0;
    }
}
```

3.3 仪表盘渲染

3.3.1 速度表

速度表采用圆形仪表盘设计，范围0-10 km/h（根据实际车辆模型限制），包含背景圆环、刻度线、指针与中心圆点。

实现特点：

1. 刻度布局：主刻度每2 km/h，次刻度每1 km/h。
2. 颜色区域：安全区域（0-6 km/h）、警告区域（6-8 km/h）、危险区域（8-10 km/h）。
3. 数值显示：当前速度显示在表盘上方，单位km/h显示在速度值上方。

代码片段：

```
 // simulate.cc 中速度表渲染的关键部分
// 速度表参数设置
const float SPEED_MAX_DISPLAY = 10.0f;
const float SPEED_ANGLE_RANGE = 240.0f * M_PI / 180.0f;
const float SPEED_START_ANGLE = 210.0f * M_PI / 180.0f;
const float SPEED_END_ANGLE = SPEED_START_ANGLE - SPEED_ANGLE_RANGE;

// 速度颜色区域指示
float speed_safe_ratio = 6.0f / SPEED_MAX_DISPLAY;
float speed_safe_end_angle = SPEED_START_ANGLE - speed_safe_ratio * SPEED_ANGLE_RANGE;
glColor4f(0.0f, 0.6f, 0.0f, 0.2f);
DrawArc(speed_cx, speed_cy, speed_radius * 0.88f, speed_safe_end_angle, SPEED_START_ANGLE, 40);

float speed_warning_ratio = 8.0f / SPEED_MAX_DISPLAY;
float speed_warning_end_angle = SPEED_START_ANGLE - speed_warning_ratio * SPEED_ANGLE_RANGE;
glColor4f(1.0f, 1.0f, 0.0f, 0.2f);
DrawArc(speed_cx, speed_cy, speed_radius * 0.88f, speed_warning_end_angle, speed_safe_end_angle,
40);

float speed_danger_end_angle = SPEED_END_ANGLE;
glColor4f(1.0f, 0.0f, 0.0f, 0.2f);
DrawArc(speed_cx, speed_cy, speed_radius * 0.88f, speed_danger_end_angle, speed_warning_end_angle,
40);
```

3.3.2 转速表

转速表设计与速度表类似，但范围调整为0-2000 RPM（渲染显示范围），而实际物理仿真中的最大转速限制为1200 RPM。这种设计区分了显示范围与实际物理限制，确保了仪表的完整可视化效果。

实现特点：

1. 刻度布局：主刻度每500 RPM，次刻度每250 RPM。
2. 颜色区域：安全区域（0-1500 RPM）、警告区域（1500-1800 RPM）、危险区域（1800-2000 RPM）。
3. 档位显示：虽然没有直接显示当前档位，但转速计算已包含档位因素。

代码片段：

```
// simulate.cc 中转速表渲染的关键部分
const float RPM_MAX = 2000.0f;
```

```
const float RPM_ANGLE_RANGE = 240.0f * M_PI / 180.0f;
const float RPM_START_ANGLE = 210.0f * M_PI / 180.0f;
const float RPM_END_ANGLE = RPM_START_ANGLE - RPM_ANGLE_RANGE;

// 转速表指针
float rpm_ratio = data.rpm / RPM_MAX;
if (rpm_ratio > 1.0f) rpm_ratio = 1.0f;
float rpm_angle = RPM_START_ANGLE - rpm_ratio * RPM_ANGLE_RANGE;

// 转速颜色区域指示
float rpm_safe_ratio = 1500.0f / RPM_MAX;
float rpm_safe_end_angle = RPM_START_ANGLE - rpm_safe_ratio * RPM_ANGLE_RANGE;
glColor4f(0.0f, 0.6f, 0.0f, 0.2f);
DrawArc(rpm_cx, rpm_cy, rpm_radius * 0.88f, rpm_safe_end_angle, RPM_START_ANGLE, 40);

float rpm_warning_ratio = 1800.0f / RPM_MAX;
float rpm_warning_end_angle = RPM_START_ANGLE - rpm_warning_ratio * RPM_ANGLE_RANGE;
glColor4f(1.0f, 1.0f, 0.0f, 0.2f);
DrawArc(rpm_cx, rpm_cy, rpm_radius * 0.88f, rpm_warning_end_angle, rpm_safe_end_angle, 40);

float rpm_danger_end_angle = RPM_END_ANGLE;
glColor4f(1.0f, 0.0f, 0.0f, 0.2f);
DrawArc(rpm_cx, rpm_cy, rpm_radius * 0.88f, rpm_danger_end_angle, rpm_warning_end_angle, 40);
```

3.3.3 仪表盘位置

```
// simulate.cc 中的仪表盘位置设置
float speed_cx = 100;                       // 速度表中心X坐标
float speed_cy = rect.height - 150;         // 速度表中心Y坐标（距离顶部150像素）
float speed_radius = 50;                    // 速度表半径

float rpm_cx = 250;                         // 转速表中心X坐标（距离速度表150像素）
float rpm_cy = rect.height - 150;           // 转速表中心Y坐标（与速度表对齐）
float rpm_radius = 50;                      // 转速表半径

float fuel_x = 50;                          // 油量表X坐标
float fuel_y = rect.height - 250;           // 油量表Y坐标（在仪表下方）
float fuel_width = 100;                     // 油量表宽度
float fuel_height = 20;                     // 油量表高度

float temp_x = 200;                         // 温度表X坐标
float temp_y = rect.height - 250;           // 温度表Y坐标（与油量表对齐）
float temp_width = 100;                     // 温度表宽度
float temp_height = 20;                     // 温度表高度
```

四、遇到的问题和解决方案

问题1:环境配置与编译失败

- **现象**: 在Ubuntu系统上执行cmake --build . -j4时出现链接错误，提示未定义引用mj_step等MuJoCo函数。
- **原因**: CMake未能正确找到MuJoCo库的路径，导致链接阶段失败。
- **解决**:

1. 检查CMakeLists.txt文件，确认已包含find_package(MuJoCo REQUIRED)。
2. 手动指定MuJoCo库路径：set(MUJOCO_DIR "/path/to/mujoco")。
3. 清理build目录后重新配置编译：rm -rf build && mkdir build && cd build && cmake .. && make -j4。
4. 最终确认是系统中存在多个MuJoCo版本导致冲突，卸载旧版本后问题解决。

问题2:仪表盘渲染时出现闪烁或覆盖3D场景

- **现象**:仪表盘绘制完成后，部分3D场景元素出现在仪表盘上方，或仪表盘内容每帧闪烁
- **原因**:OpenGL状态管理不当，特别是深度测试与混合模式设置顺序错误。
- **解决**:

1. 确保在绘制2D仪表盘前正确保存OpenGL状态（使用glPushAttrib与glPushMatrix）。
2. 绘制仪表盘前禁用深度测试（glDisable(GL_DEPTH_TEST)），确保2D内容始终在最上层。
3. 启用混合（glEnable(GL_BLEND)）并设置正确的混合函数（glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)）以实现透明效果。
4. 渲染完成后严格恢复OpenGL状态。

问题3:数据更新与渲染不同步

- **现象**:仪表盘指针更新滞后于车辆实际运动，或出现跳变。
- **原因**:数据提取与渲染在同一线程中进行，但未考虑仿真步进与渲染帧率之间的同步问题。
- **解决**:

1. 将数据提取移到仿真步进之后、渲染之前，确保每帧使用最新的仿真数据。
2. 对速度、转速等数据进行低通滤波处理，平滑输出值，避免因仿真数值波动导致的指针抖动。
3. 在DashboardData结构体中添加时间戳，并在渲染时检查数据新鲜度，避免使用陈旧数据。

五、测试与结果

5.1 功能测试

测试项目	测试方法	预期结果	实际结果	状态
环境编译	执行编译命令，检查输出	编译成功，无错误	编译成功，生成mjpc可执行文件	✅
场景加载	加载自定义车辆场景	3D窗口显示车辆模型	车辆模型正常显示，可旋转视角	✅
数据获取	控制车辆移动，查看控制台输出	速度、位置数据随运动变化	速度范围0-10 km/h，数据实时更新	✅
速度表渲染	加速/减速车辆	指针随速度平滑移动	指针在0-10 km/h范围内响应正确	✅
转速表渲染	改变车速和油门	转速随车速和油门变化	转速在0-1200 RPM范围内变化，包含自动换挡逻辑	✅
油量温度显示	长时间运行	油量递减，温度随转速升高	面板数据更新正常，模拟逻辑正确	✅
档位模拟	观察转速和车速关系	自动换挡逻辑生效	包含空挡和6档的自动变速器逻辑正常工作	✅

5.2 性能测试

- 帧率测试：在RTX 4060显卡上，运行包含车辆、地面、光源的完整场景，仪表盘全功能开启，平均帧率为105 FPS，满足实时性要求。
- CPU占用：仿真与渲染线程合计占用约15%的CPU资源（8核心）。
- 内存占用：程序运行期间内存占用稳定在约850 MB，无内存泄漏（通过Valgrind验证）。

5.3 效果展示

- 视频链接：https://github.com/167648384/-MuJoCo-MPC-/tree/main/videos

六、总结与展望

6.1 学习收获

通过完成本次大作业，我获得了以下宝贵的经验与技能：

1. 大型开源项目二次开发能力：学会了如何阅读、理解并修改工业级C++代码库，掌握了在现有框架中集成新功能的方法。
2. 跨学科知识整合：将物理仿真（MuJoCo）、控制理论（MPC）、计算机图形学（OpenGL）与软件工程（C++、CMake、

Git）有机结合，解决复杂工程问题。

3. 实时系统开发经验：深入理解了实时数据流处理、多线程同步、性能优化等关键概念，并实践了OpenGL渲染管线的使用与调试。
4. 问题解决能力：面对环境配置、编译错误、渲染异常、性能瓶颈等挑战，通过查阅文档、调试工具、社区求助等多种方式逐一攻克，提升了独立解决问题的能力。
5. 工程规范意识：按照模块化设计、代码注释、版本控制、文档撰写等工程规范进行开发，培养了良好的编程习惯。

6.2 不足之处

1. 物理模型简化：车辆动力学模型较为简单，未考虑悬挂系统、轮胎滑移、空气阻力等复杂因素，与真实车辆行为存在差距。
2. MPC集成度待加强：虽然集成了MPC框架的任务系统（残差函数和状态转移），但仪表盘数据主要来自物理仿真状态而非MPC控制器的规划输出。未来的MPC规划轨迹、预测状态等高级信息尚未在仪表盘中展示。
3. UI交互有限：仪表盘为纯显示界面，缺乏用户交互功能（如点击按钮切换显示模式）。
4. 代码优化空间：部分渲染代码仍可进一步优化，如使用现代OpenGL（3.0+）的着色器管道替代固定功能管道，提升性能与灵活性。

6.3 未来改进方向

1. 增强物理真实性：导入更精细的车辆模型，集成车辆动力学库（如CarSim、veDYNA），提升仿真逼真度。
2. 集成完整MPC控制：实现基于MPC的自动驾驶控制器，使仪表盘能够显示控制器的规划轨迹、预测状态、优化目标等信息。
3. 扩展交互功能：添加可交互的UI控件（按钮、滑块、菜单），允许用户实时调整仿真参数、切换摄像机视角、保存/加载场景等。
4. 支持多车辆与交通环境：扩展场景至多车辆交互，添加交通规则、信号灯、行人等元素，构建更丰富的驾驶仿真环境。
5. 云渲染与远程可视化：将渲染部分分离为独立服务，支持通过Web浏览器远程查看与控制仿真，便于演示与协作。

七、参考资料

1. MuJoCo官方文档. https://mujoco.readthedocs.io/
2. MuJoCo MPC GitHub仓库. https://github.com/google-deepmind/mujoco_mpc
3. LearnOpenGL CN教程. https://learnopengl-cn.github.io/
4. GLFW官方指南. https://www.glfw.org/docs/latest/
5. 《C++ Primer（第5版）》. Stanley B. Lippman, Josée Lajoie, Barbara E. Moo.