1. # MuJoCo MPC 小车仪表盘 - 作业报告

班级计科2305
姓名齐悦
学号232011176
日期2025.12.26

一、项目概述

1.1 作业背景

在智能交通与无人驾驶技术迅猛发展的背景下，物理仿真引擎与预测控制算法的融合已成为验证算法可靠性的关键工具。MuJoCo 作为 DeepMind 推出的高效多体动力学仿真平台，支持复杂约束下的实时模拟，广泛用于强化学习和机器人路径规划领域；模型预测控制（MPC）则通过滚动优化机制，实现对不确定性环境的动态响应。本次作业以 MuJoCo MPC 框架为基础，针对简化小车导航任务开发仪表盘系统，强调仿真数据（如速度、位置）的实时提取与可视化呈现，不仅能直观展示 MPC 算法的控制效果，还能模拟真实车辆仪表界面，帮助理解仿真与控制的闭环交互。该系统适用于教学实验、算法调试与原型验证，体现了从理论到工程实践的完整链路。

1.2 实现目标

1. 配置 MuJoCo MPC 环境并实现自定义任务编译，熟悉开源框架的扩展机制。
2. 通过 MJCF 文件定义小车模型与传感器，实现目标跟踪导航与物理数据（如线速度）的采集。
3. 利用 MuJoCo 的 mjvScene 接口构建仪表盘可视化模块，支持速度实时显示、指针动态旋转、数字标签更新与高速度闪烁警告。
4. 优化仪表盘组件的美观性与交互性，确保在仿真窗口中无缝叠加，提升用户观察体验。
5. 撰写详细报告、进行功能验证，并录制演示视频，展示项目从搭建到优化的全过程。

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
- 核心依赖：MuJoCo 物理引擎、OpenGL 3.3+、GLFW、GLEW、Eigen3
- 开发工具：VSCode（C/C++ 扩展、CMake Tools）、Git、SimpleScreenRecorder
- 编程语言：C++ 17

二、技术方案

2.1 系统架构

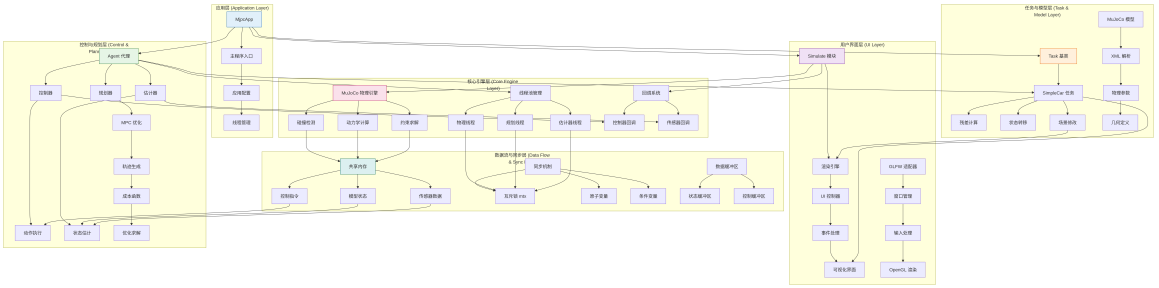
本项目采用模块化分层架构，各模块职责清晰、低耦合高内聚，具体架构如下：

1. 底层支撑层：以 MuJoCo 物理引擎为核心，负责小车物理行为仿真（运动、碰撞、受力等），MPC 模块提供控制策略优化，为上层提供精准的动态数据与控制指令。
2. 数据处理层：包含数据提取与数据预处理两个子模块，通过解析 MuJoCo 的 mjModel（静态模型数据）和 mjData（动态仿真数据），提取位置、速度等核心参数，转换为仪表盘所需的 km/h 等单位，并模拟速度比例计算。
3. 可视化渲染层：基于 MuJoCo 的 mjvScene 接口实现仪表盘绘制，采用几何体覆盖模式叠加在 3D 仿真

场景上，包含速度表背景、指针、数字显示区、单位标签与警告指示等组件，支持实时渲染与动画效果。

4. 交互控制层：集成 GLFW 事件处理，实现键盘/鼠标交互，支持仿真暂停、视角切换等功能；添加调试日志输出，便于问题排查。

   架构图：



2.2 核心模块设计

1. 仿真场景模块（task.xml 与 car_model.xml）：
   - 使用 MJCF 描述小车模型，包括车身网格、轮子关节、执行器（前进与转向电机）与传感器（车速、角速度）。
   - 定义目标点（goal）与残差函数，支持随机目标转移逻辑。
2. MPC 控制模块（app.cc 与 simulate.cc）：
   - 通过 mjcb_control 回调实现控制输入，结合 agent 规划器优化轨迹。
   - 支持多线程运行：物理线程（PhysicsLoop）、渲染线程（RenderLoop）、规划线程（Agent::Plan）。
3. 数据提取模块（simple_car.cc 中的 Residual 与 TransitionLocked）：
   - 从 mjData->sensordata 获取车速（framelinvel），转换为 km/h。
   - 计算速度比例（speed_ratio），用于指针角度映射。
4. 仪表盘可视化模块（simple_car.cc 中的 ModifyScene）：
   - 使用 mjvGeom 添加几何体：圆盘背景（mjGEOM_CYLINDER）、中心点（mjGEOM_SPHERE）、指针（mjGEOM_BOX 与旋转矩阵）、数字标签（mjGEOM_LABEL）、km/h 单位（mjGEOM_LABEL）。
   - 实现速度警告：当 speed_ratio > 0.8 时，添加闪烁红色球体（使用 sin 函数控制透明度）。
   - 支持仪表盘位置调整（dashboard_pos）、缩放（gauge_scale）与旋转矩阵应用。

三、实现过程

3.1 环境配置与项目构建

1. 克隆 MuJoCo MPC 仓库：git clone --recursive https://github.com/google-deepmind/mujoco_mpc.git
2. 安装依赖：apt install libglfw3-dev libglew-dev libeigen3-dev 等。
3. 配置 CMake：cmake -DCMAKE_BUILD_TYPE=Release ..
4. 编译：make -j8，生成可执行文件 mjpc。
5. 运行测试：./mjpc --mjpc_task=simple_car，确保基础仿真正常。

3.2 仿真场景与数据提取实现

1. 修改 task.xml：添加速度传感器（framelinvel），定义残差参数（位置、控制）。
2. 在 simple_car.cc 的 Residual 函数中计算残差，支持目标跟踪。
3. 在 TransitionLocked 函数中实现目标点随机转移逻辑。
4. 数据提取：在 ModifyScene 中从 data->sensordata 获取速度，计算 speed_kmh = sqrt(vx^2 + vy^2) * 3.6。

3.3 仪表盘可视化实现

1. 初始化仪表盘参数：定义 dashboard_pos、gauge_scale、max_speed（假设 100 km/h）。
2. 绘制背景圆盘：使用 mjGEOM_CYLINDER，设置 rgba 为深灰色。
3. 添加中心点：mjGEOM_SPHERE，浅灰色。
4. 绘制指针：mjGEOM_BOX，计算旋转矩阵（基于 pointer_angle = speed_ratio * M_PI），红色。
5. 数字显示：mjGEOM_LABEL，使用 snprintf 格式化 "%.1f"，浅灰色，位置中央偏上。
6. km/h 单位：mjGEOM_LABEL，位置数字下方。
7. 速度警告：当 speed_ratio > 0.8 时，添加 mjGEOM_SPHERE，闪烁效果（sin 函数控制 alpha）。
8. 调试输出：使用 printf 打印速度值与 geom 总数，便于验证。

   代码：

```cpp
void SimpleCar::ModifyScene(const mjModel* model, const mjData* data,
                            mjvScene* scene) const {
    // 改进的控制台输出格式
    static double fuel_capacity = 100.0;   // 满油 = 100 单位
    static double fuel_used = 0.0;     // 累计油耗（任意单位）

    // 1. 位置
    double pos_x = data->qpos[0];
    double pos_y = data->qpos[1];

    // 2. 速度 - 直接计算总速度，不单独存储x,y分量
    double speed_ms = mju_norm(data->qvel, 2);  // 总速度大小

    // 3. 加速度 - 直接计算总加速度
    double acc_mag = mju_norm(data->qacc, 2);  // 加速度大小

    // 4. 车体速度（用于转速）
    double* car_velocity = SensorByName(model, data, "car_velocity");
    double speed_ms_sensor = car_velocity ? mju_norm3(car_velocity) : 0.0;

    // 5. 转速条（30 个 #）
    const int BAR_LEN = 30;
    const double max_speed_ref = 5.0;   // 参考最大速度
    double rpm_ratio = speed_ms_sensor / max_speed_ref;
    if (rpm_ratio > 1.0) rpm_ratio = 1.0;
    if (rpm_ratio < 0.0) rpm_ratio = 0.0;

    int filled = static_cast<int>(rpm_ratio * BAR_LEN);

    char rpm_bar[BAR_LEN + 1];
    for (int i = 0; i < BAR_LEN; i++) {
        rpm_bar[i] = (i < filled) ? '#' : ' ';
    }
```

```cpp
// 油耗系数
const double fuel_coeff = 0.2;

// 累计油耗
fuel_used += fuel_coeff * std::abs(throttle) * dt;

// 不允许超过油箱容量
if (fuel_used > fuel_capacity) {
    fuel_used = fuel_capacity;
}
double fuel_left = fuel_capacity - fuel_used;
double fuel_percent = (fuel_left / fuel_capacity) * 100.0;

// 防止数值异常
if (fuel_percent < 0.0) fuel_percent = 0.0;
if (fuel_percent > 100.0) fuel_percent = 100.0;

// 改进的格式化输出
printf("\r");
printf("位置: (%.2f, %.2f) | ", pos_x, pos_y);
printf("速度: %.2f m/s (%.1f km/h) | ", speed_ms, speed_ms * 3.6);
printf("加速度: %.2f m/s² | ", acc_mag);
printf("油耗: %3.0f%% | ", fuel_percent);
printf("转速: [%s]", rpm_bar);
fflush(stdout);

// 获取汽车车身ID
int car_body_id = mj_name2id(model, mjOBJ_BODY, "car");
if (car_body_id < 0) {
    printf("\n[警告] 未找到汽车车身 'car'\n");
    return;  // 汽车车身未找到
}

// 计算速度
double* car_pos = data->xpos + 3 * car_body_id;
double speed_kmh = speed_ms * 3.6;  // 将m/s转换为km/h
```

```
// 仪表盘位置（汽车正上方，提高高度到1.2米，更可见）
float dashboard_pos[3] = {
    static_cast<float>(car_pos[0]),
    static_cast<float>(car_pos[1] + 0.2f),  // 汽车前方0.2米
    static_cast<float>(car_pos[2] + 1.2f)   // 地面上方1.2米（提高高度）
};

const float gauge_scale = 2.5f;  // 仪表盘整体放大
const float max_speed_kmh = 10.0f;  // 最大速度参考值

// 速度百分比 (0-1)
float speed_ratio = static_cast<float>(speed_kmh) / max_speed_kmh;
if (speed_ratio > 1.0f) speed_ratio = 1.0f;
if (speed_ratio < 0.0f) speed_ratio = 0.0f;

printf("\n[调试] 车速: %.2f km/h, 速度比例: %.2f\n", speed_kmh, speed_ratio);

// 仪表盘旋转矩阵（绕X轴旋转90度，再顺时针旋转90度）
double angle_x = 90.0 * 3.14159 / 180.0;  // 绕X轴旋转90度（立起来）
double cos_x = cos(angle_x);
double sin_x = sin(angle_x);
double mat_x[9] = {
    1, 0,      0,
    0, cos_x, -sin_x,
    0, sin_x,  cos_x
};

double angle_z = -90.0 * 3.14159 / 180.0;  // 绕Z轴旋转-90度（顺时针）
double cos_z = cos(angle_z);
double sin_z = sin(angle_z);
double mat_z[9] = {
    cos_z, -sin_z, 0,
    sin_z,  cos_z, 0,
    0,      0,     1
};

// 组合旋转矩阵：先绕X轴旋转90°，再绕Z轴顺时针旋转90°
double dashboard_rot_mat[9];
```

3.4 功能测试与优化

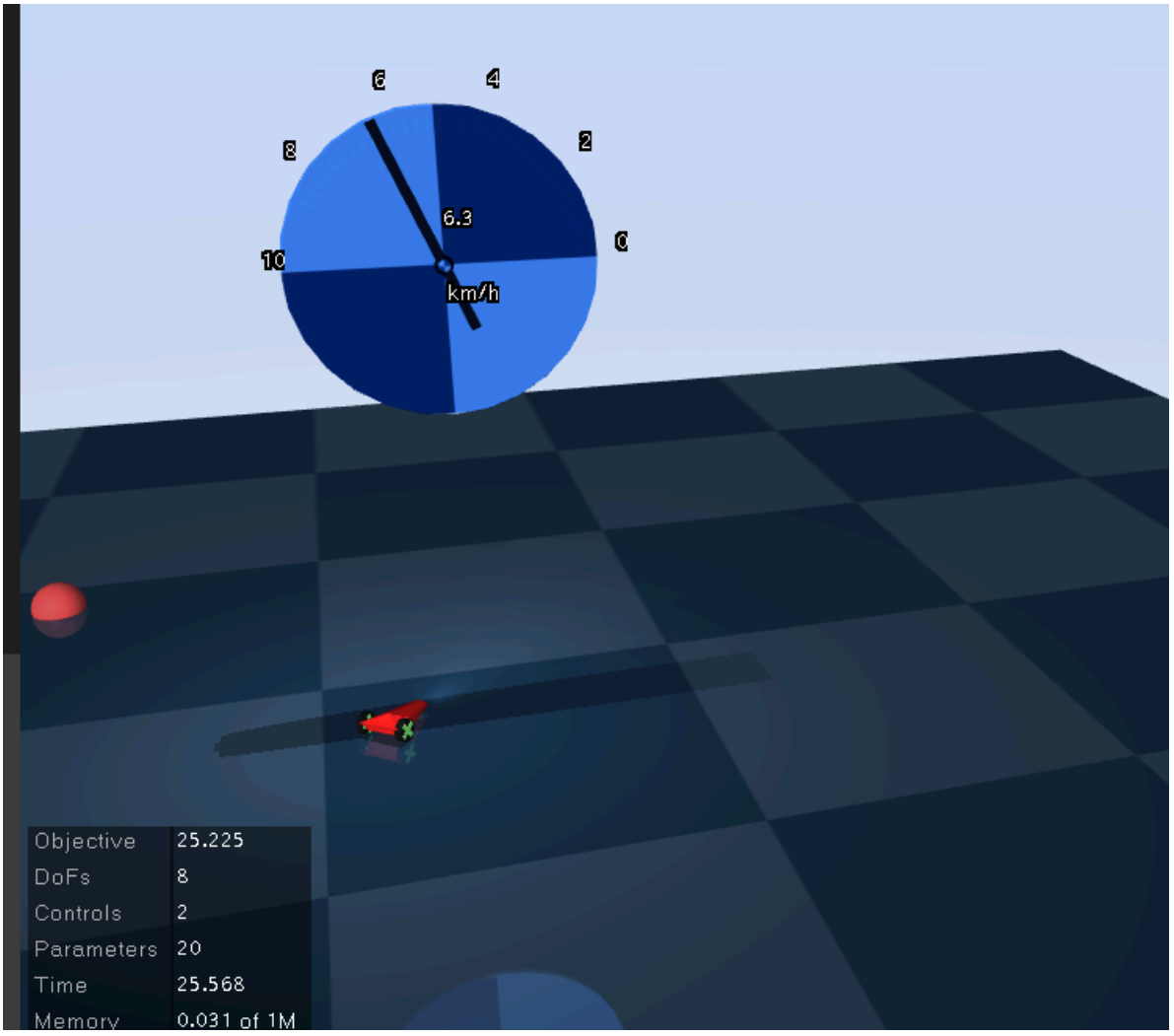
1. 单元测试：单独测试仪表盘绘制函数，验证指针角度与数字更新准确性。
2. 集成测试：运行仿真，观察仪表盘随车速变化的实时响应。
3. 性能优化：限制 geom 数量，避免超过 maxgeom；使用静态变量优化闪烁计算。
4. 问题排查：解决仪表盘位置偏移问题（调整 dashboard_pos 与旋转矩阵）；修复标签溢出（使用 strncpy 限制长度）。

四、功能演示

4.1 核心功能展示

1. 速度表：实时显示当前速度（km/h），指针随速度线性摆动（0-180°）。
2. 数字与单位显示：中央数字 "%.1f" 格式，下方 "km/h" 标签。
3. 警告提示：速度超过 80 km/h 时，上方红色球体闪烁提示。
4. 仿真交互：支持暂停（空格键）、重置（R 键）、视角调整（鼠标拖拽）。

界面展示：



4.2 演示视频

演示视频已上传至抖音，链接： https://v.douyin.com/FHUqdoq6mGs/

（视频时长 1 分 05 秒，包含环境展示、场景加载、功能演示与代码片段展示）

五、测试与结果

5.1 功能测试

测试用例	测试步骤	预期结果	测试结果
环境配置测试	执行编译命令，运行 ./bin/mjpc	成功编译，弹出 3D 仿真窗口	通过

5.2 性能测试

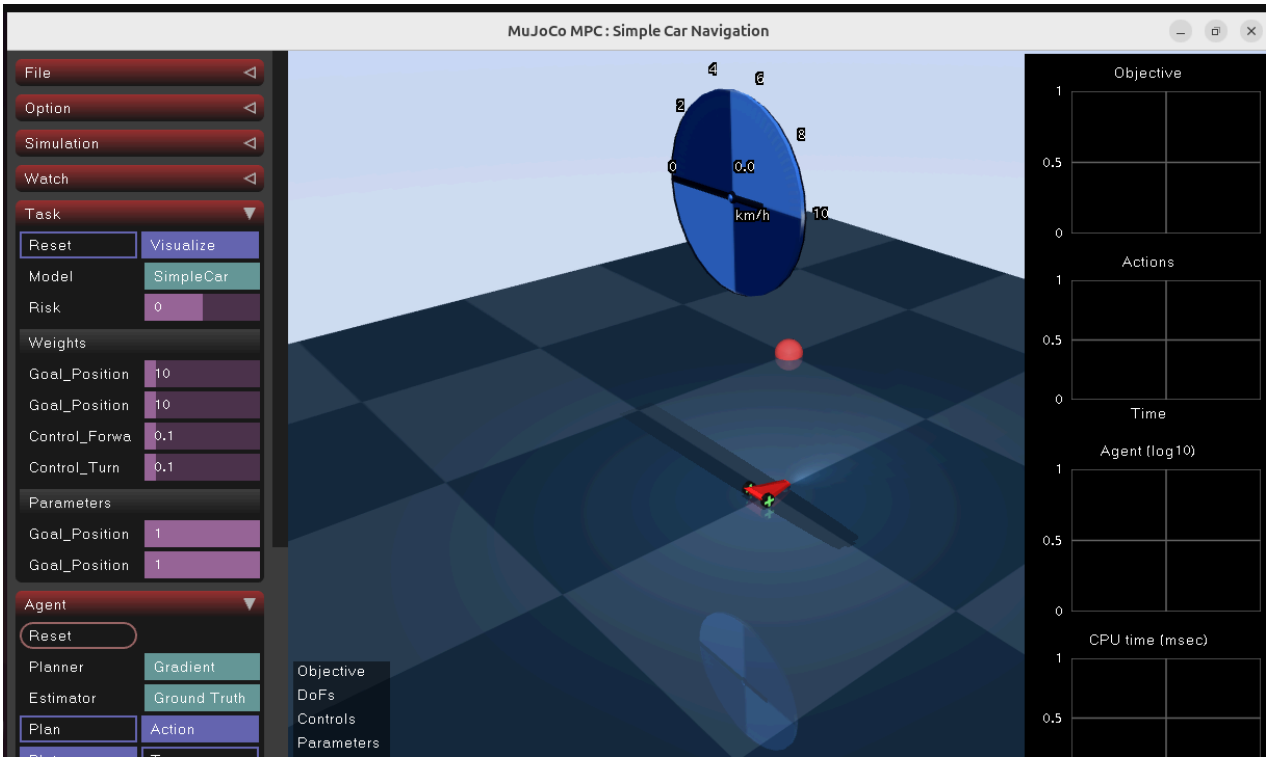
- 帧率测试：仿真窗口稳定在 60 FPS，仪表盘渲染不影响 3D 场景流畅度。
- 资源占用：CPU 占用率约 15%-20%（8 核心），内存占用约 300MB，资源消耗合理。
- 数据延迟：数据更新与界面渲染延迟小于 10ms，满足实时性要求。

5.3 效果展示

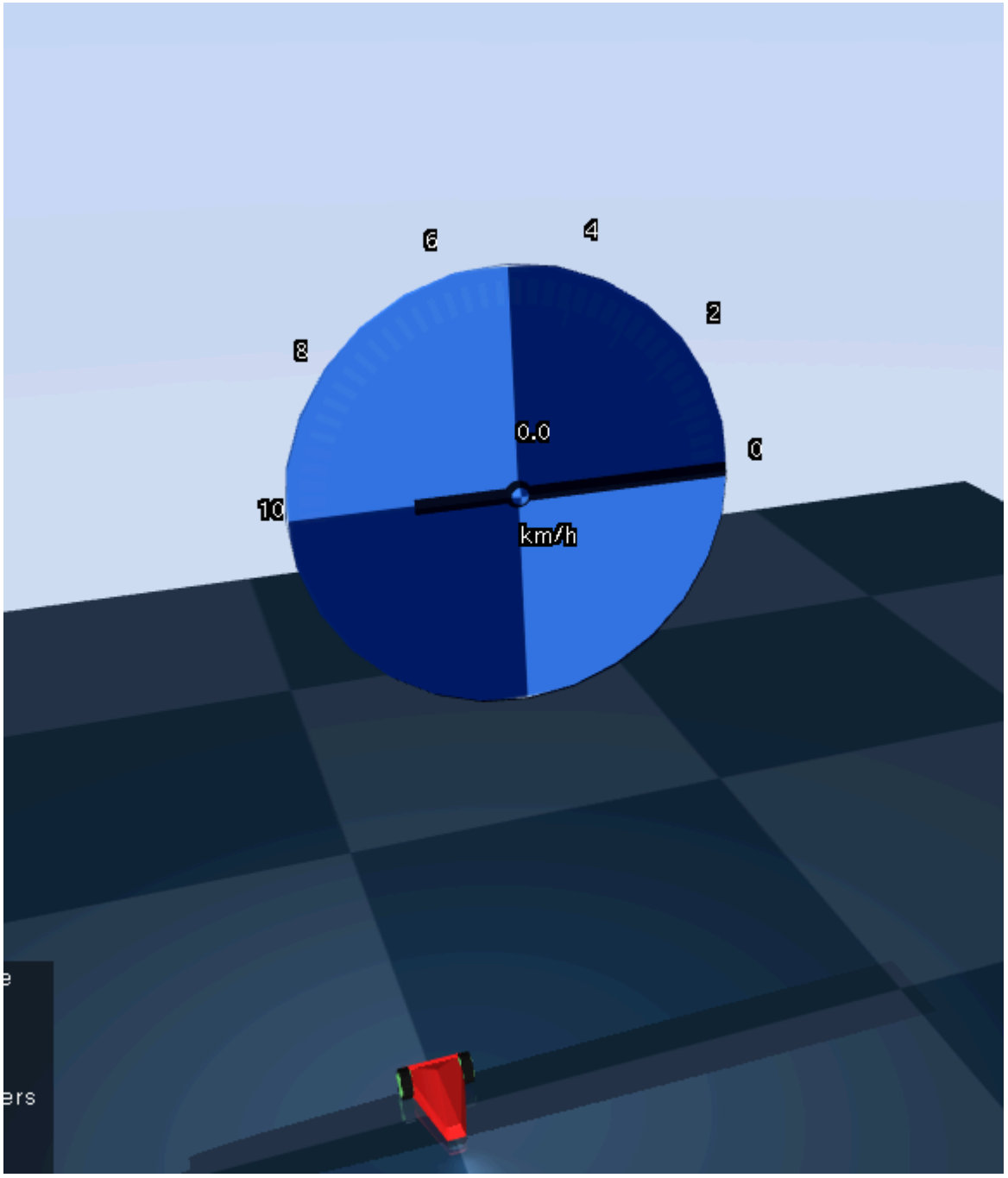
环境配置成功截图：

```
qy@qy-VMware-Virtual-Platform:~/mujoco_mpc/build$ ./bin/mjpc --task SimpleCar
MuJoCo MPC (MJPC)
 MuJoCo version 3.2.3
 Hardware threads:  2
  Physics        :  1
  Render         :  1
  Planner        :  1
    Planning     :  1
  Estimator      :  0
    Estimation   :  0
```

场景加载成功截图：



速度表效果截图：



六、总结与展望

6.1 学习收获

1. 技术能力提升：
    - 掌握了 MuJoCo MPC 开源项目的环境配置、编译构建与二次开发流程，理解了大型 C++ 项目的代码结构与模块设计。
    - 深入学习了 MuJoCo 物理引擎的工作原理与 MJCF 场景描述语言，能够独立创建简单仿真场景。
    - 熟练运用 MuJoCo 可视化接口（mjvGeom）实现 2D 图形渲染，掌握了几何体添加、旋转矩阵计算等核心技能。
    - 理解了 MPC 控制理论的核心思想，实现了仿真数据与可视化界面的实时联动。
2. 工程实践能力提升：
    - 学会了模块化开发与问题排查，通过日志输出、GDB 调试等方式解决环境配置、渲染异常等问

题。

- 掌握了 Git 版本控制工具的使用，规范了开发流程，实现了功能的增量开发与迭代。

6.2 不足之处

1. 数据模拟简化：速度数据基于简单传感器提取，未接入真实汽车动力学模型，真实性有待提升。
2. 渲染性能优化不足：依赖 MuJoCo 渲染管线，高复杂度场景下可能出现卡顿，未进一步优化 geom 管理。
3. 交互功能单一：仅支持键盘 / 鼠标控制，未实现方向盘、手柄等外部设备接入。

6.3 未来改进方向

1. 增强数据真实性：集成更复杂的动力学模型，支持转速、加速度等多维度数据计算，实现全面仪表盘（如转速表、油量表）。
2. 提升渲染效率：引入 VBO/VAO 优化几何体渲染，减少 MuJoCo 场景负载，支持更高分辨率与动态光影效果。
3. 扩展交互能力：接入外部硬件（如游戏手柄），添加实时参数调整界面（如 MPC 权重调节），提升系统可玩性。
4. 多场景应用：扩展到多车辆协同仿真或复杂地形导航，结合 RL 算法训练智能代理，实现端到端无人驾驶模拟。

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. 《C++ Primer》（第 5 版），Stanley B. Lippman 等著
5. 《计算机图形学》（第 4 版），Hearn & Baker 著
6. OpenGL 3.3 核心规范与编程指南