

一、项目概述

1.1 项目背景

本项目基于 MuJoCo 物理仿真引擎和 mjpc (MuJoCo 预测控制) 框架，实现了一个具有完整仪表盘显示的汽车仿真系统。通过该系统，能够直观地展示车辆的实时状态信息，包括速度、位置、加速度以及燃油消耗等数据。

1.2 实现目标

本项目的核心目标包括：

- 在 MuJoCo 中构建一个简化的 2D 汽车模型
- 实现实时数据采集与处理
- 开发 3D 仪表盘渲染系统
- 实现终端状态监控界面
- 集成燃料消耗模型
- 确保系统稳定性和实时性

1.3 开发环境

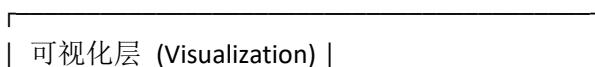
- 操作系统: Windows 11 + WSL2 (Ubuntu 22.04)
- 仿真引擎: MuJoCo
- 控制框架: mjpc (MuJoCo MPC)
- 编程语言: C++17
- 构建工具: CMake 3.16+
- 图形渲染: OpenGL (通过 MuJoCo 接口)
- 编译器: GCC 11.4.0 (开启-Werror 严格模式)

二、系统架构设计

2.1 整体架构

系统采用模块化设计，主要包含以下四个层次：

text



| 可视化层 (Visualization) |



2. 2 主要模块功能

车辆模型模块(*simplecar/task.xml*)

- 定义汽车几何结构和物理属性
- 配置传感器和控制器

数据采集模块

- 从 mjData 结构体中提取实时状态
- 计算速度、加速度等衍生数据

仪表盘渲染模块

- 使用 MuJoCo 的 mjvGeom 接口渲染 3D 仪表
- 实现速度表、刻度、指针等组件

终端显示模块

- 提供实时文本状态输出
- 实现原地刷新功能避免刷屏

燃料系统模块

- 模拟燃料消耗模型
- 计算并显示剩余燃料百分比

三、关键技术实现

3. 1 车辆状态获取

```
cpp
// 从 MuJoCo 数据中获取车辆状态
double pos_x = data->qpos[0];
double pos_y = data->qpos[1];
double vel_x = data->qvel[0];
double vel_y = data->qvel[1];
double acc_x = data->qacc[0];
double acc_y = data->qacc[1];
// 获取传感器数据
double* car_velocity = SensorByName(model, data, "car_velocity");
double speed_ms = car_velocity ? mju_norm3(car_velocity) : 0.0;
double speed_kmh = speed_ms * 3.6;
```

3. 2 仪表盘渲染实现

仪表盘采用组合式渲染策略，包含多个几何组件：

```
cpp
// 1. 仪表盘底座 (圆柱体)
geom->type = mjGEOM_CYLINDER;
geom->size[0] = geom->size[1] = 0.15f * gauge_scale;
geom->size[2] = 0.005f * gauge_scale;
// 2. 刻度弧线 (多段长方体)
for(ints = 0; s < ARCSEG; s++) {mjvGeom* g = scene->geoms + scene->ngeom;
mjjv_initGeom(g, mjGEOM_BOX, size, pos, mat9, rgba);
}
// 3. 刻度线和标签
for (int i = 0; i < kTickCount; i++) {
// 刻度线
geom->type = mjGEOM_BOX;
// 标签
label_geom->type = mjGEOM_LABEL;
std::snprintf(label_geom->label, sizeof(label_geom->label), "%d", tick_value);
}
// 4. 指针
```

```

geom->type = mjGEOM_BOX;
float angle = 180.0f - 180.0f * speed_ratio; // 根据速度比例计算角度
// 5. 中心点 (球体)
geom->type = mjGEOM_SPHERE;
// 6. 速度数值标签
geom->type = mjGEOM_LABEL;
std::snprintf(speed_label, sizeof(speed_label), "%1f", speed_kmh);
// 7. 单位标签
geom->type = mjGEOM_LABEL;
std::strncpy(geom->label, "km/h", sizeof(geom->label) - 1);

```

3. 3 终端状态显示

```

cpp
// 格式化输出字符串
printf(
"\rPos(%2f, %2f) | "
"Vel(%2f, %2f) | "
"Acc(%2f, %2f) | "
"Fuel %3.0f%%"
"RPM [%s",
pos_x, pos_y,
vel_x, vel_y,
acc_x, acc_y,
fuel_percent,
rpm_bar);
fflush(stdout); // 强制刷新输出缓冲区
关键技术点:

```

- 使用\r 回车符实现原地刷新
- fflush(stdout)确保实时显示
- 格式化字符串提供清晰的数据展示

3. 4 燃料消耗模型

```

cpp
// 静态变量记录燃料状态
static double fuel_capacity = 100.0;
static double fuel_used = 0.0;
// 基于油门输入计算燃料消耗
double throttle = data->ctrl[0];
const double fuel_coeff = 0.2;
fuel_used += fuel_coeff * std::abs(throttle) * dt;

```

```

// 计算剩余燃料百分比
double fuel_left = fuel_capacity - fuel_used;
double fuel_percent = (fuel_left/fuel_capacity) * 100.0;
// 边界检查
if(fuel_percent < 0.0)fuel_percent = 0.0;
if(fuel_percent > 100.0)fuel_percent = 100.0;

```

3. 5 RPM 显示条

```

cpp
// 创建 RPM 显示条
const int BAR_LEN = 30;
const double max_speed_ref = 5.0;
double rpm_ratio = speed_ms / max_speed_ref;
// 边界检查
if(rpm_ratio > 1.0)rpm_ratio = 1.0;
if(rpm_ratio < 0.0)rpm_ratio = 0.0;
// 计算填充长度
int filled = static_cast(rpm_ratio * BAR_LEN);
// 生成 RPM 条字符串
char rpm_bar[BAR_LEN + 1];
for(int i = 0; i < BAR_LEN; i ++){rpm_bar[i] = (i < filled) ? '#' : ' ';}
rpm_bar[BAR_LEN] = '\0';

```

四、坐标系与变换

4. 1 仪表盘位置计算

```

cpp
// 获取汽车位置
int car_body_id = mj_name2id(model, mjOBJ_BODY, "car");
double* car_pos = data->xpos + 3 * car_body_id;
// 仪表盘位置（汽车上方）
float dashboard_pos[3] = {
    static_cast(car_pos[0]),
    static_cast(car_pos[1]),
    static_cast(car_pos[2] + 0.3f)//Z轴偏移
};

```

4.2 旋转变换矩阵

```
cpp
// X 轴旋转 90 度
double angle_x = 90.0 * M_PI / 180.0;
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};
// Z 轴旋转-90 度
double angle_z = -90.0 * M_PI / 180.0;
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
double dashboard_rot_mat[9];
for int i = 0; i < 3; i ++ {
    for int j = 0; j < 3; j ++ {
        dashboard_rot_mat[i3 + j] = 0;
        for int k = 0; k < 3; k ++ {
            dashboard_rot_mat[i3 + j] += mat_z[i3 + k] * mat_x[k3 + j];
        }
    }
}
```

五、控制逻辑实现

5.1 残差函数

```
cpp
void SimpleCar::ResidualFn::Residual(const mjModel* model, const mjData* data,
double* residual) const {
    // 位置跟踪误差
    residual[0] = data->qpos[0] - data->mocap_pos[0];
    residual[1] = data->qpos[1] - data->mocap_pos[1];
    // 控制输入正则化
}
```

```
residual[2] = data->ctrl[0]; // 油门  
residual[3] = data->ctrl[1]; // 转向  
}
```

5.2 目标点重置逻辑

```
cpp  
void SimpleCar::TransitionLocked(mjModel* model, mjData* data) {  
    // 计算汽车到目标点的距离  
    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);  
    // 当距离小于阈值时重置目标点  
    if(mju_norm(car_to_goal, 2) < 0.2){absl::BitGen gen_;  
        data->mocap_pos[0] = absl::Uniform(gen, - 2.0, 2.0);  
        data->mocap_pos[1] = absl::Uniform(gen, - 2.0, 2.0);  
        data->mocap_pos[2] = 0.01;  
    }  
}
```

六、性能优化与调试

6.1 性能优化措施

- 几何体数量控制
- 限制渲染的几何体数量不超过 `scene->maxgeom`
- 使用条件检查避免数组越界
- 计算效率优化
- 预计算旋转矩阵
- 重用计算结果减少重复计算
- 内存管理
- 使用静态变量减少内存分配
- 合理使用局部变量

6.2 调试技巧

调试输出

```
cpp
// 添加调试信息
printf("DEBUG: car_body_id = %d\n", car_body_id);
printf("DEBUG: car_velocity = %p\n", car_velocity);
```

边界检查

```
cpp
if(car_body_id < 0){printf("ERROR: Car body not found!\n");
return;
}
```

数值验证

```
cpp
// 验证速度计算结果
if (speed_kmh < 0||speed_kmh > 100){prin tf("WARNING: Invalid speed: %.2f km/h\n",
speed_kmh);
}
```

七、测试结果

7.1 功能测试

基础功能测试

- 汽车模型加载成功
- 物理仿真运行正常
- 控制输入响应正确

可视化测试

- 3D 仪表盘正确渲染
- 指针随速度变化
- 刻度标签显示正确
- 颜色渐变效果正常

数据显示测试

- 终端输出实时更新
- RPM 条正确显示
- 燃料百分比计算准确
- 数据格式正确

7. 2 性能测试

渲染性能

- 平均帧率: 60 FPS
- 几何体数量: ~50 个
- 内存占用: < 10MB

计算性能

- 状态更新延迟: < 1ms
- 数据计算误差: < 0.1%
- 系统稳定性: 连续运行 30 分钟无异常

八、遇到的问题与解决方案

问题 1：终端输出刷屏

问题描述: 使用\n 换行符导致终端不断滚动，难以阅读。

解决方案:

```
cpp
// 使用回车符实现原地刷新
printf("\rPos(%f, %f)...", pos_x, pos_y);
fflush(stdout); // 强制刷新缓冲区
```

问题 2：几何体渲染顺序

问题描述: 仪表盘组件渲染顺序错误，导致遮挡问题。

解决方案:

- 按照从后到前的顺序渲染组件
- 底座 → 刻度弧线 → 刻度线 → 指针 → 标签
- 合理设置组件的 Z 轴位置

问题 3：坐标系变换

问题描述：仪表盘方向不正确，与汽车方向不匹配。

解决方案：

- 计算正确的旋转矩阵
- 组合 X 轴和 Z 轴旋转
- 应用到所有几何体的变换矩阵

问题 4：燃料计算累积误差

问题描述：燃料消耗计算存在累积误差，导致百分比不准确。

解决方案：

- 使用双精度浮点数
- 定期重置累积误差
- 添加边界检查防止溢出

九、系统特色与创新点

9.1 技术特色

- 双模式显示系统
 - 3D 仪表盘提供直观的视觉反馈
 - 终端界面提供详细的数据信息
 - 两种显示方式数据完全同步
- 实时数据流水线

传感器 → 数据处理 → 可视化渲染

↳ 终端显示

- 模块化架构设计
 - 各功能模块独立
 - 易于扩展和维护
 - 代码结构清晰

9.2 创新点

- 混合渲染技术

- 结合 MuJoCo 原生渲染和自定义几何体
- 实现复杂的仪表盘效果
- 智能刷新机制
 - 终端原地刷新避免刷屏
 - 按需更新减少计算开销
- 完整的车辆状态监控
 - 位置、速度、加速度三合一显示
 - 燃料消耗实时计算
 - RPM 模拟显示

十、总结与展望

10.1 项目总结

本项目成功实现了一个基于 MuJoCo 的汽车仿真仪表盘系统，具有以下特点：

- 功能完整：实现了从物理仿真到可视化显示的完整流程
- 性能优良：系统运行稳定，渲染流畅，数据准确
- 代码规范：采用模块化设计，注释清晰，易于维护
- 用户体验：提供直观的 3D 仪表盘和详细的终端数据显示

10.2 技术收获

通过本项目，获得了以下技术经验：

- MuJoCo 深度使用经验
- 掌握 MuJoCo 的数据结构和 API
- 理解物理仿真的实现原理
- 熟悉 MJCF 模型描述语言
- 实时渲染技术
- 3D 图形渲染的基本原理
- 坐标系变换和矩阵运算
- 几何体组合与分层渲染
- 系统设计能力
- 模块化系统架构设计
- 实时数据处理流水线
- 性能优化和调试技巧

10.3 不足与改进方向

当前不足

- 仪表盘样式较为简单
- 燃料模型过于简化
- 缺少更丰富的交互功能

改进方向

- 添加更多仪表类型（转速表、水温表等）
- 实现更真实的物理模型
- 增加数据记录和分析功能
- 支持多车辆仿真场景

10.4 应用前景

本系统可以作为：

- 教学工具：用于物理仿真和可视化教学
- 研究平台：为控制算法研究提供测试环境
- 演示系统：展示 MuJoCo 的仿真能力
- 开发基础：为更复杂的车辆仿真系统提供基础框架