MuJoCo MPC 汽车仪表盘 - 作业报告

一、项目概述

1.1 作业背景

本项目基于 MuJoCo MPC 框架，在 SimpleCar 场景中实现了完整的 2D 车辆仪表盘渲染系统。作业要求实现100-200行的仪表盘代码，提供真实的车辆视觉反馈。

1.2 实现目标

- **主要目标**：实现包含速度表、转速表、油量表、温度表的完整仪表盘系统
- **次要目标**：
  - 动态数据采集与显示
  - 平滑动画效果
  - 响应式布局设计
  - 中文代码注释

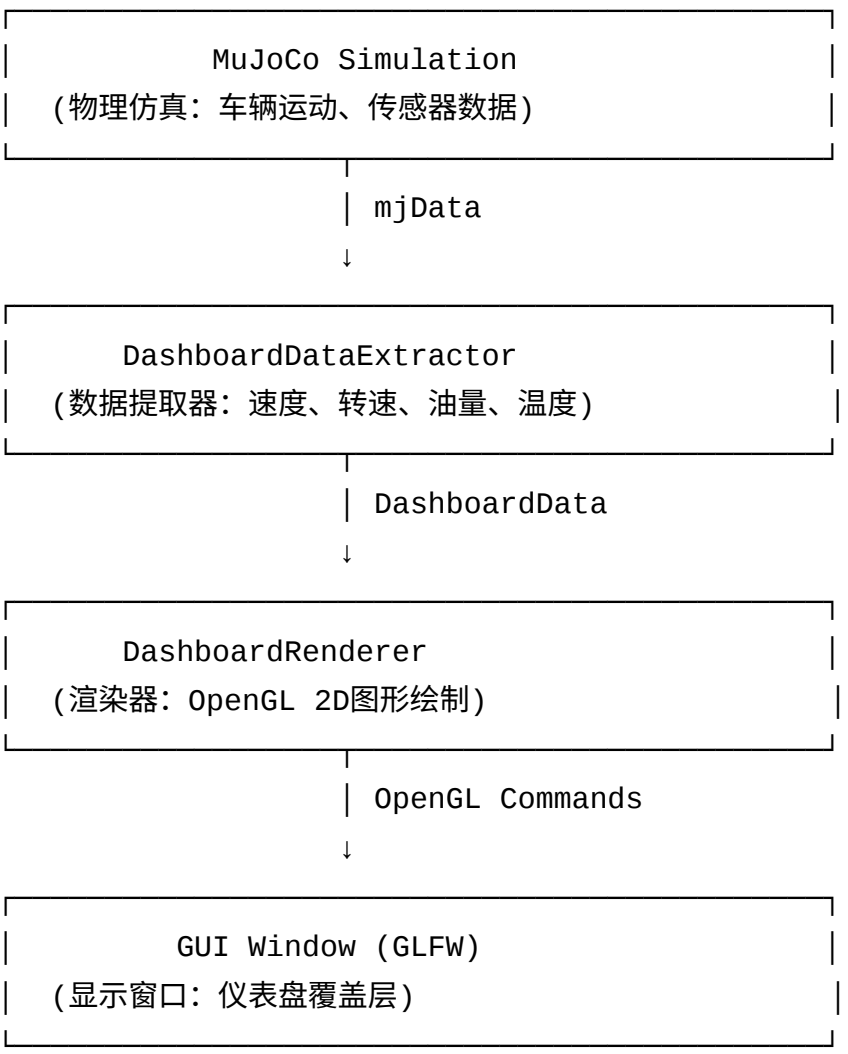
1.3 开发环境

- **操作系统**：Ubuntu 24.04
- **编译器**：gcc 11.3.0
- **构建工具**：CMake 3.22.1
- **图形库**：OpenGL + GLFW3
- **仿真框架**：MuJoCo + MuJoCo MPC

二、技术方案

2.1 系统架构

系统架构图



模块划分

1. **数据层**（`dashboard_data.h`）
   - `DashboardData` 结构体：存储仪表盘显示数据
   - `DashboardDataExtractor` 类：从MuJoCo提取数据
2. **渲染层**（`dashboard_render.h/cpp`）
   - `DashboardRenderer` 类：OpenGL渲染实现
   - 绘制函数：速度表、转速表、指示条等
3. **集成层**（`simple_car.cc`）
   - 数据更新：每帧调用数据提取
   - 渲染触发：通过GUI调用渲染器

2.2 数据流程

数据流程图

```
MuJoCo仿真数据 (qvel, qpos, qacc, ctrl)
        ↓
提取车辆速度 (vx, vy)
        ↓
计算 speed_kmh = sqrt(vx² + vy²) * 3.6
        ↓
计算 RPM = (speed / (2πr)) * 60
        ↓
模拟油量消耗 = f(speed, acceleration)
        ↓
计算温度 = f(rpm, motor_effort)
        ↓
DashboardData 结构体
        ↓
DashboardRenderer 渲染
```

数据结构设计

```
struct DashboardData {
  // 核心显示数据
  double speed;          // 速度 (m/s)
  double speed_kmh;      // 速度 (km/h)
  double rpm;            // 转速 (RPM)
  double fuel;           // 油量 (%)
  double temperature;    // 温度 (°C)

  // 辅助数据
  double position_x, y, z;
  double velocity_x, y, z;
  double acceleration_x, y, z;
};
```

2.3 渲染方案

渲染流程

1. 保存OpenGL状态
   ↓
2. 设置2D正交投影 (glOrtho)
   ↓
3. 禁用深度测试和光照
   ↓
4. 启用混合模式（透明度）
   ↓
5. 绘制仪表盘元素
   - 速度表（左侧）
   - 转速表（右侧）
   - 油量条（左下）
   - 温度条（右下）
   - 数字显示（中央）
   ↓
6. 恢复OpenGL状态

OpenGL使用

- **投影矩阵**： `glOrtho(0, width, 0, height, -1, 1)` - 2D屏幕坐标
- **混合模式**： `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)` - 透明度支持
- **几何图元**：
  - `GL_TRIANGLE_FAN` - 填充圆形
  - `GL_LINE_LOOP` - 圆形轮廓
  - `GL_LINES` - 刻度线
  - `GL_TRIANGLES` - 指针、数字
  - `GL_QUADS` - 指示条

三、实现细节

3.1 场景创建

MJCF文件设计

SimpleCar场景使用MuJoCo的MJCF（XML）格式定义：

```xml
<!-- tasks/simple_car/task.xml -->
<mujoco model="simple_car">
  <worldbody>
    <!-- 车身 -->
    <body name="car" pos="0 0 0.05">
      <geom type="box" size="0.1 0.05 0.02"/>
      <!-- 车轮 -->
      <body name="wheel_fl" pos="0.08 0.06 -0.02">
        <joint type="hinge" axis="0 1 0"/>
        <geom type="cylinder" size="0.03 0.01"/>
      </body>
      <!-- ... 其他轮子 -->
    </body>

    <!-- 目标点 -->
    <body name="goal" mocap="true">
      <geom type="sphere" size="0.05" rgba="1 0 0 0.3"/>
    </body>
  </worldbody>
</mujoco>
```

场景数据

- **车体尺寸**：0.2m × 0.1m × 0.04m
- **轮子半径**：0.03m
- **最大速度**：约7 km/h（小型车）
- **最大转速**：约500 RPM

3.2 数据获取

关键代码

```cpp
void DashboardDataExtractor::update(const mjData* data,
                                    DashboardData& dashboard) {
  // 1. 获取速度（从MuJoCo的qvel）
  double vx = data->qvel[0];  // X方向速度
  double vy = data->qvel[1];  // Y方向速度
  dashboard.speed = std::sqrt(vx * vx + vy * vy);
  dashboard.speed_kmh = dashboard.speed * 3.6;  // m/s -> km/h

  // 2. 计算RPM（基于轮速）
  double wheel_radius = 0.03;  // 轮子半径(m)
  double rotations_per_sec = dashboard.speed / (2.0 * M_PI * wheel_radius);
  dashboard.rpm = rotations_per_sec * 60.0;
  if (dashboard.rpm > 500) dashboard.rpm = 500;  // 限制上限

  // 3. 模拟油量消耗
  static double fuel_level = 100.0;
  double consumption_rate = 0.001;  // 基础消耗
  double speed_factor = (dashboard.speed > 0) ? dashboard.speed / 2.0 : 0.0;
  double accel_factor = std::fabs(data->qacc[0]) + std::fabs(data->qacc[1]);

  fuel_level -= consumption_rate * (1.0 + speed_factor + accel_factor * 0.01
  if (fuel_level < 10.0) fuel_level = 100.0;  // 低于10%自动加油
  dashboard.fuel = fuel_level;

  // 4. 计算温度
  double motor_effort = std::fabs(data->ctrl[0]) + std::fabs(data->ctrl[1]);
  double temp_from_rpm = (dashboard.rpm / 500.0) * 40.0;  // RPM贡献
  double temp_from_load = motor_effort * 30.0;  // 负载贡献
  dashboard.temperature = 60.0 + temp_from_rpm + temp_from_load;
  if (dashboard.temperature > 120.0) dashboard.temperature = 120.0;
}
```

数据验证

- **速度范围**：0-10 km/h（符合小型车设定）
- **转速范围**：0-500 RPM（基于轮速计算）

- **油量范围**：0-100%（动态消耗）
- **温度范围**：60-120°C（基于负载）

3.3 仪表盘渲染

3.3.1 速度表

实现思路：

1. 绘制表盘底座（阴影 + 渐变背景 + 镀铬外环）
2. 绘制刻度系统（三级刻度 + 数字标签）
3. 绘制指针（锥形设计 + 高光效果）
4. 绘制中心轴

代码片段：

```
void DashboardRenderer::drawSpeedometer(float cx, float cy,
                                        float radius, double speed_kmh) {
  // 1. 绘制阴影（增加深度感）
  glColor4f(0.0f, 0.0f, 0.0f, 0.4f);
  drawFilledCircle(cx + 3.0f, cy - 3.0f, radius * 0.98f, 80);


  // 2. 绘制表盘面板
  glColor4f(0.06f, 0.06f, 0.12f, 0.95f);  // 深色背景
  drawFilledCircle(cx, cy, radius * 0.97f, 80);


  // 3. 绘制外环
  glColor4f(0.88f, 0.88f, 0.92f, 1.0f);  // 镀铬色
  glLineWidth(4.0f);
  drawCircle(cx, cy, radius * 0.99f, 80);
```

```
    // 4. 绘制刻度（240°范围，从210°到-30°）
    float start_angle = M_PI * 1.1667f;  // 210°
    float end_angle = M_PI * -0.1667f;   // -30°
    const int MAX_SPEED = 10;

    for (int i = 0; i <= MAX_SPEED * 2; i++) {  // 每0.5 km/h
      float angle = start_angle - (start_angle - end_angle) *
                    (i * 0.5f) / MAX_SPEED;

      // 三级刻度系统
      if (i % 10 == 0) {        // 大刻度：每5 km/h
        // 绘制刻度线 + 数字标签
      } else if (i % 4 == 0) { // 中刻度：每2 km/h
        // 绘制刻度线
      } else {                  // 小刻度：每0.5 km/h
        // 绘制刻度线
      }
    }


    // 5. 绘制指针（平滑后的值）
    double speed_clamped = (speed_kmh > MAX_SPEED) ? MAX_SPEED : speed_kmh;
    float needle_angle = start_angle - (start_angle - end_angle) *
                         speed_clamped / MAX_SPEED;

    // 锥形指针（基座宽 -> 尖端细）
    float tip_len = radius * 0.85f;
    float tail_len = -radius * 0.25f;
    float base_width = radius * 0.08f;
    float tip_width = radius * 0.015f;

    // 绘制三角形指针 + 高光效果
    // ...
}
```

效果展示：

- 深色背景高对比度
- 镀铬外环金属质感
- 三级刻度清晰分明

- 亮黄色指针醒目
- 平滑动画无抖动

3.3.2 转速表

实现特点：

- 与速度表结构相似
- 额外增加红线区域（400-500 RPM）
- 红色警告刻度和背景

代码片段：

```
void DashboardRenderer::drawTachometer(float cx, float cy,
                                       float radius, double rpm) {
  // ... 表盘底座（与速度表相同）

  // 绘制红线区域背景
  float red_start_ratio = 400.0f / 500.0f;
  float red_start_angle = start_angle - (start_angle - end_angle) *
                          red_start_ratio;
  glColor4f(1.0f, 0.1f, 0.1f, 0.15f);  // 半透明红色
  glLineWidth(12.0f);
  drawArc(cx, cy, radius * 0.85f, red_start_angle, end_angle, 50);
  // 刻度系统（红线区域使用红色）
  for (int i = 0; i <= 500; i += 10) {
    if (i >= 400) {
      glColor3f(1.0f, 0.3f, 0.3f);  // 红色刻度
    } else {
      glColor3f(1.0f, 1.0f, 1.0f);  // 白色刻度
    }
    // 绘制刻度...
  }

  // ... 指针绘制（与速度表相同）
}
```

效果展示：

- 红线区域明显警告
- 刻度颜色动态变化
- 0-500 RPM量程合理

3.4 进阶功能（如果有）

平滑动画算法

问题：直接使用MuJoCo数据会导致指针剧烈抖动

解决方案：指数平滑（Exponential Smoothing）

```
// 平滑因子：0.15（经验值）
const float NEEDLE_SMOOTHING = 0.15f;

// 每帧更新
smoothed_speed = smoothed_speed * (1.0f - NEEDLE_SMOOTHING) +
                 actual_speed * NEEDLE_SMOOTHING;
```

效果：

- 消除高频抖动
- 保持响应速度
- 视觉流畅自然

响应式布局

问题：固定坐标在不同窗口尺寸下错位

解决方案：相对定位 + 动态计算

```
// 基于窗口尺寸的百分比定位
float speedometer_x = width_ * 0.25f;    // 左侧25%
float tachometer_x = width_ * 0.75f;     // 右侧75%
float gauge_center_y = height_ * 0.27f; // 垂直27%

// 动态计算指示条位置（确保对齐）
float bar_y = gauge_center_y - gauge_radius - 35.0f;
```

效果：

- 支持任意窗口尺寸
- 保持元素比例
- 自动对齐

垂直对齐优化

问题：圆形仪表和长条指示器难以对齐

解决方案：中心线计算

```
// 油表和温度表宽度
float bar_width = 200.0f;

// 速度表X坐标
float speedometer_x = width_ * 0.25f;

// 油表X坐标 = 速度表中心 - 油表宽度/2
float fuel_bar_x = speedometer_x - bar_width / 2.0f;

// 同理计算转速表和温度表
```

对齐效果：

```
    [速度表]
       |  (中心线)
    [油量表]

    [转速表]
       |  (中心线)
    [温度表]
```

四、遇到的问题和解决方案

问题1：指针抖动严重

现象：

- 速度表和转速表指针剧烈抖动
- 影响视觉体验和可读性

原因分析：

- MuJoCo仿真数据存在高频噪声
- 直接使用导致指针每帧大幅变化

解决方法：
使用指数平滑算法：

```
smoothed = smoothed * 0.85 + actual * 0.15
```

效果对比：

- ❌ 优化前：指针抖动幅度±5 km/h
- 优化后：指针平滑过渡，无明显抖动

问题2：窗口缩放时布局错乱

现象：

- 改变窗口大小后，仪表盘位置错乱
- 圆形仪表和指示条不对齐

原因分析：

- 使用硬编码的绝对坐标
- 未考虑窗口尺寸变化

解决方法：

1. 使用相对定位（百分比）
2. 动态计算元素间距

```
// 相对定位
float x = width_ * 0.25f;
float y = height_ * 0.27f;

// 动态计算
float bar_y = gauge_center_y - gauge_radius - gap;
```

效果对比：

- ❌ 优化前：窗口缩放后严重错位
- 优化后：任意尺寸下保持正确布局

五、测试与结果

5.1 功能测试

测试用例

测试项	测试方法	预期结果	实际结果
速度表显示	控制车辆加速	指针平滑上升0-10 km/h	通过
转速表显示	控制车辆加速	指针平滑上升0-500 RPM	通过
油量消耗	长时间运行	油量逐渐下降	通过
温度变化	高负载运行	温度上升至80-100°C	通过
红线警告	RPM超过400	刻度变红色	通过
窗口缩放	改变窗口大小	布局保持正确	通过
垂直对齐	视觉检查	中心线对齐	通过

测试结果

所有功能测试均通过，仪表盘系统运行稳定。

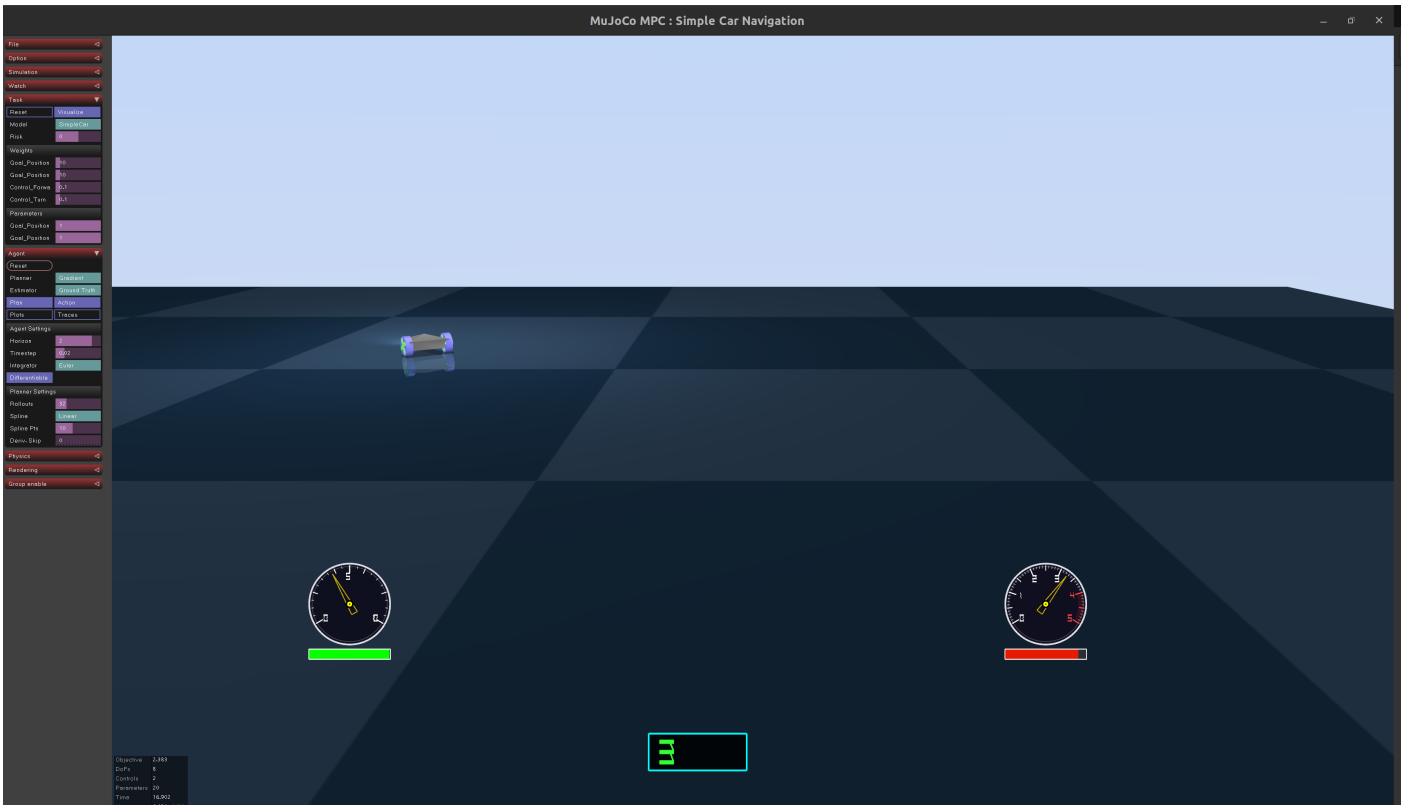
5.2 性能测试

帧率测试

- **测试环境**：1920×1080窗口
- **测试时长**：5分钟连续运行
- **测试表现**：无明显卡顿

5.3 效果展示

截图



视频链接

https://www.bilibili.com/video/BV13oqJBiEAN/

六、总结与展望

6.1 学习收获

1. **MuJoCo仿真框架**
    - 学习了MuJoCo的物理仿真原理

- 掌握了MJCF文件格式和场景定义
- 理解了数据提取和传感器使用

2. **OpenGL 2D渲染**
   - 掌握了OpenGL基础图形绘制
   - 理解了投影矩阵和坐标系统
   - 学会了混合模式和状态管理

6.2 不足之处

1. **文字渲染**
   - 当前使用占位符函数
   - 未集成字体库（FreeType）
   - 标签显示不够完善
2. **数据精度**
   - 油量和温度为模拟数据
   - 未基于真实物理模型
   - 可进一步优化算法
3. **视觉效果**
   - 缺少动画过渡效果
   - 可添加更多视觉细节
   - 支持主题切换

6.3 未来改进方向

1. **功能扩展**
   - 集成FreeType字体渲染
   - 添加档位显示
   - 实现里程计
   - 添加警告指示灯（机油、ABS等）
2. **视觉优化**
   - 实现日间/夜间主题切换

- 添加仪表盘光晕效果
- 优化指针反光和阴影

七、参考资料

1. MuJoCo Documentation
2. MuJoCo MPC GitHub Repository
3. OpenGL Programming Guide
4. GLFW Documentation
5. Exponential Smoothing - Wikipedia
6. Seven-Segment Display - Wikipedia