MuJoCo MPC SimpleCar 仪表盘系统

项目信息

- **学号**: 232011108
- **姓名**: 陈晓
- **班级**: 计科2303

项目概述

本项目在 MuJoCo MPC 框架的 SimpleCar 场景中实现了完整的 2D 车辆仪表盘渲染系统。使用 OpenGL 绘制，实现了速度表、转速表、油量指示、温度指示等功能，并集成了动态数据采集、平滑动画、响应式布局等特性。

环境要求

- **操作系统**: Ubuntu 24.04
- **编译器**: gcc 11.3.0
- **CMake**: 3.22.1
- **MuJoCo**: 最新版本
- **其他依赖**: OpenGL, GLFW3

编译和运行

编译步骤

```
cd mujoco_mpc
mkdir -p build && cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
make
```

运行

```
./bin/mjpc --task SimpleCar
```

运行后，在SimpleCar场景中可以看到左下角的完整仪表盘系统。

功能说明

已实现功能

仪表显示功能

- [x] **速度表**（圆形仪表盘）
  - 0-10 km/h 量程范围
  - 三级刻度系统（0.5/2/5 km/h）
  - 平滑指针动画
  - 七段数码管风格数字标签
- [x] **转速表**（圆形仪表盘）
  - 0-500 RPM 量程范围
  - 三级刻度系统（10/50/100 RPM）
  - 红线区域标记（400-500 RPM）
  - 平滑指针动画
- [x] **油量指示条**（水平长条）
  - 0-100% 动态显示
  - 渐变色彩（红→黄→绿）
  - 基于速度和加速度的真实消耗模拟
  - 自动加油循环
- [x] **温度指示条**（水平长条）
  - 60-120°C 动态显示
  - 渐变色彩（绿→黄→红）
  - 基于电机负载和转速的温度模拟
- [x] **数字速度显示**（底部中央）
  - 七段数码管风格
  - 高对比度颜色（绿/黄/红随速度变化）
  - 大号字体，易于阅读

技术特性

- [x] **指数平滑动画** - 消除指针抖动
- [x] **响应式布局** - 支持窗口任意缩放
- [x] **垂直对齐优化** - 圆形仪表与长条指示器中心线对齐
- [x] **动态数据计算** - 实时采集MuJoCo仿真数据
- [x] **中文代码注释** - 提高可读性和可维护性

详细功能

- [ ] 小细节（未实现）

文件说明

核心文件结构

```
mjpc/
├── dashboard_data.h            # 仪表盘数据结构及提取器（约130行）
├── dashboard_render.h          # 仪表盘渲染模块接口（约50行）
├── dashboard_render.cpp        # 仪表盘渲染核心实现（约740行）
└── tasks/simple_car/
    ├── simple_car.h           # SimpleCar任务头文件
    └── simple_car.cc          # SimpleCar任务逻辑及仪表盘集成
```

文件功能说明

1. `dashboard_data.h` - 数据层

功能: 定义仪表盘数据结构和数据提取器

核心内容:

- `DashboardData` 结构体：存储所有仪表盘显示数据
- `DashboardDataExtractor` 类：从 MuJoCo 仿真中提取数据

关键数据字段:

```
struct DashboardData {
  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)
  double position_x, y, z;     // 位置
  double velocity_x, y, z;     // 速度
  double acceleration_x, y, z; // 加速度
};
```

数据提取逻辑:

```
void update(const mjData* data, DashboardData& dashboard) {
  // 1. 从MuJoCo获取速度
  double vx = data->qvel[0];
  double vy = data->qvel[1];
  dashboard.speed = sqrt(vx*vx + vy*vy);
  dashboard.speed_kmh = dashboard.speed * 3.6;

  // 2. 计算RPM（基于轮速）
  double wheel_radius = 0.03;
  double rotations_per_sec = speed / (2*PI*wheel_radius);
  dashboard.rpm = rotations_per_sec * 60.0;

  // 3. 模拟油量消耗
  fuel_level -= consumption_rate * (1 + speed_factor + accel_factor);

  // 4. 计算温度（基于负载）
  temperature = 60.0 + temp_from_rpm + temp_from_load;
}
```

2. `dashboard_render.h` - 接口层

功能: 定义渲染器类接口

核心类:

```
class DashboardRenderer {
 public:
  // 构造函数
  DashboardRenderer(int window_width, int window_height);

  // 渲染仪表盘覆盖层
  void render(const DashboardData& data);

  // 更新窗口尺寸
  void resize(int width, int height);

 private:
  // 平滑动画变量
  float smoothed_speed_kmh_;
  float smoothed_rpm_;

  // 绘制函数
  void drawSpeedometer(...);    // 速度表
  void drawTachometer(...);     // 转速表
  void drawFuelBar(...);        // 油量条
  void drawTemperatureBar(...); // 温度条
  void drawSpeedText(...);      // 数字显示
};
```

3. `dashboard_render.cpp` - 实现层（核心）

功能: 完整的仪表盘渲染实现

代码结构:

```
1. 渲染主函数 (render)                    - 约70行
2. 几何绘制辅助函数                       - 约30行
   - drawCircle
   - drawFilledCircle
   - drawArc
3. 速度表绘制 (drawSpeedometer)           - 约140行
4. 转速表绘制 (drawTachometer)            - 约180行
5. 油量条绘制 (drawFuelBar)               - 约45行
6. 温度条绘制 (drawTemperatureBar)        - 约45行
7. 速度数字显示 (drawSpeedText)           - 约50行
8. 七段数码管渲染                         - 约60行
```

关键实现细节:

布局计算（响应式设计）

```
// 仪表盘半径（优化后）
float gauge_radius = 100.0f;

// 仪表盘中心Y坐标
float gauge_center_y = height_ * 0.27f;

// 指示条Y坐标（基于仪表盘底部）
float bar_y = gauge_center_y - gauge_radius - 35.0f;

// 油表和温度表宽度
float bar_width = 200.0f;

// 左侧：速度表与油表对齐
float speedometer_x = width_ * 0.25f;
float fuel_bar_x = speedometer_x - bar_width / 2.0f;

// 右侧：转速表与温度表对齐
float tachometer_x = width_ * 0.75f;
float temp_bar_x = tachometer_x - bar_width / 2.0f;
```

平滑动画算法

```
// 指数平滑（Exponential Smoothing）
smoothed_speed = smoothed_speed * 0.85 + actual_speed * 0.15;
smoothed_rpm = smoothed_rpm * 0.85 + actual_rpm * 0.15;
```

速度表刻度系统

```
for (int i = 0; i <= MAX_SPEED * 2; i++) {
  float speed_value = i * 0.5f;  // 每0.5 km/h

  if (i % 10 == 0) {      // 每5 km/h - 大刻度
    inner_r = 0.76f; outer_r = 0.95f; line_width = 2.5f;
    // 绘制数字标签
  }
  else if (i % 4 == 0) {  // 每2 km/h - 中刻度
    inner_r = 0.82f; outer_r = 0.95f; line_width = 2.0f;
  }
  else {                  // 每0.5 km/h - 小刻度
    inner_r = 0.88f; outer_r = 0.94f; line_width = 1.0f;
  }
}
```

指针绘制（锥形设计）

```
// 计算指针角度
float needle_angle = start_angle - (start_angle - end_angle) *
                     speed / MAX_SPEED;

// 三角形锥形指针
float tip_len = radius * 0.85f;    // 尖端长度
float tail_len = -radius * 0.25f;  // 尾部长度
float base_width = radius * 0.08f; // 基座宽度
float tip_width = radius * 0.015f; // 尖端宽度

// 绘制：基座 → 尖端的平滑过渡
// 加上高光效果和阴影
```

渐变色彩计算

```
// 油量表：红→黄→绿
float fuel_r = (fuel < 50) ? 1.0 : (100-fuel)*2/100;
float fuel_g = (fuel < 50) ? fuel*2/100 : 1.0;

// 温度表：绿→黄→红
float temp_r = temp_ratio;
float temp_g = 1.0 - temp_ratio;
```

4. `tasks/simple_car/simple_car.cc` - 集成层

功能: 将仪表盘集成到SimpleCar任务

集成步骤:

1. **数据更新**（每帧调用）

```
void SimpleCar::ModifyScene(...) const {
  UpdateDashboardData(model, data);
}

void SimpleCar::UpdateDashboardData(...) const {
  // 初始化数据提取器
  if (!data_extractor_) {
    data_extractor_ = std::make_unique<DashboardDataExtractor>(model);
  }

  // 更新仪表盘数据
  data_extractor_->update(data, dashboard_data_);
}
```

2. **渲染触发**（由GUI层调用）

```
// 在app.cc中调用
if (dashboard_renderer_) {
  dashboard_renderer_->render(dashboard_data_);
}
```

实现亮点

1. 真实物理模拟

RPM计算（基于轮速）

```
// 根据车轮半径和速度计算转速
double wheel_radius = 0.03;  // 米
double rotations_per_sec = speed / (2 * PI * wheel_radius);
double rpm = rotations_per_sec * 60.0;
```

油量动态消耗

```
// 消耗因子：基础 + 速度因子 + 加速度因子
consumption = base_rate * (1.0 + speed/2.0 + |acceleration|*0.01);
fuel_level -= consumption;

// 低于10%自动加油
if (fuel < 10%) fuel = 100%;
```

温度动态计算

```
// 基础温度 60°C
// RPM贡献：(rpm/500) * 40°C
// 负载贡献：motor_effort * 30°C
temperature = 60 + rpm_temp + load_temp;
if (temperature > 120) temperature = 120;  // 限制最高120°C
```

2. 视觉设计优化

圆形仪表盘设计

- **阴影效果**：增加深度感
- **镀铬外环**：金属质感
- **深色表盘**：高对比度背景
- **分层刻度**：清晰的视觉层次
- **高光指针**：立体感和光泽

色彩方案

- **速度表指针**：亮黄色（高可见性）
- **转速表红线**：红色警告区域
- **油量表**：红→黄→绿渐变
- **温度表**：绿→黄→红渐变
- **数字显示**：根据速度动态变色

3. 性能优化

平滑动画

```
// 指数平滑因子：0.15
// 优点：计算简单，内存占用小，效果自然
const float NEEDLE_SMOOTHING = 0.15f;
```

减少GL调用

```
// 批量绘制刻度
glBegin(GL_LINES);
for (刻度) {
  glVertex2f(...);
}
glEnd();
```

响应式计算

```
// 基于窗口尺寸的相对定位，避免硬编码
float x = width_ * 0.25f;
float y = height_ * 0.27f;
```

技术难点及解决方案

难点1：指针平滑移动

问题: 直接使用MuJoCo数据会导致指针剧烈抖动

解决: 指数平滑算法

```
smoothed = smoothed * (1 - α) + actual * α
```

其中 α = 0.15，在流畅性和响应速度间取得平衡

难点2：窗口缩放时的对齐

问题: 固定坐标在不同窗口尺寸下布局混乱

解决: 响应式相对定位

```
// 使用百分比定位
x = width_ * ratio;
y = height_ * ratio;

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

难点3：圆形仪表与长条指示器对齐

问题: 圆心和矩形中心线难以对齐

解决: 中心线计算

```
// 圆形仪表中心
float circle_x = width_ * 0.25f;

// 长条起点 = 圆心 - 宽度/2
float bar_x = circle_x - bar_width / 2.0f;
```

难点4：七段数码管渲染

问题: 没有字体库，需要绘制数字

解决: 七段数码管算法

```
// 定义7个段的显示规则
const bool segmap[10][7] = {
  {1,1,1,1,1,1,0}, // 0
  {0,1,1,0,0,0,0}, // 1
  ...
};

// 根据数字点亮对应的段
for (segment in 7_segments) {
  if (segmap[digit][segment]) {
    draw_segment();
  }
}
```

环境要求

系统要求

- 操作系统: Ubuntu 24.04
- 编译器: gcc 11.3.0
- CMake: 3.22.1

依赖库

- MuJoCo: 最新版本
- OpenGL: 支持2D渲染
- GLFW3: 窗口管理

已知问题

速度数字显示不清晰

参考资料

- MuJoCo Documentation
- MuJoCo MPC GitHub
- OpenGL 2D Rendering
- Exponential Smoothing Algorithm