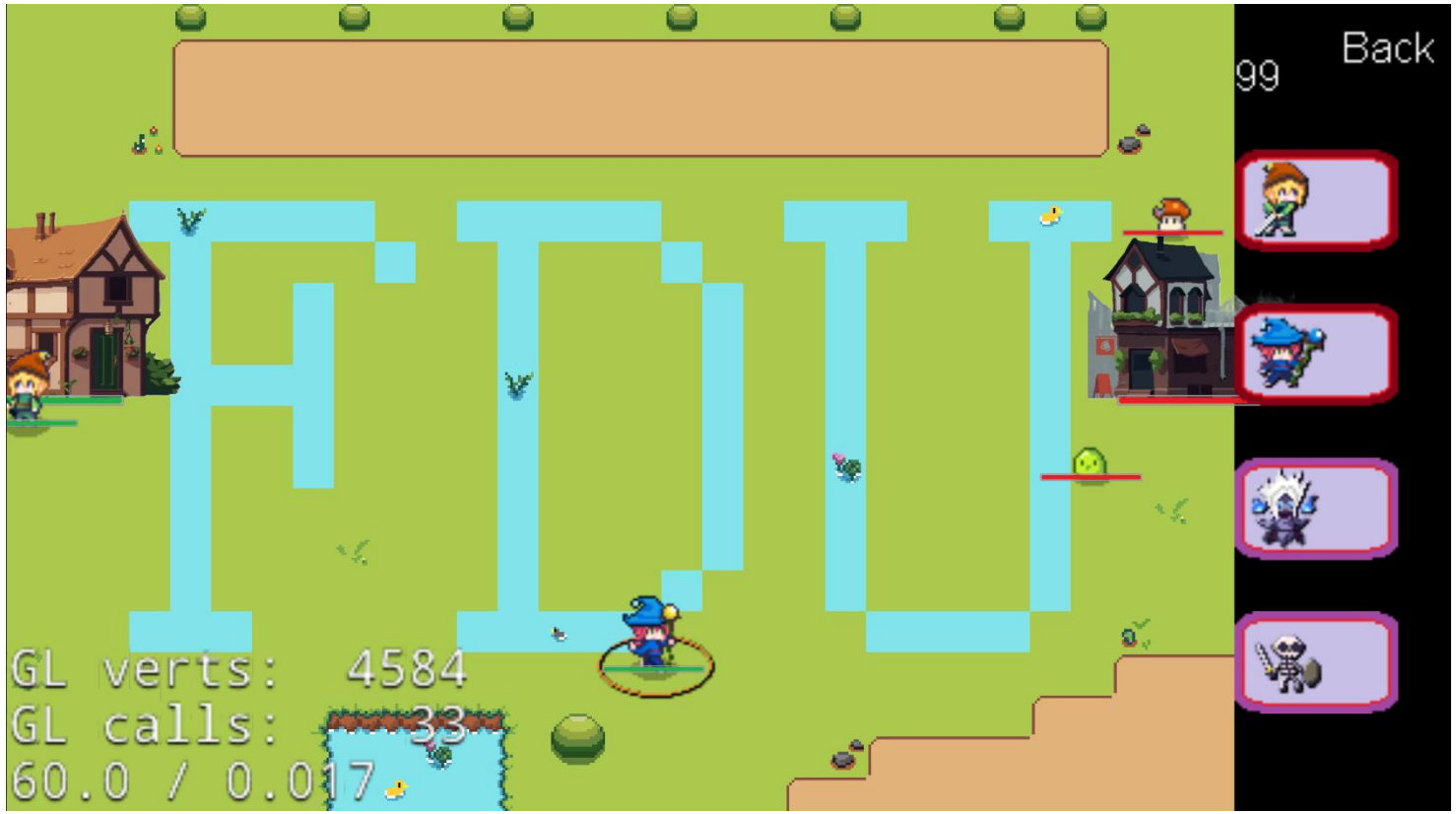
RTS游戏设计

Presenter:Zevick

目录

游戏画面

游戏画面



- RTS游戏，选择部队指定移动
- 自动索敌，战斗，追击。击破大本营获胜。

游戏配置

	A	B	C	D	E	F	G
1	template_r	state	animation_	frame_cou	frame_duration(ticks)		
2	Knight	idle	Knight_Idle	5	60		
3	Knight	run	Knight_Rur	7	60		
4	Knight	attack	Knight_Att	8	60		
5	Knight	death	Knight_Dea	6	60		
6	Wizard	idle	Wizard_Idl	5	60		
7	Wizard	run	Wizard_Ru	7	60		
8	Wizard	attack	Wizard_Att	9	60		
9	Wizard	death	Wizard_De	6	60		
10	Ghost	idle	Ghost_Idle	6	60		
11	Ghost	run	Ghost_Fly	5	60		

	A	B	C	D	E	F	G	H	I
1	template n	hp	atk	atk_range	atk_interva	atk_type	atk_radius	alert_range	move_speed
2	Knight	100	30	2	200	SINGLE	0	10	5
3	Wizard	70	40	4	300	AOE	1	10	3
4	Ghost	200	30	3	300	SINGLE	0	7	4
5	Skeleton	200	20	2	300	SINGLE	0	7	5
6	Mushroom	50	10	1	300	SINGLE	0	4	2
7	Slime	50	10	1	300	SINGLE	0	4	1
8	Building1	500	0	0	0	NULL	0	0	0
9	Building2	500	0	0	0	NULL	0	0	0

- 使用excel配置角色动画及基本信息，数据驱动

开发引擎及工具

基本工具

- 代码开源在git仓库：https://github.com/theElysia/Axmol-RTSGame

Related Tools

- 游戏开发
  - 引擎：Axmol Engine v2.10.0
  - 教程：自行查阅仓库，或者入门可以先参考cocos2d-x manual了解游戏开发基本概念
  - Windows Visual Studio 2026
  - 编程语言：C++23 (应该是至少用到了C++17的特性)
  - 补充：与Cocos2d-x引擎相似，你也可以选择使用该引擎开发。
- 角色动画制作 `.plist`
  - 工具：TexturePacker
  - 教程：https://www.codeandweb.com/texturepacker/tutorials/animations-and-spritesheets-in-axmol-engine
- 地图制作 `.tmx`
  - 工具：Tiled
  - 教程：https://www.bilibili.com/video/BV1ubjZzqEPN
- 主要美术素材来源
  - https://www.bilibili.com/video/BV1idNJz5Exu
  - https://www.bilibili.com/video/BV14T4y1Z7np

基本工具

README | Contributing | MIT license

Axmol Engine

A Multi-platform Engine for Desktop, XBOX (UWP), WebAssembly and Mobile games.

Axmol Engine is an open-source, C++ multi-platform engine designed for mobile devices, desktop, and Xbox, well-suited for 2D game development. It was launched in November 2019 as a fork of Cocos2d-x v4.0.

Please visit our Wiki to know more about Axmol.

build passing

release v2.11.1 | license MIT | code quality A | cxxstd c++20

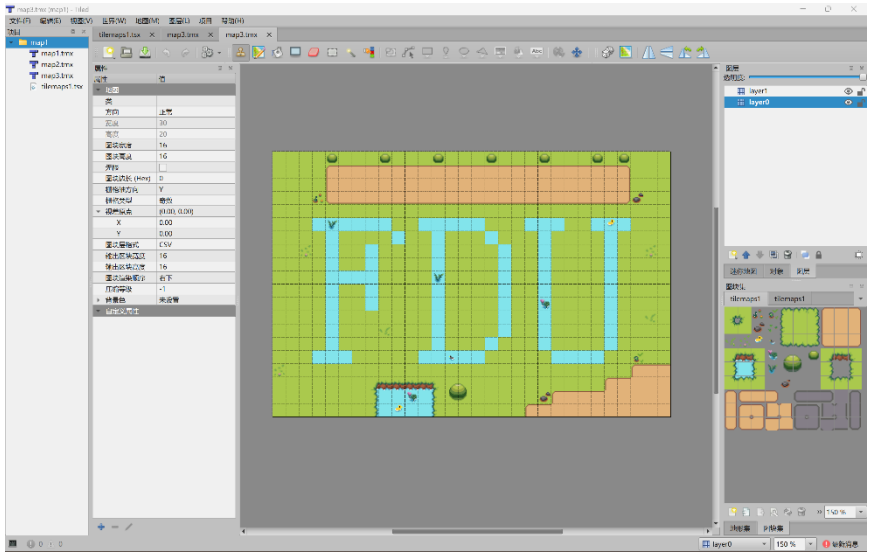
issues 10 open | forks 253 | stars 1.3k | G-Star 4 Stars | code size 36.9 MiB

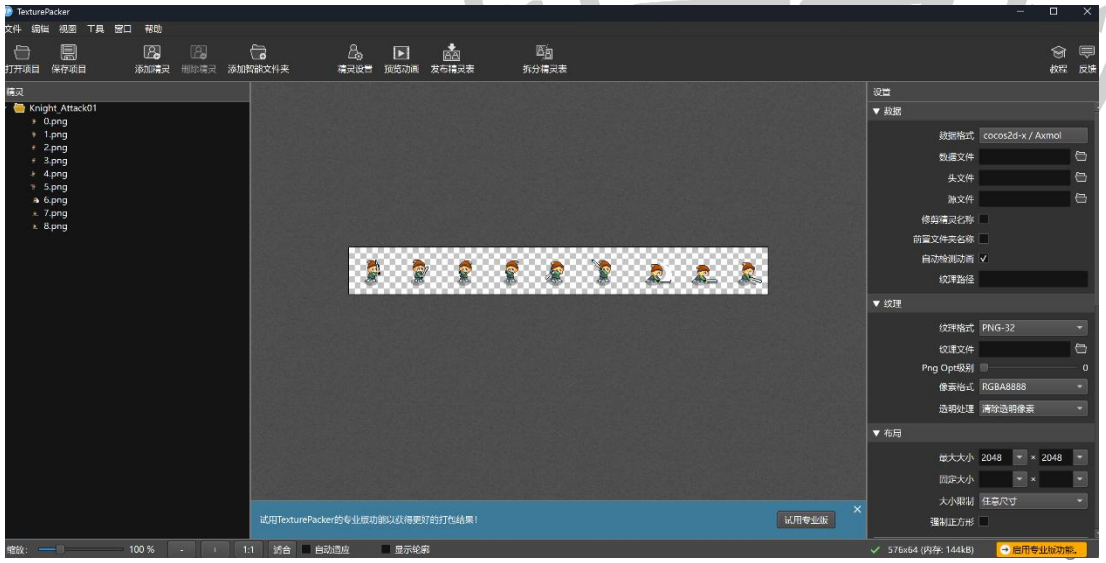
PRs welcome | discord 36 online | awesome-cpp | ossinsight

Chinese ver. / 简体中文

基本工具

- 素材制作。基础美术资源经过加工，成为游戏资源。

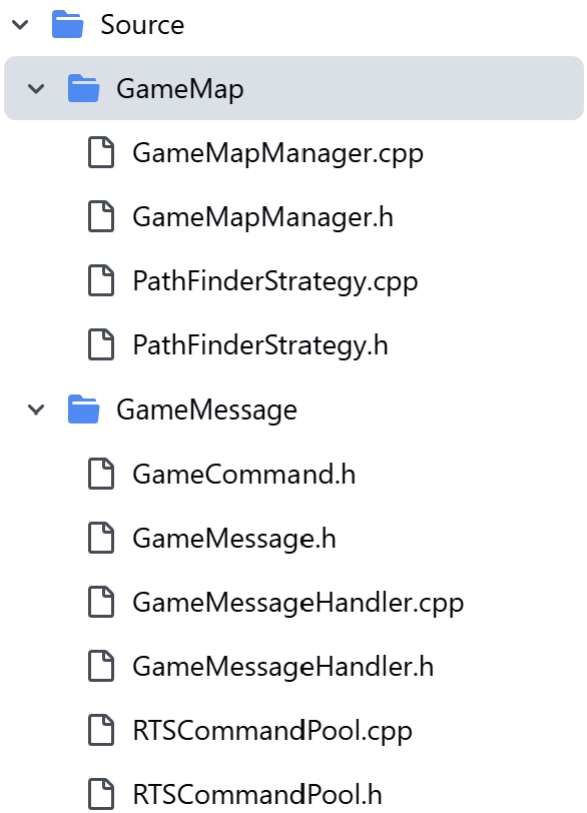


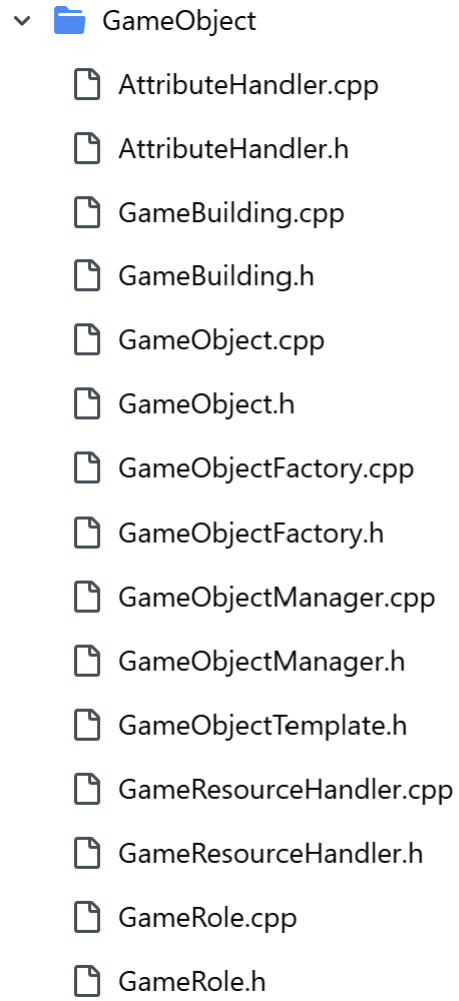


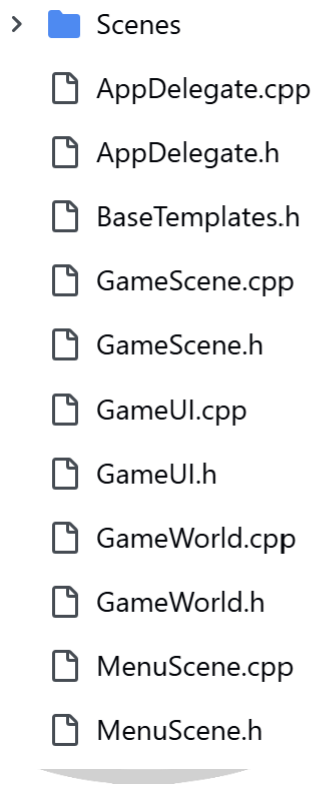
项目整体架构

使用的设计模式

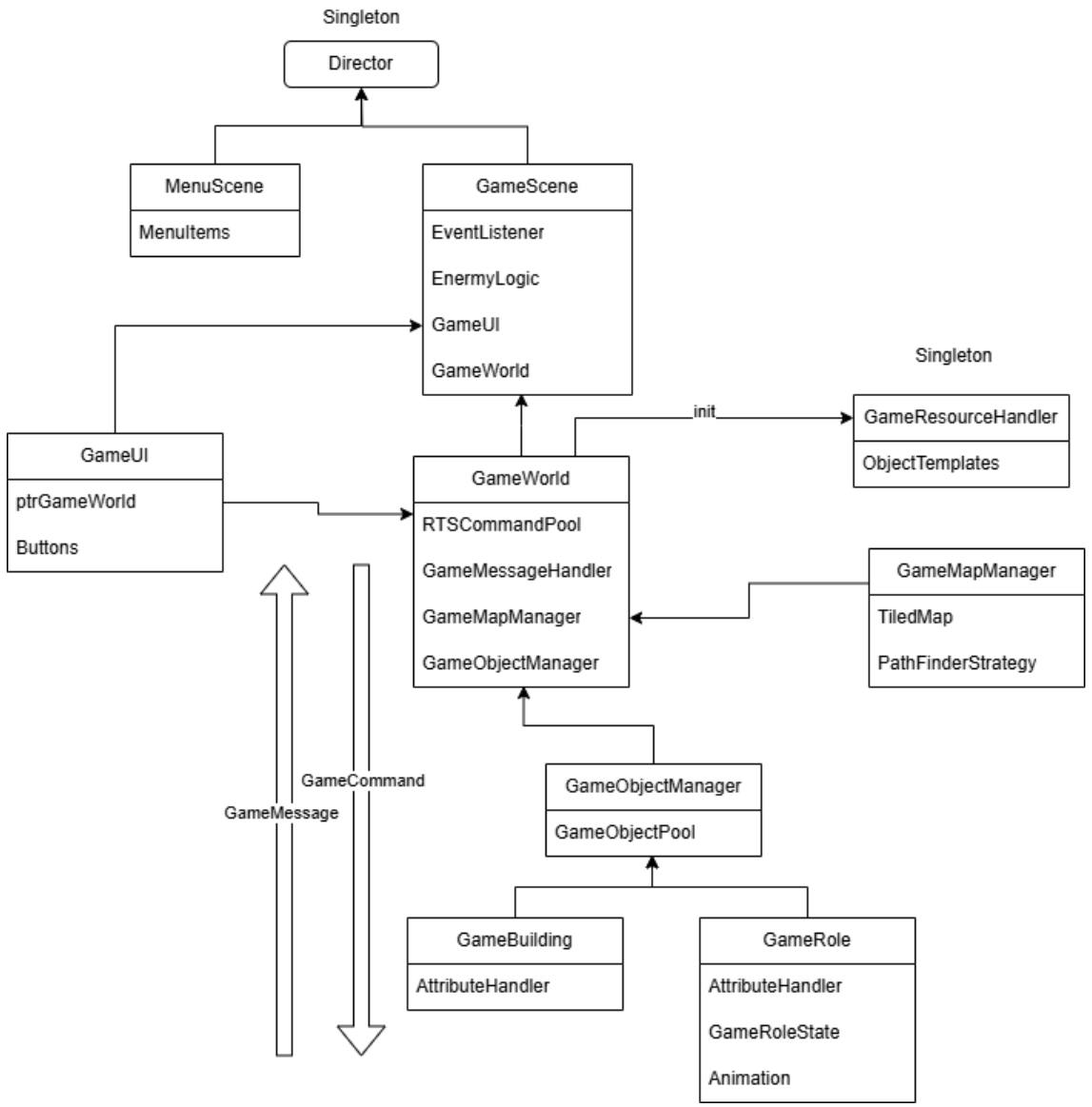
- 重要
  - 单例模式
  - 建造者模式
  - 观察者模式
  - 策略模式
  - 工厂模式
  - 蝇量/享元模式
  - 状态模式
  - 责任链模式
  - 命令模式
  - 原型模式
  - 装饰器模式
  - 组合模式
- 次要
  - 代理模式
  - 空对象模式
  - 模板方法模式
  - 迭代器模式







16/22



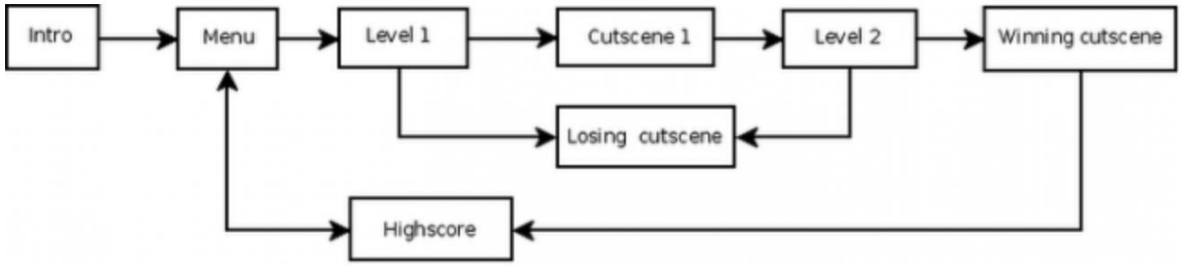
- GameMap
  - GameMapManager，管理场景地图
  - PathFinderStrategy，寻路算法策略模式，已实现适合RTS的流场算法
- GameMessage
  - GameCommand，GameObject接受的命令
  - RTSCommandPool，管理内存池并缓冲命令
  - GameMessage，GameObject发出的消息
  - GameMessageHandler，责任链模式处理消息
- GameObject
  - GameObject为抽象游戏对象类，子类具体对象包含GameBuilding与GameRole
  - GameBuilding，简单设计
  - GameRole，使用状态机，比较完善的设计
  - GameObjectFactory，简单工厂
  - AttributeHandler，具体游戏业务逻辑，用于组合设计
  - GameObjectManager，享元模式，统一管理游戏对象
  - GameResourceHandler，用于预加载游戏资源
  - GameObjectTemplate，原型模式
- Scenes
  - 测试场景，用于测试各项功能，实际不被使用
- AppDelegate
  - 程序入口，基本资源设置
  - MenuScene
    - 基本菜单
  - GameScene
    - 游戏场景
    - GameUI
    - GameWorld

代码分析（设计模式）

游戏开发基本概念

- 导演Director 负责场景转换
- 场景Scene 用渲染器renderer渲染的基本画面
- 精灵Sprite 场景中可操纵的节点Node
- 动作Action 让精灵动起来

这是一个典型的游戏流程实例。当您的游戏设计好时，`Director` 就负责场景的转换：



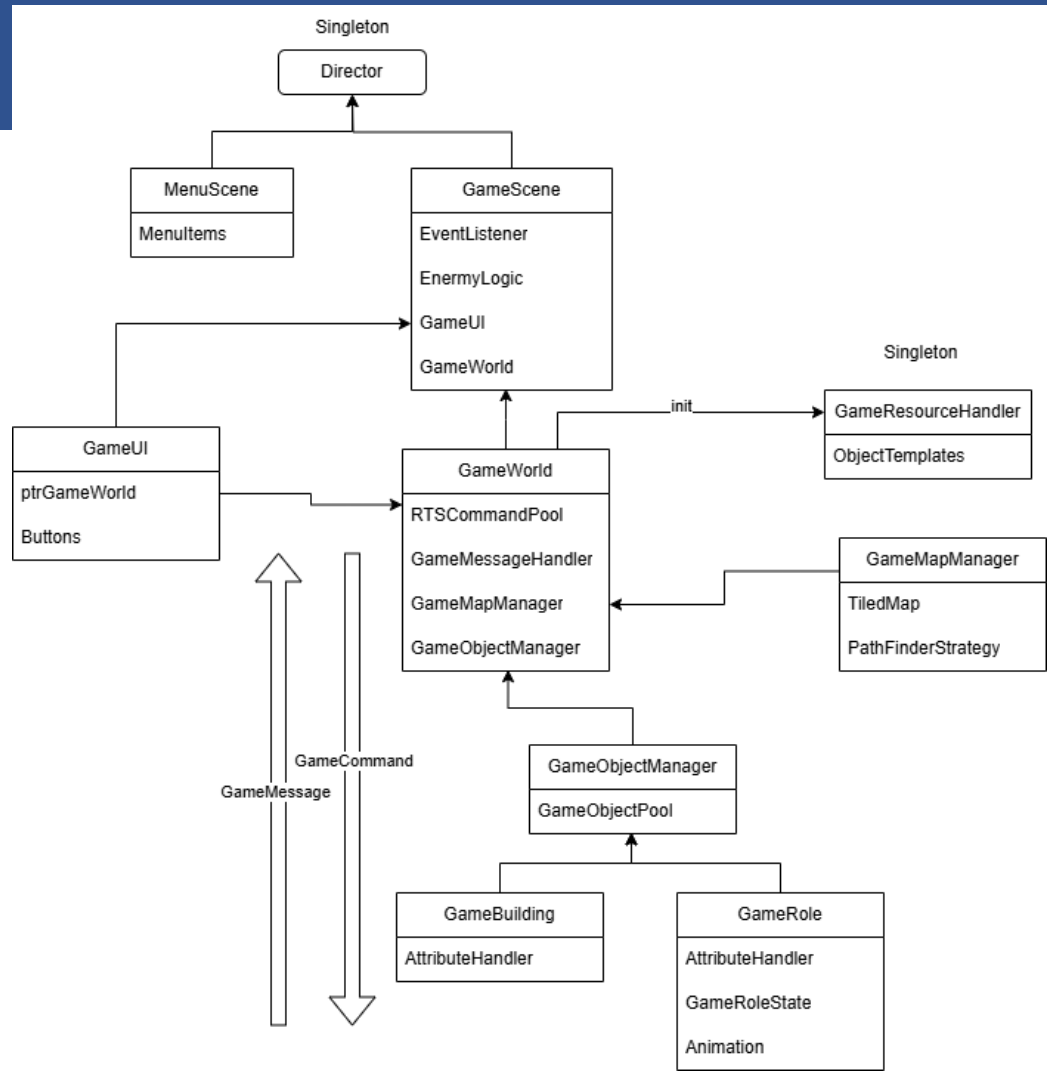
你是你的游戏的导演。你决定着发生什么，何时发生，如何发生。

单例模式(Singleton)

- 比如Director和GameResourceHandler需要单例

```cpp
// CRTP Singleton
template <typename T>
class Singleton
{
protected:
    Singleton()  = default;
    ~Singleton() = default;

    Singleton(const Singleton&)            = delete;
    Singleton& operator=(const Singleton&) = delete;

public:
    static T& instance()
    {
        static T inst;
        return inst;
    }

    static T* getInstance() { return &instance(); }
};
```

```cpp
#define DECLARE_SINGLETON(ClassName)                \
private:                                            \
    ClassName()                             = default; \
    ~ClassName()                            = default; \
    ClassName(const ClassName&)             = delete;  \
    ClassName& operator=(const ClassName&)  = delete;  \
    ClassName(ClassName&&)                  = delete;  \
    ClassName& operator=(ClassName&&)       = delete;  \
                                                       \
public:                                             \
    static ClassName& instance()                    \
    {                                               \
        static ClassName inst;                      \
        return inst;                                \
    }                                               \
    static ClassName* getInstance()                 \
    {                                               \
        return &instance();                         \
    }
```

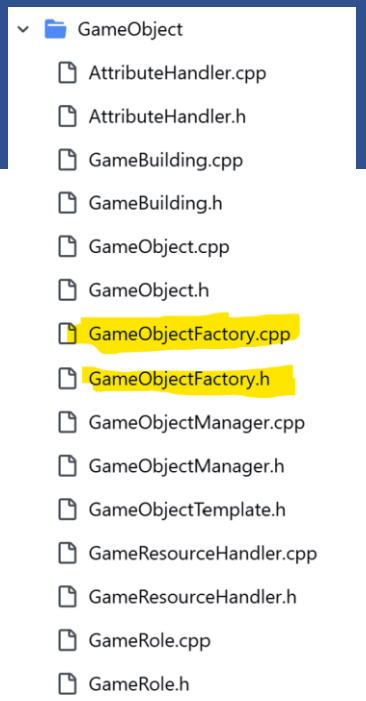


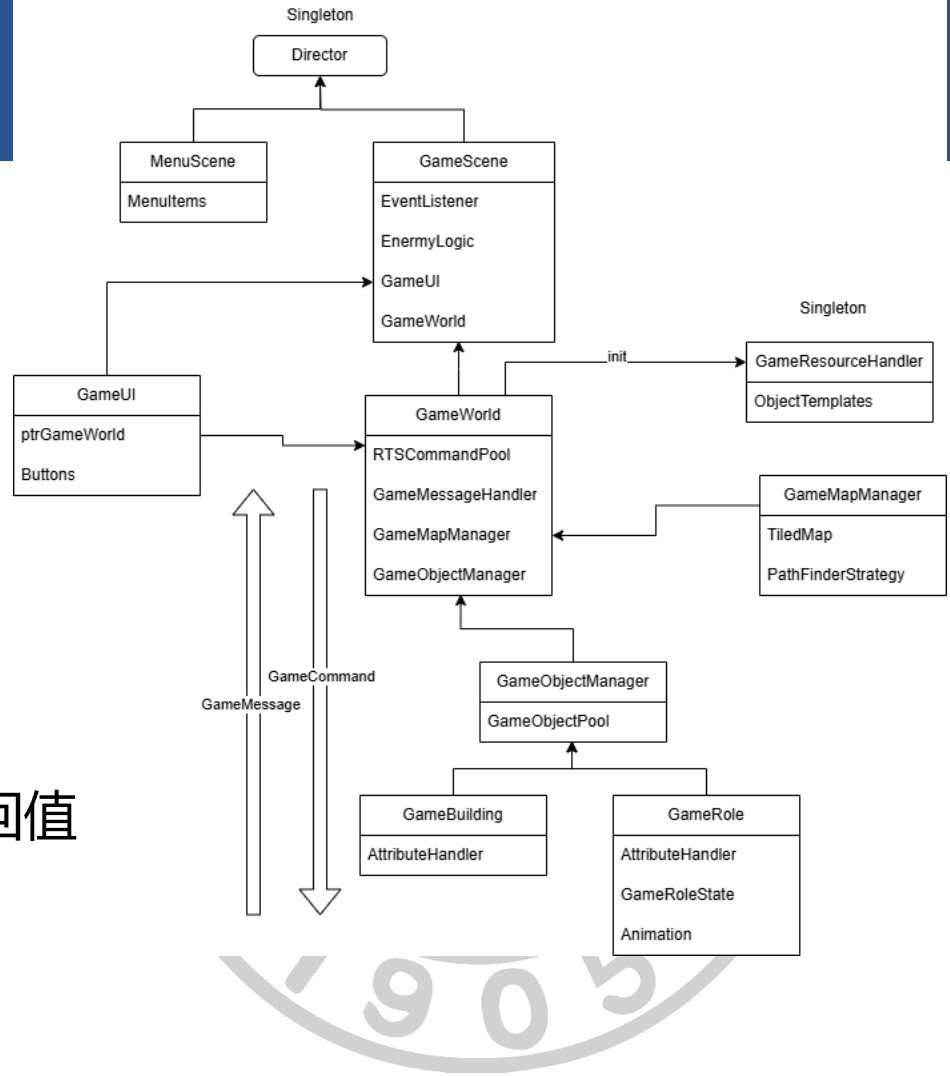
- 使用CRTP，通过模板方式提供接口与实现（类比java implements）
- 或者用宏

建造者模式(Builder)与工厂模式(Factory)

- 讲解一下静态工厂与二段构造方法。

```
1  class A{
2  public:
3      static A* create();
4      virtual bool init();
5  };
```

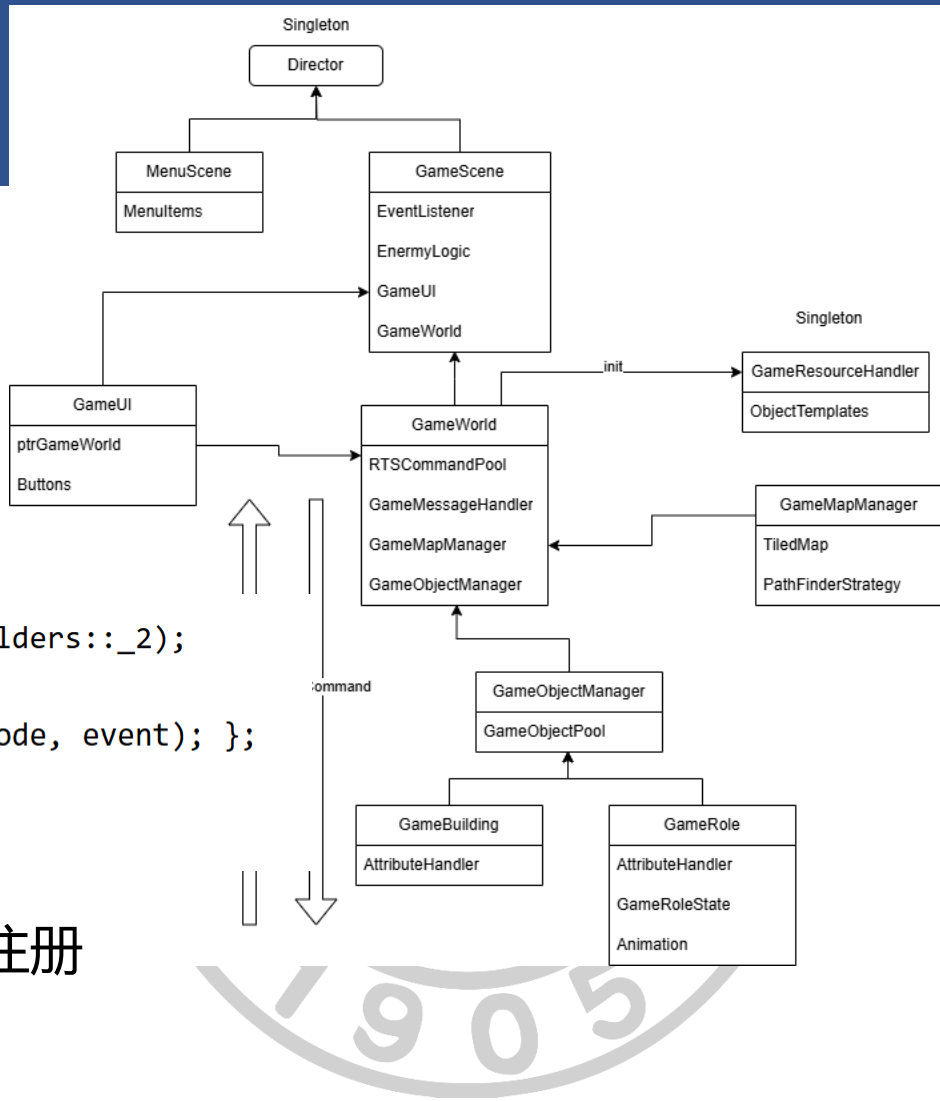




- 二段构造主要是将内存分配与初始化分离，c++构造函数没有返回值

```
01 auto game_role = utils::createInstance<GameRole>(&GameRole::init, args...);
02
03 template <typename T, typename F, typename... Ts>
04 inline T* createInstance(F&& finit, Ts&&... args)
05 {
06     T* pRet = new T();
07     if (std::mem_fn(finit)(pRet, std::forward<Ts>(args)...))
08         pRet->autorelease();
09     else
10     {
11         delete pRet;
12         pRet = nullptr;
13     }
14     return pRet;
15 }
```

观察者模式(Observer)



- C++11 std::bind方法

```
01  std::bind(&GameScene::onKeyReleased, game_scene, std::placeholders::_1, std::placeholders::_2);
02  AX_CALLBACK_2(GameScene::onKeyReleased, this);
03  [=](ax::EventKeyboard::KeyCode code, ax::Event* event) { game_scene->onKeyReleased(code, event); };
04
05  void GameScene::onKeyReleased(ax::EventKeyboard::KeyCode code, ax::Event* event);
```

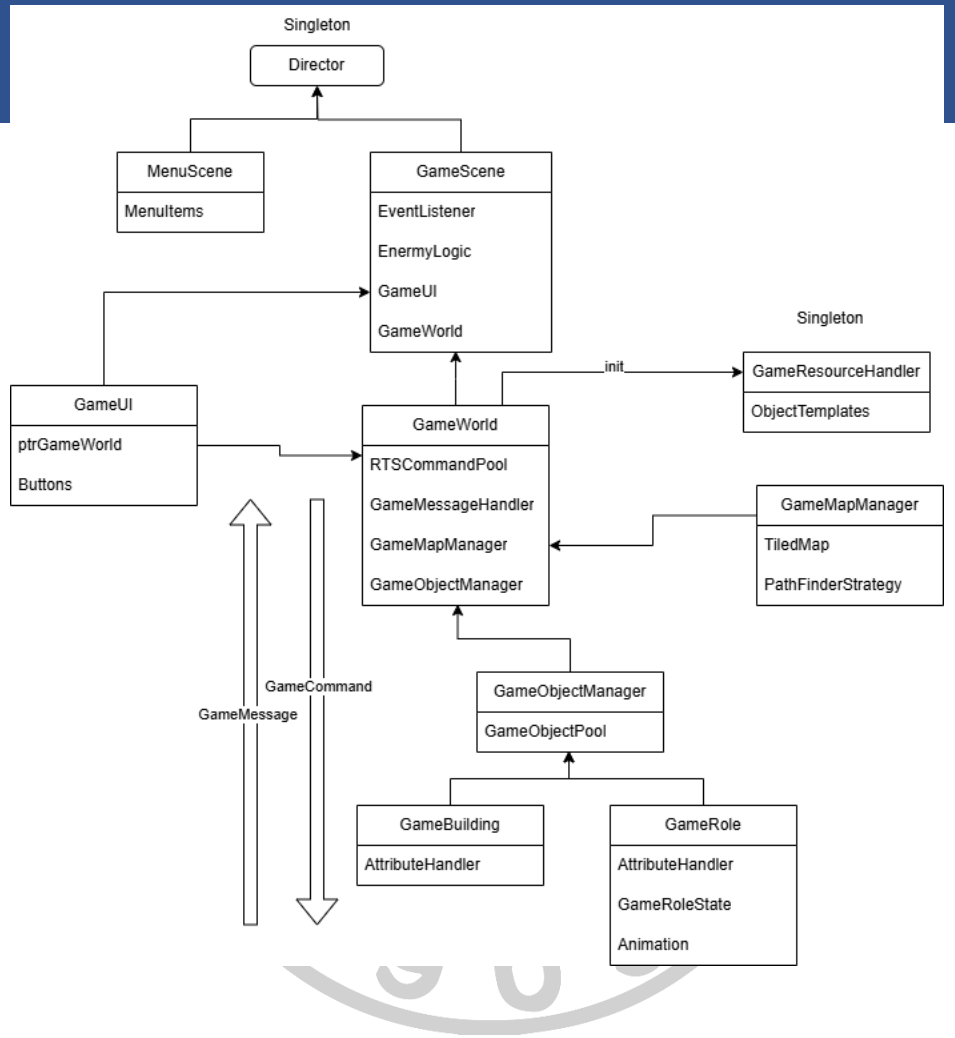
- 游戏需要响应玩家的输入（键盘，鼠标等）。通过向这些监视器注册观察者（回调函数），就可以实现各种功能。

```
1  keyboard_listener_                = EventListenerKeyboard::create();
2  keyboard_listener_->onKeyReleased = AX_CALLBACK_2(GameScene::onKeyReleased, this);
3  keyboard_listener_->onKeyPressed  = AX_CALLBACK_2(GameScene::onKeyPressed, this);
4  _eventDispatcher->addEventListenerWithSceneGraphPriority(keyboard_listener_, this);
```

策略模式(Strategy)

- 寻路逻辑使用策略模式，方便后续扩展与替换

```cpp
class PathFinderStrategy
{
public:
    virtual ~PathFinderStrategy() = default;

    virtual bool init(const std::vector<bool>& grid, int x, int y) = 0;

    virtual void setDest(const Point& dst) = 0;

    virtual std::optional<std::vector<Point>> findPath(const Point& src) = 0;
};

class PathFinder_AStar : public PathFinderStrategy;

class PathFinder_FlowField : public PathFinderStrategy;
```

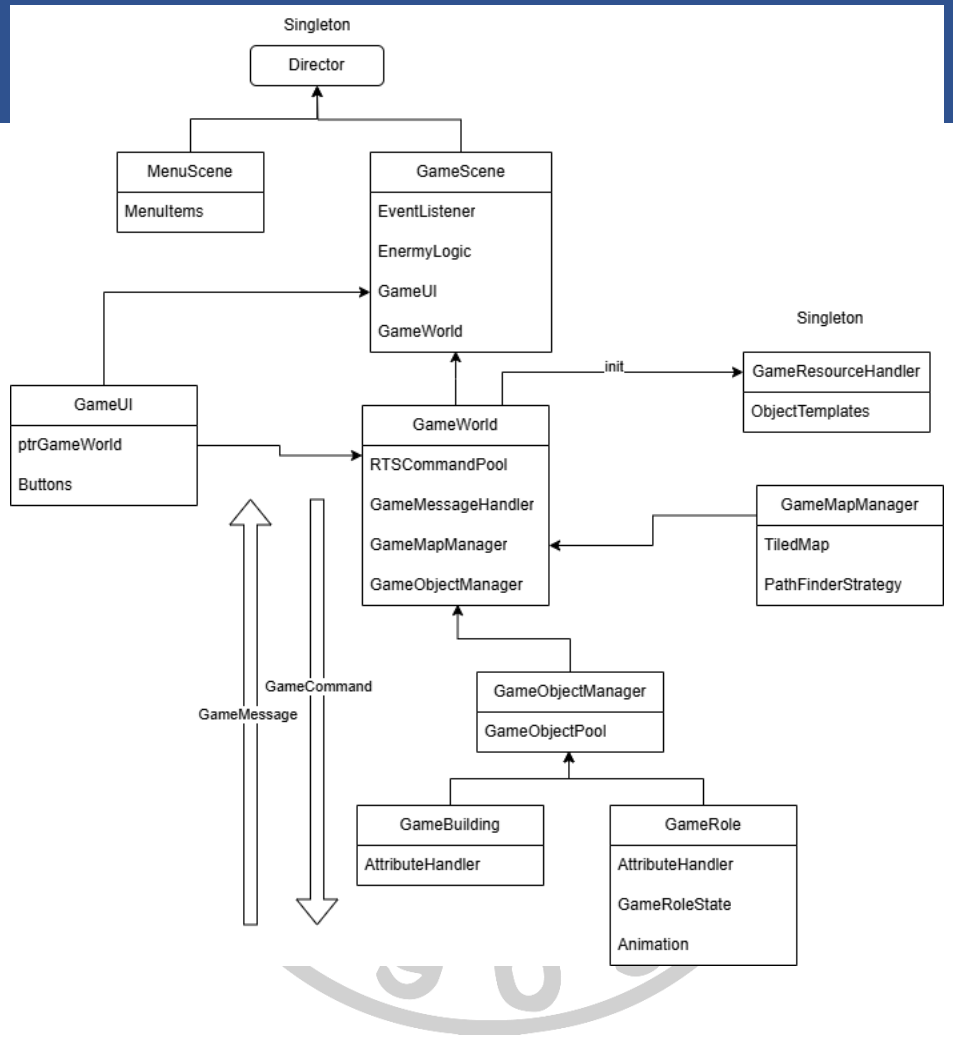


- 实现了比较简单的A*与流场算法，流场算法更适合大量角色移动场景，A*适用于对单个角色寻路

蝇量/享元模式(Flyweight)

- RTS游戏中有大量游戏对象，而且要实现选中移动等功能（对玩家输入做出响应）。若每个对象分别作为一个观察者注册到事件列表中，那么会导致严重的性能问题。

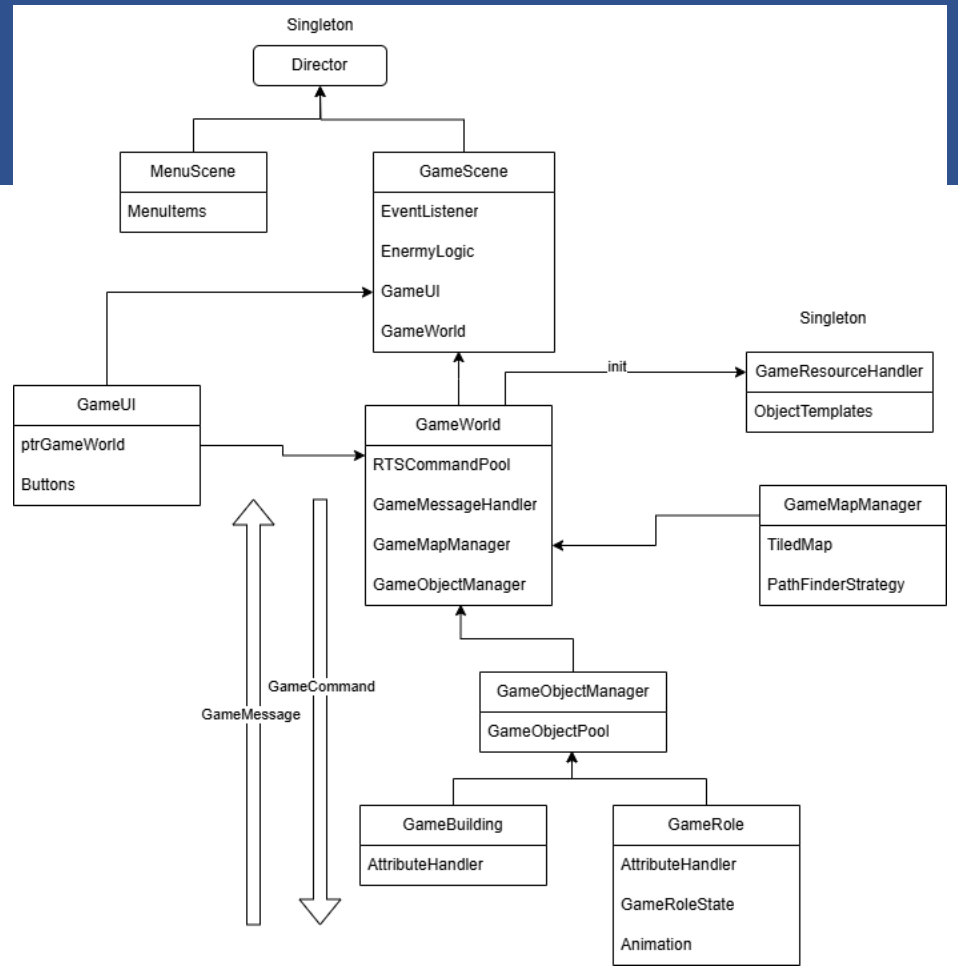
```
class GameObjectManager
{
public:
    GameObject* createGameObject(GameObject::GameObjectType gameObjectType,
                                 GameObject::CampType campType,
                                 const std::string& templateName,
                                 const ax::Vec2& position);

    void update(float delta);

    void selectWithRect(const ax::Vec2& cursorPoint1,
                        const ax::Vec2& cursorPoint2,
                        GameObject::GameObjectType gameObjectType = GameObject::GameObjectType::ROLE,
                        GameObject::CampType campType             = GameObject::CampType::PLAYER);
    void cancelAllSelected();

    void moveSelectedObjTo(const ax::Vec2& position);

    void handleMessage(GameMessage* msg);

private:
    void sendSelectCommand(bool select = true);
    // void sendMessage(GameMessage* msg);

private:
    std::unordered_map<int, GameObject*> game_object_pool_;

    std::unordered_set<int> selected_object_;
    std::vector<int> ready_to_remove_object_;
};
```



- GameObjectManager提供统一的接口，创建并管理游戏对象。

命令模式(Command)

- 最明显的就是GameCommand




```cpp
class GameObject : public ax::Sprite
{
public:
    virtual void handleCommand(GameCommand* cmd) = 0;

    void sendMessage(GameMessage* msg);
};
```

```cpp
class GameCommand
{
public:
    GameCommand(int receiverId) : receiver_id_(receiverId) {}
    virtual ~GameCommand() = default;

    virtual CommandType getType() const = 0;
    int getReceiverId() const { return receiver_id_; }

private:
    int receiver_id_ = 0;
};
```

```cpp
class CommandGetAttack : public GameCommand
{
public:
    struct Data
    {
        int source_id;
        int damage;
    };

    CommandGetAttack(int tid, int sid, int dmg) : GameCommand(tid), data_{.source_id = sid, .damage = dmg} {}

    const Data& getData() const { return data_; }

    CommandType getType() const override { return CommandType::GET_ATTACK; }

private:
    Data data_;
};
```

- 为了适应RTS游戏大量命令并发的场景，特殊设计了RTSCommandPool

- 内存池管理，所有命令均存放于固定内存池中
- 双缓冲结构，缓冲相邻帧的命令
- 批处理模式，批量处理上一帧的命令，优化手段有并行和分组（提高缓存命中）。

```
01 class RTSCommandPool
02 {
03 private:
04     // 帧缓冲区结构
05     struct FrameBuffer{...};
06
07     static constexpr size_t MAX_COMMANDS_PER_FRAME = 10000;
08     static constexpr size_t MAX_COMMAND_SIZE       = 64;  // 最大命令对象大小
09     static constexpr size_t ALIGNMENT              = alignof(std::max_align_t);
10
11     using CommandDeleter = std::function<void(GameCommand*)>;
12
13 public:
14     RTSCommandPool() = default;
15
16     RTSCommandPool(const RTSCommandPool&)            = delete;
17     RTSCommandPool& operator=(const RTSCommandPool&) = delete;
18
19     // 添加命令到当前帧缓冲区，并自动分组
20     template <typename T, typename... Args>
21     void addCommand(Args&&... args);
22
23     void processPreviousFrame(GameObjectManager* manager);
24     void swapBuffers();
25     void clearAll();
```

```
27 private:
28     void processCommandGroup(GameCommand::CommandType type, FrameBuffer& buffer, GameObjectManager* manager);
29
30     void processCommandGroupParallel(const std::vector<std::unique_ptr<GameCommand, CommandDeleter>>& commandGroup,
31                                      GameObjectManager* manager);
32
33     void executeSingleCommand(GameCommand* cmd, GameObjectManager* manager);
34
35     void handleCommandOverflow();
36
37 private:
38     // 双缓冲区
39     FrameBuffer frameBuffers[2];
40     int currentBufferIndex    = 0;  // 当前收集命令的缓冲区
41     int processingBufferIndex = 1;  // 当前正在处理的缓冲区
42
43     // 创建由memoryResource管理内存的command
44     template <typename T, typename... Args>
45     std::unique_ptr<GameCommand, CommandDeleter> createCommand(FrameBuffer& buffer, Args&&... args);
46 };
```

- 内存池管理方法
- <memory_resource>
- <functional><memory>

- Unique_ptr存储cmd
- 自定义析构函数
- 手动memory分配内存

```cpp
// 添加命令到当前帧缓冲区，并自动分组
template <typename T, typename... Args>
void RTSCommandPool::addCommand(Args&&... args)
{
    static_assert(std::is_base_of_v<GameCommand, T>, "T must be derived from GameCommand");

    FrameBuffer& currentBuffer = frameBuffers[currentBufferIndex];

    auto cmd = this->createCommand<T>(currentBuffer, std::forward<Args>(args)...);

    GameCommand::CommandType cmdType = cmd->getType();
    auto& commandGroup               = currentBuffer.getCommandGroup(cmdType);

    commandGroup.push_back(std::move(cmd));
    currentBuffer.totalCommands++;

    if (currentBuffer.totalCommands >= MAX_COMMANDS_PER_FRAME)
    {
        handleCommandOverflow();
    }
}

// 创建由memoryResource管理内存的command
template <typename T, typename... Args>
std::unique_ptr<GameCommand, CommandDeleter> RTSCommandPool::createCommand(FrameBuffer& buffer, Args&&... args)
{
    T* cmd = buffer.allocator.new_object<T>(std::forward<Args>(args)...);

    CommandDeleter deleter = [&buffer](GameCommand* p) {
        if (p)
            p->~GameCommand();
    };

    return std::unique_ptr<GameCommand, CommandDeleter>(cmd, deleter);
}
```

责任链模式(Chain of Responsibility)

- GameMessage由GameObject产生并向上传播，接受这样的消息并处理很自然想到用责任链模式。就是设计多个handler，逐次调用handler来处理一个消息，每个handler可以只处理部分消息，未处理的沿着链往后传递。
- Handler相当于依据msg替obj做决策

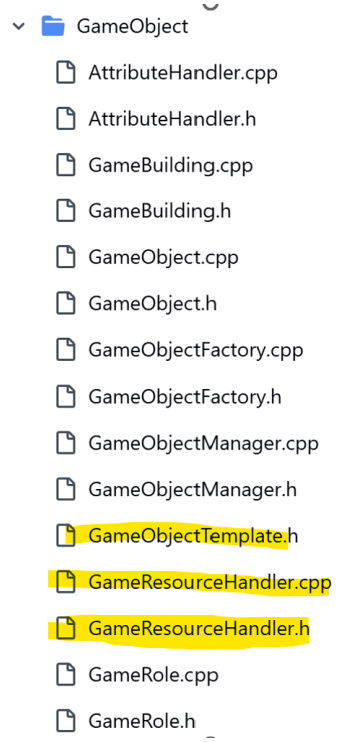
```
ResultType GameMessageHandler::handle(GameMessage* msg)
{
    GameMessageHandler* current = this;

    while (current)
    {
        if (current->canHandle(msg))
        {
            return current->process(msg);
        }
        current = current->next_.get();
    }

    return ResultType::UNHANDLED;
}
```

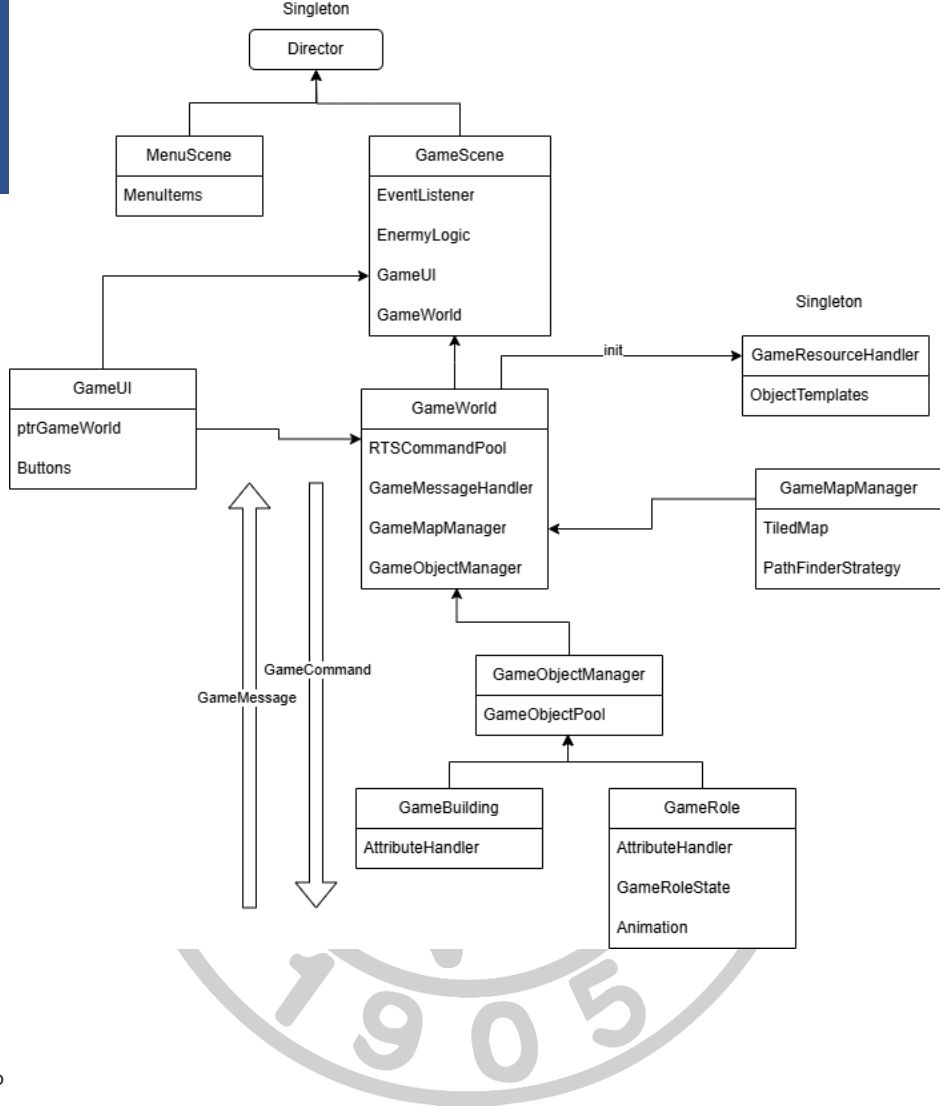
```
class GameMessageHandler
{
public:
    enum class ResultType
    {
        CONSUMED,
        UNHANDLED,
        REJECTED,
    };

public:
    virtual ~GameMessageHandler() = default;

    void setNext(std::shared_ptr<GameMessageHandler> next) { next_ = next; }

    ResultType handle(GameMessage* msg);

protected:
    virtual bool canHandle(GameMessage* msg) const = 0;
    virtual ResultType process(GameMessage* msg)   = 0;

private:
    std::shared_ptr<GameMessageHandler> next_;
};
```

原型模式(Prototype)

- 游戏会有大量重复GameObject对象，通过提前设置一个原型可以极大降低创建游戏对象的开销,在创建新对象时使用clone即可。
- 实际上这是与.plist制作结合的，需要提前将动画帧读取到内存中，并根据动画配置信息制作基本动画。

```
01  bool GameRole::init(GameObjectManager* manager,
02                      GameObjectType gameObjectType,
03                      CampType campType,
04                      const std::string& templateName,
05                      const ax::Vec2& position,
06                      int uniqueID)
07  {
08
09      if (!GameObject::init(manager, gameObjectType, campType, templateName, position, uniqueID))
10      {
11          return false;
12      }
13
14      auto& obj_template = GameResourceHandler::getInstance()->getObjTemplate(templateName);
15
16      h_atk_.init(obj_template);
17
18      anim_idle_   = RepeatForever::create(obj_template.anim_action_["idle"]->clone());
19      anim_idle_->retain();
20      anim_idle_->setTag(ActionTag::ANIMATION);
21
22      role_sprite_ = ax::Sprite::createWithSpriteFrame(obj_template.initial_sprite_frame_);
23
24      return true;
25  }
```

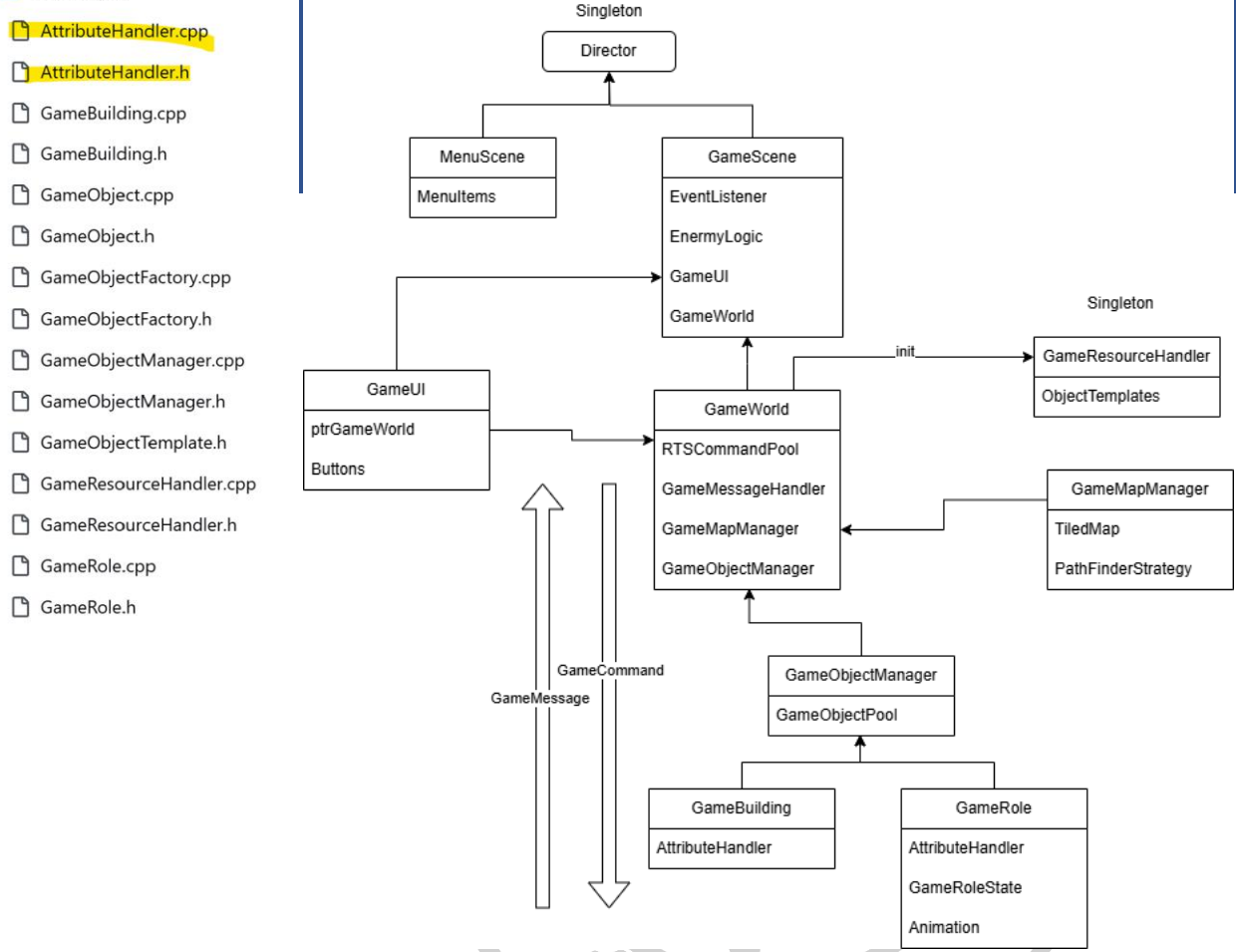




组合模式(Composite)

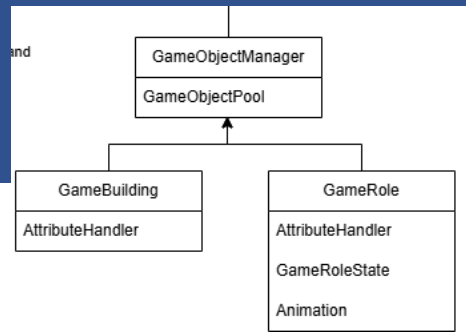
- 游戏对象属性的逻辑抽象与功能组合

```
01  class AttributeHandler_Hp
02  {
03  public:
04      void init(const GameObjectTemplate& objTemplate, const std::string& hpBarType);
05      void showHpBar();
06      void hideHpBar();
07      void reduceHp(int reduceHpAmount);
08      void addHp(int addHpAmount);
09      void updateHpBar();
10      bool isHpZero() { return hp_ == 0; }
11      ax::Node* getHpBar();
12
13  protected:
14      int hp_                     = 0;
15      int hp_max_                 = 0;
16      bool update_                = false;
17      ax::ui::LoadingBar* hp_bar_ = nullptr;
18  };
19
20  class AttributeHandler_Atk;
21  class AttributeHandler_Selectable;
22  class AttributeHandler_Movable;
```



- 大本营（生命）
- 角色（生命，攻击，选择，移动）
- 防御建筑（生命，攻击，选择）

状态模式(State)



- 随着游戏进行，一个游戏对象会有多种状态，比如战斗与空闲，行为上说就是不同状态下会有不同的基础逻辑与响应动作，那么这样就需要状态模式来介入，提供可扩展性与可维护性。
- 当然，为了进一步提升性能，避免过多子类，使用了静态状态，对象Object和上下文Context相当于每次作为参数提供。

```cpp
class GameRoleState
{
public:
    virtual ~GameRoleState() = default;

    virtual void update(GameRole* role);

    virtual void handleCommand(GameRole* role, GameCommand* cmd);
};

class GameRoleStateIdle : public GameRoleState
{
    DECLARE_SINGLETON(GameRoleStateIdle)
public:
    void update(GameRole* role) override;
    void handleCommand(GameRole* role, GameCommand* cmd) override;
};
```

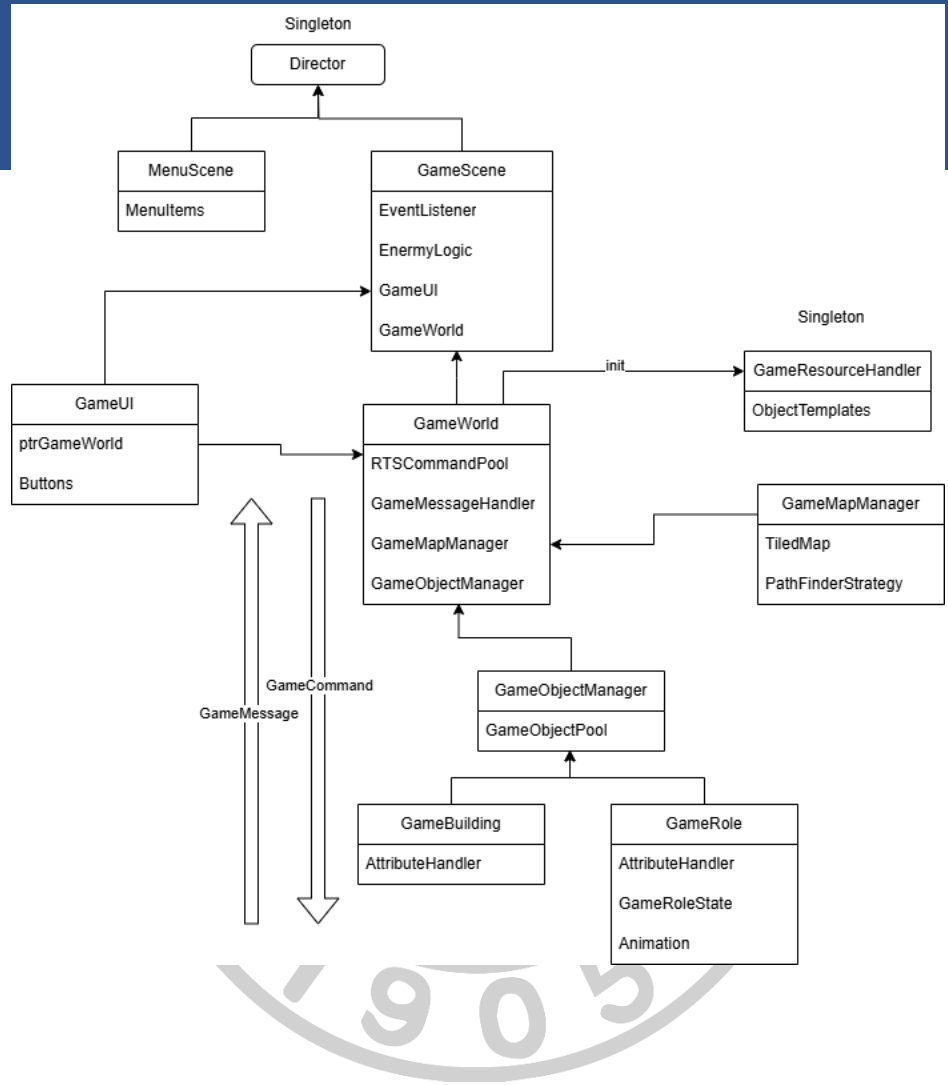
```cpp
class GameRole : public GameObject
{
public:
    void setState(GameRoleState* newState) { current_state_ = newState; }
    GameRoleState* getCurrentState() { return current_state_; }

    void update(float delta) override { current_state_->update(this); }
    void handleCommand(GameCommand* cmd) override { current_state_->handleCommand(this, cmd); }
};
```

```cpp
// 快速声明状态类
#define DECLARE_GAME_ROLE_STATE(StateName)                          \
    class StateName : public GameRoleState                          \
    {                                                               \
        DECLARE_SINGLETON(StateName)                                \
    public:                                                         \
        void update(GameRole* role) override;                       \
        void handleCommand(GameRole* role, GameCommand* cmd) override; \
    };

// DECLARE_GAME_ROLE_STATE(GameRoleStateIdle)
DECLARE_GAME_ROLE_STATE(GameRoleStateMove)
DECLARE_GAME_ROLE_STATE(GameRoleStateFight)
DECLARE_GAME_ROLE_STATE(GameRoleStateDead)

#undef DECLARE_GAME_ROLE_STATE
```

装饰器模式(Decoretor)

- 使用<functional>库，将函数作为对象传递
- 再使用lambda函数动态修改
- Sequence与Spawn组合角色动作（两种修饰符）

```
case CommandType::MOVE:
{
    auto command  = static_cast<CommandMove*>(cmd);
    auto data     = command->getData();
    auto strategy = data.strategy;
    auto action   = role->h_move_.getMoveAction(role->getPosition(), data.path);
    if (role->isMoveActionValid(strategy))
    {
        if (strategy == 0)
        {
            ax::Sequence* t = ax::Sequence::create(
                action, CallFunc::create([=]() { role->setState(GameRoleStateIdle::getInstance()); }), nullptr);
            role->moveAction(t, strategy);
            role->setState(GameRoleStateMove::getInstance());
        }
        else
        {
            role->moveAction(action, strategy);
        }
    }
}
break;
```



总结

总结

- 使用专业游戏开发引擎Axmol，基于c++开发游戏
- 了解游戏制作基本概念，使用工具Tiled与TexturePacker制作游戏素材
- 实现了RTS游戏的基本功能
- 特意使用了多种设计模式，使得项目可管理，上万行的代码成功运行
- 使用excel配置游戏数据/动画表现
- 专门为RTS游戏场景写了寻路算法以及高性能的内存池管理方案，多线程指令处理
- 游戏角色可玩家控制+自主决策，留出策略接口（通过command和message）可以接入更复杂行为逻辑

Thank You!