高级语言程序设计大作业实验报告

### 一.作业内容:

利用EasyX库完成一款2D小游戏，实现操作玩家的对象击败Boss取得胜利

### 二.开发软件

Visual Studio 2020

### 三.主要思路

本项目主要是基于“管理者模式”下的对游戏进行控制的思路。首先我们要明确的思想就是封装，同时要实现解耦合，达到不同功能的分离，防止出现当大量功能混杂在同一个函数时难以分割出需要的函数，这就需要大量的继承，虚函数，多态等知识点的灵活运用。实现对数据处理与图像绘制的解耦合也是相当重要的。

核心框架:通过资源管理器，碰撞管理器，人物管理器三大管理器对场面调度进行调控。

###### 1.资源管理器 Resources\_Manager

（1）资源预处理:

通过封装vector2,Animation,Atla，Timer等类以及单独封装的util.h文件，便于处理图像绘制与逻辑实现问题

vector2:实现二维向量的创建，重载运算符便于处理坐标关系

Timer:引入计时器的概念，利用“回调函数”(引入<functional>库)，实现更加便捷的对不同需求的满足。

Animation:用于处理图像绘制，便于处理“循环”与“非循环”类图像的处理，同时也提供了自定义的结构体Frames用于储存不同方向的图片

Atlas:利用“享元”模式，便于处理多帧的单张图片

util.h:util作为工具箱的存在，实现的功能较为杂乱，主要实现封装绘制图片函数+处理音频播放/停止+随机数函数封装。

（2）加载资源与处理图片

#pragma once

#include"atlas.h”

#include<string>

#include<graphics.h>

#include<unordered\_map>

class ResourcesManager {

public:

static ResourcesManager\* instance();

void load();

Atlas\* find\_atlas(const std::string& id)const;

IMAGE\* find\_image(const std::string& id)const;

private:

static ResourcesManager\* manager;

std::unordered\_map<std::string, Atlas\*>atlas\_pool;

std::unordered\_map<std::string, IMAGE\*>image\_pool;

public:

ResourcesManager();

~ResourcesManager();

void flip\_image(IMAGE\* src\_image, IMAGE\* dst\_image, int num\_h);

void flip\_image(IMAGE\* src\_image, IMAGE\* dst\_image, int x\_num, int y\_num);

void flip\_image(const std::string& src\_id, const std::string& dst\_id, int num\_h);

void flip\_image(const std::string& src\_id, const std::string& dst\_id, int x\_num, int y\_num);

void flip\_atlas(const std::string& src\_id, const std::string& dst\_id);

};

主要实现功能:

·通过引入“unordered\_map”头文件，实现通过键-值对应的操作，方便进行资源加载，增加对应资源池，通过直接使用对应资源池中图集名字获取对应的储存图片的变量，减少人为失误。

·通过静态函数与静态变量成员创建全局化的管理者与管理者实例的接口，实现对资源的调度

·重载四种不同的flip\_image用于处理不同类型的图集

·提供2个find函数用于通过图集名字查找对应图片，返回对应变量

###### 2.碰撞箱管理器 Collision\_Manager

(1)引入碰撞箱管理器需要先对碰撞箱进行定义,于是有了Collision\_box.h与Collision\_layer.

Collision\_layer:

enum class CollisionLayer { //用于描述某个碰撞箱自身所处层级

None,

Player,

Enemy,

};

该枚举型变量主要用于处理不同碰撞箱有不同的层级，例如玩家的受击碰撞箱和攻击碰撞箱

Collision\_box.h:

主要成员变量:bool enabled（碰撞箱是否有效），Stage stage（Stage为枚举型变量，表示碰撞箱的阶段），timer（各种计时器用于处理碰撞箱随时间变化，stage的改变），vector2 size（碰撞箱体积），vector2 position(碰撞箱位置)，CollisionLayer src\_layer与dst\_layer

解释:src\_layer用于描述自身的层级，dst\_layer用于描述该碰撞箱需要碰撞的层级

例如玩家自身的受击碰撞箱设置为:src\_layer:Player,dst\_layer:None

主要成员函数:

提供获取:get\_...() 获取碰撞箱的各个成员变量

提供修改:set\_...() 设置碰撞箱的各个成员变量

例如bool get\_enabled()const;void set\_position(const vector2& pos);

提供更新函数:void update\_size(),void update\_position(),void update\_stage()等用于更新碰撞箱的各个成员变量

(2)碰撞管理器

#pragma once

#include"collision\_box.h"

#include<vector>

class CollisionManager {

public:

static CollisionManager\* instance();

private:

CollisionBox\* attack1\_box = nullptr;

...(省略类似相同结构碰撞箱的指针初始化)

public:

CollisionBox\* get\_attack1\_box() {

return attack1\_box;

}

...(省略类似相同结构get\_attack\_box函数)

CollisionBox\* create\_collision\_box();

void destroy\_collision\_box(CollisionBox\* collision\_box);

void process\_collide();

void on\_debug\_render();

private:

static CollisionManager\* manager;

std::vector<CollisionBox\*>collision\_box\_list;

public:

CollisionManager();

~CollisionManager();

};

·采取与碰撞管理器一致思路，通过静态变量与静态函数，实现全局性的对碰撞箱的管理，提供get函数用于获取对于碰撞箱。同时提供create\_collision\_box()函数用于更好初始化碰撞箱，并pushback进入collision\_box\_list容器中统一调度管理(便于随时创建，与构造函数仅当碰撞管理器对象初始化时调用，会造成资源浪费，同时不利于释放无用的碰撞箱)，同时引入destroy\_collision\_box()与create碰撞箱效果呼应，当某个碰撞箱不再需要及时释放对应空间。

·核心逻辑:处理碰撞箱碰撞问题:

void CollisionManager::process\_collide() {

for (auto\* collision\_box\_src : collision\_box\_list) {

if (!collision\_box\_src->enabled || collision\_box\_src->layer\_dst == CollisionLayer::None) continue;

for (auto\* collision\_box\_dst : collision\_box\_list) {

if (!collision\_box\_dst || collision\_box\_src == collision\_box\_dst ||collision\_box\_src->layer\_dst != collision\_box\_dst->layer\_src) continue;

//矩形与矩形检测

bool is\_collide\_x =

(max(collision\_box\_src->position.x + collision\_box\_src->size.x / 2,

collision\_box\_dst->position.x+collision\_box\_dst->size.x/2)- min(collision\_box\_src->position.x-collision\_box\_src->size.x/ 2,

collision\_box\_dst->position.x-collision\_box\_dst->size.x/2)<=

collision\_box\_src->size.x + collision\_box\_dst->size.x);

bool is\_collide\_y =

(max(collision\_box\_src->position.y + collision\_box\_src->size.y / 2,

collision\_box\_dst->position.y + collision\_box\_dst->size.y / 2) - min(collision\_box\_src->position.y-collision\_box\_src->size.y/ 2,

collision\_box\_dst->position.y - collision\_box\_dst->size.y / 2) <= collision\_box\_src->size.y + collision\_box\_dst->size.y);

if (is\_collide\_x && is\_collide\_y && collision\_box\_dst->on\_collide) { collision\_box\_dst->on\_collide(collision\_box\_src->get\_damage());

}

}

}

}

通过碰撞层级layer的方法区分开玩家与敌人的碰撞级别，采取对场中所有未被删除的碰撞箱进行相互检测，由于为矩形碰撞箱，只需要对水平方向与竖直方向是否发生重叠，从而即可判断出矩形碰撞箱是否碰撞。

逻辑顺序：box\_src->enabled是无效的或box\_src->layer\_dst（目标碰撞箱）的层级为None不可被碰撞则跳过该box\_src,否则以该box\_src对所有除它自身之外的碰撞箱进行检测，如果box\_dst为空指针或box\_src与box\_dst为同一个碰撞箱或box\_src的目标层级layer\_dst与box\_dst的layer\_src不同或box\_dst的回调函数未被定义，则跳过该碰撞箱，继续检测其他碰撞箱。若有碰撞箱成功检测到满足条件且发生碰撞，则调用box\_dst的回调函数on\_collide()，实现碰撞效果。

·增加debug功能，对碰撞箱实现可视化，便于调试碰撞箱是否正确被检测与创建。

###### 3.角色管理器Character\_Manager

（1）Character，Player，Enemy类的实现

首先定义玩家与敌人的基类Character便于处理玩家与敌人相似的逻辑，例如update(更新)，render(渲染),位置获取与设置，图片的朝向，角色移动，受击碰撞箱以及图像资源等逻辑

继承Character实现具体Player与Enemy类，引入virtual虚函数便于超载，同时对单独的逻辑进行封装。例如实现玩家的移动需要读取操作，以及构造函数中对资源的载入。

为了让玩家与敌人能够正常的在不同状态下能够不断切换状态，例如从idle到run到attack再到idle这样一条路径，由于不存在过于复杂的逻辑，我们选择使用状态机的策略进行状态间的跳转。

（2）状态机 State\_Machine

状态机作为类似于管理器的存在，作为主导演，操控着玩家与敌人的状态间的跳转，状态的注册，状态的更新，是最核心的内容。

·引入State\_node作为类的模板，引入虚函数

class StateNode {

public:

StateNode() = default;

~StateNode() = default;

virtual void on\_enter() {}

virtual void on\_update(float delta) {}

virtual void on\_exit() {}

};

提供状态的进入，更新与退出三大功能

·引入state\_node.h头文件，对状态机进行封装

#include"state\_node.h"

#include<string>

#include<graphics.h>

#include<unordered\_map>

class StateMachine {

public:

StateMachine();

~StateMachine();

void on\_update(float delta);

void set\_entry(const std::string& id);

void switch\_to(const std::string& id);

void register\_state(const std::string& id, StateNode\* state\_node);

private:

bool need\_init = true;

StateNode\* current\_state = nullptr;

std::unordered\_map<std::string, StateNode\*>state\_pool;

};

i.同样引入<unordered\_map>头文件，便于处理不同状态间的跳转，引入状态池的概念，使得不同状态间只会在状态池中的状态进行相互跳转。

ii.提供on\_update（float delta）函数，进行统一的状态更新，同时加入set\_entry实现初始状态的初始化

iii.提供跳转(switch\_to)与注册(register\_state)功能

switch\_to是所有状态的核心

void StateMachine::switch\_to(const std::string& id) {

if (current\_state)current\_state->on\_exit();

current\_state = state\_pool[id];

if (current\_state)current\_state->on\_enter();

}

其逻辑为先退出当前状态，然后进入下一个状态，on\_exit()与on\_enter()均会在switch\_to的步骤进行执行

·玩家状态节点

以Idle\_state为例进行分析

player\_state\_nodes.h文件:

class PlayerIdleState :public StateNode {

public:

PlayerIdleState() = default;

~PlayerIdleState() = default;

void on\_enter()override;

void on\_update(float delta) override;

};

player\_state\_nodes.cpp文件:

void PlayerIdleState::on\_enter() {

CharacterManager::instance()->get\_player()->set\_animation("idle");

}

void PlayerIdleState::on\_update(float delta) {

Player\* player = (Player\*)CharacterManager::instance()->get\_player();

if (player->get\_hp() <= 0) {

player->switch\_state("dead");

}

else if (player->can\_attack1()) {

player->switch\_state("attack1");

}

else if (player->can\_attack2()) {

player->switch\_state("attack2");

}

else if (player->can\_attack3()) {

player->switch\_state("attack3");

}

else if (player->get\_velocity().y > 0) {

player->switch\_state("fall");

}

else if (player->can\_jump()) {

player->switch\_state("jump");

}

else if (player->is\_on\_floor() && player->get\_move\_axis() != 0) {

player->switch\_state("run");

}

}

在进入该状态时，设置idle对应图像，然后进入update环节，检测玩家是否出现任何满足对应跳转的条件，然后进行跳转。

·敌人状态节点

以Attack1\_state为例进行分析:

enemy\_state\_nodes.h文件:

class EnemyAttack1State :public StateNode {

public:

EnemyAttack1State();

~EnemyAttack1State() = default;

void on\_enter()override;

void on\_update(float delta) override;

void on\_exit() override;

private:

Timer timer;

float attack\_speed = 250.0f;

float attack\_dis = 300.0f;

float sum\_delta\_time = 0;

};

enemy\_state\_nodes.cpp文件:

EnemyAttack1State::EnemyAttack1State() {

timer.set\_wait\_time(0.40f);

timer.set\_one\_shot(true);

timer.set\_on\_timeout([&] {

Enemy\* enemy = (Enemy\*)CharacterManager::instance()->get\_enemy();

enemy->set\_attack1(false);

enemy->set\_attacking(false);

enemy->set\_changeDirection(true);

});

}

执行构造函数进行对计时器的初始化，完成计时器所需要计时时长，是否循环，以及对回调函数进行定义，当计时器到达规定时间后，执行对应的回调匿名函数。

void EnemyAttack1State::on\_enter() {

CharacterManager::instance()->get\_enemy()->set\_animation("enemy\_attack1");

Enemy\* enemy = (Enemy\*)CharacterManager::instance()->get\_enemy();

enemy->set\_changeDirection(false);

enemy->set\_attack1(true);

enemy->set\_attacking(true);

enemy->on\_attack1();

CollisionBox\* box = CollisionManager::instance()->get\_enemy\_attack1\_box();

box->set\_enabled(true);

box->set\_stage(CollisionBox::Stage::one);

timer.restart();

box->get\_timer\_stage1\_to\_2()->restart();

box->get\_timer\_stage2\_to\_3()->restart();

switch (range\_random(1, 3)) {

case 1:

play\_audio(\_T("player\_attack\_1"), false);

break;

case 2:

play\_audio(\_T("player\_attack\_2"), false);

break;

case 3:

play\_audio(\_T("player\_attack\_3"), false);

break;

}

}

进入状态主要是对一些敌人的成员变量进行初始化，同时开始计时器

void EnemyAttack1State::on\_update(float delta) {

timer.on\_update(delta);

Enemy\* enemy = (Enemy\*)CharacterManager::instance()->get\_enemy();

const vector2& pos\_enemy = enemy->get\_position();

const vector2& pos\_player = CharacterManager::instance()->get\_player()->get\_position();

float d = abs(pos\_enemy.x - pos\_player.x);

enemy->set\_velocity({ (pos\_enemy.x < pos\_player.x ? attack\_speed : -attack\_speed) \* (max(min(d / attack\_dis,1),0.5f)),0 });

CollisionBox\* box = CollisionManager::instance()->get\_enemy\_attack1\_box();

box->update\_enemy\_attack1\_box\_size();

box->update\_enemy\_attack1\_box\_position(enemy->get\_logic\_center(), enemy->get\_is\_facing\_left());

box->get\_timer\_stage1\_to\_2()->on\_update(delta);

if (box->get\_stage() == CollisionBox::Stage::two)

box->get\_timer\_stage2\_to\_3()->on\_update(delta);

if (enemy->get\_hp() <= 0) {

enemy->switch\_state("enemy\_dead");

}

else if (!enemy->get\_attacking()) {

if (enemy->get\_velocity().y > 0) {

enemy->switch\_state("enemy\_fall");

}

else if (enemy->is\_on\_floor()) {

enemy->switch\_state("enemy\_idle");

}

}

}

核心逻辑在于状态更新，根据条件更新 “状态timer”，“碰撞箱timer”，

碰撞箱大小，位置更新，以及stage阶段更新。最后考虑是否满足任何可以使得该状态跳转的逻辑，除非enemy血量降为0，否则只有当该攻击状态结束才会有!is\_attacking，才可能跳转到其他状态，防止不同状态相互叠加出现混乱

void EnemyAttack1State::on\_exit() {

Enemy\* enemy = (Enemy\*)CharacterManager::instance()->get\_enemy();

enemy->set\_attack1(false);

enemy->set\_attacking(false);

enemy->set\_velocity({ 0, enemy->get\_velocity().y });

}

将敌人的各种变量恢复进入前时的状态

四.收获

通过本次大作业，初步认识到游戏设计/程序设计中的各种思想，例如工厂思想，管理者模式，建造者模式，认识到解耦合，封装等思想的重要性，同时对于各种小细节，例如文件处理，摒弃全局变量，命名不符合规范等不好的习惯。

但同时依旧有部分问题仍需解决，例如优化碰撞箱繁杂的重复的代码，增添更多更”智能”的设计让角色AI的实现更加完善。