=================================“Additional.h” =================================

#pragma once

namespace game{

enum status {

PLAY,

PAUSED,

WIN,

GAME\_OVER,

RESTART,

MAIN\_MENU

};

}

namespace elements {

const size\_t SKILL\_ELEMENT\_AMOUNT = 3;

enum element {

NONE = 0,

EARTH = 8,

FIRE = 2,

WIND = 1

};

}

namespace tiles {

const unsigned int WIDTH\_SIZE\_POSITION\_IN\_STR = 9U;

const unsigned int HEIGHT\_SIZE\_POSITION\_IN\_STR = 6U;

const int STD\_SIZES\_LENGTH = 2;

struct sizes {

int width;

int height;

};

static sizes getSizesFromStr(std::string str, int widthStrPosCorrection = 2, int heightStrPosCorrection = 2) {

sizes temp;

size\_t length = str.size();

std::string width = str.substr(length - WIDTH\_SIZE\_POSITION\_IN\_STR + STD\_SIZES\_LENGTH - widthStrPosCorrection + STD\_SIZES\_LENGTH - heightStrPosCorrection,widthStrPosCorrection);

std::string height = str.substr(length - HEIGHT\_SIZE\_POSITION\_IN\_STR + STD\_SIZES\_LENGTH - heightStrPosCorrection,heightStrPosCorrection);

temp.width = atoi(width.c\_str());

temp.height = atoi(height.c\_str());

return temp;

}

const int size = 32;

const int SPRITE\_HEIGHT = 32;

const int SPRITE\_WIDTH = 32;

const int TEXTURE\_TILE\_AMOUNT\_X = 21;

const int TEXTURE\_TILE\_AMOUNT\_Y = 30;

const int TEXTURE\_FILE\_WIDHT = 672;

const int TEXTURE\_FILE\_HEIGHT = 960;

const int OB\_TEXTURE\_VALUE = 400;

}

namespace animation {

const std::string DEFAULT\_TEXTURE = "img/default.png";

//BULLETS IMGS

const std::string SWORD\_ATTACK\_TEXTURE\_FILE = "img/skills/bullet\_sword.png";

const std::string BULLET\_SMALL\_FIRE\_BALLS\_TEXTURE\_FILE = "img/skills/bullet\_small\_fire\_balls.png";

const std::string BULLET\_SMALL\_EARTH\_BALL\_TEXTURE\_FILE = "img/skills/bullet\_small\_earth\_balls.png";

const std::string BULLET\_FIRE\_BALL\_TEXTURE\_FILE = "img/skills/bullet\_fire\_ball.png";

const std::string BULLET\_EARTH\_BALL\_TEXTURE\_FILE = "img/skills/bullet\_earth\_ball.png";

const std::string BULLET\_LAVA\_TEXTURE\_FILE = "img/skills/bullet\_lava\_pool.png";

const std::string BULLET\_BANG\_BALL\_TEXTURE\_FILE = "img/skills/bullet\_bang\_ball.png";

const std::string BULLET\_EARTH\_SLAM\_TEXTURE\_FILE = "img/skills/bullet\_earth\_slam.png";

const std::string BULLET\_COMBO\_BALL\_TEXTURE\_FILE = "img/skills/bullet\_combo.png";

const std::string BULLET\_CRYSTAL\_HEAL\_TEXTURE\_FILE = "img/skills/bullet\_crystal\_heal.png";

const std::string BULLET\_FIRE\_LIGHTING\_TEXTURE\_FILE = "img/skills/bullet\_fire\_lighting.png";

const std::string BULLET\_POISION\_BALL\_TEXTURE\_FILE = "img/skills/bullet\_poision\_splash.png";

//CHARACTER IMGS

const std::string MAIN\_HERO\_TEXTURE\_FILE = "img/characters/mainHero\_32\_32.png";

const std::string MAIN\_HERO\_TEXTURE\_FILE2 = "img/characters/mainHero(2)\_32\_32.png";

const std::string ENEMY\_DEMON\_FILE = "img/characters/enemy\_demon\_32\_32.png";

const std::string ENEMY\_SLENDER\_DEMON\_FILE = "img/characters/enemy\_slenderDemon\_32\_32.png";

const std::string ENEMY\_WARRIOR\_FILE = "img/characters/enemy\_warrior\_32\_32.png";

const std::string ENEMY\_MAGE\_FILE = "img/characters/enemy\_mage\_32\_32.png";

//const std::string ENEMY\_ \_TEXURE\_FILE = "img/characters/.png";

const std::string ENEMY\_ZOMBIE\_PUDGE\_TEXURE\_FILE = "img/characters/enemy\_zombie\_pudge\_64\_64.png";

const std::string ENEMY\_SKELETON\_MAGE\_TEXTURE\_FILE = "img/characters/enemy\_skeleton\_mage\_32\_48.png";

const std::string ENEMY\_ZOMBIE\_WITCH\_TEXTURE\_FILE = "img/characters/enemy\_zombie\_witch\_32\_32.png";

const std::string ENEMY\_SOMBIE\_WARRIOR\_TEXTURE\_FILE = "img/characters/enemy\_zombie\_warrior\_32\_32.png";

const std::string BOSS\_FINALY\_DEMON\_TEXURE\_FILE = "img/characters/BOSSES/BOSS\_FINALY\_96\_48.png";

const std::string BOSS\_TREANT\_TEXURE\_FILE = "img/characters/BOSSES/BOSS\_TREANT\_96\_96.png";

const std::string BOSS\_BLACK\_DRAGON\_TEXURE\_FILE = "img/characters/BOSSES/BOSS\_BLACK\_DRAGON\_96\_96.png";

const std::string BOSS\_RED\_DRAGON\_TEXURE\_FILE = "img/characters/BOSSES/BOSS\_RED\_DRAGON\_192\_192.png";

const std::string BOSS\_ENH\_TEXURE\_FILE = "img/characters/BOSSES/BOSS\_ENH\_240\_80.png";

const int COMMON\_CHARACTER\_SPRITE = 32;

const int MAIN\_HERO\_SPRITE\_HEIGHT = 32;

const int SPRITE\_X = 0;

const int SPRITE\_Y = 0;

const int LARGE\_SKILL\_WIDTH = 64;

const int LARGE\_SKILL\_HEIGHT = 64;

const float frameRate = 3.f;

const float frameSpeed = 0.05f;

const float HEROES\_SPRITE\_COLLISION\_CORRECTION\_BORDER = 10.f;

enum direction {

BOTTOM = 0,

LEFT = 1,

RIGHT = 2,

TOP = 3,

};

}

namespace textSettings {

const std::string STD\_FONT\_FILE = "fonts/main\_font\_regular.otf";

const std::string STRIKE\_THROUGHT\_FONT\_FILE = "fonts/main\_strike\_throught\_font.ttf";

const size\_t STD\_FONT\_SIZE = 14;

const sf::Color STD\_TEXT\_COLOR = sf::Color::White;

const sf::Text::Style STD\_TEXT\_STYLE = sf::Text::Style::Regular;

const sf::Text::Style TEXT\_STYLE\_STRIKE\_THROUGH = sf::Text::Style::StrikeThrough;

};

namespace buttons {

const std::string JOURNAL\_KEY\_NAME = "journal";

const std::string GAME\_STATS\_KEY\_NAME = "gameStats";

const std::string MAP\_KEY\_NAME = "map";

const std::string SKILL\_KEY\_NAME = "skill";

}

namespace interface {

const sf::Vector2f STD\_HP\_BAR\_HEIGHT(50.f, 10.f);

const sf::Vector2f STD\_BAR\_SIZE(400.f, 20.f);

const sf::Vector2f STD\_BORDER\_SIZE(2.f, 2.f);

const sf::Vector2f STD\_WINDOW\_SIZE(48.f, 48.f);

const sf::Vector2f STD\_MARGIN\_SIZE(5.f, 5.f);

const sf::Vector2f STD\_WINDOW\_MARGIN\_SIZE(15.f, 15.f);

const sf::Vector2f STD\_SKILL\_WINDOW\_SIZE(64.f, 64.f);

const sf::Vector2f STD\_ELEMENT\_GENERATOR\_WINDOW\_SIZE(48.f, 48.f);

const std::string WIN\_CREEN\_TEXTURE\_FILE = "img/menu/winMenu.jpg";

static sf::Vector2f getScreenCoords(sf::RenderWindow \*window) {

return sf::Vector2f(window->getView().getCenter().x - window->getSize().x / 2, window->getView().getCenter().y - window->getSize().y / 2);

}

};

template <typename T> T getRand(T a, T b) {

T temp = static\_cast<T>(static\_cast<int>(a) + (rand() % (static\_cast<int>(b - a))));

return temp;

}

static bool timeIsOver(sf::Clock \*clock, sf::Int32 startTime, sf::Int32 \_time) {

sf::Int32 curTime = clock->getElapsedTime().asMilliseconds();

return (abs(curTime - startTime) > \_time) ? true : false;

}

static sf::Vector2f generateRandomSpawnCoords(const sf::Vector2i mapSize) {

sf::Vector2f vect;

vect.x = static\_cast<float>(rand() % mapSize.x);

vect.y = static\_cast<float>(rand() % mapSize.y);

return vect;

}

static size\_t findBigestLength(std::vector<std::string> list) {

size\_t maxLength = 0;

for (auto &i : list) {

maxLength = (i.size() > maxLength) ? i.size() : maxLength;

}

return maxLength;

}

================================= “BossNpc\_t.h” =================================

#pragma once

#include "Npc\_t.h"

#include <initializer\_list>

class BossNpc\_t :

public Npc\_t

{protected:

std::vector<size\_t> elementStatusList; //skills

public:

BossNpc\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple = 3.f);

BossNpc\_t(character\_t \*copyedNpc, sf::Vector2f spotPoint, float powerMultiple);

virtual ~BossNpc\_t();

virtual void setSkills(std::initializer\_list<size\_t> \_skillList);

virtual void BossNpc\_t::setSkills(std::vector<size\_t> \_skillList);

virtual std::vector<size\_t> getSkillList();

virtual void update(float \_speed);

virtual void attack();

virtual void generateSkillAndClearElemList();

virtual size\_t setElemStatus(size\_t \_elemStatus);

};

class TeantBossNpc\_t : public BossNpc\_t {

public:

TeantBossNpc\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple = 3.f);

TeantBossNpc\_t(character\_t \*copyedNpc, sf::Vector2f spotPoint, float powerMultiple);

virtual ~TeantBossNpc\_t();

};

class RedDragonBossNpc\_t : public BossNpc\_t {

public:

RedDragonBossNpc\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple = 3.f);

RedDragonBossNpc\_t(character\_t \*copyedNpc, sf::Vector2f spotPoint, float powerMultiple);

virtual ~RedDragonBossNpc\_t();

};

class BlackDragonBossNpc\_t : public BossNpc\_t {

public:

BlackDragonBossNpc\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple = 3.f);

BlackDragonBossNpc\_t(character\_t \*copyedNpc, sf::Vector2f spotPoint, float powerMultiple);

virtual ~BlackDragonBossNpc\_t();

};

class EnhBpssNpc\_t : public BossNpc\_t {

public:

EnhBpssNpc\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple = 3.f);

EnhBpssNpc\_t(character\_t \*copyedNpc, sf::Vector2f spotPoint, float powerMultiple);

virtual ~EnhBpssNpc\_t();

};

class DQBossNpc\_t : public BossNpc\_t {

public:

DQBossNpc\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple = 3.f);

DQBossNpc\_t(character\_t \*copyedNpc, sf::Vector2f spotPoint, float powerMultiple);

virtual ~DQBossNpc\_t();

};

================================= “BossNpc\_t.cpp” =================================

#include "BossNpc\_t.h"

BossNpc\_t::BossNpc\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple):

Npc\_t(\_texture, \_bulletList, \_clock, \_spotCoords, \_width, \_height, \_multiple)

{

}

BossNpc\_t::BossNpc\_t(character\_t \*copyedNpc, sf::Vector2f spotPoint, float powerMultiple) : Npc\_t(copyedNpc,spotPoint,powerMultiple) {

setSkills(copyedNpc->getSkillList());

}

BossNpc\_t::~BossNpc\_t()

{

}

void BossNpc\_t::setSkills(std::initializer\_list<size\_t> \_skillList) {

elementStatusList.insert(elementStatusList.end(), \_skillList.begin(), \_skillList.end());

}

void BossNpc\_t::setSkills(std::vector<size\_t> \_skillList) {

elementStatusList.insert(elementStatusList.end(), \_skillList.begin(), \_skillList.end());

}

std::vector<size\_t> BossNpc\_t::getSkillList() {

return elementStatusList;

}

void BossNpc\_t::update(float \_speed) {

checkAlive();

if (alive) {

physOb\_t::update(\_speed);

if (buff.get()) {

buff->checkActivity();

}

}

generateSkillAndClearElemList();

}

void BossNpc\_t::attack() {

Npc\_t::attack();

}

void BossNpc\_t::generateSkillAndClearElemList() {

size\_t skillRandomIndex = rand() % elementStatusList.size();;

elemStatus = elementStatusList[skillRandomIndex];

}

size\_t BossNpc\_t::setElemStatus(size\_t \_elemStatus) {

return elemStatus;

}

//TREANT

TeantBossNpc\_t::TeantBossNpc\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple) :

BossNpc\_t(\_texture, \_bulletList, \_clock, \_spotCoords, \_width, \_height, \_multiple)

{

setSkills({ 3,6,24,10,18,2 });

stat.stdattackPower = 50.f;

stat.attackRange = 150.f;

stat.stdPhysDef = 40.f;

stat.stdMagDef = 25.f;

stat.damageRand = 15.f;

stat.stdHP = 550.f;

stat.visionDistance = 300.f;

stat.stdAttackSpeed = 5.f;

stat.stdSpeed = 0.2f;

defaultAllStats();Npc\_t::setTypeStats();

}

TeantBossNpc\_t::TeantBossNpc\_t(character\_t \*copyedNpc, sf::Vector2f spotPoint, float powerMultiple) : BossNpc\_t(copyedNpc, spotPoint, powerMultiple) {

setSkills(copyedNpc->getSkillList());

}

TeantBossNpc\_t::~TeantBossNpc\_t()

{

}

//RED DRAGON

RedDragonBossNpc\_t::RedDragonBossNpc\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple) :

BossNpc\_t(\_texture, \_bulletList, \_clock, \_spotCoords, \_width, \_height, \_multiple)

{

setSkills({ 12,18,5,4 });

stat.stdattackPower = 60.f;

stat.attackRange = 300.f;

stat.stdPhysDef = 30.f;

stat.stdMagDef = 30.f;

stat.damageRand = 15.f;

stat.stdHP = 450.f;

stat.visionDistance = 300.f;

stat.stdAttackSpeed = 3.f;

stat.stdSpeed = 0.15f;

defaultAllStats();Npc\_t::setTypeStats();

}

RedDragonBossNpc\_t::RedDragonBossNpc\_t(character\_t \*copyedNpc, sf::Vector2f spotPoint, float powerMultiple) : BossNpc\_t(copyedNpc, spotPoint, powerMultiple) {

setSkills(copyedNpc->getSkillList());

}

RedDragonBossNpc\_t::~RedDragonBossNpc\_t()

{

}

//BLACK DRAGON

BlackDragonBossNpc\_t::BlackDragonBossNpc\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple) :

BossNpc\_t(\_texture, \_bulletList, \_clock, \_spotCoords, \_width, \_height, \_multiple)

{

setSkills({ 24,4,17,18,2 });

stat.stdattackPower = 70.f;

stat.attackRange = 200.f;

stat.stdPhysDef = 30.f;

stat.stdMagDef = 20.f;

stat.damageRand = 15.f;

stat.stdHP = 550.f;

stat.visionDistance = 320.f;

stat.stdAttackSpeed = 4.f;

stat.stdSpeed = 0.12f;

defaultAllStats();Npc\_t::setTypeStats();

}

BlackDragonBossNpc\_t::BlackDragonBossNpc\_t(character\_t \*copyedNpc, sf::Vector2f spotPoint, float powerMultiple) : BossNpc\_t(copyedNpc, spotPoint, powerMultiple) {

setSkills(copyedNpc->getSkillList());

}

BlackDragonBossNpc\_t::~BlackDragonBossNpc\_t()

{

}

//ENH

EnhBpssNpc\_t::EnhBpssNpc\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple) :

BossNpc\_t(\_texture, \_bulletList, \_clock, \_spotCoords, \_width, \_height, \_multiple)

{

setSkills({ 1,3,6,2,11,5,12 });

stat.stdattackPower = 90.f;

stat.attackRange = 300.f;

stat.stdPhysDef = 20.f;

stat.stdMagDef = 50.f;

stat.damageRand = 10.f;

stat.stdHP = 650.f;

stat.visionDistance = 250.f;

stat.stdAttackSpeed = 4.f;

stat.stdSpeed = 0.2f;

defaultAllStats();Npc\_t::setTypeStats();

}

EnhBpssNpc\_t::EnhBpssNpc\_t(character\_t \*copyedNpc, sf::Vector2f spotPoint, float powerMultiple) : BossNpc\_t(copyedNpc, spotPoint, powerMultiple) {

setSkills(copyedNpc->getSkillList());

}

EnhBpssNpc\_t::~EnhBpssNpc\_t()

{

}

//DARK QUEEN

DQBossNpc\_t::DQBossNpc\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple) :

BossNpc\_t(\_texture, \_bulletList, \_clock, \_spotCoords, \_width, \_height, \_multiple)

{

setSkills({ 6,2,11,5,12,4,7 });

stat.stdattackPower = 100.f;

stat.attackRange = 350.f;

stat.stdPhysDef = 40.f;

stat.stdMagDef = 40.f;

stat.damageRand = 30.f;

stat.stdHP = 500.f;

stat.visionDistance = 400.f;

stat.stdAttackSpeed = 2.f;

stat.stdSpeed = 0.15f;

defaultAllStats();Npc\_t::setTypeStats();

}

DQBossNpc\_t::DQBossNpc\_t(character\_t \*copyedNpc, sf::Vector2f spotPoint, float powerMultiple) : BossNpc\_t(copyedNpc, spotPoint, powerMultiple) {

setSkills(copyedNpc->getSkillList());

}

DQBossNpc\_t::~DQBossNpc\_t()

{

}

================================= “bullet\_t.h” =================================

#pragma once

#include <SFML\Graphics.hpp>

#include "physOb\_t.h"

#include "character\_t.h"

#include <list>

#include "additional.h"

struct bulletStats {

float speed;

float damage;

float range;

elements::element element = elements::NONE;

float AOE = 0.1f;

bool type = true;//0 - heal, 1 - dmg

int fraction = -1;

float mpCost = 10.f;

};

class dmgInterval\_t {

public:

physOb\_t &ob;

sf::Int32 startDmgTime;

dmgInterval\_t(physOb\_t &\_ob);

~dmgInterval\_t();

};

class character\_t;

class bullet\_t :public physOb\_t

{

private:

sf::Clock \*clock;

bulletStats stat;

character\_t \*genericObject;

std::list<dmgInterval\_t> obList;

sf::Int32 startTime;

sf::Int32 timer; //live time as Milliseconds

sf::Int32 dmgInterval;

bool mass;

float startPosX;

float startPosY;

sf::Vector2f targetPos;

float vectorLength;

void updateFrame();

void animation();

dmgInterval\_t &checkObInList(physOb\_t &Object);

public:

bullet\_t();

bullet\_t(sf::Clock \*time, character\_t \*genObj, sf::Vector2f \_targetPos);

virtual ~bullet\_t();

virtual bool checkAlive();

virtual void update(float speed);

virtual bool collisionHandler(physOb\_t &Object, float \_speed, float \_borderError = 0.f);

bool hitting(physOb\_t &Object, float \_speed, float \_borderError = 0.f);

public:

//GET

character\_t \* getGenericObject()const;

sf::Int32 getStartTime() const;

bulletStats getStats();

int getElement()const;

sf::Int32 getTimer()const;

float getDmg()const;

float getRng()const;

float getAOE() const;

bool checkAOE() const;

float getMPCost() const;

//SET

int setElement(elements::element \_element);

bool setStats(bulletStats &\_stats);

void setTimer(sf::Int32 \_timer);

void setDmg(float \_val);

void setSpeed(float \_val);

void speedMultiple(float \_val);

void setRng(float \_val);

void setAOE(float \_val);

void setType(bool \_val);

void setDmgDelay(sf::Int32 \_time);

void setMPCost(float \_val);

};

================================= “bullet\_t.cpp” =================================

#include "bullet\_t.h"

dmgInterval\_t::dmgInterval\_t(physOb\_t &\_ob):ob (\_ob){

startDmgTime = 0;

}

dmgInterval\_t::~dmgInterval\_t() {

}

bullet\_t::bullet\_t() :physOb\_t()

{

}

bullet\_t::bullet\_t(sf::Clock \*time, character\_t \*genObj, sf::Vector2f \_targetPos) : physOb\_t(genObj->getPosOfCenter().x, genObj->getPosOfCenter().y) {

genericObject = genObj;

fraction = genObj->getFraction();

clock = time;

sf::Int32 \_startTime = clock->getElapsedTime().asMilliseconds();

startTime = \_startTime;

timer = 4000;

targetPos = \_targetPos;

stat.range = genericObject->getStats().attackRange;

stat.damage = genericObject->getStats().attackPower;

float speed = 15.f;

stat.speed = speed / stat.range;

stat.AOE = 0.f;

stat.element = elements::NONE;

stat.type = true;

if (abs(stat.AOE) < FLT\_EPSILON) {

mass = false;

}

else {

mass = true;

}

dmgInterval = timer;

startPosX = posX;

startPosY = posY;

//CALC SPEED

float distanceX = targetPos.x - posX;

float distanceY = targetPos.y - posY;

float rotation = (atan2(distanceY, distanceX)) \* 180.f / 3.14159265f;

spritePref.setOriginToCenter();

spritePref.setRotation(rotation);

vectorLength = sqrt(pow(distanceX, 2) + pow(distanceY, 2)) + spritePref.getTexture().getSize().x;

//CALC DISTANCE SPEED ERROR

float k = vectorLength / stat.range;

distanceX /= k \* 100;

distanceY /= k \* 100;

//SET SPEED

dX = distanceX \* stat.speed;

dY = distanceY \* stat.speed;

}

bullet\_t::~bullet\_t()

{

}

void bullet\_t::update(float \_speed) {

if (alive) {

checkAlive();

posX += dX \* \_speed;

posY += dY \* \_speed;

updateFrame();

animation();

spritePref.setSpritePos(posX, posY);

}

}

bool bullet\_t::checkAlive() {

if (alive) {

float distanceX = startPosX - posX;

float distanceY = startPosY - posY;

float tempVectorLength = sqrt(pow(distanceX, 2) + pow(distanceY, 2));

float minRange = 0.5;

alive = (checkTimer(clock, startTime, timer) && (tempVectorLength < stat.range));//time is over or leave from range // abs(tempVectorLength - stat.range) < (minRange))

}

return alive;

}

bool bullet\_t::hitting(physOb\_t &Object, float \_speed, float \_borderError) {

if (stat.type && Object.getFraction() != fraction) {

Object.takeDamage(stat.damage, stat.type, stat.element);

if (collision) {

alive = false;

}

}

else if (!stat.type && Object.getFraction() == fraction) {

Object.takeHeal(stat.damage);

}

return true;

}

bool bullet\_t::collisionHandler(physOb\_t &Object, float \_speed, float \_borderError) {

dmgInterval\_t &temp = checkObInList(Object);//if ob exist startDmgTime != 0

if (temp.startDmgTime) {

if (!checkTimer(clock, temp.startDmgTime, dmgInterval)){

temp.startDmgTime = clock->getElapsedTime().asMilliseconds();

hitting(Object, \_speed, \_borderError);

}

} else {

if (Object.getDestroyble()) {

temp.startDmgTime = clock->getElapsedTime().asMilliseconds();

obList.push\_back(temp);

hitting(Object, \_speed, \_borderError);

return true;

}

}

return false;

}

void bullet\_t::updateFrame() {

float AnimationSpeedMultiple = 3.f;

frame += animation::frameSpeed \* AnimationSpeedMultiple;

if (frame > animation::frameRate) {

frame -= animation::frameRate;

}

}

void bullet\_t::animation() {

int spriteCoordX = static\_cast<int>(frame) \* getWidth();

spritePref.setTexturePos(spriteCoordX, spritePref.getCoordY());

}

dmgInterval\_t &bullet\_t::checkObInList(physOb\_t &Object) {

dmgInterval\_t temp(Object);

for (auto &i : obList) {

if (Object.getPtr() == i.ob.getPtr()) {

return i;

}

}

return temp;

}

//GET

character\_t \*bullet\_t::getGenericObject()const {

return genericObject;

}

sf::Int32 bullet\_t::getStartTime() const {

return startTime;

}

bulletStats bullet\_t::getStats() {

return stat;

}

int bullet\_t::getElement()const {

return stat.element;

}

sf::Int32 bullet\_t::getTimer()const {

return timer;

}

float bullet\_t::getDmg()const {

return stat.damage;

}

float bullet\_t::getRng()const {

return stat.range;

}

float bullet\_t::getAOE() const {

return stat.AOE;

}

bool bullet\_t::checkAOE() const {

return mass;

}

float bullet\_t::getMPCost() const {

return stat.mpCost;

}

//SET

int bullet\_t::setElement(elements::element \_element){

stat.element = \_element;

return stat.element;

}

bool bullet\_t::setStats(bulletStats &\_stats) {

mass = (\_stats.AOE > FLT\_EPSILON) ? true : false;

stat = \_stats;

return mass;

}

void bullet\_t::setTimer(sf::Int32 \_timer) {

timer = \_timer;

}

void bullet\_t::setDmg(float \_val) {

stat.damage = \_val;

}

void bullet\_t::setSpeed(float \_val) {

stat.speed = \_val;

}

void bullet\_t::speedMultiple(float \_val) {

stat.speed \*= \_val;

dX \*= \_val;

dY \*= \_val;

}

void bullet\_t::setRng(float \_val) {

stat.range = \_val;

}

void bullet\_t::setAOE(float \_val) {

mass = (\_val > FLT\_EPSILON) ? true : false;

stat.AOE = \_val;

}

void bullet\_t::setType(bool \_val) {

stat.type = \_val;

}

void bullet\_t::setDmgDelay(sf::Int32 \_time) {

dmgInterval = \_time;

}

void bullet\_t::setMPCost(float \_val) {

stat.mpCost = \_val;

}

================================= “Character\_t.h” =================================

#pragma once

#include <SFML\Graphics.hpp>

#include <list>

#include <numeric>

#include "characterStats\_t.h"

#include "CharacterState\_t.h"

#include "characterTimers\_t.h"

#include "physOb\_t.h"

#include "additional.h"

#include "skillObGenerator\_t.h"

#include "Effect\_t.h"

class CharacterState\_t;

class skillObGenerator\_t;

class Effect\_t;

class character\_t : public physOb\_t

{

protected:

characterStats\_t stat;

std::unique\_ptr<CharacterState\_t> state;

std::unique\_ptr<Effect\_t> buff;

std::list<elements::element> skillGeneratorArr;

size\_t elemStatus;

std::unique\_ptr<skillObGenerator\_t> skill;

sf::Clock \*clock;

characterTimers\_t timer;

sf::Vector2f targetPos;

sf::Vector2f spawnCoords;

float moveRadius;

virtual void animation();

protected:

character\_t(float \_x, float \_y, std::string fileName, int \_coordX, int \_coordY, int \_width, int \_height, sf::Clock \*\_clock, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList);

character\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, float \_x, float \_y, int \_coordX, int \_coordY, int \_width, int \_height, sf::Clock \*\_clock);

public:

virtual ~character\_t();

void changeState(CharacterState\_t \*newState);

void changeEffect(Effect\_t \*newEffect);

void defaultStats();

void defaultAllStats();

virtual void attack();

virtual bool checkAlive();

virtual float takeDamage(float \_dmg, bool \_dmgType, elements::element \_elem);

float takeHeal(float \_heal);

bool checkCollision(physOb\_t &Object);

virtual bool checkEnemy(character\_t \*ob);

virtual bool checkSkillGenerator();

virtual bool addElement(elements::element \_elem);

virtual void generateSkillAndClearElemList();

virtual void resetElemsList();

virtual size\_t setElemStatus(size\_t \_elemStatus);

virtual std::vector<size\_t> getSkillList();

virtual void setSkills(std::initializer\_list<size\_t> \_skillList);

void useMP(float \_mp);

bool kill();

void update(float \_speed);

float toHit ()const;

void updateTimerDependenceStats();

public:

//GET

Effect\_t \*getEffectPtr();

characterStats\_t &getStats();

character\_t \*getPtr();

sf::Vector2f getTargetPos();

sf::Vector2f getSpotCoords();

sf::Vector2f getSpawnCoords();

float getMoveRadius() const;

sf::Clock \*getClockPtr()const;

size\_t getElemStatus()const;

characterTimers\_t &getTimer();

std::list<elements::element> getElements();

skillObGenerator\_t \*getSkillGeneratorPtr();

float getComputedDmg()const;

CharacterState\_t \*getState()const;

//SET

float setFrame(float \_frame);

sf::Vector2f setTargetPos(sf::Vector2f \_targetPos);

void setStats(characterStats\_t &\_stats);

void setMoveRadius(float \_radius);

};

================================= “character\_t.cpp” =================================

#include "character\_t.h"

character\_t::character\_t(float \_x, float \_y, std::string fileName, int \_coordX, int \_coordY, int \_width, int \_height, sf::Clock \*\_clock, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList) : physOb\_t(\_x, \_y, fileName, \_coordX, \_coordY, \_width, \_height), timer(\_clock) {

skill.swap(std::unique\_ptr<skillObGenerator\_t>(new skillObGenerator\_t(this, \_bulletList)));

buff.swap(std::unique\_ptr<Effect\_t>(new Effect\_t(this,this->getStats())));

destroyble = true;

frame = .0f;

direction = animation::BOTTOM;

collision = false;

fraction = 1;

targetPos = spawnCoords = sf::Vector2f(\_x, \_y);

clock = \_clock;

timer.attackCDcorrection(stat.attackSpeed);

timer.castDelayCorrection(stat.castSpeed);

skillGeneratorArr = std::list<elements::element>(elements::SKILL\_ELEMENT\_AMOUNT);

moveRadius = 500.f;

}

character\_t::character\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, float \_x, float \_y, int \_coordX, int \_coordY, int \_width, int \_height, sf::Clock \*\_clock) : physOb\_t(\_x, \_y, \_texture, \_coordX, \_coordY, \_width, \_height), timer(\_clock) {

skill.swap(std::unique\_ptr<skillObGenerator\_t>(new skillObGenerator\_t(this, \_bulletList)));

buff.swap(std::unique\_ptr<Effect\_t>(new Effect\_t(this, this->getStats())));

destroyble = true;

frame = .0f;

direction = animation::BOTTOM;

collision = false;

fraction = 1;

targetPos = spawnCoords = sf::Vector2f(\_x, \_y);

clock = \_clock;

timer.attackCDcorrection(stat.attackSpeed);

timer.castDelayCorrection(stat.castSpeed);

skillGeneratorArr = std::list<elements::element>(elements::SKILL\_ELEMENT\_AMOUNT);

moveRadius = 500.f;

}

character\_t::~character\_t()

{

}

void character\_t::defaultStats() {

stat.defaultStats();

}

void character\_t::defaultAllStats() {

stat.defaultAllStats();

}

bool character\_t::kill() {

alive = false;

return false;

};

void character\_t::update(float \_speed) {

checkAlive();

if (alive) {

physOb\_t::update(\_speed);

if (buff.get()) {

buff->checkActivity();

}

}

}

void character\_t::animation() {

int spriteCoordX = static\_cast<int>(frame) \* getWidth();

int spriteCoordY = direction \* getHeight();

spritePref.setTexturePos(spriteCoordX, spriteCoordY);

}

void character\_t::setSkills(std::initializer\_list<size\_t> \_skillList) {

elemStatus = \*\_skillList.begin();

}

std::vector<size\_t> character\_t::getSkillList() {

return std::vector<size\_t>(1,elemStatus);

}

bool character\_t::checkAlive() {

if (stat.HP < FLT\_EPSILON) {

alive = false;

}

else if (stat.HP > stat.stdHP) {

stat.HP = stat.stdHP;

}

return alive;

}

float character\_t::takeDamage(float \_dmg, bool \_dmgType, elements::element \_elem) {

if (alive) {

float tempDmg = 0.f;

if (\_dmgType) {

tempDmg = (\_elem == elements::NONE) ? (\_dmg - abs(stat.physDef)) : (\_dmg - abs(stat.magDef));

if (tempDmg < 0) {

tempDmg = 1.f;

}

stat.HP -= tempDmg;

}

return tempDmg;

}

return 0.f;

}

float character\_t::takeHeal(float \_heal) {

if (alive && stat.HP<stat.stdHP) {

stat.HP += \_heal;

}

return \_heal;

}

float character\_t::toHit() const{

return stat.attackPower + getRand(-stat.damageRand, stat.damageRand);

}

bool character\_t::checkCollision(physOb\_t &Object) {

if (getFloatRect().contains(Object.getPosOfCenter())){

return true;

}

return false;

}

bool character\_t::checkEnemy(character\_t \*ob) {

float distanceX = (ob->getPosOfCenter().x )- (getPosOfCenter().x);

float distanceY = (ob->getPosOfCenter().y) - (getPosOfCenter().y);

float vectorLength = sqrt(pow(distanceX, 2) + pow(distanceY, 2));

if (vectorLength < stat.visionDistance && ob->getAlive() && fraction != ob->getFraction()) {

return true;

}

return false;

}

void character\_t::changeState(CharacterState\_t \*newState) {

state.get\_deleter();

state.swap(std::unique\_ptr<CharacterState\_t>(newState));

}

void character\_t::changeEffect(Effect\_t \*newEffect) {

buff.get\_deleter();

buff.swap(std::unique\_ptr<Effect\_t>(newEffect));

}

void character\_t::attack() {

//start cast

if (timer.attackReady()) {

timer.updateAttackCD();

skill->useSkill();

}

}

bool character\_t::checkSkillGenerator() {

if (skillGeneratorArr.size() == elements::SKILL\_ELEMENT\_AMOUNT) {

std::list<elements::element>::iterator temp = skillGeneratorArr.begin();

for (size\_t i = 0; i < skillGeneratorArr.size(); ++i, ++temp) {

if ((\*temp) == elements::NONE) return false;

}

return true;

}

else {

return false;

}

}

bool character\_t::addElement(elements::element \_elem) {

if (skillGeneratorArr.size() >= elements::SKILL\_ELEMENT\_AMOUNT) {

skillGeneratorArr.pop\_front();

skillGeneratorArr.push\_back(\_elem);

return true;

}

return false;

}

void character\_t::generateSkillAndClearElemList() {

stat.MP = stat.stdMP;

elemStatus = std::accumulate(skillGeneratorArr.begin(), skillGeneratorArr.end(), 0);

resetElemsList();

}

void character\_t::resetElemsList() {

for (auto &i : skillGeneratorArr) {

i = elements::NONE;

}

}

void character\_t::useMP(float \_mp) {

if (stat.MP > FLT\_EPSILON) {

stat.MP -= \_mp;

}

}

size\_t character\_t::setElemStatus(size\_t \_elemStatus) {

elemStatus = \_elemStatus;

return elemStatus;

}

void character\_t::updateTimerDependenceStats() {

timer.attackCDcorrection(stat.attackSpeed);

timer.castDelayCorrection(stat.castSpeed);

}

//GET

characterStats\_t &character\_t::getStats() {

return stat;

}

Effect\_t \*character\_t::getEffectPtr() {

return buff.get();

}

character\_t \*character\_t::getPtr() {

return this;

}

sf::Vector2f character\_t::getTargetPos() {

return targetPos;

}

sf::Vector2f character\_t::getSpotCoords() {

return sf::Vector2f(1800.f,1800.f);

}

sf::Vector2f character\_t::getSpawnCoords() {

return spawnCoords;

}

float character\_t::getMoveRadius() const {

return moveRadius;

}

sf::Clock \*character\_t::getClockPtr() const {

return clock;

}

size\_t character\_t::getElemStatus() const {

return elemStatus;

}

characterTimers\_t &character\_t::getTimer() {

return timer;

}

std::list<elements::element> character\_t::getElements() {

return skillGeneratorArr;

}

skillObGenerator\_t \*character\_t::getSkillGeneratorPtr() {

return skill.get();

}

float character\_t::getComputedDmg() const {

float tempDmg = stat.attackPower + getRand(-stat.damageRand, stat.damageRand);

return tempDmg;

}

CharacterState\_t \*character\_t::getState() const {

return state.get();

}

//SET

void character\_t::setStats(characterStats\_t &\_stats) {

stat = \_stats;

timer.attackCDcorrection(stat.attackSpeed);

timer.castDelayCorrection(stat.castSpeed);

}

float character\_t::setFrame(float \_frame) {

frame = \_frame;

return frame;

}

sf::Vector2f character\_t::setTargetPos(sf::Vector2f \_targetPos) {

targetPos = \_targetPos;

return targetPos;

}

void character\_t::setMoveRadius(float \_radius) {

moveRadius = \_radius;

}

================================= “Character\_state.h” =================================

#pragma once

#include "character\_t.h"

#include "bullet\_t.h"

#include <list>

#include <memory>

class character\_t;

class bullet\_t;

class CharacterState\_t

{

protected:

int stateNum;

character\_t &mainCharacter;

character\_t \*targetCharacter;

bool readyToFight;

float moveXdistanceFromSpawn;

CharacterState\_t(character\_t &\_targetOb);

CharacterState\_t(CharacterState\_t &\_state);

public:

virtual ~CharacterState\_t();

virtual void Action() = 0;

bool leaveFromSpot();

void reversXTrajectory();

character\_t &getCharacterPtr() const;

character\_t \*getTargetCharacterPtr()const;

int getStateNum() const;

void setTargetCharacter(character\_t \*character);

void resetTargetCharacter();

};

class CharacterStateMove\_t :

public CharacterState\_t

{

public:

CharacterStateMove\_t(character\_t &\_\_mainCharacter);

CharacterStateMove\_t(CharacterState\_t &\_state);

virtual ~CharacterStateMove\_t();

virtual void Action();

};

class CharacterStateFolow\_t :

public CharacterState\_t

{

public:

CharacterStateFolow\_t(character\_t &\_\_mainCharacter);

CharacterStateFolow\_t(CharacterState\_t &\_state);

virtual ~CharacterStateFolow\_t();

virtual void Action();

};

class CharacterStateAttack\_t :

public CharacterState\_t

{

public:

CharacterStateAttack\_t(character\_t &\_\_mainCharacter);

CharacterStateAttack\_t(CharacterState\_t &\_state);

virtual ~CharacterStateAttack\_t();

virtual void Action();

void Action(std::list<bullet\_t\*> &obList);

};

//PLAYER

class CharacterPlayerControll\_t :

public CharacterState\_t

{

public:

CharacterPlayerControll\_t(character\_t &\_\_mainCharacter);

virtual ~CharacterPlayerControll\_t();

virtual void Action();

};

class CharacterPlayerCast\_t :

public CharacterState\_t

{

public:

CharacterPlayerCast\_t(character\_t &\_\_mainCharacter);

virtual ~CharacterPlayerCast\_t();

virtual void Action();

};

//\*/

================================= “CharacterState.cpp” =================================

#include "CharacterState\_t.h"

CharacterState\_t::CharacterState\_t(character\_t &\_\_mainCharacter):mainCharacter(\_\_mainCharacter)

{

targetCharacter = nullptr;

stateNum = 0;

moveXdistanceFromSpawn = 200.f;

readyToFight = true;

}

CharacterState\_t::CharacterState\_t(CharacterState\_t &\_state):mainCharacter(\_state.getCharacterPtr())

{

targetCharacter = \_state.getTargetCharacterPtr();

stateNum = \_state.stateNum;

mainCharacter.setTargetPos(\_state.getCharacterPtr().getTargetPos());

moveXdistanceFromSpawn = \_state.moveXdistanceFromSpawn;

readyToFight = true;

}

CharacterState\_t::~CharacterState\_t()

{

}

bool CharacterState\_t::leaveFromSpot() {

sf::Vector2f tempSpotVector = mainCharacter.getSpawnCoords();

float distanceX = tempSpotVector.x - mainCharacter.getPosX();

float distanceY = tempSpotVector.y - mainCharacter.getPosY();

float vectorLength = sqrt(pow(distanceX, 2) + pow(distanceY, 2));

if (vectorLength > mainCharacter.getMoveRadius()) {

return true;

}

return false;

}

void CharacterState\_t::reversXTrajectory() {

moveXdistanceFromSpawn = -moveXdistanceFromSpawn;

}

//RESET

void CharacterState\_t::resetTargetCharacter() {

targetCharacter = NULL;

}

//GET

character\_t & CharacterState\_t::getCharacterPtr() const {

return mainCharacter;

}

character\_t \* CharacterState\_t::getTargetCharacterPtr()const {

return targetCharacter;

}

int CharacterState\_t::getStateNum() const {

return stateNum;

}

//SET

void CharacterState\_t::setTargetCharacter(character\_t \*character) {

targetCharacter = character;

}

//CHARACTER STATE MOVE

CharacterStateMove\_t::CharacterStateMove\_t(character\_t &\_\_mainCharacter) :CharacterState\_t(\_\_mainCharacter)

{

stateNum = 0;

targetCharacter = nullptr;

mainCharacter.setTargetPos(mainCharacter.getSpawnCoords());

readyToFight = false;

}

CharacterStateMove\_t::CharacterStateMove\_t(CharacterState\_t &\_state) : CharacterState\_t(\_state) {

stateNum = 0;

targetCharacter = nullptr;

mainCharacter.setTargetPos(mainCharacter.getSpawnCoords());

readyToFight = false;

}

CharacterStateMove\_t::~CharacterStateMove\_t()

{

}

void CharacterStateMove\_t::Action() {

float distanceX = mainCharacter.getTargetPos().x - mainCharacter.getPosX();

float distanceY = mainCharacter.getTargetPos().y - mainCharacter.getPosY();

float vectorLength = sqrt(pow(distanceX, 2) + pow(distanceY, 2));

if ((targetCharacter != nullptr) && (readyToFight) && targetCharacter->getAlive()) {

mainCharacter.changeState(new CharacterStateFolow\_t(\*this));

}

else {

resetTargetCharacter();

}

if (vectorLength > 1.f ) {

float kX = (distanceX / abs(distanceX)) \* mainCharacter.getStats().speed;

float kY = (distanceY / abs(distanceY)) \* mainCharacter.getStats().speed;

if (abs(distanceX) > mainCharacter.getWidth() / 2) {

if (abs(kX) > FLT\_EPSILON) mainCharacter.setdX(kX);

}

else if (abs(distanceY) > mainCharacter.getHeight() / 2) {

if (abs(kY) > FLT\_EPSILON) mainCharacter.setdY(kY);

}

else {

targetCharacter = nullptr;

readyToFight = true;

}

}

else {

targetCharacter = nullptr;

readyToFight = true;

}

}

//CHARACTER STATE FOLOW

CharacterStateFolow\_t::CharacterStateFolow\_t(character\_t &\_\_mainCharacter) :CharacterState\_t(\_\_mainCharacter)

{

stateNum = 1;

}

CharacterStateFolow\_t::CharacterStateFolow\_t(CharacterState\_t &\_state) : CharacterState\_t(\_state) {

stateNum = 1;

}

CharacterStateFolow\_t::~CharacterStateFolow\_t()

{

}

void CharacterStateFolow\_t::Action()

{

float distanceX = targetCharacter->getPosOfCenter().x - mainCharacter.getPosOfCenter().x;

float distanceY = targetCharacter->getPosOfCenter().y - mainCharacter.getPosOfCenter().y;

float vectorLength = sqrt(pow(distanceX, 2) + pow(distanceY, 2));

float visionMultiple = 2.f;

if (leaveFromSpot() || vectorLength > mainCharacter.getStats().visionDistance \* visionMultiple) {

mainCharacter.setTargetPos(mainCharacter.getSpawnCoords());

mainCharacter.changeState(new CharacterStateMove\_t(\*this));

}

else {

if (vectorLength < (mainCharacter.getStats().attackRange)) {

mainCharacter.changeState(new CharacterStateAttack\_t(\*this));

}

float kX = (distanceX / abs(distanceX) - FLT\_EPSILON) \* mainCharacter.getStats().speed;

float kY = (distanceY / abs(distanceY) - FLT\_EPSILON) \* mainCharacter.getStats().speed;

if (abs(distanceX) > mainCharacter.getWidth() / 2) {

if (abs(kX) > FLT\_EPSILON) mainCharacter.setdX(kX);

} else

if (abs(distanceY) > mainCharacter.getHeight() / 2) {

if (abs(kY) > FLT\_EPSILON) mainCharacter.setdY(kY);

}

}

}

//CHARACTER STATE ATTACK

CharacterStateAttack\_t::CharacterStateAttack\_t(character\_t &\_\_mainCharacter) :CharacterState\_t(\_\_mainCharacter)

{

stateNum = 2;

}

CharacterStateAttack\_t::CharacterStateAttack\_t(CharacterState\_t &\_state) : CharacterState\_t(\_state) {

stateNum = 2;

}

CharacterStateAttack\_t::~CharacterStateAttack\_t()

{

}

void CharacterStateAttack\_t::Action() {

float distanceX = targetCharacter->getPosOfCenter().x - mainCharacter.getPosOfCenter().x;

float distanceY = targetCharacter->getPosOfCenter().y - mainCharacter.getPosOfCenter().y;

float vectorLength = sqrt(pow(distanceX, 2) + pow(distanceY, 2));

float visionMultiple = 2.f;

if (vectorLength > mainCharacter.getStats().attackRange){

mainCharacter.changeState(new CharacterStateFolow\_t(\*this));

}

else if (vectorLength < mainCharacter.getStats().attackRange && targetCharacter->getAlive()) {

mainCharacter.setTargetPos(targetCharacter->getPosOfCenter());

mainCharacter.attack();

}

else {

mainCharacter.changeState(new CharacterStateMove\_t(\*this));

}

}

void CharacterStateAttack\_t::Action(std::list<bullet\_t\*> &obList) {

}

//CHARACTER STATE PLAYER CONTROLL

CharacterPlayerControll\_t::CharacterPlayerControll\_t(character\_t &\_\_mainCharacter): CharacterState\_t(\_\_mainCharacter)

{

stateNum = 3;

}

CharacterPlayerControll\_t::~CharacterPlayerControll\_t()

{

}

void CharacterPlayerControll\_t::Action() {

}

//\*/

//CHARACTER STATE PLAYER CONTROLL

CharacterPlayerCast\_t::CharacterPlayerCast\_t(character\_t &\_\_mainCharacter) : CharacterState\_t(\_\_mainCharacter)

{

stateNum = 4;

}

CharacterPlayerCast\_t::~CharacterPlayerCast\_t()

{

}

void CharacterPlayerCast\_t::Action() {

}

================================= “characterStats.h” =================================

#pragma once

struct characterStats\_t

{

float HP = 100.f;

float stdHP = 100.f;

float MP = 0.f;

float stdMP = 100.f;

float attackPower = 25.f;

float stdattackPower = 25.f;

float physDef = 10.f;

float stdPhysDef = 10.f;

float magDef = 5.f;

float stdMagDef = 3.f;

float speed = 0.15f;

float stdSpeed = 0.1f;

float castSpeed = 2.f;

float stdCastSpeed = 2.f;

float attackSpeed = 3.5f;

float stdAttackSpeed = 3.5f;

float damageRand = 5.f;

float attackRange = 60.f;

float visionDistance = 300.f;

void resetStats() {

HP = MP = attackPower =

physDef = magDef = speed = 0.f;

}

void defaultStats() {

attackPower = stdattackPower;

physDef = stdPhysDef;

magDef = stdMagDef;

speed = stdSpeed;

castSpeed = stdCastSpeed;

attackSpeed = stdAttackSpeed;

}

void defaultAllStats() {

defaultStats();

HP = stdHP;

MP = stdMP;

}

void upStat(characterStats\_t &stat) {

HP += stat.HP;

MP += stat.MP;

attackPower += stat.attackPower;

physDef += stat.physDef;

magDef += stat.magDef;

speed += stat.speed;

}

void statMiltipler(float \_mult = 1.f) {

float speedK = 0.01f;

float tempMult = abs(\_mult);

HP \*= tempMult;

stdHP \*= tempMult;

MP \*= tempMult;

stdMP \*= tempMult;

attackPower \*= tempMult;

stdattackPower \*= tempMult;

stdAttackSpeed \*= tempMult;

if (tempMult > 1.f) {

speed += speedK \* tempMult;

stdSpeed += speedK \* tempMult;

}

}

};

================================= “characterTimers\_t.h” =================================

#pragma once

#include <SFML/Graphics.hpp>

#include <map>

class characterTimers\_t

{

struct timerStats {

sf::Int32 startTime;

sf::Int32 cooldown;

bool ready = true;

};

//get init time in Milliseconds

sf::Clock \*clock;

//std::map<std::string, timerStats> timersList;

std::vector<timerStats> timersList;

const std::string attackCDkey = "attackCD";

const std::string castDelaykey = "castDelay";

const std::string skillGenerationCDkey = "skillGenerationCD";

void setAttackStartTimer(sf::Int32 \_curTime);

void setcastDelayStartTimer(sf::Int32 \_curTime);

void setswapDirectionTimerStartTimer(sf::Int32 \_curTime);

public:

characterTimers\_t();

characterTimers\_t(sf::Clock \*\_clock, int \_castSpeed = 1, int \_attackSpeed = 1);

~characterTimers\_t();

characterTimers\_t &operator =(characterTimers\_t \_secondTimer);

//GET

sf::Clock \*getClockPtr() const {

return clock;

}

sf::Int32 &getAttackStartTime();

sf::Int32 &getCastStartTime();

sf::Int32 &getSwapDirStartTime();

sf::Int32 &getAttackCD();

sf::Int32 &getCastDelay();

sf::Int32 &getDirectionSwapTime();

sf::Int32 attackCDcorrection(float \_attackSpeed);

sf::Int32 castDelayCorrection(float \_castDelay);

sf::Int32 directionSwapTimeCorrection(int \_time);

void setAttackCD(int \_attackCD);

void setCastDelay(int \_castDelay);

void setDirectionSwapTime(int \_time);

bool attackReady();

bool castReady();

bool swapDirectionReady();

void updateAttackCD();

void updateCastCD();

void updateSwapDirectionCD();

};

================================ “characterTimers.cpp” ================================

#include "characterTimers\_t.h"

const sf::Int32 STD\_ATTACK\_CD = 1000;

const sf::Int32 STD\_CAST\_DELAY = 2000;

characterTimers\_t::characterTimers\_t()

{

}

characterTimers\_t::characterTimers\_t(sf::Clock \*\_clock, int \_castSpeed, int \_attackSpeed)

{

clock = \_clock;

sf::Int32 startTime = 1;

timerStats attackCD = { startTime, STD\_ATTACK\_CD };

timersList.push\_back(attackCD);

timerStats castDelay = { startTime, STD\_CAST\_DELAY };

timersList.push\_back(castDelay);

timerStats skillGenerationCD = { startTime, 4000 };

timersList.push\_back(skillGenerationCD);

}

characterTimers\_t::~characterTimers\_t()

{

}

characterTimers\_t &characterTimers\_t::operator =(characterTimers\_t \_secondTimer) {

\_secondTimer.timersList = timersList;

return \*this;

}

sf::Int32 &characterTimers\_t::getAttackStartTime() {

return timersList[0].startTime;

}

sf::Int32 &characterTimers\_t::getCastStartTime() {

return timersList[1].startTime;

}

sf::Int32 &characterTimers\_t::getSwapDirStartTime() {

return timersList[2].startTime;

}

sf::Int32 &characterTimers\_t::getAttackCD() {

return timersList[0].cooldown;

}

sf::Int32 &characterTimers\_t::getCastDelay() {

return timersList[1].cooldown;

}

sf::Int32 &characterTimers\_t::getDirectionSwapTime() {

return timersList[2].cooldown;

}

sf::Int32 characterTimers\_t::attackCDcorrection(float \_attackCD) {

timersList[0].cooldown = STD\_ATTACK\_CD - static\_cast<sf::Int32>(\_attackCD \* 100);

return timersList[0].cooldown;

}

sf::Int32 characterTimers\_t::castDelayCorrection(float \_castDelay) {

timersList[1].cooldown = STD\_CAST\_DELAY - static\_cast<sf::Int32>(\_castDelay \* 100);

return timersList[1].cooldown;

}

sf::Int32 characterTimers\_t::directionSwapTimeCorrection(int \_time) {

timersList[2].cooldown /= \_time;

return timersList[2].cooldown;

}

void characterTimers\_t::setAttackCD(int \_attackCD) {

timersList[0].cooldown = abs(\_attackCD);

}

void characterTimers\_t::setCastDelay(int \_castDelay) {

timersList[1].cooldown = abs(\_castDelay);

}

void characterTimers\_t::setDirectionSwapTime(int \_time) {

timersList[2].cooldown = abs(\_time);}

void characterTimers\_t::setAttackStartTimer(sf::Int32 \_curTime) {

timersList[0].startTime = \_curTime;

}

void characterTimers\_t::setcastDelayStartTimer(sf::Int32 \_curTime) {

timersList[1].startTime = \_curTime;

}

void characterTimers\_t::setswapDirectionTimerStartTimer(sf::Int32 \_curTime) {

timersList[2].startTime = \_curTime;

}

bool characterTimers\_t::attackReady() {

sf::Int32 curTime = clock->getElapsedTime().asMilliseconds();

timersList[0].ready = (abs(curTime - timersList[0].startTime) > timersList[0].cooldown) ? true : false;

return timersList[0].ready;

}

bool characterTimers\_t::castReady() {

sf::Int32 curTime = clock->getElapsedTime().asMilliseconds();

timersList[1].ready = (abs(curTime - timersList[1].startTime) > timersList[1].cooldown) ? true : false;

return timersList[1].ready;

}

bool characterTimers\_t::swapDirectionReady() {

sf::Int32 curTime = clock->getElapsedTime().asMilliseconds();

timersList[2].ready = (abs(curTime - timersList[2].startTime) > timersList[2].cooldown) ? true : false;

return timersList[2].ready;

}

void characterTimers\_t::updateAttackCD() {

sf::Int32 curTime = clock->getElapsedTime().asMilliseconds();

timersList[0].ready = false;

setAttackStartTimer(curTime);

}

void characterTimers\_t::updateCastCD() {

sf::Int32 curTime = clock->getElapsedTime().asMilliseconds();

timersList[1].ready = false;

setcastDelayStartTimer(curTime);

}

void characterTimers\_t::updateSwapDirectionCD() {

sf::Int32 curTime = clock->getElapsedTime().asMilliseconds();

timersList[2].ready = false;

setswapDirectionTimerStartTimer(curTime);

}

================================= “cursor\_t.h” =================================

#pragma once

#include <SFML\Graphics.hpp>

class cursor\_t

{

sf::Texture texture;

sf::Sprite sprite;

int height;

int width;

sf::RenderWindow \*window;

public:

cursor\_t();

cursor\_t(std::string fileName, int \_height, int \_width, sf::RenderWindow \*\_window);

~cursor\_t();

void setCursorPosition();

//GET

sf::Sprite getSprite() const;

sf::Vector2f getPosition();

//SET

void setTextureFromFile(std::string fileName);

};

“cursor\_t.cpp”

#include "cursor\_t.h"

cursor\_t::cursor\_t() {

}

cursor\_t::cursor\_t(std::string fileName, int \_height, int \_width, sf::RenderWindow \*\_window)

{

height = \_height;

width = \_width;

window = \_window;

window->setMouseCursorVisible(false);

texture.loadFromFile("img/cursor\_aim.png");

sprite.setTexture(texture);

sprite.setTextureRect(sf::IntRect(0, 0, width, height));

sprite.setOrigin(static\_cast<float>(width) / 2, static\_cast<float>(height) / 2);

}

cursor\_t::~cursor\_t()

{

}

void cursor\_t::setCursorPosition() {

sf::Vector2i mousePos(sf::Mouse::getPosition(\*window));

sf::Vector2f globalPos = window->mapPixelToCoords(mousePos);

sprite.setPosition(globalPos);

}

//GET

sf::Sprite cursor\_t::getSprite() const {

return sprite;

}

sf::Vector2f cursor\_t::getPosition(){

sf::Vector2i mousePos(sf::Mouse::getPosition(\*window));

sf::Vector2f globalPos = window->mapPixelToCoords(mousePos);

return globalPos;

}

//SET

void cursor\_t::setTextureFromFile(std::string fileName) {

texture.loadFromFile(fileName);

sprite.setTexture(texture);

}

================================= “Effect\_t.h” =================================

#pragma once

#include <SFML/Graphics.hpp>

#include "character\_t.h"

class Effect\_t

{

character\_t \*character;

public:

sf::Int32 startTime;

sf::Int32 duration;

characterStats\_t stat;

bool activity;

Effect\_t(character\_t \*\_character, characterStats\_t \_stats, sf::Int32 \_duration = 5000);

virtual ~Effect\_t();

void useEffect();

bool timerIsOver();

void cancelEffect();

void checkActivity();

};

“Effect\_t.cpp”

#include "Effect\_t.h"

Effect\_t::Effect\_t(character\_t \*\_character,characterStats\_t \_stats, sf::Int32 \_duration)

{

character = \_character;

stat = \_stats;

duration = \_duration;

startTime = 0;

activity = false;

}

Effect\_t::~Effect\_t()

{

cancelEffect();

}

void Effect\_t::useEffect() {

startTime = character->getClockPtr()->getElapsedTime().asMilliseconds();

character->getStats().upStat(stat);

character->updateTimerDependenceStats();

activity = true;

}

bool Effect\_t::timerIsOver() {

if (timeIsOver(character->getClockPtr(),startTime,duration)) {

return true;

}

return false;

}

void Effect\_t::cancelEffect() {

character->getStats().defaultStats();

}

void Effect\_t::checkActivity() {

if (timerIsOver()) cancelEffect();

}

================================= “game\_t.h” =================================

#pragma once

#include <SFML/Graphics.hpp>

#include <iostream>

#include "Level\_t.h"

#include "GameEngine\_t.h"

#include "InterfaceEngine\_t.h"

#include "keyboardController.h"

#include "mouseController.h"

#include "mainMenu\_t.h"

#include "additional.h"

class mouseController;

class keyboardController;

class game\_t

{

sf::RenderWindow \*window;

sf::Clock clock;

std::unique\_ptr<Level\_t> level;

std::unique\_ptr<keyboardController> KBcontroller;

std::unique\_ptr<mouseController> Mcontroller;

std::string levelName;

size\_t difficulty;

game::status status;

public:

std::unique\_ptr<GameEngine\_t> gameEngine;

std::unique\_ptr<InterfaceEngine\_t> interfaceEngine;

game\_t(sf::RenderWindow \*\_window, std::string \_levelName, size\_t \_difficulty);

~game\_t();

sf::RenderWindow \*getWindowPtr() const {

return window;

}

void start();

void play();

void resetGame();

void keyController(sf::Event &event);

game::status getStatus() const;

void setGameStatus(game::status \_newStatus);

};

“game\_t.cpp”

#include "game\_t.h"

game\_t::game\_t(sf::RenderWindow \*\_window, std::string \_levelName, size\_t \_difficulty)

{

window = \_window;

levelName = \_levelName;

difficulty = \_difficulty;

status = game::MAIN\_MENU;

}

game\_t::~game\_t()

{

resetGame();

}

void game\_t::start() {

level.swap(std::unique\_ptr<Level\_t>(new Level\_t(levelName)));

if (!level->succesfull) {

std::cout << "MAP\_FILE\_IS\_NOT\_OPEN";

return;

}

gameEngine.swap(std::unique\_ptr<GameEngine\_t>(new GameEngine\_t(window, \*level.get(), difficulty)));

interfaceEngine.swap(std::unique\_ptr<InterfaceEngine\_t>(new InterfaceEngine\_t(window, \*level.get())));

KBcontroller.swap(std::unique\_ptr<keyboardController>(new keyboardController(level->mainHero->get(), this)));

Mcontroller.swap(std::unique\_ptr<mouseController>(new mouseController(window, this, \*interfaceEngine.get()->cursor)));

play();

}

void game\_t::play() {

using namespace sf;

Event event;

status = game::PLAY;

while (window->isOpen()) {

float timer = static\_cast<float>(clock.getElapsedTime().asMicroseconds());

clock.restart();

while (window->pollEvent(event)) {

if (event.type == Event::Closed) {

window->close();

}

}

switch (status) {

case game::status::PLAY: {

gameEngine->setSpeed(timer);

gameEngine->update();

Mcontroller->eventHandler(event);

status = gameEngine->getGameStatus();

break;

}

case game::RESTART: {

return;

}

case game::MAIN\_MENU: {

return;

}

case game::PAUSED: {

Mcontroller->menuEventHandler(event);

break;

}

case game::GAME\_OVER: {

interfaceEngine->pausedMenuIt->get()->contentList.back()->setFontColor(sf::Color::Red);

interfaceEngine->pausedMenuIt->get()->contentList.back()->setText("YOU DIE");

interfaceEngine->pausedMenuIt->get()->setDisplay(true);

Mcontroller->menuEventHandler(event);

break;

}

case game::WIN: {

interfaceEngine->pausedMenuIt->get()->contentList.back()->setFontColor(sf::Color::Yellow);

interfaceEngine->pausedMenuIt->get()->contentList.back()->setText("YOU WIN");

interfaceEngine->pausedMenuIt->get()->setDisplay(true);

interfaceEngine->winWindowIt->get()->setDisplay(true);

Mcontroller->menuEventHandler(event);

break;

}

}

keyController(event);

interfaceEngine->update();

window->clear();

gameEngine->draw();

interfaceEngine->draw();

window->display();

event.key.code = Keyboard::Unknown;

}

}

void game\_t::keyController(sf::Event &event) {

if (level->mainHero->get()->getAlive()) {

KBcontroller->eventHandler(event);

}

}

game::status game\_t::getStatus() const {

return status;

}

void game\_t::setGameStatus(game::status \_newStatus) {

status = \_newStatus;

}

void game\_t::resetGame() {

clock.restart();

KBcontroller.get\_deleter();

Mcontroller.get\_deleter();

gameEngine.get\_deleter();

interfaceEngine.get\_deleter();

level.get\_deleter();

}

================================= “GameEngine\_t.h” =================================

#pragma once

#include <SFML/Graphics.hpp>

#include <list>

#include <memory>

#include "character\_t.h"

#include "player\_t.h"

#include "Npc\_t.h"

#include "BossNpc\_t.h"

#include "bullet\_t.h"

#include "additional.h"

#include "Level\_t.h"

const float STD\_DIFFICULTY\_COEFFICIENT = 0.7f;

class GameEngine\_t

{

private:

sf :: RenderWindow \*window;

Level\_t &level;

size\_t difficulty;

std::list<std::unique\_ptr<character\_t>> npcTypesList;

std::list<std::unique\_ptr<BossNpc\_t>> npcBossesTypeList;

game::status status;

std::unique\_ptr<sf::Clock> clock;

float curTime;

float speed;

float speedMultipple;

void generateNpcTypes();

bool positionCollision(const sf::Vector2f \_obPos);

public:

GameEngine\_t(sf::RenderWindow \*\_window, Level\_t &\_level, size\_t \_difficulty);

~GameEngine\_t();

void update();

void draw();

void checkAlive();

void visionEngine();

void collisionEngine();

void bulletEngine();

void charsAction();

void generateNpc();

void generateBosses();

//GET

game::status getGameStatus() const;

float getSpeed()const;

float getCurTimeSec() const;

float getSpeedMultipple()const;

std::list<std::unique\_ptr <bullet\_t>> &getBulletList();

//SET

float setSpeed(float \_time);

};

================================= “GameEngine\_t.cpp” =================================

#include "GameEngine\_t.h"

GameEngine\_t::GameEngine\_t(sf::RenderWindow \*\_window, Level\_t &\_level, size\_t \_difficulty):level(\_level)

{

window = \_window;

difficulty = \_difficulty;

clock.swap(std::unique\_ptr<sf::Clock>(new sf::Clock));

speedMultipple = 900.f; //formula (gameSpeed = time/speedMultipple)

speed = 10.f;

using namespace animation;

std::shared\_ptr<sf::Texture> temp = std::make\_shared<sf::Texture>();

temp->loadFromFile(MAIN\_HERO\_TEXTURE\_FILE);//

tiles::sizes tempSizes = tiles::getSizesFromStr(MAIN\_HERO\_TEXTURE\_FILE);

level.charactersList.push\_back(std::unique\_ptr <character\_t>(new player\_t(temp, level.bulletsList, level.map.mainHeroSpawnCoords, tempSizes.width, tempSizes.height, clock.get())));

level.mainHero = level.charactersList.begin();

generateNpcTypes();

generateNpc();

generateBosses();

status = game::status::PLAY;

}

GameEngine\_t::~GameEngine\_t()

{

}

bool GameEngine\_t::positionCollision(const sf::Vector2f \_obPos) {

for (auto &i :level.map.mapObList) {

if (i->getFloatRect().contains(\_obPos)) {

return true;

}

}

return false;

}

void GameEngine\_t::generateNpcTypes() {

using namespace animation;

tiles::sizes tempSizes;

sf::Vector2f defaultSpawnCoords(0.f, 0.f);

//DEMON TYPE NPC

std::shared\_ptr<sf::Texture> demonTexture = std::make\_shared<sf::Texture>();

demonTexture->loadFromFile(ENEMY\_DEMON\_FILE);

tempSizes = tiles::getSizesFromStr(ENEMY\_DEMON\_FILE);

npcTypesList.push\_back(std::unique\_ptr <character\_t>(new Npc\_t(demonTexture, level.bulletsList, clock.get(), defaultSpawnCoords, tempSizes.width, tempSizes.height)));

//WARRIOR TYPE NPC

std::shared\_ptr<sf::Texture> WarriorTexture = std::make\_shared<sf::Texture>();

WarriorTexture->loadFromFile(ENEMY\_WARRIOR\_FILE);

tempSizes = tiles::getSizesFromStr(ENEMY\_WARRIOR\_FILE);

npcTypesList.push\_back(std::unique\_ptr <character\_t>(new WarriorNpc\_t(WarriorTexture, level.bulletsList, clock.get(), defaultSpawnCoords, tempSizes.width, tempSizes.height)));

//MAGE TYPE NPC

std::shared\_ptr<sf::Texture> magesTexture = std::make\_shared<sf::Texture>();

magesTexture->loadFromFile(ENEMY\_MAGE\_FILE);

tempSizes = tiles::getSizesFromStr(ENEMY\_MAGE\_FILE);

npcTypesList.push\_back(std::unique\_ptr <character\_t>(new MageNpc\_t(magesTexture, level.bulletsList, clock.get(), defaultSpawnCoords, tempSizes.width, tempSizes.height)));

//ZOMBIE WITCH NPC ZombieWitch\_t

std::shared\_ptr<sf::Texture> zombiWitchTexture = std::make\_shared<sf::Texture>();

zombiWitchTexture->loadFromFile(ENEMY\_ZOMBIE\_WITCH\_TEXTURE\_FILE);

tempSizes = tiles::getSizesFromStr(ENEMY\_ZOMBIE\_WITCH\_TEXTURE\_FILE);

npcTypesList.push\_back(std::unique\_ptr <character\_t>(new ZombieWitch\_t(zombiWitchTexture, level.bulletsList, clock.get(), defaultSpawnCoords, tempSizes.width, tempSizes.height)));

//FAT ZOMBIE CLASS FatZombie\_t

std::shared\_ptr<sf::Texture> fatZombieTexture = std::make\_shared<sf::Texture>();

fatZombieTexture->loadFromFile(ENEMY\_ZOMBIE\_PUDGE\_TEXURE\_FILE);

tempSizes = tiles::getSizesFromStr(ENEMY\_ZOMBIE\_PUDGE\_TEXURE\_FILE);

npcTypesList.push\_back(std::unique\_ptr <character\_t>(new FatZombie\_t(fatZombieTexture, level.bulletsList, clock.get(), defaultSpawnCoords, tempSizes.width, tempSizes.height)));

//SKELETON MAGE CLASS SkeletonMage\_t

std::shared\_ptr<sf::Texture> skeletonMageTexture = std::make\_shared<sf::Texture>();

skeletonMageTexture->loadFromFile(ENEMY\_SKELETON\_MAGE\_TEXTURE\_FILE);

tempSizes = tiles::getSizesFromStr(ENEMY\_SKELETON\_MAGE\_TEXTURE\_FILE);

npcTypesList.push\_back(std::unique\_ptr <character\_t>(new SkeletonMage\_t(skeletonMageTexture, level.bulletsList, clock.get(), defaultSpawnCoords, tempSizes.width, tempSizes.height)));

//BOSSES

//TREANT

std::shared\_ptr<sf::Texture> treantTexture = std::make\_shared<sf::Texture>();

treantTexture->loadFromFile(BOSS\_TREANT\_TEXURE\_FILE);

tempSizes = tiles::getSizesFromStr(BOSS\_TREANT\_TEXURE\_FILE);

npcBossesTypeList.push\_back(std::unique\_ptr <BossNpc\_t>(new TeantBossNpc\_t(treantTexture, level.bulletsList, clock.get(), defaultSpawnCoords, tempSizes.width, tempSizes.height)));

//RED DRAGON

std::shared\_ptr<sf::Texture> redDragonTexture = std::make\_shared<sf::Texture>();

redDragonTexture->loadFromFile(BOSS\_RED\_DRAGON\_TEXURE\_FILE);

tempSizes = tiles::getSizesFromStr(BOSS\_RED\_DRAGON\_TEXURE\_FILE,3,3);

npcBossesTypeList.push\_back(std::unique\_ptr <BossNpc\_t>(new RedDragonBossNpc\_t(redDragonTexture, level.bulletsList, clock.get(), defaultSpawnCoords, tempSizes.width, tempSizes.height)));

//BLACK DRAGON

std::shared\_ptr<sf::Texture> blackDragonTexture = std::make\_shared<sf::Texture>();

blackDragonTexture->loadFromFile(BOSS\_BLACK\_DRAGON\_TEXURE\_FILE);

tempSizes = tiles::getSizesFromStr(BOSS\_BLACK\_DRAGON\_TEXURE\_FILE);

npcBossesTypeList.push\_back(std::unique\_ptr <BossNpc\_t>(new BlackDragonBossNpc\_t(blackDragonTexture, level.bulletsList, clock.get(), defaultSpawnCoords, tempSizes.width, tempSizes.height)));

//ENH

std::shared\_ptr<sf::Texture> enhTexture = std::make\_shared<sf::Texture>();

enhTexture->loadFromFile(BOSS\_ENH\_TEXURE\_FILE);

tempSizes = tiles::getSizesFromStr(BOSS\_ENH\_TEXURE\_FILE,3,2);

npcBossesTypeList.push\_back(std::unique\_ptr <BossNpc\_t>(new EnhBpssNpc\_t(enhTexture, level.bulletsList, clock.get(), defaultSpawnCoords, tempSizes.width, tempSizes.height)));

//DARK QUEEN

std::shared\_ptr<sf::Texture> darkQueenTexture = std::make\_shared<sf::Texture>();

darkQueenTexture->loadFromFile(BOSS\_FINALY\_DEMON\_TEXURE\_FILE);

tempSizes = tiles::getSizesFromStr(BOSS\_FINALY\_DEMON\_TEXURE\_FILE);

npcBossesTypeList.push\_back(std::unique\_ptr <BossNpc\_t>(new DQBossNpc\_t(darkQueenTexture, level.bulletsList, clock.get(), defaultSpawnCoords, tempSizes.width, tempSizes.height)));

}

void GameEngine\_t::generateNpc() {

size\_t NpcTypeAmount = npcTypesList.size();

size\_t tempCounter = 0;

size\_t NpcAmount = 30;

sf::Vector2f tempCoords;

//\*

for (auto &i : npcTypesList) {

tempCounter = 0;

while (tempCounter++ < NpcAmount) {

do {

tempCoords = generateRandomSpawnCoords(level.map.getSize());

} while (positionCollision(tempCoords));

level.charactersList.push\_back(std::unique\_ptr <character\_t>(new BossNpc\_t(i.get(), tempCoords,STD\_DIFFICULTY\_COEFFICIENT \* static\_cast<float>(difficulty))));

}

}

//\*/

}

void GameEngine\_t::generateBosses() {

size\_t NpcTypeAmount = npcTypesList.size();

size\_t bossCounter = 0;

size\_t NpcAmount = npcBossesTypeList.size();

sf::Vector2f tempCoords(500.f,500.f);

std::list<std::unique\_ptr<BossNpc\_t>>::iterator it = npcBossesTypeList.begin();

for (auto &i : npcBossesTypeList) {

tempCoords = level.map.bossesSpawnCoords[bossCounter++];

level.charactersList.push\_back(std::unique\_ptr <character\_t>(new BossNpc\_t(i.get(), tempCoords, STD\_DIFFICULTY\_COEFFICIENT \* static\_cast<float>(difficulty))));

}

level.bossesListIt = level.charactersList.end();

for (size\_t i = 0; i < level.mission.missionsCompleteStatus.size(); ++i) {

--level.bossesListIt;

}

}

void GameEngine\_t::update() {

if (!level.gameOver) {

if (level.checkLevelComplete()) {

status = game::status::WIN;

return;

}

checkAlive();

level.mission.setTime(clock.get());

bulletEngine();

visionEngine();

charsAction();

collisionEngine();

for (auto &character : level.charactersList) {

if (character->getState()->getStateNum() != -1) {

(character)->update(speed);

}

}

for (auto &bullet : level.bulletsList) {

bullet->update(speed);

}

}

else {

status = game::status::GAME\_OVER;

return;

}

}

void GameEngine\_t::charsAction() {

for (auto &i : level.charactersList) {

i.get()->getState()->Action();

}

}

void GameEngine\_t::draw() {

for (auto &texture : level.map.groundTilesList) {

window->draw(texture->getSprite());

}

for (auto &character : level.charactersList) {

window->draw(character->getSprite());

}

for (auto &ob : level.map.mapObList) {

window->draw(ob->getSprite());

}

for (auto &bullet : level.bulletsList) {

window->draw(bullet->getSprite());

}

}

void GameEngine\_t::checkAlive() {

std::list<std::unique\_ptr <bullet\_t>>::iterator tempOb = level.bulletsList.begin();

for (auto &bullet : level.bulletsList) {

if (!bullet->getAlive()) {

std::list<std::unique\_ptr <bullet\_t>>::iterator tempOb2 = tempOb++;

level.bulletsList.erase(tempOb2);

}

}

std::list<std::unique\_ptr <character\_t>>::iterator tempCharIter = level.charactersList.begin();

//for (int i = 0; i < level.charactersList.size(); ++i, ++tempCharIter) {}

for (int i = 0; tempCharIter != level.bossesListIt; ++i, ++tempCharIter){

if (tempCharIter != level.mainHero) {

if (!(\*tempCharIter)->getAlive()) {

level.charactersList.erase(tempCharIter);

level.getMission().ånemyKilled();

}

}

else if (!(\*tempCharIter)->getAlive()) {

level.gameOver = true;

return;

}

}

for (; tempCharIter != level.charactersList.end(); ++tempCharIter) {

if (!(\*tempCharIter)->getAlive()) {

level.checkMissionsTarget();

level.getMission().bossKilled();

if (tempCharIter == level.bossesListIt) {

++level.bossesListIt;

}

level.charactersList.erase(tempCharIter);

}

}

//\*/

}

void GameEngine\_t::bulletEngine() {

// Bullet collision

for (auto &outerElement : level.bulletsList) {

for (auto &innerElement : level.map.mapObList) {

if (outerElement->checkCollision(\*innerElement)) {

outerElement->collisionHandler(\*innerElement, speed);

}

}

for (auto &innerElement : level.charactersList) {

if ((outerElement->checkCollision(\*innerElement.get()))) {

outerElement->collisionHandler(\*innerElement.get(), speed);

if (outerElement->getFraction() != innerElement->getFraction() && innerElement->getAlive()) {

innerElement.get()->getState()->setTargetCharacter((outerElement.get()->getGenericObject()));

}

}

}

}

}

void GameEngine\_t::visionEngine() {

std::list<std::unique\_ptr <character\_t>>::iterator tempCharIter = level.charactersList.begin();

for (auto &outerElement : level.charactersList) {

if (tempCharIter++ != level.mainHero) {

for (auto &innerElement : level.charactersList) {

if ( (innerElement->getAlive() && outerElement->getFraction() != innerElement->getFraction() && (outerElement->checkEnemy(innerElement.get())))) {

outerElement->getState()->setTargetCharacter(innerElement.get());

}

}

}

}

}

void GameEngine\_t::collisionEngine() {

for (auto &outerElement : level.charactersList) {

for (auto &innerElement : level.map.mapObList) {

if (outerElement->checkCollision(\*innerElement)) {

outerElement->collisionHandler(\*innerElement, speed);

}

}

}

}

//GET

game::status GameEngine\_t::getGameStatus() const {

return status;

}

float GameEngine\_t::getSpeed()const {

return speed;

}

float GameEngine\_t::getCurTimeSec() const {

return clock->getElapsedTime().asSeconds();

}

float GameEngine\_t::getSpeedMultipple()const {

return speedMultipple;

}

std::list<std::unique\_ptr <bullet\_t>> &GameEngine\_t::getBulletList() {

return level.bulletsList;

}

//SET

float GameEngine\_t::setSpeed(float \_time) {

speed = \_time/speedMultipple;

return speed;

}

================================= “ground\_t.h” =================================

#pragma once

#include "SFML\Graphics.hpp"

#include <memory>

#include "physOb\_t.h"

#include "additional.h"

class ground\_t

{

std::shared\_ptr<sf::Texture> texture;

sf::Sprite sprite;

int spriteCoordX;

int spriteCoordY;

int height;

int width;

float posX;

float posY;

int id;

public:

ground\_t();

ground\_t(std::shared\_ptr<sf::Texture>\_texture,float \_posX, float \_posY, int \_x, int \_y, int \_id);

~ground\_t();

//GET

std::shared\_ptr<sf::Texture> getTexture() const;

sf::Sprite getSprite() const;

int getCoordX() const;

int getCoordY() const;

int getHeight() const;

int getWidth() const;

float getPosX() const;

float getPosY() const;

int getId() const;

};

================================= “ground\_t.cpp” =================================

#include "ground\_t.h"

ground\_t::ground\_t()

{

texture = nullptr;

int spriteCoordX = 0;

int spriteCoordY = 0;

int height = tiles::SPRITE\_HEIGHT;

int width = tiles::SPRITE\_WIDTH;

float posX = 0;

float posY = 0;

id = 0;

}

ground\_t::ground\_t(std::shared\_ptr<sf::Texture>\_texture, float \_posX, float \_posY, int \_x, int \_y, int \_id)

{

texture = std::shared\_ptr<sf::Texture>(\_texture);

sprite.setTexture(\*texture);

spriteCoordX = \_x;

spriteCoordY = \_y;

height = tiles::SPRITE\_HEIGHT;

width = tiles::SPRITE\_WIDTH;

posX = \_posX;

posY = \_posY;

id = \_id;

sprite.setTextureRect(sf::IntRect(spriteCoordX, spriteCoordY, width, height));

sprite.setPosition(posX, posY);

}

ground\_t::~ground\_t()

{

}

//GET

std::shared\_ptr<sf::Texture> ground\_t::getTexture() const {

return texture;

}

sf::Sprite ground\_t::getSprite() const {

return sprite;

}

int ground\_t::getCoordX() const {

return spriteCoordX;

}

int ground\_t::getCoordY() const {

return spriteCoordY;

}

int ground\_t::getHeight() const {

return height;

}

int ground\_t::getWidth() const {

return width;

}

float ground\_t::getPosX() const {

return posX;

}

float ground\_t::getPosY() const {

return posY;

}

int ground\_t::getId() const {

return id;

}

================================= “InterfaceBar.h” =================================

#pragma once

#include <SFML/Graphics.hpp>

#include <memory>

#include "character\_t.h"

#include "additional.h"

class InterfaceBar

{

protected:

sf::RenderWindow \*window;

sf::Vector2f borders;

std::unique\_ptr<sf::RectangleShape> outerRect;

std::unique\_ptr<sf::RectangleShape> innerRect;

sf::Vector2f outerRectSize;

sf::Vector2f innerRectSize;

sf::Vector2f posCoords;

sf::Color outerColor;

sf::Color innerColor;

bool display;

void updateInnerRectPos();

public:

InterfaceBar(sf::RenderWindow \*\_window);

virtual ~InterfaceBar();

virtual void draw();

virtual void update();

virtual void toDefaultPosition();

//GET

sf::Vector2f getOuterRectSize() const;

sf::Vector2f getInnerRectSize() const;

sf::Vector2f getPosCoords() const;

sf::Color getOuterColor() const;

sf::Color getInnerColor() const;

bool getDisplay() const ;

//SET

void setOuterRectSize(sf::Vector2f newSize);

void setInnerRectSize(sf::Vector2f newSize);

void setPosCoords(sf::Vector2f newCoords);

void setOuterRectColor(sf::Color newColor);

void setInnerRectColor(sf::Color newColor);

void setBorders(sf::Vector2f newBorders);

void serRelativeCoords(sf::Vector2f newCoords);

};

class progressBar : public InterfaceBar {

protected:

sf::Vector2f relativePos;

float &curValue;

float &maxValue;

public:

progressBar(sf::RenderWindow \*\_window, sf::Vector2f \_relativeCoords,float &\_curVal, float &\_maxValue);

virtual ~progressBar();

virtual void update();

virtual void toDefaultPosition();

void setRelativePos(sf::Vector2f newCoords);

};

class LifeBar : public InterfaceBar {

protected:

character\_t \*character;

public:

LifeBar(sf::RenderWindow \*\_window, character\_t \*character);

virtual ~LifeBar();

virtual void update();

virtual void toDefaultPosition();

//GET

};

class castTimeBar : public LifeBar {

public:

castTimeBar(sf::RenderWindow \*\_window, character\_t \*character);

virtual ~castTimeBar();

virtual void update();

virtual void toDefaultPosition();

};

================================= “InterfaceBar.cpp” =================================

#include "InterfaceBar.h"

InterfaceBar::InterfaceBar(sf::RenderWindow \*\_window)

{

window = \_window;

display = true;

borders = interface::STD\_BORDER\_SIZE;

outerRectSize = sf::Vector2f(interface::STD\_BAR\_SIZE);

innerRectSize = sf::Vector2f((outerRectSize.x - borders.x \* 2), (outerRectSize.y - borders.y \* 2));

outerRect.swap(std::unique\_ptr<sf::RectangleShape>(new sf::RectangleShape(outerRectSize)));

innerRect.swap(std::unique\_ptr<sf::RectangleShape>(new sf::RectangleShape(innerRectSize)));

posCoords = sf::Vector2f(0.f, 0.f);

sf::Vector2f innerRectPos((posCoords.x + borders.x), (posCoords.y + borders.y));

outerRect->setPosition(posCoords);

innerRect->setPosition(innerRectPos);

outerColor = sf::Color::Black;

innerColor = sf::Color::Red;

outerRect->setFillColor(outerColor);

innerRect->setFillColor(innerColor);

}

InterfaceBar::~InterfaceBar()

{

}

void InterfaceBar::updateInnerRectPos() {

innerRect->setPosition(sf::Vector2f((posCoords.x + borders.x), (posCoords.y + borders.y)));

}

void InterfaceBar::draw() {

window->draw(\*outerRect.get());

window->draw(\*innerRect.get());

}

void InterfaceBar::update() {

outerRect->setPosition(posCoords);

updateInnerRectPos();

}

void InterfaceBar::toDefaultPosition(){

outerRect->setPosition(posCoords);

updateInnerRectPos();

}

sf::Vector2f InterfaceBar::getOuterRectSize() const {

return outerRectSize;

}

sf::Vector2f InterfaceBar::getInnerRectSize() const {

return innerRectSize;

}

sf::Vector2f InterfaceBar::getPosCoords() const{

return posCoords;

}

sf::Color InterfaceBar::getOuterColor() const{

return outerColor;

}

sf::Color InterfaceBar::getInnerColor() const{

return innerColor;

}

bool InterfaceBar::getDisplay() const {

return display;

}

//SET

void InterfaceBar::setOuterRectSize(sf::Vector2f newSize) {

outerRect->setSize(newSize);

}

void InterfaceBar::setInnerRectSize(sf::Vector2f newSize) {

innerRect->setSize(newSize);

}

void InterfaceBar::setPosCoords(sf::Vector2f newCoords) {

posCoords = newCoords;

outerRect->setPosition(posCoords);

sf::Vector2f innerRectPos((posCoords.x + borders.x), (posCoords.y + borders.y));

innerRect->setPosition(innerRectPos);

}

void InterfaceBar::setOuterRectColor(sf::Color newColor) {

outerColor = newColor;

outerRect->setFillColor(outerColor);

}

void InterfaceBar::setInnerRectColor(sf::Color newColor) {

innerColor = newColor;

innerRect->setFillColor(innerColor);

}

void InterfaceBar::setBorders(sf::Vector2f newBorders) {

borders = newBorders;

}

void InterfaceBar::serRelativeCoords(sf::Vector2f newCoords) {

}

//PROGRESS BAR

progressBar::progressBar(sf::RenderWindow \*\_window, sf::Vector2f \_relativeCoords, float &\_curVal, float &\_maxValue) : InterfaceBar(\_window), curValue(\_curVal), maxValue(\_maxValue)

{

borders = interface::STD\_BORDER\_SIZE;

relativePos = \_relativeCoords;

outerRect.swap(std::unique\_ptr<sf::RectangleShape>(new sf::RectangleShape(outerRectSize)));

innerRect.swap(std::unique\_ptr<sf::RectangleShape>(new sf::RectangleShape(innerRectSize)));

toDefaultPosition();

sf::Vector2f innerRectPos((posCoords.x + borders.x), (posCoords.y + borders.y));

outerRect->setPosition(posCoords);

innerRect->setPosition(innerRectPos);

outerColor = sf::Color::Color(0, 0, 0, 190);

innerColor = sf::Color::Color(255, 0, 0, 255);

outerRect->setFillColor(outerColor);

innerRect->setFillColor(innerColor);

}

progressBar::~progressBar() {

}

void progressBar::update() {

toDefaultPosition();

float k = curValue / maxValue;

if (k < 0) {

k = 0;

}

sf::Vector2f tempInnerRectSize(innerRectSize.x \* k, innerRectSize.y);

setInnerRectSize(tempInnerRectSize);

}

void progressBar::toDefaultPosition() {

sf::Vector2f tempPos(interface::getScreenCoords(window));

tempPos += relativePos;

setPosCoords(tempPos);

}

void progressBar::setRelativePos(sf::Vector2f newCoords) {

relativePos = newCoords;

}

//LIFE BAR

LifeBar::LifeBar(sf::RenderWindow \*\_window, character\_t \*\_character) :InterfaceBar(\_window)

{

character = \_character;

borders = interface::STD\_BORDER\_SIZE;

float barSizeDiv = 8.f;

outerRectSize = sf::Vector2f(character->getWidth() + borders.x, interface::STD\_HP\_BAR\_HEIGHT.y);

innerRectSize = sf::Vector2f((outerRectSize.x - borders.x \* 2), (outerRectSize.y - borders.y \* 2));

outerRect.swap(std::unique\_ptr<sf::RectangleShape>(new sf::RectangleShape(outerRectSize)));

innerRect.swap(std::unique\_ptr<sf::RectangleShape>(new sf::RectangleShape(innerRectSize)));

toDefaultPosition();

sf::Vector2f innerRectPos((posCoords.x + borders.x), (posCoords.y + borders.y));

outerRect->setPosition(posCoords);

innerRect->setPosition(innerRectPos);

outerColor = sf::Color::Color(0, 0, 0, 190);

innerColor = sf::Color::Color(255,0,0,255);

outerRect->setFillColor(outerColor);

innerRect->setFillColor(innerColor);

}

LifeBar::~LifeBar()

{

}

void LifeBar::update() {

if (character->getAlive()) {

toDefaultPosition();

float k = character->getStats().HP / character->getStats().stdHP;

sf::Vector2f tempInnerRectSize(innerRectSize.x \* k, innerRectSize.y);

setInnerRectSize(tempInnerRectSize);

}

else {

display = false;

}

}

void LifeBar::toDefaultPosition() {

sf::Vector2f tempPos(character->getPosX() - borders.x, character->getPosY() - borders.y \* 5.f);

setPosCoords(tempPos);

}

//CAST BAR

castTimeBar::castTimeBar(sf::RenderWindow \*\_window, character\_t \*\_character) :LifeBar(\_window, \_character)

{

character = \_character;

borders = interface::STD\_BORDER\_SIZE;

float barSizeDiv = 7.f;

outerRectSize = sf::Vector2f(interface::STD\_BAR\_SIZE.x/2, interface::STD\_BAR\_SIZE.y);

innerRectSize = sf::Vector2f((outerRectSize.x - borders.x \* 2), (outerRectSize.y - borders.y \* 2));

outerRect.swap(std::unique\_ptr<sf::RectangleShape>(new sf::RectangleShape(outerRectSize)));

innerRect.swap(std::unique\_ptr<sf::RectangleShape>(new sf::RectangleShape(innerRectSize)));

toDefaultPosition();

sf::Vector2f innerRectPos((posCoords.x + borders.x), (posCoords.y + borders.y));

outerRect->setPosition(posCoords);

innerRect->setPosition(innerRectPos);

outerColor = sf::Color::Color(0, 0, 0, 100);

innerColor = sf::Color::Color(1,255,4,255);

outerRect->setFillColor(outerColor);

innerRect->setFillColor(innerColor);

}

castTimeBar::~castTimeBar()

{

}

void castTimeBar::update() {

if (character->getAlive()) {

toDefaultPosition();

if (character->getState()->getStateNum() == 4) {

int delay = character->getTimer().getCastDelay();;

float k = (static\_cast<float>(character->getTimer().getClockPtr()->getElapsedTime().asMilliseconds()) -

static\_cast<float>(character->getTimer().getCastStartTime())) / static\_cast<float>(delay);

sf::Vector2f tempInnerRectSize(innerRectSize.x \* k, innerRectSize.y);

setInnerRectSize(tempInnerRectSize);

}

else {

sf::Vector2f tempInnerRectSize(0.f, 0.f);

setInnerRectSize(tempInnerRectSize);

}

}

else {

display = false;

}

}

void castTimeBar::toDefaultPosition() {

sf::View view = window->getView();

sf::Vector2f tempPos(view.getCenter().x - outerRectSize.x/2 + interface::STD\_BORDER\_SIZE.x, view.getCenter().y + window->getSize().y / 2 - outerRectSize.y\*2 - interface::STD\_BORDER\_SIZE.y);

setPosCoords(tempPos);

}

================================= “InterfaceButton.h” =================================

#pragma once

#include <SFML/Graphics.hpp>

#include <memory>

#include "InterfaceWindow\_t.h"

const sf::Vector2f STD\_BUTTON\_SIZE(50.f, 50.f);

class InterfaceButton : public InterfaceWindow\_t

{

protected:

InterfaceWindow\_t \*linkedOb;

bool tracking;

public:

InterfaceButton(InterfaceWindow\_t &\_linkedOb, sf::Vector2f \_pos);

virtual ~InterfaceButton();

virtual void update();

virtual bool action() = 0;

bool trackDisplay();

};

class IntefaceToggleButton : public InterfaceButton {

public:

IntefaceToggleButton(InterfaceWindow\_t &\_linkedOb, sf::Vector2f \_pos);

~IntefaceToggleButton();

virtual bool action();

};

class InterfaceRestartButton : public InterfaceButton {

public:

InterfaceRestartButton(InterfaceWindow\_t &\_linkedOb, sf::Vector2f \_pos);

~InterfaceRestartButton();

virtual bool action();

};

class IntefaceExitButton : public InterfaceButton {

public:

IntefaceExitButton(InterfaceWindow\_t &\_linkedOb, sf::Vector2f \_pos);

~IntefaceExitButton();

virtual bool action();

};

================================= “InterfaceButton.cpp” =================================

#include "InterfaceButton.h"

InterfaceButton::InterfaceButton(InterfaceWindow\_t &\_linkedOb, sf::Vector2f \_pos):InterfaceWindow\_t(\_linkedOb.getWindowPtr(), \_pos, STD\_BUTTON\_SIZE),

linkedOb(&\_linkedOb)

{

clickable = true;

borders =sf::Vector2f(2.f, 2.f);

pos = \_pos;

text.setCharacterSize(20);

title = "X";

text.setPosition(pos);

rectangle.setSize(sizes);

rectangle.setFillColor(sf::Color::Black);

rectangle.setPosition(pos);

tracking = false;

}

InterfaceButton::~InterfaceButton()

{

}

bool InterfaceButton::trackDisplay() {

display = linkedOb->getDisplayState();

return display;

}

void InterfaceButton::update() {

InterfaceWindow\_t::update();

if (tracking) {

trackDisplay();

}

}

//TOGGLE BUTTON CLASS

IntefaceToggleButton::IntefaceToggleButton(InterfaceWindow\_t &\_linkedOb, sf::Vector2f \_pos):InterfaceButton(\_linkedOb, \_pos)

{

setBgColor(sf::Color::Transparent);

setBorderColor(sf::Color::Transparent);

}

IntefaceToggleButton::~IntefaceToggleButton()

{

}

bool IntefaceToggleButton::action() {

return linkedOb->toggleDisplay();

}

//RESTART BUTTON CLASS

InterfaceRestartButton::InterfaceRestartButton(InterfaceWindow\_t &\_linkedOb, sf::Vector2f \_pos) :InterfaceButton(\_linkedOb, \_pos)

{

tracking = true;

setBgColor(sf::Color::Transparent);

setBorderColor(sf::Color::Transparent);

//setBgColor(sf::Color::Blue);

}

InterfaceRestartButton::~InterfaceRestartButton()

{

}

bool InterfaceRestartButton::action() {

return true;

}

//EXIT BUTTON CLASS

IntefaceExitButton::IntefaceExitButton(InterfaceWindow\_t &\_linkedOb, sf::Vector2f \_pos):InterfaceButton(\_linkedOb, \_pos)

{

tracking = true;

setBgColor(sf::Color::Transparent);

setBorderColor(sf::Color::Transparent);

}

IntefaceExitButton::~IntefaceExitButton()

{

}

bool IntefaceExitButton::action() {

return true;

}

================================= “InterfaceContent.h” =================================

#pragma once

#include <SFML/Graphics.hpp>

#include <memory>

#include "additional.h"

class InterfaceContent

{

protected:

sf::RenderWindow \*window;

sf::Vector2f defaultCoords;

sf::Vector2f relativePos;

sf::Vector2u sizes;

sf::Font font;

unsigned int fontSize;

sf::Color textColor;

sf::Text::Style textStyle;

public:

InterfaceContent(sf::RenderWindow \*\_window, sf::Vector2f \_defaultCoords, sf::Vector2f \_relativePos = sf::Vector2f(0.f,0.f));

virtual ~InterfaceContent();

virtual void update() = 0;

virtual void draw() = 0;

virtual void toDefaultPosition() = 0;

virtual void swapContent(std::string \_newPath = "") = 0;

virtual void resetContent() = 0;

virtual void setTexture(sf::Texture \*newTexture);

virtual void setText(std::string newStr);

//GET

sf::Vector2u getSize() const;

sf::Vector2f getRelPos()const;

unsigned int getFontSize() const;

//SET

void setRelativePos(sf::Vector2f newCoords);

virtual void setSize(sf::Vector2u newSizes);

void setFont(sf::Font newFont);

virtual void setFontSize(unsigned int newSize);

void setFontColor(sf::Color newColor);

void setFontStyle(sf::Text::Style newStyle);

};

//SPRITE CONTENT

class InterfaceSpriteContent\_t : public InterfaceContent {

public:

std::unique\_ptr<sf::Texture> texture;

sf::Sprite sprite;

public:

InterfaceSpriteContent\_t(sf::RenderWindow \*\_window, sf::Texture \*\_texture, sf::Vector2f \_defaultCoords, sf::Vector2f \_relativePos = sf::Vector2f(0.f, 0.f));

virtual ~InterfaceSpriteContent\_t();

virtual void update();

virtual void draw();

virtual void toDefaultPosition();

virtual void swapContent(std::string \_newPath = "");

virtual void resetContent();

//GET

sf::Sprite getSprite() const;

//SET

void setTexture(sf::Texture \*newTexture);

virtual void setSize(sf::Vector2u newSizes);

};

//TEXT CONTENT

class InterfaceTextContent\_t : public InterfaceContent {

public:

std::string str;

std::unique\_ptr<sf::Text> text;

void setText(sf::Text \*\_newText);

sf::Text getText();

public:

InterfaceTextContent\_t(sf::RenderWindow \*\_window, std::string \_str, sf::Vector2f \_defaultCoords, sf::Vector2f \_relativePos = sf::Vector2f(0.f, 0.f));

virtual ~InterfaceTextContent\_t();

void setText(std::string newStr);

virtual void update();

virtual void draw();

virtual void toDefaultPosition();

virtual void setFontSize(unsigned int newSize);

virtual void swapContent(std::string \_str = "");

virtual void resetContent();

};

================================= “InterfaceContent.cpp” ================================

#include "InterfaceContent.h"

InterfaceContent::InterfaceContent(sf::RenderWindow \*\_window, sf::Vector2f \_defaultCoords, sf::Vector2f \_relativePos)

{

window = \_window;

defaultCoords = \_defaultCoords;

relativePos = \_relativePos;

font.loadFromFile(textSettings::STD\_FONT\_FILE);

fontSize = textSettings::STD\_FONT\_SIZE;

textColor = textSettings::STD\_TEXT\_COLOR;

textStyle = textSettings::STD\_TEXT\_STYLE;

}

InterfaceContent::~InterfaceContent()

{

}

void InterfaceContent::setTexture(sf::Texture \*newTexture){

}

void InterfaceContent::setText(std::string newStr) {

}

//GET

sf::Vector2u InterfaceContent::getSize() const {

return sizes;

}

sf::Vector2f InterfaceContent::getRelPos()const {

return relativePos;

}

unsigned int InterfaceContent::getFontSize() const {

return fontSize;

}

//SET

void InterfaceContent::setRelativePos(sf::Vector2f newCoords) {

relativePos = newCoords;

}

void InterfaceContent::setSize(sf::Vector2u newSizes) {

sizes = newSizes;

}

void InterfaceContent::setFont(sf::Font newFont) {

font = newFont;

}

void InterfaceContent::setFontSize(unsigned int newSize){

fontSize = newSize;

}

void InterfaceContent::setFontColor(sf::Color newColor) {

textColor = newColor;

}

void InterfaceContent::setFontStyle(sf::Text::Style newStyle) {

textStyle = newStyle;

}

//SPRITE CONTENT

InterfaceSpriteContent\_t::InterfaceSpriteContent\_t(sf::RenderWindow \*\_window, sf::Texture \*\_texture, sf::Vector2f \_defaultCoords, sf::Vector2f \_relativePos):

InterfaceContent(\_window,\_defaultCoords, \_relativePos)

{

setTexture(\_texture);

sizes = texture->getSize();

sprite.setPosition(defaultCoords + relativePos);

}

InterfaceSpriteContent\_t::~InterfaceSpriteContent\_t() {

}

void InterfaceSpriteContent\_t::setTexture(sf::Texture \*newTexture) {

texture.get\_deleter();

texture.reset();

texture.swap(std::unique\_ptr<sf::Texture>(newTexture));

sprite.setTexture(\*texture);

sizes = texture->getSize();

sprite.setTextureRect(sf::IntRect(0, 0, sizes.x, sizes.y));

}

void InterfaceSpriteContent\_t::swapContent(std::string \_newPath) {

sf::Texture \*newTempTexture = new sf::Texture;

newTempTexture->loadFromFile(\_newPath);

setTexture(newTempTexture);

}

void InterfaceSpriteContent\_t::update() {

toDefaultPosition();

}

void InterfaceSpriteContent\_t::draw() {

window->draw(getSprite());

}

void InterfaceSpriteContent\_t::toDefaultPosition() {

sf::Vector2f tempPos(interface::getScreenCoords(window));

sprite.setPosition(tempPos + defaultCoords + relativePos);

}

void InterfaceSpriteContent\_t::resetContent() {

texture.get\_deleter();

}

//GET

sf::Sprite InterfaceSpriteContent\_t::getSprite() const {

return sprite;

}

//SET

void InterfaceSpriteContent\_t::setSize(sf::Vector2u newSizes) {

sizes = newSizes;

sprite.setTextureRect(sf::IntRect(0, 0, sizes.x, sizes.y));

}

//TEXT CONTENT

InterfaceTextContent\_t::InterfaceTextContent\_t(sf::RenderWindow \*\_window, std::string \_str, sf::Vector2f \_defaultCoords, sf::Vector2f \_relativePos):

InterfaceContent(\_window, \_defaultCoords, \_relativePos)

{

str = \_str;

text.swap(std::unique\_ptr<sf::Text>(new sf::Text(\_str, font, fontSize)));

text->setFillColor(textColor);

text->setStyle(textStyle);

}

InterfaceTextContent\_t::~InterfaceTextContent\_t() {

}

sf::Text InterfaceTextContent\_t::getText() {

return \*text.get();

}

void InterfaceTextContent\_t::setText(sf::Text \*\_newText) {

text.get\_deleter();

text.swap(std::unique\_ptr<sf::Text>(\_newText));

str = \_newText->getString();

}

void InterfaceTextContent\_t::swapContent(std::string \_newPath) {

font.loadFromFile(textSettings::STRIKE\_THROUGHT\_FONT\_FILE);

text->setFont(font);

text->setCharacterSize(fontSize);

setFontColor(textColor);

}

void InterfaceTextContent\_t::update() {

toDefaultPosition();

text->setFillColor(textColor);

text->setFont(font);

text->setString(str);

}

void InterfaceTextContent\_t::draw() {

window->draw(getText());

}

void InterfaceTextContent\_t::toDefaultPosition() {

sf::Vector2f tempPos(interface::getScreenCoords(window));

text->setPosition(tempPos + defaultCoords + relativePos);

}

void InterfaceTextContent\_t::resetContent() {

text.get\_deleter();

}

void InterfaceTextContent\_t::setText(std::string newStr) {

str = newStr;

}

void InterfaceTextContent\_t::setFontSize(unsigned int newSize) {

fontSize = newSize;

text->setCharacterSize(fontSize);

}

================================= “InterfaceEngine.h” =================================

#pragma once

#include <SFML\Graphics.hpp>

#include <list>

#include <map>

#include <iterator>

#include <memory>

#include "Level\_t.h"

#include "InterfaceWindow\_t.h"

#include "InterfaceButton.h"

#include "InterfaceBar.h"

#include "cursor\_t.h"

typedef std::unique\_ptr<InterfaceBar> bar\_t;

typedef std::unique\_ptr<InterfaceWindow\_t> window\_t;

typedef std::unique\_ptr<InterfaceButton> button;

class InterfaceEngine\_t

{

sf::RenderWindow \*window;

Level\_t &level;

std::unique\_ptr<sf::View> view;

//Iterators

std::list<window\_t>::iterator elemIt;

std::list<window\_t>::iterator missionWindowIt;

std::list<window\_t>::iterator gameStatsWindowIt;

std::list<window\_t>::iterator mapIt;

std::list<window\_t>::iterator skillDescriptionWindowIt;

void drawCursor();

void setCamera();

public:

std::map<std::string, std::list<button>::iterator> buttonsMap;

std::list<button>::iterator restartButton;

std::list<button>::iterator backTomainMenuButton;

std::list<window\_t>::iterator pausedMenuIt;

std::list<window\_t>::iterator winWindowIt;

std::unique\_ptr<cursor\_t> cursor;

//LISTS

std::list<bar\_t> barsList;

std::list<window\_t> windowsList;

std::list<button> buttonList;

InterfaceEngine\_t(sf::RenderWindow \*\_window, Level\_t &\_level);

~InterfaceEngine\_t();

void update();

void draw();

void generateHPbars();

void setObservedBards();

void createSkillGeneratorIterface();

void createJournalWindow();

void createGameStatsWindow();

void createMapWindow();

void createInterfaceButtons();

void createPausedMenu();

void createDescriptionMenu();

void createWinScreen();

void updateGenerator();

void updateMissionJournal();

void updateGameStats();

void updateMapWindow();

bool pause();

};

namespace icon {

//ICONS

//const std::string ICON\_ELEMENT\_\_FILE = "img/icons/.png";

const std::string ICON\_ELEMENT\_FIRE\_FILE = "img/icons/icon\_element\_fire.png";

const std::string ICON\_ELEMENT\_WIND\_FILE = "img/icons/icon\_element\_wind.png";

const std::string ICON\_ELEMENT\_EARTH\_FILE = "img/icons/icon\_element\_earth.png";

//SKILLS ICONS

const std::string ICON\_SKILL\_BANG\_BALL\_FILE = "img/icons/icon\_skill\_bang\_ball.png";

const std::string ICON\_SKILL\_COMBO\_BALL\_FILE = "img/icons/icon\_skill\_combo\_ball.png";

const std::string ICON\_SKILL\_sWORD\_ATTACK\_FILE = "img/icons/icon\_skill\_common\_attack.png";

const std::string ICON\_SKILL\_EARTH\_BALL\_FILE = "img/icons/icon\_skill\_earth\_ball.png";

const std::string ICON\_SKILL\_FIRE\_BALL\_FILE = "img/icons/icon\_skill\_fire\_ball.png";

const std::string ICON\_SKILL\_HEAL\_BALL\_FILE = "img/icons/icon\_skill\_heal\_ball.png";

const std::string ICON\_SKILL\_LAVA\_POOL\_FILE = "img/icons/icon\_skill\_lava\_pool.png";

const std::string ICON\_SKILL\_POWER\_UP\_FILE = "img/icons/icon\_skill\_powerUp.png";

const std::string ICON\_SKILL\_RUSH\_FILE = "img/icons/icon\_skill\_rush.png";

const std::string ICON\_SKILL\_SMALL\_EARTH\_BALLS\_FILE = "img/icons/icon\_skill\_small\_earth\_balls.png";

const std::string ICON\_SKILL\_SMALL\_FIRE\_BALLS\_FILE = "img/icons/icon\_skill\_small\_fire\_balls.png";

const std::string ICON\_SKILL\_\_FILE = "img/icons/.png";

//BUTTONS ICONS

const std::string ICON\_BUTTON\_JOURNAL = "img/icons/icon\_button\_journal.png";

const std::string ICON\_BUTTON\_GAME\_STATISTIC = "img/icons/icon\_button\_game\_statistic.png";

const std::string ICON\_BUTTON\_SKILLS = "img/icons/icon\_button\_skills.png";

const std::string ICON\_BUTTON\_MAP = "img/icons/icon\_button\_map.png";

//SKILL DESCRIPTION ICONS

const int ICON\_DESC\_HEIGHT = 32;

const int ICON\_DESC\_WIDTH = 32;

const std::string DESC\_ICON\_ELEMENT\_FIRE\_FILE = "img/icons/description\_icons/icon\_element\_fire.png";

const std::string DESC\_ICON\_ELEMENT\_WIND\_FILE = "img/icons/description\_icons/icon\_element\_wind.png";

const std::string DESC\_ICON\_ELEMENT\_EARTH\_FILE = "img/icons/description\_icons/icon\_element\_earth.png";

const int DESC\_ICON\_ELEMENT\_SIZE = 32;

const std::string ICON\_HEAL\_111 = "img/icons/description\_icons/111.png";

const std::string ICON\_BANG\_112 = "img/icons/description\_icons/112.png";

const std::string ICON\_STONE\_113 = "img/icons/description\_icons/113.png";

const std::string ICON\_COMBO\_123 = "img/icons/description\_icons/123.png";

const std::string ICON\_FIRE\_BREATH\_221 = "img/icons/description\_icons/221.png";

const std::string ICON\_POWER\_BUFF\_222 = "img/icons/description\_icons/222.png";

const std::string ICON\_FIRE\_BALL\_223 = "img/icons/description\_icons/223.png";

const std::string ICON\_SMALL\_EARTH\_BALLS\_331 = "img/icons/description\_icons/331.png";

const std::string ICON\_FIRE\_LIGHTING\_332 = "img/icons/description\_icons/332.png";

const std::string ICON\_SPEED\_BUFF\_333 = "img/icons/description\_icons/333.png";

}

================================= “InterfaceEngine.cpp” =================================

#include "InterfaceEngine\_t.h"

InterfaceEngine\_t::InterfaceEngine\_t(sf::RenderWindow \*\_window, Level\_t &\_level) :level(\_level)

{

window = \_window;

view.swap(std::unique\_ptr <sf::View>(new sf::View));

view->reset(sf::FloatRect(0, 0, static\_cast<float>(window->getSize().x), static\_cast<float>(window->getSize().y)));

generateHPbars();

setObservedBards();

createSkillGeneratorIterface();

createJournalWindow();

createGameStatsWindow();

createDescriptionMenu();

createMapWindow();

createWinScreen();

createPausedMenu();

createInterfaceButtons();

cursor.swap(std::unique\_ptr<cursor\_t>(new cursor\_t("img/cursor\_aim.png", 20, 20, window)));

}

InterfaceEngine\_t::~InterfaceEngine\_t()

{

window->setView(window->getDefaultView());

window->setMouseCursorVisible(true);

}

void InterfaceEngine\_t::drawCursor() {

window->draw(cursor->getSprite());

}

void InterfaceEngine\_t::update() {

setCamera();//set Camera

window->setView(\*view); // Set camera

updateMissionJournal();

updateGenerator();

if (missionWindowIt->get()->getDisplayState()) {

updateMissionJournal();

}

if (gameStatsWindowIt->get()->getDisplayState()) {

updateGameStats();

}

if (mapIt->get()->getDisplayState()) {

updateMapWindow();

}

level.mainHero->get()->setTargetPos(cursor->getPosition());

for (auto &i = barsList.begin(); i != barsList.end(); ++i) {

if (i->get()->getDisplay()) {

i->get()->update();

}

else {

barsList.erase(i);

}

}

for (auto &i : windowsList) {

i->update();

}

for (auto &i : buttonList) {

i->update();

}

cursor->setCursorPosition();

}

void InterfaceEngine\_t::draw() {

for (auto &i : barsList) {

i->draw();

}

for (auto &i : windowsList) {

i->draw();

}

for (auto &i : buttonList) {

i->draw();

}

drawCursor();

}

//INTERFACE BUILD

void InterfaceEngine\_t::generateHPbars() {

for (auto &i : level.charactersList) {

barsList.push\_back(bar\_t(new LifeBar(window, i.get())));

}

}

void InterfaceEngine\_t::setObservedBards() {

barsList.push\_back(bar\_t(new castTimeBar(window, level.charactersList.begin()->get())));

float tempBarCounter = .5f;

sf::Vector2f tempMargin(interface::STD\_BORDER\_SIZE);

barsList.push\_back(bar\_t(new progressBar(window, tempMargin, level.charactersList.begin()->get()->getStats().HP, level.charactersList.begin()->get()->getStats().stdHP)));

++tempBarCounter;

tempMargin.y += interface::STD\_BAR\_SIZE.y;

barsList.push\_back(bar\_t(new progressBar(window, tempMargin, level.charactersList.begin()->get()->getStats().MP, level.charactersList.begin()->get()->getStats().stdMP)));

barsList.back()->setInnerRectColor(sf::Color::Blue);

}

//SKILL GENERATOR INTERFACE

void InterfaceEngine\_t::createSkillGeneratorIterface() {

//SKILL GENERATOR ELEMENTS STATUS ICONS

sf::Vector2f tempPos(interface::getScreenCoords(window));

size\_t elemWindowAmount = elements::SKILL\_ELEMENT\_AMOUNT;

float betweenCorection = 7.f;

float rightWindowPadding = 60.f;

float bottomWindowPadding = 10.f;

while (elemWindowAmount-- > 0) {

float multipleCorection = static\_cast<float>(elemWindowAmount);

float positionCorrectionX = (window->getSize().x - multipleCorection \* interface::STD\_ELEMENT\_GENERATOR\_WINDOW\_SIZE.x - multipleCorection \* betweenCorection - rightWindowPadding);

float positionCorrectionY = (window->getSize().y - interface::STD\_ELEMENT\_GENERATOR\_WINDOW\_SIZE.y - bottomWindowPadding);

sf::Vector2f coordCorection(positionCorrectionX, positionCorrectionY);

windowsList.push\_back(window\_t(new InterfaceWindow\_t(window, (tempPos + coordCorection), interface::STD\_ELEMENT\_GENERATOR\_WINDOW\_SIZE)));

auto &i = windowsList.back();

i->setBgColor(sf::Color::Color(50, 50, 50, 150));

i->setBorderColor(sf::Color::Color(238, 238, 238, 150));

}

elemIt = windowsList.end();

for (size\_t i = 0; i < elements::SKILL\_ELEMENT\_AMOUNT; ++i) {

--elemIt;

}

//SKILL ICON WINDOW

rightWindowPadding = 75.f;

bottomWindowPadding = 80.f;

float positionCorrectionX = (window->getSize().x - interface::STD\_ELEMENT\_GENERATOR\_WINDOW\_SIZE.x - rightWindowPadding);

float positionCorrectionY = (window->getSize().y - interface::STD\_ELEMENT\_GENERATOR\_WINDOW\_SIZE.y - bottomWindowPadding);

sf::Vector2f coordCorection(positionCorrectionX, positionCorrectionY);

windowsList.push\_back(window\_t(new InterfaceWindow\_t(window, (tempPos + coordCorection), interface::STD\_SKILL\_WINDOW\_SIZE)));

sf::Texture \*temp = new sf::Texture;

windowsList.back().get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, temp, windowsList.back()->getPos(), sf::Vector2f(0.f, 0.f))));

windowsList.back().get()->setBgColor(sf::Color::Color(50, 50, 50, 150));

windowsList.back().get()->setBorderColor(sf::Color::Color(238, 238, 238, 150));

}

void InterfaceEngine\_t::updateGenerator() {

std::list<elements::element> &ElemsArr = level.mainHero->get()->getElements();

std::list<window\_t>::iterator tempElemIt = elemIt;

for (auto &i : ElemsArr) {

using namespace elements;

switch (i) {

case FIRE: {

tempElemIt->get()->contentList.clear();

sf::Texture \*newTempTexture = new sf::Texture;

newTempTexture->loadFromFile(icon::ICON\_ELEMENT\_FIRE\_FILE);

tempElemIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, newTempTexture, tempElemIt->get()->getPos(), sf::Vector2f(0.f, 0.f))));

break;

}

case WIND: {

tempElemIt->get()->contentList.clear();

sf::Texture \*newTempTexture = new sf::Texture;

newTempTexture->loadFromFile(icon::ICON\_ELEMENT\_WIND\_FILE);

tempElemIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, newTempTexture, tempElemIt->get()->getPos(), sf::Vector2f(0.f, 0.f))));

break;

}

case EARTH: {

tempElemIt->get()->contentList.clear();

sf::Texture \*newTempTexture = new sf::Texture;

newTempTexture->loadFromFile(icon::ICON\_ELEMENT\_EARTH\_FILE);

tempElemIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, newTempTexture, tempElemIt->get()->getPos(), sf::Vector2f(0.f, 0.f))));

break;

}

case NONE: {

tempElemIt->get()->contentList.clear();

break;

}

}

++tempElemIt;

}

//\*

switch (level.mainHero->get()->getElemStatus()) {

case 3: {

tempElemIt->get()->contentList.begin()->get()->swapContent(icon::ICON\_SKILL\_RUSH\_FILE);

break;

}

case 6: {

tempElemIt->get()->contentList.begin()->get()->swapContent(icon::ICON\_SKILL\_POWER\_UP\_FILE);

break;

}

case 24: {

tempElemIt->get()->contentList.begin()->get()->swapContent(icon::ICON\_SKILL\_HEAL\_BALL\_FILE);

break;

}

case 1: {

tempElemIt->get()->contentList.begin()->get()->swapContent(icon::ICON\_SKILL\_sWORD\_ATTACK\_FILE);

break; }

case 4: {

tempElemIt->get()->contentList.begin()->get()->swapContent(icon::ICON\_SKILL\_SMALL\_FIRE\_BALLS\_FILE);

break; }

case 10: {

tempElemIt->get()->contentList.begin()->get()->swapContent(icon::ICON\_SKILL\_SMALL\_EARTH\_BALLS\_FILE);

break; }

case 5: {

tempElemIt->get()->contentList.begin()->get()->swapContent(icon::ICON\_SKILL\_FIRE\_BALL\_FILE);

break; }

case 17: {

tempElemIt->get()->contentList.begin()->get()->swapContent(icon::ICON\_SKILL\_EARTH\_BALL\_FILE);

break; }

case 12: {

tempElemIt->get()->contentList.begin()->get()->swapContent(icon::ICON\_SKILL\_LAVA\_POOL\_FILE);

break; }

case 18: {

tempElemIt->get()->contentList.begin()->get()->swapContent(icon::ICON\_SKILL\_BANG\_BALL\_FILE);

break; }

case 11: {

tempElemIt->get()->contentList.begin()->get()->swapContent(icon::ICON\_SKILL\_COMBO\_BALL\_FILE);

break;

}

default: {

tempElemIt->get()->contentList.begin()->get()->resetContent();

break;

}

}

//\*/

}

//MISSION JOURNAL INTERFACE

void InterfaceEngine\_t::createJournalWindow() {

size\_t contentStringsAmount = level.getMission().missionsContent.size();

//MAIN MISSION WINDOW

sf::Vector2f windowPosition(STD\_BUTTON\_SIZE.x + interface::STD\_WINDOW\_MARGIN\_SIZE.x, window->getSize().y / 4.f);

sf::Vector2f windowSize(findBigestLength(level.getMission().missionsContent) \* textSettings::STD\_FONT\_SIZE / 1.5f + interface::STD\_BORDER\_SIZE.x, (contentStringsAmount + 1) \* (textSettings::STD\_FONT\_SIZE + interface::STD\_BORDER\_SIZE.y));

windowsList.push\_back(window\_t(new InterfaceWindow\_t(window, windowPosition, windowSize)));

missionWindowIt = windowsList.end();

--missionWindowIt;

windowsList.back()->setTitle("MISSION JOURNAL");

windowsList.back()->setDisplay(false);

missionWindowIt->get()->setBgColor(sf::Color::Color(50, 50, 50, 80));

missionWindowIt->get()->setBorderColor(sf::Color::Color(238, 238, 238, 80));

sf::Vector2f contentPos(0.f, 0.f);

contentPos.x = textSettings::STD\_FONT\_SIZE + interface::STD\_BORDER\_SIZE.x;

//MISSIONS LIST

for (size\_t i = 0; i < contentStringsAmount; ++i) {

contentPos.y = static\_cast<float>((i + 1) \* textSettings::STD\_FONT\_SIZE) + interface::STD\_BORDER\_SIZE.x;

windowsList.back()->contentList.push\_back(content(new InterfaceTextContent\_t(window, level.getMission().missionsContent[i], windowsList.back()->getPos(), contentPos)));

}

}

void InterfaceEngine\_t::updateMissionJournal() {

std::vector<content>::iterator tempIt = missionWindowIt->get()->contentList.begin();

for (size\_t i = 0; i < missionWindowIt->get()->contentList.size(); ++i) {

if (level.getMission().missionsCompleteStatus[i]) {

tempIt->get()->swapContent();

}

++tempIt;

}

}

void InterfaceEngine\_t::createWinScreen() {

sf::Texture \*winScreenTexture = new sf::Texture();

winScreenTexture->loadFromFile(interface::WIN\_CREEN\_TEXTURE\_FILE);

sf::Vector2f windowSize(window->getSize());

sf::Vector2f windowPosition(windowSize.x/2 - winScreenTexture->getSize().x/2, 0.f);

windowsList.push\_back(window\_t(new InterfaceWindow\_t(window,sf::Vector2f(0.f,0.f),windowSize)));

winWindowIt = windowsList.end();

--winWindowIt;

winWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window,winScreenTexture,windowPosition)));

winWindowIt->get()->setDisplay(false);

}

//MISSION STATISTIC INTERFACE

void InterfaceEngine\_t::createGameStatsWindow() {

size\_t contentStringsAmount = level.getMission().gameStats.statFields.size();

//MAIN MISSION WINDOW

sf::Vector2f windowPosition(STD\_BUTTON\_SIZE.x, windowsList.back()->getPos().y + windowsList.back()->getSizes().y);

sf::Vector2f windowSize(findBigestLength(level.getMission().gameStats.statFields) \* textSettings::STD\_FONT\_SIZE / 1.2f + interface::STD\_BORDER\_SIZE.x, (contentStringsAmount + 1) \* (textSettings::STD\_FONT\_SIZE + interface::STD\_BORDER\_SIZE.y));

windowPosition += interface::STD\_WINDOW\_MARGIN\_SIZE;

windowsList.push\_back(window\_t(new InterfaceWindow\_t(window, windowPosition, windowSize)));

gameStatsWindowIt = windowsList.end();

--gameStatsWindowIt;

windowsList.back()->setDisplay(false);

windowsList.back()->setTitle("STATISTIC");

gameStatsWindowIt->get()->setBgColor(sf::Color::Color(50, 50, 50, 80));

gameStatsWindowIt->get()->setBorderColor(sf::Color::Color(238, 238, 238, 80));

sf::Vector2f contentPos(0.f, 0.f);

contentPos.x = textSettings::STD\_FONT\_SIZE + interface::STD\_BORDER\_SIZE.x;

//STATS LIST

for (size\_t i = 0; i < contentStringsAmount; ++i) {

contentPos.y = static\_cast<float>((i + 1) \* textSettings::STD\_FONT\_SIZE) + interface::STD\_BORDER\_SIZE.x;

std::string tempStr = (level.getMission().gameStats.statFields[i] + level.getMission().gameStats.statStrValues[i]);

windowsList.back()->contentList.push\_back(content(new InterfaceTextContent\_t(window, tempStr, windowsList.back()->getPos(), contentPos)));

}

}

void InterfaceEngine\_t::updateGameStats() {

level.getMission().gameStats.fillStrStats();

for (size\_t i = 0; i < level.getMission().gameStats.statFields.size(); ++i) {

std::string tempStr = (level.getMission().gameStats.statFields[i] + level.getMission().gameStats.statStrValues[i]);

gameStatsWindowIt->get()->contentList[i]->setText(tempStr);

}

}

void InterfaceEngine\_t::createMapWindow() {

sf::Texture \*miniMapTexure(new sf::Texture);

std::string mapFileName = "img/maps/" + level.map.getLvlName() + ".png";

miniMapTexure->loadFromFile(mapFileName);

sf::Vector2f mapSize(miniMapTexure->getSize());

sf::Vector2f mapPosition(window->getSize().x/2 - mapSize.x /2 , window->getSize().y / 2 - mapSize.y / 2);

windowsList.push\_back(window\_t(new InterfaceWindow\_t(window, mapPosition, mapSize)));

mapIt = windowsList.end();

--mapIt;

mapIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window,miniMapTexure,mapIt->get()->getPos())));

mapIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, ".", mapIt->get()->getPos())));

unsigned int characterSize = 80;

mapIt->get()->contentList.back()->setFontSize(characterSize);

mapIt->get()->toggleDisplay();

}

void InterfaceEngine\_t::updateMapWindow() {

sf::Vector2f mapk;

mapk.x = mapIt->get()->getSizes().x / level.map.getFloatSize().x;

mapk.y = mapIt->get()->getSizes().y / level.map.getFloatSize().y;

sf::Vector2f posCharOnMap (level.mainHero->get()->getPos().x \* mapk.x, level.mainHero->get()->getPos().y \* mapk.y);

float k = static\_cast<float>(mapIt->get()->contentList.back()->getFontSize());

sf::Vector2f posCorrection(k/4.f, k);

mapIt->get()->contentList.back()->setRelativePos(posCharOnMap - posCorrection);

}

void InterfaceEngine\_t::createPausedMenu() {

sf::Vector2f windowSize(static\_cast<float>(window->getSize().x / 2), static\_cast<float>(window->getSize().y / 2));

sf::Vector2f windowPosition(static\_cast<float>(window->getSize().x / 2) - windowSize.x / 2.f, static\_cast<float>(window->getSize().y / 2) - windowSize.y / 2.f);

windowsList.push\_back(window\_t(new InterfaceWindow\_t(window, windowPosition, windowSize)));

pausedMenuIt = windowsList.end();

--pausedMenuIt;

pausedMenuIt->get()->setBgColor(sf::Color::Color(50, 50, 50, 150));

pausedMenuIt->get()->setBorderColor(sf::Color::Color(238, 238, 238, 150));

pausedMenuIt->get()->setDisplay(false);

sf::Vector2f statusMassageSize(windowSize.x / 2.f, windowSize.y / 7.f);

sf::Vector2f statusMassagePos(windowSize.x / 2.13f, statusMassageSize.y);

pausedMenuIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "PAUSE", windowPosition, statusMassagePos)));

pausedMenuIt->get()->contentList.back()->setFontSize(40);

pausedMenuIt->get()->contentList.back()->setFontColor(sf::Color::White);

sf::Vector2f restartButtonsSize(windowSize.x / 2.f, windowSize.y / 7.f);

sf::Vector2f restartButtonPos(windowPosition.x + windowSize.x / 2.f - restartButtonsSize.x / 2.f, windowPosition.y + statusMassageSize.y + restartButtonsSize.y \* 3.f);

buttonList.push\_back(button(new InterfaceRestartButton(\*pausedMenuIt->get(), restartButtonPos)));

restartButton = buttonList.end();

--restartButton;

restartButton->get()->setSizes(restartButtonsSize);

sf::Vector2f restartButtonContentCorrectionPos(restartButtonsSize.x / 2.24f,0.f);

restartButton->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "RESTART", restartButtonPos, restartButtonContentCorrectionPos)));

restartButton->get()->contentList.back()->setFontSize(28);

restartButton->get()->contentList.back()->setFontColor(sf::Color::White);

sf::Vector2f backToMainMenuButtonsSize(windowSize.x / 2.f, windowSize.y / 7.f);

sf::Vector2f backToMainMenuButtonPos(windowPosition.x + windowSize.x / 2.f - backToMainMenuButtonsSize.x / 2.f, restartButtonPos.y + backToMainMenuButtonsSize.y);

buttonList.push\_back(button(new IntefaceExitButton(\*pausedMenuIt->get(), backToMainMenuButtonPos)));

backTomainMenuButton = buttonList.end();

--backTomainMenuButton;

backTomainMenuButton->get()->setSizes(backToMainMenuButtonsSize);

sf::Vector2f backToMainMenuButtonButtonContentCorrectionPos(backToMainMenuButtonsSize.x / 4.f, 0.f);

backTomainMenuButton->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "BACK TO MAIN MENU", backToMainMenuButtonPos, backToMainMenuButtonButtonContentCorrectionPos)));

backTomainMenuButton->get()->contentList.back()->setFontSize(28);

backTomainMenuButton->get()->contentList.back()->setFontColor(sf::Color::White);

}

bool InterfaceEngine\_t::pause() {

missionWindowIt->get()->setDisplay(false);

gameStatsWindowIt->get()->setDisplay(false);

mapIt->get()->setDisplay(false);

skillDescriptionWindowIt->get()->setDisplay(false);

return pausedMenuIt->get()->toggleDisplay();

}

//BUTTONS

void InterfaceEngine\_t::createInterfaceButtons() {

sf::Vector2f buttonPosition(interface::STD\_MARGIN\_SIZE.x, window->getSize().y / 4.f);

sf::Vector2f buttonMargin(0.f, interface::STD\_MARGIN\_SIZE.y);

//JOURNAL BUTTON

buttonList.push\_back(button(new IntefaceToggleButton(\*missionWindowIt->get(), buttonPosition)));

sf::Texture \*temp = new sf::Texture;

temp->loadFromFile(icon::ICON\_BUTTON\_JOURNAL);

buttonList.back().get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, temp, buttonList.back()->getPos())));

std::list<button>::iterator journalButton = buttonList.end();

--journalButton;

buttonsMap.insert(std::pair<std::string, std::list<button>::iterator>(buttons::JOURNAL\_KEY\_NAME, journalButton));

//GAME STATISTIC BUTTON

buttonPosition.y = buttonPosition.y + buttonList.back()->getSizes().y;

buttonList.push\_back(button(new IntefaceToggleButton(\*gameStatsWindowIt->get(), buttonPosition + buttonMargin)));

sf::Texture \*temp2 = new sf::Texture;

temp2->loadFromFile(icon::ICON\_BUTTON\_GAME\_STATISTIC);

buttonList.back().get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, temp2, buttonList.back()->getPos())));

std::list<button>::iterator statisticButton = buttonList.end();

--statisticButton;

buttonsMap.insert(std::pair<std::string, std::list<button>::iterator>(buttons::GAME\_STATS\_KEY\_NAME, statisticButton));

//MAP BUTTON

buttonPosition.y = buttonPosition.y + buttonList.back()->getSizes().y;

buttonList.push\_back(button(new IntefaceToggleButton(\*mapIt->get(), buttonPosition + buttonMargin \* 2.f)));

sf::Texture \*temp3 = new sf::Texture;

temp3->loadFromFile(icon::ICON\_BUTTON\_MAP);

buttonList.back().get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, temp3, buttonList.back()->getPos())));

std::list<button>::iterator mapButton = buttonList.end();

--mapButton;

buttonsMap.insert(std::pair<std::string, std::list<button>::iterator>(buttons::MAP\_KEY\_NAME, mapButton));

//SKILL BUTTON

buttonPosition.y = buttonPosition.y + buttonList.back()->getSizes().y ;

buttonList.push\_back(button(new IntefaceToggleButton(\*skillDescriptionWindowIt->get(), buttonPosition + buttonMargin \* 3.f)));

sf::Texture \*temp4 = new sf::Texture;

temp4->loadFromFile(icon::ICON\_BUTTON\_SKILLS);

buttonList.back().get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, temp4, buttonList.back()->getPos())));

std::list<button>::iterator skillsButton = buttonList.end();

--skillsButton;

buttonsMap.insert(std::pair<std::string, std::list<button>::iterator>(buttons::SKILL\_KEY\_NAME, skillsButton));

}

void InterfaceEngine\_t::setCamera() {

float \_x = level.mainHero->get()->getPosOfCenter().x;

float \_y = level.mainHero->get()->getPosOfCenter().y;

//EDIT THIS FOR CAMERA CONTROLL

float leftBorder = static\_cast<float>(window->getSize().x) / 2;

float topBorder = static\_cast<float>(window->getSize().y) / 2;

float rightBorder = level.map.getSize().x - (static\_cast<float>(window->getSize().x) / 2);

float bottomBorder = level.map.getSize().y - (static\_cast<float>(window->getSize().y) / 2);

float error = 5.0f;

if (\_x < leftBorder) {

\_x = leftBorder;

}

else if (\_x > rightBorder) {

\_x = rightBorder;

}

if (\_y > bottomBorder) {

\_y = bottomBorder;

}

else if (\_y < topBorder) {

\_y = topBorder;

}

view->setCenter(\_x, \_y);

}

//SKILL DESCRIPTION INTERFACE

void InterfaceEngine\_t::createDescriptionMenu() {

size\_t skillsAmount = 10;

sf::Vector2f windowPosition(STD\_BUTTON\_SIZE.x + interface::STD\_WINDOW\_MARGIN\_SIZE.x, window->getSize().y / 2.f);

sf::Vector2f windowSize((icon::ICON\_DESC\_WIDTH + interface::STD\_BORDER\_SIZE.x) \* 9.f, (skillsAmount + 1) \* (icon::ICON\_DESC\_HEIGHT + interface::STD\_BORDER\_SIZE.y \* 2));

windowsList.push\_back(window\_t(new InterfaceWindow\_t(window,windowPosition,windowSize)));

skillDescriptionWindowIt = windowsList.end();

--skillDescriptionWindowIt;

skillDescriptionWindowIt->get()->setTitle("SKILL DESCRIPTION");

skillDescriptionWindowIt->get()->setDisplay(false);

skillDescriptionWindowIt->get()->setBgColor(sf::Color::Color(50, 50, 50, 120));

skillDescriptionWindowIt->get()->setBorderColor(sf::Color::Color(238, 238, 238, 120));

sf::Vector2f contentPos(0.f, 0.f);

contentPos.x = textSettings::STD\_FONT\_SIZE + interface::STD\_BORDER\_SIZE.x;

using namespace icon;

int i = 1; //content string number

contentPos.y = static\_cast<float>((i) \* ICON\_DESC\_HEIGHT/2) + interface::STD\_BORDER\_SIZE.x;

sf::Texture \*EARTH\_ICON = new sf::Texture; EARTH\_ICON->loadFromFile(DESC\_ICON\_ELEMENT\_EARTH\_FILE);

sf::Texture \*FIRE\_ICON = new sf::Texture; FIRE\_ICON->loadFromFile(DESC\_ICON\_ELEMENT\_FIRE\_FILE);

sf::Texture \*WIND\_ICON= new sf::Texture; WIND\_ICON->loadFromFile(DESC\_ICON\_ELEMENT\_WIND\_FILE);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE\*2;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, EARTH\_ICON, skillDescriptionWindowIt->get()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window,"Num1" , windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE\*2;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, FIRE\_ICON, skillDescriptionWindowIt->get()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "Num2", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE \*2;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, WIND\_ICON, skillDescriptionWindowIt->get()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "Num3", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

//111

contentPos.x = textSettings::STD\_FONT\_SIZE + interface::STD\_BORDER\_SIZE.x;

sf::Texture \*HEAL = new sf::Texture;

HEAL->loadFromFile(ICON\_HEAL\_111);

++i;

contentPos.y = static\_cast<float>((i) \* ICON\_DESC\_HEIGHT) + interface::STD\_BORDER\_SIZE.x;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, HEAL, skillDescriptionWindowIt->get()->getPos(), contentPos)));

sf::Texture \*EARTH\_ICON1 = new sf::Texture; EARTH\_ICON1->loadFromFile(DESC\_ICON\_ELEMENT\_EARTH\_FILE);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE \* 2;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, EARTH\_ICON1, skillDescriptionWindowIt->get()->getPos(), contentPos)));

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "x3", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

//112

contentPos.x = textSettings::STD\_FONT\_SIZE + interface::STD\_BORDER\_SIZE.x;

sf::Texture \*BANG\_BALL= new sf::Texture;

BANG\_BALL->loadFromFile(ICON\_BANG\_112);

++i;

contentPos.y = static\_cast<float>((i) \* ICON\_DESC\_HEIGHT) + interface::STD\_BORDER\_SIZE.x;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, BANG\_BALL, skillDescriptionWindowIt->get()->getPos(), contentPos)));

sf::Texture \*EARTH\_ICON2 = new sf::Texture; EARTH\_ICON2->loadFromFile(DESC\_ICON\_ELEMENT\_EARTH\_FILE);

sf::Texture \*FIRE\_ICON2 = new sf::Texture; FIRE\_ICON2->loadFromFile(DESC\_ICON\_ELEMENT\_FIRE\_FILE);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE \* 2;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, EARTH\_ICON2, skillDescriptionWindowIt->get()->getPos(), contentPos)));

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "x2", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, FIRE\_ICON2, skillDescriptionWindowIt->get()->getPos(), contentPos)));

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "x1", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

//113

contentPos.x = textSettings::STD\_FONT\_SIZE + interface::STD\_BORDER\_SIZE.x;

sf::Texture \*STONE= new sf::Texture;

STONE->loadFromFile(ICON\_STONE\_113);

++i;

contentPos.y = static\_cast<float>((i) \* ICON\_DESC\_HEIGHT) + interface::STD\_BORDER\_SIZE.x;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, STONE, skillDescriptionWindowIt->get()->getPos(), contentPos)));

sf::Texture \*EARTH\_ICON3 = new sf::Texture; EARTH\_ICON3->loadFromFile(DESC\_ICON\_ELEMENT\_EARTH\_FILE);

sf::Texture \*WIND\_ICON3 = new sf::Texture; WIND\_ICON3->loadFromFile(DESC\_ICON\_ELEMENT\_WIND\_FILE);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE \* 2;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, EARTH\_ICON3, skillDescriptionWindowIt->get()->getPos(), contentPos)));

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "x2", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, WIND\_ICON3, skillDescriptionWindowIt->get()->getPos(), contentPos)));

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "x1", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

//123

contentPos.x = textSettings::STD\_FONT\_SIZE + interface::STD\_BORDER\_SIZE.x;

sf::Texture \*COMBO = new sf::Texture;

COMBO->loadFromFile(ICON\_COMBO\_123);

++i;

contentPos.y = static\_cast<float>((i) \* ICON\_DESC\_HEIGHT) + interface::STD\_BORDER\_SIZE.x;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, COMBO, skillDescriptionWindowIt->get()->getPos(), contentPos)));

sf::Texture \*EARTH\_ICON4 = new sf::Texture; EARTH\_ICON4->loadFromFile(DESC\_ICON\_ELEMENT\_EARTH\_FILE);

sf::Texture \*FIRE\_ICON4 = new sf::Texture; FIRE\_ICON4->loadFromFile(DESC\_ICON\_ELEMENT\_FIRE\_FILE);

sf::Texture \*WIND\_ICON4 = new sf::Texture; WIND\_ICON4->loadFromFile(DESC\_ICON\_ELEMENT\_WIND\_FILE);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE \* 2;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, EARTH\_ICON4, skillDescriptionWindowIt->get()->getPos(), contentPos)));

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "x1", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, FIRE\_ICON4, skillDescriptionWindowIt->get()->getPos(), contentPos)));

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "x1", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, WIND\_ICON4, skillDescriptionWindowIt->get()->getPos(), contentPos)));

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "x1", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

//221

contentPos.x = textSettings::STD\_FONT\_SIZE + interface::STD\_BORDER\_SIZE.x;

sf::Texture \*FIRE\_BREATH = new sf::Texture;

FIRE\_BREATH->loadFromFile(ICON\_FIRE\_BREATH\_221);

++i;

contentPos.y = static\_cast<float>((i) \* ICON\_DESC\_HEIGHT) + interface::STD\_BORDER\_SIZE.x;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, FIRE\_BREATH, skillDescriptionWindowIt->get()->getPos(), contentPos)));

sf::Texture \*EARTH\_ICON5 = new sf::Texture; EARTH\_ICON5->loadFromFile(DESC\_ICON\_ELEMENT\_EARTH\_FILE);

sf::Texture \*FIRE\_ICON5= new sf::Texture; FIRE\_ICON5->loadFromFile(DESC\_ICON\_ELEMENT\_FIRE\_FILE);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE\*2;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, FIRE\_ICON5, skillDescriptionWindowIt->get()->getPos(), contentPos)));

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "x2", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, EARTH\_ICON5, skillDescriptionWindowIt->get()->getPos(), contentPos)));

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "x1", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

//222

contentPos.x = textSettings::STD\_FONT\_SIZE + interface::STD\_BORDER\_SIZE.x;

sf::Texture \*POWER\_BUFF = new sf::Texture;

POWER\_BUFF->loadFromFile(ICON\_POWER\_BUFF\_222);

++i;

contentPos.y = static\_cast<float>((i) \* ICON\_DESC\_HEIGHT) + interface::STD\_BORDER\_SIZE.x;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, POWER\_BUFF, skillDescriptionWindowIt->get()->getPos(), contentPos)));

sf::Texture \*FIRE\_ICON6 = new sf::Texture; FIRE\_ICON6->loadFromFile(DESC\_ICON\_ELEMENT\_FIRE\_FILE);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE \* 2;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, FIRE\_ICON6, skillDescriptionWindowIt->get()->getPos(), contentPos)));

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "x3", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

//223

contentPos.x = textSettings::STD\_FONT\_SIZE + interface::STD\_BORDER\_SIZE.x;

sf::Texture \*FIRE\_BALL = new sf::Texture;

FIRE\_BALL->loadFromFile(ICON\_FIRE\_BALL\_223);

++i;

contentPos.y = static\_cast<float>((i) \* ICON\_DESC\_HEIGHT) + interface::STD\_BORDER\_SIZE.x;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, FIRE\_BALL, skillDescriptionWindowIt->get()->getPos(), contentPos)));

sf::Texture \*FIRE\_ICON7 = new sf::Texture; FIRE\_ICON7->loadFromFile(DESC\_ICON\_ELEMENT\_FIRE\_FILE);

sf::Texture \*WIND\_ICON7 = new sf::Texture; WIND\_ICON7->loadFromFile(DESC\_ICON\_ELEMENT\_WIND\_FILE);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE \* 2;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, FIRE\_ICON7, skillDescriptionWindowIt->get()->getPos(), contentPos)));

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "x2", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, WIND\_ICON7, skillDescriptionWindowIt->get()->getPos(), contentPos)));

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "x1", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

//331

contentPos.x = textSettings::STD\_FONT\_SIZE + interface::STD\_BORDER\_SIZE.x;

sf::Texture \*SMALL\_EARTH\_BALL = new sf::Texture;

SMALL\_EARTH\_BALL->loadFromFile(ICON\_SMALL\_EARTH\_BALLS\_331);

++i;

contentPos.y = static\_cast<float>((i) \* ICON\_DESC\_HEIGHT) + interface::STD\_BORDER\_SIZE.x;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, SMALL\_EARTH\_BALL, skillDescriptionWindowIt->get()->getPos(), contentPos)));

sf::Texture \*EARTH\_ICON8 = new sf::Texture; EARTH\_ICON8->loadFromFile(DESC\_ICON\_ELEMENT\_EARTH\_FILE);

sf::Texture \*WIND\_ICON8 = new sf::Texture; WIND\_ICON8->loadFromFile(DESC\_ICON\_ELEMENT\_WIND\_FILE);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE \* 2;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, WIND\_ICON8, skillDescriptionWindowIt->get()->getPos(), contentPos)));

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "x2", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, EARTH\_ICON8, skillDescriptionWindowIt->get()->getPos(), contentPos)));

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "x1", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

//332

contentPos.x = textSettings::STD\_FONT\_SIZE + interface::STD\_BORDER\_SIZE.x;

sf::Texture \*FIRE\_LIGHTING = new sf::Texture;

FIRE\_LIGHTING->loadFromFile(ICON\_FIRE\_LIGHTING\_332);

++i;

contentPos.y = static\_cast<float>((i) \* ICON\_DESC\_HEIGHT) + interface::STD\_BORDER\_SIZE.x;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, FIRE\_LIGHTING, skillDescriptionWindowIt->get()->getPos(), contentPos)));

sf::Texture \*FIRE\_ICON9 = new sf::Texture; FIRE\_ICON9->loadFromFile(DESC\_ICON\_ELEMENT\_EARTH\_FILE);

sf::Texture \*WIND\_ICON9 = new sf::Texture; WIND\_ICON9->loadFromFile(DESC\_ICON\_ELEMENT\_WIND\_FILE);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE \* 2;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, WIND\_ICON9, skillDescriptionWindowIt->get()->getPos(), contentPos)));

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "x2", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, FIRE\_ICON9, skillDescriptionWindowIt->get()->getPos(), contentPos)));

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "x1", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

//333

contentPos.x = textSettings::STD\_FONT\_SIZE + interface::STD\_BORDER\_SIZE.x;

sf::Texture \*SPEED\_BUFF = new sf::Texture;

SPEED\_BUFF->loadFromFile(ICON\_SPEED\_BUFF\_333);

++i;

contentPos.y = static\_cast<float>((i) \* ICON\_DESC\_HEIGHT) + interface::STD\_BORDER\_SIZE.x;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, SPEED\_BUFF, skillDescriptionWindowIt->get()->getPos(), contentPos)));

sf::Texture \*WIND\_ICON10 = new sf::Texture; WIND\_ICON10->loadFromFile(DESC\_ICON\_ELEMENT\_WIND\_FILE);

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE \* 2;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceSpriteContent\_t(window, WIND\_ICON10, skillDescriptionWindowIt->get()->getPos(), contentPos)));

contentPos.x += DESC\_ICON\_ELEMENT\_SIZE;

skillDescriptionWindowIt->get()->contentList.push\_back(content(new InterfaceTextContent\_t(window, "x3", windowsList.back()->getPos(), contentPos)));

skillDescriptionWindowIt->get()->contentList.back()->setFontSize(18);

}

================================= “IntefaceWindow.h” =================================

#pragma once

#include <SFML/Graphics.hpp>

#include <memory>

#include <list>

#include "additional.h"

#include "InterfaceContent.h"

typedef std::unique\_ptr<InterfaceContent> content;

class InterfaceWindow\_t

{

protected:

bool display;

bool clickable;

sf::Vector2f pos;

sf::Vector2f sizes;

sf::RenderWindow \*window;

sf::Font font;

sf::Text text;

std::string title;

sf::Vector2f textRelativePos;

sf::RectangleShape rectangle;

sf::Vector2f borders;

sf::RectangleShape outerBorderRect;

sf::Int32 startTime;

sf::Int32 displayDuration;

public:

std::vector<content> contentList;

InterfaceWindow\_t(sf::RenderWindow \*\_window, sf::Vector2f \_pos, sf::Vector2f \_sizes);

virtual ~InterfaceWindow\_t();

virtual void draw();

bool toggleDisplay();

virtual void update();

virtual void toDefaultPosition();

//GET

sf::Vector2f getPos() const;

sf::Vector2f getSizes() const;

sf::FloatRect getFloatRect()const;

sf::IntRect getIntRect() const;

bool getClickable() const;

bool getDisplayState() const;

sf::RenderWindow \*getWindowPtr()const;

sf::Vector2f getBorders() const;

//SET

void setPos(sf::Vector2f &newpos);

void setSizes(sf::Vector2f &newSizes);

void setBgColor(sf::Color newColor);

void setClickble(bool newState);

void setBorderColor(sf::Color newColor);

void setDisplay(bool newState);

void setTitle(std::string newTitle);

void setTitlePos(sf::Vector2f \_newpos);

void setTextFontSize(unsigned int size);

void setFont(std::string \_path);

void setTextColor(sf::Color newColor);

};

=============================== “InterfaceWindow\_t.cpp” ================================

#include "InterfaceWindow\_t.h"

//INTERFACE OB

InterfaceWindow\_t::InterfaceWindow\_t(sf::RenderWindow \*\_window, sf::Vector2f \_pos, sf::Vector2f \_sizes): rectangle(\_sizes),borders(interface::STD\_BORDER\_SIZE)

{

display = true;

clickable = false;

window = \_window;

pos = \_pos;

sizes = \_sizes;

rectangle.setSize(sizes);

rectangle.setFillColor(sf::Color(0, 0, 0, 255));

rectangle.setPosition(pos);

outerBorderRect.setSize(sizes + borders\*2.f);

outerBorderRect.setFillColor(sf::Color::Color(238, 238, 238, 255));

font.loadFromFile(textSettings::STD\_FONT\_FILE);

text.setCharacterSize(textSettings::STD\_FONT\_SIZE);

text.setFont(font);

text.setPosition(pos + textRelativePos);

text.setString(title);

}

InterfaceWindow\_t::~InterfaceWindow\_t()

{

}

bool InterfaceWindow\_t::toggleDisplay() {

display = display ? false : true;

return display;

}

void InterfaceWindow\_t::draw() {

if (display) {

window->draw(outerBorderRect);

window->draw(rectangle);

window->draw(text);

for (auto &i : contentList) {

i->draw();

}

}

}

void InterfaceWindow\_t::update() {

toDefaultPosition();

for (auto &i : contentList) {

i->update();

}

}

void InterfaceWindow\_t::toDefaultPosition() {

sf::Vector2f tempPos(interface::getScreenCoords(window));

rectangle.setPosition(tempPos + pos);

text.setPosition(tempPos + pos + textRelativePos);

outerBorderRect.setPosition(tempPos + pos - borders);

}

//GET

sf::Vector2f InterfaceWindow\_t::getPos() const {

return pos;

}

sf::Vector2f InterfaceWindow\_t::getSizes() const {

return sizes;

}

sf::FloatRect InterfaceWindow\_t::getFloatRect()const {

return sf::FloatRect(pos,sizes);

}

sf::IntRect InterfaceWindow\_t::getIntRect() const {

return sf::IntRect(static\_cast<int>(pos.x), static\_cast<int>(pos.y), static\_cast<int>(sizes.x), static\_cast<int>(sizes.y));

}

bool InterfaceWindow\_t::getClickable() const {

return clickable;

}

bool InterfaceWindow\_t::getDisplayState() const {

return display;

}

sf::RenderWindow \*InterfaceWindow\_t::getWindowPtr()const {

return window;

}

sf::Vector2f InterfaceWindow\_t::getBorders() const {

return borders;

}

//SET

void InterfaceWindow\_t::setPos(sf::Vector2f &newpos) {

pos = newpos;

rectangle.setPosition(pos);

}

void InterfaceWindow\_t::setSizes(sf::Vector2f &newSizes) {

sizes = newSizes;

rectangle.setSize(sizes);

}

void InterfaceWindow\_t::setBgColor(sf::Color newColor) {

rectangle.setFillColor(newColor);

}

void InterfaceWindow\_t::setClickble(bool newState) {

clickable = newState;

}

void InterfaceWindow\_t::setBorderColor(sf::Color newColor) {

outerBorderRect.setFillColor(newColor);

}

void InterfaceWindow\_t::setDisplay(bool newState) {

display = newState;

}

void InterfaceWindow\_t::setTitle(std::string newTitle) {

title = newTitle;

text.setString(title);

}

void InterfaceWindow\_t::setTitlePos(sf::Vector2f \_newpos) {

textRelativePos = \_newpos;

}

void InterfaceWindow\_t::setTextFontSize(unsigned int size) {

text.setCharacterSize(size);

}

void InterfaceWindow\_t::setFont(std::string \_path) {

font.loadFromFile(\_path);

}

void InterfaceWindow\_t::setTextColor(sf::Color newColor) {

text.setFillColor(newColor);

}

================================= “keyboardController.h” ================================

#pragma once

#include <SFML/Graphics.hpp>

#include "character\_t.h"

#include "game\_t.h"

class game\_t;

class keyboardController{

character\_t \*character;

game\_t \*game;

void checkCharacterStateAndChangeDefault();

public:

keyboardController(character\_t \*\_mainHero, game\_t \*\_game);

virtual ~keyboardController();

void eventHandler(sf::Event &event);

character\_t \* getPlayerCharPtr()const;

};

=============================== “keyboardController.cpp” ================================

#include "keyboardController.h"

//keyboardController

keyboardController::keyboardController(character\_t \*mainHero, game\_t \*\_game)

{

character = mainHero;

game = \_game;

}

keyboardController::~keyboardController()

{

}

void keyboardController::checkCharacterStateAndChangeDefault() {

if ((character->getState()->getStateNum() != 3)) {

character->changeState(new CharacterPlayerControll\_t(\*character));

}

}

void keyboardController::eventHandler(sf::Event &event) {

using namespace sf;

if (game->getStatus() != game::PAUSED) {

if (sf::Mouse::isButtonPressed(sf::Mouse::Left)){

character->attack();

checkCharacterStateAndChangeDefault();

}

}

if (event.type == Event::KeyReleased) {

if (event.key.code == Keyboard::Escape) {

checkCharacterStateAndChangeDefault();

game->interfaceEngine->pause();

if (game->interfaceEngine->pausedMenuIt->get()->getDisplayState()) {

game->setGameStatus(game::PAUSED);

}

else {

game->setGameStatus(game::PLAY);

}

}

if (game->getStatus() != game::PAUSED) {

switch (event.key.code) {

case Keyboard::E: {

game->setGameStatus(game::WIN);

checkCharacterStateAndChangeDefault();

break;

}

case Keyboard::Num1: {

character->addElement(elements::EARTH);

checkCharacterStateAndChangeDefault();

break;

}

case Keyboard::Num2: {

character->addElement(elements::FIRE);

checkCharacterStateAndChangeDefault();

break;

}

case Keyboard::Num3: {

character->addElement(elements::WIND);

checkCharacterStateAndChangeDefault();

break;

}

case Keyboard::J: {

game->interfaceEngine->buttonsMap[buttons::JOURNAL\_KEY\_NAME]->get()->action();

break;

}

case Keyboard::T: {

game->interfaceEngine->buttonsMap[buttons::GAME\_STATS\_KEY\_NAME]->get()->action();

break;

}

case Keyboard::M: {

game->interfaceEngine->buttonsMap[buttons::MAP\_KEY\_NAME]->get()->action();

break;

}

case Keyboard::K: {

game->interfaceEngine->buttonsMap[buttons::SKILL\_KEY\_NAME]->get()->action();

break;

}

}

}

}

if (Keyboard::isKeyPressed(Keyboard::Space)) {

if (character->getState()->getStateNum() != 4) {

character->getTimer().updateCastCD();

character->changeState(new CharacterPlayerCast\_t(\*character));

}

}

if (Keyboard::isKeyPressed(Keyboard::W)) {

character->setdY(-character->getStats().speed);

checkCharacterStateAndChangeDefault();

}

else if (Keyboard::isKeyPressed(Keyboard::A)) {

character->setdX(-character->getStats().speed);

checkCharacterStateAndChangeDefault();

}

else if (Keyboard::isKeyPressed(Keyboard::S)) {

character->setdY(character->getStats().speed);

checkCharacterStateAndChangeDefault();

}

else if (Keyboard::isKeyPressed(Keyboard::D)) {

character->setdX(character->getStats().speed);

checkCharacterStateAndChangeDefault();

}

}

//GET

character\_t \* keyboardController::getPlayerCharPtr()const {

return character;

}

================================= “Level\_t.h” =================================

#pragma once

#include <vector>

#include <memory>

#include "character\_t.h"

#include "player\_t.h"

#include "Npc\_t.h"

#include "map\_t.h"

#include "bullet\_t.h"

#include "additional.h"

#include "mission\_t.h"

class Level\_t

{

public:

bool succesfull;

bool levelComplete;

bool gameOver;

mission\_t mission;

map\_t map;

std::list<std::unique\_ptr <character\_t>> charactersList;

std::list<std::unique\_ptr <bullet\_t>> bulletsList;

std::list<std::unique\_ptr <character\_t>>::iterator bossesListIt;

std::list<std::unique\_ptr <character\_t>>::iterator mainHero;

Level\_t(std::string \_levelName);

~Level\_t();

mission\_t &getMission();

void checkMissionsTarget();

bool checkLevelComplete();

};

================================= “Level\_t.cpp” =================================

#include "Level\_t.h"

Level\_t::Level\_t(std::string \_levelName): map(\_levelName)

{

succesfull = (map.fillTheMapObj() && map.fillTheMapTiles());

levelComplete = false;

gameOver = false;

}

Level\_t::~Level\_t()

{

}

mission\_t &Level\_t::getMission() {

return mission;

}

void Level\_t::checkMissionsTarget() {

std::list<std::unique\_ptr <character\_t>>::iterator it = bossesListIt;

for (size\_t i = 0; it != charactersList.end(); ++i, ++it) {

if (!it->get()->getAlive()){

mission.missionsCompleteStatus[i] = true;

}

}

}

bool Level\_t::checkLevelComplete() {

for (auto &i : mission.missionsCompleteStatus) {

if (!i) {

return false;

}

}

return true;

}

================================= “mainMenu\_t.h” =================================

#pragma once

#include <SFML\Graphics.hpp>

#include <vector>

#include <memory>

const std::string MENU\_BG\_TEXTURE\_FILE = "img/menu/menu\_bg.jpg";

const std::string NAME\_TEXTURE\_FILE = "img/menu/nameTexture.png";

const std::string START\_MESSAGE\_TEXTURE\_FILE = "img/menu/start\_mess.png";

const std::string FONT\_FILE = "fonts/hercules\_modern.ttf";

typedef std::unique\_ptr<sf::Texture> texture;

typedef std::unique\_ptr<sf::Texture> sprite;

typedef std::vector<std::string> strVec;

class mainMenu\_t

{

sf::RenderWindow \*window;

texture bgTexture;

texture nameTexture;

texture startMessageTexture;

sf::Sprite bgSprite;

sf::Sprite nameSprite;

sf::Sprite startMessageSprite;

strVec menuItems;

strVec levelNamesList;

strVec difficulValuesList;

sf::Text levelNameText;

sf::Text difficultyText;

std::vector<sf::Text> menuItemsText;

std::vector<sf::FloatRect> menuItemsRectList;

sf::Font textFont;

unsigned int fontSize;

unsigned int targetItemFontSize;

sf::Color itemsMainColor;

sf::Color targetItemColor;

sf::Color variativeItemsColor;

void draw();

int itemChoice();

std::string &levelName;

size\_t &difficulty;

public:

bool active;

mainMenu\_t(sf::RenderWindow \*\_window,std::string &\_levelName,size\_t &\_difficulty);

~mainMenu\_t();

void action();

void makeMenu();

};

================================= “mainMenu\_t.cpp” =================================

#include "mainMenu\_t.h"

mainMenu\_t::mainMenu\_t(sf::RenderWindow \*\_window, std::string &\_levelName, size\_t &\_difficulty):levelName(\_levelName),difficulty(\_difficulty)

{

window = \_window;

//TEXTURE PREFERENCES

bgTexture.swap(texture(new sf::Texture));

nameTexture.swap(texture(new sf::Texture));

startMessageTexture.swap(texture(new sf::Texture));

bgTexture->loadFromFile(MENU\_BG\_TEXTURE\_FILE);

nameTexture->loadFromFile(NAME\_TEXTURE\_FILE);

startMessageTexture->loadFromFile(START\_MESSAGE\_TEXTURE\_FILE);

//SPRITES PREFERENCES

bgSprite.setTexture(\*bgTexture.get());

nameSprite.setTexture(\*nameTexture.get());

startMessageSprite.setTexture(\*startMessageTexture.get());

//TEXT PREFERENCES

textFont.loadFromFile(FONT\_FILE);

fontSize = static\_cast<unsigned int>(window->getSize().x / 32);

targetItemFontSize = fontSize + 1;

itemsMainColor = sf::Color::Color(255, 255, 255, 255);

variativeItemsColor = sf::Color::Yellow;

targetItemColor = sf::Color::Green;

menuItems = {

"START GAME",

"LEVEL: ",

"DIFFICULTY: ",

"EXIT"

};

levelNamesList = {

"level0",

"West Land",

"East Land (in dev)",

"Evil Land (in dev)",

"Russia Land (in dev)"

};

difficulValuesList = {

"EASY",

"MEDIUM",

"HARD"

};

levelNameText.setFont(textFont);

levelNameText.setCharacterSize(fontSize);

levelNameText.setFillColor(variativeItemsColor);

difficultyText.setFont(textFont);

difficultyText.setCharacterSize(fontSize);

difficultyText.setFillColor(variativeItemsColor);

active = true;

}

mainMenu\_t::~mainMenu\_t()

{

}

void mainMenu\_t::makeMenu() {

sf::Vector2u winSize = window->getSize();//CALCULATE WINDOW SIZE

float kX = (static\_cast<float>(window->getSize().x) / static\_cast<float>(bgTexture->getSize().x));

float kY = (static\_cast<float>(window->getSize().y) / static\_cast<float>(bgTexture->getSize().y));

bgSprite.setScale(kX, kY);

nameSprite.setScale(kX, kY);

nameSprite.setOrigin(static\_cast<float>(nameTexture->getSize().x) / 2, static\_cast<float>(nameTexture->getSize().y) / 2);//Ïîìåíÿòü êîîðäèíàòû!!!!!!!!!!!!

nameSprite.setPosition(static\_cast<float>(winSize.x) / 2, static\_cast<float>(nameTexture->getSize().y) / 2);

//MAKE START MESSAGE SPRITE

startMessageSprite.scale(kX, kY);

startMessageSprite.setPosition(50.f, window->getSize().y / 4.f);

//MAKE MENU ITEMS TEXT

int menuItemsVerticalMargin = winSize.y / 10;

float itemskX = 0.5f;

float itemskY = 0.45f;

sf::Vector2f menuPosition(static\_cast<float>(winSize.x) \* itemskX, static\_cast<float>(winSize.y) \* itemskY);//ITEMS POSITION

for (size\_t i = 0; i < menuItems.size(); ++i) {

menuItemsText.push\_back(sf::Text(menuItems[i], textFont, fontSize));

menuItemsText.back().setPosition(menuPosition.x, menuPosition.y + menuItemsVerticalMargin \* i);

menuItemsText.back().setFillColor(itemsMainColor);

sf::FloatRect tempRect(menuItemsText.back().getPosition().x, menuItemsText.back().getPosition().y,

menuItemsText.back().getLocalBounds().width, menuItemsText.back().getLocalBounds().height);

menuItemsRectList.push\_back(tempRect);

}

//LEVEL NAME POSITION

sf::Vector2f levelTextPos(menuItemsText[1].getLocalBounds().width, 0.f);

levelNameText.setPosition(menuItemsText[1].getPosition() + levelTextPos);

sf::FloatRect levelNameRect(levelNameText.getPosition().x, levelNameText.getPosition().y,

levelNameText.getLocalBounds().width, levelNameText.getLocalBounds().height);

//DIFFICULITY VALUE POSITION

sf::Vector2f diffTextPos(menuItemsText[2].getLocalBounds().width, 0.f);

difficultyText.setPosition(menuItemsText[2].getPosition() + diffTextPos);

sf::FloatRect diffTextRect(difficultyText.getPosition().x, difficultyText.getPosition().y,

difficultyText.getLocalBounds().width, difficultyText.getLocalBounds().height);

}

void mainMenu\_t::draw() {

window->clear();

window->draw(bgSprite);

window->draw(nameSprite);

window->draw(startMessageSprite);

for (auto &i : menuItemsText) {

window->draw(i);

}

window->draw(difficultyText);

window->draw(levelNameText);

window->display();

}

void mainMenu\_t::action() {

//Int RECT LSIT

//LEVEL

size\_t currentLevelItem = 0;

size\_t levelAmount = levelNamesList.size() - 1;

//DIFFICALTY

size\_t currentDifficulityItem = 0;

size\_t maxDifficulity = difficulValuesList.size() - 1;

sf::Event event;

int currentItem = -1;

while (active) {

while (window->pollEvent(event)) {

if (event.type == sf::Event::Closed) {

window->close();

}

}

//COLOR TO DEFAULT

for (auto &i : menuItemsText) {

i.setFillColor(itemsMainColor);

i.setCharacterSize(fontSize);

}

difficultyText.setFillColor(variativeItemsColor);

levelNameText.setFillColor(variativeItemsColor);

//SET LEVEL & DIFFICULTY

difficultyText.setString(difficulValuesList[currentDifficulityItem]);

levelNameText.setString(levelNamesList[currentLevelItem]);

currentItem = itemChoice();

if (sf::Keyboard::isKeyPressed(sf::Keyboard::Escape)) {

window->close();

}

if (event.type == sf::Event::MouseButtonReleased && event.key.code == sf::Mouse::Left) {

switch (currentItem) {

case 0: {

levelName = levelNamesList[currentLevelItem];

difficulty = currentDifficulityItem + 1;

active = false;

}

case 1: {

if (currentLevelItem < levelAmount) {

++currentLevelItem;

}

else {

currentLevelItem = 0;

}

break;

}

case 2: {

if (currentDifficulityItem < maxDifficulity) {

++currentDifficulityItem;

}

else {

currentDifficulityItem = 0;

}

break;

}

case 3: {

exit(0);

}

}

}

event.key.code = sf::Keyboard::Unknown;

draw();

}

}

int mainMenu\_t::itemChoice() {

int currentItem = -1;

sf::Vector2f mousePos(static\_cast<float>(sf::Mouse::getPosition(\*window).x), static\_cast<float>(sf::Mouse::getPosition(\*window).y));

if (menuItemsRectList[0].contains(mousePos)) {

menuItemsText[0].setFillColor(targetItemColor);

menuItemsText[0].setCharacterSize(targetItemFontSize);

currentItem = 0;

}

if (menuItemsRectList[1].contains(mousePos)) {

menuItemsText[1].setFillColor(targetItemColor);

menuItemsText[1].setCharacterSize(targetItemFontSize);

currentItem = 1;

}

if (menuItemsRectList[2].contains(mousePos)) {

menuItemsText[2].setFillColor(targetItemColor);

menuItemsText[2].setCharacterSize(targetItemFontSize);

currentItem = 2;

}

if (menuItemsRectList[3].contains(mousePos)) {

menuItemsText[3].setFillColor(targetItemColor);

menuItemsText[3].setCharacterSize(targetItemFontSize);

currentItem = 3;

}

return currentItem;

}

================================= “map\_t.h” =================================

#pragma once

#include <SFML/Graphics.hpp>

#include "physOb\_t.h"

#include "ground\_t.h"

#include "additional.h"

#include <memory>

#include <list>

const std::string MAP\_PATH = "maps/";

const std::string MAP\_OBJ\_FILE\_NAME = "/obj.csv";

const std::string MAP\_BG\_TILES\_FILE\_NAME = "/map\_bg.csv";

const std::string MAP\_OUT\_TILES\_FILE\_NAME = "/map\_out.csv";

const std::string TILE\_FILE\_PATH = "img/objects/TilesTexture.png";

class map\_t

{

private:

sf::Vector2i size;

int tileAmountX;

int tileAmountY;

std::string levelName;

std::string mapObjectsFile;

std::string mapBgTilesFile;

std::string mapOutTilesFile;

std::string tileFileName;

std::shared\_ptr<sf::Texture>tile\_texture;

public:

sf::Vector2f mainHeroSpawnCoords;

std::vector<sf::Vector2f> bossesSpawnCoords;

sf::Sprite mapBgSprite;

std::list<std::unique\_ptr <ground\_t>> groundTilesList; //If needed layers view

std::list<std::unique\_ptr <physOb\_t>> mapObList;

map\_t();

map\_t(std::string \_levelName);

~map\_t();

public:

bool fillTheMapObj();

bool fillTheMapTiles();

std::string getLvlName() const;

public:

//GET

sf::Vector2i getSize() const;

sf::Vector2f getFloatSize() const;

sf::Vector2i getTileAmount() const;

};

“map\_t.cpp”

#include "map\_t.h"

#include <iostream>

#include <fstream>

#include <iomanip>

#include <ctype.h>

map\_t::map\_t()

{

}

//GET

std::string map\_t::getLvlName() const {

return levelName;

}

map\_t::map\_t(std::string \_levelName) {

levelName = \_levelName;

//LOAD MAP BG

mapObjectsFile = MAP\_PATH + levelName + MAP\_OBJ\_FILE\_NAME;

mapBgTilesFile = MAP\_PATH + levelName + MAP\_BG\_TILES\_FILE\_NAME;

mapOutTilesFile = MAP\_PATH + levelName + MAP\_OUT\_TILES\_FILE\_NAME;;

//LOAD MAP OBJECTS

tileFileName = TILE\_FILE\_PATH;

tile\_texture = std::make\_shared<sf::Texture>();

tile\_texture->loadFromFile(tileFileName);

//CALC MAP SIZEs

tileAmountX = 272;

tileAmountY = 143;

size.x = tileAmountX \* tiles::size;

size.y = tileAmountY\*tiles::size;

size\_t bossesAmount = 5;

bossesSpawnCoords = std::vector<sf::Vector2f>(bossesAmount);

}

map\_t::~map\_t()

{

}

bool map\_t::fillTheMapObj() {

using namespace std;

ifstream MAP\_FILE(mapObjectsFile, ios::in);

if (!MAP\_FILE) {

cout << "OPEN FILE ERROR" << endl;

return false;

}

int tileId;

for (int i = 0; i < tileAmountY; ++i) {

for (int j = 0; j < tileAmountX; ++j) {

MAP\_FILE >> tileId;

MAP\_FILE.get();

int coordX = j \* tiles::size;

int coordY = i \* tiles::size;

int spritePosY = ((tileId / tiles::TEXTURE\_TILE\_AMOUNT\_X)\*tiles::size);

int spritePosX = ((tileId % tiles::TEXTURE\_TILE\_AMOUNT\_X)\*tiles::size);

//\*/

if (tileId != (-1)) {

mapObList.push\_back(std::unique\_ptr<physOb\_t>(new physOb\_t(static\_cast<float>(coordX), static\_cast<float>(coordY), tile\_texture, spritePosX, spritePosY, tiles::size, tiles::size)));

}

}

}

return true;

}

bool map\_t::fillTheMapTiles() {

using namespace std;

ifstream MAP\_BG\_FILE(mapBgTilesFile, ios::in);

if (!MAP\_BG\_FILE) {

cout << "OPEN FILE1 ERROR" << endl;

return false;

}

ifstream MAP\_OUT\_FILE(mapOutTilesFile, ios::in);

if (!MAP\_OUT\_FILE) {

cout << "OPEN FILE2 ERROR" << endl;

return false;

}

int tileId;

for (int i = 0; i < tileAmountY; ++i) {

for (int j = 0; j < tileAmountX; ++j) {

MAP\_BG\_FILE >> tileId;

MAP\_BG\_FILE.get();

int coordX = j \* tiles::size;

int coordY = i \* tiles::size;

int spritePosY = ((tileId / tiles::TEXTURE\_TILE\_AMOUNT\_X)\*tiles::size);

int spritePosX = ((tileId % tiles::TEXTURE\_TILE\_AMOUNT\_X)\*tiles::size);

//\*//IF needed layers view should return this fragment

if (tileId != -1) {

groundTilesList.push\_back(std::unique\_ptr<ground\_t>(new ground\_t(tile\_texture, static\_cast<float>(coordX), static\_cast<float>(coordY), spritePosX, spritePosY, tileId)));

}

//\*/

}

}

for (int i = 0; i < tileAmountY; ++i) {

for (int j = 0; j < tileAmountX; ++j) {

MAP\_OUT\_FILE >> tileId;

MAP\_OUT\_FILE.get();

int coordX = j \* tiles::size;

int coordY = i \* tiles::size;

int spritePosY = ((tileId / tiles::TEXTURE\_TILE\_AMOUNT\_X)\*tiles::size);

int spritePosX = ((tileId % tiles::TEXTURE\_TILE\_AMOUNT\_X)\*tiles::size);

//\*//IF needed layers view should return this fragment

switch (tileId) {

case 1281: {

sf::Vector2f temp(static\_cast<float>(coordX), static\_cast<float>(coordY));

mainHeroSpawnCoords = temp;

break;

}

case 1282: {

sf::Vector2f temp(static\_cast<float>(coordX), static\_cast<float>(coordY));

bossesSpawnCoords[0] = temp; // TREANT SPAWN

break;

}

case 1283: {

sf::Vector2f temp(static\_cast<float>(coordX), static\_cast<float>(coordY));

bossesSpawnCoords[1] = temp; //RED DRAGON SPAWN

break;

}case 1284: {

sf::Vector2f temp(static\_cast<float>(coordX), static\_cast<float>(coordY));

bossesSpawnCoords[2] = temp; //BLACK DRAGON SPAWN

break;

}case 1285: {

sf::Vector2f temp(static\_cast<float>(coordX), static\_cast<float>(coordY));

bossesSpawnCoords[3] = temp; //ENH SPAWN

break;

}case 1286: {

sf::Vector2f temp(static\_cast<float>(coordX), static\_cast<float>(coordY));

bossesSpawnCoords[4] = temp; //DARK QUEEN SPAWN

break;

}

default: {

if (tileId != -1) {

groundTilesList.push\_back(std::unique\_ptr<ground\_t>(new ground\_t(tile\_texture, static\_cast<float>(coordX), static\_cast<float>(coordY), spritePosX, spritePosY, tileId)));

}

}

}

//\*/

}

}

return true;

}

//GET

sf::Vector2i map\_t::getSize() const {

return size;

}

sf::Vector2f map\_t::getFloatSize() const {

return sf::Vector2f(static\_cast<float>(size.x), static\_cast<float>(size.y));

}

sf::Vector2i map\_t::getTileAmount() const {

sf::Vector2i temp(tileAmountX,tileAmountY);

return temp;

}

================================= “mission\_t.h” =================================

#pragma once

#include <SFML\Graphics.hpp>

#include <sstream>

#include <vector>

#include <memory>

class gameStatistic\_t {

std::string convertTime(size\_t \_val);

public:

gameStatistic\_t();

~gameStatistic\_t();

std::vector<std::string> statFields;

std::vector<std::string> statStrValues;

std::vector <size\_t> statDataVect;

void fillStrStats();

void ånemyKilled();

void bossKilled();

void setTime(sf::Clock \*clock);

};

const std::vector<std::string> STD\_MISSION\_TASKS = {

"- KILL THE HOLY CRYSTAL GUARDIAN",

"- KILL RED DRAGON",

"- KILL BLACK DRAGON",

"- KILL THE ENH",

"- KILL THE QUEEN OF DARKNESS (MAIN)"

};

class mission\_t

{

public:

gameStatistic\_t gameStats;

std::vector<std::string> missionsContent;

std::vector<bool> missionsCompleteStatus;

mission\_t();

mission\_t(std::string \_fileName);

~mission\_t();

void ånemyKilled();

void bossKilled();

void setTime(sf::Clock \*clock);

bool checkComplete();

};

const std::vector<std::string> STAT\_LIST = {

"KILLED ENEMYES: ",

"KILLED BOSSES: ",

"ALIVE TIME: "

};

“mission\_t.cpp”

#include "mission\_t.h"

//GAME STATS

gameStatistic\_t::gameStatistic\_t() {

statFields = STAT\_LIST;

statStrValues = std::vector<std::string>(statFields.size());

statDataVect = std::vector <size\_t>(statFields.size(), 0);

fillStrStats();

}

gameStatistic\_t::~gameStatistic\_t() {

}

void gameStatistic\_t::ånemyKilled() {

++statDataVect[0];

}

void gameStatistic\_t::bossKilled() {

++statDataVect[1];

}

void gameStatistic\_t::setTime(sf::Clock \*clock) {

statDataVect[2] = static\_cast<size\_t>(clock->getElapsedTime().asSeconds());

}

std::string gameStatistic\_t::convertTime(size\_t \_val) {

size\_t minutes = \_val / 60;

size\_t seconds = \_val % 60;

std::ostringstream tempStream;

tempStream << minutes << ":" << seconds;//minutes + " : " + seconds

return std::string(tempStream.str());

}

void gameStatistic\_t::fillStrStats() {

std::ostringstream tempStream;

for (size\_t i = 0; i < statStrValues.size(); ++i) {

if (i != 2) {

tempStream.str(std::string());

tempStream << statDataVect[i];

statStrValues[i] = tempStream.str();

}

}

statStrValues[2] = convertTime(statDataVect[2]);

}

mission\_t::mission\_t(): gameStats()

{

missionsContent = STD\_MISSION\_TASKS;

missionsCompleteStatus = std::vector<bool>(missionsContent.size(),false);

}

mission\_t::mission\_t(std::string \_fileName) {

//OPEN FILE AND READ DATA

missionsCompleteStatus = std::vector<bool>(missionsContent.size(), false);

}

mission\_t::~mission\_t()

{

}

void mission\_t::ånemyKilled() {

gameStats.ånemyKilled();

}

void mission\_t::bossKilled() {

gameStats.bossKilled();

}

void mission\_t::setTime(sf::Clock \*clock) {

gameStats.setTime(clock);

}

bool mission\_t::checkComplete() {

for (auto &i : missionsCompleteStatus) {

if (!i) return false;

}

return true;

}

================================= “mouseController.h” =================================

#pragma once

#include <SFML\Graphics.hpp>

#include "InterfaceButton.h"

#include "game\_t.h"

#include "cursor\_t.h"

typedef std::unique\_ptr<InterfaceButton> button;

class game\_t;

class mouseController

{

sf::RenderWindow \*window;

game\_t \*game;

cursor\_t &cursor;

public:

mouseController(sf::RenderWindow \*\_window, game\_t \*\_game, cursor\_t &\_cursor);

~mouseController();

void eventHandler(sf::Event &event);

void menuEventHandler(sf::Event &event);

};

“mouseController.cpp”

#include "mouseController.h"

mouseController::mouseController(sf::RenderWindow \*\_window, game\_t \*\_game, cursor\_t &\_cursor) : cursor(\_cursor)

{

window = \_window;

game = \_game;

}

mouseController::~mouseController()

{

}

void mouseController::eventHandler(sf::Event &event) {

if (event.type == sf::Event::MouseButtonReleased && event.key.code == sf::Mouse::Left) {

for (auto &i : game->interfaceEngine.get()->buttonList) {

if (i->getIntRect().contains(sf::Mouse::getPosition(\*window))) {

i->action();

}

}

}

}

void mouseController::menuEventHandler(sf::Event &event) {

game->interfaceEngine->restartButton->get()->contentList.back()->setFontColor(sf::Color::White);

game->interfaceEngine->backTomainMenuButton->get()->contentList.back()->setFontColor(sf::Color::White);

if (game->interfaceEngine->restartButton->get()->getIntRect().contains(sf::Mouse::getPosition(\*window))) {

game->interfaceEngine->restartButton->get()->contentList.back()->setFontColor(sf::Color::Green);

if (sf::Mouse::isButtonPressed(sf::Mouse::Left)) {

game->setGameStatus(game::RESTART);

}

}

else if (game->interfaceEngine->backTomainMenuButton->get()->getIntRect().contains(sf::Mouse::getPosition(\*window))) {

game->interfaceEngine->backTomainMenuButton->get()->contentList.back()->setFontColor(sf::Color::Green);

if (sf::Mouse::isButtonPressed(sf::Mouse::Left)) {

game->setGameStatus(game::MAIN\_MENU);

}

}

}

================================= “Npc\_t.h” =================================

#pragma once

#include "character\_t.h"

#include "additional.h"

class Npc\_t :

public character\_t

{

protected:

sf::Vector2f spotCoords;

sf::Int32 spawnTime;

float powerMultiple;

virtual void setTypeStats();

public:

Npc\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock,float \_x, float \_y, int \_width, int \_height, float \_statMultiple = 1.f);

Npc\_t(std::shared\_ptr<sf::Texture>texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple = 1.f);

Npc\_t(character\_t \*copyedNpc, sf::Vector2f sporPoint, float powerMultiple = 1.f);

virtual ~Npc\_t();

virtual void attack();

sf::Vector2f getSpotCoords() const;

float getPowerMultiple() const;

};

class MageNpc\_t : public Npc\_t {

virtual void setTypeStats();

public:

MageNpc\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple = 1.f);

MageNpc\_t(character\_t \*copyedNpc, sf::Vector2f sporPoint, float powerMultiple = 1.f);

~MageNpc\_t();

};

class WarriorNpc\_t : public Npc\_t {

virtual void setTypeStats();

public:

WarriorNpc\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple = 1.f);

WarriorNpc\_t(character\_t \*copyedNpc, sf::Vector2f sporPoint, float powerMultiple = 1.f);

~WarriorNpc\_t();

};

class ZombieWitch\_t : public Npc\_t {

virtual void setTypeStats();

public:

ZombieWitch\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple = 1.f);

ZombieWitch\_t(character\_t \*copyedNpc, sf::Vector2f sporPoint, float powerMultiple = 1.f);

~ZombieWitch\_t();

};

class FatZombie\_t : public Npc\_t {

virtual void setTypeStats();

public:

FatZombie\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple = 1.f);

FatZombie\_t(character\_t \*copyedNpc, sf::Vector2f sporPoint, float powerMultiple = 1.f);

~FatZombie\_t();

};

class SkeletonMage\_t : public Npc\_t {

virtual void setTypeStats();

public:

SkeletonMage\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple = 1.f);

SkeletonMage\_t(character\_t \*copyedNpc, sf::Vector2f sporPoint, float powerMultiple = 1.f);

~SkeletonMage\_t();

};

================================= “Npc\_t.cpp” =================================

#include "Npc\_t.h"

Npc\_t::Npc\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, float \_x, float \_y, int \_width, int \_height, float \_statMultiple)

:character\_t(\_texture, \_bulletList, \_x, \_y, 0,0,\_width, \_height,\_clock)

{

spotCoords = sf::Vector2f(\_x,\_y);

spawnCoords = spotCoords;

powerMultiple = \_statMultiple;

stat.statMiltipler(powerMultiple);

state.swap(std::unique\_ptr<CharacterState\_t>(new CharacterStateMove\_t(\*this)));

spawnTime = std::numeric\_limits<sf::Int32>::max();

elemStatus = 1;

collision = false;

}

Npc\_t::Npc\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple) :

character\_t(\_texture, \_bulletList,\_spotCoords.x, \_spotCoords.y, 0, 0, \_width, \_height, \_clock)

{

spotCoords = \_spotCoords;

spawnCoords.x = spotCoords.x;

spawnCoords.y = spotCoords.y;

powerMultiple = \_multiple;

stat.statMiltipler(powerMultiple);

state.swap(std::unique\_ptr<CharacterState\_t>(new CharacterStateMove\_t(\*this)));

spawnTime = std::numeric\_limits<sf::Int32>::max();

elemStatus = 1;

collision = false;

}

Npc\_t::Npc\_t(character\_t \*copyedNpc, sf::Vector2f spotPoint, float powerMultiple):

character\_t(copyedNpc->getSpritePref().getTextureSharedPtr(), copyedNpc->getSkillGeneratorPtr()->getBulletList(), spotPoint.x, spotPoint.y, 0, 0,

copyedNpc->getWidth(), copyedNpc->getHeight(), copyedNpc->getClockPtr()) {

setStats(copyedNpc->getStats());

elemStatus = copyedNpc->getElemStatus();

spotCoords = spotPoint;

spawnCoords.x = spotPoint.x;

spawnCoords.y = spotPoint.y;

powerMultiple = powerMultiple;

stat.statMiltipler(powerMultiple);

state.swap(std::unique\_ptr<CharacterState\_t>(new CharacterStateMove\_t(\*this)));

spawnTime = std::numeric\_limits<sf::Int32>::max();

collision = copyedNpc->getCollision();

alive = copyedNpc->getAlive();

clock = copyedNpc->getClockPtr();

moveRadius = copyedNpc->getMoveRadius();

fraction = copyedNpc->getFraction();

timer = copyedNpc->getTimer();

}

Npc\_t::~Npc\_t()

{

}

void Npc\_t::attack() {

character\_t::attack();

}

void Npc\_t::setTypeStats() {

timer.attackCDcorrection(stat.attackSpeed);

timer.castDelayCorrection(stat.castSpeed);

}

//GET

sf::Vector2f Npc\_t::getSpotCoords() const{

return spotCoords;

}

float Npc\_t::getPowerMultiple() const {

return powerMultiple;

}

//MAGE CLASS

MageNpc\_t::MageNpc\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple) :

Npc\_t(\_texture,\_bulletList,\_clock,\_spotCoords,\_width, \_height, \_multiple)

{

setTypeStats();

}

MageNpc\_t::MageNpc\_t(character\_t \*copyedNpc, sf::Vector2f spotPoint, float powerMultiple) :

Npc\_t(copyedNpc, spotPoint, powerMultiple)

{

setTypeStats();

}

MageNpc\_t::~MageNpc\_t() {

}

void MageNpc\_t::setTypeStats() {

elemStatus = 5;//FIRE BALLS

stat.attackRange = 200.f;

stat.stdattackPower = 20.f;

stat.stdPhysDef = 5.f;

stat.stdMagDef = 15.f;

stat.damageRand = 2.f;

stat.stdHP = 50.f;

stat.visionDistance = 220.f;

stat.stdAttackSpeed = 3.f;

stat.stdSpeed = 0.1f;

defaultAllStats();

Npc\_t::setTypeStats();

}

//WARRIOR CLASS

WarriorNpc\_t::WarriorNpc\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple) :

Npc\_t(\_texture, \_bulletList, \_clock, \_spotCoords, \_width, \_height, \_multiple)

{

setTypeStats();

}

WarriorNpc\_t::WarriorNpc\_t(character\_t \*copyedNpc, sf::Vector2f spotPoint, float powerMultiple) :

Npc\_t(copyedNpc, spotPoint, powerMultiple)

{

setTypeStats();

}

WarriorNpc\_t::~WarriorNpc\_t() {

}

void WarriorNpc\_t::setTypeStats() {

elemStatus = 1;//SWORD ATTACK

stat.attackRange = 40.f;

stat.stdPhysDef = 15.f;

stat.stdMagDef = 5.f;

stat.stdattackPower = 20.f;

stat.damageRand = 10.f;

stat.stdHP = 70.f;

stat.visionDistance = 250.f;

stat.stdAttackSpeed = 5.f;

stat.stdSpeed = 0.16f;

defaultAllStats();

Npc\_t::setTypeStats();

}

//ZOMBIE WITCH CLASS

ZombieWitch\_t::ZombieWitch\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple) :

Npc\_t(\_texture, \_bulletList, \_clock, \_spotCoords, \_width, \_height, \_multiple)

{

setTypeStats();

}

ZombieWitch\_t::ZombieWitch\_t(character\_t \*copyedNpc, sf::Vector2f spotPoint, float powerMultiple) :

Npc\_t(copyedNpc, spotPoint, powerMultiple)

{

setTypeStats();

}

ZombieWitch\_t::~ZombieWitch\_t() {

}

void ZombieWitch\_t::setTypeStats() {

elemStatus = 4;//FIRE LIGHTING

stat.attackRange = 250.f;

stat.stdPhysDef = 5.f;

stat.stdMagDef = 5.f;

stat.stdattackPower = 30.f;

stat.damageRand = 7.f;

stat.stdHP = 50.f;

stat.visionDistance = 300.f;

stat.stdAttackSpeed = 1.f;

stat.stdSpeed = 0.09f;

defaultAllStats();

Npc\_t::setTypeStats();

}

//FAT ZOMBIE CLASS

FatZombie\_t::FatZombie\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple) :

Npc\_t(\_texture, \_bulletList, \_clock, \_spotCoords, \_width, \_height, \_multiple)

{

setTypeStats();

}

FatZombie\_t::FatZombie\_t(character\_t \*copyedNpc, sf::Vector2f spotPoint, float powerMultiple) :

Npc\_t(copyedNpc, spotPoint, powerMultiple)

{

setTypeStats();

}

FatZombie\_t::~FatZombie\_t() {

}

void FatZombie\_t::setTypeStats() {

elemStatus = 2;//POISION BALL

stat.attackRange = 70.f;

stat.stdPhysDef = 30.f;

stat.stdMagDef = 30.f;

stat.stdattackPower = 50.f;

stat.damageRand = 7.f;

stat.stdHP = 100.f;

stat.visionDistance = 250.f;

stat.stdAttackSpeed = 1.f;

stat.stdSpeed = 0.08f;

defaultAllStats();

Npc\_t::setTypeStats();

}

//SKELETON MAGE CLASS

SkeletonMage\_t::SkeletonMage\_t(std::shared\_ptr<sf::Texture>\_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Clock \*\_clock, sf::Vector2f \_spotCoords, int \_width, int \_height, float \_multiple) :

Npc\_t(\_texture, \_bulletList, \_clock, \_spotCoords, \_width, \_height, \_multiple)

{

setTypeStats();

}

SkeletonMage\_t::SkeletonMage\_t(character\_t \*copyedNpc, sf::Vector2f spotPoint, float powerMultiple) :

Npc\_t(copyedNpc, spotPoint, powerMultiple)

{

setTypeStats();

}

SkeletonMage\_t::~SkeletonMage\_t() {

}

void SkeletonMage\_t::setTypeStats() {

elemStatus = 12;

stat.attackRange = 300.f;

stat.stdPhysDef = 20.f;

stat.stdMagDef = 20.f;

stat.stdattackPower = 100.f;

stat.damageRand = 15.f;

stat.stdHP = 110.f;

stat.visionDistance = 350.f;

stat.stdAttackSpeed = -2.f;

stat.stdSpeed = 0.11f;

defaultAllStats();

Npc\_t::setTypeStats();

}

================================= “obPreference.h” =================================

#pragma once

#include <SFML\Graphics.hpp>

#include <memory>

const int DEFAULT\_HEIGHT = 32;

const int DEFAULT\_WIDTH = 32;

class obPreference

{

private:

std::shared\_ptr<sf::Texture> texture;

sf::Sprite sprite;

int spriteCoordX;

int spriteCoordY;

int height;

int width;

public:

obPreference();

obPreference(std::string \_fileTexturePath);

obPreference(std::string \_fileTexturePath, int \_coordX, int \_coordY, int \_width, int \_height);

obPreference(std::shared\_ptr<sf::Texture>&\_texture, int \_coordX, int \_coordY, int \_width, int \_height);

virtual ~obPreference();

void setOriginToCenter();

//GET

std::shared\_ptr<sf::Texture> &getTextureSharedPtr();

sf::Texture getTexture() const;

sf::Sprite &getSprite();

int getCoordX() const;

int getCoordY() const;

int getHeight() const;

int getWidth() const;

//SET

void setTexturePtr(std::shared\_ptr<sf::Texture> &newTexture);

void setTexturePos(int coordX, int coordY);

void setSize( int newWidth,int newHeight);

void setSpritePos(float coordX, float coordY);

void setRotation(float \_rotation);

//FOR CHECK COLLISION

sf::FloatRect getRect()const;

};

================================= “obPreference.cpp” =================================

#include "obPreference.h"

obPreference::obPreference() {

spriteCoordX = 0;

spriteCoordY = 0;

height = DEFAULT\_HEIGHT;

width = DEFAULT\_WIDTH;

texture = std::make\_shared<sf::Texture>();

texture->loadFromFile("img/default.png");

sprite.setTexture(\*texture);

sprite.setTextureRect(sf::IntRect(spriteCoordX, spriteCoordY,width,height));

}

obPreference::obPreference(std::string \_fileTexturePath)

{

spriteCoordX = 0;

spriteCoordY = 0;

height = DEFAULT\_HEIGHT;

width = DEFAULT\_WIDTH;

texture = std::make\_shared<sf::Texture>();

texture->loadFromFile(\_fileTexturePath);

sprite.setTexture(\*texture);

sprite.setTextureRect(sf::IntRect(spriteCoordX, spriteCoordY, width, height));

}

obPreference::obPreference(std::string \_fileTexturePath, int \_coordX, int \_coordY, int \_width, int \_height){

spriteCoordX = \_coordX;

spriteCoordY = \_coordY;

height = \_height;

width = \_width;

texture = std::make\_shared<sf::Texture>();

texture->loadFromFile(\_fileTexturePath);

sprite.setTexture(\*texture);

sprite.setTextureRect(sf::IntRect(spriteCoordX, spriteCoordY, width, height));

}

obPreference::obPreference(std::shared\_ptr<sf::Texture> &\_texture, int \_coordX, int \_coordY, int \_width, int \_height) {

spriteCoordX = \_coordX;

spriteCoordY = \_coordY;

height = \_height;

width = \_width;

texture = \_texture;

sprite.setTexture(\*texture);

sprite.setTextureRect(sf::IntRect(spriteCoordX, spriteCoordY, width, height));

}

obPreference::~obPreference()

{

}

void obPreference::setOriginToCenter() {

sprite.setOrigin(static\_cast<float>(width) / 2, static\_cast<float>(height) / 2);

}

//GET

std::shared\_ptr<sf::Texture> &obPreference::getTextureSharedPtr(){

return texture;

}

sf::Texture obPreference::getTexture() const{

return \*texture;

}

sf::Sprite &obPreference::getSprite() {

return sprite;

}

int obPreference::getCoordX() const {

return spriteCoordX;

}

int obPreference::getCoordY() const {

return spriteCoordY;

}

int obPreference::getHeight() const {

return height;

}

int obPreference::getWidth() const {

return width;

}

//SET

void obPreference::setTexturePtr(std::shared\_ptr<sf::Texture> &newTexture) {

texture = newTexture;

sprite.setTexture(\*texture);

}

void obPreference::setTexturePos(int coordX, int coordY) {

spriteCoordX = coordX;

spriteCoordY = coordY;

sprite.setTextureRect(sf::IntRect(coordX,coordY,width,height));

}

void obPreference::setSize( int newWidth,int newHeight) {

width = newWidth;

height = newHeight;

sprite.setTextureRect(sf::IntRect(spriteCoordX, spriteCoordY, width, height));

}

void obPreference::setSpritePos(float coordX, float coordY) {

sprite.setPosition(coordX, coordY);

}

void obPreference::setRotation(float \_rotation) {

sprite.setRotation(\_rotation);

}

//FOR CHECK COLLISION

sf::FloatRect obPreference::getRect()const {

sf::FloatRect rect(static\_cast<float>(spriteCoordX), static\_cast<float>(spriteCoordY), static\_cast<float>(width), static\_cast<float>(height));

return rect;

}

================================= “physOb.h” =================================

#pragma once

#include <SFML\Graphics.hpp>

#include "obPreference.h"

#include "additional.h"

class physOb\_t

{

protected:

obPreference spritePref;

float posX;

float posY;

float dX;

float dY;

bool destroyble;

bool alive;

int hitsToDestroy;

bool collision;

int fraction;

animation::direction direction;

std::vector<int> dropList;

float frame;

virtual void animation();

virtual void updateFrame();

virtual bool checkTimer(sf::Clock \*clock, sf::Int32 startTime, sf::Int32 \_time);

public:

physOb\_t();

physOb\_t(float \_posX, float \_posY);

physOb\_t(float \_posX, float \_posY, std::string fileName, int \_coordX, int \_coordY, int \_width, int \_height);

physOb\_t(float \_posX, float \_posY, std::shared\_ptr<sf::Texture>\_texture, int \_coordX, int \_coordY, int \_width, int \_height);

virtual ~physOb\_t();

virtual bool kill();

virtual void update(float \_speed);

bool checkCollision(physOb\_t &Object);

virtual bool collisionHandler(physOb\_t &Object, float \_speed, float \_borderError = -15.f);

virtual bool checkAlive();

virtual float takeDamage(float \_dmg, bool \_dmgType, elements::element \_elem);

virtual float takeHeal(float \_heal);

//GET

sf::FloatRect getCollisionRect() const;

physOb\_t \*getPtr();

float getFrame() const;

obPreference &getSpritePref();

void setSpriteSize(int \_x, int \_y);

void setRotation(float \_val);

sf::Sprite &getSprite();

int getWidth() const;

int getHeight()const;

int getFraction() const;

float getdX() const;

float getdY() const;

float getPosX() const;

float getPosY() const;

bool getDestroyble() const;

bool getAlive() const;

bool getCollision() const;

int gethitsToDestroy() const;

virtual sf::Int32 getStartTime() const;

virtual sf::Int32 getTimeCd()const;

sf::Vector2f getPos() const;

sf::Vector2f getPosOfCenter() const;

sf::FloatRect getFloatRect() const;

//SET

float setPosX(float \_x);

float setPosY(float \_y);

int setFraction(int \_fract);

bool setDestroyble(bool \_d);

bool setAlive(bool \_live);

bool setCollision(bool \_collision);

int sethitsToDestroy(int \_hitsToDestroy);

void setTexturePtr(std::shared\_ptr<sf::Texture> &newTexture);

float setdX(float \_x);

float setdY(float \_y);

};

================================= “physOb.cpp” =================================

#include "physOb\_t.h"

bool equal(float &a, float &b) {

return (abs(a - b) < std::numeric\_limits<float>::epsilon()) ? true : false;

}

bool inInterval(float &x, float a, float b) {

return (x > a && x < b) ? true : false;

}

physOb\_t::physOb\_t()

{

dX = 0.f;

dY = 0.f;

posX = 0;

posY = 0;

destroyble = false;

alive = true;

hitsToDestroy = 1;

collision = true;

spritePref.setSpritePos(posX, posY);

fraction = -1;

}

physOb\_t::physOb\_t(float \_posX, float \_posY) {

dX = 0.f;

dY = 0.f;

posX = \_posX;

posY = \_posY;

destroyble = false;

alive = true;

hitsToDestroy = 1;

collision = true;

spritePref.setSpritePos(posX, posY);

fraction = -1;

}

physOb\_t::physOb\_t(float \_posX, float \_posY, std::string fileName, int \_coordX, int \_coordY, int \_width, int \_height) :spritePref(fileName, \_coordX, \_coordY, \_width, \_height) {

dX = 0.f;

dY = 0.f;

destroyble = false;

alive = true;

hitsToDestroy = 1;

collision = true;

fraction = -1;

posX = \_posX;

posY = \_posY;

spritePref.setSpritePos(posX, posY);

}

physOb\_t::physOb\_t(float \_posX, float \_posY, std::shared\_ptr<sf::Texture>\_texture, int \_coordX, int \_coordY, int \_width, int \_height) : spritePref(\_texture, \_coordX, \_coordY, \_width, \_height) {

dX = 0.f;

dY = 0.f;

destroyble = false;

alive = true;

hitsToDestroy = 1;

collision = true;

fraction = -1;

posX = \_posX;

posY = \_posY;

spritePref.setSpritePos(posX, posY);

}

physOb\_t::~physOb\_t()

{

}

bool physOb\_t::kill() {

alive = false;

return alive;

}

void physOb\_t::update(float \_speed) {

if (alive) {

if ((dX > 0) && (abs(dY) < FLT\_EPSILON)) {

direction = animation::RIGHT;

}

else if ((dX < 0) && (abs(dY) < FLT\_EPSILON)) {

direction = animation::LEFT;

}

else if ((dY > 0) && (abs(dX) < FLT\_EPSILON)) {

direction = animation::BOTTOM;

}

else if ((dY < 0) && (abs(dX) < FLT\_EPSILON)) {

direction = animation::TOP;

}

posX += dX \* \_speed;

posY += dY \* \_speed;

dX = 0.f;

dY = 0.f;

spritePref.setSpritePos(posX, posY);

updateFrame();

animation();

}

}

bool physOb\_t::checkCollision(physOb\_t &Object) {

if (getFloatRect().intersects(Object.getCollisionRect())) {

return true;

}

return false;

}

bool physOb\_t::collisionHandler(physOb\_t &Object, float \_speed, float \_borderError) {

if (collision && Object.collision) {

float zero = std::numeric\_limits<float>::epsilon();

float speedX = abs(dX) \* \_speed;

float speedY = abs(dY) \* \_speed;

if (direction == animation::RIGHT) {

posX = Object.getPosX() - getWidth() - \_borderError;

} else if (direction == animation::LEFT) {

posX = Object.getPosX() + Object.getWidth() + \_borderError;

} else if (direction == animation::BOTTOM) {

posY = Object.getPosY() - getHeight() - \_borderError;

} else if (direction == animation::TOP) {

posY = Object.getPosY() + Object.getHeight() + \_borderError;

}

dX = 0;

dY = 0;

return true;

}

return false;

}

bool physOb\_t::checkTimer(sf::Clock \*clock, sf::Int32 startTime, sf::Int32 \_time) {

sf::Int32 curTime = clock->getElapsedTime().asMilliseconds();

return (abs(curTime - startTime) > \_time) ? false : true;

}

bool physOb\_t::checkAlive() {

if (alive) {

alive = (hitsToDestroy > 0) ? true : false;

}

return alive;

}

float physOb\_t::takeDamage(float \_dmg, bool \_dmgType, elements::element \_elem) {

if (alive) {

--hitsToDestroy;

}

return \_dmg;

}

float physOb\_t::takeHeal(float \_heal) {

if (alive) {

++hitsToDestroy;

}

return \_heal;

}

void physOb\_t::updateFrame() {

frame += animation::frameSpeed;

if (frame > animation::frameRate) {

frame -= animation::frameRate;

}

}

void physOb\_t::animation() {

}

//GET

sf::FloatRect physOb\_t::getCollisionRect() const {

float correctionError = animation::HEROES\_SPRITE\_COLLISION\_CORRECTION\_BORDER;

return sf::FloatRect(posX + correctionError, posY + correctionError, getWidth()- correctionError,getHeight() - correctionError);

}

physOb\_t \*physOb\_t::getPtr() {

return this;

}

float physOb\_t::getFrame() const {

return frame;

}

obPreference &physOb\_t::getSpritePref() {

return spritePref;

}

sf::Sprite &physOb\_t::getSprite() {

return spritePref.getSprite();

}

int physOb\_t::getWidth() const {

return spritePref.getWidth();

}

int physOb\_t::getHeight()const {

return spritePref.getHeight();

}

int physOb\_t::getFraction() const {

return fraction;

}

float physOb\_t::getdX() const {

return dX;

}

float physOb\_t::getdY() const {

return dY;

}

float physOb\_t::getPosX() const {

return posX;

}

float physOb\_t::getPosY() const {

return posY;

}

bool physOb\_t::getDestroyble() const {

return destroyble;

}

bool physOb\_t::getAlive() const {

return alive;

}

bool physOb\_t::getCollision() const {

return collision;

}

int physOb\_t::gethitsToDestroy() const {

return hitsToDestroy;

}

sf::Int32 physOb\_t::getStartTime() const {

return 0;

}

sf::Int32 physOb\_t::getTimeCd()const {

return std::numeric\_limits<sf::Int32>::max();

}

sf::Vector2f physOb\_t::getPos() const {

return sf::Vector2f (posX, posY);

}

sf::Vector2f physOb\_t::getPosOfCenter() const {

return sf::Vector2f(posX + getWidth()/2, posY + getHeight() / 2);

}

sf::FloatRect physOb\_t::getFloatRect() const {

return sf::FloatRect(posX, posY, static\_cast<float>(getWidth()), static\_cast<float>(getHeight()));

}

//SET

void physOb\_t::setSpriteSize(int \_x, int \_y){

spritePref.setSize(\_x, \_y);

}

void physOb\_t::setRotation(float \_val) {

spritePref.setRotation(\_val);

}

float physOb\_t::setPosX(float \_x){

posX = \_x;

return posX;

}

float physOb\_t::setPosY(float \_y){

posY = \_y;

return posY;

}

int physOb\_t::setFraction(int \_fract) {

fraction = \_fract;

return fraction;

}

bool physOb\_t::setDestroyble(bool \_d){

destroyble = \_d;

return destroyble;

}

bool physOb\_t::setAlive(bool \_live){

alive = \_live;

return alive;

}

bool physOb\_t::setCollision(bool \_collision){

collision = \_collision;

return collision;

}

int physOb\_t::sethitsToDestroy(int \_hitsToDestroy){

hitsToDestroy = \_hitsToDestroy;

return hitsToDestroy;

}

void physOb\_t::setTexturePtr(std::shared\_ptr<sf::Texture> &newTexture) {

spritePref.setTexturePtr(newTexture);

}

float physOb\_t::setdX(float \_x) {

dX = \_x;

return dX;

}

float physOb\_t::setdY(float \_y) {

dY = \_y;

return dY;

}

================================= “player\_t.h” =================================

#pragma once

#include "character\_t.h"

#include "bullet\_t.h"

#include "additional.h"

class player\_t : public character\_t

{

public:

player\_t(sf::Vector2f \_pos, std::string filename, int \_width, int \_height, sf::Clock \*\_clock, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList);

player\_t(std::shared\_ptr<sf::Texture> \_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Vector2f \_pos, int \_width, int \_height, sf::Clock \*\_clock);

virtual ~player\_t();

virtual void attack();

void useMP(float \_mp);

void update(float \_speed);

};

================================= “player\_t.cpp” =================================

#include "player\_t.h"

//\*

player\_t::player\_t(sf::Vector2f \_pos, std::string fileName, int \_width, int \_height, sf::Clock \*\_clock, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList) : character\_t(\_pos.x, \_pos.y, fileName, 0,0, \_width, \_height, \_clock, \_bulletList)

{

fraction = 0;

elemStatus = 5;

collision = true;

state.swap(std::unique\_ptr<CharacterState\_t>(new CharacterPlayerControll\_t(\*this)));

stat.stdattackPower = 20.f;

stat.attackRange = 250.f;

stat.stdPhysDef = 15.f;

stat.stdMagDef = 15.f;

stat.damageRand = 10.f;

stat.stdHP = 350.f;

stat.visionDistance = 300.f;

stat.stdAttackSpeed = 5.f;

stat.stdSpeed = 0.15f;

stat.stdCastSpeed = 17.f;

defaultAllStats();

timer.attackCDcorrection(stat.stdAttackSpeed);

timer.castDelayCorrection(stat.stdCastSpeed);

}

player\_t::player\_t(std::shared\_ptr<sf::Texture> \_texture, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList, sf::Vector2f \_pos, int \_width, int \_height, sf::Clock \*\_clock) :

character\_t(\_texture, \_bulletList, \_pos.x, \_pos.y, 0,0, \_width, \_height, \_clock)

{

fraction = 0;

elemStatus = 5;

collision = true;

state.swap(std::unique\_ptr<CharacterState\_t>(new CharacterPlayerControll\_t(\*this)));;

stat.stdattackPower = 20.f;

stat.attackRange = 250.f;

stat.stdPhysDef = 25.f;

stat.stdMagDef = 15.f;

stat.damageRand = 10.f;

stat.stdHP = 350.f;

stat.visionDistance = 300.f;

stat.stdAttackSpeed = 5.f;

stat.stdSpeed = 0.15f;

stat.stdCastSpeed = 17.f;

defaultAllStats();

timer.attackCDcorrection(stat.stdAttackSpeed);

timer.castDelayCorrection(stat.stdCastSpeed);

}

//\*/

player\_t::~player\_t()

{

}

void player\_t::update(float \_speed) {

character\_t::update(\_speed);

if (((this)->getState()->getStateNum() == 4) && (this)->checkSkillGenerator() && ((this)->getTimer().castReady())) {

(this)->generateSkillAndClearElemList();

(this)->changeState(new CharacterPlayerControll\_t(\*this));

}

else if ((this)->getTimer().castReady()){

(this)->changeState(new CharacterPlayerControll\_t(\*this));

}

}

void player\_t::attack() {

if (stat.MP > 0) {

if (timer.attackReady()) {

timer.updateAttackCD();

skill->useSkill();

}

}

}

void player\_t::useMP(float \_mp) {

character\_t::useMP(\_mp);

if (stat.MP < FLT\_EPSILON){

elemStatus = 0;

}

}

================================= “skillObGenerator\_t.h” ================================

#pragma once

#include <SFML/Graphics.hpp>

#include <list>

#include <memory>

#include "character\_t.h"

#include "bullet\_t.h"

#include "additional.h"

class character\_t;

class skillObGenerator\_t

{

character\_t \*character;

std::list<std::unique\_ptr <bullet\_t>> &skillGeneratorBulletList;

public:

skillObGenerator\_t(character\_t \*\_character, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList);

virtual ~skillObGenerator\_t();

void useSkill();

void upCharacterStat(characterStats\_t &\_stats);

std::list<std::unique\_ptr <bullet\_t>> &getBulletList();

};

=============================== “skillObGenerator\_t.cpp” ================================

#include "skillObGenerator\_t.h"

skillObGenerator\_t::skillObGenerator\_t(character\_t \*\_character, std::list<std::unique\_ptr <bullet\_t>> &\_bulletList):skillGeneratorBulletList(\_bulletList)

{

character = \_character;

}

skillObGenerator\_t::~skillObGenerator\_t()

{

}

void skillObGenerator\_t::useSkill() {

/\*//SKILL LIST

1 - sword attack

2 - poision ball

3 - speed buff

4 - fire lighting

5 - fire ball

6 - power buff

7 - attack speed buff

10 - mass earth balls

11 - combo ball

12 - fire breath

17 - earth ball

18 - bang ball

24 - heal stone

//\*/

size\_t skillType = character->getElemStatus();

switch (skillType) {

case 3: {

characterStats\_t tempStat;

tempStat.resetStats();

tempStat.speed += 0.05f;

upCharacterStat(tempStat);

character->changeEffect(new Effect\_t(character, tempStat));

character->getEffectPtr()->useEffect();

float buffMPcost = 50.f;

character->useMP(buffMPcost);

break;

}

case 6: {

characterStats\_t tempStat;

tempStat.resetStats();

tempStat.attackPower += 20.f;

tempStat.physDef += 20.f;

tempStat.magDef += 20.f;

upCharacterStat(tempStat);

character->changeEffect(new Effect\_t(character, tempStat, 10000));

character->getEffectPtr()->useEffect();

float buffMPcost = 50.f;

character->useMP(buffMPcost);

break;

}

case 7: {

characterStats\_t tempStat;

tempStat.resetStats();

tempStat.attackSpeed += 20.f;

upCharacterStat(tempStat);

character->changeEffect(new Effect\_t(character, tempStat, 3000));

character->getEffectPtr()->useEffect();

float buffMPcost = 50.f;

character->useMP(buffMPcost);

break;

}

case 24: {//HEAL BALL

std::shared\_ptr<sf::Texture>tempTexture = std::make\_shared<sf::Texture>();

tempTexture->loadFromFile(animation::BULLET\_CRYSTAL\_HEAL\_TEXTURE\_FILE);

std::unique\_ptr<bullet\_t> tempBullet(new bullet\_t(character->getClockPtr(), character, character->getTargetPos()));

tempBullet->setRotation(0.f);

tempBullet->getSprite().setScale(2.f,2.f);

tempBullet->setTexturePtr(tempTexture);

tempBullet->setSpriteSize(tempTexture->getSize().x / 3, tempTexture->getSize().y);

tempBullet->setRng(character->getStats().attackRange);

tempBullet->setCollision(false);

tempBullet->setType(false);

tempBullet->setdX(0.f);

tempBullet->setdY(0.f);

tempBullet->setSpeed(0.f);

tempBullet->setElement(elements::EARTH);

tempBullet->setTimer(5000);

tempBullet->setAOE(50.f);

tempBullet->setDmgDelay(500);

tempBullet->setRng(1000.f);

float tempDmg = 5.f + character->getComputedDmg() \* 0.1f;

tempBullet->setDmg(tempDmg);

tempBullet->setMPCost(30.f);

character->useMP(tempBullet->getMPCost());

skillGeneratorBulletList.push\_back(std::move(tempBullet));

break;

}

case 1: {//PhysAttack

std::shared\_ptr<sf::Texture>tempTexture = std::make\_shared<sf::Texture>();

tempTexture->loadFromFile(animation::SWORD\_ATTACK\_TEXTURE\_FILE);

std::unique\_ptr<bullet\_t> tempBullet(new bullet\_t(character->getClockPtr(), character, character->getTargetPos()));

tempBullet->setTexturePtr(tempTexture);

tempBullet->setSpriteSize(tempTexture->getSize().x/3, tempTexture->getSize().y);

tempBullet->setCollision(false);

tempBullet->speedMultiple(1.5f);

tempBullet->setElement(elements::NONE);

tempBullet->setTimer(500);

tempBullet->setAOE(20.f);

tempBullet->setRng((character->getStats().attackRange > 150.f)?150.f: character->getStats().attackRange);

float tempDmg = 30.f + character->getComputedDmg();

tempBullet->setDmg(tempDmg);

character->useMP(0.f);

skillGeneratorBulletList.push\_back(std::move(tempBullet));

break;

}

case 2: {//POISION BALL

std::shared\_ptr<sf::Texture>tempTexture = std::make\_shared<sf::Texture>();

tempTexture->loadFromFile(animation::BULLET\_POISION\_BALL\_TEXTURE\_FILE);

std::unique\_ptr<bullet\_t> tempBullet(new bullet\_t(character->getClockPtr(), character, character->getTargetPos()));

tempBullet->setRotation(0.f);

tempBullet->setSpriteSize(tempTexture->getSize().x / 3, tempTexture->getSize().y);

tempBullet->setTexturePtr(tempTexture);

tempBullet->setRng(character->getStats().attackRange);

tempBullet->setCollision(false);

tempBullet->speedMultiple(1.5f);

tempBullet->setElement(elements::EARTH);

tempBullet->setTimer(4000);

tempBullet->setAOE(80.f);

tempBullet->setMPCost(10.f);

float tempDmg = 10.f + character->getComputedDmg();

tempBullet->setDmg(tempDmg);

character->useMP(tempBullet->getMPCost());

skillGeneratorBulletList.push\_back(std::move(tempBullet));

break;

}

case 4: {//FIRE LIGHTING

std::shared\_ptr<sf::Texture>tempTexture = std::make\_shared<sf::Texture>();

tempTexture->loadFromFile(animation::BULLET\_FIRE\_LIGHTING\_TEXTURE\_FILE);

std::unique\_ptr<bullet\_t> tempBullet(new bullet\_t(character->getClockPtr(), character, character->getTargetPos()));

tempBullet->setSpriteSize(tempTexture->getSize().x / 3, tempTexture->getSize().y);

tempBullet->setTexturePtr(tempTexture);

tempBullet->setCollision(true);

tempBullet->speedMultiple(2.f);

tempBullet->setElement(elements::FIRE);

tempBullet->setTimer(2000);

tempBullet->setRng(character->getStats().attackRange);

float tempDmg = 5.f + character->getComputedDmg();

tempBullet->setDmg(tempDmg);

tempBullet->setMPCost(10.f);

character->useMP(tempBullet->getMPCost());

skillGeneratorBulletList.push\_back(std::move(tempBullet));

break;

}

case 10: {//SMALL EARTH BALLS

std::shared\_ptr<sf::Texture>tempTexture = std::make\_shared<sf::Texture>();

tempTexture->loadFromFile(animation::BULLET\_SMALL\_EARTH\_BALL\_TEXTURE\_FILE);

std::unique\_ptr<bullet\_t> tempBullet(new bullet\_t(character->getClockPtr(), character, character->getTargetPos()));

tempBullet->setSpriteSize(tempTexture->getSize().x / 3, tempTexture->getSize().y);

tempBullet->setTexturePtr(tempTexture);

tempBullet->setAOE(50.f);

tempBullet->setCollision(false);

tempBullet->speedMultiple(1.5f);

tempBullet->setElement(elements::EARTH);

tempBullet->setTimer(2000);

tempBullet->setRng(character->getStats().attackRange);

float tempDmg = 7.f + character->getComputedDmg();

tempBullet->setDmg(tempDmg);

tempBullet->setMPCost(20.f);

character->useMP(tempBullet->getMPCost());

skillGeneratorBulletList.push\_back(std::move(tempBullet));

break;

}

case 5: {//FIRE BALL

std::shared\_ptr<sf::Texture>tempTexture = std::make\_shared<sf::Texture>();

tempTexture->loadFromFile(animation::BULLET\_FIRE\_BALL\_TEXTURE\_FILE);

std::unique\_ptr<bullet\_t> tempBullet(new bullet\_t(character->getClockPtr(), character, character->getTargetPos()));

tempBullet->setTexturePtr(tempTexture);

tempBullet->setSpriteSize(tempTexture->getSize().x /3, tempTexture->getSize().y);

tempBullet->setCollision(true);

tempBullet->speedMultiple(1.5f);

tempBullet->setElement(elements::FIRE);

tempBullet->setTimer(4000);

tempBullet->setRng(100.f + character->getStats().attackRange);

float tempDmg = 15.f + character->getComputedDmg();

tempBullet->setDmg(tempDmg);

tempBullet->setMPCost(5.f);

character->useMP(tempBullet->getMPCost());

skillGeneratorBulletList.push\_back(std::move(tempBullet));

break;

}

case 17: {//EARTH BALL

std::shared\_ptr<sf::Texture>tempTexture = std::make\_shared<sf::Texture>();

tempTexture->loadFromFile(animation::BULLET\_EARTH\_BALL\_TEXTURE\_FILE);

std::unique\_ptr<bullet\_t> tempBullet(new bullet\_t(character->getClockPtr(), character, character->getTargetPos()));

tempBullet->setTexturePtr(tempTexture);

tempBullet->setSpriteSize(tempTexture->getSize().x / 3, tempTexture->getSize().y);

tempBullet->setCollision(true);

tempBullet->speedMultiple(1.f);

tempBullet->setElement(elements::EARTH);

tempBullet->setTimer(4000);

tempBullet->setRng(100.f + character->getStats().attackRange);

tempBullet->setAOE(10.f);

tempBullet->setMPCost(20.f);

float tempDmg = 30.f + character->getComputedDmg();

tempBullet->setDmg(tempDmg);

character->useMP(tempBullet->getMPCost());

skillGeneratorBulletList.push\_back(std::move(tempBullet));

break;

}

case 12: {//FIRE BREATH

std::shared\_ptr<sf::Texture>tempTexture = std::make\_shared<sf::Texture>();

tempTexture->loadFromFile(animation::BULLET\_LAVA\_TEXTURE\_FILE);

int amountPools = static\_cast<int>(character->getStats().attackRange) / animation::LARGE\_SKILL\_WIDTH;

{

std::unique\_ptr<bullet\_t> tempBullet(new bullet\_t(character->getClockPtr(), character, character->getTargetPos()));

tempBullet->setRotation(0.f);

tempBullet->setTexturePtr(tempTexture);

tempBullet->setSpriteSize(tempTexture->getSize().x / 3, tempTexture->getSize().y);

tempBullet->setPosX(character->getPosOfCenter().x - character->getWidth()/2);

tempBullet->setPosY(character->getPosOfCenter().y - character->getHeight() / 2);

tempBullet->speedMultiple(1.2f);

tempBullet->setCollision(false);

tempBullet->setElement(elements::FIRE);

tempBullet->setTimer(4000);

tempBullet->setDmgDelay(300);

tempBullet->setAOE(60.f);

tempBullet->setMPCost(10.f);

float tempDmg = (2.f + character->getComputedDmg() \* 0.5f);

tempBullet->setDmg(tempDmg);

character->useMP(tempBullet->getMPCost());

skillGeneratorBulletList.push\_back(std::move(tempBullet));

}

break;

}

case 18: {//BANG BALL

std::shared\_ptr<sf::Texture>tempTexture = std::make\_shared<sf::Texture>();

tempTexture->loadFromFile(animation::BULLET\_BANG\_BALL\_TEXTURE\_FILE);

std::unique\_ptr<bullet\_t> tempBullet(new bullet\_t(character->getClockPtr(), character, character->getTargetPos()));

tempBullet->setSpriteSize(tempTexture->getSize().x / 3, tempTexture->getSize().y);

tempBullet->setTexturePtr(tempTexture);

tempBullet->setRng(character->getStats().attackRange /2);

tempBullet->setCollision(false);

tempBullet->speedMultiple(0.5f);

tempBullet->setElement(elements::EARTH);

tempBullet->setTimer(2000);

float tempDmg = 60.f + character->getComputedDmg();

tempBullet->setDmg(tempDmg);

tempBullet->setMPCost(30.f);

character->useMP(tempBullet->getMPCost());

skillGeneratorBulletList.push\_back(std::move(tempBullet));

break;

}

case 11: {//COMBO BALL

std::shared\_ptr<sf::Texture>tempTexture = std::make\_shared<sf::Texture>();

tempTexture->loadFromFile(animation::BULLET\_COMBO\_BALL\_TEXTURE\_FILE);

std::unique\_ptr<bullet\_t> tempBullet(new bullet\_t(character->getClockPtr(), character, character->getTargetPos()));

tempBullet->setSpriteSize(tempTexture->getSize().x / 3, tempTexture->getSize().y);

tempBullet->setTexturePtr(tempTexture);

tempBullet->speedMultiple(2.f);

tempBullet->setAOE(30.f);

tempBullet->setCollision(true);

tempBullet->setElement(elements::WIND);

tempBullet->setTimer(4000);

tempBullet->setRng(character->getStats().attackRange);

float tempDmg = character->getComputedDmg();

tempBullet->setDmg(tempDmg);

tempBullet->setMPCost(15.f);

character->useMP(tempBullet->getMPCost());

skillGeneratorBulletList.push\_back(std::move(tempBullet));

break;

}

}

}

void skillObGenerator\_t::upCharacterStat(characterStats\_t &\_stats) {

character->getStats().upStat(\_stats);

}

//GET

std::list<std::unique\_ptr <bullet\_t>> &skillObGenerator\_t::getBulletList(){

return skillGeneratorBulletList;

}