

 

**Описание логики коллизий**

Game::loop()->game()

// Apply game objects moves

applyPlayerMoves(); // Keyboard moves, Mouse moves, checking player move border, Fire

Когда выстрел ->

bullets->addPlayerBullet(engine->getManager(), engine->getVideo(), texManager, enemies->createSelectors(engine->getManager()), rPos);

timer.lastFire = timer.now;

applyEnemiesMoves(); // move up/down, check Flying down(если враги опустились до начала экрана то проигрыш), Random enemy fires

Когда выстрел ->

// function createTriangleSelector from class GameObject1

bullets->addEnemyBullet(engine->getManager(), engine->getVideo(), texManager, player->createTriangleSelector(engine->getManager()), rPos);

timer.lastEnemyFire = timer.now;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

applyBulletsMoves(); // Move thier ways, Getting all nodes bullets hited & indecies for bullets

// Возвращает массив узлов с котрыми пересеклась пуля

array<ISceneNode \*> checkBullets(array<u32> \*Pindecies, array<u32> \*Eindecies);

Вызываем checkBullets () и затем в цикле проверяем есть ли в этом списке плеер или враги а так же Checking for player & enemy damage and death

array<ISceneNode \*> nodes = bullets->checkBullets(Pindecies, Eindecies);

\*\*\*\*\*\*\*\*

В checkBullets() проверяем пули плеера и врагов

array<ISceneNode \*> nodes;

u32 count = 0;

for (auto &i : pBullets)

{

if (!i.isDead())

{

// если пуля вышла за пределы установленного пути то удаляем ее и кладем номер в Findecies

if (i.checkEnd())

{

i.drop();

i.setHealth(0);

Pindecies->push\_back(count);

continue;

}

//if the function checkCollision() returns the node that was collided with bullet

// кладем этот узел в nodes, удаляем пулю и кладем ее номер в Findecies

ISceneNode \*buff = i.checkCollision();

if (buff)

{

nodes.push\_back(buff);

i.drop();

i.setHealth(0);

Pindecies->push\_back(count);

}

}

count++;

}

count = 0;

for (auto &i : eBullets){-||-}

return nodes;

}

ISceneNode \*Bullet::checkCollision()

{

vector3df pos = node->getPosition();

// проверка выхода пули за пределы установленного пути

if (pos.X > path.start.X && pos.X >= path.end.X)

health = 0;

else if (pos.X < path.start.X && pos.X <= path.end.X)

health = 0;

// if health <= 0

if (isDead())

return NULL;

// проверяем столкновение текущей пули со списком коллизий (class GameObject

// Special scene node animator for doing automatic collision detection and response.

//list<ISceneNodeAnimatorCollisionResponse \*> collisions;)

for (auto i : collisions)

{

// collisionOccurred() Возвращает true, если столкновение произошло во время последнего события animateNode ()

// Если true то возвращаем узел с коллизией

if (i->collisionOccurred())

{

health = 0;

return (i->getCollisionNode()); //! Returns the node that was collided with.

}

}

return NULL;

}

***Code:***

**Irrlicht Part**

*Events*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

class GameEvents : public IEventReceiver

{

private:

// Boolean array to store the state of the keys

bool keys[KEY\_KEY\_CODES\_COUNT];

position2di mousePosition;

bool leftButtonPressed;

bool mouseMove;

public:

GameEvents();

bool OnEvent(const SEvent &event);

bool isKeyDown(EKEY\_CODE key) const;

position2di getMousePosition() const;

bool isLeftButtonPressed() const;

bool isMouseMoveActive() const;

void setMousePosition(vector3df position);

void zeroMouse();

void zeroMouseMove();

};

GameEvents::GameEvents() : leftButtonPressed(false)

{

for (bool &i : keys)

i = false;

}

bool GameEvents::OnEvent(const SEvent &event)

{

if (event.EventType == EET\_KEY\_INPUT\_EVENT)

keys[event.KeyInput.Key] = event.KeyInput.PressedDown;

else

{

switch (event.MouseInput.Event)

{

case EMIE\_LMOUSE\_PRESSED\_DOWN:

leftButtonPressed = true;

break;

case EMIE\_LMOUSE\_LEFT\_UP:

leftButtonPressed = false;

break;

case EMIE\_MOUSE\_MOVED:

mousePosition.X = event.MouseInput.X;

mousePosition.Y = event.MouseInput.Y;

break;

default:

break;

}

}

return false;

}

bool GameEvents::isKeyDown(EKEY\_CODE key) const

{

return keys[key];

}

position2di GameEvents::getMousePosition() const

{

return mousePosition;

}

bool GameEvents::isLeftButtonPressed() const

{

return leftButtonPressed;

}

void GameEvents::setMousePosition(const vector3df position)

{

mousePosition.Y = position.X;

mousePosition.X = position.Z;

}

void GameEvents::zeroMouse()

{

leftButtonPressed = false;

}

bool GameEvents::isMouseMoveActive() const {

return mouseMove;

}

void GameEvents::zeroMouseMove()

{

mouseMove = false;

}

Global

#include "irrLicht\irrlicht.h"

#include "IrrKlang\irrKlang.h"

#include <iostream>

#pragma comment(lib, "libs/Irrlicht.lib")

using namespace irr;

using namespace core;

using namespace video;

using namespace scene;

using namespace io;

using namespace gui;

using namespace irrklang;

enum Direction

{

UP,

DOWN,

LEFT,

RIGHT

};

#define IDLE\_VECTOR {0.f, 0.f, 0.f}

#define WINDOW\_WIDTH 1024

#define WINDOW\_HEIGHT 768

Timer

#include "Global.h"

struct Time

{

u32 lastTime;

u32 now;

u32 lastFire;

u32 lastStep;

u32 lastEnemyFire;

f32 delta;

u32 totalTime;

u32 pauseTime;

u32 lastFrame;

void zeroTimes()

{

lastTime = 0;

now = 0;

lastFire = 0;

lastStep = 0;

lastEnemyFire = 0;

totalTime = 0;

pauseTime = 0;

lastFrame = 0;

}

void addDelta(u32 delta)

{

lastTime += delta;

lastFire += delta;

lastStep += delta;

lastEnemyFire += delta;

totalTime += delta;

lastFrame += delta;

}

};

Irrlicht

#include "Global.h"

#include "GameEvents.h"

#define WINDOW\_COLOR\_DEPTH 32

#define WINDOW\_BACKGROUND\_COLOR {0, 0, 0, 0}

#define CAMERA\_POSITION { 0.f, -300.f, 0.f }

#define LIGHT\_POSITION { 0.f, -500.f, 0.f }

#define LIGHT\_RADIUS 5000

class Irrlicht {

private:

GameEvents \*reciever;

IrrlichtDevice \*device;

IVideoDriver \*video;

ISceneManager \*manager;

IGUIEnvironment \*gui;

dimension2du windowSize;

ICameraSceneNode \*camera;

public:

Irrlicht();

Irrlicht(wchar\_t \*name, dimension2du size);

ISceneManager \* getManager() const;

IGUIEnvironment \* getGUI() const;

IVideoDriver \* getVideo() const;

ISceneNode \* getSkyBox(path fileName) const;

dimension2du getCenter() const;

u32 getNow() const;

void setTime(u32 newTime);

bool isKeyPressed(EKEY\_CODE key) const;

void hideCursor(const bool flag);

void setCursorPosition(const dimension2du position);

void setCursorToCenter();

vector2di getCursorPosition() const;

bool isLeftMouseButtonPressed() const;

void resetInput();

void resetMouseMove();

bool isMouseActive();

bool run();

void drawAll();

bool drop();

};

#include "Irrlicht.h"

Irrlicht::Irrlicht() : reciever(NULL), device(NULL), video(NULL), manager(NULL), gui(NULL), windowSize(dimension2du(0, 0)), camera(NULL)

{

}

Irrlicht::Irrlicht(wchar\_t \*name, dimension2du size) : windowSize(size)

{

reciever = new GameEvents();

device = createDevice(EDT\_OPENGL, windowSize, WINDOW\_COLOR\_DEPTH, false, false, false, reciever);

if (!device)

return;

device->setWindowCaption(name);

video = device->getVideoDriver();

manager = device->getSceneManager();

gui = device->getGUIEnvironment();

camera = manager->addCameraSceneNode(NULL, CAMERA\_POSITION, IDLE\_VECTOR);

}

ISceneManager \*Irrlicht::getManager() const

{

return manager;

}

IGUIEnvironment \*Irrlicht::getGUI() const

{

return gui;

}

IVideoDriver \*Irrlicht::getVideo() const

{

return video;

}

ISceneNode \*Irrlicht::getSkyBox(path fileName) const

{

ISceneNode \*box = manager->addSkyBoxSceneNode(video->getTexture(fileName), NULL, NULL, NULL, NULL, NULL);

return box;

}

dimension2du Irrlicht::getCenter() const

{

return dimension2du(windowSize.Width / 2, windowSize.Height / 2);

}

u32 Irrlicht::getNow() const

{

return device->getTimer()->getTime();

}

void Irrlicht::setTime(u32 newTime)

{

device->getTimer()->setTime(newTime);

}

bool Irrlicht::isKeyPressed(EKEY\_CODE key) const

{

return reciever->isKeyDown(key);

}

void Irrlicht::hideCursor(const bool flag)

{

device->getCursorControl()->setVisible(!flag);

}

void Irrlicht::setCursorPosition(const dimension2du position)

{

s32 x = position.Width;

s32 y = position.Height;

device->getCursorControl()->setPosition(x, y);

}

void Irrlicht::setCursorToCenter()

{

s32 x = windowSize.Width / 2;

s32 y = windowSize.Height / 2;

device->getCursorControl()->setPosition(x, y);

}

vector2di Irrlicht::getCursorPosition() const

{

return device->getCursorControl()->getPosition();

}

bool Irrlicht::isLeftMouseButtonPressed() const

{

return reciever->isLeftButtonPressed();

}

void Irrlicht::resetInput()

{

reciever->zeroMouse();

//reciever->zeroMouseMove();

}

bool Irrlicht::isMouseActive() {

return reciever->isMouseMoveActive();

}

void Irrlicht::resetMouseMove()

{

reciever->zeroMouseMove();

}

bool Irrlicht::run()

{

return device->run();

}

void Irrlicht::drawAll()

{

video->beginScene(true, true, WINDOW\_BACKGROUND\_COLOR);

manager->drawAll();

gui->drawAll();

video->endScene();

}

bool Irrlicht::drop()

{

return device->drop();

}

\_\_\_\_\_\_\_\_\_\_\_

IrrKlang

#include "IrrKlang\irrKlang.h"

#pragma comment(lib, "libs/Irrklang.lib")

using namespace irrklang;

\_\_\_\_\_\_\_\_\_\_\_

#include "Game.h"

int main() {

Game game;

game.loop();

return 0;

}

GameObject

#include "Global.h"

#define MAX\_HEALTH 100

class GameObject

{

protected:

IAnimatedMeshSceneNode \*node;

list<ISceneNodeAnimator \*> animators;

// Special scene node animator for doing automatic collision detection and response.

list<ISceneNodeAnimatorCollisionResponse \*> collisions;

path name;

//s32 -> int

s32 health;

private:

void addAllAnimators();

public:

GameObject();

GameObject(IAnimatedMeshSceneNode \*node, list<ISceneNodeAnimator \*> animators, list<ISceneNodeAnimatorCollisionResponse \*> collisions, path name);

GameObject(const GameObject &object);

~GameObject();

/\*\* A scene node is a node in the hierarchical scene graph. Every scene

node may have children, which are also scene nodes. Children move

relative to their parent's position. If the parent of a node is not

visible, its children won't be visible either. In this way, it is for

example easily possible to attach a light to a moving car, or to place

a walking character on a moving platform on a moving ship.

\*/

ISceneNode \*getNode() const;

list<ISceneNodeAnimator \*> getAnimators() const;

//! Animates a scene node. Can animate position, rotation, material, and so on.

/\*\* A scene node animator is able to animate a scene node in a very simple way. It may

change its position, rotation, scale and/or material. There are lots of animators

to choose from. You can create scene node animators with the ISceneManager interface.

\*/

ISceneNodeAnimator \*getAnimatorByIndex(u32 index) const;

list<ISceneNodeAnimatorCollisionResponse \*> getCollisions() const;

ISceneNodeAnimatorCollisionResponse \*getCollisionByIndex(u32 index) const;

path getName() const;

s32 getHealth() const;

vector3df getPosition() const;

vector3df getRotation() const;

vector3df getScale() const;

void setPosition(const vector3df position);

void setRotation(const vector3df rotation);

void setScale(const vector3df scale);

void setNode(IAnimatedMeshSceneNode \*newNode);

void setAnimators(list<ISceneNodeAnimator \*> newAnimators);

void setCollisions(list<ISceneNodeAnimatorCollisionResponse \*> newCollisions);

void setName(const path newName);

void setHealth(const s32 newHealth);

bool createNode(ISceneManager \*manager);

ITriangleSelector \*createTriangleSelector(ISceneManager \*manager) const;

void addAnimator(ISceneNodeAnimator \*animator);

void addCollision(ISceneNodeAnimatorCollisionResponse \*collision);

virtual bool move(f32 delta);

void damage(u32 dmg);

bool isDead() const;

virtual void drop();

};

#include "GameObject.h"

/////////////////////////////////////////

// Constructors and Distructors

/////////////////////////////////////////

GameObject::GameObject() : node(NULL), animators({}), collisions({}), name(""), health(MAX\_HEALTH)

{

}

GameObject::GameObject(IAnimatedMeshSceneNode \*node, list<ISceneNodeAnimator \*> animators, list<ISceneNodeAnimatorCollisionResponse \*> collisions, path name) : node(node), animators(animators), collisions(collisions), name(name), health(MAX\_HEALTH)

{

if (node)

{

for (auto &i : animators)

node->addAnimator(i);

for (auto &i : collisions)

node->addAnimator(i);

}

}

GameObject::GameObject(const GameObject &object)

{

node = object.node;

for (auto &i : object.animators)

animators.push\_back(i);

for (auto &i : object.collisions)

collisions.push\_back(i);

name = object.name;

health = object.health;

}

GameObject::~GameObject()

{

if (node)

node = NULL;

animators.clear();

collisions.clear();

}

/////////////////////////////////////////

// Getters

/////////////////////////////////////////

ISceneNode \*GameObject::getNode() const

{

return node;

}

list<ISceneNodeAnimator \*> GameObject::getAnimators() const

{

return animators;

}

ISceneNodeAnimator \*GameObject::getAnimatorByIndex(u32 index) const

{

if (index >= animators.getSize())

return NULL;

else

{

auto iter = animators.begin();

for (u32 i = 0; i < index; i++)

iter++;

return \*iter;

}

}

list<ISceneNodeAnimatorCollisionResponse \*> GameObject::getCollisions() const

{

return collisions;

}

ISceneNodeAnimatorCollisionResponse \*GameObject::getCollisionByIndex(u32 index) const

{

if (index >= collisions.getSize())

return NULL;

else

{

auto iter = collisions.begin();

for (u32 i = 0; i < index; i++)

iter++;

return \*iter;

}

}

path GameObject::getName() const

{

return name;

}

s32 GameObject::getHealth() const

{

return health;

}

vector3df GameObject::getPosition() const

{

return node->getPosition();

}

vector3df GameObject::getRotation() const

{

return node->getRotation();

}

vector3df GameObject::getScale() const

{

return node->getScale();

}

/////////////////////////////////////////

// Setters

/////////////////////////////////////////

void GameObject::setPosition(const vector3df position)

{

node->setPosition(position);

}

void GameObject::setRotation(const vector3df rotation)

{

node->setRotation(rotation);

}

void GameObject::setScale(const vector3df scale)

{

node->setScale(scale);

}

void GameObject::setNode(IAnimatedMeshSceneNode \*newNode)

{

node->removeAnimators();

node->remove();

node = newNode;

addAllAnimators();

}

void GameObject::setAnimators(list<ISceneNodeAnimator \*> newAnimators)

{

animators.clear();

for (auto &i : newAnimators)

animators.push\_back(i);

if (node)

{

for (auto &i : animators)

node->addAnimator(i);

}

}

void GameObject::setCollisions(list<ISceneNodeAnimatorCollisionResponse\* > newCollisions)

{

collisions.clear();

for (auto &i : newCollisions)

collisions.push\_back(i);

if (node)

{

for (auto &i : collisions)

node->addAnimator(i);

}

}

void GameObject::setName(const path newName)

{

name = newName;

}

void GameObject::setHealth(const s32 newHealth)

{

health = newHealth;

}

/////////////////////////////////////////

// Additional functions

/////////////////////////////////////////

void GameObject::addAllAnimators()

{

for (auto &i : animators)

node->addAnimator(i);

for (auto &i : collisions)

node->addAnimator(i);

}

bool GameObject::createNode(ISceneManager \*manager)

{

IAnimatedMesh \*mesh = manager->getMesh(name);

if (!mesh)

return false;

node = manager->addAnimatedMeshSceneNode(mesh);

if (!node)

return false;

addAllAnimators();

return true;

}

ITriangleSelector \*GameObject::createTriangleSelector(ISceneManager \*manager) const

{

return manager->createTriangleSelectorFromBoundingBox(node);

}

void GameObject::addAnimator(ISceneNodeAnimator \*animator)

{

animators.push\_back(animator);

node->addAnimator(animator);

}

void GameObject::addCollision(ISceneNodeAnimatorCollisionResponse \*collision)

{

collisions.push\_back(collision);

node->addAnimator(collision);

}

bool GameObject::move(f32 delta)

{

return false;

}

void GameObject::damage(u32 dmg)

{

health -= dmg;

}

bool GameObject::isDead() const

{

return health <= 0;

}

void GameObject::drop()

{

if (node)

{

node->removeAnimators();

node->remove();

node = NULL;

animators.clear();

collisions.clear();

}

}

BulletStorage

#include "Global.h"

#include "Bullet.h"

#include "TextureManager.h"

#define BULLET\_PLAYER\_SPEED 800

#define BULLET\_ENEMY\_SPEED 400

#define BULLET\_ELIPSE\_BOUND { 5.f, 5.f, 5.f }

class BulletStorage

{

private:

list<Bullet> pBullets;

list<Bullet> eBullets;

public:

list<Bullet> getPBullets() const;

list<Bullet> getEBullets() const;

void setPBullets(list<Bullet> rockets);

void setEBullets(list<Bullet> rockets);

void addPlayerBullet(ISceneManager \*manager, IVideoDriver \*video, TextureManager \*&texManager, IMetaTriangleSelector \*world, vector3df position);

void addEnemyBullet(ISceneManager \*manager, IVideoDriver \*video, TextureManager \*&texManager, ITriangleSelector \*world, vector3df position);

// The method checkBullets() returns an array of nodes with which the bullet cuts off

array<ISceneNode \*> checkBullets(array<u32> \*Pindecies, array<u32> \*Eindecies);

void move(f32 delta);

void drop();

};

#include "BulletStorage.h"

list<Bullet> BulletStorage::getPBullets() const

{

return pBullets;

}

list<Bullet> BulletStorage::getEBullets() const

{

return eBullets;

}

void BulletStorage::setPBullets(list<Bullet> bullets)

{

pBullets = bullets;

}

void BulletStorage::setEBullets(list<Bullet> bullets)

{

eBullets = bullets;

}

//! Creates a special scene node animator for doing automatic collision detection and response.

/\*\* See ISceneNodeAnimatorCollisionResponse for details.

\param world: Triangle selector holding all triangles of the world with which

the scene node may collide. You can create a triangle selector with

ISceneManager::createTriangleSelector();

\param sceneNode: SceneNode which should be manipulated. After you added this animator

to the scene node, the scene node will not be able to move through walls and is

affected by gravity. If you need to teleport the scene node to a new position without

it being effected by the collision geometry, then call sceneNode->setPosition(); then

animator->setTargetNode(sceneNode);

\param ellipsoidRadius: Radius of the ellipsoid with which collision detection and

response is done. If you have got a scene node, and you are unsure about

how big the radius should be, you could use the following code to determine

it:

\code

const core::aabbox3d<f32>& box = yourSceneNode->getBoundingBox();

core::vector3df radius = box.MaxEdge - box.getCenter();

\endcode

\param gravityPerSecond: Sets the gravity of the environment, as an acceleration in

units per second per second. If your units are equivalent to metres, then

core::vector3df(0,-10.0f,0) would give an approximately realistic gravity.

You can disable gravity by setting it to core::vector3df(0,0,0).

\param ellipsoidTranslation: By default, the ellipsoid for collision detection is created around

the center of the scene node, which means that the ellipsoid surrounds

it completely. If this is not what you want, you may specify a translation

for the ellipsoid.

\param slidingValue: DOCUMENTATION NEEDED.

\return The animator. Attach it to a scene node with ISceneNode::addAnimator()

and the animator will cause it to do collision detection and response.

If you no longer need the animator, you should call ISceneNodeAnimator::drop().

See IReferenceCounted::drop() for more information. \*/

void BulletStorage::addPlayerBullet(ISceneManager \*manager, IVideoDriver \*video, TextureManager \*&texManager, IMetaTriangleSelector \*world, vector3df position)

{

Bullet bullet;

if (bullet.createBullet (manager, video, position, UP))

{

// Anim contains list of all tringleselectors for enemy

ISceneNodeAnimatorCollisionResponse \*anim = manager->createCollisionResponseAnimator(world, bullet.getNode(), BULLET\_ELIPSE\_BOUND, IDLE\_VECTOR);

// Add Collisions with enemys for each bullet (we add anim to collisions lists from GameObject class).

if (anim)

bullet.addCollision(anim);

pBullets.push\_back(bullet);

}

}

void BulletStorage::addEnemyBullet(ISceneManager \*manager, IVideoDriver \*video, TextureManager \*&texManager, ITriangleSelector \*world, vector3df position)

{

Bullet bullet;

if (bullet.createBullet(manager, video, position, DOWN))

{

ISceneNodeAnimatorCollisionResponse \*anim = manager->createCollisionResponseAnimator(world, bullet.getNode(), BULLET\_ELIPSE\_BOUND, IDLE\_VECTOR);

if (anim)

bullet.addCollision(anim);

eBullets.push\_back(bullet);

}

}

array<ISceneNode \*> BulletStorage::checkBullets(array<u32> \*Pindecies, array<u32> \*Eindecies)

{

array<ISceneNode \*> nodes;

u32 count = 0;

for (auto &i : pBullets)

{

if (!i.isDead())

{

// If the bullet is outside the established path, then delete it and put the number to Findecies

if (i.checkEnd())

{

i.drop();

i.setHealth(0);

Pindecies->push\_back(count);

continue;

}

// If the method checkCollision() returns the node that was collided with bullet -

// put this node to "nodes", then delete the bullet and put its number in Findecies

ISceneNode \*buff = i.checkCollision();

if (buff)

{

nodes.push\_back(buff);

i.drop();

i.setHealth(0);

Pindecies->push\_back(count);

}

}

count++;

}

count = 0;

for (auto &i : eBullets)

{

if (!i.isDead())

{

if (i.checkEnd())

{

i.drop();

i.setHealth(0);

Eindecies->push\_back(count);

continue;

}

ISceneNode \*buff = i.checkCollision();

if (buff)

{

nodes.push\_back(buff);

i.drop();

i.setHealth(0);

Eindecies->push\_back(count);

}

}

count++;

}

return nodes;

}

void BulletStorage::move(f32 delta)

{

for (auto &i : pBullets)

{

if (!i.isDead())

{

vector3df pos = i.getPosition();

pos.Z += delta \* BULLET\_PLAYER\_SPEED;

i.setPosition(pos);

}

}

for (auto &i : eBullets)

{

if (!i.isDead())

{

vector3df pos = i.getPosition();

pos.Z -= delta \* BULLET\_ENEMY\_SPEED;

i.setPosition(pos);

}

}

}

void BulletStorage::drop()

{

for (auto &i : pBullets)

i.drop();

for (auto &i : eBullets)

i.drop();

pBullets.clear();

eBullets.clear();

}

Bullet

#include "Global.h"

#include "GameObject.h"

#define BULLET\_END\_PATH 250

class Bullet : public GameObject

{

private:

line3df path;

public:

Bullet();

line3df getPath() const;

void setPath(const line3df newPath);

bool createBullet(ISceneManager \*manager, IVideoDriver \*video, vector3df position, Direction direction);

// The method checkCollision() returns the node that was collided with bullet or NULL

// 1. If the collision occurred: health = 0 and Returns the node that was collided with.

// 2. Check the bullet for the limit of the installed path (If the bullet is outside the path: health = 0 & return NULL)

ISceneNode \*checkCollision();

// Check the bullet for the limit of the installed path

bool checkEnd() const;

};

#include "Bullet.h"

Bullet::Bullet() : GameObject(), path()

{

}

line3df Bullet::getPath() const

{

return path;

}

void Bullet::setPath(const line3df newPath)

{

path = newPath;

}

bool Bullet::createBullet(ISceneManager \*manager, IVideoDriver \*video, vector3df position, Direction direction)

{

vector3df end = position;

switch (direction)

{

case UP:

end.X = BULLET\_END\_PATH;

break;

case DOWN:

end.X = -BULLET\_END\_PATH;

break;

}

path = line3df(position, end);

name = "models/cube.x";

createNode(manager);

node->setPosition(position);

node->setScale({1.5, 1.5, 1.5});

health = MAX\_HEALTH;

return true;

}

ISceneNode \*Bullet::checkCollision()

{

vector3df pos = node->getPosition();

// Check the bullet for the limit of the installed path(If the bullet is outside the path : health = 0 & return NULL)

if (pos.X > path.start.X && pos.X >= path.end.X)

health = 0;

else if (pos.X < path.start.X && pos.X <= path.end.X)

health = 0;

// If health <= 0

if (isDead())

return NULL;

// Check the collision of the current bullet with the list of collisions (class GameObject

// Special scene node animator for doing automatic collision detection and response.

// list<ISceneNodeAnimatorCollisionResponse \*> collisions;)

for (auto i : collisions)

{

// The method collisionOccurred() returns true if a collision occurred during the last animateNode()

if (i->collisionOccurred())

{

// If true, then return the node with a collision

health = 0;

return (i->getCollisionNode()); // Returns the node that was collided with.

}

}

return NULL;

}

// Check the bullet for the limit of the installed path

bool Bullet::checkEnd() const

{

if (node->getPosition().X > path.start.X && node->getPosition().X >= path.end.X)

return true;

if (node->getPosition().X < path.start.X && node->getPosition().X <= path.end.X)

return true;

return false;

}

EnemyStorage

#include "Global.h"

#include "Enemy.h"

#include "TextureManager.h"

#define ENEMY\_COUNT 8

#define ENEMY\_LINE\_COUNT 5

#define ENEMY\_VERTICAL\_MARGIN 50

#define ENEMY\_HOTIZONTAL\_MARGIN 100

class EnemyStorage

{

private:

list<Enemy> enemyList;

public:

list<Enemy> getEnemyList() const;

void setEnemyList(list<Enemy> newList);

vector3df getRandomEnemyPosition() const;

bool createEnemies(ISceneManager \*manager, TextureManager \*&texManager, rectf limits);

// IMetaTriangleSelector - Interface for making multiple triangle selectors work as one big selector.

/\*\* This is nothing more than a collection of one or more triangle selectors

providing together the interface of one triangle selector. In this way,

collision tests can be done with different triangle soups in one pass.

\*/

// The method creates triangleselector for each enemy from enemyList & add it to the IMetaTriangleSelector.

IMetaTriangleSelector \*createSelectors(ISceneManager \*manager);

// The method checks for enemies damage and death

bool checkNode(ISceneNode \*check, ISceneManager \*manager, u32 \*score, ISoundEngine \*sound, bool soundPlay, vector3df \*position);

void move(f32 delta);

bool step();

void drop();

};

#include "EnemyStorage.h"

list<Enemy> EnemyStorage::getEnemyList() const

{

return enemyList;

}

void EnemyStorage::setEnemyList(list<Enemy> newList)

{

enemyList = newList;

}

vector3df EnemyStorage::getRandomEnemyPosition() const

{

vector3df result = IDLE\_VECTOR;

while (result == vector3df(IDLE\_VECTOR))

{

u32 index = rand() % enemyList.size();

u32 count = 0;

for (auto &i : enemyList)

{

if (count == index && !i.isDead())

{

result = i.getPosition();

return result;

}

count++;

}

}

return result;

}

bool EnemyStorage::createEnemies(ISceneManager \*manager, TextureManager \*&texManager, rectf limits)

{

Direction direction = DOWN;

for (u32 i = 0; i < 10; i++)

{

u32 type = rand() % ENEMY\_TYPE\_COUNT;

if (i % ENEMY\_LINE\_COUNT == 0 && i)

direction = (direction == DOWN ? UP : DOWN);

f32 yStep = i % ENEMY\_LINE\_COUNT;

if (direction == UP)

yStep++;

vector3df position = { limits.UpperLeftCorner.X + yStep \* ENEMY\_HOTIZONTAL\_MARGIN, 0.f, limits.UpperLeftCorner.Y - (i / ENEMY\_LINE\_COUNT) \* ENEMY\_VERTICAL\_MARGIN };

vector3df end = position;

vector3df start = position;

switch (direction)

{

case UP:

end.X += ENEMY\_VERTICAL\_MARGIN;

break;

case DOWN:

start.X -= ENEMY\_VERTICAL\_MARGIN;

break;

}

end.X = limits.LowerRightCorner.Y;

rectf path = { start.Z, start.X, end.Z, end.X };

enemyList.push\_back({ manager, texManager, type, direction, position, path });

}

return true;

}

bool EnemyStorage::checkNode(ISceneNode \*check, ISceneManager \*manager, u32 \*score, ISoundEngine \*sound, bool soundPlay, vector3df \*position)

{

for (auto &i : enemyList)

{

if (check == i.getNode())

{

i.damage(i.getDamage());

(\*score) += i.getDamage();

// проверка на уничтожение

if (i.isDead())

{

if (soundPlay)

{

ISound \*bang = sound->play2D("sounds/crash.wav", false, true);

bang->setVolume(10.0f);

bang->setIsPaused(false);

bang->drop();

}

i.deathAnimation();

(\*score) += i.getDamage() \* 2;

(\*position) = i.getPosition();

// Move the object behind the screen & delete collisions and animators from node i.moveFromField();

break;

}

}

}

for (auto &i : enemyList)

if (!i.isDead())

return false;

return true;

}

IMetaTriangleSelector \*EnemyStorage::createSelectors(ISceneManager \*manager)

{

IMetaTriangleSelector \*result = manager->createMetaTriangleSelector();

for (auto &i : enemyList)

result->addTriangleSelector(i.createTriangleSelector(manager));

return result;

}

void EnemyStorage::move(f32 delta)

{

for (auto &i : enemyList)

if (!i.isDead())

i.move(delta);

}

bool EnemyStorage::step()

{

for (auto &i : enemyList)

{

if (!i.isDead())

{

Direction temp = i.getDirection();

i.setDirection(LEFT);

if (i.move(0))

return true;

i.setDirection(temp);

}

}

return false;

}

void EnemyStorage::drop()

{

for (auto &i : enemyList)

i.drop();

enemyList.clear();

}

Player

#include "IrrLicht.h"

#include "GameObject.h"

#define PLAYER\_CANON\_MARGIN 8

#define PLAYER\_SPEED 200

#define PLAYER\_FIRE\_TIME 400

#define PLAYER\_DAMAGE 10

#define PLAYER\_ROTATION { 0, 270, 180 }

enum playerState

{

IDLE,

ATTACK,

RUNNING

};

class Player : public GameObject

{

private:

playerState pState;

public:

Player();

bool move(f32 delta);

vector3df fire();

void setIdleAnimation();

void setRunningAnimation();

void setAttackAnimation();

};

#include "Player.h"

Player::Player() : GameObject()

{

}

bool Player::move(f32 delta)

{

vector3df pos = node->getPosition();

f32 path = PLAYER\_SPEED \* delta;

pos.X += path;

node->setPosition(pos);

return false;

}

vector3df Player::fire()

{

vector3df pos = node->getPosition();

return pos;

}

void Player::setIdleAnimation()

{

if (pState != playerState::IDLE)

{

node->setMD2Animation(EMAT\_STAND);

node->setAnimationSpeed(30);

pState = playerState::IDLE;

}

}

void Player::setRunningAnimation()

{

if (pState != playerState::RUNNING)

{

node->setMD2Animation(EMAT\_RUN);

node->setAnimationSpeed(30);

pState = playerState::RUNNING;

}

}

void Player::setAttackAnimation()

{

if (pState != playerState::ATTACK)

{

node->setMD2Animation(EMAT\_ATTACK);

node->setAnimationSpeed(20);

pState = playerState::ATTACK;

}

}

GUI

#include "Global.h"

#include "TextureManager.h"

#include <string>

#define START\_BUTTON 101

#define RESUME\_BUTTON 102

#define BACK\_BUTTON 103

#define QUIT\_BUTTON 104

#define OPTIONS\_BUTTON 105

#define MUSIC\_ENABLE 201

#define MUSIC\_DISABLE 202

#define SOUND\_ENABLE 203

#define SOUND\_DISABLE 204

#define UNKNOWN -1

#define BUTTON\_WIDTH 170

#define BUTTON\_HEIGHT 45

#define BUTTON\_MARGIN 30

#define COMBOBOX\_WIDTH 100

#define COMBOBOX\_HEIGHT 20

#define COMBOBOX\_MARGIN 20

#define SCORE\_HEALTH\_WIDTH 60

#define SCORE\_HEALTH\_HEIGHT 10

#define TEXT\_MARGIN 10

#define SCORE\_POSITION\_H 490

#define SCORE\_POSITION\_V 10

#define HEALTH\_POSITION\_H 500

#define HEALTH\_POSITION\_V 30

#define TRIPLE\_BUTTON\_START\_V 299

#define TRIPLE\_BUTTON\_START\_H 437

#define SINGLE\_BUTTON\_START\_H 227

#define WHITE\_COLOR {255, 255, 255, 255}

class GUI

{

private:

list<IGUIButton \*> buttons;

IGUIStaticText \*score;

IGUIStaticText \*health;

IGUIStaticText \*result;

IGUIComboBox \*musicBox;

IGUIComboBox \*soundBox;

IGUIImage \*logo;

IGUIImage \*life;

void addButton(IGUIEnvironment \*gui, ITexture \*&texture, path name, const recti size, u32 btn);

public:

GUI();

void setScore(const u32 score);

void setHealth(const s32 health);

void createLife(IGUIEnvironment \*gui, IVideoDriver \*drv);

bool isButtonPressed(u32 id) const;

s32 getMusicSelection() const;

s32 getSoundSelection() const;

void menu(IGUIEnvironment \*gui, TextureManager \*&manager, IVideoDriver \*drv);

void options(IGUIEnvironment \*gui, TextureManager \*&manager);

void pause(IGUIEnvironment \*gui, TextureManager \*&manager);

void game(IGUIEnvironment \*gui, TextureManager \*&manager, IVideoDriver \*drv);

void win(IGUIEnvironment \*gui, TextureManager \*&manager);

void lose(IGUIEnvironment \*gui, TextureManager \*&manager);

void drop(bool fScore = false, bool fHealth = false);

};

#include "Gui.h"

GUI::GUI() : buttons({}), score(NULL), health(NULL), result(NULL), musicBox(NULL), soundBox(NULL), life(NULL), logo(NULL)

{

}

void GUI::addButton(IGUIEnvironment \*gui, ITexture \*&texture, path name, const recti size, u32 id)

{

size\_t length = strlen(name.c\_str()) + 1;

wchar\_t \*text = new wchar\_t[length];

std::mbstowcs(text, name.c\_str(), length);

IGUIButton \*button = gui->addButton(size, NULL, id, text);

button->setImage(texture);

buttons.push\_back(button);

}

void GUI::setScore(const u32 newScore)

{

std::wstring text = L"Score ";

text += std::to\_wstring(newScore);

score->setText(text.c\_str());

}

void GUI::createLife(IGUIEnvironment \*gui, IVideoDriver \*drv) {

life = gui->addImage(drv->getTexture("media/heart.png"), core::position2d<int>(10, 10));

}

void GUI::setHealth(const s32 newHealth)

{

std::wstring text = std::to\_wstring(newHealth);

health->setText(text.c\_str());

}

bool GUI::isButtonPressed(u32 id) const

{

for (auto &i : buttons)

if (i && i->getID() == id && i->isPressed())

return true;

return false;

}

s32 GUI::getMusicSelection() const

{

if (musicBox)

return musicBox->getItemData(musicBox->getSelected());

return UNKNOWN;

}

s32 GUI::getSoundSelection() const

{

if (soundBox)

return soundBox->getItemData(soundBox->getSelected());

return UNKNOWN;

}

void GUI::menu(IGUIEnvironment \*gui, TextureManager \*&manager, IVideoDriver \*drv)

{

drop();

logo = gui->addImage(drv->getTexture("textures/poster.png"), core::position2d<int>(10, -65));

addButton(gui, manager->getButtonsTexture()[0], "Start Game",

{

SINGLE\_BUTTON\_START\_H,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2 + BUTTON\_MARGIN,

SINGLE\_BUTTON\_START\_H + BUTTON\_WIDTH,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2 + BUTTON\_MARGIN + BUTTON\_HEIGHT

}, START\_BUTTON);

addButton(gui, manager->getButtonsTexture()[1], "Options",

{

SINGLE\_BUTTON\_START\_H + BUTTON\_WIDTH + BUTTON\_MARGIN,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2 + BUTTON\_MARGIN,

SINGLE\_BUTTON\_START\_H + BUTTON\_WIDTH \* 2 + BUTTON\_MARGIN,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2 + BUTTON\_MARGIN + BUTTON\_HEIGHT

}, OPTIONS\_BUTTON);

addButton(gui, manager->getButtonsTexture()[3], "Quit",

{

SINGLE\_BUTTON\_START\_H + BUTTON\_WIDTH \* 2 + BUTTON\_MARGIN \* 2,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2 + BUTTON\_MARGIN,

SINGLE\_BUTTON\_START\_H + BUTTON\_WIDTH \* 3 + BUTTON\_MARGIN \* 2,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2 + BUTTON\_MARGIN + BUTTON\_HEIGHT

}, QUIT\_BUTTON);

}

void GUI::options(IGUIEnvironment \*gui, TextureManager \*&manager)

{

drop();

musicBox = gui->addComboBox(

{

WINDOW\_WIDTH / 2 - COMBOBOX\_WIDTH / 2,

WINDOW\_HEIGHT / 2 - COMBOBOX\_HEIGHT \* 2 - COMBOBOX\_MARGIN,

WINDOW\_WIDTH / 2 + COMBOBOX\_WIDTH / 2,

WINDOW\_HEIGHT / 2 - COMBOBOX\_HEIGHT - COMBOBOX\_MARGIN,

});

soundBox = gui->addComboBox(

{

WINDOW\_WIDTH / 2 - COMBOBOX\_WIDTH / 2,

WINDOW\_HEIGHT / 2 - COMBOBOX\_HEIGHT / 2,

WINDOW\_WIDTH / 2 + COMBOBOX\_WIDTH / 2,

WINDOW\_HEIGHT / 2 + COMBOBOX\_HEIGHT / 2 });

musicBox->addItem(L"Enable music", MUSIC\_ENABLE);

musicBox->addItem(L"Disable Music", MUSIC\_DISABLE);

soundBox->addItem(L"Enable sound", SOUND\_ENABLE);

soundBox->addItem(L"Disable sound", SOUND\_DISABLE);

addButton(gui, manager->getButtonsTexture()[2], "Back",

{

TRIPLE\_BUTTON\_START\_H,

WINDOW\_HEIGHT / 2 + COMBOBOX\_HEIGHT / 2 + BUTTON\_MARGIN,

TRIPLE\_BUTTON\_START\_H + BUTTON\_WIDTH,

WINDOW\_HEIGHT / 2 + COMBOBOX\_HEIGHT / 2 + BUTTON\_MARGIN + BUTTON\_HEIGHT

}, BACK\_BUTTON);

}

void GUI::pause(IGUIEnvironment \*gui, TextureManager \*&manager)

{

drop(true, true);

addButton(gui, manager->getButtonsTexture()[0], "Resume",

{

TRIPLE\_BUTTON\_START\_H,

TRIPLE\_BUTTON\_START\_V,

TRIPLE\_BUTTON\_START\_H + BUTTON\_WIDTH,

TRIPLE\_BUTTON\_START\_V + BUTTON\_HEIGHT

}, RESUME\_BUTTON);

addButton(gui, manager->getButtonsTexture()[2], "Back",

{

TRIPLE\_BUTTON\_START\_H,

TRIPLE\_BUTTON\_START\_V + BUTTON\_HEIGHT + BUTTON\_MARGIN,

TRIPLE\_BUTTON\_START\_H + BUTTON\_WIDTH,

TRIPLE\_BUTTON\_START\_V + BUTTON\_HEIGHT \* 2 + BUTTON\_MARGIN

}, BACK\_BUTTON);

addButton(gui, manager->getButtonsTexture()[3], "Quit",

{

TRIPLE\_BUTTON\_START\_H,

TRIPLE\_BUTTON\_START\_V + BUTTON\_HEIGHT \* 2 + BUTTON\_MARGIN \* 2,

TRIPLE\_BUTTON\_START\_H + BUTTON\_WIDTH,

TRIPLE\_BUTTON\_START\_V + BUTTON\_HEIGHT \* 3 + BUTTON\_MARGIN \* 2

}, QUIT\_BUTTON);

}

void GUI::game(IGUIEnvironment \*gui, TextureManager \*&manager, IVideoDriver \*drv)

{

drop(false, false);

life = gui->addImage(drv->getTexture("textures/heart.png"), core::position2d<int>(HEALTH\_POSITION\_H - 9, HEALTH\_POSITION\_V-10));

score = gui->addStaticText(L"Score was 0 but it shows nothing", { SCORE\_POSITION\_H, SCORE\_POSITION\_V, SCORE\_POSITION\_H + SCORE\_HEALTH\_WIDTH, SCORE\_POSITION\_V + SCORE\_HEALTH\_WIDTH });

score->setOverrideColor(WHITE\_COLOR);

score->enableOverrideColor(true);

health = gui->addStaticText(L"100", { HEALTH\_POSITION\_H, HEALTH\_POSITION\_V, HEALTH\_POSITION\_H + SCORE\_HEALTH\_WIDTH, HEALTH\_POSITION\_V + SCORE\_HEALTH\_HEIGHT });

health->setOverrideColor(WHITE\_COLOR);

health->enableOverrideColor(true);

}

void GUI::win(IGUIEnvironment \*gui, TextureManager \*&manager)

{

drop(true);

result = gui->addStaticText(L"Good Work",

{

WINDOW\_WIDTH / 2 - SCORE\_HEALTH\_WIDTH / 2,

WINDOW\_HEIGHT / 2 - SCORE\_HEALTH\_HEIGHT / 2 - TEXT\_MARGIN - SCORE\_HEALTH\_HEIGHT,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2,

WINDOW\_HEIGHT / 2 - SCORE\_HEALTH\_HEIGHT / 2 - TEXT\_MARGIN

});

result->setOverrideColor(WHITE\_COLOR);

result->enableOverrideColor(true);

score->setRelativePosition(

{

WINDOW\_WIDTH / 2 - SCORE\_HEALTH\_WIDTH / 2,

WINDOW\_HEIGHT / 2 - SCORE\_HEALTH\_HEIGHT / 2,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2,

WINDOW\_HEIGHT / 2 + SCORE\_HEALTH\_HEIGHT / 2

});

addButton(gui, manager->getButtonsTexture()[0], "Start Game",

{

SINGLE\_BUTTON\_START\_H,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2 + BUTTON\_MARGIN,

SINGLE\_BUTTON\_START\_H + BUTTON\_WIDTH,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2 + BUTTON\_MARGIN + BUTTON\_HEIGHT

}, START\_BUTTON);

addButton(gui, manager->getButtonsTexture()[2], "Back",

{

SINGLE\_BUTTON\_START\_H + BUTTON\_WIDTH + BUTTON\_MARGIN,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2 + BUTTON\_MARGIN,

SINGLE\_BUTTON\_START\_H + BUTTON\_WIDTH \* 2 + BUTTON\_MARGIN,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2 + BUTTON\_MARGIN + BUTTON\_HEIGHT

}, BACK\_BUTTON);

addButton(gui, manager->getButtonsTexture()[3], "Quit",

{

SINGLE\_BUTTON\_START\_H + BUTTON\_WIDTH \* 2 + BUTTON\_MARGIN \* 2,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2 + BUTTON\_MARGIN,

SINGLE\_BUTTON\_START\_H + BUTTON\_WIDTH \* 3 + BUTTON\_MARGIN \* 2,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2 + BUTTON\_MARGIN + BUTTON\_HEIGHT

}, QUIT\_BUTTON);

}

void GUI::lose(IGUIEnvironment \*gui, TextureManager \*&manager)

{

drop(true);

result = gui->addStaticText(L"Mission failed",

{

WINDOW\_WIDTH / 2 - SCORE\_HEALTH\_WIDTH / 2,

WINDOW\_HEIGHT / 2 - SCORE\_HEALTH\_HEIGHT / 2 - TEXT\_MARGIN - SCORE\_HEALTH\_HEIGHT,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2,

WINDOW\_HEIGHT / 2 - SCORE\_HEALTH\_HEIGHT / 2 - TEXT\_MARGIN

});

result->setOverrideColor(WHITE\_COLOR);

result->enableOverrideColor(true);

score->setRelativePosition(

{

WINDOW\_WIDTH / 2 - SCORE\_HEALTH\_WIDTH / 2,

WINDOW\_HEIGHT / 2 - SCORE\_HEALTH\_HEIGHT / 2,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2,

WINDOW\_HEIGHT / 2 + SCORE\_HEALTH\_HEIGHT / 2

});

addButton(gui, manager->getButtonsTexture()[0], "Start Game",

{

SINGLE\_BUTTON\_START\_H,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2 + BUTTON\_MARGIN,

SINGLE\_BUTTON\_START\_H + BUTTON\_WIDTH,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2 + BUTTON\_MARGIN + BUTTON\_HEIGHT

}, START\_BUTTON);

addButton(gui, manager->getButtonsTexture()[2], "Back",

{

SINGLE\_BUTTON\_START\_H + BUTTON\_WIDTH + BUTTON\_MARGIN,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2 + BUTTON\_MARGIN,

SINGLE\_BUTTON\_START\_H + BUTTON\_WIDTH \* 2 + BUTTON\_MARGIN,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2 + BUTTON\_MARGIN + BUTTON\_HEIGHT

}, BACK\_BUTTON);

addButton(gui, manager->getButtonsTexture()[3], "Quit",

{

SINGLE\_BUTTON\_START\_H + BUTTON\_WIDTH \* 2 + BUTTON\_MARGIN \* 2,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2 + BUTTON\_MARGIN,

SINGLE\_BUTTON\_START\_H + BUTTON\_WIDTH \* 3 + BUTTON\_MARGIN \* 2,

WINDOW\_WIDTH / 2 + SCORE\_HEALTH\_WIDTH / 2 + BUTTON\_MARGIN + BUTTON\_HEIGHT

}, QUIT\_BUTTON);

}

void GUI::drop(bool fScore, bool fHealth)

{

for (auto &i : buttons)

{

if (i)

{

i->remove();

i = NULL;

}

}

buttons.clear();

if (score && !fScore)

{

score->remove();

score = NULL;

}

if (health && !fHealth)

{

health->remove();

health = NULL;

}

if (life && !fHealth)

{

life->remove();

life = NULL;

}

if (logo)

{

logo->remove();

logo = NULL;

}

if (result)

{

result->remove();

result = NULL;

}

if (musicBox)

{

musicBox->remove();

musicBox = NULL;

}

if (soundBox)

{

soundBox->remove();

soundBox = NULL;

}

}

TextureManager

#include "Global.h"

#define TEXTURE\_PATH "textures/"

#define PLAYER\_NAME "sydney.bmp"

#define ENEMY01\_NAME "faerie2.bmp"

#define ENEMY02\_NAME "Faerie5.bmp"

#define ENEMY03\_NAME "nskinbl.jpg"

#define ENEMY04\_NAME "nskinrd.jpg"

#define ENEMY05\_NAME "lostsoul.bmp"

#define BUTTON\_START\_NAME "Start1.png"

#define BUTTON\_OPTIONS\_NAME "Start1.png"

#define BUTTON\_BACK\_NAME "Start1.png"

#define BUTTON\_QUIT\_NAME "Start1.png"

#define ENEMY\_TEXTURES\_COUNT 5

#define BUTTON\_TEXTURES\_COUNT 4

class TextureManager

{

private:

ITexture \*\*playerTex;

ITexture \*\*enemyTex;

ITexture \*\*buttonTex;

ITexture \*\*buttonsTex;

public:

TextureManager();

ITexture \*\*getPlayerTexture() const;

ITexture \*\*getEnemyTexture() const;

ITexture \*\*getButtonsTexture() const;

void setPlayerTexture(ITexture \*\*texture);

void setEnemyTexture(ITexture \*\*texture);

void setButtonsTexture(ITexture \*\*texture);

void loadTextures(IVideoDriver \*video);

void drop();

};

#include "TextureManager.h"

TextureManager::TextureManager() : playerTex(NULL), enemyTex(NULL), buttonsTex(NULL)

{

}

ITexture \*\*TextureManager::getPlayerTexture() const

{

return playerTex;

}

ITexture \*\*TextureManager::getEnemyTexture() const

{

return enemyTex;

}

ITexture \*\*TextureManager::getButtonsTexture() const

{

return buttonsTex;

}

void TextureManager::setPlayerTexture(ITexture \*\*texture)

{

playerTex = texture;

}

void TextureManager::setEnemyTexture(ITexture \*\*texture)

{

enemyTex = texture;

}

void TextureManager::setButtonsTexture(ITexture \*\*texture)

{

buttonsTex = texture;

}

void TextureManager::loadTextures(IVideoDriver \*video)

{

path name = TEXTURE\_PATH;

playerTex = new ITexture \*;

playerTex[0] = video->getTexture(name + PLAYER\_NAME);

enemyTex = new ITexture \*[ENEMY\_TEXTURES\_COUNT];

enemyTex[0] = video->getTexture(name + ENEMY01\_NAME);

enemyTex[1] = video->getTexture(name + ENEMY02\_NAME);

enemyTex[2] = video->getTexture(name + ENEMY03\_NAME);

enemyTex[3] = video->getTexture(name + ENEMY04\_NAME);

enemyTex[4] = video->getTexture(name + ENEMY05\_NAME);

buttonsTex = new ITexture \*[BUTTON\_TEXTURES\_COUNT];

buttonsTex[0] = video->getTexture(name + BUTTON\_START\_NAME);

buttonsTex[1] = video->getTexture(name + BUTTON\_OPTIONS\_NAME);

buttonsTex[2] = video->getTexture(name + BUTTON\_BACK\_NAME);

buttonsTex[3] = video->getTexture(name + BUTTON\_QUIT\_NAME);

}

void TextureManager::drop()

{

if (playerTex)

{

if (playerTex[0])

playerTex[0] = NULL;

delete playerTex;

}

if (enemyTex)

{

for (u32 i = 0; i < ENEMY\_TEXTURES\_COUNT; i++)

{

if (enemyTex[i])

enemyTex[i] = NULL;

}

delete enemyTex;

}

if (buttonsTex)

{

for (u32 i = 0; i < BUTTON\_TEXTURES\_COUNT; i++)

{

if (buttonsTex[i])

buttonsTex[i] = NULL;

}

delete buttonsTex;

}

}

Game

#include "IrrLicht.h"

#include "Irrklang.h"

#include "Player.h"

#include "EnemyStorage.h"

#include "BulletStorage.h"

#include "Timer.h"

#include "Gui.h"

#include "Effect.h"

#include "TextureManager.h"

#include <time.h>

enum State

{

MENU,

OPTIONS,

INGAME,

PAUSE,

OVER,

EXIT

};

#define ENEMY\_FIELD\_SIZE { -250.f, 180.f, 250.f, -160.f }

#define PLAYER\_LINE\_SIZE { -200, 0.f, -500, 200, 0.f, 0 }

#define SCENE\_SKYBOX\_ROTATION { 0.f, 270.f, 0.f }

#define SCENE\_SKYBOX\_SCALE { 1.1f, 1.f, 0.75f }

class Game

{

private:

Irrlicht \*engine;

ISoundEngine \*sound;

Player \*player;

EnemyStorage \*enemies;

BulletStorage \*bullets;

GUI \*gui;

Effect \*effects;

ISound \*gameMusic;

TextureManager \*texManager;

rectf enemyField;

// 3D line between two points with intersection methods.

line3df playerLine;

State state;

Time timer;

u32 score;

bool soundPlay;

void toLevel(bool pause);

void menu();

void game();

bool addPlayer();

bool addEnemies();

void applyEnemiesMoves();

void applyPlayerMoves();

void applyBulletsMoves();

void applyEffects();

void drop();

public:

Game();

void loop();

};

#include "Game.h"

Game::Game() :

enemyField(ENEMY\_FIELD\_SIZE),

playerLine(line3df(PLAYER\_LINE\_SIZE)),

enemies(new EnemyStorage()),

player(new Player()),

bullets(new BulletStorage()),

gui(new GUI()),

effects(new Effect()),

texManager(new TextureManager()),

engine(new Irrlicht(L"Monster Attack", dimension2du(WINDOW\_WIDTH, WINDOW\_HEIGHT))),

state(MENU),

sound(createIrrKlangDevice()),

score(0),

soundPlay(true)

{

srand(time(NULL));

}

void Game::toLevel(bool pause)

{

engine->hideCursor(true);

engine->setCursorToCenter();

engine->resetInput();

if (!pause)

{

drop();

if (!addPlayer())

exit(1);

if (!addEnemies())

exit(1);

timer.zeroTimes();

engine->setTime(0);

}

gui->game(engine->getGUI(), texManager, engine->getVideo());

}

void Game::menu()

{

switch (state)

{

case MENU:

if (gui->isButtonPressed(START\_BUTTON))

{

state = INGAME;

toLevel(false);

}

else if (gui->isButtonPressed(OPTIONS\_BUTTON))

{

state = OPTIONS;

gui->options(engine->getGUI(), texManager);

}

else if (gui->isButtonPressed(QUIT\_BUTTON))

state = EXIT;

break;

case OPTIONS:

if (gui->isButtonPressed(BACK\_BUTTON))

{

state = MENU;

gui->menu(engine->getGUI(), texManager, engine->getVideo());

}

else if (gui->getMusicSelection() == MUSIC\_ENABLE)

{

if (gameMusic->getIsPaused())

gameMusic = sound->play2D("sounds/soundtrack.mp3", true, false, true);

}

else if (gui->getMusicSelection() == MUSIC\_DISABLE)

gameMusic->setIsPaused(true);

if (gui->getSoundSelection() == SOUND\_ENABLE)

soundPlay = true;

else if (gui->getSoundSelection() == SOUND\_DISABLE)

soundPlay = false;

break;

case PAUSE:

if (gui->isButtonPressed(BACK\_BUTTON))

{

score = 0;

state = MENU;

drop();

engine->hideCursor(false);

gui->menu(engine->getGUI(), texManager, engine->getVideo());

}

else if (gui->isButtonPressed(RESUME\_BUTTON))

{

state = INGAME;

u32 delta = engine->getNow() - timer.pauseTime;

timer.addDelta(delta);

toLevel(true);

}

else if (gui->isButtonPressed(QUIT\_BUTTON))

state = EXIT;

break;

case OVER:

score = 0;

if (gui->isButtonPressed(START\_BUTTON))

{

state = INGAME;

toLevel(false);

}

else if (gui->isButtonPressed(BACK\_BUTTON))

{

state = MENU;

drop();

engine->hideCursor(false);

gui->menu(engine->getGUI(), texManager, engine->getVideo());

}

else if (gui->isButtonPressed(QUIT\_BUTTON))

state = EXIT;

break;

}

}

void Game::game()

{

// Counting time since last frame

timer.now = engine->getNow();

timer.delta = (f32)(timer.now - timer.lastTime) / 1000.f;

timer.lastTime = timer.now;

// Apply game objects moves

applyPlayerMoves();

applyEnemiesMoves();

applyBulletsMoves();

applyEffects();

if (engine->isKeyPressed(KEY\_ESCAPE))

{

state = PAUSE;

timer.pauseTime = engine->getNow();

gui->pause(engine->getGUI(), texManager);

engine->hideCursor(false);

}

}

void Game::drop()

{

player->drop();

enemies->drop();

bullets->drop();

effects->drop();

}

void Game::loop()

{

// Creating background

ISceneNode \*skybox = engine->getSkyBox("textures/background1.jpg");

skybox->setRotation(SCENE\_SKYBOX\_ROTATION);

skybox->setScale(SCENE\_SKYBOX\_SCALE);

// Textures

texManager->loadTextures(engine->getVideo());

// Light

ILightSceneNode \*light = engine->getManager()->addLightSceneNode(0, LIGHT\_POSITION, WHITE\_COLOR, LIGHT\_RADIUS);

// Menu

gui->menu(engine->getGUI(), texManager, engine->getVideo());

// Music

gameMusic = sound->play2D("sounds/soundtrack.mp3", true, false, true);

bool exitFlag = false;

while (engine->run())

{

switch (state)

{

case MENU:

case OPTIONS:

case PAUSE:

case OVER:

menu();

break;

case INGAME:

game();

break;

case EXIT:

exitFlag = true;

break;

}

if (exitFlag)

break;

engine->drawAll();

}

texManager->drop();

engine->drop();

sound->drop();

}

bool Game::addPlayer()

{

player->setName("models/sydney.md2");

engine->resetMouseMove();

bool result = player->createNode(engine->getManager());

if (result)

{

player->setPosition(playerLine.getMiddle());

player->setRotation(PLAYER\_ROTATION);

player->setHealth(MAX\_HEALTH);

}

player->getNode()->setMaterialTexture(0, texManager->getPlayerTexture()[0]);

player->setIdleAnimation();

return result;

}

void Game::applyPlayerMoves()

{

vector3df initPos = player->getPosition();

// Keyboard moves

if (engine->isKeyPressed(KEY\_KEY\_S)) {

player->move(-timer.delta);

player->setRunningAnimation();

engine->resetMouseMove();

}

else if (engine->isKeyPressed(KEY\_KEY\_W)) {

player->move(timer.delta);

player->setRunningAnimation();

engine->resetMouseMove();

}

else if( (!engine->isKeyPressed(KEY\_KEY\_S) || !engine->isKeyPressed(KEY\_KEY\_W)) && !engine->isKeyPressed(KEY\_SPACE))

player->setIdleAnimation();

// Mouse moves

dimension2du center = engine->getCenter();

vector2di mouse = engine->getCursorPosition();

if (center.Width != mouse.X)

{

f32 move = (f32)(mouse.X - (s32)center.Width);

player->move(timer.delta \* move);

engine->setCursorPosition(center);

}

// checking player move border

vector3df pos = player->getPosition();

if (pos != initPos)

{

// Checking for bound.

if (pos.X < playerLine.start.X)

pos.X = playerLine.start.X;

else if (pos.X > playerLine.end.X)

pos.X = playerLine.end.X;

player->setPosition(pos);

}

// Fire

if ((engine->isKeyPressed(KEY\_SPACE) || engine->isLeftMouseButtonPressed()) && (timer.now - timer.lastFire) >= PLAYER\_FIRE\_TIME)

{

player->setAttackAnimation();

ISound \*bang = sound->play2D("sounds/lazer.wav", false, true);

bang->setVolume(0.5f);

bang->setIsPaused(false);

bang->drop();

vector3df rPos = player->fire();

bullets->addPlayerBullet(engine->getManager(), engine->getVideo(), texManager, enemies->createSelectors(engine->getManager()), rPos);

timer.lastFire = timer.now;

}

}

bool Game::addEnemies()

{

return enemies->createEnemies(engine->getManager(), texManager, enemyField);

}

void Game::applyEnemiesMoves()

{

// Move up/down

enemies->move(timer.delta);

// Check left movement

if ((timer.now - timer.lastStep) >= ENEMY\_STEP\_TIME)

{

// If the enemies were on the left side of the screen

if (enemies->step())

{

state = OVER;

engine->hideCursor(false);

gui->lose(engine->getGUI(), texManager);

player->setHealth(0);

}

timer.lastStep = timer.now;

}

// Random enemy fires

if ((timer.now - timer.lastEnemyFire) >= ENEMY\_FIRE\_TIME)

{

ISound \*bang = sound->play2D("sounds/lazer.wav", false, true);

bang->setVolume(0.5f);

bang->setIsPaused(false);

bang->drop();

vector3df rPos = enemies->getRandomEnemyPosition();

// function createTriangleSelector from class GameObject

bullets->addEnemyBullet(engine->getManager(), engine->getVideo(), texManager, player->createTriangleSelector(engine->getManager()), rPos);

timer.lastEnemyFire = timer.now;

}

}

void Game::applyBulletsMoves()

{

// Move thier ways

bullets->move(timer.delta);

// Getting all nodes bullets hited & indecies for bullets

array<u32> \*Pindecies, \*Eindecies;

Pindecies = new array<u32>();

Eindecies = new array<u32>();

array<ISceneNode \*> nodes = bullets->checkBullets(Pindecies, Eindecies);

for (u32 i = 0; i < nodes.size(); i++)

{

vector3df pos = IDLE\_VECTOR;

// Checking for player damage and death

if (nodes[i] == player->getNode())

{

player->damage(PLAYER\_DAMAGE);

if (player->isDead())

{

if (soundPlay)

{

ISound \*bang = sound->play2D("sounds/bang.wav", false, true);

bang->setVolume(0.1f);

bang->setIsPaused(false);

bang->drop();

}

state = OVER;

engine->hideCursor(false);

drop();

gui->lose(engine->getGUI(), texManager);

gui->setScore(score);

return;

}

}

// Checking for enemies damage and death

else if (enemies->checkNode(nodes[i], engine->getManager(), &score, sound, soundPlay, &pos))

{

state = OVER;

engine->hideCursor(false);

drop();

gui->win(engine->getGUI(), texManager);

return;

}

if (pos != vector3df(IDLE\_VECTOR))

effects->addDeath(engine->getManager(), pos, engine->getVideo());

}

gui->setScore(score);

if (!player->isDead())

gui->setHealth(player->getHealth());

}

void Game::applyEffects()

{

if (timer.now - timer.lastFrame >= EFFECT\_DEATH\_TIME)

{

effects->drop();

timer.lastFrame = timer.now;

}

}