```cpp
#include "AirStrike.h"

AirStrike::AirStrike(int id, b2World& world, float posX, float posY) :
//Weapon(w_air_strike),
world(world) {
    this->id = id;
    this->deploy_x = posX;
    this->deploy_y = posY;
    createMissils();
}

AirStrike::~AirStrike() {
    // for(std::vector<Missil*>::iterator it = this->missils.begin(); it != this
->missils.end(); ++it) {
    //     delete (*it);
    // }
}

void AirStrike::createMissils() {
    for (int i = 1; i <= gConfiguration.AIR_STRIKE_MISSIL_QUANTITY; ++i) {
        this->missils.push_back(new Missil(this->id+i,
        this->world,
        this->deploy_x + (i),
        this->deploy_y,
        w_air_strike));
    }
}

std::vector<Missil*> AirStrike::getMissils() {
    return this->missils;
}
```



```cpp
#ifndef AIR_STRIKE_H
#define AIR_STRIKE_H

//#include "Weapon.h"
#include "Box2D.h"
#include "Configuration.h"
#include "types.h"
#include "Missil.h"

class AirStrike {
private:
    b2World& world;
    std::vector<Missil*> missils;
    float deploy_x;
    float deploy_y;
    int id;
    void createMissils();
public:
    AirStrike(int id, b2World& world, float posX, float posY);
    ~AirStrike();
    std::vector<Missil*> getMissils();
};

#endif
```

jun 26, 18 11:13 **Bat.cpp** Page 1/1

```
#include "Bat.h"
#include "types.h"

Bat::Bat(b2World& world, float posX, float posY, bool mirrored, float angle) :
world(world) {
    this->posX = posX;
    this->posY = posY;
    this->mirrored = mirrored;
    this->angle = angle;
    this->rayLength = gConfiguration.BAT_LENGTH;
    rayCast();
}

void Bat::rayCast() {
    float c_angle;
    if (mirrored) {
        c_angle = 90 - angle;
    } else c_angle = 270 + angle;

    c_angle = c_angle * gConfiguration.DEGTORAD;
    b2Vec2 rayDir(sinf(c_angle), cosf(c_angle));
    b2Vec2 center(posX, posY);
    rayDir.y = -rayDir.y;
    b2Vec2 p2 = center + rayLength * rayDir;
    RayCastClosestCallBack callback;

    this->world.RayCast(&callback, center, p2);
    if (callback.body) {
        entity_t entity_type = static_cast<Entity*>(callback.body->GetUserData()
)->getEntityType();
        if (entity_type == WORM) {
            Worm * worm = static_cast<Worm*>(callback.body->GetUserData());

            atack(worm, callback.body);
        }
    }

}

void Bat::atack(Worm* worm, b2Body* body) {
    float impulse = body->GetMass() * gConfiguration.BAT_IMPULSE;
    float x_impulse = impulse * cosf(angle * gConfiguration.DEGTORAD);
    float y_impulse = impulse * sinf(angle * gConfiguration.DEGTORAD);
    if (!mirrored) x_impulse = x_impulse * -1;
    body->ApplyLinearImpulse(b2Vec2(x_impulse, -y_impulse), body->GetWorldCenter
(),true);
    worm->hurt(gConfiguration.BAT_DAMAGE);
    worm->setAffectedByExplosion();
}
```

jun 25, 18 20:09 **Bat.h** Page 1/1

```
#ifndef BAT_H
#define BAT_H

#include "Weapon.h"
#include "Box2D.h"
#include "Worm.h"
#include "types.h"
#include "Configuration.h"
#include "PhysicEntity.h"
#include "RayCastClosestCallBack.h"
#include "Entity.h"

class Bat {
private:
    b2World& world;
    float posX;
    float posY;
    bool mirrored;
    float angle;
    int rayLength;
public:
    Bat(b2World& world, float posX, float posY, bool mirrored, float angle);
    void rayCast(void);
    void atack(Worm* worm, b2Body* body);
};

#endif
```

```
#include "Bazooka.h"
#include "types.h"
#include <iostream>

Bazooka::Bazooka(int id, b2World& world, float posX, float posY, float mirrored,

float shooting_angle, int power_factor, weapon_t type) :
Weapon(type),
world(world) {
    b2BodyDef bazookaDef;
    bazookaDef.type = b2_dynamicBody;
    bazookaDef.position.Set(posX, posY);
    b2Body* body = world.CreateBody(&bazookaDef);
    body->SetUserData(this);

    b2PolygonShape bazookaShape;
    bazookaShape.SetAsBox(BAZOOKA_WIDTH/2, BAZOKOA_HEIGHT/2);

    b2FixtureDef bazookaFixture;
    bazookaFixture.shape = &bazookaShape;
    bazookaFixture.density = 1;
    bazookaFixture.friction = 0.3;
    body->CreateFixture(&bazookaFixture);
    this->body = body;

    this->wind_affected = true;
    this->exploded = false;
    this->power_factor = power_factor;
    this->mirrored = mirrored;
    this->shooting_angle = shooting_angle;
    this->id = id;
    this->blast_power = gConfiguration.BAZOOKA_BLAST_POWER;
    this->blast_radius = gConfiguration.BAZOOKA_BLAST_RADIUS;
    shoot();
}

Bazooka::~Bazooka() {
    this->world.DestroyBody(this->body);
}

void Bazooka::explode() {
    ExplosionManager explosionManager(this->world);
    b2Vec2 center = this->body->GetPosition();
    explosionManager.manageExplosion(center, blast_radius, blast_power);
    this->exploded = true;
}

float Bazooka::getPosX() {
    return this->body->GetPosition().x;
}

float Bazooka::getPosY() {
    return this->body->GetPosition().y;
}

void Bazooka::shoot() {
    float impulse = this->body->GetMass() * power_factor;
    float x_impulse = impulse * cosf(shooting_angle * gConfiguration.DEGTORAD);
    float y_impulse = impulse * sinf(shooting_angle * gConfiguration.DEGTORAD);
    if (!mirrored) x_impulse = x_impulse * -1;
    this->body->ApplyLinearImpulse(b2Vec2(x_impulse, -y_impulse), this->body->Ge
tWorldCenter(), true);
```



```
}

void Bazooka::update(int currenTime, int wind_force) {
    if (getPosY() > gConfiguration.WORLD_Y_LIMIT /*|| contact*/) {
        this->explode();
    }
    if (wind_affected) {
        this->body->ApplyForce(body->GetMass() * b2Vec2(wind_force,0), body->Get
WorldCenter(), true);
    }
    b2Vec2 mov_speed = this->body->GetLinearVelocity();

    if (round(mov_speed.x) == 0) {
        if (mov_speed.y > 0) {
            this->direction_angle = 180;
            return;
        }

        if (mov_speed.y < 0) {
            this->direction_angle = 0;
            return;
        }
    }

    if (round(mov_speed.y) == 0) {
        if (mov_speed.x > 0) {
            this->direction_angle = 90;
            return;
        }

        if (mov_speed.x < 0) {
            this->direction_angle = 270;
            return;
        }
    }

    int ang = atan(mov_speed.x/mov_speed.y) * gConfiguration.RADTODEG;

    // Primer cuadrante
    if (mov_speed.y < 0 && mov_speed.x > 0) {
        this->direction_angle = -ang;
    }

    // Segundo cuadrante
    if (mov_speed.y < 0 && mov_speed.x < 0) {
        this->direction_angle = 360 - ang;
    }

    // Tercer cuadrante
    if (mov_speed.y > 0 && mov_speed.x < 0) {
        this->direction_angle = 180 - ang;
    }

    // Cuarto cuadrante
    if (mov_speed.y > 0 && mov_speed.x > 0) {
        this->direction_angle = 180 - ang;
    }
}

bool Bazooka::isMoving() {
    b2Vec2 speed = this->body->GetLinearVelocity();
    if (!speed.x && !speed.y) return false;
```

```
    return true;
}
```



```
#ifndef BAZOOKA_H
#define BAZOOKA_H

#include "Weapon.h"
#include "Box2D.h"
#include "types.h"
#include "Configuration.h"
#include "ExplosionManager.h"

#define BAZOOKA_WIDTH 0.7f
#define BAZOKOA_HEIGHT 0.2f

class Bazooka : public Weapon {
protected:
    b2World& world;
    b2Body* body;
    int shooting_angle;
    int power_factor;
    bool mirrored;
    int blast_power;
public:
    Bazooka(int id, b2World& world, float posX, float posY, float mirrored, floa
t shooting_angle, int power_factor, weapon_t type);
    ~Bazooka();
    void explode(void);
    float getPosX();
    float getPosY();
    void shoot(void);
    bool isMoving(void);
    entity_t getEntityType() {return BAZOOKA;}
    void update(int currentTime, int wind_force);
};

#endif
```

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include "client.h"
#include "protocol.h"
#include "event.h"
#include "yaml.h"

Client::Client(Protocol prt, std::string & pn) :
protocol(std::move(prt)){
    this->player_name = pn;
    this->status = lobby;
    this->in_match_id = 0;
    this->defeated = false;
    this->exited = false;
}

void Client::sendGamesStatus(YAML::Node gameStatusNode) {
    protocol.sendGameStatus(gameStatusNode);
}

Event Client::rcvEvent(void) {
    return this->protocol.rcvEvent();
}

std::string Client::getPlayerName(void) {
    return this->player_name;
}

void Client::setStatus(client_status_t new_status) {
    this->status = new_status;
}

client_status_t Client::getStatus(void) {
    return this->status;
}

void Client::sendResponse(int code, std::string & msg) {
    YAML::Node response;
    response["code"] = code;
    response["msg"] = msg;
    this->protocol.sendMsg(response);
}

void Client::setJoinedMatchGameCreator(std::string & jmn) {
    this->joined_match_creator_name = jmn;
}

void Client::clearJoinedMatchGameCreator(void) {
    this->joined_match_creator_name.clear();
}

std::string Client::getJoinedMatchCreatorName(void) {
    return this->joined_match_creator_name;
}

void Client::sendWaitingPlayers(std::vector<std::string> players) {
    YAML::Node response;
    response["waiting_players"];
    response["waiting_players"] = players;
    this->protocol.sendMsg(response);
```



```cpp
}

void Client::sendGameStart(int code, std::string & msg, std::string & team_id) {
    YAML::Node response;
    response["code"] = code;
    response["msg"] = msg;
    response["team_id"] = team_id;
    this->protocol.sendMsg(response);
}

void Client::sendMapGame(std::fstream & map_file) {
    this->protocol.sendFile(map_file);
}

void Client::sendSnapShot(std::stringstream & ss) {
    this->protocol.sendGameMapAsString(ss);
}

void Client::setIdInMatch(size_t id) {
    this->defeated = false;
    this->exited = false;
    this->in_match_id = id;
}

size_t Client::getIdInMatch(void) {
    return this->in_match_id;
}

void Client::rcvMapGame(std::fstream & map_file) {
    this->protocol.rcvFile(map_file);
}

void Client::setExited(void) {
    this->exited = true;
}

void Client::setDefeated(void) {
    this->defeated = true;
}

bool Client::isDefeated(void) {
    return this->defeated;
}

bool Client::isExited(void) {
    return this->exited;
}
```

jun 25, 18 20:09 **client.h** Page 1/1

```
#ifndef __CLIENT_H__
#define __CLIENT_H__

#include "protocol.h"
#include "yaml.h"
#include "types.h"
#include <string>
#include <vector>
#include <sstream>
#include <fstream>

class Client {
    private:
        Protocol protocol;
        std::string player_name;
        client_status_t status;
        std::string joined_match_creator_name;
        size_t in_match_id;
        bool defeated;
        bool exited;

    public:
        Client(Protocol, std::string &);
        void sendGamesStatus(YAML::Node);
        Event rcvEvent(void);
        std::string getPlayerName(void);
        void setStatus(client_status_t);
        client_status_t getStatus(void);
        void sendResponse(int, std::string &);
        void sendGameStart(int, std::string &, std::string &);
        void setJoinedMatchGameCreator(std::string &);
        void clearJoinedMatchGameCreator(void);
        std::string getJoinedMatchCreatorName(void);
        void sendWaitingPlayers(std::vector<std::string>);
        void sendMapGame(std::fstream &);
        void sendSnapShot(std::stringstream &);
        void setIdInMatch(size_t);
        size_t getIdInMatch(void);
        void rcvMapGame(std::fstream &);
        void setExited(void);
        void setDefeated(void);
        bool isDefeated(void);
        bool isExited(void);
};

#endif
```

jun 25, 18 20:58 **Configuration.cpp** Page 1/5

```
#include <iostream>
#include "Configuration.h"

Configuration::Configuration() {
    //TURN
    this->TURN_DURATION = 50; //
    //CONVERSION
    this->SCALING_FACTOR = 0.0416; //
    this->DEGTORAD = 0.0174533; //
    this->RADTODEG = 57.2958; //

    //WORLD
    this->WORLD_TIME_STEP = 1.0f/60.0f; //
    this->WORLD_VELOCITY_ITERATIONS = 6; //
    this->WORLD_POSITION_ITERATIONS = 2; //
    this->WORLD_Y_LIMIT = 58.24f;

    //WORM
    this->WORM_SPEED = 2;   //
    this->WORM_FRONT_JUMP_X_IMPULSE = 4; //
    this->WORM_FRONT_JUMP_Y_IMPULSE = 4; //
    this->WORM_BACK_JUMP_X_IMPULSE = 4;
    this->WORM_BACK_JUMP_Y_IMPULSE = 4;
    this->WORM_MAX_FALL_DISTANCE = 4; //

    //BAZOOKA
    this->BAZOOKA_BLAST_RADIUS = 2; //
    this->BAZOOKA_BLAST_POWER = 50; //

    //MORTAR
    this->MORTAR_BLAST_RADIUS = 2; //
    this->MORTAR_BLAST_POWER = 50; //
    this->MORTAR_FRAGMENT_QUANTITY = 6; //
    this->MORTAR_FRAGMENT_BLAST_RADIUS = 2; //
    this->MORTAR_FRAGMENT_BLAST_POWER = 10; //

    //WIND
    this->MAX_WIND_FORCE = 6; //
    this->MIN_WIND_FORCE = -6; //

    //GRENADE
    this->GRENADE_RESTITUTION = 0.6; //

    //RED GRENADE
    this->RED_GRENADE_BLAST_RADIUS = 2; //
    this->RED_GRENADE_BLAST_POWER = 30; //
    this->RED_GRENADE_FRAGMENT_QUANTITY = 50; //
    this->RED_GRENADE_FRAGMENT_BLAST_RADIUS = 2; //
    this->RED_GRENADE_FRAGMENT_BLAST_POWER = 10; //

    //BANANA
    this->BANANA_BLAST_RADIUS = 4; //
    this->BANANA_BLAST_POWER = 70; //

    //HOLY GRENADE
    this->HOLY_GRENADE_BLAST_RADIUS = 8; //
    this->HOLY_GRENADE_BLAST_POWER = 110; //

    //AIRSTRIKE
    this->AIR_STRIKE_BLAST_RADIUS = 2; //
    this->AIR_STRIKE_BLAST_POWER = 40; //
    this->AIR_STRIKE_MISSIL_QUANTITY = 6; //
```

```
    //BAT
    this->BAT_DAMAGE = 10; //
    this->BAT_LENGTH = 1; //
    this->BAT_IMPULSE = 10; //

    //DYNAMITE
    this->DYNAMITE_BLAST_RADIUS = 4; //
    this->DYNAMITE_BLAST_POWER =50; //

    //GREEN GRENADE
    this->GREEN_GRENADE_BLAST_RADIUS = 2; //
    this->GREEN_GRENADE_BLAST_POWER = 30; //
}

void Configuration::loadConfigFile(YAML::Node & configNode) {
    size_t counter = 0;

    if (configNode["match"]["turn_duration"]) {
        this->TURN_DURATION = configNode["match"]["turn_duration"].as<float>(); //
        counter++;
    }

    if (configNode["world_physics"]["scaling_factor_pixels_meters"]) {
        this->SCALING_FACTOR = configNode["world_physics"]["scaling_factor_pixels_meters"].
as<float>(); //
        counter++;
    }

    if (configNode["world_physics"]["scaling_degree_radian"]) {
        this->DEGTORAD = configNode["world_physics"]["scaling_degree_radian"].as<float>()
; //
        counter++;
    }

    if (configNode["world_physics"]["scaling_radian_degree"]) {
        this->RADTODEG = configNode["world_physics"]["scaling_radian_degree"].as<float>()
; //
        counter++;
    }

    if (configNode["world_physics"]["world_time_step"]) {
        this->WORLD_TIME_STEP = configNode["world_physics"]["world_time_step"].as<float
>(); //
        counter++;
    }

    if (configNode["world_physics"]["world_velocity_iterations"]) {
        this->WORLD_VELOCITY_ITERATIONS = configNode["world_physics"]["world_velocity_it
erations"].as<float>(); //
        counter++;
    }

    if (configNode["world_physics"]["world_position_iterations"]) {
        this->WORLD_POSITION_ITERATIONS = configNode["world_physics"]["world_position_it
erations"].as<float>(); //
        counter++;
    }

    if (configNode["world_physics"]["world_y_limit"]) {
        this->WORLD_Y_LIMIT = configNode["world_physics"]["world_y_limit"].as<float>()
; //
```



```
        counter++;
    }

    if (configNode["world_physics"]["max_wind_force"]) {
        this->MAX_WIND_FORCE = configNode["world_physics"]["max_wind_force"].as<int>(
); //
        counter++;
    }

    if (configNode["world_physics"]["min_wind_force"]) {
        this->MIN_WIND_FORCE = configNode["world_physics"]["min_wind_force"].as<int>()
; //
        counter++;
    }

    if (configNode["worms"]["worms_speed"]) {
        this->WORM_SPEED = configNode["worms"]["worms_speed"].as<float>(); //
        counter++;
    }

    if (configNode["worms"]["worm_front_jump_x_impulse"]) {
        this->WORM_FRONT_JUMP_X_IMPULSE = configNode["worms"]["worm_front_jump_x_imp
ulse"].as<float>(); //
        counter++;
    }


    if (configNode["worms"]["worm_front_jump_y_impulse"]) {
        this->WORM_FRONT_JUMP_Y_IMPULSE = configNode["worms"]["worm_front_jump_y_imp
ulse"].as<float>(); //
        counter++;
    }

    if (configNode["worms"]["worm_back_jump_x_impulse"]) {
        this->WORM_BACK_JUMP_X_IMPULSE = configNode["worms"]["worm_back_jump_x_impu
lse"].as<float>(); //
        counter++;
    }

    if (configNode["worms"]["worm_back_jump_y_impulse"]) {
        this->WORM_BACK_JUMP_Y_IMPULSE = configNode["worms"]["worm_back_jump_y_impu
lse"].as<float>(); //
        counter++;
    }

    if (configNode["worms"]["worm_max_fall_distance"]) {
        this->WORM_MAX_FALL_DISTANCE = configNode["worms"]["worm_max_fall_distance"].
as<int>(); //
        counter++;
    }

    if (configNode["weapons"]["physics"]["grenade_restitution"]) {
        this->GRENADE_RESTITUTION = configNode["weapons"]["physics"]["grenade_restitution
"].as<float>(); //
        counter++;
    }

    if (configNode["weapons"]["bazooka"]) {
        this->BAZOOKA_BLAST_RADIUS = configNode["weapons"]["bazooka"]["radious"].as<
int>(); //
        this->BAZOOKA_BLAST_POWER = configNode["weapons"]["bazooka"]["damage"].as<i
nt>(); //
```

```
        counter += 2;
    }

    if (configNode["weapons"]["mortar"]) {
        this->MORTAR_BLAST_RADIUS = configNode["weapons"]["mortar"]["radious"].as<int>(); //
        this->MORTAR_BLAST_POWER = configNode["weapons"]["mortar"]["damage"].as<int>(); //
        this->MORTAR_FRAGMENT_QUANTITY = configNode["weapons"]["mortar"]["fragments_qty"].as<int>(); //
        this->MORTAR_FRAGMENT_BLAST_RADIUS = configNode["weapons"]["mortar"]["fragments_radious"].as<int>(); //
        this->MORTAR_FRAGMENT_BLAST_POWER = configNode["weapons"]["mortar"]["fragments_damage"].as<int>(); //
        counter += 5;
    }

    if (configNode["weapons"]["red_grenade"]) {
        this->RED_GRENADE_BLAST_RADIUS = configNode["weapons"]["red_grenade"]["radious"].as<int>(); //
        this->RED_GRENADE_BLAST_POWER = configNode["weapons"]["red_grenade"]["damage"].as<int>(); //
        this->RED_GRENADE_FRAGMENT_QUANTITY = configNode["weapons"]["red_grenade"]["fragments_qty"].as<int>(); //
        this->RED_GRENADE_FRAGMENT_BLAST_RADIUS = configNode["weapons"]["red_grenade"]["fragments_radious"].as<int>(); //
        this->RED_GRENADE_FRAGMENT_BLAST_POWER = configNode["weapons"]["red_grenade"]["fragments_damage"].as<int>(); //
        counter += 5;
    }

    if (configNode["weapons"]["banana"]) {
        this->BANANA_BLAST_RADIUS = configNode["weapons"]["banana"]["radious"].as<int>(); //
        this->BANANA_BLAST_POWER = configNode["weapons"]["banana"]["damage"].as<int>(); //
        counter += 2;
    }

    if (configNode["weapons"]["holy_grenade"]) {
        this->HOLY_GRENADE_BLAST_RADIUS = configNode["weapons"]["holy_grenade"]["radious"].as<int>(); //
        this->HOLY_GRENADE_BLAST_POWER = configNode["weapons"]["holy_grenade"]["damage"].as<int>(); //
        counter += 2;
    }

    if (configNode["weapons"]["air_strike"]) {
        this->AIR_STRIKE_BLAST_RADIUS = configNode["weapons"]["air_strike"]["radious"].as<int>(); //
        this->AIR_STRIKE_BLAST_POWER = configNode["weapons"]["air_strike"]["damage"].as<int>(); //
        this->AIR_STRIKE_MISSIL_QUANTITY = configNode["weapons"]["air_strike"]["qty"].as<int>(); //
        counter += 3;
    }

    if (configNode["weapons"]["bat"]) {
        this->BAT_DAMAGE = configNode["weapons"]["bat"]["damage"].as<int>(); //
        this->BAT_LENGTH = configNode["weapons"]["bat"]["length"].as<int>(); //
        this->BAT_IMPULSE = configNode["weapons"]["bat"]["impulse"].as<int>(); //
        counter += 3;
```



```
    }

    if (configNode["weapons"]["dynamite"]) {
        this->DYNAMITE_BLAST_RADIUS = configNode["weapons"]["dynamite"]["radious"].as<int>(); //
        this->DYNAMITE_BLAST_POWER = configNode["weapons"]["dynamite"]["damage"].as<int>(); //
        counter += 2;
    }

    if (configNode["weapons"]["green_grenade"]) {
        this->GREEN_GRENADE_BLAST_RADIUS = configNode["weapons"]["green_grenade"]["radious"].as<int>(); //
        this->GREEN_GRENADE_BLAST_POWER = configNode["weapons"]["green_grenade"]["damage"].as<int>(); //
        counter += 2;
    }
}
```

```
#ifndef CONFIGURATION_H
#define CONFIGURATION_H

#include "yaml.h"

class Configuration {
public:
    Configuration();
    void loadConfigFile(YAML::Node &);
    //TURN
    float TURN_DURATION;

    //CONVERSIONS
    float DEGTORAD;
    float RADTODEG;
    float SCALING_FACTOR;

    //WORLD
    float WORLD_TIME_STEP;
    float WORLD_VELOCITY_ITERATIONS;
    float WORLD_POSITION_ITERATIONS;
    float WORLD_Y_LIMIT;

    //WORM
    float WORM_SPEED;
    float WORM_JUMP_IMPULSE;
    int WORM_MAX_FALL_DISTANCE;
    int WORM_FRONT_JUMP_X_IMPULSE;
    int WORM_FRONT_JUMP_Y_IMPULSE;
    int WORM_BACK_JUMP_X_IMPULSE;
    int WORM_BACK_JUMP_Y_IMPULSE;

    //BAZOOKA
    int BAZOOKA_BLAST_RADIUS;
    int BAZOOKA_BLAST_POWER;

    //MORTAR
    int MORTAR_BLAST_RADIUS;
    int MORTAR_BLAST_POWER;
    int MORTAR_FRAGMENT_QUANTITY;
    int MORTAR_FRAGMENT_BLAST_RADIUS;
    int MORTAR_FRAGMENT_BLAST_POWER;

    //GRENADE
    float GRENADE_RESTITUTION;

    //RED GRENADE
    int RED_GRENADE_BLAST_RADIUS;
    int RED_GRENADE_BLAST_POWER;
    int RED_GRENADE_FRAGMENT_QUANTITY;
    int RED_GRENADE_FRAGMENT_BLAST_RADIUS;
    int RED_GRENADE_FRAGMENT_BLAST_POWER;

    //WIND
    int MAX_WIND_FORCE;
    int MIN_WIND_FORCE;

    //BANANA
    int BANANA_BLAST_RADIUS;
    int BANANA_BLAST_POWER;

    //HOLY GRENADE
```



```
    int HOLY_GRENADE_BLAST_RADIUS;
    int HOLY_GRENADE_BLAST_POWER;

    //AIRSTRIKE
    int AIR_STRIKE_BLAST_RADIUS;
    int AIR_STRIKE_BLAST_POWER;
    int AIR_STRIKE_MISSIL_QUANTITY;

    //BAT
    int BAT_DAMAGE;
    int BAT_LENGTH;
    int BAT_IMPULSE;

    //DYNAMITE
    int DYNAMITE_BLAST_RADIUS;
    int DYNAMITE_BLAST_POWER;

    //GREEN GRENADE
    int GREEN_GRENADE_BLAST_RADIUS;
    int GREEN_GRENADE_BLAST_POWER;
};

extern Configuration gConfiguration;

#endif
```

```
#include "ContactListener.h"
#include <iostream>

ContactListener::ContactListener(){}
ContactListener::~ContactListener() {}

void ContactListener::BeginContact(b2Contact* contact) {

    b2WorldManifold worldManifold;
    contact->GetWorldManifold(&worldManifold);

    void* bodyAUserData = contact->GetFixtureA()->GetBody()->GetUserData();
    void* bodyBUserData = contact->GetFixtureB()->GetBody()->GetUserData();

    if (bodyAUserData && bodyBUserData) {
        entity_t entityA_type = static_cast<Entity*>(bodyAUserData)->getEntityTy
pe();
        entity_t entityB_type = static_cast<Entity*>(bodyBUserData)->getEntityTy
pe();

        //WORM FOOT CONTACT
        if (entityA_type == WORM && entityB_type == STRUCTURE) {
            float angle = static_cast<Girder*>(bodyBUserData)->getAngle();
            static_cast<Worm*>(bodyAUserData)->setNormal(worldManifold.normal);
            if (angle <= 0.8 && angle >= -0.8) {
                static_cast<Worm*>(bodyAUserData)->setAngle(angle);
                static_cast<Worm*>(bodyAUserData)->addFootContact();
            }
        }

        if (entityB_type == WORM && entityA_type == STRUCTURE) {
            float angle = static_cast<Girder*>(bodyAUserData)->getAngle();
            static_cast<Worm*>(bodyBUserData)->setNormal(worldManifold.normal);
            if (angle <= 0.8 && angle >= -0.8) {
                static_cast<Worm*>(bodyBUserData)->setAngle(angle);
                static_cast<Worm*>(bodyBUserData)->addFootContact();
            }
        }

        //WORM WATER CONTACT
        if (entityA_type == WORM && entityB_type == WATER) {
           static_cast<Worm*>(bodyAUserData)->kill();
        }

        if (entityB_type == WORM && entityA_type == WATER) {
           static_cast<Worm*>(bodyBUserData)->kill();
        }

        //GRENADE WALL/WATER CONTACT
        if (entityA_type == GRENADE && (entityB_type == WALL || entityB_type ==
WATER)) {
            static_cast<Grenade*>(bodyAUserData)->explode();
        }

        if (entityB_type == GRENADE && (entityA_type == WALL || entityA_type ==
WATER)) {
            static_cast<Grenade*>(bodyBUserData)->explode();
        }

        //BAZOOKA WORM/STRUCTURE CONTACT
        if (entityA_type == BAZOOKA && (entityB_type == WORM || entityB_type ==
STRUCTURE || entityB_type == WATER || entityB_type == WALL)) {
```



```
            static_cast<Bazooka*>(bodyAUserData)->explode();
        }

        if (entityB_type == BAZOOKA && (entityA_type == WORM || entityA_type ==
STRUCTURE || entityA_type == WATER || entityA_type == WALL)) {
            static_cast<Bazooka*>(bodyBUserData)->explode();
        }

        //MISSIL STRUCTURE/WATER CONTACT
        if (entityA_type == MISSIL && (entityB_type == WORM || entityB_type == S
TRUCTURE || entityB_type == WATER || entityB_type == WALL)) {
            static_cast<Missil*>(bodyAUserData)->explode();
        }

        if (entityB_type == MISSIL && (entityA_type == WORM || entityA_type == S
TRUCTURE || entityA_type == WATER || entityA_type == WALL)) {
            static_cast<Missil*>(bodyBUserData)->explode();
        }

        // FRAGMENT STRUCTURE/WATER CONTACT
        if (entityA_type == FRAGMENT && (entityB_type == WORM || entityB_type ==
 STRUCTURE || entityB_type == WATER)) {
            static_cast<Fragment*>(bodyAUserData)->explode();
        }

        if (entityB_type == FRAGMENT && (entityA_type == WORM || entityA_type ==
 STRUCTURE || entityA_type == WATER)) {
            static_cast<Fragment*>(bodyBUserData)->explode();
        }
    }
}

void ContactListener::EndContact(b2Contact* contact) {
    void* bodyAUserData = contact->GetFixtureA()->GetBody()->GetUserData();
    void* bodyBUserData = contact->GetFixtureB()->GetBody()->GetUserData();

    if (bodyAUserData && bodyBUserData) {
        entity_t entityA_type = static_cast<Entity*>(bodyAUserData)->getEntityTy
pe();
        entity_t entityB_type = static_cast<Entity*>(bodyBUserData)->getEntityTy
pe();

        //WORM FOOT CONTACT
        if (entityA_type == WORM && entityB_type == STRUCTURE) {
            float angle = static_cast<Girder*>(bodyBUserData)->getAngle();
            if (angle <= 0.8 && angle >= -0.8) {
                static_cast<Worm*>(bodyAUserData)->deleteFootContact();
                static_cast<Worm*>(bodyAUserData)->setNormal(b2Vec2(0,0));

            }
        }

        if (entityB_type == WORM && entityA_type == STRUCTURE) {
            float angle = static_cast<Girder*>(bodyAUserData)->getAngle();
            if (angle <= 0.8 && angle >= -0.8) {
                static_cast<Worm*>(bodyBUserData)->deleteFootContact();
                static_cast<Worm*>(bodyBUserData)->setNormal(b2Vec2(0,0));

            }
        }
    }
}
```

```
#ifndef CONTACT_LISTENER_H
#define CONTACT_LISTENER_H

#include "Box2D.h"
#include "Entity.h"
#include "Girder.h"
#include "Worm.h"
#include "Bat.h"
#include "Bazooka.h"
#include "Missil.h"
#include "Fragment.h"
#include "Grenade.h"

class ContactListener : public b2ContactListener {
public:
    ContactListener();
    virtual ~ContactListener();
    void BeginContact(b2Contact* contact);
    void EndContact(b2Contact* contact);
};

#endif
```



```
#include "Dynamite.h"
#include <iostream>
#include "types.h"

Dynamite::Dynamite(int id, b2World& world, float posX, float posY, int delay, in
t currentTime) :
Weapon(w_dynamite),
detonationTime(currentTime + delay),
world(world) {
    b2BodyDef dynamiteDef;
    dynamiteDef.type = b2_dynamicBody;
    dynamiteDef.fixedRotation = true;
    dynamiteDef.position.Set(posX, posY);
    dynamiteDef.allowSleep = false;
    b2Body* body = world.CreateBody(&dynamiteDef);
    body->SetUserData(this);

    b2PolygonShape dynamiteShape;
    dynamiteShape.SetAsBox(0.1f, 0.2f);

    b2FixtureDef dynamiteFixture;
    dynamiteFixture.shape = &dynamiteShape;
    dynamiteFixture.density = 1;
    dynamiteFixture.friction = 1;
    body->CreateFixture(&dynamiteFixture);

    this->body = body;
    this->exploded = false;
    this->id = id;
    this->blast_radius = gConfiguration.DYNAMITE_BLAST_RADIUS;
    this->blast_power = gConfiguration.DYNAMITE_BLAST_POWER;
}

Dynamite::~Dynamite() {
    this->world.DestroyBody(this->body);
}

void Dynamite::update(int currentTime, int wind_force) {
    if (currentTime >= this->detonationTime && !exploded) {
        explode();
    }
    if (wind_affected) {
       this->body->ApplyForce(body->GetMass() * b2Vec2(wind_force,0), body->GetW
orldCenter(), true);
    }
    this->countdown = this->detonationTime - currentTime;
}

void Dynamite::explode() {
    ExplosionManager explosionManager(this->world);
    b2Vec2 center = this->body->GetPosition();
    explosionManager.manageExplosion(center, blast_radius, blast_power);
    this->exploded = true;
}

float Dynamite::getPosX() {
    return this->body->GetPosition().x;
}

float Dynamite::getPosY() {
    return this->body->GetPosition().y;
}
```

```
bool Dynamite::isMoving() {
    return true;
}
```



```
#ifndef DYNAMITE_H
#define DYNAMITE_H


#include "Weapon.h"
#include "Box2D/Box2D.h"
#include "types.h"
#include "Configuration.h"
#include "ExplosionManager.h"

#define DYNAMITE_WIDTH 0.1f
#define DYNAMITE_HEIGHT 0.2f

class Dynamite : public Weapon{
private:
    int detonationTime;
    b2World& world;
    b2Body* body;
    int blast_power;
public:
    Dynamite(int id, b2World& world, float posX, float posY, int delay, int curr
entTime);
    ~Dynamite();
    void explode(void);
    void update(int currentTime, int wind_force);
    bool hasExploded(void);
    float getPosX(void);
    float getPosY(void);
    bool isMoving(void);
    entity_t getEntityType() {return DYNAMITE;}
};

#endif
```

jun 25, 18 20:09 **Entity.h** Page 1/1

```
#ifndef ENTITY_H
#define ENTITY_H

typedef enum {
    WORM,
    STRUCTURE,
    DYNAMITE,
    BAT,
    WATER,
    BAZOOKA,
    MISSIL,
    GRENADE,
    FRAGMENT,
    WALL,
} entity_t;

class Entity {
public:
    virtual entity_t getEntityType() = 0;
    virtual float getPosX() = 0;
    virtual float getPosY() = 0;
    //virtual void update(int currentTime) = 0;
};

#endif
```

jun 25, 18 20:09 **event_receiver.cpp** Page 1/1

```
#include "event_receiver.h"
#include "protocol.h"
#include "event.h"
#include "match.h"

EventReceiver::EventReceiver(Client * cli, World & w, Match & match, size_t id)
:
client(cli),
world(w),
match(match),
team_id(id) {
    this->keep_running = true;
    this->quit_event = false;
}

bool EventReceiver::isRunning(void) const {
    return this->keep_running;
}

size_t EventReceiver::getId(void) const {
    return this->team_id;
}

void EventReceiver::run(void) {
    while (keep_running) {
        Event new_event = this->client->rcvEvent();
        if (new_event.quit()) {
            std::cout << "Recibido evento de quit." << std::endl;
            this->quit_event = true;
            stop();
            return;
        }
        int team_turn = match.getTeamTurn();
        if ((int) new_event.getTeamId() != team_turn) {
            continue;
        }
        if (match.getTurnTimeleft() > 0) {
            // if (match.extraTime() && new_event.getNode()["event"]["action"].a
s<int>() == a_shoot) {
            //     continue;
            // }
            this->world.executeAction(new_event, match.getWormTurn(team_turn));

        }
    }
}

bool EventReceiver::quitEvent(void) {
    return this->quit_event;
}

void EventReceiver::stop(void) {
    this->keep_running = false;
}
```

jun 25, 18 20:09 **event_receiver.h** Page 1/1

```
#ifndef __EVENT_RECEIVER_H__
#define __EVENT_RECEIVER_H__

#include "thread.h"
#include "World.h"
#include "client.h"
#include "match.h"

class EventReceiver : public Thread {
    private:
        Client * client;
        World & world;
        Match & match;
        size_t team_id;
        bool keep_running;
        bool quit_event;

    public:
        EventReceiver(Client *, World &, Match &, size_t);
        virtual void run(void);
        void stop(void);
        bool quitEvent(void);
        virtual bool isRunning(void) const;
        virtual size_t getId(void) const;
};

#endif
```

jun 25, 18 20:09 **ExplosionManager.cpp** Page 1/1

```
#include "ExplosionManager.h"

/*
Fuente: https://www.iforce2d.net/b2dtut/explosions
*/

ExplosionManager::ExplosionManager(b2World& world) :
world(world) {
}

void ExplosionManager::manageExplosion(b2Vec2 center, float radius, float power)
 {
    QueryCallback queryCallback;
    b2AABB aabb;
    aabb.lowerBound = center - b2Vec2(radius, radius);
    aabb.upperBound = center + b2Vec2(radius, radius);
    this->world.QueryAABB(&queryCallback, aabb);

    for (unsigned int i = 0; i < queryCallback.foundBodies.size(); i++) {
        b2Body* body = queryCallback.foundBodies[i];
        b2Vec2 bodyCom = body->GetWorldCenter();
        if ((bodyCom - center).Length() > radius)
            continue;
        this->applyBlastImpulse(body, center, bodyCom, power, radius);
    }
}

int ExplosionManager::calculateDamage(float blastPower, float radius, float dist
ance) {
    return blastPower * ((-distance/radius) + 1);
}

void ExplosionManager::applyBlastImpulse(b2Body* body, b2Vec2 blastCenter, b2Vec
2 applyPoint, float blastPower, float radius) {
        b2Vec2 blastDir = applyPoint - blastCenter;
        float distance = blastDir.Normalize();
    int damage = this->calculateDamage(blastPower, radius, distance);
    if (blastDir.y > 0) blastDir.y = -blastDir.y;
        if (distance <= 1) distance = 1;
        float invDistance = 1/distance;
        float impulseMag = (blastPower/REDUCE_FACTOR) * invDistance;
    entity_t entity_type = static_cast<Entity*>(body->GetUserData())->getEntityT
ype();
    if (entity_type == WORM) {
        body->ApplyLinearImpulse(impulseMag * blastDir, applyPoint, true);
        static_cast<Worm*>(body->GetUserData())->hurt(damage);
        static_cast<Worm*>(body->GetUserData())->setAffectedByExplosion();
    }
}
```

jun 25, 18 20:09 **ExplosionManager.h** Page 1/1

```
#ifndef EXPLOSION_MANAGER_H
#define EXPLOSION_MANAGER_H

#define GRADTORAD 0.0174533
#define REDUCE_FACTOR 5

#include "Box2D.h"
#include "Entity.h"
#include "Worm.h"
#include "QueryCallback.h"

class ExplosionManager {
private:
    b2World& world;
    void applyBlastImpulse(b2Body* body, b2Vec2 blastCenter, b2Vec2 applyPoint,
float blastPower, float radius);
    int calculateDamage(float blastPower, float radius, float distance);
public:
    ExplosionManager(b2World& world);
    void manageExplosion(b2Vec2 center, float radius, float power);

};

#endif
```

jun 25, 18 20:09 **Fragment.cpp** Page 1/2

```
#include "Fragment.h"

Fragment::Fragment(int id, b2World& world, float posX, float posY, weapon_t type
) :
Weapon(type),
world(world) {
    b2BodyDef fragmentDef;
    fragmentDef.type = b2_dynamicBody;
    fragmentDef.position.Set(posX, posY);
    b2Body* body = world.CreateBody(&fragmentDef);
    body->SetUserData(this);

    b2CircleShape fragmentShape;
    fragmentShape.m_radius = FRAGMENT_RADIUS;

    b2FixtureDef fragmentFixture;
    fragmentFixture.shape = &fragmentShape;
    fragmentFixture.density = 1;
    fragmentFixture.friction = 0.3;

    body->CreateFixture(&fragmentFixture);

    this->body = body;
    this->blast_radius = gConfiguration.AIR_STRIKE_BLAST_RADIUS;
    this->blast_power = gConfiguration.AIR_STRIKE_BLAST_POWER;
    this->exploded = false;
    this->id = id;
    this->wind_affected = true;
    shoot();
}

Fragment::~Fragment() {
    this->world.DestroyBody(this->body);
}

void Fragment::explode() {
    ExplosionManager explosionManager(this->world);
    b2Vec2 center = this->body->GetPosition();
    explosionManager.manageExplosion(center, blast_radius, blast_power);
    this->exploded = true;
}

float Fragment::getPosX() {
    return this->body->GetPosition().x;
}

float Fragment::getPosY() {
    return this->body->GetPosition().y;
}

void Fragment::update(int currentTime, int wind_force) {
    if (this->body->GetPosition().y > gConfiguration.WORLD_Y_LIMIT /*|| contact*
/) {
        this->explode();
    }

    if (wind_affected) {
        this->body->ApplyForce(body->GetMass() * b2Vec2(wind_force, 0), body->Ge
tWorldCenter(), true);
    }

    b2Vec2 mov_speed = this->body->GetLinearVelocity();
```

```
    if (round(mov_speed.x) == 0) {
        if (mov_speed.y > 0) {
            this->direction_angle = 180;
            return;
        }

        if (mov_speed.y < 0) {
            this->direction_angle = 0;
            return;
        }
    }

    if (round(mov_speed.y) == 0) {
        if (mov_speed.x > 0) {
            this->direction_angle = 90;
            return;
        }

        if (mov_speed.x < 0) {
            this->direction_angle = 270;
            return;
        }
    }

    int ang = atan(mov_speed.x/mov_speed.y) * gConfiguration.RADTODEG;

    // Primer cuadrante
    if (mov_speed.y < 0 && mov_speed.x > 0) {
        this->direction_angle = -ang;
    }

    // Segundo cuadrante
    if (mov_speed.y < 0 && mov_speed.x < 0) {
        this->direction_angle = 360 - ang;
    }

    // Tercer cuadrante
    if (mov_speed.y > 0 && mov_speed.x < 0) {
        this->direction_angle = 180 - ang;
    }

    // Cuarto cuadrante
    if (mov_speed.y > 0 && mov_speed.x > 0) {
        this->direction_angle = 180 - ang;
    }
}

bool Fragment::isMoving() {
    b2Vec2 speed = this->body->GetLinearVelocity();
    if (!speed.x && !speed.y) return false;
    return true;
}

void Fragment::shoot() {
    float impulse = this->body->GetMass() * 7;
    int angle = rand()%(181);
    float x_impulse = cosf(angle * gConfiguration.DEGTORAD) * impulse;
    float y_impulse = sinf(angle * gConfiguration.DEGTORAD) * impulse;
    this->body->ApplyLinearImpulse(b2Vec2(x_impulse, -y_impulse), this->body->Ge
tWorldCenter(), true);
}
```



```
#ifndef FRAGMENT_H
#define FRAGMENT_H

#include "Weapon.h"
#include "Box2D.h"
#include "ExplosionManager.h"
#include "Configuration.h"
#include <stdlib.h>

#define FRAGMENT_RADIUS 0.25

class Fragment : public Weapon {
private:
    int blast_power;
    b2World& world;
    b2Body* body;
    void shoot();
public:
    Fragment(int id, b2World& world, float posX, float posY, weapon_t type);
    ~Fragment();
    void explode();
    float getPosX();
    float getPosY();
    bool isMoving();
    entity_t getEntityType() {return FRAGMENT;}
    void update(int currentTime, int wind_force);
};

#endif
```

```
#include "Girder.h"

Girder::Girder(b2World& world, float posX, float posY, float angle, float height
, float width) :
world(world) {
    b2BodyDef girderDef;
    girderDef.type = b2_staticBody;
    girderDef.position.Set(posX, posY);
    girderDef.allowSleep = true;
    girderDef.angle = angle;
    b2Body* body = world.CreateBody(&girderDef);
    body->SetUserData(this);
    body->SetAwake(false);

    b2PolygonShape girderShape;
    girderShape.SetAsBox(width/2, height/2);

    b2FixtureDef girderFixture;
    girderFixture.shape = &girderShape;
    girderFixture.filter.categoryBits = STRUCTURE_PHYSIC;
    girderFixture.filter.maskBits = WORM_PHYSIC;
    if (angle <= 0.8 && angle >= -0.8) {
        girderFixture.friction = GIRDER_FRICTION;
    } else {
        girderFixture.friction = SLIPPERY_GIRDER;
    }
    girderFixture.density = GIRDER_DENSITY;

    body->CreateFixture(&girderFixture);

    this->body = body;
}

float Girder::getPosX() {
        return this->body->GetPosition().x;
}

float Girder::getPosY() {
    return this->body->GetPosition().y;
}

float Girder::getAngle() {
    return this->body->GetAngle();
}

Girder::~Girder(void) {
    this->world.DestroyBody(this->body);
}

void Girder::update() {
    this->body->SetAwake(false);
}
```



```
#ifndef GIRDER_H
#define GIRDER_H

#include "Box2D.h"
#include "PhysicEntity.h"
#include "Entity.h"
#include "Configuration.h"

#define GIRDER_DENSITY 1
#define GIRDER_FRICTION 5
#define SLIPPERY_GIRDER 0

class Girder : public Entity {
private:
    b2World& world;
    b2Body* body;
public:
    Girder(b2World& world, float posX, float posY, float angle, float height, fl
oat width);
    virtual ~Girder(void);
    float getPosX();
    float getPosY();
    float getAngle();
    entity_t getEntityType() {return STRUCTURE;}
    void update(void);
};

#endif
```

```
#include "Grenade.h"
#include "types.h"
#include <iostream>

Grenade::Grenade(int id, b2World& world, float posX, float posY, bool mirrored,
float shooting_angle, int power_factor, int delay, int currentTime, weapon_t typ
e) :
Weapon(type),
detonationTime(currentTime + delay),
world(world) {
    b2BodyDef grenadeDef;
    grenadeDef.type = b2_dynamicBody;
    grenadeDef.position.Set(posX, posY);
    b2Body* body = world.CreateBody(&grenadeDef);
    body->SetUserData(this);

    b2CircleShape grenadeShape;
    grenadeShape.m_radius = GRENADE_RADIUS;

    b2FixtureDef grenadeFixture;
    grenadeFixture.shape = &grenadeShape;
    grenadeFixture.density = GRENADE_DENSITY;
    grenadeFixture.friction = 50000000;
    grenadeFixture.restitution = gConfiguration.GRENADE_RESTITUTION;
    body->CreateFixture(&grenadeFixture);

    this->body = body;

    this->exploded = false;
    this->power_factor = power_factor;
    this->mirrored = mirrored;
    this->shooting_angle = shooting_angle;
    this->id = id;

    if (type == w_banana) {
        this->blast_power = gConfiguration.BANANA_BLAST_POWER;
        this->blast_radius = gConfiguration.BANANA_BLAST_RADIUS;
    } else if (type == w_holy_grenade) {
        this->blast_power = gConfiguration.HOLY_GRENADE_BLAST_POWER;
        this->blast_radius = gConfiguration.HOLY_GRENADE_BLAST_RADIUS;
    } else if (type == w_green_grenade) {
        this->blast_power = gConfiguration.GREEN_GRENADE_BLAST_POWER;
        this->blast_radius = gConfiguration.GREEN_GRENADE_BLAST_RADIUS;
    } else if (type == w_cluster) {
        this->blast_power = gConfiguration.RED_GRENADE_BLAST_POWER;
        this->blast_radius = gConfiguration.RED_GRENADE_BLAST_RADIUS;
    }
    shoot();
}

Grenade::~Grenade() {
    this->world.DestroyBody(this->body);
}

void Grenade::update(int currentTime, int wind_force) {
    if (currentTime >= this->detonationTime && !exploded) {
        explode();
    }
    this->countdown = this->detonationTime - currentTime;

    if (wind_affected) {
        this->body->ApplyForce(body->GetMass() * b2Vec2(wind_force,0), body->Get
```



```
WorldCenter(), true);
    }
}

void Grenade::explode() {
    ExplosionManager explosioManager(this->world);
    b2Vec2 center = this->body->GetPosition();
    explosioManager.manageExplosion(center, blast_radius, blast_power);
    this->exploded = true;
}

float Grenade::getPosX() {
    return this->body->GetPosition().x;
}

float Grenade::getPosY() {
    return this->body->GetPosition().y;
}

void Grenade::shoot() {
    float impulse = this->body->GetMass() * power_factor;
    float x_impulse = cosf(shooting_angle * gConfiguration.DEGTORAD) * impulse;
    float y_impulse = sinf(shooting_angle * gConfiguration.DEGTORAD) * impulse;
    if (!mirrored) x_impulse = x_impulse * -1;
    this->body->ApplyLinearImpulse(b2Vec2(x_impulse, -y_impulse), this->body->Ge
tWorldCenter(), true);
}

bool Grenade::isMoving() {
    b2Vec2 speed = this->body->GetLinearVelocity();
    if (!speed.x && !speed.y) return false;
    return true;
}
```

```
#ifndef GRENADE_H
#define GRENADE_H

#include "Box2D.h"
#include "Weapon.h"
#include "types.h"
#include "Configuration.h"
#include "ExplosionManager.h"

#define GRENADE_RADIUS 0.25f
#define GRENADE_DENSITY 1

class Grenade : public Weapon {
protected:
    int detonationTime;
    b2World & world;
    b2Body* body;
    int blast_power;
    bool mirrored;
    int shooting_angle;
    int power_factor;
public:
    Grenade(int id, b2World& world, float posX, float posY, bool mirrored, float
 shooting_angle, int power_factor, int delay, int currentTime, weapon_t type);
    ~Grenade();
    void shoot(void);
    void update(int current_time, int wind_force);
    float getPosX();
    float getPosY();
    bool isMoving();
    entity_t getEntityType() {return GRENADE;}
    void explode(void);
};

#endif
```



```
#include <fstream>
#include "lobby_attendant.h"
#include "thread.h"
#include "client.h"
#include "yaml.h"

LobbyAttendant::LobbyAttendant(Client * c, ProtectedWaitingGames & wg) :
client(c),
waiting_games (wg) {
    this->keep_running = true;
    this->player_name = client->getPlayerName();
}

bool LobbyAttendant::isRunning(void) const {
    return this->keep_running;
}

size_t LobbyAttendant::getId(void) const {
    return 0;
}

void LobbyAttendant::stop(void) {
    this->keep_running = false;
}

void LobbyAttendant::run(void) {
    while (this->keep_running) {
        Event new_event = this->client->rcvEvent();
        if (new_event.quit()) {
            std::cout << "El cliente " << this->client->getPlayerName() << " ha saldo de
l lobby." << std::endl;
            this->client->setStatus(quited);
            this->keep_running = false;
            return;
        }
        if (new_event.goToMatch()) {
            std::string creator_match_name = this->client->getJoinedMatchCreator
Name();
            this->waiting_games.waitGameUntilFinish(creator_match_name);
            continue;
        }
        processEvent(new_event);
    }
}

void LobbyAttendant::processEvent(Event & event) {
    YAML::Node event_node = event.getNode();

    action_t action = (action_t) event_node["event"]["action"].as<int>();

    switch(action) {
        case a_refreshLobby: {
            refreshLobby();
            break;
        }
        case a_createMatch: {
            std::string match_name = event_node["event"]["match_name"].as<std::str
ing>();
            size_t players_qty = event_node["event"]["map_players_qty"].as<size_t>()
;
            createMatch(match_name, players_qty);
            break;
```

```
        }
        case a_rmWaitingMatch: {
            removeWaitingMatch();
            break;
        }
        case a_joinWaitingMatch: {
            std::string match_creator_name = event_node["event"]["creator_name"].as<
std::string>();
            joinWaitingMatch(match_creator_name);
            break;
        }
        case a_exitWaitingMatch: {
            exitWaitingMatch();
            break;
        }
        case a_refreshWaitingList: {
            refreshWaitingList();
            break;
        }
        case a_startMatch: {
            startMatch();
            break;
        }
        default: break;
    }
}

void LobbyAttendant::refreshLobby(void) {
    this->client->sendGamesStatus(this->waiting_games.getGamesInfoNode());
}

void LobbyAttendant::createMatch(std::string & match_name, size_t map_players_qt
y) {
    std::cout << "El cliente " << this->player_name << " ha creado una partida." << std::end
l;
    this->client->setStatus(creator);
    WaitingGame * new_waiting_game = new WaitingGame(this->client, match_name, m
ap_players_qty);
    this->waiting_games.addNewWaitingGame(this->player_name, new_waiting_game);
    this->client->setJoinedMatchGameCreator(match_name);
}

void LobbyAttendant::removeWaitingMatch(void) {
    std::cout << "El creador de la partida en espera " << this->waiting_games.getGameName(th
is->player_name) << " ha cancelado la partida." << std::endl;
    this->client->setStatus(lobby);
    this->client->clearJoinedMatchGameCreator();
    this->waiting_games.notifyAllCancellGame(this->player_name);
    this->waiting_games.removeGame(this->player_name);
}

void LobbyAttendant::joinWaitingMatch(std::string & match_creator_name) {
    std::cout << "El cliente " << this->player_name << " intenta joinearse a la partida de " << m
atch_creator_name << std::endl;
    if (this->waiting_games.gameHasFreeSlots(match_creator_name)) {
        this->waiting_games.addPlayerToGame(match_creator_name, this->client);
        this->client->setStatus(joined);
        std::string msg = "";
        this->client->sendResponse(1, msg);
        this->client->setJoinedMatchGameCreator(match_creator_name);
    } else {
        std::string msg = "La partida estÃ¡ llena.";
```



```
        this->client->sendResponse(0, msg);
    }
}

void LobbyAttendant::exitWaitingMatch(void) {
    std::string joined_match_creator_name = this->client->getJoinedMatchCreatorN
ame();
    this->waiting_games.rmvPlayerFromGame(joined_match_creator_name, this->playe
r_name);
    this->client->clearJoinedMatchGameCreator();
    std::string msg = "exited";
    this->client->sendResponse(1, msg);
}

void LobbyAttendant::refreshWaitingList(void) {
    std::cout << "El creador de partida " << this->player_name << " quiere hacer refresh de la lista
 de jugadores en espera." << std::endl;
    this->client->sendWaitingPlayers(this->waiting_games.getWaitingPlayers(this-
>player_name));
}

void LobbyAttendant::startMatch(void) {
    std::cout << "El jugador " << this->player_name << " intenta iniciar su partida." << std::
endl;
    if (this->waiting_games.gameHasFreeSlots(this->player_name)) {
        // No se puede iniciar la partida, pues faltan lugares que ocupar.
        std::string msg = "La partida no tiene los suficientes jugadores para iniciar.";
        this->client->sendResponse(0, msg);
    } else {
        // La partida puede comenzar
        std::cout << "La partida puede comenzar, se le informarÃ¡ a todos los participantes." << std::e
ndl;
        this->waiting_games.notifyAllStartGame(this->player_name);
        std::string map_path = this->player_name + "-map.tar.gz";
        std::fstream map_file(map_path, std::fstream::out | std::fstream::binary
 | std::fstream::trunc);
        this->client->rcvMapGame(map_file);
        std::cout << "Recibido el mapa del cliente creador. Va a iniciarse la partida." << std::endl;
        map_file.close();
        this->waiting_games.startWaitingGame(this->player_name, map_path);
        this->waiting_games.removeGame(this->player_name);
        std::cout << "Waiting game removido." << std::endl;
    }
}
```

```
#ifndef __LOBBY_ATTENDANT_H__
#define __LOBBY_ATTENDANT_H__

#include <map>
#include <string>
#include "thread.h"
#include "client.h"
#include "event.h"
#include "waiting_game.h"
#include "protected_waiting_games.h"

class LobbyAttendant : public Thread {
    private:
        Client * client;
        ProtectedWaitingGames & waiting_games;
        bool keep_running;
        std::string player_name;

        size_t getId(void) const;
        void processEvent(Event &);
        void refreshLobby(void);
        void createMatch(std::string &, size_t);
        void removeWaitingMatch(void);
        void joinWaitingMatch(std::string &);
        void exitWaitingMatch(void);
        void refreshWaitingList(void);
        void startMatch(void);

    public:
        LobbyAttendant(Client *, ProtectedWaitingGames &);
        bool isRunning(void) const;
        virtual void run(void);
        void stop(void);
};

#endif
```



```
#include <iostream>
#include <sstream>
#include <string>
#include <fstream>
#include <map>
#include <unistd.h>
#include "yaml.h"
#include "socket.h"
#include "socket_error.h"
#include "protocol.h"
#include "protocol_error.h"
#include "types.h"
#include "World.h"
#include "snapshot_sender.h"
#include "blocking_queue.h"
#include "event.h"
#include "match.h"
#include "event_receiver.h"
#include "Configuration.h"
#include "snapshot.h"
#include "server.h"
#include "server_error.h"
#include <sys/stat.h>

#define MIN_SRV_ARGS_QTY 2
#define MAX_SRV_ARGS_QTY 3
#define SRV_CONFIG_FILE_POS 2
#define SRV_PORT_POS 1

Configuration gConfiguration;


int main(int argc, char *argv[]) try {
     if (argc < MIN_SRV_ARGS_QTY || argc > MAX_SRV_ARGS_QTY) {
        std::cout << "Servidor mal invocado." << std::endl;
        std::cout << "Forma de uso: './server <port> [config-file-path]" << std::endl;
        return 0;
    } else if (argc == MAX_SRV_ARGS_QTY) {
        try {
            std::cout << "Cargando archivo de configuraciÃ³n desde " << argv[SRV_CONFIG_FIL
E_POS] << std::endl;
            YAML::Node config_node = YAML::LoadFile(argv[SRV_CONFIG_FILE_POS]);
            gConfiguration.loadConfigFile(config_node);
        } catch (const YAML::Exception & err) {
            std::cout << "No se pudo abrir el archivo de configuraciÃ³n espcificado. Se usarÃ¡n valores d
e configuraciÃ³n por defecto: " << err.what() << std::endl;
        }
    } else {
        struct stat info;
        if (stat("/usr/etc/worms", &info) == 0) {
            std::cout << "Cargando archivo de configuraciÃ³n en /usr/etc/worms" << std::endl;
            YAML::Node config_node = YAML::LoadFile("/usr/etc/worms/server_config.yml");
            gConfiguration.loadConfigFile(config_node);
        }
    }

    std::string port(argv[SRV_PORT_POS]);
    Server server(port);

    server.start();

    char c;
```

```
    do {
        c = getchar();
    } while (c != 'q');

    std::cout << "Servidor detenido." << std::endl;
    server.stop();
    server.join();

    return 0;

} catch(const SocketError & e) {
    std::cout << e.what() << std::endl;
} catch(const ServerError & s) {
    std::cout << s.what() << std::endl;
} catch(const ProtocolError & p) {
    std::cout << p.what() << std::endl;
} catch(const std::exception & g) {
    std::cout << g.what() << std::endl;
}
```



```
#include <iostream>
#include <unistd.h>
#include <map>
#include <vector>
#include <algorithm>
#include "team.h"
#include "match.h"

#define US_SEC_FACTOR 1000000

Match::Match(std::map<int, Worm*>& worms, std::map<int, Team*> & teams, Wind* wi
nd, size_t td) :
teams(teams),
worms(worms),
wind(wind) {
    this->turn_duration_sec = td;
    this->actual_turn_start_time = 0;
    this->turn_timeleft_sec = td;
    this->match_finished = false;
    this->winner_team = -1;
    this->turn_finished = false;
    this->worms_moving = false;
    this->alive_projectiles = false;
    this->worms_affected_by_explosion = false;
    this->protagonic_worm_got_hurt = false;
    this->protagonic_worm_did_shoot = false;
    this->extra_time = false;
    createTeams();
}

void Match::createTeams() {
    std::map<int, Team*>::const_iterator it;
    for (it = this->teams.begin(); it != this->teams.end(); it++) {
        this->team_turn_order.push((it)->first);
        std::map<int, Worm*> worms_team = (it)->second->getWorms();
        std::map<int, Worm*>::const_iterator worms_it;

        for(worms_it = worms_team.begin(); worms_it != worms_team.end(); worms_i
t++) {
            this->worm_turn_order[(it)->second->getTeamId()].push(worms_it->seco
nd->getId());
        }

    }
}

void Match::printTeams(void) {
    std::map<int, Team *>::const_iterator it;
    for (it = this->teams.begin(); it != this->teams.end(); it++) {
        std::vector<int> worms_id = it->second->getWormsID();
        std::vector<int>::const_iterator it2;
        std::cout << "Equipo NÃºmero " << it->second->getTeamId() << std::endl;
        for (it2 = worms_id.begin(); it2 != worms_id.end(); it2++) {
            std::cout << "Integrante: " << *it2 << std::endl;
        }
    }
}

int Match::getTeamTurn(void) {
    return this->team_turn_order.front();
}
```

```
int Match::getWormTurn(int team_id) {
    return this->worm_turn_order[team_id].front();
}

int Match::nextTurn(void) {
    this->wind->updateWindForce();
    this->extra_time = false;
    int alive_teams;

    int actual_team_turn = getTeamTurn();

    alive_teams = removeDeadTeamsTurns();
    if (alive_teams == 1) {
        this->winner_team = this->team_turn_order.front();
        //Partida con un ganador
        return -1;
    } else if (alive_teams == 0) {
        this->winner_team = 0;
        // Partida sin ganadores
        return -2;
    }

    int actual_worm_turn;
    if (getTeamTurn() == actual_team_turn) {
        actual_worm_turn = getWormTurn(actual_team_turn);
        removeDeadWormsTurns();
        if (getWormTurn(actual_team_turn) == actual_worm_turn) {
            this->worm_turn_order[actual_team_turn].pop();
            this->worm_turn_order[actual_team_turn].push(actual_worm_turn);
        }
        this->team_turn_order.pop();
        this->team_turn_order.push(actual_team_turn);
    }

    refreshWormsFlagsByNewTurn();

    return 0;
}

void Match::refreshWormsFlagsByNewTurn(void) {
    std::map<int, Worm *>::iterator it;
    for (it = this->worms.begin(); it != this->worms.end(); it++) {
        it->second->refreshByNewTurn();
    }
}

int Match::removeDeadTeamsTurns(void) {
    int teams_qty = this->team_turn_order.size();
    for (int i = 0; i < teams_qty; i++) {
        int team_id = this->team_turn_order.front();
        this->team_turn_order.pop();
        if (this->teams[team_id]->haveAliveMember()) {
            this->team_turn_order.push(team_id);
        }
    }
    return this->team_turn_order.size();
}

void Match::removeDeadWormsTurns(void) {
    std::map<int, std::queue<int>>::iterator it;
    for (it = this->worm_turn_order.begin(); it != this->worm_turn_order.end();
it++) {
```



```
        int queue_size = it->second.size();
        for (int i = 0; i < queue_size; i++) {
            int worm_id = it->second.front();
            it->second.pop();
            if (!this->worms[worm_id]->isDead()) {
                it->second.push(worm_id);
            }
        }
    }
}

std::map<size_t, int> Match::getTeamInfo(void) {
    std::map<size_t, int> alive_teams;
    std::map<int, Team *>::const_iterator it;
    for (it = this->teams.begin(); it != this->teams.end(); it++) {
            alive_teams[it->second->getTeamId()] = it->second->getTotalLife();
    }
    return alive_teams;
}

void Match::start(unsigned int actual_time_sec) {
    this->actual_turn_start_time = actual_time_sec;
    std::cout << "Es el turno del equipo " << getTeamTurn() << " con su Worm " << getWormTu
rn(getTeamTurn()) << std::endl;
}

void Match::setMovingWormsFlag(bool flag) {
    this->worms_moving = flag;
}

void Match::setAliveProjectilesFlag(bool flag) {
    this->alive_projectiles = flag;
}

void Match::setWormsAffectedByExplosion(bool flag) {
    this->worms_affected_by_explosion = flag;
}

void Match::setProtagonicWormGotHurt(bool flag) {
    this->protagonic_worm_got_hurt = flag;
}

void Match::setProtagonicWormDidShoot(bool flag) {
    this->protagonic_worm_did_shoot = flag;
}

void Match::update(unsigned int time_passed) {
    //this->turn_timeleft_sec = this->turn_timeleft_sec - time_passed;
    if (this->match_finished) return;
    if (aliveTeams() < 2) {
        this->match_finished = true;
    }

    if (this->turn_timeleft_sec <= 0) {
        this->turn_finished = true;
        this->extra_time = false;
    } else {
        this->turn_timeleft_sec = this->turn_timeleft_sec - time_passed;
    }

    if (this->protagonic_worm_did_shoot && this->turn_timeleft_sec > 3) {
        this->turn_timeleft_sec = 3;
```

```
        this->protagonic_worm_did_shoot = false;
        this->extra_time = true;
    }

    if ((this->protagonic_worm_did_shoot || this->protagonic_worm_got_hurt || th
is->turn_finished)
    && !this->worms_moving && !this->alive_projectiles && !this->worms_affected_
by_explosion && !this->extra_time) {
        //this->extra_time = false;
        if (nextTurn() < 0) {
            std::cout << "No se pudo cambiar de turno, la partida finalizÃ³." << std::endl;
            this->match_finished = true;
        } else {
            std::cout << "Es el turno del equipo " << getTeamTurn() << " con su Worm " << g
etWormTurn(getTeamTurn()) << std::endl;
            this->turn_timeleft_sec = this->turn_duration_sec;
            this->turn_finished = false;
            return;
        }
    }
}

int Match::getTurnTimeleft(void) {
    return this->turn_timeleft_sec;
}

bool Match::finished(void) {
    return this->match_finished;
}

int Match::getWinner(void) {
    return this->winner_team;
}

int Match::getTeamTotalLife(size_t team_id) {
    return this->teams[team_id]->getTotalLife();
}

int Match::getWindForce() {
    return this->wind->getWindForce();
}

bool Match::extraTime() {
    return this->extra_time;
}

void Match::removePlayer(size_t tid) {
    this->teams[(int)tid]->killAll();
    if (getTeamTurn() == (int) tid) {
        if (nextTurn() < 0) {
            std::cout << "No se pudo cambiar de turno, la partida finalizÃ³." << std::endl;
            this->match_finished = true;
        } else {
            std::cout << "Es el turno del equipo " << getTeamTurn() << " con su Worm " << g
etWormTurn(getTeamTurn()) << std::endl;
            this->turn_timeleft_sec = this->turn_duration_sec;
            this->turn_finished = false;
            return;
        }
    }
}
```



```
size_t Match::aliveTeams(void) {
    size_t counter = 0;
    std::map<int, Team *>::const_iterator it;
    for (it = this->teams.begin(); it != this->teams.end(); it++) {
        if (it->second->haveAliveMember()) {
            counter++;
        }
    }
    return counter;
}
```

jun 25, 18 20:09 **match.h** Page 1/1

```
#ifndef __MATCH_H__
#define __MATCH_H__

#include <map>
#include <vector>
#include <queue>
#include "team.h"
#include "Worm.h"
#include "thread.h"
#include "Wind.h"

class Match {
    private:
        std::map<int, Team *> & teams;
        std::map<int, Worm *> & worms;
        Wind* wind;
        std::queue<int> team_turn_order;
        std::map<int, std::queue<int>> worm_turn_order;
        unsigned int turn_duration_sec;
        unsigned int actual_turn_start_time;
        bool match_finished;
        int winner_team;
        int turn_timeleft_sec;
        bool turn_finished;
        bool worms_moving;
        bool alive_projectiles;
        bool worms_affected_by_explosion;
        bool protagonic_worm_got_hurt;
        bool protagonic_worm_did_shoot;
        bool extra_time;

        void createTeams();
        int removeDeadTeamsTurns(void);
        void removeDeadWormsTurns(void);
        void refreshWormsFlagsByNewTurn(void);
        size_t aliveTeams(void);

    public:
        Match(std::map<int, Worm*>& worms, std::map<int, Team*> &,Wind* wind, si
ze_t);
        void printTeams(void);
        int getTeamTurn(void);
        int getWormTurn(int);
        void start(unsigned int);
        int nextTurn(void);
        void update(unsigned int);
        bool finished(void);
        int getWinner(void);
        int getTurnTimeleft(void);
        std::map<size_t, int> getTeamInfo(void);
        void setAliveProjectilesFlag(bool);
        void setMovingWormsFlag(bool);
        void setWormsAffectedByExplosion(bool);
        void setProtagonicWormGotHurt(bool);
        void setProtagonicWormDidShoot(bool);
        int getTeamTotalLife(size_t);
        int getWindForce();
        bool extraTime();
        void removePlayer(size_t);
};

#endif
```

jun 25, 18 20:09 **Missil.cpp** Page 1/2

```
#include "Missil.h"
#include <iostream>

Missil::Missil(int id, b2World& world, float posX, float posY, weapon_t type) :
Weapon(type),
world(world) {
    b2BodyDef missilDef;
    missilDef.type = b2_dynamicBody;
    missilDef.position.Set(posX, posY);
    b2Body* body = world.CreateBody(&missilDef);
    body->SetUserData(this);

    b2PolygonShape missilShape;
    missilShape.SetAsBox(MISSIL_WIDTH/2, MISSIL_HEIGHT/2);

    b2FixtureDef missilFixture;
    missilFixture.shape = &missilShape;
    missilFixture.density = 1;
    missilFixture.friction = 0.3;

    body->CreateFixture(&missilFixture);

    this->body = body;
    this->blast_radius = gConfiguration.AIR_STRIKE_BLAST_RADIUS;
    this->blast_power = gConfiguration.AIR_STRIKE_BLAST_POWER;
    this->exploded = false;
    this->id = id;
    this->wind_affected = true;
}

Missil::~Missil() {
    this->world.DestroyBody(this->body);
}

void Missil::explode() {
    ExplosionManager explosionManager(this->world);
    b2Vec2 center = this->body->GetPosition();
    explosionManager.manageExplosion(center, blast_radius, blast_power);
    this->exploded = true;
}

float Missil::getPosX() {
    return this->body->GetPosition().x;
}

float Missil::getPosY() {
    return this->body->GetPosition().y;
}

void Missil::update(int currentTime, int wind_force) {
    if (this->body->GetPosition().y > gConfiguration.WORLD_Y_LIMIT /*|| contact*
/) {
        this->explode();
    }

    if (wind_affected) {
        this->body->ApplyForce(body->GetMass() * b2Vec2(wind_force, 0), body->Ge
tWorldCenter(), true);
    }

    b2Vec2 mov_speed = this->body->GetLinearVelocity();
```

```
    if (round(mov_speed.x) == 0) {
        if (mov_speed.y > 0) {
            this->direction_angle = 180;
            return;
        }

        if (mov_speed.y < 0) {
            this->direction_angle = 0;
            return;
        }
    }

    if (round(mov_speed.y) == 0) {
        if (mov_speed.x > 0) {
            this->direction_angle = 90;
            return;
        }

        if (mov_speed.x < 0) {
            this->direction_angle = 270;
            return;
        }
    }

    int ang = atan(mov_speed.x/mov_speed.y) * gConfiguration.RADTODEG;

    // Primer cuadrante
    if (mov_speed.y < 0 && mov_speed.x > 0) {
        this->direction_angle = -ang;
    }

    // Segundo cuadrante
    if (mov_speed.y < 0 && mov_speed.x < 0) {
        this->direction_angle = 360 - ang;
    }

    // Tercer cuadrante
    if (mov_speed.y > 0 && mov_speed.x < 0) {
        this->direction_angle = 180 - ang;
    }

    // Cuarto cuadrante
    if (mov_speed.y > 0 && mov_speed.x > 0) {
        this->direction_angle = 180 - ang;
    }
}

// void Missil::setContact(bool made_contact) {
//     this->contact = made_contact;
// }

bool Missil::isMoving() {
    b2Vec2 speed = this->body->GetLinearVelocity();
    if (!speed.x && !speed.y) return false;
    return true;
}
```



```
#ifndef MISSIL_H
#define MISSIL_H

#include "Weapon.h"
#include "Box2D.h"
#include "types.h"
#include "ExplosionManager.h"
#include "Configuration.h"

#define MISSIL_WIDTH 0.2f
#define MISSIL_HEIGHT 0.7f

class Missil : public Weapon {
private:
    int blast_power;
    b2World& world;
    b2Body* body;
public:
    Missil(int id, b2World& world, float posX, float posY, weapon_t type);
    ~Missil();
    void explode();
    float getPosX();
    float getPosY();
    bool isMoving();
    entity_t getEntityType() {return MISSIL;}
    void update(int currentTime, int wind_force);
};

#endif
```

```cpp
#include "Mortar.h"
#include "types.h"

Mortar::Mortar(int id, b2World& world, float posX, float posY, float mirrored, f
loat shooting_angle, int power_factor, weapon_t type) :
Bazooka(id, world, posX, posY, mirrored, shooting_angle, power_factor, type) {

}

int Mortar::addProjectiles(std::map<int, Weapon*> & weapons) {
    for (int i = 1; i < gConfiguration.MORTAR_FRAGMENT_QUANTITY; ++i) {
        Fragment* fragment = new Fragment(this->id + i,
        this->world,
        getPosX(),
        getPosY(),
        w_air_strike);
        weapons.insert(std::pair<int, Weapon*>(fragment->getId(), fragment));
    }
    return gConfiguration.MORTAR_FRAGMENT_QUANTITY;
}
```



```cpp
#ifndef MORTAR_H
#define MORTAR_H

#include "Weapon.h"
#include "Box2D.h"
#include "types.h"
#include "Bazooka.h"
#include "Configuration.h"
#include "Fragment.h"

class Mortar : public Bazooka {
public:
    Mortar(int id, b2World& world, float posX, float posY, float mirrored, float
 shooting_angle, int power_factor, weapon_t type);
    int addProjectiles(std::map<int, Weapon*> & weapons);
};

#endif
```

```
#ifndef PHYSIC_ENTITY_H
#define PHYSIC_ENTITY_H

#include "Box2D.h"

enum _entityCategory {
    WORM_PHYSIC = 0x0001,
    STRUCTURE_PHYSIC = 0x0002,
    WATER_PHYSIC = 0x0004,
    BAT_PHYSIC = 0x0008,
    BAZOOKA_PHYSIC = 0x0010,
};

class PhysicEntity {
public:
private:
};

#endif
```



```
#include "protected_waiting_games.h"
#include <mutex>
#include <map>
#include <string>
#include "waiting_game.h"
#include "yaml.h"

ProtectedWaitingGames::ProtectedWaitingGames(void) {

}

void ProtectedWaitingGames::addNewWaitingGame(std::string & creator_name, Waitin
gGame * new_waiting_game) {
    std::lock_guard<std::mutex> lck(this->mutex);
    this->waiting_games[creator_name] = new_waiting_game;
}

YAML::Node ProtectedWaitingGames::getGamesInfoNode(void) {
    std::lock_guard<std::mutex> lck(this->mutex);
    YAML::Node waiting_games;
    std::map<std::string, WaitingGame*>::const_iterator it;
    for (it = this->waiting_games.begin(); it != this->waiting_games.end(); it++
) {
        YAML::Node a_waiting_game_node;
        a_waiting_game_node["match_name"] = it->second->getMatchName();
        a_waiting_game_node["creator"] = it->second->getCreatorName();
        a_waiting_game_node["required_players"] = it->second->getPlayersQty();
        a_waiting_game_node["joined_players"] = it->second->getJoinedPlayersQty();
        waiting_games["waiting_games"].push_back(a_waiting_game_node);
    }
    return waiting_games;
}

std::string ProtectedWaitingGames::getGameName(std::string & creator_name) {
    std::lock_guard<std::mutex> lck(this->mutex);
    return this->waiting_games[creator_name]->getMatchName();
}

void ProtectedWaitingGames::removeGame(std::string & creator_name) {
    std::lock_guard<std::mutex> lck(this->mutex);
    delete this->waiting_games[creator_name];
    this->waiting_games.erase(creator_name);
}

bool ProtectedWaitingGames::gameHasFreeSlots(std::string & creator_name) {
    std::lock_guard<std::mutex> lck(this->mutex);
    return this->waiting_games[creator_name]->hasFreeSlots();
}

void ProtectedWaitingGames::addPlayerToGame(std::string & creator_name, Client *
 new_player) {
    std::lock_guard<std::mutex> lck(this->mutex);
    this->waiting_games[creator_name]->addPlayer(new_player);
}

void ProtectedWaitingGames::rmvPlayerFromGame(std::string & creator_name, std::s
tring & rm_player_name) {
    std::lock_guard<std::mutex> lck(this->mutex);
    this->waiting_games[creator_name]->rmPlayer(rm_player_name);
}

std::vector<std::string> ProtectedWaitingGames::getWaitingPlayers(std::string &
```

```
creator_name) {
    std::lock_guard<std::mutex> lck(this->mutex);
    return this->waiting_games[creator_name]->getWaitingPlayersName();
}

void ProtectedWaitingGames::notifyAllStartGame(std::string & creator_name) {
    std::lock_guard<std::mutex> lck(this->mutex);
    this->waiting_games[creator_name]->notifyAllStartGame();
}

void ProtectedWaitingGames::notifyAllCancellGame(std::string & creator_name) {
    std::lock_guard<std::mutex> lck(this->mutex);
    this->waiting_games[creator_name]->notifyAllCancellGame();
}

void ProtectedWaitingGames::startWaitingGame(std::string & creator_name, std::st
ring & map_path) {
    this->waiting_games[creator_name]->startGame(map_path);
}

void ProtectedWaitingGames::waitGameUntilFinish(std::string & creator_name) {
    this->waiting_games[creator_name]->waitUntilFinish();
}
```



```
#ifndef __PROTECTED_WAITING_GAMES_H__
#define __PROTECTED_WAITING_GAMES_H__

#include <mutex>
#include <map>
#include <string>
#include <vector>
#include <string>
#include "waiting_game.h"
#include "yaml.h"

class ProtectedWaitingGames {
    private:
        std::mutex mutex;
        std::map<std::string, WaitingGame*> waiting_games;
    public:
        ProtectedWaitingGames(void);
        void addNewWaitingGame(std::string &, WaitingGame *);
        YAML::Node getGamesInfoNode(void);
        std::string getGameName(std::string &);
        void removeGame(std::string &);
        bool gameHasFreeSlots(std::string &);
        void addPlayerToGame(std::string &, Client *);
        void rmvPlayerFromGame(std::string &, std::string &);
        std::vector<std::string> getWaitingPlayers(std::string &);
        void notifyAllStartGame(std::string &);
        void notifyAllCancellGame(std::string &);
        void startWaitingGame(std::string &, std::string &);
        void waitGameUntilFinish(std::string &);
};

#endif
```

jun 25, 18 20:09 **QueryCallback.cpp** Page 1/1

```cpp
#include "QueryCallback.h"

bool QueryCallback::ReportFixture(b2Fixture* fixture) {
    foundBodies.push_back(fixture->GetBody());
    return true;
}
```

jun 25, 18 20:09 **QueryCallback.h** Page 1/1

```cpp
#ifndef QUERY_CALLBACK_H
#define QUERY_CALLBACK_H

#include <vector>
#include "Box2D.h"

class QueryCallback : public b2QueryCallback {
public:
    std::vector<b2Body*> foundBodies;
    bool ReportFixture(b2Fixture* fixture);
};

#endif
```

jun 25, 18 20:09 **RayCastClosestCallBack.cpp** Page 1/1

```cpp
#include "RayCastClosestCallBack.h"

RayCastClosestCallBack::RayCastClosestCallBack() {
    this->body = NULL;
}

float32 RayCastClosestCallBack::ReportFixture(b2Fixture* fixture, const b2Vec2&
point, const b2Vec2& normal, float32 fraction) {
    body = fixture->GetBody();
    this->point = point;
    return fraction;
}
```

jun 25, 18 20:09 **RayCastClosestCallBack.h** Page 1/1

```cpp
#ifndef RAYCAST_CLOSEST_CALL_BACK_H
#define RAYCAST_CLOSEST_CALL_BACK_H

#include "Box2D.h"

class RayCastClosestCallBack : public b2RayCastCallback {
public:
    RayCastClosestCallBack ();
    b2Body* body;
    b2Vec2 point;

    float32 ReportFixture(b2Fixture* fixture, const b2Vec2& point, const b2Vec2&
 normal, float32 fraction);
};

#endif
```

```cpp
#include "RedGrenade.h"
#include "types.h"

RedGrenade::RedGrenade(int id, b2World& world, float posX, float posY, bool mirr
ored, float shooting_angle, int power_factor, int delay, int currentTime, weapon
_t type) :
Grenade(id, world, posX, posY, mirrored, shooting_angle, power_factor, delay, cu
rrentTime, type) {
}

int RedGrenade::addProjectiles(std::map<int, Weapon*> & weapons) {
    for (int i = 1; i < gConfiguration.RED_GRENADE_FRAGMENT_QUANTITY + 1; ++i) {
        Fragment* fragment = new Fragment(this->id+i,
        this->world,
        getPosX(),
        getPosY(),
        w_air_strike);
        weapons.insert(std::pair<int, Weapon*>(fragment->getId(), fragment));
    }
    return gConfiguration.RED_GRENADE_FRAGMENT_QUANTITY;
}
```



```cpp
#ifndef RED_GRENADE_H
#define RED_GRENADE_H

#include "Weapon.h"
#include "Box2D.h"
#include "types.h"
#include "Configuration.h"
#include "Grenade.h"
#include "ExplosionManager.h"
#include <list>
#include "Fragment.h"
#include <map>

class RedGrenade : public Grenade {
public:
    RedGrenade(int id, b2World& world, float posX, float posY, bool mirrored, fl
oat shooting_angle, int power_factor, int delay, int currentTime, weapon_t type)
;
    int addProjectiles(std::map<int, Weapon*> & weapons);
};

#endif
```

```
#include <fstream>
#include <map>
#include <sstream>
#include "server.h"
#include "socket.h"
#include "socket_error.h"
#include "protocol.h"
#include "server_error.h"
#include "client.h"
#include "lobby_attendant.h"

Server::Server(std::string & port) :
skt(port) {
    this->keep_running = true;
}

void Server::run(void) {
    while (1) {
        try {
            Protocol newsktprotocol(std::move(this->skt.accept_connection()));
            std::string player_name;
            newsktprotocol.getPlayerName(player_name);

            cleanLobby();
            cleanQuitedClients();

            if (this->clients.find(player_name) != this->clients.end()) {
                player_name = findFreeName(player_name);
            }

            std::cout << "Bautizando al cliente como " << player_name << std::endl;
            newsktprotocol.sendName(player_name);

            YAML::Node match_status = this->protected_waiting_games.getGamesInfo
Node();
            newsktprotocol.sendGameStatus(match_status);

            Client * client = new Client(std::move(newsktprotocol), player_name)
;
            this->clients.insert(std::pair<std::string, Client*>(player_name, cl
ient));
            LobbyAttendant * new_lobby_attendant = new LobbyAttendant(client, th
is->protected_waiting_games);
            new_lobby_attendant->start();
            this->clients_in_lobby.insert(std::pair<std::string, LobbyAttendant*
>(player_name, new_lobby_attendant));
        } catch(const SocketError & e) {
            std::cout << "Server acceptor se detiene por cierre del socket listener." << std::endl;
            break;
        }
    }
}

void Server::cleanLobby(void) {
    std::map<std::string, LobbyAttendant*>::iterator it;
    for (it = this->clients_in_lobby.begin(); it != this->clients_in_lobby.end()
;) {
        if (!it->second->isRunning()) {
            it->second->join();
            std::cout << "Deleteando Lobby Attendant del cliente " << it->first << std::end
l;
            delete it->second;
            this->clients_in_lobby.erase(it++);
        } else {
            ++it;
        }
    }
}

void Server::cleanQuitedClients(void) {
    std::map<std::string, Client*>::iterator it;
    for (it = this->clients.begin(); it != this->clients.end();) {
        if (it->second->getStatus() == quited) {
            std::cout << "Deleteando al cliente " << it->first << " que salio del Lobby." << s
td::endl;
            delete it->second;
            this->clients.erase(it++);
        } else {
            ++it;
        }
    }
}

 bool Server::isRunning(void) const {
     return this->keep_running;
 }

Server::~Server(void) {
    // Cerrar server ordenadamente
    std::map<std::string, Client*>::iterator it;
    for (it = this->clients.begin(); it != this->clients.end(); it++) {
        delete it->second;
    }
}

size_t  Server::getId(void) const{
    return 0;
}

void Server::stop(void) {
    this->keep_running = false;
    // Destraba el accept del metodo run()
    this->skt.stopListening();
}

std::string Server::findFreeName(std::string & old_name) {
    int counter = 1;
    std::string number;
    std::string new_name;
    std::string tmp;
    while(1) {
        number.clear();
        number.append("-" + std::to_string(counter));
        tmp = old_name;
        if (this->clients.find(tmp.append(number)) == this->clients.end()) {
            new_name = tmp;
            break;
        }
        counter++;
    }
    return new_name;
}
```

```cpp
#include <iostream>
#include <string>
#include "server_error.h"

ServerError::ServerError(const std::string & msg) {
    this->msg = msg;
}

const char * ServerError::what() const noexcept {
    return this->msg.c_str();
}

ServerError::~ServerError() {
}
```



```cpp
#ifndef __SERVER_ERROR_H__
#define __SERVER_ERROR_H__

#include <exception>
#include <iostream>
#include <string>

/*
Clase para generar excepciones del tipo 'servidor'.
*/
class ServerError : public std::exception {
    private:
        std::string msg;
    public:
        explicit ServerError(const std::string &);
        virtual const char * what() const noexcept;
        virtual ~ServerError() noexcept;
};

#endif
```

```
#include <string>
#include <unistd.h>
#include "server_game.h"
#include "protocol.h"
#include "yaml.h"
#include "blocking_queue.h"
#include "World.h"
#include "match.h"
#include <fstream>
#include "snapshot_sender.h"
#include "event_receiver.h"

#define MAX_QUEUE_SNAPSHOTS 256
#define MAP_YML_NAME "map.yml"

ServerGame::ServerGame(std::vector<Client*> cl, std::string & map_path) :
clients(cl),
map_path(map_path) {
    std::vector<Client*>::const_iterator it;
    // Envio a todos los clientes el mapa del juego.
    std::fstream file_map(this->map_path, std::fstream::in | std::fstream::binar
y);
    for (it = this->clients.begin(); it != this->clients.end(); it++) {
        std::cout << "Iterando por clientes." << std::endl;
        if ((*it)->getIdInMatch() == 1) continue;
        std::cout << "EnviÃ¡ndole el mapa al jugador nro. " << (*it)->getIdInMatch() << std
::endl;
        (*it)->sendMapGame(file_map);
    }
    std::string map_yml_name(MAP_YML_NAME);
        std::string cmd_unzip_tar_gz = "tar -xf " + map_path;
        std::system(cmd_unzip_tar_gz.c_str());
}

ServerGame::~ServerGame(void) {
    removePreviousTempFiles();
}

void ServerGame::removePreviousTempFiles(void) {
        std::string cmd_rm_map_yml = "rm map.yml background.png " + this->map_path;
        std::system(cmd_rm_map_yml.c_str());
}

void ServerGame::startGame(void) {
    std::cout << "Inicio de server game." << std::endl;
    Queue<Snapshot*> snapshots(MAX_QUEUE_SNAPSHOTS);
    std::cout << "Creando world." << std::endl;
    std::string map_yml_name(MAP_YML_NAME);
    World world(map_yml_name, snapshots);
    std::cout << "Creando match." << std::endl;
    Match match(world.getWorms(), world.getTeams(), world.getWind(), gConfigurat
ion.TURN_DURATION);
    std::cout << "Creando snapshot sender." << std::endl;
    SnapshotSender snapshot_sender(snapshots, match, this->clients);

    std::vector<Client*>::const_iterator it;
    for (it = this->clients.begin(); it != this->clients.end(); it++) {
        this->event_receiver.push_back(new EventReceiver((*it), world, match, (*
it)->getIdInMatch()));
    }

    world.start();
```



```
    snapshot_sender.start();
    //Lanzo hilos de event receiver
    std::vector<EventReceiver*>::const_iterator it2;
    for (it2 = this->event_receiver.begin(); it2 != this->event_receiver.end();
it2++) {
        (*it2)->start();
    }

    std::cout << "Inicio de game loop de servidor." << std::endl;
    gameLoop(match, world);
    std::cout << "Fin de game loop de servidor." << std::endl;

    if (match.finished()) {
        std::cout << "La partida finalizÃ³ y el ganador es " << match.getWinner() << std::en
dl;
    }

    snapshot_sender.stop();
    //Stops y joins de los hilos lanzados
    world.stop();
    world.join();
    snapshot_sender.join();
    for (it2 = this->event_receiver.begin(); it2 != this->event_receiver.end();
it2++) {
        (*it2)->stop();
    }
    for (it2 = this->event_receiver.begin(); it2 != this->event_receiver.end();
it2++) {
        (*it2)->join();
    }

    std::cout << "Fin de server game." << std::endl;
}

void ServerGame::gameLoop(Match & match, World & world) {
    unsigned int timer = 0;
    match.start(world.getTimeSeconds());
    while(!match.finished()) {
        timer = world.getTimeSeconds();
        match.setAliveProjectilesFlag(world.hasAliveProjectiles());
        match.setMovingWormsFlag(world.hasWormsMoving());
        match.setWormsAffectedByExplosion(world.hasWormsAffectedByExplosion());
        match.setProtagonicWormGotHurt(world.hasWormGotHurt(match.getWormTurn(ma
tch.getTeamTurn())));
        match.setProtagonicWormDidShoot(world.hasWormShooted(match.getWormTurn(m
atch.getTeamTurn())));
        usleep(16666);
        match.update(world.getTimeSeconds() - timer);
        cleanClients(match);
    }
}

void ServerGame::cleanClients(Match & match) {
    std::vector<EventReceiver*>::iterator it;
    for (it = this->event_receiver.begin(); it != this->event_receiver.end();) {
        if (!(*it)->isRunning()) {
            (*it)->join();
            match.removePlayer((*it)->getId());
            it = this->event_receiver.erase(it);
            std::cout << "Jugador removido por quit." << std::endl;
        } else {
            it++;
```

```
        }
    }
}
```



```
#ifndef __SERVER_GAME_H__
#define __SERVER_GAME_H__

#include <vector>
#include "protocol.h"
#include "yaml.h"
#include "client.h"
#include "match.h"
#include "event_receiver.h"

class ServerGame {
    private:
        std::vector<Client*> clients;
        //std::map<size_t, Protocol*> protocols;
        //YAML::Node mapNode;
        std::string map_path;
        std::vector<EventReceiver*> event_receiver;
        void gameLoop(Match &, World &);
        void removePreviousTempFiles(void);
        void cleanClients(Match &);

    public:
        ServerGame(std::vector<Client*>, std::string &);
        ~ServerGame(void);
        void startGame(void);
};

#endif
```

```
#ifndef __SERVER_H__
#define __SERVER_H__
#include <string>
#include <map>
#include "thread.h"
#include "protocol.h"
#include "socket.h"
#include "client.h"
#include "lobby_attendant.h"
#include "waiting_game.h"
#include "protected_waiting_games.h"

#define MSG_CANT_OPEN_CFG_FILE "No se pudo abrir el archivo de configuracion."

class Server : public Thread {
    private:
        SocketListener skt;
        std::map<std::string, Client*> clients;
        std::map<std::string, LobbyAttendant*> clients_in_lobby;
        ProtectedWaitingGames protected_waiting_games;
        std::map<std::string, WaitingGame*> waiting_games;
        bool keep_running;

        bool isRunning(void) const;
        size_t getId(void) const;
        std::string findFreeName(std::string &);
        void cleanLobby(void);
        void cleanQuitedClients(void);
    public:
        void stop(void);
        Server(std::string & port);
        virtual void run(void);
        ~Server(void);
};

#endif
```



```
#include <string>
#include <vector>
#include "snapshot.h"
#include "yaml.h"
#include "World.h"

Snapshot::Snapshot() {
}

void Snapshot::updateTeams(std::map<int, Team*> & teams) {
    snapshot << YAML::BeginMap;
    snapshot << YAML::Key<< "worms_teams";
    snapshot << YAML::Value << YAML::BeginMap;

    std::map<int, Team*>::const_iterator teamss_it;
    for (teamss_it = teams.begin(); teamss_it != teams.end(); ++teamss_it) {

        int team_id = teamss_it->second->getTeamId();
        std::map<int, Worm*> & worms = teamss_it->second->getWorms();
        std::map<int, Worm*>::const_iterator worm_it;


        snapshot << YAML::Key << (int) team_id;
        snapshot << YAML::Value << YAML::BeginMap;
        snapshot << YAML::Key << "worms";
        snapshot << YAML::Value << YAML::BeginSeq;

        for (worm_it = worms.begin(); worm_it != worms.end(); ++worm_it) {
            Worm* worm = worm_it->second;
            snapshot << YAML::BeginMap;
            snapshot << YAML::Key << "id" << YAML::Value << worm->getId();
            //snapshot << YAML::Key << "name" << YAML::Value << worm->getName();
            snapshot << YAML::Key << "health" << YAML::Value << worm->getHealth()
;
            snapshot << YAML::Key << "x" << YAML::Value << (int)(worm->getPosX()
 / gConfiguration.SCALING_FACTOR);
            snapshot << YAML::Key << "y" << YAML::Value << (int)(worm->getPosY()
 / gConfiguration.SCALING_FACTOR);
            snapshot << YAML::Key << "status";
            snapshot << YAML::Value << YAML::BeginMap;
            snapshot << YAML::Key << "grounded" << YAML::Value << (int) worm->isG
rounded();
            snapshot << YAML::Key << "falling" << YAML::Value << (int) worm->isFal
ling();
            snapshot << YAML::Key << "mirrored" << YAML::Value << (int) worm->isMi
rrored();
            snapshot << YAML::Key << "walking" << YAML::Value << (int) worm->isWa
lking();
            snapshot << YAML::Key << "inclination" << YAML::Value << (int) worm->ge
tInclination();
            snapshot << YAML::Key << "affected_by_explosion" << YAML::Value << (int)
worm->isAffectedByExplosion();
            snapshot << YAML::Key << "angle_direction" << YAML::Value <<    worm->ge
tDirectionAngle();
            snapshot << YAML::EndMap;
            snapshot << YAML::EndMap;
        }
        snapshot << YAML::EndSeq;

        std::map<weapon_t, int> inventory = teamss_it->second->getInventory();
        std::map<weapon_t, int>::const_iterator inventory_it;
```

```
        snapshot << YAML::Key << "inventory";
        snapshot << YAML::Value << YAML::BeginMap;

    for (inventory_it = inventory.begin(); inventory_it != inventory.end(); ++in
ventory_it) {
        snapshot << YAML::Key << inventory_it->first;
        snapshot << YAML::Value <<YAML::BeginMap;
        snapshot << YAML::Key << "supplies" << YAML::Value << inventory_it->second
;
        snapshot << YAML::EndMap;
    }
    snapshot << YAML::EndMap;
    snapshot << YAML::EndMap;
    }

    snapshot << YAML::EndMap;
}

void Snapshot::updateProjectiles(std::map<int, Weapon*> & weapons) {
    snapshot << YAML::Key << "projectiles";
    snapshot << YAML::Value << YAML::BeginSeq;

    for (std::map <int, Weapon*>::iterator it = weapons.begin(); it != weapons.e
nd(); ++it) {
        Weapon* weapon = it->second;
        snapshot << YAML::BeginMap;
        snapshot << YAML::Key << "id" << YAML::Value << weapon->getId();
        snapshot << YAML::Key << "type" << YAML::Value << std::to_string(weapon->
getType());
        snapshot << YAML::Key << "x" << YAML::Value << (int) (weapon->getPosX()
/  gConfiguration.SCALING_FACTOR);
        snapshot << YAML::Key << "y" << YAML::Value << (int) (weapon->getPosY()
/  gConfiguration.SCALING_FACTOR);
        snapshot << YAML::Key << "countdown" << YAML::Value << weapon->getCountdo
wn();
        snapshot << YAML::Key << "exploded" << YAML::Value << (int) weapon->hasEx
ploded();
        snapshot << YAML::Key << "blast_radius" << YAML::Value << (int) (weapon->ge
tBlastRadius() / gConfiguration.SCALING_FACTOR);
        snapshot << YAML::Key << "moving" << YAML::Value << (int) (weapon->isMov
ing());
        snapshot << YAML::Key << "angle_direction" << YAML::Value << weapon->getDire
ctionAngle();
        snapshot << YAML::EndMap;
    }
    snapshot << YAML::EndSeq;
}

void Snapshot::updateGameStatus(Match & match) {
    snapshot << YAML::Key << "game_status";
    snapshot << YAML::Value << YAML::BeginMap;
    snapshot << YAML::Key << "teams_health" << YAML::Value << match.getTeamInfo();
    snapshot << YAML::Key << "wind_force" << YAML::Value << match.getWindForce();
    snapshot << YAML::Key << "team_turn" << YAML::Value << match.getTeamTurn();
    snapshot << YAML::Key << "protagonic_worm" << YAML::Value << match.getWormTurn(
match.getTeamTurn());
    snapshot << YAML::Key << "turn_timeleft" << YAML::Value << std::to_string(match.
getTurnTimeleft());
    snapshot << YAML::Key << "finished" << YAML::Value << std::to_string(match.fin
ished());
    snapshot << YAML::EndMap;
}
```



```
const char* Snapshot::getSnapshot() {
    return this->snapshot.c_str();
}

void Snapshot::updateGameStatusLastSnapshot(Match & match) {
    snapshot << YAML::Key << "game_status";
    snapshot << YAML::Value << YAML::BeginMap;
    snapshot << YAML::Key << "teams_health" << YAML::Value << match.getTeamInfo();
    snapshot << YAML::Key << "wind_force" << YAML::Value << match.getWindForce();
    snapshot << YAML::Key << "protagonic_worm" << YAML::Value << match.getWormTurn(
match.getTeamTurn());
    snapshot << YAML::Key << "turn_timeleft" << YAML::Value << std::to_string(match.
getTurnTimeleft());
    snapshot << YAML::Key << "finished" << YAML::Value << std::to_string(1);
    snapshot << YAML::EndMap;
}
```

jun 25, 18 20:09 **snapshot.h** Page 1/1

```
#ifndef __SNAPSHOT_H__
#define __SNAPSHOT_H__

#include <map>
#include "Worm.h"
#include "yaml.h"
#include "match.h"
#include "Weapon.h"
#include "Configuration.h"

class Snapshot {
    private:
        YAML::Emitter snapshot;
    public:
        Snapshot();
        void updateTeams(std::map<int, Team*> & teams);
        void updateProjectiles(std::map<int, Weapon*> &);
        void updateGameStatus(Match &);
        void updateGameStatusLastSnapshot(Match &);
        const char* getSnapshot();
        void updateGameStatusLastSnapshot();
};

#endif
```

jun 26, 18 12:27 **snapshot_sender.cpp** Page 1/2

```
#include "snapshot.h"
#include "snapshot_sender.h"
#include "socket_error.h"
#include <iostream>
#include <unistd.h>
#include "client.h"

SnapshotSender::SnapshotSender(Queue<Snapshot*> & snapshots, Match & m, std::vec
tor<Client*> cl) :
snapshots(snapshots) ,
match(m),
clients(cl) {
    this->keep_running = true;
}

SnapshotSender::~SnapshotSender() {
}

void SnapshotSender::run() {
    while (keep_running) {
        usleep(16666);
        Snapshot* snapshot = this->snapshots.pop();
        if (keep_running && snapshot) {
            sendSnapshot(snapshot);
        }
    }
    std::cout << "Sacando la ultima foto con partida finished." << std::endl;
    Snapshot* snapshot = this->snapshots.pop();
    sendLastSnapshot(snapshot);
}

void SnapshotSender::sendSnapshot(Snapshot * snapshot) {
    snapshot->updateGameStatus(this->match);
    std::stringstream ss;
    ss << snapshot->getSnapshot();
    std::vector<Client*>::const_iterator it;
    for (it = this->clients.begin(); it != this->clients.end(); it++) {
        (*it)->sendSnapShot(ss);
    }

    delete snapshot;
}

void SnapshotSender::sendLastSnapshot(Snapshot * last_snapshot) {
    last_snapshot->updateGameStatusLastSnapshot(this->match);
    std::stringstream ss;
    ss << last_snapshot->getSnapshot();
    std::cout << ss.str() << std::endl;
    std::vector<Client*>::const_iterator it;
    for (it = this->clients.begin(); it != this->clients.end(); it++) {
        std::cout << "Enviando ultima snapshot a cliente." << std::endl;
        (*it)->sendSnapShot(ss);
    }

    delete last_snapshot;
}

size_t SnapshotSender::getId(void) const{
    return 0;
}

bool SnapshotSender::isRunning(void) const {
```

```
    return this->keep_running;
}

void SnapshotSender::stop() {
    this->keep_running = false;
}
```



```
#ifndef SNAPSHOT_SENDER_H
#define SNAPSHOT_SENDER_H

#include "snapshot.h"
#include "thread.h"
#include <vector>
#include <string>
#include "protocol.h"
#include "blocking_queue.h"
#include "yaml.h"
#include "match.h"
#include "client.h"

class SnapshotSender : public Thread {
    private:
        Queue<Snapshot*> & snapshots;
        Match & match;
        std::vector<Client*> clients;
        bool keep_running;

        virtual bool isRunning(void) const;
        virtual size_t getId(void) const;
        void sendSnapshot(Snapshot *);
        void sendLastSnapshot(Snapshot *);

    public:
        SnapshotSender(Queue<Snapshot*> &, Match &, std::vector<Client*>);
        ~SnapshotSender();
        virtual void run(void);
        void stop();
};

#endif
```

```
#include <iostream>
#include <map>
#include "team.h"
#include "Worm.h"
#include "types.h"

Team::Team(int id) {
    this->team_id = id;
    this->member_qty = 0;
}

void Team::addMember(Worm * worm) {
    this->worms[worm->getId()] = worm;
    this->member_qty++;
}

void Team::print(void) const {
    std::map<int, Worm*>::const_iterator it;
    size_t i = 1;
    std::cout << "**********************" << std::endl;
    for (it = this->worms.begin(); it != this->worms.end(); it++) {
        std::cout << "Equipo ID: " << this->team_id << std::endl;
        std::cout << "Miembro " << i++ << std::endl;
        std::cout << "Nombre: " << it->second->getName() << std::endl;
        std::cout << "Vida: " << it->second->getHealth() << std::endl;
        std::cout << "**********" << std::endl;
    }
    std::cout << "**********************" << std::endl;
}

void Team::initializeInventory(YAML::Node inventory_node) {
    YAML::Node::const_iterator it;
    for (it = inventory_node.begin(); it != inventory_node.end(); it++) {
        weapon_t weapon_id = (weapon_t) it->first.as<int>();
        int supplies = it->second["supplies"].as<int>();
        this->inventory.insert(std::pair<weapon_t,int>(weapon_id, supplies));
    }
}

std::vector<int> Team::getWormsID(void) {
    std::vector<int> ids;
    std::map<int, Worm*>::const_iterator it;
    for (it = this->worms.begin(); it != this->worms.end(); it++) {
        ids.push_back(it->second->getId());
    }
    return ids;
}

int Team::getTeamId(void) {
    return this->team_id;
}

Team::~Team(void) {
}

bool Team::haveAliveMember(void) {
    std::map<int, Worm*>::const_iterator it;
    for (it = this->worms.begin(); it != this->worms.end(); it++) {
        if (!it->second->isDead()) {
            return true;
        }
    }
```



```
    return false;
}

int Team::getTotalLife(void) {
    int counter = 0;
    std::map<int, Worm*>::const_iterator it;
    for (it = this->worms.begin(); it != this->worms.end(); it++) {
        counter += it->second->getHealth();
    }
    return counter;
}

std::map<int, Worm*> & Team::getWorms() {
    return this->worms;
}

std::map<weapon_t, int> & Team::getInventory() {
    return this->inventory;
}

void Team::killAll(void) {
    std::map<int, Worm*>::iterator it;
    for (it = this->worms.begin(); it != this->worms.end(); it ++) {
        it->second->kill();
    }
}

bool Team::hasSupplies(weapon_t weapon_id) {
    return this->inventory[weapon_id] > 0;
}

void Team::reduceSupplie(weapon_t weapon_id) {
    this->inventory[weapon_id] -= 1;
}
```

jun 25, 18 20:09 **team.h** Page 1/1

```
#ifndef __TEAM_H__
#define __TEAM_H__

#include <map>
#include <vector>
#include "Worm.h"
#include "yaml.h"
#include "types.h"

class Team {
    private:
        int team_id;
        int member_qty;
        std::map<int, Worm*> worms;
        std::map<weapon_t, int> inventory;

    public:
        Team(int);
        ~Team(void);
        std::map<int, Worm*>  & getWorms();
        void addMember(Worm *);
        void initializeInventory(YAML::Node inventory_node);
        std::map<weapon_t, int> & getInventory();
        std::vector<int> getWormsID(void);
        void print(void) const;
        int getTeamId(void);
        bool haveAliveMember(void);
        int getTotalLife(void);
        void killAll(void);
        bool hasSupplies(weapon_t);
        void reduceSupplie(weapon_t);
};

#endif
```

jun 25, 18 20:09 **Teleportation.cpp** Page 1/1

```
#include "Teleportation.h"
#include "types.h"
#include <iostream>

Teleportation::Teleportation(Worm* worm, float posX, float posY) :
posX(posX),
posY(posY) {
    this->worm = worm;

}
void Teleportation::teleport() {
    this->worm->setPosition(posX, posY);
    //this->worm->addFootContact();
}
```

jun 25, 18 20:09 **Teleportation.h** Page 1/1

```
#ifndef TELEPORT_H
#define TELEPORT_H

#include "Weapon.h"
#include "Worm.h"
#include "types.h"

class Teleportation {
private:
    Worm* worm;
    float posX;
    float posY;

public:
    Teleportation(Worm* worm, float posX, float posY);
    void teleport(void);
};

#endif
```

jun 26, 18 12:27 **waiting_game.cpp** Page 1/2

```
#include <string>
#include "waiting_game.h"
#include "client.h"
#include <unistd.h>
#include "server_game.h"

WaitingGame::WaitingGame(Client * cn, std::string & mn, size_t pq) {
    this->members.push_back(cn);
    this->match_name = mn;
    this->players_qty = pq;
    this->joined_players = 1;
    this->finished = false;
}

void WaitingGame::addPlayer(Client * new_member) {
    this->members.push_back(new_member);
    this->joined_players++;
}

std::string WaitingGame::getCreatorName(void) {
    return this->members.front()->getPlayerName();
}

std::string WaitingGame::getMatchName(void) {
    return this->match_name;
}

size_t WaitingGame::getPlayersQty(void) {
    return this->players_qty;
}

bool WaitingGame::hasFreeSlots(void) {
    return (this->players_qty - this->joined_players) > 0;
}

size_t WaitingGame::getJoinedPlayersQty(void) {
    return this->joined_players;
}

void WaitingGame::rmPlayer(std::string & player_name) {
    std::vector<Client *>::iterator it;
    for (it = this->members.begin(); it != this->members.end(); it++) {
        if ((*it)->getPlayerName() == player_name) {
            this->members.erase(it);
            this->joined_players--;
            return;
        }
    }
}

std::vector<std::string> WaitingGame::getWaitingPlayersName(void) {
    std::vector<std::string> players_names;
    std::vector<Client *>::iterator it;
    for (it = this->members.begin(); it != this->members.end(); it++) {
        players_names.push_back((*it)->getPlayerName());
    }
    return players_names;
}

void WaitingGame::notifyAllStartGame(void) {
    std::vector<Client*>::iterator it;
    std::string msg = "started";
```

```
    size_t team_id = 1;
    for (it = this->members.begin(); it != this->members.end(); it++) {
        std::string tid = std::to_string(team_id);
        (*it)->setIdInMatch(team_id);
        (*it)->sendGameStart(1, msg, tid);
        team_id++;
    }
}

void WaitingGame::notifyAllCancellGame(void) {
    std::vector<Client*>::iterator it;
    std::string msg = "aborted";
    for (it = this->members.begin(); it != this->members.end(); it++) {
        if ((*it)->getPlayerName() == this->getCreatorName()) continue;
        (*it)->sendResponse(0, msg);
    }
}

void WaitingGame::startGame(std::string & map_path) {
    std::unique_lock<std::mutex> lock(this->mutex);
    std::cout << "Iniciando partida." << std::endl;
    ServerGame new_server_game(members, map_path);
    new_server_game.startGame();
    this->finished = true;
    this->cv.notify_all();
    std::cout << "Partida finalizada" << std::endl;
}

bool WaitingGame::hasFinished(void) {
    return this->finished;
}

void WaitingGame::waitUntilFinish(void) {
    std::cout << "Esperando a que termine la partida." << std::endl;
    std::unique_lock<std::mutex> lock(this->mutex);
    while (!this->finished) {
        this->cv.wait(lock);
    }
    std::cout << "La partida termino, ya no esperarÃ© mas." << std::endl;
}
```



```
#ifndef __WAITING_GAME_H__
#define __WAITING_GAME_H__

#include <string>
#include <vector>
#include <mutex>
#include <condition_variable>
#include "client.h"

class WaitingGame {
    private:
        std::vector<Client*> members;
        std::string match_name;
        size_t players_qty;
        size_t joined_players;
        std::mutex mutex;
        std::condition_variable cv;
        bool finished;

    public:
        WaitingGame(Client *, std::string &, size_t);
        std::string getCreatorName(void);
        std::string getMatchName(void);
        size_t getPlayersQty(void);
        size_t getJoinedPlayersQty(void);
        bool hasFreeSlots(void);
        void addPlayer(Client *);
        void rmPlayer(std::string &);
        std::vector<std::string> getWaitingPlayersName(void);
        void notifyAllStartGame(void);
        void notifyAllCancellGame(void);
        void startGame(std::string &);
        bool hasFinished(void);
        void waitUntilFinish(void);
};

#endif
```

jun 25, 18 20:09 **Wall.cpp** Page 1/1

```
#include "Wall.h"

Wall::Wall(b2World& world, float posX, float posY, float width, float height) :
world(world) {
    b2BodyDef wallDef;
    wallDef.type = b2_staticBody;
    wallDef.position.Set(posX, posY);
    wallDef.allowSleep = true;
    b2Body* body = world.CreateBody(&wallDef);
    body->SetAwake(false);
    body->SetUserData(this);

    b2PolygonShape wallShape;
    wallShape.SetAsBox(width/2, height/2);

    b2FixtureDef wallFixture;
    wallFixture.shape = &wallShape;
    wallFixture.density = 1;
    wallFixture.friction = 0.3;
    wallFixture.filter.categoryBits = STRUCTURE_PHYSIC;
    wallFixture.filter.maskBits = WORM_PHYSIC;

    body->CreateFixture(&wallFixture);
    this->body = body;
}

Wall::~Wall() {
    this->world.DestroyBody(this->body);
}

void Wall::update() {
    this->body->SetAwake(false);
}

float Wall::getPosX() {
    return this->body->GetPosition().x;
}

float Wall::getPosY() {
    return this->body->GetPosition().y;
}
```

jun 25, 18 20:09 **Wall.h** Page 1/1

```
#ifndef WALL_H
#define WALL_H

#include "Box2D.h"
#include "Entity.h"
#include "PhysicEntity.h"

class Wall : public Entity {
private:
    b2World& world;
    b2Body* body;
public:
    Wall(b2World& world, float posX, float posY, float width, float height);
    virtual ~Wall();
    entity_t getEntityType() {return WALL;}
    void update(void);
    float getPosX(void);
    float getPosY(void);
};

#endif
```

jun 25, 18 20:09 **Water.cpp** Page 1/1

```
#include "Water.h"

Water::Water(b2World& world, float posX, float posY, float width, float height)
:
world(world) {
    b2BodyDef waterDef;
    waterDef.type = b2_staticBody;
    waterDef.position.Set(posX, posY);
    waterDef.allowSleep = true;
    b2Body* body = world.CreateBody(&waterDef);
    body->SetUserData(this);
    body->SetAwake(false);
    b2PolygonShape waterShape;
    waterShape.SetAsBox(width/2, height/2);

    b2FixtureDef waterFixture;
    waterFixture.shape = &waterShape;
    waterFixture.density = 1;
    waterFixture.friction = 1;
    waterFixture.restitution = 0;
    waterFixture.filter.categoryBits = WATER_PHYSIC;
    waterFixture.filter.maskBits = WORM_PHYSIC;
    body->CreateFixture(&waterFixture);

    this->body = body;
}

Water::~Water() {
    this->world.DestroyBody(this->body);
}

void Water::update() {
    this->body->SetAwake(false);
}

float Water::getPosX() {
    return this->body->GetPosition().x;
}

float Water::getPosY() {
    return this->body->GetPosition().y;
}
```

jun 25, 18 20:09 **Water.h** Page 1/1

```
#ifndef WATER_H
#define WATER_H

#include "Box2D.h"
#include "Entity.h"
#include "PhysicEntity.h"

class Water : public Entity{
private:
    b2World& world;
    b2Body* body;

public:
    Water(b2World& world, float posX, float posY, float width, float height);
    virtual ~Water();
    entity_t getEntityType() {return WATER;}
    void update(void);
    float getPosX();
    float getPosY();
};

#endif
```

jun 25, 18 20:09 **Weapon.cpp** Page 1/1

```cpp
#include "Weapon.h"

Weapon::Weapon(weapon_t t) {
    this->type = t;
    this->countdown = -1;
    this->exploded = false;
    this->direction_angle = 0;
    this->wind_affected = false;
}

Weapon::~Weapon() {
}

bool Weapon::hasExploded() {
    return this->exploded;
}

int Weapon::getCountdown() {
    return this->countdown;
}

int Weapon::getId() {
    return this->id;
}

weapon_t Weapon::getType(void) {
    return this->type;
}

int Weapon::getBlastRadius() {
    return this->blast_radius;
}

int Weapon::getDirectionAngle() {
    return this->direction_angle;
}

int Weapon::addProjectiles(std::map<int, Weapon*> & weapons) {
    return 0;
}
```

jun 25, 18 20:09 **Weapon.h** Page 1/1

```cpp
#ifndef WEAPON_H
#define WEAPON_H

#include "Entity.h"
#include "types.h"
#include <map>

class Weapon : public Entity {
public:
    virtual ~Weapon();
    virtual void update(int current_time, int wind_force) = 0;
    virtual void explode(void) = 0;
    virtual weapon_t getType(void);
    virtual bool isMoving() = 0;
    virtual int getDirectionAngle();
    virtual int addProjectiles(std::map<int, Weapon*> & weapons);
    bool hasExploded(void);
    virtual int getCountdown();
    int getBlastRadius();
    virtual int getId();
protected:
    explicit Weapon(weapon_t type);
    weapon_t type;
    bool exploded;
    bool wind_affected;
    int blast_radius;
    int blast_power;
    int countdown;
    int id;
    int direction_angle;
};

#endif
```

```
#include "WeaponManager.h"

WeaponManager::WeaponManager(std::map<int, Worm*> & worms, std::map<int, Team*>
& teams, WorldPhysic & world) :
worms(worms),
world(world),
teams(teams) {
    this->weaponCounter = 0;
}

void WeaponManager::manageShoot(Event & event, size_t id, unsigned int currentTi
me) {
    if ((this->weapons.size() != 0) || this->worms[id]->didShootInTurn()) {
        return;
    }

    const YAML::Node & nodeEvent = event.getNode();
    weapon_t weapon = (weapon_t) nodeEvent["event"]["weapon"].as<int>();

    if (!this->teams[this->worms[id]->getTeam()]->hasSupplies(weapon)) {
        std::cout << "Disparo ignorado, no quedan supplies." << std::endl;
        return;
    }

    Weapon* newWeapon = NULL;

    if (weapon == w_dynamite) {
        newWeapon = new Dynamite(this->weaponCounter,
        this->world.getWorld(),
        this->worms[id]->getPosX(),
        this->worms[id]->getPosY(),
        nodeEvent["event"]["countdown"].as<int>(),
        currentTime
        );
    } else if (weapon == w_green_grenade || weapon == w_holy_grenade || weapon =
= w_banana) {
        newWeapon = new Grenade(
        this->weaponCounter,
        this->world.getWorld(),
        this->worms[id]->getPosX(),
        this->worms[id]->getPosY(),
        this->worms[id]->isMirrored() ,
        nodeEvent["event"]["sight_angle"].as<int>() ,
        nodeEvent["event"]["power"].as<int>(),
        nodeEvent["event"]["countdown"].as<int>(),
        currentTime,
        weapon
        );
    } else if (weapon == w_bazooka) {
        newWeapon = new Bazooka(this->weaponCounter,
        this->world.getWorld(),
        this->worms[id]->getPosX(),
        this->worms[id]->getPosY(),
        this->worms[id]->isMirrored(),
        nodeEvent["event"]["sight_angle"].as<int>(),
        nodeEvent["event"]["power"].as<int>(),
        weapon
        );
    } else if (weapon == w_air_strike) {
        AirStrike air_strike(this->weaponCounter,
        this->world.getWorld(),
        nodeEvent["event"]["remote_control_x"].as<int>() * gConfiguration.SCALING_FAC
```



```
TOR,
        1
        );

        std::vector<Missil*> missils = air_strike.getMissils();
        for (std::vector<Missil*>::iterator it = missils.begin(); it != missils.
end(); ++it) {
            this->weapons.insert(std::pair<int, Weapon*>((*it)->getId(), (*it)))
;
            this->weaponCounter++;
            this->worms[id]->shoot();
        }
    } else if (weapon == w_cluster) {
        newWeapon = new RedGrenade(this->weaponCounter,
        this->world.getWorld(),
        this->worms[id]->getPosX(),
        this->worms[id]->getPosY(),
        this->worms[id]->isMirrored(),
        nodeEvent["event"]["sight_angle"].as<int>(),
        nodeEvent["event"]["power"].as<int>(),
        nodeEvent["event"]["countdown"].as<int>(),
        currentTime,
        weapon
        );
    } else if (weapon == w_mortar) {
        newWeapon = new Mortar(this->weaponCounter,
        this->world.getWorld(),
        this->worms[id]->getPosX(),
        this->worms[id]->getPosY(),
        this->worms[id]->isMirrored(),
        nodeEvent["event"]["sight_angle"].as<int>(),
        nodeEvent["event"]["power"].as<int>(),
        weapon
        );
    } else if (weapon == w_bat) {
        Bat bat(this->world.getWorld(),
        this->worms[id]->getPosX(),
        this->worms[id]->getPosY(),
        this->worms[id]->isMirrored(),
        nodeEvent["event"]["sight_angle"].as<int>());
    } else if (weapon == w_teleport) {
        Teleportation teleportation(this->worms[id],
        (float) nodeEvent["event"]["remote_control_x"].as<int>() * gConfiguration.SCA
LING_FACTOR,
        (float) nodeEvent["event"]["remote_control_y"].as<int>() * gConfiguration.SCA
LING_FACTOR);
        teleportation.teleport();
    }

    if (newWeapon) {
        this->weapons.insert(std::pair<int, Weapon*>(this->weaponCounter, newWea
pon));
        this->weaponCounter++;
    }

    this->teams[this->worms[id]->getTeam()]->reduceSupplie(weapon);
    this->worms[id]->shoot();
}

void WeaponManager::updateWeapons(unsigned int currentTime, int windForce) {
    std::map<int, Weapon*>::iterator it;
    for(it=this->weapons.begin();it != this->weapons.end();) {
```

```
        if ((it)->second->hasExploded()) {
            int q_added = it->second->addProjectiles(this->weapons);
            this->weaponCounter = this->weaponCounter + q_added;
            delete (it->second);
            it = this->weapons.erase(it);
        } else {
            (it)->second->update(currentTime, windForce);
            it++;
        }
    }
}

std::map<int, Weapon*> & WeaponManager::getWeapons() {
    return this->weapons;
}

bool WeaponManager::hasAliveWeapons() {
    return this->weapons.size();
}
```



```
#ifndef WEAPON_MANGER_H
#define WEAPON_MANGER_H

#include "Weapon.h"
#include "WorldPhysic.h"
#include "types.h"
#include "Worm.h"
#include "event.h"
#include "Dynamite.h"
#include "Bazooka.h"
#include "AirStrike.h"
#include "Bat.h"
#include "Teleportation.h"
#include "RedGrenade.h"
#include "Mortar.h"
#include "Missil.h"
#include "team.h"
#include <map>
#include <iostream>

class WeaponManager {
private:
    std::map<int, Weapon*> weapons;
    std::map<int, Worm*> & worms;
    WorldPhysic & world;
    std::map<int, Team*> & teams;

    int weaponCounter;
public:
    WeaponManager(std::map<int, Worm*> & , std::map<int, Team*> &, WorldPhysic &
);
    void manageShoot(Event & event, size_t id, unsigned int currentTime);
    void updateWeapons(unsigned int currentTime, int windForce);
    std::map<int, Weapon*> & getWeapons();
    bool hasAliveWeapons();
};

#endif
```

jun 25, 18 20:09 **Wind.cpp** Page 1/1

```
#include "Wind.h"
#include <iostream>

Wind::Wind() {
    this->max_wind_force = gConfiguration.MAX_WIND_FORCE;
    this->min_wind_force = gConfiguration.MIN_WIND_FORCE;
    this->wind_force = (rand()%(max_wind_force - min_wind_force + 1 )) + min_win
d_force;
}

Wind::~Wind() {

}

int Wind::getWindForce() {
    return this->wind_force;
}

void Wind::updateWindForce() {
    this->wind_force = (rand()%(max_wind_force - min_wind_force + 1 )) + min_win
d_force;
}
```

jun 25, 18 20:09 **Wind.h** Page 1/1

```
#ifndef WIND_H
#define WIND_H

#include "Box2D.h"
#include "Configuration.h"
#include <stdlib.h>

class Wind {
private:
    int max_wind_force;
    int min_wind_force;
    int wind_force;
public:
    Wind();
    ~Wind();
    void updateWindForce();
    int getWindForce();
};

#endif
```

```
#include "World.h"
#include <unistd.h>
#include <iostream>

#define MAP_WIDTH 2500
#define MAP_HEIGTH 1500

World::World(std::string & map_path, Queue<Snapshot*> & snps) :
snapshots(snps),
map_path(map_path) {
    this->map_node = YAML::LoadFile(map_path);
    initializeWorld();
    this->keep_running = true;
    this->time_sec = 0;
    this->weapon_counter = 0;
}

World::~World() {
    for (std::map<int, Girder*>::iterator it = this->girders.begin(); it!= this-
>girders.end(); ++it) {
        delete it->second;
    }

    for (std::map<int, Worm*>::iterator it = this->worms.begin(); it != this->wo
rms.end(); ++it) {
        delete it->second;
    }

    for (std::map<int, Team*>::iterator it = this->teams.begin(); it != this->te
ams.end(); ++it) {
        delete it->second;
    }

    delete this->water;
    delete this->weaponManager;
    delete this->wind;
    delete this->ceiling;
    delete this->leftWall;
    delete this->rightWall;
}

size_t World::getId(void) const {
    return 0;
}

bool World::isRunning(void) const {
    return true;
}

std::map<int, Worm *> & World::getWorms() {
    return this->worms;
}

Wind* World::getWind() {
    return this->wind;
}

bool World::hasWormsMoving() {
    for (std::map<int, Worm*>::iterator it = this->worms.begin(); it != this->wo
rms.end(); ++it) {
        if (it->second->isMoving() && !it->second->isDead()) return true;
    }
```



```
    return false;
}

bool World::hasWormGotHurt(size_t worm_id) {
    return this->worms[worm_id]->gotHurtInTurn();
}

bool World::hasAliveProjectiles() {
    return this->weaponManager->hasAliveWeapons();
}

bool World::hasWormsAffectedByExplosion() {
    for (std::map<int, Worm*>::iterator it = this->worms.begin(); it != this->wo
rms.end(); ++it) {
        if (it->second->isAffectedByExplosion() && !it->second->isDead()) return
 true;
    }
    return false;
}

bool World::hasWormShooted(size_t worm_id) {
    return this->worms[worm_id]->didShootInTurn();
}

std::map<int, Team*> & World::getTeams() {
    return this->teams;
}

void World::initializeWorld() {
    float water_posX = (MAP_WIDTH / 2) * gConfiguration.SCALING_FACTOR;
    float water_posY = (MAP_HEIGTH - 100) * gConfiguration.SCALING_FACTOR ;
    float water_height = this->map_node["static"]["water_level"].as<int>() * gConfigu
ration.SCALING_FACTOR;
    float water_width =   MAP_WIDTH * gConfiguration.SCALING_FACTOR;

    this->water = new Water(this->worldPhysic.getWorld(), water_posX, water_posY
, water_width, water_height);

    this->wind = new Wind();


    float ceiling_posX = (MAP_WIDTH / 2) * gConfiguration.SCALING_FACTOR;
    float ceiling_posY = 0;
    float ceiling_height = 1;
    float ceiling_width = MAP_WIDTH * gConfiguration.SCALING_FACTOR;
    float wall_height = MAP_HEIGTH * gConfiguration.SCALING_FACTOR;
    float wall_width = 1;
    float left_wall_posX = 0;
    float right_wall_posX = MAP_WIDTH * gConfiguration.SCALING_FACTOR;
    float wall_posY = (MAP_HEIGTH / 2) * gConfiguration.SCALING_FACTOR;

    this->ceiling = new Wall(this->worldPhysic.getWorld(), ceiling_posX, ceiling
_posY, ceiling_width, ceiling_height);
    this->leftWall = new Wall(this->worldPhysic.getWorld(), left_wall_posX, wall
_posY, wall_width, wall_height);
    this->rightWall = new Wall(this->worldPhysic.getWorld(), right_wall_posX, wa
ll_posY, wall_width, wall_height);

    const YAML::Node & static_node = this->map_node["static"];
    const YAML::Node & dynamic_node = this->map_node["dynamic"];

    const YAML::Node& short_girders_node = static_node["short_girders"];
```

```
    const YAML::Node& long_girders_node = static_node["long_girders"];
    const YAML::Node& worms_teams_node = dynamic_node["worms_teams"];

    // Initialize de vigas cortas
    for (YAML::const_iterator it = short_girders_node.begin(); it != short_girde
rs_node.end(); ++it) {
        const YAML::Node &  short_girder = *it;
        int id = short_girder["id"].as<int>();
        float posX = (float) short_girder["x"].as<int>() * gConfiguration.SCALIN
G_FACTOR;
        float posY = (float) short_girder["y"].as<int>() * gConfiguration.SCALIN
G_FACTOR;
        float angle = (float) short_girder["angle"].as<int>() * GRADTORAD;
        Girder* girder_ptr = new Girder(this->worldPhysic.getWorld(), posX, posY
, -angle, 0.8, 3);
        this->girders.insert(std::pair<int, Girder*>(id, girder_ptr));
    }

    // Initialize de vigas largas
    for (YAML::const_iterator it = long_girders_node.begin(); it != long_girders
_node.end(); ++it) {
        const YAML::Node&  long_girder = *it;
        int id = long_girder["id"].as<int>();
        float posX = (float) long_girder["x"].as<int>() * gConfiguration.SCALING
_FACTOR;
        float posY = (float) long_girder["y"].as<int>() * gConfiguration.SCALING
_FACTOR;
        float angle = (float) long_girder["angle"].as<int>() * GRADTORAD;
        Girder* girder_ptr = new Girder(this->worldPhysic.getWorld(), posX, posY
, -angle, 0.8, 6);
        this->girders.insert(std::pair<int, Girder*>(id, girder_ptr));
    }

    // Initialize team
    int id, tid, health;
    float x;
    float y;
    std::string name;
    int aux = 0;
    int wormsHealth = this->map_node["static"]["worms_health"].as<int>();
    int maxTeamHealtlh = 0;

    for (YAML::const_iterator it = worms_teams_node.begin(); it != worms_teams_n
ode.end(); it++) {
        aux = it->second["worms"].size() * wormsHealth;
        if (aux > maxTeamHealtlh) {
            maxTeamHealtlh = aux;
        }
    }


    for (YAML::const_iterator it = worms_teams_node.begin(); it != worms_teams_n
ode.end(); it++) {
        tid = it->first.as<int>();
        Team* new_team = new Team(tid);
        const YAML::Node& worms_node = it->second["worms"];
        int wormsQuantity = it->second["worms"].size();
        for (YAML::const_iterator worms_it = worms_node.begin(); worms_it != wor
ms_node.end(); worms_it++) {
            const YAML::Node& worm = *worms_it;
            name = worm["name"].as<std::string>();
            id = worm["id"].as<int>();
```



```
            health = maxTeamHealtlh / wormsQuantity;
            x = (float) worm["x"].as<int>() * gConfiguration.SCALING_FACTOR;
            y = (float) worm["y"].as<int>() * gConfiguration.SCALING_FACTOR;
            Worm * new_worm = new Worm(name, id, tid, health, this->worldPhysic.
getWorld(), x, y);
            this->worms.insert(std::pair<int, Worm*>(id, new_worm));
            new_team->addMember(new_worm);
        }
        new_team->initializeInventory(dynamic_node["worms_teams"][tid]["inventory"])
;
        this->teams.insert(std::pair<int, Team*>(tid, new_team));
    }

    this->weaponManager = new WeaponManager(this->worms, this->teams, this->worl
dPhysic);
}

void World::updateBodies() {
    this->weaponManager->updateWeapons(getTimeSeconds(), wind->getWindForce());
    for (std::map<int,Worm*>::iterator it=this->worms.begin(); it != this->worms
.end(); ++it) {
        Worm* worm = it->second;
        //if (!worm->isDead())
            worm->update();
    }
}

void World::run() {
    unsigned int step_counter = 0;
    while (this->keep_running) {
        this->worldPhysic.step();
        this->worldPhysic.clearForces();
        step_counter++;
        if (this->worldPhysic.aliveBodies() || step_counter == 60 /*|| this->wea
pons.size() > 0*/) {
            Snapshot* snapshot = new Snapshot();
            snapshot->updateTeams(this->teams);
            snapshot->updateProjectiles(this->weaponManager->getWeapons());
            this->snapshots.push(snapshot);
        }
        updateBodies();
        if (step_counter == 60) {
            this->time_sec++;
            step_counter = 0;
        }
        usleep(16666);
    }

    for (int i = 0; i < 2; i++) {
        Snapshot* snapshot = new Snapshot();
        snapshot->updateTeams(this->teams);
        snapshot->updateProjectiles(this->weaponManager->getWeapons());
        this->snapshots.push(snapshot);
    }

}

void World::stop() {
    this->keep_running = false;
}

unsigned int World::getTimeSeconds(void) {
```

```
    return this->time_sec;
}

void World::executeAction(Event & event, size_t id) {
    YAML::Node eventNode = event.getNode();
    action_t action = (action_t) eventNode["event"]["action"].as<int>();
    switch(action) {
        case a_moveLeft:
            this->worms[id]->moveLeft();
            break;
        case a_moveRight:
            this->worms[id]->moveRight();
            break;
        case a_frontJump:
            this->worms[id]->frontJump();
            break;
        case a_backJump:
            this->worms[id]->backJump();
            break;
        case a_shoot : {
            this->weaponManager->manageShoot(event, id, getTimeSeconds());
            break;
        }
        case a_pickWeapon:
            this->worms[id]->pickWeapon((weapon_t) eventNode["event"]["weapon"].as
<int>());
        default: break;
    }
}
```



```
#ifndef WORLD_H
#define WORLD_H

#include <map>
#include "WorldPhysic.h"
#include "Girder.h"
#include "Worm.h"
#include "yaml.h"
#include "Water.h"
#include "thread.h"
#include "types.h"
#include "ContactListener.h"
#include <sstream>
#include "blocking_queue.h"
#include "event.h"
#include "snapshot.h"
#include "Configuration.h"
#include "Wind.h"
#include "Teleportation.h"
#include "Wall.h"
#include "WeaponManager.h"

#define GRADTORAD 0.0174533

class World : public Thread {
private:
    Queue<Snapshot*> & snapshots;
    bool keep_running;
    WorldPhysic worldPhysic;
    std::map<int, Girder *> girders;
    std::map<int, Worm *> worms;
    std::map<int, Team *> teams;
    int weapon_counter;
    Water * water;
    unsigned int time_sec;
    std::string & map_path;
    Wind* wind;
    Wall* ceiling;
    Wall* leftWall;
    Wall* rightWall;
    virtual bool isRunning(void) const;
    virtual size_t getId(void) const;
    void updateBodies();
    YAML::Node map_node;
    WeaponManager* weaponManager;

public:
    World(std::string &, Queue<Snapshot*> &);
    ~World(void);
    void initializeWorld();
    std::map<int, Worm *> & getWorms();
    std::map<int, Team*> & getTeams();
    Wind* getWind();
    virtual void run(void);
    void stop();
    void executeAction(Event &, size_t);
    unsigned int getTimeSeconds(void);
    bool hasWormsMoving(void);
    bool hasAliveProjectiles(void);
    bool hasWormsAffectedByExplosion(void);
    bool hasWormGotHurt(size_t);
    bool hasWormShooted(size_t);
```

```
};

#endif
```



```
#include "WorldPhysic.h"
#include <iostream>
b2Vec2 WorldPhysic::_generateGravity() {
    b2Vec2 gravity(0.0f, 9.8f);
    return gravity;
}

WorldPhysic::WorldPhysic() : world(_generateGravity()) {
    this->world.SetAllowSleeping(true);
    this->world.SetContactListener(&contactListener);
}

b2World& WorldPhysic::getWorld() {
    return this->world;
}

bool WorldPhysic::aliveBodies() {
    for (b2Body* b = this->world.GetBodyList(); b; b = b->GetNext()) {
        if (b->IsAwake()) {
            entity_t entity = static_cast<Entity*>(b->GetUserData())->getEntityT
ype();
            if (entity == STRUCTURE || entity == WATER || entity == WALL) {
                continue;
            }
            return true;
        }
    }
    return false;
}

void WorldPhysic::step() {
    this->world.Step(gConfiguration.WORLD_TIME_STEP, gConfiguration.WORLD_VELOCI
TY_ITERATIONS, gConfiguration.WORLD_POSITION_ITERATIONS);
}

void WorldPhysic::clearForces() {
    this->world.ClearForces();
}
```

jun 25, 18 20:09 **WorldPhysic.h** Page 1/1

```
#ifndef WORLD_PHYSIC_H
#define WORLD_PHYSIC_H

#include "Box2D.h"
#include "ContactListener.h"
#include "Configuration.h"

class WorldPhysic {
public:
    WorldPhysic();
    b2World& getWorld();
    void step();
    void clearForces();
    bool aliveBodies();
private:
    ContactListener contactListener;
    b2Vec2 _generateGravity();
    b2World world;
};

#endif
```

jun 25, 18 20:58 **Worm.cpp** Page 1/6

```
#include "Worm.h"
#include <iostream>

Worm::Worm(std::string n, int id, int team_id, int h, b2World& world, float posX
, float posY) :
world(world) {

    b2BodyDef wormDef;
    wormDef.type = b2_dynamicBody;
    wormDef.fixedRotation = true;
    wormDef.allowSleep = true;
    wormDef.position.Set(posX, posY);
    b2Body* body = world.CreateBody(&wormDef);
    body->SetUserData(this);

    //b2PolygonShape wormShape;
    //wormShape.SetAsBox(WORM_WIDTH/2, WORM_HEIGHT/2);

    b2CircleShape wormShape;
    wormShape.m_radius = WORM_RADIUS;

    b2FixtureDef wormFixture;
    wormFixture.shape = &wormShape;
    wormFixture.density = WORM_DENSITY;
    wormFixture.friction = WORM_FRICTION;
    wormFixture.filter.categoryBits = WORM_PHYSIC;
    wormFixture.filter.maskBits = STRUCTURE_PHYSIC | WATER_PHYSIC;
    body->CreateFixture(&wormFixture);

    this->body = body;
    this->numFootContacts = 0;


    this->id = id;
    this->inclination = NONE;
    this->health = h;
    this->team_id = team_id;
    this->name = n;
    this->angle = 0;
    this->falling = false;
    this->mirrored = false;
    this->hurtInTurn = false;
    this->affectedByExplosion = false;
    this->shootedInTurn = false;
    this->weapon = w_null;
    this->direction_angle = 0;
    this->fallenDistance = 0;
    this->falled = false;
}

Worm::~Worm(void) {
    if (this->body) {
        this->world.DestroyBody(this->body);
    }
}

void Worm::frontJump(void) {
    if (this->numFootContacts <= 0) return;
    float factor;
    mirrored == true ? factor = 1.0 : factor = -1.0;
    float x_impulse = this->body->GetMass() * gConfiguration.WORM_FRONT_JUMP_X_I
MPULSE;
```

```
    float y_impulse = this->body->GetMass() * gConfiguration.WORM_FRONT_JUMP_Y_I
MPULSE;
    this->body->ApplyLinearImpulse(b2Vec2(x_impulse * factor,-y_impulse), this->
body->GetWorldCenter(), true);
}

void Worm::backJump(void) {
    if (this->numFootContacts <= 0) return;
    float factor;
    mirrored == true ? factor = 1.0 : factor = -1.0;
    float x_impulse = this->body->GetMass() * gConfiguration.WORM_BACK_JUMP_X_IM
PULSE;
    float y_impulse = this->body->GetMass() * gConfiguration.WORM_BACK_JUMP_Y_IM
PULSE;
    this->body->ApplyLinearImpulse(b2Vec2(-x_impulse * factor, -y_impulse), this
->body->GetWorldCenter(), true);
}

void Worm::moveRight(void) {
    if (isGrounded() && !affectedByExplosion) {
        b2Vec2 velocity;
        velocity.x = cosf(angle) * gConfiguration.WORM_SPEED;
        velocity.y = sinf(angle) * gConfiguration.WORM_SPEED;
        this->body->SetLinearVelocity(velocity);
        this->mirrored = true;
    }
}

void Worm::moveLeft(void) {
    if (isGrounded() && !affectedByExplosion) {
        b2Vec2 velocity; // = this->body->GetLinearVelocity();
        velocity.x = cosf(angle) * -gConfiguration.WORM_SPEED;
        velocity.y = sinf(angle) * -gConfiguration.WORM_SPEED;
        this->body->SetLinearVelocity(velocity);
        this->mirrored = false;
    }
}

bool Worm::isMirrored(void) {
    return this->mirrored;
}

float Worm::getPosX(void) {
    return this->body->GetPosition().x;
}

float Worm::getPosY(void) {
    return this->body->GetPosition().y;
}

int Worm::getId(void) {
    return this->id;
}

int Worm::getTeam(void) {
    return this->team_id;
}

int Worm::getHealth(void) {
    return this->health;
}
```



```
std::string Worm::getName(void) {
    return this->name;
}

void Worm::addFootContact(void) {
    this->numFootContacts++;
}

void Worm::deleteFootContact(void) {
    this->numFootContacts--;
}

void Worm::shoot() {
    this->shootedInTurn = true;
    this->weapon = w_null;
}

void Worm::hurt(int damage) {
    if (this->health - damage < 0) {
        kill();
    } else {
        this->health -=damage;
    }
    this->hurtInTurn = true;
}

bool Worm::isWalking(void) {
    b2Vec2 velocity = this->body->GetLinearVelocity();
    return (velocity.y || velocity.x) && isGrounded();
}

bool Worm::isMoving(void) {
    b2Vec2 velocity = this->body->GetLinearVelocity();
    return velocity.y || velocity.x;
    //return this->wormPhysic.haveHorizontalSpeed() || this->wormPhysic.haveVert
icalSpeed();
}

bool Worm::isFalling(void) {
    b2Vec2 velocity = this->body->GetLinearVelocity();
    return !isGrounded() && velocity.y;// && !jumping;
}

bool Worm::isGrounded(void) {
    return numFootContacts > 0;
}

bool Worm::isDead(void) {
    return this->health == 0;
}

void Worm::setAngle(float angle) {
    this->angle = angle;
}

void Worm::setFalling(bool falling) {
    this->falling = falling;
}

void Worm::update() {
    if (!this->isGrounded()) {
        if (this->falled && (getPosY() < this->fallenDistance)) {
```

```
            this->fallenDistance = getPosY();
        } else if (!this->falled){
            this->fallenDistance = getPosY();
            this->falled = true;
        }
    } else if (isGrounded() && (falled)) {
        this->fallenDistance = getPosY() - this->fallenDistance;
        if (this->fallenDistance > gConfiguration.WORM_MAX_FALL_DISTANCE) {
            hurt(this->fallenDistance);
        }
        this->falled = false;
        this->fallenDistance = 0;
    }

    if (!this->body->GetLinearVelocity().x && !this->body->GetLinearVelocity().y
) {
        this->affectedByExplosion = false;
    }

    b2Vec2 mov_speed = this->body->GetLinearVelocity();

    if (round(mov_speed.x) == 0) {
        if (mov_speed.y > 0) {
            this->direction_angle = 180;
            return;
        }

        if (mov_speed.y < 0) {
            this->direction_angle = 0;
            return;
        }
    }

    if (round(mov_speed.y) == 0) {
        if (mov_speed.x > 0) {
            this->direction_angle = 90;
            return;
        }

        if (mov_speed.x < 0) {
            this->direction_angle = 270;
            return;
        }
    }

    int ang = atan(mov_speed.x/mov_speed.y) * gConfiguration.RADTODEG;

    // Primer cuad rante
    if (mov_speed.y < 0 && mov_speed.x > 0) {
        this->direction_angle = -ang;
    }

    // Segundo cuadrante
    if (mov_speed.y < 0 && mov_speed.x < 0) {
        this->direction_angle = 360 - ang;
    }

    // Tercer cuadrante
    if (mov_speed.y > 0 && mov_speed.x < 0) {
        this->direction_angle = 180 - ang;
    }
```



```
    // Cuarto cuadrante
    if (mov_speed.y > 0 && mov_speed.x > 0) {
        this->direction_angle = 180 - ang;
    }



    if (getPosY() > gConfiguration.WORLD_Y_LIMIT) {
        this->kill();
    }

    if (normalX < 0 && normalY < 0 && !mirrored) {
        this->inclination = DOWN;
    } else if (normalX > 0 && normalY  < 0 && !mirrored) {
        this->inclination = UP;
    } else if (normalX < 0 && normalY  < 0 && mirrored) {
        this->inclination = UP;
    } else if (normalX > 0 && normalY  < 0 && mirrored) {
        this->inclination = DOWN;
    } else this->inclination = NONE;
}

bool Worm::isAffectedByExplosion() {
    return this->affectedByExplosion;
}

void Worm::setAffectedByExplosion(){
    this->affectedByExplosion = true;
}

void Worm::kill() {
    this->health = 0;
    this->hurtInTurn = true;
}

void Worm::refreshByNewTurn(void) {
    this->hurtInTurn = false;
    this->shootedInTurn = false;
}

bool Worm::gotHurtInTurn(void) {
    return this->hurtInTurn;
}

bool Worm::didShootInTurn(void) {
    return this->shootedInTurn;
}

void Worm::setPosition(float posX, float posY) {
    this->body->SetTransform(b2Vec2(posX, posY), this->body->GetAngle());
    this->body->SetAwake(true);
}

void Worm::setNormal(b2Vec2 normal) {
    this->normalX = normal.x;
    this->normalY = normal.y;
}

worm_inclination_t Worm::getInclination() {
    return this->inclination;
}
```

```
int Worm::getDirectionAngle() {
    return this->direction_angle;
}

void Worm::pickWeapon(weapon_t weapon) {
    this->weapon = weapon;
}
```



```
#ifndef WORM_H
#define WORM_H

#include "Box2D/Box2D.h"
#include "Entity.h"
#include <string>
#include "PhysicEntity.h"
#include "Configuration.h"
#include "types.h"

#define WORM_HEIGHT 0.8f
#define WORM_RADIUS 0.54f
#define WORM_WIDTH 0.8f
#define WORM_DENSITY 1.0f
#define WORM_FRICTION 0.3f

class Worm : public Entity {
    private:
        int health;
        int id;
        int team_id;

        float posX;
        float posY;
        float angle;
        float fallenDistance;
        worm_inclination_t inclination;
        weapon_t weapon;

        bool affectedByExplosion;
        bool falled;
        bool mirrored;
        bool falling;
        bool hurtInTurn;
        bool shootedInTurn;
        int numFootContacts;
        float normalX;
        float normalY;
        int direction_angle;
        std::string name;
        b2World& world;
        b2Body* body;
    public:
        // mirrored = true significa mirando a derecha
        Worm(std::string, int id, int team_id, int h, b2World& World, float posX
, float posY);
        virtual ~Worm(void);
        void hurt(int);
        void frontJump(void);
        void backJump(void);
        void moveLeft(void);
        void moveRight(void);
        void pointHigher(void);
        void pointMoreDown(void);
        float getPosY(void);
        float getPosX(void);
        void setAngle(float angle);
        int getId(void);
        int getTeam(void);
        int getHealth(void);
        std::string getName(void);
        entity_t getEntityType() {return WORM;}
```

```
        void addFootContact(void);
        void deleteFootContact(void);
        bool isMirrored(void);
        void shoot(/* entity_t weapon */);
        bool isWalking(void);
        bool isFalling(void);
        bool isGrounded(void);
        bool isDead(void);
        bool gotHurtInTurn(void);
        bool didShootInTurn(void);
        void setAffectedByExplosion();
        void setFalling(bool);
        void update(void);
        void kill(void);
        bool isMoving(void);
        bool isAffectedByExplosion(void);
        void refreshByNewTurn(void);
        void setPosition(float posX, float posY);
        void setNormal(b2Vec2 normal);
        worm_inclination_t getInclination();
        int getDirectionAngle();
        void pickWeapon(weapon_t weapon);
};

#endif
```


Table of Contents