```cpp
#include "ClientHandler.h"
#include "MapsList.h"
#include <iostream>
#include <string>

ClientHandler::ClientHandler(ServerProtocol&& client,
                             GamesList& games, std::mutex& mutex_cout):
    client(std::move(client)), games(games),
    connected(false), mutex_cout(mutex_cout){}

ClientHandler::~ClientHandler(){}

void ClientHandler::run(){
    try{
        while(!this->connected){
            char action = this->client.getProtocol().receiveChar();
            std::string player_name = this->client.getProtocol().receiveString()
;
            this->client.setName(player_name);
            if (action == CREATE_GAME_ACTION) {
                this->createGame();
            } else if (action == JOIN_GAME_ACTION) {
                this->joinGame();
            }
        }
    } catch(const SocketException& e){
        std::lock_guard<std::mutex> lock(this->mutex_cout);
        std::cout << "[INFO] Un cliente se desconecto" << std::endl;
    } catch(const std::exception& e){
        std::lock_guard<std::mutex> lock(this->mutex_cout);
        std::cout << "[ERROR] Error con un cliente: " << e.what() << std::endl;
    }
    this->running = false;
}

void ClientHandler::stop(){
    this->client.getProtocol().stop();
}

void ClientHandler::createGame(){
    maps_list_t maps_list = MapsList::getAllMaps();

    size_t size = maps_list.size();
    this->client.getProtocol().sendLength(size);

    for (size_t i = 0; i < size; i++){
        this->client.getProtocol().sendString(maps_list[i]);
    }

    if (size == 0){
        return;
    }

    std::string map = this->client.getProtocol().receiveString();
    if (map.empty()){
        return;
    }
    std::string game_name = this->client.getProtocol().receiveString();
    int max_players = this->client.getProtocol().receiveLength();

    this->games.checkGames();

    bool result = this->games.addGame(game_name, map, max_players, this->client)
;
    if (!result){
        this->client.getProtocol().sendChar(false);
```



```cpp
    } else {
        this->connected = true;
    }
}

void ClientHandler::joinGame(){
    games_list_t games_list = this->games.getJoinableGames(this->client.getName(
));

    size_t size = games_list.size();
    this->client.getProtocol().sendLength(size);

    for (size_t i = 0; i < size; i++){
        this->client.getProtocol().sendString(games_list[i]);
    }

    if (size == 0){
        return;
    }

    std::string game_name = this->client.getProtocol().receiveString();
    if (game_name.empty()){
        return;
    }

    bool result = this->games.addPlayer(game_name, this->client);

    if (!result){
        this->client.getProtocol().sendChar(false);
    } else {
        this->connected = true;
    }
}
```

jun 10, 18 19:29 **ClientHandler.h** Page 1/1

```
#ifndef __CLIENTHANDLER_H__
#define __CLIENTHANDLER_H__

#include "Socket.h"
#include "Server.h"
#include "Thread.h"
#include "Player.h"
#include "GamesList.h"
#include <mutex>

class ClientHandler: public Thread{
    private:
        Player client;
        GamesList& games;
        bool connected;
        std::mutex& mutex_cout;

        /* Crea una partida nueva */
        void createGame();

        /* Agrega un jugador a una partida */
        void joinGame();

    public:
        /* Constructor */
        ClientHandler(ServerProtocol&& cli, GamesList& games, std::mutex& mtx_co
ut);

        /* Destructor */
        ~ClientHandler();

        /* Ejecuta el client handler */
        void run();

        /* Se desconecta abruptamente del cliente */
        void stop();
};

#endif
```

jun 20, 18 18:42 **GamesList.cpp** Page 1/2

```
#include "GamesList.h"
#include "Path.h"
#include "Server.h"
#include <iostream>
#include <string>

typedef std::unordered_map<std::string, std::unique_ptr<Game>>::iterator games_i
t;

GamesList::GamesList(Server& server, std::mutex& mutex_cout):
    server(server), mutex_cout(mutex_cout){}

GamesList::~GamesList(){
    for (games_it it = this->games.begin(); it != this->games.end(); ++it){
        it->second->join();
        std::lock_guard<std::mutex> lock(this->mutex_cout);
        std::cout << "[INFO] Partida terminada: " << it->first << std::endl;
    }
}

bool GamesList::addGame(string& name, string& map, int max, Player& player){
    std::lock_guard<std::mutex> lock(this->mutex);
    auto it = this->games.find(name);
    if (it != this->games.end()){
        return false;
    }

    try{
        Game* g = new Game(max, SERVER_CONFIG_FILE, MAPS_PATH + map, this->serve
r);
        std::unique_ptr<Game> game(g);
        this->games[name] = std::move(game);
        std::lock_guard<std::mutex> lock(this->mutex_cout);
        std::cout << "[INFO] Nueva partida creada: " << name << std::endl;
    } catch(const std::exception& e){
        std::lock_guard<std::mutex> lock(this->mutex_cout);
        std::cout << "[ERROR] Error al crear partida: " << name;
        std::cout << "-> " << e.what() << std::endl;
        return false;
    }

    std::string player_name = player.getName();
    bool result = this->games[name]->addPlayer(player);
    if (result){
        std::lock_guard<std::mutex> lock(this->mutex_cout);
        std::cout << "[INFO] El jugador '" << player_name;
        std::cout << "' se unio a la partida '" <<  name << "'" << std::endl;
    }

    return result;
}

games_list_t GamesList::getJoinableGames(const std::string& player_name){
    std::lock_guard<std::mutex> lock(this->mutex);
    games_list_t joinables;

    for (games_it it = this->games.begin(); it != this->games.end(); ++it){
        if (it->second->playerCanJoin(player_name)){
            joinables.push_back(it->first);
        }
    }
    return std::move(joinables);
}

bool GamesList::addPlayer(const std::string& game_name, Player& player){
    std::lock_guard<std::mutex> lock(this->mutex);
```

jun 20, 18 18:42 **GamesList.cpp** Page 2/2

```
65      std::string player_name = player.getName();
66      bool result = this->games[game_name]->addPlayer(player);
67      if (result){
68          std::lock_guard<std::mutex> lock(this->mutex_cout);
69          std::cout << "[INFO] El jugador '" << player_name;
70          std::cout << "' se unio a la partida '" << game_name << "'" << std::endl;
71      }
72      if (this->games[game_name]->isFull()){
73          std::lock_guard<std::mutex> lock(this->mutex_cout);
74          std::cout << "[INFO] Partida iniciada: " << game_name << std::endl;
75          this->games[game_name]->start();
76      }
77      return result;
78  }
79
80  void GamesList::checkGames(){
81      std::lock_guard<std::mutex> lock(this->mutex);
82      auto it = this->games.begin();
83      while (it != this->games.end()){
84          if (! it->second->isRunning()){
85              it->second->join();
86              std::lock_guard<std::mutex> lock(this->mutex_cout);
87              std::cout << "[INFO] Partida terminada: " << it->first << std::endl;
88              it = this->games.erase(it);
89          } else {
90              ++it;
91          }
92      }
93  }
```

jun 10, 18 19:29 **GamesList.h** Page 1/1

```
1   #ifndef __GAMESLIST_H__
2   #define __GAMESLIST_H__
3
4   #include <vector>
5   #include <string>
6   #include <unordered_map>
7   #include <memory>
8   #include <mutex>
9   #include "Game.h"
10
11  typedef std::vector<std::string> games_list_t;
12
13  class Server;
14
15  class GamesList{
16      private:
17          Server& server;
18          std::unordered_map<std::string, std::unique_ptr<Game>> games;
19          std::mutex mutex;
20          std::mutex& mutex_cout;
21
22          typedef const std::string string;
23
24      public:
25          /* Constructor */
26          GamesList(Server& server, std::mutex& mutex_cout);
27
28          /* Destructor */
29          ~GamesList();
30
31          /* Agrega una patida nueva a la lista */
32          bool addGame(string& game_name, string& map, int max_players, Player& pl
    ayer);
33
34          /* Devuelve una lista con las partidas a las cuales se puede
35           * unir el jugador */
36          games_list_t getJoinableGames(const std::string& player_name);
37
38          /* Agrega un jugador a la partida */
39          bool addPlayer(const std::string& game_name, Player& player);
40
41          /* Verifica las partidas que terminaron */
42          void checkGames();
43  };
44
45  #endif
```

jun 11, 18 17:06 **MapsList.cpp** Page 1/1

```cpp
#include "MapsList.h"
#include "Path.h"
#include <string>

maps_list_t MapsList::getAllMaps(){
    maps_list_t maps_list;

    struct dirent* entry;
    DIR* dir = opendir(MAPS_PATH.c_str());
    if (!dir){
        std::move(maps_list);
    }

    while((entry = readdir(dir))){  // NOLINT (La solucion que propone esta deprecated)
        std::string file(entry->d_name);
        if (file.rfind(YAML_EXTENSION) != std::string::npos){
            maps_list.push_back(file);
        }
    }

    closedir(dir);
    return std::move(maps_list);
}
```

may 28, 18 14:41 **MapsList.h** Page 1/1

```cpp
#ifndef __MAPSLIST_H__
#define __MAPSLIST_H__

#include <dirent.h>
#include <vector>
#include <string>

typedef std::vector<std::string> maps_list_t;

class MapsList{
    public:
        /* Devuelve una lista con todos los mapas */
        static maps_list_t getAllMaps();
};

#endif
```

```cpp
#include <string>
#include <memory>
#include <iostream>
#include "Server.h"
#include "ClientHandler.h"

#define MAX_CLIENT_WAIT 100

Server::Server(const std::string& service, std::mutex& mutex_cout):
    socket(Socket::Server(service.c_str(), MAX_CLIENT_WAIT)),
    games_list(*this, mutex_cout), mutex_cout(mutex_cout){}

Server::~Server(){
    for (std::unique_ptr<Thread>& client: this->clients){
        client->stop();
        client->join();
    }
}

void Server::run(){
    while (this->running){
        try{
            Socket client = this->socket.acceptClient();
            {
                std::lock_guard<std::mutex> lock(this->mutex_cout);
                std::cout << "[INFO] Nuevo cliente conectado." << std::endl;
            }
            ServerProtocol protocol(std::move(client));
            this->addConnectedClient(std::move(protocol));

            this->check();
        } catch(const std::exception& e){
            if (this->running){
                std::lock_guard<std::mutex> lock(this->mutex_cout);
                std::cout << "[ERROR] " << e.what() << std::endl;
            }
        }
    }
}

void Server::stop(){
    this->running = false;
    this->socket.stop();
}

void Server::check(){
    //Elimino threads que ya terminaron
    this->games_list.checkGames();
    std::lock_guard<std::mutex> lock(this->mutex);
    auto it = this->clients.begin();
    while (it != this->clients.end()){
        if (!(*it)->isRunning()){
            (*it)->join();
            it = this->clients.erase(it);
        } else {
            ++it;
        }
    }
}

void Server::addConnectedClient(ServerProtocol&& protocol){
    std::lock_guard<std::mutex> lock(this->mutex);
    Thread* t = new ClientHandler(std::move(protocol), games_list, mutex_cout);
    std::unique_ptr<Thread> th(t);
    th->start();
    this->clients.push_back(std::move(th));
```



```cpp
}
```

jun 07, 18 23:33 **Server.h** Page 1/1

```
#ifndef __SERVER_H__
#define __SERVER_H__

#include <string>
#include <list>
#include <memory>
#include <mutex>
#include "Socket.h"
#include "Thread.h"
#include "GamesList.h"

class Server: public Thread{
    private:
        Socket socket;
        std::list<std::unique_ptr<Thread>> clients;
        GamesList games_list;
        std::mutex& mutex_cout;
        std::mutex mutex;

        /* Elimina los clientes que terminaron su comunicacion
         * de la lista */
        void check();

    public:
        /* Crea el server y lo asocia al puerto indicado */
        Server(const std::string& service, std::mutex& mutex_cout);

        /* Desconecta el server */
        ~Server();

        /* Ejecuta el server */
        void run();

        /* Avisa al server que debe dejar de ejecutarse */
        void stop();

        /*Agrega un nuevo cliente ya conectado */
        void addConnectedClient(ServerProtocol&& protocol);
};

#endif
```

jun 20, 18 18:42 **DataSender.cpp** Page 1/3

```
#include "DataSender.h"
#include <map>
#include <string>
#include <vector>

typedef std::vector<Player> Players;

DataSender::DataSender(World& world, Players& players, GameParameters& params):
    objects(world.getObjectsList()), girders(world.getGirdersList()),
    players(players), mutex(world.getMutex()), active(false),
    sleep_time(params.getDataSenderSleep()){
        for (size_t i = 0; i < this->players.size(); i++){
            std::unique_ptr<PlayerDataSender> s(new PlayerDataSender(this->playe
rs[i]));
            this->players_data_senders.push_back(std::move(s));
            this->players_data_senders[i]->start();
        }
    }

DataSender::~DataSender(){
    for (size_t i = 0; i < this->players.size(); i++){
        this->players_data_senders[i]->stop();
        this->players_data_senders[i]->join();
    }
}

void DataSender::run(){
    while(this->running){
        std::this_thread::sleep_for(std::chrono::milliseconds(this->sleep_time))
;
        std::lock_guard<std::mutex> lock(this->mutex);
        this->active = false;
        std::list<physical_object_ptr>::iterator it = this->objects.begin();

        while(it != this->objects.end()){
            if ((*it)->isDead() && !(*it)->getBody()){
                Buffer data = ServerProtocol::sendDeadObject(*it);

                this->sendBuffer(data);
                it = this->objects.erase(it);
                continue;
            }

            if ((*it)->isMoving()){
                Buffer data = ServerProtocol::sendObject(*it);
                this->sendBuffer(data);
                this->active = true;
            }
            ++it;
        }

        this->notifyAll();
    }
}

void DataSender::sendBackgroundImage(Buffer& image){
    this->sendBuffer(image);
    this->notifyAll();
}

void DataSender::sendStartGame(){
    Buffer data = ServerProtocol::sendStartGame();
    this->sendBuffer(data);
    this->notifyAll();
}

```

```
65  void DataSender::sendTurnData(int turn_time, int time_after_shoot){
66      Buffer data = ServerProtocol::sendTurnData(turn_time, time_after_shoot);
67      this->sendBuffer(data);
68      this->notifyAll();
69  }
70
71  void DataSender::sendPlayersId(){
72      Buffer length = ServerProtocol::sendLengthBuffer(this->players.size());
73      this->sendBuffer(length);
74      for (Player& player: this->players){
75          Buffer data = ServerProtocol::sendPlayerId(player);
76          this->sendBuffer(data);
77      }
78      this->notifyAll();
79  }
80
81  void DataSender::sendGirders(){
82      Buffer length = ServerProtocol::sendLengthBuffer(this->girders.size());
83      this->sendBuffer(length);
84      for (physical_object_ptr& girder: this->girders){
85          Buffer data = ServerProtocol::sendGirder(girder);
86          this->sendBuffer(data);
87      }
88      this->notifyAll();
89  }
90
91  void DataSender::sendWeaponsAmmo(std::map<std::string, unsigned int>& weapons){
92      Buffer length = ServerProtocol::sendLengthBuffer(weapons.size());
93      this->sendBuffer(length);
94      std::map<std::string, unsigned int>::iterator it;
95      for (it = weapons.begin(); it != weapons.end(); ++it){
96          Buffer data = ServerProtocol::sendWeaponAmmo(it->first, it->second);
97          this->sendBuffer(data);
98      }
99      this->notifyAll();
100 }
101
102 void DataSender::sendStartTurn(int worm_id, int player_id, float wind){
103     Buffer data = ServerProtocol::sendStartTurn(worm_id, player_id, wind);
104     this->sendBuffer(data);
105     this->notifyAll();
106 }
107
108 void DataSender::sendWeaponChanged(const std::string &weapon){
109     Buffer data = ServerProtocol::sendWeaponChanged(weapon);
110     this->sendBuffer(data);
111     this->notifyAll();
112 }
113
114 void DataSender::sendWeaponShot(const std::string& weapon){
115     Buffer data = ServerProtocol::sendWeaponShot(weapon);
116     this->sendBuffer(data);
117     this->notifyAll();
118 }
119
120 void DataSender::sendMoveAction(char action){
121     if (action == MOVE_RIGHT || action == MOVE_LEFT){
122         return;
123     }
124     Buffer data = ServerProtocol::sendMoveAction(action);
125     this->sendBuffer(data);
126     this->notifyAll();
127 }
128
129 void DataSender::sendUpdateScope(int angle) {
130     Buffer data = ServerProtocol::sendUpdateScope(angle);
```



```
131     this->sendBuffer(data);
132     this->notifyAll();
133 }
134
135 void DataSender::sendEndGame(const std::string& winner){
136     Buffer data = ServerProtocol::sendEndGame(winner);
137     this->sendBuffer(data);
138     this->notifyAll();
139 }
140
141 void DataSender::sendEndTurn(){
142     Buffer data = this->players[0].getProtocol().sendEndTurn();
143     this->sendBuffer(data);
144     this->notifyAll();
145 }
146
147 bool DataSender::isActive(){
148     std::lock_guard<std::mutex> lock(this->mutex);
149     return this->active;
150 }
151
152 void DataSender::sendBuffer(const Buffer& buffer){
153     for (size_t i = 0; i < this->players.size(); i++){
154         if (this->players[i].isConnected()){
155             this->players_data_senders[i]->sendData(buffer);
156         }
157     }
158 }
159
160 void DataSender::notifyAll(){
161     for (size_t i = 0; i < this->players.size(); i++){
162         if (this->players[i].isConnected()){
163             this->players_data_senders[i]->notify();
164         }
165     }
166 }
```

```
1  #ifndef __DATASENDER_H__
2  #define __DATASENDER_H__
3
4  #include "Thread.h"
5  #include "World.h"
6  #include "PhysicalObject.h"
7  #include "Player.h"
8  #include "ServerProtocol.h"
9  #include "PlayerDataSender.h"
10 #include "Buffer.h"
11 #include <list>
12 #include <memory>
13 #include <vector>
14 #include <map>
15 #include <string>
16
17 // Clase que se encarga de enviar datos a los jugadores
18 class DataSender: public Thread{
19     private:
20         std::list<physical_object_ptr>& objects;
21         std::list<physical_object_ptr>& girders;
22         std::vector<Player>& players;
23         std::vector<std::unique_ptr<PlayerDataSender>> players_data_senders;
24         std::mutex& mutex;
25         bool active;
26         int sleep_time;
27
28         // Envia la informacion del buffer a todos los jugadores
29         void sendBuffer(const Buffer& buffer);
30         void notifyAll();
31
32     public:
33         DataSender(World& world, std::vector<Player>& players, GameParameters& p
   aram);
34         ~DataSender();
35
36         //Envia constantemente los datos de los objetos
37         void run() override;
38
39         //Envia la imagen de fondo
40         void sendBackgroundImage(Buffer& image);
41
42         //Envia los datos del turno
43         void sendTurnData(int turn_time, int time_after_shoot);
44
45         //Envia los datos de los jugadores
46         void sendPlayersId();
47
48         //Envia los datos de las vigas
49         void sendGirders();
50
51         //Envia las municiones de las armas
52         void sendWeaponsAmmo(std::map<std::string, unsigned int>& weapons);
53
54         //Envia que el jugador cambio de arma
55         void sendWeaponChanged(const std::string &weapon);
56
57         //Envia que el gusano actual salto
58         void sendMoveAction(char action);
59
60         //Envia que el jugador cambio el angulo de la mira
61         void sendUpdateScope(int angle);
62
63         //Envia que el jugador disparo un arma
64         void sendWeaponShot(const std::string& weapon);
65
```



```
66         //Envia la senial de comienzo del juego
67         void sendStartGame();
68
69         //Envia la senial de que inicia un nuevo turno
70         void sendStartTurn(int worm_id, int player_id, float wind);
71
72         //Envia la senial de terminar turno
73         void sendEndTurn();
74
75         //Envia la senial de que el juego termino
76         void sendEndGame(const std::string& winner);
77
78         //Devuelve true si sigue enviando datos
79         bool isActive();
80 };
81
82
83 #endif
```

```cpp
#include "PlayerDataReceiver.h"
#include <string>

PlayerDataReceiver::PlayerDataReceiver(Player& player, DataSender& sender):
    player(player), data_sender(sender), is_my_turn(false){}

PlayerDataReceiver::~PlayerDataReceiver(){}

void PlayerDataReceiver::run(){
    try{
        while (this->running){
            Buffer data = this->player.getProtocol().receiveBuffer();
            std::lock_guard<std::mutex> lock(this->mutex);
            if (this->is_my_turn){
                this->analizeReceivedData(data);
            }
        }
    } catch(const std::exception& e){
        this->player.disconnect();
    }
}

void PlayerDataReceiver::beginTurn(){
    std::lock_guard<std::mutex> lock(this->mutex);
    this->is_my_turn = true;
}

void PlayerDataReceiver::endTurn(){
    std::lock_guard<std::mutex> lock(this->mutex);
    this->is_my_turn = false;
}

void PlayerDataReceiver::analizeReceivedData(Buffer& buffer){
    char action = buffer.getNext();

    if (action == ACTION) {
        char worm_action = buffer.getNext();
        if (worm_action == MOVE_ACTION){
            char move = buffer.getNext();
            if (this->player.getCurrentWorm().move(move)){
                this->data_sender.sendMoveAction(move);
            }
        } else if (worm_action == CHANGE_WEAPON_ACTION) {
            std::string weapon(ServerProtocol::receiveStringBuffer(buffer));
            this->data_sender.sendWeaponChanged(weapon);
            this->player.changeWeapon(weapon);
        } else if (worm_action == MOVE_SCOPE) {
            int32_t angle = ServerProtocol::receiveIntBuffer(buffer);
            this->data_sender.sendUpdateScope(angle);
        } else if (worm_action == SHOOT_WEAPON) {
            int angle = ServerProtocol::receiveIntBuffer(buffer);
            int power = ServerProtocol::receiveIntBuffer(buffer);
            int time = ServerProtocol::receiveIntBuffer(buffer);
            const std::string& weapon = this->player.getCurrentWorm().getCurrent
Weapon();
            this->data_sender.sendWeaponShot(weapon);
            this->player.getCurrentWorm().shoot(angle, power, time);
        } else if (worm_action == SHOOT_SELF_DIRECTED) {
            int pos_x = ServerProtocol::receiveIntBuffer(buffer) / UNIT_TO_SEND;
            int pos_y = ServerProtocol::receiveIntBuffer(buffer) / UNIT_TO_SEND;
            const std::string& weapon = this->player.getCurrentWorm().getCurrent
Weapon();
            this->data_sender.sendWeaponShot(weapon);
            this->player.getCurrentWorm().shoot(b2Vec2(pos_x, pos_y));
        }
    }
```



```cpp
}
```

jun 10, 18 16:09 **PlayerDataReceiver.h** Page 1/1

```h
#ifndef __PLAYERDATARECEIVER_H__
#define __PLAYERDATARECEIVER_H__

#include "Thread.h"
#include "Player.h"
#include "DataSender.h"
#include <mutex>

/* Clase que se encarga de recibir datos del jugador
 * y de analizarlos */
class PlayerDataReceiver: public Thread{
    private:
        Player& player;
        DataSender& data_sender;
        bool is_my_turn;
        std::mutex mutex;

        /* Analiza los datos recibidos */
        void analizeReceivedData(Buffer& data);

    public:
        /* Constructor */
        PlayerDataReceiver(Player& player, DataSender& data_sender);

        /* Destructor */
        ~PlayerDataReceiver();

        /* Comienza a recibir datos */
        void run() override;

        /* Comienza el turno */
        void beginTurn();

        /* Termina el turno */
        void endTurn();
};

#endif
```

jun 09, 18 19:06 **PlayerDataSender.cpp** Page 1/1

```cpp
#include "PlayerDataSender.h"

PlayerDataSender::PlayerDataSender(Player& player): player(player){}

PlayerDataSender::~PlayerDataSender(){}

void PlayerDataSender::run(){
    while (true){
        std::unique_lock<std::mutex> lock(this->mutex);
        while (this->queue.empty() && this->running){
            this->condition_variable.wait(lock);
        }

        if (!this->running){
            break;
        }
        try{
            this->player.getProtocol().sendBuffer(this->queue.front());
            this->queue.pop();
        } catch(const SocketException& e){
            this->player.disconnect();
        }
    }
}

void PlayerDataSender::sendData(Buffer buffer){
    std::unique_lock<std::mutex> lock(this->mutex);
    this->queue.push(buffer);
}

void PlayerDataSender::notify(){
    this->condition_variable.notify_one();
}

void PlayerDataSender::stop(){
    Thread::stop();
    this->notify();
}
```

jun 10, 18 16:09 **PlayerDataSender.h** Page 1/1

```
#ifndef __PLAYERDATASENDER_H__
#define __PLAYERDATASENDER_H__

#include "Thread.h"
#include "Player.h"
#include "Buffer.h"
#include <mutex>
#include <condition_variable>
#include <queue>

//Cola bloqueante para enviar datos a un jugador
class PlayerDataSender: public Thread{
    private:
        std::mutex mutex;
        std::condition_variable condition_variable;
        Player& player;
        std::queue<Buffer> queue;

    public:
        explicit PlayerDataSender(Player& player);

        ~PlayerDataSender();

        //Envia datos al jugador
        void run() override;

        //Agrega un nuevo dato a la cola
        void sendData(Buffer buffer);

        //Notifica que hay nuevos datos
        void notify();

        //Termina el envio de datos
        void stop() override;
};

#endif
```

jun 10, 18 16:09 **ServerProtocol.cpp** Page 1/3

```
#include "ServerProtocol.h"
#include "Game.h"
#include "Weapon.h"
#include "Girder.h"
#include "ObjectSizes.h"
#include "Player.h"
#include "DataSender.h"
#include <string>

ServerProtocol::ServerProtocol(Socket&& socket): Protocol(std::move(socket)){}

ServerProtocol::ServerProtocol(ServerProtocol&& other):
    Protocol(std::move(other)) {}

ServerProtocol::~ServerProtocol(){}

Buffer ServerProtocol::sendObject(physical_object_ptr& object){
    Buffer buffer;
    buffer.setNext(MOVING_OBJECT);

    const std::string& type = object->getType();
    if (type == TYPE_WORM){
        ServerProtocol::send_worm(object, buffer);
    } else if (type == TYPE_WEAPON){
        ServerProtocol::send_weapon(object, buffer);
    }
    return std::move(buffer);
}

Buffer ServerProtocol::sendDeadObject(physical_object_ptr& object){
    Buffer buffer;
    buffer.setNext(DEAD_OBJECT);

    const std::string& type = object->getType();
    if (type == TYPE_WORM){
        buffer.setNext(WORM_TYPE);
    } else if (type == TYPE_WEAPON){
        buffer.setNext(WEAPON_TYPE);
    }

    uint32_t id = object->getId();
    ServerProtocol::sendIntBuffer(buffer, id);

    return std::move(buffer);
}

void ServerProtocol::send_worm(physical_object_ptr& object, Buffer& buffer){
    Worm* worm = (Worm*)object.get();
    buffer.setNext(WORM_TYPE);
    int32_t id = worm->getId();

    b2Vec2 position = worm->getPosition();

    ServerProtocol::sendIntBuffer(buffer, id);
    ServerProtocol::sendIntBuffer(buffer, worm->getPlayerId());
    ServerProtocol::sendIntBuffer(buffer, position.x * UNIT_TO_SEND);
    ServerProtocol::sendIntBuffer(buffer, position.y * UNIT_TO_SEND);
    ServerProtocol::sendIntBuffer(buffer, worm->getLife());
    buffer.setNext(worm->getDir());
    buffer.setNext(worm->isColliding());
}

void ServerProtocol::send_weapon(physical_object_ptr& object, Buffer& buffer){
    buffer.setNext(WEAPON_TYPE);
    ServerProtocol::sendIntBuffer(buffer, object->getId());

```

```
67
68      b2Vec2 position = object->getPosition();
69      Weapon* weapon = (Weapon*)object.get();
70      std::string name = weapon->getName();
71
72      ServerProtocol::sendStringBuffer(buffer, name);
73      ServerProtocol::sendIntBuffer(buffer, position.x * UNIT_TO_SEND);
74      ServerProtocol::sendIntBuffer(buffer, position.y * UNIT_TO_SEND);
75  }
76
77  Buffer ServerProtocol::sendStartGame(){
78      Buffer buffer;
79      buffer.setNext(START_GAME_ACTION);
80      return buffer;
81  }
82
83  Buffer ServerProtocol::sendEndTurn(){
84      Buffer buffer;
85      buffer.setNext(END_TURN);
86      return buffer;
87  }
88
89  Buffer ServerProtocol::sendStartTurn(int worm_id, int player_id, float wind){
90      Buffer buffer;
91      buffer.setNext(START_TURN);
92      ServerProtocol::sendIntBuffer(buffer, worm_id);
93      ServerProtocol::sendIntBuffer(buffer, player_id);
94      ServerProtocol::sendIntBuffer(buffer, wind * UNIT_TO_SEND);
95      return buffer;
96  }
97
98  Buffer ServerProtocol::sendTurnData(int turn_time, int time_after_shoot){
99      Buffer buffer;
100     ServerProtocol::sendIntBuffer(buffer, turn_time);
101     ServerProtocol::sendIntBuffer(buffer, time_after_shoot);
102     return buffer;
103 }
104
105 Buffer ServerProtocol::sendPlayerId(const Player& player){
106     Buffer buffer;
107     ServerProtocol::sendIntBuffer(buffer, player.getId());
108     ServerProtocol::sendStringBuffer(buffer, player.getName());
109     return buffer;
110 }
111
112 Buffer ServerProtocol::sendGirder(physical_object_ptr& object){
113     Girder* girder = (Girder*)object.get();
114
115     Buffer buffer;
116     ServerProtocol::sendIntBuffer(buffer, girder->getSize());
117
118     b2Vec2 position = object->getPosition();
119     ServerProtocol::sendIntBuffer(buffer, position.x * UNIT_TO_SEND);
120     ServerProtocol::sendIntBuffer(buffer, position.y * UNIT_TO_SEND);
121     ServerProtocol::sendIntBuffer(buffer, girder->getRotation());
122     return buffer;
123 }
124
125 Buffer ServerProtocol::sendWeaponAmmo(const std::string& weapon_name, int ammo){
126     Buffer buffer;
127     ServerProtocol::sendStringBuffer(buffer, weapon_name);
128     ServerProtocol::sendIntBuffer(buffer, ammo);
129     return buffer;
130 }
131
132 Buffer ServerProtocol::sendWeaponChanged(const std::string &weapon){
```



```
133     Buffer buffer;
134     buffer.setNext(CHANGE_WEAPON_ACTION);
135     ServerProtocol::sendStringBuffer(buffer, weapon);
136     return buffer;
137 }
138
139 Buffer ServerProtocol::sendWeaponShot(const std::string &weapon){
140     Buffer buffer;
141     buffer.setNext(SHOOT_WEAPON_ACTION);
142     ServerProtocol::sendStringBuffer(buffer, weapon);
143     return buffer;
144 }
145
146 Buffer ServerProtocol::sendMoveAction(char action){
147     Buffer buffer;
148     buffer.setNext(MOVE_ACTION);
149     buffer.setNext(action);
150     return buffer;
151 }
152
153 Buffer ServerProtocol::sendUpdateScope(int angle) {
154     Buffer buffer;
155     buffer.setNext(MOVE_SCOPE);
156     ServerProtocol::sendIntBuffer(buffer, angle);
157     return buffer;
158 }
159
160 Buffer ServerProtocol::sendEndGame(const std::string& winner){
161     Buffer buffer;
162     buffer.setNext(END_GAME);
163     ServerProtocol::sendStringBuffer(buffer, winner);
164     return buffer;
165 }
```

jun 10, 18 16:09 **ServerProtocol.h** Page 1/2

```
1  #ifndef __SERVERPROTOCOL_H__
2  #define __SERVERPROTOCOL_H__
3
4  #include "Socket.h"
5  #include "Protocol.h"
6  #include "PhysicalObject.h"
7  #include <mutex>
8  #include <string>
9
10 class Player;
11
12 class ServerProtocol : public Protocol{
13     private:
14         //Carga los datos del gusano en el buffer
15         static void send_worm(physical_object_ptr& object, Buffer& buffer);
16
17         //Carga los datos del arma en el buffer
18         static void send_weapon(physical_object_ptr& weapon, Buffer& buffer);
19
20     public:
21         explicit ServerProtocol(Socket&& socket);
22         ServerProtocol(ServerProtocol&& other);
23         ~ServerProtocol();
24
25         //Carga un nuevo objeto en el buffer
26         static Buffer sendObject(physical_object_ptr& object);
27
28         //Carga la informacion de un objeto muerto en el buffer
29         static Buffer sendDeadObject(physical_object_ptr& object);
30
31         //Carga la informacion de comienzo de juego
32         static Buffer sendStartGame();
33
34         //Carga la informacion de nuevo turno en el buffer
35         static Buffer sendStartTurn(int worm_id, int player_id, float wind);
36
37         //Carga la informacion del turno en el buffer
38         static Buffer sendTurnData(int turn_time, int time_after_shoot);
39
40         //Carga la informacion de un nuevo jugador en el buffer
41         static Buffer sendPlayerId(const Player& player);
42
43         //Carga la informacion de una viga en el buffer
44         static Buffer sendGirder(physical_object_ptr& girder);
45
46         //Carga la informacion de un arma en el buffer
47         static Buffer sendWeaponAmmo(const std::string& weapon_name, int ammo);
48
49         //Carga la informacion de cambio de arma en el buffer
50         static Buffer sendWeaponChanged(const std::string &weapon);
51
52         //Carga la informacion de arma disparada en el buffer
53         static Buffer sendWeaponShot(const std::string &weapon);
54
55         //Carga la informacion de que el gusano salto
56         static Buffer sendMoveAction(char action);
57
58         //Carga la informacion de cambio de angulo en el buffer
59         static Buffer sendUpdateScope(int angle);
60
61         //Carga la informacion de fin del juego en el buffer
62         static Buffer sendEndGame(const std::string& winner);
63
64         //Carga la informacion de fin del turno
65         static Buffer sendEndTurn();
66 };
```

jun 10, 18 16:09 **ServerProtocol.h** Page 2/2

```
67
68 #endif
```

jun 20, 18 18:42 **Game.cpp** Page 1/3

```cpp
#include "Game.h"
#include "Girder.h"
#include "WeaponFactory.h"
#include "Server.h"
#include <map>
#include <string>
#include <vector>

#define TURN_STEP 100 //milliseconds

Game::Game(size_t players, const std::string& config_file,
                    const std::string& map, Server& server):
    players(players), server(server),
    parameters(config_file, map),
    world(this->parameters){
        this->running = true;
    }

Game::~Game(){
    this->world.stop();
    this->world.join();
    if (data_sender){
        this->data_sender->stop();
        this->data_sender->join();
    }
}

bool Game::addPlayer(Player& player){
    if (this->isFull()){
        return false;
    }

    return this->turn.addPlayer(player);
}

bool Game::isFull(){
    return this->players <= this->turn.getPlayersSize();
}

bool Game::playerCanJoin(const std::string& player_name){
    if (this->isFull()){
        return false;
    }
    return this->turn.playerCanJoin(player_name);
}

void Game::run(){
    this->configure();
    this->world.start();
    this->data_sender->start();

    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    this->waitToWorld();

    while (!this->turn.gameEnded(this->world.getMutex())){
        this->player_turn_active = true;
        this->turn.beginTurn();
        int worm_id = this->turn.getCurrentPlayer().getCurrentWorm().getId();
        int player_id = this->turn.getCurrentPlayer().getId();
        this->data_sender->sendStartTurn(worm_id, player_id, this->world.getWind());

        size_t current_turn_time = 0;
        size_t max_turn_time = this->parameters.getTurnTime() * 1000;
        bool time_reduced = false;
        while(current_turn_time < max_turn_time){
```

jun 20, 18 18:42 **Game.cpp** Page 2/3

```cpp
            std::this_thread::sleep_for(std::chrono::milliseconds(TURN_STEP));
            current_turn_time += TURN_STEP;
            Worm& current_worm = this->turn.getCurrentPlayer().getCurrentWorm();
            if (current_worm.damageReceived() || this->turn.gameEnded(world.getMutex())){
                current_turn_time = max_turn_time;
            }else if (!time_reduced && current_worm.hasShot()){
                current_turn_time = max_turn_time - parameters.getTimeAfterShoot() * 1000;
                time_reduced = true;
            }
        }

        this->turn.endTurn();
        this->data_sender->sendEndTurn();
        this->waitToWorld();
        this->world.update();
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(50));
    this->data_sender->sendEndGame(this->turn.getWinner());
    this->world.stop();
    this->data_sender->stop();
    this->data_sender->join();
    for (Player& player: this->turn.getPlayers()){
        if (player.isConnected()){
            this->server.addConnectedClient(std::move(player.getProtocol()));
        }
    }
    this->running = false;
}

void Game::configure(){
    DataSender* s = new DataSender(world, turn.getPlayers(), parameters);
    this->data_sender.reset(s);
    this->turn.startGame(*this->data_sender);

    this->data_sender->sendStartGame();
    this->data_sender->sendBackgroundImage(this->parameters.getBackgroundImage());
    int turn_time = this->parameters.getTurnTime();
    int time_after_shoot = this->parameters.getTimeAfterShoot();
    this->data_sender->sendTurnData(turn_time, time_after_shoot);
    this->data_sender->sendPlayersId();

    //Asignacion de gusanos
    std::vector<b2Vec2>& worms_list = this->parameters.getWorms();
    size_t size = worms_list.size();
    for (size_t i = 0; i < size; i++){
        this->turn.addWorm(this->world, this->parameters, worms_list[i], i);
    }
    this->turn.distributeWorms(size, this->parameters.getWormsLifeToAdd());

    //Creacion de vigas
    int max_height = 0;
    std::vector<GirderParams>& list = this->parameters.getGirders();
    size = list.size();
    for (size_t i = 0; i < size; i++){
        Girder* g = new Girder(world, parameters, list[i].len, list[i].rotation);
        physical_object_ptr girder(g);
        this->world.addObject(girder, b2Vec2(list[i].pos_x, list[i].pos_y));
        if (list[i].pos_y > max_height){
            max_height = list[i].pos_y;
        }
    }
    this->parameters.setMaxHeight(max_height);
```

jun 20, 18 18:42 **Game.cpp** Page 3/3

```
128     this->data_sender->sendGirders();
129
130     //Municion de las armas
131     std::map<std::string, unsigned int>& ammo = this->parameters.getWeaponsAmmo(
    );
132     this->data_sender->sendWeaponsAmmo(ammo);
133     for (Player& player: this->turn.getPlayers()){
134         player.setWeaponsAmmo(ammo);
135     }
136 }
137
138 void Game::endTurn(){
139     this->player_turn_active = false;
140 }
141
142 void Game::waitToWorld(){
143     while (this->world.isActive() || this->data_sender->isActive()){
144         int sleep = this->parameters.getGameWaitingWorldSleep();
145         std::this_thread::sleep_for(std::chrono::milliseconds(sleep));
146     }
147 }
```

jun 10, 18 19:29 **Game.h** Page 1/1

```
1  #ifndef __GAME_H__
2  #define __GAME_H__
3
4  #include <vector>
5  #include <memory>
6  #include "Turn.h"
7  #include "GameParameters.h"
8  #include "Thread.h"
9  #include "Player.h"
10 #include "Worm.h"
11 #include "World.h"
12 #include "DataSender.h"
13 #include <string>
14
15 class Player;
16 class Server;
17
18 class Game: public Thread{
19     private:
20         size_t players;
21         Server& server;
22         GameParameters parameters;
23         World world;
24         Turn turn;
25         std::unique_ptr<DataSender> data_sender;
26         bool player_turn_active;
27
28         /* Realiza la configuracion inicial de la partida */
29         void configure();
30
31         /* Espera a que los objetos dejen de moverse */
32         void waitToWorld();
33
34         typedef const std::string string;
35
36     public:
37         /* Constructor */
38         Game(size_t players, string& config_file, string& map, Server& server);
39
40         /* Destructor */
41         ~Game();
42
43         /* Agrega un jugador a la partida */
44         bool addPlayer(Player& player);
45
46         /* Devuelve true si la partida esta llena */
47         bool isFull();
48
49         /* Devuelve true si el jugador puede unirse a la partida */
50         bool playerCanJoin(const std::string& player_name);
51
52         /* Comienza la partida */
53         void run() override;
54
55         /* Finaliza el turno */
56         void endTurn();
57 };
58
59 #endif
```

```cpp
#include "GameParameters.h"
#include "ConfigFields.h"
#include "Path.h"
#include <algorithm>
#include <random>
#include <map>
#include <string>
#include <vector>

#define WORLD_MAX_HEIGHT "world_max_height"

typedef std::vector<std::vector<float>> worms_vector;
typedef std::vector<std::vector<float>> girders_vector;

GameParameters::GameParameters(const std::string& config_file,
                               const std::string& config_editor_file){
    //Compruebo que existan todos los parametros necesarios
    YAML::Node config(YAML::LoadFile(config_file));
    YAML::Node config_editor(YAML::LoadFile(config_editor_file));

    params[DATA_SENDER_SLEEP] = config[DATA_SENDER_SLEEP].as<float>();
    params[GAME_WAIT_WORLD_SLEEP] = config[GAME_WAIT_WORLD_SLEEP].as<float>();
    params[WORLD_SLEEP_AFTER_STEP] = config[WORLD_SLEEP_AFTER_STEP].as<float>();
    params[WORLD_TIME_STEP] = config[WORLD_TIME_STEP].as<float>();
    params[TURN_TIME] = config[TURN_TIME].as<float>();
    params[TIME_AFTER_SHOOT] = config[TIME_AFTER_SHOOT].as<float>();

    params[WORMS_LIFE] = config_editor[WORMS_LIFE].as<float>();
    params[WORMS_LIFE_TO_ADD] = config[WORMS_LIFE_TO_ADD].as<float>();
    params[WORM_VELOCITY] = config[WORM_VELOCITY].as<float>();
    params[WORM_EXPLOSION_VELOCITY] = config[WORM_EXPLOSION_VELOCITY].as<float>(
);
    params[WORM_JUMP_VELOCITY] = config[WORM_JUMP_VELOCITY].as<float>();
    params[WORM_ROLLBACK_VELOCITY] = config[WORM_ROLLBACK_VELOCITY].as<float>();
    params[WORM_JUMP_HEIGHT] = config[WORM_JUMP_HEIGHT].as<float>();
    params[WORM_ROLLBACK_HEIGHT] = config[WORM_ROLLBACK_HEIGHT].as<float>();
    params[WORM_HEIGHT_TO_DAMAGE] = config[WORM_HEIGHT_TO_DAMAGE].as<float>();
    params[WORM_MAX_HEIGHT_DAMAGE] = config[WORM_MAX_HEIGHT_DAMAGE].as<float>();
    params[WEAPONS_VELOCITY] = config[WEAPONS_VELOCITY].as<float>();
    params[WIND_MIN_VELOCITY] = config[WIND_MIN_VELOCITY].as<float>();
    params[WIND_MAX_VELOCITY] = config[WIND_MAX_VELOCITY].as<float>();
    params[GRAVITY] = config[GRAVITY].as<float>();
    params[AIR_MISSILES_SEPARATION] = config[AIR_MISSILES_SEPARATION].as<float>(
);
    params[GIRDER_ANGLE_FRICTION] = config[GIRDER_ANGLE_FRICTION].as<float>();
    params[WORLD_MAX_HEIGHT] = 99999;

    weapon_radius = config[WEAPON_RADIUS].as<std::map<std::string, int>>();
    weapon_ammo =
                config_editor[WEAPON_AMMO].as<std::map<std::string, unsigned int
>>();
    weapon_damage = config[WEAPON_DAMAGE].as<std::map<std::string, int>>();
    weapon_fragments = config[WEAPON_FRAGMENTS].as<std::map<std::string, int>>()
;

    worms_vector worms_file = config_editor[WORMS_DATA].as<worms_vector>();
    for (std::vector<float>& worm: worms_file){
        this->worms.push_back(b2Vec2(worm[0], worm[1]));
    }

    girders_vector girders_file = config_editor[GIRDERS_DATA].as<girders_vector>
();
    for (std::vector<float>& girder: girders_file){
        this->girders.push_back(GirderParams(girder[0], girder[1], girder[2], gi
rder[3]));
    }
```



```cpp

    std::vector<int> bg = config_editor[BACKGROUND_IMAGE].as<std::vector<int>>()
;
    Buffer buffer(bg.size());
    for (int byte: bg){
        buffer.setNext(byte);
    }
    this->background_image = std::move(buffer);
}

GameParameters::~GameParameters(){}

int GameParameters::getWormLife(){
    return this->params[WORMS_LIFE];
}

int GameParameters::getWormsLifeToAdd(){
    return this->params[WORMS_LIFE_TO_ADD];
}

std::vector<b2Vec2>& GameParameters::getWorms(){
    std::random_device rd;
    std::mt19937 random(rd());

    std::shuffle(this->worms.begin(), this->worms.end(), random);
    return this->worms;
}

std::vector<GirderParams>& GameParameters::getGirders(){
    return this->girders;
}

std::map<std::string, unsigned int>& GameParameters::getWeaponsAmmo(){
    return this->weapon_ammo;
}

float GameParameters::getWormVelocity(){
    return this->params[WORM_VELOCITY];
}

float GameParameters::getWormExplosionVelocity(){
    return this->params[WORM_EXPLOSION_VELOCITY];
}

float GameParameters::getWormJumpVelocity(){
    return this->params[WORM_JUMP_VELOCITY];
}

float GameParameters::getWormRollbackVelocity(){
    return this->params[WORM_ROLLBACK_VELOCITY];
}

float GameParameters::getWormJumpHeight(){
    return this->params[WORM_JUMP_HEIGHT];
}

float GameParameters::getWormRollbackHeight(){
    return this->params[WORM_ROLLBACK_HEIGHT];
}

int GameParameters::getWormHeightToDamage(){
    return this->params[WORM_HEIGHT_TO_DAMAGE];
}

int GameParameters::getWormMaxHeightDamage(){
    return this->params[WORM_MAX_HEIGHT_DAMAGE];
```

```
126 }
127
128 float GameParameters::getWeaponsVelocity(){
129     return this->params[WEAPONS_VELOCITY];
130 }
131
132 int GameParameters::getWeaponDamage(const std::string& weapon){
133     return this->weapon_damage[weapon];
134 }
135
136 int GameParameters::getWeaponRadius(const std::string& weapon){
137     return this->weapon_radius[weapon];
138 }
139
140 int GameParameters::getWeaponFragments(const std::string& weapon){
141     return this->weapon_fragments[weapon];
142 }
143
144 float GameParameters::getWindMinVelocity(){
145     return this->params[WIND_MIN_VELOCITY];
146 }
147
148 float GameParameters::getWindMaxVelocity(){
149     return this->params[WIND_MAX_VELOCITY];
150 }
151
152 float GameParameters::getGravity(){
153     return this->params[GRAVITY];
154 }
155
156 float GameParameters::getAirMissilesSeparation(){
157     return this->params[AIR_MISSILES_SEPARATION];
158 }
159
160 int GameParameters::getMaxGirderRotationToFriction(){
161     return this->params[GIRDER_ANGLE_FRICTION];
162 }
163
164 void GameParameters::setMaxHeight(int height){
165     this->params[WORLD_MAX_HEIGHT] = height + 10;
166 }
167
168 int GameParameters::getMaxHeight(){
169     return this->params[WORLD_MAX_HEIGHT];
170 }
171
172 int GameParameters::getDataSenderSleep(){
173     return this->params[DATA_SENDER_SLEEP];
174 }
175
176 int GameParameters::getGameWaitingWorldSleep(){
177     return this->params[GAME_WAIT_WORLD_SLEEP];
178 }
179
180 int GameParameters::getWorldSleepAfterStep(){
181     return this->params[WORLD_SLEEP_AFTER_STEP];
182 }
183
184 float GameParameters::getWorldTimeStep(){
185     return this->params[WORLD_TIME_STEP];
186 }
187
188 int GameParameters::getTurnTime(){
189     return this->params[TURN_TIME];
190 }
191
```



```
192 int GameParameters::getTimeAfterShoot(){
193     return this->params[TIME_AFTER_SHOOT];
194 }
195
196 Buffer& GameParameters::getBackgroundImage(){
197     return this->background_image;
198 }
199
200 GameParameters::GirderParams::GirderParams(size_t len, float pos_x,
201                                            float pos_y, int rotation):
202     len(len), pos_x(pos_x), pos_y(pos_y), rotation(rotation){}
```

```
1   #ifndef __GAMEPARAMETERS_H__
2   #define __GAMEPARAMETERS_H__
3
4   #include <string>
5   #include <vector>
6   #include <map>
7   #include "b2Math.h"
8   #include "yaml.h"
9   #include "Buffer.h"
10
11  // Clase que lee los archivos de configuracion
12  // y devuelve los parametros obtenidos
13  class GameParameters{
14      public:
15          class GirderParams;
16
17      private:
18          std::map<std::string, float> params;
19          std::map<std::string, int> weapon_radius;
20          std::map<std::string, unsigned int> weapon_ammo;
21          std::map<std::string, int> weapon_damage;
22          std::map<std::string, int> weapon_fragments;
23
24          std::vector<b2Vec2> worms;
25          std::vector<GirderParams> girders;
26          Buffer background_image;
27
28      public:
29          //Inicializa todos los parametros necesarios para la partida
30          GameParameters(const std::string& config, const std::string& editor_file
    );
31          ~GameParameters();
32
33          //Devuelve la vida del worm
34          int getWormLife();
35
36          //Devuelve la vida a agregar de los worms
37          int getWormsLifeToAdd();
38
39          //Devuelve los worms del mapa
40          std::vector<b2Vec2>& getWorms();
41
42          //Devuelve la vigas del mapa
43          std::vector<GirderParams>& getGirders();
44
45          //Devuelve la municion de las armas
46          std::map<std::string, unsigned int>& getWeaponsAmmo();
47
48          //Devuelve la velocidad del worm
49          float getWormVelocity();
50
51          //Devuelve la velocidad del worm debido a una explosion
52          float getWormExplosionVelocity();
53
54          //Devuelve la velocidad de salto del worm
55          float getWormJumpVelocity();
56
57          //Devuelve la velocidad del rollback del worm
58          float getWormRollbackVelocity();
59
60          //Devuelve la altura de salto del worm
61          float getWormJumpHeight();
62
63          //Devuelve la altura del rollback del worm
64          float getWormRollbackHeight();
65
```



```
66
67          //Devuelve la altura en la cual el worm sufre daÃ±o
68          int getWormHeightToDamage();
69
70          //Devuelve el daÃ±o maximo por caida
71          int getWormMaxHeightDamage();
72
73
74          //Devuelve la velocidad del arma
75          float getWeaponsVelocity();
76
77          //Devuelve el daÃ±o del arma
78          int getWeaponDamage(const std::string& weapon);
79
80          //Devuelve el radio de daÃ±o del arma
81          int getWeaponRadius(const std::string& weapon);
82
83          //Devuelve la cantidad de fragmentos del arma
84          int getWeaponFragments(const std::string& weapon);
85
86          //Devuelve la velocidad minima del viento
87          float getWindMinVelocity();
88
89          //Devuelve la velocidad maxima del viento
90          float getWindMaxVelocity();
91
92          //Devuelve la gravedad
93          float getGravity();
94
95          //Devuelve la separacion de los misiles aereos
96          float getAirMissilesSeparation();
97
98          //Devuelve la rotacion maxima para la cual
99          //el gusano no desliza
100         int getMaxGirderRotationToFriction();
101
102         //Establece la altura maxima
103         void setMaxHeight(int height);
104
105         //Devuelve la altura maxima
106         int getMaxHeight();
107
108         //Devuelve el tiempo de sleep del DataSender
109         int getDataSenderSleep();
110
111         //Devuelve el tiempo de sleep del World
112         int getGameWaitingWorldSleep();
113
114         //Devuelve el tiempo de sleep del step del World
115         int getWorldSleepAfterStep();
116
117         //Devuelve el time step del World
118         float getWorldTimeStep();
119
120         //Devuelve el tiempo del turno
121         int getTurnTime();
122
123         //Devuelve el tiempo adicional luego de un disparo
124         int getTimeAfterShoot();
125
126         //Devuelve la imagen de fondo
127         Buffer& getBackgroundImage();
128 };
129
130 class GameParameters::GirderParams{
131     public:
```

jun 19, 18 14:51 **GameParameters.h** Page 3/3

```
132         size_t len;
133         float pos_x;
134         float pos_y;
135         int rotation;
136 
137         GirderParams(size_t len, float pos_x, float pos_y, int rotation);
138 };
139 
140 typedef GameParameters::GirderParams GirderParams;
141 
142 #endif
```

jun 10, 18 19:29 **Player.cpp** Page 1/2

```
1  #include "Player.h"
2  #include <map>
3  #include <string>
4  
5  Player::Player(ServerProtocol&& protocol): protocol(std::move(protocol)),
6      id(-1), connected(true){}
7  
8  Player::Player(Player&& other):
9      protocol(std::move(other.protocol)), name(std::move(other.name)),
10     worms(std::move(other.worms)), id(other.id), connected(other.connected){}
11 
12 Player::~Player(){}
13 
14 void Player::setId(int id){
15     this->id = id;
16 }
17 
18 int Player::getId() const{
19     return this->id;
20 }
21 
22 Worm& Player::getCurrentWorm(){
23     return this->worms.getCurrentWorm();
24 }
25 
26 void Player::beginTurn(){
27     this->worms.beginTurn();
28 }
29 
30 void Player::addWorm(World& world,
31                      GameParameters& params, const b2Vec2& position, int id){
32     physical_object_ptr worm(new Worm(world, params, id, this->id, this->weapons
   ));
33     this->worms.add(worm);
34     world.addObject(worm, position);
35 }
36 
37 void Player::distributeWorms(size_t max, int life_to_add){
38     this->worms.distribute(max, life_to_add);
39 }
40 
41 bool Player::isDead(){
42     return this->worms.isEmpty();
43 }
44 
45 ServerProtocol& Player::getProtocol(){
46     return this->protocol;
47 }
48 
49 void Player::setName(const std::string& name){
50     this->name = name;
51 }
52 
53 const std::string& Player::getName() const{
54     return this->name;
55 }
56 
57 bool Player::isConnected() const{
58     return this->connected;
59 }
60 
61 void Player::disconnect(){
62     this->connected = false;
63     this->worms.kill();
64 }
65 
```

jun 10, 18 19:29 **Player.cpp** Page 2/2

```
66  void Player::setWeaponsAmmo(const std::map<std::string, unsigned int>& ammo){
67      this->weapons.updateAmmo(ammo);
68  }
69
70  void Player::changeWeapon(const std::string& weapon){
71      this->weapons.changeWeapon(weapon);
72  }
```

jun 10, 18 19:29 **Player.h** Page 1/2

```
1   #ifndef __PLAYER_H__
2   #define __PLAYER_H__
3
4   #include "WormsList.h"
5   #include "ServerProtocol.h"
6   #include "Worm.h"
7   #include "World.h"
8   #include "GameParameters.h"
9   #include "WeaponList.h"
10  #include <string>
11  #include <map>
12
13  class Player{
14      private:
15          ServerProtocol protocol;
16          std::string name;
17          WormsList worms;
18          WeaponList weapons;
19          int id;
20          bool connected;
21
22      public:
23          explicit Player(ServerProtocol&& protocol);
24
25          Player(Player&& other);
26
27          ~Player();
28
29          //Setea el id del jugador por el pasado
30          void setId(int id);
31
32          //Devuelve el id del jugador
33          int getId() const;
34
35          //Devuelve el gusano actual del jugador
36          Worm& getCurrentWorm();
37
38          //Empieza el turno del jugador
39          void beginTurn();
40
41          //Agrega un nuevo gusano al jugador
42          void addWorm(World& world, GameParameters& param, const b2Vec2& pos, int
        id);
43
44          //Agrega vida a los gusanos del jugador
45          //en caso de que tenga menos gusanos que otros jugadores
46          void distributeWorms(size_t max, int life_to_add);
47
48          //Devuelve true si el jugador esta muerto
49          bool isDead();
50
51          //Devuelve true si el jugador esta desconectado
52          bool isConnected() const;
53
54          //Desconecta al jugador
55          void disconnect();
56
57          //Setea la municion de las armas
58          void setWeaponsAmmo(const std::map<std::string, unsigned int>& ammo);
59
60          //Cambia el arma actual del jugador
61          void changeWeapon(const std::string& weapon);
62
63          //Setea el nombre del jugador
64          void setName(const std::string& name);
65
```

jun 10, 18 19:29 **Player.h** Page 2/2

```
66          //Devuelve el nombre del jugador
67          const std::string& getName() const;
68
69          //Devuelve el protocolo del jugador
70          ServerProtocol& getProtocol();
71  };
72
73  #endif
```

jun 20, 18 18:42 **Turn.cpp** Page 1/2

```
1   #include "Turn.h"
2   #include <string>
3   #include <vector>
4
5   Turn::Turn(): current(0){}
6
7   Turn::~Turn(){
8       for (std::unique_ptr<PlayerDataReceiver>& receiver: this->receivers){
9           receiver->stop();
10          receiver->join();
11      }
12  }
13
14  bool Turn::addPlayer(Player& player){
15      if (!this->playerCanJoin(player.getName())){
16          return false;
17      }
18      player.setId(this->players.size());
19      player.getProtocol().sendChar(true);
20      this->players.push_back(std::move(player));
21      return true;
22  }
23
24  bool Turn::playerCanJoin(const std::string& player_name){
25      for (Player& player: this->players){
26          if (player.getName() == player_name){
27              return false;
28          }
29      }
30      return true;
31  }
32
33  size_t Turn::getPlayersSize() const{
34      return this->players.size();
35  }
36
37  Player& Turn::getCurrentPlayer(){
38      return this->players.at(this->current);
39  }
40
41  void Turn::startGame(DataSender& data_sender){
42      for (Player& player: this->players){
43          PlayerDataReceiver* r = new PlayerDataReceiver(player, data_sender);
44          std::unique_ptr<PlayerDataReceiver> receiver(r);
45          receiver->start();
46          this->receivers.push_back(std::move(receiver));
47      }
48  }
49
50  void Turn::beginTurn(){
51      do {
52          this->advanceCurrent();
53      } while (this->getCurrentPlayer().isDead());
54      this->getCurrentPlayer().beginTurn();
55      this->receivers[this->current]->beginTurn();
56  }
57
58  void Turn::endTurn(){
59      this->receivers[this->current]->endTurn();
60  }
61
62  std::vector<Player>& Turn::getPlayers(){
63      return this->players;
64  }
65
66  void Turn::advanceCurrent(){
```

jun 20, 18 18:42 **Turn.cpp** Page 2/2

```
67      this->current++;
68      if (this->current >= this->players.size()){
69          this->current = 0;
70      }
71  }
72
73  void Turn::addWorm(World& world, GameParameters& params, b2Vec2 pos, int id){
74      this->players[this->current].addWorm(world, params, pos, id);
75      this->advanceCurrent();
76  }
77
78  void Turn::distributeWorms(size_t size, int life_to_add){
79      int quantity = (size / this->players.size());
80      if (size % this->players.size() != 0){
81          quantity += 1;
82      }
83
84      for (Player& player: this->players){
85          player.distributeWorms(quantity, life_to_add);
86      }
87  }
88
89  bool Turn::gameEnded(std::mutex& mutex){
90      std::lock_guard<std::mutex> lock(mutex);
91      this->winner.clear();
92      size_t players_alive = 0;
93      for (Player& player: this->players){
94          if (!player.isDead()){
95              players_alive++;
96              this->winner = player.getName();
97          }
98      }
99      return players_alive <= 1;
100 }
101
102 const std::string& Turn::getWinner(){
103     for (std::unique_ptr<PlayerDataReceiver>& receiver: this->receivers){
104         receiver->stop();
105     }
106     return this->winner;
107 }
```

jun 10, 18 19:29 **Turn.h** Page 1/1

```
1   #ifndef __SERVERTURN_H__
2   #define __SERVERTURN_H__
3
4   #include "Player.h"
5   #include "PlayerDataReceiver.h"
6   #include "DataSender.h"
7   #include <vector>
8   #include <string>
9   #include <memory>
10
11  class Turn{
12      private:
13          std::vector<Player> players;
14          std::vector<std::unique_ptr<PlayerDataReceiver>> receivers;
15          std::string winner;
16          size_t current;
17
18          void advanceCurrent();
19
20      public:
21          Turn();
22          ~Turn();
23
24          //Agrega un nuevo jugador
25          bool addPlayer(Player& player);
26
27          //Devuelve true si el jugador se puede unir a la partida
28          bool playerCanJoin(const std::string& player_name);
29
30          //Devuelve la cantidad de jugadores
31          size_t getPlayersSize() const;
32
33          //Devuelve un vector con los jugadores
34          std::vector<Player>& getPlayers();
35
36          //Devuelve el jugador actual
37          Player& getCurrentPlayer();
38
39          //Realiza la configuracion inicial
40          void startGame(DataSender& data_sender);
41
42          //Empieza un nuevo turno, cambiando el jugador actual
43          void beginTurn();
44
45          //Termina el turno del jugador actual
46          void endTurn();
47
48          //Agrega un gusano al proximo jugador
49          void addWorm(World& world, GameParameters& params, b2Vec2 pos, int id);
50
51          //Agrega vida a los jugadores con menos gusanos
52          void distributeWorms(size_t size, int life_to_add);
53
54          //Devuelve true si queda uno o ningun jugador vivo
55          bool gameEnded(std::mutex& mutex);
56
57          //Devuelve el nombre del jugador ganador
58          const std::string& getWinner();
59  };
60
61  #endif
```

jun 10, 18 19:29 **WeaponList.cpp** Page 1/1

```cpp
#include "WeaponList.h"
#include "WeaponNames.h"
#include "WeaponFactory.h"
#include <map>
#include <string>

WeaponList::WeaponList(): current_weapon(DEFAULT_WEAPON){}

WeaponList::~WeaponList(){}

void WeaponList::updateAmmo(const std::map<std::string, unsigned int>& ammo){
    this->ammo = ammo;
}

bool WeaponList::shoot(){
    if (this->ammo[this->current_weapon] == 0){
        return false;
    }
    this->ammo[this->current_weapon]--;
    return true;
}

physical_object_ptr WeaponList::getCurrentWeapon(World& world,
                                        GameParameters& parameters){
    WeaponFactory factory(world, parameters);
    return factory.getWeapon(this->current_weapon);
}

void WeaponList::changeWeapon(const std::string& weapon){
    this->current_weapon = weapon;
}
```

jun 10, 18 19:29 **WeaponList.h** Page 1/1

```cpp
#ifndef __WEAPONLIST_H__
#define __WEAPONLIST_H__

#include <map>
#include <string>
#include "PhysicalObject.h"

class GameParameters;

class WeaponList{
    private:
        std::map<std::string, unsigned int> ammo;
        std::string current_weapon;

    public:
        WeaponList();

        ~WeaponList();

        //Actualiza la municion de las armas
        void updateAmmo(const std::map<std::string, unsigned int>& ammo);

        //Devuelve si puede disparar el arma, y disminuye la municion
        bool shoot();

        //Devuelve el arma actual
        physical_object_ptr getCurrentWeapon(World& world, GameParameters& param
s);

        //Cambia el arma actual
        void changeWeapon(const std::string& weapon);
};

#endif
```

jun 20, 18 18:42 **WormsList.cpp** Page 1/1

```cpp
#include "WormsList.h"

WormsList::WormsList(): current(0){}

WormsList::~WormsList(){}

Worm& WormsList::getCurrentWorm(){
    Worm* worm = (Worm*)this->list[this->current].get();
    return *worm;
}

void WormsList::beginTurn(){
    do {
        this->current++;
        if (this->current >= this->list.size()){
            this->current = 0;
        }
    } while (this->getCurrentWorm().isDead());
    this->getCurrentWorm().beginTurn();
}

void WormsList::add(physical_object_ptr worm){
    this->list.push_back(worm);
}

WormsList::WormsList(WormsList&& other):
    list(std::move(other.list)), current(other.current){}

void WormsList::distribute(size_t max, int life_to_add){
    if (this->list.size() < max){
        for (physical_object_ptr& worm_ptr: this->list){
            Worm* worm = (Worm*)worm_ptr.get();
            worm->addLife(life_to_add);
        }
    }
}

bool WormsList::isEmpty(){
    for (physical_object_ptr& worm: this->list){
        if (!worm->isDead()){
            return false;
        }
    }
    return true;
}

void WormsList::kill(){
    for (physical_object_ptr& worm: this->list){
        if (!worm->isDead()){
            worm->kill();
        }
    }
}
```

jun 09, 18 19:06 **WormsList.h** Page 1/1

```cpp
#ifndef __WORMSLIST_H__
#define __WORMSLIST_H__

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

class WormsList{
    private:
        std::vector<physical_object_ptr> list;
        size_t current;

    public:
        /* Constructor */
        WormsList();

        /* Destructor */
        ~WormsList();

        /* Devuelve el worm actual */
        Worm& getCurrentWorm();

        /* Comienza el turno, cambiando el gusano actual */
        void beginTurn();

        /* Agrega un worm a la lista */
        void add(physical_object_ptr worm);

        /* Constructor por movimiento */
        WormsList(WormsList&& other);

        /* Aumenta la vida de los worms si la cantidad de
         * worms es menor que la de otros jugadores */
        void distribute(size_t max, int life_to_add);

        /* Devuelve true si todos los worms estan muertos */
        bool isEmpty();

        /* Mata a todos los worms */
        void kill();
};

#endif
```

jun 10, 18 19:29 **CollisionData.cpp** Page 1/1

```cpp
#include "CollisionData.h"
#include "PhysicalObject.h"
#include <string>

CollisionData::CollisionData(std::string type, PhysicalObject* object):
    type(type), object(object){}

CollisionData::~CollisionData(){}

const std::string& CollisionData::getType(){
    return this->type;
}

PhysicalObject* CollisionData::getObject(){
    return this->object;
}
```

jun 09, 18 21:42 **CollisionData.h** Page 1/1

```cpp
#ifndef __COLLISIONDATA_H__
#define __COLLISIONDATA_H__

#include <string>


class PhysicalObject;

//Datos de un objeto para determinar colisiones
class CollisionData{
    private:
        std::string type;
        PhysicalObject* object;

    public:
        CollisionData(std::string type, PhysicalObject* object);
        ~CollisionData();

        //Devuelve el tipo del objeto fisico
        const std::string& getType();

        //Devuelve el objeto fisico
        PhysicalObject* getObject();
};

#endif
```

jun 10, 18 19:29 **CollisionListener.cpp** Page 1/2

```
#include "CollisionListener.h"
#include "PhysicalObject.h"
#include "Worm.h"
#include "Girder.h"

CollisionListener::CollisionListener(){}

CollisionListener::~CollisionListener(){}

void CollisionListener::BeginContact(b2Contact* contact){
    CollisionData* dataA =
                (CollisionData*)contact->GetFixtureA()->GetBody()->GetUserData()
;
    CollisionData* dataB =
                (CollisionData*)contact->GetFixtureB()->GetBody()->GetUserData()
;

    if (dataA->getObject()->isDead() || dataB->getObject()->isDead()){
        return;
    }

    if (dataA->getType() == TYPE_WEAPON){
        if (dataB->getType() == TYPE_WORM){
            int shooter_id = ((Weapon*)dataA->getObject())->getShooterId();
            int worm_id = dataB->getObject()->getId();
            if (shooter_id == worm_id){
                return;
            }
        }
        dataA->getObject()->collideWithSomething(dataB);
    } else if (dataB->getType() == TYPE_WEAPON){
        if (dataA->getType() == TYPE_WORM){
            int shooter_id = ((Weapon*)dataB->getObject())->getShooterId();
            int worm_id = dataA->getObject()->getId();
            if (shooter_id == worm_id){
                return;
            }
        }
        dataB->getObject()->collideWithSomething(dataA);
    }

    if (dataA->getType() == TYPE_WORM && contact->GetFixtureA()->IsSensor() &&
            (dataB->getType() == TYPE_GIRDER || dataB->getType() == TYPE_BORDER)
){
        dataA->getObject()->collideWithSomething(dataB);

    } else if (dataB->getType() == TYPE_WORM &&
                contact->GetFixtureB()->IsSensor() &&
                (dataA->getType() == TYPE_GIRDER || dataA->getType() == TYPE_BOR
DER)){
        dataB->getObject()->collideWithSomething(dataA);
    }
}

void CollisionListener::EndContact(b2Contact* contact){
    CollisionData* dataA =
                (CollisionData*)contact->GetFixtureA()->GetBody()->GetUserData()
;
    CollisionData* dataB =
                (CollisionData*)contact->GetFixtureB()->GetBody()->GetUserData()
;

    if (dataA->getType() == TYPE_WORM &&
            contact->GetFixtureA()->IsSensor() && dataB->getType() == TYPE_GIRDE
R){
        bool friction = ((Girder *) dataB->getObject())->hasFriction();
```

jun 10, 18 19:29 **CollisionListener.cpp** Page 2/2

```
        ((Worm *) dataA->getObject())->endCollissionGirder(friction);
    } else if (dataB->getType() == TYPE_WORM &&
                contact->GetFixtureB()->IsSensor() && dataA->getType() == TYPE_G
IRDER){
        bool friction = ((Girder *) dataA->getObject())->hasFriction();
        ((Worm *) dataB->getObject())->endCollissionGirder(friction);
    }

    if (dataA->getType() == TYPE_WEAPON){
        ((Weapon*)dataA->getObject())->removeShooterId();
    }
    if (dataB->getType() == TYPE_WEAPON){
        ((Weapon*)dataB->getObject())->removeShooterId();
    }
}

bool CollisionListener::ShouldCollide(b2Fixture* fixtureA, b2Fixture* fixtureB){
    CollisionData* dataA = (CollisionData*)fixtureA->GetBody()->GetUserData();
    CollisionData* dataB = (CollisionData*)fixtureB->GetBody()->GetUserData();

    if (dataA->getType() == TYPE_WORM && dataB->getType() == TYPE_WORM){
        return false;
    }
    if (dataA->getType() == TYPE_WEAPON && dataB->getType() == TYPE_WEAPON){
        return false;
    }
    return true;
}
```

jun 09, 18 19:06 **CollisionListener.h** Page 1/1

```
#ifndef __COLLISIONLISTENER_H__
#define __COLLISIONLISTENER_H__

#include <string>
#include "CollisionData.h"
#include "b2WorldCallbacks.h"
#include "b2Contact.h"
#include <list>

class CollisionListener: public b2ContactListener, public b2ContactFilter{
    public:
        CollisionListener();
        ~CollisionListener();

        //Analiza la colision entre dos objetos
        void BeginContact(b2Contact* contact) override;

        //Analiza el fin de colision entre dos objetos
        void EndContact(b2Contact* contact) override;

        //Analiza si dos objetos deben colisionar o no
        bool ShouldCollide(b2Fixture* fixtureA, b2Fixture* fixtureB) override;
};

#endif
```

jun 20, 18 18:42 **RayCastWeaponExploded.cpp** Page 1/1

```
#include "RayCastWeaponExploded.h"
#include "Worm.h"

RayCastWeaponExploded::RayCastWeaponExploded(): closest(NULL){}

RayCastWeaponExploded::~RayCastWeaponExploded(){}

b2Body* RayCastWeaponExploded::getClosestWorm(){
    if (!this->closest){
        return NULL;
    }
    CollisionData* data = (CollisionData*)this->closest->GetUserData();
    if (data->getType() != TYPE_WORM){
        this->closest = NULL;
        return NULL;
    }

    this->affected_worms.push_back(this->closest);
    b2Body* closest_worm = this->closest;
    this->closest = NULL;
    return closest_worm;
}

float32 RayCastWeaponExploded::ReportFixture(b2Fixture* fixture,
                        const b2Vec2& point, const b2Vec2& normal, float32 fract
ion){
    b2Body* closest_body = fixture->GetBody();
    for (b2Body* affected_worm: this->affected_worms){
        if (affected_worm == closest_body){
            return -1;
        }
    }
    this->closest = closest_body;
    return fraction;
}
```

jun 10, 18 19:29 **RayCastWeaponExploded.h** Page 1/1

```
#ifndef __RAYCASTWEAPONEXPLODED_H__
#define __RAYCASTWEAPONEXPLODED_H__

#include "b2Body.h"
#include "b2Fixture.h"
#include "b2WorldCallbacks.h"
#include <vector>

class RayCastWeaponExploded: public b2RayCastCallback{
    private:
        std::vector<b2Body*> affected_worms;
        b2Body* closest;

    public:
        RayCastWeaponExploded();
        ~RayCastWeaponExploded();

        //Devuelve el gusano mas cercano a la explosion, si hay
        b2Body* getClosestWorm();

        //Busca al objeto mas cercano a la explosion
        float32 ReportFixture(b2Fixture* fixture, const b2Vec2& point,
                        const b2Vec2& normal, float32 fraction) override;
};

#endif

```

jun 10, 18 19:29 **BottomBorder.cpp** Page 1/1

```
#include "BottomBorder.h"
#include "b2PolygonShape.h"
#include "b2Fixture.h"

BottomBorder::BottomBorder(World& world):
    PhysicalObject(world, 0, TYPE_BORDER){}

BottomBorder::~BottomBorder(){}

void BottomBorder::getBodyDef(b2BodyDef& body_def, const b2Vec2& pos){
    body_def.type = b2_staticBody;
    body_def.position.Set(pos.x, pos.y);
}

void BottomBorder::createFixtures(){
    b2PolygonShape boxShape;
    boxShape.SetAsBox(100000,1);

    b2FixtureDef boxFixtureDef;
    boxFixtureDef.shape = &boxShape;
    boxFixtureDef.density = 1;
    this->body->CreateFixture(&boxFixtureDef);
}
```

jun 10, 18 19:29 **BottomBorder.h** Page 1/1

```
#ifndef __BOTTOMBORDER_H__
#define __BOTTOMBORDER_H__

#include "PhysicalObject.h"
#include <string>

//Determina el borde inferior del mundo
class BottomBorder: public PhysicalObject{
    private:
        std::string type;

    protected:
        void getBodyDef(b2BodyDef& body_def, const b2Vec2& pos) override;
        void createFixtures() override;

    public:
        explicit BottomBorder(World& world);
        ~BottomBorder();
};

#endif
```

jun 10, 18 19:29 **Girder.cpp** Page 1/1

```
#include "Girder.h"
#include "b2PolygonShape.h"
#include "b2Fixture.h"
#include "Math.h"

Girder::Girder(World& world, GameParameters& params, size_t size, int angle):
    PhysicalObject(world, 0, TYPE_GIRDER), size(size), rotation(angle),
    max_rotation_to_friction(params.getMaxGirderRotationToFriction()){}

Girder::~Girder(){}

void Girder::getBodyDef(b2BodyDef& body_def, const b2Vec2& pos){
    body_def.type = b2_staticBody;
    body_def.position.Set(pos.x, pos.y);
}

void Girder::createFixtures(){
    b2PolygonShape boxShape;
    boxShape.SetAsBox(this->size / 2.0, girder_height / 2,
            b2Vec2(0, 0), Math::degreesToRadians(this->rotation));

    b2FixtureDef boxFixtureDef;
    boxFixtureDef.shape = &boxShape;
    boxFixtureDef.density = 1;
    this->body->CreateFixture(&boxFixtureDef);
}

size_t Girder::getSize(){
    return this->size;
}

int Girder::getRotation(){
    return this->rotation;
}

bool Girder::hasFriction(){
    int angle = this->getAngle();
    return angle < this->max_rotation_to_friction || angle == 90;
}

int Girder::getAngle(){
    int angle = this->rotation;
    if (angle > 90){
        angle = 180 - angle;
    }
    return angle;
}
```

jun 10, 18 19:29 **Girder.h** Page 1/1

```
#ifndef __GIRDER_H__
#define __GIRDER_H__

#include "PhysicalObject.h"
#include "GameParameters.h"

class Girder: public PhysicalObject{
    private:
        size_t size;
        int rotation;
        int max_rotation_to_friction;

    protected:
        void getBodyDef(b2BodyDef& body_def, const b2Vec2& pos) override;
        void createFixtures() override;

    public:
        Girder(World& world, GameParameters& parameters, size_t size, int rotati
on);
        ~Girder();

        //Devuelve la longitud de la viga
        size_t getSize();

        //Devuelve la rotacion de la viga
        int getRotation();

        //Devuelve true si la viga tiene friccion
        bool hasFriction();

        //Devuelve la rotacion normalizada
        int getAngle();
};

#endif
```

jun 10, 18 19:29 **PhysicalObject.cpp** Page 1/2

```
#include "PhysicalObject.h"
#include "World.h"
#include <string>

PhysicalObject::PhysicalObject(World& world, int id, const std::string& type):
    world(world), body(NULL), is_dead(false), id(id),
    type(type), last_position(-1, -1),
    last_position_sent(false), data_updated(false),
    collision_data(type, this){}

PhysicalObject::~PhysicalObject(){}

void PhysicalObject::initializeBody(b2Body* body){
    this->body = body;
    this->body->SetUserData(&this->collision_data);
    this->createFixtures();
    this->setInitialVelocity();
}

void PhysicalObject::destroyBody(){
    this->body = NULL;
    this->is_dead = true;
}

b2Vec2 PhysicalObject::getPosition(){
    if (this->body){
        return this->body->GetPosition();
    }
    return b2Vec2(-100, 0);
}

b2Body* PhysicalObject::getBody(){
    return this->body;
}

bool PhysicalObject::isMoving(){
    if (!this->body || this->is_dead){
        return false;
    }
    b2Vec2 pos = this->body->GetPosition();
    int last_x = this->last_position.x * UNIT_TO_SEND;
    int last_y = this->last_position.y * UNIT_TO_SEND;
    bool moved_x = (int)(pos.x * UNIT_TO_SEND) != last_x;
    bool moved_y = (int)(pos.y * UNIT_TO_SEND) != last_y;
    this->last_position = pos;
    bool moved = moved_x || moved_y;
    if (moved || this->data_updated){
        this->last_position_sent = false;
        this->data_updated = false;
        return true;
    }
    if (!this->body->IsAwake() && !this->last_position_sent){
        this->last_position_sent = true;
        this->data_updated = false;
        return true;
    }
    return false;
}

bool PhysicalObject::isActive(){
    if (!this->body){
        return false;
    }
    return this->body->IsAwake();
}

```

jun 10, 18 19:29 **PhysicalObject.cpp** Page 2/2

```cpp
bool PhysicalObject::isDead(){
    return this->is_dead;
}

bool PhysicalObject::isWindAffected(){
    return false;
}

void PhysicalObject::kill(){
    this->is_dead = true;
}

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

const std::string& PhysicalObject::getType(){
    return this->type;
}

void PhysicalObject::setInitialVelocity(){}

void PhysicalObject::collideWithSomething(CollisionData *other){}
```

jun 09, 18 21:42 **PhysicalObject.h** Page 1/2

```cpp
#ifndef __PHYSICALOBJECT_H__
#define __PHYSICALOBJECT_H__

#include "b2Body.h"
#include "CollisionData.h"
#include "ObjectSizes.h"
#include "ObjectTypes.h"
#include <string>
#include <memory>

class World;

class PhysicalObject{
    protected:
        World& world;
        b2Body* body;
        bool is_dead;
        int id;
        const std::string& type;
        b2Vec2 last_position;
        bool last_position_sent;
        bool data_updated;
        CollisionData collision_data;

        virtual void createFixtures() = 0;
        virtual void setInitialVelocity();

    public:
        PhysicalObject(World& world, int id, const std::string& type);
        virtual ~PhysicalObject();

        //Inicializa el cuerpo del objeto
        void initializeBody(b2Body* body);

        //Destruye el cuerpo del objeto
        void destroyBody();

        //Devuelve la posicion del objeto
        b2Vec2 getPosition();

        //Devuelve el cuerpo del objeto
        b2Body* getBody();

        //Devuelve true si el objeto se esta moviendo
        virtual bool isMoving();

        //Devuelve true si el objeto esta activo
        virtual bool isActive();

        //Devuelve true si el objeto esta muerto
        virtual bool isDead();

        //Devuelve true si el objeto es afectado por el viento
        virtual bool isWindAffected();

        //Mata al objeto
        void kill();

        //Devuelve el id del objeto
        int getId();

        //Devuelve el tipo del objeto
        const std::string& getType();

        virtual void getBodyDef(b2BodyDef& body_def, const b2Vec2& pos) = 0;

```

jun 09, 18 21:42 **PhysicalObject.h** Page 2/2

```
67          //Colisiona con otro objeto
68          virtual void collideWithSomething(CollisionData *other);
69  };
70
71  typedef std::shared_ptr<PhysicalObject> physical_object_ptr;
72
73  #endif
```

jun 10, 18 19:29 **AirAttack.cpp** Page 1/1

```
1   #include "AirAttack.h"
2   #include "WeaponFactory.h"
3   #include "Worm.h"
4   #include <string>
5
6   AirAttack::AirAttack(World& world, GameParameters& parameters):
7       Weapon(world, parameters, 0),
8       missiles_separation(parameters.getAirMissilesSeparation()){}
9
10  AirAttack::~AirAttack(){}
11
12  const std::string& AirAttack::getName(){
13      return AIR_ATTACK_NAME;
14  }
15
16  void AirAttack::shoot(char dir, int angle, int power, int time, int shooter){}
17
18  void AirAttack::shoot(Worm& shooter, b2Vec2 pos){
19      int missiles = this->parameters.getWeaponFragments(AIR_ATTACK_NAME);
20      float pos_x = pos.x - missiles * this->missiles_separation / 2;
21      float pos_y = this->parameters.getMaxHeight();
22      WeaponFactory factory(this->world, this->parameters);
23      for (int i = 0; i < missiles; i++, pos_x += this->missiles_separation){
24          physical_object_ptr missile = factory.getWeapon(AIR_ATTACK_MISSILE_NAME)
    ;
25          this->world.addObject(missile, b2Vec2(pos_x, pos_y));
26      };
27  }
```

jun 10, 18 19:29 **AirAttack.h** Page 1/1

```h
#ifndef __SERVERAIRATTACK_H__
#define __SERVERAIRATTACK_H__

#include "Weapon.h"
#include <string>

class AirAttack: public Weapon{
    private:
        float missiles_separation;

    public:
        AirAttack(World& world, GameParameters& parameters);
        ~AirAttack();

        const std::string& getName() override;

        void shoot(char dir, int angle, int power, int time, int shooter_id) override;

        void shoot(Worm& shooter, b2Vec2 pos) override;
};

#endif
```

jun 10, 18 19:29 **AirAttackMissile.cpp** Page 1/1

```cpp
#include "AirAttackMissile.h"
#include <string>

AirAttackMissile::AirAttackMissile(World& world, GameParameters& parameters):
    Weapon(world, parameters,
        parameters.getWeaponDamage(AIR_ATTACK_MISSILE_NAME),
        parameters.getWeaponRadius(AIR_ATTACK_MISSILE_NAME)){}

AirAttackMissile::~AirAttackMissile(){}

const std::string& AirAttackMissile::getName(){
    return AIR_ATTACK_MISSILE_NAME;
}

bool AirAttackMissile::isWindAffected(){
    return true;
}
```

jun 10, 18 19:29 **AirAttackMissile.h** Page 1/1

```
#ifndef __SERVERAIRATTACKMISSILE_H__
#define __SERVERAIRATTACKMISSILE_H__

#include "Weapon.h"
#include <string>

class AirAttackMissile: public Weapon{
    public:
        AirAttackMissile(World& world, GameParameters& parameters);
        ~AirAttackMissile();

        const std::string& getName() override;

        bool isWindAffected() override;
};

#endif
```

jun 10, 18 19:29 **Banana.cpp** Page 1/1

```
#include "Banana.h"
#include "b2Fixture.h"
#include "b2CircleShape.h"
#include <string>

Banana::Banana(World& world, GameParameters& parameters):
    Weapon(world, parameters,
        parameters.getWeaponDamage(BANANA_NAME),
        parameters.getWeaponRadius(BANANA_NAME)){}

Banana::~Banana(){}

const std::string& Banana::getName(){
    return BANANA_NAME;
}

void Banana::createFixtures(){
    b2CircleShape circleShape;
    circleShape.m_p.Set(0, 0);
    circleShape.m_radius = weapon_size / 2;

    b2FixtureDef fixtureDef;
    fixtureDef.shape = &circleShape;
    fixtureDef.density = 4;
    fixtureDef.restitution = 0.9; //rebotable
    this->body->CreateFixture(&fixtureDef);
}
```

jun 10, 18 19:29 **Banana.h** Page 1/1

```
#ifndef __SERVERBANANA_H__
#define __SERVERBANANA_H__

#include "Weapon.h"
#include <string>

class Banana: public Weapon{
    protected:
        void createFixtures() override;

    public:
        Banana(World& world, GameParameters& parameters);
        ~Banana();

        const std::string& getName() override;
};

#endif
```

jun 10, 18 19:29 **Bat.cpp** Page 1/1

```
#include "Bat.h"
#include <string>

Bat::Bat(World& world, GameParameters& parameters):
    Weapon(world, parameters, parameters.getWeaponDamage(BAT_NAME),
        parameters.getWeaponRadius(BAT_NAME)){}

Bat::~Bat(){}

const std::string& Bat::getName(){
    return BAT_NAME;
}

void Bat::setInitialVelocity(){
    this->explode();
}

void Bat::explode(){
    b2Vec2 center = this->body->GetPosition();
    this->attackWormExplosion(center, this->angle);

    this->waiting_to_explode = false;
    this->is_dead = true;
}
```

jun 10, 18 19:29 **Bat.h** Page 1/1

```
1  #ifndef __SERVERBAT_H__
2  #define __SERVERBAT_H__
3
4  #include "Weapon.h"
5  #include <string>
6
7  class Bat: public Weapon{
8      public:
9          Bat(World& world, GameParameters& parameters);
10         ~Bat();
11
12         const std::string& getName() override;
13
14         void setInitialVelocity() override;
15
16         void explode() override;
17 };
18
19 #endif
```

jun 10, 18 19:29 **Bazooka.cpp** Page 1/1

```
1  #include "Bazooka.h"
2  #include <string>
3
4  Bazooka::Bazooka(World& world, GameParameters& parameters):
5      Weapon(world, parameters,
6          parameters.getWeaponDamage(BAZOOKA_NAME),
7          parameters.getWeaponRadius(BAZOOKA_NAME)){}
8
9  Bazooka::~Bazooka(){}
10
11 const std::string& Bazooka::getName(){
12     return BAZOOKA_NAME;
13 }
14
15 bool Bazooka::isWindAffected(){
16     return true;
17 }
```

jun 10, 18 19:29 **Bazooka.h** Page 1/1

```
#ifndef __SERVERBAZOOKA_H__
#define __SERVERBAZOOKA_H__

#include "Weapon.h"
#include <string>

class Bazooka: public Weapon{
    public:
        Bazooka(World& world, GameParameters& parameters);
        ~Bazooka();

        const std::string& getName() override;

        bool isWindAffected() override;
};

#endif
```

jun 10, 18 19:29 **Dynamite.cpp** Page 1/1

```
#include "Dynamite.h"
#include <string>

Dynamite::Dynamite(World& world, GameParameters& parameters):
    Weapon(world, parameters, parameters.getWeaponDamage(DYNAMITE_NAME),
        parameters.getWeaponRadius(DYNAMITE_NAME)){}

Dynamite::~Dynamite(){}

const std::string& Dynamite::getName(){
    return DYNAMITE_NAME;
}
```

jun 10, 18 19:29 **Dynamite.h** Page 1/1

```h
#ifndef __SERVERDYNAMITE_H__
#define __SERVERDYNAMITE_H__

#include "Weapon.h"
#include <string>

class Dynamite: public Weapon{
    public:
        Dynamite(World& world, GameParameters& parameters);
        ~Dynamite();

        const std::string& getName() override;
};

#endif
```

jun 10, 18 19:29 **FragmentableWeapon.cpp** Page 1/1

```cpp
#include "FragmentableWeapon.h"
#include "WeaponFactory.h"
#include "Fragment.h"
#include "Math.h"
#include <string>

FragmentableWeapon::FragmentableWeapon(World& world, GameParameters& parameters,
                                                    int damage, int fragments, i
nt radius):
    Weapon(world, parameters, damage, radius), fragments(fragments){}

FragmentableWeapon::~FragmentableWeapon(){}

void FragmentableWeapon::explode(){
    WeaponFactory factory(this->world, this->parameters);
    for (float angle = 0; angle < 360; angle+= (360 / fragments)){
        physical_object_ptr fragment = factory.getWeapon(this->getName() + FRAGM
ENT);

        b2Vec2 center = this->body->GetPosition() +
                        0.3 * b2Vec2(Math::cosDegrees(angle), Math::sinDegrees(a
ngle));
        ((Fragment *) fragment.get())->setShootPosition(center);
        ((Fragment*)fragment.get())->shoot(angle);
        this->world.addWeaponFragment(fragment);
    }
    Weapon::explode();
}
```

jun 10, 18 19:29 **FragmentableWeapon.h** Page 1/1

```
#ifndef __FRAGMENTABLEWEAPON_H__
#define __FRAGMENTABLEWEAPON_H__

#include "Weapon.h"
#include <string>

class FragmentableWeapon: public Weapon{
    protected:
        int fragments;

    public:
        FragmentableWeapon(World& world, GameParameters& parameters,
                                        int damage, int fragments, int radius);
        virtual ~FragmentableWeapon();

        //Explota el arma y lanza fragmentos
        void explode();
};

#endif
```

jun 10, 18 19:29 **Fragment.cpp** Page 1/1

```
#include "Fragment.h"

Fragment::Fragment(World& world, GameParameters& params, int dam, int radius):
    Weapon(world, params, dam, radius){}

Fragment::~Fragment(){}

void Fragment::setShootPosition(b2Vec2 pos){
    this->shoot_position = pos;
}

b2Vec2 Fragment::getShootPosition(){
    return this->shoot_position;
}

void Fragment::shoot(int angle){
    Weapon::shoot(1, angle, -1, -1, -1);
}
```

jun 10, 18 19:29 **Fragment.h** Page 1/1

```
1  #ifndef __SERVERFRAGMENT_H__
2  #define __SERVERFRAGMENT_H__
3
4  #include "Weapon.h"
5
6  class Fragment: public Weapon{
7      private:
8          b2Vec2 shoot_position;
9
10     public:
11         Fragment(World& world, GameParameters& parameters, int damage, int radiu
   s);
12         ~Fragment();
13
14         void setShootPosition(b2Vec2 pos);
15         b2Vec2 getShootPosition();
16
17         void shoot(int angle);
18 };
19
20 #endif
```

jun 10, 18 19:29 **GreenGrenade.cpp** Page 1/1

```
1  #include "GreenGrenade.h"
2  #include <string>
3
4  GreenGrenade::GreenGrenade(World& world, GameParameters& parameters):
5      Weapon(world, parameters,
6          parameters.getWeaponDamage(GREEN_GRENADE_NAME),
7          parameters.getWeaponRadius(GREEN_GRENADE_NAME)){}
8
9  GreenGrenade::~GreenGrenade(){}
10
11 const std::string& GreenGrenade::getName(){
12     return GREEN_GRENADE_NAME;
13 }
```

jun 10, 18 19:29 **GreenGrenade.h** Page 1/1

```
1  #ifndef __SERVERGREENGRENADE_H__
2  #define __SERVERGREENGRENADE_H__
3
4  #include "Weapon.h"
5  #include <string>
6
7  class GreenGrenade: public Weapon{
8      public:
9          GreenGrenade(World& world, GameParameters& parameters);
10         ~GreenGrenade();
11
12         const std::string& getName() override;
13 };
14
15 #endif
```

jun 10, 18 19:29 **HolyGrenade.cpp** Page 1/1

```
1  #include "HolyGrenade.h"
2  #include <string>
3
4  HolyGrenade::HolyGrenade(World& world, GameParameters& parameters):
5      Weapon(world, parameters,
6          parameters.getWeaponDamage(HOLY_GRENADE_NAME),
7          parameters.getWeaponRadius(HOLY_GRENADE_NAME)){}
8
9  HolyGrenade::~HolyGrenade(){}
10
11 const std::string& HolyGrenade::getName(){
12     return HOLY_GRENADE_NAME;
13 }
```

jun 10, 18 19:29 **HolyGrenade.h** Page 1/1

```h
#ifndef __SERVERHOLYGRENADE_H__
#define __SERVERHOLYGRENADE_H__

#include "Weapon.h"
#include <string>

class HolyGrenade: public Weapon{
    public:
        HolyGrenade(World& world, GameParameters& parameters);
        ~HolyGrenade();

        const std::string& getName() override;
};

#endif
```

jun 10, 18 19:29 **Mortar.cpp** Page 1/1

```cpp
#include "Mortar.h"
#include <string>

Mortar::Mortar(World& world, GameParameters& parameters):
    FragmentableWeapon(world, parameters,
        parameters.getWeaponDamage(MORTAR_NAME),
        parameters.getWeaponFragments(MORTAR_NAME),
        parameters.getWeaponRadius(MORTAR_NAME)){}

Mortar::~Mortar(){}

const std::string& Mortar::getName(){
    return MORTAR_NAME;
}

bool Mortar::isWindAffected(){
    return true;
}
```

jun 10, 18 19:29 **MortarFragment.cpp** Page 1/1

```cpp
#include "MortarFragment.h"
#include <string>

MortarFragment::MortarFragment(World& world, GameParameters& parameters):
    Fragment(world, parameters,
        parameters.getWeaponDamage(MORTAR_FRAGMENTS_NAME),
        parameters.getWeaponRadius(MORTAR_FRAGMENTS_NAME)){}

MortarFragment::~MortarFragment(){}

const std::string& MortarFragment::getName(){
    return MORTAR_FRAGMENTS_NAME;
}

bool MortarFragment::isWindAffected(){
    return true;
}
```

jun 10, 18 19:29 **MortarFragment.h** Page 1/1

```cpp
#ifndef __SERVERMORTARFRAGMENT_H__
#define __SERVERMORTARFRAGMENT_H__

#include "Fragment.h"
#include <string>

class MortarFragment: public Fragment{
    public:
        MortarFragment(World& world, GameParameters& parameters);
        ~MortarFragment();

        const std::string& getName() override;

        bool isWindAffected() override;
};

#endif
```

jun 10, 18 19:29 **Mortar.h** Page 1/1

```
1  #ifndef __SERVERMORTAR_H__
2  #define __SERVERMORTAR_H__
3
4  #include "FragmentableWeapon.h"
5  #include <string>
6
7  class Mortar: public FragmentableWeapon{
8      public:
9          Mortar(World& world, GameParameters& parameters);
10         ~Mortar();
11
12         const std::string& getName() override;
13
14         bool isWindAffected() override;
15 };
16
17 #endif
```

jun 10, 18 19:29 **RedGrenade.cpp** Page 1/1

```
1  #include "RedGrenade.h"
2  #include <string>
3
4  RedGrenade::RedGrenade(World& world, GameParameters& parameters):
5      FragmentableWeapon(world, parameters,
6          parameters.getWeaponDamage(RED_GRENADE_NAME),
7          parameters.getWeaponFragments(RED_GRENADE_NAME),
8          parameters.getWeaponRadius(RED_GRENADE_NAME)){}
9
10 RedGrenade::~RedGrenade(){}
11
12 const std::string& RedGrenade::getName(){
13     return RED_GRENADE_NAME;
14 }
```

jun 10, 18 19:29 **RedGrenadeFragment.cpp** Page 1/1

```cpp
#include "RedGrenadeFragment.h"
#include <string>

RedGrenadeFragment::RedGrenadeFragment(World& world, GameParameters& params):
    Fragment(world, params,
        params.getWeaponDamage(RED_GRENADE_FRAGMENTS_NAME),
        params.getWeaponRadius(RED_GRENADE_FRAGMENTS_NAME)){}

RedGrenadeFragment::~RedGrenadeFragment(){}

const std::string& RedGrenadeFragment::getName(){
    return RED_GRENADE_FRAGMENTS_NAME;
}
```

jun 10, 18 19:29 **RedGrenadeFragment.h** Page 1/1

```cpp
#ifndef __SERVERREDGRENADEFRAGMENT_H__
#define __SERVERREDGRENADEFRAGMENT_H__

#include "Fragment.h"
#include <string>

class RedGrenadeFragment: public Fragment{
    public:
        RedGrenadeFragment(World& world, GameParameters& parameters);
        ~RedGrenadeFragment();

        const std::string& getName() override;
};

#endif
```

jun 10, 18 19:29 **RedGrenade.h** Page 1/1

```h
#ifndef __SERVERREDGRENADE_H__
#define __SERVERREDGRENADE_H__

#include "FragmentableWeapon.h"
#include <string>

class RedGrenade: public FragmentableWeapon{
    public:
        RedGrenade(World& world, GameParameters& parameters);
        ~RedGrenade();

        const std::string& getName() override;
};

#endif
```

jun 10, 18 19:29 **Teleportation.cpp** Page 1/1

```cpp
#include "Teleportation.h"
#include "Worm.h"
#include <mutex>
#include <string>

Teleportation::Teleportation(World& world, GameParameters& parameters):
    Weapon(world, parameters, 0){}

Teleportation::~Teleportation(){}

const std::string& Teleportation::getName(){
    return TELEPORT_NAME;
}

void Teleportation::shoot(char dir, int angle, int power,
                                int time, int shooter_id){}

void Teleportation::shoot(Worm& shooter, b2Vec2 pos){
    pos.x += (worm_size / 2);
    pos.y += (worm_size / 2);
    std::lock_guard<std::mutex> lock(this->world.getMutex());
    b2Body* body = shooter.getBody();
    if (body){
        shooter.getBody()->SetTransform(pos, 0);
        shooter.getBody()->SetAwake(true);
    }
}
```

jun 10, 18 19:29 **Teleportation.h** Page 1/1

```
#ifndef __SERVERTELEPORTATION_H__
#define __SERVERTELEPORTATION_H__

#include "Weapon.h"
#include <string>

class Teleportation: public Weapon{
    public:
        Teleportation(World& world, GameParameters& parameters);
        ~Teleportation();

        const std::string& getName() override;

        void shoot(char dir, int angle, int power, int time, int shooter_id) override;

        //Teletransporta al gusano
        void shoot(Worm& shooter, b2Vec2 pos) override;
};

#endif
```

jun 10, 18 19:29 **Weapon.cpp** Page 1/2

```
#include "Weapon.h"
#include "b2Fixture.h"
#include "b2CircleShape.h"
#include "CollisionData.h"
#include "Worm.h"
#include "Math.h"

int Weapon::weapon_id = 1;

Weapon::Weapon(World& world, GameParameters& params, int damage, int radius):
    PhysicalObject(world, Weapon::weapon_id++, TYPE_WEAPON),
    parameters(params),
    damage(damage), radius(radius),
    waiting_to_explode(false), time_to_explode(-1),
    angle(MAX_WEAPON_ANGLE + 1), power(-1),
    shooter_id(-1), explode_time(world, *this){}

Weapon::~Weapon(){
    this->explode_time.join();
}

bool Weapon::isActive(){
    return this->waiting_to_explode || PhysicalObject::isActive();
}

void Weapon::shoot(char dir, int angle, int power, int time, int shooter_id){
    if (dir == -1 && angle <= MAX_WEAPON_ANGLE){
        angle = 180 - angle;
    }
    this->time_to_explode = time;
    this->angle = angle;
    this->power = power;
    this->shooter_id = shooter_id;
}

void Weapon::shoot(Worm& shooter, b2Vec2 pos){}

void Weapon::getBodyDef(b2BodyDef& body_def, const b2Vec2& pos){
    body_def.type = b2_dynamicBody;
    body_def.position.Set(pos.x, pos.y);
    body_def.fixedRotation = true;
    body_def.bullet = true;
}

void Weapon::createFixtures(){
    b2CircleShape circleShape;
    circleShape.m_p.Set(0, 0);
    circleShape.m_radius = weapon_size / 2;

    b2FixtureDef fixtureDef;
    fixtureDef.shape = &circleShape;
    fixtureDef.density = 4;
    this->body->CreateFixture(&fixtureDef);
}

void Weapon::setInitialVelocity(){
    if (this->angle <= 360){
        int velocity = this->parameters.getWeaponsVelocity();
        if (this->power != -1){
            velocity *= this->power / 1000;
        }
        b2Vec2 linear_velocity(velocity * Math::cosDegrees(this->angle),
                               velocity * Math::sinDegrees(this->angle));
        this->body->SetLinearVelocity(linear_velocity);
    }
    this->waiting_to_explode = true;
```

jun 10, 18 19:29 **Weapon.cpp** Page 2/2

```
67      this->explode_time.setTime(this->time_to_explode);
68      this->explode_time.start();
69  }
70
71
72  void Weapon::explode(){
73      b2Vec2 center = this->body->GetPosition();
74      for (float bullet_angle = 0; bullet_angle < 360; bullet_angle+= 5){
75          this->attackWormExplosion(center, bullet_angle);
76      }
77
78      this->explode_time.stop();
79      this->waiting_to_explode = false;
80      this->is_dead = true;
81  }
82
83  void Weapon::attackWormExplosion(const b2Vec2& center, int angle){
84      b2Vec2 end = center + this->radius *
85                               b2Vec2(Math::cosDegrees(angle), Math::sinDegrees(ang
    le));
86      b2Body* closest = this->world.getClosestObject(&explosion, center, end);
87      if (closest){
88          Worm* worm = (Worm*)((CollisionData*)closest->GetUserData())->getObject(
    );
89          float distance = b2Distance(center, worm->getPosition());
90
91          //Justo en el borde hace la mitad de danio
92          int worm_damage = this->damage * (1 - distance / (2 * this->radius));
93
94          worm->receiveWeaponDamage(worm_damage, center);
95      }
96  }
97
98  void Weapon::collideWithSomething(CollisionData *other){
99      if (this->time_to_explode == -1){
100         this->explode_time.stop();
101         this->explode();
102     } else if (other->getType() == TYPE_BORDER){
103         this->explode_time.stop();
104         this->is_dead = true;
105     }
106 }
107
108 int Weapon::getShooterId() const{
109     return this->shooter_id;
110 }
111
112 void Weapon::removeShooterId(){
113     this->shooter_id = -1;
114 }
```

jun 10, 18 19:29 **WeaponExplodeTime.cpp** Page 1/1

```
1   #include "WeaponExplodeTime.h"
2   #include "Weapon.h"
3   #include "World.h"
4
5   WeaponExplodeTime::WeaponExplodeTime(World& world, Weapon& weapon):
6       weapon(weapon), world(world), time(-1){}
7
8   WeaponExplodeTime::~WeaponExplodeTime(){}
9
10  void WeaponExplodeTime::setTime(int time){
11      this->time = time;
12  }
13
14  void WeaponExplodeTime::run(){
15      if (this->time > 0){
16          int passed = 0;
17          while (this->running && passed < this->time){
18              std::this_thread::sleep_for(std::chrono::seconds(1));
19              passed++;
20          }
21          if (this->running){
22              std::lock_guard<std::mutex> lock(this->world.getMutex());
23              if (!this->weapon.isDead()){
24                  this->weapon.explode();
25                  this->world.removeTimedWeapon(this->weapon);
26              }
27          }
28      }
29  }
```

jun 10, 18 19:29 **WeaponExplodeTime.h** Page 1/1

```
#ifndef __WEAPONEXPLODETIME_H__
#define __WEAPONEXPLODETIME_H__

#include "Thread.h"
#include <mutex>

class Weapon;
class World;

class WeaponExplodeTime: public Thread{
    private:
        Weapon& weapon;
        World& world;
        int time;

    public:
        WeaponExplodeTime(World& world, Weapon& weapon);
        ~WeaponExplodeTime();

        //Setea el tiempo de explosion
        void setTime(int time);

        //Cuenta el tiempo que falta para que el arma explote
        void run() override;
};

#endif
```

jun 10, 18 19:29 **WeaponFactory.cpp** Page 1/1

```
#include "WeaponFactory.h"
#include "WeaponNames.h"
#include <string>

#include "Bazooka.h"
#include "Dynamite.h"
#include "RedGrenade.h"
#include "RedGrenadeFragment.h"
#include "GreenGrenade.h"
#include "HolyGrenade.h"
#include "Banana.h"
#include "Teleportation.h"
#include "AirAttack.h"
#include "AirAttackMissile.h"
#include "Mortar.h"
#include "MortarFragment.h"
#include "Bat.h"

WeaponFactory::WeaponFactory(World& world, GameParameters& parameters):
    world(world), parameters(parameters){}

WeaponFactory::~WeaponFactory(){}

physical_object_ptr WeaponFactory::getWeapon(const std::string& name){
    if (name == BAZOOKA_NAME){
        return physical_object_ptr(new Bazooka(world, parameters));
    } else if (name == DYNAMITE_NAME){
        return physical_object_ptr(new Dynamite(world, parameters));
    } else if (name == RED_GRENADE_NAME){
        return physical_object_ptr(new RedGrenade(world, parameters));
    } else if (name == RED_GRENADE_FRAGMENTS_NAME){
        return physical_object_ptr(new RedGrenadeFragment(world, parameters));
    } else if (name == GREEN_GRENADE_NAME){
        return physical_object_ptr(new GreenGrenade(world, parameters));
    } else if (name == HOLY_GRENADE_NAME){
        return physical_object_ptr(new HolyGrenade(world, parameters));
    } else if (name == MORTAR_NAME){
        return physical_object_ptr(new Mortar(world, parameters));
    } else if (name == MORTAR_FRAGMENTS_NAME){
        return physical_object_ptr(new MortarFragment(world, parameters));
    } else if (name == BANANA_NAME){
        return physical_object_ptr(new Banana(world, parameters));
    } else if (name == BAT_NAME){
        return physical_object_ptr(new Bat(world, parameters));
    } else if (name == TELEPORT_NAME){
        return physical_object_ptr(new Teleportation(world, parameters));
    } else if (name == AIR_ATTACK_NAME){
        return physical_object_ptr(new AirAttack(world, parameters));
    } else if (name == AIR_ATTACK_MISSILE_NAME){
        return physical_object_ptr(new AirAttackMissile(world, parameters));
    }

    throw std::runtime_error(name + ": El arma no existe.");
}
```

jun 10, 18 19:29 **WeaponFactory.h** Page 1/1

```
#ifndef __WEAPONFACTORY_H__
#define __WEAPONFACTORY_H__

#include "World.h"
#include "GameParameters.h"
#include <string>

class WeaponFactory{
    private:
        World& world;
        GameParameters& parameters;

    public:
        WeaponFactory(World& world, GameParameters& parameters);
        ~WeaponFactory();

        //Devuelve el arma pedida
        physical_object_ptr getWeapon(const std::string& name);
};

#endif
```

jun 10, 18 19:29 **Weapon.h** Page 1/2

```
#ifndef __WEAPON_H__
#define __WEAPON_H__

#include "PhysicalObject.h"
#include "GameParameters.h"
#include "World.h"
#include "WeaponExplodeTime.h"
#include <string>
#include "WeaponNames.h"
#include "RayCastWeaponExploded.h"

class Worm;

class Weapon: public PhysicalObject{
    protected:
        GameParameters& parameters;
        int damage;
        int radius;
        bool waiting_to_explode;
        int time_to_explode;
        float angle;
        float power;
        int shooter_id;
        WeaponExplodeTime explode_time;
        RayCastWeaponExploded explosion;

        virtual void createFixtures() override;
        virtual void setInitialVelocity() override;

        //Ataca a los gusanos en el radio de explosion
        void attackWormExplosion(const b2Vec2& center, int angle);

    public:
        static int weapon_id;

        Weapon(World& world, GameParameters& parameters, int damage, int radius
= 0);
        virtual ~Weapon();

        //Devuelve true si el arma esta en movimiento o esperando para explotar
        bool isActive() override;

        //Carga los datos para disparar el arma
        virtual void shoot(char dir, int angle, int power, int time, int shooter
_id);

        //Dispara un arma teledirigida
        virtual void shoot(Worm& shooter, b2Vec2 pos);

        //Explota el arma
        virtual void explode();

        //Establece la accion a realizar cuando el arma colisiona
        virtual void collideWithSomething(CollisionData *other) override;

        void getBodyDef(b2BodyDef& body_def, const b2Vec2& pos) override;

        //Devuelve el nombre del arma
        virtual const std::string& getName() = 0;

        //Devuelve el id del tirador
        int getShooterId() const;

        //Remueve el id del tirador
        void removeShooterId();
};
```

jun 10, 18 19:29 **Weapon.h** Page 2/2

```
65
66  #endif
```

jun 10, 18 19:29 **Worm.cpp** Page 1/4

```
1   #include "Worm.h"
2   #include "b2CircleShape.h"
3   #include "b2PolygonShape.h"
4   #include "b2Fixture.h"
5   #include "Protocol.h"
6   #include "WeaponFactory.h"
7   #include "Girder.h"
8   #include "Math.h"
9   #include <algorithm>
10  #include <string>
11
12  Worm::Worm(World& world, GameParameters& parameters,
13                                  int id, int player_id, WeaponList& weapons):
14      PhysicalObject(world, id, TYPE_WORM), player_id(player_id),
15      life(parameters.getWormLife()),
16      dir(1), parameters(parameters), weapons(weapons), max_height(0),
17      colliding_with_girder(0), friction(0),
18      movement_allowed(false), angle(0), has_shot(false), damage_received(false){}
19
20  Worm::~Worm(){}
21
22  void Worm::getBodyDef(b2BodyDef& body_def, const b2Vec2& pos){
23      body_def.type = b2_dynamicBody;
24      body_def.position.Set(pos.x, pos.y);
25  }
26
27  void Worm::createFixtures(){
28      b2CircleShape circleShape;
29      circleShape.m_p.Set(0, 0);
30      circleShape.m_radius = worm_size / 2;
31
32      b2FixtureDef fixtureDef;
33      fixtureDef.shape = &circleShape;
34      fixtureDef.density = 10;
35      this->body->CreateFixture(&fixtureDef);
36      this->body->SetFixedRotation(true);
37
38      //Sensor para colisiones
39      b2PolygonShape sensorShape;
40      sensorShape.SetAsBox(worm_size * 0.5 * 0.7, worm_size / 5,
41                                        b2Vec2(0, -1 * worm_size / 2), 0);
42
43      b2FixtureDef sensorFixtureDef;
44      sensorFixtureDef.shape = &sensorShape;
45      sensorFixtureDef.isSensor = true;
46      this->body->CreateFixture(&sensorFixtureDef);
47  }
48
49  int Worm::getPlayerId() const{
50      return this->player_id;
51  }
52
53  int Worm::getLife() const{
54      return this->life;
55  }
56
57  char Worm::getDir() const{
58      return this->dir;
59  }
60
61  bool Worm::isColliding() const{
62      return this->colliding_with_girder && !this->movement_allowed;
63  }
64
65  const std::string& Worm::getCurrentWeapon() const{
66      physical_object_ptr weapon = weapons.getCurrentWeapon(world, parameters);
```

```
67      return ((Weapon*)weapon.get())->getName();
68  }
69
70  void Worm::addLife(int life){
71      this->life += life;
72  }
73
74  void Worm::reduceLife(size_t damage){
75      this->life -= damage;
76      this->damage_received = true;
77      this->data_updated = true;
78      if (this->life <= 0){
79          this->life = 0;
80          this->is_dead = true;
81      }
82  }
83
84  bool Worm::move(char action){
85      if (!this->colliding_with_girder || this->movement_allowed){
86          return false;
87      }
88      this->movement_allowed = false;
89      if (action == MOVE_RIGHT){
90          this->dir = action;
91          b2Vec2 velocity(parameters.getWormVelocity(), 0);
92          this->world.setLinearVelocity(*this, velocity);
93      } else if (action == MOVE_LEFT){
94          this->dir = action;
95          b2Vec2 velocity(-1 * parameters.getWormVelocity(), 0);
96          this->world.setLinearVelocity(*this, velocity);
97      } else {
98          this->movement_allowed = true;
99          if (action == JUMP){
100             b2Vec2 velocity(parameters.getWormJumpVelocity(),
101                                         parameters.getWormJumpHeight());
102             velocity.x *= this->dir;
103             this->world.setLinearVelocity(*this, velocity);
104         } else if (action == ROLLBACK){
105             b2Vec2 velocity(parameters.getWormRollbackVelocity(),
106                                         parameters.getWormRollbackHeight());
107             velocity.x *= -1 * this->dir;
108             this->world.setLinearVelocity(*this, velocity);
109         }
110     }
111     return true;
112 }
113
114 void Worm::shoot(int angle, int power, int time){
115     if (!this->weapons.shoot()){
116         return;
117     }
118     b2Vec2 pos = this->getPosition();
119     int shooter_id = this->id;
120     float x_add = (worm_size * this->dir);;
121     float y_add = worm_size;
122     if (angle > MAX_WEAPON_ANGLE){
123         shooter_id = -1;
124         x_add *= Math::cosDegrees(this->angle);
125         y_add *= Math::sinDegrees(this->angle);
126     } else {
127         float factor = (this->getCurrentWeapon() == BAT_NAME ? 0.2 : 0.7);
128         x_add *= Math::cosDegrees(angle) * factor;
129         y_add *= Math::sinDegrees(angle) * factor;
130     }
131
132     pos.x += x_add;
```



```
133     pos.y += y_add;
134
135     physical_object_ptr weapon = weapons.getCurrentWeapon(world, parameters);
136     ((Weapon*)weapon.get())->shoot(this->dir, angle, power, time, shooter_id);
137     this->world.addObject(weapon, pos);
138     this->has_shot = true;
139 }
140
141 void Worm::shoot(b2Vec2 pos){
142     if (!this->weapons.shoot()){
143         return;
144     }
145     physical_object_ptr weapon = weapons.getCurrentWeapon(world, parameters);
146     ((Weapon*)weapon.get())->shoot(*this, pos);
147     this->has_shot = true;
148 }
149
150 void Worm::receiveWeaponDamage(int damage, const b2Vec2 &epicenter){
151     this->reduceLife(damage);
152     b2Vec2 direction = this->body->GetPosition() - epicenter;
153     direction.Normalize();
154     this->body->SetGravityScale(1);
155     this->movement_allowed = true;
156     this->body->SetLinearVelocity(
157                     damage * parameters.getWormExplosionVelocity() * direction);
158 }
159
160 void Worm::collideWithSomething(CollisionData *other){
161     if (other->getType() == TYPE_BORDER){
162         this->kill();
163     } else if (other->getType() == TYPE_GIRDER){
164         int min_height = parameters.getWormHeightToDamage();
165         float current_height = this->body->GetPosition().y;
166         this->max_height -= current_height;
167
168         if (this->max_height >= min_height){
169             this->reduceLife(std::min((int) this->max_height - min_height + 1,
170                                                 parameters.getWormMaxHeightDamag
    e()));
171         }
172         this->max_height = 0;
173         this->colliding_with_girder ++;
174         Girder* girder = (Girder*)other->getObject();
175         if (girder->hasFriction()){
176             this->friction++;
177             this->movement_allowed = false;
178             this->angle = girder->getAngle();
179         }
180     }
181 }
182
183 void Worm::endCollissionGirder(char has_friction){
184     this->friction -= has_friction;
185     this->colliding_with_girder --;
186     if (this->friction <= 0){
187         this->friction = 0;
188         this->body->SetGravityScale(1);
189         this->angle = 0;
190     }
191 }
192
193 bool Worm::isActive(){
194     if (!this->colliding_with_girder){
195         float height = this->body->GetPosition().y;
196         this->max_height = std::max(this->max_height, height);
197     } else if (this->friction && !this->movement_allowed){
```

jun 10, 18 19:29 **Worm.cpp** Page 4/4

```
198         this->body->SetGravityScale(0);
199         this->body->SetLinearVelocity(b2Vec2(0, 0));
200     }
201     if (!this->body->IsAwake()){
202         this->movement_allowed = false;
203     } else if (!this->friction){
204         this->movement_allowed = true;
205     }
206     return PhysicalObject::isActive();
207 }
208
209 bool Worm::hasShot() const{
210     return this->has_shot;
211 }
212
213 bool Worm::damageReceived() const{
214     return this->damage_received || this->is_dead;
215 }
216
217 void Worm::beginTurn(){
218     this->has_shot = false;
219     this->damage_received = false;
220 }
```

jun 10, 18 19:29 **Worm.h** Page 1/2

```
1  #ifndef __WORM_H__
2  #define __WORM_H__
3
4  #include "PhysicalObject.h"
5  #include "GameParameters.h"
6  #include "Weapon.h"
7  #include "WeaponList.h"
8  #include <string>
9
10 class Worm: public PhysicalObject{
11     private:
12         int player_id;
13         int life;
14         char dir;
15         GameParameters& parameters;
16         WeaponList& weapons;
17         float max_height;
18         int colliding_with_girder;
19         int friction;
20         bool movement_allowed;
21         int angle;
22
23         bool has_shot;
24         bool damage_received;
25
26     protected:
27         void getBodyDef(b2BodyDef& body_def, const b2Vec2& pos) override;
28         void createFixtures() override;
29
30     public:
31         Worm(World& world, GameParameters& parameters,
32                     int id, int player_id, WeaponList& weapons);
33         ~Worm();
34
35         //Devuelve el id del jugador
36         int getPlayerId() const;
37
38         //Devuelve la vida del worm
39         int getLife() const;
40
41         //Devuelve la direccion del worm
42         char getDir() const;
43
44         //Devuelve true si esta colisionando
45         bool isColliding() const;
46
47         //Devuelve el arma actual
48         const std::string& getCurrentWeapon() const;
49
50         //Aumenta la vida del gusano
51         void addLife(int life);
52
53         //Reduce la vida del gusano
54         void reduceLife(size_t damage);
55
56         //Ejecuta una accion de movimiento del gusano
57         bool move(char action);
58
59         //Dispara un arma no teledirigida
60         void shoot(int angle, int power, int time);
61
62         //Dispara un arma teledirigida
63         void shoot(b2Vec2 pos);
64
65         //Analiza la colision con el objeto
66         void collideWithSomething(CollisionData *other) override;
```

jun 10, 18 19:29 **Worm.h** Page 2/2

```
67
68          //Analiza el fin del contacto con una viga
69          void endCollissionGirder(char friction);
70
71          //Recibe danio de un arma o una explosion
72          void receiveWeaponDamage(int damage, const b2Vec2 &epicenter);
73
74          //Devuelve true si el gusano esta en movimiento
75          bool isActive() override;
76
77          //Devuelve true si el gusano disparo
78          bool hasShot() const;
79
80          //Devuelve true si el gusano recibio danio
81          bool damageReceived() const;
82
83          //Empieza el turno del gusano
84          void beginTurn();
85  };
86
87  #endif
```

jun 10, 18 19:29 **Wind.cpp** Page 1/1

```
1   #include "Wind.h"
2   #include <random>
3
4   Wind::Wind(GameParameters& parameters):
5       min_velocity(parameters.getWindMinVelocity()),
6       max_velocity(parameters.getWindMaxVelocity()){
7           this->update();
8       }
9
10  Wind::~Wind(){}
11
12  float Wind::getVelocity() const{
13      return this->velocity;
14  }
15
16  void Wind::update(){
17      std::mt19937 rng;
18      rng.seed(std::random_device()());
19      std::uniform_real_distribution<float> dist(min_velocity, max_velocity);
20      std::uniform_int_distribution<int> direction(-1, 1); //Acepto velocidad 0
21
22      this->velocity = dist(rng);
23      this->velocity *= direction(rng);
24  }
```

jun 10, 18 19:29 **Wind.h** Page 1/1

```
#ifndef __WIND_H__
#define __WIND_H__

#include "GameParameters.h"

class Wind{
    private:
        float min_velocity;
        float max_velocity;
        float velocity;

    public:
        explicit Wind(GameParameters& parameters);
        ~Wind();

        //Devuelve la velocidad del viento
        float getVelocity() const;

        //Actualiza la velocidad del viento
        void update();
};


#endif
```

jun 20, 18 18:42 **World.cpp** Page 1/3

```
#include "World.h"
#include "Weapon.h"
#include "BottomBorder.h"
#include "b2WorldCallbacks.h"
#include "Fragment.h"
#include <list>

World::World(GameParameters& parameters):
    world(b2Vec2(0, parameters.getGravity())),
    wind(parameters), is_active(true),
    sleep_time(parameters.getWorldSleepAfterStep()),
    time_step(parameters.getWorldTimeStep()){
        this->world.SetAllowSleeping(true);
        this->world.SetContinuousPhysics(true);
        this->world.SetContactListener(&this->collision_listener);
        this->world.SetContactFilter(&this->collision_listener);
        this->initialize();
}

World::~World(){}

void World::run(){
    int32 velocityIterations = 8;   //how strongly to correct velocity
    int32 positionIterations = 3;   //how strongly to correct position

    while(this->running){
        std::this_thread::sleep_for(std::chrono::milliseconds(this->sleep_time))
;

        this->addAllFragments();

        std::lock_guard<std::mutex> lock(this->mutex);

        this->world.Step(this->time_step, velocityIterations, positionIterations
);

        this->is_active = false;
        for (physical_object_ptr& object: this->objects){
            if (object->isDead()){
                this->removeObject(object);
            } else if (object->isActive()){
                this->is_active = true;
                b2Body* body = object->getBody();
                if (body && object->isWindAffected()){
                    body->ApplyForceToCenter(b2Vec2(this->wind.getVelocity(), 0)
, false);
                }
            }
        }
    }
}

void World::addAllFragments(){
    std::lock_guard<std::mutex> lock(this->mutex);
    for (physical_object_ptr& fragment: this->fragments_to_add){
        b2BodyDef body_def;
        b2Vec2 pos = ((Fragment *) fragment.get())->getShootPosition();
        fragment->getBodyDef(body_def, pos);
        this->initializeObject(fragment, &body_def);
    }
    this->fragments_to_add.clear();
}

bool World::isActive(){
    std::lock_guard<std::mutex> lock(this->mutex);
    return this->is_active;
```

```
64  }
65
66  void World::update(){
67      std::lock_guard<std::mutex> lock(this->mutex);
68      this->wind.update();
69  }
70
71  void World::addObject(physical_object_ptr object, const b2Vec2& pos){
72      b2BodyDef body_def;
73      object->getBodyDef(body_def, pos);
74
75      std::lock_guard<std::mutex> lock(this->mutex);
76      this->initializeObject(object, &body_def);
77  }
78
79  void World::initializeObject(physical_object_ptr object, b2BodyDef* body_def){
80      object->initializeBody(this->world.CreateBody(body_def));
81      if (body_def->type != b2_staticBody){
82          this->objects.push_back(object);
83      } else {
84          this->girders.push_back(object);
85      }
86  }
87
88  void World::addWeaponFragment(physical_object_ptr fragment){
89      this->fragments_to_add.push_back(fragment);
90  }
91
92  void World::removeTimedWeapon(Weapon& weapon){
93      b2Body* body = weapon.getBody();
94      if (body){
95          this->world.DestroyBody(body);
96          weapon.destroyBody();
97      }
98  }
99
100 void World::removeObject(physical_object_ptr object){
101     b2Body* body = object->getBody();
102     if (body){
103         this->world.DestroyBody(body);
104         object->destroyBody();
105     }
106 }
107
108 void World::initialize(){
109     physical_object_ptr bottom_border(new BottomBorder(*this));
110     this->addObject(bottom_border, b2Vec2(0, 0));
111 }
112
113 void World::setLinearVelocity(PhysicalObject& object, b2Vec2& velocity){
114     std::lock_guard<std::mutex> lock(this->mutex);
115     b2Body* body = object.getBody();
116     if (body){
117         body->SetGravityScale(1);
118         body->SetLinearVelocity(velocity);
119     }
120 }
121
122 b2Body* World::getClosestObject(RayCastWeaponExploded* callback,
123                                 b2Vec2 center, b2Vec2 end){
124     this->world.RayCast(callback, center, end);
125     return callback->getClosestWorm();
126 }
127
128 float World::getWind() const{
129     return this->wind.getVelocity();
```



```
130 }
131
132 std::list<physical_object_ptr>& World::getObjectsList(){
133     return this->objects;
134 }
135
136 std::list<physical_object_ptr>& World::getGirdersList(){
137     return this->girders;
138 }
139
140 std::mutex& World::getMutex(){
141     return this->mutex;
142 }
```

jun 10, 18 19:29 **World.h** Page 1/2

```
1  #ifndef __WORLD_H__
2  #define __WORLD_H__
3
4  #include "Thread.h"
5  #include "b2World.h"
6  #include "b2Body.h"
7  #include "PhysicalObject.h"
8  #include "CollisionListener.h"
9  #include "RayCastWeaponExploded.h"
10 #include "Wind.h"
11 #include <mutex>
12 #include <list>
13
14 class Weapon;
15
16 class World: public Thread{
17     private:
18         b2World world;
19         Wind wind;
20         std::mutex mutex;
21         CollisionListener collision_listener;
22         std::list<physical_object_ptr> objects;
23         std::list<physical_object_ptr> girders;
24         std::list<physical_object_ptr> fragments_to_add;
25         bool is_active;
26         int sleep_time;
27         float time_step;
28
29         //Inicializa el mundo
30         void initialize();
31
32         //Remueve un objeto del mundo
33         void removeObject(physical_object_ptr object);
34
35         //Inicializa un objeto recien agregado al mundo
36         void initializeObject(physical_object_ptr object, b2BodyDef* body_def);
37
38         //Agrega todos los fragmentos de armas al mundo
39         void addAllFragments();
40
41     public:
42         explicit World(GameParameters& parameters);
43         ~World();
44
45         void run() override;
46
47         //Agrega el objeto al mundo en la posicion indicada
48         void addObject(physical_object_ptr object, const b2Vec2& pos);
49
50         //Agrega un fragmento de arma
51         void addWeaponFragment(physical_object_ptr fragment);
52
53         //Elimina una arma del mundo
54         void removeTimedWeapon(Weapon& weapon);
55
56         //Setea la velocidad de un objeto
57         void setLinearVelocity(PhysicalObject& object, b2Vec2& velocity);
58
59         //Devuelve true si alguno de los objetos esta en movimiento
60         bool isActive();
61
62         //Actualiza el mundo
63         void update();
64
65         //Devuelve la velocidad del viento
66         float getWind() const;
```

jun 10, 18 19:29 **World.h** Page 2/2

```
67
68         //Devuelve el objeto mas cercano entre al centro en la direccion end - c
   enter
69         b2Body* getClosestObject(RayCastWeaponExploded* callback,
70                                  b2Vec2 center, b2Vec2 end);
71
72         //Devuelve la lista de objetos
73         std::list<physical_object_ptr>& getObjectsList();
74
75         //Devuelve la lista de vigas
76         std::list<physical_object_ptr>& getGirdersList();
77
78         //Devuelve el mutex
79         std::mutex& getMutex();
80 };
81
82
83 #endif
```

jun 10, 18 16:09 **main.cpp** Page 1/1

```cpp
#include "Server.h"
#include "yaml.h"
#include "ConfigFields.h"
#include "Path.h"
#include <iostream>
#include <string>
#include <mutex>

#define EXIT_CHAR 'q'

int main(int argc, const char* argv[]){
    std::mutex mutex_cout;
    try{
        YAML::Node config(YAML::LoadFile(SERVER_CONFIG_FILE));
        Server server(config[SERVER_PORT].as<std::string>(), mutex_cout);
        std::cout << "[LOG] Server iniciado." << std::endl;
        server.start();
        while (std::cin.get() != EXIT_CHAR){}
        {
            std::lock_guard<std::mutex> lock(mutex_cout);
            std::cout << "[LOG] Comenzando el cierre del servidor." << std::endl;
        }
        server.stop();
        server.join();
    } catch(const std::exception& e){
        std::lock_guard<std::mutex> lock(mutex_cout);
        std::cout << "[ERROR] " << e.what() << std::endl;
    }
    return 0;
}
```

jun 26, 18 13:25 **Table of Content** Page 1/2

jun 26, 18 13:25 **Table of Content** Page 2/2