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

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){
    } 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);
    } 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 07, 18 20:19 **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&& client, GamesList& games, std::mutex& mutex_cout);

        /* Destructor */
        ~ClientHandler();

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

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

#endif
```

Jun 09, 18 14:59 **GamesList.cpp** Page 1/2

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

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

GamesList::~GamesList(){
    for (auto 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(const std::string& game_name, const std::string& map, int max_players, Player& player){
    std::lock_guard<std::mutex> lock(this->mutex);
    auto it = this->games.find(game_name);
    if (it != this->games.end()){
        return false;
    }

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

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

Jun 09, 18 14:59 **GamesList.cpp** Page 2/2

```
63          std::cout << "[INFO] El jugador '" << player_name << "' se unio a la partida '" << game
    _name << "'" << std::endl;
64      }
65      if (this->games[game_name]->isFull()){
66          std::lock_guard<std::mutex> lock(this->mutex_cout);
67          std::cout << "[INFO] Partida iniciada: " << game_name << std::endl;
68          this->games[game_name]->start();
69      }
70      return result;
71  }
72
73  void GamesList::checkGames(){
74      std::lock_guard<std::mutex> lock(this->mutex);
75      auto it = this->games.begin();
76      while (it != this->games.end()){
77          if (! it->second->isRunning()){
78              it->second->join();
79              std::lock_guard<std::mutex> lock(this->mutex_cout);
80              std::cout << "[INFO] Partida terminada: " << it->first << std::endl;
81              it = this->games.erase(it);
82          } else {
83              ++it;
84          }
85      }
86  }
```

Jun 07, 18 20:21 **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      public:
23          /* Constructor */
24          GamesList(Server& server, std::mutex& mutex_cout);
25
26          /* Destructor */
27          ~GamesList();
28
29          /* Agrega una patida nueva a la lista */
30          bool addGame(const std::string& game_name, const std::string& map, int m
    ax_players, Player& player);
31
32          /* Devuelve una lista con las partidas a las cuales se puede
33           * unir el jugador */
34          games_list_t getJoinableGames(const std::string& player_name);
35
36          /* Agrega un jugador a la partida */
37          bool addPlayer(const std::string& game_name, Player& player);
38
39          /* Verifica las partidas que terminaron */
40          void checkGames();
41  };
42
43  #endif
```

May 29, 18 13:35 **MapsList.cpp** Page 1/1

```cpp
#include "MapsList.h"
#include "Path.h"

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))){
        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 18:21 **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 (auto it = this->clients.begin(); it != this->clients.end(); ++it){
        (*it)->stop();
        (*it)->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);
    std::unique_ptr<Thread> t(new ClientHandler(std::move(protocol), this->games_list, this->mutex_cout));
    t->start();
    this->clients.push_back(std::move(t));
```



```cpp
}
```

Jun 07, 18 20:28 **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 07, 18 18:40 **DataSender.cpp** Page 1/3

```
#include "DataSender.h"

DataSender::DataSender(World& world, std::vector<Player>& players, GameParameter
s& parameters):
    objects(world.getObjectsList()), girders(world.getGirdersList()),
    players(players), mutex(world.getMutex()), active(false), sleep_time(paramet
ers.getDataSenderSleep()){

        for (size_t i = 0; i < this->players.size(); i++){
            std::unique_ptr<PlayerDataSender> sender(new PlayerDataSender(this->
players[i]));
            this->players_data_senders.push_back(std::move(sender));
            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;
        auto 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(File& image){
    Buffer data = ServerProtocol::sendFile(image);
    this->sendBuffer(data);
    this->notifyAll();
}

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

void DataSender::sendTurnData(int turn_time, int time_after_shoot){
    Buffer data = ServerProtocol::sendTurnData(turn_time, time_after_shoot);
```

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



```
129
130 void DataSender::sendEndGame(const std::string& winner){
131     Buffer data = ServerProtocol::sendEndGame(winner);
132     this->sendBuffer(data);
133     this->notifyAll();
134 }
135
136 void DataSender::sendEndTurn(){
137     Buffer data = this->players[0].getProtocol().sendEndTurn();
138     this->sendBuffer(data);
139     this->notifyAll();
140 }
141
142 bool DataSender::isActive(){
143     std::lock_guard<std::mutex> lock(this->mutex);
144     return this->active;
145 }
146
147 void DataSender::sendBuffer(const Buffer& buffer){
148     for (size_t i = 0; i < this->players.size(); i++){
149         if (this->players[i].isConnected()){
150             this->players_data_senders[i]->sendData(buffer);
151         }
152     }
153 }
154
155 void DataSender::notifyAll(){
156     for (size_t i = 0; i < this->players.size(); i++){
157         if (this->players[i].isConnected()){
158             this->players_data_senders[i]->notify();
159         }
160     }
161 }
```

```
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 <list>
11 #include <memory>
12
13 class DataSender: public Thread{
14     private:
15         std::list<physical_object_ptr>& objects;
16         std::list<physical_object_ptr>& girders;
17         std::vector<Player>& players;
18         std::vector<std::unique_ptr<PlayerDataSender>> players_data_senders;
19         std::mutex& mutex;
20         bool active;
21         int sleep_time;
22
23         void sendBuffer(const Buffer& buffer);
24         void notifyAll();
25
26     public:
27         DataSender(World& world, std::vector<Player>& players, GameParameters& p
   arameters);
28         ~DataSender();
29
30         //Envia constantemente los datos de los objetos
31         void run() override;
32
33         //Envia la imagen de fondo
34         void sendBackgroundImage(File& image);
35
36         //Envia los datos del turno
37         void sendTurnData(int turn_time, int time_after_shoot);
38
39         //Envia los datos de los jugadores
40         void sendPlayersId();
41
42         //Envia los datos de las vigas
43         void sendGirders();
44
45         //Envia las municiones de las armas
46         void sendWeaponsAmmo(std::map<std::string, unsigned int>& weapons);
47
48         //Envia que el jugador cambio de arma
49         void sendWeaponChanged(const std::string &weapon);
50
51         //Envia que el gusano actual salto
52         void sendMoveAction(char action);
53
54         //Envia que el jugador cambio el angulo de la mira
55         void sendUpdateScope(int angle);
56
57         //Envia que el jugador disparo un arma
58         void sendWeaponShot(const std::string& weapon);
59
60         //Envia la senial de comienzo del juego
61         void sendStartGame();
62
63         //Envia la senial de que inicia un nuevo turno
64         void sendStartTurn(int worm_id, int player_id, float wind);
65
```



```
66         //Envia la senial de terminar turno
67         void sendEndTurn();
68
69         //Envia la senial de que el juego termino
70         void sendEndGame(const std::string& winner);
71
72         //Devuelve true si sigue enviando datos
73         bool isActive();
74 };
75
76
77 #endif
```

Jun 07, 18 21:08 **PlayerDataReceiver.cpp** Page 1/1

```cpp
#include "PlayerDataReceiver.h"

PlayerDataReceiver::PlayerDataReceiver(Player& player, DataSender& data_sender):
    player(player), data_sender(data_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);
            this->data_sender.sendWeaponShot(this->player.getCurrentWorm().getCu
rrentWeapon());
            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;
            this->data_sender.sendWeaponShot(this->player.getCurrentWorm().getCu
rrentWeapon());
            this->player.getCurrentWorm().shoot(b2Vec2(pos_x, pos_y));
        }
    }
}
```

Jun 07, 18 11:55 **PlayerDataReceiver.h** Page 1/1

```cpp
#ifndef __PLAYERDATARECEIVER_H__
#define __PLAYERDATARECEIVER_H__

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

class PlayerDataReceiver: public Thread{
    private:
        Player& player;
        DataSender& data_sender;
        bool is_my_turn;
        std::mutex mutex;

        void analizeReceivedData(Buffer& data);

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

        ~PlayerDataReceiver();

        void run() override;

        void beginTurn();

        void endTurn();

};

#endif
```

May 28, 18 19:58 **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();
}
```

May 30, 18 20:03 **PlayerDataSender.h** Page 1/1

```cpp
#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:
        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
```

```cpp
#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());

```



```cpp

    b2Vec2 position = object->getPosition();
    Weapon* weapon = (Weapon*)object.get();
    std::string name = weapon->getName();

    ServerProtocol::sendStringBuffer(buffer, name);
    ServerProtocol::sendIntBuffer(buffer, position.x * UNIT_TO_SEND);
    ServerProtocol::sendIntBuffer(buffer, position.y * UNIT_TO_SEND);
}

Buffer ServerProtocol::sendStartGame(){
    Buffer buffer;
    buffer.setNext(START_GAME_ACTION);
    return buffer;
}

Buffer ServerProtocol::sendEndTurn(){
    Buffer buffer;
    buffer.setNext(END_TURN);
    return buffer;
}

Buffer ServerProtocol::sendStartTurn(int current_worm_id, int current_player_id, float wind){
    Buffer buffer;
    buffer.setNext(START_TURN);
    ServerProtocol::sendIntBuffer(buffer, current_worm_id);
    ServerProtocol::sendIntBuffer(buffer, current_player_id);
    ServerProtocol::sendIntBuffer(buffer, wind * UNIT_TO_SEND);
    return buffer;
}

Buffer ServerProtocol::sendTurnData(int turn_time, int time_after_shoot){
    Buffer buffer;
    ServerProtocol::sendIntBuffer(buffer, turn_time);
    ServerProtocol::sendIntBuffer(buffer, time_after_shoot);
    return buffer;
}

Buffer ServerProtocol::sendPlayerId(const Player& player){
    Buffer buffer;
    ServerProtocol::sendIntBuffer(buffer, player.getId());
    ServerProtocol::sendStringBuffer(buffer, player.getName());
    return buffer;
}

Buffer ServerProtocol::sendGirder(physical_object_ptr& object){
    Girder* girder = (Girder*)object.get();

    Buffer buffer;
    ServerProtocol::sendIntBuffer(buffer, girder->getSize());

    b2Vec2 position = object->getPosition();
    ServerProtocol::sendIntBuffer(buffer, position.x * UNIT_TO_SEND);
    ServerProtocol::sendIntBuffer(buffer, position.y * UNIT_TO_SEND);
    ServerProtocol::sendIntBuffer(buffer, girder->getRotation());
    return buffer;
}

Buffer ServerProtocol::sendWeaponAmmo(const std::string& weapon_name, int ammo){
    Buffer buffer;
    ServerProtocol::sendStringBuffer(buffer, weapon_name);
    ServerProtocol::sendIntBuffer(buffer, ammo);
    return buffer;
}
```

Jun 07, 18 14:35 **ServerProtocol.cpp** Page 3/3

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

Jun 07, 18 14:34 **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
9  class Player;
10
11 class ServerProtocol : public Protocol{
12     private:
13         //Carga los datos del gusano en el buffer
14         static void send_worm(physical_object_ptr& object, Buffer& buffer);
15
16         //Carga los datos del arma en el buffer
17         static void send_weapon(physical_object_ptr& weapon, Buffer& buffer);
18
19     public:
20         ServerProtocol(Socket&& socket);
21         ServerProtocol(ServerProtocol&& other);
22         ~ServerProtocol();
23
24         //Carga un nuevo objeto en el buffer
25         static Buffer sendObject(physical_object_ptr& object);
26
27         //Carga la informacion de un objeto muerto en el buffer
28         static Buffer sendDeadObject(physical_object_ptr& object);
29
30         //Carga la informacion de comienzo de juego
31         static Buffer sendStartGame();
32
33         //Carga la informacion de nuevo turno en el buffer
34         static Buffer sendStartTurn(int current_worm_id, int current_player_id,
   float wind);
35
36         //Carga la informacion del turno en el buffer
37         static Buffer sendTurnData(int turn_time, int time_after_shoot);
38
39         //Carga la informacion de un nuevo jugador en el buffer
40         static Buffer sendPlayerId(const Player& player);
41
42         //Carga la informacion de una viga en el buffer
43         static Buffer sendGirder(physical_object_ptr& girder);
44
45         //Carga la informacion de un arma en el buffer
46         static Buffer sendWeaponAmmo(const std::string& weapon_name, int ammo);
47
48         //Carga la informacion de cambio de arma en el buffer
49         static Buffer sendWeaponChanged(const std::string &weapon);
50
51         //Carga la informacion de arma disparada en el buffer
52         static Buffer sendWeaponShot(const std::string &weapon);
53
54         //Carga la informacion de que el gusano salto
55         static Buffer sendMoveAction(char action);
56
57         //Carga la informacion de cambio de angulo en el buffer
58         static Buffer sendUpdateScope(int angle);
59
60         //Carga la informacion de fin del juego en el buffer
61         static Buffer sendEndGame(const std::string& winner);
62
63         //Carga la informacion de fin del turno
64         static Buffer sendEndTurn();
65 };
```

Jun 07, 18 14:34 **ServerProtocol.h** Page 2/2

```
66
67  #endif
```

Jun 07, 18 20:54 **Game.cpp** Page 1/3

```
1   #include "Game.h"
2   #include "Girder.h"
3   #include "WeaponFactory.h"
4   #include "Server.h"
5
6   #define TURN_STEP 100 //milliseconds
7
8   Game::Game(size_t players, const std::string& config_file, const std::string& ma
    p, Server& server):
9       players(players), server(server), parameters(config_file, map), world(this->
    parameters){
10          this->running = true;
11      }
12
13  Game::~Game(){
14      this->world.stop();
15      this->world.join();
16      if (data_sender){
17          this->data_sender->stop();
18          this->data_sender->join();
19      }
20  }
21
22  bool Game::addPlayer(Player& player){
23      if (this->isFull()){
24          return false;
25      }
26
27      return this->turn.addPlayer(player);
28  }
29
30  bool Game::isFull(){
31      return this->players <= this->turn.getPlayersSize();
32  }
33
34  bool Game::playerCanJoin(const std::string& player_name){
35      if (this->isFull()){
36          return false;
37      }
38      return this->turn.playerCanJoin(player_name);
39  }
40
41  void Game::run(){
42      this->configure();
43      this->world.start();
44      this->data_sender->start();
45
46      std::this_thread::sleep_for(std::chrono::milliseconds(100));
47      this->waitToWorld();
48
49      while (!this->turn.gameEnded(this->world.getMutex())){
50          this->player_turn_active = true;
51          this->turn.beginTurn();
52          int worm_id = this->turn.getCurrentPlayer().getCurrentWorm().getId();
53          int player_id = this->turn.getCurrentPlayer().getId();
54          this->data_sender->sendStartTurn(worm_id, player_id, this->world.getWind
    ());
55
56          size_t current_turn_time = 0;
57          size_t max_turn_time = this->parameters.getTurnTime() * 1000;
58          bool time_reduced = false;
59          while(current_turn_time < max_turn_time){
60              std::this_thread::sleep_for(std::chrono::milliseconds(TURN_STEP));
61              current_turn_time += TURN_STEP;
62              Worm& current_worm = this->turn.getCurrentPlayer().getCurrentWorm();
63              if (current_worm.damageReceived() || this->turn.gameEnded(this->worl
```

```
    d.getMutex())){
64                  current_turn_time = max_turn_time;
65              }else if (!time_reduced && current_worm.hasShot()){
66                  current_turn_time = max_turn_time - this->parameters.getTimeAfte
    rShoot() * 1000;
67                  time_reduced = true;
68              }
69          }
70
71          this->turn.endTurn();
72          this->data_sender->sendEndTurn();
73          this->waitToWorld();
74          this->world.update();
75      }
76      std::this_thread::sleep_for(std::chrono::milliseconds(50));
77      this->data_sender->sendEndGame(this->turn.getWinner());
78      this->world.stop();
79      this->data_sender->stop();
80      this->data_sender->join();
81      auto& player_list = this->turn.getPlayers();
82      for (auto it = player_list.begin(); it != player_list.end(); ++it){
83          if (it->isConnected()){
84              this->server.addConnectedClient(std::move(it->getProtocol()));
85          }
86      }
87      this->running = false;
88  }
89
90  void Game::configure(){
91      this->data_sender.reset(new DataSender(this->world, this->turn.getPlayers(),
     this->parameters));
92      this->turn.startGame(*this->data_sender);
93
94      this->data_sender->sendStartGame();
95      this->data_sender->sendBackgroundImage(this->parameters.getBackgroundImage()
    );
96      this->data_sender->sendTurnData(this->parameters.getTurnTime(), this->parame
    ters.getTimeAfterShoot());
97      this->data_sender->sendPlayersId();
98
99      //Asignacion de gusanos
100     std::vector<b2Vec2>& worms_list = this->parameters.getWorms();
101     size_t size = worms_list.size();
102     for (size_t i = 0; i < size; i++){
103         this->turn.addWorm(this->world, this->parameters, worms_list[i], i);
104     }
105     this->turn.distributeWorms(size, this->parameters.getWormsLifeToAdd());
106
107     //Creacion de vigas
108     int max_height = 0;
109     std::vector<GirderParams>& girders_list = this->parameters.getGirders();
110     size = girders_list.size();
111     for (size_t i = 0; i < size; i++){
112         physical_object_ptr girder(new Girder(this->world, this->parameters, gir
    ders_list[i].len, girders_list[i].rotation));
113         this->world.addObject(girder, b2Vec2(girders_list[i].pos_x, girders_list
    [i].pos_y));
114         if (girders_list[i].pos_y > max_height){
115             max_height = girders_list[i].pos_y;
116         }
117     }
118     this->parameters.setMaxHeight(max_height);
119     this->data_sender->sendGirders();
120
121     //Municion de las armas
122     std::map<std::string, unsigned int>& ammo = this->parameters.getWeaponsAmmo(
```



```
    );
123     this->data_sender->sendWeaponsAmmo(ammo);
124     auto& player_list = this->turn.getPlayers();
125     for (auto it = player_list.begin(); it != player_list.end(); ++it){
126         it->setWeaponsAmmo(ammo);
127     }
128 }
129
130 void Game::endTurn(){
131     this->player_turn_active = false;
132 }
133
134 void Game::waitToWorld(){
135     while (this->world.isActive() || this->data_sender->isActive()){
136         std::this_thread::sleep_for(std::chrono::milliseconds(this->parameters.g
    etGameWaitingWorldSleep()));
137     }
138 }
```

Jun 07, 18 20:22 **Game.h** Page 1/1

```
#ifndef __GAME_H__
#define __GAME_H__

#include <vector>
#include <memory>
#include "Turn.h"
#include "GameParameters.h"
#include "Thread.h"
#include "Player.h"
#include "Worm.h"
#include "World.h"
#include "DataSender.h"

class Player;
class Server;

class Game: public Thread{
    private:
        size_t players;
        Server& server;
        GameParameters parameters;
        World world;
        Turn turn;
        std::unique_ptr<DataSender> data_sender;
        bool player_turn_active;

        /* Realiza la configuracion inicial de la partida */
        void configure();

        /* Espera a que los objetos dejen de moverse */
        void waitToWorld();

    public:
        /* Constructor */
        Game(size_t players, const std::string& config_file, const std::string&
map, Server& server);

        /* Destructor */
        ~Game();

        /* Agrega un jugador a la partida */
        bool addPlayer(Player& player);

        /* Devuelve true si la partida esta llena */
        bool isFull();

        /* Devuelve true si el jugador puede unirse a la partida */
        bool playerCanJoin(const std::string& player_name);

        /* Comienza la partida */
        void run() override;

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

#endif
```

Jun 07, 18 18:39 **GameParameters.cpp** Page 1/4

```
#include "GameParameters.h"
#include "ConfigFields.h"
#include "Path.h"
#include <algorithm>
#include <random>

#define WORLD_MAX_HEIGHT "world_max_height"

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));

    this->float_parameters[DATA_SENDER_SLEEP] = config[DATA_SENDER_SLEEP].as<flo
at>();
    this->float_parameters[GAME_WAITING_WORLD_SLEEP] = config[GAME_WAITING_WORLD
_SLEEP].as<float>();
    this->float_parameters[WORLD_SLEEP_AFTER_STEP] = config[WORLD_SLEEP_AFTER_ST
EP].as<float>();
    this->float_parameters[WORLD_TIME_STEP] = config[WORLD_TIME_STEP].as<float>(
);
    this->float_parameters[TURN_TIME] = config[TURN_TIME].as<float>();
    this->float_parameters[TIME_AFTER_SHOOT] = config[TIME_AFTER_SHOOT].as<float
>();

    this->float_parameters[WORMS_LIFE] = config_editor[WORMS_LIFE].as<float>();
    this->float_parameters[WORMS_LIFE_TO_ADD] = config[WORMS_LIFE_TO_ADD].as<flo
at>();
    this->float_parameters[WORM_VELOCITY] = config[WORM_VELOCITY].as<float>();
    this->float_parameters[WORM_EXPLOSION_VELOCITY] = config[WORM_EXPLOSION_VELO
CITY].as<float>();
    this->float_parameters[WORM_JUMP_VELOCITY] = config[WORM_JUMP_VELOCITY].as<f
loat>();
    this->float_parameters[WORM_ROLLBACK_VELOCITY] = config[WORM_ROLLBACK_VELOCI
TY].as<float>();
    this->float_parameters[WORM_JUMP_HEIGHT] = config[WORM_JUMP_HEIGHT].as<float
>();
    this->float_parameters[WORM_ROLLBACK_HEIGHT] = config[WORM_ROLLBACK_HEIGHT].
as<float>();
    this->float_parameters[WORM_HEIGHT_TO_DAMAGE] = config[WORM_HEIGHT_TO_DAMAGE
].as<float>();
    this->float_parameters[WORM_MAX_HEIGHT_DAMAGE] = config[WORM_MAX_HEIGHT_DAMA
GE].as<float>();
    this->float_parameters[WEAPONS_VELOCITY] = config[WEAPONS_VELOCITY].as<float
>();
    this->float_parameters[WIND_MIN_VELOCITY] = config[WIND_MIN_VELOCITY].as<flo
at>();
    this->float_parameters[WIND_MAX_VELOCITY] = config[WIND_MAX_VELOCITY].as<flo
at>();
    this->float_parameters[GRAVITY] = config[GRAVITY].as<float>();
    this->float_parameters[AIR_MISSILES_SEPARATION] = config[AIR_MISSILES_SEPARA
TION].as<float>();
    this->float_parameters[MAX_GIRDER_ROTATION_FRICTION] = config[MAX_GIRDER_ROT
ATION_FRICTION].as<float>();
    this->float_parameters[WORLD_MAX_HEIGHT] = 99999;

    this->weapon_radius = config[WEAPON_RADIUS].as<std::map<std::string, int>>()
;
    this->weapon_ammo = config_editor[WEAPON_AMMO].as<std::map<std::string, unsi
gned int>>();
    this->weapon_damage = config[WEAPON_DAMAGE].as<std::map<std::string, int>>()
;
    this->weapon_fragments = config[WEAPON_FRAGMENTS].as<std::map<std::string, i
nt>>();
```

```
44
45      std::vector<std::vector<float>> worms_file = config_editor[WORMS_DATA].as<st
    d::vector<std::vector<float>>>();
46      for (auto it = worms_file.begin(); it != worms_file.end(); ++it){
47          this->worms.push_back(b2Vec2((*it)[0], (*it)[1]));
48      }
49
50      std::vector<std::vector<float>> girders_file = config_editor[GIRDERS_DATA].a
    s<std::vector<std::vector<float>>>();
51      for (auto it = girders_file.begin(); it != girders_file.end(); ++it){
52          this->girders.push_back(GirderParams((*it)[0], (*it)[1], (*it)[2], (*it)
    [3]));
53      }
54
55      std::string background = BACKGROUND_PATH + config_editor[BACKGROUND_IMAGE].a
    s<std::string>();
56      this->background_image = std::move(File(background, FILE_READ_MODE));
57  }
58
59  GameParameters::~GameParameters(){}
60
61  int GameParameters::getWormLife(){
62      return this->float_parameters[WORMS_LIFE];
63  }
64
65  int GameParameters::getWormsLifeToAdd(){
66      return this->float_parameters[WORMS_LIFE_TO_ADD];
67  }
68
69  std::vector<b2Vec2>& GameParameters::getWorms(){
70      std::random_device rd;
71      std::mt19937 random(rd());
72
73      std::shuffle(this->worms.begin(), this->worms.end(), random);
74      return this->worms;
75  }
76
77  std::vector<GirderParams>& GameParameters::getGirders(){
78      return this->girders;
79  }
80
81  std::map<std::string, unsigned int>& GameParameters::getWeaponsAmmo(){
82      return this->weapon_ammo;
83  }
84
85  float GameParameters::getWormVelocity(){
86      return this->float_parameters[WORM_VELOCITY];
87  }
88
89  float GameParameters::getWormExplosionVelocity(){
90      return this->float_parameters[WORM_EXPLOSION_VELOCITY];
91  }
92
93  float GameParameters::getWormJumpVelocity(){
94      return this->float_parameters[WORM_JUMP_VELOCITY];
95  }
96
97  float GameParameters::getWormRollbackVelocity(){
98      return this->float_parameters[WORM_ROLLBACK_VELOCITY];
99  }
100
101 float GameParameters::getWormJumpHeight(){
102     return this->float_parameters[WORM_JUMP_HEIGHT];
103 }
104
105 float GameParameters::getWormRollbackHeight(){
```



```
106     return this->float_parameters[WORM_ROLLBACK_HEIGHT];
107 }
108
109 int GameParameters::getWormHeightToDamage(){
110     return this->float_parameters[WORM_HEIGHT_TO_DAMAGE];
111 }
112
113 int GameParameters::getWormMaxHeightDamage(){
114     return this->float_parameters[WORM_MAX_HEIGHT_DAMAGE];
115 }
116
117 float GameParameters::getWeaponsVelocity(){
118     return this->float_parameters[WEAPONS_VELOCITY];
119 }
120
121 int GameParameters::getWeaponDamage(const std::string& weapon){
122     return this->weapon_damage[weapon];
123 }
124
125 int GameParameters::getWeaponRadius(const std::string& weapon){
126     return this->weapon_radius[weapon];
127 }
128
129 int GameParameters::getWeaponFragments(const std::string& weapon){
130     return this->weapon_fragments[weapon];
131 }
132
133 float GameParameters::getWindMinVelocity(){
134     return this->float_parameters[WIND_MIN_VELOCITY];
135 }
136
137 float GameParameters::getWindMaxVelocity(){
138     return this->float_parameters[WIND_MAX_VELOCITY];
139 }
140
141 float GameParameters::getGravity(){
142     return this->float_parameters[GRAVITY];
143 }
144
145 float GameParameters::getAirMissilesSeparation(){
146     return this->float_parameters[AIR_MISSILES_SEPARATION];
147 }
148
149 int GameParameters::getMaxGirderRotationToFriction(){
150     return this->float_parameters[MAX_GIRDER_ROTATION_FRICTION];
151 }
152
153 void GameParameters::setMaxHeight(int height){
154     this->float_parameters[WORLD_MAX_HEIGHT] = height + 15;
155 }
156
157 int GameParameters::getMaxHeight(){
158     return this->float_parameters[WORLD_MAX_HEIGHT];
159 }
160
161 int GameParameters::getDataSenderSleep(){
162     return this->float_parameters[DATA_SENDER_SLEEP];
163 }
164
165 int GameParameters::getGameWaitingWorldSleep(){
166     return this->float_parameters[GAME_WAITING_WORLD_SLEEP];
167 }
168
169 int GameParameters::getWorldSleepAfterStep(){
170     return this->float_parameters[WORLD_SLEEP_AFTER_STEP];
171 }
```

Jun 07, 18 18:39 **GameParameters.cpp** Page 4/4

```
172
173 float GameParameters::getWorldTimeStep(){
174     return this->float_parameters[WORLD_TIME_STEP];
175 }
176
177 int GameParameters::getTurnTime(){
178     return this->float_parameters[TURN_TIME];
179 }
180
181 int GameParameters::getTimeAfterShoot(){
182     return this->float_parameters[TIME_AFTER_SHOOT];
183 }
184
185 File& GameParameters::getBackgroundImage(){
186     return this->background_image;
187 }
188
189 GameParameters::GirderParams::GirderParams(size_t len, float pos_x, float pos_y,
     int rotation):
190     len(len), pos_x(pos_x), pos_y(pos_y), rotation(rotation){}
```

Jun 07, 18 18:39 **GameParameters.h** Page 1/2

```
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 "File.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> float_parameters;
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         File background_image;
27
28     public:
29         //Inicializa todos los parametros necesarios para la partida
30         GameParameters(const std::string& config_file, const std::string& config
   _editor_file);
31         ~GameParameters();
32
33         int getWormLife();
34         int getWormsLifeToAdd();
35
36         std::vector<b2Vec2>& getWorms();
37         std::vector<GirderParams>& getGirders();
38         std::map<std::string, unsigned int>& getWeaponsAmmo();
39
40
41         float getWormVelocity();
42         float getWormExplosionVelocity();
43         float getWormJumpVelocity();
44         float getWormRollbackVelocity();
45         float getWormJumpHeight();
46         float getWormRollbackHeight();
47
48         int getWormHeightToDamage();
49         int getWormMaxHeightDamage();
50
51         float getWeaponsVelocity();
52
53         int getWeaponDamage(const std::string& weapon);
54         int getWeaponRadius(const std::string& weapon);
55         int getWeaponFragments(const std::string& weapon);
56
57         float getWindMinVelocity();
58         float getWindMaxVelocity();
59         float getGravity();
60         float getAirMissilesSeparation();
61
62         int getMaxGirderRotationToFriction();
63         void setMaxHeight(int height);
64         int getMaxHeight();
65
```

Jun 07, 18 18:39 **GameParameters.h** Page 2/2

```
66          int getDataSenderSleep();
67          int getGameWaitingWorldSleep();
68          int getWorldSleepAfterStep();
69          float getWorldTimeStep();
70
71          int getTurnTime();
72          int getTimeAfterShoot();
73
74          File& getBackgroundImage();
75  };
76
77  class GameParameters::GirderParams{
78      public:
79          size_t len;
80          float pos_x;
81          float pos_y;
82          int rotation;
83
84          GirderParams(size_t len, float pos_x, float pos_y, int rotation);
85  };
86
87  typedef GameParameters::GirderParams GirderParams;
88
89  #endif
```

Jun 07, 18 18:50 **Player.cpp** Page 1/2

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

Jun 07, 18 18:50 **Player.cpp** Page 2/2

```
65  }
66
67  void Player::changeWeapon(const std::string& weapon){
68      this->weapons.changeWeapon(weapon);
69  }
```

Jun 07, 18 18:49 **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
12  class Player{
13      private:
14          ServerProtocol protocol;
15          std::string name;
16          WormsList worms;
17          WeaponList weapons;
18          int id;
19          bool connected;
20
21      public:
22          Player(ServerProtocol&& protocol);
23
24          Player(Player&& other);
25
26          ~Player();
27
28          void setId(int id);
29
30          int getId() const;
31
32          //Devuelve el gusano actual del jugador
33          Worm& getCurrentWorm();
34
35          //Empieza el turno del jugador
36          void beginTurn();
37
38          //Agrega un nuevo gusano al jugador
39          void addWorm(World& world, GameParameters& parameters, const b2Vec2& pos
    ition, int id);
40
41          //Agrega vida a los gusanos del jugador
42          //en caso de que tenga menos gusanos que otros jugadores
43          void distributeWorms(size_t max, int life_to_add);
44
45          //Devuelve true si el jugador esta muerto
46          bool isDead();
47
48          //Devuelve true si el jugador esta desconectado
49          bool isConnected() const;
50
51          //Desconecta al jugador
52          void disconnect();
53
54          //Setea la municion de las armas
55          void setWeaponsAmmo(const std::map<std::string, unsigned int>& ammo);
56
57          //Cambia el arma actual del jugador
58          void changeWeapon(const std::string& weapon);
59
60          void setName(const std::string& name);
61
62          const std::string& getName() const;
63
64          ServerProtocol& getProtocol();
65
```

Jun 07, 18 18:49 **Player.h** Page 2/2

```
66  };
67
68  #endif
```

Jun 09, 18 15:00 **Turn.cpp** Page 1/2

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

Jun 09, 18 15:00 **Turn.cpp** Page 2/2

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

Jun 07, 18 12:44 **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& parameters, b2Vec2 position,
    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
62  #endif
```

Jun 07, 18 19:00 **WeaponList.cpp** Page 1/1

```cpp
#include "WeaponList.h"
#include "WeaponNames.h"
#include "WeaponFactory.h"

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 07, 18 20:05 **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& parameters);

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

#endif
```

Jun 07, 18 13:10 **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(ot
her.current){}

void WormsList::distribute(size_t max, int life_to_add){
    if (this->list.size() < max){
        for (auto it = this->list.begin(); it != this->list.end(); ++it){
            Worm* worm = (Worm*)it->get();
            worm->addLife(life_to_add);
        }
    }
}

bool WormsList::isEmpty(){
    for (auto it = this->list.begin(); it != this->list.end(); ++it){
        if (!(*it)->isDead()){
            return false;
        }
    }
    return true;
}

void WormsList::kill(){
    for (auto it = this->list.begin(); it != this->list.end(); ++it){
        if (!(*it)->isDead()){
            (*it)->kill();
        }
    }
}
```

Jun 05, 18 15:24 **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
```

May 26, 18 12:13 **CollisionData.cpp** Page 1/1

```cpp
#include "CollisionData.h"
#include "PhysicalObject.h"

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;
}
```

May 30, 18 20:03 **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();

        const std::string& getType();
        PhysicalObject* getObject();
};

#endif
```

Jun 06, 18 21:17 **CollisionListener.cpp** Page 1/2

```cpp
#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_BORDER)){
        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_GIRDER){
        bool friction = ((Girder *) dataB->getObject())->hasFriction();
        ((Worm *) dataA->getObject())->endCollissionGirder(friction);
    } else if (dataB->getType() == TYPE_WORM && contact->GetFixtureB()->IsSensor() && dataA->getType() == TYPE_GIRDER){
        bool friction = ((Girder *) dataA->getObject())->hasFriction();
        ((Worm *) dataB->getObject())->endCollissionGirder(friction);
```

Jun 06, 18 21:17 **CollisionListener.cpp** Page 2/2

```cpp
    }

    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 06, 18 20:55 **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
```

May 26, 18 12:13 **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& p
oint, const b2Vec2& normal, float32 fraction){
    b2Body* closest_body = fixture->GetBody();
    for (auto it = this->affected_worms.begin(); it != this->affected_worms.end(
); ++it){
        if (*it == closest_body){
            return -1;
        }
    }
    this->closest = closest_body;
    return fraction;
}
```

May 30, 18 20:03 **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

```

May 26, 18 12:13 **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);
}
```

May 30, 18 20:03 **BottomBorder.h** Page 1/1

```
#ifndef __BOTTOMBORDER_H__
#define __BOTTOMBORDER_H__

#include "PhysicalObject.h"

//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:
        BottomBorder(World& world);
        ~BottomBorder();

};

#endif
```

Jun 09, 18 14:13 **Girder.cpp** Page 1/1

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

Girder::Girder(World& world, GameParameters& parameters, size_t size, int rotation):
    PhysicalObject(world, 0, TYPE_GIRDER), size(size), rotation(rotation),
    max_rotation_to_friction(parameters.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(){
    return this->getAngle() < this->max_rotation_to_friction || this->getAngle() == 90;
}

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

Jun 05, 18 14:07 **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 rotation);
        ~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 06, 18 21:34 **PhysicalObject.cpp** Page 1/2

```
#include "PhysicalObject.h"
#include "World.h"

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();
    bool moved_x = (int)(pos.x * UNIT_TO_SEND) != (int)(this->last_position.x * UNIT_TO_SEND);
    bool moved_y = (int)(pos.y * UNIT_TO_SEND) != (int)(this->last_position.y * UNIT_TO_SEND);
    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();
}

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

Jun 06, 18 21:34 **PhysicalObject.cpp** Page 2/2

```cpp
}

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 06, 18 21:33 **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();

        int getId();

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

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

        //Colisiona con otro objeto
```

Jun 06, 18 21:33 **PhysicalObject.h** Page 2/2

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

Jun 02, 18 13:11 **AirAttack.cpp** Page 1/1

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

Jun 02, 18 13:11 **AirAttack.h** Page 1/1

```
1  #ifndef __SERVERAIRATTACK_H__
2  #define __SERVERAIRATTACK_H__
3
4  #include "Weapon.h"
5
6  class AirAttack: public Weapon{
7      private:
8          float missiles_separation;
9
10     public:
11
12         AirAttack(World& world, GameParameters& parameters);
13         ~AirAttack();
14
15         const std::string& getName() override;
16
17         void shoot(char dir, int angle, int power, int time, int shooter_id) ove
   rride;
18
19         void shoot(Worm& shooter, b2Vec2 pos) override;
20
21 };
22
23 #endif
```

May 29, 18 14:09 **AirAttackMissile.cpp** Page 1/1

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

May 26, 18 12:13 **AirAttackMissile.h** Page 1/1

```
#ifndef __SERVERAIRATTACKMISSILE_H__
#define __SERVERAIRATTACKMISSILE_H__

#include "Weapon.h"

class AirAttackMissile: public Weapon{
    public:

        AirAttackMissile(World& world, GameParameters& parameters);
        ~AirAttackMissile();

        const std::string& getName() override;

        bool isWindAffected() override;

};

#endif
```

Jun 05, 18 14:08 **Banana.cpp** Page 1/1

```
#include "Banana.h"
#include "b2Fixture.h"
#include "b2CircleShape.h"

Banana::Banana(World& world, GameParameters& parameters):
    Weapon(world, parameters, parameters.getWeaponDamage(BANANA_NAME), parameter
s.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);
}
```

May 26, 18 12:13 **Banana.h** Page 1/1

```
#ifndef __SERVERBANANA_H__
#define __SERVERBANANA_H__

#include "Weapon.h"

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

    public:

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

        const std::string& getName() override;

};

#endif
```

May 26, 18 12:13 **Bat.cpp** Page 1/1

```
#include "Bat.h"

Bat::Bat(World& world, GameParameters& parameters):
    Weapon(world, parameters, parameters.getWeaponDamage(BAT_NAME), parameters.g
etWeaponRadius(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;
}
```

May 26, 18 12:13 **Bat.h** Page 1/1

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

May 26, 18 12:13 **Bazooka.cpp** Page 1/1

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

May 26, 18 12:13 **Bazooka.h** Page 1/1

```
#ifndef __SERVERBAZOOKA_H__
#define __SERVERBAZOOKA_H__

#include "Weapon.h"

class Bazooka: public Weapon{
    public:

        Bazooka(World& world, GameParameters& parameters);
        ~Bazooka();

        const std::string& getName() override;
        bool isWindAffected() override;

};

#endif
```

Jun 05, 18 14:09 **Dynamite.cpp** Page 1/1

```
#include "Dynamite.h"

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 05, 18 14:09 **Dynamite.h** Page 1/1

```
#ifndef __SERVERDYNAMITE_H__
#define __SERVERDYNAMITE_H__

#include "Weapon.h"

class Dynamite: public Weapon{

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

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

#endif
```

Jun 06, 18 20:08 **FragmentableWeapon.cpp** Page 1/1

```
#include "FragmentableWeapon.h"
#include "WeaponFactory.h"
#include "Fragment.h"
#include "Math.h"

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

FragmentableWeapon::~FragmentableWeapon(){}

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

        b2Vec2 center = this->body->GetPosition() + 0.3 * b2Vec2(Math::cosDegree
s(fragment_angle),
                                                                 Math::sinDegrees(frag
ment_angle));
        ((Fragment *) fragment.get())->setShootPosition(center);
        ((Fragment*)fragment.get())->shoot(fragment_angle);
        this->world.addWeaponFragment(fragment);
    }
    Weapon::explode();
}
```

May 30, 18 20:03 **FragmentableWeapon.h** Page 1/1

```
#ifndef __FRAGMENTABLEWEAPON_H__
#define __FRAGMENTABLEWEAPON_H__

#include "Weapon.h"

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 06, 18 20:08 **Fragment.cpp** Page 1/1

```
#include "Fragment.h"

Fragment::Fragment(World& world, GameParameters& parameters, int damage, int radius):
    Weapon(world, parameters, damage, 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 06, 18 20:08 **Fragment.h** Page 1/1

```
#ifndef __SERVERFRAGMENT_H__
#define __SERVERFRAGMENT_H__

#include "Weapon.h"

class Fragment: public Weapon{
    private:
        b2Vec2 shoot_position;

    public:

        Fragment(World& world, GameParameters& parameters, int damage, int radius);
        ~Fragment();

        void setShootPosition(b2Vec2 pos);
        b2Vec2 getShootPosition();

        void shoot(int angle);

};

#endif
```

May 26, 18 12:13 **GreenGrenade.cpp** Page 1/1

```
#include "GreenGrenade.h"

GreenGrenade::GreenGrenade(World& world, GameParameters& parameters):
    Weapon(world, parameters, parameters.getWeaponDamage(GREEN_GRENADE_NAME), parameters.getWeaponRadius(GREEN_GRENADE_NAME)){}

GreenGrenade::~GreenGrenade(){}

const std::string& GreenGrenade::getName(){
    return GREEN_GRENADE_NAME;
}
```

May 26, 18 12:13 **GreenGrenade.h** Page 1/1

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

May 26, 18 12:13 **HolyGrenade.cpp** Page 1/1

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

May 26, 18 12:13 **HolyGrenade.h** Page 1/1

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

May 26, 18 12:13 **Mortar.cpp** Page 1/1

```
1  #include "Mortar.h"
2
3  Mortar::Mortar(World& world, GameParameters& parameters):
4      FragmentableWeapon(world, parameters, parameters.getWeaponDamage(MORTAR_NAME
   ), parameters.getWeaponFragments(MORTAR_NAME), parameters.getWeaponRadius(MORTAR
   _NAME)){}
5
6  Mortar::~Mortar(){}
7
8  const std::string& Mortar::getName(){
9      return MORTAR_NAME;
10 }
11
12 bool Mortar::isWindAffected(){
13     return true;
14 }
```

May 26, 18 12:13 **MortarFragment.cpp** Page 1/1

```cpp
#include "MortarFragment.h"

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;
}
```

May 26, 18 12:13 **MortarFragment.h** Page 1/1

```cpp
#ifndef __SERVERMORTARFRAGMENT_H__
#define __SERVERMORTARFRAGMENT_H__

#include "Fragment.h"

class MortarFragment: public Fragment{
    public:

        MortarFragment(World& world, GameParameters& parameters);
        ~MortarFragment();

        const std::string& getName() override;

        bool isWindAffected() override;

};

#endif
```

May 26, 18 12:13 **Mortar.h** Page 1/1

```
#ifndef __SERVERMORTAR_H__
#define __SERVERMORTAR_H__

#include "FragmentableWeapon.h"

class Mortar: public FragmentableWeapon{
    public:

        Mortar(World& world, GameParameters& parameters);
        ~Mortar();

        const std::string& getName() override;

        bool isWindAffected() override;
};

#endif
```

May 26, 18 12:13 **RedGrenade.cpp** Page 1/1

```
#include "RedGrenade.h"

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

RedGrenade::~RedGrenade(){}

const std::string& RedGrenade::getName(){
    return RED_GRENADE_NAME;
}
```

May 26, 18 12:13 **RedGrenadeFragment.cpp** Page 1/1

```cpp
#include "RedGrenadeFragment.h"

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

RedGrenadeFragment::~RedGrenadeFragment(){}

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

May 26, 18 12:13 **RedGrenadeFragment.h** Page 1/1

```cpp
#ifndef __SERVERREDGRENADEFRAGMENT_H__
#define __SERVERREDGRENADEFRAGMENT_H__

#include "Fragment.h"

class RedGrenadeFragment: public Fragment{
    public:

        RedGrenadeFragment(World& world, GameParameters& parameters);
        ~RedGrenadeFragment();

        const std::string& getName() override;

};

#endif
```

May 26, 18 12:13 **RedGrenade.h** Page 1/1

```h
#ifndef __SERVERREDGRENADE_H__
#define __SERVERREDGRENADE_H__

#include "FragmentableWeapon.h"

class RedGrenade: public FragmentableWeapon{
    public:

        RedGrenade(World& world, GameParameters& parameters);
        ~RedGrenade();

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

#endif
```

Jun 03, 18 21:28 **Teleportation.cpp** Page 1/1

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

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 02, 18 13:11 **Teleportation.h** Page 1/1

```
#ifndef __SERVERTELEPORTATION_H__
#define __SERVERTELEPORTATION_H__

#include "Weapon.h"

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 06, 18 21:18 **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& parameters, int damage, int radius)
:
    PhysicalObject(world, Weapon::weapon_id++, TYPE_WEAPON), parameters(parameters),
    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);
    }
```

Jun 06, 18 21:18 **Weapon.cpp** Page 2/2

```
63      this->waiting_to_explode = true;
64      this->explode_time.setTime(this->time_to_explode);
65      this->explode_time.start();
66  }
67
68
69  void Weapon::explode(){
70      b2Vec2 center = this->body->GetPosition();
71      for (float bullet_angle = 0; bullet_angle < 360; bullet_angle+= 5){
72          this->attackWormExplosion(center, bullet_angle);
73      }
74
75      this->explode_time.stop();
76      this->waiting_to_explode = false;
77      this->is_dead = true;
78  }
79
80  void Weapon::attackWormExplosion(const b2Vec2& center, int angle){
81      b2Vec2 end = center + this->radius * b2Vec2(Math::cosDegrees(angle), Math::s
    inDegrees(angle));
82      b2Body* closest_body = this->world.getClosestObject(&this->explosion, center
    , end);
83      if (closest_body){
84          Worm* worm = (Worm*)((CollisionData*)closest_body->GetUserData())->getOb
    ject();
85          float distance = b2Distance(center, worm->getPosition());
86          int worm_damage = this->damage * (1 - distance / (2 * this->radius)); //
    Justo en el borde hace la mitad de danio
87          worm->receiveWeaponDamage(worm_damage, center);
88      }
89  }
90
91  void Weapon::collideWithSomething(CollisionData *other){
92      if (this->time_to_explode == -1){
93          this->explode_time.stop();
94          this->explode();
95      } else if (other->getType() == TYPE_BORDER){
96          this->explode_time.stop();
97          this->is_dead = true;
98      }
99  }
100
101 int Weapon::getShooterId() const{
102     return this->shooter_id;
103 }
104
105 void Weapon::removeShooterId(){
106     this->shooter_id = -1;
107 }
```

May 26, 18 12:13 **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
30  }
```

May 30, 18 20:03 **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();

        void setTime(int time);

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

};

#endif
```

May 26, 18 12:13 **WeaponFactory.cpp** Page 1/1

```
#include "WeaponFactory.h"
#include "WeaponNames.h"

#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(this->world, this->parameters));
    } else if (name == DYNAMITE_NAME){
        return physical_object_ptr(new Dynamite(this->world, this->parameters));
    } else if (name == RED_GRENADE_NAME){
        return physical_object_ptr(new RedGrenade(this->world, this->parameters)
);
    } else if (name == RED_GRENADE_FRAGMENTS_NAME){
        return physical_object_ptr(new RedGrenadeFragment(this->world, this->par
ameters));
    } else if (name == GREEN_GRENADE_NAME){
        return physical_object_ptr(new GreenGrenade(this->world, this->parameter
s));
    } else if (name == HOLY_GRENADE_NAME){
        return physical_object_ptr(new HolyGrenade(this->world, this->parameters
));
    } else if (name == MORTAR_NAME){
        return physical_object_ptr(new Mortar(this->world, this->parameters));
    } else if (name == MORTAR_FRAGMENTS_NAME){
        return physical_object_ptr(new MortarFragment(this->world, this->paramet
ers));
    } else if (name == BANANA_NAME){
        return physical_object_ptr(new Banana(this->world, this->parameters));
    } else if (name == BAT_NAME){
        return physical_object_ptr(new Bat(this->world, this->parameters));
    } else if (name == TELEPORT_NAME){
        return physical_object_ptr(new Teleportation(this->world, this->paramete
rs));
    } else if (name == AIR_ATTACK_NAME){
        return physical_object_ptr(new AirAttack(this->world, this->parameters))
;
    } else if (name == AIR_ATTACK_MISSILE_NAME){
        return physical_object_ptr(new AirAttackMissile(this->world, this->param
eters));
    }

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

May 30, 18 20:03 **WeaponFactory.h** Page 1/1

```
#ifndef __WEAPONFACTORY_H__
#define __WEAPONFACTORY_H__

#include "World.h"
#include "GameParameters.h"

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 06, 18 21:17 **Weapon.h** Page 1/1

```
#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();

        virtual void collideWithSomething(CollisionData *other) override;

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

        virtual const std::string& getName() = 0;

        int getShooterId() const;

        void removeShooterId();

};

#endif
```

```cpp
#include "Worm.h"
#include "b2CircleShape.h"
#include "b2PolygonShape.h"
#include "b2Fixture.h"
#include "Protocol.h"
#include "WeaponFactory.h"
#include "Girder.h"
#include "Math.h"
#include <algorithm>

Worm::Worm(World& world, GameParameters& parameters, int id, int player_id, WeaponList& weapons):
    PhysicalObject(world, id, TYPE_WORM), player_id(player_id), life(parameters.getWormLife()),
    dir(1), parameters(parameters), weapons(weapons), max_height(0), colliding_with_girder(0), friction(0),
    movement_allowed(false), angle(0), has_shot(false), damage_received(false){}

Worm::~Worm(){}

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

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

    b2FixtureDef fixtureDef;
    fixtureDef.shape = &circleShape;
    fixtureDef.density = 10;
    this->body->CreateFixture(&fixtureDef);
    this->body->SetFixedRotation(true);

    //Sensor para colisiones
    b2PolygonShape sensorShape;
    sensorShape.SetAsBox(worm_size * 0.5 * 0.7, worm_size / 5, b2Vec2(0, -1 * worm_size / 2), 0);

    b2FixtureDef sensorFixtureDef;
    sensorFixtureDef.shape = &sensorShape;
    sensorFixtureDef.isSensor = true;
    this->body->CreateFixture(&sensorFixtureDef);
}

int Worm::getPlayerId() const{
    return this->player_id;
}

int Worm::getLife() const{
    return this->life;
}

char Worm::getDir() const{
    return this->dir;
}

bool Worm::isColliding() const{
    return this->colliding_with_girder && !this->movement_allowed;
}

const std::string& Worm::getCurrentWeapon() const{
    physical_object_ptr weapon = this->weapons.getCurrentWeapon(this->world, this->parameters);
```



```cpp
    return ((Weapon*)weapon.get())->getName();
}

void Worm::addLife(int life){
    this->life += life;
}

void Worm::reduceLife(size_t damage){
    this->life -= damage;
    this->damage_received = true;
    this->data_updated = true;
    if (this->life <= 0){
        this->life = 0;
        this->is_dead = true;
    }
}

bool Worm::move(char action){
    if (!this->colliding_with_girder || this->movement_allowed){
        return false;
    }
    this->movement_allowed = false;
    if (action == MOVE_RIGHT){
        this->dir = action;
        b2Vec2 velocity(parameters.getWormVelocity(), 0);
        this->world.setLinearVelocity(*this, velocity);
    } else if (action == MOVE_LEFT){
        this->dir = action;
        b2Vec2 velocity(-1 * parameters.getWormVelocity(), 0);
        this->world.setLinearVelocity(*this, velocity);
    } else {

        this->movement_allowed = true;
        if (action == JUMP){
            b2Vec2 velocity(parameters.getWormJumpVelocity(), parameters.getWormJumpHeight());
            velocity.x *= this->dir;
            this->world.setLinearVelocity(*this, velocity);
        } else if (action == ROLLBACK){
            b2Vec2 velocity(parameters.getWormRollbackVelocity(), parameters.getWormRollbackHeight());
            velocity.x *= -1 * this->dir;
            this->world.setLinearVelocity(*this, velocity);
        }
    }
    return true;
}

void Worm::shoot(int angle, int power, int time){
    if (!this->weapons.shoot()){
        return;
    }
    b2Vec2 pos = this->getPosition();
    int shooter_id = this->id;
    float x_add = (worm_size * this->dir);;
    float y_add = worm_size;
    if (angle > MAX_WEAPON_ANGLE){
        shooter_id = -1;
        x_add *= Math::cosDegrees(this->angle);
        y_add *= Math::sinDegrees(this->angle);
    } else {
        float factor = (this->getCurrentWeapon() == BAT_NAME ? 0.2 : 0.7);
        x_add *= Math::cosDegrees(angle) * factor;
        y_add *= Math::sinDegrees(angle) * factor;
    }

```

```
126     pos.x += x_add;
127     pos.y += y_add;
128 
129     physical_object_ptr weapon = this->weapons.getCurrentWeapon(this->world, thi
    s->parameters);
130     ((Weapon*)weapon.get())->shoot(this->dir, angle, power, time, shooter_id);
131     this->world.addObject(weapon, pos);
132     this->has_shot = true;
133 }
134 
135 void Worm::shoot(b2Vec2 pos){
136     if (!this->weapons.shoot()){
137         return;
138     }
139     ((Weapon*)this->weapons.getCurrentWeapon(this->world, this->parameters).get(
    ))->shoot(*this, pos);
140     this->has_shot = true;
141 }
142 
143 void Worm::receiveWeaponDamage(int damage, const b2Vec2 &epicenter){
144     this->reduceLife(damage);
145     b2Vec2 direction = this->body->GetPosition() - epicenter;
146     direction.Normalize();
147     this->body->SetGravityScale(1);
148     this->movement_allowed = true;
149     this->body->SetLinearVelocity(damage * parameters.getWormExplosionVelocity()
     * direction);
150 }
151 
152 void Worm::collideWithSomething(CollisionData *other){
153     if (other->getType() == TYPE_BORDER){
154         this->kill();
155     } else if(other->getType() == TYPE_GIRDER){
156         int min_height = parameters.getWormHeightToDamage();
157         float current_height = this->body->GetPosition().y;
158         this->max_height -= current_height;
159 
160         if (this->max_height >= min_height){
161             this->reduceLife(std::min((int) this->max_height - min_height + 1, p
    arameters.getWormMaxHeightDamage()));
162         }
163         this->max_height = 0;
164         this->colliding_with_girder ++;
165         Girder* girder = (Girder*)other->getObject();
166         if (girder->hasFriction()){
167             this->friction++;
168             this->movement_allowed = false;
169             this->angle = girder->getAngle();
170         }
171     }
172 }
173 
174 void Worm::endCollissionGirder(char has_friction){
175     this->friction -= has_friction;
176     this->colliding_with_girder --;
177     if (this->friction <= 0){
178         this->friction = 0;
179         this->body->SetGravityScale(1);
180         this->angle = 0;
181     }
182 }
183 
184 bool Worm::isActive(){
185     if (!this->colliding_with_girder){
186         float height = this->body->GetPosition().y;
187         this->max_height = std::max(this->max_height, height);
```



```
188     } else if (this->friction && !this->movement_allowed){
189         this->body->SetGravityScale(0);
190         this->body->SetLinearVelocity(b2Vec2(0, 0));
191     }
192     if (!this->body->IsAwake()){
193         this->movement_allowed = false;
194     } else if (!this->friction){
195         this->movement_allowed = true;
196     }
197     return PhysicalObject::isActive();
198 }
199 
200 bool Worm::hasShot() const{
201     return this->has_shot;
202 }
203 
204 bool Worm::damageReceived() const{
205     return this->damage_received || this->is_dead;
206 }
207 
208 void Worm::beginTurn(){
209     this->has_shot = false;
210     this->damage_received = false;
211 }
```

```
#ifndef __WORM_H__
#define __WORM_H__

#include "PhysicalObject.h"
#include "GameParameters.h"
#include "Weapon.h"
#include "WeaponList.h"

class Worm: public PhysicalObject{
    private:
        int player_id;
        int life;
        char dir;
        GameParameters& parameters;
        WeaponList& weapons;
        float max_height;
        int colliding_with_girder;
        int friction;
        bool movement_allowed;
        int angle;

        bool has_shot;
        bool damage_received;

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

    public:
        Worm(World& world, GameParameters& parameters, int id, int player_id, We
aponList& weapons);
        ~Worm();

        int getPlayerId() const;
        int getLife() const;
        char getDir() const;
        bool isColliding() const;
        const std::string& getCurrentWeapon() const;

        //Aumenta la vida del gusano
        void addLife(int life);

        //Reduce la vida del gusano
        void reduceLife(size_t damage);

        //Ejecuta una accion de movimiento del gusano
        bool move(char action);

        //Dispara un arma no teledirigida
        void shoot(int angle, int power, int time);

        //Dispara un arma teledirigida
        void shoot(b2Vec2 pos);

        //Analiza la colision con el objeto
        void collideWithSomething(CollisionData *other) override;

        //Analiza el fin del contacto con una viga
        void endCollissionGirder(char friction);

        //Recibe danio de un arma o una explosion
        void receiveWeaponDamage(int damage, const b2Vec2 &epicenter);

        //Devuelve true si el gusano esta en movimiento
        bool isActive() override;

```



```
        //Devuelve true si el gusano disparo
        bool hasShot() const;

        //Devuelve true si el gusano recibio danio
        bool damageReceived() const;

        //Empieza el turno del gusano
        void beginTurn();
};

#endif
```

May 26, 18 12:13 **Wind.cpp** Page 1/1

```cpp
#include "Wind.h"
#include <random>

Wind::Wind(GameParameters& parameters):
    min_velocity(parameters.getWindMinVelocity()),
    max_velocity(parameters.getWindMaxVelocity()){
        this->update();
    }

Wind::~Wind(){}

float Wind::getVelocity() const{
    return this->velocity;
}

void Wind::update(){
    std::mt19937 rng;
    rng.seed(std::random_device()());
    std::uniform_real_distribution<float> distribution(this->min_velocity, this->max_velocity);
    std::uniform_int_distribution<int> direction(-1, 1); //Acepto velocidad 0

    this->velocity = distribution(rng);
    this->velocity *= direction(rng);
}
```

Jun 05, 18 14:07 **Wind.h** Page 1/1

```cpp
#ifndef __WIND_H__
#define __WIND_H__

#include "GameParameters.h"

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

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

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

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


#endif
```

Jun 05, 18 14:07 **World.cpp** Page 1/3

```cpp
#include "World.h"
#include "Weapon.h"
#include "BottomBorder.h"
#include "b2WorldCallbacks.h"
#include "Fragment.h"

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 (auto it = this->objects.begin(); it != this->objects.end(); it++){
            if ((*it)->isDead()){
                this->removeObject(*it);
            } else if ((*it)->isActive()){
                this->is_active = true;
                b2Body* body = (*it)->getBody();
                if (body && (*it)->isWindAffected()){
                    body->ApplyForceToCenter(b2Vec2(this->wind.getVelocity(), 0), false);
                }
            }
        }
    }
}

void World::addAllFragments(){
    std::lock_guard<std::mutex> lock(this->mutex);

    for (auto it = this->fragments_to_add.begin(); it != this->fragments_to_add.end(); it++){
        b2BodyDef body_def;
        b2Vec2 pos = ((Fragment *) it->get())->getShootPosition();
        (*it)->getBodyDef(body_def, pos);
        this->initializeObject(*it, &body_def);
    }
    this->fragments_to_add.clear();
}

bool World::isActive(){
```

Jun 05, 18 14:07 **World.cpp** Page 2/3

```cpp
    std::lock_guard<std::mutex> lock(this->mutex);
    return this->is_active;
}

void World::update(){
    std::lock_guard<std::mutex> lock(this->mutex);
    this->wind.update();
}

void World::addObject(physical_object_ptr object, const b2Vec2& pos){
    b2BodyDef body_def;
    object->getBodyDef(body_def, pos);

    std::lock_guard<std::mutex> lock(this->mutex);
    this->initializeObject(object, &body_def);
}

void World::initializeObject(physical_object_ptr object, b2BodyDef* body_def){
    object->initializeBody(this->world.CreateBody(body_def));
    if (body_def->type != b2_staticBody){
        this->objects.push_back(object);
    } else {
        this->girders.push_back(object);
    }
}

void World::addWeaponFragment(physical_object_ptr fragment){
    this->fragments_to_add.push_back(fragment);
}

void World::removeTimedWeapon(Weapon& weapon){
    b2Body* body = weapon.getBody();
    if (body){
        this->world.DestroyBody(body);
        weapon.destroyBody();
    }
}

void World::removeObject(physical_object_ptr object){
    b2Body* body = object->getBody();
    if (body){
        this->world.DestroyBody(body);
        object->destroyBody();
    }
}

void World::initialize(){
    physical_object_ptr bottom_border(new BottomBorder(*this));
    this->addObject(bottom_border, b2Vec2(0, 0));
}

void World::setLinearVelocity(PhysicalObject& object, b2Vec2& velocity){
    std::lock_guard<std::mutex> lock(this->mutex);
    b2Body* body = object.getBody();
    if (body){
        body->SetGravityScale(1);
        body->SetLinearVelocity(velocity);
    }
}

b2Body* World::getClosestObject(RayCastWeaponExploded* callback, b2Vec2 center, b2Vec2 end){
    this->world.RayCast(callback, center, end);
    return callback->getClosestWorm();
}

```

Jun 05, 18 14:07 **World.cpp** Page 3/3

```
126 float World::getWind() const{
127     return this->wind.getVelocity();
128 }
129
130 std::list<physical_object_ptr>& World::getObjectsList(){
131     return this->objects;
132 }
133
134 std::list<physical_object_ptr>& World::getGirdersList(){
135     return this->girders;
136 }
137
138 std::mutex& World::getMutex(){
139     return this->mutex;
140 }
```

Jun 05, 18 14:07 **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         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 05, 18 14:07 **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, b2Vec2 center,
     b2Vec2 end);
70
71          std::list<physical_object_ptr>& getObjectsList();
72          std::list<physical_object_ptr>& getGirdersList();
73
74          std::mutex& getMutex();
75  };
76
77
78  #endif
```

Jun 02, 18 13:00 **main.cpp** Page 1/1

```
1   #include "Server.h"
2   #include "yaml.h"
3   #include "ConfigFields.h"
4   #include "Path.h"
5   #include <iostream>
6   #include <mutex>
7
8   #define EXIT_CHAR 'q'
9
10  int main(int argc, const char* argv[]){
11      std::mutex mutex_cout;
12      try{
13          YAML::Node config(YAML::LoadFile(SERVER_CONFIG_FILE));
14          Server server(config[SERVER_PORT].as<std::string>(), mutex_cout);
15          std::cout << "[LOG] Server iniciado." << std::endl;
16          server.start();
17          while (std::cin.get() != EXIT_CHAR){}
18          {
19              std::lock_guard<std::mutex> lock(mutex_cout);
20              std::cout << "[LOG] Comenzando el cierre del servidor." << std::endl;
21          }
22          server.stop();
23          server.join();
24      } catch(const std::exception& e){
25          std::lock_guard<std::mutex> lock(mutex_cout);
26          std::cout << "[ERROR] " << e.what() << std::endl;
27      }
28      return 0;
29  }
```

Jun 09, 18 18:50 **Table of Content** Page 1/2

Table of Contents

Jun 09, 18 18:50 **Table of Content** Page 2/2