May 31, 18 12:08 **ExplosionView.cpp** Page 1/1

```cpp
#include "ExplosionView.h"
#include <gtkmm/image.h>
#include <glibmm/main.h>
#include "Path.h"

ExplosionView::ExplosionView(BulletView&& bullet) : bulletView(std::move(bullet)){
    this->animation = Gdk::Pixbuf::create_from_file(EXPLOSION_ANIMATION);
    int width = this->animation->get_width();
    int height = this->animation->get_height();
    for (int i = 0; i < height/width; i++) {
        Glib::RefPtr<Gdk::Pixbuf> aux = Gdk::Pixbuf::create_subpixbuf(this->animation, 0, i * width, width, width);
        this->animation_vector.push_back(aux);
    }
    this->iter = this->animation_vector.begin();
}

ExplosionView::~ExplosionView() {}

ExplosionView::ExplosionView(ExplosionView&& other) :
        bulletView(std::move(other.bulletView)){
    this->animation_vector = other.animation_vector;
    this->animation = other.animation;
    this->iter = this->animation_vector.begin();
}

bool ExplosionView::startCallBack() {
    Gtk::Image& image = (Gtk::Image&)this->bulletView.getWidget();
    image.set(*(this->iter));
    this->iter++;
    if (this->iter == this->animation_vector.end()) {
        this->bulletView.removeFromWorld();
        return false;
    }
    return true;
}

void ExplosionView::start() {
    Glib::signal_timeout().connect(sigc::mem_fun(*this, &ExplosionView::startCallBack), 40);
}

bool ExplosionView::hasFinished() {
    return this->iter == this->animation_vector.end();
}
```

May 31, 18 12:08 **ExplosionView.h** Page 1/1

```cpp
#ifndef __CLIENTEXPLOSIONVIEW_H__
#define __CLIENTEXPLOSIONVIEW_H__

#include <vector>
#include <gdkmm/pixbuf.h>
#include "BulletView.h"

/* Clase que se encarga de reproducir la animacion de una explosion */
class ExplosionView {
    private:
        BulletView bulletView;
        std::vector<Glib::RefPtr<Gdk::Pixbuf>> animation_vector;
        Glib::RefPtr<Gdk::Pixbuf> animation;
        std::vector<Glib::RefPtr<Gdk::Pixbuf>>::iterator iter;

        /* Callback de start */
        bool startCallBack();

    public:
        /* Constructor */
        ExplosionView(BulletView&& bullet);

        /* Destructor */
        ~ExplosionView();

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


        /* Realiza la animacion de la explosion */
        void start();

        /* Devuelve true si la animacion de la explosion finalizo */
        bool hasFinished();
};


#endif
```

```cpp
#include "ExplosionViewList.h"

ExplosionViewList::ExplosionViewList() {}

ExplosionViewList::~ExplosionViewList() {}

void ExplosionViewList::check() {
    std::list<ExplosionView>::iterator iter;
    iter = this->animations.begin();
    while (iter != this->animations.end()) {
        if (iter->hasFinished()) {
            iter = this->animations.erase(iter);
        } else {
            ++iter;
        }
    }
}

void ExplosionViewList::addAndStart(ExplosionView&& animation) {
    this->check();
    this->animations.push_back(std::move(animation));
    this->animations.back().start();
}
```

```cpp
#ifndef WORMS_EXPLOSIONVIEWLIST_H
#define WORMS_EXPLOSIONVIEWLIST_H

#include <list>
#include "ExplosionView.h"

/* Clase que se encarga de almacenar animaciones de explosiones */
class ExplosionViewList {
    private:
        std::list<ExplosionView> animations;

        /* Verifica si alguna animacion de la lista finalizo y las
         * elimina de la lista */
        void check();

    public:
        /* Constructor */
        ExplosionViewList();

        /* Destructor */
        ~ExplosionViewList();


        /* Agrega una animacion de explosion a la lista y la reproduce */
        void addAndStart(ExplosionView&& animation);

};


#endif //WORMS_EXPLOSIONVIEWLIST_H
```

```cpp
#include "WalkingAnimation.h"
#include "Path.h"
#include "ObjectSizes.h"

#define DIR_RIGHT 1
#define DIR_LEFT -1

WalkingAnimation::WalkingAnimation(Gtk::Image* worm_image) : worm_image(worm_ima
ge),
    dir(DIR_RIGHT) {
    this->walk_image = Gdk::Pixbuf::create_from_file(WORMS_PATH + "walk.png");
    int width = this->walk_image->get_width();
    int height = this->walk_image->get_height();
    for (int i = 0; i < height / WORM_IMAGE_WIDTH; i++) {
        walk_queue.push(Gdk::Pixbuf::create_subpixbuf(this->walk_image, 0, i * W
ORM_IMAGE_WIDTH, width, WORM_IMAGE_WIDTH));
    }
}

WalkingAnimation::~WalkingAnimation() {}

WalkingAnimation::WalkingAnimation(WalkingAnimation&& other) :
    walk_queue(std::move(other.walk_queue)), walk_image(std::move(other.walk_ima
ge)),
    worm_image(other.worm_image), dir(other.dir) {}

void WalkingAnimation::setMovementImage(char new_dir) {
    if (new_dir == this->dir){
        this->walk_queue.push(std::move(this->walk_queue.front()));
        this->walk_queue.pop();
    }
    this->dir = new_dir;
    this->setStaticImage();
}

void WalkingAnimation::setStaticImage() {
    this->worm_image->set(Gdk::Pixbuf::create_subpixbuf(this->walk_queue.back(),
 WORM_IMAGE_WIDTH + this->dir * WORM_IMAGE_WIDTH, 0, WORM_IMAGE_WIDTH, WORM_IMAG
E_WIDTH));
}

void WalkingAnimation::updateWormImage(Gtk::Image* worm_image) {
    this->worm_image = worm_image;
}

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

```cpp
#ifndef WORMS_WALKINGANIMATION_H
#define WORMS_WALKINGANIMATION_H

#include <gtkmm/image.h>
#include <gdkmm/pixbuf.h>
#include <queue>

/* Clase que se encarga de actualizar la imagen del worm al
 * moverse obteniendo una animacion del worm caminando */
class WalkingAnimation {
    private:
        std::queue<Glib::RefPtr<Gdk::Pixbuf>> walk_queue;
        Glib::RefPtr<Gdk::Pixbuf> walk_image;
        Gtk::Image* worm_image;
        char dir;

    public:
        /* Constructor*/
        WalkingAnimation(Gtk::Image* worm_image);

        /* Destructor */
        ~WalkingAnimation();

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


        /* Actualiza la imagen del worm por la siguiente
         * imagen del worm caminando */
        void setMovementImage(char new_dir);

        /* Setea la imagen del worm por la imagen actual del
         * worm caminando */
        void setStaticImage();

        /* Devuelve la direccion del worm */
        char getDir() const;

        /* Actualiza el puntero de la imagen del worm */
        void updateWormImage(Gtk::Image* worm_image);
};


#endif //WORMS_WALKINGANIMATION_H
```

```cpp
#include "WeaponAnimation.h"
#include <glibmm/main.h>
#include "WormView.h"
#include "Path.h"
#include "ObjectSizes.h"
#include "WeaponNames.h"

#define DIR_RIGHT 1

WeaponAnimation::WeaponAnimation(const std::string& weapon, Gtk::Image* worm_image) :
    worm_image(worm_image), angle(DEFAULT_ANGLE) {
    this->updateWeaponImage(weapon);
}

WeaponAnimation::~WeaponAnimation() {}

WeaponAnimation::WeaponAnimation(WeaponAnimation&& other) :
    scope_vector(std::move(other.scope_vector)),
    scope_image(std::move(other.scope_image)),
    worm_image(other.worm_image),
    angle(other.angle) {}

void WeaponAnimation::updateWeaponImage(const std::string& weapon) {
    this->scope_vector.clear();
    this->scope_image = Gdk::Pixbuf::create_from_file(WORMS_PATH + weapon + "_scope.png");
    int width = this->scope_image->get_width();
    int height = this->scope_image->get_height();
    for (int i = 0; i < height / WORM_IMAGE_WIDTH; i++) {
        this->scope_vector.push_back(Gdk::Pixbuf::create_subpixbuf(scope_image, 0, i * WORM_IMAGE_WIDTH, width, WORM_IMAGE_WIDTH));
    }
}

void WeaponAnimation::changeWeapon(const std::string& weapon, char dir) {
    this->updateWeaponImage(weapon);
    this->setWeaponImage(dir);
}

void WeaponAnimation::setWeaponImage(char dir) {
    int width = this->scope_vector[(90 + this->angle) / 6]->get_width() / 3;
    int height = this->scope_vector[(90 + this->angle) / 6]->get_height();
    this->worm_image->set(Gdk::Pixbuf::create_subpixbuf(this->scope_vector[(90 + this->angle) / 6], width + dir * width, 0, width, height));
}

bool WeaponAnimation::batHitCallBack(std::vector<Glib::RefPtr<Gdk::Pixbuf>>::iterator& iter, const int width, char dir) {
    this->worm_image->set(Gdk::Pixbuf::create_subpixbuf(*iter, 0, 0, width, WORM_IMAGE_WIDTH));
    ++iter;
    if (iter == this->scope_vector.end()) {
        this->updateWeaponImage(BAT_NAME);
        this->setWeaponImage(dir);
        return false;
    }
    return true;
}

void WeaponAnimation::weaponShootAnimation(const std::string &weapon, char dir)
{
    if (weapon != BAT_NAME) {
        return;
    }
    this->scope_image = Gdk::Pixbuf::create_from_file(BAT_HIT_ANIMATION);
```

```cpp
    int width = this->scope_image->get_width() / 3;
    int height = this->scope_image->get_height();
    int pos_x = width + dir * width;
    this->scope_vector.clear();
    for (int i = 0; i < height / WORM_IMAGE_WIDTH; i++) {
        this->scope_vector.push_back(Gdk::Pixbuf::create_subpixbuf(scope_image, pos_x, i * WORM_IMAGE_WIDTH, width, WORM_IMAGE_WIDTH));
    }
    std::vector<Glib::RefPtr<Gdk::Pixbuf>>::iterator iter = this->scope_vector.begin();
    sigc::slot<bool> my_slot = sigc::bind(sigc::mem_fun(*this, &WeaponAnimation::batHitCallBack), iter, width, dir);
    Glib::signal_timeout().connect(my_slot, 12);
}

void WeaponAnimation::changeAngle(int angle, char dir) {
    this->angle = angle;
    this->setWeaponImage(dir);
}

void WeaponAnimation::updateWormImage(Gtk::Image* worm_image) {
    this->worm_image = worm_image;
}
```

```cpp
#ifndef WORMS_WEAPONANIMATION_H
#define WORMS_WEAPONANIMATION_H

#include <gtkmm/image.h>
#include <gdkmm/pixbuf.h>
#include <vector>
#include <string>

class WormView;

class WeaponAnimation {
    private:
        std::vector<Glib::RefPtr<Gdk::Pixbuf>> scope_vector;
        Glib::RefPtr<Gdk::Pixbuf> scope_image;
        Gtk::Image* worm_image;
        int angle;

        /* Actualiza las imagenes por las imagenes del arma nueva */
        void updateWeaponImage(const std::string& weapon);

        /* Callback */
        bool batHitCallBack(std::vector<Glib::RefPtr<Gdk::Pixbuf>>::iterator& it
er, const int width, char dir);

    public:
        /* Constructor */
        WeaponAnimation(const std::string& weapon, Gtk::Image* worm_image);

        /* Destructor */
        ~WeaponAnimation();

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


        /* Cambia la imagen del worm con el arma actual por una imagen
         * del worm con la nueva arma */
        void changeWeapon(const std::string& weapon, char dir);

        /* Setea la imagen del worm con el arma actual apuntando
         * con el angulo especifico */
        void setWeaponImage(char dir);

        /* Realiza la animacion del disparo del arma */
        void weaponShootAnimation(const std::string &weapon, char dir);

        /* Actualiza el angulo, cambiando la imagen del arma
         * por la correspondiente */
        void changeAngle(int angle, char dir);

        /* Actualiza el puntero de la imagen del worm */
        void updateWormImage(Gtk::Image* worm_image);
};


#endif //WORMS_WEAPONANIMATION_H
```

```cpp
#include <gtkmm/application.h>
#include <gtkmm/window.h>
#include "ServerMenu.h"
#include "Path.h"

int main(int argc, char* argv[]){
    auto app = Gtk::Application::create(argc, argv);
    Gtk::Window window;
    window.maximize();

    window.set_title(CLIENT_WINDOW_NAME);

    window.set_icon_from_file(ICON_PATH);

    ServerMenu server_menu(window);

    app->run(window);

    return 0;
}
```

Jun 01, 18 13:12 **ButtonBuilder.cpp** Page 1/1

```
#include "ButtonBuilder.h"
#include <string>
#include <gtkmm/label.h>
#include <gdkmm/rgba.h>

void ButtonBuilder::buildButton(Gtk::Button* button) {
    std::string text = button->get_label();
    Gtk::Label* label = (Gtk::Label*)button->get_child();
    label->set_markup("<b>" + text + "</b>");
    label->override_color(Gdk::RGBA("black"));
}
```

Jun 03, 18 12:56 **ButtonBuilder.h** Page 1/1

```
#ifndef WORMS_BUTTONBUILDER_H
#define WORMS_BUTTONBUILDER_H

#include <gtkmm/button.h>

class ButtonBuilder {
    public:
        /* Modifica la visualizaciÃ³n del label del boton */
        static void buildButton(Gtk::Button* button);
};


#endif //WORMS_BUTTONBUILDER_H
```

```cpp
#include "CreateGameMenu.h"
#include <gtkmm/builder.h>
#include <glibmm/main.h>
#include "Path.h"
#include "GamePlayers.h"
#include "ButtonBuilder.h"

CreateGameMenu::CreateGameMenu(Gtk::Window& window, MenuView& first_menu, ClientProtocol& protocol, std::string&& name, int quantity):
    SelectableListMenu(window, first_menu, protocol, std::move(name)){

    Glib::RefPtr<Gtk::Builder> builder = Gtk::Builder::create_from_file(GLADE_PATH + "client_CreateGameMenu.glade");

    builder->get_widget("error", this->error);
    builder->get_widget("game_name", this->game_name);
    builder->get_widget("players_number", this->players_number);
    builder->get_widget("games", this->games);
    builder->get_widget("quit_game", this->quit_game);

    this->configure(quantity);

    ButtonBuilder::buildButton(quit_game);

    builder->get_widget("create_game_menu", this->menu);

    this->addMenu();
    this->quit_game->signal_clicked().connect(sigc::mem_fun(*this, &CreateGameMenu::quitButtonPressed));
}

CreateGameMenu::~CreateGameMenu(){}

void CreateGameMenu::selectButtonPressed(Glib::ustring map_name){
    std::string name(this->game_name->get_text());
    if (name.empty()){
        this->error->set_label("Debe ingresar el nombre de la partida");
        return;
    }

    size_t players = this->players_number->get_value_as_int();
    if (players < min_players || players > max_players){
        std::string message("El numero de jugadores debe estar entre ");
        message += std::to_string(min_players) + std::string(" y ") + std::to_string(max_players);
        this->error->set_label(message);
        return;
    }

    try{
        this->protocol.sendString(map_name);
        this->protocol.sendString(name);
        this->protocol.sendLength(players);
        bool result = this->protocol.receiveChar();
        if (!result){
            this->showErrorAndRestart("Ocurrio un error al crear la partida");
        } else {
            this->waitToPlayers();
        }
    } catch (const SocketException& e){
        this->showFatalError();
    }
}
```

```cpp
#ifndef __CREATEGAMEMENU__
#define __CREATEGAMEMENU__

#include <gtkmm/button.h>
#include <gtkmm/entry.h>
#include <gtkmm/spinbutton.h>
#include "SelectableListMenu.h"

/* Clase que se encarga de los pasos necesarios para que el
 * jugador cree una partida */
class CreateGameMenu: public SelectableListMenu{
    private:
        Gtk::Entry* game_name;
        Gtk::SpinButton* players_number;
        Gtk::Button* quit_game;

        /* Handler del boton de seleccion */
        void selectButtonPressed(Glib::ustring map_name) override;

    public:
        /* Constructor */
        CreateGameMenu(Gtk::Window& window, MenuView& first_menu, ClientProtocol& protocol, std::string&& name, int quantity);

        /* Destructor */
        ~CreateGameMenu();
};

#endif
```

```cpp
#include "GameMenu.h"
#include <gtkmm/builder.h>
#include "Path.h"
#include "CreateGameMenu.h"
#include "JoinGameMenu.h"
#include "ButtonBuilder.h"

GameMenu::GameMenu(Gtk::Window& window, ClientProtocol& protocol):
    MenuView(window, *this, protocol){
    Glib::RefPtr<Gtk::Builder> builder = Gtk::Builder::create_from_file(GLADE_PATH + "client_GameMenu.glade");

    builder->get_widget("error", this->error);
    builder->get_widget("player_name", this->player_name);

    builder->get_widget("game_menu", this->menu);

    this->addMenu();

    Gtk::Button *create_game, *join_game, *quit_game;

    builder->get_widget("create_game", create_game);
    builder->get_widget("join_game", join_game);
    builder->get_widget("quit_game", quit_game);

    ButtonBuilder::buildButton(create_game);
    ButtonBuilder::buildButton(join_game);
    ButtonBuilder::buildButton(quit_game);

    create_game->signal_clicked().connect(sigc::mem_fun(*this, &GameMenu::createButtonPressed));
    join_game->signal_clicked().connect(sigc::mem_fun(*this, &GameMenu::joinButtonPressed));
    quit_game->signal_clicked().connect(sigc::mem_fun(*this, &GameMenu::quitButtonPressed));
}

GameMenu::~GameMenu(){}

void GameMenu::createButtonPressed(){
    if (this->selectAction(CREATE_GAME_ACTION)){
        std::string name(this->player_name->get_text());
        int quantity = this->protocol.receiveLength();
        if (quantity == 0){
            this->showErrorAndRestart("No hay mapas para crear una partida");
        } else {
            this->next_menu = std::unique_ptr<MenuView>(new CreateGameMenu(this->window, *this, this->protocol, std::move(name), quantity));
        }
    }
}

void GameMenu::joinButtonPressed(){
    if (this->selectAction(JOIN_GAME_ACTION)){
        std::string name(this->player_name->get_text());
        int quantity = this->protocol.receiveLength();
        if (quantity == 0){
            this->showErrorAndRestart("No hay partidas disponibles");
        } else {
            this->next_menu = std::unique_ptr<MenuView>(new JoinGameMenu(this->window, *this, this->protocol, std::move(name), quantity));
        }
    }
}

bool GameMenu::selectAction(char action){
```

```cpp
    std::string name(this->player_name->get_text());
    if (name.empty()){
        this->error->set_label("Debe ingresar su nombre");
        return false;
    }
    try{
        this->protocol.sendChar(action);
        this->protocol.sendString(name);
        this->window.remove();
        return true;
    } catch (const SocketException& e){
        this->showFatalError();
        return false;
    }
}
```

Jun 02, 18 12:18 **GameMenuField.cpp** Page 1/1

```
#include "GameMenuField.h"
#include <gdkmm/rgba.h>
#include "Path.h"
#include "ButtonBuilder.h"

GameMenuField::GameMenuField(const std::string& title): container(true, 20){
    size_t extension = title.rfind(YAML_EXTENSION);
    this->title.set_markup(title.substr(0, extension));
    this->title.override_color(Gdk::RGBA("black"));
    this->title.override_background_color(Gdk::RGBA("white"));
    this->container.pack_start(this->title);
    this->container.pack_end(this->button);

    this->button.set_label("Seleccionar");
    ButtonBuilder::buildButton(&this->button);
}

GameMenuField::~GameMenuField(){}

GameMenuField::GameMenuField(GameMenuField&& other): title(std::move(other.title
)),
    button(std::move(other.button)), container(std::move(other.container)){}

Gtk::Container& GameMenuField::getContainer(){
    return this->container;
}

Gtk::Button& GameMenuField::getButton(){
    return this->button;
}
```

May 28, 18 18:21 **GameMenuField.h** Page 1/1

```
#ifndef __GAMEMENUFIELD_H__
#define __GAMEMENUFIELD_H__

#include <gtkmm/hvbox.h>
#include <gtkmm/label.h>
#include <gtkmm/button.h>
#include <string>

class GameMenuField{
    private:
        Gtk::Label title;
        Gtk::Button button;
        Gtk::HBox container;

    public:
        /* Constructor */
        GameMenuField(const std::string& title);

        /* Destructor */
        ~GameMenuField();

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


        /* Devuelve el contenedor del menu */
        Gtk::Container& getContainer();

        /* Devuelve el boton del menu */
        Gtk::Button& getButton();
};


#endif
```

```cpp
#ifndef __GAMEMENU__
#define __GAMEMENU__

#include <gtkmm/button.h>
#include <gtkmm/entry.h>
#include <gtkmm/window.h>
#include <string>
#include <memory>
#include "ClientProtocol.h"
#include "MenuView.h"

/* Clase que se encarga de controlar el menu del juego */
class GameMenu: public MenuView{
    private:
        Gtk::Entry* player_name;

        /* Crea el boton de creacion de partida */
        void createButtonPressed();

        /* Crea el boton de unirse a partida */
        void joinButtonPressed();

        /* Envia la accion implementada */
        bool selectAction(char action);

    public:
        /* Constructor */
        GameMenu(Gtk::Window& window, ClientProtocol& protocol);

        /* Destructor */
        ~GameMenu();
};

#endif
```

```cpp
#include "JoinGameMenu.h"
#include <gtkmm/builder.h>
#include <glibmm/main.h>
#include "Path.h"
#include "WaitingLabel.h"
#include "ButtonBuilder.h"

JoinGameMenu::JoinGameMenu(Gtk::Window& window, MenuView& first_menu, ClientProtocol& protocol, std::string&& name, int quantity):
    SelectableListMenu(window, first_menu, protocol, std::move(name)){

    Glib::RefPtr<Gtk::Builder> builder = Gtk::Builder::create_from_file(GLADE_PATH + "client_JoinGameMenu.glade");

    builder->get_widget("error", this->error);
    builder->get_widget("game", this->games);
    builder->get_widget("quit_game", this->quit_game);

    this->configure(quantity);

    ButtonBuilder::buildButton(quit_game);

    builder->get_widget("join_game_menu", this->menu);

    this->addMenu();

    quit_game->signal_clicked().connect(sigc::mem_fun(*this, &JoinGameMenu::quitButtonPressed));

}

JoinGameMenu::~JoinGameMenu(){}


void JoinGameMenu::selectButtonPressed(Glib::ustring game_name){
    try{
        this->protocol.sendString(game_name);
        bool result = this->protocol.receiveChar();
        if (!result){
            this->showErrorAndRestart("Ocurrio un error al unirse a la partida");
        } else {
            this->waitToPlayers();
        }
    } catch (const SocketException& e){
        this->showFatalError();
    }
}
```

Jun 01, 18 13:12 **JoinGameMenu.h** Page 1/1

```
#ifndef __JOINGAMEMENU__
#define __JOINGAMEMENU__

#include "SelectableListMenu.h"


/* Clase que se encarga de los pasos necesarios para que el
 * jugador se una a una partida */
class JoinGameMenu: public SelectableListMenu{
    private:
        Gtk::Button* quit_game;

        /* Handler del boton de unirse a partida */
        void selectButtonPressed(Glib::ustring game_name) override;

    public:
        /* Constructor */
        JoinGameMenu(Gtk::Window& window, MenuView& first_menu, ClientProtocol& protocol, std::string&& name, int quantity);

        /* Destructor */
        ~JoinGameMenu();
};

#endif
```

Jun 03, 18 12:56 **MenuView.cpp** Page 1/1

```
#include "MenuView.h"
#include "ServerFatalError.h"

MenuView::MenuView(Gtk::Window& window, MenuView& main_menu, ClientProtocol& protocol):
    window(window), protocol(protocol), main_menu(main_menu) {

    Glib::RefPtr<Gdk::Pixbuf> aux = Gdk::Pixbuf::create_from_file(BACKGROUND_MENU_IMAGE);
    this->background.set(aux);
    this->menu_container.add_overlay(this->background);
}

MenuView::~MenuView(){
    delete this->menu;
}

void MenuView::showFatalError(){
    ServerFatalError error(this->window);
}

void MenuView::showErrorAndRestart(const std::string& error){
    this->window.remove();
    this->main_menu.showError(error);
    this->window.add(this->main_menu.menu_container);
}

void MenuView::showError(const std::string& error){
    this->error->set_label(error);
}

void MenuView::quitButtonPressed() {
    this->window.close();
}

void MenuView::addMenu() {
    this->menu_container.add_overlay(*this->menu);
    this->window.add(this->menu_container);
    this->window.show_all();
}
```

```
#ifndef __MENUVIEW_H__
#define __MENUVIEW_H__

#include <gtkmm/hvbox.h>
#include <gtkmm/label.h>
#include <gtkmm/window.h>
#include <gtkmm/overlay.h>
#include <gtkmm/image.h>
#include <memory>
#include "ClientProtocol.h"

class MenuView{
    private:
        /* Muestra un mensaje de error */
        void showError(const std::string& error);

    protected:
        Gtk::Window& window;
        ClientProtocol& protocol;
        Gtk::Label* error;
        std::unique_ptr<MenuView> next_menu;
        MenuView& main_menu;
        Gtk::Container* menu;

        Gtk::Overlay menu_container;
        Gtk::Image background;

        /* Muestra un mensaje de error y cierra la aplicacion*/
        void showFatalError();

        /* Muestra un mensaje de error y reinicia */
        void showErrorAndRestart(const std::string& error);

        /* Agrega el menu al world y el world al window */
        void addMenu();

        /* Handler del boton de salir */
        void quitButtonPressed();

    public:
        /* Constructor */
        MenuView(Gtk::Window& window, MenuView& main_menu, ClientProtocol& proto
col);

        /* Destructor */
        virtual ~MenuView();
};

#endif
```

```
#include "SelectableListMenu.h"

SelectableListMenu::SelectableListMenu(Gtk::Window& window, MenuView& first_menu
, ClientProtocol& protocol, std::string&& name):
    MenuView(window, first_menu, protocol), player_name(std::move(name)){}

SelectableListMenu::~SelectableListMenu(){}

void SelectableListMenu::configure(int quantity){
    try{
        for (int i = 0; i < quantity; i++){
            std::string field = this->protocol.receiveString();
            this->addField(field);
        }
    }catch (const SocketException& e){
        this->showFatalError();
    }

    for (auto it = this->fields.begin(); it != this->fields.end(); ++it){
        this->games->pack_start(it->getContainer());
    }
    this->games->show();
}

void SelectableListMenu::addField(const std::string& field_name){
    GameMenuField field(field_name);
    this->fields.push_back(std::move(field));
    this->fields.back().getButton().signal_clicked().connect(sigc::bind<Glib::us
tring>(sigc::mem_fun(*this,

                   &SelectableListMenu::selectButtonPressed), field_name));
}

bool SelectableListMenu::createPlayer(){
    try{
        this->player = std::unique_ptr<Player>(new Player(std::move(this->protoc
ol), this->player_name, this->window));
    } catch (const std::exception& e){
        this->showFatalError();
    }
    return false;
}

void SelectableListMenu::waitToPlayers(){
    this->window.remove();
    this->window.add(this->waiting_label.getWidget());
    this->window.show_all();
    sigc::slot<bool> my_slot = sigc::mem_fun(*this, &SelectableListMenu::createP
layer);
    Glib::signal_idle().connect(my_slot);
}
```

```
#ifndef __SELECTABLELISTMENU_H__
#define __SELECTABLELISTMENU_H__

#include <gtkmm/hvbox.h>
#include <gtkmm/label.h>
#include <gtkmm/window.h>
#include <memory>
#include <string>
#include <vector>
#include "ClientProtocol.h"
#include "MenuView.h"
#include "WaitingLabel.h"
#include "Player.h"
#include "GameMenuField.h"

class SelectableListMenu: public MenuView{
    protected:
        Gtk::Box* games;
        std::string player_name;
        WaitingLabel waiting_label;
        std::vector<GameMenuField> fields;
        std::unique_ptr<Player> player;

        /* Realiza la configuracion del juego */
        void configure(int quantity);

        /* Agrega un campo a la lista */
        void addField(const std::string& field_name);

        /* Crea un nuevo jugador */
        bool createPlayer();

        /* Handler del boton de seleccion */
        virtual void selectButtonPressed(Glib::ustring field_name) = 0;

        /* Muestra el mensaje esperando jugadores */
        void waitToPlayers();

    public:
        /* Constructor */
        SelectableListMenu(Gtk::Window& window, MenuView& first_menu, ClientProt
ocol& protocol, std::string&& name);

        /* Destructor */
        ~SelectableListMenu();
};

#endif
```

```
#include "ServerFatalError.h"
#include <gtkmm/messagedialog.h>

ServerFatalError::ServerFatalError(Gtk::Window& window){
    Gtk::MessageDialog dialog(window, "Ocurrio un error con la conexion del servidor", false,
Gtk::MESSAGE_ERROR, Gtk::BUTTONS_CLOSE, true);
    dialog.run();
    window.close();
}

ServerFatalError::~ServerFatalError(){}
```

```cpp
#ifndef __SERVERFATALERROR_H__
#define __SERVERFATALERROR_H__

#include <gtkmm/window.h>

class ServerFatalError{
    public:
        ServerFatalError(Gtk::Window& window);

        ~ServerFatalError();
};

#endif
```

```cpp
#include "ServerMenu.h"
#include <gtkmm/builder.h>
#include "Path.h"
#include "ButtonBuilder.h"

ServerMenu::ServerMenu(Gtk::Window& window): window(window) {
    Glib::RefPtr<Gtk::Builder> builder = Gtk::Builder::create_from_file(GLADE_PATH + "client_ServerMenu.glade");

    builder->get_widget("error", this->error);
    builder->get_widget("host", this->host);
    builder->get_widget("service", this->service);
    builder->get_widget("connect", this->connect);
    builder->get_widget("quit_game", this->quit);

    ButtonBuilder::buildButton(this->quit);
    ButtonBuilder::buildButton(this->connect);

    builder->get_widget("server_menu", this->menu);

    builder->get_widget("background", this->background);
    Glib::RefPtr<Gdk::Pixbuf> aux = Gdk::Pixbuf::create_from_file(BACKGROUND_MENU_IMAGE);
    this->background->set(aux);

    this->window.add(*this->menu);
    this->window.show_all();

    this->connect->signal_clicked().connect(sigc::mem_fun(*this, &ServerMenu::connectButtonPressed));
    this->quit->signal_clicked().connect(sigc::mem_fun(*this, &ServerMenu::quitButtonPressed));
}

ServerMenu::~ServerMenu(){
    delete this->menu;
}

void ServerMenu::connectButtonPressed(){
    std::string host(this->host->get_text());
    if (host.empty()){
        this->error->set_label("Debe ingresar un host");
        return;
    }

    std::string service(this->service->get_text());
    if (service.empty()){
        this->error->set_label("Debe ingresar un servicio");
        return;
    }

    this->connectToServer(host, service);
}

void ServerMenu::quitButtonPressed() {
    this->window.close();
}

void ServerMenu::connectToServer(const std::string &host, const std::string &service){
    try{
        Socket socket(Socket::Client(host.c_str(), service.c_str()));
        this->protocol.reset(new ClientProtocol(std::move(socket), this->window));
        this->window.remove();
        this->next_menu = std::unique_ptr<MenuView>(new GameMenu(this->window, *
    this->protocol));
    } catch (const SocketException& e){
        this->error->set_label("No pudo conectarse al servidor");
    }
}
```

```cpp
#ifndef __SERVERMENU__
#define __SERVERMENU__

#include <gtkmm/application.h>
#include <gtkmm/hvbox.h>
#include <gtkmm/button.h>
#include <gtkmm/entry.h>
#include <gtkmm/label.h>
#include <gtkmm/window.h>
#include <gtkmm/overlay.h>
#include <gtkmm/image.h>
#include <string>
#include <memory>
#include "ClientProtocol.h"
#include "GameMenu.h"
#include "MenuView.h"

/* Menu de conexion con el servidor */
class ServerMenu{
    private:
        Gtk::Label* error;
        Gtk::Entry* host;
        Gtk::Entry* service;
        Gtk::Button* connect;
        Gtk::Button* quit;
        Gtk::Window& window;
        Gtk::Container* menu;
        std::unique_ptr<MenuView> next_menu;
        std::unique_ptr<ClientProtocol> protocol;
        Gtk::Image* background;

        /* Handler del boton de conexion */
        void connectButtonPressed();

        /* Handler del boton de salir */
        void quitButtonPressed();

        /* Intenta realizar una conexion con el servidor */
        void connectToServer(const std::string &host, const std::string &service
);

    public:
        /* Constructor */
        ServerMenu(Gtk::Window& window);

        /* Destructor */
        ~ServerMenu();
};

#endif
```

May 27, 18 22:46 **WaitingLabel.cpp** Page 1/1

```cpp
#include "WaitingLabel.h"

const std::string begining("<span size='20000'>");
const std::string ending("</span>");

WaitingLabel::WaitingLabel(){
    this->label.set_use_markup(true);
    this->label.set_markup(begining + "Esperando jugadores..." + ending);
    this->label.show();
}

WaitingLabel::~WaitingLabel(){}

Gtk::Widget& WaitingLabel::getWidget(){
    return this->label;
}
```

Jun 03, 18 12:56 **WaitingLabel.h** Page 1/1

```cpp
#ifndef __WAITINGLABEL_H__
#define __WAITINGLABEL_H__

#include <gtkmm/label.h>

/* Label de que indica la espera a otros jugadores */
class WaitingLabel{
    private:
        Gtk::Label label;

    public:
        /* Constructor */
        WaitingLabel();

        /* Destructor */
        ~WaitingLabel();

        /* Devuelve el contenedor del mensaje */
        Gtk::Widget& getWidget();
};


#endif
```

```cpp
#ifndef WORMS_MUSICPATH_H
#define WORMS_MUSICPATH_H

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

const std::string BACKGROUND_MUSIC = SOUNDS_PATH + "BackgroundMusic.mp3";
const std::string START_TURN_SOUND = SOUNDS_PATH + "Misc/StartRound.wav";
const std::string TICK_SOUND = SOUNDS_PATH + "Misc/TimerTick.wav";
const std::string RUN_AWAY_SOUND = SOUNDS_PATH + "Worms/RunAway.wav";
const std::string DEATH_SOUND = SOUNDS_PATH + "Worms/Death.wav";
const std::string DAMAGE_RECEIVE_SOUND = SOUNDS_PATH + "Worms/DamageReceive.wav";
const std::string EXPLOSION_SOUND = SOUNDS_PATH + "Weapons/Explosion.wav";
const std::string TELEPORT_SOUND = SOUNDS_PATH + "Weapons/Teleportation.wav";
const std::string BAT_SOUND = SOUNDS_PATH + "Weapons/BaseballSound.wav";
const std::string HOLY_GRENADE_SOUND = SOUNDS_PATH + "Weapons/HolyGrenade.wav";
const std::string AIR_ATTACK_SOUND = SOUNDS_PATH + "Weapons/AirAttack.wav";
const std::string SHOOT_SOUND  = SOUNDS_PATH + "Weapons/ShootWeapon.wav";
const std::string ROLLBACK_SOUND = SOUNDS_PATH + "Misc/RollBack.wav";
const std::string JUMP_SOUND = SOUNDS_PATH + "Misc/Jump.wav";
const std::string SELECT_WEAPON_SOUND = SOUNDS_PATH + "Misc/SelectWeapon.wav";
const std::string NO_AMMO_SOUND = SOUNDS_PATH + "Misc/NoAmmo.wav";
const std::string VICTORY_SOUND = SOUNDS_PATH + "Worms/Victory.WAV";

#endif //WORMS_MUSICPATH_H
```

```cpp
#include "MusicPlayer.h"
#include "MusicPlayerException.h"
#include "WeaponNames.h"
#include "Protocol.h"
#include "MusicPath.h"

MusicPlayer::MusicPlayer() {
    this->music = NULL;
    // Initialize SDL.
    if (SDL_Init(SDL_INIT_AUDIO) < 0) {
        throw MusicPlayerException("Error al inicializar SDL");
    }

    //Initialize SDL_mixer
    if (Mix_OpenAudio( 22050, MIX_DEFAULT_FORMAT, 2, 4096) == -1) {
        throw MusicPlayerException("Error al inicializar SDL mixer");
    }

    // Load background music
    this->music = Mix_LoadMUS(BACKGROUND_MUSIC.c_str());
    if (this->music == NULL) {
    }
}

MusicPlayer::~MusicPlayer() {
    Mix_HaltChannel(-1);
    this->stop();
    if (this->music != NULL) {
        Mix_FreeMusic(this->music);
    }
    std::map<int, Mix_Chunk*>::iterator iter;
    for (iter = this->effects.begin(); iter != this->effects.end(); iter++) {
        Mix_FreeChunk(iter->second);
    }
    // quit SDL_mixer
    Mix_CloseAudio();
    Mix_Quit();
    SDL_Quit();
}

void MusicPlayer::check(int channel) {
    if (this->effects.find(channel) != this->effects.end()) {
        // elimino el audio anterior de este canal
        Mix_FreeChunk(this->effects.at(channel));
        this->effects.erase(channel);
    }
    std::map<int, Mix_Chunk*>::iterator iter = this->effects.begin();
    while (iter != this->effects.end()) {
        if (!Mix_Playing(iter->first)) {
            Mix_FreeChunk(iter->second);
            iter = this->effects.erase(iter);
        } else {
            iter++;
        }
    }
}

void MusicPlayer::addEffect(const std::string& audio) {
    int channel;
    Mix_Chunk* effect = NULL;
    effect = Mix_LoadWAV(audio.c_str());
    if (effect == NULL) {
        return;
    }
    if ((channel = Mix_PlayChannel(-1, effect, 0)) == -1) {
        Mix_FreeChunk(effect);
        return;
    }
    this->check(channel);
    this->effects.insert(std::make_pair(channel, effect));
}

void MusicPlayer::playMusic() {
    Mix_PlayMusic(this->music, -1);
    Mix_VolumeMusic(MIX_MAX_VOLUME / 4);
}

void MusicPlayer::playStartTurnSound() {
    this->addEffect(START_TURN_SOUND);
}

void MusicPlayer::playTickSound() {
    this->addEffect(TICK_SOUND);
}

void MusicPlayer::playDeathSound() {
    this->addEffect(DEATH_SOUND);
}

void MusicPlayer::playDamageReceiveSound() {
    this->addEffect(DAMAGE_RECEIVE_SOUND);
}

void MusicPlayer::playExplosionSound(const std::string& weapon) {
    if (weapon == HOLY_GRENADE_NAME) {
        this->addEffect(HOLY_GRENADE_SOUND);
    } else {
        this->addEffect(EXPLOSION_SOUND);
    }
}

void MusicPlayer::playVictory() {
    this->addEffect(VICTORY_SOUND);
}

void MusicPlayer::playNoAmmo() {
    this->addEffect(NO_AMMO_SOUND);
}

void MusicPlayer::stop() {
    Mix_HaltMusic();
}

void MusicPlayer::playWeaponShotSound(const std::string& weapon){
    if (weapon == TELEPORT_NAME) {
        this->addEffect(TELEPORT_SOUND);
    } else if (weapon == BAT_NAME) {
        this->addEffect(BAT_SOUND);
    } else if (weapon == DYNAMITE_NAME) {
        this->addEffect(RUN_AWAY_SOUND);
    } else if (weapon == AIR_ATTACK_NAME) {
        this->addEffect(AIR_ATTACK_SOUND);
    } else {
        this->addEffect(SHOOT_SOUND);
    }
}

void MusicPlayer::playJumpSound(char action) {
    if (action == ROLLBACK) {
        this->addEffect(ROLLBACK_SOUND);
    } else if (action == JUMP){
        this->addEffect(JUMP_SOUND);
```

```cpp
    }
}

void MusicPlayer::playSelectWeaponSound() {
    this->addEffect(SELECT_WEAPON_SOUND);
}
```

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

MusicPlayerException::MusicPlayerException(std::string msg): msg(msg){
    this->msg.insert(0, "Error en Music Player: ");
}

MusicPlayerException::~MusicPlayerException(){}

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

```cpp
#ifndef __MUSICPLAYEREXCEPTION_H__
#define __MUSICPLAYEREXCEPTION_H__

#include <exception>
#include <string>

class MusicPlayerException: public std::exception{
    private:
        std::string msg;

    public:
        //Crea la excepcion
        explicit MusicPlayerException(std::string msg);

        //Destruye la excepcion
        virtual ~MusicPlayerException();

        //Devuelve el mensaje de error
        virtual const char* what() const noexcept;
};

#endif
```

```
#ifndef __MUSICPLAYER_H__
#define __MUSICPLAYER_H__

#include <SDL2/SDL.h>
#include <SDL2/SDL_mixer.h>
#include <map>
#include <string>

/* Clase que se enecarga de reproducir musica y efectos
 * de sonido */
class MusicPlayer {
    private:
        Mix_Music* music; // Musica de fondo
        std::map<int, Mix_Chunk*> effects;

        /* Verifica si algunos efectos de la lista finalizaon y los
         * libera. AdemÃ¡s libera el efecto que se encuentre guardado
         * en la lista con clave channel */
        void check(int channel);

        /* Agrega un nuevo efecto a la lista y lo reproduce */
        void addEffect(const std::string& audio);

    public:
        /* Constructor */
        MusicPlayer();

        /* Destructor */
        ~MusicPlayer();

        /* Reproduce la musica de fondo */
        void playMusic();

        /* Reproduce el sonido de inicio de turno */
        void playStartTurnSound();

        /* Reproduce el sonido de falta de tiempo */
        void playTickSound();

        /* Reproduce el sonido de muerte de un worm */
        void playDeathSound();

        /* Reproduce el sonido de daÃ±o recibido */
        void playDamageReceiveSound();

        /* Reproduce el sonido de la explosion */
        void playExplosionSound(const std::string& weapon);

        /* Reproduce el sonido de arma disparada */
        void playWeaponShotSound(const std::string& weapon);

        /* Reproduce el sonido de salto o rollback */
        void playJumpSound(char action);

        /* Reproduce el sonido de arma seleccionada */
        void playSelectWeaponSound();

        /* Reproduce el sonido de victoria */
        void playVictory();

        /* Reproduce el sonido de arma descargada */
        void playNoAmmo();

        /* Detiene la reproduccion de la musica de fondo */
        void stop();
};
```

```


#endif
```

```cpp
#include "Player.h"
#include "WeaponNames.h"

Player::Player(ClientProtocol protocol, const std::string& name, Gtk::Window& window):
    protocol(std::move(protocol)), name(name),
    screen(window, *this, this->weapons),
    turn(*this, this->screen.getTurnLabel()),
    view_list(this->screen.getWorld(), *this, this->screen.getPlayersView(), musicPlayer),
    data_receiver(*this),
    handlers(*this, this->view_list, this->weapons, this->screen.getWorld()){

    this->musicPlayer.playMusic();
    this->data_receiver.start();
}

Player::~Player() {
    this->data_receiver.stop();
    this->data_receiver.join();
}

void Player::startTurn(int worm_id, int player_id, float wind){
    this->view_list.setCurrentWorm(worm_id);
    this->screen.getWindView().update(wind);
    const std::string& current_player = this->screen.getPlayersView().getPlayer(player_id);
    if (current_player == this->name){
        //Es mi turno
        this->musicPlayer.playStartTurnSound();
        this->handlers.enableAll();
        this->changeWeapon(this->weapons.getCurrentWeapon().getName());
        this->screen.getTurnLabel().beginTurn();
        this->turn.start();
    } else {
        this->screen.getTurnLabel().beginTurn(current_player);
    }
}

void Player::endTurn() {
    this->screen.getTurnLabel().endTurn();
    this->handlers.disableAll();
    this->view_list.removeScopeVisibility();
    this->protocol.sendEndTurn();
}

void Player::endGame(const std::string& winner){
    this->data_receiver.stop();
    this->handlers.disableAll();
    this->screen.getTurnLabel().setWinner(winner, this->name == winner);
    this->view_list.setVictory();
}

void Player::endTurnEarly(){
    this->turn.stop();
}

void Player::shootWeapon() {
    this->turn.reduceTime();
    this->weapons.getCurrentWeapon().shoot();
}

void Player::changeWeapon(std::string weapon) {
    this->musicPlayer.playSelectWeaponSound();
    this->weapons.changeWeapon(weapon);
    if (this->handlers.isEnabled()){
```

```cpp
        this->protocol.sendChangeWeapon(weapon);
    }
}

void Player::shoot(Position position) {
    this->shootWeapon();
    this->protocol.sendWeaponSelfDirectedShoot(position);
    this->screen.getWeaponsView().updateAmmo(this->weapons.getCurrentWeapon());
}

void Player::playTickTime() {
    this->musicPlayer.playTickSound();
}

void Player::shoot(int angle, int power, int time) {
    this->shootWeapon();
    if (!this->weapons.getCurrentWeapon().isTimed()) {
        time = -1;
    }
    if (!this->weapons.getCurrentWeapon().hasScope()) {
        angle = MAX_WEAPON_ANGLE * 8;
    }
    this->protocol.sendWeaponShoot(angle, power, time);
    this->view_list.removeScopeVisibility();
    this->screen.getWeaponsView().updateAmmo(this->weapons.getCurrentWeapon());
}

ViewsList& Player::getViewsList() {
    return this->view_list;
}

ScreenView& Player::getScreen(){
    return this->screen;
}

WeaponList& Player::getWeapons(){
    return this->weapons;
}

ClientProtocol& Player::getProtocol(){
    return this->protocol;
}

MusicPlayer& Player::getMusicPlayer() {
    return this->musicPlayer;
}
```

```
#ifndef __CLIENTPLAYER_H__
#define __CLIENTPLAYER_H__

#include <memory>
#include <gtkmm/window.h>
#include "ClientProtocol.h"
#include "Turn.h"
#include "Weapon.h"
#include "WeaponList.h"
#include "ScreenView.h"
#include "ViewsList.h"
#include "Position.h"
#include "DataReceiver.h"
#include "Handlers.h"
#include "MusicPlayer.h"

class Player {
    private:
        ClientProtocol protocol;
        std::string name;
        WeaponList weapons;
        ScreenView screen;
        Turn turn;
        ViewsList view_list;
        DataReceiver data_receiver;
        Handlers handlers;
        MusicPlayer musicPlayer;

        /* Reduce el tiempo del turno y actualiza la municion */
        void shootWeapon();

    public:
        /* Constructor */
        Player(ClientProtocol protocol, const std::string& name, Gtk::Window& wi
ndow);

        /* Destructor */
        ~Player();


        /* Comienza el turno. Si es el turno del jugador entonces,
           habilita los handlers, sino muestra los movimientos realizados
           por el otro jugador */
        void startTurn(int worm_id, int player_id, float wind);

        /* Finaliza el turno del jugador actual */
        void endTurn();

        /* Finaliza el juego */
        void endGame(const std::string& winner);

        /* El jugador debe terminar su turno antes */
        void endTurnEarly();

        /* Cambia el arma actual por la espeificada */
        void changeWeapon(std::string weapon);

        /* Realiza el disparo del arma con el angulo, potencia
           y tiempo pasados */
        void shoot(int angle, int power, int time);

        /* Realiza el disparo del arma en la posicion pasada */
        void shoot(Position position);

        /* Reproduce el sonido de falta de tiempo */
        void playTickTime();
```

```

        /* Devuelve la lista de los elementos presentes en la vista */
        ViewsList& getViewsList();

        /* Devuelve la vista */
        ScreenView& getScreen();

        /* Devuelve la lista de armas */
        WeaponList& getWeapons();

        /* Devuelve el protocolo */
        ClientProtocol& getProtocol();

        /* Devuelve el music player */
        MusicPlayer& getMusicPlayer();
};

#endif
```

```cpp
#include "Scope.h"
#include "Path.h"
#include "WeaponNames.h"

Scope::Scope(WorldView& world): world(world){
    this->scope.set(SCOPE_IMAGE);
    this->angle = DEFAULT_ANGLE;
    this->world.addElement(this->scope, Position(0,0), 0, 0);
}

Scope::~Scope(){}

void Scope::update(int angle, WormView& worm){
    this->angle = angle;
    char dir = worm.getDir();
    if (dir == DIR_LEFT)
        angle = 180 - angle;
    this->world.moveScope(this->scope, worm.getWidget(), angle);
    this->scope.show();
    worm.updateScope(this->angle);
}

void Scope::update(WormView& worm){
    this->update(this->angle, worm);
}


void Scope::hide(){
    if (this->scope.is_visible()){
        this->scope.hide();
    }
}
```

```cpp
#ifndef __SCOPE_H__
#define __SCOPE_H__

#include <gtkmm/image.h>
#include "WorldView.h"
#include "WormView.h"

class Scope{
    private:
        Gtk::Image scope;
        WorldView& world;
        int angle;

    public:
        /* Constructor */
        Scope(WorldView& world);

        /* Destructor */
        ~Scope();

        /* Actualiza la posicion del scope */
        void update(int angle, WormView& worm);

        /* Actualiza la posicion del scope */
        void update(WormView& worm);

        /* Esconde el scope */
        void hide();

};

#endif
```

```cpp
#include "ViewsList.h"
#include <glibmm/main.h>
#include "ObjectSizes.h"
#include "WeaponNames.h"
#include "Player.h"

ViewsList::ViewsList(WorldView& world, Player& player, PlayersList& players_list, MusicPlayer& musicPlayer):
    world(world), player(player), players_list(players_list), scope(world), musicPlayer(musicPlayer) {

    this->current_worm_id = -1;
    this->weapon_focused = -1;
    this->worm_focused = -1;
}

ViewsList::~ViewsList(){}


void ViewsList::removeWorm(int id){
    auto it = this->worms.find(id);
    if (it != this->worms.end()) {
        this->players_list.reducePlayerLife(it->second.getPlayerId(), it->second.getLife());
        if (id == this->current_worm_id){
            this->player.endTurnEarly();
        }
        it->second.removeFromWorld();
        this->worms.erase(it);
        this->musicPlayer.playDeathSound();
        this->checkMovingWorms();
    }
}

void ViewsList::removeWeapon(int id){
    auto it = this->weapons.find(id);
    if (it != this->weapons.end()) {
        if(it->second.getName() != BAT_NAME) {
            this->musicPlayer.playExplosionSound(it->second.getName());
            ExplosionView explosion(std::move(it->second));
            this->animation.addAndStart(std::move(explosion));
        }
        this->weapons.erase(it);

        if (this->weapon_focused == id){
            this->weapon_focused = -2;
            this->checkMovingWorms();
        }
    }
}

void ViewsList::updateWormData(int id, int player_id, float pos_x, float pos_y, int life, char dir, bool colliding){
    auto it = this->worms.find(id);
    Position pos(pos_x / UNIT_TO_SEND, pos_y / UNIT_TO_SEND);
    if (it == this->worms.end()){
        //Worm no existe
        WormView worm(this->world, life, dir, pos, player_id);
        this->worms.insert(std::make_pair(id, std::move(worm)));
        this->players_list.addPlayerLife(player_id, life);
    } else {
        //Worm existe
        int current_life = it->second.getLife();
        if (current_life != life){
            this->players_list.reducePlayerLife(player_id, current_life - life);
            if (id == this->current_worm_id){
```

```cpp
                this->musicPlayer.playDamageReceiveSound();
                this->player.endTurnEarly();
            }
        }
        it->second.updateData(life, dir, pos, colliding, id == this->current_worm_id, this->weapon_focused != -1);
        this->checkMovingWorms();
    }
}

void ViewsList::updateWeaponData(int id, const std::string& weapon_name, float pos_x, float pos_y){
    auto it = this->weapons.find(id);
    Position pos(pos_x / UNIT_TO_SEND, pos_y / UNIT_TO_SEND);
    if (it == this->weapons.end()){
        //Weapon no existe
        BulletView weapon(this->world, weapon_name, pos);
        if (this->weapon_focused < 0){
            weapon.setFocus(true);
            this->weapon_focused = id;
            this->removeWormFocus();
        }
        this->weapons.insert(std::make_pair(id, std::move(weapon)));
    } else {
        //Weapon existe
        it->second.updateData(pos);
    }
}

void ViewsList::changeWeapon(const std::string& weapon_name) {
    auto it = this->worms.find(this->current_worm_id);
    it->second.changeWeapon(weapon_name);
    if (WeaponsFactory().createWeapon(weapon_name, 1)->hasScope()) {
        this->scope.update(it->second);
    }
}

void ViewsList::updateScope(int angle) {
    auto it = this->worms.find(this->current_worm_id);
    if (it == this->worms.end()) {
        return;
    }
    this->scope.update(angle, it->second);
}

void ViewsList::removeScopeVisibility() {
    this->scope.hide();
}

bool ViewsList::addGirderCallBack(size_t size, Position pos, int rotation){
    GirderView girder(this->world, size, pos, rotation);
    this->girders.push_back(std::move(girder));
    return false;
}

void ViewsList::addGirder(size_t size, float pos_x, float pos_y, int rotation){
    sigc::slot<bool> my_slot = sigc::bind(sigc::mem_fun(*this, &ViewsList::addGirderCallBack), size, Position(pos_x, pos_y), rotation);
    Glib::signal_idle().connect(my_slot);
}

void ViewsList::setCurrentWorm(int id){
    this->removeWormFocus();
    for (auto it = this->worms.begin(); it != this->worms.end(); ++it){
        it->second.resetFocus();
    }
    this->current_worm_id = id;
    this->worm_focused = id;
    this->weapon_focused = -1;
    WormView& worm = this->worms.at(id);
    this->world.setFocus(worm.getWidget());
    worm.setFocus(true);
}

void ViewsList::removeWormFocus(){
    auto it = this->worms.find(this->worm_focused);
    if (it != this->worms.end()){
        it->second.resetFocus();
    }
    this->worm_focused = -1;
}

void ViewsList::checkMovingWorms(){
    if (this->weapon_focused != -2){
        return;
    }

    auto it = this->worms.find(this->worm_focused);
    if (it == this->worms.end() || !it->second.isMoving()){
        this->removeWormFocus();
        for (auto it2 = this->worms.begin(); it2 != this->worms.end(); ++it2){
            if (it2->second.isMoving()){
                this->worm_focused = it2->first;
                it2->second.setFocus(true);
                this->world.setFocus(it2->second.getWidget());
                return;
            }
        }
    }

}

void ViewsList::setVictory() {
    if (this->worms.empty()){
        return;
    }
    for (auto iter = this->worms.begin(); iter != this->worms.end(); iter++) {
        this->musicPlayer.playVictory();
        iter->second.setVictory();
        this->world.setFocus(iter->second.getWidget());
    }
}

void ViewsList::shoot(const std::string& weapon) {
    this->worms.at(this->current_worm_id).weaponShoot(weapon);
}
```

```
#ifndef __VIEWSLIST_H__
#define __VIEWSLIST_H__

#include <unordered_map>
#include <vector>
#include <string>
#include "WorldView.h"
#include "WormView.h"
#include "BulletView.h"
#include "GirderView.h"
#include "PlayersList.h"
#include "ExplosionView.h"
#include "ExplosionViewList.h"
#include "MusicPlayer.h"
#include "Scope.h"

/* Clase que se encarga de almacenar los objetos visibles */
class ViewsList{
    private:
        WorldView& world;
        Player& player;
        PlayersList& players_list;
        std::unordered_map<int, WormView> worms;
        std::unordered_map<int, BulletView> weapons;
        std::vector<GirderView> girders;
        int current_worm_id;
        int weapon_focused;
        int worm_focused;
        ExplosionViewList animation;
        Scope scope;
        MusicPlayer& musicPlayer;

        /* Elimina el focus sobre el worm */
        void removeWormFocus();

        /* CallBacks */
        bool addGirderCallBack(size_t size, Position pos, int rotation);

    public:
        /* Constructor */
        ViewsList(WorldView& world, Player& player, PlayersList& players_list, M
usicPlayer& musicPlayer);

        /* Destructor */
        ~ViewsList();

        /* Elimina al worm de la vista actualizando la vida del player */
        void removeWorm(int id);

        /* Elimina la vista del arma y la reemplaza por la animacion de la explo
sion */
        void removeWeapon(int id);

        /* Actualiza la posicion y la vida del worm */
        void updateWormData(int id, int player_id, float pos_x, float pos_y, int
 life, char dir, bool colliding);

        /* Actualiza la posicion del arma */
        void updateWeaponData(int id, const std::string& weapon_name, float pos_
x, float pos_y);

        /* CallBack de changeWeapon */
        bool changeWeaponCallBack(const std::string &weapon_name);

        /* Actualiza la vista del worm con el arma nueva */
        void changeWeapon(const std::string &weapon_name);
```

```cpp

        /* Actualiza la posicion del scope */
        void updateScope(int angle);

        /* Esconde la vista del scope */
        void removeScopeVisibility();

        /* Agrega una viga a la vista en la posicion indicada y
         * con la rotacion indicada */
        void addGirder(size_t size, float pos_x, float pos_y, int rotation);

        /* Actualiza el worm actual y hace focus en este */
        void setCurrentWorm(int id);

        /* Actualiza la imagen de los worms ganadores por la animacion
         * de los worms festejando */
        void setVictory();

        /* Chequea si el gusano actual se esta moviendo, caso contario
        le da el focus a otro */
        void checkMovingWorms();

        /* Realiza la animacion del disparo del arma */
        void shoot(const std::string& weapon);
};


#endif
```

```cpp
#include "WeaponPowerAccum.h"
#include "Handlers.h"

const int TIME_STEP = 50;
const int MINIMUM_POWER = 1000;
const int POWER_STEP = 15;

WeaponPowerAccum::WeaponPowerAccum(Handlers& handlers, int time) :
    actual_time(0), max_time(time), handlers(handlers) {}

WeaponPowerAccum::~WeaponPowerAccum() {}

bool WeaponPowerAccum::startCallBack() {
    this->actual_time += TIME_STEP;
    this->power += POWER_STEP;

    if (this->actual_time == this->max_time) {
        this->handlers.powerAccumStopped(this->power);
        return false;
    }
    return true;
}

void WeaponPowerAccum::start() {
    this->actual_time = 0;
    this->power = MINIMUM_POWER;
    this->my_connection = Glib::signal_timeout().connect(sigc::mem_fun(*this, &W
eaponPowerAccum::startCallBack), TIME_STEP);
}

void WeaponPowerAccum::stop() {
    if (this->my_connection.connected()) {
        this->my_connection.disconnect();
        this->handlers.powerAccumStopped(this->power);
    }
}
```

```
#ifndef __CLIENTTIMER_H__
#define __CLIENTTIMER_H__

#include <glibmm/main.h>

class Handlers;

/* Clase que simula a un contador */
class WeaponPowerAccum {
    private:
        int actual_time;
        int max_time;
        int power;
        Handlers& handlers;
        sigc::connection my_connection;

        /* Callback de start */
        bool startCallBack();

    public:
        /* Constructor */
        WeaponPowerAccum(Handlers& handlers, int time);

        /* Destructor */
        ~WeaponPowerAccum();

        /* Cuenta el tiempo transcurrido y llama al metodo timerStopped
           de la clase Handler con este tiempo */
        void start();

        /* Detiene el contador */
        void stop();
};

#endif
```

```
#include "ClientProtocol.h"
#include <string>
#include "Player.h"
#include "WeaponList.h"
#include "ObjectSizes.h"
#include "ServerFatalError.h"

ClientProtocol::ClientProtocol(Socket&& socket, Gtk::Window& window): Protocol(s
td::move(socket)), window(window){}

ClientProtocol::ClientProtocol(ClientProtocol&& other): Protocol(std::move(other
)), window(other.window) {}

ClientProtocol::~ClientProtocol(){}

void ClientProtocol::sendMoveAction(char action){
    Buffer buffer;
    buffer.setNext(ACTION);
    buffer.setNext(MOVE_ACTION);
    buffer.setNext(action);
    this->sendBuffer(buffer);
}

void ClientProtocol::sendChangeWeapon(const std::string &weapon){
    Buffer buffer;
    buffer.setNext(ACTION);
    buffer.setNext(CHANGE_WEAPON_ACTION);
    this->sendStringBuffer(buffer, weapon);
    this->sendBuffer(buffer);
}

void ClientProtocol::sendWeaponShoot(int32_t angle, int32_t power, int32_t time)
{
    Buffer buffer;
    buffer.setNext(ACTION);
    buffer.setNext(SHOOT_WEAPON);
    this->sendIntBuffer(buffer, angle);
    this->sendIntBuffer(buffer, power);
    this->sendIntBuffer(buffer, time);
    this->sendBuffer(buffer);
}

void ClientProtocol::sendWeaponSelfDirectedShoot(const Position &pos) {
    Buffer buffer;
    buffer.setNext(ACTION);
    buffer.setNext(SHOOT_SELF_DIRECTED);

    this->sendIntBuffer(buffer, pos.getX() * UNIT_TO_SEND);
    this->sendIntBuffer(buffer, pos.getY() * UNIT_TO_SEND);

    this->sendBuffer(buffer);
}

void ClientProtocol::updateScope(int angle) {
    Buffer buffer;
    buffer.setNext(ACTION);
    buffer.setNext(MOVE_SCOPE);

    this->sendIntBuffer(buffer, angle);

    this->sendBuffer(buffer);
}

void ClientProtocol::sendEndTurn(){
    Buffer buffer;
    buffer.setNext(END_TURN);
```

```cpp
    this->sendBuffer(buffer);
}

void ClientProtocol::receiveStartGame(){
    Buffer buffer = std::move(this->receiveBuffer());
}

void ClientProtocol::receiveBackgroundImage(WorldView& world){
    Buffer buffer = std::move(this->receiveBuffer());
    world.setBackgroundImage(buffer);
}

void ClientProtocol::receivePlayers(PlayersList& players_list){
    Buffer buffer = std::move(this->receiveBuffer());
    int quantity = this->receiveIntBuffer(buffer);

    for (int i = 0; i < quantity; i++){
        Buffer buffer = std::move(this->receiveBuffer());

        int id = this->receiveIntBuffer(buffer);
        std::string name = this->receiveStringBuffer(buffer);

        players_list.addPlayer(id, name);
    }
}

void ClientProtocol::receiveGirders(ViewsList& viewsList){
    Buffer buffer = std::move(this->receiveBuffer());
    int quantity = this->receiveIntBuffer(buffer);

    for (int i = 0; i < quantity; i++){
        Buffer buffer = std::move(this->receiveBuffer());;

        int size = this->receiveIntBuffer(buffer);
        float pos_x = this->receiveIntBuffer(buffer) / UNIT_TO_SEND;
        float pos_y = this->receiveIntBuffer(buffer) / UNIT_TO_SEND;
        int rotation = this->receiveIntBuffer(buffer);
        viewsList.addGirder(size, pos_x, pos_y, rotation);
    }
}

void ClientProtocol::receiveWeaponsAmmo(WeaponList& weapon_list){
    Buffer buffer = std::move(this->receiveBuffer());
    int quantity = this->receiveIntBuffer(buffer);

    for (int i = 0; i < quantity; i++){
        Buffer buffer = std::move(this->receiveBuffer());

        std::string name = this->receiveStringBuffer(buffer);
        int ammo = this->receiveIntBuffer(buffer);
        weapon_list.add(name, ammo);
    }
}

void ClientProtocol::sendBuffer(Buffer &buffer){
    try{
        Protocol::sendBuffer(buffer);
    } catch (const std::exception& e){
        ServerFatalError error(this->window);
    }
}
```

```cpp
#ifndef __CLIENTPROTOCOL_H__
#define __CLIENTPROTOCOL_H__

#include "Socket.h"
#include "Protocol.h"
#include "Position.h"
#include "ViewsList.h"
#include "PlayersList.h"
#include <gtkmm/window.h>

class Player;
class WeaponList;

/* Clase que se encarga de enviar y recibir mensajes del socket
 * con un formato determinado */
class ClientProtocol: public Protocol {
    private:
        Gtk::Window& window;

    public:
        /* Constructor */
        ClientProtocol(Socket&& socket, Gtk::Window& window);

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

        /* Destructor */
        ~ClientProtocol();

        /* Envia un mensaje que indica una accion de movimiento */
        void sendMoveAction(char action);

        /* Envia un mensaje que indica una accion de cambio de arma
         * con el nombre del arma */
        void sendChangeWeapon(const std::string &weapon);

        /* Envia un mensaje de accion de disparo, con el angulo, la potencia
         * y el tiempo de explosion */
        void sendWeaponShoot(int32_t angle, int32_t power, int32_t time);

        /* Envia un mensaje de accion de disparo teledirigido con
         * la posicion del disparo */
        void sendWeaponSelfDirectedShoot(const Position &pos);

        /* Envia un mesaje que indica el cambio del angulo del scope */
        void updateScope(int angle);

        /* Envia un mensaje de finalizacion de turno */
        void sendEndTurn();

        /* Recibe el comienzo del juego */
        void receiveStartGame();

        /* Recibe y setea la imagen de fondo */
        void receiveBackgroundImage(WorldView& world);

        /* Recibe los jugadores de la partida junto con su
         * id y su nombre */
        void receivePlayers(PlayersList& players_list);

        /* Recibe la vigas presentes en el mapa junto con su tamaÃ±o,
         * su posicion y su rotacion */
        void receiveGirders(ViewsList& viewsList);

        /* Recibe las armas presentes en el juego junto con
         * su municion */
        void receiveWeaponsAmmo(WeaponList& weapon_list);

        /* Envia el contenido del buffer */
        void sendBuffer(Buffer &buffer) override;
};

#endif
```

```
#include "DataReceiver.h"
#include "Player.h"
#include <glibmm/main.h>
#include "ObjectSizes.h"

DataReceiver::DataReceiver(Player& player):
    player(player), protocol(player.getProtocol()){}

DataReceiver::~DataReceiver(){
    this->protocol.stop();
}

void DataReceiver::run(){
    try{
        this->initialConfig();
        while(this->running){
            Buffer data = this->protocol.receiveBuffer();
            sigc::slot<bool> my_slot = sigc::bind(sigc::mem_fun(*this, &DataRece
iver::analizeReceivedData), data);
            Glib::signal_idle().connect(my_slot);
        }

    } catch (const std::exception& e){
        if (this->running){
            this->player.getScreen().close();
        }
    }
}

void DataReceiver::initialConfig(){
    this->protocol.receiveStartGame();
    this->protocol.receiveBackgroundImage(this->player.getScreen().getWorld());
    this->protocol.receivePlayers(this->player.getScreen().getPlayersView());
    this->protocol.receiveGirders(this->player.getViewsList());
    this->protocol.receiveWeaponsAmmo(this->player.getWeapons());
    this->player.getScreen().show();
}

bool DataReceiver::analizeReceivedData(Buffer buffer){
    char action = buffer.getNext();

    if (action == START_TURN){
        int worm_id = Protocol::receiveIntBuffer(buffer);
        int player_id = Protocol::receiveIntBuffer(buffer);
        float wind = Protocol::receiveIntBuffer(buffer) / UNIT_TO_SEND;
        this->player.startTurn(worm_id, player_id, wind);
    } else if (action == END_GAME){
        std::string winner = Protocol::receiveStringBuffer(buffer);
        this->player.endGame(winner);
    } else if (action == END_TURN){
        this->player.endTurnEarly();
    } else if (action == CHANGE_WEAPON_ACTION) {
        std::string weapon(Protocol::receiveStringBuffer(buffer));
        this->player.getViewsList().removeScopeVisibility();
        this->player.getViewsList().changeWeapon(weapon);
    } else if (action == MOVE_SCOPE) {
        int angle = Protocol::receiveIntBuffer(buffer);
        this->player.getViewsList().updateScope(angle);
    } else if (action == SHOOT_WEAPON_ACTION) {
        std::string weapon(Protocol::receiveStringBuffer(buffer));
        this->player.getViewsList().removeScopeVisibility();
        this->player.getViewsList().shoot(weapon);
        this->player.getMusicPlayer().playWeaponShotSound(weapon);
    } else if (action == MOVING_OBJECT) {
        char type = buffer.getNext();
        int id = Protocol::receiveIntBuffer(buffer);
```

```cpp

        if (type == WORM_TYPE){
            int player_id = Protocol::receiveIntBuffer(buffer);
            int pos_x = Protocol::receiveIntBuffer(buffer);
            int pos_y = Protocol::receiveIntBuffer(buffer);
            int life = Protocol::receiveIntBuffer(buffer);
            char dir = buffer.getNext();
            bool colliding = buffer.getNext();
            this->player.getViewsList().updateWormData(id, player_id, pos_x, pos_y, life, dir, colliding);
            this->player.getViewsList().removeScopeVisibility();
        } else if (type == WEAPON_TYPE){
            std::string weapon(Protocol::receiveStringBuffer(buffer));

            int pos_x = Protocol::receiveIntBuffer(buffer);
            int pos_y = Protocol::receiveIntBuffer(buffer);
            this->player.getViewsList().updateWeaponData(id, weapon, pos_x, pos_y);
        }
    } else if (action == DEAD_OBJECT){
        char type = buffer.getNext();
        int id = Protocol::receiveIntBuffer(buffer);
        if (type == WORM_TYPE){
            this->player.getViewsList().removeWorm(id);
        } else if (type == WEAPON_TYPE){
            this->player.getViewsList().removeWeapon(id);
        }
    } else if (action == MOVE_ACTION){
        char movement = buffer.getNext();
        this->player.getMusicPlayer().playJumpSound(movement);
    }
    return false;
}
```

```cpp
#ifndef __DATARECEIVER_H__
#define __DATARECEIVER_H__

#include "Thread.h"
#include "ClientProtocol.h"

class Player;

/* Clase que se encarga de recibir los mensajes enviados por el servidor */
class DataReceiver: public Thread{
    private:
        Player& player;
        ClientProtocol& protocol;

        /* Recibe los datos de la configuracion inicial */
        void initialConfig();

        /* Analiza los datos recibidos */
        bool analizeReceivedData(Buffer buffer);

    public:
        /* Constructor */
        DataReceiver(Player& player);

        /* Destructor */
        ~DataReceiver();

        /* Comienza a recibir mensajes del protocolo */
        void run() override;
};


#endif
```

```
#include "Handlers.h"
#include <gtkmm/adjustment.h>
#include <gdk/gdkkeysyms.h>
#include "Player.h"
#include "ViewPositionTransformer.h"
#include "WeaponNames.h"

const char SPACE = ' ';
const int WEAPONS_DEFAULT_TIME = 3;
const char ASCII_OFFSET = 48;
const char ASCII_1 = 49;
const char ASCII_5 = 53;
const int MAX_TIME = 3000;
const int ANGLE_STEP = 6;

Handlers::Handlers(Player& player, ViewsList& view_list, WeaponList& weapons, WorldView& world):
    player(player), view_list(view_list), weapons(weapons), world(world),
    scroll_handler(world.getWindow()), power_accumulator(*this, MAX_TIME){
        this->has_shoot = false;
        this->current_angle = DEFAULT_ANGLE;
        this->weapons_time = WEAPONS_DEFAULT_TIME;
        this->enabled = false;
    }

Handlers::~Handlers() {}

void Handlers::enableAll(){
    this->weapons_time = WEAPONS_DEFAULT_TIME;
    this->current_angle = DEFAULT_ANGLE;
    this->has_shoot = false;

    this->player.getProtocol().updateScope(DEFAULT_ANGLE);

    this->world.getWindow().get_parent()->get_parent()->set_can_focus(true);
    this->world.getWindow().get_parent()->get_parent()->grab_focus();

    this->world.getWindow().get_parent()->get_parent()->signal_key_press_event().connect(sigc::mem_fun(*this,
                                   &Handlers::keyPressHandler));
    this->world.getWindow().get_parent()->get_parent()->signal_key_release_event().connect(sigc::mem_fun(*this,
                                     &Handlers::keyReleaseHandler));
    this->world.getWindow().signal_button_press_event().connect(sigc::mem_fun(*this, &Handlers::onButtonPressEvent));

    this->enabled = true;
}

void Handlers::disableAll() {
    this->world.getWindow().get_parent()->get_parent()->signal_key_press_event().connect(sigc::mem_fun(*this,
                                   &Handlers::inactiveKeyHandler));
    this->world.getWindow().get_parent()->get_parent()->signal_key_release_event().connect(sigc::mem_fun(*this,
                                     &Handlers::inactiveKeyHandler));
    this->world.getWindow().signal_button_press_event().connect(sigc::mem_fun(*this, &Handlers::inactiveButtonHandler));

    this->enabled = false;
}

```

```
void Handlers::powerAccumStopped(int power){
    this->player.shoot(this->current_angle, power, this->weapons_time);
}

bool Handlers::keyPressHandler(GdkEventKey *key_event) {
    if (key_event->keyval == GDK_KEY_Left) {
        this->player.getProtocol().sendMoveAction(MOVE_LEFT);
    } else if (key_event->keyval == GDK_KEY_Right) {
        this->player.getProtocol().sendMoveAction(MOVE_RIGHT);
    } else if (key_event->keyval == GDK_KEY_Return) {
        this->player.getProtocol().sendMoveAction(JUMP);
    } else if (key_event->keyval == GDK_KEY_BackSpace) {
        this->player.getProtocol().sendMoveAction(ROLLBACK);
    } else if (key_event->keyval == GDK_KEY_Up) {
        if (!this->weapons.getCurrentWeapon().hasScope()) {
            return true;
        }
        if (this->current_angle < MAX_WEAPON_ANGLE) {
            this->current_angle += ANGLE_STEP;
        }
        this->player.getProtocol().updateScope(this->current_angle);
    } else if (key_event->keyval == GDK_KEY_Down) {
        if (!this->weapons.getCurrentWeapon().hasScope()) {
            return true;
        }
        if (this->current_angle > MIN_WEAPON_ANGLE) {
            this->current_angle -= ANGLE_STEP;
        }
        this->player.getProtocol().updateScope(this->current_angle);
    } else if (key_event->keyval >= ASCII_1 && key_event->keyval <= ASCII_5) {
        this->weapons_time = key_event->keyval - ASCII_OFFSET;
    } else if (key_event->keyval == SPACE && key_event->type == GDK_KEY_PRESS) {
        if (this->weapons.getCurrentWeapon().isSelfDirected()) {
            return true;
        }
        if (!this->weapons.getCurrentWeapon().hasAmmo()) {
            return true;
        }
        if (this->has_shoot) {
            return true;
        }
        this->has_shoot = true;
        if (!this->weapons.getCurrentWeapon().hasVariablePower()) {
            this->player.shoot(this->current_angle, -1, this->weapons_time);
        } else {
            this->power_accumulator.start();
        }
    }
    return true;
}

bool Handlers::keyReleaseHandler(GdkEventKey *key_event) {
    if (key_event->type == GDK_KEY_RELEASE) {
        if (key_event->keyval == SPACE) {
            if (this->weapons.getCurrentWeapon().isSelfDirected()) {
                return true;
            }
            if (!this->weapons.getCurrentWeapon().hasVariablePower()) {
                return true;
            }
            if (!this->weapons.getCurrentWeapon().hasAmmo()) {
                this->player.getMusicPlayer().playNoAmmo();
                return true;
            }
            this->power_accumulator.stop();
        }
    }
    return true;
}

bool Handlers::onButtonPressEvent(GdkEventButton *event) {
    if (!this->weapons.getCurrentWeapon().isSelfDirected()) {
        return true;
    }
    if (!this->weapons.getCurrentWeapon().hasAmmo()) {
        this->player.getMusicPlayer().playNoAmmo();
        return true;
    }
    if (this->has_shoot) {
        return true;
    }
    if ((event->type == GDK_BUTTON_PRESS) && (event->button == 1)) {
        float x = event->x;
        float y = event->y;
        x += this->world.getWindow().get_hadjustment()->get_value();
        y += this->world.getWindow().get_vadjustment()->get_value();
        Position position(x, y);
        Position newPosition = ViewPositionTransformer(this->world.getLayout()).
transformToPosition(position);
        this->has_shoot = true;
        this->player.shoot(newPosition);
    }
    return true;
}

bool Handlers::inactiveKeyHandler(GdkEventKey *key_event) {
    return true;
}

bool Handlers::inactiveButtonHandler(GdkEventButton *event) {
    return true;
}

int Handlers::getCurrentAngle() const{
    return this->current_angle;
}

bool Handlers::isEnabled(){
    return this->enabled;
}
```

```
#ifndef __HANDLERS__H__
#define __HANDLERS__H__

#include <gdk/gdk.h>
#include "WeaponPowerAccum.h"
#include "ScrollHandler.h"

class Player;
class ViewsList;
class WeaponList;
class WorldView;

/* Clase que se encarga de definir los handlers del teclado y
   del mouse. */
class Handlers{
    private:
        Player& player;
        ViewsList& view_list;
        WeaponList& weapons;
        WorldView& world;
        ScrollHandler scroll_handler;

        bool has_shoot;
        int current_angle;
        int weapons_time;
        bool enabled;

        WeaponPowerAccum power_accumulator;


    public:
        /* Constructor */
        Handlers(Player& player, ViewsList& view_list, WeaponList& weapons, Worl
dView& world);

        /* Destructor */
        ~Handlers();

        /* Handler completo para el presionado de teclas. Indica
           los pasos que se deben realizar al presionar una tecla
           especifica */
        bool keyPressHandler(GdkEventKey *key_event);

        /* Handler completo para la liberaciÃ³n de teclas. Indica
           los pasos que se deben realizar al liberar una tecla
           especifica */
        bool keyReleaseHandler(GdkEventKey *key_event);

        /* Handler del mouse. Indica los pasos que se deben realizar
           al utilizar el mouse */
        bool onButtonPressEvent(GdkEventButton *event);

        /* Handler inactivo de las teclas. Indica que no se debe
           realizar ninguna accion al producirse un evento */
        bool inactiveKeyHandler(GdkEventKey *key_event);

        /* Handler inactivo del mouse. Indica que no se debe
           realizar ninguna accion al producirse un evento */
        bool inactiveButtonHandler(GdkEventButton *event);

        /* Habilita todos los handlers */
        void enableAll();

        /* Deshabilita todos los handlers */
        void disableAll();

        /* Realiza el shoot del player */
        void powerAccumStopped(int power);

        /* Devuelve el angulo actual del scope */
        int getCurrentAngle() const;

        /* Devuelve true si estan habilitados los handlers */
        bool isEnabled();
};

#endif
```

```
#include "PlayerLifeLabel.h"
#include "GamePlayers.h"

const std::string begining("<span color='");
const std::string middle("'>");
const std::string ending("</span>");

PlayerLifeLabel::PlayerLifeLabel(): id(0), player_name(""), life(0){
    this->label.set_use_markup(true);
}

PlayerLifeLabel::~PlayerLifeLabel(){}

void PlayerLifeLabel::setPlayerName(int id, const std::string& player_name){
    this->id = id;
    this->player_name = player_name;
    this->updateLabel();
}

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

void PlayerLifeLabel::reduceLife(int life){
    this->life -= life;
    this->updateLabel();
}

Gtk::Label& PlayerLifeLabel::getLabel(){
    return this->label;
}

void PlayerLifeLabel::updateLabel(){
    std::string message = begining + colors[this->id] + middle;
    message += std::to_string(this->id) + "- " + this->player_name;
    message += ": " + std::to_string(this->life) + ending;
    this->label.set_markup(message);
}
```

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

```cpp
#ifndef __PLAYERLIFELABEL_H__
#define __PLAYERLIFELABEL_H__

#include <gtkmm/label.h>

/* Clase que se encarga de controlar el indicador de vida del jugador */
class PlayerLifeLabel{
    private:
        int id;
        std::string player_name;
        int life;
        Gtk::Label label;

        /* Actualiza la informacion del label */
        void updateLabel();

    public:
        /* Constructor */
        PlayerLifeLabel();

        /* Destructor */
        ~PlayerLifeLabel();


        /* Establece el nombre del jugador */
        void setPlayerName(int id, const std::string& player_name);

        /* Agrega la vida al label */
        void addLife(int life);

        /* Disminuye la vida y actualiza la vista del label */
        void reduceLife(int life);

        /* Devuelve el label del jugador */
        Gtk::Label& getLabel();
};


#endif
```

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

```cpp
#include "PlayersList.h"
#include <glibmm/main.h>

#define SPACING 20

PlayersList::PlayersList(): container(false, SPACING){
    this->title.set_use_markup(true);
    this->title.set_markup("<span><b><u>Jugadores</u></b></span>");
    this->container.pack_start(this->title, Gtk::PACK_SHRINK);
}

PlayersList::~PlayersList(){}

void PlayersList::addPlayer(int id, const std::string& name){
    sigc::slot<bool> my_slot = sigc::bind(sigc::mem_fun(*this, &PlayersList::add
PLayerCallBack), id, name);
    Glib::signal_idle().connect(my_slot);
}

bool PlayersList::addPLayerCallBack(int id, std::string name){
    this->players[id] = name;
    this->labels[id].setPlayerName(id, name);
    this->container.pack_start(this->labels[id].getLabel(), Gtk::PACK_SHRINK);
    return false;
}

const std::string& PlayersList::getPlayer(int id) const{
    return this->players.at(id);
}

Gtk::Container& PlayersList::getWindow(){
    return this->container;
}

void PlayersList::addPlayerLife(int player_id, int life){
    this->labels[player_id].addLife(life);
}

void PlayersList::reducePlayerLife(int player_id, int life){
    this->labels[player_id].reduceLife(life);
}
```

```
#ifndef __PLAYERSLIST_H__
#define __PLAYERSLIST_H__

#include <map>
#include <string>
#include <gtkmm/hvbox.h>
#include <gtkmm/label.h>
#include "PlayerLifeLabel.h"

/* Clase que se encarga de almacenar los nombres y las vidas
 * de todos los jugadores */
class PlayersList{
    private:
        std::map<int, std::string> players;
        std::map<int, PlayerLifeLabel> labels;
        Gtk::VBox container;
        Gtk::Label title;

        bool addPLayerCallBack(int id, std::string name);

    public:
        /* Constructor */
        PlayersList();

        /* Destructor */
        ~PlayersList();

        /* Agrega al jugador a la lista de jugadores y agrega su
         * informacion a la vista */
        void addPlayer(int id, const std::string& name);

        /* Devuelve el nombre del jugador */
        const std::string& getPlayer(int id) const;

        /* Devuelve el contenedor de los jugadores */
        Gtk::Container& getWindow();

        /* Agrega la informacion de la vida del jugador a la vista */
        void addPlayerLife(int player_id, int life);

        /* Reduce la vida del jugador y actualiza la vista */
        void reducePlayerLife(int player_id, int life);
};

#endif
```

```
#include "ScreenView.h"
#include "ServerFatalError.h"
#include <glibmm/main.h>

#define PADDING 10
#define SPACING 30

ScreenView::ScreenView(Gtk::Window& window, Player& player, WeaponList& weapons)
 :
    left_view(false, SPACING), window(window), weapons_view(weapons, player) {
    this->left_view.pack_start(this->wind_view.getWindow(), Gtk::PACK_SHRINK);
    this->left_view.pack_start(this->players.getWindow(), Gtk::PACK_SHRINK);
    this->world_box.pack_start(this->left_view, Gtk::PACK_SHRINK, PADDING);
    this->world_box.pack_start(this->world.getContainer());
    this->world_box.pack_end(this->weapons_view.getWindow(), Gtk::PACK_SHRINK);

    this->screen.pack_start(this->turn_label.getWindow(), Gtk::PACK_SHRINK);
    this->screen.pack_end(this->world_box);
}

ScreenView::~ScreenView() {}

void ScreenView::show(){
    sigc::slot<bool> my_slot = sigc::mem_fun(*this, &ScreenView::showCallBack);
    Glib::signal_idle().connect(my_slot);
}

bool ScreenView::showCallBack(){
    this->weapons_view.update();
    this->window.remove();
    this->window.add(this->screen);
    this->window.show_all();
    return false;
}

void ScreenView::close(){
    sigc::slot<bool> my_slot = sigc::mem_fun(*this, &ScreenView::closeCallBack);
    Glib::signal_idle().connect(my_slot);
}

bool ScreenView::closeCallBack(){
    ServerFatalError error(this->window);
    return false;
}

WorldView& ScreenView::getWorld() {
    return this->world;
}

WeaponView& ScreenView::getWeaponsView() {
    return this->weapons_view;
}

TurnLabel& ScreenView::getTurnLabel(){
    return this->turn_label;
}

PlayersList& ScreenView::getPlayersView(){
    return this->players;
}

WindView& ScreenView::getWindView(){
    return this->wind_view;
}
```

```cpp
#ifndef __CLIENTSCREENVIEW_H__
#define __CLIENTSCREENVIEW_H__

#include <gtkmm/hvbox.h>
#include <gtkmm/label.h>
#include <gtkmm/window.h>
#include "WorldView.h"
#include "WeaponView.h"
#include "TurnLabel.h"
#include "PlayersList.h"
#include "WindView.h"

/* Clase que se encarga de almacenar los contenedores principales
 * de la vista y mostrar su contenido */
class ScreenView {
    private:
        Gtk::VBox screen;
        Gtk::HBox world_box;
        Gtk::VBox left_view;
        Gtk::Window& window;

        WorldView world;
        WeaponView weapons_view;
        TurnLabel turn_label;
        PlayersList players;
        WindView wind_view;

        /* CallBacks */
        bool showCallBack();
        bool closeCallBack();

    public:
        /* Constructor */
        ScreenView(Gtk::Window& window, Player& player, WeaponList& weapons);

        /* Destructor */
        ~ScreenView();

        /* Muestra la pantalla en la ventana */
        void show();

        /* Cierra la ventana completamente */
        void close();

        /* Devuelve el WorldView */
        WorldView& getWorld();

        /* Devuelve el WeaponView */
        WeaponView& getWeaponsView();

        /* Devuelve el TurnLabel */
        TurnLabel& getTurnLabel();

        /* Devuelve el Players view */
        PlayersList& getPlayersView();

        /* Devuelve el wind view */
        WindView& getWindView();
};

#endif
```

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

const std::string begining("<span size='20000'>");
const std::string ending("</span>");

TurnLabel::TurnLabel() {
    this->message.set_use_markup(true);
    this->message.set_markup(begining + "Worms" + ending);
    this->label.pack_start(this->message);
    this->label.pack_end(this->time);
}

TurnLabel::~TurnLabel() {}

void TurnLabel::beginTurn() {
    std::string message = begining + "Tu turno" + ending;
    this->message.set_markup(message);
}

void TurnLabel::beginTurn(const std::string& player_name) {
    std::string message = begining + "Turno de " + player_name + ending;
    this->message.set_markup(message);
}

void TurnLabel::endTurn() {
    this->time.set_markup("");
    this->message.set_markup(begining + "Termino tu turno" + ending);
}

void TurnLabel::setTime(int time) {
    this->time.set_markup(begining + std::to_string(time) + ending);
}

void TurnLabel::setWinner(const std::string& winner, bool i_win){
    this->message.set_markup(begining + "Termino el juego" + ending);
    std::string winner_message;
    if (winner.empty()){
        winner_message = "Empate";
    } else if (i_win) {
        winner_message = "GANASTE!!!!";
    } else {
        winner_message = "Perdiste :( El ganador fue: " + winner;
    }
    this->time.set_markup(begining + winner_message + ending);
}

Gtk::Container& TurnLabel::getWindow() {
    return this->label;
}
```

```
#ifndef __TURNLABEL_H__
#define __TURNLABEL_H__

#include <gtkmm/hvbox.h>
#include <gtkmm/label.h>

/* Clase que se encarga de controlar los labels que indican
 * el estado del turno */
class TurnLabel{
    private:
        Gtk::Label message;
        Gtk::Label time;
        Gtk::HBox label;

    public:
        /* Constructor */
        TurnLabel();

        /* Destructor */
        ~TurnLabel();


        /* Cambia el label indicando que es el turno del jugador */
        void beginTurn();

        /* Cambia el label indicando que es el turno del jugador
         * con nombre pasado por parametro */
        void beginTurn(const std::string& player_name);

        /* Cambia el label indicando que finalizo el turno del jugador */
        void endTurn();

        /* Cambia el label mostrando al ganador */
        void setWinner(const std::string& winner, bool i_win);

        /* Cambia el label de tiempo al tiempo pasado por parametro */
        void setTime(int time);

        /* Devuelve el contenedor de la vista */
        Gtk::Container& getWindow();
};


#endif
```

```
#include "WeaponButton.h"
#include "Player.h"
#include "Path.h"

WeaponButton::WeaponButton(const std::string& weapon_name, unsigned int ammo, Pl
ayer& player) :
    weapon_name(weapon_name), player(player) {
    this->setLabel(ammo);
    std::string path = WEAPONS_PATH;
    path += weapon_name + ".png";
    this->image.set(path);
    this->button.set_image(this->image);
    this->button.set_always_show_image(true);
    this->button.signal_clicked().connect(sigc::mem_fun(*this, &WeaponButton::on
ClickedButton));
}

WeaponButton::~WeaponButton() {}

void WeaponButton::onClickedButton() {
    this->player.changeWeapon(weapon_name);
}

Gtk::Widget& WeaponButton::getButton() {
    return this->button;
}

void WeaponButton::setLabel(unsigned int ammo){
    std::string label = "Ammo:\n  ";
    if (!ammo){
        label += "0";
        button.set_sensitive(false);
    }
    else if (ammo > 100){
        label += "âM−^HM−^^";
    } else {
        label += std::to_string(ammo);
    }
    this->button.set_label(label);
}
```

```
#ifndef __CLIENTWEAPONBUTTON_H__
#define __CLIENTWEAPONBUTTON_H__

#include <gtkmm/togglebutton.h>
#include <gtkmm/image.h>
#include <string>

class Player;

/* Clase que se encarga de mostrar el boton de un arma
 * junto con la informacion correspondiente a esa arma */
class WeaponButton {
    private:
        std::string weapon_name;
        Player& player;
        Gtk::Button button;
        Gtk::Image image;

    public:
        /* Constructor */
        WeaponButton(const std::string& weapon_name, unsigned int ammo, Player& player);

        /* Destructor */
        ~WeaponButton();

        /* Devuelve el wiget del boton */
        Gtk::Widget& getButton();

        /* Setea el label del boton */
        void setLabel(unsigned int ammo);

        /* Handler del boton al ser clickeado */
        void onClickedButton();
};


#endif
```

```
#include "WeaponView.h"
#include <glibmm/main.h>
#include "Player.h"
#include "WeaponList.h"
#include "WeaponButton.h"

WeaponView::WeaponView(WeaponList& weapons, Player& player) :
                    weapons(weapons), player(player) {}

WeaponView::~WeaponView() {}

void WeaponView::update() {
    WeaponList::iterator iter;
    int row = 1, column = 1;
    for (iter = this->weapons.begin(); iter != this->weapons.end(); iter++) {
        std::unique_ptr<WeaponButton> p(new WeaponButton(iter->second->getName(), iter->second->getAmmo(), this->player));
        this->buttons.insert(std::pair<std::string, std::unique_ptr<WeaponButton>>(iter->second->getName(), std::move(p)));
        this->window.attach(this->buttons.at(iter->second->getName())->getButton(), column, row, 1, 1);
        row++;
    }
}

Gtk::Grid& WeaponView::getWindow() {
    return this->window;
}

void WeaponView::updateAmmo(const Weapon& weapon){
    this->buttons[weapon.getName()]->setLabel(weapon.getAmmo());
}
```

Jun 03, 18 12:56 **WeaponView.h** Page 1/1

```cpp
#ifndef __CLIENTWEAPONVIEW_H__
#define __CLIENTWEAPONVIEW_H__

#include <gtkmm/grid.h>
#include <unordered_map>
#include <memory>
#include <string>

class Player;
class WeaponList;
class WeaponButton;
class Weapon;

/* Clase que se encarga de mostrar los datos de las armas del juego
 * y de almacenar todos los botones de las armas */
class WeaponView {
    private:
        WeaponList& weapons;
        Gtk::Grid window;
        Player& player;
        std::unordered_map<std::string, std::unique_ptr<WeaponButton>> buttons;

    public:
        /* Constructor */
        WeaponView(WeaponList& weapons, Player& player);

        /* Destructor */
        ~WeaponView();


        /* Actualiza la informacion de todos los botones */
        void update();

        /* Actualiza la informacion de la municion del arma especifica */
        void updateAmmo(const Weapon& weapon);

        /* Devuelve el contenedor de la vista */
        Gtk::Grid& getWindow();
};

#endif
```

Jun 02, 18 13:59 **WindView.cpp** Page 1/1

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

WindView::WindView(): container(false, 7){
    this->container.pack_start(this->velocity, Gtk::PACK_SHRINK);
    this->container.pack_start(this->direction, Gtk::PACK_SHRINK);
    this->velocity.set_use_markup(true);
}

WindView::~WindView(){}

void WindView::update(float wind){
    wind *= 10;
    std::string message = "<span><b><u>Viento</u></b>\n\n";
    std::string direction = "right";
    if (wind == 0){
        direction = "no";
    } else if (wind < 0){
        wind *= -1;
        direction = "left";
    }
    std::string velocity = std::to_string(wind);
    message += velocity.substr(0,4) + "</span>";
    this->velocity.set_markup(message);
    this->direction.set(IMAGES_PATH + "arrow_" + direction + ".png");
}

Gtk::VBox& WindView::getWindow(){
    return this->container;
}
```

```cpp
#ifndef __WINDVIEW_H__
#define __WINDVIEW_H__

#include <gtkmm/hvbox.h>
#include <gtkmm/label.h>
#include <gtkmm/image.h>

class WindView{
    private:
        Gtk::VBox container;
        Gtk::Label velocity;
        Gtk::Image direction;

    public:
        WindView();
        ~WindView();

        //Actualiza la vista del viento
        void update(float wind);

        Gtk::VBox& getWindow();
};

#endif

```

```cpp
#include "WorldView.h"
#include <gtkmm/adjustment.h>
#include <glibmm/main.h>
#include <giomm/memoryinputstream.h>
#include "ViewPositionTransformer.h"
#include "Player.h"
#include "Math.h"
#include "Path.h"
#include "ObjectSizes.h"

WorldView::WorldView() {
    this->container.add_overlay(this->background);
    this->world.set_size(map_width, map_height);
    this->window.add_events(Gdk::BUTTON_PRESS_MASK);
    this->window.add(this->world);
    this->container.add_overlay(this->window);

    this->water.show(this->world);
    this->window.get_hadjustment()->set_value(map_width / 2);
    this->window.get_vadjustment()->set_value(map_height);
}

WorldView::~WorldView() {}

void WorldView::moveElement(Gtk::Widget& element, const Position& position, float width, float height, bool focus){
    Position newPosition = ViewPositionTransformer(this->world).transformToScreenAndMove(position, width, height);
    this->world.move(element, newPosition.getX(), newPosition.getY());
    if (focus){
        this->setFocus(element);
    }
}

void WorldView::moveScope(Gtk::Widget& scope, Gtk::Widget& worm, int angle) {
    float pos_x = this->world.child_property_x(worm).get_value();
    float pos_y = this->world.child_property_y(worm).get_value();
    pos_x += 50 * Math::cosDegrees(angle);
    pos_y -= 50 * Math::sinDegrees(angle);
    pos_x -= worm.get_width() / 2; // Para que quede referenciado a la mitad de la imagen
    this->world.move(scope, pos_x, pos_y);
}

void WorldView::removeElement(Gtk::Widget& element){
    this->world.remove(element);
}

void WorldView::addElement(Gtk::Widget& element, const Position& position, float width, float height, bool focus){
    Position newPosition = ViewPositionTransformer(this->world).transformToScreenAndMove(position, width, height);
    this->world.put(element, newPosition.getX(), newPosition.getY());
    element.show_all();
    if (focus){
        this->setFocus(element);
    }
}

Gtk::ScrolledWindow& WorldView::getWindow(){
    return this->window;
}

Gtk::Layout& WorldView::getLayout(){
    return this->world;
}
```

```cpp

void WorldView::setFocus(Gtk::Widget& element){
    this->window.get_hadjustment()->set_value(element.get_allocation().get_x() - this->window.get_hadjustment()->get_page_size() / 2);
    this->window.get_vadjustment()->set_value(element.get_allocation().get_y() - this->window.get_vadjustment()->get_page_size() / 2);
}

void WorldView::setBackgroundImage(const Buffer& image){
    sigc::slot<bool> my_slot = sigc::bind(sigc::mem_fun(*this, &WorldView::setBackgroundImageCallBack), image);
    Glib::signal_idle().connect(my_slot);
}

bool WorldView::setBackgroundImageCallBack(Buffer image){
    auto screen = this->container.get_screen();
    size_t screen_width = screen->get_width();
    size_t screen_height = screen->get_height();
    auto pixbuf = Gio::MemoryInputStream::create();
    pixbuf->add_data(image.getPointer(), image.getMaxSize());
    auto aux = Gdk::Pixbuf::create_from_stream (pixbuf);
    size_t img_width = aux->get_width();
    size_t img_height = aux->get_height();
    for (size_t x = 0; x < screen_width; x += img_width) {
        for (size_t y = 0; y < screen_height; y += img_height) {
            Gtk::Image background_image(aux);
            background_image.show();
            this->background.put(background_image, x, y);
            this->background_images.push_back(std::move(background_image));
        }
    }
    return false;
}

Gtk::Container& WorldView::getContainer(){
    return this->container;
}
```

```cpp
#ifndef __WORLDVIEW_H__
#define __WORLDVIEW_H__

#include <gtkmm/widget.h>
#include <gtkmm/layout.h>
#include <gtkmm/hvbox.h>
#include <gtkmm/scrolledwindow.h>
#include <gtkmm/overlay.h>
#include <string>
#include "Position.h"
#include "Water.h"
#include "Buffer.h"

class Player;

/* Clase que se encarga de mostrar objetos en posiciones
 * especificas, moverlos y eliminarlos de la vista*/
class WorldView{
    private:
        Gtk::Overlay container;
        Gtk::Layout background;
        Gtk::Layout world;
        Gtk::ScrolledWindow window;
        std::vector<Gtk::Image> background_images;
        Water water;

        bool setBackgroundImageCallBack(Buffer image);

    public:
        /* Constructor */
        WorldView();

        /* Destructor */
        ~WorldView();

        /* Setea la imagen de fondo */
        void setBackgroundImage(const Buffer& image);

        /* Mueve el elemento pasado a la posicion especificada */
        void moveElement(Gtk::Widget& element, const Position& position, float width, float height, bool focus = false);

        /* Mueve la mira a la posicion correspondiente para que tenga el angulo
         * especificado por parametro */
        void moveScope(Gtk::Widget& scope, Gtk::Widget& worm, int angle);

        /* Remueve el elemento de la vista */
        void removeElement(Gtk::Widget& element);

        /* Agrega un elemento a la vista en la posicion especificada */
        void addElement(Gtk::Widget& element, const Position& position, float width, float height, bool focus = false);

        /* Devuelve la vista del scrolledWindow */
        Gtk::ScrolledWindow& getWindow();

        /* Devuelve el container */
        Gtk::Container& getContainer();

        /* Devuelve la vista del Layout */
        Gtk::Layout& getLayout();

        /* Realiza focus en el elemento pasado */
        void setFocus(Gtk::Widget& element);
};

```

```cpp

#endif
```

```cpp
#include "Turn.h"
#include <glibmm/main.h>
#include "Player.h"

const int TIMER = 60;
const int REDUCTION_TIME = 3;
const int LIMIT_TIME = 10;

Turn::Turn(Player& player, TurnLabel& time_label):
    actual_time(TIMER), player(player), time_label(time_label){}

Turn::~Turn() {}

bool Turn::startCallBack() {
    this->time_label.setTime(this->actual_time);
    if (this->actual_time <= LIMIT_TIME){
        this->player.playTickTime();
    }
    if (this->actual_time == 0) {
        this->player.endTurn();
    }
    this->actual_time--;
    return this->actual_time >= 0;
}

void Turn::start() {
    this->actual_time = TIMER;
    this->my_connection = Glib::signal_timeout().connect(sigc::mem_fun(*this, &T
urn::startCallBack), 1000);
}

void Turn::reduceTime() {
    this->actual_time = REDUCTION_TIME;
}

void Turn::stop() {
    if (this->my_connection.connected()) {
        this->my_connection.disconnect();
        this->player.endTurn();
    }
}
```

```cpp
#ifndef __CLIENTTURN_H__
#define __CLIENTTURN_H__

#include "TurnLabel.h"

class Player;

/* Clase que se encarga de contar el tiempo del turno */
class Turn {
    private:
        int actual_time;
        Player& player;
        TurnLabel& time_label;
        sigc::connection my_connection;

        /* Callback de start */
        bool startCallBack();

    public:
        /* Constructor */
        Turn(Player& player, TurnLabel& time_label);

        /* Destructor */
        ~Turn();


        /* Comienza la cuenta regresiva del turno actualizando el
         * label que muestra el tiempo */
        void start();

        /* Reduce el tiempo restante del turno a 3 segundos */
        void reduceTime();

        /* Detiene el contador y finaliza el turno */
        void stop();
};

#endif
```

```cpp
#include "BulletView.h"
#include "ObjectSizes.h"

BulletView::BulletView(WorldView& worldView, std::string weapon, Position pos):
    Viewable(worldView), weapon_name(std::move(weapon)) {

    std::string path(BULLETS_PATH);
    path += this->weapon_name;
    path += ".png";
    this->image.set(path);
    this->addToWorld(pos, weapon_size, weapon_size);
}

BulletView::~BulletView() {}

BulletView::BulletView(BulletView&& other): Viewable(std::move(other)),
    image(std::move(other.image)), weapon_name(std::move(other.weapon_name)) {}

void BulletView::updateData(const Position& new_pos){
    this->move(new_pos, weapon_size, weapon_size);
}

Gtk::Widget& BulletView::getWidget(){
    return this->image;
}

std::string BulletView::getName() {
    return this->weapon_name;
}

```

```cpp
#ifndef __CLIENTBULLETVIEW_H__
#define __CLIENTBULLETVIEW_H__

#include <gtkmm/widget.h>
#include <gtkmm/image.h>
#include <string>
#include "Viewable.h"

/* Clase que se encarga de controlar la vista de las balas */
class BulletView: public Viewable{
    private:
        Gtk::Image image;
        std::string weapon_name;

    public:
        /* Constructor */
        BulletView(WorldView& worldView, std::string weapon, Position pos);

        /* Destructor */
        ~BulletView();

        /* Constructor por movimient */
        BulletView(BulletView&& other);

        /* Actualiza la posicion de la bala en la vista */
        void updateData(const Position& new_pos);

        /* Devuelve el contenedor de la bala */
        Gtk::Widget& getWidget() override;

        /* Devuelve el nombre del arma de la bala */
        std::string getName();
};


#endif
```

```cpp
#include "GirderView.h"
#include "GirderSize.h"

GirderView::GirderView(WorldView& worldView, size_t size, Position pos, int rota
tion):
    Viewable(worldView), size(size), rotation(rotation){

    std::string path(GIRDER_PATH);
    path += std::to_string(size);
    path += "_";
    path += std::to_string(rotation);
    path += ".png";
    this->image.set(path);
    float width = GirderSize::getGirderWidthMeters(size, rotation);
    float height = GirderSize::getGirderHeightMeters(size, rotation);
    this->addToWorld(pos, width, height);
}

GirderView::~GirderView(){}

GirderView::GirderView(GirderView&& other): Viewable(std::move(other)),
    image(std::move(other.image)), size(other.size), rotation(other.rotation){}

Gtk::Widget& GirderView::getWidget(){
    return this->image;
}
```

```cpp
#ifndef __GIRDERVIEW_H__
#define __GIRDERVIEW_H__

#include <gtkmm/widget.h>
#include <gtkmm/image.h>
#include <string>
#include "Viewable.h"

/* Clase que se encaga de controlar la vista de las vigas */
class GirderView: public Viewable{
    private:
        Gtk::Image image;
        int size;
        int rotation;

    public:
            /* Constructor */
        GirderView(WorldView& worldView, size_t size, Position pos, int rotation);

        /* Destructor */
        ~GirderView();

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

        /* Devuelve el contenedor de la viga */
        Gtk::Widget& getWidget() override;
};


#endif
```

```cpp
#include "Viewable.h"

Viewable::Viewable(WorldView& worldView): worldView(worldView), has_focus(false){}

Viewable::~Viewable(){}

void Viewable::move(const Position& pos, float width, float height){
    this->worldView.moveElement(this->getWidget(), pos, width, height, this->has_focus);
}

void Viewable::removeFromWorld(){
    this->worldView.removeElement(this->getWidget());
}

void Viewable::addToWorld(const Position& pos, float width, float height){
    this->worldView.addElement(this->getWidget(), pos, width, height, this->has_focus);
}

Viewable::Viewable(Viewable&& other): worldView(other.worldView), has_focus(other.has_focus){}

void Viewable::setFocus(bool focus){
    this->has_focus = focus;
}

bool Viewable::hasFocus() const{
    return this->has_focus;
}
```

```cpp
#ifndef __VIEWABLE_H__
#define __VIEWABLE_H__

#include <gtkmm/widget.h>
#include "WorldView.h"
#include "Position.h"
#include "Path.h"

/* Clase que se encarga de controlar los objetos visuales */
class Viewable{
    private:
        WorldView& worldView;
        bool has_focus;

    protected:
        /* Agrega al objeto visual a la vista */
        void addToWorld(const Position& pos, float width, float height);

        /* Mueve al objeto visual a la posicion especificada */
        void move(const Position& pos, float width, float height);

    public:
        /* Constructor */
        Viewable(WorldView& worldView);

        /* Destructor */
        virtual ~Viewable();

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

        /* Devuelve el contenedor del objeto visual */
        virtual Gtk::Widget& getWidget() = 0;

        /* Remueve al objeto visual de la vista */
        void removeFromWorld();

        /* Establece si al objeto visual se le puede hacer focus o no */
        void setFocus(bool focus);

        /* Devuelve true si el objeto visual es focuseable */
        bool hasFocus() const;
};

#endif
```

```cpp
#include "WormLifeView.h"

const std::string begining("<span color='");
const std::string middle("'><b>");
const std::string ending("</b></span>");

WormLifeView::WormLifeView(int life, const std::string& color): color(color){
    this->label.set_use_markup(true);
    this->updateLife(life);
}

WormLifeView::~WormLifeView(){}

WormLifeView::WormLifeView(WormLifeView&& other):
    label(std::move(other.label)), color(std::move(other.color)){}

void WormLifeView::updateLife(int life){
    this->label.override_background_color(Gdk::RGBA(this->color));
    this->label.set_markup(begining + "white" + middle + std::to_string(life) + e
nding);
}

Gtk::Widget& WormLifeView::getWidget(){
    return this->label;
}
```

```
#ifndef __WORMLIFEVIEW_H__
#define __WORMLIFEVIEW_H__

#include <gtkmm/label.h>

/* Clase que se encarga de controlar el label de la vida
 * del worm */
class WormLifeView{
    private:
        Gtk::Label label;
        std::string color;

    public:
        /* Constructor */
        WormLifeView(int life, const std::string& color);

        /* Destructor */
        ~WormLifeView();

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

        /* Actualiza el label de vida del worm */
        void updateLife(int life);

        /* Devuelve el contenedor de la vida */
        Gtk::Widget& getWidget();
};


#endif
```

```
#include "WormView.h"
#include <string>
#include <glibmm/main.h>
#include "ObjectSizes.h"
#include "WeaponNames.h"
#include "GamePlayers.h"

WormView::WormView(WorldView& worldView, int life, char dir, Position pos, int player_id):
    Viewable(worldView), player_id(player_id), life(life), is_moving(false),
    last_position(Position(-1, -1)), label(life, colors[player_id]),
    walkingAnimation(&this->image), weaponAnimation(DEFAULT_WEAPON, &this->image) {
        this->worm.attach(this->label.getWidget(), 0, 0, 1, 1);
        this->worm.attach(this->image, 0, 1, 1, 1);
        this->walkingAnimation.setStaticImage();
        this->addToWorld(pos, worm_size, worm_size + 0.5);
}

WormView::~WormView(){}

WormView::WormView(WormView&& other): Viewable(std::move(other)), player_id(other.player_id),
    life(other.life), is_moving(other.is_moving),
    last_position(other.last_position), label(std::move(other.label)),
    image(std::move(other.image)),
    worm(std::move(other.worm)), walkingAnimation(std::move(other.walkingAnimation)),
    weaponAnimation(std::move(other.weaponAnimation)) {
    this->weaponAnimation.updateWormImage(&this->image);
    this->walkingAnimation.updateWormImage(&this->image);
}

void WormView::updateData(int new_life, char new_dir, const Position& new_pos, bool colliding, bool is_current_worm, bool has_shot) {
    if (new_life != this->life){
        this->label.updateLife(new_life);
    }
    this->life = new_life;
    this->is_moving = !(this->last_position == new_pos);
    this->last_position = new_pos;
    this->setNewImage(new_dir, colliding, is_current_worm, has_shot);
    this->move(new_pos, worm_size, worm_size + 0.5);
}

void WormView::updateScope(int angle) {
    this->weaponAnimation.changeAngle(angle, this->getDir());
}

void WormView::changeWeapon(const std::string& weapon) {
    this->weaponAnimation.changeWeapon(weapon, this->getDir());
}

void WormView::setNewImage(char dir, bool colliding, bool is_current_worm, bool has_shot){
    if (is_current_worm){
        if (!this->is_moving && !has_shot && colliding){
            this->weaponAnimation.setWeaponImage(dir);
        } else if (colliding){
            this->walkingAnimation.setMovementImage(dir);
        }
        return;
    }
    this->walkingAnimation.setStaticImage();
}

```

```cpp
Gtk::Widget& WormView::getWidget(){
    return this->worm;
}

Gtk::Image& WormView::getImage() {
    return this->image;
}

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

char WormView::getDir() const {
    return this->walkingAnimation.getDir();
}

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

bool WormView::isMoving() const{
    return this->is_moving;
}

void WormView::setVictory() {
    this->image.set(VICTORY_ANIMATION);
}

void WormView::weaponShoot(const std::string& weapon) {
    this->weaponAnimation.weaponShootAnimation(weapon, this->getDir());
}

void WormView::resetFocus(){
    this->is_moving = false;
    this->setFocus(false);
    this->walkingAnimation.setStaticImage();
}
```

```cpp
#ifndef __WORMVIEW_H__
#define __WORMVIEW_H__

#include <gtkmm/widget.h>
#include <gtkmm/image.h>
#include <gtkmm/grid.h>
#include <gdkmm/pixbuf.h>
#include <vector>
#include "Viewable.h"
#include "WormLifeView.h"
#include "WalkingAnimation.h"
#include "WeaponAnimation.h"

#define DIR_RIGHT 1
#define DIR_LEFT -1

/* Clase que se encarga de controlar la vista de los worms */
class WormView: public Viewable {
    private:
        int player_id;
        int life;
        bool is_moving;
        Position last_position;
        WormLifeView label;
        Gtk::Image image;
        Gtk::Grid worm;
        WalkingAnimation walkingAnimation;
        WeaponAnimation weaponAnimation;

        /* Actualiza la imagen del worm a la correspondiente segun las
         * condiciones en las que se encuentra este */
        void setNewImage(char dir, bool colliding, bool is_current_worm, bool ha
s_shot);

        /* Cambia la imagen actual por la del arma actual */
        void setWeaponImage();

        /* Actualiza las imagenes de las armas */
        void updateWeaponImage();

        /* Callback */
        bool batHitCallBack(std::vector<Glib::RefPtr<Gdk::Pixbuf>>::iterator& it
er, const int width);

    public:
        /* Constructor */
        WormView(WorldView& worldView, int life, char dir, Position pos, int pla
yer_id);

        /* Destructor */
        ~WormView();

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


        /* Actualiza la posicion y vida del worm */
        void updateData(int new_life, char new_dir, const Position& new_pos, boo
l colliding, bool is_current_worm, bool has_shot);

        /* Actualiza la imagen del arma con el angulo actual */
        void updateScope(int angle);

        /* Actualiza el arma del worm y cambia la imagen */
        void changeWeapon(const std::string &weapon);

        /* Devuelve la direccion del worm */
        char getDir() const;

        /* Elimina la imagen del arma del worm */
        void removeWeaponImage();

        /* Devuelve la vida del worm */
        int getLife() const;

        /* Devuelve el id del player que controla al worm */
        int getPlayerId() const;

        /* Devuelve el contenedor donde se encuentra la vista del worm */
        Gtk::Widget& getWidget() override;

        /* Devuelve la imagen que contiene al worm */
        Gtk::Image& getImage();

        /* Cambia la imagen del worm por la animacion del worm
         * festejando la victoria */
        void setVictory();

        /* Devuelve true si el gusano se esta moviendo */
        bool isMoving() const;

        /* Realiza la animacion del disparo del arma */
        void weaponShoot(const std::string& weapon);

        /* Resetea el focus del gusano */
        void resetFocus();
};


#endif
```

```
#include "DistanceWeapon.h"

DistanceWeapon::DistanceWeapon(std::string name, int ammo, bool time) :
    Weapon(name, ammo) {
    this->has_Scope = true;
    this->is_Timed = time;
}

DistanceWeapon::~DistanceWeapon() {}

DistanceWeapon::DistanceWeapon(DistanceWeapon&& other) : Weapon(std::move(other)
){}

bool DistanceWeapon::hasVariablePower() const{
    return true;
}

```

May 27, 18 21:56 **DistanceWeapon.h** Page 1/1

```
#ifndef __CLIENTDISTANCEWEAPON_H__
#define __CLIENTDISTANCEWEAPON_H__

#include "Weapon.h"

/* Clase que se encarga de representar a las armas de distancia */
class DistanceWeapon: public Weapon{
    public:
        /* Constructor */
        DistanceWeapon(std::string name, int ammo, bool time = false);

        /* Destructor */
        ~DistanceWeapon();

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


        /* Devuelve true si el arma tiene potencia variable */
        bool hasVariablePower() const override;
};

#endif
```

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

```
#include "MeleeWeapon.h"
#include <limits>

MeleeWeapon::MeleeWeapon(std::string name, int ammo, bool scope, bool time) :
    Weapon(name, ammo) {
    this->has_Scope = scope;
    this->is_Timed = time;
}

MeleeWeapon::MeleeWeapon(MeleeWeapon&& other) : Weapon(std::move(other)) {}
```

```
#ifndef __CLIENTMELEEWEAPON_H__
#define __CLIENTMELEEWEAPON_H__

#include "Weapon.h"

/* Clase que se encarga de representar las armas de cuerpo a cuerpo */
class MeleeWeapon : public Weapon {
    public:
        /* Constructor */
        MeleeWeapon(std::string name, int ammo, bool scope, bool time = false);

        /* Destructor */
        ~MeleeWeapon() {}

        /* Constructor por movimiento */
        MeleeWeapon(MeleeWeapon&& other);
};

#endif
```

```
#include "AirAttack.h"
#include "WeaponNames.h"

AirAttack::AirAttack(int ammo) : SelfDirectedWeapon(AIR_ATTACK_NAME, ammo) {}

AirAttack::~AirAttack() {}

AirAttack::AirAttack(AirAttack&& other) : SelfDirectedWeapon(std::move(other)) {
}
```

May 27, 18 21:56 **AirAttack.h** Page 1/1

```
#ifndef __CLIENTAIRATTACK_H__
#define __CLIENTAIRATTACK_H__

#include "SelfDirectedWeapon.h"

/* Clase que representa al arma AirStrike */
class AirAttack: public SelfDirectedWeapon {
    public:
        /* Constructor */
        AirAttack(int ammo);

        /* Destructor */
        ~AirAttack();

        /* Constructor por movimiento */
        AirAttack(AirAttack&& other);
};

#endif
```

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

```
#include "Banana.h"
#include "WeaponNames.h"

Banana::Banana(int ammo) : DistanceWeapon(BANANA_NAME, ammo, true) {}

Banana::~Banana() {}

Banana::Banana(Banana&& other) : DistanceWeapon(std::move(other)) {}

```

```cpp
#ifndef __CLIENTBANANA_H__
#define __CLIENTBANANA_H__

#include "DistanceWeapon.h"

/* Clase que representa al arma Banana */
class Banana: public DistanceWeapon {
    public:
        /* Constructor */
        Banana(int ammo);

        /* Destructor */
        ~Banana();

        /* Constructor por movimiento */
        Banana(Banana&& other);
};

#endif
```

```cpp
#include "Bat.h"
#include "WeaponNames.h"

Bat::Bat(int ammo): MeleeWeapon(BAT_NAME, ammo, true) {}

Bat::~Bat() {}

Bat::Bat(Bat&& other) : MeleeWeapon(std::move(other)) {}

```

```
#ifndef __CLIENTBAT_H__
#define __CLIENTBAT_H__

#include "MeleeWeapon.h"

/* Clase que representa al arma Bat de baseball */
class Bat: public MeleeWeapon {
    public:
        /* Constructor */
        Bat(int ammo);

        /* Destructor */
        ~Bat();

        /* Constructor por movimiento */
        Bat(Bat&& other);
};

#endif
```

```
#include "Bazooka.h"
#include "WeaponNames.h"

Bazooka::Bazooka(int ammo) : DistanceWeapon(BAZOOKA_NAME, ammo) {}

Bazooka::~Bazooka(){}

Bazooka::Bazooka(Bazooka&& other) : DistanceWeapon(std::move(other)) {}

```

```cpp
#ifndef __CLIENTBAZOOKA_H__
#define __CLIENTBAZOOKA_H__

#include "DistanceWeapon.h"

/* Clase que representa al arma Bazooka */
class Bazooka: public DistanceWeapon {
    public:
        /* Constructor */
        Bazooka(int ammo);

        /* Destructor */
        ~Bazooka();

        /* Constructor por movimiento */
        Bazooka(Bazooka&& other);
};

#endif
```

```cpp
#include "Dynamite.h"
#include "WeaponNames.h"

Dynamite::Dynamite(int ammo): MeleeWeapon(DYNAMITE_NAME, ammo, false, true) {}

Dynamite::~Dynamite() {}

Dynamite::Dynamite(Dynamite&& other) : MeleeWeapon(std::move(other)) {}

```

```
#ifndef __CLIENTDYNAMITE_H__
#define __CLIENTDYNAMITE_H__

#include "MeleeWeapon.h"

/* Clase que representa al arma Dinamita */
class Dynamite: public MeleeWeapon {
    public:
        /* Constructor */
        Dynamite(int ammo);

        /* Destructor */
        ~Dynamite();

        /* Constructor por movimiento */
        Dynamite(Dynamite&& other);
};

#endif
```

```
#include "GreenGrenade.h"
#include "WeaponNames.h"

GreenGrenade::GreenGrenade(int ammo):
    DistanceWeapon(GREEN_GRENADE_NAME, ammo, true) {}

GreenGrenade::~GreenGrenade(){}

GreenGrenade::GreenGrenade(GreenGrenade&& other) : DistanceWeapon(std::move(othe
r)) {}

```

```
#ifndef __CLIENTGREENGRENADE_H__
#define __CLIENTGREENGRENADE_H__

#include "DistanceWeapon.h"

/* Clase que representa al arma Granada verde */
class GreenGrenade: public DistanceWeapon {
    public:
        /* Constructor */
        GreenGrenade(int ammo);

        /* Destructor */
        ~GreenGrenade();

        /* Constructor por movimiento */
        GreenGrenade(GreenGrenade&& other);
};

#endif
```

```
#include "HolyGrenade.h"
#include "WeaponNames.h"

HolyGrenade::HolyGrenade(int ammo) :
    DistanceWeapon(HOLY_GRENADE_NAME, ammo, true) {}

HolyGrenade::~HolyGrenade(){}

HolyGrenade::HolyGrenade(HolyGrenade&& other) : DistanceWeapon(std::move(other))
 {}
```

```
#ifndef __CLIENTHOLYGRENADE_H__
#define __CLIENTHOLYGRENADE_H__

#include "DistanceWeapon.h"

/* Clase que representa al arma Granada santa */
class HolyGrenade: public DistanceWeapon {
    public:
        /* Constructor */
        HolyGrenade(int ammo);

        /* Destructor */
        ~HolyGrenade();

        /* Constructor por movimiento */
        HolyGrenade(HolyGrenade&& other);
};

#endif
```

```
#include "Mortar.h"
#include "WeaponNames.h"

Mortar::Mortar(int ammo): DistanceWeapon(MORTAR_NAME, ammo, false) {}

Mortar::~Mortar() {}

Mortar::Mortar(Mortar&& other) : DistanceWeapon(std::move(other)) {}

```

```cpp
#ifndef __CLIENTMORTAR_H__
#define __CLIENTMORTAR_H__

#include "DistanceWeapon.h"

/* Clase que representa al arma Mortero */
class Mortar: public DistanceWeapon {
    public:
        /* Constructor */
        Mortar(int ammo);

        /* Destructor */
        ~Mortar();

        /* Constructor por movimiento */
        Mortar(Mortar&& other);
};

#endif
```

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

RedGrenade::RedGrenade(int ammo):
    DistanceWeapon(RED_GRENADE_NAME, ammo, true) {}

RedGrenade::~RedGrenade() {}

RedGrenade::RedGrenade(RedGrenade&& other) : DistanceWeapon(std::move(other)) {}

```

May 31, 18 12:08 **RedGrenade.h** Page 1/1

```
#ifndef __CLIENTREDGRENADE_H__
#define __CLIENTREDGRENADE_H__

#include "DistanceWeapon.h"

/* Clase que representa al arma Granada roja */
class RedGrenade: public DistanceWeapon {
    public:
        /* Constructor */
        RedGrenade(int ammo);

        /* Destructor */
        ~RedGrenade();

        /* Constructor por movimiento */
        RedGrenade(RedGrenade&& other);
};

#endif
```

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

```
#include "Teleportation.h"
#include "WeaponNames.h"

Teleportation::Teleportation(int ammo): SelfDirectedWeapon(TELEPORT_NAME, ammo)
{}

Teleportation::~Teleportation(){}

Teleportation::Teleportation(Teleportation&& other) : SelfDirectedWeapon(std::mo
ve(other)) {}
```

```
#ifndef __CLIENTTELEPORTATION_H__
#define __CLIENTTELEPORTATION_H__

#include "SelfDirectedWeapon.h"

/* Clase que representa al arma Teletransportador */
class Teleportation: public SelfDirectedWeapon {
    public:
        /* Constructor */
        Teleportation(int ammo);

        /* Destructor */
        ~Teleportation();

        /* Constructor por movimiento */
        Teleportation(Teleportation&& other);
};

#endif
```

```
#include "SelfDirectedWeapon.h"

SelfDirectedWeapon::SelfDirectedWeapon(std::string name, int ammo) : Weapon(name
, ammo) {}

SelfDirectedWeapon::~SelfDirectedWeapon() {}

SelfDirectedWeapon::SelfDirectedWeapon(SelfDirectedWeapon&& other) : Weapon(std:
:move(other)) {}

bool SelfDirectedWeapon::isSelfDirected() const{
    return true;
}
```

May 31, 18 12:08 **SelfDirectedWeapon.h** Page 1/1

```cpp
#ifndef __SELFDIRECTEDWEAPON_H__
#define __SELFDIRECTEDWEAPON_H__

#include "Weapon.h"

/* Clase que representa las armas teledirigidas */
class SelfDirectedWeapon: public Weapon{
    public:
        /* Constructor */
        SelfDirectedWeapon(std::string name, int ammo);

        /* Destructor */
        ~SelfDirectedWeapon();

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

        /* Devuelve true si es teledirigida */
        bool isSelfDirected() const override;
};

#endif
```

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

```cpp
#include "Weapon.h"

Weapon::Weapon(std::string name, int ammo) :
    name(name), ammo(ammo), has_Scope(false), is_Timed(false){}

Weapon::~Weapon() {}

Weapon::Weapon(Weapon&& other) {
    this->name = std::move(other.name);
    this->ammo = std::move(other.ammo);
    this->has_Scope = std::move(other.has_Scope);
    this->is_Timed = std::move(other.is_Timed);
}

Weapon& Weapon::operator=(Weapon&& other) {
    this->name = std::move(other.name);
    this->ammo = std::move(other.ammo);
    this->has_Scope = std::move(other.has_Scope);
    this->is_Timed = std::move(other.is_Timed);
    return *this;
}

bool Weapon::hasScope() const{
    return this->has_Scope;
}

bool Weapon::isSelfDirected() const{
    return false;
}

bool Weapon::isTimed() const{
    return this->is_Timed;
}

bool Weapon::hasVariablePower() const{
    return false;
}

const std::string& Weapon::getName() const{
    return this->name;
}

void Weapon::shoot() {
    if (this->ammo <= 100)
        this->ammo--;
}

bool Weapon::hasAmmo() const{
    return this->ammo > 0;
}

unsigned int Weapon::getAmmo() const{
    return this->ammo;
}

```

```cpp
#ifndef __CLIENTWEAPON_H__
#define __CLIENTWEAPON_H__

#include <string>

/* Clase que se encarga de representar a las armas del juego */
class Weapon {
    protected:
        std::string name;
        unsigned int ammo;
        bool has_Scope;
        bool is_Timed;

    public:
        /* Constructor */
        Weapon(std::string name, int ammo);

        /* Destructor */
        ~Weapon();

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

        /* Operador = por movimiento */
        Weapon& operator=(Weapon&& other);


        /* Devuelve true si el arma tiene mira */
        virtual bool hasScope() const;

        /* Devuelve true si el arma es teledirigida */
        virtual bool isSelfDirected() const;

        /* Devuelve true si el arma es por tiempo */
        virtual bool isTimed() const;

        /* Devuelve true si el arma tiene potencia variable */
        virtual bool hasVariablePower() const;

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

        /* Disminuye la cantidad de municiones del arma */
        virtual void shoot();

        /* Devuelve true si el arma tiene balas */
        virtual bool hasAmmo() const;

        /* Devuelve la cantidad de balas */
        unsigned int getAmmo() const;
};
#endif
```

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

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

WeaponList::~WeaponList() {}

void WeaponList::add(std::string weapon, int ammo) {
    WeaponsFactory factory;
    this->weapons.insert(std::pair<std::string, weapon_ptr>(weapon, std::move(fa
ctory.createWeapon(weapon, ammo))));
}

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

Weapon& WeaponList::getCurrentWeapon() {
    return *this->weapons.at(this->current_weapon);
}

WeaponList::iterator WeaponList::begin() {
    return this->weapons.begin();
}

WeaponList::iterator WeaponList::end() {
    return this->weapons.end();
}

```

```cpp
#ifndef __CLIENTWEAPONLIST_H__
#define __CLIENTWEAPONLIST_H__

#include <map>
#include "Weapon.h"
#include "WeaponsFactory.h"

/* Clase que se encarga de almacenar las armas del juego */
class WeaponList {
    private:
        typedef std::map<std::string, weapon_ptr> WeaponsList;
        WeaponsList weapons;
        std::string current_weapon;

    public:
        /* Constructor */
        WeaponList();

        /* Destructor */
        ~WeaponList();


        /* Agrega un arma a la lista */
        void add(std::string weapon, int ammo);

        /* Devuelve el arma actual */
        Weapon& getCurrentWeapon();

        /* Cambia el arma actual por la especificada */
        void changeWeapon(std::string weapon);

        typedef WeaponsList::iterator iterator;
        typedef WeaponsList::const_iterator const_iterator;
        iterator begin();
        iterator end();
};


#endif
```

```cpp
#include "WeaponsFactory.h"
#include "WeaponNames.h"

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


WeaponsFactory::WeaponsFactory() {}

WeaponsFactory::~WeaponsFactory() {}

weapon_ptr WeaponsFactory::createWeapon(std::string weapon, int ammo) {
    if (weapon == AIR_ATTACK_NAME)
        return weapon_ptr(new AirAttack(ammo));
    else if (weapon == BANANA_NAME)
        return weapon_ptr(new Banana(ammo));
    else if (weapon == BAT_NAME)
        return weapon_ptr(new Bat(ammo));
    else if (weapon == BAZOOKA_NAME)
        return weapon_ptr(new Bazooka(ammo));
    else if (weapon == DYNAMITE_NAME)
        return weapon_ptr(new Dynamite(ammo));
    else if (weapon == GREEN_GRENADE_NAME)
        return weapon_ptr(new GreenGrenade(ammo));
    else if (weapon == HOLY_GRENADE_NAME)
        return weapon_ptr(new HolyGrenade(ammo));
    else if (weapon == MORTAR_NAME)
        return weapon_ptr(new Mortar(ammo));
    else if (weapon == RED_GRENADE_NAME)
        return weapon_ptr(new RedGrenade(ammo));
    return weapon_ptr(new Teleportation(ammo));
}
```

```cpp
#ifndef __CLIENTWEAPONSFACTORY_H__
#define __CLIENTWEAPONSFACTORY_H__

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

typedef std::unique_ptr<Weapon> weapon_ptr;

/* Clase que se encarga de crear las armas del juego */
class WeaponsFactory {
    public:
        /* Constructor */
        WeaponsFactory();

        /* Destructor */
        ~WeaponsFactory();


        /* Crea el arma especificada con las municiones especificadas */
        weapon_ptr createWeapon(std::string weapon, int ammo);
};


#endif
```

Table of Contents