
SGG player_movement.h Library

Επιμέλεια: Α.Δρακόπουλος

Πίνακας περιεχομένων

Βιβλιοθήκη player_movement.h.....	3
Βιβλιοθήκη player_movement.h (τεκμηρίωση στα Αγγλικά).....	3
Βιβλιοθήκη player_movement_es.h (τεκμηρίωση στα Ισπανικά).....	7
Βιβλιοθήκη player_movement_gr.h (τεκμηρίωση στα Ελληνικά).....	11
Πρόγραμμα ελέγχου βιβλιοθήκης player_movement.h.....	15

Βιβλιοθήκη player_movement.h

Βιβλιοθήκη player_movement.h (τεκμηρίωση στα Αγγλικά)

```
#pragma once
#ifndef PLAYER_MOVEMENT_H
#define PLAYER_MOVEMENT_H

#include <chrono>
#include <thread>
#include <iostream>
#include "sgg/graphics.h"
#define M_PI 3.14159265358979323846
// Player properties and state
struct Player {
    float x;           // Player's x-axis position
    float y;           // Player's y-axis position
    float width, height; ///< Dimensions of the player (width and height).
    float speed;       // Current speed of the player
    float angle;       ///< Orientation of the player in degrees.
    bool isJumping;    // Indicates if the player is currently jumping
    bool isCrouching;  // Indicates if the player is crouching
    const float defaultHeight = 1.0f; // Default height for standing position
    const float crouchHeight = 0.5f;  // Height when crouching

    graphics::Brush frontBrush; ///< Color used for the player's front section.
    graphics::Brush backBrush;

};

/**
 * Delays keyboard response by a specified amount of time.
 * This function pauses the program's execution to simulate a "slowed" input response,
 * effectively ignoring any keyboard inputs during the delay period.
 *
 * @param delay_ms The delay duration in milliseconds before accepting keyboard input.
 *
 * @note This function can be used to limit input frequency in scenarios where
 * rapid input responses are undesired.
 */
void slowKeyboardInput(int delay_ms) {
    // Ensure the delay is positive
    if (delay_ms > 0) {
        // Put the program to sleep for the specified duration
        std::this_thread::sleep_for(std::chrono::milliseconds(delay_ms));
    }
}

/**
 * Makes the player jump along the y-axis.
 * @param player Reference to the Player struct.
 * @param jumpHeight The height of the jump.
 * This function updates the player's position along the y-axis
 * by increasing the y-coordinate to simulate a jump.
 */
void jumpY(Player& player, float jumpHeight) {
    if (!player.isJumping) {
        player.y += jumpHeight;
        player.isJumping = true;
        std::cout << "Player jumped on Y-axis to: " << player.y << std::endl;
    }
    player.isJumping = false;
}
```

```

/**
 * Makes the player jump along the x-axis.
 * @param player Reference to the Player struct.
 * @param jumpDistance The distance of the jump.
 * This function updates the player's position along the x-axis
 * by increasing the x-coordinate to simulate a jump.
 */
void jumpX(Player& player, float jumpDistance) {
    if (!player.isJumping) {
        player.x += jumpDistance;
        player.isJumping = true;
        std::cout << "Player jumped on X-axis to: " << player.x << std::endl;
    }
    player.isJumping = false;
}

/**
 * Increases the player's speed for a short duration, simulating a sprint.
 * @param player Reference to the Player struct.
 * @param sprintMultiplier Multiplier applied to the player's speed.
 * The function applies a temporary speed increase to the player,
 * giving them a boost for faster movement.
 */
void sprint(Player& player, float sprintMultiplier) {
    float originalSpeed = player.speed;
    player.speed *= sprintMultiplier;
    std::cout << "Player is sprinting at speed: " << player.speed << std::endl;

    // Reset speed to normal after a short duration (simple simulation)
    //player.speed = originalSpeed;
}

/**
 * Slows down the player's movement speed for a stealthier approach.
 * @param player Reference to the Player struct.
 * @param walkMultiplier Multiplier applied to reduce the player's speed.
 * This function applies a slower speed to the player, making movement more cautious.
 */
void walk(Player& player, float walkMultiplier) {
    float originalSpeed = player.speed;
    player.speed *= walkMultiplier;
    std::cout << "Player is walking at speed: " << player.speed << std::endl;

    // Reset speed to normal after walking
    //player.speed = originalSpeed;
}

// Movement and rotation functions

/**
 * @brief Moves the player forward in the direction they are facing.
 *
 * This function updates the player's position based on their speed and orientation.
 * @param player The player to be moved forward.
 */
void moveForward(Player& player) {
    player.x += player.speed * cos(player.angle * M_PI / 180.0f);
    player.y += player.speed * sin(player.angle * M_PI / 180.0f);
}

/**
 * @brief Moves the player backward in the opposite direction they are facing.
 *

```

```

    * This function updates the player's position to move them backward relative to their
    current orientation.
    * @param player The player to be moved backward.
    */
void moveBackward(Player& player) {
    player.x -= player.speed * cos(player.angle * M_PI / 180.0f);
    player.y -= player.speed * sin(player.angle * M_PI / 180.0f);
}

/**
 * @brief Rotates the player to the left by a specified degree.
 *
 * This function adjusts the player's orientation to the left.
 * @param player The player to be rotated.
 * @param degrees The amount in degrees to rotate the player left.
 */
void rotateLeft(Player& player, float degrees) {
    player.angle -= degrees;
    if (player.angle < 0) player.angle += 360;
}

/**
 * @brief Rotates the player to the right by a specified degree.
 *
 * This function adjusts the player's orientation to the right.
 * @param player The player to be rotated.
 * @param degrees The amount in degrees to rotate the player right.
 */
void rotateRight(Player& player, float degrees) {
    player.angle += degrees;
    if (player.angle >= 360) player.angle -= 360;
}

// Position and boundary checking functions

/**
 * @brief Checks if the player is within the boundaries of the window.
 *
 * This function verifies that the player is within the visible window area,
    considering
    * the player's dimensions and position.
    * @param player The player to check.
    * @param windowWidth The width of the window boundary.
    * @param windowHeight The height of the window boundary.
    * @return true if the player is within bounds; otherwise false.
 */
bool isWithinBounds(const Player& player, float windowWidth, float windowHeight) {
    return player.x >= player.width / 2 && player.x <= windowWidth - player.width / 2
    &&
        player.y >= player.height / 2 && player.y <= windowHeight - player.height / 2;
}

/**
 * @brief Ensures that the player stays within the window boundaries.
 *
 * This function adjusts the player's position if they exceed the window's bounds.
 * @param player The player to enforce bounds on.
 * @param windowWidth The width of the window boundary.
 * @param windowHeight The height of the window boundary.
 */
void enforceBounds(Player& player, float windowWidth, float windowHeight) {
    if (player.x < player.width / 2) player.x = player.width / 2;
    if (player.x > windowWidth - player.width / 2) player.x = windowWidth -
player.width / 2;
    if (player.y < player.height / 2) player.y = player.height / 2;

```

```

    if (player.y > windowHeight - player.height / 2) player.y = windowHeight -
player.height / 2;
}

// Graphical representation functions

/**
 * @brief Renders the player as a rectangle with a distinguishable front and back.
 *
 * The player's orientation is taken into account, with a slight offset to visually
indicate the front of the player.
 * @param player The player to render.
 */
void drawPlayer(const Player& player) {
    graphics::setOrientation(player.angle); // Set orientation to match the player's
angle

    // Define back brush color to distinguish the back
    graphics::Brush backBrush = player.backBrush;
    backBrush.fill_color[0] = 0.6f; // A darker shade for the back
    backBrush.fill_color[1] = 0.6f;
    backBrush.fill_color[2] = 0.6f;

    // Define front brush color for distinction
    graphics::Brush frontBrush = player.frontBrush;
    frontBrush.fill_color[0] = 1.0f; // A lighter shade for the front
    frontBrush.fill_color[1] = 0.8f;
    frontBrush.fill_color[2] = 0.0f;

    // Draw the back side of the player rectangle
    graphics::drawRect(player.x, player.y, player.width, player.height, backBrush);

    // Draw the front side with a slight offset for visual distinction
    float frontOffset = player.width * 0.2f;
    float frontX = player.x + frontOffset * cos(player.angle * M_PI / 180.0f);
    float frontY = player.y + frontOffset * sin(player.angle * M_PI / 180.0f);
    graphics::drawRect(frontX, frontY, player.width * 0.8f, player.height * 0.8f,
frontBrush); // Slightly smaller for emphasis

    graphics::resetPose(); // Reset orientation after drawing
}

#endif // PLAYER_MOVEMENT_H

```

Βιβλιοθήκη player_movement_es.h (τεκμηρίωση στα Ισπανικά)

```
#pragma once
#ifndef PLAYER_MOVEMENT_ES_H
#define PLAYER_MOVEMENT_ES_H

#include <chrono>
#include <thread>
#include <iostream>
#include "sgg/graphics.h"
#define M_PI 3.14159265358979323846

// Propiedades y estado del jugador
struct Player {
    float x;           // Posición del jugador en el eje x
    float y;           // Posición del jugador en el eje y
    float width, height; ///< Dimensiones del jugador (ancho y alto).
    float speed;       // Velocidad actual del jugador
    float angle;       ///< Orientación del jugador en grados.
    bool isJumping;    // Indica si el jugador está saltando actualmente
    bool isCrouching;  // Indica si el jugador está agachado
    const float defaultHeight = 1.0f; // Altura predeterminada para la posición de
    const float crouchHeight = 0.5f; // Altura cuando está agachado

    graphics::Brush frontBrush; ///< Color utilizado para la parte frontal del
    graphics::Brush backBrush;
};

/**
 * Retarda la respuesta del teclado por un tiempo especificado.
 * Esta función pausa la ejecución del programa para simular una respuesta "lenta" a
 * las entradas del teclado,
 * ignorando efectivamente cualquier entrada de teclado durante el período de demora.
 *
 * @param delay_ms Duración del retardo en milisegundos antes de aceptar la entrada
 * del teclado.
 *
 * @nota Esta función se puede usar para limitar la frecuencia de entrada en
 * escenarios donde
 * se desean respuestas de entrada rápidas.
 */
void slowKeyboardInput(int delay_ms) {
    // Asegura que el retardo sea positivo
    if (delay_ms > 0) {
        // Pone el programa en espera durante la duración especificada
        std::this_thread::sleep_for(std::chrono::milliseconds(delay_ms));
    }
}

/**
 * Hace que el jugador salte a lo largo del eje y.
 * @param player Referencia a la estructura Player.
 * @param jumpHeight La altura del salto.
 * Esta función actualiza la posición del jugador a lo largo del eje y
 * aumentando la coordenada y para simular un salto.
 */
void jumpY(Player& player, float jumpHeight) {
    if (!player.isJumping) {
        player.y += jumpHeight;
        player.isJumping = true;
    }
}
```

```

        std::cout << "El jugador saltó en el eje Y a: " << player.y << std::endl;
    }
    player.isJumping = false;
}

/**
 * Hace que el jugador salte a lo largo del eje x.
 * @param player Referencia a la estructura Player.
 * @param jumpDistance La distancia del salto.
 * Esta función actualiza la posición del jugador a lo largo del eje x
 * aumentando la coordenada x para simular un salto.
 */
void jumpX(Player& player, float jumpDistance) {
    if (!player.isJumping) {
        player.x += jumpDistance;
        player.isJumping = true;
        std::cout << "El jugador saltó en el eje X a: " << player.x << std::endl;
    }
    player.isJumping = false;
}

/**
 * Aumenta la velocidad del jugador por un corto tiempo, simulando un sprint.
 * @param player Referencia a la estructura Player.
 * @param sprintMultiplier Multiplicador aplicado a la velocidad del jugador.
 * La función aplica un aumento temporal de velocidad al jugador,
 * dándole un impulso para moverse más rápido.
 */
void sprint(Player& player, float sprintMultiplier) {
    float originalSpeed = player.speed;
    player.speed *= sprintMultiplier;
    std::cout << "El jugador está corriendo a velocidad: " << player.speed <<
std::endl;
}

/**
 * Ralentiza la velocidad de movimiento del jugador para un enfoque más sigiloso.
 * @param player Referencia a la estructura Player.
 * @param walkMultiplier Multiplicador aplicado para reducir la velocidad del jugador.
 * Esta función aplica una velocidad más lenta al jugador, haciendo que el movimiento
 * sea más cauteloso.
 */
void walk(Player& player, float walkMultiplier) {
    float originalSpeed = player.speed;
    player.speed *= walkMultiplier;
    std::cout << "El jugador está caminando a velocidad: " << player.speed <<
std::endl;
}

// Funciones de movimiento y rotación

/**
 * @brief Mueve al jugador hacia adelante en la dirección en la que está mirando.
 *
 * Esta función actualiza la posición del jugador en función de su velocidad y
 * orientación.
 * @param player El jugador que se moverá hacia adelante.
 */
void moveForward(Player& player) {
    player.x += player.speed * cos(player.angle * M_PI / 180.0f);
    player.y += player.speed * sin(player.angle * M_PI / 180.0f);
}

/**
 * @brief Mueve al jugador hacia atrás en la dirección opuesta a la que está mirando.

```



```

*
* Esta función actualiza la posición del jugador para moverlo hacia atrás en relación
con su orientación actual.
* @param player El jugador que se moverá hacia atrás.
*/
void moveBackward(Player& player) {
    player.x -= player.speed * cos(player.angle * M_PI / 180.0f);
    player.y -= player.speed * sin(player.angle * M_PI / 180.0f);
}

/**
* @brief Rota al jugador hacia la izquierda en una cantidad de grados especificada.
*
* Esta función ajusta la orientación del jugador hacia la izquierda.
* @param player El jugador que se rotará.
* @param degrees La cantidad en grados para rotar al jugador hacia la izquierda.
*/
void rotateLeft(Player& player, float degrees) {
    player.angle -= degrees;
    if (player.angle < 0) player.angle += 360;
}

/**
* @brief Rota al jugador hacia la derecha en una cantidad de grados especificada.
*
* Esta función ajusta la orientación del jugador hacia la derecha.
* @param player El jugador que se rotará.
* @param degrees La cantidad en grados para rotar al jugador hacia la derecha.
*/
void rotateRight(Player& player, float degrees) {
    player.angle += degrees;
    if (player.angle >= 360) player.angle -= 360;
}

// Funciones de verificación de posición y límites

/**
* @brief Verifica si el jugador está dentro de los límites de la ventana.
*
* Esta función verifica que el jugador esté dentro del área visible de la ventana,
considerando
* las dimensiones y posición del jugador.
* @param player El jugador a verificar.
* @param windowWidth El ancho del límite de la ventana.
* @param windowHeight La altura del límite de la ventana.
* @return true si el jugador está dentro de los límites; de lo contrario, false.
*/
bool isWithinBounds(const Player& player, float windowWidth, float windowHeight) {
    return player.x >= player.width / 2 && player.x <= windowWidth - player.width / 2
&&
        player.y >= player.height / 2 && player.y <= windowHeight - player.height / 2;
}

/**
* @brief Asegura que el jugador se mantenga dentro de los límites de la ventana.
*
* Esta función ajusta la posición del jugador si excede los límites de la ventana.
* @param player El jugador al que se le aplicarán los límites.
* @param windowWidth El ancho del límite de la ventana.
* @param windowHeight La altura del límite de la ventana.
*/
void enforceBounds(Player& player, float windowWidth, float windowHeight) {
    if (player.x < player.width / 2) player.x = player.width / 2;
    if (player.x > windowWidth - player.width / 2) player.x = windowWidth -
player.width / 2;

```

```

        if (player.y < player.height / 2) player.y = player.height / 2;
        if (player.y > windowHeight - player.height / 2) player.y = windowHeight -
player.height / 2;
    }

// Funciones de representación gráfica

/**
 * @brief Renderiza al jugador como un rectángulo con una parte frontal y trasera
distinguibiles.
 *
 * La orientación del jugador se toma en cuenta, con un ligero desplazamiento para
indicar visualmente la parte frontal del jugador.
 * @param player El jugador a renderizar.
 */
void drawPlayer(const Player& player) {
    graphics::setOrientation(player.angle); // Establece la orientación para coincidir
con el ángulo del jugador

    // Definir el color del pincel trasero para distinguir la parte trasera
    graphics::Brush backBrush = player.backBrush;
    backBrush.fill_color[0] = 0.6f; // Un tono más oscuro para la parte trasera
    backBrush.fill_color[1] = 0.6f;
    backBrush.fill_color[2] = 0.6f;

    // Definir el color del pincel frontal para distinción
    graphics::Brush frontBrush = player.frontBrush;
    frontBrush.fill_color[0] = 1.0f; // Un tono más claro para la parte frontal
    frontBrush.fill_color[1] = 0.8f;
    frontBrush.fill_color[2] = 0.0f;

    // Dibuja la parte trasera del rectángulo del jugador
    graphics::drawRect(player.x, player.y, player.width, player.height, backBrush);

    // Dibuja la parte frontal con un ligero desplazamiento para distinción visual
    float frontOffset = player.width * 0.2f;
    float frontX = player.x + frontOffset * cos(player.angle * M_PI / 180.0f);
    float frontY = player.y + frontOffset * sin(player.angle * M_PI / 180.0f);
    graphics::drawRect(frontX, frontY, player.width * 0.8f, player.height * 0.8f,
frontBrush); // Ligeramente más pequeño para énfasis

    graphics::resetPose(); // Restablece la orientación después de dibujar
}

#endif // PLAYER_MOVEMENT_ES_H

```

Βιβλιοθήκη player_movement_gr.h (τεκμηρίωση στα Ελληνικά)

```
#pragma once
#ifndef PLAYER_MOVEMENT_GR_H
#define PLAYER_MOVEMENT_GR_H

#include <chrono>
#include <thread>
#include <iostream>
#include "sgg/graphics.h"
#define M_PI 3.14159265358979323846

// Ιδιότητες και κατάσταση του παίκτη
struct Player {
    float x;           // Θέση του παίκτη στον άξονα x
    float y;           // Θέση του παίκτη στον άξονα y
    float width, height; ///< Διαστάσεις του παίκτη (πλάτος και ύψος).
    float speed;       // Τρέχουσα ταχύτητα του παίκτη
    float angle;       ///< Προσανατολισμός του παίκτη σε μοίρες.
    bool isJumping;    // Υποδεικνύει αν ο παίκτης εκτελεί άλμα
    bool isCrouching;  // Υποδεικνύει αν ο παίκτης είναι σκυμμένος
    const float defaultHeight = 1.0f; // Προεπιλεγμένο ύψος για την όρθια θέση
    const float crouchHeight = 0.5f;  // Ύψος όταν ο παίκτης είναι σκυμμένος

    graphics::Brush frontBrush; ///< Χρώμα που χρησιμοποιείται για το εμπρόσθιο τμήμα
    του παίκτη.
    graphics::Brush backBrush;
};

/**
 * Καθυστερεί την απόκριση του πληκτρολογίου για συγκεκριμένο χρονικό διάστημα.
 * Η συνάρτηση αυτή σταματά την εκτέλεση του προγράμματος για να προσομοιώσει
 * καθυστέρηση στην απόκριση του πληκτρολογίου, αγνοώντας ουσιαστικά
 * εισόδους από το πληκτρολόγιο κατά τη διάρκεια της καθυστέρησης.
 *
 * @param delay_ms Διάρκεια καθυστέρησης σε χιλιοστά του δευτερολέπτου πριν αποδεχτεί
 * εισόδους.
 *
 * @note Αυτή η συνάρτηση μπορεί να χρησιμοποιηθεί για να περιορίσει τη συχνότητα
 * εισόδων σε περιπτώσεις
 * όπου οι γρήγορες αποκρίσεις δεν είναι επιθυμητές.
 */
void slowKeyboardInput(int delay_ms) {
    if (delay_ms > 0) {
        std::this_thread::sleep_for(std::chrono::milliseconds(delay_ms));
    }
}

/**
 * Κάνει τον παίκτη να εκτελέσει άλμα στον άξονα y.
 * @param player Αναφορά στη δομή Player.
 * @param jumpHeight Το ύψος του άλματος.
 * Αυτή η συνάρτηση ενημερώνει τη θέση του παίκτη στον άξονα y
 * αυξάνοντας τη συντεταγμένη y για την προσομοίωση άλματος.
 */
void jumpY(Player& player, float jumpHeight) {
    if (!player.isJumping) {
        player.y += jumpHeight;
        player.isJumping = true;
        std::cout << "Ο παίκτης εκτέλεσε άλμα στον άξονα Y στη θέση: " << player.y <<
        std::endl;
    }
    player.isJumping = false;
}
```

```

}

/**
 * Κάνει τον παίκτη να εκτελέσει άλμα στον άξονα x.
 * @param player Αναφορά στη δομή Player.
 * @param jumpDistance Η απόσταση του άλματος.
 * Αυτή η συνάρτηση ενημερώνει τη θέση του παίκτη στον άξονα x
 * αυξάνοντας τη συντεταγμένη x για την προσομοίωση άλματος.
 */
void jumpX(Player& player, float jumpDistance) {
    if (!player.isJumping) {
        player.x += jumpDistance;
        player.isJumping = true;
        std::cout << "Ο παίκτης εκτέλεσε άλμα στον άξονα X στη θέση: " << player.x <<
std::endl;
    }
    player.isJumping = false;
}

/**
 * Αυξάνει την ταχύτητα του παίκτη για μικρό χρονικό διάστημα, προσομοιώνοντας σπριντ.
 * @param player Αναφορά στη δομή Player.
 * @param sprintMultiplier Πολλαπλασιαστής που εφαρμόζεται στην ταχύτητα του παίκτη.
 * Η συνάρτηση εφαρμόζει προσωρινή αύξηση της ταχύτητας, προσφέροντας στον παίκτη
 * ώθηση για πιο γρήγορη κίνηση.
 */
void sprint(Player& player, float sprintMultiplier) {
    player.speed *= sprintMultiplier;
    std::cout << "Ο παίκτης σπριντάρει με ταχύτητα: " << player.speed << std::endl;
}

/**
 * Μειώνει την ταχύτητα κίνησης του παίκτη για πιο αθόρυβη προσέγγιση.
 * @param player Αναφορά στη δομή Player.
 * @param walkMultiplier Πολλαπλασιαστής που εφαρμόζεται για τη μείωση της ταχύτητας
 του παίκτη.
 * Η συνάρτηση αυτή επιβραδύνει την ταχύτητα του παίκτη, επιτρέποντας προσεκτικότερη
 κίνηση.
 */
void walk(Player& player, float walkMultiplier) {
    player.speed *= walkMultiplier;
    std::cout << "Ο παίκτης περπατά με ταχύτητα: " << player.speed << std::endl;
}

// Συναρτήσεις μετακίνησης και περιστροφής

/**
 * Μετακινεί τον παίκτη προς τα εμπρός προς την κατεύθυνση που κοιτάει.
 * @param player Ο παίκτης που θα μετακινηθεί προς τα εμπρός.
 */
void moveForward(Player& player) {
    player.x += player.speed * cos(player.angle * M_PI / 180.0f);
    player.y += player.speed * sin(player.angle * M_PI / 180.0f);
}

/**
 * Μετακινεί τον παίκτη προς τα πίσω προς την αντίθετη κατεύθυνση που κοιτάει.
 * @param player Ο παίκτης που θα μετακινηθεί προς τα πίσω.
 */
void moveBackward(Player& player) {
    player.x -= player.speed * cos(player.angle * M_PI / 180.0f);
    player.y -= player.speed * sin(player.angle * M_PI / 180.0f);
}

/**

```

```

* Περιστρέφει τον παίκτη προς τα αριστερά κατά συγκεκριμένες μοίρες.
* @param player O παίκτης που θα περιστραφεί.
* @param degrees Η ποσότητα περιστροφής σε μοίρες.
*/
void rotateLeft(Player& player, float degrees) {
    player.angle -= degrees;
    if (player.angle < 0) player.angle += 360;
}

/**
* Περιστρέφει τον παίκτη προς τα δεξιά κατά συγκεκριμένες μοίρες.
* @param player O παίκτης που θα περιστραφεί.
* @param degrees Η ποσότητα περιστροφής σε μοίρες.
*/
void rotateRight(Player& player, float degrees) {
    player.angle += degrees;
    if (player.angle >= 360) player.angle -= 360;
}

// Συναρτήσεις ελέγχου θέσης και ορίων

/**
* Ελέγχει αν ο παίκτης βρίσκεται εντός των ορίων του παραθύρου.
* @param player O παίκτης που θα ελεγχθεί.
* @param windowWidth Το πλάτος του παραθύρου.
* @param windowHeight Το ύψος του παραθύρου.
* @return true αν ο παίκτης είναι εντός των ορίων, αλλιώς false.
*/
bool isWithinBounds(const Player& player, float windowWidth, float windowHeight) {
    return player.x >= player.width / 2 && player.x <= windowWidth - player.width / 2
&&
    player.y >= player.height / 2 && player.y <= windowHeight - player.height / 2;
}

/**
* Διασφαλίζει ότι ο παίκτης παραμένει εντός των ορίων του παραθύρου.
* @param player O παίκτης για τον οποίο θα εφαρμοστούν τα όρια.
* @param windowWidth Το πλάτος του παραθύρου.
* @param windowHeight Το ύψος του παραθύρου.
*/
void enforceBounds(Player& player, float windowWidth, float windowHeight) {
    if (player.x < player.width / 2) player.x = player.width / 2;
    if (player.x > windowWidth - player.width / 2) player.x = windowWidth -
player.width / 2;
    if (player.y < player.height / 2) player.y = player.height / 2;
    if (player.y > windowHeight - player.height / 2) player.y = windowHeight -
player.height / 2;
}

// Συναρτήσεις γραφικής απεικόνισης

/**
* Σχεδιάζει τον παίκτη ως ορθογώνιο με διακριτή μπροστινή και πίσω πλευρά.
* O προσανατολισμός του παίκτη λαμβάνεται υπόψη, με ελαφριά μετατόπιση
* για να διακρίνεται οπτικά η μπροστινή πλευρά.
* @param player O παίκτης που θα σχεδιαστεί.
*/
void drawPlayer(const Player& player) {
    graphics::setOrientation(player.angle); // Θέτει τον προσανατολισμό σύμφωνα με τη
γωνία του παίκτη

    // Ορισμός χρώματος για την πίσω πλευρά
    graphics::Brush backBrush = player.backBrush;
    backBrush.fill_color[0] = 0.6f;
    backBrush.fill_color[1] = 0.6f;

```

```

backBrush.fill_color[2] = 0.6f;

// Ορισμός χρώματος για την μπροστινή πλευρά
graphics::Brush frontBrush = player.frontBrush;
frontBrush.fill_color[0] = 1.0f;
frontBrush.fill_color[1] = 0.8f;
frontBrush.fill_color[2] = 0.0f;

// Σχεδίαση πίσω πλευράς
graphics::drawRect(player.x, player.y, player.width, player.height, backBrush);

// Σχεδίαση μπροστινής πλευράς με ελαφριά μετατόπιση
float frontOffset = player.width * 0.2f;
float frontX = player.x + frontOffset * cos(player.angle * M_PI / 180.0f);
float frontY = player.y + frontOffset * sin(player.angle * M_PI / 180.0f);
graphics::drawRect(frontX, frontY, player.width * 0.8f, player.height * 0.8f,
frontBrush);

    graphics::resetPose(); // Επαναφορά προσανατολισμού μετά τη σχεδίαση
}

#endif // PLAYER_MOVEMENT_GR_H

```

Πρόγραμμα ελέγχου βιβλιοθήκης player_movement.h

```
#include "Player_Movement.h"
#include "sgg/graphics.h"
#include <iostream>

// Ρυθμίσεις παραθύρου
const float WINDOW_WIDTH = 800.0f;
const float WINDOW_HEIGHT = 600.0f;

// Δημιουργία του παίκτη
Player player = {
    400.0f, 300.0f,      // Αρχική θέση
    50.0f, 30.0f,       // Διαστάσεις του παίκτη
    5.0f,               // Ταχύτητα του παίκτη
    0.0f,               // Αρχικός προσανατολισμός
    false,
    false,
    1.0f,
    0.5f,
    { {1.0f, 0.0f, 0.0f} }, // Κόκκινη μπροστινή πλευρά
    { {0.6f, 0.0f, 0.0f} }  // Σκουρότερη κόκκινη πίσω πλευρά
};

/**
 * Συνάρτηση σχεδίασης που καλείται σε κάθε καρέ.
 */
void draw() {
    // Καθαρισμός παραθύρου
    graphics::Brush bgBrush;
    bgBrush.fill_color[0] = 1.0f;
    bgBrush.fill_color[1] = 1.0f;
    bgBrush.fill_color[2] = 1.0f;
    graphics::drawRect(WINDOW_WIDTH / 2, WINDOW_HEIGHT / 2, WINDOW_WIDTH,
        WINDOW_HEIGHT, bgBrush);

    // Σχεδίαση του παίκτη
    drawPlayer(player);
}

/**
 * Συνάρτηση ενημέρωσης που καλείται περιοδικά.
 * @param ms Τα χιλιοστά του δευτερολέπτου από την τελευταία κλήση της συνάρτησης.
 */
void update(float ms) {
    slowKeyboardInput(50);
    // Movement controls
    if (graphics::getKeyState(graphics::SCANCODE_UP)) {
        moveForward(player);
        enforceBounds(player, WINDOW_WIDTH, WINDOW_HEIGHT); // Διασφάλιση ορίων
    }
    if (graphics::getKeyState(graphics::SCANCODE_DOWN)) {
        moveBackward(player);
        enforceBounds(player, WINDOW_WIDTH, WINDOW_HEIGHT); // Διασφάλιση ορίων
    }
    if (graphics::getKeyState(graphics::SCANCODE_LEFT)) {
        rotateLeft(player, 5.0f);
    }
    if (graphics::getKeyState(graphics::SCANCODE_RIGHT)) {
        rotateRight(player, 5.0f);
    }
}
```

```

// Action controls
if (graphics::getKeyState(graphics::SCANCODE_1)) {
    jumpY(player, 30.0f); // Simulate jump
    enforceBounds(player, WINDOW_WIDTH, WINDOW_HEIGHT);
}

if (graphics::getKeyState(graphics::SCANCODE_0)) {
    jumpX(player, 30.0f); // Simulate jump
    enforceBounds(player, WINDOW_WIDTH, WINDOW_HEIGHT);
}

if (graphics::getKeyState(graphics::SCANCODE_R)) {
    sprint(player, 2.0f); // Sprint with speed multiplier
    enforceBounds(player, WINDOW_WIDTH, WINDOW_HEIGHT);
}
if (graphics::getKeyState(graphics::SCANCODE_W)) {
    walk(player, 0.5f); // Walk with speed reduction
    enforceBounds(player, WINDOW_WIDTH, WINDOW_HEIGHT);
}
}

int main() {
    // Δημιουργία παραθύρου
    graphics::createWindow(WINDOW_WIDTH, WINDOW_HEIGHT, "Player Movement");

    // Αρχικοποίηση και ορισμός της συνάρτησης σχεδίασης
    graphics::setDrawFunction(draw);

    // Ορισμός της συνάρτησης ενημέρωσης
    graphics::setUpdateFunction(update);

    // Έναρξη του κύκλου μηνυμάτων
    graphics::startMessageLoop();

    return 0;
}

```