Sistemas Electrónicos Digitales

Proyecto Final:

DOOM en Mbed LPC-1768

Unai Mendoza y Jorge Ameneiro

13/05/2025

INDICE

Introducción

Este documento presenta el desarrollo del proyecto final de la asignatura **Sistemas Electrónicos Digitales** (18485), perteneciente al Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación en la **Universidad Autónoma de Madrid**. El objetivo del proyecto es aplicar los conocimientos adquiridos a lo largo del curso para diseñar e implementar un sistema digital funcional, partiendo de especificaciones concretas y utilizando componentes reales.

Como propuesta de trabajo, se ha desarrollado una versión simplificada del clásico videojuego **DOOM** (1995), adaptada a las capacidades de la **mbed LPC1768**, una plataforma basada en el microcontrolador ARM Cortex-M3. El proyecto ha sido programado íntegramente en lenguaje **C**, y hace uso intensivo de técnicas de diseño estructurado, control de periféricos y gestión eficiente de recursos hardware.

Este enfoque permite abordar aspectos clave del desarrollo embebido, como el manejo de entradas/salidas digitales, la sincronización de procesos, la representación gráfica básica y la lógica de juego, todo ello dentro de las restricciones propias del sistema.

La implementación está organizada en módulos funcionales que permiten una lectura clara del código, facilitando tanto la depuración como la futura reutilización o ampliación del sistema. Todo el código y la documentación están disponibles en el repositorio de GitHub del proyecto (https://github.com/unimen04/DOOM-SED), con el fin de promover la transparencia, la reutilización y el aprendizaje colaborativo.

Recursos de prácticas previas

(LPC1768_func.c y Buffer.c)

Este proyecto es la octava práctica de la asignatura. En ella se juntan todos los conocimientos adquiridos, al igual que el material que hemos ido desarrollando a lo largo de la asignatura. El objetivo de este apartado es explicar tanto las funciones que hemos utilizado de prácticas anteriores, las modificaciones que hemos hecho en ellas y el material que se nos ha proporcionado. Por poner en contexto, las prácticas realizadas han sido las siguientes:

1. LEDs y Joystick
2. Timer e interrupciones
3. LCD (Liquid Crystal Display)
4. ADC (Analog-Digital Converter)
5. Acelerómetro
6. PWM (Pulse Width Modulation)
7. DAC (Digital-Analog Converter)

En este proyecto, no se utiliza ni el acelerómetro, ni el PWM, ni el DAC.

El resto de las funciones a utilizar se encuentran dentro de LPC1768_func.c y Buffer.c (y sus respectivos .h). Además se cuenta con SPI_LCD.c y Delay.c, que han sido provistos por el profesor.

Los códigos están divididos en diferentes secciones que se explicarán detalladamente:

LPC1768_func.c

JOYSTICK

- **`void initPin_JOY(uint32_t port_number, uint32_t pin_number)`**
  Configura un pin como entrada digital con resistencia de pulldown, dados un puerto y un pin, ya que dependiendo del puerto un pin puede tener diferentes funciones.
- **`void read_center()`**
  Lee el botón central del joystick y lo registra si se ha presionado, guardando su estado en center_registered.
- **`void read_joy()`**
  Lee el resto de los botones del joystick y actualiza sus respectivas flags.

LEDS

- **`void initPin_LED(uint32_t port_number, uint32_t pin_number)`**
  Misma función que initPin_JOY().
- **`void display_leds(uint8_t led1, uint8_t led2, uint8_t led3, uint8_t led4)`**
  Enciende o apaga los leds que se le pasen como parámetro.

RGB

- **`void initPin_RGB(uint32_t port_number, uint32_t pin_number)`**
  Igual que initPin_LED().
- **`void display_rgb(uint8_t red, uint8_t green, uint8_t blue)`**
  Enciende o apaga los diferentes colores del RGB, dependiendo de los parámetros recibidos .

TIMER

- **`void init_timer(int maxCount)`**
  Configura el timer: comienza habilitando las interrupciones. Después, reinicia el contador si este ha llegado a su valor máximo. Continúa asignando un valor al prescaler (0 en nuestro caso), y después, comprueba si el timer ha alcanzado el máximo. Una vez hecho esto, activa las interrupicones y, finalmente, activa el contador.

ADC

- **`void init_ADC(void)`**
  Configura el ADC: lo primero, se activa. Después, configura ambos potenciómetros (aunque solo utilizamos uno) con función 3, es decir, con el ADC. Por último, se reinician los valores de SEL y de CLKDIV.
- **`uint32_t read_ADC(uint8_t canal, uint8_t canal_previo)`**
  Es la encargada de leer el valor de un canal del ADC y transformarlo, devolviendo este valor transformado.
- **`uint8_t read_pot(void)`**
  La función devuelve 1 si el valor del potenciómetro ha sido modificado, y 0 si no.

START

Por último, en esta sección del código se llama a todas las funciones anteriores (con sus parámetros previamente definidos) para inicializar correctamente la placa con todo lo necesario para funcionar.

Buffer.c

- **`static const uint8_t digitos[10][FONT_SIZE]`**
  Contiene los dígitos del 0 al 9. Estos números son de 4x5 píxeles.
- **`static const uint8_t letras [NUM_LETRAS_ABECEDARIO+1][FONT_SIZE]`**
  Contiene las letras (mayúsculas) del alfabeto (exceptuando la Ñ) y el carácter de espacio. Estas letras son de 5x5 píxeles.
- **`void ZBuffer_Reset(uint8_t *zbuffer)`**
  Reinicia el ZBuffer poniéndolo a su valor máximo, 255.
- **`void Buffer_Reset(uint8_t* buffer)`**
  Reinicia el buffer poniéndolo a 0.
- **`void Buffer_SetPixel(uint8_t* buffer, uint8_t x, uint8_t y)`**
  Enciende un píxel de la pantalla dadas sus coordenadas en la misma.
- **`void Buffer_ClearPixel(uint8_t* buffer, uint8_t x, uint8_t y)`**
  Apaga un píxel de la pantalla dadas sus coordenadas en la misma.
- **`void Buffer_DrawLineH(uint8_t* buffer, uint8_t xi, uint8_t xf, uint8_t y)`**
  Dibuja una línea horizontal desde (xi, y) hasta (xf, y).
- **`void Buffer_DrawLineV(uint8_t* buffer, uint8_t x, uint8_t yi, uint8_t yf)`**
  Dibuja una línea vertical desde (x, yi) hasta (x, yf).
- **`void Buffer_DrawLine(uint8_t* buffer, uint8_t x1, uint8_t y1, uint8_t x2, uint8_t y2)`**
  Dibuja una línea cualquiera desde (x1, y1) hasta (x2, y2). Utiliza el algoritmo de Bresenham. Es más lento que Buffer_DrawLineV() y Buffer_DrawLineH().
- **`void Buffer_DrawRect(uint8_t* buffer, uint8_t x, uint8_t y, uint8_t width, uint8_t height)`**
  Dibuja un rectángulo sólido en (x,y) (su esquina superior izquierda), de tamaño width x height.
- **`void Buffer_DrawBRect(uint8_t* buffer, uint8_t x, uint8_t y, uint8_t width, uint8_t height)`**
  Igual que la anterior, pero sin ser sólido.
- **`void Buffer_DrawBRect(uint8_t* buffer, uint8_t x, uint8_t y, uint8_t width, uint8_t height)`**
  Igual que la anterior, pero sin ser sólido.

- **`void Buffer_DrawCircle(uint8_t* buffer, uint8_t xc, uint8_t yc, uint8_t radius)`**
  Dibuja un círculo utilizando el algoritmo de Bresenham.
- **`void Buffer_DrawDigit(uint8_t* buffer, uint8_t digito, uint8_t x, uint8_t y)`**
  Dibuja un dígito pasado como parámetro en (x,y).
- **`void Buffer_DrawNum(uint8_t* buffer, int num, uint8_t x, uint8_t y)`**
  Dibuja un numero de uno o varios dígitos pasado como parámetro empezando en (x,y), separados entre sí por un offset.
- **`void Buffer_DrawLetter(uint8_t* buffer, char letter, uint8_t x, uint8_t y)`**
  Igual que Buffer_DrawDigit() pero con letras.
- **`void Buffer_DrawWord(uint8_t* buffer, char *texto, uint8_t x, uint8_t y`**
  Igual que Buffer_DrawNum pero con palabras.
- **`void Buffer_DrawMenu(uint8_t *buffer, const uint8_t *sprite, uint8_t width, uint8_t height, uint8_t x, uint8_t y)`**
  Dibuja el menú inicial del juego. También se utiliza para dibujar la pistola y su animación. Hemos reutilizado el código ya que tanto el menú como la pistola (estática y disparando) han sido diseñadas por nosotros utilizando la web https://javl.github.io/image2cpp/.
  La web recibe una imagen y te devuelve el array de bits(en hexadecimal de 8 bits), donde cada byte representa 8 bits de la imagen en horizontal. Por defecto, entiende los bits blancos de la imagen como 1, pero nosotros lo necesitamos al revés, por lo que cambiamos *Background color* a *Black*, e invertimos los colores de la imagen (en la propia web). Además, cambiamos *Scaling* a *scale to fit, keeping proportions*, además de centrar la imagen tanto horizontal como verticalmente.
- **`void Buffer_DrawSprite(uint8_t *buffer, uint8_t *zbuffer, const uint8_t *bitmap, const uint8_t *mask, uint8_t width, uint8_t height, uint8_t x, uint8_t y, float distance)`**
  Dibuja los diferentes sprites del juego. Estos sprites no los hemos diseñado nosotros, los hemos obtenido de https://github.com/potat-dev/milandr-doom, quien utiliza un microcontrolador Milandr 1986VE91T con una pantalla de 128x64 pixeles, por lo que esta función es capaz de modificar su tamaño para ajustarlo a la nuestra, además de escalar el sprite dependiendo de la distancia.

A pesar de ser códigos de otras prácticas, estos han sido modificados con nuevas funciones necesarias en el juego.

structure.c

Este código contiene todo lo necesario para calcular posiciones y colisiones. En un primer momento, a la hora de desarrollar el raycasting (del que hablaremos más tarde), el videojuego tenía una vista cenital, en 2D. Hemos decidido dejar las funciones que se utilizaron a pesar de que son inútiles y no afectan a la lógica del videojuego en 3D, pero pueden ser de utilidad a la hora de corregir errores.

Por este motivo, el mapa inicialmente venía limitado por el tamaño de la pantalla: 128x32 píxeles (aunque este era realmente de 32x8 píxeles, ya que está formado por un sistema de celdas que explicaremos a continuación). En 3D, este mapa puede ser de tamaño mayor. La definición del mapa viene dada por:

- **`uint8_t map[MAP_HEIGHT][MAP_WIDTH]`**
  Este array está formado por los siguientes números, cada cual representa una entidad diferente:
  - 1 -> PARED
  - 2 -> ENEMIGO
  - 3 -> SPAWNPOINT DEL JUGADOR
  - 4 -> BOTIQUÍN

  Cada uno de estos números representa una CELDA.

Fue necesario implementar un sistema de celdas para el correcto funcionamiento del raycasting, ya que este necesita, para funcionar correctamente, coordenadas decimales para mayor precisión. No tiene sentido estar en un punto decimal en una pantalla de píxeles, ya que estos toman números enteros. Para solucionar este problema, se crean las celdas: unidades de 4x4 píxeles que forman el mapa, quedando este de 32x8 celdas. Las esquinas de las celdas toman valores enteros (por ejemplo, la celda superior izquierda es la celda 0, cuya esquina superior izquierda toma las coordenadas (0,0)), pero la ventaja de este sistema es que puedes estar en medio de la celda con un valor decimal. El funcionamiento del motor raycaster se explicará más adelante.

A continuación, se detalla la función de las diferentes funciones:

- **`coords set_coords(float x, float y)`**
  Dadas unas posiciones (x,y) , las añade a una estructura.
- **`cell obtainCell(coords p1)`**
  Dadas las coordenadas de un objeto, devuelve la celda en la que se encuentra.
- **`coords obtainCoords(cell cellPos)`**
  Funciona a la inversa que la función anterior: dada la celda en la que se encuentra un objeto, devuelve sus coordenadas en la pantalla.
- **`float distance2(coords p1, coords p2)`**
  Dados dos puntos, devuelve la distancia entre ellos.
- **`uint8_t colissionCell(cell cellCheck)`**
  Esta función se utiliza para comprobar si la celda está OCUPADA (y que es lo que la ocupa, ya que devuelve el número que contiene dicha celda, siendo esta otra gran ventaja de utilizar el sistema de celdas junto a la forma que hemos utilizado para crear el mapa). Con ella podemos detectar las colisiones. Además, si las coordenadas de cellCheck son superiores al tamaño establecido del mapa, automáticamente se detecta como colisión con una pared (para evitar que el jugador pueda salir del mapa).
- **`uint8_t colission(coords pos)`**
  Esta función hace uso de la anterior para, dadas las coordenadas del jugador, detectar si ha sufrido una colisión, y con qué.

También están definidas **`uint8_t min(uint8_t a, uint8_t b)`** y **`uint8_t max(uint8_t a, uint8_t b)`**, que devuelven, respectivamente, el mínimo y el máximo entre dos valores dados, y por último **`posLCD coords2LCD(coords pos)`**, que no es utilizada en la representación 3D, pero sirve para, dadas las coordenadas del jugador en el mapa (que recordemos, pueden ser decimales), devolver su posición en la pantalla LCD (números enteros).

entity.c

Este código contiene la creación, lógica y gestión de sprites de todas las entidades del juego. Para optimizar el juego, se ha limitado a un número máximo de entidades (15 de cada entidad en nuestro caso).

JUGADOR

El jugador se inicializa con: **`player set_player(float x, float y)`**. Tiene una vida de 100PS (puntos de salud). Este inicia en las posiciones definidas en el mapa, con un ángulo inicial de 0º. Además, tiene un timer para evitar ataques continuos (cooldown). El ataque del jugador quita 25PS a los enemigos.

El movimiento es gestionado por la función **`void move_player()`**. El movimiento depende de la entrada del joystick, y el ángulo de visión del potenciómetro. Es decir, si se pulsa el botón UP, irá en dirección al ángulo de mirada. En 2D ocurre lo mismo. El movimiento solo se actualiza si no se detecta una colisión.

La salud del jugador se representa con los 4 LEDs integrados en la placa (utilizando la función **`void display_health()`**). Cada uno representa 25PS. Cuando están todos apagados, el jugador ha muerto.

El estado del jugador se actualiza con la función **`void update_player(void)`**. Esta se encarga de actualizar la posición, el ángulo de mirada y la salud del jugador.

ENTIDADES: Estructuras

Las entidades que tiene el videojuego (al menos, hasta el momento) son: enemigos (enemy), botiquines (healthpack) y bolas de fuego (fireballs).

Al igual que los jugadores, tanto los enemigos como los botiquines se inicializan en las posiciones definidas por el mapa (**`enemy_t set_enemy(float x, float y)`** y **`healthpack_t set_healthpack(float x, float y)`**). Los enemigos tienen 100PS, sus ataques hacen 25PS al jugador y también tienen un tiempo de cooldown para evitar ataques continuos. Se inicializan en su estado “IDLE”, y si el número de enemigos es superior al número máximo de entidades, el contador de enemigos se reinicia. Los botiquines funcionan de manera similar, solo que sin estado y con 1PS para ser eliminados cuando el jugador los toque.

Las bolas de fuego son el ataque de los enemigos (son definidas con **`fireball_t set_fireball(coords pos)`**). Al igual que los botiquines, tienen 1PS. Estos, en el momento que el enemigo los lanza, fijan su dirección a la posición en la que se encontraba el jugador en el momento en el que se lanzaron, y se mueven con una velocidad previamente definida.

Tanto la velocidad del jugador como de los enemigos y bolas de fuego pueden ser cambiadas en su definición en **`entity.h`**.

ENEMIGO

- **`void move_enemy(enemy_t *e)`**
  Es la encargada del movimiento del enemigo. Su dirección depende de la posición del jugador, y solo se actualiza si no ha sufrido una colisión.
- **`void update_enemy(enemy_t *e)`**
  Esta función actualiza el comportamiento de un enemigo según su estado actual, la distancia al jugador y ciertos timers. El enemigo puede estar inactivo, en movimiento, atacando a distancia o cuerpo a cuerpo (melé), herido o muerto. Más adelante se adjunta la maquina de estados del mismo. Los estados son los siguientes:
  - ENEMY_STATE_INACTIVE
    Si el enemigo está inactivo (despawneado), no hace nada y sale de la función.
  - ENEMY_STATE_IDLE
    Si el enemigo está en reposo, se actualiza su sprite al de reposo y se mantiene en este estado hasta que el jugador entra en su zona de detección, que cambia a ENEMY_STATE_MOVING.
  - ENEMY_STATE_MOVING
    Este estado indica que el enemigo se está moviendo hacia el jugador. El sprite alterna entre dos diferentes para dar sensación de movimiento. Si el jugador sale de la zona de detección, vuelve a ENEMY_STATE_IDLE. Si el jugador está lo suficientemente cerca, dentro del radio de ataque a distancia, cambia el estado a ENEMY_STATE_RANGED_ATTACKING y reinicia el timer.
  - ENEMY_STATE_RANGED_ATTACKING
    Si el jugador se aleja, vuelve a ENEMY_STATE_MOVING. Si se acerca aún más y entra en el rango cuerpo a cuerpo, el estado pasa a ENEMY_STATE_ATTACKING. Si este se mantiene en el rango de ataque a distancia y el timer ha llegado a 0, lanza una bola de fuego y reinicia el timer. El sprite se actualiza cuando ataca.
  - ENEMY_STATE_ATTACKING
    Si el jugador se aleja, vuelve a ENEMY_STATE_RANGED_ATTACKING. Si el temporizador llega a 0, ataca al jugador (-25PS) y reinicia el timer. El sprite también se actualiza.
  - ENEMY_STATE_HIT
    Por otro lado, si el enemigo recibe daño, actualiza el sprite para mostrar que ha sido golpeado. Si su vida llega a 0, cambia el estado a ENEMY_STATE_DEAD y actualiza el contador de muertes, y si aún tiene vida y el temporizador ha llegado a 0, vuelve a ENEMY_STATE_IDLE.
  - ENEMY_STATE_DEAD
    Por último, si el enemigo muere, espera a que el temporizador llegue a 0 y pasa a ENEMY_STATE_INACTIVE, eliminando al enemigo.

BOTIQUÍN

La gestión del botiquín viene dada por **`void updateHealthpack(healthpack_t *h)`**. Si el botiquín ha sido recogido, no hace nada. Si no, se actualiza la posición respecto al jugador. Si el jugador entra en el rango de detección del botiquín, le sube la vida (50PS) y se marca como recogido (su salud pasa a 0PS).

BOLA DE FUEGO

La bola de fuego, como ya se ha mencionado, se mueve hacia la dirección en la que estaba el jugador en el momento que se lanzó (con el uso de **`void move_fireball(fireball_t *f)`**). En el momento que entra en la zona de colisión del jugador, le resta vida (-25PS) y se pone su propia vida a 0PS para destruirse. Si el jugador logra esquivarla, seguirá su recorrido hasta que colisione con una pared, donde su vida se pondrá a 0PS. Su estado es controlado por **`void update_fireball(fireball_t *f)`**.

Por último, las funciones **`void spawn_entities(void)`** y **`void update_entities(void)`** se encargan, respectivamente, de hacer aparecer las distintas entidades leyendo el array del mapa, y actualizarlas recorriendo un bucle independiente para cada una.

DOOM.c

DOOM.c contiene las funciones principales para el funcionamiento del programa. Incluye los bucles principales del juego, el renderizado del mapa usando raycsting y los sprites de las entidades y la función de disparar para poder restarle vida a los enemigos.

A continuación, se explican las funciones básicas de este archivo que sirven para controlar el flujo del juego :

- **`void loop(void)`**
  Esta función se encarga de controlar si el juego está en el menú o en el gameplay mediante una variable global: scene. Esta será puesta a 1 cuando se quiera iniciar el juego. Es la única función disponible en `DOOM.h` y será llamada desde `main.c` dentro de un bucle infinito.
- **`void loop_menu(void)`**
  Muestra el menú inicial y espera hasta que el joystick central esté pulsado. Cuando eso ocurra, pondrá el valor de scene a 1, dando asi inicio al gameplay.
- **`void loop_gameplay(void)`**
  Actualiza todos los eventos del juego (excepto el joystick, el cual se lee siempre) cada vez que llega una interrupción. Controla las funciones de disparar, actualizar las entidades y renderizar todo lo que se muestra por pantalla. Para disparar, se emplea un timer para el jugador que decrece en cada interrupción, implementando así un "cooldown" en el ataque del jugador (para evitar disparos continuos). También maneja los sprites del arma: si el timer es 0 y no se ha disparado se pone el sprite 1 (arma normal), y si el timer es 0 y si se ha disparado, se pone el sprite 2 (arma disparando) y se llama la función `shoot`.
- **`void shoot(void)`**
  Utiliza la función `translateIntoView` para obtener las coordenadas relativas de cada enemigo al que se apunta con respecto al jugador, y calcula el ángulo de diferencia. Si el ángulo entra dentro de un rango definido en la variable ANLGE_TOLERANCE, el disparo se efectúa y se le restará vida al enemigo.
- **`renderEntities(void)`**
  Implementa 3 bucles (uno para enemigos, otro para botiquines y otro para fireballs) que renderizan el sprite de la entidad si este está dentro de tu campo de visión, lo cual se sabe viendo la posición relativa usando la función `translateIntoView`.
- **`coords translateIntoView(coords entityPos)`**
  Esta función obtiene la posición relativa de una entidad al jugador utilizando una matriz de transformación. Para más información, ver https://lodev.org/cgtutor/raycasting3.html
- **`void drawImpSprite(uint8_t spriteIndex, uint8_t x, uint8_t y, float distance)`**
  Esta función simplifica el dibujar el sprite de los enemigos. Estos tienen 4 sprites diferentes a los cuales se accede utilizando una máscara. Esta función calcula dicha máscara dado el número de sprite que se quiere usar.

La función **`renderMap(void)`** es la más compleja de todas y el corazón del programa. Es un motor raycasting como el utilizado en el Wolfesetin 3D (1992). El motor raycaster fue el primer motor 3D a nivel comercial utilizado en la industria. La técnica se basa en falsear el 3D dado un mapa 2D isométrico lanzando rayos desde la posición del jugador y comprobando donde se chocan esos rayos con las paredes, para después calcular la altura de la pared basándose en la distancia del jugador a dicha pared. No es el motor que se empleó para programar el DOOM originalmente, ya que el DOOM Engine utiliza BSP (Binary Space Partitioning) para generar una vista pseudo 3D similar al raycaster, pero con numerosas ventajas, como la capacidad de poner diferentes alturas, paredes que no necesitan formar ángulos de 90 grados (para hacer, por ejemplo, rampas), iluminación dinámica o sprites orientados. En nuestro caso, se ha empleado un motor raycaster dadas las limitaciones de la placa LPC 1768, que no podría soportar la complejidad del DOOM Engine.

Para trazar los rayos se emplea el algoritmo DDA (Digitial Differential Analyzer) explicado a continuación:

Este algoritmo tiene la peculiaridad de necesitar un espacio dividido en celdas unitarias (explicado en la Página 10 de esta memoria), en la que se podrá marcar como '1' una celda para obtener una pared.

Primero necesitamos obtener el vector unitario del rayo, para ello fijamos un vector de dirección v=(cos(α), sen(α)) que será la dirección del rayo, una vez obtenido la dirección del rayo queremos saber el avance unitario en esa dirección para ello:

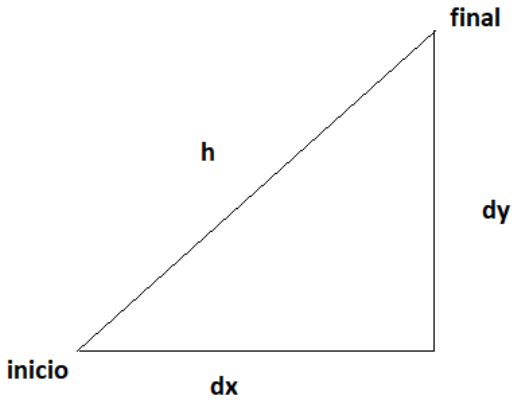

Figura 1: Obtención vector unitario del rayo (1/2)

Dado el triángulo rectángulo donde inicio es la posición del jugador y final es una posición arbitraria en la dirección del vector v hay poca gente que no sabe que: $h = \sqrt{dx^2 + dy^2}$ y que la recta que une inicio con final tiene como pendiente $m = \frac{dy}{dx}$. Si sustituimos $dx = 1$ el triángulo quedara así:

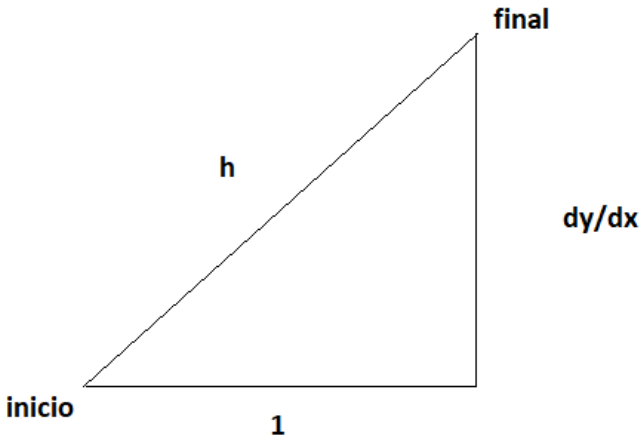

Figura 2: Obtención vector unitario del rayo (2/2)

Por lo tanto sustituyendo los nuevos valores en Pitágoras obtenemos $Sx = \sqrt{1 + (\frac{dy}{dx})^2}$, utilizando un proceso análogo para $Sy = 1$ obtenemos $Sy = \sqrt{1 + (\frac{dx}{dy})^2}$ siendo este el vector S que definirá el avance unitario de nuestro rayo.

Una vez hallado este vector, se puede comenzar con el algoritmo. Los primeros pasos serán inicializar el rayo en la posición del jugador y calcular el vector S. Los siguientes pasos para el algoritmo son:

1. Escoger el rayo más corto (en posición x o y) y avanzar esa coordenada usando el vector S. En la primera iteración no importa que rayo se avance primero.
2. Comprobar si la celda inspeccionada está ocupada. Si lo esta se acaba el algoritmo y se devuelve la distancia obtenida.
3. Si no lo está, pasar a la siguiente celda.
4. Repetir hasta terminar.

Como se puede ver el algoritmo es sencillo y eficiente, requiriendo pocas comprobaciones por cada rayo lanzado.

A continuación se ilustran varios pasos de este algoritmo.

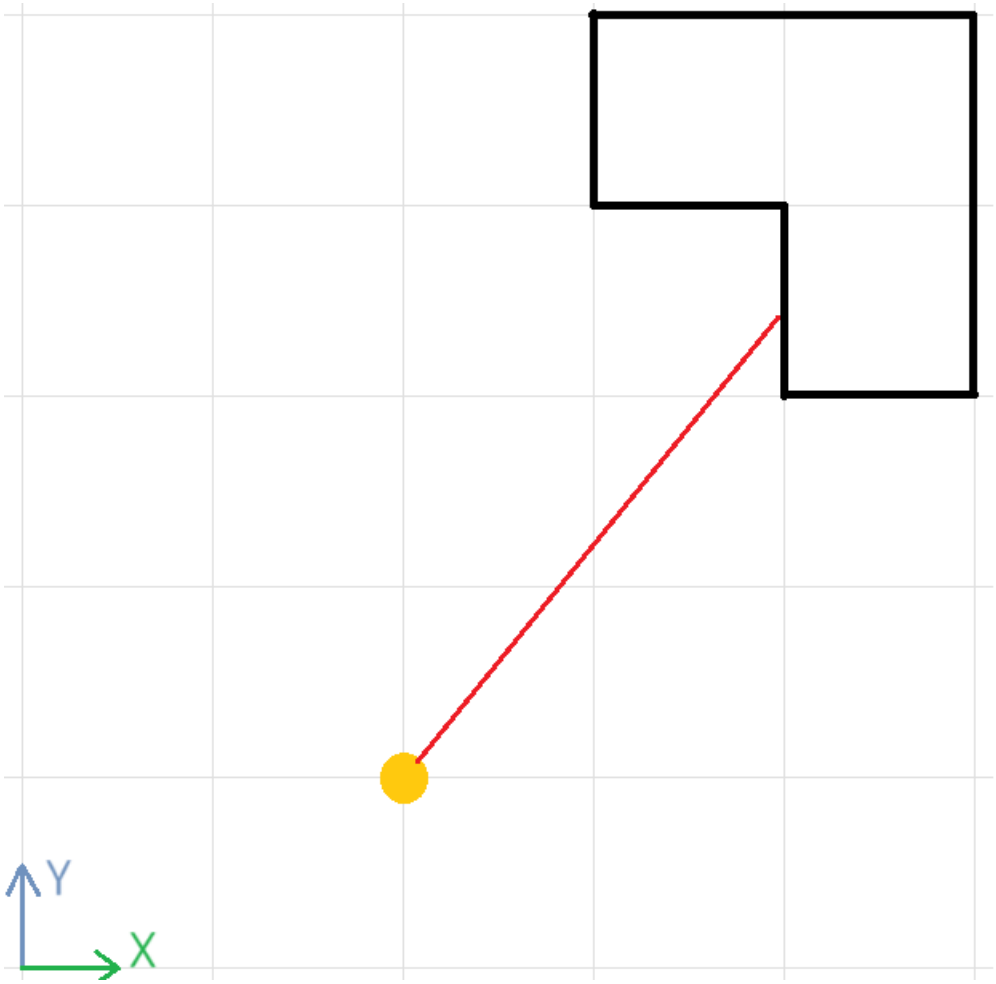

Figura 3: Explicación algoritmo raycasting (1/5)

Aquí podemos ver la inicialización del algoritmo. El punto amarillo es el inicio (donde se encuentra el jugador) y la línea roja es el resultado esperado.

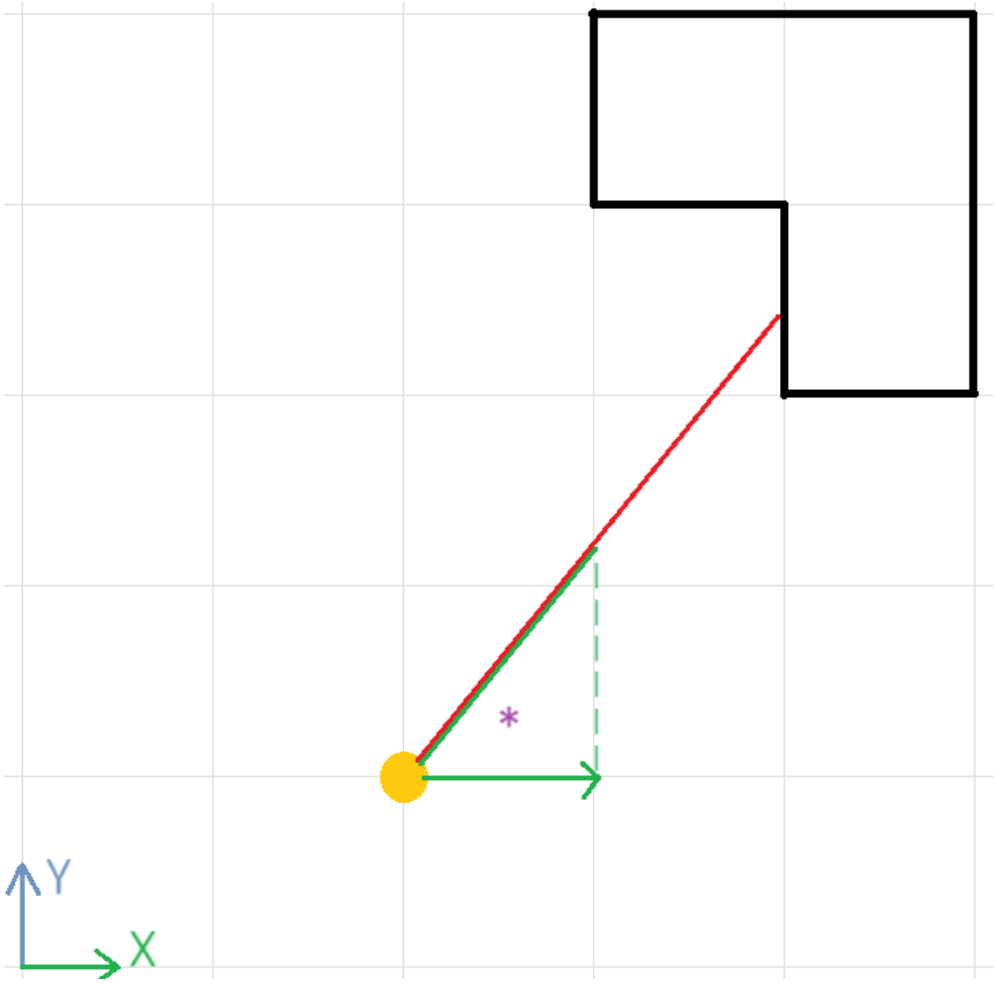

Figura 4: Explicación algoritmo raycasting (2/5)

Se avanza el rayo en el sentido de las X (avanzar primero el eje y también seria valido para la primera iteración). El asterisco (*) indica la celda que se está inspeccionando, que está vacía por lo que el algoritmo continua. Como el rayo en el eje Y es más corto (todavía no hay, tiene tamaño 0) se avanza dicho rayo.

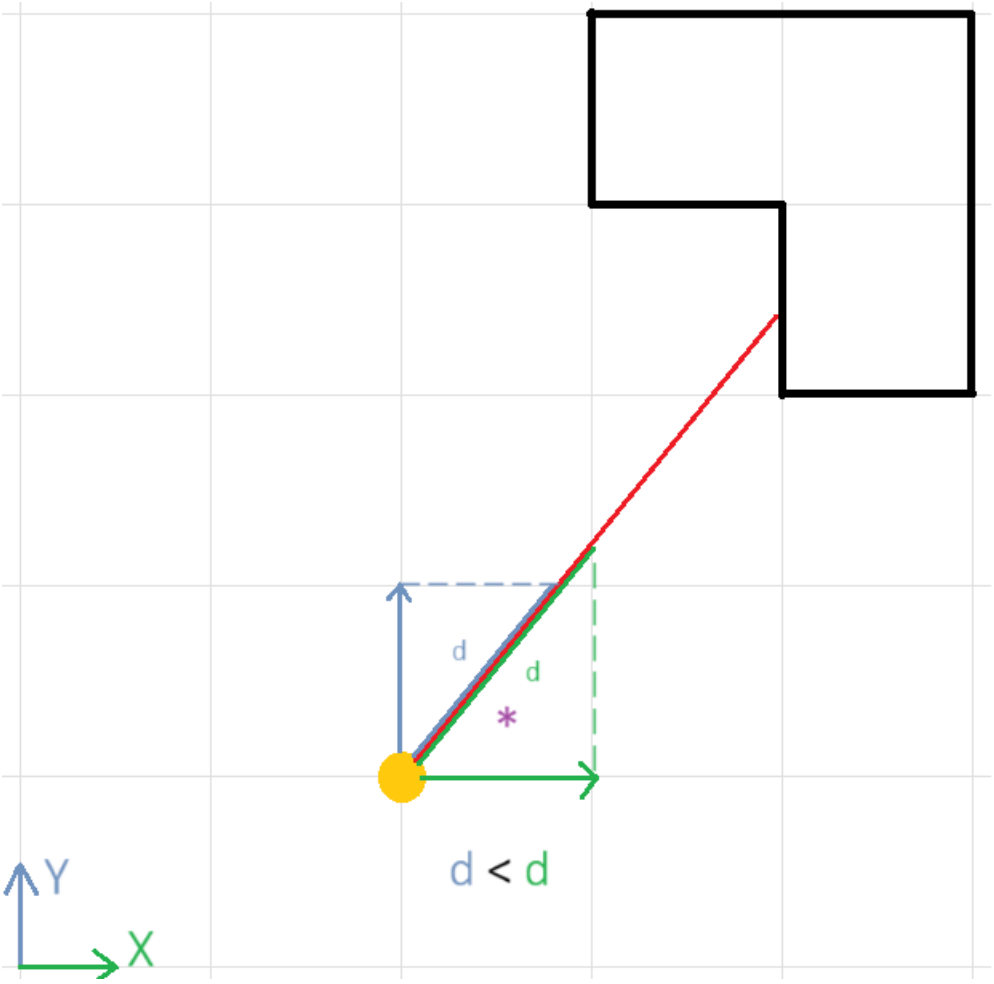

Figura 5: Explicación algoritmo raycasting (3/5)

El rayo en Y sigue siendo más corto, ya que, avanzando por la componente Y de nuestra dirección (línea roja) recorremos menos distancia que avanzando por la componente X, por lo que se sigue avanzando en Y.

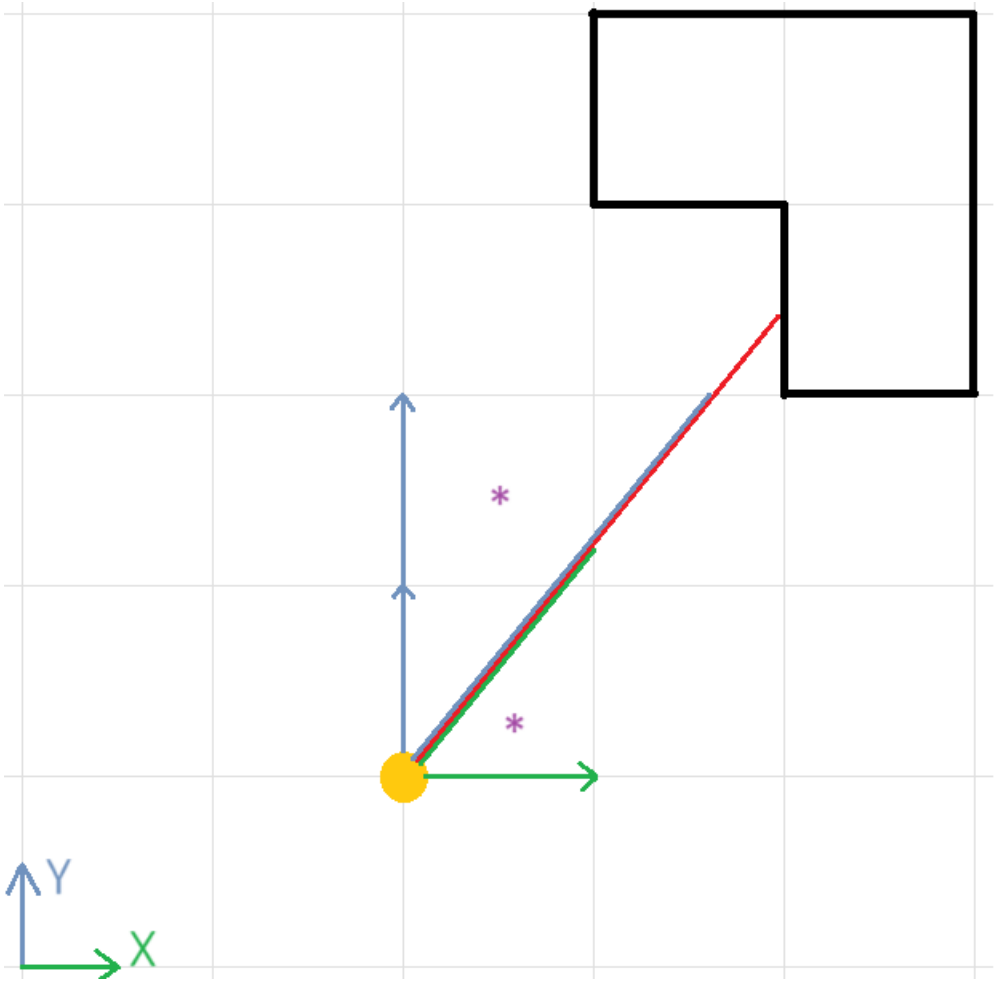

Figura 6: Explicación algoritmo raycasting (4/5)

Se comprueba que la celda que estamos inspeccionando ahora está vacía y se avanza en x.

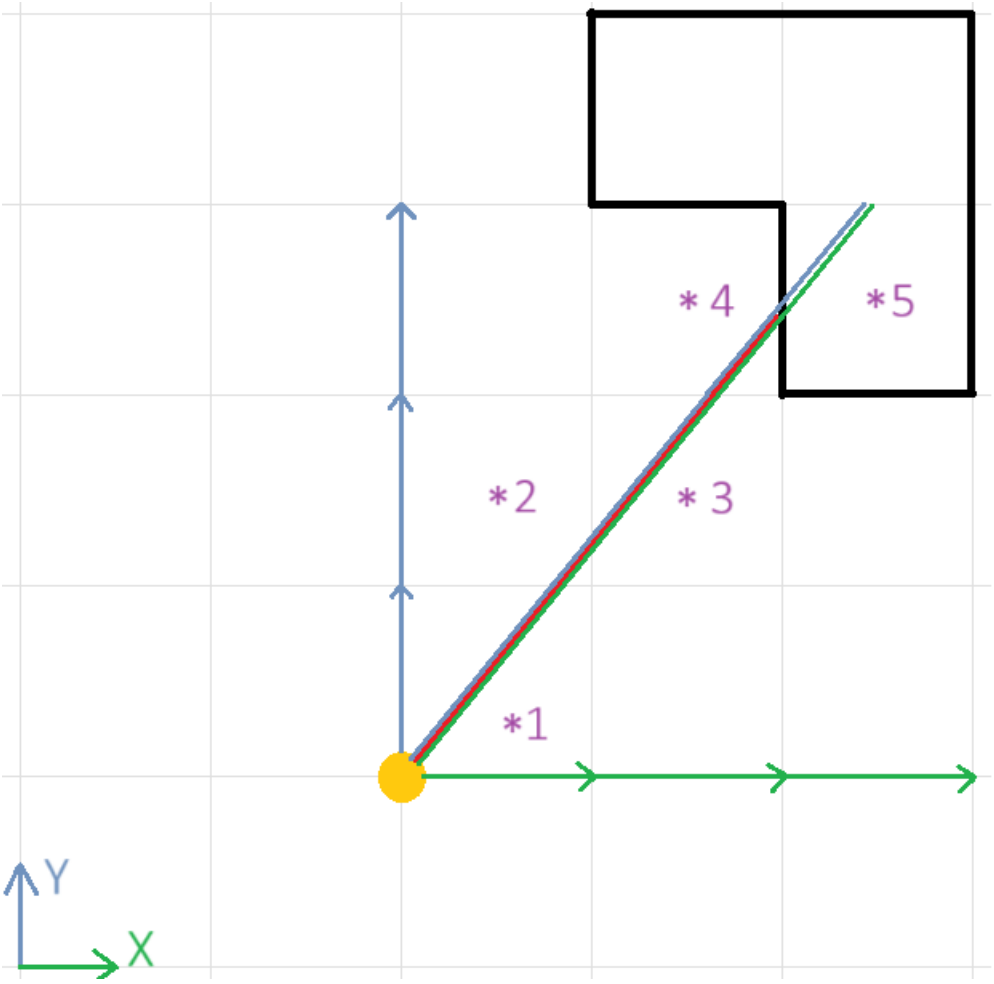

Figura 7: Explicación algoritmo raycasting (5/5)

Continuando con el algoritmo se puede ver que cuando se inspeccione la celda numero 5, la función de colisión devolverá un 1, poniendo fin al algoritmo.

Ahora se obtiene la distancia desde el jugador hasta el punto de colisión, y esta distancia hay que transformarla a la altura de la pared que se va a dibujar:

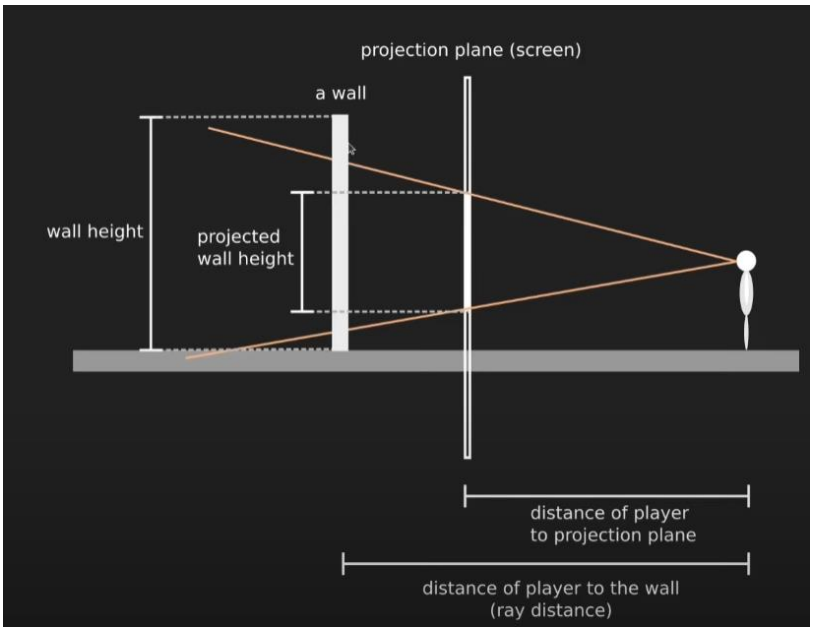

Figura 8: Cómo se obtiene la altura de la pared a dibujar

Para eso es necesario calcular la altura de la pared proyectada dada la altura de la pared (32 pixeles en nuestro caso, ya que esta es la altura de la pantalla LCD) y la distancia del jugador a la pared que se ha obtenido en el algoritmo. Solo queda hallar la distancia al plano de proyección. Para ello se aprovecha la similitud de triángulos establecida entre la distancia a la pared y la distancia a la pared proyectada. Se establece que:

$$\frac{altura\ de\ la\ pared\ proyectada}{distancia\ al\ plano\ de\ proyeccion} = \frac{altura\ de\ la\ pared}{distancia\ a\ la\ pared}$$

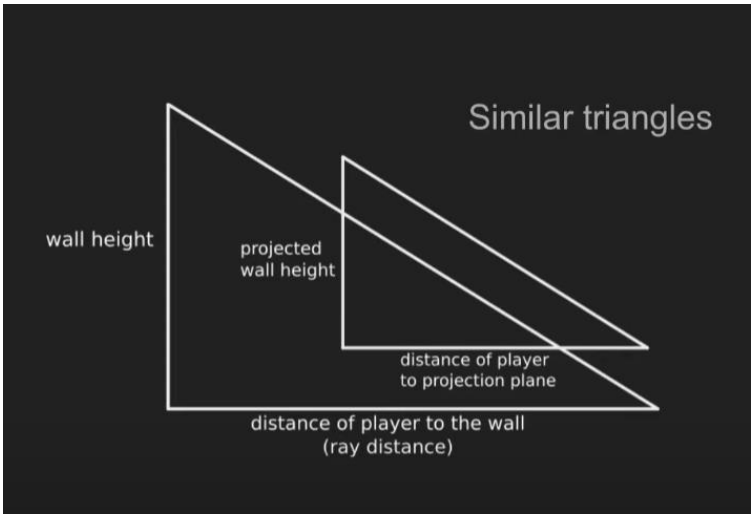

Figura 9: Similitud de triángulos para hallar la distancia al plano de proyección

Para calcular la distancia al plano de proyección se considera la mitad del ancho de la pantalla, 64 pixeles en este caso y la mitad del FOV (Field Of Vision), 32º. Se utiliza la siguiente fórmula:

$$d = \frac{ancho\ de\ la\ pantalla/2}{\tan\left(FOV/2\right)}$$

El problema es que, al ser la altura de la pantalla 32 píxeles (mucho menor que el ancho las alturas de las paredes proyectadas), casi siempre van a ser mayores a la altura de la pantalla, por lo que se realiza un ajuste cambiando la formula a

$$d = \frac{alto\ de\ la\ pantalla/2}{\tan\left(FOV/2\right)}$$

El FOV es el campo de visión del jugador. En nuestro caso, es de 64º. Más adelante se explicará el porqué.

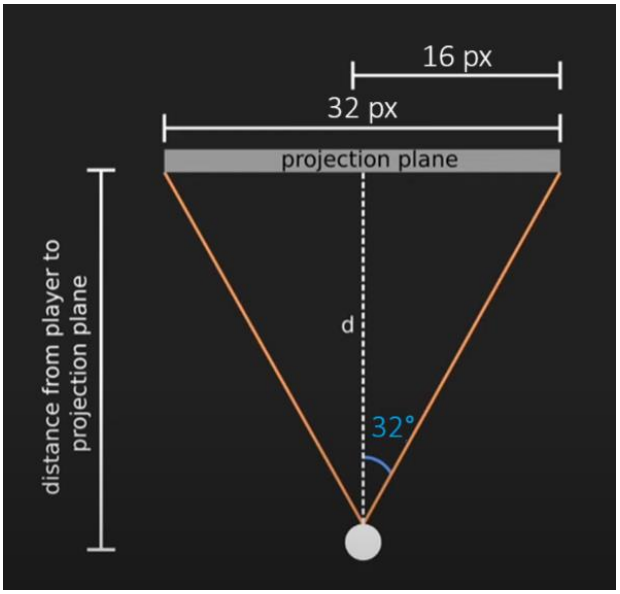

Figura 10: Campo de visión

Finalmente, la altura de la pared proyectada queda:

$$altura\ de\ la\ pared\ proyectada = \frac{altura\ de\ la\ pared}{distancia\ a\ la\ pared} * d$$

Ahora solo queda lanzar más rayos para crear un campo de visión. Como ya se ha mencionado, el campo de visión del jugador es de 64º. Se lanzarán el mismo número de rayos que grados del FOV. Se itera desde 0 hasta el número de rayos y se normaliza el ángulo de la cámara usando la siguiente formula:

$$cx = \frac{2 * r}{N} - 1$$

Dónde r es el número del rayo actual (de 0 a 64) y N el número total de rayos (64). Esto devolverá un numero entre -1 y 1 que se usará para calcular el ángulo de cada rayo utilizando la siguiente fórmula:

$$\theta = \alpha + \text{atan}\ (cx * \tan\ (FOV)\ )$$

Dónde α el ángulo del jugador (que varía entre 0 y 360º, siendo ajustado por el potenciómetro como ya se ha explicado). Después, se ejecutará el algoritmo usando θ como ángulo inicial para calcular el vector v.

Como la pantalla es de 128 pixeles y se lanzan solo 64 rayos se establece un divisor de resolución = 2 para que lance un rayo cada 2 pixeles en la pantalla quedando las paredes con un patrón rayado. Se pueden rellenar las líneas intermedias a las que no se ha lanzado un rayo pero en este caso al ser la pantalla monocromática y los sprites de las entidades de color negro, un patrón rayado ayuda a diferenciar a las entidades de las paredes.

Una corrección común necesaria en la mayoría de raycasters es corregir el efecto de ojo de pez generado porque los rayos lanzados desde la izquierda y la derecha tienen más distancia que los rayos lanzados desde el centro.

Para ello se calcula la distancia al plano del jugador. Si D es la distancia del rayo, entonces : $D' = D * \cos\ (\beta)$

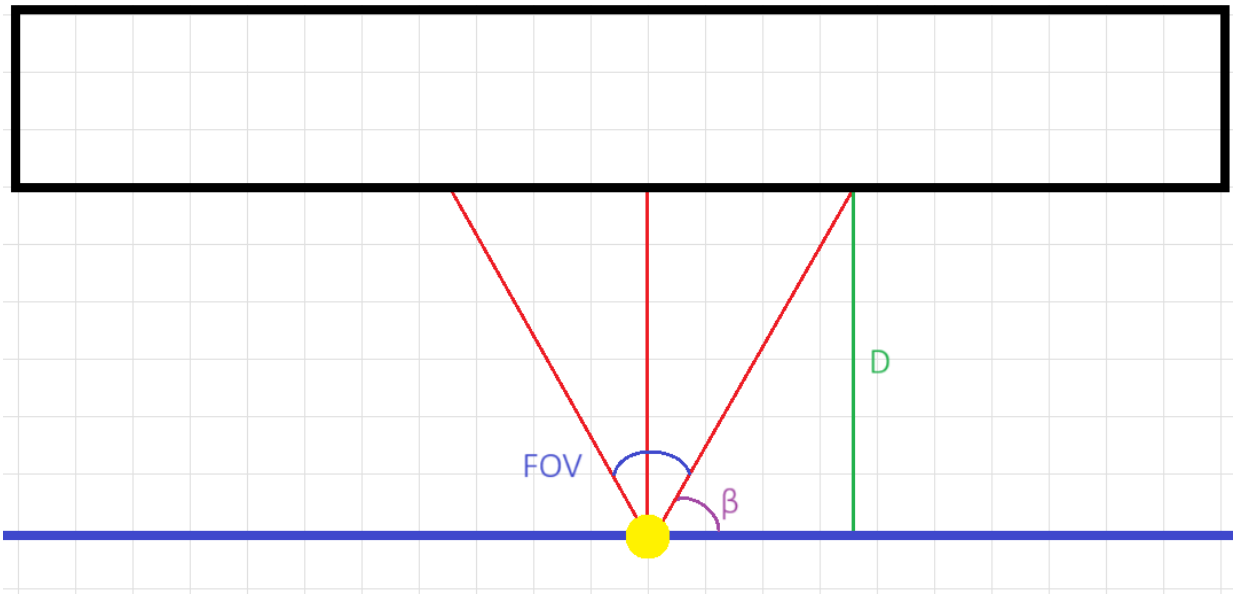

Figura 11: Corrección ojo de pez

Esta corrección no ha sido necesaria emplearla en la práctica. Se desconoce el motivo por el que no ha sido necesario, pero una hipótesis es que por la limitada altura de la pantalla este efecto no es capaz de apreciarse ya que al estar cerca de una pared todas las paredes se van a ver de altura máxima. Aun así, éste es un problema común de los motores de raycasting.

Imágenes

En esta sección se incluyen diferentes imágenes que hemos considerado de interés.

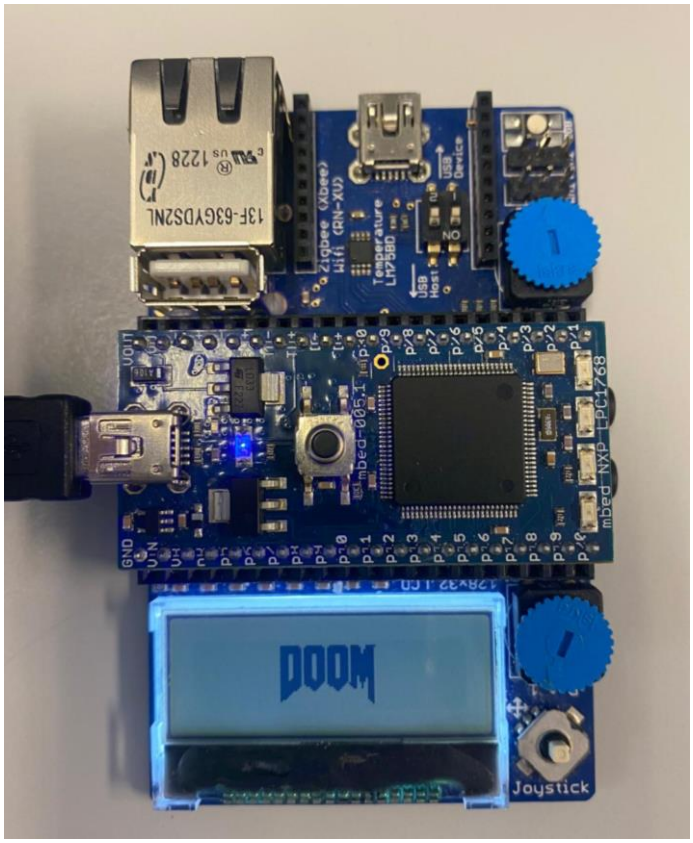

Figura 12: Pantalla principal

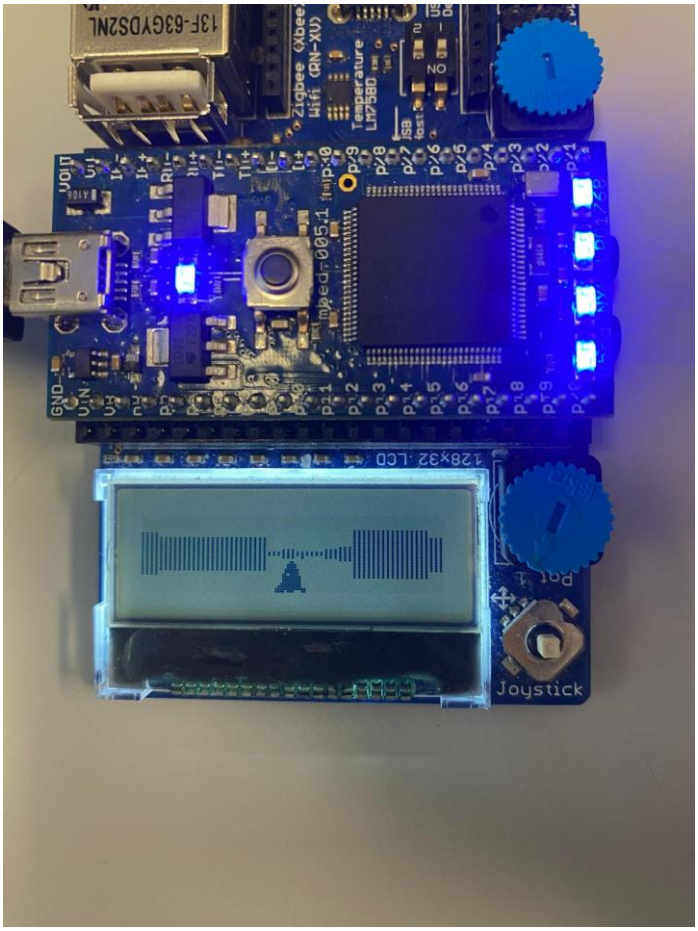

Figura 13: Vista del juego

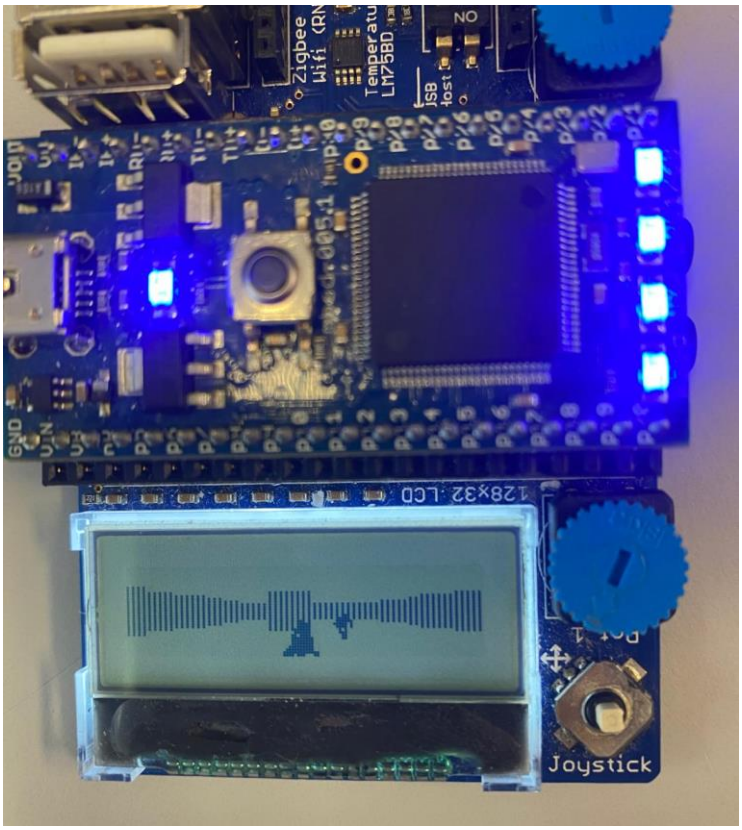

Figura 14: Enemigo acercándose al jugador

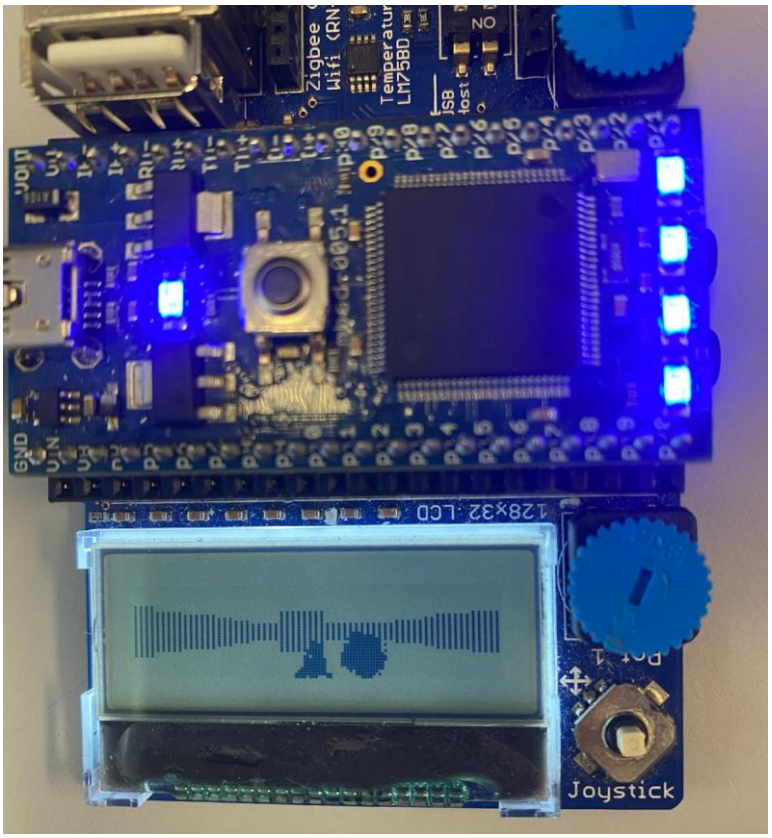

Figura 15: Enemigo atacando a distancia (Bola de fuego)

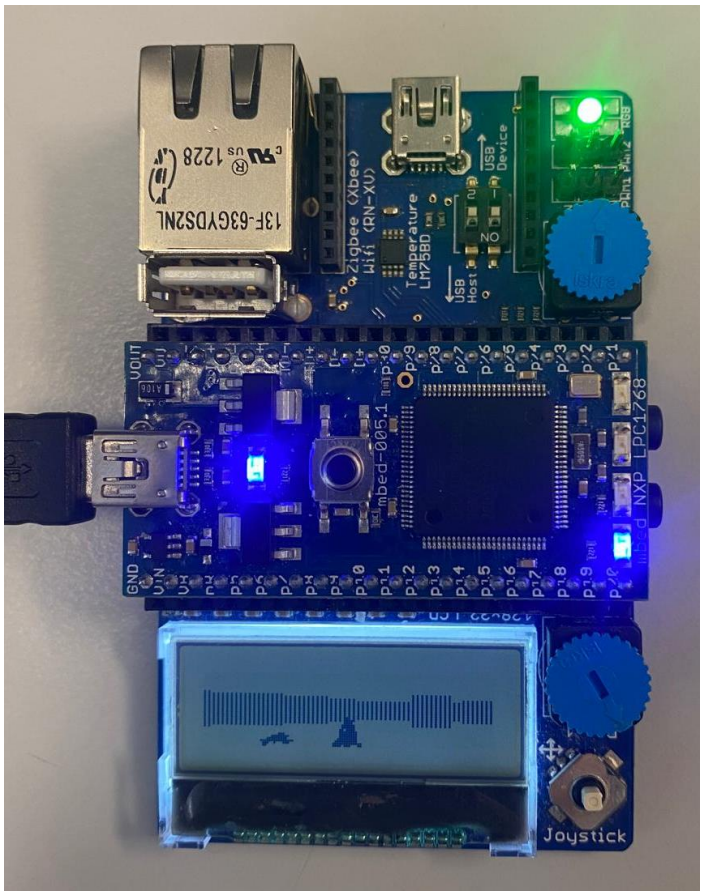

Figura 16: Todos los enemigos eliminados (LED verde)

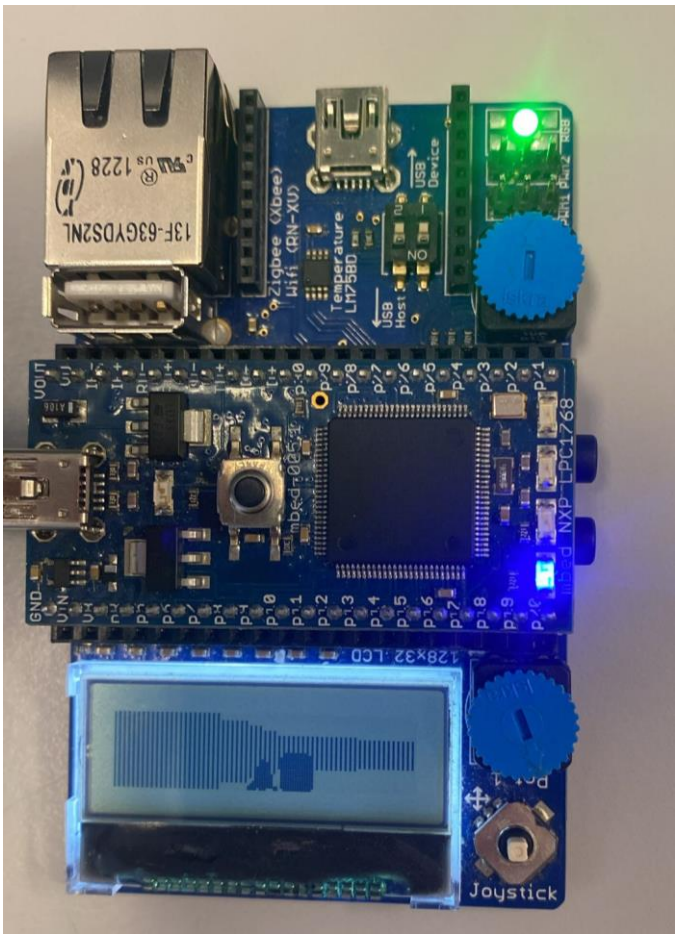

Figura 17: Botiquín

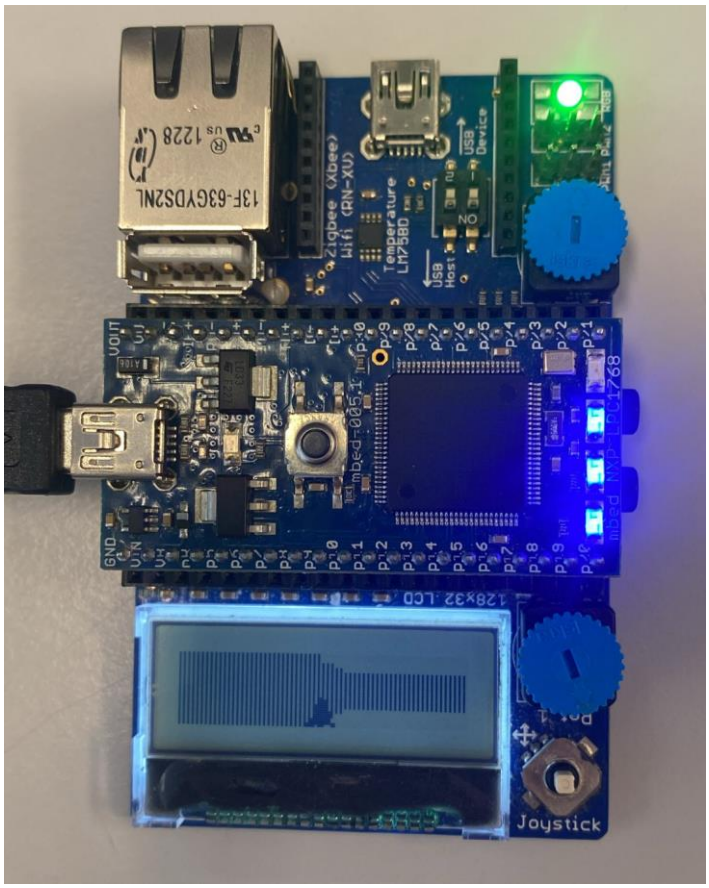

Figura 18: Salud restaurada tras coger botiquín (+50 PS)

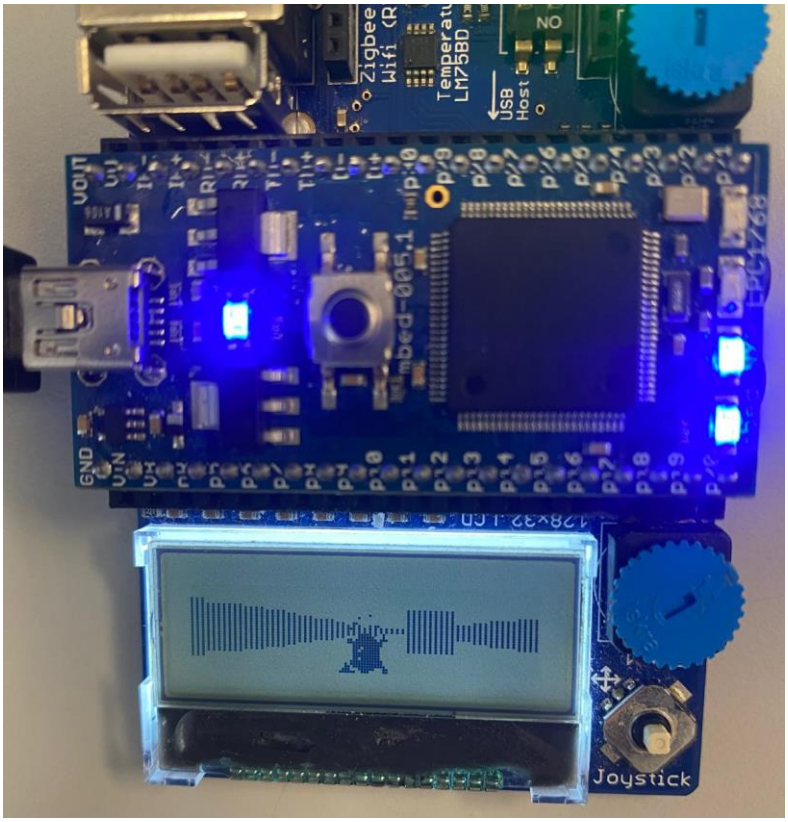

Figura 19: Pistola disparando

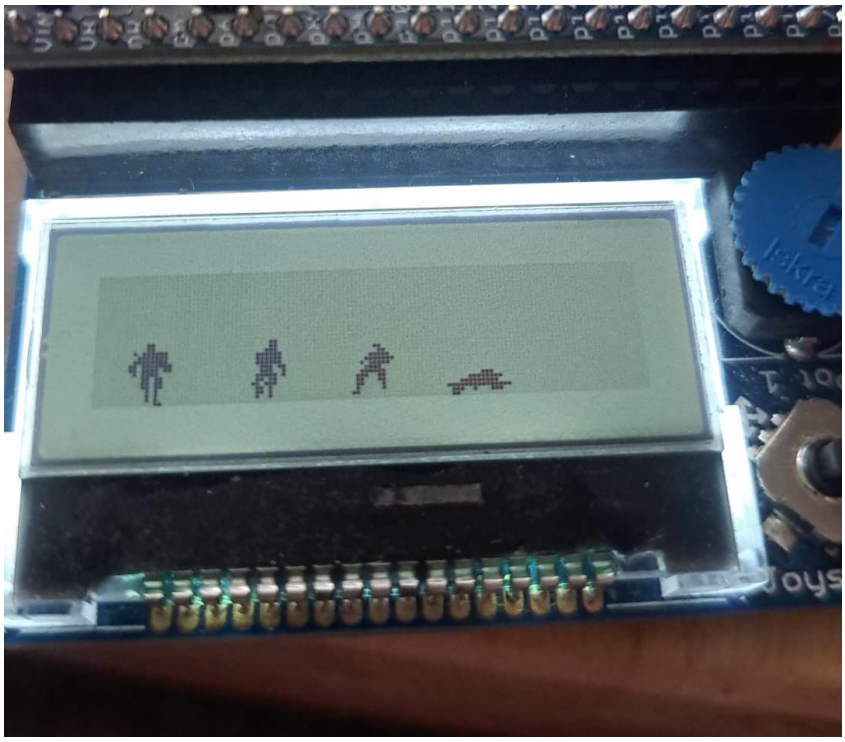

Figura 20: Diferentes sprites del enemigo (1 y 2: Caminando, 3: Daño recibido, 4: Muerto)

Problemas y futuras mejoras

El código final sigue teniendo problemas y aún hay muchas mejoras a implementar. En esta sección se ven con más detalle.

PROBLEMAS

- Los enemigos se ven a través de las paredes, y además, se les puede disparar mientras que ellos a ti no. Desconocemos el motivo.
- El sprite del botiquín no carga correctamente, se ve como un rectángulo negro, mientras que debería verse así:

Figura 21: Sprite botiquín

También desconocemos el motivo.

- Para disparar al enemigo hay que apuntar ligeramente a la izquierda por como está diseñado el sprite.

MEJORAS

- Sistema de munición, en este momento es infinita.
- Sistema de llaves para abrir puertas, o al eliminar a todos los enemigos que se desbloquee una puerta, o que salga que has ganado… En este momento, al eliminar a los enemigos (en el código solo hay 1 puesto, pero esto como ya hemos visto se puede cambiar) no ocurre nada.
- Añadir diferentes niveles.
- Añadir la banda sonora del DOOM, al igual que sonidos de disparo, enemigos… ya que la placa cuenta con altavoz.

Bibliografía

Motor de raycasting :

- https://lodev.org/cgtutor/raycasting.html
- https://www.youtube.com/watch?v=NbSee-XM7WA&t=748s
- https://www.youtube.com/watch?v=gYRrGTC7GtA

Capturas de pantalla y metodología para calcular la altura de las paredes:

- https://www.youtube.com/watch?v=E18bSJezaUE&t=6046s

Renderizado de sprites:

- https://lodev.org/cgtutor/raycasting3.html

Modelos de los sprites:

- https://github.com/potat-dev/milandr-doom

Diseño de nuestros propios sprites:

- https://javl.github.io/image2cpp/

Logo DOOM:

- https://es.m.wikipedia.org/wiki/Archivo:Doom_logo.png

Capturas de pantalla para los sprites de la pistola:

- https://www.youtube.com/watch?v=MnqLJpgq7jc&t=17s