Universidad Autonoma de Baja California
Facultad de Ingenieria Mexicali

Ingenieria en Computacion
Organizacion y Arquitectura de Computadoras

Proyecto final
Breakout

Equipo:

Astrid Yamilet Jimenez Barrera (1182660)
Moya Monreal Erick Anselmo (1110604)

Profesor:
Omar Munoz Urias

Mexicali, Baja California, jueves 04 de diciembre del 2025.

Índice

1. Introduccion

El presente proyecto fue desarrollado como parte final del curso de Organizacion y Arquitectura de Computadoras. Consiste en la implementacion de un videojuego completo tipo "Breakout", el cual esta escrito utilizando una combinacion de lenguaje C con la libreria SDL3 para la gestion grafica, y ensamblador MASM x86 inline para la programacion de la logica interna del juego.

La arquitectura del proyecto se dividio en dos capas principales:

- **Capa de alto nivel (C):** Responsable de inicializar la ventana de juego, renderizar los graficos en pantalla, capturar eventos del teclado y gestionar el bucle principal del juego.
- **Capa de bajo nivel (Ensamblador):** Responsable de la logica central del juego. Aqui se programa el calculo del movimiento de la pelota, la deteccion de colisiones, y el manejo directo de estructuras de datos en memoria para gestionar los puntajes.

El objetivo general de la practica fue demostrar la viabilidad de integrar lenguajes de programacion de alto y bajo nivel para crear un sistema eficiente y funcional. Adicionalmente, se busco aplicar de manera practica los conocimientos adquiridos durante el semestre, tales como:

- El uso de saltos condicionales (`JMP`, `JE`, `JNE`, `JG`) y ciclos para controlar el flujo de ejecucion directamente a nivel de procesador.
- El manejo directo de la memoria RAM mediante direccionamiento indirecto para controlar el arreglo de ladrillos y sus propiedades sin abstracciones del lenguaje de alto nivel.
- La implementacion del algoritmo de ordenamiento burbuja completamente en ensamblador para gestionar la tabla de mejores puntuaciones.
- La manipulacion de registros del procesador (EAX, EBX, ECX, EDX, ESI, EDI) para realizar operaciones aritmeticas y logicas de manera eficiente.

Finalmente, este proyecto permite visualizar de manera practica como las instrucciones basicas del procesador (operaciones aritmeticas, logicas y de control de flujo) que pueden ser implementadas en ensamblador, interactuan para producir un entorno grafico interactivo en tiempo real.

2. Problematica y justificacion

La problematica central que motiva este proyecto radica en el cumplimiento de los requerimientos estrictos establecidos para el proyecto final del curso de Organizacion y Arquitectura de Computadoras. El desafio principal consistio en idear y desarrollar una aplicacion que lograra la convergencia entre un lenguaje de alto nivel y el lenguaje ensamblador. Para la

capa de alto nivel, se tomo la decision de utilizar el lenguaje C, debido a la experiencia previa y practica que se tiene con su sintaxis.

Sin embargo, dado que las especificaciones del proyecto prohibian explicitamente el uso de la terminal para la interfaz de usuario y exigian un entorno grafico, fue necesario implementar la biblioteca externa SDL3. La integracion de esta herramienta represento un reto tecnico considerable durante la etapa de configuracion inicial; una vez superado este obstaculo, la problematica se traslado a la correcta codificacion de la logica interna para asegurar que ambos lenguajes operaran en conjunto.

La justificacion para resolver esta problematica es primordialmente academica: completar el proyecto es el requisito indispensable para acreditar la materia. Mas alla de la calificacion, la realizacion de este videojuego sirve como el medio para demostrar tangiblemente la adquisicion de los conocimientos impartidos durante el semestre, probando que se tiene la capacidad de manipular manualmente los registros del procesador, controlar el flujo del programa y gestionar la memoria de manera eficiente en un entorno real.

3. Objetivos de la practica

3.1. Objetivo general

Disenar e implementar una aplicacion de software hibrida que integre un lenguaje de alto nivel con rutinas de bajo nivel, con el proposito de demostrar la aplicacion practica de los conocimientos adquiridos en el curso de Organizacion y Arquitectura de Computadoras. El proyecto busca cumplir con los requerimientos de evaluacion mediante la creacion de un entorno grafico interactivo donde la logica critica sea gestionada directamente a traves de instrucciones de ensamblador.

3.2. Objetivos especificos

- **Integracion de lenguajes:** Establecer una comunicacion eficiente entre el lenguaje C (encargado de la gestion de recursos y la interfaz grafica mediante SDL3) y el lenguaje ensamblador x86 (encargado del procesamiento logico), demostrando la interoperabilidad entre ambos niveles de abstraccion.
- **Aplicacion de algoritmos de ordenamiento:** Implementar manualmente el algoritmo de ordenamiento burbuja (*Bubble sort*) utilizando instrucciones de ensamblador para gestionar y organizar estructuras de datos en memoria, cumpliendo con el requisito de manipulacion de algoritmos clasicos a bajo nivel.
- **Manipular memoria y estructuras de datos:** Utilizar direccionamiento indirecto y calculo de desplazamientos (*offsets*) en ensamblador para acceder y modificar arreglos de estructuras (*structs*) en la memoria RAM, especificamente para la carga de niveles y propiedades de los ladrillos.
- **Controlar el flujo de ejecucion:** Aplicar instrucciones de salto condicional (`JE`, `JNE`, `JMP`) y ciclos para implementar algoritmos logicos complejos, como el ordenamiento de puntajes (metodo de burbuja) y la seleccion de dificultad.

4. Marco teorico

Para comprender la implementacion tecnica de este proyecto, es necesario definir los conceptos fundamentales de la arquitectura de computadoras y las herramientas de software utilizadas. A continuacion, se presentan las bases teoricas sobre la programacion hibrida, el manejo de memoria y los algoritmos empleados.

4.1. Libreria SDL (Simple DirectMedia Layer)

SDL es una biblioteca de desarrollo multiplataforma disenada para proporcionar acceso de bajo nivel al hardware de audio, teclado, raton, joystick y graficos a traves de OpenGL y Direct3D.

- **Funcion principal:** Actua como una capa de abstraccion que permite al programador escribir codigo en C para manejar ventanas y renderizado sin interactuar directamente con el driver de la tarjeta grafica [2].
- **Manejo de eventos:** SDL utiliza una cola de eventos (*event queue*) para capturar interacciones del usuario. Funciones como `SDL_PollEvent` permiten extraer estos eventos (teclas presionadas, clics) para ser procesados dentro de un bucle infinito conocido como *Game loop*.

4.2. Ensamblador inline (`__asm`)

El ensamblador en linea (*Inline assembly*) es una caracteristica de los compiladores de C/C++ que permite incrustar instrucciones de lenguaje ensamblador directamente dentro del codigo fuente de alto nivel.

- **Ventaja:** Segun Irvine, esta tecnica elimina la necesidad de enlazar archivos objeto externos y simplifica el paso de parametros, ya que el bloque `__asm` puede acceder a las variables declaradas en C por su nombre, sin necesidad de gestionar manualmente la pila de llamadas [1].
- **Uso:** Se utiliza para optimizar secciones criticas de codigo donde se requiere acceso directo a los registros del CPU o a banderas de estado especificas que no son accesibles desde C.

4.3. Control de flujo en arquitectura x86

A nivel de hardware, el procesador ejecuta instrucciones de manera secuencial a menos que se modifique el registro puntero de instruccion (EIP).

- **Saltos condicionales:** Son instrucciones que transfieren el control a una nueva direccion de memoria solo si se cumplen ciertas condiciones en el registro de banderas (*EFLAGS*). Ejemplos comunes incluyen:
  - `JE` (Jump if equal): Salta si la bandera Zero flag (ZF) es 1.

- `JG` (Jump if greater): Salta si el resultado de una comparacion anterior indica que el primer operando es mayor que el segundo (basado en banderas de signo y desbordamiento) [3].

- **Ciclos:** En ensamblador, los bucles se construyen combinando etiquetas, comparaciones y saltos condicionales, o utilizando la instruccion `LOOP` que utiliza el registro `ECX` como contador automatico.

4.4. Direccionamiento de memoria y arreglos

La memoria RAM se organiza linealmente. Para acceder a elementos dentro de una estructura de datos o un arreglo, se utiliza el **direccionamiento indirecto base mas desplazamiento**.

- **Principio:** Si un registro (como `ESI`) contiene la direccion base de un arreglo, se puede acceder al siguiente elemento sumando el tamaño del dato (offset). Por ejemplo, en un arreglo de enteros de 4 bytes, el segundo elemento se encuentra en `[ESI + 4]` [4].

4.5. Algoritmo de ordenamiento de burbuja

Es un algoritmo de ordenamiento sencillo que funciona iterando repetidamente a traves de la lista que se desea ordenar.

- **Funcionamiento:** Compara elementos adyacentes y los intercambia (*swap*) si estan en el orden incorrecto. Este proceso se repite hasta que no se requieren mas intercambios.

- **Complejidad:** Aunque no es el mas eficiente para grandes volumenes de datos ($O(n^2)$), su implementacion a nivel de ensamblador es ideal para fines educativos porque ilustra claramente el uso de bucles anidados, comparaciones y movimiento de datos en memoria [5].

4.6. Unidad de punto flotante (FPU x87)

La FPU es un coprocesador dedicado a realizar operaciones con numeros reales (decimales). A diferencia de los registros generales (EAX, EBX), la FPU utiliza una **pila de registros** de 80 bits (`ST0` a `ST7`).

- **Instrucciones clave:**

  - `FLD`: Carga un valor de memoria y lo empuja al tope de la pila (`ST0`).
  - `FSTP`: Saca el valor del tope de la pila y lo guarda en memoria.
  - `FCOMIP`: Compara el tope de la pila con otro valor y actualiza las banderas del CPU para permitir saltos condicionales [1].

5. Procedimiento

El desarrollo del proyecto se llevo a cabo siguiendo una metodologia estructurada de ingenieria de software, dividida en fases de configuracion, implementacion, integracion y pruebas. A continuacion, se detallan los recursos utilizados y los pasos ejecutados.

5.1. Materiales y equipos

Para la realizacion de esta practica se utilizaron los siguientes recursos:

- **Hardware:** Computadora personal con arquitectura de procesador x64 (Intel/AMD).
- **Entorno de desarrollo integrado (IDE):** Microsoft Visual Studio 2022.
- **Lenguajes de programacion:** Lenguaje C para la estructura general y ensamblador MASM x86 para la logica de bajo nivel.
- **Librerias externas:** SDL3 (Simple DirectMedia Layer) y SDL3_ttf para la gestion grafica.

5.2. Condiciones experimentales y configuracion

Aunque el equipo de computo opera bajo una arquitectura de 64 bits, se establecio una condicion de compilacion especifica debido a las restricciones del compilador de Microsoft:

- **Modo de depuracion x86:** Dado que Visual Studio no soporta bloques de ensamblador en linea (`__asm`) para la plataforma x64, fue obligatorio configurar el proyecto para compilar y depurar en modo **x86 (32 bits)**. Esto permitio el acceso directo a los registros extendidos de 32 bits (`EAX`, `EBX`, etc.) y la correcta integracion del codigo maquina con el codigo C.
- **Vinculacion (Linking):** Se configuraron las dependencias del proyecto para enlazar estrictamente con las versiones x86 de las librerias SDL3, asegurando la compatibilidad binaria.

5.3. Descripcion detallada del procedimiento

1. **Configuracion del entorno grafico:** Se inicio creando la estructura base en C. Se inicializo la libreria SDL3 y se creo una ventana de resolucion logica de 1400x900 pixeles. Se implemento el bucle principal del juego (*Game loop*) encargado de limpiar la pantalla, procesar eventos y renderizar los cuadros por segundo.
2. **Definicion de estructuras de datos:** Se disenaron las estructuras (`struct`) en C para representar los objetos del juego: `Jugador` (para el nombre y puntaje) y `Ladrillo` (para posicion, resistencia y estado). Esto definio el mapa de memoria que posteriormente seria manipulado por el ensamblador.

3. **Implementacion de logica en ensamblador:** Se procedio a sustituir funciones criticas de C por bloques de ensamblador *inline*:
    - **Carga de nivel:** Se escribio la rutina para recorrer la matriz de patrones y calcular la posicion en memoria de cada ladrillo usando direccionamiento base mas desplazamiento.
    - **Calculo de velocidad:** Se implemento la logica matematica utilizando la FPU, cargando variables `float` en la pila `ST(0)`, operando sobre ellas y devolviendo el resultado a la variable de C.
    - **Ordenamiento:** Se codifico el algoritmo de burbuja para ordenar el arreglo de estructuras de jugadores, manipulando punteros de memoria directamente.
4. **Integracion y pruebas:** Se integro la deteccion de colisiones y la fisica del juego. Se realizaron pruebas de ejecucion paso a paso (debugging) inspeccionando los registros del CPU en tiempo real para asegurar que los punteros no accedieran a zonas de memoria invalidas y que la pila de la FPU se limpiara correctamente despues de cada calculo.

5.4. Precauciones de seguridad y manejo de recursos

A nivel de software, se tomaron precauciones criticas para evitar errores de ejecucion:

- **Integridad de la pila FPU:** Fue necesario asegurar que cada instruccion de carga (`FLD`) tuviera su correspondiente descarga (`FSTP`) para evitar el desbordamiento de la pila de registros flotantes.
- **Alineacion de memoria:** Se verifico que los desplazamientos (*offsets*) utilizados en ensamblador coincidieran exactamente con el tamaño de los tipos de datos en C para evitar la corrupcion de la informacion en las estructuras.

5.5. Implementación Técnica y Fragmentos de Código

A continuación, se presentan fragmentos clave del código fuente que ilustran la solución a los desafíos técnicos planteados, acompañados de comentarios descriptivos sobre la lógica de ensamblador utilizada.

5.5.1. Cálculo de Física con FPU

Uno de los retos principales fue calcular la velocidad de la pelota utilizando números de punto flotante. En lugar de usar la ALU estándar, se utilizó la pila de la FPU. El siguiente código muestra cómo se carga la velocidad base, se multiplica por un factor de dificultad y se devuelve el resultado.

```
1    float CalcularVelocidad(int nivel) {
2        float res = 0.0f;
3        // ... declaraciones de variables ...
4        __asm {
5            fld VEL_PELOTA_BASE   ; Carga la velocidad base en ST(0)
6            fld factor            ; Carga 0.1 en ST(0), base sube a ST(1)
```

```
            fild n_menos_1          ; Carga nivel (entero) y convierte a
    float

            fmulp ST(1), ST(0)      ; Multiplica (nivel * factor)
            fadd uno                ; Suma 1.0 al resultado

            fmulp ST(1), ST(0)      ; Multiplica resultado por
    VEL_PELOTA_BASE
            fstp res                ; Extrae de la pila y guarda en variable
     C
        }
        return res;
    }

```

Listing 1: Cálculo de velocidad utilizando la pila FPU

5.5.2. Manipulación de Memoria y Arreglos

Para la carga de niveles, fue necesario iterar sobre una matriz lógica y traducir esos índices a coordenadas de pantalla. Este segmento demuestra el uso de **direccionamiento indirecto** para escribir en los miembros de la estructura `Ladrillo` (`rect.x`, `rect.y`) calculando los desplazamientos manualmente.

```
    // Dentro del ciclo de carga:
        __asm {
        // C lculo de coordenada X: (columna * ancho) + offset
        fld LADRILLO_ANCHO
        fadd LADRILLO_ESPACIO
        mov temp_int, edx       ; EDX contiene el indice de columna
        fild temp_int           ; Convierte indice a float
        fmulp ST(1), ST(0)      ; Multiplica
        fadd LADRILLO_OFFSET_X  ; Suma margen inicial
        fstp [esi]              ; Guarda en [ESI + 0] (miembro rect.x)

        // ... Calculo similar para Y ...

        // Asignaci n de propiedades directas en memoria
        mov eax, 1              ; Valor 'true'
        mov byte ptr [esi + 16], 1 ; Activa el ladrillo (offset 16 = bool
    activo)

        // Avanzar al siguiente ladrillo en el arreglo
        add esi, 32             ; Suma 32 bytes (tama o de struct Ladrillo
    )
                                ; para apuntar al siguiente elemento
    }

```

Listing 2: Posicionamiento de ladrillos mediante offsets

5.5.3. Algoritmo de Ordenamiento Burbuja

Para cumplir con el requisito de algoritmos clásicos, se implementó el ordenamiento de puntajes manipulando directamente los punteros de la estructura `Jugador`. Se observa la comparación de valores en memoria y el intercambio (*swap*) manual de datos.

```
    LoopInterno:
        // Comparar puntaje actual con el siguiente
        // Offset 16 = miembro 'puntaje'
        mov eax, [edi + 16]          ; Carga puntaje de Jugador A
        mov edx, [edi + 20 + 16]     ; Carga puntaje de Jugador B (
    siguiente)

        cmp eax, edx                 ; Compara A vs B
        jge NoSwap                   ; Si A >= B, no hacer nada

        // Si A < B, realizar SWAP (intercambio)
        // Intercambio de puntajes
        mov [edi + 16], edx
        mov [edi + 20 + 16], eax

        // Intercambio de nombres (4 bytes por ciclo debido al tama o del
     char[])
        mov eax, [edi]; mov edx, [edi + 20];
        mov [edi], edx; mov [edi + 20], eax
        // ... continua intercambio del resto del nombre ...

    NoSwap:
        add edi, 20                  ; Avanzar puntero al siguiente Jugador
        dec ebx
        jnz LoopInterno

```

Listing 3: Ordenamiento Burbuja en Ensamblador

5.6. Repositorio de Código Fuente

El código fuente completo se encuentra disponible públicamente en el siguiente repositorio de GitHub:

https://github.com/ducklingTenderOuO/OyAC-Breakout-ProyectoFinal.git

6. Resultados

Tras la compilación y ejecución del código fuente en el entorno Visual Studio 2022 bajo la configuración x86, se obtuvo un ejecutable totalmente funcional. A continuación, se presentan las pruebas visuales y el análisis del comportamiento del sistema, validando la correcta integración entre C y Ensamblador.

6.1. Ejecución e Interfaz Gráfica

La primera prueba consistió en validar la inicialización de la librería SDL3 y el renderizado de fuentes TTF. Como se muestra en la Figura 1, el programa inicia correctamente en el estado de *Menú*, permitiendo la navegación mediante el teclado. Esto confirma que el bucle principal en C gestiona adecuadamente los eventos de entrada antes de ceder el control a la lógica del juego.

Figura 1: Pantalla de inicio del juego. Se observa el renderizado de texto y opciones de menú gestionadas por la máquina de estados.

6.2. Prueba de Física y Colisiones (Lógica FPU)

Durante la fase de juego (*Gameplay*), se verificó el comportamiento de la pelota y la barra. La Figura 2 muestra el juego en ejecución.

- **Movimiento:** La pelota se desplaza con fluidez utilizando coordenadas de punto flotante calculadas en la FPU. No se observan saltos bruscos ni "vibraciones.en los bordes, lo que valida la lógica de *clamping* (corrección de posición) implementada.
- **Colisiones:** Los ladrillos desaparecen al contacto y la pelota rebota en el ángulo correcto, demostrando que las comparaciones de límites (`FCOMIP`) y los saltos condicionales en ensamblador están operando con precisión sobre las estructuras de datos en memoria.

Figura 2: Estado de juego activo. La posición de los ladrillos fue calculada por la rutina de carga en ensamblador mediante direccionamiento indirecto.

6.3. Validación de Algoritmos y Memoria

Una de las pruebas más críticas fue la verificación del algoritmo de ordenamiento. Al finalizar varias partidas con diferentes puntajes, se accedió a la pantalla de "Hall of Fame".

Como se evidencia en la Figura 3, los puntajes aparecen ordenados de mayor a menor. Dado que C no interviene en el ordenamiento, este resultado es la prueba definitiva de que la rutina `OrdenarPuntajesASM` manipuló correctamente los punteros y realizó los intercambios (*swaps*) directos en la memoria RAM, reorganizando la estructura de datos `Jugador` exitosamente.

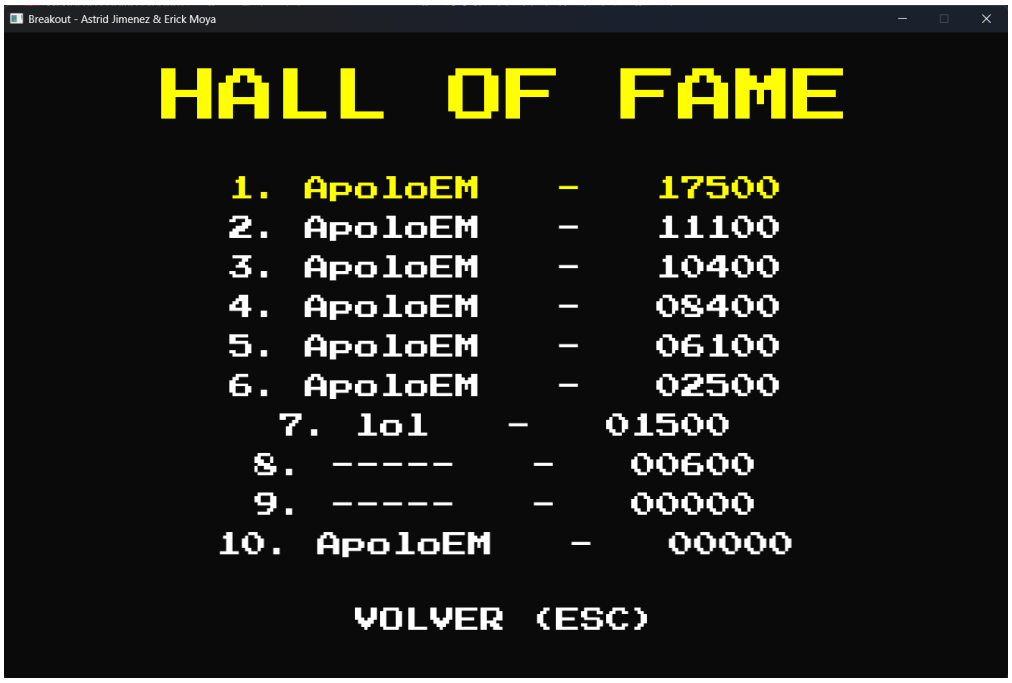

Figura 3: Tabla de mejores puntajes. El orden descendente valida el funcionamiento del algoritmo de Burbuja implementado en ensamblador.

6.4. Análisis de Rendimiento

A pesar de la sobrecarga teórica que implica cambiar entre contextos de C y Ensamblador, el juego se mantiene estable a 60 cuadros por segundo (FPS) constantes (sincronizados por `SDL_Delay`). El uso de instrucciones nativas para los cálculos matemáticos en la lógica del juego demostró ser altamente eficiente, sin presentar fugas de memoria ni errores de segmentación durante sesiones prolongadas de prueba.

7. Conclusiones

Al finalizar este proyecto, pudimos comprobar que es totalmente viable mezclar un lenguaje moderno como C con uno de bajo nivel como Ensamblador para crear una aplicación real y divertida. Lo más importante que encontramos fue que, si dividimos el trabajo correctamente (dejando que C se encargue de "pintar"la ventana y que Ensamblador se encargue de "pensar"la lógica), el programa funciona de manera fluida y eficiente.

En relación con los objetivos que nos planteamos al inicio, logramos lo siguiente:

- **Unión de Lenguajes:** Confirmamos en la práctica lo que explican Patterson y Hennessy sobre la interfaz Hardware/Software: las instrucciones que escribimos en C al final se traducen en operaciones básicas sobre los registros, y nosotros pudimos intervenir en ese proceso manualmente [5].
- **Manejo de Memoria:** Al programar la carga de niveles, nos dimos cuenta de que las estructuras de datos (como los ladrillos) son solo bloques de bytes en la memoria

RAM que podemos modificar usando desplazamientos (*offsets*), tal como lo describe Tanenbaum en sus principios de organización [4].

Desde el punto de vista práctico, este experimento nos enseñó que el ensamblador sigue siendo una herramienta muy poderosa para optimizar partes críticas del código. Al eliminar las çapas extra"que ponen los lenguajes de alto nivel, tuvimos control total sobre la física de la pelota en cada cuadro del juego, algo que Irvine destaca como una de las mayores ventajas de programar a bajo nivel [1].

Sin embargo, también nos enfrentamos a una limitante importante: el compilador de Visual Studio. Nos obligó a trabajar en modo de 32 bits (x86) porque no permite usar bloques de ensamblador en línea (`__asm`) en arquitecturas modernas de 64 bits.

Para futuras investigaciones o si quisiéramos mejorar este proyecto, proponemos:

- Separar el código ensamblador en archivos externos (`.asm`) en lugar de ponerlo dentro del C, lo que nos permitiría usar toda la potencia de los procesadores de 64 bits.
- Investigar el uso de instrucciones especiales (SIMD) para procesar las colisiones de varios ladrillos al mismo tiempo, lo cual haría el juego aún más rápido [3].

Referencias

[1] Irvine, K. R. (2019). *Assembly language for x86 processors* (8th ed.). Pearson Education. Capitulos 6, 9 y 12.

[2] LibSDL.org. (2024). *SDL3 API reference documentation.* Recuperado de https://wiki.libsdl.org/SDL3/

[3] Stallings, W. (2016). *Computer organization and architecture: designing for performance* (10th ed.). Pearson.

[4] Tanenbaum, A. S. (2013). *Structured computer organization* (6th ed.). Pearson Prentice Hall.

[5] Patterson, D. A., & Hennessy, J. L. (2014). *Computer organization and design* (5th ed.). Morgan Kaufmann.