Documentación Técnica - Mario Assembler

Desarrollo y Explicación de las Consignas

Autores: Matías Berardo, Alexis Byrne, Agustina Caffaratti, Lucas Ruberto, Franco Rubin

Fecha: Noviembre 2025

Abstract

Este documento presenta una descripción completa y detallada del desarrollo de cada una de las consignas del proyecto asignado. Su objetivo principal es servir como guía técnica que permita comprender el razonamiento, las decisiones de diseño, la implementación y las pruebas asociadas a cada punto solicitado en el trabajo. La documentación está pensada tanto para el docente evaluador como para cualquier desarrollador o lector técnico que desee entender en profundidad cómo se resolvieron las distintas partes del sistema. A lo largo del documento se explica el propósito de cada módulo, la lógica aplicada, los algoritmos utilizados y los desafíos encontrados durante el proceso de desarrollo.

1. Introducción

El presente informe tiene como propósito documentar de manera clara y organizada todas las etapas y consignas correspondientes al proyecto. En él se detalla tanto el análisis teórico como la implementación práctica de las soluciones propuestas, manteniendo un enfoque orientado a la comprensión técnica del código y la justificación de las decisiones tomadas.

El documento se estructura de forma que cada consigna o apartado planteado en el enunciado sea explicado en profundidad. Se incluye información sobre la arquitectura general del sistema, los módulos desarrollados, las funciones principales, las estructuras de datos utilizadas y las rutinas clave.

De esta manera, esta documentación busca no solo cumplir con los requerimientos formales del trabajo práctico, sino también dejar un registro técnico de calidad, útil para referencia y mantenimiento del proyecto a futuro.

La estructura del documento sigue el siguiente esquema general:

- **Arquitectura del sistema:** descripción general del programa y su organización interna.
- **Explicación de las consignas:** desarrollo detallado de cada punto del enunciado, con su respectiva implementación.
- **Análisis de funcionamiento:** explicación del comportamiento esperado, casos de prueba y validaciones.

2. Division de Tareas

En el grupo noa dividimos las tareas de la siguiente manera:

- **Producer:** Matías Berardo
- **Programación:**Franco Rubin, Alexis Byrne
- **Arte y diseño:** Caffaratti Agustina, Lucas Ruberto

3. Objetivos del Documento

El objetivo de esta documentación es garantizar que cada aspecto del proyecto pueda ser comprendido sin necesidad de revisar el código fuente directamente. Se pretende que cualquier persona con conocimientos técnicos pueda seguir la lógica del desarrollo y entender cómo cada consigna fue resuelta, qué decisiones se tomaron, y por qué.

4. Estructura General del Proyecto

Antes de entrar en el detalle de las consignas, se presenta una visión general del sistema, sus módulos principales y la forma en que interactúan entre sí. Este contexto inicial permite entender mejor la relación entre los distintos componentes antes de abordar cada punto individual.

El proyecto está organizado en módulos que dividen las tareas principales del juego, de manera que cada bloque del código cumple una función específica dentro del ciclo de ejecución. Esta separación favorece la comprensión, el mantenimiento y la posibilidad de realizar pruebas por partes.

4.1. Estructura Principal del Programa

En la raíz del programa se encuentra la rutina `main`, que cumple el rol de inicialización y control general. Desde allí se configuran las variables del entorno, se establecen las direcciones base de los periféricos simulados (el *Bitmap Display* y el *Keyboard MMIO*) y se da inicio al bucle principal del juego. Este bucle coordina la lectura de entradas, la actualización física, la detección de colisiones y el renderizado en pantalla, garantizando el funcionamiento continuo del sistema.

4.2. Motor de Física

El motor de física, implementado en la rutina `update_mario_physics`, es responsable de simular el movimiento y las fuerzas que actúan sobre el personaje principal. Su función incluye la aplicación de la gravedad, el control de la fricción y la actualización de las posiciones de Mario según sus velocidades en los ejes `x` e `y`. Estos cálculos se realizan en cada iteración del bucle principal, manteniendo un comportamiento coherente y fluido.

4.3. Sistema de Entrada

El sistema de entrada está representado por la rutina `process_input`, que interpreta las teclas presionadas mediante el uso de memoria mapeada en `0xFFFF0000`. Esta lectura directa desde el registro MMIO permite que las acciones del jugador (moverse, saltar o salir del juego) se reflejen en tiempo real, modificando los valores de movimiento del personaje en los registros correspondientes.

4.4. Sistema de Renderizado

La parte gráfica del proyecto está controlada por el módulo de renderizado, implementado principalmente en la rutina `render_frame`. Este módulo escribe directamente en el framebuffer ubicado en memoria, en la dirección `0x10080000`. Allí, cada píxel es dibujado mediante rutinas auxiliares como `draw_pixel` y `fill_rect`, que se encargan de representar todos los elementos visibles del juego: plataformas, enemigos, objetos decorativos y al propio Mario.

4.5. Sistema de Colisiones

El sistema de colisiones se basa en comparaciones rectangulares del tipo AABB (*Axis-Aligned Bounding Box*). Este mecanismo permite verificar superposiciones entre Mario y los distintos objetos del entorno, como tubos, plataformas o enemigos. Según el tipo de colisión detectada, el programa determina si debe detener la caída del personaje, permitir un salto o eliminar un enemigo cuando Mario lo pisa desde arriba.

4.6. Gestión de Entidades y Estados

Además de Mario, el juego cuenta con otras entidades como los *Goombas* (enemigos), monedas y objetos del entorno. Cada una tiene su propio comportamiento definido en rutinas específicas, que controlan su movimiento y estado. Estas entidades se reinician con la rutina `init_game_state`, la cual restablece las posiciones iniciales, banderas de activación y variables de progreso del jugador. La cámara también se gestiona en esta rutina, asegurando que el desplazamiento del escenario acompañe el movimiento de Mario.

4.7. Resumen de la Arquitectura

En conjunto, el sistema mantiene una arquitectura simple pero eficaz, donde cada módulo depende de datos globales bien definidos en el segmento `.data`. Esto permite mantener sincronía entre la lógica del juego, las físicas, las entradas y el renderizado, garantizando un flujo estable y coherente durante toda la ejecución. La estructura modular facilita además la depuración y ampliación del código, permitiendo añadir nuevas mecánicas o mejorar el rendimiento sin comprometer la organización general del proyecto.

5. Consignas Principales

5.1. Sistema de Física y Movimiento

5.1.1 Implementación del Motor Físico

El motor de física implementa un sistema de simulación basado en integración de Euler, donde la velocidad y posición se actualizan en cada frame. Las constantes físicas definidas en `.data` (`GRAVITY=1`, `JUMP_VELOCITY=-12`, `MAX_FALL_SPEED=6`) controlan el comportamiento:

```
update_mario_physics:
    lw $t0, mario_vy
    lw $t1, GRAVITY
    add $t0, $t0, $t1       # vy += gravity
    # Limitar velocidad de caida
    lw $t1, MAX_FALL_SPEED
    blt $t0, $t1, vy_ok
    move $t0, $t1
```

5.1.2 Sistema de Aceleración Diferenciada

El código implementa dos modos de aceleración según el estado del personaje:

- **En suelo:** `MAX_SPEED=5`, `ACCELERATION=3` (control preciso)
- **En aire:** `AIR_MAX_SPEED=8`, `AIR_ACCELERATION=5` (mayor libertad de movimiento)

Esto permite mantener el control durante saltos mientras se limita el deslizamiento en tierra.

5.2. Detección de Colisiones AABB

5.2.1 Sistema de Colisión Rectangular

Todas las colisiones usan el algoritmo AABB (*Axis-Aligned Bounding Box*), verificando superposición en ambos ejes:

```
# Verificacion X: mario_right > obj_left AND mario_left <
   obj_right
# Verificacion Y: mario_bottom > obj_top AND mario_top <
   obj_bottom
```

5.2.2 Resolución de Colisiones por Prioridad

El sistema resuelve colisiones en orden específico:

1. **Plataformas verticales:** Se verifica la posición previa (`prev_mario_y`) para determinar si Mario venía desde arriba. Solo permite aterrizaje si `mario_bottom < platform_top` en el frame anterior.
2. **Colisiones laterales:** Cuando hay overlap vertical, se calcula la distancia de penetración desde cada lado y se empuja al personaje por el lado con menor overlap:

```
sub $t3, $t2, $s1    # overlap_left = mario_right -
   platform_left
sub $t5, $t4, $t0    # overlap_right = platform_right -
   mario_x
blt $t3, $t5, push_mario_left
```

3. **Tubos (pipes):** Similar a plataformas pero con detección especial para aterrizaje desde arriba (distancia máxima de 8 píxeles).

5.3. Sistema de Cámara y Scroll

5.3.1 Scroll Horizontal con Dead Zone

La cámara mantiene a Mario centrado con una zona muerta de 80 píxeles desde el borde izquierdo:

```
sub $t2, $t0, $t1       # distancia = mario_x - camera_x
li $t3, 80              # dead zone
ble $t2, $t3, camera_bounds_check
sub $t1, $t0, $t3       # camera_x = mario_x - 80
```

El sistema limita la cámara entre 0 y `WORLD_WIDTH-128`, evitando mostrar áreas fuera del nivel.

5.4. Sistema de Animación

5.4.1 Animación de Sprites Multi-Frame

Mario cuenta con 4 sprites diferentes:

- Frame 1, 2, 3: Caminando (ciclo cada 5 frames)
- Frame jump: En el aire

El sistema detecta automáticamente la dirección mediante `mario_facing_right` y espeja horizontalmente los sprites cuando es necesario:

```
mario_flip_pixel:
    li $t5, 11
    sub $t6, $t5, $s4      # columna_invertida = 11 -
       columna_actual
```

Los Goombas también tienen 2 frames que alternan según `goomba_move_counter`.

5.5. Sistema de Enemigos con IA

5.5.1 Patrullaje con Límites Dinámicos

Cada Goomba tiene definidos límites min/max en la estructura de datos (20 bytes por entidad). El movimiento incluye:

1. **Detección de bordes:** Invierte dirección al alcanzar límites
2. **Detección de tubos:** Usa `goomba_pipe_collision` para evitar atravesar obstáculos
3. **Movimiento por frames:** Solo se mueve cada 2 frames (`goomba_move_delay=2`)

5.5.2 Colisión Mario-Goomba con Zona de Muerte

El sistema diferencia entre pisar y ser golpeado:

```
# Calcular centros verticales
add $t0, mario_y, mario_height/2
add $t2, goomba_y, goomba_height/2

# Si centro de Mario esta arriba Y esta cayendo
bge $t0, $t2, mario_dies_goomba
bgtz mario_vy, kill_goomba_stomp
```

Solo mata al enemigo si Mario cae desde arriba Y la distancia es < 6 píxeles.

5.6. Sistema de Mystery Blocks y Power-ups

5.6.1 Bloques con Estado Persistente

Cada bloque tiene estructura: `(x, y, hit_flag, item_type)`. Al golpear desde abajo:

```
# Verificar golpe: mario_top debe estar MUY cerca de
   block_bottom
sub $t6, mario_top, block_bottom
bgtz $t6, next_mystery      # No hay contacto
li $t7, -10                 # Tolerancia de 10 pixeles
blt $t6, $t7, next_mystery  # Muy lejos
```

5.6.2 Sistema de Power-ups con Física

Los power-ups spawneados tienen física independiente:

- Gravedad: `POWERUP_GRAVITY=1`
- Rebote inicial: `POWERUP_BOUNCE=-6`
- Colisión con suelo para detener caída

Tipos disponibles:

- Estrella (`type=0`): +5000 puntos
- Corazón (`type=1`): +1 vida (máximo 3)

5.7. Optimizaciones de Renderizado

5.7.1 Culling de Entidades

Todas las funciones de dibujo verifican visibilidad antes de renderizar:

```
li $t2, -20             # Margen izquierdo extendido
blt screen_x, $t2, skip_entity
li $t2, 140             # Margen derecho
bge screen_x, $t2, skip_entity
```

5.7.2 Sprites con Transparencia

Los sprites usan color `0x00000000` como transparente, saltando píxeles durante el renderizado:

```
lw $t0, 0($s2)          # Cargar color del sprite
beqz $t0, skip_pixel    # Si es 0, no dibujar
```

6. Análisis del Funcionamiento

6.1. Flujo del Bucle Principal

6.1.1 Ciclo de Ejecución

El juego ejecuta 60 iteraciones por segundo (delay de 1ms por frame), siguiendo esta secuencia:

1. `process_input`: Lee teclado MMIO (`0xFFFF0000`)
2. `update_mario_physics`: Aplica física y actualiza posición
3. `check_mystery_block_hits`: Detecta golpes a bloques
4. `update_powerups`: Física de ítems
5. `check_powerup_collection`: Recolecta power-ups
6. `update_camera`: Ajusta scroll
7. `update_goombas`: IA y movimiento de enemigos
8. `check_goomba_collisions`: Combate con enemigos
9. `check_coin_collisions`: Recolección de monedas
10. `check_pit_death`: Caída al vacío
11. `render_frame`: Dibuja todo en pantalla

6.2. Casos de Prueba Críticos

6.2.1 Prueba 1: Salto Preciso entre Plataformas

Escenario: Mario en `x=190`, debe saltar a plataforma en `x=220, y=20`

Validación:

- Velocidad horizontal en aire permite alcanzar distancia (`AIR_MAX_SPEED=8`)
- `JUMP_VELOCITY=-12` da altura suficiente
- Detección de aterrizaje verifica `prev_mario_y` para evitar atravesar desde abajo

Resultado esperado: Aterrizaje suave sin rebotes

6.2.2 Prueba 2: Colisión con Tubería

Escenario: Mario corriendo hacia pipe en `x=350` (altura 24 píxeles)

Validación:

- Si `mario_bottom` está dentro de 8 píxeles del tope de la pipe: permite aterrizaje
- Si no: resuelve colisión lateral empujando a Mario hacia atrás
- Función `check_pipe_landing` diferencia ambos casos

Resultado esperado: Aterrizaje en tubos bajos, bloqueo en tubos altos

6.2.3 Prueba 3: Eliminar Goomba con Salto

Escenario: Mario salta sobre Goomba en `x=60, y=ground-8`

Validación:

```
mario_center_y = 50 + 6 = 56
goomba_center_y = 56 + 4 = 60
56 < 60     Mario esta arriba
mario_vy = 3     Esta cayendo
distancia = mario_bottom - goomba_top = 64 - 56 = 8
8 > 6     NO elimina (demasiado lejos)
```

Ajuste necesario: Mario debe estar más cerca (< 6 píxeles) del tope del Goomba

6.2.4 Prueba 4: Mystery Block Golpeado Múltiples Veces

Escenario: Mario salta bajo bloque en `x=52, y=8`

Validación:

- Primer golpe: `hit_flag=0` → Spawnea power-up, cambia a sprite "used"
- Segundo golpe: `hit_flag=1` → Ignora colisión (`bnez $t1, next_mystery`)

Resultado esperado: Solo un power-up por bloque

6.2.5 Prueba 5: Caída al Vacío entre Islas

Escenario: Mario en `x=170` (gap entre isla 1 y 2)

Validación:

```
lw $t0, mario_y          # y = 65 (debajo de pantalla)
li $t1, 64
blt $t0, $t1, pit_check_done
jal mario_hit            # Pierde vida y resetea
```

Resultado esperado: Mario pierde vida y reaparece en `x=10, y=50`

6.3. Limitaciones Conocidas

6.3.1 Problema 1: Atravesar Plataformas a Alta Velocidad

Si `mario_vy > 10`, puede atravesar plataformas de 8 píxeles de altura en un solo frame.

Solución: `MAX_FALL_SPEED=6` limita esto.

6.3.2 Problema 2: Colisión Lateral con Esquinas

Al caer cerca del borde de una plataforma, ocasionalmente se resuelve como colisión lateral en lugar de aterrizaje.

Mitigación: Priorización de colisiones verticales.

6.3.3 Problema 3: Goomba Atascado en Tubería

Si los límites de patrulla del Goomba coinciden exactamente con un pipe, puede quedar oscilando.

Solución: Función `goomba_pipe_collision` que detecta y revierte movimiento.

6.4. Validación de Integridad del Sistema

6.4.1 Sincronización Física-Render

El sistema garantiza coherencia mediante ejecución secuencial estricta:

- Física se calcula ANTES de colisiones
- Colisiones se resuelven ANTES de cámara
- Cámara se actualiza ANTES de render

6.4.2 Gestión de Memoria sin Fugas

Todas las entidades usan memoria estática en `.data`. No hay asignación dinámica, eliminando riesgo de memory leaks.

6.4.3 Recuperación de Errores

El sistema implementa reseteo completo con `reset_game`:

- Restaura posiciones iniciales
- Limpia flags de colección (monedas, bloques)
- Reinicia estados de enemigos (`alive=1`)
- Resetea power-ups activos (`x=-1`)

Esto garantiza que cada partida comience en estado limpio y predecible.