CE1106 - DonCEy Kong Jr
Paradigmas de Programación
Escuela de Ingeniería en Computadores
Estudiantes:
Eduardo José Canessa Quesada
Luis Felipe Chaves Mena
Deiler Morera Valverde
Fecha: Noviembre 2025

A continuación se documentan los aspectos fundamentales del proyecto DonCEy Kong Jr desarrollado como parte del curso CE3104 - Paradigmas de programación. Este proyecto implementa una versión simplificada del clasíco videojuego Donkey Kong Jr, utilizando dos paradigmas de programación distintos, **programación imperativa en C** para los clientes (jugadores y espectadores) y **Programación orientada a objetos en java** para el servidor encargado de la lógica del juego.
El sistema integra comunicación por sockets, estructuras de datos personalizadas y elementos básicos de un videojuego como movimiento, colisiones

1. Estructura de datos desarrolladas

1.1. Estructuras para el cliente

En la implementación del cliente, se desarrollan varias estructuras de datos que permiten representar los elementos fundamentales del juego, el jugador, los cocodrilos, las frutas generadoras de puntos, estas estructuras se emplean para almacenar el estado recibido por el servidor de java y después ser representadas en la interfaz gráfica desarrollada con la librería Raylib. A con se presentan las estructuras, con una descripción detallada y su función dentro del sistema.

1. Estructura **Jugador**

```c
typedef struct {
    int x;
    int y;
    int climbing;
    int right;
    int points;
} Jugador;
```

los campos

- **x** y **y** son las coordenadas del jugador dentro del mapa,
- **climbing** indica si el jugador esta subiendo una liana.
- **right** indica si el jugador esta mirando hacia la izquierda o derecha.
- **points** indica el puntaje acumulado por el jugador.

El servidor envía constantemente actualizaciones JSON con el estado del jugador. Estas actualizaciones son procesadas por la función **procesarJSON()** del cliente, que modifica directamente la instancia global **Jugador jugador;** y esta variable es utilizada por el modulo gráfico **DrawSpriteAt()** para dibujar el sprite adecuado según su orientación

2. Estructura **Entidad**

```c
typedef struct {
    int x;
    int y;
    char type[20];
    bool view;
} Entidad;
```

Esta estructura modela los enemigos, los cocodrilos rojos y azules

- **x** y **y** son la posición del enemigo dentro del mapa,
- **type** es el tipo de cocodrilo, rojo o azul

- **view** indica si el cocodrilo esta mirando hacia abajo o hacia arriba.

El cliente mantiene un arreglo dinámico que almacena todas las entidades presentes:

```
        #define MAX_ENTIDADES 100
        Entidad entidades[MAX_ENTIDADES];
        int numEntidades;
```

El servidor java envía cada actualización mediante JSON y el cliente interpreta cada enemigo, lo guarda en el arreglo y luego el módulo gráfico se encarga de dibujarlos con la textura correspondiente según su tipo y orientación.

3. Estructura **Fruta**

```
        typedef struct {
            int x;
            int y;
            char type[20];
            int points;
        } Fruta;
```

Representa los ítems que otorga puntos al jugador.

- **x** y **y** es la posición de la fruta en el mapa.
- **type** tipo de fruta.
- **points** valor en puntos que otorga al ser consumida

El cliente administra otro arreglo:

```
        #define MAX_FRUTAS 100
        Fruta frutas[MAX_FRUTAS];
        int numFrutas;
```

cada fruta recibida desde el servidor se almacena aquí y el módulo gráfico selecciona el sprite adecuado para cada una

1.2. Estructuras para el servidor Java

El servidor implementado en Java utiliza un conjunto de clases y estructuras que representan el estado completo del juego para cada jugador conectado. A diferencia del cliente, que solo almacena una copia simplificada del estado, el servidor mantiene toda la lógica, el mundo del jugador, frutas, las entidades activas y los sistemas de física y colisiones.

A continuación, se describen las estructuras principales utilizadas por el servidor Java, su propósito y la función que cumplen dentro del sistema.

1. **Estructura Entity**

```
public abstract class Entity {
    protected Coords position;

    public Entity(int x, int y) {
        this.position = new Coords(x, y);
    }

    public Coords getPosition() { return position; }
    public void setPosition(Coords p) { this.position = p; }
    public int getX() { return position.getX(); }
    public int getY() { return position.getY(); }
}
```

Esta clase es la base para todas las entidades del juego (jugadores, enemigos y frutas). Sus campos:

- **position**: almacena coordenadas en la grilla del mapa mediante la clase `Coords`.

El servidor utiliza esta clase como estructura fundamental para poder manejar cualquier objeto dentro del mundo independientemente de su tipo.

2. **Estructura Player**

```java
public class Player extends Entity {
    private final int jumpStrength;
    private boolean facingRight;
    private boolean onGround;
    private boolean onVine;
    private boolean dead;
    private int points;
}
```

Campos principales:

- **jumpStrength**: altura máxima del salto.
- **facingRight**: indica si el jugador mira a la derecha.
- **onGround**: si está sobre plataforma sólida.
- **onVine**: si está agarrado a una liana.
- **dead**: estado de muerte.
- **points**: puntaje acumulado.

El servidor actualiza esta estructura en cada ciclo de juego, aplicando gravedad, colisiones y movimientos recibidos desde el cliente. Además, sus valores se envían al cliente mediante JSON para actualizar la interfaz gráfica.

3. **Estructura RedCoco y BlueCoco**

```java
public abstract class Coco extends Entity {
    protected boolean active;
    protected String type;
    protected boolean isFacingDown;
    protected int movementSpeed;
    protected int movementCounter;
}
```

Los cocodrilos son enemigos controlados completamente por el servidor. Campos:

- **active**: indica si el enemigo sigue en juego.
- **type**: tipo de cocodrilo (rojo o azul).
- **movementSpeed**: cada cuántos ticks se mueve.

RedCoco patrulla una liana subiendo y bajando. **BlueCoco** baja por la liana y luego cae hasta desactivarse.

El servidor mantiene listas independientes:

- Para Jugador 1: `List<Coco>cocodrilosJ1;`
- Para Jugador 2: `List<Coco>cocodrilosJ2;`

Estas listas son las que se envían al cliente en cada actualización JSON.

4. **Estructura Fruit**

```java
public class Fruit extends Entity {
    private boolean active;
    private final int points;
    private final String type;
}
```

Campos:

- **type**: tipo de fruta (BANANA, ORANGE, STRAWBERRY...)
- **points**: puntos otorgados al jugador
- **active**: indica si aún puede ser recogida

Las frutas se manejan en listas:

- `List<Fruit>frutasJ1;`

- `List<Fruit>frutasJ2;`

Cuando un jugador colisiona con una fruta, el servidor modifica el puntaje y elimina la fruta de la lista con el sistema de colisiones.

5. **Estructura Coords**

```
public class Coords {
    private final int x;
    private final int y;
}
```

Se utiliza para representar posiciones dentro del mundo. Es inmutable, lo que evita inconsistencias durante la simulación.

6. **Estructura World y enum TileType**

```
public class World {
    private final Map<Coords, TileType> grid;
    private List<Coords> validVinePositions;
}
```

- El servidor mantiene una copia completa del mapa para cada jugador.
- Todo el terreno se almacena en un `HashMap<Coords, TileType>`.
- `TileType` define si una celda es plataforma, liana, agua, vacío, etc.

Esto permite simulaciones personalizadas: cada jugador tiene su propio mundo aislado.

7. **Estructura CollisionSystem y GravitySystem**

Ambos sistemas actúan directamente sobre las estructuras del jugador y los enemigos.

- **CollisionSystem**: detecta colisiones con cocodrilos, plataformas, lianas y frutas.
- **GravitySystem**: aplica la gravedad al jugador mientras no esté en una liana o plataforma.

Estas clases no representan datos, sino que modifican las estructuras previamente descritas.

8. **Estructura Server** y **AdapterJSON**

El servidor gestiona conexiones, colas de mensajes y la serialización del estado del juego:

```
Map<String, Socket> jugadores;
Map<String, List<Socket>> espectadoresPorJugador;
List<String> mensajes_j1;
List<String> mensajes_j2;
AdapterJSON adapter;
```

- **jugadores**: relación nombre → socket.
- **espectadoresPorJugador**: espectadores asociados a cada jugador.
- **mensajes_j1, mensajes_j2**: cola de movimientos pendientes por procesar.
- **AdapterJSON**: convierte jugadores, frutas y cocodrilos en JSON para enviarlos al cliente.

2. Algoritmos desarrollados

2.1. Algoritmos desarrollados para el cliente

1. **Algoritmo de comunicación cliente-servidor mediante sockets**
   Propósito: La función **clientLoop** tiene como propósito establecer y mantener la comunicación entre el cliente en C y el servidor Java.

```
ALGORITMO ComunicacionClienteServidor()

    Crear socket TCP

    Configurar direccion del servidor (IP, puerto)

    Intentar conectar con servidor
    SI la conexion falla:
        Mostrar error y finalizar

    // Recepción de bienvenida
    bienvenida_completa <- FALSO
    MIENTRAS bienvenida_completa = FALSO:
        mensaje <- recibirDatos()
        Mostrar mensaje
        SI mensaje contiene "Conexión Exitosa.":
            bienvenida_completa <- VERDADERO

    // Envío del tipo de cliente (Jugador / Espectador)
    enviarDatos(tipoCliente)

    // Bucle de comunicación principal
    MIENTRAS cliente_activo:

        Esperar actividad en:
            - socket del servidor
            - teclado del usuario (stdin)

        SI hay datos desde el servidor:
            mensaje <- recibirDatos()
            procesarJSON(mensaje)

        SI el usuario presiona una tecla:
            tecla <- leerTecla()
            accion <- convertirTeclaAComando(tecla)
            enviarDatos(accion)

FIN ALGORITMO
```

   Su funcionamiento se desarrolla en cuatro etapas principales: primero crea y configura un socket TCP para iniciar la conexión con el servidor. Luego recibe el mensaje de bienvenida y envía la información necesaria para registrar el tipo de cliente que se conecta. Posteriormente, entra en un ciclo de comunicación controlado por **select()**, que permite atender simultáneamente los mensajes enviados por el servidor y las teclas presionadas por el usuario sin bloquear la ejecución. Finalmente, cuando la sesión termina, cierra el socket y restaura el modo habitual del terminal, completando así el proceso de comunicación.

2. **Manejo de teclado**
   Propósito: Permitir que el cliente capture teclas individuales en tiempo real sin requerir que el usuario presione la tecla **enter**.

```
    ALGORITMO set_input_mode()
        Obtener configuración del teclado
        Desactivar modo canónico y eco
        Aplicar configuración
    FIN

    ALGORITMO reset_input_mode()
        Obtener configuración del teclado
        Activar modo canónico y eco
        Aplicar configuración
    FIN
```

   La función **set_input_mode** modifica la configuración del terminal para habilitar un modo de entrada no canónico, deshabilitando el eco de caracteres y permitiendo que cada tecla sea procesada inmediatamente. Esto permite detectar acciones del jugador sin necesidad de presionar **enter**. Su inversa, **reset_input_mode**, restaura la configuración original del terminal al finalizar la ejecución del cliente. En conjunto, ambas funciones encapsulan el algoritmo necesario para gestionar una entrada de teclado rápida, directa y adecuada para la jugabilidad del sistema.

3. **Procesamiento del estado del juego en formato JSON**
propósito: Interpretar el mensaje **JSON** enviado por el servidor, extraer la información del jugador, las entidades enemigas, las frutas y actualizar las estructuras internas del cliente para sincronizar el estado del juego.

```
ALGORITMO ProcesarJSON(json)

    // Procesar jugador
    SI existe sección "jugador" en json:
        leer x, y, puntos
        leer estados climbing y right
        actualizar estructura Jugador

    // Procesar entidades (cocodrilos)
    numEntidades <- 0
    SI existe lista "entidades" en json:
        PARA cada entidad en la lista:
            leer x, y, tipo y orientación
            agregar entidad a arreglo entidades

    // Procesar frutas
    numFrutas <- 0
    SI existe lista "frutas" en json:
        PARA cada fruta en la lista:
            leer x, y, tipo y puntos
            agregar fruta a arreglo frutas

FIN
```

La función **procesarJSON** tiene 3 etapas, primero identifica y extrae la sección correspondiente al jugador, obteniendo su posición, dirección, estado de trepado y puntaje. Luego recorre el arreglo JSON de entidades, interpretando cada cocodrilo y almacenando su tipo, orientación y posición dentro de un arreglo estructurado. Finalmente procesa la lista de frutas, extrayendo sus coordenadas, tipo y valor en puntos. Al finalizar, las estructuras globales del cliente contienen una representación completa y actualizada del estado del juego, lista para ser dibujada por el módulo gráfico.

4. **Renderizado gráfico**
Propósito: Dibujar en pantalla el estado más reciente del juego, representando el mapa, el jugador, los enemigos y las frutas mediante sprites 2D, actualizados en cada ciclo de dibujo para reflejar lo que envía el servidor en tiempo real.

```
ALGORITMO RenderizarPantalla

    MIENTRAS la ventana esté abierta:

        Iniciar dibujo y limpiar pantalla

        Dibujar mapa completo

        // Dibujar jugador
        SI jugador está trepando:
            dibujar sprite de trepar
        SINO SI está mirando a la derecha:
            dibujar sprite derecha
        SINO:
            dibujar sprite izquierda

        // Dibujar enemigos
        PARA cada entidad:
            elegir sprite según tipo y orientación
            dibujar entidad

        // Dibujar frutas
        PARA cada fruta:
            elegir sprite según tipo
            dibujar fruta

        Finalizar dibujo del frame

FIN
```

El algoritmo de renderizado opera en tres fases claramente definidas. Primero limpia la pantalla y dibuja el mapa base utilizando un recorrido completo por la matriz del escenario. Luego representa los elementos dinámicos del juego: el jugador, los cocodrilos, las frutas y el puntaje actual del jugador, seleccionando de manera condicional el sprite adecuado según los atributos almacenados en sus estructuras de datos. Finalmente actualiza el frame mediante **EndDrawing()**, logrando así

un ciclo de renderizado continuo que produce una representación gráfica coherente y sincronizada del estado del juego.

5. **Manejo Multihilo del Cliente**
Proposito: Separar la lógica de comunicación del servidor y el proceso de renderizado gráfico en dos hilos independientes, asegurando que el juego reciba actualizaciones en tiempo real sin afectar el rendimiento ni bloquear la interfaz visual del cliente.

```
ALGORITMO ManejoMultihilo()

    // Hilo secundario: comunicación con servidor
    Crear hilo que ejecuta clientLoop()

    // Hilo principal: renderizado del juego
    MIENTRAS la ventana esté abierta:

        Iniciar dibujo y limpiar pantalla

        Dibujar mapa
        Dibujar jugador según su estado
        Dibujar todas las entidades
        Dibujar todas las frutas

        Finalizar dibujo del frame

FIN
```

El cliente implementa un modelo multihilo donde la función clientLoop se ejecuta en un hilo independiente encargado exclusivamente de manejar la comunicación con el servidor. Este hilo procesa mensajes, interpreta el JSON recibido y actualiza las estructuras de datos compartidas. Paralelamente, el hilo principal ejecuta el ciclo de renderizado gráfico, dibujando el mapa, el jugador, las entidades y las frutas utilizando los valores actualizados por el hilo de comunicación. Esta separación permite que la recepción de datos y la actualización visual ocurran de manera simultánea y fluida, evitando bloqueos y garantizando una experiencia de juego estable.

2.2. Algoritmos desarrollados para el servidor

A continuación se detallan los algoritmos más importantes implementados en el servidor Java, los cuales permiten administrar las conexiones, procesar los movimientos del jugador, actualizar el mundo y generar la información enviada al cliente.

1. **Procesamiento de movimientos del jugador**
Propósito: Interpretar el comando numérico enviado por el cliente y actualizar la posición del jugador verificando colisiones, gravedad y restricciones del entorno.

```
ALGORITMO procesarMovimientoJugador(mensaje, jugador, colision, gravedad, cocos, frutas)
    SI jugador es nulo O esta muerto ENTONCES retornar
    convertir mensaje a entero (movimiento)
    SEGUN movimiento HACER
        1: mover jugador hacia arriba (salto o subir liana)
        2: mover jugador a la derecha
        3: mover jugador hacia abajo si esta en liana
        4: mover jugador a la izquierda
    FIN SEGUN
    actualizar estado del jugador mediante CollisionSystem
    enviar mensaje de confirmacion y estado actualizado al cliente
FIN
```

Este algoritmo es uno de los núcleos del servidor. Cada acción enviada por el cliente se traduce en movimiento, salto o interacción con el entorno. Posteriormente, se actualiza el estado mediante colisiones y se envía la respuesta al cliente.

2. **Actualización de enemigos**
Propósito: Ejecutar el comportamiento autónomo de cada enemigo, ya sea patrullaje o caída, y remover aquellos que ya no están activos.

```
ALGORITMO actualizarCocodrilos()
    PARA cada cocodrilo en la lista
        SI cocodrilo esta activo ENTONCES
            llamar cocodrilo.update(world)
        FIN SI
    FIN PARA
    eliminar cocodrilos inactivos de la lista
FIN
```

Cada uno ejecuta su propia logica de movimiento y se desactiva automáticamente cuando deja de ser relevante.

3. **Algoritmo de gravedad del jugador**
   Propósito: Simular la caída del jugador cuando no está en plataforma ni agarrado a una liana, aplicando aceleración controlada.

```
ALGORITMO applyGravity(jugador)
    SI jugador esta muerto ENTONCES retornar
    SI jugador esta en el suelo O en liana ENTONCES
        reiniciar velocidad de caida
        retornar
    FIN SI
    incrementar contador de gravedad
    SI no corresponde aplicar este tick ENTONCES retornar
    aumentar velocidad de caida hasta un maximo
    calcular posicion debajo del jugador
    SI colisionSystem.canMoveTo(posAbajo) ENTONCES
        mover jugador hacia abajo
    FIN SI
    actualizar estado mediante CollisionSystem
FIN
```

Se encarga de que la física sea consistente: caída natural, aceleración progresiva y respeto por colisiones con el escenario.

4. **Algoritmo de colisiones del jugador**
   Propósito: Determinar si el jugador está en liana, suelo, agua, cerca de un enemigo o sobre una fruta, y actualizar su estado o puntaje.

```
ALGORITMO updatePlayerState(jugador, enemigos, frutas)
    SI jugador muerto ENTONCES retornar
    obtener posicion actual
    jugador.onVine  = hay liana en esa posicion
    jugador.onGround = hay plataforma justo debajo
    SI tile actual es mortal ENTONCES jugador.muere()
    PARA cada cocodrilo activo
        SI posiciones coinciden ENTONCES jugador.muere()
    FIN PARA
    PARA cada fruta activa
        SI posiciones coinciden ENTONCES
            sumar puntos
            desactivar fruta
        FIN SI
    FIN PARA
FIN
```

Este algoritmo unifica todas las interacciones del jugador con el mundo: muerte, recolección y estado ambiental.

5. **Generación de estado para el cliente (JSON)**
   Propósito: Convertir todo el estado del juego (jugador, enemigos, frutas y espectadores) en una cadena JSON que será enviada al cliente C.

```
ALGORITMO generarJSON(jugador, frutas, cocos, nombre, espectadores)
    iniciar objeto JSON
    agregar datos del jugador:
        x, y, puntos, orientacion, si esta en liana
    agregar lista de cocos:
        tipo, x, y, orientacion
    agregar lista de frutas:
        tipo, x, y, puntos
    agregar datos adicionales:
        nombre del jugador, cantidad de espectadores
    devolver JSON como String
FIN
```

Este algoritmo es esencial: permite sincronizar el estado entre servidor y cliente de manera eficiente y ligera. El cliente usa este JSON para actualizar gráficos y lógica.

6. **Gestión de conexión y creación de mundos**
   Propósito: Detectar conexiones o desconexiones, instancia de los mundos y crear los sistemas físicos para cada jugador.

```
ALGORITMO gestionarJugadores()
    llamar gestionarConexionJugador1()
    llamar gestionarConexionJugador2()
    llamar limpiarJugadoresDesconectados()
FIN
```

Este algoritmo se encarga coordina todo el ciclo de vida de los jugadores: creación del mundo, activación, desconexión y limpieza.

3. Problemas sin solución

A pesar del correcto funcionamiento general del proyecto, durante el desarrollo surgieron ciertos inconvenientes que no lograron resolverse por completo dentro del tiempo y alcance establecidos. Estos problemas no afectan de manera crítica la jugabilidad, pero sí representan limitaciones técnicas que podrían abordarse en versiones futuras para mejorar la experiencia y la eficiencia del sistema. Entre los principales aspectos pendientes se destacan los siguientes:

1. **Fluidez del movimiento matricial.** Debido a que el desplazamiento se realiza por bloques discretos, el jugador puede percibir ligeros "saltos" en la animación del movimiento. La incorporación de interpolación o transiciones suavizadas hubiese requerido una reestructuración significativa del motor gráfico.

2. **Implementación de animaciones en los sprites.** La inclusión de ciclos de animación resultó problemática, ya que interfería con la lógica gráfica establecida en el cliente en C. Lograr un sistema de animación robusto implicaba ajustes importantes en el manejo de texturas y estados.

3. **Sincronización entre comunicación y renderizado.** En situaciones puntuales, algunos eventos de red no se reflejan visualmente con la precisión temporal deseada, generando pequeños desfases. Optimizar esta sincronización exige un manejo más avanzado de hilos y control de tiempos.

Estos puntos representan oportunidades de mejora que podrían abordarse en fases posteriores del proyecto, con el objetivo de refinar aún más el rendimiento, la estabilidad y la calidad visual del sistema.

4. Plan de Actividades

Cuadro 1: Plan de Actividades del Proyecto (Planificación Inicial)

No.	Actividad	Responsable(s)	Tiempo
1	Investigación inicial del juego, requisitos y comunicación C–Java	Todos	1 día
2	Diseño preliminar del cliente y servidor (arquitectura general)	Eduardo, Luis	1 día
3	Diseño de estructuras de datos (Jugador, Entidad, Fruta, etc.)	Deiler	1 día
4	Implementación del mapa y carga gráfica (Raylib)	Eduardo	2 días
5	Implementación del servidor Java: lógica del juego y clases base	Luis	2 días
6	Implementación del cliente C: conexión mediante sockets	Deiler	2 días
7	Desarrollo del sistema de comunicación JSON entre servidor y cliente	Luis, Deiler	2 días
8	Implementación de control del jugador (movimiento, trepar, colisiones)	Eduardo	1 día
9	Implementación de creación y movimiento de cocodrilos en el servidor	Luis	1 día
10	Implementación de frutas (creación, puntos y eliminación)	Eduardo	1 día
11	Integración completa cliente–servidor y pruebas básicas	Todos	2 días
12	Pruebas de rendimiento, corrección de errores y mejora de sincronización	Todos	2 días
13	Redacción de la documentación: estructuras de datos	Deiler	1 día
14	Redacción de algoritmos implementados y problemas encontrados	Eduardo	1 día
15	Elaboración de conclusiones, recomendaciones y bibliografía	Luis	1 día
16	Revisión final, edición y entrega del documento y del proyecto completo	Todos	1 día

5. Problemas Encontrados

Durante el desarrollo del proyecto se identificaron diversos desafíos técnicos y de diseño que, mediante análisis y ajustes progresivos, lograron resolverse satisfactoriamente. A continuación, se presentan los principales problemas enfrentados y la solución implementada en cada caso:

1. **Interoperabilidad entre lenguajes.** La conexión entre el servidor implementado en Java y el cliente desarrollado en C presentó dificultades iniciales debido a diferencias en manejo de sockets y formatos de datos. Esto se resolvió estandarizando el protocolo de comunicación y utilizando estructuras de mensaje consistentes.

2. **Selección de la librería gráfica.** La elección de una librería adecuada para C resultó clave. Tras evaluar varias opciones, **Raylib** se identificó como la alternativa más eficiente por su sencillez, portabilidad y facilidad de integración con código existente, lo que permitió acelerar el desarrollo gráfico.

3. **Estabilidad de la lógica del juego.** La construcción de una mecánica estable requirió depuración constante de eventos, ciclos de actualización y validaciones internas. Se implementaron controles adicionales para asegurar coherencia en los estados del jugador, enemigos y elementos del entorno.

4. **Gestión de múltiples clientes.** Se estableció un límite de dos jugadores activos y dos espectadores por jugador, para un total máximo de seis conexiones simultáneas. Para lograrlo, se diseñó una estructura de control de sockets que asigna roles y gestiona permisos según el tipo de usuario.

5. **Conectividad en red local.** La conexión a través de Wi-Fi implicó configurar direcciones **IPv4** y asegurar que todos los dispositivos se encontraran en la misma red. Se resolvió mediante la identificación y uso de direcciones locales fijas para el servidor durante las pruebas.

6. **Arquitectura del sistema.** Crear una arquitectura estable que separara adecuadamente la lógica del servidor, la capa gráfica y la administración de sockets fue un reto inicial. La solución consistió en modularizar el código y definir responsabilidades claras para cada componente.

7. **Organización de recursos.** La administración de carpetas y archivos, especialmente para sprites, texturas y niveles, requería un control riguroso. Se optó por una estructura jerárquica estandarizada que facilitó la carga dinámica de recursos.

8. **Selección de sprites.** Encontrar sprites compatibles en estilo, tamaño y legibilidad fue un desafío. Esto se resolvió unificando las dimensiones y ajustando manualmente aquellos elementos que no coincidían con la estética general del juego.

9. **Modelado de objetos.** La creación de un modelo claro para cada tipo de entidad (jugador, enemigo, frutas, paredes, etc.) requirió definir propiedades comunes y comportamientos particulares. La implementación final permitió un manejo más simple y coherente de cada objeto dentro del motor del juego.

10. **Fluidez y carga de la matriz.** La representación matricial en el cliente en C debía actualizarse rápidamente sin afectar el rendimiento. Se optimizó el proceso de carga y refresco de la matriz, reduciendo tiempos de espera y mejorando la fluidez percibida.

11. **Transmisión de mensajes.** El envío y recepción de información se resolvió utilizando mensajes en formato **JSON**, lo que facilitó la serialización, la lectura de datos y la expansión futura del protocolo gracias a su estructura clara y flexible.

6. Conclusiones

1. El modelo Cliente-Servidor implemetado resultó adecuado para aun videojuego ligero, permitiendo mantener la lógica principal en el servidor mientras el cliente, se encarga de renderizar. Esto redujo la complejidad del cliente, permitiendo mejorar la estabilidad general del sistema.

2. La separación entre mundo, entidades, sistemas de física permitió un diseño modular, permitiendo que cada componente puede modificarse o ampliarse sin afectar el resto del sistema.

3. El uso de la librería Raylib en el cliente permitió una interfaz gráfica simple pero eficiente, esto brindo una buena integración con **C** y simplificando el manejo de ventanas, texturas y las entradas del usuario.

4. El proyecto permitió fortalecer conocimientos fundamentales sobre la programación concurrente, redes y videojuegos, esto integrando hilos, sockets, programación orientada a objetos.

5. El uso de JSON como formato de intercambio demostró ser eficiente, permitiendo transmitir el estado del juego entre el servidor o el cliente (modo adaptador entre los lenguajes) de forma clara y fácilmente extensible a futuro.

7. Recomendaciones

- **Considerar un formato de datos más eficiente.** Aunque el uso de JSON facilitó la comunicación entre el cliente desarrollado en C y el servidor implementado en Java, este formato puede resultar costoso en términos de tiempo de procesamiento, especialmente cuando se transmiten múltiples entidades o actualizaciones frecuentes del estado del juego. Por ello, se recomienda evaluar el uso de formatos más compactos y eficientes como *Protocol Buffers* o incluso estructuras binarias personalizadas. Estas alternativas permitirían disminuir la latencia, reducir el tamaño de los mensajes y mejorar el rendimiento general del sistema en escenarios de mayor carga o futuras expansiones del proyecto.

- **Preparar el código para expansiones futuras.** El proyecto cuenta con una base sólida, pero se recomienda aplicar principios de diseño modular para facilitar su crecimiento. Separar responsabilidades en capas más definidas (por ejemplo: lógica de juego, renderizado, comunicación y manejo de datos) permitiría añadir nuevas funcionalidades sin alterar el núcleo del sistema. Asimismo, sería conveniente abstraer elementos como enemigos, frutas o plataformas dentro de estructuras o clases más generales, lo que habilitaría la creación de nuevos elementos del juego o niveles adicionales sin necesidad de modificar secciones extensas del código ya existente.

- **Buscar mejores librerías para la parte gráfica.** Aunque Raylib resultó ser una librería accesible, liviana y adecuada para los requerimientos del proyecto, su capacidad gráfica es limitada para desarrollos más avanzados. En caso de que se desee ampliar el juego con animaciones más fluidas, físicas complejas o soporte multiplataforma más robusto, se recomienda evaluar librerías gráficas

como SDL2, SFML o motores como Godot y Unity. La selección de una herramienta gráfica más completa permitiría mejorar la calidad visual del juego y ofrecer una experiencia más inmersiva en caso de que el proyecto evolucione hacia objetivos más ambiciosos.

8. Bibliografía

Referencias

[1] Oracle. *Java SE Documentation*. Disponible en: `https://docs.oracle.com/javase/`

[2] Oracle. *Java Networking (Sockets) Tutorial*. Disponible en: `https://docs.oracle.com/javase/tutorial/networking/sockets/`

[3] DevDocs. *C Language Documentation*. Disponible en: `https://devdocs.io/c/`

[4] Raylib Technologies. *Raylib Official Documentation*. Disponible en: `https://www.raylib.com/`

[5] Valladolid, R. *Raylib: Learn Videogame Programming in C*. Raylib Technologies, 2023.

[6] Bray, T. *The JavaScript Object Notation (JSON) Data Interchange Format (RFC 8259)*. IETF, 2017. Disponible en: `https://www.rfc-editor.org/rfc/rfc8259`