UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

Introducción a la Computación Gráfica

Edición 2025

Laboratorio 1
Transformar juego de 2D a 3D

Docentes:

Pedro Piñeyro

Sandro Moscatelli

Integrantes:

Daniel García - 5.066.865-4

Valentín Dutra - 5.231.787-7

Benjamín Montenegro - 4.925.516-3

Índice

1. Descripción del juego

El juego se compone de tres niveles. El objetivo es guiar al gusano hasta el portal, pero para lograrlo deberá comer todas las manzanas del nivel correspondiente y evitar caer al vacío.

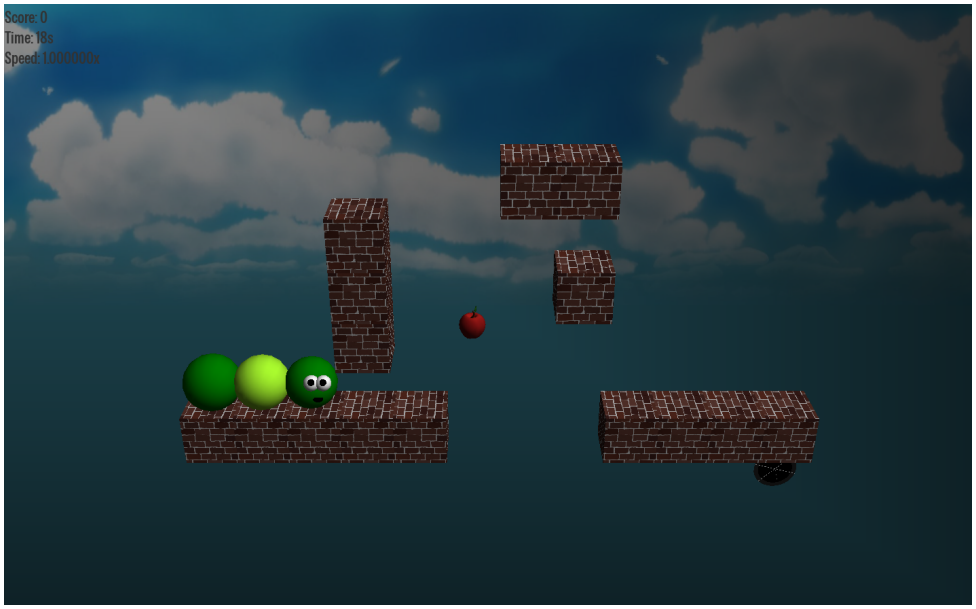

Figura 1: Nivel 1

1.1. Controles

El movimiento del gusano se controla con las flechas del teclado. La tecla **v** permite alternar entre las distintas cámaras disponibles: vista isométrica, cámara libre[1] y tercera persona.

Con la tecla **t** se activa o desactiva el renderizado de texturas, mientras que la tecla **w** cambia al modo *wireframe*.

Las teclas del **1** al **4** ajustan la velocidad del juego. La tecla **l** cambia la posición de la luz, y con **k** se modifica su color.

La tecla **p** pone el juego en pausa o lo reanuda, y finalmente, con la tecla **q** se cierra el juego.

2. Arquitectura de la aplicación

En la figura 2 se muestra un esquema general de las clases utilizadas en la aplicación.

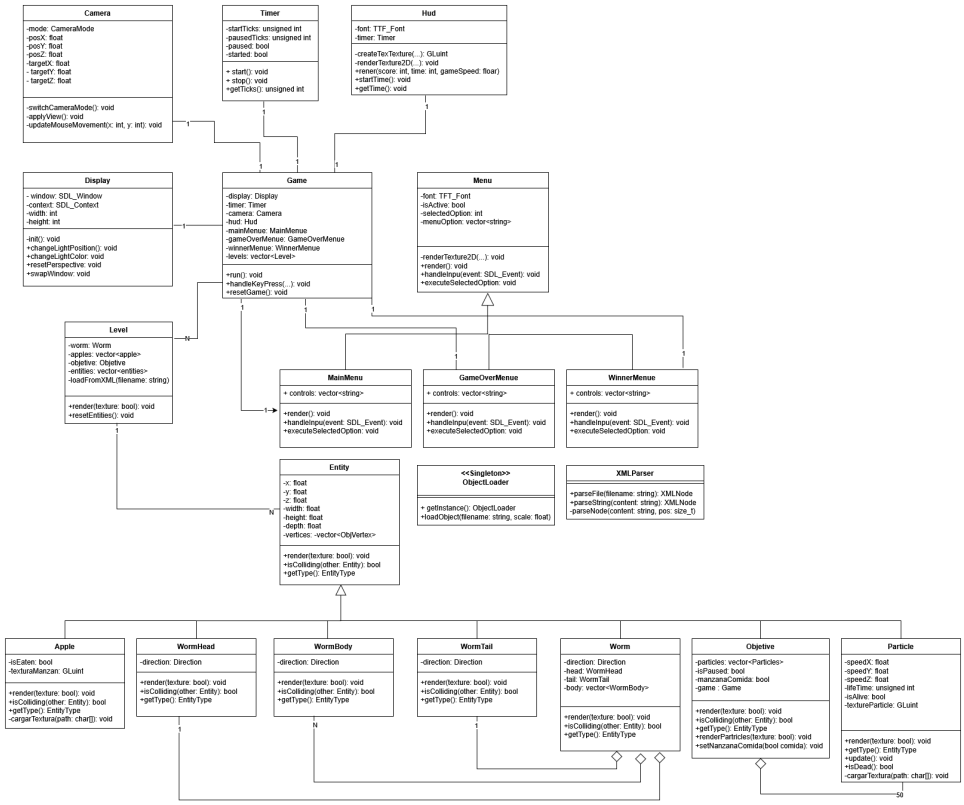

Figura 2: Diagrama de clases

2.1. Componentes principales

2.1.1. Clase `Game`

La clase `Game` actúa como el núcleo principal de la aplicación. Esta clase es responsable de inicializar todos los componentes del juego, manejar los distintos estados (menú, juego en curso, pausa, fin del juego, victoria), gestionar la lógica de avance entre niveles y ejecutar el bucle principal del juego.

Funciones principales

- **`run()`**: Ejecuta el bucle principal del juego, que:
  - Maneja los eventos del usuario.
  - Actualiza el estado del juego (movimiento del gusano, detección de colisiones, progreso entre niveles).
  - Controla la cámara y el renderizado.

    - Administra la transición entre estados (menú, jugando, pausa, game over, victoria).

- **`handleKeyPress(...)`**: Gestiona la entrada del teclado. Permite controlar al gusano, cambiar el modo de cámara, activar o desactivar texturas, cambiar parámetros de iluminación y modificar la velocidad del juego.

Estados del juego El flujo del juego se controla mediante una máquina de estados que incluye:

- `MENU`: Se muestra el menú principal con opciones para jugar o salir.
- `PLAYING`: El juego está en curso.
- `PAUSED`: El juego se encuentra pausado.
- `GAME_OVER`: El jugador ha perdido; se muestra el menú de fin de juego.
- `WINNER`: El jugador ha ganado; se muestra el puntaje y el tiempo final.

Resumen funcional Es una clase que mantiene todo el juego funcionando, desde el menú hasta el *gameplay*, pasando por todos los estados posibles del juego.

2.1.2. Clase `Level`

La clase `Level` está encargada de representar y gestionar toda la información y el comportamiento asociado a un nivel del juego. Cada instancia de esta clase carga los datos del nivel desde un archivo XML y crea las entidades correspondientes: gusano, manzanas, bloques y objetivo.

Funciones principales

- **Constructor `Level(...)`**: Carga los datos del nivel desde un archivo `.xml`.
- **`initialize()`**: Inicializa las entidades del nivel (gusano, manzanas y objetivo) en base a la información cargada previamente.
- **`loadFromXML(...)`**: Carga los datos del nivel desde un archivo XML, utilizando un *parser* personalizado. Extrae información sobre la posición y longitud inicial del gusano, las manzanas, el objetivo y los bloques del escenario.

Resumen funcional Se encarga de la lógica de construcción, administración y renderizado de un nivel del juego. Su diseño está orientado a datos, permitiendo definir niveles fácilmente a través de archivos `XML`. Esto facilita la expansión del juego y la edición de niveles sin modificar el código fuente. Además, centraliza la relación entre el gusano, las manzanas, los bloques del escenario y el objetivo final del nivel.

2.1.3. Clase `Entity`

La clase `Entity`, la cual sirve como una clase base para la representación de objetos en un entorno 3D, probablemente dentro de un contexto de juego.

Funciones principales

- **Constructores**:
  - Constructor principal: Inicializa una entidad con su posición (x, y, z) y sus dimensiones $(width, height, depth)$. Además, guarda las posiciones iniciales como previas $(prevX, prevY, prevZ)$.
  - Constructor con carga de modelo 3D: Permite especificar una ruta a un archivo `.obj` para cargar los vértices del modelo utilizando un `ObjectLoader`.
- **Detección de colisiones (`isColliding`)** [2, 1]:
  - Recibe un puntero a otra entidad (`other`) para verificar si existe una colisión.
  - Define márgenes de colisión específicos (`WALL_MARGIN`, `OBJECTIVE_MARGIN`, `APPLE_MARGIN`, `WORM_MARGIN`) para diferentes tipos de entidades.
  - Ajusta los márgenes (`thisMargin`, `otherMargin`) basándose en el `EntityType` de las entidades involucradas (WORM, APPLE, OBJECTIVE).
  - Calcula los límites (izquierda, derecha, arriba, abajo) de ambas entidades considerando sus posiciones y los márgenes aplicados.
  - Verifica la superposición en los ejes X e Y, incorporando un pequeño `COLLISION_MARGIN` para manejar la precisión.
  - Retorna `true` si hay superposición en ambos ejes, lo que indica una colisión.

Resumen funcional La clase `Entity` es la base de todos los objetos del juego. Controla dónde están los objetos, cómo se mueven, cuándo chocan entre sí y cómo se ven en 3D. Cada objeto (gusano, manzana, pared) sigue las reglas de `Entity` para funcionar correctamente en el juego.

2.1.4. Clase `Apple`

La clase `Apple`, que hereda de `Entity` y representa una manzana en el entorno 3D del juego. Gestiona la carga de su modelo 3D, su textura y su estado (si ha sido comida o no).

Funciones principales

- **`render(bool texture)`**: Renderiza la manzana en la escena 3D.
- **`setEaten(bool eaten)`**: Establece el estado de la manzana a comida (‘true‘) o no comida (‘false‘).

Resumen funcional La clase `Apple` es una manzana que el gusano puede comer en el juego. Hereda de `Entity` y tiene características especiales: puede cargar su modelo 3D y su textura para verse como una manzana real, sabe si ya fue comida o no, y puede mostrarse en el juego con textura o con un color rojo simple.

2.1.5. Clase `WormHead`

La clase `WormHead`, que hereda de `Entity` y representa la cabeza del gusano en el juego. Esta clase es responsable de la lógica de renderizado específica de la cabeza, incluyendo animaciones de los ojos y la boca, así como el control de su dirección.

Funciones principales

- **`render(bool texture)`**: Renderiza la cabeza del gusano en la escena 3D, incluyendo animaciones de ojos y boca.

Resumen funcional La clase `WormHead` es la cabeza del gusano en el juego. Hereda de `Entity` y se encarga de darle vida al gusano con detalles como ojos que parpadean y pupilas que se mueven según la dirección. La cabeza controla hacia dónde mira el gusano

(arriba, abajo, izquierda o derecha) y se mueve de forma suave gracias a la interpolación. Es la parte más visible y expresiva del gusano, haciendo que el juego sea más atractivo visualmente.

2.1.6. Clase `WormBody`

La clase `WormBody`, que hereda de `Entity` y representa un segmento del cuerpo del gusano en el juego. Su función principal es la de ser un componente visual y lógico del gusano, manejando su posición y dirección.

Resumen funcional La clase `WormBody` es cada parte del cuerpo del gusano. Hereda de `Entity` y se encarga de mostrar cada segmento como una esfera verde claro. Cada parte del cuerpo sabe en qué dirección está y se mueve de forma suave gracias a la interpolación.

2.1.7. Clase `WormTail`

La clase `WormTail`, que hereda de `Entity` y representa la cola del gusano en el juego. Su principal función es la de ser un componente visual del gusano, manejando su posición y dirección de forma análoga a otras partes del cuerpo.

Resumen funcional La clase `WormTail` es la cola del gusano. Hereda de `Entity` y se encarga de mostrar el último segmento del gusano como una esfera verde. La cola se mueve de forma suave siguiendo al resto del cuerpo y tiene un pequeño desplazamiento que hace que parezca que se arrastra. Es la parte final del gusano que completa su apariencia y movimiento.

2.1.8. Clase `Objective`

La clase `Objective`, que hereda de `Entity` y representa el objetivo final en un nivel del juego. Esta clase es responsable de su renderizado visual, incluyendo una animación de rotación y un sistema de partículas, así como la gestión de su estado (pausado o si las manzanas han sido comidas).

Funciones principales

- **`renderParticles(bool texture)`**: Gestiona y renderiza el sistema de partículas del objetivo.

Resumen funcional La clase `Objective` es la meta que el jugador debe alcanzar en cada nivel. Hereda de `Entity` y se muestra como un objeto que gira constantemente. Cuando el jugador come todas las manzanas, el objetivo muestra un efecto de partículas que indica que se puede alcanzar.

2.1.9. Clase `Particle`

La clase `Particle`, que hereda de `Entity` y representa una partícula individual en el sistema de partículas del juego. Cada partícula tiene un tiempo de vida, una velocidad aleatoria y una textura asignada para su renderizado.

Funciones principales

- `update()`: Actualiza el estado de la partícula en cada fotograma.
  - Si la partícula está viva, actualiza su posición sumando sus velocidades ($speedX, speedY, speedZ$) a sus coordenadas.
  - Decrementa su tiempo de vida (‘lifeTime‘).
  - Si el tiempo de vida llega a cero o menos, la partícula se marca como muerta (‘isAlive = false‘).

Resumen funcional La clase `Particle` es un pequeño elemento visual que se usa para crear efectos especiales en el juego. Hereda de `Entity` y cada partícula se mueve de forma aleatoria en diferentes direcciones. Tiene un tiempo de vida limitado y puede mostrar diferentes texturas (roja, verde o azul). Se usa principalmente para crear efectos visuales, como cuando el objetivo está activo, haciendo que el juego se vea más atractivo y dinámico.

2.1.10. Clase `Wall`

La clase `Wall`, que representa las paredes en el juego 3D.

Funciones principales

Resumen funcional La clase `Wall` es esencial para construir el laberinto o los límites del juego. Permite que las paredes se muestren correctamente en 3D con su modelo y textura, formando los obstáculos para el jugador.

2.1.11. Clase `Display`

La clase `Display`, que gestiona la ventana de SDL y el contexto de OpenGL para el renderizado del juego. También se encarga de la configuración inicial de la cámara, la iluminación y de la gestión del intercambio de buffers.

Funciones principales

- **Constructor `Display()`**: Inicializa SDL y crea la ventana de juego y el contexto de OpenGL asociado. Configura el tamaño de la ventana y maneja los errores que puedan surgir durante la inicialización de SDL y OpenGL. Inicializa los índices para el control de la posición y el color de la luz.
- **`init()`**: Realiza la configuración inicial de la matriz de proyección (perspectiva de la cámara) y la vista del modelo. Habilita funcionalidades clave de OpenGL como la prueba de profundidad (`GL_DEPTH_TEST`), la iluminación (`GL_LIGHTING`, `GL_LIGHT0`) y el material de color (`GL_COLOR_MATERIAL`). Establece propiedades de luz ambiental y difusa, y posiciona una luz inicial en la escena 3D. La cámara se sitúa para proporcionar una vista diagonal desde arriba.
- **`swapWindow()`**: Intercambia los buffers de la ventana de SDL, mostrando el contenido que ha sido renderizado en el buffer trasero al buffer frontal, lo que genera una animación fluida en pantalla.

Resumen funcional La clase `Display` es la encargada de mostrar todo lo que vemos en el juego. Se encarga de crear la ventana del juego y configurar cómo se ven las cosas en 3D. Controla la iluminación del juego, permitiendo cambiar dónde está la luz y de qué color es. También maneja la cámara y cómo se ve el juego desde diferentes ángulos.

2.1.12. Clase `Camera`

La clase `Camera`, que controla la vista del jugador en el juego 3D, ofreciendo varios modos de visualización.

Funciones principales

- `updateMouseMovement(int x, int y)`: Permite mover la cámara libremente ajustando su orientación con el movimiento del mouse.
- `switchCameraMode()`: Alterna entre los diferentes modos de cámara disponibles.

Resumen funcional La clase `Camera` es la que controla cómo vemos el juego. Permite cambiar entre tres formas de ver el juego: una vista normal fija, una vista que sigue al gusano y una vista libre que podemos controlar con el mouse. Esto hace que el juego sea más interesante porque podemos ver las cosas desde diferentes ángulos y seguir al gusano mientras se mueve.

2.1.13. Clase `ObjLoader`

La clase `ObjLoader`, un cargador de modelos 3D en formato OBJ. [3, 4, 5] Esta clase es crucial para importar la geometría (vértices, normales y coordenadas de textura) de objetos complejos en el juego.

Funciones principales

- `getInstance()`: Implementa el patrón Singleton, asegurando que solo haya una instancia de `ObjLoader` en toda la aplicación. Esto permite un acceso global y eficiente al cargador de modelos.
- `loadOBJ(..)`: Carga un modelo 3D desde un archivo OBJ especificado por su nombre.

Resumen funcional La clase `ObjLoader` es la que se encarga de cargar los modelos 3D en el juego. Usa un patrón Singleton para asegurarse de que solo existe una instancia que maneja todas las cargas. Su método principal, `loadOBJ`, lee archivos OBJ y extrae la información necesaria (posición, textura y normales) para que los objetos 3D se puedan mostrar correctamente en el juego.

2.1.14. Clase `XMLParser`

El archivo `XMLParser.cpp` implementa la clase `XMLParser`, la cual se encarga de leer y analizar datos en formato XML.

Funciones principales

- `parseFile(..)`: Carga y analiza un archivo XML completo.
- `parseString(..)`: Analiza una cadena de texto que contiene datos XML.
- `parseNode(...)`: Es el método principal que procesa recursivamente cada sección (nodo) del XML, extrayendo su nombre, atributos y contenido.

Resumen funcional La clase `XMLParser` es la que lee los archivos XML del juego. Estos archivos contienen información importante, como la configuración de los niveles. El parser convierte el texto XML en una estructura que el juego puede entender y usar fácilmente.

2.1.15. Clase `Hud`

La clase `Hud`, encargada de mostrar información importante al jugador directamente en la pantalla de juego, como la puntuación, el tiempo y la velocidad[2, 1].

Resumen funcional La clase `Hud` es vital para mantener al jugador informado sobre el estado del juego en todo momento. Se encarga de tomar datos numéricos (puntuación, tiempo, velocidad) y presentarlos de forma clara y visible en la interfaz gráfica del juego.

Referencias

[1] M. Shah, "Sdl c++ series," 2022.

[2] L. F. Productions, "Lazy foo' productions." https://lazyfoo.net/tutorials/SDL/index.php, 2025.

[3] opengl tutorial.org, "Model loading." www.opengl-tutorial.org/beginners-tutorials/tutorial-7-model-loading/, 2017.

[4] wikibooks.org, "Modern opengl tutorial load obj." https://en.wikibooks.org/wiki/OpenGL_Programming/Modern_OpenGL_Tutorial_Load_OBJ, 2020.

[5] J. de Vries, "Learn opengl." https://learnopengl.com/Introduction, 2014.