ESCUELA DE INGENIERÍA DE FUENLABRADA

GRADO EN INGENIERÍA EN SISTEMAS AUDIOVISUALES Y MULTIMEDIA

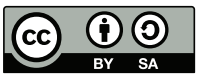
TRABAJO FIN DE GRADO

CHOMP CRAWLER: ADAPTACIÓN AL ENTORNO WEB DEL PAC-MAN CLÁSICO.

Autor: Diego Sota Rebollo

Tutor: Dr. Jesús María González Barahona

Curso académico 2025/2026



Dedicado a Amparo Rebollo,
por enseñarme el valor de la memoria.

II

Agradecimientos

Resumen

Summary

Índice general

Índice de figuras

Capítulo 1

Introducción

Capítulo 2

Objetivos

2.1. Objetivo general

Este trabajo tiene como objetivo crear un videojuego completo y funcional, inspirado en el clásico juego de arcade Pacman, con mecánicas originales del género Rogue-like, tilizando tecnologías web.

2.2. Objetivos específicos

Mediante la realización de este trabajo, se busca alcanzar los siguientes objetivos:

Exploración de tecnologías web relacionadas con gráficos 3D y desarrollo de interfaces de usuario, a fin de comprender sus capacidades y limitaciones en el contexto del desarrollo de videojuegos.

Desarrollar un proyecto completo que abarque desde la concepción inicial hasta la implementación final, incluyendo diseño, programación, pruebas y documentación.

Revisitar y poner en práctica algunos de los conceptos centrales del grado de Ingeniería en Sistemas Audiovisuales y Multimedia, aplicándolos en un proyecto que sintetice los conocimientos adquiridos durante el mismo.

Se ha buscado la obtención de una experiencia de juego con la rejugabilidad como máxima, mediante la re-implementación de un algoritmo de generación procedural de laberintos, representados en forma de *Tilemaps*. Para conseguir una experiencia accesible desde una amplia gama de dispositivos, se ha seleccionado la tecnología web, mediante la representación de

gráficos en tres dimensiones.

A nivel de estructura, se ha optado por diseñar un sistema propio, basado en *ECS*, que permite separar correctamente las responsabilidades dentro del flujo de ejecución de la lógica del juego.

2.3. Planificación temporal

Capítulo 3

Estado del arte

3.1. Análisis Técnico de Pac-Man

Para comprender la magnitud de la ingeniería detrás de *Pac-Man*, es indispensable acudir a la documentación técnica exhaustiva recopilada por Jamey Pittman en *The Pac-Man Dossier* [17]. Este compendio, fruto de la ingeniería inversa y el análisis del desensamblado del código original en ROM, revela que la aparente simplicidad del juego esconde un sistema determinista de alta complejidad. A diferencia de las aproximaciones modernas basadas en heurísticas complejas, *Pac-Man* demuestra cómo un conjunto de reglas lógicas elementales, ejecutadas sobre hardware limitado, puede generar comportamientos emergentes sofisticados. A continuación, se desgranan los aspectos críticos de su diseño basándose en esta referencia técnica.

3.1.1. Historia y relevancia en la industria

Desarrollado por Namco y diseñado por Toru Iwatani, *Pac-Man* (originalmente *Puck-Man*) debutó en Japón el 22 de mayo de 1980. En un mercado saturado por *shooters* espaciales derivados del éxito de *Space Invaders*, Iwatani buscó expandir la demografía de los salones recreativos hacia el público femenino y las parejas. Para ello, conceptualizó una mecánica centrada en "comer" (*taberu* en japonés), alejándose de la violencia explícita. El diseño de personajes, inspirado en la cultura *kawaii*, y el uso pionero de colores pastel sobre un fondo negro, rompieron con la estética militarista predominante.

La llegada del título a Estados Unidos en octubre de 1980, licenciado por Midway, supuso

un fenómeno cultural sin precedentes. A nivel técnico, el juego introdujo innovaciones que definirían el medio: fue uno de los primeros en implementar *cutscenes* o interludios narrativos y, más importante aún, dotó a los enemigos de "personalidad" mediante algoritmos de comportamiento diferenciados. Esta decisión de diseño transformó a los fantasmas de meros obstáculos móviles a agentes con roles tácticos específicos, sentando las bases de la inteligencia artificial en videojuegos.

3.1.2. Mecánicas fundamentales y sistema de teselas

La lógica de *Pac-Man* opera sobre una cuadrícula de teselas (*tilemap*) de 28×36 celdas, donde cada tesela equivale a 8×8 píxeles. El movimiento de los actores (Pac-Man y los fantasmas) no es libre, sino que está restringido a los centros de estas teselas. El motor del juego evalúa la posición de cada entidad basándose en su píxel central; si este punto entra en una nueva tesela, se actualiza la lógica de ocupación. Este sistema discretizado permite simplificar la detección de colisiones: la muerte del jugador ocurre únicamente cuando su tesela ocupada coincide con la de un fantasma.

Una restricción mecánica crucial es la prohibición de detenerse o invertir el sentido de la marcha voluntariamente. Los fantasmas siempre deben avanzar hacia la siguiente tesela disponible, evaluando las intersecciones antes de llegar a ellas. Esta anticipación es clave: la decisión de giro se toma un paso antes de la intersección física, basándose en la distancia euclidiana hacia una "tesela objetivo" (*Target Tile*). Este determinismo permite a los jugadores expertos predecir y manipular el flujo del juego, transformando la experiencia de reacción en una de planificación estratégica.

3.1.3. Inteligencia Artificial: El sistema de objetivos

La "personalidad" de cada fantasma es el resultado de asignar una tesela objetivo (*Target Tile*) diferente a cada uno, calculada en tiempo real según la posición del jugador. Según el análisis de Pittman, los comportamientos se definen de la siguiente manera:

- **Blinky (Rojo - *Shadow*):** Su objetivo es la tesela que ocupa actualmente Pac-Man. Esto genera una persecución directa e implacable, actuando como una sombra que sigue la ruta más corta hacia el jugador.

- **Pinky (Rosa - *Speedy*):** Su lógica apunta a cuatro teselas por delante de la posición actual de Pac-Man. Esto provoca que intente emboscar al jugador, cortándole el paso en las intersecciones en lugar de seguirlo desde atrás.

- **Inky (Cian - *Bashful*):** Utiliza una lógica vectorial compleja. Se traza un vector desde la posición de Blinky hasta dos teselas por delante de Pac-Man, y se duplica esa distancia en la dirección opuesta. Esto hace que su movimiento dependa de la posición relativa de su compañero rojo, resultando en maniobras de flanqueo difíciles de predecir.

- **Clyde (Naranja - *Pokey*):** Posee un comportamiento dual basado en la proximidad. Si está a más de ocho teselas de Pac-Man, su objetivo es el jugador (como Blinky). Sin embargo, si se acerca a menos de ocho teselas, su objetivo cambia inmediatamente a su esquina de patrulla (esquina inferior izquierda), haciéndole huir y parecer desinteresado o "tonto".

3.1.4. Ciclos de comportamiento y dificultad

El ritmo del juego está dictado por la alternancia cíclica entre dos modos globales de comportamiento: *Scatter* (Dispersión) y *Chase* (Persecución). Durante el modo *Scatter*, cada fantasma ignora al jugador y trata de alcanzar una tesela objetivo fija en una de las cuatro esquinas del laberinto, lo que les obliga a patrullar zonas específicas y da un respiro al jugador. Tras un tiempo predefinido, cambian a modo *Chase*, activando sus algoritmos de persecución individuales descritos anteriormente. Estos ciclos se repiten en intervalos fijos (ej. 7 segundos de dispersión, 20 de persecución) que varían según el nivel.

La curva de dificultad no se basa solo en la velocidad, sino en la alteración de estos tiempos. A medida que se superan niveles, los periodos de *Scatter* se reducen drásticamente hasta desaparecer, manteniendo a los fantasmas en persecución constante. Adicionalmente, se introduce la mecánica de "Cruise Elroy": cuando quedan pocos puntos en el mapa, Blinky aumenta su velocidad porcentualmente, llegando a superar la del propio Pac-Man, lo que intensifica la tensión final de cada nivel.

3.2. Técnicas utilizadas

3.2.1. Generación procedural de laberintos.

Si bien la literatura académica estandariza algoritmos de árboles de expansión como *Prim* o *Recursive Backtracking* para la creación de laberintos, su aplicación resulta inviable para mecánicas tipo *Pac-Man*. Estos métodos generan matemáticamente "laberintos perfectos" —estructuras acíclicas repletas de callejones sin salida—, lo que anula las rutas de evasión y la dinámica de persecución cíclica esencial en el género arcade. Debido a estas limitaciones estructurales, se ha descartado el uso de generadores de grafos convencionales en favor de la implementación basada en restricciones de teselas propuesta por Lebron [10]. Esta solución específica permite imponer condiciones de diseño no triviales, tales como la simetría axial del tablero, la eliminación de *dead-ends* (*braiding*) y la reserva de zonas funcionales para la IA ("casa de fantasmas"), garantizando niveles topológicamente válidos para la navegación de agentes autónomos.

3.2.2. Arquitectura Entity Component System (ECS)

El desarrollo de videojuegos modernos ha evidenciado las limitaciones estructurales de la Programación Orientada a Objetos (OOP) clásica, especialmente cuando se enfrentan problemas de jerarquías de herencia rígidas y cuellos de botella en el rendimiento de la memoria. Se ha considerado la arquitectura *Entity Component System* (ECS), un patrón arquitectónico que prioriza la composición sobre la herencia y separa estrictamente los datos del comportamiento. A diferencia del modelo tradicional donde un objeto encapsula ambos aspectos, en ECS una *Entidad* se reduce a un identificador único (generalmente un entero) que carece de lógica o datos propios; su función es actuar como un contenedor abstracto que agrupa *Componentes* [16]. Estos componentes son estructuras de datos puras (structs), desprovistas de métodos, que almacenan el estado específico de un aspecto del juego (e.g., 'Posición', 'Velocidad', 'Renderable').

La lógica del juego se traslada a los *Sistemas*, que operan de manera transversal sobre conjuntos de entidades que poseen una firma de componentes específica. Por ejemplo, un *Sistema de Movimiento* iterará sobre todas las entidades que posean tanto 'Posición' como 'Velocidad', actualizando los datos de la primera en función de la segunda, ignorando cualquier otro atribu-

to. Este desacoplamiento total permite una flexibilidad extrema en tiempo de diseño: es posible crear nuevos tipos de enemigos o comportamientos emergentes simplemente combinando componentes existentes, sin necesidad de refactorizar complejas cadenas de herencia ni enfrentarse al "problema del diamante" típico de la OOP [7].

3.3. Tecnologías

3.3.1. HTML5 y los navegadores web modernos.

HTML5 representa la evolución más significativa del estándar HyperText Markup Language (HTML), establecido por el World Wide Web Consortium (W3C) como la base para el desarrollo de aplicaciones web ricas y multimedia [4]. A diferencia de versiones anteriores centradas en la presentación de contenido estático, HTML5 introduce APIs nativas que habilitan funcionalidades avanzadas, tales como la reproducción de audio y vídeo sin plugins externos, el almacenamiento local de datos y la manipulación de gráficos en dos y tres dimensiones mediante WebGL. Esta expansión de capacidades ha transformado los navegadores web en plataformas completas para el desarrollo de aplicaciones interactivas, eliminando la dependencia de tecnologías propietarias como Flash.

La selección de HTML5 se justifica por su accesibilidad multiplataforma y soporte nativo en navegadores modernos como Chrome y Firefox, facilitando la ejecución sin instalación y la integración con WebGL para gráficos 3D. Además, simplifica el despliegue y promueve la interoperabilidad, demostrando la viabilidad de los navegadores para videojuegos complejos [14] [5].

3.3.2. EcmaScript 6 y Typescript.

La base del desarrollo se fundamenta en JavaScript moderno, entendido como la implementación del estándar internacional ECMA-262 (ECMAScript) [1]. Se toma como punto de inflexión la especificación ES6 (ES2015) y sus sucesoras, que introdujeron la sintaxis de módulos, promesas y clases, permitiendo al lenguaje evolucionar desde simples scripts a arquitecturas de software complejas [22]. Esta evolución resulta esencial para proyectos web de escala, ya que facilita la modularización del código y la gestión asíncrona, alineándose con las necesidades

de un videojuego interactivo que requiere actualizaciones en tiempo real y manejo eficiente de eventos.

Para garantizar la escalabilidad y la seguridad del código en un entorno de producción, se adopta TypeScript. Esta tecnología opera como un superconjunto sintáctico estricto de JavaScript que añade tipado estático opcional y comprobación de errores en tiempo de compilación [3]. TypeScript no reemplaza a JavaScript, sino que mejora la experiencia de desarrollo (DX) y la mantenibilidad, transpilándose finalmente a código JavaScript estándar compatible con cualquier navegador o entorno Node.js, asegurando así un código autodocumentado y resiliente ante errores de tipo comunes en tiempo de ejecución. Esta elección se justifica por la complejidad inherente a un proyecto de videojuego, donde la detección temprana de errores reduce significativamente el tiempo de depuración.

3.3.3. Gráficos 3D en la web con WebGL. *Three.js.*

WebGL (Web Graphics Library) surge como la estandarización llevada a cabo por el Khronos Group para trasladar las capacidades de OpenGL ES 2.0 a los navegadores web, permitiendo renderizado de gráficos 3D acelerados por hardware directamente en el elemento canvas de HTML5 sin necesidad de plugins [9]. Esta API de bajo nivel proporciona acceso directo a la GPU mediante programas de sombreado (shaders) escritos en GLSL, habilitando efectos visuales complejos y rendimiento óptimo en aplicaciones web [15]. Sin embargo, su complejidad técnica resulta excesivamente verbosa para el desarrollo ágil, requiriendo gestión manual de matrices, buffers y estados gráficos.

Para mitigar esta complejidad, se integra Three.js, una biblioteca de alto nivel que abstrae las primitivas de WebGL mediante una arquitectura orientada a objetos [24][1]. Three.js gestiona el grafo de escena, las cámaras, la iluminación y los materiales, permitiendo focalizar el desarrollo en la lógica visual y la experiencia de usuario en lugar de en la gestión matemática de matrices y buffers de memoria crudos. Esta abstracción resulta esencial para proyectos de videojuegos web, donde la productividad del desarrollador y la mantenibilidad del código son críticas para cumplir con plazos y objetivos de calidad.

Se ha elegido esta tecnología por su adopción generalizada en la industria del desarrollo web. Además, resulta muy pertinente en el contexto del Grado en Ingeniería en Sistemas Au-

[1] `https://github.com/mrdoob/three.js`

diovisuales y Multimedia, siendo una tecnología en la que se ha profundizado durante el plan de estudios.

3.3.4. Interfaces Gráficas de Usuario con *React.js*. Estados globales con *Zustand*

React.js, desarrollado originalmente por Facebook (ahora Meta) en 2013 para solucionar problemas de rendimiento en interfaces con actualizaciones de datos masivas y constantes, se ha establecido como la biblioteca estándar de facto para la construcción de interfaces de usuario modernas [13]. Su filosofía se centra en una arquitectura basada en componentes reutilizables y un paradigma declarativo, donde el desarrollador define "qué" debe mostrarse y la biblioteca gestiona el "cómo" [2]. La clave de su eficiencia reside en el Virtual DOM, una representación en memoria del árbol de elementos que permite ejecutar un algoritmo de reconciliación (Diffing Algorithm), actualizando en el DOM real únicamente aquellos nodos que han sufrido cambios. Esto supone una optimización drástica en términos de rendimiento frente a la manipulación directa del navegador.

La selección de React se justifica por su madurez en el ecosistema de desarrollo web y su capacidad para gestionar interfaces complejas con múltiples actualizaciones de estado, como es el caso de un videojuego donde la UI debe reflejar constantemente cambios en puntuaciones, vidas y estados de juego. Además, resulta pertinente en el contexto del Grado en Ingeniería en Sistemas Audiovisuales y Multimedia, siendo una tecnología que permite aplicar los conocimientos adquiridos en desarrollo de aplicaciones web de forma ágil. Siendo un framework ampliamente adoptado en la industria, su uso en este proyecto facilita la adquisición de habilidades prácticas y relevantes para el mercado laboral.

Complementando a React, se integra Zustand para la gestión del estado global de la aplicación. Esta librería ha experimentado una adopción exponencial debido a su capacidad para simplificar el flujo de datos sin el *boilerplate* o código repetitivo excesivo asociado a soluciones tradicionales como *Redux* [19]. Zustand utiliza un modelo de almacén (store) centralizado basado en hooks que permite a cualquier componente acceder y modificar el estado de manera directa y reactiva, resolviendo problemas de paso de propiedades por múltiples niveles de componentes (prop-drilling) con una huella de memoria mínima y una API extremadamente

concisa. Esta elección resulta crítica para la gestión del estado de un videojuego, donde posiciones de entidades, puntuaciones y eventos requieren actualizaciones frecuentes sin comprometer el rendimiento.

3.3.5. *React Three Fiber*. Integración de Three con React

React Three Fiber (R3F) se implementa como un renderizador personalizado de React para Three.js, actuando como un puente arquitectónico que traslada el paradigma declarativo y basado en componentes de React al entorno imperativo de los gráficos 3D [18]. A diferencia de una simple envoltura (wrapper), R3F crea un grafo de escena dinámico donde cada objeto 3D se instancia como un componente nativo de React, permitiendo gestionar el ciclo de vida, los eventos y el estado de la escena (cámaras, mallas, luces) mediante el ecosistema estándar de hooks, props y contexto. Esta arquitectura es posible gracias a la API de renderizadores personalizados de React, que permite extender el framework más allá del DOM tradicional [12].

La principal fortaleza técnica de R3F radica en que delega la gestión del bucle de renderizado y la optimización de recursos al propio reconciliador de React, garantizando que no exista sobrecarga de rendimiento (overhead) respecto al uso de Three.js vainilla, al tiempo que facilita la integración de interactividad compleja en entornos web.

La selección de R3F se justifica por su capacidad para unificar la gestión de la interfaz de usuario 2D y la escena 3D bajo un único modelo de programación, eliminando la necesidad de sincronizar manualmente el estado entre React y Three.js. Esta decisión arquitectónica permite aprovechar las ventajas de ambas tecnologías sin comprometer el rendimiento, demostrando la viabilidad de paradigmas declarativos en aplicaciones gráficas interactivas complejas.

3.3.6. Algorítmos con *Python 3*. *NumPy*

La elección de Python 3 como lenguaje vehicular para los algoritmos de generación procedural no responde únicamente a su sintaxis legible, sino a su consolidación como un entorno de producción robusto y versátil. A diferencia de su concepción original como lenguaje de *scripting*, las iteraciones modernas de Python 3 han introducido características como las anotaciones de tipo y mejoras significativas en la gestión asíncrona, lo que ha permitido su adopción masiva en infraestructuras críticas de ingeniería de software [21]. Su naturaleza multiparadigmática

facilita la implementación de lógica compleja mediante patrones idiomáticos que reducen la deuda técnica y favorecen la mantenibilidad del código a largo plazo. Además, su extensa biblioteca estándar y la madurez de su ecosistema de terceros permiten desacoplar la lógica de generación matemática de la capa de presentación web, asegurando una arquitectura modular donde los algoritmos pueden evolucionar independientemente del cliente gráfico.

No obstante, la naturaleza interpretada de Python puede suponer un cuello de botella en operaciones de cálculo intensivo. Para mitigar esta limitación en el procesamiento de las matrices del laberinto, se ha integrado NumPy, la biblioteca fundamental para la computación científica en este ecosistema [11]. La ventaja crítica de NumPy reside en su objeto principal, el *ndarray*, que a diferencia de las listas nativas de Python —que son colecciones de punteros a objetos dispersos en memoria—, almacena los datos en bloques de memoria contiguos, similar a C [8]. Esta arquitectura permite la ejecución de operaciones vectorizadas y *broadcasting*, eliminando la necesidad de bucles explícitos en el nivel del intérprete y delegando el cómputo a rutinas optimizadas de bajo nivel.

3.3.7. WASM y ejectución de código nativo en navegadores web. Pyodide.

La necesidad de ejecutar operaciones computacionalmente costosas en el lado del cliente, superando las limitaciones de rendimiento inherentes a JavaScript, motivó el surgimiento de *WebAssembly* (WASM). Concebido originalmente por el W3C, WASM se define como un formato de instrucciones binario diseñado para ser un objetivo de compilación eficiente para lenguajes de alto nivel como C, C++ o Rust [6]. Su adopción en la actualidad responde a la capacidad de ejecutar código a velocidades cercanas a las nativas dentro del entorno seguro (*sandbox*) del navegador, permitiendo portar motores de física, decodificadores de video o algoritmos científicos complejos que anteriormente requerían ejecución en servidor. Al tratarse de un estándar abierto soportado por los principales motores de navegación, WASM actúa como una capa de abstracción sobre el hardware subyacente, garantizando una ejecución determinista y eficiente independientemente de la plataforma del usuario [23].

En el contexto específico de este proyecto, para habilitar la ejecución del generador de laberintos escrito en Python dentro del navegador, se ha integrado *Pyodide*[2]. Esta solución no es un simple transpilador, sino una distribución completa de CPython portada a WebAssembly

[2] `https://github.com/pyodide/pyodide`

mediante la cadena de herramientas *Emscripten* [20]. Su arquitectura se distingue por incluir un sistema de archivos virtual (MEMFS) que simula el entorno de disco necesario para la biblioteca estándar de Python y por su capacidad de cargar dinámicamente paquetes científicos compilados, como NumPy, que mantienen sus optimizaciones en C/Fortran. Un aspecto crítico de su diseño es la interfaz de funciones foráneas (FFI) bidireccional, que permite al código Python manipular el DOM y acceder a objetos JavaScript directamente, y viceversa, facilitando una integración transparente entre ambos contextos.

Capítulo 4

Diseño e implementación

El

4.1. Arquitectura general

4.1.1. ECS

Para estructurar la lógica del juego, se ha optado por utilizar una adaptación del patrón de arquitectura *Entity-Component-System* (ECS). Este patrón permite separar claramente las responsabilidades dentro del flujo de ejecución de la lógica del juego, facilitando la escalabilidad y el mantenimiento del código. Siguiendo esta idea, los elementos del juego se representan como *entidades* compuestas por *componentes* que encapsulan datos específicos. Esto fuerza a los distintos elementos del estado a ser estructurados de manera homogénea, facilitando la generalización de las operaciones sobre ellos. Estas operaciones ocurren dentro de *sistemas*, que procesan las entidades que poseen los componentes relevantes para su función.

4.2. Motor Lógico

El núcleo de la aplicación es un bucle que ejecuta la lógica del juego. Para esto, se ha implementado una clase `GameLoop`, que se encarga de orquestar la actualización del estado del juego y la renderizado de la escena gráfica. Exte bucle ejecuta en cada iteración la logica del juego. Esto incluye la lógica global de la aplicación, el manejo de los estados de la partida

y la ejectución de los sistemas ECS cuando la partida se encuentra en ejectución.

4.2.1. Generación

Uno de los pilares fundamentales del proyecto es la generación procedural de laberintos. La rejugabilidad, identificada como objetivo principal en la sección 2.2, requiere que cada partida presente un escenario diferente, lo cual hace inviable el diseño manual de niveles. Se optó por implementar un sistema de generación procedural capaz de producir laberintos con las características topológicas del clásico Pac-Man: un espacio cerrado con pasillos interconectados, esquinas redondeadas y zonas de alta densidad de conexiones que favorecen tanto la huida como la persecución.

Para este propósito, se llevó a cabo una reimplementación del algoritmo desarrollado por Shaun Lebron [**?**] [10]. La elección de este algoritmo se fundamenta en su capacidad demostrada para producir laberintos que preservan la esencia del diseño original de Pac-Man, evitando tanto la generación de espacios excesivamente abiertos como de corredores lineales predecibles.

El sistema genera representaciones de laberintos en forma de mapas de teselas o *tilemaps*, es decir, matrices bidimensionales donde cada elemento es un identificador del tipo de tesela que ocupa esa posición (pared, pasillo, esquina, etc.). Esta representación matricial facilita tanto el procesamiento algorítmico como la posterior traducción a geometría tridimensional. Para la implementación de este módulo, se seleccionó el lenguaje Python debido a la madurez de su ecosistema en el manejo de estructuras de datos multidimensionales, específicamente a través de la biblioteca NumPy.

A continuación, se detalla el flujo de generación del laberinto:

Celdas

Para poder comprender el funcionamiento del algoritmo, es necesario introducir el concepto de *celdas* (*cells*). Una celda es un nodo con propiedades utilizado para construir una versión abstracta del laberinto. Los laberintos en Pac-Man se caracterizan por buscar deliberadamente la presencia de bucles. Por esto, en términos de forma, es de mayor utilidad entenderlos nó como un conjunto de caminos interconectados, sino como un puzzle formado por poliominós, figuras formadas por la unión de cuadrados. Cada celda representa uno de estos cuadrados.

La Matriz de Celdas y sus Conexiones

Las celdas presentan una dualidad intrínseca: por un lado, son unidades discretas que componen la estructura de una matriz bidimensional; por otro, actúan como nodos en un grafo que interconecta cada celda con sus vecinas ortogonalmente adyacentes (vecindad de Von Newmann). Esto resulta muy útil, pues permite recorrer las celdas mediante la indexación matricial, pero también permite acceder de una celda a otra, a partir de las conexiones que se establecen entre ellas.

El algoritmo comienza inicializando una matriz de celdas, donde cada celda está desconectada de sus vecinas. Resulta fundamental precisar que todas las celdas contienen una referencia a sus vecinas, independientemente de si pertenecen o no a la misma figura poliominal. Por tanto, se define *la celda siguiente en una dirección* como aquella que se encuentre adyacente en dicha dirección para una cierta celda. Esta es siempre es accesible desde la referencia que posee cada una de sus vecinas y solo se considerarán conectadas las celdas que pertenezcan a la misma figura.

El algorítmo comienza creando las referencias entre las celdas adyacentes, manteniendo todas las conexiones inicialmente deshabilitadas. En este punto, el algoritmo admite la presencia de conexiones predefinidas entre ciertas celdas, lo cual permite reservar zonas del laberinto para funciones específicas, como la “casa de los fantasmas”.

Generación Mediante Figuras

El núcleo del algoritmo radica en la generación iterativa de figuras poliominales que conectan las celdas adyacentes. En esta sección se describe de forma detallada el proceso de generación:

Validación del Grafo Generado

Generación de Túneles

El Laberinto de Teselas

4.3. Sistema Gráfico

4.3.1. Interfaz de Usuario

4.3.2. Head-Up Display

4.3.3. Escena

4.4. Estado

4.5. Prototipos y Metodología

A fin de organizar mejor el proceso de desarrollo del proyecto, se ha requerido un sistema que permita planificar el trabajo de forma efectiva y sostenible en el medio plazo. Pese a que el proyecto ha sido desarrollado en su totalidad por el alumno, con la ayuda y supervisión del tutor, la metodología de trabajo se ha asemejado a la propuesta en [[Scrum]]. Se han definido distintos sprints, asociados a una serie de objetivos. Cada uno de estos sprints debe resultar en una versión estable del juego. Las ideas que pudieran surgir durante el transcurso de un sprint han sido apuntadas y revisadas al finalizar cada uno de los mismos. Todos los elementos de la arquitectura se relacionan mediante el estado.

Capítulo 5

Experimentos y validación

Este capítulo se introdujo como requisito en 2019. Describe los experimentos y casos de test que tuviste que implementar para validar tus resultados. Incluye también los resultados de validación que permiten afirmar que tus resultados son correctos.

Capítulo 6

Resultados

En este capítulo se incluyen los resultados de tu trabajo fin de grado.

Si es una herramienta de análisis lo que has realizado, aquí puedes poner ejemplos de haberla utilizado para que se vea su utilidad.

Capítulo 7

Conclusiones

7.1. Consecución de objetivos

Esta sección es la sección espejo de las dos primeras del capítulo de objetivos, donde se planteaba el objetivo general y se elaboraban los específicos.

Es aquí donde hay que debatir qué se ha conseguido y qué no. Cuando algo no se ha conseguido, se ha de justificar, en términos de qué problemas se han encontrado y qué medidas se han tomado para mitigar esos problemas.

Y si has llegado hasta aquí, siempre es bueno pasarle el corrector ortográfico, que las erratas quedan fatal en la memoria final. Para eso, en Linux tenemos aspell, que se ejecuta de la siguiente manera desde la línea de *shell*:

```
aspell --lang=es_ES -c memoria.tex
```

7.2. Aplicación de lo aprendido

Aquí viene lo que has aprendido durante el Grado/Máster y que has aplicado en el TFG/TFM. Una buena idea es poner las asignaturas más relacionadas y comentar en un párrafo los conocimientos y habilidades puestos en práctica.

1. a

2. b

7.3. Lecciones aprendidas

Aquí viene lo que has aprendido en el Trabajo Fin de Grado/Máster.

1. Aquí viene uno.
2. Aquí viene otro.

7.4. Trabajos futuros

Ningún proyecto ni software se termina, así que aquí vienen ideas y funcionalidades que estaría bien tener implementadas en el futuro.

Es un apartado que sirve para dar ideas de cara a futuros TFGs/TFMs.

Apéndice A

Manual de usuario

Esto es un apéndice. Si has creado una aplicación, siempre viene bien tener un manual de usuario. Pues ponlo aquí.

Bibliografía

[1] Ecmascript 2023 language specification, 2023. Especificación técnica base sobre la que se construye el ecosistema JavaScript moderno.

[2] A. Banks and E. Porcello. *Learning React: Modern Patterns for Developing React Apps*. O'Reilly Media, Sebastopol, CA, 2nd edition, 2020. Texto de referencia sobre arquitectura de componentes y patrones de diseño en React.

[3] B. Cherny. *Programming TypeScript: Making Your JavaScript Applications Scale*. O'Reilly Media, Sebastopol, CA, 2019. Manual de referencia técnica sobre el sistema de tipos y la escalabilidad en TypeScript.

[4] S. Faulkner, A. Eicholz, T. Leithead, A. Danilo, and S. Moon. Html5.2 recommendation. W3c recommendation, World Wide Web Consortium (W3C), 2017. Estándar oficial que define las APIs de conectividad y multimedia.

[5] A. Freeman. *The Definitive Guide to HTML5*. Apress, New York, 2011. Enfocado en HTML5 como plataforma de desarrollo de aplicaciones ricas.

[6] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200. ACM, 2017. Publicación fundamental que introduce la arquitectura y métricas de rendimiento de WebAssembly frente a JavaScript.

[7] T. Härkönen. Advantages and implementation of entity-component-systems. Master's thesis, Tampere University, 2019. Estudio académico que compara el rendimiento y la mantenibilidad entre arquitecturas OOP y ECS en entornos de tiempo real.

[8] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, et al. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020. Artículo fundamental que describe la arquitectura interna, la vectorización y el impacto científico de NumPy.

[9] D. Jackson and J. Gilbert. Webgl specification 1.0. Standard specification, The Khronos Group, 2011. Estándar base que define la interfaz entre JavaScript y OpenGL ES.

[10] S. LeBron. Pac-man maze generation, 2018. https://shaunlebron.github.io/pacman-mazegen.

[11] W. McKinney. *Python for Data Analysis: Data Wrangling with pandas, NumPy, and Jupyter*. O'Reilly Media, Sebastopol, CA, 3rd edition, 2022. Manual técnico sobre el ecosistema de análisis de datos y computación numérica en Python.

[12] Meta Platforms Inc. *React Custom Renderers*. Meta Open Source, 2024. Documentación sobre la arquitectura interna de React que permite la creación de renderizadores personalizados como R3F.

[13] Meta Platforms Inc. *React Documentation: The Library for Web and Native User Interfaces*. Meta Open Source, 2024. Documentación oficial que describe el algoritmo de reconciliación y el ciclo de vida de componentes.

[14] Mozilla Developer Network. Introduction to web apis, 2024. Documentación técnica sobre la implementación de WebGL y WebRTC.

[15] Mozilla Developer Network. Webgl: 2d and 3d graphics on the web, 2024. Documentación de referencia sobre la integración de la API gráfica en el contexto del navegador.

[16] R. Nystrom. *Game Programming Patterns*. Genever Benning, Brighton, 2014. Referencia fundamental que detalla la implementación de patrones arquitectónicos específicos para el desarrollo de motores de juego.

[17] J. Pittman. The pac-man dossier, 2011. Análisis técnico exhaustivo sobre la lógica de programación, inteligencia artificial y mecánicas internas del arcade original de Namco.

[18] Poimandres Group. *React Three Fiber Documentation*. Poimandres Open Source Collective, 2024. Especificación técnica del renderizador y guía de integración con el ecosistema Three.js.

[19] Poimandres Group. *Zustand Documentation: Bear necessities for state management in React*. Poimandres Open Source Collective, 2024. Portal oficial de documentación técnica. Detalla la API de hooks y patrones de integración con React.

[20] Pyodide Contributors. *Pyodide: Python distribution for the browser and Node.js based on WebAssembly*. Mozilla and Open Source Community, 2024. Documentación técnica que detalla la arquitectura de portado de CPython vía Emscripten y la interfaz FFI.

[21] L. Ramalho. *Fluent Python: Clear, Concise, and Effective Programming*. O'Reilly Media, Sebastopol, CA, 2nd edition, 2022. Referencia exhaustiva sobre las características idiomáticas y el modelo de datos de Python 3 moderno.

[22] A. Rauschmayer. *Exploring ES6: Upgrade to the next version of JavaScript*. Leanpub, 2014. Análisis profundo de las características modulares introducidas en ECMAScript 6.

[23] A. Rossberg, B. L. Titzer, A. Haas, D. L. Schuff, et al. Bringing the web up to speed with WebAssembly. *Communications of the ACM*, 61(12):107–117, 2018. Análisis profundo sobre el diseño de seguridad, portabilidad y el formato binario del estándar.

[24] Three.js Authors. *Three.js Documentation*. Three.js Project, 2024. Documentación oficial de la biblioteca de abstracción para renderizado WebGL.