ESCUELA DE INGENIERÍA DE FUENLABRADA

GRADO EN INGENIERÍA EN SISTEMAS AUDIOVISUALES Y MULTIMEDIA

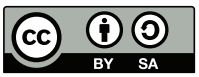
TRABAJO FIN DE GRADO

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

Autor: Diego Sota Rebollo

Tutor: Dr. Jesús María González Barahona

Curso académico 2025/2026



©2026 Diego Sota Rebollo

Algunos derechos reservados

Este documento se distribuye bajo la licencia

“Atribución-CompartirIgual 4.0 Internacional” de Creative Commons,

disponible en

`https://creativecommons.org/licenses/by-sa/4.0/deed.es`

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

II

Agradecimientos

Quiero agradecer a mi madre, Esther, por su apoyo incondicional, por su paciencia y por confiar en mí durante este proceso. Por los mismos motivos, quiero agradecer a Mario, Lucía y Piter.

Resumen

Chomp Crawler es un videojuego tridimensional que se ejecuta íntegramente en el navegador web, sin instalación, y que toma las mecánicas fundamentales de *Pac-Man* como punto de partida para construir una experiencia de juego nueva. El jugador recorre una mazmorra formada por múltiples salas interconectadas, cada una de ellas un laberinto al estilo del arcade original, perseguido por enemigos cuyo comportamiento mantiene la tensión característica del clásico.

El elemento central del proyecto es la generación procedural de niveles. Cada partida produce una mazmorra distinta aprovechando dos capas algorítmicas complementarias: una que genera salas individuales con las propiedades topológicas del laberinto de *Pac-Man* —espacio cerrado, cíclico y sin callejones sin salida—, y otra que las ensambla en una estructura coherente y completamente navegable, con garantía algorítmica de conectividad. El resultado es que ningún nivel se repite, pero todos son justos y explorables: el jugador puede recorrer la mazmorra entera en cada partida sin encontrar regiones inaccesibles, manteniendo la sensación de descubrimiento que define al género *rogue-like*.

Para sostener esta experiencia en el navegador se ha construido una arquitectura que combina un motor lógico basado en el patrón *Entity Component System*, un generador de laberintos escrito en Python y ejecutado en el cliente mediante WebAssembly, y una representación tridimensional implementada con React Three Fiber sobre WebGL. El proyecto constituye un ejemplo concreto de integración de tecnologías heterogéneas —Python, TypeScript, WebAssembly, WebGL— dentro de un único producto funcional accesible desde cualquier navegador moderno.

Summary

Chomp Crawler is a three-dimensional video game that runs entirely in the web browser, requiring no installation, and takes the core mechanics of *Pac-Man* as a foundation to build a new gaming experience. The player navigates a dungeon made up of multiple interconnected rooms, each one a maze in the style of the original arcade game, pursued by enemies whose behaviour preserves the characteristic tension of the classic.

The central element of the project is procedural level generation. Every playthrough produces a different dungeon through two complementary algorithmic layers: one that generates individual rooms with the topological properties of the *Pac-Man* maze —enclosed, cyclic, free of dead ends—, and another that assembles them into a coherent and fully navigable structure, with an algorithmic guarantee of connectivity. The result is that no two levels are alike, yet every one is fair and explorable: the player can traverse the entire dungeon in each run without encountering unreachable areas, preserving the sense of discovery that defines the *rogue-like* genre.

To sustain this experience in the browser, an architecture has been built that combines a logic engine based on the *Entity Component System* pattern, a maze generator written in Python and executed client-side via WebAssembly, and a three-dimensional renderer implemented with React Three Fiber on top of WebGL. The project stands as a concrete example of integrating heterogeneous technologies —Python, TypeScript, WebAssembly, WebGL— into a single functional product accessible from any modern browser.

Índice general

Índice de figuras

Capítulo 1

Introducción

Desde sus orígenes, la industria del videojuego ha avanzado de la mano de la tecnología disponible en cada momento. En los últimos años, los navegadores web han protagonizado uno de los saltos más significativos de esa evolución: gracias a la estandarización de tecnologías como WebGL y WebAssembly, el navegador ha dejado de ser un visor de documentos para convertirse en una plataforma de ejecución capaz de sostener experiencias interactivas complejas sin instalaciones ni descargas. En este contexto, el desarrollo de videojuegos para la web ha ganado una viabilidad técnica que hasta hace poco era impensable.

Dentro de la historia del medio, pocos títulos han tenido un impacto tan duradero como *Pac-Man*. Publicado en 1980, definió un género y estableció mecánicas que siguen siendo reconocibles cuatro décadas después: el jugador recorre un tablero comiendo puntos mientras evita a unos enemigos con comportamientos diferenciados. Sin embargo, el tablero de *Pac-Man* no es un laberinto en el sentido estricto. Es una arena cerrada y simétrica, diseñada para la persecución y la fluidez del movimiento, no para la exploración ni para el desafío de orientarse. El jugador lo memoriza rápidamente y, con ello, la sorpresa desaparece.

La pregunta que motiva este trabajo es directa: ¿qué ocurre si tomamos las mecánicas de *Pac-Man* y las situamos en un laberinto real, uno que el jugador no haya visto nunca? Un espacio que tenga comienzo y final, que exija orientarse, que obligue a tomar decisiones de ruta y que sea distinto en cada partida. Esta idea conecta con el género *rogue-like*, donde la generación procedural de niveles es el mecanismo que sostiene la rejugabilidad. La combinación de ambas tradiciones es el núcleo de *Chomp Crawler*, accesible en `https://chompcrawler.com`.

1.1. Objetivos

1.1.1. Objetivo principal

Diseñar e implementar un videojuego completo, funcional y rejugable, inspirado en *Pac-Man*, en el que los niveles sean laberintos genuinos generados proceduralmente: con entrada y salida definidas, completamente navegables, y distintos en cada partida.

1.1.2. Objetivos instrumentales

Los siguientes objetivos son condición necesaria para alcanzar el principal:

1. Implementar el motor lógico del videojuego desde cero: movimiento, colisiones, comportamiento de enemigos y ciclo de vida de la partida.

2. Diseñar las mecánicas de juego y el apartado visual adaptados al contexto tridimensional y explorable.

3. Desarrollar un generador procedural que produzca en cada partida un nivel único con garantía algorítmica de conectividad y navegabilidad completa.

1.1.3. Restricciones

El proyecto se ha desarrollado bajo las siguientes restricciones técnicas, que acotan el espacio de soluciones y forman parte de la propuesta en sí misma:

1. Toda la lógica, generación y renderizado se ejecutan en el cliente; no existe componente en servidor.

2. La aplicación emplea un *stack* de *frontend* moderno basado en React y Next.js.

3. La representación del juego es tridimensional, implementada sobre WebGL mediante React Three Fiber.

4. El generador de niveles, escrito en Python, se ejecuta en el navegador a través de WebAssembly mediante Pyodide.

1.2. Estructura del documento

El Capítulo 2 introduce las técnicas y tecnologías relevantes sobre las que se apoya el proyecto. El Capítulo 3 describe el diseño e implementación: mecánicas, generador procedural, motor ECS y arquitectura de la aplicación. El Capítulo 5 detalla la planificación temporal del desarrollo. El Capítulo 6 recoge los experimentos de rendimiento y las pruebas empíricas realizadas durante el desarrollo. Finalmente, el Capítulo 7 evalúa el cumplimiento de los objetivos, extrae las lecciones aprendidas y apunta líneas de trabajo futuro.

Capítulo 2

Técnicas y Tecnologías Relevantes

2.1. Pac-Man

Pac-Man es un videojuego arcade en el que el jugador controla a un personaje circular que recorre un laberinto comiendo puntos mientras evita a cuatro fantasmas que lo persiguen. El objetivo de cada pantalla es consumir todos los puntos; cuatro *power pellets* distribuidos en las esquinas permiten, durante unos segundos, invertir los roles y cazar a los fantasmas. El laberinto no cambia entre partidas: es siempre el mismo tablero simétrico, con los mismos túneles y las mismas intersecciones.

Figura 2.1: El tablero clásico de *Pac-Man*: un laberinto simétrico, idéntico en cada partida, con sus túneles e intersecciones característicos.

La aparente sencillez del diseño esconde una complejidad técnica notable, bien documentada por Jamey Pittman en *The Pac-Man Dossier* [1]. La implementación demuestra cómo un conjunto reducido de reglas ejecutadas en hardware muy limitado puede generar comportamien-

tos emergentes sofisticados. El análisis de Pittman es la referencia principal de esta sección.

2.1.1. 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 tras el éxito de *Space Invaders*, el estudio buscó expandir su público adoptando una estética más familiar: una mecánica centrada en "comer" (*taberu* en japonés), lejos de la violencia explícita. El diseño de personajes, inspirado en la cultura *kawaii*, y el uso pionero de colores pastel sobre fondo negro rompieron con la estética militarista predominante.

La llegada del título a Estados Unidos en octubre de 1980 supuso un fenómeno cultural sin precedentes. A nivel técnico, introdujo innovaciones que definirían el medio: fue uno de los primeros juegos en incorporar *cutscenes* narrativas y, más relevante aún, dotó a los enemigos de "personalidad" mediante algoritmos de comportamiento diferenciados. Esto transformó a los fantasmas de simples obstáculos en agentes con roles específicos, siendo uno de los primeros ejemplos de lo que en el contexto de los videojuegos de la época se denominaba inteligencia artificial: sistemas de reglas deterministas diseñados para simular comportamiento con intención, sin ninguna relación con el aprendizaje automático que el término evoca hoy.

2.1.2. 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 no es libre: está restringido a los centros de estas teselas. El motor evalúa la posición de cada entidad por su píxel central; cuando este entra en una nueva tesela, se actualiza la lógica de ocupación. Este comportamiento discreto simplifica la detección de colisiones: la muerte ocurre únicamente cuando la tesela de Pac-Man coincide con la de un fantasma.

Una restricción central sobre los fantasmas es la prohibición de detenerse o invertir la marcha voluntariamente. Siempre deben avanzar hacia la siguiente tesela disponible, evaluando todos los caminos al llegar a una encrucijada antes de seleccionar una dirección. Esta decisión se toma en función de la distancia euclidiana hacia una "tesela objetivo". La naturaleza determinista de este comportamiento permite a jugadores expertos predecir y manipular el flujo del

juego, dando lugar a numerosas estrategias y a la identificación de puntos ciegos.

2.1.3. El sistema de objetivos de los fantasmas

La "personalidad" de cada fantasma resulta de asignar una tesela objetivo distinta, calculada en tiempo real a partir de la posición del jugador. Según el análisis de Pittman, los comportamientos son los siguientes:

- **Blinky (Rojo):** Su objetivo es la tesela que ocupa Pac-Man, generando una persecución directa e implacable.
- **Pinky (Rosa):** Apunta a cuatro teselas por delante de Pac-Man en su dirección de movimiento, buscando emboscarlo en las intersecciones.
- **Inky (Cian):** Traza un vector desde Blinky hasta dos teselas por delante de Pac-Man y lo duplica en sentido opuesto, creando una emboscada coordinada que simula estrategia conjunta.
- **Clyde (Naranja):** Comportamiento dual: persigue a Pac-Man cuando está lejos, pero huye hacia su esquina al acercarse a menos de ocho teselas, resultando en un enemigo aparentemente impredecible.

2.1.4. Ciclos de comportamiento y dificultad

El ritmo del juego está dictado por la alternancia entre dos modos globales: *Scatter* (los fantasmas ignoran al jugador y se dirigen a sus esquinas asignadas) y *Chase* (activan sus algoritmos individuales). Estos ciclos se repiten en intervalos que se acortan conforme aumenta el nivel, hasta que *Scatter* desaparece y los fantasmas persiguen de forma ininterrumpida. La dificultad no reside en la velocidad, sino en la gestión de estos tiempos.

2.1.5. La franquicia y su influencia

La popularidad de *Pac-Man* trascendió los salones recreativos: existe merchandising, una serie de animación, una película de 2015 y decenas de entregas oficiales y no oficiales. La

franquicia ha demostrado una capacidad de reinvención notable, y varias de esas iteraciones han servido de referencia directa o de inspiración durante el desarrollo de *Chomp Crawler*.

Ms. Pac-Man (1982) fue la primera continuación relevante, e introdujo cuatro laberintos distintos que rotan entre pantallas, rompiendo por primera vez la monotonía del tablero único. *Pac-Mania* (1987) exploró tímidamente el espacio tridimensional con una perspectiva isométrica y añadió la capacidad de saltar, la primera modificación significativa a la movilidad del personaje. *Pac-Man Arrangement* (1996) profundizó en el diseño de enemigos con la introducción de Kinky, un nuevo fantasma que al fusionarse con otro lo potencia, generando versiones mejoradas con comportamientos más agresivos y complejos; ese mismo título incluyó un jefe final, algo inédito en la franquicia. Más recientemente, *Pac-Man 256* (2015) adaptó el juego al género *endless runner*: el jugador asciende por un laberinto proceduralmente generado de forma infinita, esquivando fantasmas y acumulando puntos mientras una glitch que corrompe el laberinto avanza desde abajo.

La alta reimplementabilidad del concepto original queda ilustrada por el hecho de que Google embebió en su buscador una versión jugable cuyo laberinto tiene la forma del logo de la empresa. Estas iteraciones, junto con numerosas reimplementaciones de la comunidad, confirman que las mecánicas de *Pac-Man* son un punto de partida fértil para la experimentación, tal como lo es en este trabajo.

2.1.6. Mecánicas de Pac-Man en Chomp Crawler

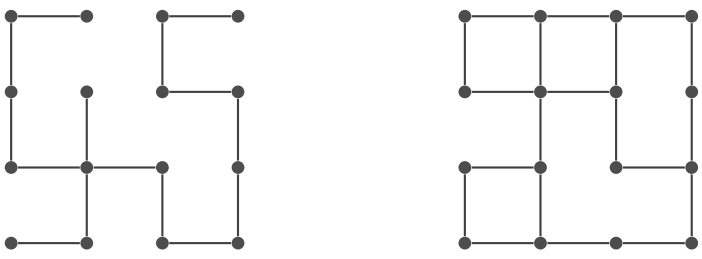
Chomp Crawler no reimplementa *Pac-Man*: toma sus mecánicas como punto de partida y las reinterpreta en un contexto nuevo. La persecución, la dinámica de comer puntos, los *power pellets* y el comportamiento dual de los enemigos están presentes, pero subordinados a un objetivo distinto: explorar y superar un laberinto genuino. El análisis técnico de las secciones anteriores, especialmente el sistema de teselas y el diseño de comportamientos, ha sido material de referencia directo para las decisiones de implementación descritas en el Capítulo 3.

2.2. Algoritmos y técnicas de generación procedural

Los laberintos son una de las estructuras más antiguas de la cultura humana. Desde el mítico laberinto de Creta hasta los setos recortados de los jardines renacentistas, la idea de un espacio

diseñado para desorientar y desafiar al que lo recorre ha fascinado a arquitectos, matemáticos y jugadores por igual. En su forma más esencial, un laberinto es un espacio con una entrada y una salida en el que el camino que las une no es evidente: el explorador debe orientarse, tomar decisiones y, a menudo, retroceder. Lo que varía entre laberintos es la naturaleza de ese reto: algunos tienen un único camino correcto, otros ofrecen múltiples rutas con distintos grados de dificultad, y otros aún son espacios abiertos donde la desorientación surge de la escala y la repetición.

La generación procedural de laberintos es la disciplina que estudia cómo producir estas estructuras de forma algorítmica, sin diseño manual. Desde el punto de vista computacional, un laberinto puede modelarse como un grafo en el que los nodos son posiciones transitables y las aristas son conexiones entre ellas, lo que permite aplicar herramientas clásicas de la teoría de grafos a su construcción y análisis. El campo cuenta con una larga tradición y ha dado lugar a una diversidad amplia de algoritmos con propiedades distintas: algunos producen laberintos perfectos (con exactamente un camino entre cualquier par de puntos, sin ciclos), otros generan espacios con múltiples rutas alternativas, y otros permiten controlar la densidad de bifurcaciones, la longitud media de los corredores o la presencia de grandes salas abiertas. Algoritmos como *Recursive Backtracking*, *Prim*, *Eller* o *Wilson* representan aproximaciones clásicas, cada una con compromisos diferentes entre complejidad de implementación, rendimiento y características del laberinto resultante.



(a) Laberinto perfecto: árbol, sin ciclos (b) Laberinto cíclico: grafo con ciclos

Figura 2.2: Modelado de laberintos como grafos. En un *laberinto perfecto* (a) existe un único camino entre cualquier par de celdas, lo que corresponde a un árbol sin ciclos y abundantes callejones sin salida. Un *laberinto cíclico* (b) añade conexiones adicionales que cierran bucles, ofreciendo rutas alternativas como las que exige la jugabilidad de *Pac-Man*.

2.2.1. Laberintos al estilo Pac-Man

Si bien la literatura académica estandariza algoritmos como *Prim* o *Recursive Backtracking* para la creación de laberintos, su aplicación resulta inviable para mecánicas tipo *Pac-Man*. Estos métodos generan "laberintos perfectos": estructuras acíclicas repletas de callejones sin salida (figura 2.2a). Ese tipo de topología restringiría demasiado la movilidad del jugador e invalidaría las mecánicas de persecución, haciendo el juego injusto y frustrante.

Por este motivo se ha descartado el uso de generadores convencionales en favor de la implementación basada en restricciones de teselas propuesta por ShaunLebron [2]. Esta solución tiene por objetivo crear laberintos cíclicos con las proporciones exactas de los laberintos de *Pac-Man* y *Ms. Pac-Man*, logrando resultados extremadamente fidedignos mediante un sistema de generación basado en estructuras poliominales que garantiza la navegabilidad completa característica del juego original.

2.2.2. Árboles de Expansión Mínimos y el Algoritmo de Kruskal.

La garantía de conectividad en grafos se presenta como un problema clásico de la teoría de grafos, abordado formalmente mediante la búsqueda de un *árbol de expansión mínimo* (*Minimum Spanning Tree*, MST). Dado un grafo con aristas ponderadas, el MST es el subgrafo que conecta todos los nodos minimizando la suma de pesos de sus aristas. El algoritmo de Kruskal, formulado en 1956 [3], resuelve este problema mediante la siguiente estrategia:

Las aristas se ordenan por peso ascendente y se añaden secuencialmente al árbol, descartando aquellas que formarían un ciclo. La detección de ciclos se delega en la estructura de datos Conjunto Disjunto, que mantiene particiones de nodos y responde a consultas de pertenencia en tiempo casi constante, logrando una complejidad total de $\mathcal{O}(E \log E)$.

Los árboles de expansión tienen una relación directa con la generación procedural de laberintos: un árbol de expansión sobre una rejilla regular es, por definición, un laberinto perfecto, ya que conecta todas las celdas sin formar ciclos. Algoritmos como *Prim* y *Kruskal* se emplean habitualmente con este propósito en la literatura de generación procedural. En este proyecto, sin embargo, se aplican a un nivel superior de abstracción: no para generar el laberinto interno de una sala, sino para decidir qué salas de la mazmorra quedan conectadas entre sí, garantizando que el nivel completo sea siempre transitable.

2.2.3. Arquitectura Entity Component System

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* es un identificador único que carece de lógica o datos propios; su función es actuar como un contenedor abstracto que agrupa *Componentes* [4]. Estos componentes son estructuras de datos carentes de funcionalidad, que almacenan el estado específico de un aspecto del juego.

La lógica del juego se traslada a los *Sistemas*, funciones 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 atributo. 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. En la programación orientada a objetos clásica, la herencia múltiple introduce el llamado **problema del diamante**: cuando una clase hereda de dos clases que a su vez comparten un ancestro común, el compilador o el intérprete se enfrenta a una ambigüedad sobre qué versión de los atributos y métodos heredar, lo que obliga a soluciones ad hoc que complican la jerarquía. ECS evita este problema por diseño, ya que las entidades no heredan comportamiento sino que lo adquieren mediante la composición de componentes independientes [5].

2.3. Tecnologías

Este apartado describe las tecnologías empleadas en el desarrollo de Chomp Crawler, organizadas en cuatro grupos. En primer lugar se exponen las APIs nativas que el navegador web moderno pone a disposición de las aplicaciones (HTML5, WebGL y WebAssembly), que constituyen la plataforma de ejecución del proyecto. A continuación se detallan los lenguajes de programación utilizados y, seguidamente, las bibliotecas y *frameworks* específicos construidos

sobre dichas APIs. Por último, se recogen las tecnologías auxiliares empleadas en la redacción de la memoria, el entorno de desarrollo y el despliegue de la aplicación.

2.3.1. El navegador web moderno y sus APIs

El navegador web ha dejado de ser un simple visor de documentos para convertirse en una plataforma de ejecución completa, capaz de albergar aplicaciones interactivas comparables en complejidad a las nativas de escritorio. Esta transformación se sustenta en un conjunto de APIs estandarizadas que exponen, de forma segura y portable, capacidades antes reservadas al sistema operativo. El proyecto se apoya directamente en tres de ellas: la base estructural y multimedia de HTML5, el renderizado 3D acelerado por hardware de WebGL y la ejecución de código nativo de WebAssembly.

HTML5

HTML5 representa una evolución 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 [6]. A diferencia de versiones anteriores centradas en la presentación de contenido estático, HTML5 introduce APIs nativas que amplían notablemente las posibilidades en este contexto, incorporando funcionalidades 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, facilitando la ejecución sin instalación. Además, simplifica el despliegue y promueve la interoperabilidad, demostrando la viabilidad de los navegadores para videojuegos complejos [7] [8].

WebGL: gráficos 3D acelerados por hardware

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, permi-

tiendo 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 [10]. Sin embargo, su complejidad técnica resulta excesivamente verbosa para el desarrollo ágil, requiriendo gestión manual de matrices, buffers y estados gráficos. Por este motivo, el proyecto no consume WebGL directamente, sino a través de la abstracción de alto nivel que proporciona Three.js, descrita más adelante.

WebAssembly

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 [11]. 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 [12]. En este proyecto, WASM es la API que habilita la ejecución del generador de laberintos escrito en Python dentro del navegador, a través de la distribución Pyodide descrita en el grupo de bibliotecas.

2.3.2. Lenguajes de programación

La práctica totalidad del proyecto está escrita en TypeScript: el motor lógico, la aplicación web, la interfaz de usuario y la integración con las bibliotecas de renderizado. Python se reserva exclusivamente para el módulo de generación procedural de laberintos, que se ejecuta en el navegador a través de Pyodide. Esta partición mantiene cada lenguaje en el dominio en el que ofrece mayor valor: TypeScript para la arquitectura de un sistema interactivo complejo, Python

para la expresividad algorítmica del generador.

JavaScript y el estándar ECMAScript

La base del desarrollo en el cliente se sustenta en JavaScript moderno, entendido como la implementación del estándar internacional ECMA-262 (ECMAScript) [13]. 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 hasta hacer viable el desarrollo de arquitecturas de software complejas [14]. Esta evolución resulta esencial para el desarrollo de aplicaciones web, ya que facilita la modularización del código y enriquece las posibilidades de gestión asíncrona, alineándose con las necesidades de un videojuego interactivo que requiere actualizaciones en tiempo real y manejo eficiente de eventos.

TypeScript

Para mejorar la experiencia de desarrollo y dificultar la generación de errores en tiempo de ejecución, 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 [15]. TypeScript no reemplaza a JavaScript, sino que ofrece una capa adicional de seguridad durante el desarrollo, transpilándose finalmente a código JavaScript estándar compatible con cualquier navegador o entorno Node.js.

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 y el tipado explícito documenta los contratos entre los numerosos sistemas del motor lógico.

Python 3 y 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 [16]. 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 [17]. 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 [18]. 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.

2.3.3. Bibliotecas y *frameworks*

Three.js

Three.js es una biblioteca de alto nivel que abstrae las primitivas de WebGL mediante una arquitectura orientada a objetos [19][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 Audiovisuales y Multimedia, siendo una tecnología en la que se ha profundizado durante el plan de estudios.

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

React y Next.js

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 [20]. 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" [21]. 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. Para facilitar la orquestación general de la aplicación y su despliegue, la biblioteca se ha integrado dentro del entorno de *Next.js* [22], un marco de trabajo que proporciona una estructura sólida para aplicaciones React sin añadir complejidad innecesaria a la lógica de cliente. 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.

Zustand

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* [23]. 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.

Tailwind CSS

En el ámbito de la estilización visual, se ha adoptado *Tailwind CSS* [24]. Este marco de trabajo basado en clases de utilidad (*utility-first*) permite aplicar estilos directamente sobre la estructura de los componentes de React, evitando la necesidad de mantener complejas hojas de estilo globales. Su integración agiliza el diseño de las interfaces de usuario y garantiza una estricta consistencia estética en toda la aplicación web.

React Three Fiber

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 [25]. 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 [26].

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.

Pyodide

Para habilitar la ejecución del generador de laberintos escrito en Python dentro del navegador, sobre la API de WebAssembly antes descrita, se ha integrado *Pyodide*[2]. Esta solución no es un simple transpilador, sino una distribución completa de CPython portada a WebAssembly mediante la cadena de herramientas *Emscripten* [27]. 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.

2.3.4. Otras tecnologías utilizadas

Más allá de las tecnologías que conforman el producto final, el desarrollo del proyecto se ha apoyado en un conjunto de herramientas auxiliares para la redacción de la documentación, la escritura de código y el despliegue en producción.

LaTeX y TikZ

La presente memoria se ha redactado íntegramente en LaTeX, el sistema de composición tipográfica orientado a documentos técnicos y científicos [28]. Frente a los procesadores de texto convencionales, LaTeX desacopla el contenido de la presentación, garantizando una maquetación consistente, una gestión automática de referencias cruzadas y bibliografía, y un tratamiento riguroso de la notación matemática. Para la elaboración de los diagramas técnicos que ilustran la arquitectura del sistema y los algoritmos de generación procedural se ha empleado TikZ, el paquete de dibujo vectorial programático integrado en LaTeX [29], que permite construir figuras precisas y reproducibles directamente desde el código fuente del documento, manteniendo la coherencia estética con el resto de la maquetación.

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

Entorno de desarrollo

Durante el desarrollo se ha experimentado con un flujo de trabajo basado íntegramente en la terminal, articulado en torno al editor Neovim [30] configurado mediante la distribución LazyVim. El aprendizaje de las *vim motions* y el aprovechamiento de los plugins integrados en LazyVim han agilizado notablemente la edición de código, reduciendo además la dependencia de asistentes externos al resolver de forma manual tareas que de otro modo se habrían delegado. Este editor se ha complementado con *lazygit* como interfaz para el control de versiones con Git y con *Zellij* como multiplexor de terminal, que permite organizar el espacio de trabajo en múltiples paneles y sesiones persistentes.

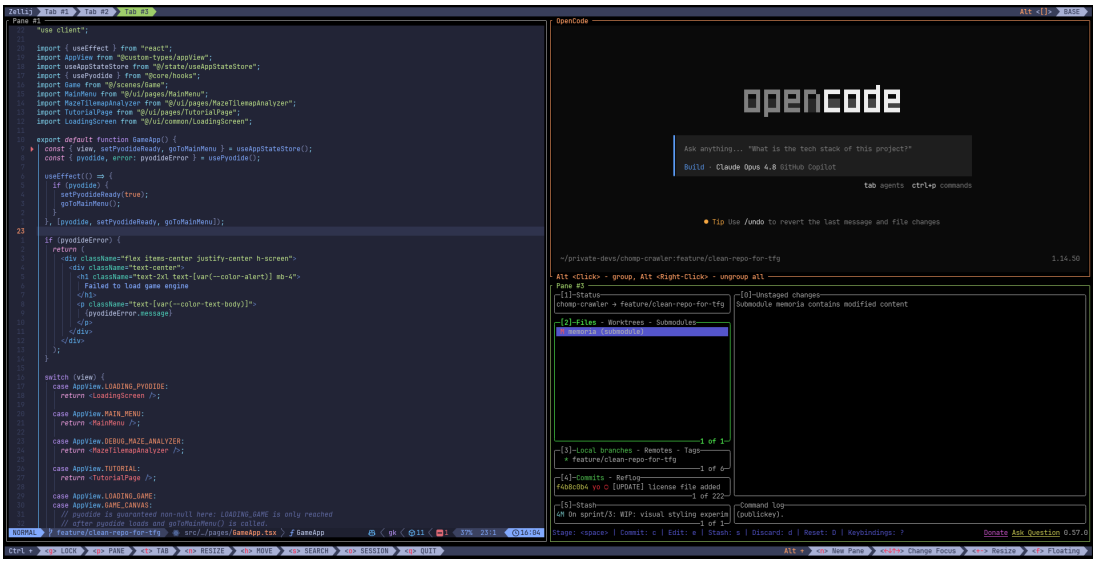


Figura 2.3: Entorno de desarrollo empleado durante el proyecto: Neovim (LazyVim) para la edición de código, *lazygit* como interfaz de Git y *Zellij* como multiplexor de terminal, organizados en paneles dentro de una misma sesión.

Como asistente de inteligencia artificial para el desarrollo se ha empleado *opencode*, una herramienta de código abierto que permite construir flujos de trabajo agénticos orientados a la asistencia en el desarrollo de software. A diferencia de alternativas propietarias como Claude Code, opencode es agnóstica respecto al proveedor del modelo de lenguaje, pudiendo conectarse con una amplia variedad de proveedores de LLM, lo que ofrece flexibilidad tanto en coste como en capacidades.

Despliegue e infraestructura

La aplicación se ha desplegado en un servidor privado virtual (VPS) provisto de un sistema operativo Ubuntu, configurado desde cero con criterios de seguridad. Para facilitar y reproducir el despliegue se ha empleado Docker [31], que encapsula la aplicación y sus dependencias en contenedores aislados del sistema anfitrión. El servidor aloja, junto a Chomp Crawler, otros proyectos personales; la convivencia y el enrutado entre ellos se resuelve mediante un *proxy* inverso gestionado con Nginx [32], que dirige el tráfico al servicio correspondiente y habilita HTTPS para cifrar las comunicaciones. El videojuego es accesible públicamente a través del dominio *chompcrawler.com*, adquirido específicamente para el proyecto.

La configuración del servidor prioriza la reducción de la superficie de exposición: únicamente permanecen abiertos los puertos estrictamente necesarios, esto es, el 80 (HTTP) y el 443 (HTTPS) para el tráfico web y el 22 (SSH) para la administración remota, cuyo acceso se restringe mediante autenticación por clave criptográfica con huella SHA-256 en lugar de contraseña.

La figura 2.4 resume esta topología de despliegue.

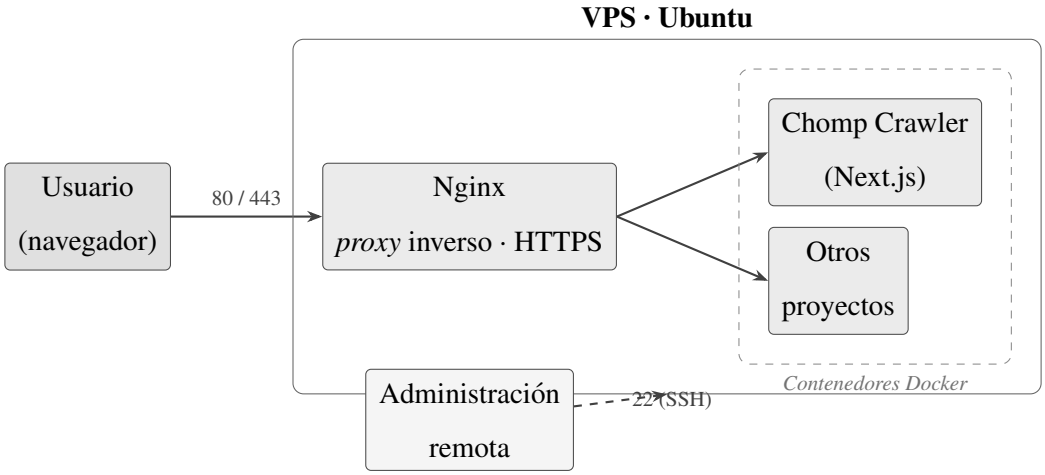


Figura 2.4: Topología de despliegue. El tráfico web llega por los puertos 80 y 443 a un *proxy* inverso Nginx, que enruta cada petición al contenedor Docker correspondiente; junto a *Chomp Crawler* conviven otros proyectos en el mismo VPS. La administración remota se realiza por SSH (puerto 22) con autenticación por clave.

Capítulo 3

Diseño

Este capítulo describe el videojuego desde la perspectiva del diseño: qué es *Chomp Crawler*, cómo se juega y qué aspecto tiene. Se presenta primero la jugabilidad —la estructura del nivel, los enemigos y las mecánicas que el jugador maneja durante la partida— y, a continuación, la dirección de arte y los criterios estéticos que dotan de identidad visual al conjunto. La descripción se mantiene deliberadamente en el plano conceptual; los detalles de cómo estas ideas se traducen en código se abordan en el capítulo siguiente, dedicado a la implementación.

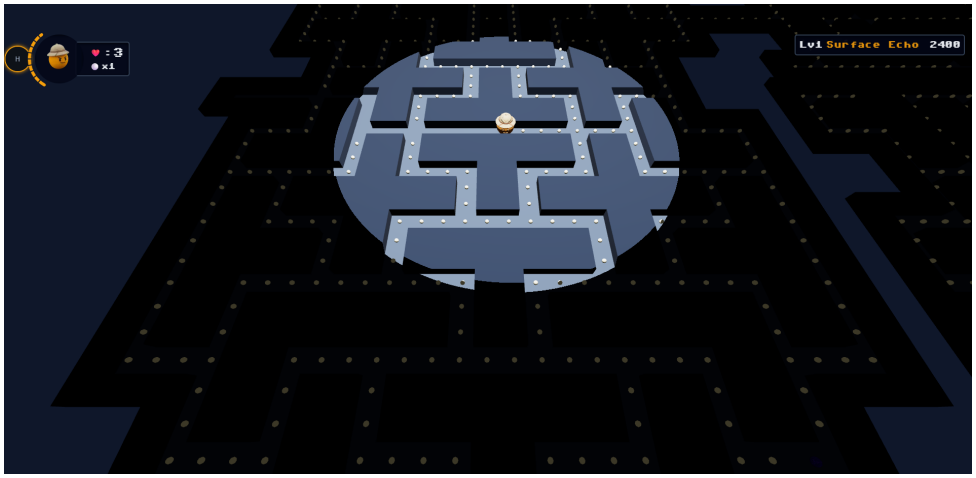


Figura 3.1: Vista de juego habitual de una mazmorra de *Chomp Crawler*, donde se aprecian los corredores de estilo *Pac-Man* y los túneles que enlazan salas contiguas.

3.1. Jugabilidad

3.1.1. Una Mazmorra de Laberintos

Los laberintos de estilo *Pac-Man* sacrifican el concepto fundamental del laberinto clásico —explorar un espacio desde una entrada hasta encontrar una salida— en favor de un espacio cerrado, cíclico y continuo que prioriza la fluidez del movimiento y la persecución. *Chomp Crawler* aprovecha precisamente esta característica para construir sobre ella una experiencia más profunda: en lugar de un único laberinto, el jugador se adentra en una *mazmorra*, un nivel completo formado por múltiples salas interconectadas, cada una de ellas constituida por un laberinto de estilo *Pac-Man* generado de forma procedural. Los túneles laterales del juego original, que en el clásico arcade permitían atravesar el tablero de lado a lado, adquieren aquí un nuevo rol estructural: actúan como pasarelas que conectan salas horizontalmente contiguas, tejiendo la red de navegación de la mazmorra.

La disposición de estos pasadizos entre salas es la que otorga a la mazmorra su carácter de laberinto: algunas salas conducen al jugador hacia la salida del nivel, mientras que otras constituyen callejones sin salida que obligan a retroceder y reorientar la ruta. Las salas verticalmente adyacentes son visibles desde la posición del jugador, actuando como referencias espaciales, pero únicamente son accesibles a través de una sala contigua en el eje horizontal, garantizando que la mazmorra completa sea siempre navegable en su totalidad. El tamaño de las salas varía de forma aleatoria en cada partida, aspecto que se detalla en profundidad en la sección dedicada a la generación de niveles, asegurando que ninguna mazmorra se repita y que el jugador no pueda memorizar su disposición de una partida a la siguiente.

3.1.2. Ecos en el Laberinto

Los *ecos* son los enemigos que habitan la mazmorra, una simplificación de los fantasmas de *Pac-Man* adaptada a la nueva estructura de juego. Su comportamiento alterna de forma aleatoria entre dos estados base: un estado de reposo (*idle*), en el que permanecen estáticos en su posición, y un estado de deambulación (*scatter*), en el que recorren el laberinto de manera aleatoria sin un objetivo definido. Esta alternancia impredecible mantiene al jugador en tensión incluso cuando no es detectado, ya que un eco en reposo puede activarse en cualquier momento.

La detección está mediada por el foco de luz direccional que emite el jugador, una representación visual del sonido que *Chomp* genera al moverse por el laberinto. Este foco ilumina el área inmediatamente frente al personaje, mientras que el resto del nivel permanece en una oscuridad pronunciada que limita drásticamente la visibilidad del entorno. Cualquier eco que entre dentro de este radio iluminado percibe al jugador y activa su modo de persecución, siguiéndolo activamente hasta que logra abandonar el foco, momento en el que pierde el rastro y retoma su comportamiento habitual. La oscuridad circundante actúa, por tanto, como aliada natural del jugador, ocultando su posición frente a los ecos que deambulan fuera del foco.

Cuando el jugador utiliza una *Esfera de Ruido Blanco*, los ecos cercanos quedan perturbados y, mientras permanezcan dentro del radio del foco, huyen activamente del jugador. Al salir del área iluminada dejan de huir, pero el estado de perturbación se mantiene: si vuelven a entrar en el foco reanudan la huida de inmediato, sin necesidad de ser afectados de nuevo.

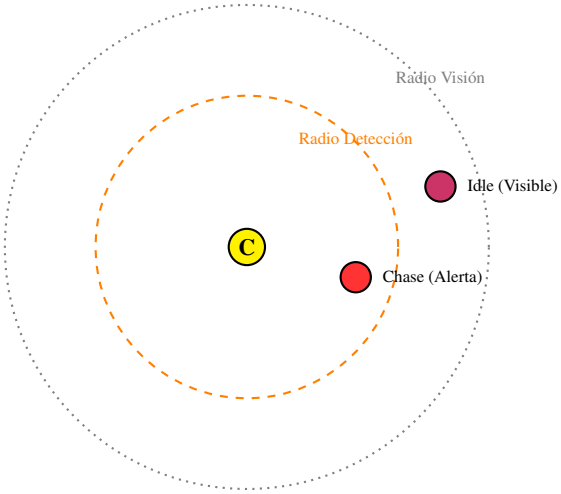


Figura 3.2: Mecánica de detección auditiva y visual (foco). Los ecos solo reaccionan y persiguen al jugador (C) si se encuentran dentro del radio de detección interior.

3.1.3. Esencia, Dash y Esferas de Ruido Blanco

En el diseño original de *Pac-Man*, los elementos del nivel cumplen roles muy concretos: los *pacdots* son puntuación y condición de victoria, mientras que los *power pellets* otorgan temporalmente la capacidad de cazar a los fantasmas. Al transformar la estructura del juego en una mazmorra de múltiples salas, estas dinámicas requieren una reinterpretación que las integre en el nuevo sistema jugable.

Los *pacdots* se convierten en *bolas de esencia*, un recurso que el jugador acumula al recorrer las salas del nivel. La esencia así recolectada actúa como combustible para los sistemas jugables del personaje: a medida que se consume, se carga la barra de *dash*, y al alcanzar el umbral necesario, el jugador puede ejecutar dicha acción a cambio de la esencia requerida. El *dash* es una mecánica de movimiento puntual: al pulsarlo mientras el personaje avanza en una dirección, *Chomp* realiza un desplazamiento acelerado en ese sentido, durante el cual es invulnerable. Esta invulnerabilidad lo convierte en una herramienta versátil: permite esquivar enemigos atravesándolos, ganar distancia de escape en la dirección contraria o acortar la distancia para cazarlos. La esencia es, además, el recurso central del sistema de medallones, que se detalla en la sección siguiente.

Los *power pellets* del original encuentran su equivalente en las *Esferas de Ruido Blanco* (*White Noise Balls*). Su efecto sobre los enemigos es análogo al del juego clásico —el comportamiento concreto de estos se describe en su sección correspondiente—, pero su funcionamiento como objeto ha sido modificado: en lugar de activarse al pisarlas, las esferas son recogidas y almacenadas como un consumible acumulativo. El jugador decide el momento de su uso, añadiendo una dimensión táctica a una mecánica que en el original era puramente reactiva.

3.1.4. Medallones

Los medallones constituyen el sistema de progresión pasiva y activa de *Chomp Crawler*, alimentado por la esencia recolectada durante el nivel. Cuando la barra de *dash* está completamente cargada, los puntos de esencia adicionales comienzan a acumularse en el medallón que el jugador tenga seleccionado en ese momento, visible en el HUD como una imagen que se va descubriendo a medida que se llena. Cada medallón dispone de cinco niveles de mejora; al superar cada umbral, el jugador recibe una mejora pasiva asociada al medallón. Al alcanzar el nivel máximo, el medallón pasa a funcionar como acumulador: una vez cargado completamente, el jugador puede activar su habilidad activa a voluntad pulsando el botón correspondiente, vaciando el medallón en el acto. Los medallones deben ser encontrados durante el nivel, a excepción del medallón de Salud, que siempre está disponible desde el inicio de la partida.

Medallón de Salud

El medallón de Salud es el único presente desde el comienzo de cada partida. Opera siempre como un medallón en nivel máximo, por lo que únicamente acumula esencia. Al completarse, genera automáticamente una vida adicional para el jugador y se vacía, comenzando un nuevo ciclo de acumulación.

Medallón de Visión

El área perceptible por el jugador se divide en dos regiones concéntricas. La región interior determina el radio de detección por parte de los ecos. Cuando este radio se reduce —ya sea por el medallón de Sigilo o por el propio entorno— emerge una región exterior en forma de aro en la que el jugador puede ver sin ser detectado. Las mejoras pasivas del medallón de Visión amplían progresivamente este aro exterior y mejoran el campo de visión general. Su habilidad activa permite al jugador visualizar la mazmorra en vista cenital: sin volverse invulnerable, puede observar el laberinto desde arriba hasta que decide moverse, momento en el que la vista regresa a la perspectiva habitual. El elevado coste de esta habilidad la reserva para decisiones tácticas clave.

Medallón de Sigilo

Las mejoras pasivas del medallón de Sigilo reducen progresivamente el radio de la región interior del foco, es decir, el área en la que el jugador es detectable por los ecos, sin afectar al aro exterior de visión. Su habilidad activa comprime este radio de detección de forma muy agresiva durante unos segundos, permitiendo al jugador atravesar zonas de alta densidad de ecos con un riesgo mínimo de ser descubierto.

Medallón de Velocidad

El medallón de Velocidad incrementa pasivamente la velocidad de desplazamiento de *Chomp* con cada nivel alcanzado. Su habilidad activa eleva la velocidad a valores extremos durante un breve intervalo, orientada a jugadores que priorizan la movilidad como herramienta de supervivencia.

Medallón de Esencia

Las mejoras pasivas del medallón de Esencia incrementan la eficiencia con la que la esencia recolectada alimenta la barra de *dash*, de modo que la misma cantidad de *pacdots* consumidos produce una carga proporcionalmente mayor. Su habilidad activa rellena la barra de *dash* al completo de forma instantánea.

Medallón de Grito

El medallón de Grito está íntimamente ligado al uso de las *Esferas de Ruido Blanco*. Sus mejoras pasivas incrementan la duración del estado de perturbación infligido a los ecos al consumir una esfera, prolongando el tiempo durante el cual huyen al volver a entrar en el foco del jugador. Su habilidad activa invierte la lógica de consumo: en lugar de requerir esferas para asustar a los ecos, el medallón completamente cargado genera directamente *Esferas de Ruido Blanco* al llenarse, recompensando al jugador con munición adicional.

3.2. Arte y Diseño Visual

El planteamiento artístico del proyecto adopta la sencillez como herramienta creativa fundamental. Si bien la creación de recursos gráficos avanzados excede el alcance técnico central de este trabajo, se ha considerado que una dirección de arte cohesionada resulta imprescindible para redondear la experiencia interactiva y dotar al videojuego de identidad propia.

Figura 3.3: Ilustración promocional generada con *Nano Banana* que condensa la identidad visual del juego: *Chomp*, el explorador, rodeado por los ecos sobre un laberinto de inspiración *arcade*.

La premisa narrativa sitúa a *Chomp* como un explorador que se adentra en un laberinto sobrenatural, cuyas propiedades alteran el comportamiento del sonido, materializando entidades hostiles —los ecos— a partir del ruido generado por el propio jugador. Para dar forma visual a este concepto, el diseño de los personajes se ha apoyado en herramientas de inteligencia artificial. Se ha empleado el modelo de generación de imágenes *Nano Banana* para explorar y definir las distintas variantes conceptuales tanto del protagonista como de los enemigos. Varias de estas imágenes han sido posteriormente procesadas y adaptadas de forma manual para su correcta integración en el videojuego, como es el caso de los elementos ilustrativos presentes en la Pantalla de Visualización Frontal. Asimismo, estas ilustraciones han servido como referencia base para la obtención de la geometría tridimensional, utilizando el servicio *Meshy AI* para producir los modelos 3D que finalmente pueblan la escena.

Figura 3.4: Modelos tridimensionales generados con *Meshy AI* a partir de las ilustraciones de referencia. A la izquierda, *Chomp*, el protagonista; a la derecha, un eco enemigo.

Un aspecto crucial para la cohesión estética ha sido la estructuración cromática, dividida en dos dominios independientes. Por un lado, la interfaz gráfica de la aplicación se rige por una paleta de color global compuesta por tres tonos de azul que establecen la jerarquía estructural, sobre un fondo azul oscuro profundo. Como contraste principal, se emplea un color de acento naranja —que evoca cromáticamente a *Chomp*— para guiar la atención hacia elementos interactivos clave, complementándose con tonos blancos para maximizar la legibilidad tipográfica y colores rojo y verde reservados semánticamente para acciones destructivas o de confirmación.

Por otro lado, la representación tridimensional de la mazmorra exige un tratamiento distinto. Los componentes estructurales de aparición masiva, como las baldosas de suelo y los bloques de pared, prescinden deliberadamente de texturas complejas, priorizando materiales de color plano. Esta decisión busca la limpieza visual y evita recargar la escena, favoreciendo a su vez el rendimiento del renderizado instanciado. Para compensar esta simplicidad y evitar la monotonía, se ha diseñado un sistema de paletas de color dependientes del nivel. Cada fase adopta una temática cromática compuesta por tres colores armónicos asignados al suelo, los muros y el vacío que envuelve el laberinto, garantizando así variedad visual y una atmósfera renovada conforme el jugador avanza en la partida.

Capítulo 4

Implementación

Este capítulo describe cómo se construye técnicamente el videojuego presentado en el capítulo anterior. El sistema se articula en torno a un *sistema lógico* —el núcleo que simula el juego, ajeno a cualquier consideración gráfica— y una *aplicación web* que lo envuelve, lo hace accesible desde el navegador y lo dota de interfaz visual.

El sistema lógico, a su vez, se compone de dos piezas independientes que conviene presentar por separado antes de unirlas. La primera es el *generador de niveles*, un módulo de generación procedural que produce, para cada partida, la mazmorra sobre la que se juega; su salida es una representación de datos del laberinto (un *tilemap*), no todavía un juego. La segunda es el *motor lógico ECS*, que toma ese *tilemap*, lo puebla de entidades y simula la partida *frame* a *frame*. El punto de unión entre ambas piezas es el ciclo de vida del motor: al comenzar cada nivel, el motor invoca al generador, ensambla el *Mundo* a partir del resultado y arranca el bucle de simulación. Finalmente, la aplicación web orquesta este conjunto y lo conecta con la interfaz gráfica y la escena tridimensional.

Siguiendo este orden, el capítulo aborda primero la generación de niveles, después el motor lógico ECS y sus detalles de implementación, y por último la aplicación web que integra todas las piezas. El código fuente completo del proyecto se encuentra disponible en su repositorio público[1], al que se remite a lo largo del capítulo para facilitar la consulta de los módulos concretos que se describen.

[1] `https://github.com/yodak025/chomp-crawler`

4.1. Generación de Niveles

La rejugabilidad, identificada como objetivo principal en la sección 1.1, exige que cada partida presente un escenario distinto, lo cual descarta el diseño manual de niveles y motiva un sistema de generación procedural. Tal y como se ha presentado en la sección dedicada a la jugabilidad, un nivel de *Chomp Crawler* no es un único laberinto, sino una mazmorra formada por múltiples salas interconectadas, cada una de ellas un laberinto al estilo *Pac-Man*. Esta dualidad estructural se traslada directamente al diseño del módulo de generación, que opera en dos capas claramente diferenciadas:

- Una capa *micro*, encargada de producir cada sala individual como un laberinto que preserva las propiedades topológicas características del diseño original de *Pac-Man*: espacio cerrado con pasillos interconectados, abundancia de bucles y túneles laterales que conectan ambos extremos horizontales.

- Una capa *macro*, encargada de organizar varias salas en una mazmorra navegable, decidir su disposición espacial, garantizar la conectividad global mediante la asignación de túneles compartidos entre salas adyacentes y distribuir las entidades del juego sobre el resultado.

La capa micro se aborda mediante una reimplementación del algoritmo desarrollado por Shaun Lebron [2], descrito en la sección 2. La elección se fundamenta en su capacidad demostrada para producir laberintos que preservan la esencia del diseño original, evitando tanto los espacios excesivamente abiertos como los corredores lineales predecibles. La capa macro, en cambio, constituye una aportación propia de este trabajo: articula las salas generadas en una mazmorra coherente apoyándose en un algoritmo de árbol de expansión mínima sobre un grafo de adyacencias entre salas.

Ambas capas producen como salida representaciones de laberintos en forma de mapas de teselas o *tilemaps*: matrices bidimensionales donde cada elemento es un identificador del tipo de tesela que ocupa esa posición (pared, pasillo, *pacdot*, *power pellet*, 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, es-

pecíficamente a través de la biblioteca NumPy. El código de este módulo reside en el directorio `src/maze-gen/` del repositorio.

4.1.1. Generación de Laberintos

Esta subsección describe la capa micro: la generación de una sala individual entendida como un laberinto al estilo *Pac-Man*. La salida de este proceso es una matriz de caracteres que representa el *tilemap* de una sala, sobre la cual la capa macro operará posteriormente.

Celdas

Para comprender el funcionamiento del algoritmo es necesario introducir el concepto de *celda* (*cell*). 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 ello, en términos de forma, resulta más útil entenderlos no 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.

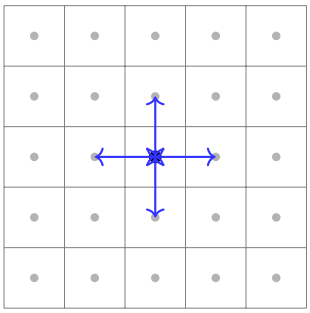
La Matriz de Celdas y sus Conexiones

Las celdas presentan una dualidad intrínseca: por un lado, son unidades discretas que componen una matriz bidimensional; por otro, actúan como nodos de un grafo que interconecta cada celda con sus vecinas ortogonalmente adyacentes (vecindad de Von Neumann). Esto permite recorrer las celdas mediante indexación matricial y, simultáneamente, navegar entre ellas a través de las conexiones establecidas.

El algoritmo comienza inicializando una matriz de celdas, donde cada celda está desconectada de sus vecinas. Es importante precisar que todas las celdas mantienen una referencia a sus adyacentes independientemente de si pertenecen o no a la misma figura poliominal. Por tanto, se define *la celda siguiente en una dirección* como la celda colindante en dicha dirección para una celda dada. Esta es siempre accesible desde la referencia que cada celda guarda de sus vecinas, y solo se considerarán conectadas las celdas que pertenezcan a la misma figura.

En este punto, el algoritmo admite la presencia de *conexiones predefinidas* entre ciertas celdas. Esta capacidad, aparentemente menor en el contexto de una sala aislada, resulta clave en

la capa macro: permite forzar la conectividad entre salas adyacentes obligando a la generación a respetar túneles previamente comprometidos.



Matriz de celdas mostrando vecindad de Von Neumann

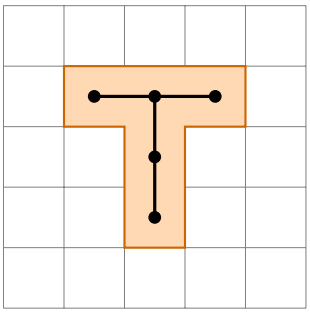


Figura poliominal (T) formada por celdas conectadas

Figura 4.1: Representación dual de las celdas: como matriz ortogonal con vecindades direccionales (izquierda) y conformando un subgrafo conectado o figura poliominal (derecha).

Generación Mediante Figuras

El núcleo del algoritmo radica en la generación iterativa de figuras poliominales que conectan celdas adyacentes. El algoritmo itera las columnas de la matriz de izquierda a derecha; en cada columna selecciona, mediante distribución uniforme, una celda aún no asignada a ninguna figura —denominada *vacía*— y comienza a construir una figura poliominal añadiéndole celdas adyacentes. La principal virtud del algoritmo de Lebron es la riqueza con la que define las reglas de expansión, manejando casos específicos y aplicando un sistema de pesos que controla la probabilidad de crecimiento en función del tamaño actual de la figura. Tras agotar la expansión, la figura se consolida marcando todas sus celdas como ocupadas. El proceso continúa hasta que no queda ninguna celda vacía. Es posible encontrar figuras conectadas con los bordes superior, inferior o derecho de la matriz.

La búsqueda deliberada de simetría axial vertical es un rasgo distintivo del diseño de los laberintos de *Pac-Man* y se conserva como decisión de diseño en *Chomp Crawler*: aporta legibilidad espacial al jugador y resulta coherente con la estética arcade que el proyecto reivindica. Para materializarla, la matriz de celdas representa únicamente una mitad del laberinto, incluyendo la columna central que actúa como eje de simetría; la otra mitad se obtiene posteriormente mediante reflexión durante la fase de proyección a teselas. Las figuras que contengan una celda

en la columna central resultan, tras la reflexión, en una única figura simétrica.

Una vez completada la fase generativa, el grafo resultante atraviesa una fase de validación, encarnada en la función `is_desirable`, que decide si la configuración obtenida puede aceptarse o si, por el contrario, debe descartarse y reiniciarse el proceso desde cero. El algoritmo original de Lebron incluye en este paso un conjunto exhaustivo de comprobaciones orientadas a preservar la fidelidad estética del laberinto canónico de *Pac-Man*, entre las que destacan el rechazo de configuraciones cuyas celdas de las esquinas superior derecha o inferior derecha conectan hacia el exterior —rompiendo la solidez visual de las esquinas del tablero— y un sistema auxiliar que rastrea celdas candidatas a ser estiradas o estrechadas para encajar el resultado en las dimensiones canónicas del tablero original.

La aplicación directa de esta lógica sobre los laberintos generados en *Chomp Crawler*, de dimensiones notablemente superiores a las del juego original, evidenció problemas de rendimiento críticos asociados a la elevada tasa de descartes. Dado que el objetivo del proyecto no exige reproducir la fidelidad geométrica del arcade, se optó por una simplificación pragmática de esta fase: el sistema de seguimiento de celdas redimensionables se elimina por completo al considerarse que su complejidad no se compensa con el valor que aporta en este nuevo contexto, y las dos comprobaciones de esquinas se relajan, ya que en el paradigma de mazmorra de salas encadenadas las paredes perimetrales no son inmediatamente visibles ni juegan el rol estético que tenían en el tablero único del original. Para cuantificar el impacto de esta decisión se instrumentó el algoritmo y se ejecutó con las distintas combinaciones de comprobaciones activas, registrando la proporción de reinicios atribuibles a cada condición; los resultados, anotados sobre el propio código fuente, sitúan la responsabilidad de la comprobación de la esquina superior derecha en torno al 33 % de los descartes y la de la esquina inferior derecha en torno al 26 %. Conviene matizar que estas cifras, medidas sobre la capa micro de forma aislada, subestiman el ahorro real: al invocarse la generación repetidamente desde la capa macro, el efecto compuesto sobre el tiempo total de generación de un nivel resulta sustancialmente mayor.

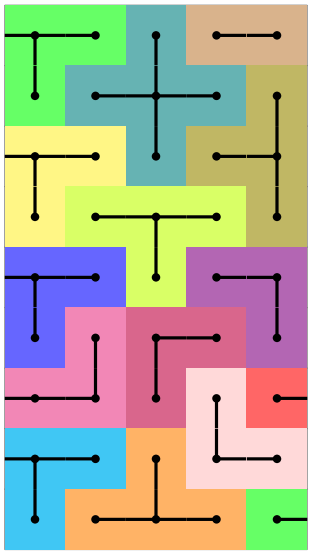


Figura 4.2: Ejemplo de matriz de celdas de 9×5 tras la generación mediante figuras poliominales. Cada color representa un grupo estructural independiente conectado por vecindad ortogonal.

Validación del Grafo Generado

Tras la fase de generación se ejecuta una validación que detecta la presencia de configuraciones 2×2 de celdas plenamente conectadas mediante análisis de patrones de conectividad. Estas formaciones reciben un tratamiento diferenciado según su ubicación:

- **Rechazo en el eje de simetría:** las figuras 2×2 situadas en la columna adyacente al eje de simetría son consideradas inválidas. Dado que la columna central no se duplica durante la reflexión, estas configuraciones generarían en el laberinto final regiones rectangulares de teselas completamente inaccesibles, violando el principio de navegabilidad total del espacio de juego. Su detección provoca el descarte de la configuración y el reinicio de la generación.

- **Corrección fuera del eje:** las figuras 2×2 localizadas en otras posiciones se corrigen automáticamente abriendo conexiones internas entre sus cuatro celdas constituyentes y unificándolas bajo un mismo grupo de pertenencia. Esta transformación convierte la formación potencialmente problemática en una figura plenamente conectada, eliminando el riesgo de huecos intransitables sin requerir regeneración.

Esta estrategia maximiza el aprovechamiento de las configuraciones generadas, requiriendo

regeneración completa únicamente ante violaciones irrecuperables. Una vez superada la validación, la generación prosigue hacia la fase de construcción de túneles.

Generación de Túneles

Los túneles laterales constituyen un elemento característico del diseño de *Pac-Man*, proporcionando rutas que conectan ambos extremos horizontales del laberinto. En el clásico arcade permiten al jugador evadir situaciones de acorralamiento atravesando los límites laterales y reapareciendo en el extremo opuesto. En *Chomp Crawler*, en cambio, los túneles adquieren un rol estructural adicional: son el mecanismo por el que la capa macro conecta salas horizontalmente contiguas, tal y como se detalla en la subsección 4.1.2. Esta doble función obliga a generalizar la generación de túneles en dos aspectos: primero, deben poder producirse *varios túneles* por sala (no un único par como en el original); segundo, los túneles del extremo izquierdo y los del extremo derecho deben poder situarse en filas distintas, ya que cada uno conecta con una sala vecina diferente. La asimetría de los túneles es, por tanto, la única ruptura intencional de la simetría general del laberinto.

La generación procedural de túneles se aborda mediante un sistema de clasificación jerárquica de candidatos sobre las celdas del borde correspondiente, distinguiendo tres categorías ordenadas por preferencia:

Túneles de vacío (*void tunnels*). Representan la configuración óptima. La celda candidata y su vecina superior no pertenecen a ninguna figura poliominal y se encuentran directamente conectadas al borde del laberinto. Esta disposición garantiza que el túnel atraviese un espacio limpio, sin interferir con la estructura interna de figuras, produciendo pasillos horizontales continuos que respetan la topología del resto del laberinto.

Callejones sin salida adyacentes (*dead-end tunnels*). Constituyen una alternativa viable cuando no se dispone de suficientes candidatos de vacío. Se sitúan junto a extremos de figuras que forman callejones sin salida en el borde. La construcción del túnel transforma el callejón en un pasillo transitable, integrándolo funcionalmente en la red de navegación. Se distinguen dos subtipos: callejones simples, donde únicamente uno de los vecinos verticales forma parte de una figura, y callejones dobles, donde tanto el vecino superior como el inferior constituyen

extremos de figuras distintas.

Túneles de borde genéricos (*edge tunnels*). Se emplean como solución de última instancia, en posiciones intermedias del borde sin consideraciones específicas sobre la estructura adyacente, siempre que no interfieran con conexiones verticales que pudieran fragmentar el túnel.

El proceso de selección opera por priorización: se evalúan primero los candidatos de vacío, recurriendo a las categorías inferiores solo en caso de insuficiencia, y la elección final dentro de la categoría más preferente disponible se realiza aleatoriamente para garantizar variabilidad.

La generalización a múltiples túneles asimétricos se implementa parametrizando la generación con dos listas independientes de filas objetivo, una para el borde izquierdo y otra para el derecho. La capa macro proporciona estas listas en función de las salas vecinas con las que cada lado deba conectarse (véase la subsección 4.1.2). Tras la selección de los túneles, una fase de validación verifica la ausencia de caminos verticales que los atraviesen —lo cual generaría discontinuidades o colisiones con la geometría vertical—; en caso de detectar incompatibilidades, se descarta la configuración completa y se reinicia la generación de la sala. Finalmente, una fase de consolidación elimina los callejones sin salida residuales creados por la inserción de los túneles, conectándolos a las estructuras adyacentes para preservar la coherencia topológica.

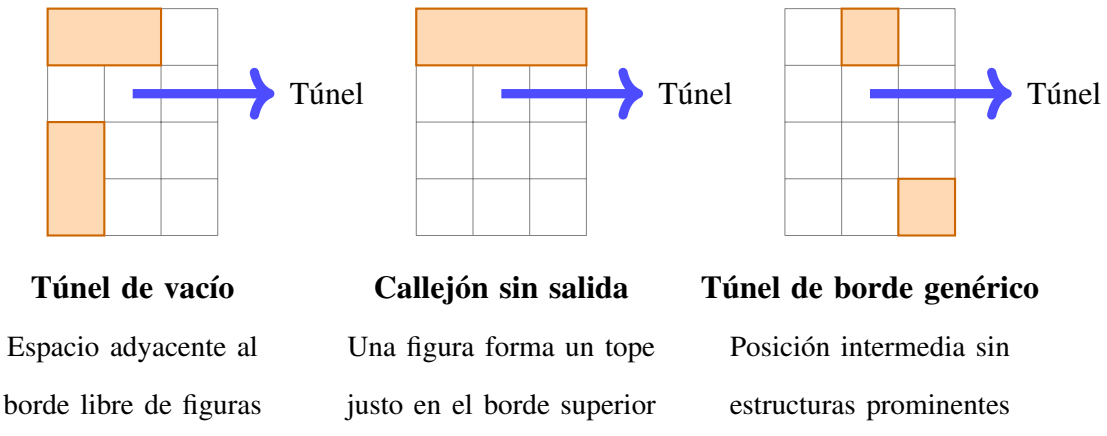


Figura 4.3: Comparativa esquemática de los tres tipos de candidatos a túnel en el borde derecho de una sala. Las zonas naranjas representan celdas ocupadas por figuras poliominales en el momento de la evaluación.

El Laberinto de Teselas

El grafo de celdas conectadas constituye una representación abstracta del laberinto, inadecuada para su renderizado directo o para la lógica de navegación discreta del juego. Se requiere una transformación que traduzca las relaciones topológicas entre celdas en una matriz de teselas (*tilemap*) donde cada elemento represente un tipo específico de componente: pasillo navegable, pared o *pacdot*.

El proceso de conversión opera mediante una expansión espacial donde cada celda del grafo se proyecta en una región de 3×3 teselas. Esta expansión permite traducir las conexiones entre celdas adyacentes en pasillos explícitos, mientras que las ausencias de conexión se materializan como paredes. La matriz resultante representa únicamente la mitad derecha del laberinto, incluyendo la columna central que actúa como eje de simetría; la mitad izquierda se obtiene mediante reflexión especular, garantizando la simetría bilateral característica del diseño original. El algoritmo identifica las fronteras entre grupos de celdas distintos y las terminaciones de figuras para establecer los pasillos navegables (marcados con `'.'`); a continuación, un proceso de relleno marca como muros (`'|'`) todas las teselas vacías adyacentes a pasillos, delimitando visualmente la geometría navegable.

Los túneles generados en la fase anterior se materializan en el *tilemap* mediante la apertura de pasillos en las posiciones extremas de las filas correspondientes. Previo a la generación de paredes, un proceso de limpieza recorre los caminos desde los bordes hacia el interior eliminando fragmentos lineales hasta alcanzar intersecciones, evitando así la aparición de callejones sin salida espurios en los límites de la sala.

A diferencia del diseño original de *Pac-Man*, en esta capa no se distribuyen *power pellets* ni elementos estructurales como la "cámara de los fantasmas" (*ghost house*). La distribución de *power pellets* y otras entidades de juego se desplaza a la capa macro (subsección 4.1.2), donde puede aplicarse criterios globales sobre la totalidad de la mazmorra, no sobre salas aisladas. En cuanto a la *ghost house*, la propia estructura jugable de *Chomp Crawler* la hace innecesaria: los ecos no necesitan un punto de respawn centralizado y, en su lugar, se distribuyen directamente sobre las salas según el criterio descrito más adelante.

El resultado final de esta capa es una matriz bidimensional de caracteres donde cada símbolo representa un tipo de tesela específico para una sala. Esta representación intermedia se entrega a la capa macro, que la integra en la mazmorra completa.

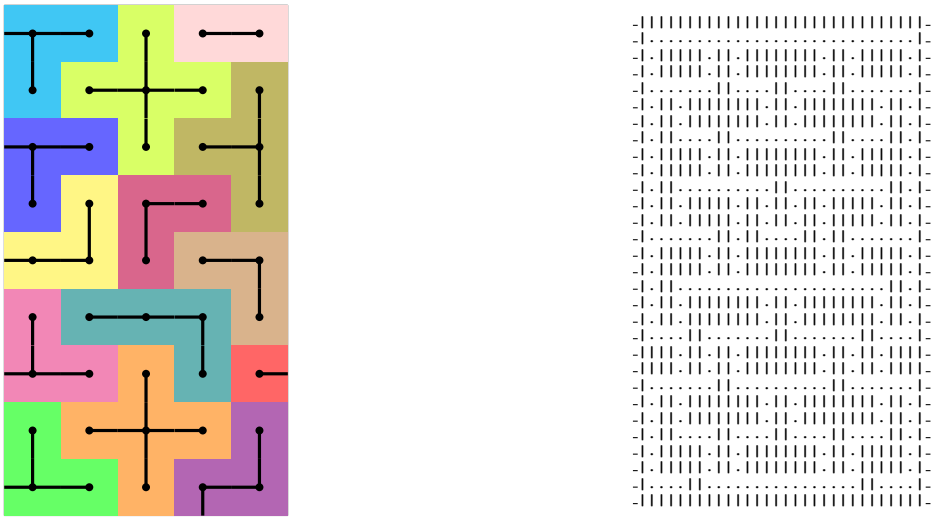


Figura 4.4: A la izquierda, representación abstracta de las celdas conectadas (una mitad del laberinto) con sus grupos estructurales marcados en color. A la derecha, el *tilemap* resultante generado a partir de estas conexiones, ya reflejado simétricamente y renderizado en formato de texto (las teselas vacías se representan con _, los caminos con . y los muros con |). Ambos elementos se derivan de la misma generación algorítmica de tamaño duplicado (9×5).

4.1.2. Generación de Mazmorras

La capa macro toma como punto de partida la capacidad descrita en la subsección anterior —generar una sala completa con un conjunto arbitrario de túneles asimétricos— y la utiliza para construir una mazmorra: una rejilla de salas dispuestas en capas horizontales, conectadas entre sí mediante los túneles laterales y, en su conjunto, navegable en su totalidad desde una sala de entrada hasta una sala de salida. Esta capa constituye la principal contribución algorítmica de este trabajo sobre el diseño original de Lebron.

Estructura en Capas y Salas

La mazmorra se organiza en *capas* (*layers*) verticales yuxtapuestas en el eje horizontal: cada capa abarca la altura completa de la mazmorra y, internamente, se subdivide en una o varias *salas* (*chunks*) apiladas verticalmente. El número de capas y la altura total de la mazmorra son parámetros de configuración fijos del nivel; el número y la altura de las salas dentro de cada capa, en cambio, se decide de manera procedural en cada generación.

Una decisión estructural importante de esta organización es que las salas *dentro de una*

misma capa permanecen aisladas entre sí: tras el ensamblado conservan íntegras sus paredes perimetrales, de modo que están separadas por un muro sólido y no son directamente alcanzables unas desde otras. La conectividad de la mazmorra se construye exclusivamente *entre capas adyacentes*, a través de túneles laterales compartidos entre una sala de una capa y una sala de la capa contigua que solapen verticalmente al menos una fila. Por tanto, para que el jugador viaje entre dos salas de la misma capa debe necesariamente rodear a través de salas de capas vecinas.

Definimos como *salas vecinas* las salas pertenecientes a capas adyacentes cuyos rangos verticales se solapan al menos parcialmente, ya que solo entre ellas es físicamente posible abrir un túnel lateral compartido. Esta noción de vecindad es la única relevante para la construcción del grafo de la mazmorra.

Particionado Probabilístico de Capas

El primer paso del proceso macro consiste en decidir, para cada capa, cuántas salas la componen y qué altura tiene cada una. Este particionado opera de forma iterativa: para cada capa se sortea el número de salas según una distribución de probabilidad fija —fuertemente sesgada hacia capas con dos o tres salas— y, una vez fijado dicho número, las alturas individuales se determinan muestreando posiciones de corte sobre la altura total de la capa, sujetas a una altura mínima que garantiza que cada sala disponga de espacio suficiente para alojar un laberinto no trivial.

Conectividad mediante Árbol de Expansión Mínima

Una vez particionadas todas las capas, se debe decidir qué pares de salas vecinas quedarán efectivamente conectadas mediante un túnel lateral compartido. La intuición de partida es que conectar todas las parejas de salas vecinas resultaría en una mazmorra con demasiados ciclos, en la que la exploración perdería sentido al existir múltiples rutas redundantes entre cualesquiera dos salas. En el otro extremo, una conectividad insuficiente fragmentaría la mazmorra en componentes inalcanzables. Se busca, por tanto, un punto intermedio: una mazmorra conexa pero suficientemente escasa en ciclos como para que el jugador deba elegir una ruta concreta y, en ocasiones, retroceder sobre sus pasos.

Esta caracterización corresponde exactamente a la noción de *árbol de expansión* sobre un grafo: un subconjunto de aristas que conecta todos los vértices sin formar ciclos. Para materia-

lizarla, se construye un grafo no dirigido donde cada vértice representa una sala y existe una arista por cada par de salas vecinas, es decir, por cada par de salas pertenecientes a capas adyacentes cuyos rangos verticales se solapan. Cada arista lleva asociado el rango de filas en el que las dos salas que conecta se solapan, ya que es dentro de ese rango donde podrá situarse el túnel compartido si la arista resulta finalmente seleccionada.

Sobre este grafo se aplica el algoritmo de Kruskal, que construye un árbol de expansión mínima seleccionando aristas de menor peso y descartando aquellas que cerrarían un ciclo, comprobándolo mediante una estructura *Union-Find*. En este caso particular los pesos se asignan aleatoriamente, de modo que el árbol resultante es uno cualquiera entre los posibles árboles de expansión del grafo, elegido con probabilidad uniforme. El árbol así obtenido garantiza dos propiedades fundamentales de la mazmorra: que sea *conexa* —toda sala es alcanzable desde cualquier otra— y que esté *libre de ciclos a nivel de salas*, lo cual evita la aparición de rutas redundantes a gran escala. Las aristas seleccionadas determinan, finalmente, el conjunto exacto de túneles que la capa micro deberá materializar en cada sala.

A las aristas seleccionadas por el árbol de expansión se les añaden dos conexiones adicionales, ajenas al MST, que materializan la entrada y la salida de la mazmorra: una sala de la primera capa recibe un túnel adicional en su borde izquierdo, que actuará como entrada del nivel, y una sala de la última capa recibe un túnel adicional en su borde derecho, que actuará como salida. Estas dos aristas no participan en el cómputo del árbol porque su existencia es una restricción del nivel y no una decisión del algoritmo, pero quedan registradas en el grafo a efectos de verificación posterior.

Una vez completada la asignación de aristas, se ejecuta una verificación de consistencia mediante una búsqueda en anchura (*Breadth-First Search*) sobre el propio grafo de salas, no sobre el *tilemap*: partiendo de una sala arbitraria del primer layer, se comprueba que todas las salas resulten alcanzables siguiendo las aristas del árbol y las aristas de entrada/salida. Esta verificación es redundante por construcción —un árbol de expansión es siempre conexo— pero actúa como salvaguarda frente a errores de *bookkeeping* en la asignación de túneles, principalmente en el cálculo de los desplazamientos relativos entre los sistemas de coordenadas locales de salas pertenecientes a capas distintas.

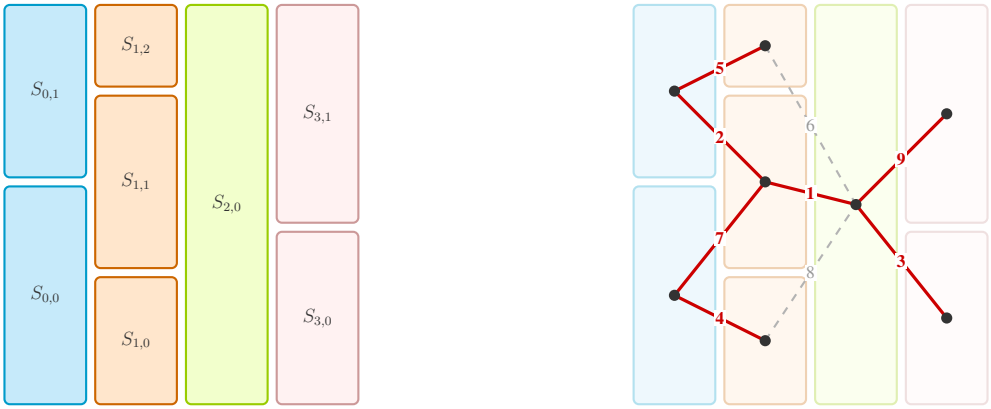


Figura 4.5: A la izquierda, esquema de una mazmorra dividida en cuatro capas verticales, particionadas probabilísticamente en distintas salas (*chunks*). A la derecha, grafo de adyacencias superpuesto donde cada nodo representa una sala y las aristas marcan el solapamiento vertical entre salas de capas contiguas; en rojo, las aristas seleccionadas por el algoritmo de Kruskal (Árbol de Expansión Mínima) que materializarán los túneles definitivos.

Asignación de Túneles a las Salas

Una vez determinado el árbol, cada arista seleccionada se traduce en una pareja de túneles laterales concretos en las dos salas que conecta. Para cada arista se sortea una fila dentro del rango vertical compartido por ambas salas, expresada en coordenadas globales de la mazmorra. Esta fila se registra como túnel del borde derecho de la sala perteneciente a la capa izquierda y como túnel del borde izquierdo de la sala perteneciente a la capa derecha. Dado que cada sala mantiene su propio sistema de coordenadas local, el algoritmo aplica el desplazamiento vertical necesario al traducir la fila global a la fila correspondiente dentro de cada sala, asegurando que ambos túneles queden efectivamente alineados tras el ensamblado.

El conjunto de túneles asignados a cada sala se traduce, finalmente, en las dos listas de filas objetivo descritas en la subsección 4.1.1 (una para el borde izquierdo y otra para el derecho), que se entregan al algoritmo de generación de la capa micro como restricción.

Pipeline de Ensamblado

Con los túneles asignados, la generación de cada sala es independiente: la capa micro recibe la configuración de túneles y produce el *tilemap* correspondiente. La capa macro orquesta este proceso reintentando la generación de una sala individual hasta un número máximo de veces

si el algoritmo micro fracasa; si tras agotar los reintentos no se obtiene un resultado válido, se descarta la mazmorra completa y se reinicia el proceso desde el particionado, dado que la configuración de túneles asignada podría ser inviable.

Una vez generadas todas las salas, sus *tilemaps* se ensamblan en una única matriz mediante dos operaciones de concatenación encadenadas. En primer lugar, dentro de cada capa, las salas se apilan verticalmente conservando íntegras sus paredes perimetrales, lo cual produce el muro sólido entre salas de la misma capa al que se aludía anteriormente. En segundo lugar, las capas resultantes se yuxtaponen horizontalmente; los túneles laterales asignados por el MST en los bordes derecho e izquierdo de salas vecinas quedan así enfrentados y producen, en el *tilemap* resultante, los pasajes que conectan capas adyacentes. Finalmente, se añaden dos columnas frontera, una a cada lado de la mazmorra, en las que se abre exclusivamente la celda correspondiente al túnel de entrada (a la izquierda) y al túnel de salida (a la derecha), materializándolos como puertas del nivel.

Distribución de Entidades

La última fase de la capa macro consiste en distribuir las entidades de juego sobre el *tilemap* ensamblado. A diferencia de la capa micro —que produce salas pobladas únicamente de pasillos transitables y *pacdots*—, esta distribución se realiza con visión global de la mazmorra, lo cual permite aplicar criterios espaciales coherentes a gran escala.

Las entidades a distribuir son las *Esferas de Ruido Blanco* (equivalentes a los *power pellets* del original), el personaje del jugador (*Chomp*) y los *ecos* enemigos. La distribución sigue una estrategia de *estratificación espacial*: la mazmorra se divide en una rejilla regular de regiones y cada entidad se sitúa aleatoriamente dentro de una región concreta, sobre una tesela transitable. Esta estratificación garantiza una dispersión homogénea, evitando concentraciones que desequilibren la dinámica de juego: ni todos los enemigos en una misma sala, ni todas las esferas próximas al jugador, ni la entrada y la salida en regiones inmediatamente adyacentes. La sala que contiene la posición inicial del jugador se designa como sala de entrada y se utiliza como origen para la verificación de conectividad descrita en el apartado anterior.

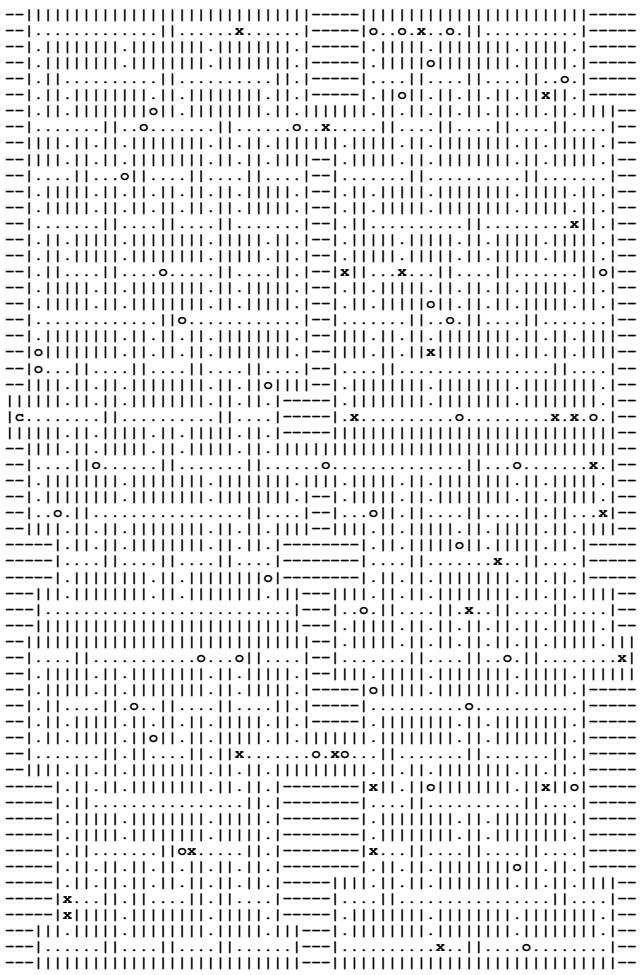


Figura 4.6: Ejemplo de una mazmorra completa ensamblada por la capa macro, compuesta por 2 capas verticales con múltiples salas (*chunks*) conectadas mediante túneles asimétricos. Se observa la distribución espacial de entidades: jugador (`c`), ecos (`x`), bolas de ruido blanco (`o`), pacdots (`.`), muros (`|`) y zonas de vacío (`_`).

El *tilemap* resultante, junto con las posiciones iniciales de las entidades, constituye la salida completa del módulo de generación. Esta representación se entrega al motor lógico para su posterior conversión a una matriz numérica y su consumo tanto por el motor gráfico tridimensional como por los sistemas de navegación y colisión del juego.

4.2. Motor Lógico ECS

El núcleo computacional del videojuego se ha construido siguiendo la arquitectura *Entity Component System* (ECS). La adopción de este paradigma responde a la necesidad de gestionar

una gran cantidad de elementos dinámicos interactuando de forma concurrente, evitando los problemas de rigidez, acoplamiento y fragmentación de memoria que tradicionalmente limitan a las jerarquías de herencia de la programación orientada a objetos convencional. En esta sección se ahonda en los detalles asociados a su implementación.

4.2.1. El Mundo como Registro Central

En pocas palabras, el *Mundo* (*World*) es la estructura de datos que guarda todo el estado de la partida en un único lugar: dónde está cada entidad, qué propiedades tiene y cuál es la situación general del juego en cada instante. El motor no almacena nada relevante para la lógica fuera de él, de modo que basta inspeccionar el *Mundo* para conocer la situación completa de la simulación. El código correspondiente reside en el módulo `src/core/` del repositorio.

El motor se articula en torno a este objeto único, que actúa como contenedor exclusivo de la simulación: la totalidad del estado interno del juego reside en él y, en sentido inverso, ningún dato relevante para la lógica del juego se mantiene fuera de su ámbito. Esta centralización es el rasgo arquitectónico que vertebra el resto del motor, en la medida en que erige al *Mundo* como única fuente de verdad sobre la que operan los distintos sistemas. En su interior, las *entidades* son meros identificadores opacos, carentes de estado y comportamiento propios; toda su naturaleza queda determinada por el conjunto de *componentes* que se les asocia en un instante dado.

La organización interna del *Mundo* es **orientada a datos**: los componentes no se almacenan agrupados por entidad, como sería natural en un modelo orientado a objetos, sino agrupados por tipo. En términos conceptuales, el *Mundo* mantiene una colección de tablas, una por cada tipo de componente, en las que cada fila asocia un identificador de entidad con la instancia concreta del componente correspondiente. Esta disposición condiciona la forma en que los *sistemas* interrogan al estado: un sistema no formula la pregunta de qué componentes posee una entidad determinada, sino la pregunta inversa de qué entidades poseen una combinación dada de componentes. A dicha combinación se la denomina *firma de componentes*, y constituye el criterio uniforme mediante el cual cada sistema selecciona en cada *frame* la población de entidades sobre la que actúa. De este modelo se sigue una consecuencia arquitectónica inmediata: toda información que dos sistemas requieran compartir ha de expresarse necesariamente como un componente, imponiendo así una disciplina explícita sobre la representación del estado del

juego.

4.2.2. Tipología de Componentes

Los componentes son estructuras de datos puras, desprovistas de métodos y de lógica, que en su conjunto desempeñan tres roles funcionales claramente diferenciados dentro del motor. Si bien el catálogo exhaustivo excede el alcance de esta exposición, conviene presentar las tres familias junto con algunos ejemplos representativos de cada una, a fin de que la descripción posterior de los sistemas resulte inteligible.

- **Componentes de identidad**: actúan como etiquetas que clasifican cualitativamente a la entidad. No portan estado dinámico significativo, sino que indican qué *es* la entidad dentro del juego; tal es el caso de las etiquetas que distinguen a Chomp de un eco. Su utilidad principal consiste en restringir la firma de componentes que los sistemas exigen, garantizando que cada sistema actúe únicamente sobre la categoría de entidades que le concierne.

- **Componentes de estado**: contienen los datos que evolucionan a lo largo de la partida y describen la situación instantánea de la entidad. En ellos reside el grueso de la información del juego: la posición sobre la mazmorra, la dirección de movimiento solicitada, el modo de comportamiento de un eco, su objetivo de persecución, los puntos de salud de Chomp, su reserva de esencia, los temporizadores que regulan el ritmo de movimiento o la lista de medallones equipados. Estos componentes son leídos y mutados por los sistemas en cada *frame*.

- **Componentes de evento**: marcadores efímeros que registran un suceso puntual y solo resultan pertinentes durante el *frame* en que han sido emitidos. Constituyen el mecanismo por el que un sistema notifica a otro la ocurrencia de un suceso, ya sea una colisión entre Chomp y un eco, la recolección de una bola de esencia o la eliminación de un eco vulnerable, sin necesidad de establecer un canal de comunicación externo al *Mundo*. Una fase de limpieza ulterior los retira al término de cada ciclo.

La distinción entre componentes de estado y componentes de evento responde a una decisión deliberada: los primeros describen *lo que la entidad es ahora*, mientras que los segundos

describen *lo que le ha sucedido en este instante*. Tal demarcación simplifica el razonamiento sobre el flujo de información dentro del motor y previene la contaminación del estado persistente con datos de naturaleza puntual.

4.2.3. Inventario de Entidades del Juego

Definidos el *Mundo*, los componentes y la noción de entidad, procede enumerar qué entidades existen realmente en *Chomp Crawler*. Pese a la multitud de elementos que el jugador percibe en pantalla, el *Mundo* solo modela como entidad aquello que posee comportamiento dinámico propio, esto es, que se mueve o toma decisiones a lo largo de la partida. En la práctica, el inventario se reduce a dos tipos de entidad, cada una caracterizada por la combinación de *componentes* que se le adjudica al instanciarse:

- **Chomp**: la única entidad bajo control del jugador. Combina componentes de identidad que la marcan como tal, componentes de posición que registran su localización sobre la mazmorra, componentes de intención que recogen las órdenes de movimiento, componentes de estado vital que gestionan su salud y su invulnerabilidad temporal, y componentes de recursos que regulan las mecánicas de *dash*, esencia y medallones equipados.

- **Ecos**: las entidades antagonistas, instanciadas en número variable según la configuración del nivel. Cada eco lleva componentes de identidad, posición sobre la mazmorra, modo de comportamiento, que selecciona entre dispersión, persecución, vulnerabilidad y eliminación, un objetivo dinámico que orienta sus decisiones de movimiento y un temporizador que regula su cadencia.

Cabe señalar una ausencia deliberada en este inventario: las paredes de la mazmorra y los elementos coleccionables —bolas de esencia, esferas de ruido blanco y medallones depositados— no se modelan como entidades, pese a ser entes presentes en el universo del juego. Por tratarse de una desviación respecto al modelo canónico de ECS, su justificación se desarrolla más adelante, en la subsección 4.3.2, dedicada a las adaptaciones pragmáticas del modelo.

4.2.4. Sistemas: Funciones que Transforman el Mundo

Los *sistemas* son las unidades funcionales encargadas de hacer evolucionar el estado del juego de un *frame* al siguiente. Cada sistema se concibe como una función pura que opera sobre el *Mundo* y que articula su trabajo en tres fases conceptuales: la selección de entidades cuya firma de componentes coincide con la requerida, la lectura de los componentes pertinentes de dichas entidades y la mutación del estado del *Mundo* en consecuencia. Esta mutación puede traducirse en la modificación de componentes existentes, en la incorporación o retirada de componentes a una entidad e incluso en la creación o destrucción de entidades completas.

Los sistemas no mantienen estado interno propio entre invocaciones: toda la información que precisan debe leerse del *Mundo* en el momento de su ejecución, y todo efecto que pretendan dejar ha de escribirse de vuelta en él. Algunos sistemas reciben, además del propio *Mundo*, un parámetro adicional con el tiempo transcurrido desde el *frame* anterior (*delta time*), valor que les permite operar con dinámicas continuas de manera independiente a la cadencia concreta de ejecución del bucle.

De esta concepción se sigue que la comunicación entre sistemas no se produce por invocación directa entre ellos, sino exclusivamente a través del *Mundo*. Los componentes de estado actúan como canal implícito de comunicación persistente, en la medida en que un sistema escribe en ellos lo que otros leerán más adelante, y los componentes de evento actúan como canal explícito de comunicación puntual entre sistemas que detectan sucesos y sistemas que aplican sus consecuencias. Esta uniformidad en el medio de comunicación es la que permite presentar el bucle de juego como una secuencia ordenada y determinista de sistemas, descrita en la sección siguiente.

4.3. Detalles de implementación del motor

Las secciones anteriores han presentado el motor ECS en su forma conceptual: el *Mundo*, los componentes, las entidades y los sistemas. Esta sección reúne los detalles concretos de su implementación efectiva, que se apartan en distintos grados del modelo teórico. Se aborda, en este orden, la composición exacta del bucle de juego como *pipeline* determinista, las adaptaciones pragmáticas respecto al ECS canónico, la separación entre estado frío y estado caliente que conecta el motor con la interfaz, y el ciclo de vida completo de una partida.

4.3.1. El Bucle de Juego como Pipeline Determinista

El motor lógico se ejecuta como un bucle continuo articulado en torno al mecanismo de solicitudes de fotograma del navegador (*frame requests*), que delega en él la decisión de cuándo invocar cada nueva iteración. En cada *frame*, el motor calcula el tiempo transcurrido desde la iteración anterior (*delta time*) y a continuación ejecuta, en un orden estricto y predefinido, la totalidad de los sistemas que componen el *pipeline*. Esta secuencialidad rigurosa es el mecanismo que garantiza la coherencia del estado: la salida de cada sistema constituye la entrada validada del siguiente, eliminando de raíz cualquier ambigüedad derivada de un orden de evaluación no determinado.

El *pipeline* se organiza en ocho fases consecutivas, cada una de las cuales agrupa los sistemas afines por su rol funcional dentro del flujo del *frame*. La descripción que sigue enumera los sistemas en su orden exacto de ejecución, acompañando cada uno de una caracterización sintética de su cometido.

- **Fase 1 — Captura de entrada.**
  - *Sistema de captura de entrada*: traduce el estado instantáneo del teclado a un componente de pulsaciones accesible por el resto del motor.
  - *Sistema de entrada de habilidades*: registra las pulsaciones específicas asociadas a las habilidades del jugador, separándolas de las teclas de movimiento.
  - *Sistema de atributos del jugador*: recalcula los atributos efectivos de Chomp (radios de visión y agresión, multiplicadores de velocidad, capacidades de *dash* y recolección) a partir de los medallones equipados en cada *frame*.
- **Fase 2 — Resolución de intención del jugador.**
  - *Sistema de intención del jugador*: traduce las pulsaciones de movimiento en una dirección efectiva, respetando las reglas de giro en intersecciones y la posibilidad de almacenar transitoriamente una dirección pretendida no satisfacible de inmediato.
- **Fase 3 — Movimiento del jugador.**
  - *Sistema de energía de* dash: actualiza los temporizadores asociados a la habilidad de *dash*, restaurando la velocidad ordinaria una vez expirada la ventana de impulso.

- *Sistema de* dash: activa la habilidad de impulso cuando concurren la intención del jugador y la disponibilidad de energía suficiente.
- *Sistema de movimiento continuo*: desplaza la posición continua de Chomp en función de su dirección efectiva, su velocidad y el *delta time*, deteniéndolo ante muros.
- *Sistema de alineación*: corrige la posición continua de Chomp para alinearla con la rejilla del *tilemap* cuando el jugador no impone una dirección activa.
- *Sistema de sincronización de posición discreta*: deriva las coordenadas enteras de Chomp a partir de su posición continua, de modo que el resto de sistemas puedan razonar sobre ellas en términos de celdas del *tilemap*.

- **Fase 4 — Inteligencia y movimiento de los ecos.**
  - *Sistema de modos de comportamiento*: gobierna las transiciones entre los modos de cada eco (dispersión, persecución, vulnerabilidad, eliminación) en función del estado del juego y de los temporizadores asociados.
  - *Sistema de selección de objetivo*: calcula, para cada eco, la celda objetivo que guiará su próximo movimiento, en consonancia con su modo actual.
  - *Sistema de dirección heurística*: elige la dirección de movimiento que minimiza la distancia al objetivo evitando muros e inversiones de marcha.
  - *Sistema de movimiento discreto*: aplica un desplazamiento de una celda completa a cada eco cuando su temporizador interno señala el momento de moverse.

- **Fase 5 — Detección de interacciones, aplicación de efectos y temporizadores.**
  - *Sistema de colisión entre entidades*: detecta cualquier coincidencia espacial entre Chomp y un eco y emite el correspondiente componente de evento de colisión.
  - *Sistema de detección de recolección*: detecta la presencia de un coleccionable bajo la posición de Chomp y emite el correspondiente componente de evento de recolección.
  - *Sistema de daño*: procesa los eventos de colisión, aplicando daño a Chomp o, cuando el eco implicado se encuentra en estado vulnerable, emitiendo a su vez un evento de eco eliminado.

- *Sistema de eco eliminado*: procesa los eventos de eco eliminado, otorgando los beneficios correspondientes al jugador y marcando al eco como retirado del juego.
- *Sistema de efectos de recolección*: procesa los eventos de recolección, retirando el coleccionable del *Mundo* y aplicando sus consecuencias específicas (puntuación, recarga de *dash*, incorporación de medallones o reservas de Esferas de Ruido Blanco).
- *Sistema de activación de Esfera de Ruido Blanco*: consume una esfera de la reserva del jugador y aplica el estado vulnerable a los ecos comprendidos dentro del radio de visión.
- *Sistema de carga de medallones*: gestiona la rotación entre los medallones disponibles en respuesta a las pulsaciones del jugador.
- *Sistema de activación de medallones*: dispara la habilidad asociada al medallón seleccionado cuando el jugador así lo solicita.
- *Sistema de invulnerabilidad*: decrementa el temporizador de invulnerabilidad temporal de Chomp tras recibir daño.
- *Sistema de actualización de temporizadores*: avanza los temporizadores genéricos de las entidades en función del *delta time*, marcando aquellos que han alcanzado el umbral para el siguiente paso.
- *Sistema de tics de comportamiento*: decrementa los contadores específicos que regulan las transiciones entre modos de los ecos.

- **Fase 6 — Condiciones de finalización.**
  - *Sistema de victoria*: comprueba si Chomp ha alcanzado la celda de salida de la mazmorra y, en tal caso, transita el estado global del juego a la fase de victoria.
  - *Sistema de derrota*: comprueba si la salud de Chomp se ha agotado y, en tal caso, transita el estado global del juego a la fase de derrota.
- **Fase 7 — Sincronización con el estado externo.**
  - *Sistema de sincronización*: proyecta el subconjunto del estado del *Mundo* relevante para la representación visual e informativa hacia las estructuras externas que consume la interfaz del juego, según se detalla en la sección 4.3.3.

- **Fase 8 — Limpieza.**

  - *Sistema de limpieza de eventos*: retira del *Mundo* los componentes de evento emitidos durante el *frame*, garantizando que el siguiente ciclo comience con un estado libre de notificaciones residuales.

Concluida la fase de limpieza, el motor solicita al navegador el siguiente *frame* y reinicia el ciclo. La rigidez del orden de fases y la independencia de cada sistema respecto a sus pares constituyen, en conjunto, las dos garantías arquitectónicas que sostienen la previsibilidad del comportamiento del juego.

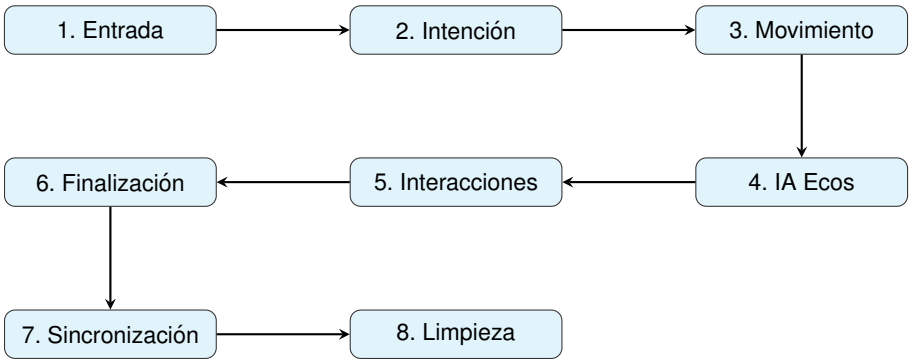


Figura 4.7: Ejecución secuencial de los sistemas lógicos en el *pipeline* del motor ECS.

4.3.2. Adaptaciones Pragmáticas del Modelo Canónico

La arquitectura descrita en las subsecciones precedentes responde, en términos generales, a las prescripciones del modelo *Entity-Component-System* expuesto en la sección 2.2.3. No obstante, su implementación efectiva en este proyecto incorpora ciertas concesiones pragmáticas respecto al canon teórico, motivadas por las particularidades del dominio del juego y por el coste que tendría una adhesión estricta. Conviene explicitarlas, pues delimitan con precisión la frontera entre la arquitectura ideal y la arquitectura realmente desplegada [5].

La primera adaptación, ya anticipada en la subsección 4.2.3, consiste en la exclusión deliberada de las paredes y los coleccionables del registro de entidades del *Mundo*. Esta exclusión se justifica por dos consideraciones de orden práctico. En primer lugar, dichos elementos carecen de comportamiento dinámico: no se mueven, no toman decisiones y su único cambio de estado posible es la propia desaparición en el caso de los coleccionables. Modelarlos como

entidades ECS supondría dotarlos de una maquinaria conceptual que jamás se ejercitaría. En segundo lugar, su número es muy elevado, del orden de millares de celdas para una mazmorra de tamaño habitual, y los sistemas que interactúan con ellos —en particular los responsables de la detección de colisiones contra paredes y de la recolección de objetos— requieren consultar en cada *frame* si una posición concreta de la mazmorra contiene un elemento de estos tipos. Una representación basada en entidades obligaría a recorrer linealmente la población candidata para cada consulta, lo que penalizaría sustancialmente el rendimiento del bucle de juego. Por estas razones, paredes y coleccionables se representan como *estructuras espaciales indexadas por posición*, ajenas al registro de entidades del *Mundo*, que ofrecen acceso de coste constante a la información asociada a cada celda.

La segunda adaptación afecta al jugador. Chomp posee, simultáneamente, una posición continua expresada en coordenadas reales y una posición discreta expresada en celdas del *tilemap*. La primera gobierna su desplazamiento fluido y su representación visual; la segunda permite resolver con eficiencia las consultas de adyacencia, las colisiones con la geometría del laberinto y la detección de coleccionables. Un sistema dedicado actúa como puente entre ambas representaciones, proyectando la posición continua sobre la rejilla discreta una vez por *frame*. Esta duplicidad rompe la ortogonalidad de los componentes propia del modelo canónico, pero resulta inevitable dada la coexistencia, dentro de la misma escena, de un protagonista de movimiento continuo y de antagonistas de movimiento discreto.

La tercera adaptación concierne a la comunicación entre sistemas. El modelo canónico postula que los sistemas se comunican exclusivamente a través del estado persistente de los componentes. Sin embargo, ciertas interacciones del juego —colisiones, recolecciones, eliminaciones de ecos— son puntuales por naturaleza y no requieren persistencia entre *frames*. Modelarlas mediante componentes ordinarios obligaría a sus consumidores a comprobar marcadores y a sus productores a gestionar la limpieza, complicando innecesariamente la lógica. La solución adoptada consiste en emplear componentes de evento, descritos en la subsección 4.2.2, que viven un único *frame* y son retirados en bloque por un sistema de limpieza al cierre del ciclo. Esta convención permite preservar el principio de comunicación a través del *Mundo* sin incurrir en el coste de un sistema de mensajería adicional.

La cuarta adaptación se refiere al estado global del juego. Magnitudes como la fase actual de la partida, la puntuación acumulada o las dimensiones del *tilemap* no pertenecen propiamente a

ninguna entidad del juego, sino que describen el contexto en el que todas ellas se desenvuelven. El modelo canónico recomendaría representarlas mediante una entidad singleton portadora de los componentes correspondientes. En este proyecto, sin embargo, dichas magnitudes se almacenan directamente como atributos del *Mundo*, accesibles a todos los sistemas sin mediación de consultas por componente. Esta decisión simplifica la implementación a costa de introducir una asimetría entre el estado global y el estado de las entidades, asimetría que se ha considerado aceptable dada la naturaleza acotada del proyecto.

4.3.3. Estado Frío y Estado Caliente

La descripción del *Mundo* presentada hasta el momento lo caracteriza como un registro autosuficiente, capaz de albergar la totalidad del estado del juego y de servir como sustrato común a los sistemas. Esta caracterización es exacta desde la perspectiva de la lógica del juego, pero requiere un complemento cuando se considera la integración con la capa de presentación. La interfaz gráfica, construida sobre el paradigma reactivo de la biblioteca empleada, demanda un modelo de estado distinto: uno que notifique automáticamente a los componentes visuales cuando los valores observados cambian, evitando así la consulta explícita en cada *frame*. El *Mundo* no satisface esta exigencia, pues su naturaleza es la de una estructura mutable accedida de forma imperativa por los sistemas.

La solución adoptada distingue dos categorías complementarias de estado, denominadas, respectivamente, estado frío y estado caliente. El estado frío reside en el *Mundo* y constituye la fuente canónica de verdad de la simulación: contiene la totalidad de las entidades, sus componentes y las magnitudes globales de la partida. Sobre él operan los sistemas, modificándolo durante la ejecución de cada fase del bucle. Su acceso es directo, síncrono y no notifica a observadores; su rendimiento es, en consecuencia, óptimo para las operaciones intensivas que la lógica del juego requiere a sesenta *frames* por segundo.

El estado caliente reside, en cambio, en una colección de almacenes reactivos externos al *Mundo*. Cada almacén agrupa un subconjunto coherente de magnitudes observables —el estado general de la partida, los atributos visibles del jugador, los modos de comportamiento de los ecos, los metadatos del laberinto, los indicadores de depuración— y expone interfaces de suscripción que los componentes de la interfaz gráfica consumen para sincronizar su representación. La separación en múltiples almacenes obedece al principio de cohesión: cada componente

visual se suscribe únicamente al subconjunto de estado que efectivamente consume, evitando re-renderizados innecesarios y preservando la independencia entre las regiones de la interfaz, según se detalla en la sección 2.

La relación entre ambos tipos de estado es estrictamente unidireccional: el estado frío alimenta al estado caliente, nunca a la inversa. Un sistema dedicado, ejecutado en la séptima fase del bucle descrita en la subsección 4.3.1, recorre el subconjunto del *Mundo* relevante para la presentación y proyecta sus valores actualizados sobre los almacenes reactivos. Esta proyección es selectiva: no todo el estado frío se replica, sino únicamente aquellas magnitudes que la interfaz observa. La frontera entre ambos mundos queda así nítidamente establecida y la lógica del juego permanece desacoplada de las particularidades del sistema de presentación.

Esta arquitectura dual presenta dos virtudes que justifican su complejidad aparente. En primer lugar, preserva la pureza del *Mundo* como modelo computacional: los sistemas operan sobre estructuras de datos planas, sin la sobrecarga que impondría un mecanismo de notificación reactivo en cada modificación de componente. En segundo lugar, aísla la lógica del juego de la tecnología concreta empleada para la interfaz: una eventual migración del sistema de almacenes reactivos no exigiría modificación alguna en los sistemas, sino únicamente en el sistema de sincronización que actúa como puente entre ambas capas.

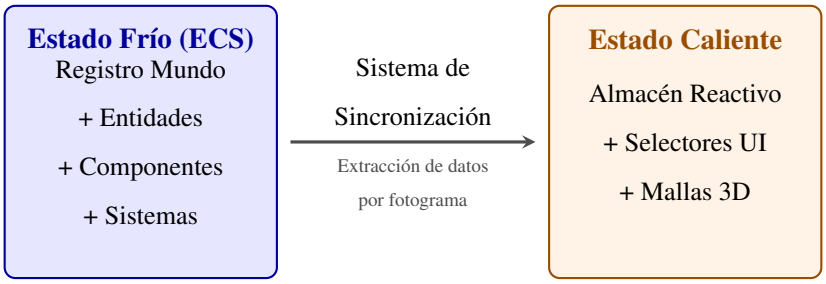


Figura 4.8: Esquema conceptual de la arquitectura dual de estados. El motor (izquierda) procesa la simulación de forma aislada e imperativa, sincronizando fotograma a fotograma las variables observables hacia los almacenes reactivos (derecha) que consumen los componentes visuales.

4.3.4. El Ciclo de Vida de la Partida

El motor lógico, una vez instanciado, no se encuentra inmediatamente en disposición de ejecutar el bucle descrito en las subsecciones precedentes. Su estado inicial es el de una estructura inerte: el *Mundo* existe pero está vacío de entidades, los sistemas están definidos pero no se

invocan, y el bucle de animación no ha comenzado a solicitar *frames* al navegador. Esta inactividad inicial responde a una decisión arquitectónica deliberada: la construcción del motor y la preparación de una partida concreta son operaciones distintas, separadas por una orden explícita de comienzo.

Cuando dicha orden se recibe, el motor entra en una fase de carga durante la cual reinicia el *Mundo*, invoca al subsistema de generación procedural para obtener el laberinto del nivel y traduce el *tilemap* resultante en las entidades y estructuras espaciales que pueblan el *Mundo*, según los criterios expuestos en la subsección 4.2.3. Concluida la carga, el motor declara la partida lista pero aún suspendida; el bucle no comienza hasta que se recibe una segunda orden, separada de la primera, que transita el estado global al régimen de juego activo y emite la primera solicitud de *frame*. Esta doble orden permite que la capa de presentación interponga las transiciones visuales que considere oportunas entre la preparación del estado y el inicio de la simulación.

Durante el régimen activo, el bucle ejecuta el pipeline completo de manera ininterrumpida. La pausa constituye la única forma de suspender este régimen sin terminar la partida. Cuando se solicita, el motor deja de pedir nuevos *frames* al navegador y transita el estado global al régimen pausado, proyectando una última vez el estado caliente para que la interfaz refleje el cambio. Mientras la pausa persiste, ningún sistema se ejecuta, ningún temporizador avanza y el estado frío permanece exactamente como estaba en el último *frame* procesado. La reanudación es estrictamente simétrica: una orden explícita devuelve el estado global al régimen activo y reinicia las solicitudes de *frame*, retomando el bucle desde el punto donde se detuvo.

La partida termina por una de tres rutas. La victoria se produce cuando el sistema correspondiente, ejecutado en la sexta fase del pipeline descrito en la subsección 4.3.1, detecta que Chomp ha alcanzado la celda de salida del nivel; el motor entonces se detiene, sincroniza una última vez el estado caliente y aguarda la orden de avanzar al siguiente nivel. Esta orden desencadena un procedimiento específico de transición que persiste el progreso acumulado del jugador —medallones recolectados, barra de esencia, salud restante, energía de *dash* e inventario de Esferas de Ruido Blanco— en el *Mundo*, lo reinicia preservando puntuación y nivel, incrementa este último, suma una bonificación y reinvoca la secuencia de carga, devolviendo así el ciclo a su segunda fase. La derrota, detectada por el sistema homólogo cuando la salud de Chomp se agota, sigue el mismo patrón de detención y sincronización pero conduce a un estado

terminal del que solo se sale reiniciando completamente el ciclo, descartando todo el progreso. El abandono, finalmente, es una transición que el jugador puede solicitar desde el régimen activo, desde la pausa o desde las pantallas terminales: detiene el bucle y reinicia el *Mundo* de manera incondicional, devolviendo el motor a su estado inicial inerte y cerrando así el ciclo de vida.

4.4. Aplicación Web

El resultado de los conceptos y sistemas descritos en las secciones anteriores es una aplicación web desarrollada con Next.js. Esta ofrece el videojuego como una única página, junto con una página de aterrizaje que actúa como punto de entrada y como sitio web del proyecto. Además, se incluyen páginas secundarias con información adicional del proyecto. En esta sección se detallan los aspectos de esta aplicación.

4.4.1. Arquitectura

La aplicación está compuesta por distintas páginas. En la ruta raíz se ubica una página de aterrizaje que permite al usuario conocer información sobre el proyecto y avanzar hacia el resto de páginas de la aplicación. La página principal se ubica en la ruta */game* y es el punto de entrada del videojuego. Adicionalente se ha añadido la ruta */styleguide*, que reune los componentes gráficos del proyecto.

El videojuego funciona gracias a la interacción de tres áreas diferenciadas: El sistema lógico, las interfaces gráficas y la escena tridimensional. Estas áreas son independientes y se relacionan mediante un estado global. A continuación, se detalla el flujo de ejecución en clave temporal.

Al comenzar la partida, la aplicación prepara una instancia de pyodide. Esto es necesario para poder ejecutar el algoritmo de generación en este entorno web, y se realiza una única vez de forma independiente a la partida. El proceso comienza descargando el *runtime* de pyodide, el componente escrito en WebAssembly que permite ejecutar código python en el navegador. A continuación, se descargan los módulos y las dependencias del generador de laberintos. La interfaz gráfica muestra un mensaje de carga al usuario durante el proceso y al finalizar muestra el menú principal del juego.

Cuando el jugador pulsa el botón *PLAY* se crea una instancia del motor lógico, que recibe

la instancia de pyodide y prepara el estado inicial del nivel. Para esto, genera un tilemap y lo procesa, creando una instancia interna del *Mundo* y poblándola a partir del mismo con entidades *ECS*. Cuando el mundo se encuentra preparado para compenzar, el motor activa el bucle principal del juego, que ejecuta los sistemas *ECS*. El jugador podrá observar de nuevo un mensaje de carga durante este proceso, que puede demorar algunos segundos debido a los tiempos de generación. Cuando el nivel está listo, la interfaz pasa a mostrar la escena tridimensional que representa los elementos del *mundo*, así como una pantalla de visualización frontal superpuesta, que ofrece información relevante al jugador sobre el estado de la partida, como el número de vidas o la cantidad de esencia disponible.

Durante el transcurso del nivel, el jugador podrá pausar el juego, accediendo a un menú que permite reanudar la partida, reiniciar el juego o volver al menú principal.

La partida finaliza si el jugador pierde todas sus vidas. En este escenario, el juego se pausa y la aplicación muestra un elemento gráfico que informa al jugador del fin del juego. Se ofrece la opción de reinicio y también de regreso al menú principal.

Si el jugador comleta el nivel, alcanzando la salida de la mazmorra, el juego se pausa y se muestra un elemento gráfico informando de la victoria. Se muestra la puntuación incrementada tras alcanzar la salida y se ofrece la opción de avanzar al siguiente nivel. También se ofrecen las habituales opciones de reinicio y vuelta al menú principal.

4.4.2. Estado

La convivencia de tres entornos diferenciados dentro del proyecto —el motor lógico, la interfaz gráfica bidimensional y la escena tridimensional— exige un mecanismo de comunicación sólido. Para evitar la transmisión sucesiva e innecesaria de propiedades a través de toda la jerarquía de componentes, se ha optado por implementar un estado global centralizado mediante la biblioteca *Zustand*. Este enfoque proporciona un único punto de acceso reactivo, permitiendo que cualquier elemento visual, independientemente de su profundidad en el árbol estructural, consuma la información compartida.

La orquestación general del programa se delega a un registro dedicado al estado de la navegación. Este almacén controla el flujo de la aplicación de página única, gestionando las transiciones entre las distintas vistas estructurales. Se encarga de mostrar los menús principales, coordinar las pantallas de carga durante la inicialización de dependencias externas como *Pyo-*

dide y habilitar el lienzo principal del juego. Esta separación garantiza que el ciclo de vida de la aplicación opere de forma completamente ajena a la lógica de la simulación del videojuego.

En lo relativo a la partida, se ha establecido una frontera arquitectónica estricta entre la simulación algorítmica y la representación gráfica. Toda la lógica subyacente reside en el registro central del motor, denominado estado frío. Se trata del registro del mundo mencionado en la sección previa, sobre el funcionamiento de la partida a nivel lógico. Este componente opera exclusivamente sobre estructuras de datos planas, lo que permite al motor procesar los cálculos continuos de la simulación sin cargar con la complejidad y sobrecarga de un sistema reactivo de interfaz.

Para comunicar el resultado de la simulación al jugador, el sistema *ECS* de sincronización interviene al término de cada ciclo lógico. Su labor consiste en extraer del motor únicamente la información estrictamente necesaria para la representación gráfica y volcarla en el estado caliente, gestionado por *Zustand*. A su vez, este estado global expone funciones selectoras granulares que permiten a cada elemento de la interfaz gráfica y a cada malla de la escena tridimensional suscribirse de forma exclusiva a los datos que le conciernen.

Esta estrategia dual justifica la presencia de un estado global. Por un lado, habilita a las capas de presentación el uso de las utilidades propias del ecosistema de *React*. Por otro, exime al motor lógico de acoplarse a las herramientas específicas de la interfaz gráfica, limitando el uso de la reactividad a los dominios donde realmente aporta valor estructural.

4.4.3. Escena 3D

La representación espacial del entorno y de los actores del juego se desarrolla íntegramente mediante la biblioteca *React Three Fiber*. Esta tecnología actúa como un puente que traslada el paradigma declarativo de componentes al lienzo gráfico, abstrayendo la complejidad intrínseca de WebGL. Al tratar cada malla tridimensional, fuente de iluminación y cámara virtual como un componente *React* se logra una integración nativa y coherente con el resto de la interfaz gráfica, facilitando su composición y mantenimiento.

En este diseño, la escena actúa como un consumidor puro del estado global de la simulación. Gracias a la división establecida por el estado caliente, los componentes gráficos se suscriben de forma selectiva a las variables que determinan su aspecto. Esta conexión directa permite que las posiciones, animaciones y estados de los actores se reflejen en la pantalla instantáneamente

cuando el motor lógico emite una actualización, sin necesidad de orquestar manualmente el redibujado de la escena completa.

La escala espacial generada por el algoritmo de construcción de mazmorras implica la existencia de miles de elementos simultáneos. Para evitar la saturación del procesamiento gráfico, el renderizado del laberinto recurre al uso de mallas instanciadas. Esta técnica reduce drásticamente las llamadas de dibujado de la tarjeta gráfica al reutilizar la misma geometría para múltiples instancias, técnica que se aplica tanto a elementos estáticos como los muros, como a grupos dinámicos como los coleccionables o los propios ecos.

Finalmente, el uso de este marco de trabajo permite delegar la responsabilidad de la suavidad visual fuera del estricto ciclo de actualización del motor lógico. Utilizando las herramientas de bucle de fotogramas que proporciona *React Three Fiber*, se calculan interpolaciones puramente visuales, como el seguimiento continuo de la cámara, las rotaciones de los modelos o las transiciones de escala de los enemigos. De esta forma, los efectos visuales enriquecen la experiencia del usuario sin comprometer ni mezclar su lógica con el determinismo matemático que rige el transcurso de la partida.

4.4.4. Interfaces Gráficas

La capa de presentación bidimensional de la aplicación, que abarca desde la página de aterrizaje hasta los menús interactivos y la visualización de información en pantalla, se ha desarrollado utilizando tecnologías web consolidadas. La estructuración de estas vistas se apoya en *React*, cuya arquitectura basada en componentes permite construir elementos modulares y reutilizables, separando adecuadamente la navegación de la aplicación de la propia experiencia de juego. Para la estilización visual se ha adoptado *Tailwind CSS*, un marco de trabajo basado en clases de utilidad que integra el diseño directamente en la jerarquía de cada componente. Esta combinación técnica asegura una estricta coherencia estética a lo largo de todas las pantallas, facilita la iteración del diseño y previene los problemas de mantenibilidad típicos de las hojas de estilo globales, resultando en un sistema visual escalable e independiente del motor lógico.

Jerarquía de Componentes

La arquitectura de componentes de interfaz se organiza en una jerarquía de cuatro niveles inspirada en la metodología de diseño atómico y los principios SOLID. El primer nivel comprende componentes atómicos sin dependencias internas como botones, entradas de texto o iconos, que no acceden al estado global y operan exclusivamente mediante propiedades. El segundo nivel agrupa componentes compuestos que combinan entre uno y tres componentes atómicos para encapsular patrones reutilizables como modales o grupos de botones. El tercer nivel contiene componentes de distribución espacial que integran múltiples almacenes de estado y organizan secciones completas de la interfaz. El cuarto nivel orquesta vistas completas de aplicación que coordinan todos los niveles inferiores. Esta jerarquía impone un flujo de dependencias unidireccional estricto; los niveles superiores pueden importar componentes de niveles inferiores, pero la relación inversa está prohibida, garantizando modularidad y facilitando el mantenimiento mediante responsabilidad única por componente.

Página de Aterrizaje

La ruta principal de la aplicación web se ha reservado para alojar una página de presentación del proyecto. Este espacio tiene un carácter marcadamente informativo y promocional, destinado a ofrecer un contexto general sobre el desarrollo, las tecnologías empleadas y el propósito del videojuego antes de que el usuario inicie la partida.

Menús

La coordinación entre los distintos estados globales de la aplicación y el flujo de la partida se manipula mediante sistemas de menús interactivos, compuestos principalmente por agrupaciones de botones. Esta capa abarca el menú principal de acceso al juego, el menú de pausa durante la partida, así como las pantallas de transición tras superar un nivel o perder todas las vidas. La implementación de estas pantallas recurre intensivamente a la arquitectura de componentes de *React*, reutilizando elementos atómicos y estructuras de agrupación. Esta estrategia permite aplicar el principio de programación *Don't Repeat Yourself*), reduciendo la duplicidad de código y garantizando un estilo visual completamente homogéneo en todas las interacciones del usuario fuera del tiempo de juego activo.

Pantalla de Visualización Frontal

Durante el transcurso de la simulación, es imprescindible proporcionar al jugador información continua sobre el estado frío de la partida. La Pantalla de Visualización Frontal se encarga de representar de forma clara estas variables críticas, tales como el medallón actualmente seleccionado, el nivel de carga de la habilidad de impulso *dash*), la cantidad de vidas restantes, el inventario de esferas de ruido blanco, la puntuación acumulada y el nivel en curso.

Capítulo 5

Planificación temporal

El desarrollo de *Chomp Crawler* se ha extendido a lo largo de dos años, desde julio de 2024 hasta junio de 2026, articulándose en torno a una metodología de trabajo iterativa e incremental. Lejos de seguir un calendario rígido y continuo, el proyecto se organizó en cuatro sprints, cada uno orientado a un objetivo tangible que sentaba las bases del siguiente. Esta segmentación permitió afrontar la considerable complejidad del trabajo de forma escalonada, validando cada hito antes de aumentar el alcance.

Conviene señalar que la dedicación no fue homogénea. La compatibilización del proyecto con la actividad académica y profesional impuso un ritmo variable, salpicado de periodos de abandono, especialmente acusados en las fases iniciales. El cómputo temporal total, por tanto, no refleja una ocupación constante, sino la maduración progresiva de un sistema que requirió tanto trabajo activo como tiempo de reflexión entre etapas. La figura 5.1 sintetiza este recorrido.

5.1. Sprint 1: Investigación y generación procedural

El primer sprint, comprendido entre julio de 2024 y mayo de 2025, se consagró por completo a la pieza algorítmica fundamental del proyecto: la generación procedural de laberintos. Durante esta fase se llevó a cabo una investigación en profundidad sobre los distintos algoritmos disponibles, culminando en la selección del algoritmo de Shaun LeBron como base estructural. Una vez seleccionado, se estudió a fondo su funcionamiento y se reimplementó por completo en Python, adaptándolo a las necesidades específicas del juego. La dedicación durante esta etapa se limitó a los fines de semana y estuvo marcada por grandes periodos de abandono.

5.2. Sprint 2: Prototipado y viabilidad

El segundo sprint se desarrolló entre mayo y septiembre de 2025. Su propósito fue demostrar la viabilidad práctica de la generación de niveles a la escala propia de *Pac-Man*, materializándola en varios prototipos sucesivos. Como primer acercamiento a la jugabilidad, se incorporó un fantasma básico capaz de perseguir al protagonista. Si bien la dedicación continuó siendo discontinua, en esta etapa adquirió ya una cadencia diaria, lo que aceleró notablemente el avance respecto al sprint anterior.

5.3. Sprint 3: Arquitectura y flujo de juego

El tercer sprint, prolongado hasta diciembre de 2025, abordó la consolidación arquitectónica del proyecto. En él se implementó la arquitectura *Entity Component System* y se construyó el flujo completo del juego, dotando al prototipo de un ciclo de partida coherente y de un núcleo lógico sólido y mantenible. La dedicación se mantuvo en la línea diaria pero discontinua del sprint precedente.

5.4. Sprint 4: Identidad propia y pulido

El cuarto y último sprint, desarrollado entre marzo y junio de 2026, transformó el prototipo funcional en el producto final. Se migró la aplicación al marco de trabajo Next.js y se definieron sus componentes conforme a un estilo visual propio de color y tipografía. Se generaron mapas de gran tamaño, se implementaron los nuevos sistemas jugables y se modelaron las nuevas mallas (*meshes*), forjando una identidad estética y mecánica diferenciada respecto a *Pac-Man*. El ritmo de trabajo fue análogo al de las etapas anteriores.

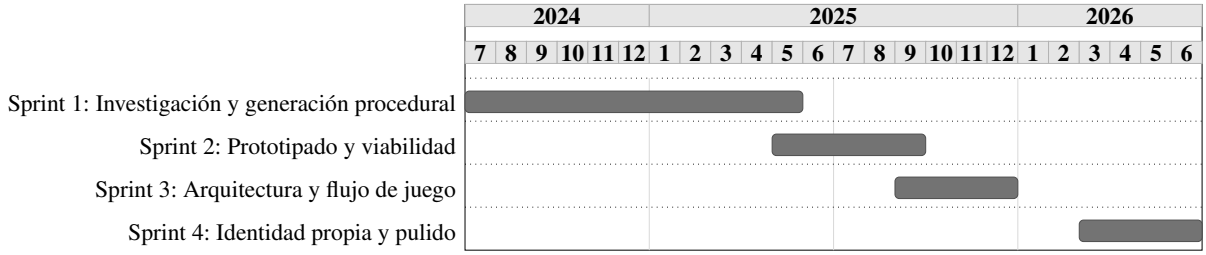


Figura 5.1: Cronograma del desarrollo de *Chomp Crawler*, organizado en cuatro sprints entre julio de 2024 y junio de 2026. El eje horizontal representa los meses del calendario.

Capítulo 6

Experimentos y validación

El proceso de desarrollo de este proyecto se ha apoyado en pruebas de rendimiento y sesiones de validación empírica para guiar la toma de decisiones técnicas y de diseño. Dado el enfoque iterativo del trabajo, estos experimentos se han centrado en resolver los cuellos de botella surgidos durante la implementación y en ajustar la experiencia de juego mediante las observaciones de jugadores externos.

6.1. Experimentos de rendimiento

6.1.1. Optimización de la generación procedural

La implementación inicial del algoritmo de generación de laberintos presentó problemas de rendimiento inasumibles, demorando varios minutos en la creación de un único nivel. Inicialmente, se experimentó modificando el número máximo de reintentos permitidos para la generación de cada sala de la mazmorra. Sin embargo, limitar los reintentos provocaba el descarte de la mazmorra completa y forzaba un reinicio desde el principio en cuanto el algoritmo se atascaba en una configuración compleja, por lo que el problema de los tiempos de carga persistía.

Para abordar este cuello de botella, se implementó un banco de pruebas en lenguaje Python que ejecutaba el algoritmo iterativamente sobre cientos de muestras, registrando el motivo de reinicio a través de los mensajes de *log* del programa para cada iteración y generando un reporte que permitió estudiar la tendencia. El posterior análisis de estos datos reveló que aproximadamente el 59 % de los descartes se debían a dos comprobaciones estéticas heredadas del

algoritmo original de *Pac-Man*: el rechazo de configuraciones en la esquina superior derecha (responsable del 33 % de los descartes) y en la esquina inferior derecha (26 %). Como se ha detallado en el capítulo de diseño, se concluyó que mantener esta fidelidad geométrica no compensaba el alto coste computacional. La relajación de estas restricciones redujo los tiempos de generación de minutos a apenas unos pocos segundos, garantizando una inicialización ágil para el jugador.

6.1.2. Optimización del renderizado tridimensional

Un segundo experimento técnico se llevó a cabo tras integrar el generador procedural en el cliente web y comenzar a proyectar mazmorras gigantes. Durante la exploración, el navegador experimentaba caídas severas en la tasa de fotogramas, afectando directamente a la jugabilidad.

Para diagnosticar el problema, se emplearon las herramientas de perfilado de rendimiento integradas en el navegador web. El análisis del flujo de ejecución demostró que el cuello de botella residía en el excesivo número de llamadas de actualización y dibujado que el procesador enviaba a la tarjeta gráfica, originadas por la instanciación individual de cada baldosa, bloque de pared y entidad recolectable de la escena. Este diagnóstico empírico forzó un cambio arquitectónico en la representación 3D hacia el uso masivo de mallas instanciadas. Esta técnica mitigó el problema al colapsar miles de operaciones gráficas en una única llamada, estabilizando el rendimiento sin importar la extensión de la mazmorra.

6.2. Validación empírica con usuarios

6.2.1. Diseño del experimento

Se realizaron cinco sesiones individuales de prueba con usuarios externos al proyecto, reclutados entre familiares y amigos. Cada sesión fue independiente y presencial: el usuario jugaba solo, sin observar la experiencia de otros participantes ni recibir instrucciones previas más allá de los controles básicos. El perfil de los participantes era variado: la mayoría tenía poca o ninguna experiencia con videojuegos, mientras que dos de los cinco eran jugadores habituales.

El objetivo no era obtener métricas cuantitativas, sino identificar puntos de fricción reales en la experiencia —confusión, frustración, comportamientos no anticipados— y observar hasta

qué punto las mecánicas diseñadas resultaban comprensibles sin mediación del autor.

6.2.2. Observaciones y correcciones

A partir de las sesiones se detectaron dos problemas con impacto directo en el diseño, que derivaron en cambios concretos:

- **Claridad del objetivo y navegación.** Tres de los cinco participantes se perdieron durante sus primeras partidas, sin comprender de forma intuitiva cuál era el objetivo de la mazmorra. La tendencia observada era intentar limpiar el laberinto de todas las bolas de esencia —como en el *Pac-Man* clásico— en lugar de buscar la salida del nivel. Uno de estos tres usuarios era jugador habitual, lo que descarta que la confusión se debiese únicamente a la falta de experiencia con videojuegos.
- **Balanceo y curva de dificultad.** Uno de los jugadores habituales llegó al nivel catorce en su primera partida sin encontrar resistencia significativa por parte de la simulación. Este resultado evidenció un déficit importante en la escalabilidad de la dificultad: el sistema no penalizaba suficientemente al jugador experimentado. Para corregirlo, se introdujo un sistema de escalado que incrementa ligeramente la velocidad base de los ecos al superar cada nivel, garantizando un desafío sostenido a largo plazo.

En conjunto, los cinco usuarios completaron al menos un nivel y expresaron una valoración positiva de la experiencia, lo que confirmó que las mecánicas principales funcionaban de forma comprensible una vez superado el obstáculo inicial de orientación.

Capítulo 7

Conclusiones

El resultado final de este Trabajo Fin de Grado es *Chomp Crawler*, un videojuego web completamente funcional que moderniza las mecánicas clásicas de *Pac-Man* introduciendo elementos del género *rogue-like*. Se ha logrado materializar una experiencia interactiva fluida y desafiante, accesible directamente desde navegadores web modernos sin necesidad de instalaciones previas en `https://chompcrawler.com`. El proyecto culmina en una aplicación cohesionada que integra con éxito tecnologías diversas y avanzadas, desde la ejecución de algoritmos de generación procedural en Python mediante WebAssembly, hasta el renderizado tridimensional interactivo y la gestión reactiva de interfaces de usuario.

A nivel de ingeniería, el logro más significativo reside en la implementación e integración profunda del sistema en su conjunto, construido desde sus cimientos. Más allá de la utilización de herramientas o arquitecturas concretas, el verdadero valor del proyecto radica en como se aborda a bajo nivel tanto el núcleo lógico de la simulación como el intrincado proceso de generación procedural de laberintos. Esta aproximación ha supuesto enfrentarse de manera directa a complejos desafíos teóricos y arquitectónicos —tales como la manipulación de grafos, la garantía algorítmica de conectividad espacial y el control determinista del estado— exigiendo un riguroso ejercicio de ingeniería de software. El resultado es un producto cohesionado que cristaliza la aplicación práctica de conceptos computacionales fundamentales para crear y sostener una experiencia interactiva compleja.

7.1. Consecución de objetivos

La evaluación de los objetivos planteados al inicio de este Trabajo Fin de Grado arroja un balance muy positivo, aunque con matices enriquecedores fruto de la experiencia del propio desarrollo. A continuación se revisan el objetivo principal, los objetivos instrumentales y las restricciones formulados en la sección 1.1.

El **objetivo principal** —diseñar e implementar un videojuego completo, funcional y rejugable, inspirado en *Pac-Man*, con niveles que sean laberintos genuinos generados proceduralmente— se ha alcanzado con creces. El producto final es una aplicación robusta y plenamente jugable en la que cada partida transcurre en un laberinto único, con entrada y salida definidas y completamente navegable. Gracias a la solidez de los algoritmos de generación procedural y al comportamiento emergente del sistema lógico, el diseño aleatorio sostiene la jugabilidad a largo plazo, que era la apuesta central del proyecto.

Los **objetivos instrumentales** se han satisfecho en su totalidad, si bien con un esfuerzo desigual:

- El **motor lógico** se ha implementado íntegramente desde cero —movimiento, colisiones, comportamiento de enemigos y ciclo de vida de la partida—, constituyendo el núcleo de ingeniería más exigente del trabajo y la pieza sobre la que descansa el resto del sistema.

- El **diseño de mecánicas y apartado visual** adaptados al contexto tridimensional y explorable se ha completado, aunque fue precisamente aquí donde la ambición inicial obligó a descartar numerosas ideas y a acotar el alcance para preservar la viabilidad del proyecto.

- El **generador procedural** produce en cada partida un nivel único con garantía algorítmica de conectividad y navegabilidad completa, cumpliendo su cometido de forma rigurosa y verificada empíricamente.

En cuanto a las **restricciones** —ejecución íntegra en cliente sin componente de servidor, *stack* de *frontend* basado en React y Next.js, representación tridimensional sobre WebGL mediante React Three Fiber y generación de niveles en Python ejecutada en el navegador a través de Pyodide y WebAssembly—, se han respetado todas sin excepción. Lejos de ser meras limitaciones, estas condiciones forman parte sustancial de la propuesta y han dirigido buena parte

de las decisiones de arquitectura. La conclusión extraída es que la tecnología web resulta apasionante y perfectamente capaz de sostener el desarrollo de videojuegos; la principal autocrítica es no haber explorado las capacidades de red o multijugador inherentes al navegador, una línea que se retoma en la sección de trabajos futuros.

Más allá del cumplimiento estricto, el desarrollo ha dejado una lección de incalculable valor: la importancia de comedir las expectativas, acotar el alcance y asumir la inherente dificultad de la planificación temporal en el desarrollo de software, asunto sobre el que se profundiza en las secciones siguientes.

7.2. Aplicación de lo aprendido

Este trabajo se erige como una síntesis transversal de las competencias adquiridas durante el Grado en Ingeniería en Sistemas Audiovisuales y Multimedia, aplicando de forma directa los conocimientos de múltiples asignaturas:

- **Tecnologías Web y Arquitectura de Aplicaciones:** El uso intensivo de JavaScript moderno y su estructuración gráfica consolidan lo aprendido en *Construcción de Servicios y Aplicaciones Audiovisuales en Internet*. La adopción de librerías como React y Tailwind CSS supone una expansión natural de este temario. Asimismo, el despliegue del proyecto sobre el marco de trabajo Next.js representa un avance cualitativo respecto a los conceptos abordados en *Laboratorio de Tecnologías Audiovisuales en la Web*, yendo un paso más allá del desarrollo tradicional de interfaces en cliente (*frontend*) y servidores (*backend*) aislados.

- **Renderizado y Gráficos Tridimensionales:** La implementación visual mediante React Three Fiber guarda una relación directa y profunda con la asignatura *Gráficos 3D y Visualización*. De hecho, la representación espacial de este proyecto nace como una expansión natural de la práctica final de dicha materia, demostrando la aplicabilidad de la programación gráfica en la web.

- **Algoritmia y Matemáticas Computacionales:** La integración del algoritmo de Kruskal para la generación estructural de las mazmorras continúa y amplía el temario de *Gestión y Optimización de Recursos* —donde se estudian estrategias análogas como Prim o

Dijkstra—, y materializa los fundamentos de la teoría de grafos introducida en *Matemáticas II*.

- **Fundamentos Transversales:** Por su propia naturaleza, la conceptualización y desarrollo de una aplicación web de esta envergadura exige movilizar conocimientos fundacionales sobre redes de comunicaciones, programación orientada a objetos y estructuras de datos.

Más allá del despliegue de tecnologías concretas, el Grado ha inculcado una filosofía de trabajo de incalculable valor. La ingeniería es, en su raíz, el uso del ingenio para la resolución sistemática de problemas. Llevar a término este proyecto ha requerido pensar de forma analítica, cuestionar suposiciones, medir el rendimiento empíricamente y, en definitiva, aplicar un criterio técnico fundamentado en cada fase de la toma de decisiones.

7.3. Lecciones aprendidas

A lo largo del desarrollo de este Trabajo Fin de Grado se han adquirido competencias que exceden el temario reglado del Grado. Si la sección anterior recoge cómo el proyecto ha servido para aplicar lo aprendido en las aulas, esta pone en valor el conjunto de conocimientos abordados de forma autónoma —por necesidad del propio proyecto— y que constituyen, en su mayoría, bloques formativos completos no cubiertos por la carrera:

1. **Desarrollo *frontend* moderno con el ecosistema *React*:** El Grado introduce el desarrollo web tradicional, pero no el paradigma declarativo ni la arquitectura basada en componentes que hoy vertebran la industria. Su estudio y aplicación práctica ha supuesto incorporar un bloque de conocimiento autónomo y plenamente vigente en el mercado laboral, comprobando de primera mano las virtudes de construir interfaces modulares y reactivas.

2. **Arquitectura *Entity Component System*:** La adopción del patrón *ECS*, propio del desarrollo de videojuegos y ajeno al plan de estudios, ha exigido comprender sus mecánicas internas y razonar sobre la separación entre datos y comportamiento. Constituye un modelo arquitectónico de pleno valor profesional, asimilado e implementado desde cero.

3. **Tipado estático y análisis de código legado:** El estudio de código fuente de terceros —escrito en versiones antiguas de JavaScript y carente de comentarios— ha sido un ejer-

cicio de ingeniería inversa tan instructivo como exigente. La experiencia ha reafirmado el valor del código limpio y motivado la adopción de *TypeScript*, asumiendo tanto sus beneficios preventivos como las fricciones que el tipado estático impone al desarrollo.

4. **Diseño visual y dirección de arte:** Asumir responsabilidades propias del diseño gráfico, materia inexistente en una titulación de corte ingenieril, ha permitido adquirir competencias fundamentales en gestión de paletas de color, contraste y tipografía, así como dimensionar la complejidad real de un rol que condiciona directamente la experiencia del usuario.

5. **Integración prudente de la Inteligencia Artificial:** Trabajar con herramientas de generación de código asistida por IA ha resultado ser una disciplina en sí misma, mucho más exigente de lo que su aparente comodidad sugiere. La lección esencial es que la IA tiende a avanzar más rápido de lo que el desarrollador puede comprender y validar: aprobar cambios atómicos, uno a uno, transmite una sensación de control que es en parte ilusoria, pues el entendimiento global del sistema solo se preserva navegando y releyendo el código en su contexto completo, con una actitud crítica y escéptica que discrimina activamente qué *no* debe incorporarse. Es un ejercicio agotador que demanda disciplina sostenida: tras varias horas de revisión aparece la tentación de aceptar sugerencias sin el debido escrutinio, momento en el que conviene alternar la asistencia con fases de investigación y de diseño sobre el papel. En este equilibrio ha resultado especialmente valioso reservar la limpieza y reestructuración final del código a un trabajo manual y deliberado, donde el dominio del editor permite recuperar por completo la autoría y la comprensión del resultado.

6. **Gestión de contextos tecnológicos concurrentes:** Compaginar este desarrollo con una beca en paralelo ha evidenciado la dificultad intrínseca de operar simultáneamente sobre dos proyectos con pilas tecnológicas (*stacks*) divergentes, lo que exige un gran esfuerzo de adaptación y penaliza los cambios de contexto.

7. **Sostenibilidad y planificación del trabajo:** En el ámbito de la gestión de proyectos se ha constatado la importancia de fijar objetivos realistas y mantener el orden. La exigencia desmedida y la privación de descanso para solventar imprevistos resultan contraproducentes a medio y largo plazo; la sostenibilidad del ritmo de trabajo es prioritaria frente a

la sobrecarga táctica para asegurar la calidad y viabilidad de cualquier proyecto.

7.4. Trabajos futuros

A pesar de haber alcanzado los objetivos planteados y de obtener una aplicación completamente funcional, la propia naturaleza modular y escalable del proyecto abre un amplio abanico de posibilidades para su futura expansión. Algunas de las funcionalidades y mejoras que enriquecerían significativamente la experiencia incluyen:

- **Funcionalidades en línea y persistencia:** Implementación de un sistema de autenticación de usuarios que permita el guardado en la nube y el establecimiento de clasificaciones (*rankings*) globales, fomentando el aspecto competitivo.
- **Multijugador cooperativo:** Explorar las capacidades de red de los navegadores modernos integrando un modo cooperativo simultáneo para 2 a 4 jugadores, fundamentado en conexiones entre pares (P2P) mediante tecnología WebRTC.
- **Expansión de mecánicas de juego:** Ampliación del elenco de enemigos con nuevos tipos de ecos que presenten patrones de comportamiento únicos. Además, se contempla el desarrollo de batallas contra jefes finales y la inclusión de diferentes finales alternativos para la aventura.
- **Nuevos elementos y eventos espaciales:** Integración de un sistema de objetos ofensivos especiales que otorguen al jugador nuevas herramientas de interacción agresiva frente a los enemigos, inspirándose en iteraciones modernas de la franquicia original como *Pac-Man 256*. Asimismo, la generación procedural podría sofisticarse para incluir salas singulares, con eventos o misiones especiales transitorias.
- **Personalización estética:** Incorporación de un sistema de progresión que recompense al jugador con elementos cosméticos desbloqueables, como sombreros o variaciones visuales para el protagonista, permitiendo personalizar la experiencia visual sin alterar las mecánicas subyacentes.

Bibliografía

[1] Jamey Pittman. The pac-man dossier, 2011. Análisis técnico exhaustivo sobre la lógica de programación, inteligencia artificial y mecánicas internas del arcade original de Namco. URL: `https://pacman.holenet.info`.

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

[3] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 1956.

[4] Robert 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.

[5] Teemu 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. URL: `https://trepo.tuni.fi/handle/123456789/27593`.

[6] Steve Faulkner, Arron Eicholz, Travis Leithead, Alex Danilo, and Sangwhan Moon. Html5.2 recommendation. W3c recommendation, World Wide Web Consortium (W3C), 2017. Estándar oficial que define las APIs de conectividad y multimedia. URL: `https://www.w3.org/TR/html52/`.

[7] Mozilla Developer Network. Introduction to web apis, 2024. Documentación técnica sobre la implementación de WebGL y WebRTC. URL: `https://developer.mozilla.org/en-US/docs/Web/API`.

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

[9] Dean Jackson and Jeff Gilbert. Webgl specification 1.0. Standard specification, The Khronos Group, 2011. Estándar base que define la interfaz entre JavaScript y OpenGL ES. URL: `https://www.khronos.org/registry/webgl/specs/1.0/`.

[10] 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. URL: `https://developer.mozilla.org/es/docs/Web/API/WebGL_API`.

[11] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF 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.

[12] Andreas Rossberg, Ben L Titzer, Andreas Haas, Derek 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.

[13] Ecmascript 2023 language specification, 2023. Especificación técnica base sobre la que se construye el ecosistema JavaScript moderno. URL: `https://www.ecma-international.org/publications-and-standards/standards/ecma-262/`.

[14] Axel Rauschmayer. *Exploring ES6: Upgrade to the next version of JavaScript*. Leanpub, 2014. Análisis profundo de las características modulares introducidas en ECMAScript 6. URL: `http://exploringjs.com/es6/`.

[15] Boris 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.

[16] Luciano 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.

[17] Wes 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.

[18] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David 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. doi:10.1038/s41586-020-2649-2.

[19] Three.js Authors. *Three.js Documentation*. Three.js Project, 2024. Documentación oficial de la biblioteca de abstracción para renderizado WebGL. URL: https://threejs.org/docs/.

[20] 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. URL: https://react.dev/.

[21] Alex Banks and Eve 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.

[22] Vercel Inc. *Next.js Documentation*, 2024. URL: https://nextjs.org/docs.

[23] 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. URL: https://zustand.docs.pmnd.rs/getting-started/introduction.

[24] Tailwind Labs Inc. *Tailwind CSS Documentation*, 2024. URL: https://tailwindcss.com/docs.

[25] Poimandres Group. *React Three Fiber Documentation*. Poimandres Open Source Collective, 2024. Especificación técnica del renderizador y guía de integración con el ecosistema Three.js. URL: `https://docs.pmndrs.assets/react-three-fiber/getting-started/introduction`.

[26] 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. URL: `https://react.dev/reference/react-dom/server`.

[27] 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. URL: `https://pyodide.org/en/stable/`.

[28] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, MA, 2nd edition, 1994. Manual de referencia del sistema de composición tipográfica empleado en la redacción de la memoria.

[29] Till Tantau. *The TikZ and PGF Packages: Manual for Version 3.1.10*. PGF/TikZ Project, 2023. Documentación oficial del paquete de dibujo vectorial programático para LaTeX. URL: `https://github.com/pgf-tikz/pgf`.

[30] Neovim Contributors. *Neovim Documentation*. Neovim Project, 2024. Documentación oficial del editor de texto extensible basado en terminal. URL: `https://neovim.io/doc/`.

[31] Docker Inc. *Docker Documentation*. Docker Inc., 2024. Documentación oficial de la plataforma de contenedores empleada en el despliegue. URL: `https://docs.docker.com/`.

[32] Nginx Inc. *nginx Documentation*. F5, Inc., 2024. Documentación oficial del servidor web y proxy inverso empleado en la infraestructura. URL: `https://nginx.org/en/docs/`.