CPES SEIM BHIP

PROYECTO

Tetraversus

Autor:
Xabier GARROTE

Tutor:
Daniel Miguel BABON

Presentación del proyecto para obtener
el título de FP de Grado Superior en Desarrollo de Aplicaciones Multiplataforma

20 de febrero de 2026

«*Subjectivity is tempered by the realization that any work is good if it is accepted by its audience. This means that we aren't dealing with individual preferences and prejudices, but rather, with the desires of a collective audience.*»

Robert C. Martin [Mar25]

CPES SEIM BHIP

Resumen

FP de Grado Superior en Desarrollo de Aplicaciones Multiplataforma

Tetraversus

por Xabier GARROTE

Este proyecto tiene como objetivo desarrollar un videojuego inspirado en el clásico Tetris, respetando sus reglas originales e incorporando funcionalidades actuales. Se implementa una interfaz gráfica intuitiva que favorece una interacción fluida, así como un modo multijugador en línea que permite la competición entre jugadores en tiempo real. El juego se desarrolla con el motor Unity y el lenguaje C#, poniendo el foco en una arquitectura modular y en buenas prácticas de programación. El resultado es una reinterpretación del juego tradicional que combina fidelidad mecánica, juego competitivo y una experiencia de usuario actualizada.

Agradecimientos

A mi aita, por introducirme en el mundo de la informática cuando apenas tenía cinco años.

A mi ama, por su apoyo incondicional, por su paciencia y por ser mi mayor fan.

A mi tutor Dani, por darme la oportunidad de desarrollar este proyecto, por su confianza y por acompañarme durante todo el proceso.

A Robert C. Martin, por su inspiración constante y por ser un referente en el ámbito del desarrollo de software.

A Marian Rojas Estapé, por su obra y por animarme a entrevistar a Robert C. Martin.

A mis amigos de las partidas de cartas de los viernes, por las risas, los buenos momentos y por ser un apoyo imprescindible.

Índice general

Dedicado a mi aita, te quiero...

Preámbulo

Voy a empezar con una historia que me ha marcado mucho porque me apasiona el deporte y su capacidad de sufrimiento así como de hacer brillar el talento y creo que es muy ilustrativa.

Si hay una prueba mítica en atletismo, considerada la prueba reina, son los 100 metros lisos. Durante muchos décadas, se pensaba que era imposible que el ser humano bajase de los 10 segundos. 10,05; 10,03; 10,02... pero nunca por debajo de 10. Muchos médicos afirmaron que el ser humano no estaba diseñado para bajar de los 10 segundos en los 100 metros lisos.

Parecía que esto se había convertido en una verdad absoluta hasta que, en el verano tórrido de de los Juegos Olímpicos de México de 1968, se presenta un atleta llamado James Hines. Suena el disparo, sale como una bala y marca 9,95. Esto es lo que hace el libro *Clean Code*[Mar25]:**demostrar que se pueden hacer mejor las cosas y romper con lo que hasta entonces estaba establecido.**

Después del logro de James Hines pasó algo increíble: unos cuantos atletas más bajaron de los 10 segundos. Creyeron que era posible porque lo habían visto. Lo mismo pasó en informática: aparecieron *Clean Code with C#* y *Clean Code in Python*...

Y cuando ves a alguien que ha hecho algo y te inspira, el cerebro empieza a trabajar a tu favor.

No pude leer *Clean Code* hasta la edad adulta ya que aprendí inglés muy tarde, con 22 años. Cuando lo leí, me encontré con un libro con mucho conocimiento, fácil de entender y con consejos muy prácticos. Además, algo impresionante ocurrió cuando probé a llevar sus consejos a la práctica: funcionaban. ¡Sí, funcionaban! Mi código era más sencillo, más práctico y con menos errores.

El 30 de septiembre de este año Robert C. Martin publico la segunda edición del libro *Clean Code*. He tenido la ocasión de leerlo y bate los diez segundos, y también a todos los que están por debajo de los diez segundos. Robert C. Martin se ha colocado con el mejor tiempo de toda la historia en el mundo de la informática. Es el Usain Bolt de la informática. Ahora que lo pienso, todavía tiene más mérito con 74 años, cuatro años más que Bill Gates.

Mi tutor Dani me dio a conocer el libro *Clean Code* y la obra *Cómo hacer que te pasen cosas buenas*, de Marian Rojas Estapé, me animó a entrevistar a su autor, Robert C. Martin.

En este proyecto final de grado encontraréis todo el conocimiento de *Clean Code* volcado, junto con ideas de otros libros que también marcan muy buenos tiempos, como *The Pragmatic Programmer* [HT19] o *Design Patterns* [Gam+94]. y espero que lo disfrutéis tanto como lo he disfrutado yo escribiéndolo.

Capítulo 1

Planificación del proyecto

1.1. Introducción

1.1.1. Contexto y motivación

En los últimos años, los videojuegos han dejado de ser un simple pasatiempo para consolidarse como un punto de encuentro entre arte y tecnología y, en muchos casos, también como una forma de comunicación. Jugar ya no consiste únicamente en superar niveles: puede implicar habitar una historia, explorar una idea o, sencillamente, desconectar durante un rato en otro mundo.

Este proyecto nace de la motivación por comprender, de manera práctica y completa, cómo se construye un videojuego desde cero: concebir, diseñar y, finalmente, programar. Además, se trata del primer videojuego que desarrollo, lo que convierte el proceso en una experiencia especialmente significativa.

El recorrido ha estado marcado por pruebas, errores, pequeños avances y un aprendizaje constante. Como dijo Antonio Machado: «*Caminante, no hay camino, se hace camino al andar*». **En mi caso, ese camino ha sido aprender a dar vida a una idea, convertir líneas de código en un producto jugable y entender todo lo que existe detrás de un juego, más allá de lo que se ve en pantalla.**

1.2. Objetivos

En este apartado se recogen los objetivos específicos del proyecto junto con sus criterios de aceptación. Estos objetivos se han definido para estructurar el desarrollo en hitos funcionales, de forma que cada bloque aporte valor verificable al producto final.

1.2.1. Elegir lenguaje y motor

- Elaborar una *shortlist* (2–3 opciones) con pros y contras.
- Evaluar: Godot (GDScript/C#), Unity (C#), Phaser (JS/TS). Alternativas: MonoGame (C#) y LÖVE (Lua).
- Comparar criterios: soporte 2D (*tilemaps*, físicas simples, *input*), rendimiento, exportación (PC/Web), tamaño de *build*.
- Analizar licencia/costes, curva de aprendizaje, *tooling* (animación, audio, depuración) y comunidad/recursos en español.

- **Aceptación:** decisión final justificada con base en los criterios anteriores.

1.2.2. Tablero y sistema de piezas

- Implementar una rejilla visible de 10×20 con un *buffer* superior de 2–4 filas.
- Implementar las piezas I, O, T, S, Z, J y L.
- Diseñar e incluir 1–2 piezas especiales del modo alternativo (por ejemplo, bloque bomba o pieza fantasma).
- Implementar un generador aleatorio tipo *bolsa* de 7 piezas.
- **Aceptación:** no hay solapes ni desbordes; la pieza aparece centrada y cae correctamente.

1.2.3. Movimiento, rotación y colisiones

- Soportar movimiento izquierda/derecha, rotación horaria y antihoraria, *soft drop* y *hard drop*.
- Incorporar un *lock delay* configurable (p. ej., 500 ms) y *wall kicks* simplificados (SRS básico).
- **Aceptación:** las pruebas de colisión (paredes/piezas/suelo) pasan en todos los tetrominós; los *wall kicks* funcionan en esquinas.

1.2.4. Líneas, puntuación y niveles

- Implementar la detección de líneas completas y su eliminación.
- Diseñar un sistema de puntuación básico con progresión de niveles y aumento de velocidad.
- **Aceptación:** las líneas se detectan y eliminan correctamente; la puntuación y el nivel se actualizan.

1.2.5. Mecánicas de calidad de vida

- Implementar *hold* (guardar pieza) con un uso por caída y una cola de próximas piezas (*next*: 3–5).
- Implementar *ghost piece* (sombra de aterrizaje) con opción de activación/desactivación.
- **Aceptación:** la interfaz muestra *hold*/*next*; la *ghost piece* coincide con la posición final real.

1.2.6. Base de datos y servicios web

- Diseñar la base de datos (tablas, relaciones, restricciones e integridad).
- Implementar servicios web para registrar usuarios, autenticar y gestionar datos de juego.
- Diseñar la API siguiendo un estilo REST y documentar los *endpoints*.

1.2.7. Diseño final de la base de datos

- Verificar la coherencia del modelo relacional, la normalización y la integridad referencial.

1.2.8. Diseño final de la REST API

- Verificar la coherencia de los *endpoints*, la adecuación de los métodos HTTP y la claridad de las rutas.
- Verificar la consistencia de los esquemas de datos, los códigos de respuesta y la documentación.

Capítulo 2

Elegir lenguaje y motor

2.1. Introducción y criterios de evaluación

La elección del motor condiciona el rendimiento, el tamaño de las *builds*, la velocidad de desarrollo y la sostenibilidad del proyecto. En este documento se evalúan tres opciones principales para un juego 2D con *tilemaps*, físicas simples e *input*, con exportación a PC y Web: **Godot (GDScript/C#)**, **Unity (C#)** y **Phaser (JavaScript/TypeScript)**. Los criterios de comparación son:

- **Capacidades 2D**: *tilemaps*, físicas 2D, sistema de escenas, animación, *input*.
- **Rendimiento**: FPS, consumo de memoria, tiempos de carga.
- **Exportación y tamaño de *build***: PC (Windows/Linux/macOS) y Web (HTML5/WebGL/WASM), peso inicial y facilidad de automatización.
- **Licencia/costos**: modelo, riesgos y restricciones.
- **Curva de aprendizaje**: documentación, ergonomía del lenguaje, arquitectura.
- ***Tooling***: animación, audio, *profiling*, depuración.
- **Comunidad/recursos en español**: foros, tutoriales, repositorios y ejemplos.

2.2. Shortlist de motores

2.2.1. Godot (GDScript/C#)

Visión general. Motor *open source* (licencia MIT) con enfoque sólido en 2D, editor ligero y nodos dedicados (p. ej., `TileMap`, `AnimatedSprite2D`, `CollisionShape2D`). Lenguajes: *GDScript* y C#. Las guías y referencias oficiales de Godot cubren *tilemaps*, *tilesets* y clases relacionadas [God25e; God25f; God25c; God25d].

Pros.

- **2D nativo** muy completo: `TileMap`/`TileSet`, físicas 2D integradas, señales y sistema de escenas [God25e; God25c].
- **Iteración rápida** con GDScript (recarga ágil y sintaxis clara).
- **Exportación directa** a Web (HTML5/WASM) y a PC con plantillas oficiales [God25b].
- **Tamaño de *build*** contenido para 2D (dependiente de *assets* y configuración de exportación).

- **Licencia MIT**: cero *royalties* [God25h; God25a].

Contras.

- Ecosistema de extensiones menor que Unity.
- C# menos pulido que GDScript dentro del propio editor.
- En Web, tiempos de carga que pueden requerir compresión y *tuning* [God25b].

2.2.2. Unity (C#)

Visión general. Motor comercial maduro con editor completo, ecosistema masivo y herramientas profesionales para 2D: *Tilemap*, 2D Animation/Sprite Editor, Cinemachine 2D, *Input System* moderno y *Profiler* avanzado [Uni25a; Uni25e].

Pros.

- ***Tooling* todo en uno** muy robusto para 2D (*Tilemap*, 2D Animation, *Profiler*) [Uni25a].
- **Ecosistema** extenso (Asset Store) y documentación abundante.
- **Rendimiento** sólido en PC; opciones de optimización (Burst, Jobs/DOTS, URP 2D).
- **Multiplataforma**: exportación a muchas plataformas, WebGL incluida [Uni25a; Uni25d].
- **C#** y excelente integración con IDEs (*Rider*/VS) y pruebas. Cámaras avanzadas con *Cinemachine* [Uni25b; Uni25c] e *Input System* moderno [Uni25f].

Contras.

- **Tamaño de *build*** mayor (especialmente WebGL); requiere compresión (Brotli/Gzip) y configuración de servidor para descargas óptimas [Uni25d].
- **Curva del editor** más pronunciada por la amplitud de opciones.
- **Licencia/políticas** sujetas a cambio, a vigilar en hitos.

2.2.3. Phaser (JavaScript/TypeScript)

Visión general. *Framework* 2D *web-first*, ideal si el *target* principal es navegador. Integración natural con el ecosistema web (npm, *bundlers*, CI/CD). La documentación oficial cubre *tilemaps* y sistemas de físicas (Arcade y Matter) [Pha25d; Pha25a; Pha25b; Pha25c].

Pros.

- **Web nativo** y *builds* muy ligeras.
- **Iteración** rápida con *dev servers* y recarga.
- ***Tooling* web**: depuración con *DevTools*, despliegue simple (estáticos).

Contras.

- Menos ***tooling* visual** integrado (editor de escenas/animación).
- Físicas y *tilemaps* dependen de librerías/editores externos (p. ej., Tiled) [Pha25d; Pha25a; Pha25b].

- Exportación a PC vía *wrappers* (Electron/Tauri) que aumentan el peso.

2.3. Comparativa por criterios

- **Capacidades 2D.** Godot ofrece un *stack* 2D muy cohesionado [God25e; God25c]; Unity proporciona herramientas profesionales y extensibles [Uni25a; Uni25e]; Phaser cubre lo esencial con flexibilidad, apoyándose en Tiled y librerías [Pha25d; Pha25c].
- **Rendimiento.** En PC, Godot y Unity rinden muy bien en 2D. Unity dispone de rutas de optimización avanzadas si surge la necesidad [Uni25a]. En Web, Phaser y Godot son competitivos; Unity WebGL funciona pero tiende a mayor consumo (optimizable con compresión y ajustes) [Uni25d; God25b].
- **Exportación y tamaño de *build*.** Godot exporta a Web/PC con pesos contenidos [God25b]. Unity exporta a casi todo, con *builds* más pesadas, especialmente en WebGL [Uni25d]. Phaser es óptimo para Web; para PC necesita envoltorios que añaden MBs.
- **Licencia y costos.** Godot (MIT) no exige *royalties* [God25h; God25a]. Unity ofrece plan gratuito con condiciones que deben revisarse en hitos. Phaser es libre para la mayoría de usos; el coste radica en infraestructura y equipo [Pha25e].
- **Curva de aprendizaje.** Godot (GDScript) y Phaser (JS/TS) facilitan la entrada y la iteración. Unity exige familiarización con su editor, pero recompensa con *tooling* y ecosistema cuando el proyecto crece [Uni25a; Uni25e].
- ***Tooling*.** Unity lidera en *profiling*, *timeline*, cámaras (Cinemachine) y flujos complejos [Uni25b; Uni25a]. Godot tiene un conjunto suficiente para 2D [God25e]. Phaser se apoya en herramientas web y editores externos [Pha25c].
- **Comunidad en español.** Unity y Godot disponen de comunidades activas (docs extensas y foros); Phaser es más nicho (con recursos oficiales y MIT) [Uni25a; God25g; Pha25e].

2.4. Alternativas breves: MonoGame (C#) y LÖVE (Lua)

MonoGame (C#) es un *framework* sin editor: otorga control y rendimiento excelentes, pero exige más código y construir *tooling* propio. **LÖVE (Lua)** es ligero y ágil para 2D, con iteración muy rápida, aunque menos industrializado para exportación que Godot/Unity.

2.5. Aceptación

Decisión: Adoptar **Unity (C#)** como motor principal para el proyecto 2D con *tilemaps*, físicas simples e *input*, con exportación a **PC** y soporte **WebGL** cuando sea necesario.

Justificación.

1. ***Tooling* y flujo profesional "todo en uno".** Unity ofrece el mejor conjunto de herramientas integradas para 2D: *Tilemap* (reglas, capas), 2D Animation/Sprite Editor, Cinemachine 2D, nuevo *Input System*, *Profiler* y depuración de físicas.

Este ecosistema reduce deuda técnica y acorta integraciones [Uni25a; Uni25b; Uni25f].

2. **Ecosistema y escalabilidad.** La Asset Store y la base de conocimiento permiten resolver con rapidez necesidades de producción (*pathfinding*, efectos, *shaders*, diálogo). Si el proyecto crece (*DLC*, consolas), existen *pipelines* probados y soporte multiplataforma [Uni25e].
3. **Alineación con el *stack* del equipo (C#/.NET).** Conocimiento previo de C# maximiza productividad desde el primer *sprint*. Tipado fuerte, pruebas y *tooling* IDE mejoran mantenibilidad.
4. **Rendimiento suficiente y opciones de optimización.** Para 2D convencional, Unity rinde sobrado en PC y ofrece rutas de optimización (Burst, Jobs/DOTS, Sprite Atlas, Addressables, URP 2D) para futuros requisitos [Uni25a].
5. **Gestión del riesgo y previsibilidad.** Aunque ha habido cambios de licencia, Unity mantiene un plan gratuito adecuado para escalas iniciales. Se agenda revisión de licencia en preproducción y *vertical slice* para asegurar cumplimiento (ver documentación de despliegue y requisitos WebGL) [Uni25d].

***Caveats* y mitigaciones.**

- *Tamaño de build (PC/WebGL) elevado.* **Mitigación:** IL2CPP + Stripping, compresión Brotli/Gzip en WebGL, Addressables para carga progresiva, Sprite Atlas y texturas comprimidas [Uni25d].
- *Curva del editor.* **Mitigación:** guía interna de proyecto (convenciones de carpetas, *prefabs*/variantes, ScriptableObjects), plantillas de escena y *cookbook* de patrones [Uni25e].
- *WebGL con consumo de memoria/tiempos de carga.* **Mitigación:** presupuesto de *assets*, audio comprimido, formatos adecuados, limitar *plugins* pesados, pruebas en hardware objetivo y *gates* de *performance* en CI [Uni25d].

Conclusión. Unity equilibra potencia de herramientas, ecosistema y previsibilidad para un proyecto 2D con opción de crecer en alcance o plataformas. Aun asumiendo *builds* más pesadas que alternativas como Godot/Phaser, las mitigaciones de tamaño y el *tooling* profesional justifican la elección. La decisión queda aceptada.

Capítulo 3

Tablero y sistema de piezas

3.1. Implementación de una rejilla visible de 10×20

En Unity, un *Tilemap* es un sistema 2D para construir escenarios mediante *tiles* (baldosas) organizadas sobre una cuadrícula [Uni25i]. Cada *tile* suele estar asociado a un *sprite* reutilizable, lo que facilita la creación de superficies, bordes y elementos decorativos con un coste reducido de tiempo y memoria [Uni25i]. Los *Tilemaps* se alojan dentro de un objeto *Grid*, que define el tamaño de celda y la orientación (cuadrada, isométrica o hexagonal) [Uni25i].

La herramienta *Tile Palette* permite pintar, borrar y reemplazar baldosas de forma similar a un editor de niveles, mientras que el *Tilemap Renderer* dibuja el conjunto de *tiles* de manera eficiente y permite controlar el orden de renderizado mediante capas (*sorting layers*) [Uni25i]. Además, si se requiere interacción física, puede añadirse un *Tilemap Collider 2D* para dotar a las baldosas de colisión [Uni25h].

Para acelerar la construcción de patrones repetitivos, Unity proporciona *Rule Tiles*, que aplican reglas para colocar bordes y esquinas de manera automática según la vecindad [Uni25g]. También es habitual separar el escenario en múltiples *Tilemaps* (por ejemplo, suelo, bordes y decoración) dentro del mismo *Grid*, con el fin de organizar el contenido y ajustar el renderizado de cada capa de forma independiente [Uni25i].

En términos de rendimiento, los *Tilemaps* reducen *draw calls* al agrupar sprites, por lo que son una solución habitual en juegos 2D como plataformas o RPGs [Uni25i]. Asimismo, se integran con iluminación 2D, efectos y scripts, lo que permite desde animaciones hasta modificación de baldosas en *runtime* [Uni25i].

3.1.1. Configuración en Unity

A continuación se resumen los pasos principales seguidos para preparar el tablero visible de 10×20 en Unity:

- Se renombra la escena dentro de la carpeta `Scenes` a `Tetraversus`.

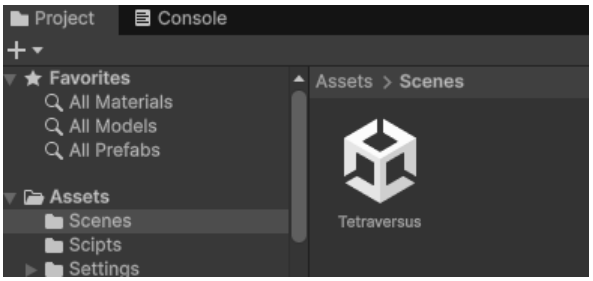


FIGURA 3.1: Escena `Tetraversus` renombrada.

- Se ajusta el color de fondo de la cámara a `#2F353D`.

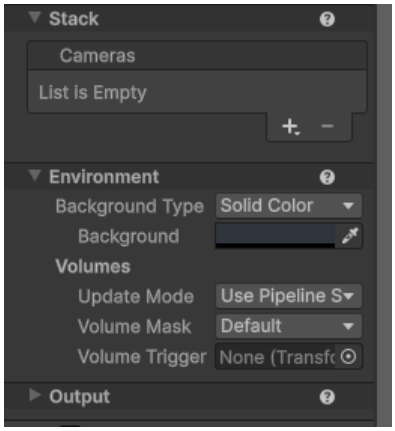


FIGURA 3.2: Configuración del color de fondo de la cámara.

- Se configura el tamaño de la cámara a 12 para que el tablero quede correctamente encuadrado.

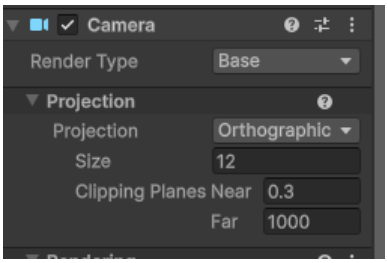


FIGURA 3.3: Ajuste del tamaño de la cámara a 12.

- Se establece el *draw mode* a *Tile* y se define el área de dibujo con dimensiones 10×20.

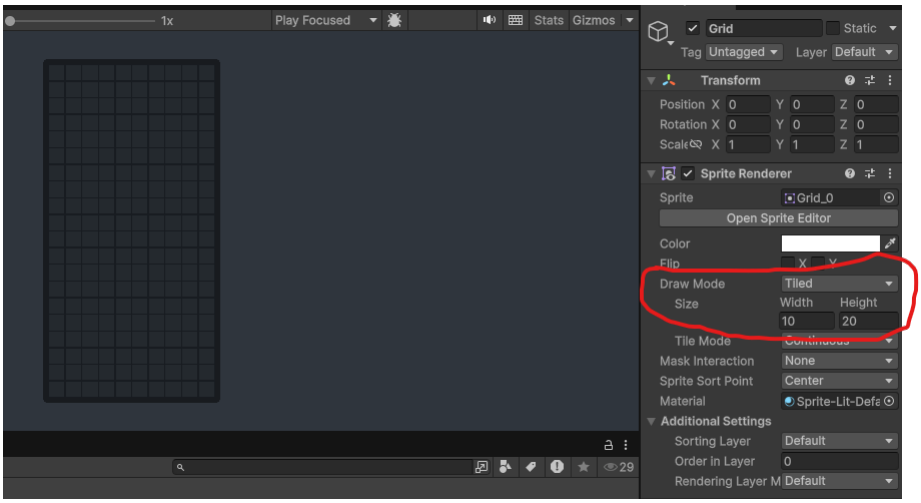


FIGURA 3.4: Opciones del *draw mode*.

- Se crea el tablero y se ajusta el orden de renderizado de las capas.

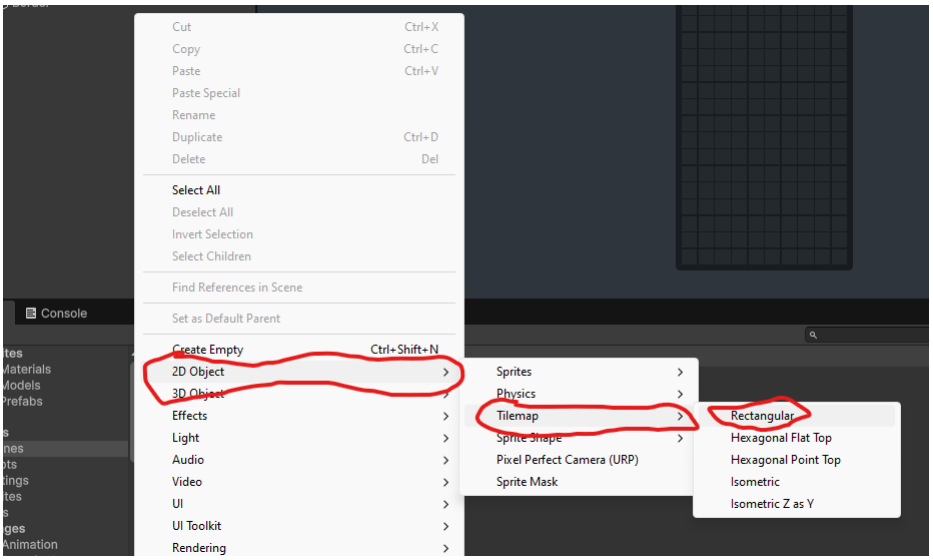


FIGURA 3.5: Creación de objetos 2D y configuración del tablero.

- Se organiza el orden de capas para evitar solapes visuales.

FIGURA 3.6: Menú de ordenación de capas.

- Orden de capas utilizado:

- `Grid`: 0
- `Ghost`: 1
- `Board`: 2
- `Border`: 3

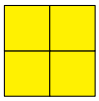
3.2. Piezas: I, O, T, S, Z, J y L

Un tetrominó es una figura formada por cuatro cuadrados unidos por sus lados. Desde un punto de vista matemático existen cinco tetrominós "libres" si se considera equivalente una pieza y su reflejo, pero Tetris utiliza siete piezas al distinguir las versiones espejadas [Wik25]. Las siete piezas del juego son: I, O, T, S, Z, J y L.

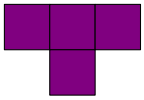
A continuación se muestran las siete piezas representadas sobre una rejilla:

I (barra)

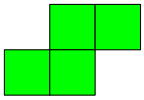
O (cuadrado)



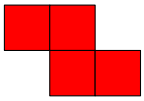
T



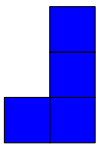
S (escalera derecha)



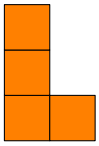
Z (escalera izquierda)



J (L invertida)



L



En esta fase del proyecto, las coordenadas de cada pieza se mantienen fijadas en el código (*hardcoded*). Como mejora futura, podrían externalizarse a un fichero de configuración y cargarse al iniciar el juego. Para almacenar dichas coordenadas se utilizan *arrays* de tipo `Vector2Int`, que permiten representar posiciones enteras en rejilla.

En el fichero `Data.cs` se definen las coordenadas asociadas a cada tetrominó. Finalmente, al *GameObject* `Board` se le añade el script `Board.cs`, encargado de gestionar el tablero, la pieza activa y la generación de nuevas piezas. La posición inicial de aparición se define como:

```
public Vector3Int spawnPosition { get; } = new Vector3Int(-1, 8, 0);
```

Nota: los *Tilemaps* trabajan con `Vector3Int` porque se apoyan en un espacio tridimensional, aunque el juego se renderice en 2D.

3.3. Generador aleatorio tipo *bolsa* de siete piezas

Para aproximarse al comportamiento estándar de Tetris, se utiliza un generador aleatorio basado en una *bolsa* de siete piezas. La idea es evitar rachas excesivas de una misma pieza: se construye una lista con los siete tetrominós, se baraja y se van extrayendo uno a uno. Cuando la bolsa se vacía, se vuelve a rellenar y barajar.

En la clase `Data.cs` se define un array con los siete tetrominós:

```
public TetrominoData[] tetrominoes;
```

Al ser un campo público, se inicializa desde el *Inspector* de Unity:

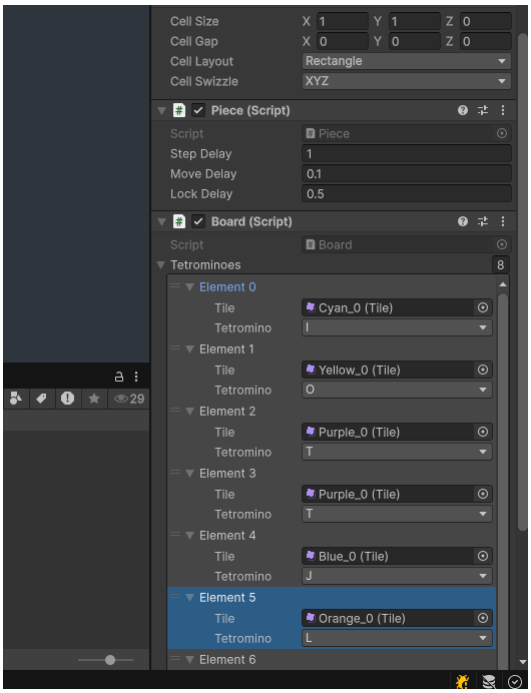


FIGURA 3.7: Configuración del array `tetrominoes` en el *Inspector* de Unity.

Para iniciar la partida, el `Board` genera la primera pieza invocando `SpawnPiece()` desde `Start()`. A partir de ese momento, el método `Update()` se ejecuta en cada *frame* para aplicar la lógica de caída según el temporizador interno:

```
private void Update()
```

Cuando la pieza llega al fondo del tablero o colisiona con bloques ya fijados, se ejecuta el flujo típico del sistema:

```
Step()
Move()
Lock()
ClearLines()
SpawnPiece()
```

Estas funciones se encargan, respectivamente, de avanzar el estado de caída, procesar movimientos, fijar la pieza en el tablero, eliminar líneas completas y generar la siguiente pieza.

3.4. Aceptación

En las pruebas realizadas, la pieza aparece en una posición inicial centrada dentro del área jugable y desciende correctamente sin solaparse con bloques existentes ni salirse de los límites del tablero. Asimismo, la lógica de bloqueo y generación de la siguiente pieza mantiene la consistencia del estado del tablero, garantizando que el flujo de juego básico se comporta como se espera.

Capítulo 4

Movimiento, rotación y colisiones

4.1. Matemática de matrices

En el juego, la posición de cada tetrominó puede modelarse como un conjunto de puntos en una rejilla discreta. Para describir transformaciones como rotaciones, resulta útil apoyarse en álgebra lineal y, en particular, en el producto de matrices por vectores.

Sea el punto (vector columna):

$$\mathbf{p} = \begin{bmatrix} x \\ y \end{bmatrix}$$

y una matriz 2×2:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}.$$

El producto de la matriz por el punto queda definido como:

$$A\,\mathbf{p} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}.$$

Ejemplo. Si $A = \begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix}$ y $\mathbf{p} = \begin{bmatrix} 4 \\ -1 \end{bmatrix}$, entonces:

$$A\,\mathbf{p} = \begin{bmatrix} 2 \cdot 4 + 1 \cdot (-1) \\ 0 \cdot 4 + 3 \cdot (-1) \end{bmatrix} = \begin{bmatrix} 7 \\ -3 \end{bmatrix}.$$

Para obtener las nuevas coordenadas de un punto rotado en el plano se utiliza la matriz de rotación de ángulo θ:

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

En particular, para un giro horario de -90° se tiene:

$$R(-90^\circ) = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}.$$

4.2. Matriz de rotación horaria

En ejes matemáticos (x hacia la derecha, y hacia arriba) y utilizando **vectores columna**, un giro horario de -90° alrededor del **origen** viene dado por:

$$R_{-90} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \qquad \begin{pmatrix} x' \\ y' \end{pmatrix} = R_{-90} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} y \\ -x \end{pmatrix}.$$

Regla especial para I y O. Para las piezas `I` y `O` el pivote se encuentra en un centro de subcelda $(0{,}5, 0{,}5)$. Por ello, para rotarlas en rejilla se aplica el siguiente procedimiento: restar $(0{,}5, 0{,}5)$ a cada celda, aplicar R_{-90}, y finalmente usar $\lceil \cdot \rceil$ coordenada a coordenada para volver a posiciones enteras.

Pieza T — giro horario -90° en el origen

Original: $\mathcal{T} = \{(0,0), (1,0), (-1,0), (0,1)\}$.

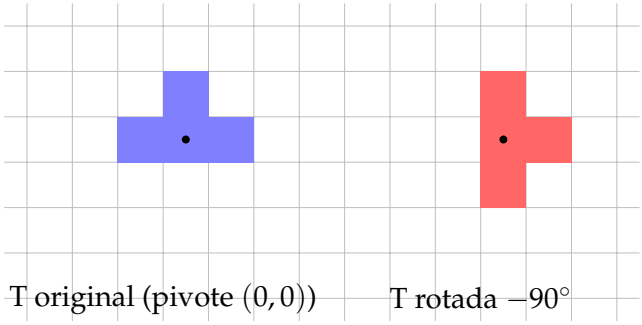
$$(0,0) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} = (0,0),$$

$$(1,0) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \end{pmatrix} = (0,-1),$$

$$(-1,0) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} -1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = (0,1),$$

$$(0,1) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = (1,0).$$

Rotada: $\{(0,0), (0,-1), (0,1), (1,0)\}$.



T original (pivote $(0,0)$) T rotada -90°

Pieza L — giro horario -90° en el origen

Original: $\mathcal{L} = \{(0,0), (0,1), (0,2), (1,0)\}$.

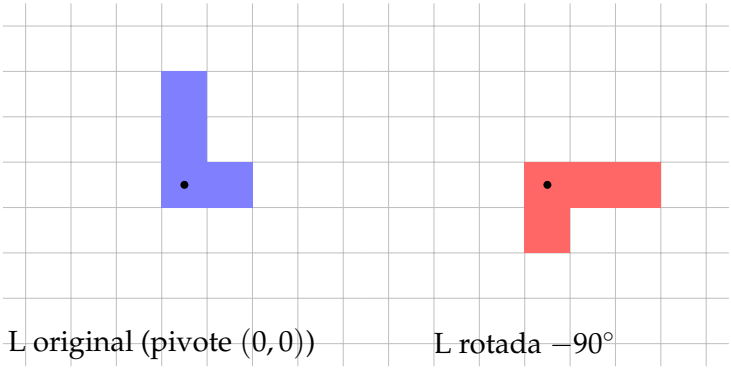
$$(0,0) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} = (0,0),$$

$$(0,1) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = (1,0),$$

$$(0,2) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \end{pmatrix} = (2,0),$$

$$(1,0) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \end{pmatrix} = (0,-1).$$

Rotada: $\{(0,0), (1,0), (2,0), (0,-1)\}$.



L original (pivote $(0,0)$) L rotada -90°

Pieza J — giro horario -90° en el origen

Original: $\mathcal{J} = \{(0,0), (0,1), (0,2), (-1,0)\}$.

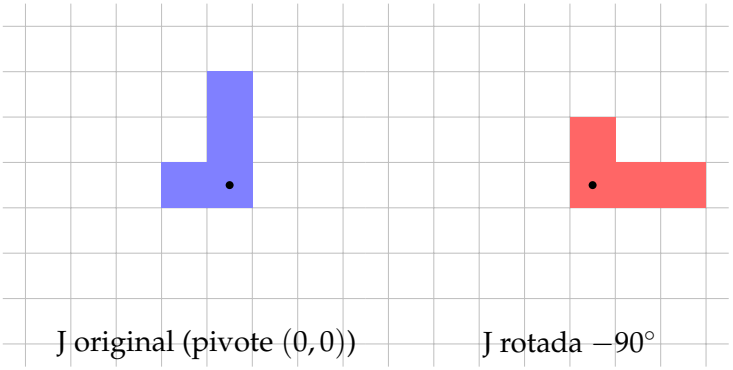
$$(0,0) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} = (0,0),$$

$$(0,1) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = (1,0),$$

$$(0,2) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \end{pmatrix} = (2,0),$$

$$(-1,0) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} -1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = (0,1).$$

Rotada: $\{(0,0), (1,0), (2,0), (0,1)\}$.



J original (pivote $(0,0)$) J rotada $-90°$

Pieza S — giro horario $-90°$ en el origen

Original: $\mathcal{S} = \{(0,0),(1,0),(-1,1),(0,1)\}$.

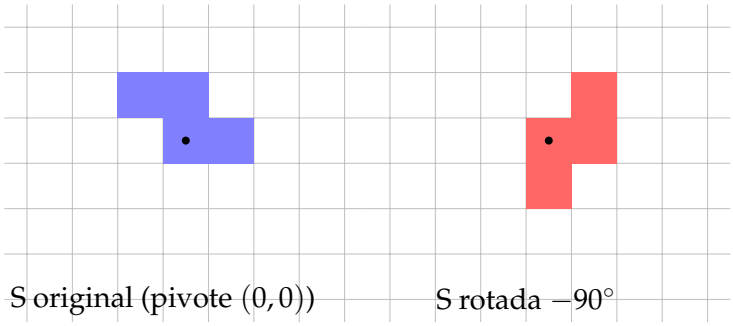
$$(0,0) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} = (0,0),$$

$$(1,0) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \end{pmatrix} = (0,-1),$$

$$(-1,1) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} -1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} = (1,1),$$

$$(0,1) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = (1,0).$$

Rotada: $\{(0,0),(0,-1),(1,1),(1,0)\}$.



S original (pivote $(0,0)$) S rotada $-90°$

Pieza Z — giro horario -90° en el origen

Original: $\mathcal{Z} = \{(0,0), (-1,0), (0,1), (1,1)\}$.

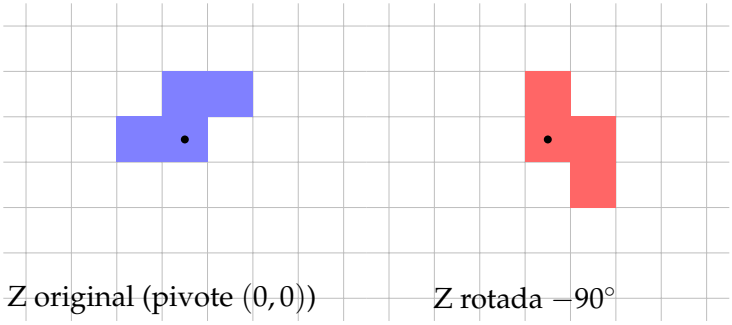
$$(0,0) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} = (0,0),$$

$$(-1,0) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} -1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = (0,1),$$

$$(0,1) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = (1,0),$$

$$(1,1) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \end{pmatrix} = (1,-1).$$

Rotada: $\{(0,0), (0,1), (1,0), (1,-1)\}$.



Pieza I — restar $(0{,}5, 0{,}5)$, rotar -90° y aplicar $\lceil \cdot \rceil$

Original: $\mathcal{I} = \{(-1,0), (0,0), (1,0), (2,0)\}$.

Paso 1: restar $(0{,}5, 0{,}5)$.

$$\{(-1{,}5, -0{,}5),\ (-0{,}5, -0{,}5),\ (0{,}5, -0{,}5),\ (1{,}5, -0{,}5)\}.$$

Paso 2: multiplicar por $R_{-90} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$.

$$(-1{,}5, -0{,}5) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} -1{,}5 \\ -0{,}5 \end{pmatrix} = \begin{pmatrix} -0{,}5 \\ 1{,}5 \end{pmatrix},$$

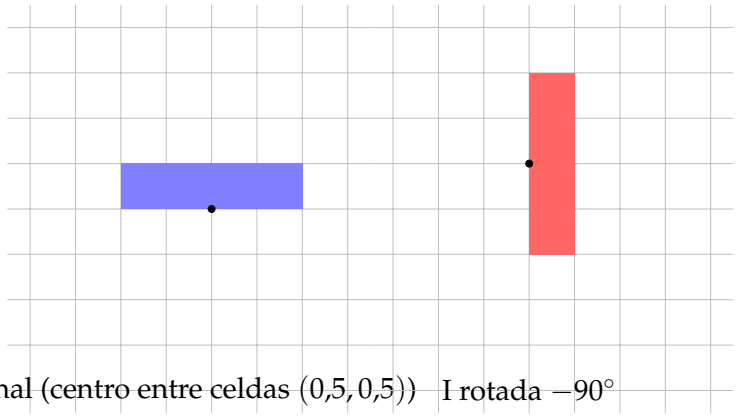
$$(-0{,}5, -0{,}5) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} -0{,}5 \\ -0{,}5 \end{pmatrix} = \begin{pmatrix} -0{,}5 \\ 0{,}5 \end{pmatrix},$$

$$(0{,}5, -0{,}5) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 0{,}5 \\ -0{,}5 \end{pmatrix} = \begin{pmatrix} -0{,}5 \\ -0{,}5 \end{pmatrix},$$

$$(1{,}5, -0{,}5) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 1{,}5 \\ -0{,}5 \end{pmatrix} = \begin{pmatrix} -0{,}5 \\ -1{,}5 \end{pmatrix}.$$

Paso 3: aplicar $\lceil\cdot\rceil$ por coordenada.

$$\{(0,2),(0,1),(0,0),(0,-1)\} \Rightarrow \text{rotada} = \{(0,-1),(0,0),(0,1),(0,2)\}.$$



I original (centro entre celdas $(0{,}5, 0{,}5)$) I rotada $-90°$

Pieza O — restar $(0{,}5, 0{,}5)$, rotar $-90°$ y aplicar $\lceil\cdot\rceil$

Original: $\mathcal{O} = \{(0,0),(1,0),(0,1),(1,1)\}$.

Paso 1: restar $(0{,}5, 0{,}5)$**.**

$(0,0) \to (-0{,}5, -0{,}5)$, $(1,0) \to (0{,}5, -0{,}5)$, $(0,1) \to (-0{,}5, 0{,}5)$, $(1,1) \to (0{,}5, 0{,}5)$.

Paso 2: multiplicar por R_{-90}.

$$(-0{,}5,-0{,}5) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} -0{,}5 \\ -0{,}5 \end{pmatrix} = \begin{pmatrix} -0{,}5 \\ 0{,}5 \end{pmatrix},$$

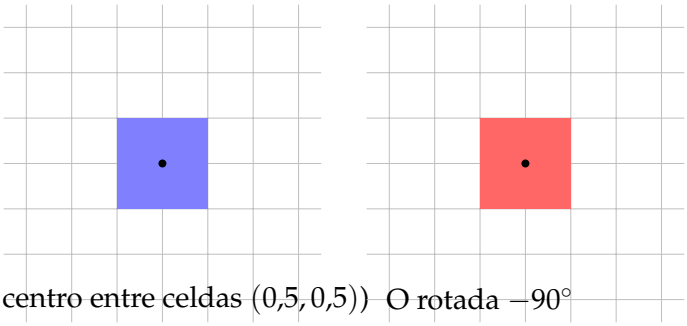
$$(0{,}5,-0{,}5) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 0{,}5 \\ -0{,}5 \end{pmatrix} = \begin{pmatrix} -0{,}5 \\ -0{,}5 \end{pmatrix},$$

$$(-0{,}5,0{,}5) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} -0{,}5 \\ 0{,}5 \end{pmatrix} = \begin{pmatrix} 0{,}5 \\ 0{,}5 \end{pmatrix},$$

$$(0{,}5,0{,}5) \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 0{,}5 \\ 0{,}5 \end{pmatrix} = \begin{pmatrix} 0{,}5 \\ -0{,}5 \end{pmatrix}.$$

Paso 3: aplicar $\lceil \cdot \rceil$**.**

$\{(0,1),(0,0),(1,1),(1,0)\} \Longrightarrow \{(0,0),(1,0),(0,1),(1,1)\}$ (misma forma).



O original centro entre celdas $(0{,}5,0{,}5)$) O rotada $-90°$

4.3. Todas las piezas con sus respectivos centros

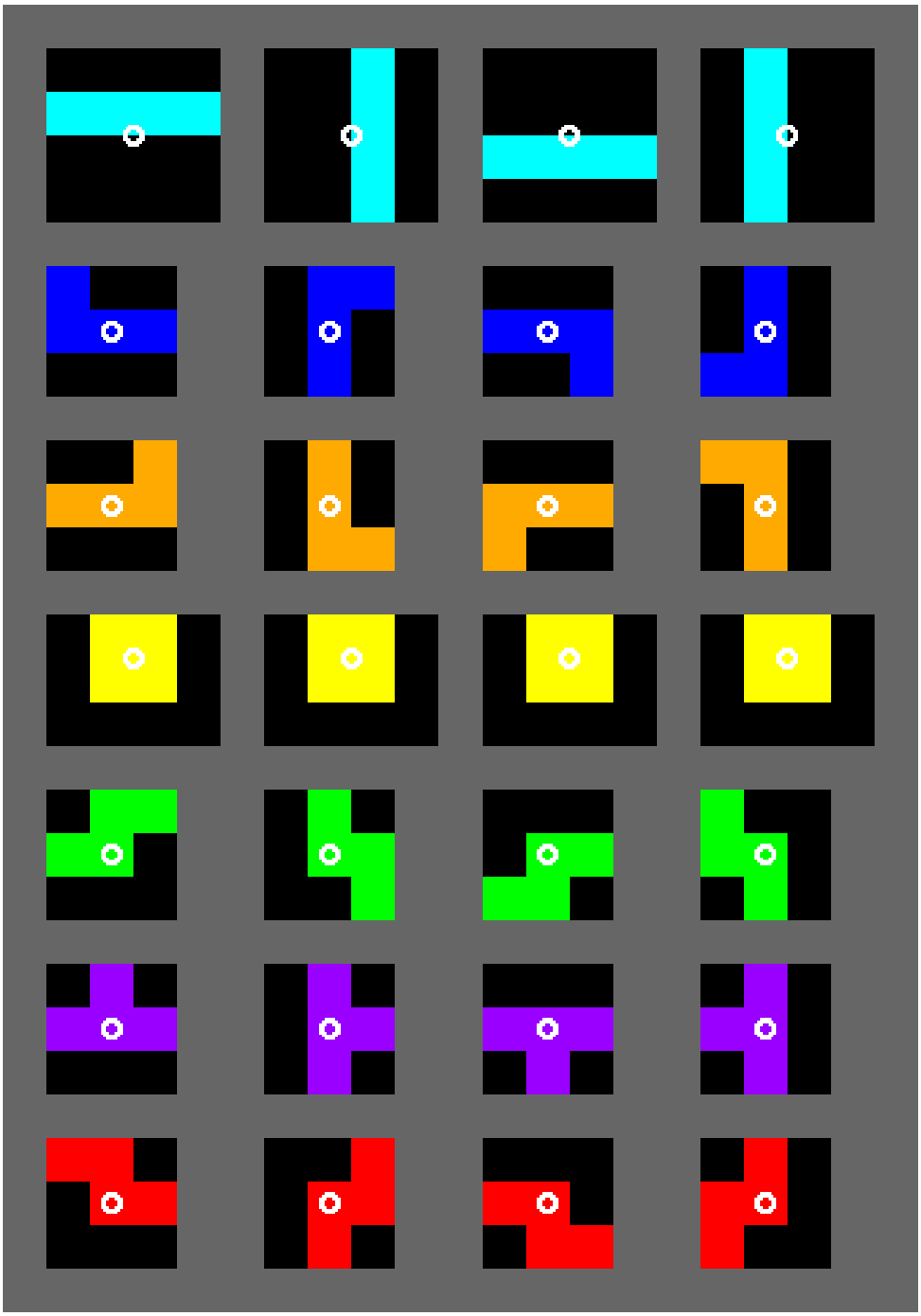


FIGURA 4.1: Representación de las piezas T, L, J, S, Z, I y O con sus respectivos centros de rotación (pivotes).

4.4. Colisiones

Las colisiones determinan si un movimiento o una rotación de una pieza es válido. En este proyecto, el tablero se representa como una rejilla discreta de celdas; cada celda puede estar vacía u ocupada por un bloque "fijado" (parte de una pieza que

ya ha aterrizado). La pieza activa se describe mediante un conjunto de coordenadas (x, y) relativas a un pivote, como se ha mostrado en las secciones anteriores.

En cada paso, el sistema no actualiza la pieza "a ciegas"; primero construye una posición candidata. Para un movimiento de traslación, se suma un vector de desplazamiento $(\Delta x, \Delta y)$ a todas las celdas de la pieza. Para una rotación, se aplica la matriz correspondiente (por ejemplo, R_{-90} en un giro horario de $-90°$). El resultado es un nuevo conjunto de coordenadas $\{(x', y')\}$ que representa la posición propuesta.

Se considera que existe colisión si alguna celda de la posición candidata:

- Queda fuera de los límites del tablero (pared izquierda, pared derecha o por debajo del suelo), o
- Coincide con una celda ya ocupada por bloques fijados.

Si no hay colisión, el movimiento se acepta y se actualiza el estado de la pieza activa. Si hay colisión, el movimiento se rechaza y la pieza permanece en su estado anterior. En el caso de *soft drop* o *hard drop*, si al intentar descender se detecta colisión con el suelo o con bloques fijados, se considera que la pieza ha aterrizado: sus celdas pasan a formar parte del tablero y se desencadena la comprobación de líneas completas para su posible eliminación.

4.4.1. Rotación, orientación y *wall kicks*

El algoritmo de rotación sigue el enfoque de "probar antes de aceptar", incorporando el concepto de *orientación* y el comportamiento de *wall kicks*. Cada pieza dispone de un `rotationIndex` que indica su orientación dentro de las cuatro posibles (0, 1, 2 o 3). Cuando el jugador solicita una rotación, el método `Rotate(direction)` incrementa o decrementa este índice (según el giro sea horario o antihorario) y realiza un *wrap* para mantenerlo en el rango $[0, 3]$.

A continuación, `ApplyRotationMatrix(direction)` aplica la matriz de rotación a las coordenadas de la pieza y genera la posición candidata rotada. Esta posición no se acepta de inmediato: se comprueba la colisión y, si procede, se evalúan una serie de desplazamientos adicionales predefinidos mediante `TestWallKicks`. El objetivo es intentar "encajar" la rotación cerca de paredes o en configuraciones estrechas (comportamiento típico del sistema SRS en variantes modernas de Tetris).

Si alguno de estos desplazamientos produce una configuración válida (sin colisión), la rotación se acepta en la nueva posición. En caso contrario, `TestWallKicks` devuelve `false` y el sistema deshace la operación, restaurando el `rotationIndex` previo y manteniendo la pieza en su estado original. Desde el punto de vista del jugador, la rotación simplemente no se ejecuta porque violaría las reglas de colisión.

4.5. Aceptación

Se han implementado las mecánicas de movimiento, rotación y colisiones para todos los tetrominós. Cada pieza responde correctamente a los comandos de desplazamiento lateral, rotación (horaria y antihoraria), *soft drop* y *hard drop*. Además, se han realizado pruebas específicas para comprobar que las colisiones con paredes, suelo y bloques fijados se detectan de forma consistente en todas las piezas y orientaciones.

Asimismo, se verificó el funcionamiento de los *wall kicks* en situaciones de esquina y proximidad a paredes. El sistema intenta desplazamientos correctivos antes de rechazar una rotación, permitiendo rotaciones naturales en espacios reducidos sin invadir celdas inválidas del tablero. En conjunto, las pruebas confirman que el sistema de colisiones y rotaciones mantiene la integridad de la rejilla y produce un comportamiento coherente con las expectativas del género.

Capítulo 5

Líneas, puntuación y niveles

5.1. Detección de líneas completas y su eliminación

La detección y eliminación de líneas se realiza recorriendo el tablero de abajo hacia arriba. Este enfoque permite procesar correctamente el desplazamiento de filas tras una eliminación y evita omitir posibles líneas completas.

- Se recorre el tablero desde la fila inferior hacia la superior.
- Cuando una fila está completa, se elimina y se desplazan hacia abajo todas las filas situadas por encima.
- Tras el desplazamiento, no se avanza de fila, ya que en esa misma posición puede haber caído una nueva fila que también podría estar completa.
- Durante el desplazamiento, se garantiza que nunca se accede fuera de los límites del tablero y, al finalizar, se vacía explícitamente la fila superior.

5.2. Sistema de puntuación con progresión de niveles y aumento de velocidad

El sistema de puntuación se ha planteado con el objetivo de recompensar la eliminación de líneas y reflejar la progresión del jugador mediante niveles. A nivel de diseño, la puntuación no debería depender únicamente del número total de líneas eliminadas, sino también del tipo de jugada realizada: eliminar varias líneas simultáneamente debería otorgar una mayor recompensa mediante un multiplicador. Además, como mejora habitual en el género, también puede contemplarse un multiplicador adicional por rachas (*combos*), es decir, por eliminar líneas en turnos consecutivos sin fallar.

- Al eliminar una línea, se suma una cantidad base de puntos.
- Si se eliminan varias líneas simultáneamente, se aplica un multiplicador de puntuación.
- De forma teórica, si el jugador elimina líneas de manera consecutiva (racha o *combo*), podría aplicarse un multiplicador adicional para incentivar la consistencia y el juego avanzado.
- Cada 10 líneas eliminadas, se incrementa el nivel del juego, representando el progreso de la partida.

- Idealmente, al subir de nivel se reduciría el tiempo entre caídas, aumentando la velocidad de descenso de las piezas y, por tanto, la dificultad.
- La interfaz muestra en todo momento la puntuación y el nivel actuales.

Aunque el diseño contempla el aumento de velocidad como parte natural de la progresión de dificultad, esta mecánica no se llegó a implementar en la versión final por falta de tiempo. De igual forma, el multiplicador por líneas eliminadas de manera consecutiva (*combo*) quedó definido a nivel conceptual, pero no se incorporó en esta versión por la misma limitación temporal. Ambas mejoras se consideran prioritarias para futuras iteraciones, ya que aportarían una progresión de dificultad y una profundidad de puntuación más completas.

5.3. Contadores de puntos y nivel en la interfaz gráfica

Para centralizar el estado de la partida (puntuación, líneas eliminadas y nivel) y asegurar que todas las piezas actualizan los contadores de forma consistente, se ha aplicado el patrón *Singleton* en el controlador de puntuación.

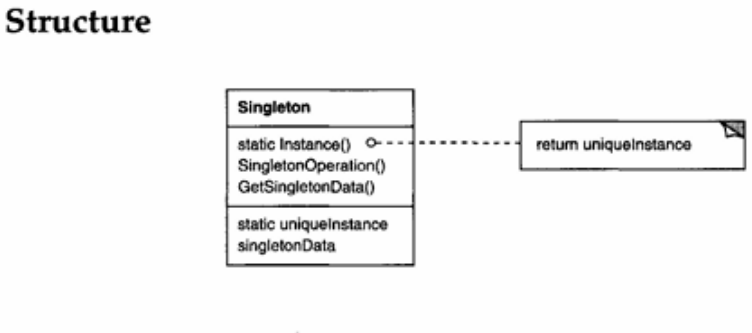


FIGURA 5.1: Diagrama UML del patrón *Singleton*.[Gam+94]

La Figura 5.2 muestra el diagrama de clases del sistema de puntuación y niveles.

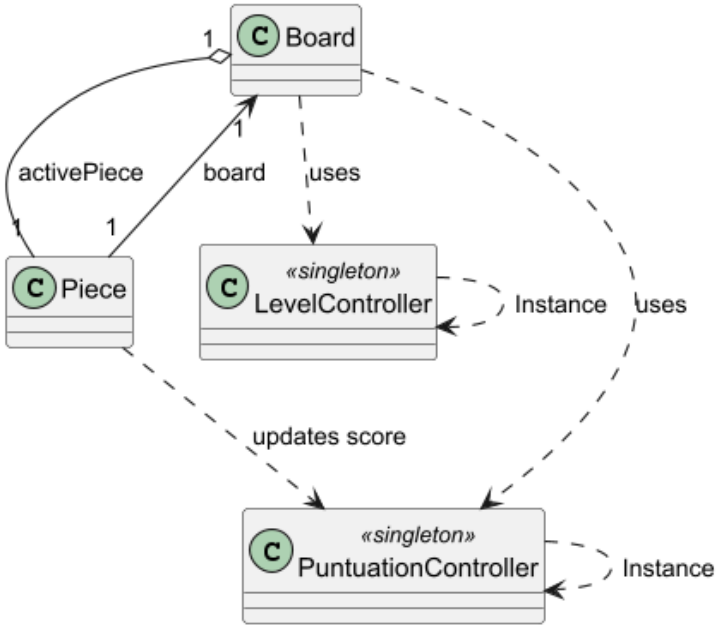


FIGURA 5.2: Diagrama de clases del sistema de puntuación y niveles.

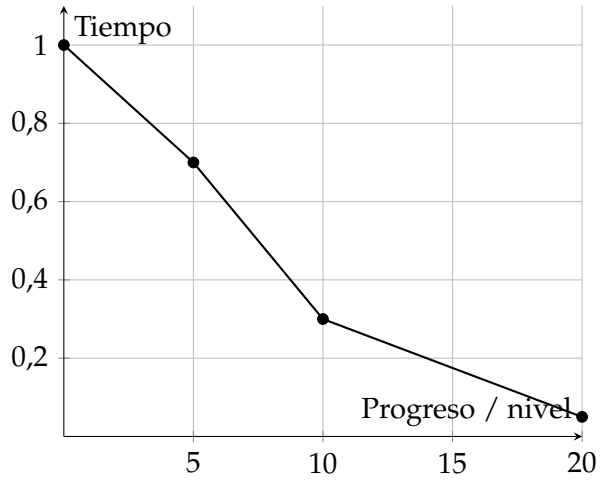
La clase `Piece` notifica los eventos relevantes para la puntuación (por ejemplo, si la pieza ha descendido mediante *soft drop* o *hard drop*) al controlador `PuntuationController`, que actualiza la puntuación acumulada. Dado que cada pieza generada es una nueva instancia de `Piece`, resulta necesario disponer de un punto de acceso común para mantener el estado global de la partida. Por este motivo, `PuntuationController` se implementa como un *Singleton*, garantizando una única instancia compartida por todas las piezas y evitando inconsistencias.

Por su parte, `LevelController` podría implementarse como una clase convencional, ya que se invoca principalmente desde `Board`. No obstante, se ha optado por aplicar también *Singleton* para mantener coherencia en el diseño del conjunto de controladores del sistema de puntuación y niveles.[Gam+94]

5.4. Implementación del gráfico de nivel de dificultad del juego

Con el objetivo de representar la progresión de dificultad, se plantea modelar el tiempo de caída en función del nivel mediante una función a trozos:

$$D(x) = \begin{cases} -0{,}06x + 1, & 0 \leq x \leq 5, \\ -0{,}08x + 1{,}10, & 5 < x \leq 10, \\ -0{,}025x + 0{,}55, & 10 < x \leq 20. \end{cases}$$



Actualmente, esta parte queda planteada a nivel de diseño y se encuentra pendiente de implementación.

5.5. Aceptación

Se ha implementado la detección y eliminación de líneas completas recorriendo el tablero de abajo hacia arriba, lo que permite identificar filas completas y eliminarlas de forma correcta. Tras cada eliminación, las filas superiores se desplazan hacia abajo y se garantiza que la fila superior se vacía explícitamente, evitando lecturas fuera de los límites del tablero.

Asimismo, se ha desarrollado un sistema de puntuación que incrementa los puntos al eliminar líneas y aplica un multiplicador cuando se eliminan varias líneas de forma simultánea. El nivel se incrementa cada línea eliminada pero se podría poner el valor deseado y tanto la puntuación como el nivel se muestran en la interfaz, proporcionando información clara y continua al jugador.

En cuanto a las mejoras previstas para el sistema, el aumento de velocidad asociado a la subida de nivel se definió como parte del diseño, pero no se incluyó en la versión final por limitaciones de tiempo. Del mismo modo, el multiplicador adicional por eliminación consecutiva de líneas (*combo*) no se llegó a implementar en esta entrega. Ambas funcionalidades quedan planteadas como trabajo futuro para enriquecer la progresión de dificultad y la profundidad del sistema de puntuación.

Para verificar el comportamiento implementado, se realizaron pruebas en diferentes situaciones de juego, confirmando que las líneas se detectan y eliminan correctamente, y que la puntuación y el nivel se actualizan conforme a las reglas establecidas.

Capítulo 6

Mecánicas de calidad de vida

6.1. Ghost piece (sombra de aterrizaje) con botón *on*/*off*

Esta mecánica muestra una sombra de la pieza que se está moviendo, indicando la posición exacta en la que aterrizaría si se realizara la caída en ese instante. De este modo, el jugador puede anticipar colocaciones y planificar movimientos con mayor precisión, reduciendo errores y mejorando la toma de decisiones.

Además, podría incorporarse una opción de activación/desactivación para adaptarse a las preferencias del jugador. Para ello, sería suficiente con añadir un botón en el menú de opciones que permita alternar la visibilidad de la sombra de aterrizaje. Cuando esta opción estuviera habilitada y el jugador desplazara una pieza, el sistema calcularía su posición final y representaría una silueta translúcida en dicha ubicación. En cambio, si el jugador desactivara la función, la sombra dejaría de mostrarse, ofreciendo una experiencia de juego más tradicional.

6.1.1. Implementación por código de la *ghost piece*

Para implementar la mecánica de *ghost piece*, se modificó el método `LateUpdate()` del script principal. En cada ciclo de actualización se limpia el tablero visual, se copian las coordenadas de la pieza activa y se invoca `HardDrop()`, que calcula la posición de aterrizaje. A continuación, se dibuja la pieza y, si la opción está activada, también la sombra en la posición calculada.

El código del método `LateUpdate()` es el siguiente:

```
private void LateUpdate()
{
    Clear();
    CopyToNewPieceCoordenates();
    HardDrop();
    DrawPiece();
}
```

6.1.2. Criterios de aceptación

La interfaz del juego muestra correctamente los paneles de *hold* y *next*, y la *ghost piece* coincide con la posición final real de la pieza al soltarla. Para comprobarlo, pueden realizarse pruebas moviendo piezas en distintas situaciones y verificando que la sombra indica con precisión el punto de aterrizaje.

Asimismo, de forma opcional, podría validarse que la función puede activarse y desactivarse desde el menú de opciones sin afectar a la fluidez del juego. Por último, también sería recomendable efectuar comprobaciones adicionales para asegurar que la visualización es clara, no interfiere con otras mecánicas y se integra correctamente en la experiencia de juego.

En conclusión, la implementación de la *ghost piece* puede considerarse satisfactoria siempre que cumpla los criterios de aceptación definidos, especialmente en lo relativo a la precisión de la posición de aterrizaje y a su correcta integración con la interfaz.

Capítulo 7

Base de datos y servicios web

Para gestionar la autenticación de usuarios, se ha creado la tabla `users`, compuesta por tres campos: `id`, `username` y `password`. El campo `id` es numérico y se ha definido como clave primaria (*Primary Key*, PK), lo que garantiza que cada registro disponga de un identificador único y no nulo. Por su parte, el campo `username` incorpora una restricción de unicidad (*Unique Key*, UK), evitando el registro de dos usuarios con el mismo nombre. Finalmente, el campo `password` almacena la contraseña del usuario en forma de *hash* (no en texto plano), reforzando la seguridad del sistema. En conjunto, esta estructura permite identificar de manera inequívoca a cada usuario y previene duplicidades en los nombres de usuario.

7.1. Diseño de la base de datos

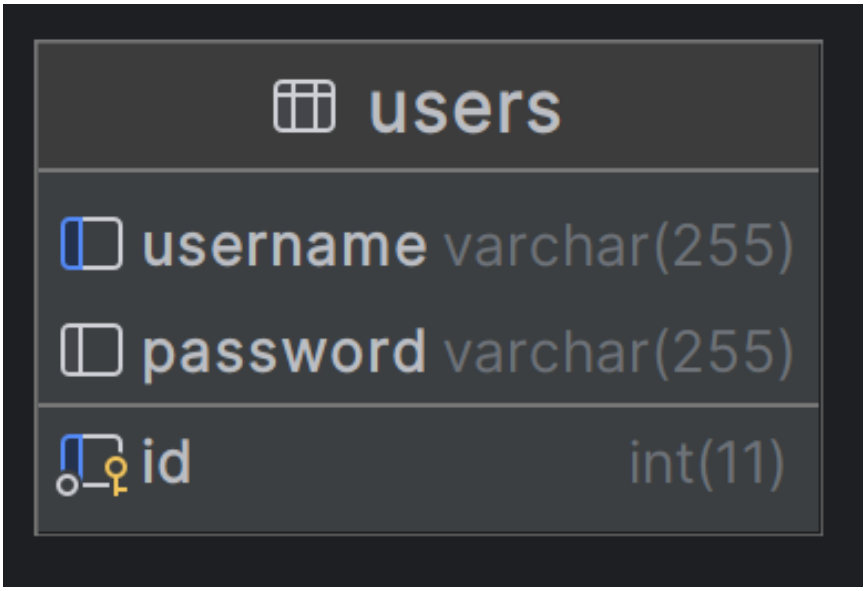


FIGURA 7.1: Diagrama entidad-relación de la base de datos.

7.2. Servicios web

API significa *Application Programming Interface* (interfaz de programación de aplicaciones). Una API web es un conjunto de interfaces expuestas a través de la Web que permite a un cliente comunicarse con un servidor para solicitar y manipular información. Por ejemplo, al reservar un vuelo desde una página web, el navegador envía

solicitudes al servidor de la aerolínea mediante una API web para consultar su base de datos. A continuación, el servidor devuelve la información del vuelo y el cliente puede completar la reserva.

Las organizaciones ofrecen APIs desde hace décadas, pero con la aparición y consolidación de la *World Wide Web* se hizo imprescindible estandarizar la comunicación entre clientes y servidores en entornos heterogéneos.

Podemos construir APIs web utilizando distintas tecnologías (Java, Python, Ruby, PHP, .NET, etc.) y siguiendo diferentes estilos arquitectónicos. Es habitual encontrar términos como SOAP, *web services* y REST. Aunque todos suelen apoyarse en HTTP para el transporte, difieren en el formato de mensajes, en la semántica de la comunicación y en el grado de acoplamiento entre cliente y servidor.

En la década de 1990 el foco se situó en mejorar la comunicación dentro de redes internas, donde TCP/IP se consolidó como estándar. Con el tiempo, la interoperabilidad entre plataformas se volvió crítica y surgieron los *servicios web*, basados en *Simple Object Access Protocol* (SOAP), orientados a escenarios empresariales y al intercambio de datos entre sistemas heterogéneos.

No obstante, el XML utilizado por SOAP resulta relativamente pesado, lo que incrementa el consumo de ancho de banda. A principios de la década de 2000, Microsoft lanzó *Windows Communication Foundation* (WCF), una tecnología que facilitó la gestión de la complejidad asociada a SOAP. WCF se apoya en el enfoque de llamadas a procedimiento remoto (RPC, por sus siglas en inglés), manteniendo SOAP como protocolo subyacente. Con el tiempo, parte de estos estándares han ido cediendo protagonismo a las APIs REST, que se describen en la siguiente sección.

7.3. APIs REST

REST, también conocido como *Representational State Transfer* (transferencia de estado representacional), es un estilo arquitectónico para APIs web propuesto por Roy Fielding en su tesis doctoral "Architectural Styles and the Design of Network-based Software Architectures"[Fie00] (2000). En la práctica, las APIs REST suelen implementarse sobre HTTP; sin embargo, la propuesta original describe conceptos y restricciones arquitectónicas, sin imponer un protocolo concreto. Aun así, dado que HTTP es el estándar predominante en la Web, en este proyecto se utilizará HTTP como base para la API.

Conviene recordar que REST es un estilo, no una norma estricta. Al diseñar una API web no es obligatorio seguir REST: pueden emplearse otros enfoques. No obstante, REST resulta recomendable porque introduce restricciones que promueven un diseño más mantenible, escalable e interoperable, facilitando además la integración con sistemas de terceros cuando se adoptan convenciones comunes.

El concepto central de REST es la transferencia de estado mediante representaciones. Un sistema web puede entenderse como una colección de recursos. Por ejemplo, una colección `books` (libros) es un recurso, y cada libro individual también lo es. Si el cliente solicita la lista de libros a través de un enlace (por ejemplo, `http://www.example.com/books`), el servidor devuelve una representación (habitualmente JSON) con los libros disponibles. Esa representación puede incluir enlaces a otros recursos, como el de un libro concreto (por ejemplo, `http://www.example.com/books/1`). Al

navegar por dichos enlaces, el cliente va obteniendo representaciones y, con ellas, el estado necesario para operar en la aplicación.

A continuación se resumen las restricciones principales de REST.

7.3.1. Las restricciones de REST

El trabajo[Fie00] de Roy Fielding define seis restricciones para los sistemas REST:

- **Cliente-servidor (*Client-server*):** Aplica el principio de separación de responsabilidades. Cliente y servidor evolucionan de forma independiente: el cliente envía solicitudes y el servidor devuelve respuestas. El cliente no necesita conocer los detalles de implementación del servidor, y viceversa.
- **Ausencia de estado (*Statelessness*):** El servidor no mantiene el estado de sesión del cliente. En cada petición, el cliente debe incluir toda la información necesaria para que el servidor pueda procesarla. Esta característica favorece la escalabilidad, ya que el servidor no necesita recordar estado entre peticiones.
- **Cacheabilidad (*Cacheability*):** Las respuestas deben indicar explícita o implícitamente si pueden almacenarse en caché. La caché puede residir en el cliente, en el navegador o en una CDN, mejorando rendimiento y escalabilidad al reducir peticiones repetidas al servidor.
- **Sistema en capas (*Layered system*):** El cliente no tiene por qué conocer la topología completa hasta el servidor final. Pueden existir capas intermedias (proxy, balanceadores, pasarelas, etc.) sin afectar al código cliente/servidor, siempre que se mantenga la interfaz.
- **Código bajo demanda (*Code on demand*):** De forma opcional, el servidor puede enviar código ejecutable al cliente (por ejemplo, JavaScript) para ampliar funcionalidades en el lado cliente.
- **Interfaz uniforme (*Uniform interface*):** Es la restricción clave en REST. Incluye la identificación de recursos, la manipulación mediante representaciones, mensajes autodescriptivos e hipermedia como motor del estado de la aplicación. Su objetivo es simplificar y desacoplar el sistema para que cada componente evolucione de manera independiente.

7.3.2. Diferencias entre URI y URL

Algunos documentos utilizan el término URI. Un *Uniform Resource Identifier* (URI) es una secuencia única de caracteres que identifica un recurso lógico o físico en tecnologías web. Puede basarse en una ubicación, un nombre o ambos.
Un *Uniform Resource Locator* (URL) es un tipo de URI que apunta a un recurso a través de una red e indica el protocolo de acceso (por ejemplo, `http`, `https` o `ftp`). En la práctica, el término URL se utiliza ampliamente. **Aun así, conviene recordar que sus alcances son distintos: URI es el superconjunto de URL.**

7.4. Pasos para crear nuestra API

En los últimos años, el enfoque *API-first* ha ganado relevancia. Este enfoque trata las APIs como ciudadanos de primera clase dentro del proyecto, definiendo primero un contrato que especifica cómo debe comportarse la API antes de escribir implementación. Esto permite que los equipos trabajen en paralelo: por ejemplo, los desarrolladores *front-end* o móviles pueden simular (*mock*) y validar integraciones basándose en dicho contrato. Con herramientas como Swagger[Wik], es posible automatizar parte del proceso, incluyendo documentación, APIs simuladas, SDKs, etc., acelerando el desarrollo y reduciendo el tiempo de salida al mercado.

Estos son algunos pasos habituales para diseñar una API basada en REST:

- Identificar los recursos.
- Definir las relaciones entre los recursos.
- Identificar las operaciones (eventos) relevantes.
- Diseñar las rutas (*paths*) de los recursos.
- Mapear las operaciones a métodos HTTP (GET, POST, PUT, DELETE, etc.).
- Asignar códigos de respuesta coherentes.
- Documentar la API.

7.5. Tabla de nuestros endpoints

Método HTTP	URL	Operación	Descripción
POST	`/api/auth`	Verificar credenciales	Verifica las credenciales del usuario y devuelve un token JWT.
POST	`/api/users`	Crear usuario	Crea un usuario nuevo; devuelve `201 Created` o `409 Conflict` si el usuario ya existe.
PUT	`/api/users`	Actualizar contraseña	Actualiza la contraseña de un usuario existente; devuelve `200 OK` o `401 Unauthorized`.

CUADRO 7.1: Endpoints de autenticación y usuarios.

7.5.1. ¿Es mi API web RESTful?

Como se ha indicado anteriormente, REST no es una regla ni una especificación formal. No existe un estándar oficial que certifique una API como "RESTful". Tampoco exige JSON ni obliga a seguir patrones CRUD. Sin embargo, muchas implementaciones REST en la práctica emplean convenciones ampliamente aceptadas (HTTP, URLs, códigos de estado, representaciones JSON, etc.). En consecuencia, es habitual que se etiqueten como RESTful APIs que no cumplen de forma estricta todas las restricciones descritas por Roy Fielding.

En términos prácticos, rara vez compensa invertir esfuerzo en discutir si una API es "lo bastante REST". El objetivo es construir un sistema funcional, mantenible y

consistente. Además, no todo el mundo ha leído la tesis original y la tecnología evoluciona con rapidez. Como dice el proverbio: "No importa si el gato es blanco o negro; mientras cace ratones, es un buen gato".

Dicho esto, cuando se inicia un proyecto desde cero (*greenfield*), resulta recomendable seguir convenciones para reducir fricción y mejorar la interoperabilidad. En general, una API de inspiración REST suele caracterizarse por:

- Una URL base que actúa como raíz de la API (por ejemplo, `http://api.example.com`).
- El uso consistente de la semántica de los métodos HTTP (GET, POST, PUT, DELETE, etc.).
- Un *media type* que define el formato de las representaciones (por ejemplo, `application/json` o `application/xml`).

7.6. JWT: ¿qué es y cómo se utiliza?

Un JWT (*JSON Web Token*) es un estándar abierto para representar *claims* (declaraciones) entre dos partes. Es compacto y fácil de transportar. El siguiente es un ejemplo de JWT:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJl
... (contenido omitido) ...
9WXtQHO-hJydVJou-YiQYQpkq8
```

Un JWT se compone de tres partes, separadas por puntos:

- *Header* (cabecera)
- *Payload* (carga útil)
- *Signature* (firma)

La Figura 7.2 ilustra esta estructura.

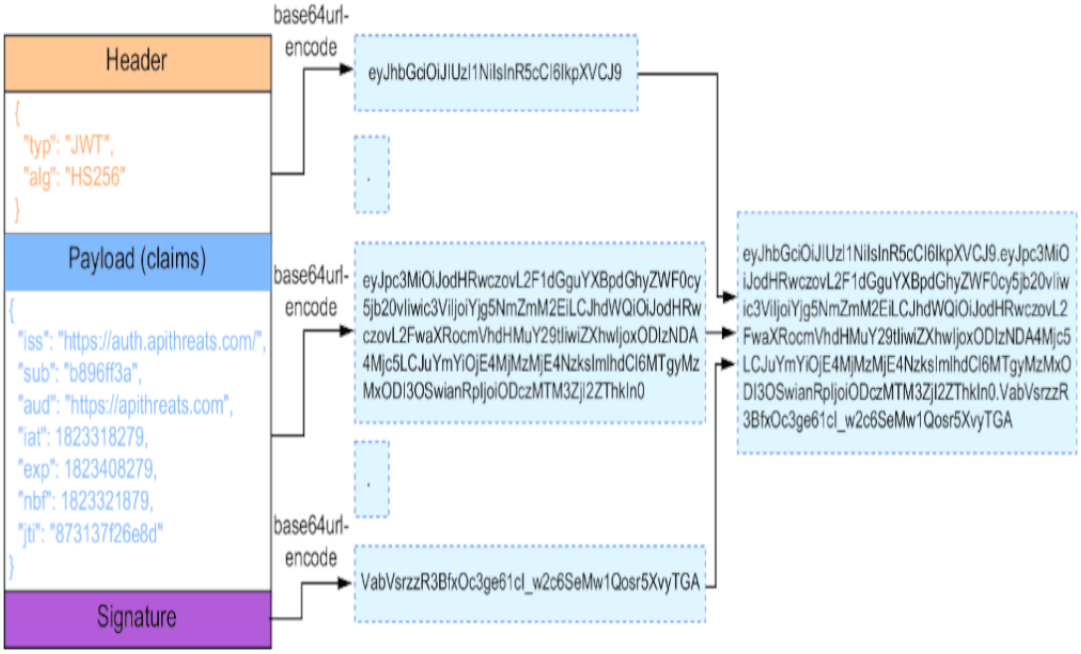


FIGURA 7.2: Estructura de un JWT.

7.6.1. Header

El encabezado de un JWT se conoce como encabezado JOSE (*JSON Object Signing and Encryption*, por sus siglas en inglés). Incluye metadatos sobre el token, como el algoritmo de firma y, opcionalmente, un identificador de clave. Los parámetros más habituales del encabezado JOSE son `typ`, `alg` y `kid`:

- `typ`: indica el tipo de token (en este caso, JWT).
- `alg`: especifica el algoritmo utilizado para firmar el token.
- `kid`: identifica la clave utilizada para firmar el token (útil cuando se emplean varias claves).

Un ejemplo de encabezado JOSE sería:

```
"typ": "JWT",
"alg": "RS256",
"kid": "080E8CtRFAnnLlgK3dk8Y"
```

7.6.2. Payload

El siguiente elemento es la *carga útil* (*payload*). Contiene *claims* (declaraciones) sobre el sujeto del token, que puede ser un usuario o una identidad no humana (por ejemplo, un servicio). En el caso de *ID tokens*, la carga útil suele contener información identificativa del usuario, mientras que en *access tokens* incluye *claims* sobre los permisos para acceder a una API.

La RFC 7519 [IET15b] define siete *claims* registradas:

- `iss` (*issuer*) — identifica el emisor del token.
- `sub` (*subject*) — identifica el sujeto al que pertenece el token.

- `aud` (*audience*) — identifica la audiencia prevista (por ejemplo, la API destinataria).
- `exp` (*expiration*) — instante de expiración del token.
- `nbf` (*not before*) — instante a partir del cual el token es válido.
- `iat` (*issued at time*) — instante de emisión del token.
- `jti` (*JWT ID*) — identificador único del token.

Estas *claims* no son obligatorias, por lo que es posible emitir tokens sin ellas. Aun así, suele considerarse una buena práctica incluirlas cuando aplica, porque mejora la interoperabilidad y facilita la validación por parte de bibliotecas y componentes estándar (por ejemplo, *API gateways*), que ya esperan este conjunto de campos según la RFC 7519 [IET15b].

7.6.3. Signature

El elemento final es la firma. El proceso de firma se describe en la RFC 7515 [IET15a]. La firma se obtiene aplicando un algoritmo criptográfico sobre la cabecera y la carga útil, aportando integridad y autenticidad (es decir, permite comprobar que el token no ha sido modificado). En términos generales, la firma se calcula a partir de la cabecera y el *payload* codificados en `base64url`, concatenados y separados por un punto.

Capítulo 8

Diseño final de la base de datos

En este capítulo se presenta el diseño final de la base de datos, incluyendo todas las tablas, sus atributos y las relaciones entre ellas. Asimismo, se documentan los procedimientos almacenados y las funciones desarrolladas para dar soporte a la lógica de acceso a datos. Con respecto al diseño inicial, se han introducido varios cambios orientados a mejorar la eficiencia, la mantenibilidad y la integridad de la información.

8.1. Tablas

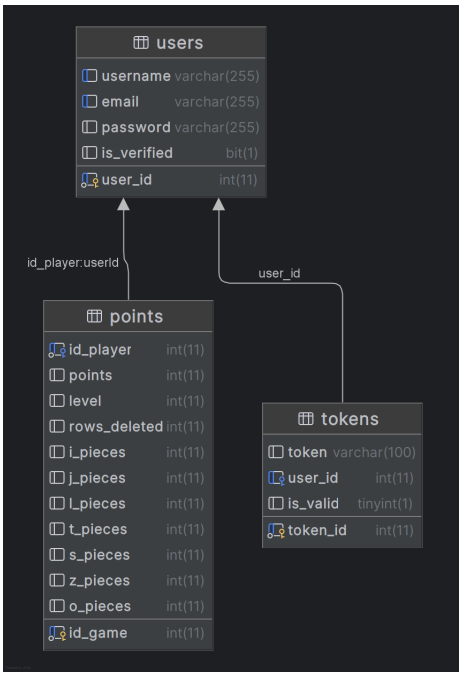


FIGURA 8.1: Diagrama final de la base de datos

En la Figura 8.1 se muestran las tablas de la base de datos, junto con sus atributos y las relaciones definidas entre ellas. A continuación, se detallan los procedimientos almacenados y las funciones creadas para implementar las operaciones principales del sistema.

8.2. Procedimientos almacenados y funciones

FIGURA 8.2: Procedimientos almacenados y funciones

De forma general, las funciones almacenadas se emplean cuando se requiere devolver un valor (por ejemplo, un indicador *booleano*), mientras que los procedimientos almacenados suelen utilizarse para ejecutar operaciones que no devuelven un valor único o que pueden retornar un conjunto de resultados. En este proyecto se han utilizado procedimientos almacenados para operaciones de inserción, actualización y eliminación, y funciones para comprobaciones que devuelven verdadero o falso.

La única excepción es la función `fn_verify_email_token`, cuyo comportamiento combina validación y actualización: si el token es válido, devuelve `1` y actualiza el campo `email_verified` a `true`; en caso contrario, devuelve `0` sin modificar ningún dato.

En cuanto a la nomenclatura, se ha seguido una convención que facilita identificar de inmediato el propósito y el tipo de cada objeto. Dado que una función o un procedimiento almacenado ejecuta una acción concreta, su nombre comienza con un verbo que describe dicha acción y va precedido por un prefijo que indica su naturaleza (`fn_` para funciones y `pr_` para procedimientos). Esta aproximación, similar a la notación húngara, resulta útil en este contexto para diferenciar rápidamente entre ambos tipos. Por ejemplo, `pr_insert_user` indica un procedimiento almacenado destinado a insertar un usuario, mientras que `fn_verify_email_token` identifica una función que valida un token de verificación de correo.

8.3. Obtención de datos desde la base de datos en el servicio REST

Al comenzar el desarrollo del servicio REST, fue necesario definir un enfoque para acceder a la base de datos de forma eficiente y segura. En ese momento existían dos alternativas principales: incorporar sentencias SQL directamente en el código del servicio o delegar dichas operaciones en procedimientos almacenados. Finalmente, se optó por la segunda opción por varias razones.

En primer lugar, los procedimientos almacenados permiten encapsular la lógica de acceso a datos en la propia base de datos, lo que mejora la seguridad y la mantenibilidad del sistema. Además, este enfoque reduce el acoplamiento del servicio web

con el esquema físico, evitando depender directamente de los nombres concretos de las tablas. En segundo lugar, los procedimientos y funciones almacenadas pueden beneficiarse de optimizaciones del motor de la base de datos, lo que puede traducirse en un mejor rendimiento. Por último, al poder reutilizarse desde distintos puntos del servicio REST, se reduce la duplicación de código y se facilita el mantenimiento evolutivo.

8.3.1. Insertar sentencias SQL directamente en el servicio REST

FIGURA 8.3: Sentencias SQL embebidas directamente en el servicio REST

8.3.2. Utilizar procedimientos y funciones almacenadas

FIGURA 8.4: Acceso a datos mediante procedimientos y funciones almacenadas

Capítulo 9

Diseño final de la REST API

En este capítulo se presenta el diseño final de la API REST, incluyendo todas las rutas, sus métodos HTTP y los datos que se envían y reciben. Asimismo, se describen los controladores y servicios implementados para dar soporte a la API. Con respecto al diseño inicial, se han introducido diversos cambios orientados a mejorar la eficiencia y reforzar la seguridad.

La API REST se ha diseñado siguiendo los principios REST, utilizando el método HTTP adecuado en cada operación y definiendo rutas claras y descriptivas para cada recurso.

Además, se han aplicado medidas de seguridad destinadas a mitigar ataques habituales, como la inyección SQL, el *cross-site scripting* (XSS) y el *cross-site request forgery* (CSRF). Aunque podría haberse incorporado un esquema adicional de autenticación y autorización para restringir el acceso a determinadas rutas, se ha optado por no hacerlo con el objetivo de simplificar el alcance del proyecto.

En cuanto a la tecnología empleada, podría haberse utilizado cualquier *framework* de *backend*; sin embargo, se ha elegido .NET Core 8 por ser la plataforma con la que tengo mayor experiencia y porque permite desarrollar la API de forma rápida y eficiente. Además, .NET Core es multiplataforma, lo que facilita el despliegue en distintos sistemas operativos, como Windows, Linux y macOS, y en diferentes entornos de ejecución.

Por último, la consulta de recursos especializados, como *Web API Development with ASP.NET Core 8* [Yan24], ha sido clave para consolidar conceptos y adoptar buenas prácticas en el desarrollo de APIs con .NET. Una parte importante de las recomendaciones del libro se ha aplicado en el diseño y la implementación de la API, lo que ha contribuido a mejorar la calidad y la eficiencia del proyecto. Gran parte de la documentación incluida en este capítulo se apoya en dicho recurso, si bien se ha adaptado a las necesidades y requisitos específicos del proyecto.

9.1. Diferentes metodos para llamar a la REST API

HTTP method	Safe	Idempotent	Common operations
GET	Yes	Yes	Read, list, view, search, show, and retrieve
HEAD	Yes	Yes	HEAD is used to check the availability of a resource without actually downloading it.
OPTIONS	Yes	Yes	OPTIONS is used to retrieve the available HTTP methods for a given resource.
TRACE	Yes	Yes	TRACE is used to get diagnostic information about a request/response cycle.
PUT	No	Yes	Update and replace
DELETE	No	Yes	Delete, remove, and clear
POST	No	No	Create, add, and update
PATCH	No	No	Update

CUADRO 9.1: Los diferentes metodos HTTP y sus propiedades de seguridad e idempotencia

9.2. Swagger

9.2.1. Historia y que es

Swagger es un conjunto de herramientas para el diseño, la documentación y la prueba de APIs, basado en la especificación OpenAPI. Permite describir de forma estandarizada los endpoints, parámetros, cuerpos de petición, respuestas y esquemas de datos. A partir de esa descripción, genera documentación interactiva que facilita la exploración y validación de la API. Además, ofrece utilidades para editar la especificación, simular llamadas y automatizar flujos de desarrollo e integración. En conjunto, mejora la comunicación entre equipos y contribuye a la mantenibilidad y consistencia del ciclo de vida de la API. [Wik]

9.2.2. Configurar Swagger en la REST API

Solo hace falta escribir el siguiente comando para crear un nuevo proyecto Web API con .NET Core 8 y configurar Swagger de forma automática.

```
dotnet new webapi -n EnvironmentDemo -controllers
```

9.3. Vistazo final de como han quedado los *endpoints* de la REST API

En las Figuras 9.1 y 9.2 se presenta el diseño final de los *endpoints* de la API REST, incluyendo sus rutas, métodos HTTP y los datos que se envían y reciben.

FIGURA 9.1: Diseño final de los *endpoints* de la API REST (I)

Debido a limitaciones de espacio, el resto de *endpoints* se muestra en la Figura 9.2.

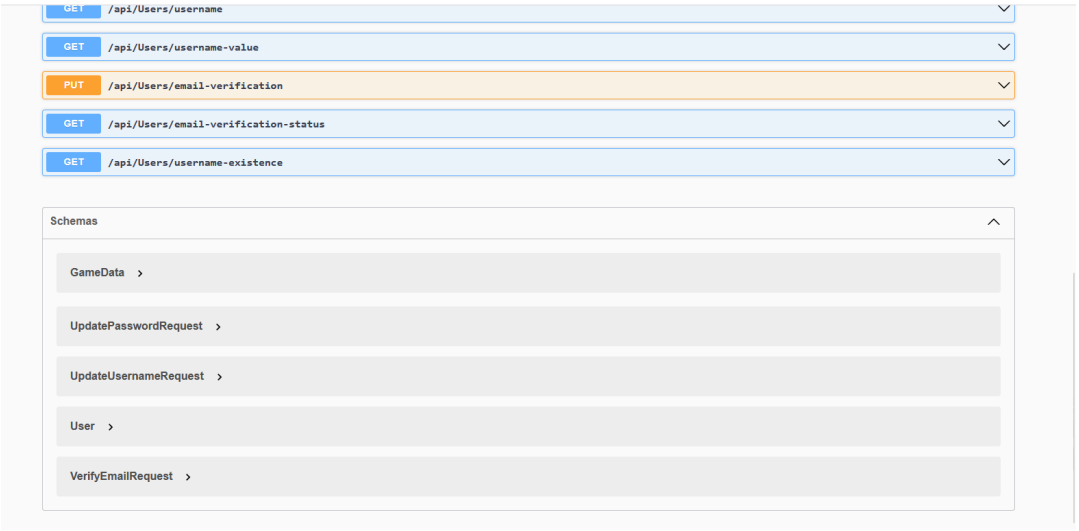


FIGURA 9.2: Diseño final de los *endpoints* de la API REST (II)

Capítulo 10

Errores que cometi durante el desarrollo del proyecto

En este capítulo expongo algunos de los errores que cometí durante el desarrollo del proyecto y explico cómo los solucioné. Aunque no se trata de un apartado especialmente técnico, considero importante documentar estos fallos y su resolución, ya que pueden ayudar a otros desarrolladores a evitar problemas similares y a aprender de mi experiencia.

Además, incluir estas incidencias permite comprender mejor el proceso de desarrollo y valorar el esfuerzo, la dedicación y los inevitables "dolores de cabeza" que conlleva sacar adelante un proyecto de este tipo.

10.1. No llamar de manera correcta a los *endpoints* de la REST API desde el cliente

En este caso, el cliente no es una aplicación de escritorio en .NET ni una aplicación móvil, sino un videojuego desarrollado con Unity.

Al principio no tenía claro cómo consumir los *endpoints* de la API REST desde el cliente y pensé que bastaba con realizar una petición HTTP a la ruta correspondiente. Sin embargo, pronto comprobé que el proceso era más complejo: además del formato de los datos enviados y recibidos, era necesario contemplar el tratamiento de errores, la autenticación y las particularidades de Unity a la hora de realizar solicitudes HTTP, entre otros aspectos.

A continuación, presento dos fragmentos de código: el primero muestra un enfoque incorrecto, planteado como si se tratara de una aplicación de escritorio; el segundo refleja la implementación adecuada desde el cliente en Unity.

- Llamada incorrecta desde el cliente, planteada como si se tratara de una aplicación de escritorio.

```
HttpClient client = new HttpClient();
var response = await client.
                    GetAsync("https://localhost:5001/api/users");
if (response.IsSuccessStatusCode)
{
    var users = await response.Content.ReadAsAsync<List<User>>();
    // Hacer algo con los usuarios
}
```

```
else
{
    // Manejar el error
}
```

- Llamada correcta desde el cliente, que es un videojuego desarrollado con Unity

```
using (var req = new UnityWebRequest(authUrl, UnityWebRequest.kHttpVerbPOST))
{
    req.uploadHandler = new UploadHandlerRaw(Encoding.UTF8.GetBytes(json));
    req.downloadHandler = new DownloadHandlerBuffer();
    req.timeout = 15;

    req.SetRequestHeader("Content-Type", "application/json");
    req.SetRequestHeader("Accept", "text/plain");

    // Pinning as you had it (optional)
    req.certificateHandler =
        new PinnedCertHandler(config.certificatePinSha256);
    req.disposeCertificateHandlerOnDispose = true;

    yield return req.SendWebRequest();

    if (req.result != UnityWebRequest.Result.Success)
    {
        var headers = req.GetResponseHeaders();
        var headersText = headers == null
            ? "(none)"
            : string.Join("\n", System.Linq.Enumerable.
            Select(headers, kv => $"{kv.Key}: {kv.Value}"));

        var msg =
            $"HTTP {req.responseCode} {req.error}\n";

        onError?.Invoke(new AuthenticationException(msg));
        yield break;
    }

    var body = req.downloadHandler.text;
    var tokenResponse = JsonConvert.DeserializeObject
    <TokenResponseDto>(body);

    onSuccessToken?.Invoke(tokenResponse.Token);
}
}
```

10.2. Desalineación entre los nombres de los parámetros (cliente y API)

En este caso, el error que cometí fue que los nombres de los parámetros enviados desde el cliente no coincidían con los que esperaba el *endpoint* de la API REST.

Para evitar este tipo de problemas, es fundamental que los DTO (*data transfer object*) utilizados para enviar y recibir datos desde el cliente mantengan la misma nomenclatura de propiedades que el modelo que consume el *endpoint* (ya sea en el cuerpo de la petición, en los parámetros de consulta o en las cabeceras, según corresponda). Además, en *Swagger* [Wik] se muestran de forma clara los nombres y la estructura de los datos esperados en cada *endpoint*: si no hay correspondencia con las propiedades del DTO, la serialización no se realiza como se espera, el servidor recibe valores nulos o incompletos y, en consecuencia, el *endpoint* devuelve un error.

Una herramienta que me resultó especialmente útil para identificar y solucionar este problema fue *Postman* [Pos], ya que me permitió enviar peticiones HTTP directamente a los *endpoints* y analizar tanto el contenido de las solicitudes como las respuestas del servidor, comprobando así si los nombres de los parámetros estaban correctamente alineados.

10.2.1. Pensar que a la API siempre se le van a enviar los datos correctos

En este caso, el error que cometi fue pensar que a la API siempre se le van a enviar los datos correctos, y no tener en cuenta el manejo de errores. Para solucionar este error, decidi implementar un manejo de errores adecuado en la REST API,

10.3. No utilizar procedimientos almacenados para acceder a la base de datos

En este caso, el error que cometi fue no utilizar procedimientos almacenados para acceder a la base de datos, y en su lugar, insertar codigo SQL directamente en el servicio REST. Esto no solo es una mala practica, sino que tambien es un error de seguridad, ya que puede permitir ataques de inyección SQL, y tambien puede hacer que el servicio REST quede acoplado al nombre que tienen las tablas en la base de datos, lo que dificulta el mantenimiento y la escalabilidad del servicio REST. Para solucionar este error, decidi utilizar procedimientos almacenados para acceder a la base de datos, y asi encapsular la lógica de acceso a datos en la base de datos, mejorar la seguridad y la mantenibilidad del código, y evitar que el servicio REST quede acoplado al nombre que tienen las tablas en la base de datos.

10.4. Unity no detectaba el paquete Newtonsoft.Json

En este caso, el error que cometí fue no configurar correctamente el paquete Newtonsoft.Json en Unity. Inicialmente, lo instalaba desde el IDE (en mi caso, Rider). Sin embargo, al cerrar el IDE y volver a abrir Unity, el paquete no quedaba disponible en el proyecto y aparecía un error indicando que no se encontraba el *namespace* Newtonsoft.Json.

Para solucionarlo, configuré el paquete correctamente desde Unity. Para ello, accedí a `Edit >Project Settings >Preferences` y, en la sección de generación de archivos `.csproj`, habilité la opción correspondiente a los paquetes del registro (`Registry packages`). A continuación, busqué el paquete por su nombre completo para que Unity lo detectara e incorporara correctamente al proyecto.

De este modo, el paquete pasó a estar disponible y el error del *namespace* dejó de producirse.

10.5. Acoplar la configuración de los endpoints al código del cliente

En este caso, el error que cometí fue definir la URL de la API REST de forma estática en el cliente, es decir, escribirla directamente en el código. Esta práctica deja el cliente innecesariamente acoplado a la dirección del servicio y dificulta tanto su mantenimiento como su evolución (por ejemplo, al cambiar de entorno, dominio o puerto).

Para solucionarlo, opté por externalizar esa configuración en un archivo específico, de modo que la URL de la API REST pueda modificarse sin tocar el código del cliente. Con ello se reduce el acoplamiento, se simplifica la gestión de entornos y se mejora la mantenibilidad del proyecto.

Capítulo 11

Conclusión

En este capítulo se muestra el resumen de los objetivos alcanzados y los objetivos no alcanzados.

11.1. Resumen de los objetivos alcanzados

En este proyecto se han alcanzado los siguientes objetivos:

- Diseñar y desarrollar una base de datos relacional para almacenar la información de los usuarios, los juegos, las partidas, etc.
- Diseñar y desarrollar un servicio RESTful para exponer la funcionalidad de la base de datos a través de una API.
- Diseñar y desarrollar un cliente para consumir la API del servicio RESTful, en este caso un videojuego desarrollado con Unity.
- Implementar medidas de seguridad para proteger la información de los usuarios y evitar ataques de seguridad.

11.2. Resumen de los objetivos no alcanzados

En el proyecto no se han alcanzado los siguientes objetivos:

- Introducir una bonificacion por rachas de filas eliminadas consecutivamente.
- Implementar un sistema de logros y recompensas para incentivar a los jugadores a seguir jugando y mejorar su rendimiento.
- Implementar un sistema de clasificación global para que los jugadores puedan comparar su rendimiento con el de otros jugadores de todo el mundo.
- Que el videojuego tenga una version multijugador.
- Implementar un sistema de pago para comprar juegos y contenido adicional.
- Implementar un sistema de recomendaciones de juegos basado en el historial de partidas y las preferencias de los usuarios.

Capítulo 12

Opinión final

El proyecto ha sido un éxito, ya que se ha cumplido la mayor parte de los objetivos planteados y se han superado prácticamente todos los retos y dificultades surgidas durante su desarrollo. Aunque no se ha alcanzado el objetivo de incluir una versión multijugador ni se han implementado un sistema de pago y un sistema de recomendaciones de juegos, considero que los objetivos principales del proyecto se han logrado de forma satisfactoria.

A pesar de los contratiempos y los "dolores de cabeza" que han acompañado el proceso, la experiencia ha sido muy enriquecedora y gratificante, porque me ha permitido aprender numerosas cosas nuevas. Además, he tenido la oportunidad de aplicar los conocimientos adquiridos a lo largo de mi trayectoria académica y de enfrentarme a problemas reales, similares a los que aparecen en un proyecto profesional.

Por todo ello, es el proyecto más grande y complejo que he desarrollado jamás, y también del que me siento más orgulloso.

Bibliografía

[Fie00]
Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. 2000. URL: https://ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[Gam+94]
Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, oct. de 1994, pág. 395. ISBN: 978-0-201-63361-0.

[God25a]
Godot Documentation. *Complying with licenses*. Godot Engine. 2025. URL: https://docs.godotengine.org/en/4.4/about/complying_with_licenses.html (visitado 03-10-2025).

[God25b]
Godot Documentation. *Exporting for the Web*. Godot Engine. 2025. URL: https://docs.godotengine.org/en/latest/tutorials/export/exporting_for_web.html (visitado 03-10-2025).

[God25c]
Godot Documentation. *TileMap — Class Reference*. Godot Engine. 2025. URL: https://docs.godotengine.org/en/4.4/classes/class_tilemap.html (visitado 03-10-2025).

[God25d]
Godot Documentation. *TileSet — Class Reference*. Godot Engine. 2025. URL: https://docs.godotengine.org/en/4.4/classes/class_tileset.html (visitado 03-10-2025).

[God25e]
Godot Documentation. *Using TileMaps*. Godot Engine. 2025. URL: https://docs.godotengine.org/en/latest/tutorials/2d/using_tilemaps.html (visitado 03-10-2025).

[God25f]
Godot Documentation. *Using TileSets*. Godot Engine. 2025. URL: https://docs.godotengine.org/en/stable/tutorials/2d/using_tilesets.html (visitado 03-10-2025).

[God25g]
Godot Engine. *Godot Engine — GitHub repository*. Repository README and license note. 2025. URL: https://github.com/godotengine/godot (visitado 03-10-2025).

[God25h]
Godot Engine. *License*. Godot is released under the MIT License. 2025. URL: https://godotengine.org/license/ (visitado 03-10-2025).

[HT19]
Andrew Hunt y David Thomas. *The Pragmatic Programmer: Your Journey to Mastery*. 20th Anniversary Edition. Addison-Wesley Professional, 2019. ISBN: 9780135957059.

[IET15a]
IETF JWT Working Group. *RFC 7515: JSON Web Signature (JWS)*. DOI: 10.17487/RFC7515. 2015. URL: https://datatracker.ietf.org/doc/html/rfc7515 (visitado 10-11-2025).

[IET15b]
IETF JWT Working Group. *RFC 7519: JSON Web Token (JWT)*. DOI: 10.17487/RFC7519. 2015. URL: https://datatracker.ietf.org/doc/html/rfc7519 (visitado 10-11-2025).

[Mar25]
Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, sep. de 2025, pág. 672. ISBN: 978-0-135-39858-6.

[Pha25a]
Phaser Documentation. *Arcade Physics — Concepts*. Phaser Studio Inc. 2025. URL: https://docs.phaser.io/phaser/concepts/physics/arcade (visitado 03-10-2025).

[Pha25b]
Phaser Documentation. *Matter Physics — Concepts*. Phaser Studio Inc. 2025. URL: https://docs.phaser.io/phaser/concepts/physics/matter (visitado 03-10-2025).

[Pha25c]
Phaser Documentation. *Phaser 3 API Documentation*. Phaser Studio Inc. 2025. URL: https://docs.phaser.io/api-documentation/api-documentation (visitado 03-10-2025).

[Pha25d]
Phaser Documentation. *Phaser Tilemaps API*. Phaser Studio Inc. 2025. URL: https://docs.phaser.io/api-documentation/class/tilemaps-tilemap (visitado 03-10-2025).

[Pha25e]
Phaser Studio Inc. *The MIT License (MIT) — Phaser*. Phaser is distributed under the MIT License. 2025. URL: https://phaser.io/download/license (visitado 03-10-2025).

[Pos]
Postman, Inc. *Postman: The World's Leading API Platform*. URL: https://www.postman.com/ (visitado 19-02-2026).

[Uni25a]
Unity Technologies. *2D game development*. Unity. 2025. URL: https://docs.unity3d.com/6000.2/Documentation/Manual/Unity2D.html (visitado 03-10-2025).

[Uni25b]
Unity Technologies. *Cinemachine — 2D graphics*. Unity. 2025. URL: https://docs.unity3d.com/Packages/com.unity.cinemachine@2.8/manual/Cinemachine2D.html (visitado 03-10-2025).

[Uni25c]
Unity Technologies. *Cinemachine — Package Documentation*. Unity. 2025. URL: https://docs.unity3d.com/Packages/com.unity.cinemachine@2.3/ (visitado 03-10-2025).

[Uni25d]
Unity Technologies. *Deploy a Web application (WebGL)*. Compression methods (gzip/Brotli) and server configuration. Unity. 2025. URL: https://docs.unity3d.com/6000.2/Documentation/Manual/webgl-deploying.html (visitado 03-10-2025).

[Uni25e]
Unity Technologies. *Get started with 2D game development*. Unity. 2025. URL: https://docs.unity3d.com/6000.2/Documentation/Manual/2d-game-development-landing.html (visitado 03-10-2025).

[Uni25f]
Unity Technologies. *Input System — Package Documentation*. Unity. 2025. URL: https://docs.unity3d.com/Packages/com.unity.inputsystem@latest/ (visitado 03-10-2025).

[Uni25g]
Unity Technologies. *Rule Tile (2D Tilemap Extras)*. Unity. 2025. URL: https://docs.unity3d.com/Packages/com.unity.2d.tilemap.extras@1.6/manual/RuleTile.html (visitado 05-10-2025).

[Uni25h]
Unity Technologies. *Tilemap Collider 2D*. Unity. 2025. URL: https://docs.unity3d.com/2023.1/Documentation/Manual/class-TilemapCollider2D.html (visitado 05-10-2025).

[Uni25i]
Unity Technologies. *Tilemaps*. Unity. 2025. URL: https://docs.unity3d.com/6000.2/Documentation/Manual/tilemaps/tilemaps-landing.html (visitado 05-10-2025).

[Wik]
Wikipedia contributors. *Swagger (software)*. Wikipedia. Last edited 15 February 2026. URL: https://en.wikipedia.org/wiki/Swagger_(software) (visitado 17-02-2026).

[Wik25]
Wikipedia contributors. *Tetrominó*. Wikipedia, la enciclopedia libre. 22 de mayo de 2025. URL: https://es.wikipedia.org/wiki/Tetromin%C3%B3 (visitado 10-11-2025).

[Yan24]
Xiaodi Yan. *Web API Development with ASP.NET Core 8: Learn techniques, patterns, and tools for building high-performance, robust, and scalable web APIs*. Packt Publishing, abr. de 2024, pág. 804. ISBN: 978-1804610954.