Videojuego de ajedrez con modo de juego contra la máquina

Anexo V: Manual del programador

Trabajo de Fin de Grado

GRADO EN INGENIERÍA INFORMÁTICA

Junio de 2025

Autor

Óscar Sánchez Rubio

Tutores

Luis Augusto Silva Zendron

Gabriel Villarrubia González

Índice

Contenidos

Tablas

1. Introducción

Este manual del programador ha sido creado con el objetivo de proporcionar una guía clara y detallada para el entendimiento del desarrollo del código creado para el proyecto, En él se incluyen las características principales del sistema, la estructura del código, las convenciones de programación utilizadas, y ejemplos prácticos que facilitan la comprensión y el trabajo con el código fuente.

Este documento se organiza en los siguientes apartados:

- **Bibliotecas externas**: Se enumeran y explican las principales dependencias externas utilizadas en el proyecto, incluyendo sus propósitos y cómo han sido integradas en el motor.

- **Paradigma de programación**: Se analiza el enfoque de diseño adoptado para el desarrollo del software, haciendo hincapié en el paradigma de programación principal (como programación orientada a objetos, funcional o estructurada) y justificando su elección en el contexto del motor de ajedrez.

- **Estructura del código**: Se describe la organización general del proyecto, indicando los módulos o paquetes más relevantes, su jerarquía, y las responsabilidades principales de cada uno.

- **Funciones principales**: Se detallan las funciones o métodos clave que constituyen la lógica del motor, explicando su funcionamiento, entrada y salida de datos, así como su interacción con otras partes del sistema.

Está manual también puede estar dirigido a desarrolladores que participen en el proyecto, así como a futuros colaboradores que deseen entender su funcionamiento interno para aportar mejoras o corregir errores.

2. Bibliotecas externas

2.1. TextMeshPro

TextMeshPro es una biblioteca de Unity que ofrece un sistema avanzado de renderizado de texto. A diferencia del sistema estándar de texto de Unity, *TextMeshPro* proporciona una mayor calidad visual, mejor control sobre el estilo y formato del texto, y opciones avanzadas como efectos de sombreado, contornos y mapas de bits de alta resolución.

Esta biblioteca es especialmente útil para proyectos que requieren interfaces de usuario con texto claro y estéticamente atractivo, ya que permite manipular fuentes y estilos con precisión, mejorar la legibilidad y optimizar el rendimiento en la representación de texto.

En este proyecto, TextMeshPro se utiliza para mostrar la totalidad del texto que aparece en las interfaces de las escenas.

2.2. New Input System Package

Se trata de una biblioteca oficial de Unity que ofrece un sistema avanzado y flexible para gestionar las entradas del usuario. Sustituye al sistema de entrada tradicional, permitiendo un manejo más potente y compatible con una enorme cantidad de teclados, ratones, etc.

En este proyecto, el *New Input System Package* se emplea para controlar todas las interacciones del usuario, asegurando una configuración flexible y una respuesta eficiente ante distintos dispositivos de entrada.

3. Paradigma de programación

Para este proyecto, se ha utilizado una combinación del paradigma de programación orientada a objetos (OOP), en combinación del ***data-oriented design***. Este *data-oriented design*, aunque no puede no ser considerado del todo un paradigma de programación, al ser compatible con otros, se trata de un acercamiento de programación más óptimo, a comparación del clásico OOP (Data-Oriented Design (or why you might be shooting yourself in the foot with OOP), 2009).

Debido al gran uso de memoria empleado durante la búsqueda de movimientos, ha sido necesario para este proyecto, ha sido necesario aplicar el *data-oriented design* a ciertas estructuras como han sido los movimientos. Para representar los movimientos internamente en la aplicación, necesitamos los siguientes campos:

- Casilla de origen: Un campo de al menos 6 bits.
- Casilla de destino: Un campo de al menos 6 bits.
- *Flag* de movimiento: En esta ocasión se ha contado con un total de 8 *flags*, es decir, al menos se necesitan 3 bits.

A simple vista, la mejor opción para crear una clase en la haya 3 variables, una para cada campo. Sin embargo, esto es bastante ineficiente a nivel de memoria, puesto que el espacio en memoria que menos puede ocupar una variable en C# es de 8 bits, lo que su pone un mínimo de 24 bits aplicando esta solución. Si pensamos en asignar una variable como un *ushort*, el cual ocupa 16 bits en memoria, y asociamos ciertos bits de esta variable a los campos antes mencionados, podemos optimizar la representación:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Origen						Destino						Flag			

Tabla 1: Representación interna de los movimientos

Esto es posible gracias a operaciones con máscaras y desplazamientos a nivel de bit, que permiten elaborar una *setters* y *getters* que requieren de más tiempo de ejecución al tener que "*extraer*" el valor de los campos de la variable, pero a cambio de una considerable reducción del uso de memoria.

4. Estructura del código

La estructura del código de la aplicación se ha organizado siguiendo el uso de ***namespaces***, lo que facilita la mantenibilidad, escalabilidad y comprensión del proyecto. Cada *namespace* agrupa clases con responsabilidades claramente definidas dentro del sistema, y relacionadas con los subsistemas identificados durante la disciplina de diseño. A continuación, se describen los principales espacios de nombres utilizados:

- **Ajedrez.Core**: Este *namespace* contiene las clases entidad fundamentales del juego, que representan los elementos esenciales del dominio del ajedrez. Entre ellas se incluyen:
    - **Jugador**: Clase abstracta que representa a cada jugador de la partida.
    - **JugadorHumano**: Clase que hereda de Jugador y que representa a un jugador manejado por el usuario.
    - **JugadorIA**: Clase que hereda de Jugador y que representa a un jugador manejado por el ordenador. Dentro de esta clase, se hallan métodos que hacen posible la búsqueda y la gestión del tiempo de búsqueda.
    - **Tablero**: Clase que modela el tablero, con su distribución de las piezas mediante *bitboards*. Además, contiene métodos tan importantes como *HacerMovimiento()*, *DeshacerMovimiento* y *GenerarMovimientosLegales()*.
    - **EstadoTablero**: Estructura que modela el estado del tablero, necesaria para poder deshacer movimientos durante la búsqueda.
    - **Movimiento**: Estructura encargada de describe la información necesaria de los movimientos de la partida.
    - **Pieza**: Estructura que representa una pieza mediante su tipo y su color.
    - **Partida**: Clase que almacena todos los parámetros ajustables de una partida, como son: la duración, el incremento de tiempo, el tipo de jugadores, etc.
    - **PilaRepeticiones**: Se trata de una clase que modela una implementación especial de una pila dedicada a almacenar posiciones repetidas durante la partida. El hecho de que sea una implementación especial es necesario para poder hallar posiciones repetidas durante el proceso de búsqueda.
- **Ajedrez.Managers**: Incluye las clases responsables del control y coordinación del flujo del programa, comunicando las clases de la interfaz con las de la entidad. Esencialmente, se ha creado una clase *manager* para cada escena:
    - **PartidaManager**: Controla todos los aspectos relativos al flujo durante la partida. Es la encargada de comunicar los movimientos elegidos por el usuario, a las clases entidad del programa.
    - **ConfiguracionesPreviasManager**: Se encarga de instanciar la clase partida en base a las opciones elegidas por el usuario como configuraciones previas.

- **Ajedrez.UI**: Contiene las clases encargadas de la interacción con el usuario. En este espacio de nombres se agrupan los métodos que generan y gestionan la interfaz gráfica o textual, permitiendo desacoplar la lógica del motor del funcionamiento de dicha interfaz. Las clases que conforman este *namespace* son:

    - **InputManager**: Clase encargada de recibir las entradas del ratón del usuario para poder mover las piezas del tablero durante la partida.

    - **TableroUI**: Representación visual del tablero con la que se comunican aquellos componentes que quieren efectuar un movimiento en la interfaz del usuario.

    - **PartidaUI**: Interfaz encargada de mostrar el tiempo restante de cada jugador, así como sus nombres durante la partida, e informa de la situación del tablero (jaque, jaque mata, rey ahogado, etc.).

    - **PromociónUI**: Clase que gestiona el panel de promoción de ambos jugadores. Haciendo comunicar la elección del usuario de la clase PartidaManager.

    - **ConfiguracionesPreviasUI**: Gestiona los elementos del *canvas*, pertenecientes a la interfaz de las configuraciones previas. Comunica los valores establecidos por el usuario a ConfiguracionesPreviasManager.

    - **MenuPrincipalUI**: Gestiona los elementos del *canvas*, perteneciones a la interfaz del menú principal.

- **Ajedrez.IA**: Reúne las clases que implementan la lógica de la inteligencia artificial del motor. Aquí se encuentran los algoritmos de evaluación, búsqueda y toma de decisiones que permiten al motor jugar contra un oponente humano u otro motor:

    - **Busqueda**: Encargada de implementar el algoritmo de búsqueda.

    - **Evaluación**: Elabora las tareas de evaluación del tablero durante la búsqueda.

    - **GestorTiempo**: Clase que implementa los métodos necesarios para poder conocer el tiempo que será invertido durante la búsqueda.

    - **LibroAperturas**: Clase que maneja la elección de los movimientos de apertura en base a un fichero *Aperturas.txt*.

    - **MapasPiezas**: Clase que implementa toda la funcionalidad relativa a los *piece-square tables*.

    - **OrdenadorMovimientos**: Se encarga de ordenar una lista de movimientos, en base a una puntuación estimada de los mismos.

    - **TablaTransposición**: Implementa la funcionalidad de las tablas de transposición.

- **Ajedrez.Debugging:** Este espacio de nombres fue creado para facilitar el proceso de depuración del proyecto durante su desarrollo, debido a que fue necesaria la creación de herramientas particulares especializadas en la depuración. Contiene clases, métodos y utilidades diseñadas para el análisis del comportamiento del programa, la detección

de errores y la validación del flujo lógico de las operaciones internas. Los métodos de estas clases son llamados desde el InputManager:

  - **BitboardDebugger**: Clase encargada de la depuración de los bitboards.
  - **BusquedaDebugger**: Clase encargada de la depuración de la búsqueda.
  - **EstadoDebugger**: Clase encargada de la depuración de la clase EstadoTablero.

- ***Ajedrez.Utils***: Este *namespace* agrupa un conjunto de clases y métodos auxiliares que ofrecen funcionalidades de apoyo al resto del sistema. Se trata de utilidades generales que no encajan directamente en los otros espacios de nombres por no formar parte del núcleo del dominio ni de los controladores principales, pero que resultan esenciales para casi todas las clases del código. Las clases que lo componen son:
  - **AjedrezUtilities**: Se refiere a utilidades generales del tablero, como la conversión de coordenadas del tablero a índice.
  - **BitboardUtilities**: Se tratan de funcionalidades relativas a los *bitboards*.
  - **ZobristHashing**: Utilidades relacionadas con el uso de *Zobrist hashes*.
- ***Ajedrez.Systems***: Este *namespace* contiene clases fundamentales que no forman parte directa del dominio del ajedrez, pero que son componentes clave para ofrecer una experiencia de usuario completa y funcional. Las dos clases que lo conforman son:
  - **AudioSystem**: Encargada de la gestión del sonido dentro de la aplicación.
  - **PersistenciaSystem**: Gestiona el almacenamiento y recuperación de datos como las partidas a la hora de ser cargadas con los parámetros de configuración elegidos.

5. Funciones principales

5.1. Alfabeta

Parámetros:

- ***profundidadRestante***: Número de niveles restantes por explorar en el árbol de juego. Actúa como condición de parada.
- ***profundidadDesdeRaiz***: Número de movimientos desde la raíz hasta la posición actual. Se utiliza para ajustar la evaluación en caso de jaque mate (preferir mates rápidos o evitar derrotas tempranas).
- ***alfa***: Mejor valor que el jugador Max (el que inicia la búsqueda, la máquina) puede garantizar hasta el momento.
- ***beta***: Mejor valor que el jugador Min (el oponente) puede forzar.
- **Valor de salida**: Un entero que representa la evaluación numérica de la posición, desde la perspectiva del jugador que mueve.

Esta función implementa la variante **Negamax** (Negamax) con poda Alfa-Beta del clásico Minimax, que simplifica el código asumiendo que ambos jugadores intentan maximizar la misma función de evaluación, pero con signo invertido.

Primero, comprueba si la búsqueda ha sido cancelada o si la posición ya está almacenada en la tabla de transposición, en cuyo caso reutiliza el resultado. Si se alcanza la profundidad máxima, se llama a una búsqueda especializada que sólo explora capturas (búsqueda de quietud (Berent, 2019)).

Si hay movimientos legales disponibles, se exploran recursivamente. Si se encuentra un valor igual o superior a beta, se poda la rama (no se evalúan más movimientos). Si se mejora alfa, se actualiza como mejor valor local. Al final, se guarda la evaluación en la tabla de transposición para futuras consultas.

5.2. ForzarMovimientoReyEndgame (Mop-up)

Parámetros:

- ***color***: Color del jugador que evalúa la posición (blanco o negro).
- ***valorMaterialAliado***: Suma del valor del material del jugador actual.
- ***valorMaterialEnemigo***: Suma del valor del material del oponente.
- ***valorEndgameEnemigo***: Valor de coma flotante comprendido entre 0 y 1 que indica si el oponente se encuentra en el final de partida
- **Valor de salida**: Devuelve una evaluación entera que fomenta maniobras de rey en finales ganados. Si no se cumplen las condiciones, devuelve 0.

Esta función proporciona una bonificación adicional a la evaluación cuando se detecta un final de partida favorable para el jugador actual, provocando el movimiento de rey hacia el centro del tablero y arrinconando al rey enemigo.

Cuando el material aliado supera al enemigo por al menos dos peones (*VALOR_PEON * 2*) y el factor de final del enemigo (*valorEndgameEnemigo*) es positivo, se calcula una bonificación en base a dos criterios estratégicos:

1. Acercamiento del rey aliado al rey rival: Se favorece que el rey del jugador actual se aproxime al del oponente, incentivando maniobras típicas de mate o dominación. Se calcula como la distancia Manhattan entre ambos reyes.

2. Empuje del rey enemigo hacia el borde: Se recompensa que el rey rival se aleje del centro del tablero, algo común en posiciones ganadoras donde se busca arrinconar al rey para forzar el mate.

La suma de ambos factores se multiplica por *valorEndgameEnemigo*, ajustando el peso según lo avanzado que esté el final de partida. Si no se cumplen las condiciones (por ejemplo, si no hay ventaja clara), la función retorna 0.

5.3.CrearZobristHash

Parámetros:

- ***tablero***: Objeto que representa el estado completo del tablero.

- **Valor de salida**: Devuelve un valor un entero sin signo de 64 bits que representa el Zobrist *hash* único de la posición actual en el tablero.

  El hash se construye combinando varias fuentes de información de la posición:

1. Piezas en el tablero: Se recorre cada casilla (0–63), y si contiene una pieza, se aplica una operación XOR con el valor *hash* predefinido correspondiente a ese tipo de pieza en esa casilla.

2. Derechos de enroque: Los 4 derechos de enroque posibles (blanco corto, blanco largo, negro corto, negro largo) se codifican como un entero entre 0 y 15, que se usa como índice para acceder al valor correspondiente en *HashesEnroquesDisponibles*, el cual se incorpora también mediante XOR.

3. Peón al paso: Si hay un peón vulnerable al paso, se añade al hash el valor asociado a su columna.

4. Turno: si el turno es de las negras, se aplica un XOR adicional con *HashTurno*.

El resultado final es un valor hash único (con altísima probabilidad) para cada combinación legal de posición en el tablero. Esta técnica permite comparaciones muy rápidas de posiciones y es esencial para la detección de repeticiones y la reutilización de evaluaciones en la tabla de transposición.

5.4.PrimerBitActivo

Parámetros:

- ***bb***: Un número entero sin signo de 64 bits (ulong) que representa un bitboard, es decir, una codificación de las 64 casillas del tablero de ajedrez donde cada bit indica si una condición se cumple (por ejemplo, presencia de una pieza).

- **Valor de salida**: Devuelve un entero que representa el índice (0–63) del primer bit activo (más bajo) en el *bitboard*. Si no hay ningún bit activo (*bb == 0*), devuelve -1.

La función *PrimerBitActivo* localiza el índice del bit menos significativo activado en un *bitboard* de 64 bits, es decir, la casilla con el bit más bajo que está encendido. Para ello utiliza una técnica eficiente basada en una multiplicación con una constante mágica de **De Bruijn** (De Brujin Sequence), que permite determinar la posición del bit activo sin necesidad de recorrer todos los bits.

Este método combina operaciones bit a bit y un desplazamiento para convertir el bit aislado en un índice válido (0–63), que luego se consulta en una tabla precomputada (Knuth, 2009). Así, la función obtiene rápidamente la posición del primer bit activo, optimizando cálculos comunes en motores de ajedrez que usan *bitboards*.

5.5.ContarBitsActivos

Parámetros:

- ***x***: un número entero sin signo de 64 bits que representa un *bitboard*.

- **Valor de salida**: Devuelve un entero que indica el número total de bits activos en el *bitboard*.

Esta función cuenta de manera eficiente cuántos bits están activos en un entero de 64 bits, operación conocida como ***popcount*** (Portable pop count (hamming weight)). Este algoritmo, en lugar de examinar bit a bit, agrupa y suma las cantidades parciales de bits activos en bloques de 2, luego 4, y finalmente 8 bits. Al final, multiplica y desplaza para combinar estas sumas parciales en el total final.

Este método es muy rápido y no depende de instrucciones especiales de *hardware*, por lo que es muy utilizado en motores de ajedrez basados en *bitboards* para mantener la eficiencia en el cálculo.

6. Bibliografía

Berent, A. (2019). *Quiescence Search and Extensions*. Retrieved from Adam Berent Software & Hobbies: https://adamberent.com/quiescencesearch-andextensions/

Data-Oriented Design (or why you might be shooting yourself in the foot with OOP). (2009). Retrieved from Games from Within: https://gamesfromwithin.com/data-oriented-design

De Brujin Sequence. (n.d.). Chess Programming Wiki. Retrieved from https://www.chessprogramming.org/De_Bruijn_Sequence

Knuth, D. (2009). Bitwise Tricks & Techniques; Binary Decision Diagrams. In *The Art of Computer Programming* (Vol. 4). Addison-Wesley Professional.

Negamax. (n.d.). Chess Programming Wiki. Retrieved from https://www.chessprogramming.org/Negamax

Portable pop count (hamming weight). (n.d.). Unity Discussions. Retrieved from https://discussions.unity.com/t/portable-pop-count-hamming-weight/807752