Universidad San Jorge

Escuela de Arquitectura y Tecnología

Grado en Diseño y Desarrollo de Videojuegos

Proyecto Final

Investigación para el desarrollo de un motor de ajedrez y su integración en un videojuego multiplataforma

Autor del proyecto: Daniel Alexander Gracia Machado

Director del proyecto: Eduardo Jiménez Chapresto

Zaragoza, xx de septiembre de 2022

Este trabajo constituye parte de mi candidatura para la obtención del título de Graduado en Ingeniería Informática por la Universidad San Jorge y no ha sido entregado previamente (o simultáneamente) para la obtención de cualquier otro título.

Este documento es el resultado de mi propio trabajo, excepto donde de otra manera esté indicado y referido.

Doy mi consentimiento para que se archive este trabajo en la biblioteca universitaria de Universidad San Jorge, donde se puede facilitar su consulta.

Firma Fecha

Dedicatoria y Agradecimiento

Le dedico este trabajo en primer lugar a mi madre, quien siempre ha luchado por que mi futuro y el de mi hermano sean mejor que el suyo. Si algo tengo claro es que sin ella no hubiera llegado nunca donde estoy y siempre se lo agradeceré. A mi hermano porque, aunque discutimos más de lo que me gustaría reconocer, siempre ha estado ahí cuando he necesitado consejo de quien nadie más me lo podía dar.

A mi pareja por aguantar mis momentos de estrés, y más de una charla sobre mis proyectos y exámenes. Gracias por el cariño y la paciencia.

A mis amigos y compañeros de clase, en especial a Gabriel Villalonga por estar siempre dispuesto a ayudarme y obligarme a mejorar profesional y académicamente, a Jorge Solano y a Francisco Núñez. Creo que esta experiencia académica no hubiera sido igual si no hubiera coincidido con vosotros en mi paso por la universidad; realmente me siento afortunado de haberos conocido.

A mi tutor académico Eduardo Jiménez, por aceptar tutelar este trabajo y por lograr exprimir mi potencial en el aula y fuera de ella y por enseñarme a ser autoexigente.

Finalmente, a toda mi familia y amigos que han tenido que soportarme durante los años de mi paso por la universidad. Muchas gracias por confiar en mí y apoyarme cuando más lo he necesitado.

Tabla de contenido

Resumen **1**
Abstract **1**
1. Introducción **3**
2. Estado del arte **6**
2.1. Motor de ajedrez **6**
2.2. Piezas **6**
2.2.1. Peón *6*
2.2.2. Caballo *6*
2.2.3. Alfil *7*
2.2.4. Torre *7*
2.2.5. Dama *7*
2.2.6. Rey *7*
2.3. Movimientos **8**
2.3.1. Medios movimientos (half move o ply) *8*
2.3.2. Movimiento (full move o movimiento completo) *8*
2.3.3. Doble movimiento de peón *8*
2.3.4. Promoción/Coronación *9*
2.3.5. Enroque *9*
2.4. Capturas **10**
2.4.1. Captura al paso *10*
2.5. Cadena FEN **11**
2.5.1. Posicionamiento de las piezas *12*
2.5.2. Color activo *12*
2.5.3. Derechos de enroque *12*
2.5.4. Posible captura al paso *13*
2.5.5. Medios movimientos *13*
2.5.6. Movimientos completos *13*
2.6. Evaluación de movimientos **13**
2.7. Búsqueda de movimientos **14**
2.7.1. Minimax *14*
2.7.2. Algoritmo Negamax y Alpha-Beta pruning *15*
2.7.3. Búsqueda Quiescente (Quiescense Search) *15*
2.7.4. MVV–LVA (Most Valuable Victim - Least Valuable Attacker) *16*
2.7.5. PV (Principal Variation) *16*
2.7.6. Late Move Reduction (LMR, Reducción de Movimientos Finales) *17*
2.7.7. Null move pruning (Poda de Movimientos Vacíos) *17*
2.8. Bitboards **17**
2.8.1. Máscaras de ataques *20*
2.8.2. Ocupaciones *20*
2.9. Magic Bitboards **21**
2.10. Codificación de movimientos **22**
3. Objetivos **25**
4. Metodología **27**
4.1. Notion **27**

4.1.1. Registro del Backlog ... 27
4.1.2. Registro de las reuniones ... 28
4.2. Discord ... 29
4.3. Metodología ágil para el desarrollo ... 29
4.4. Control de versiones (Git) ... 29
5. Análisis e implementación ... 30
5.1. Funcionalidades implementadas ... 30
5.2. Arquitectura y estructura del motor ... 30
5.3. Representación en Bitboards y utilidades ... 32
5.3.1. Enumeraciones y constantes ... 32
5.3.2. Operaciones de bits ... 34
5.4. Tablas de ataques ... 37
5.4.1. Peones ... 38
5.4.2. Caballos ... 40
5.4.3. Rey ... 42
5.4.4. Alfiles ... 43
5.4.5. Torres ... 46
5.4.6. Dama ... 47
5.5. Bitboards y Ocupaciones ... 48
5.6. Parseo de cadenas FEN ... 49
5.7. Generación y codificación de movimientos ... 52
5.8. Generación de movimientos de peón ... 53
5.8.1. Movimientos sin captura del peón ... 53
5.9. Generación de movimientos de enroque ... 55
5.10. Generación de movimientos del resto de piezas ... 56
5.11. Solución Copy/Make ... 57
5.12. Ejecución de movimientos sobre el tablero ... 58
5.13. Evaluación ... 60
5.14. Búsqueda ... 63
5.14.1. Negamax & Alpha-Beta Pruning ... 63
5.14.2. Búsqueda quiescente ... 67
5.14.3. MVV LVA ... 69
6. Estudio económico ... 72
6.1. Costes del desarrollo ... 72
6.2. Coste de los materiales ... 72
6.3. Coste total ... 73
6.4. Perspectiva de negocio ... 73
7. Resultados ... 74
8. Conclusiones y futuros desarrollos ... 82
9. Bibliografía ... 84

Resumen

BBChess es un motor de ajedrez basado en *Bitboards* desarrollado en C y portado posteriormente a Unreal Engine 5 en C++ para el próximo juego de la empresa Eclipse Games. Este motor está basado en los llamados *Bitboards*, que no deja de ser otro nombre que se le ha dado a un tipo de dato primitivo ya existente y soportado por prácticamente todos los lenguajes de programación modernos, el número entero de 64 bits sin signo, o *uint64*. Esta aproximación a la programación de un motor de ajedrez es conocida por estar centrada en las piezas y su representación en el tablero. El concepto de esta solución se basa en usar la representación binaria de un *uint64* para representar el estado en el tablero de cada casilla, dado que el número de bits en esta representación coincide con el número de casillas en el tablero de ajedrez. De esta manera podemos realizar operaciones que normalmente conllevarían entradas y salidas a memoria como simples operaciones de bits. También, como en el juego al que este motor está destinado se centra en partidas contra la máquina, se va a desarrollar las funcionalidades completas de un motor de búsqueda y evaluación de movimientos; estos últimos también serán manipulados para ser codificados en enteros de 32 bits con el fin de aligerar el uso de memoria por parte del motor a la hora de generar e interpretar los movimientos en las búsquedas y evaluaciones. Este proyecto se centra en salir de la zona de confort que nos proporciona la programación orientada a objetos más cercana al lenguaje humano y ofrecer un cambio de perspectiva y saber adaptar nuestra programación al problema que se quiere resolver.

Abstract

BBChess is a chess engine based on *Bitboards* developed in C and later ported to Unreal Engine 5 in C++ for the upcoming game of the company Eclipse Games. This engine is based on so-called *Bitboards*, which is just another name for an existing primitive data type supported by virtually all modern programming languages, the 64-bit unsigned integer, or *uint64*. This approach to programming a chess engine is known to be piece centric, focusing on their representation on the chessboard. The concept of this solution is based on using the binary representation of a *uint64* to represent the board state of each square, since the number of bits in this representation matches the number of squares on the chessboard. In this way we can perform operations that would normally involve many memory read and write operations as simple bit calculations. Also, as the game for which this engine is intended focuses on games against the machine, the full functionalities of a move search and evaluation engine will be developed; the latter will also be manipulated to be encoded in 32-bit integers in order to lighten the use of memory by the engine when generating and interpreting the moves in the searches and evaluations. This project focuses on leaving the comfort zone provided by object-oriented programming, which is closer to the human language, and offering a change of perspective and knowing how to adapt our programming to the problem to be solved.

1. Introducción

Cuando investigamos un poco sobre la programación de motores de ajedrez, descubrimos que hay distintas maneras de afrontar este problema en cuanto al diseño del proyecto y del manejo de datos, entre las que destacan las arquitecturas basadas en vectores bidimensionales, unidimensionales, en representación 0x88 o, la seleccionada para el desarrollo de este proyecto, la arquitectura basada en Bitboards.

Esta arquitectura nos ofrece una versión centrada en las piezas a la hora de representar datos como la ocupación de las piezas o sus ataques sobre este. Esto se debe a que el uso que damos en este tipo de arquitectura a los Bitboards es la de representar con estos las ocupaciones en el tablero de todas las piezas, generando hasta doce Bitboards (uno por cada tipo y color de pieza) con las posiciones en las que hay alojada una pieza del tipo que mantienen activada.

Estas "activaciones" no son más que bits activados o apagados ya que, como su propio nombre indica, los Bitboards definen el tablero mediante la representación binaria de números enteros sin signo de 64 bits, los cuales podemos manipular mediante operaciones de bits para realizar operaciones como capturas o movimientos sobre el tablero, reduciendo así la carga computacional a la hora, sobre todo, de la búsqueda de movimientos.

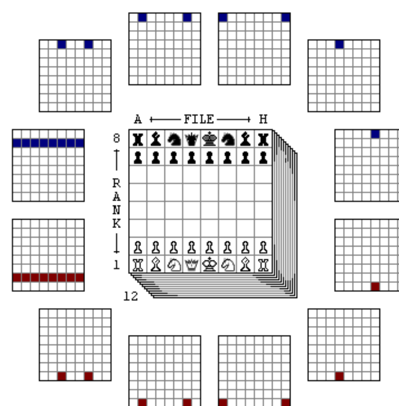

Ilustración 1-1. Máscaras de bits para la ocupación de cada pieza en el tablero inicial

La búsqueda de movimientos es uno de los dos principales pilares de los motores de ajedrez, siendo la evaluación de posiciones su otra gran funcionalidad. Sin una manera de evaluar una posición, un motor no es capaz de decidir el mejor movimiento a ejecutar para una posición dada. Esta evaluación se puede realizar en función de varios parámetros, pero los más comunes para evaluar una posición son el valor del material (las piezas) y su posicionamiento en el tablero. Siempre se pueden añadir variables a la ecuación, pero esta es la base para poder evaluar de una manera básica y fiable una posición.

En cuanto a la búsqueda, el estándar en este tipo de programas es optar por el algoritmo por excelencia para este tipo de problemas, el algoritmo Negamax. Este algoritmo basa su implementación en la generación de un árbol de búsqueda en el que cada nodo representa una posición; el resultado de haber ejecutado uno de todos los movimientos disponibles en cada uno de estos nodos, alternando la evaluación en la siguiente llamada recursiva para compararla con la puntuación que el jugador contrario obtendría. De esta manera, maximizamos la puntuación que gana el jugador que haya comenzado la búsqueda, minimizando la del oponente, evaluando cada nodo para calcular esta diferencia de puntuación.

Ahora bien, representar el tablero en forma de bits es tremendamente útil computacionalmente, pero es tremendamente abstracto para nosotros como humanos. ¿Cómo traducimos todo esto a algo que podamos entender visualmente? Por suerte, existe una solución para esta pregunta llamada FEN, cadenas de caracteres que encapsulan toda la información referente a una posición del tablero de ajedrez. Estas cadenas de caracteres se componen de 6 partes que recogen el posicionamiento de las piezas, el lado al que le toca jugar, los derechos a enrocarse disponibles para ambos lados, la casilla abierta para captura al paso (si la hubiera) y los contadores de medios movimientos y movimientos completos.

Ilustración 1-2. Ejemplo de representación de un tablero en forma de cadena FEN

Finalmente, una vez podemos abstraer el tablero y traducirlo desde y hasta estados de partidas enteros, solo nos queda hacer que nuestro programa pueda jugar partidas aplicando todas las técnicas anteriormente mencionadas. Toda esta complejidad añadida a la hora de programar la veremos traducida en pura eficiencia. Habremos conseguido adaptar nuestro programa a la máquina, ya que un ordenador entiende mejor y prefiere operar con ceros y unos, y nosotros se lo damos para facilitarle el trabajo.

BBChess ha conseguido que mi visión se amplie, haciendo que las máscaras de bits no sean algo que temer o repudiar, sino un recurso más que poder aprovechar cuando la eficiencia sea un aspecto clave para la solución del problema que se me ponga por delante.

2. Estado del arte

2.1. Motor de ajedrez

Un motor de ajedrez es un programa ejecutado desde un ordenador que es capaz de analizar movimientos y tableros, y generar una respuesta en forma de movimiento el cual, mediante los cálculos y evaluaciones pertinentes, clasifique como óptimo para la posición dada. Este programa conforma normalmente un *back-end* sin gráficos manejado desde una interfaz de consola de comandos.

Además, con el fin de que un jugador no tenga que aprender a usar cada uno de los diferentes motores de ajedrez que existan, se ha establecido un protocolo estandarizado el cual deberán seguir los motores de ajedrez para poder ser ejecutados en cualquier interfaz gráfica. Este protocolo se llama UCI, que significa Universal Chess Interface (Interfaz Universal de Ajedrez).

En esencia, los motores de ajedrez se pueden dividir en dos partes; la evaluación y la búsqueda.

2.2. Piezas

En el ajedrez, existen 6 tipos de piezas y cada una puede ser o blanca o negra; distinción la cual solo influye en quién mueve primero al comienzo de la partida (siempre juegan primero las piezas blancas, abriendo la partida) y en la dirección en la que se mueven los peones de cada color.

Las piezas tienen cada una un valor asociado denominado "calidad", que recoge el valor de la ventaja que otorga al jugador que captura dicha pieza.

2.2.1. Peón

Es la pieza básica del ajedrez. Esta pieza solo puede moverse en dirección frontal y en referencia a la posición del jugador. Es decir, los peones blancos sólo pueden avanzar hacia la fila 8 mientras que los peones negros sólo pueden avanzar hacia la fila 1.

Los peones tienen un valor en calidad de 1 punto.

2.2.2. Caballo

El caballo es la primera de las dos piezas menores. Se caracteriza por ser la única pieza que puede moverse por encima de sus propias piezas, siendo capaz de "atravesarlas". El movimiento del caballo se suele comparar al de la letra L, ya que se puede mover a una casilla situada o bien a 2 casillas en vertical y una casilla en horizontal o viceversa.

El caballo tiene un valor en calidad de 3 puntos, siendo la menos valiosa de las dos piezas menores.

2.2.3. Alfil

El alfil es la segunda pieza menor. Su movimiento es exclusivamente en diagonal y totalmente libre mientras no haya piezas que se interpongan en cualquiera de las cuatro diagonales en las que se mueva. Una característica de los alfiles es que cada uno de los alfiles con los que un jugador comienza la partida sólo puede moverse en el color de casilla desde el que comienza. Esto nos permite diferenciarlos entre alfil de negras o alfil de blancas, siendo este color el de la casilla por la que se mueve, no el color de la pieza.

El valor en calidad del alfil es 3.5, siendo este ligeramente más valioso que el caballo.

2.2.4. Torre

La torre es la primera pieza mayor. Esta puede moverse exclusivamente en vertical u horizontal con total libertad, mientras no se encuentre una pieza que bloquee su camino, igual que el alfil.

Esta pieza recompensa con una calidad de 5 puntos a quien la capture.

2.2.5. Dama

La dama es la pieza más valiosa que un jugador puede tener, sólo por detrás del rey. Puede moverse en línea recta, tanto en vertical, horizontal o diagonal, las casillas que quiera hasta toparse con una pieza o el borde del tablero, es decir, puede tener un comportamiento tanto de alfil como de torre. Precisamente por esta gran agilidad es la pieza que ningún jugador debería echar a perder; incluso es un acto de deportividad rendirse si se pierde la dama de una forma inadecuada.

Su valor casi dobla al de la torre, la otra pieza mayor, otorgando 9 puntos de calidad para quien capture la dama de su oponente.

2.2.6. Rey

El rey es la pieza más valiosa por el mero hecho de que quien capture el rey de su oponente gana automáticamente la partida, aunque nunca se llega realmente a capturar al rey, sino que se le da *jaque mate*. El rey posee la misma capacidad de dirección que la dama, pero sólo puede moverse una única casilla en cualquiera de las 8 direcciones que elija, siempre y cuando no se exponga a *jaque*, lo cual no es considerado un movimiento legal.

El valor de esta pieza es literalmente incalculable, dado que o la tienes sobre el tablero o has perdido la partida.

2.3. Movimientos

El término de "movimiento" en ajedrez utilizado a la ligera puede llegar a ser ambiguo, sobre todo cuando comienzas a leer documentación y artículos sobre el ajedrez, ya que algunos utilizan el término "movimiento" para hacer referencia a que un jugador mueva otra pieza, mientras que en otras fuentes se emplea como la secuencia de dos movimientos de pieza, uno por cada jugador. Para aclarar esto, me apoyaré en la notación FEN, más desarrollada en el apartado Cadena FEN.

2.3.1. Medios movimientos (half move o ply)

Los medios movimientos (en inglés "half move" empleado en la notación FEN, o "ply" en el contexto de motores de ajedrez) hace referencia a cada movimiento individual que produce un jugador al mover una pieza.

2.3.2. Movimiento (full move o movimiento completo)

El movimiento completo define una secuencia de dos plies (plural de ply) consecutivos, es decir, dos movimientos de pieza individuales, cada uno ejecutado por un jugador distinto alternativamente.

A lo largo del documento para no mezclar nomenclaturas, siempre que no sea estrictamente necesario por el contexto emplear el término ply para diferenciar medios movimientos de movimientos completos, se empleará el término "movimiento" para hacer referencia a lo que sería un medio movimiento o ply. En caso de utilizarlo para nombrar un movimiento completo se indicará adecuadamente.

Como ya muchos sabemos, en el ajedrez existen varios tipos de movimientos. Estos se clasifican en función del tablero resultante de realizar dicho movimiento.

2.3.3. Doble movimiento de peón

El doble movimiento de peón es un movimiento especial de los peones, y sólo puede realizarlo un peón que se encuentre en la fila desde la que ha iniciado la partida, siendo esta la fila 2 para las blancas y la 7 para las negras. En caso de poder realizarlo y como su nombre indica, el peón podrá avanzar dos casillas hacia delante en lugar de una única casilla, pero no es obligatorio tener que mover dos casillas siempre que esté abierta la posibilidad a hacerlo.

2.3.4. Promoción/Coronación

La promoción es un movimiento exclusivo de los peones que se produce al alcanzar la fila más cercana a tu oponente con uno de estos y, en este caso, el peón coronado [1] es sustituido por bien un caballo, un alfil, una torre o una dama, a elección del jugador que ha logrado coronar.

2.3.5. Enroque

El enroque es el único movimiento que involucra que dos piezas acaben en casillas distintas en el mismo movimiento, el rey y una de las dos torres. En caso de enrocar hacia el lado de la torre más cercana al rey, el rey se desplazará en horizontal hasta la columna G y la torre de la columna H se moverá también en horizontal a la columna F, y se denomina enroque corto (O-O en notación algebraica [2]). Por otra parte, si el enroque se produce hacia el lado de la dama, el rey se desplazará en horizontal hasta la columna C y la torre de la columna A ocupará la casilla contigua al monarca en la columna D, enrocando así en largo (O-O-O en notación algebraica).

El enroque es un movimiento que está sujeto a poder ser ejecutado bajo unas condiciones bastante estrictas. Estas condiciones son:

1) El enroque sólo es posible si ni el rey ni la torre con la que enrocarse se han movido de sus casillas, aunque hayan vuelto a las mismas posteriormente.
2) No puede haber ninguna pieza, aliada o enemiga, entre el rey y la torre con la que se vaya a realizar el enroque.
3) El rey no puede estar en jaque.
4) La casilla a la que el rey va a parar no puede estar en jaque, ni ninguna de las casillas entre la casilla de origen y la de destino.

Realmente, ambos jugadores, al comienzo de la partida, tienen todos los derechos de enrocar disponibles (tanto en corto como en largo), y únicamente se perderían estos derechos si moviéramos una torre (en cuyo caso perderíamos el derecho al enroque hacia el lado de la torre desplazada), si moviéramos el rey (entonces se perderían ambos derechos de enroque) o si hubiéramos enrocado ya anteriormente, en cuyo caso perdemos automáticamente el derecho a enrocar del lado contrario.

1 Coronar hace referencia al hecho de llegar con el peón a la fila más cercana al oponente.

2 Notación basada en coordenadas para definir un movimiento en ajedrez.

2.4. Capturas

Las capturas son un movimiento que termina con, como su nombre indica, la captura de una pieza del rival, otorgando así una ventaja momentánea (hasta que el rival complete el movimiento no se puede valorar la posición de manera fiable) a quien ha capturado.

En el ajedrez, las capturas son realizadas por las piezas menores y mayores de la misma manera en la que se mueven por el tablero, a excepción del peón, que no captura hacia delante, sino en las casillas diagonales estrictamente frontales, es decir, las dos casillas que el peón tiene una casilla por delante a cada lado de la casilla a la que podría moverse. Esto puede desembocar en posiciones donde un jugador que captura con peón ahora tenga dos peones en la misma columna, en cuyo caso el peón posterior quedaría bloqueado por el que tiene delante.

La única captura especial en el ajedrez también la tiene el peón, y se conoce como captura al paso, la cual mucha gente incluso aficionada no suele conocer.

2.4.1. Captura al paso

La posibilidad de capturar al paso se activa cuando un peón realiza un doble movimiento, y se abre en la casilla inmediatamente posterior al peón (la casilla posterior en la fila 3 para los peones blancos y en la 6 para los peones negros que hayan terminado en las filas 4 y 5 respectivamente por un doble movimiento de peón) la posibilidad de capturar dicho peón al paso

Una vez se activa esta casilla y es el turno del oponente, si este tenía un peón ubicado en cualquiera de las dos casillas inmediatamente contiguas lateralmente al peón que el oponente ha avanzado doblemente, el peón ocupa la casilla abierta para la captura al paso y el peón rival (el que había avanzado dos casillas) es capturado.

La siguiente ilustración se verá un ejemplo de una captura al paso.

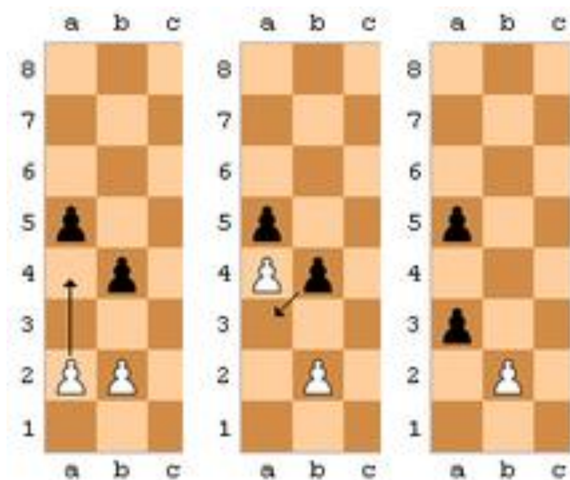

Ilustración 2-1. Secuencia de movimientos en las que el peón negro de la columna B captura el peón blanco de la columna A al paso, abierto en la casilla A3 tras el movimiento a4.

Como Podemos observar, primero el peón ubicado en A2 avanza dos casillas hasta A4 (movimiento "a4" [3] en notación algebraica), abriendo la casilla A3 para su captura al paso. El peón negro de la columna B acaba estando en la casilla contigua horizontalmente al peón de la columna A, que ha realizado un movimiento doble. El peón negro entonces captura al paso el peón blanco de A2 y termina en A3, la casilla que se había abierto para capturar al paso.

2.5. Cadena FEN

Como primer paso a la hora de conectar el mundo real (tableros, piezas, jugadores…) con el virtual (ordenadores, motores de ajedrez…) encontramos las cadenas FEN (Forsyth-Edwards Notation), que definen de manera estandarizada únicamente la posición de un tablero de ajedrez con todos los datos necesarios para poder continuar una partida en el punto exacto que representa, haciendo posible traducir cualquier posición a una línea de texto plano.

Para poder representar la posición completa, esta cadena está formada por 6 partes separadas por espacios entre sí, y que cada una representa un aspecto de la posición distinto; empleando únicamente caracteres ASCII para que cualquier ordenador moderno pueda analizarla con facilidad.

Esta cadena emplea las iniciales de los términos no numéricos en inglés, así que cada carácter hace referencia a la inicial de la palabra en inglés a la que representa.

3 En notación algebraica, los movimientos de peón que no involucran ninguna captura sólo enumeran la casilla en la que termina el peón, ya sea un movimiento simple o doble.

2.5.1. Posicionamiento de las piezas

Este es el primer campo que contiene una cadena FEN. Consiste en 8 cadenas de caracteres y dígitos separadas entre ellas por el carácter "/" que recogen la posición de las piezas sobre el tablero de ajedrez, ordenadas de en orden descendente desde la fila 8 hasta la 1.

Para ello, se emplea un carácter para cada tipo de pieza y su capitalización para determinar el color de dicha pieza, siendo los caracteres "p", "n", "b", "r", "q" y "k" para representar los peones, los caballos, los alfiles, las torres, la dama y el rey, respectivamente, y empleando caracteres en mayúscula para las piezas blancas y en minúscula para las negras. Los dígitos representan la cantidad de casillas vacías consecutivas, contadas de izquierda a derecha.

Ilustración 2-2. La representación FEN para el posicionamiento de las piezas para este tablero sería r1bk3r/p2pBpNp/n4n2/1p1NP2P/6P1/3P4/P1P1K3/q5b1

2.5.2. Color activo

Este campo dentro de la cadena FEN indica qué jugador es el siguiente en mover. Se emplea un único carácter para representar este dato, siendo "w" para determinar que mueven las piezas blancas y "b" en caso de que sean las piezas negras quienes muevan a continuación.

2.5.3. Derechos de enroque

Esta secuencia de entre uno y cuatro caracteres determina qué derechos de enroques siguen abiertos en el momento de haber transcrito la posición para cada jugador. Un enroque corto se representa con un carácter "k", representando que el jugador tiene derecho a enrocar hacia el lado del rey, o una "q" si es hacia el lado de la dama. De la misma forma que en el posicionamiento

de las piezas, la capitalización nos indica el color del derecho a enroque. Finalmente, si a ningún jugador le quedan derechos de enroque, este campo se representa con el carácter "-".

Entonces, unos ejemplos de derechos de enroque representados en una cadena FEN serían:

- "KQkq": tanto el jugador de blancas como el jugador de negras pueden enrocar tanto en corto como en largo (lado de rey y lado de dama, respectivamente)
- "Kq": el jugador de blancas ha perdido el enroque largo, mientras que el de negras ha perdido el enroque corto, pero siguen teniendo los otros derechos correspondientes.
- "-": En este caso, ningún jugador tiene derechos de enroque.

2.5.4. Posible captura al paso

Indica la casilla que haya quedado abierta para captura al paso. En caso de haber una casilla abierta para esta captura especial, se emplean dos caracteres, una letra de la "a" a la "h" representando la columna, y un dígito de los dos posibles, "3" o "6", representando la fila donde se encuentra. Esta explicación se puede resumir como la representación algebraica de la casilla donde se haya abierto una captura al paso.

Si no hubiera ninguna casilla abierta, se emplea únicamente el carácter "-".

2.5.5. Medios movimientos

Este carácter (o pareja de caracteres si llegase o superase los 10 medios movimientos) representa la cuenta de medios movimientos o plies realizados desde el último movimiento de peón o captura, en cuyo caso este contador se reinicia a 0.

Este campo es utilizado para terminar la partida en tablas por la regla de los 50 movimientos, que rige que si este contador llega a 100 medios movimientos (50 movimientos completos sin capturas o movimientos de peones), el juego termina en tablas.

2.5.6. Movimientos completos

El contador de movimientos completos cuenta, como su nombre indica, movimientos completos, indicando así el número de turnos completos en la partida.

2.6. Evaluación de movimientos

En la evaluación, el motor toma posiciones individuales y decide cuál es la mejor. Normalmente, los motores de ajedrez toman en consideración un sistema de puntuación similar (si no idéntico) al que emplean los jugadores; asignar un valor numérico a cada pieza y contar los puntos de un jugador frente al otro. Otros, en cambio, no sólo emplean este sistema, sino que le añaden

complejidad para poder evaluar una posición de una manera más precisa, previendo ataques o amenazas, la seguridad del rey o la estructura de peones.

2.7. Búsqueda de movimientos

Una búsqueda de movimientos rudimentaria nos ofrece que, para una posición y una profundidad de búsqueda dadas, el motor nos generará una lista con todos los movimientos para una posición y, para cada uno de esos movimientos, continuará buscando hasta llegar a la profundidad deseada. Una vez haya recorrido todo el árbol de movimientos, nos devolverá el movimiento que mejor puntuación proporcione. Obviamente, cuanta más profundidad requiramos a la búsqueda, mejor será el movimiento dada la exhaustiva comparación para largas secuencias de movimientos.

Cada capa de profundidad de la búsqueda se define como *ply*, que representa cada movimiento individual hecho por un jugador, siendo un movimiento entero la ejecución de 2 *plies* consecutivos, uno por cada jugador. Normalmente, para un *ply* de 20 (10 movimientos, 2 *plies* por jugador), la máquina ya está mirando mucho más lejos de lo que lo haría un jugador humano.

El principal problema de los algoritmos de búsqueda es que, como podemos intuir, su complejidad de tiempo crece de manera exponencial, generando problemas de rendimiento a la hora de tener que calcular movimientos de forma rápida en partidas competitivas. Es por esto por lo que se han desarrollado algoritmos de *pruning* (poda en inglés) que nos permiten "podar" el gran árbol generado en la búsqueda global.

2.7.1. Minimax

Minimax es un algoritmo para determinar una puntuación en juegos de Suma Cero [4] de acuerdo con la Teoría de Juegos [5]. En el ajedrez este algoritmo determina dicha puntuación tras ser realizados una serie de movimientos, idealmente siendo estos los mejores movimientos de acuerdo con una función de evaluación. Este algoritmo se ejecuta recursivamente, recibiendo dos parámetros, *alpha* y *beta*, correspondientes a la puntuación de cada jugador. Esta función calcula estos valores en cada llamada recursiva y los intercambia en la siguiente. De esta manera, se minimiza la puntuación del rival y se maximiza la del jugador realizando la búsqueda.

Para una búsqueda de un único *ply*, la puntuación es determinada al comparar todos los movimientos que el jugador que mueve (jugador "max", al que maximizar la puntuación) tiene disponibles y encontrar el que le proporcione mayor puntuación en la evaluación.

4 Representación matemática en la que un resultado supone una ventaja para un jugador equivalente a la pérdida del contrario.

5 Área de la matemática aplicada que estudia interacciones en estructuras formalizadas de incentivos (juegos).

2.7.2. Algoritmo Negamax y Alpha-Beta pruning

El algoritmo Negamax es una especificación del algoritmo Minimax. En Negamax, en lugar de utilizar dos subrutinas para calcular la mínima puntuación para el jugador y la máxima, podemos calcular la puntuación negada para cada *ply* dada la siguiente relación matemática:

$$\max(a, b) == -\min(-a, -b)$$

El algoritmo Alpha-Beta es un algoritmo que mejora significativamente la eficiencia de una búsqueda Minimax. Consiste en reducir la búsqueda en el árbol de movimientos generado gracias a poder determinar, mediante técnicas de poda, ramas del árbol que ignorar y, por tanto, evitar todos los cálculos que hubieran sido necesarios para buscar movimientos en dicha rama.

Lo bueno de este algoritmo es que no pasa por alto posibles movimientos mejores que el que ya se haya encontrado en el momento de decidir si "podar" o no, gracias a que si ya se ha encontrado un movimiento significativamente bueno, una sola refutación [6] de este es suficiente para evitar el movimiento encontrado.

Esta mejora es altamente considerable. Para una búsqueda Minimax donde se generan X nodos, una búsqueda que implemente Alpha-Beta puede reducir la cantidad de nodos a $\sqrt{(X)}$ idealmente, aunque la cantidad de nodos real que se poden dependerá, sobre todo, de la ordenación de los movimientos generados. Si los mejores movimientos aparecen al comienzo de la búsqueda, podremos podar más fácilmente y muchas más ramas que si primero buscásemos los peores.

2.7.3. Búsqueda Quiescente (Quiescense Search)

Esta búsqueda es una que no busca en grandes colecciones de movimientos, sino en pequeñas cantidades de estos, con el fin de sólo evaluar movimientos silenciosos [7]. Estas búsquedas son necesarias para evitar el llamado Efecto Horizonte [8].

Parar la búsqueda en la profundidad establecida y evaluar en ese punto puede llevar a pérdida de material [9]. Imaginemos que el último movimiento de nuestra búsqueda es torre captura un peón. Esta es una clara ganancia de material, en concreto de un peón. Pero ¿qué pasaría si el siguiente movimiento del rival resulta en la captura de nuestra torre? Esto sería una clara pérdida

---

6 Movimiento que, dentro de una búsqueda, causa una gran desventaja y, por tanto, refuta un movimiento anterior.

7 Movimientos que no llevan a posiciones estratégicas, capturas o ataques.

8 Causado por la limitación de profundidad en los algoritmos de búsqueda, una posición de desventaja no es detectada en la búsqueda y es, aunque aplazable en el tiempo, inevitable.

9 En ajedrez, la pérdida de material se produce cuando, teniendo en cuenta los valores asociados a cada pieza, tras un movimiento completo, terminas con menos puntos (piezas) que tu oponente.

de material, ya que al principio pensábamos que habíamos ganado un peón cuando realmente estamos perdiendo una pieza mayor.

Precisamente para evitar estas situaciones donde existe el potencial peligro de perder material por no poder evaluar y buscar más allá de la profundidad establecida, se evalúan únicamente posiciones quiescentes.

2.7.4. MVV–LVA (Most Valuable Victim - Least Valuable Attacker)

MVV-LVA es una heurística utilizada para ordenar movimientos con captura optimizada. Para comprender esta heurística, sólo hace falta analizar el nombre que la define. Traducido al español, recibe el nombre de Víctima Más Valiosa-Atacante Menos Valioso. ¿Qué nos quiere decir esto?

Para una posición dada, se debe encontrar la pieza más valiosa del rival que, si fuera capturada, generaría una posición ventajosa para nosotros, y convertirla en nuestra víctima. Una vez encontrada, una segunda búsqueda deberá encontrar el atacante menos valioso que pueda capturar dicha víctima.

Los órdenes por valores de las piezas en orden ascendente son el peón, el caballo, el alfil, la torre, la dama y el rey, el cual recibe un valor infinito, ya que si es capturado el juego termina automáticamente. Una cosa a tener en cuenta es que esta heurística no es segura por completo si la víctima atacada por piezas más valiosas esta defendida.

2.7.5. PV (Principal Variation)

La Variación Principal (PV) consiste en una secuencia de movimientos que nuestro motor estima como mejores y, por tanto, espera que sean jugados. Todos los nodos dentro de la Variación Principal se llaman *PV-nodes*. Esta secuencia de movimientos es, dentro de un algoritmo de búsqueda en profundidad, la consideración más importante dentro de cada iteración.Profundización Iterativa

La Profundización Iterativa es la estrategia de gestión de tiempo más básica para los algoritmos DFS [10] y resulta ser una de las mejoras más significativas en la ordenación de movimientos en algoritmos como Alpha-Beta.

Esta mejora se puede interpretar como un intento de unificar una búsqueda en profundidad y una búsqueda en anchura al mismo tiempo. Esta estrategia de búsqueda es tan buena como una búsqueda en anchura, pero empleando mucha menos memoria. Esto es posible gracias a que

10 *Depth-First Search* es un algoritmo de búsqueda para estructuras basadas en nodos conectados (como árboles o grafos) en el que se prioriza visitar nodos en niveles de profundidad mayores (hijos) en lugar de nodos en la misma profundidad.

para cada iteración, se visita un nodo como si se tratase de una búsqueda en profundidad, pero el orden en el que los nodos son visitados realmente corresponde con el de una búsqueda en anchura.

2.7.6. Late Move Reduction (LMR, Reducción de Movimientos Finales)

Esta técnica de reducción de memoria reduce la carga computacional de la búsqueda reduciendo los movimientos que están hacia el final de la ordenación. Este proceso normalmente no evita por completo que se lleguen a buscar movimientos en dichas ramas, sino que, si no se encuentran movimientos relevantes en una profundidad intermedia/baja, no se siga profundizando más en la búsqueda.

2.7.7. Null move pruning (Poda de Movimientos Vacíos)

Esta metodología se emplea para reducir el espacio de la búsqueda pasando movimientos "vacíos" y viendo si la puntuación del sub-árbol generado sigue siendo lo suficientemente óptima como para producir una poda o no.

Esta técnica está basada en la Observación de Movimientos Nulos, la cual propone que, para el lado que deba mover, casi siempre hay un mejor movimiento que uno que no hace nada. Esta observación se puede considerar como un bonus de *tempo*, pero en situaciones conocidas como Zugzwang [11] no es de gran ayuda.

2.8. Bitboards

Los Bitboards son agrupaciones de 64 bits que se utilizan para representar, entre otras cosas, un tablero de ajedrez centrado en el posicionamiento y ocupación de las piezas en el mismo. Es necesario que dichas colecciones de bits contengan 64 bits, pues es la cantidad de casillas que existen en el tablero de ajedrez; empleando de esta forma un bit para representar una casilla del tablero. Esta relación de bit-casilla no es arbitraria, pues sino realizar movimientos o ataques sería tedioso y por completo contraproducente; precisamente se emplean sets ordenados de bits para poder aproximar el tablero a esta representación abstracta.

Un tablero de ajedrez completo tiene, como máximo, 12 tipos de piezas diferentes. Esto nos obliga a tener, por lo menos, 12 Bitboards para poder representar las ocupaciones de cada pieza en el tablero. De esta manera, marcaremos como bits activos los bits que ocupe una pieza en la casilla correspondiente en el tablero. A lo largo de esta memoria, se van a ver representaciones de Bitboards manipuladas para poder comprender mejor (de una manera más visual que si

11 Posición en la que mover desemboca en una posición desventajosa, sea el movimiento que sea, normalmente hasta una posición de pérdida. Suele darse en la fase final de la partida, especialmente en finales de peones. (Chess Programming Wiki, n.d.) (Chess Programming Wiki, n.d.) (Chess Programming Wiki, n.d.)

viéramos la representación binaria en crudo) su extrapolación al tablero. Dichas representaciones tendrán el formato de la Ilustración 2-1.

	A	B	C	D	E	F	G	H
8	0	1	2	3	4	5	6	7
7	8	9	10	11	12	13	14	15
6	16	17	18	19	20	21	22	23
5	24	25	26	27	28	29	30	31
4	32	33	34	35	36	37	38	39
3	40	41	42	43	44	45	46	47
2	48	49	50	51	52	53	54	55
1	56	57	58	59	60	61	62	63

Ilustración 2-3. Plantilla de representación, adaptada a un tablero, de un Bitboard

En esta ilustración podemos apreciar que las coordenadas de cada eje corresponden a las coordenadas típicas a las que encontramos en el tablero de ajedrez común. Además, en cada casilla encontramos el índice de esta, que comienza en la casilla A8 e incremente hacia la columna H en horizontal y hacia la fila 8 en vertical, numerando cada casilla de izquierda a derecha y de arriba abajo. Este índice hace referencia al índice del bit correspondiente en la representación de un número entero sin signo de 64 bits; a dicha representación se hará referencia a partir de ahora como *uint64*.

De este modo, el estado del bit menos significativo (el bit situado más a la izquierda) de un *uint64* representará el estado de la casilla A8 (índice 0), y el bit más significativo (el bit situado más a la derecha) del mismo *uint64*, representa el estado de la casilla H1 (índice 63).

Una vez podemos representar el estado de las casillas con un Bitboard, podemos emplear un Bitboard arbitrario para el fin que se desee, teniendo en cuenta que toda esta conversión tiene el fin de facilitar el uso de operaciones binarias entre parejas de Bitboards. Los Bitboards no solo se emplean para determinar las posiciones de las piezas, sino que también se pueden emplear para representar las casillas a las que se puede mover una pieza desde una casilla si realizamos operaciones *shift* de bits, o para saber si una pieza esta atacada o no mediante operaciones lógicas AND entre Bitboards.

	A	B	C	D	E	F	G	H
8	0	0	0	0	0	0	0	1
7	0	0	0	0	0	0	1	0
6	0	0	0	0	0	1	0	0
5	1	0	0	0	1	0	0	0
4	0	1	0	1	0	0	0	0
3	0	0	0	0	0	0	0	0
2	0	1	0	1	0	0	0	0
1	1	0	0	0	1	0	0	0

Ilustración 2-4. Máscara de bits representando todas las casillas a las que potencialmente puede moverse un alfil situado en C3

	A	B	C	D	E	F	G	H
8	1	1	1	1	1	1	1	1
7	1	1	0	0	1	1	0	1
6	0	1	0	1	0	0	1	0
5	0	0	1	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0

Ilustración 2-5. Bitboard de la ocupación en el tablero de todas las piezas negras para una posición arbitraria

A modo de ejemplo, gracias a tener estas representaciones, podemos saber que el alfil (supongamos blanco por el bien de este ejemplo) de la Ilustración 2-4 está atacando a la pieza negra situada en H8 (presumiblemente una torre) de la Ilustración 2-5 si sometiéramos ambos Bitboards a una operación lógica AND, que nos daría como resultado un Bitboard en el que el único bit activo es el situado en H8.

¿Qué ocurriría si hubiera alguna pieza blanca entre el anterior alfil blanco y la casilla H8? ¿Una operación AND para este caso no nos aseguraría que ese alfil puede capturar la pieza de H8? En efecto, así lo haría, pero la clave se encuentra en que para las piezas que se mueven deslizándose sobre el tablero como los alfiles, la torre o la dama, no podemos usar directamente la máscara de bits precalculada que nos indique todas las casillas a las que se pueden mover, sino que debemos calcular estas máscaras de ataques en tiempo real, ya que no podemos predecir la

ocupación del tablero en cada turno. Todas estas explicaciones se podrán ver detalladas en el apartado Análisis e implementación

2.8.1. Máscaras de ataques

Como ya se ha introducido en el apartado anterior, las máscaras de ataques no representan la ocupación de una pieza en el tablero, sino todas las casillas a las que una pieza potencialmente puede moverse desde una casilla. Se representan a modo de tablas donde solo es necesario el tipo de pieza y, en el caso de los peones, el color de este.

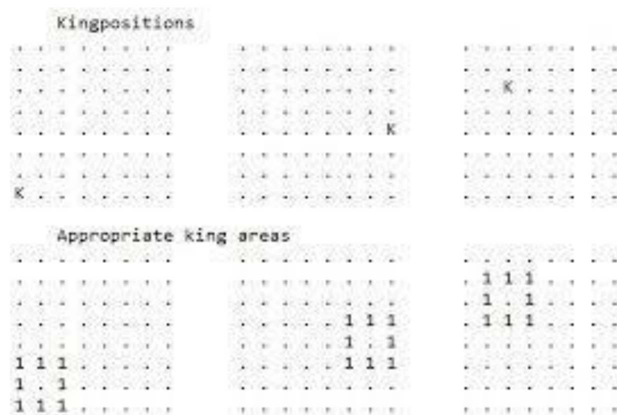

Ilustración 2-6. Máscaras de posibles ataques de un rey situado en diferentes posiciones del tablero

Estas tablas deben ser precalculadas, ya que no varía la casilla a la que se puede mover una pieza desde una casilla arbitraria, solo cambiará para las piezas que se deslizan sobre el tablero, y para ellas sí que hace falta generar una máscara en tiempo real, pero para esto se hace uso de la máscara precalculada.

2.8.2. Ocupaciones

Por último, podemos representar con Bitboards las ocupaciones de un tablero para una pieza en concreto o para un color entero. Las ocupaciones de las piezas por separado también nos sirven para conocer el estado del tablero, y a partir de ellas podemos conseguir la máscara de bits que define la ocupación de un color entero.

Como ya hemos podido ver en el ejemplo anterior, estas máscaras de ocupaciones se usan entre otras cosas para saber si una pieza que se desliza puede capturar una pieza enemiga si hacemos operaciones lógicas con la ocupación de su propio color.

Esta generación de máscaras de bits en tiempo real puede ralentizar nuestro programa, ya que una ocupación arbitraria es completamente impredecible de un turno a otro. Si tuviéramos una manera de poder precalcular muchas de estas ocupaciones, podríamos evitar tener que calcular

tantas máscaras en tiempo real y poder emplear precálculos si fuera posible. Bueno, pues este es otro uso de los Bitboards, y se conoce como Magic Bitboards.

2.9. Magic Bitboards

Los Bitboards Mágicos definen un algoritmo de *hash perfecto* [12] con el que definir unas tablas de ataque que para alfiles y torres de una pasada. Este método se ha convertido en un estándar en los programas de ajedrez basados en Bitboards dada la gran capacidad de computación de los microprocesadores modernos.

Esta técnica de hash viene de la idea de Lasse Hansen, un ingeniero noruego aficionado a la programación de ajedrez, y consiste en hacer un hash de las hasta 12 ocupaciones relevantes de bits en ambas direcciones de un alfil o una torre al mismo tiempo.

La manera en la que estos Bitboards Mágicos se generan consiste en:

1) Generar una máscara de bits con las ocupaciones relevantes para una clave (índice). De esta manera, si tuviéramos una torre en A1, los bits relevantes de ocupación serían los que van de A2 a A7 y los que van de B1 a G1.
2) Multiplicar la clave por un "número mágico" para obtener un mapeo para el índice. Este número normalmente se genera mediante prueba y error con fuerza bruta de una manera sencilla, aunque esto introduce la posibilidad de no escoger el número mágico óptimo.
3) Realizar operaciones binarias de *shift* a la derecha 64-n bits, siendo "n" el número de bits en el índice. Conectando con el apartado 2), un número mágico óptimo tendrá menos bits en el índice.
4) Utilizar ese índice para referenciar una tabla de ataques preinicializada.

En la siguiente ilustración se puede ver este proceso gráficamente, aunque no es para nada un algoritmo ni intuitivo ni amigable.

12 El hash perfecto se da cuando se conocen todas las claves de hash se conocen en tiempo de compilación y componen una colección pequeña; entonces se genera una tabla hash donde no hay colisiones por haber claves únicas.

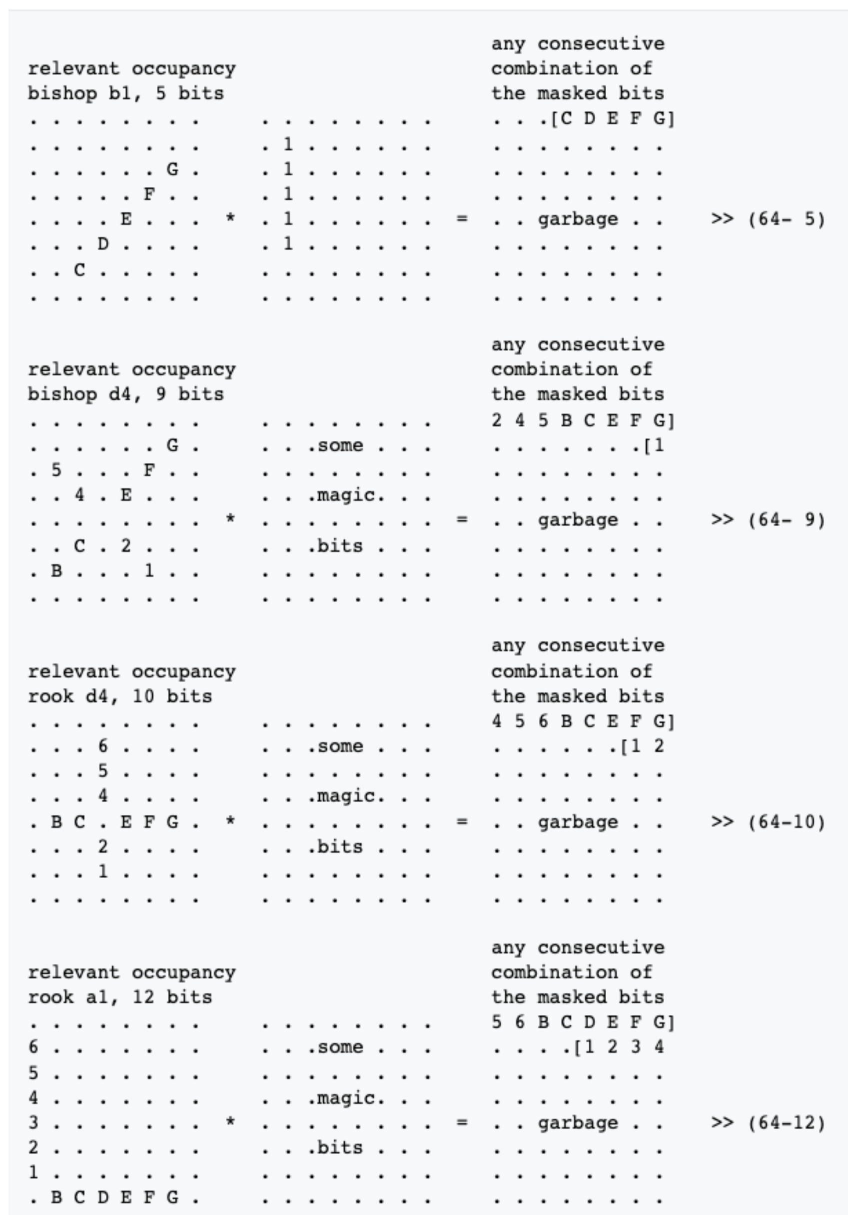

Ilustración 2-7. Representación gráfica del proceso de generación de Bitboards Mágicos

2.10. Codificación de movimientos

Los movimientos, para su evaluación, deben ser almacenados en colecciones que serán manipuladas posteriormente. Estas colecciones pueden potencialmente almacenar millones de movimientos, por lo que una encontrar una manera de intentar reducir la memoria empleada para almacenarlos es clave para no sacrificar mucha memoria.

Siguiendo tema principal de los Bitboards (los bits y su manipulación), podemos tratar de encontrar una forma de codificar un movimiento intentando usar representaciones binarias, aunque en este caso no serán necesarios 64 bits, sino unos cuantos menos.

Entonces, ¿cómo almacenamos toda la información necesaria para definir un movimiento en un set de bits? Bueno, para eso primero tenemos que saber qué propiedades del movimiento vamos a guardar y cuántos bits nos hacen falta para representar un valor de dicha propiedad; para los Bitboards la cantidad es una, una casilla activa o no, pero para estas propiedades, la cantidad de bits que necesitamos es variada para cada uno de estos campos.

Realizando unos cálculos y tratando de minimizar el espacio a ocupar por movimiento, se ha llegado a la conclusión de que serán necesarios ni más ni menos que 24 bits para representar un movimiento y, por tanto, podemos emplear un número entero de 32 bits para codificarlo.

Las propiedades que codificar y la cantidad de bits necesaria para codificarlas son:

- Casilla origen: define la casilla desde la que la pieza a mover se mueve, y dado que existen 64 casillas en el ajedrez representadas en enteros dentro del rango [0-63], emplearemos 6 bits para codificar este campo.
- Casilla destino: define la casilla a la que la pieza a mover va a parar al finalizar el movimiento. Siguiendo la misma lógica que para la casilla origen, se hacen uso de otros 6 bits.
- Pieza: Define la pieza que se desplaza en el movimiento a codificar. Existen 12 tipos de piezas distinguidas por color y tipo de pieza, por tanto harán falta 4 bits como mínimo para codificarla (con 4 bits puedes representar hasta 16 valores, mientras que con 3 solo puedes representar 8).
- Pieza a la que promocionar: defina la pieza (por tanto, se emplearán 6 bits de nuevo para codificar este campo) a la que un peón coronaría tras llegar al fondo del lado opuesto.
- Flag [13] de captura: Flag que nos revela si el movimiento es una captura o no. Para las 4 flags restantes, se empleará un único bit para cada una.
- Flag de doble movimiento de peón: Flag que nos revela si este movimiento ha sido un doble movimiento de peón. Se emplea, sobre todo, para poder actualizar el estado de la casilla que se abre para una captura al paso.
- Flag de caputura al paso: nos indica si la captura que define este movimiento ha sido realizada al paso o no.
- Flag de enroque: activa en caso de tratarse de un movimiento de enroque. Nos sirve para poder eliminar los enroques que el jugador que lo ha realizado tenía disponibles.

El reparto de bits resultante, si contamos todos los campos para codificar un movimiento, suman 26 bits. En la siguiente ilustración se puede ver reflejado dicho reparto a lo largo del entero.

---

13 Flag en programación hace referencia un bit, y se emplean para almacenar un estado binario.

```
0000 0000 0000 0000 0000 0000 0011 1111   source square          0×0000003F
0000 0000 0000 0000 0000 1111 1100 0000   target square          0×00000FC0
0000 0000 0000 0000 1111 0000 0000 0000   piece                  0×0000F000
0000 0000 0000 1111 0000 0000 0000 0000   promoted piece         0×000F0000
0000 0000 0001 0000 0000 0000 0000 0000   capture flag           0×00100000
0000 0000 0010 0000 0000 0000 0000 0000   double pawn push flag  0×00200000
0000 0000 0100 0000 0000 0000 0000 0000   enpassant flag         0×00400000
0000 0000 1000 0000 0000 0000 0000 0000   castling flag          0×00800000
```

Ilustración 2-8. Reparto de bits por propiedades para codificar un movimiento en un int32

3. Objetivos

En una primera estimación, se establecieron los siguientes objetivos para el desarrollo del motor:

- Jugar al ajedrez en base a las normas de la Federación Internacional de Ajedrez (FIDE).
- Regular la dificultad para adaptarse a jugadores de diferentes niveles de experiencia.
- Buscar las mejores jugadas para un determinado estado del tablero.
- Analizar qué jugador tiene ventaja en una partida y dar un valor a dicha ventaja
- Identificar mates inevitables.
- Interpretar la notación de ajedrez humana y traducirla a notación para ordenadores.
- Trabajar con notación para ordenadores y poder traducirla a notación humana.
- Analizar y realizar un estudio de una partida, con posibles variaciones a una profundidad establecida.
- Estar lo suficientemente optimizado como para poder ejecutarse en dispositivos móviles o consolas portátiles sin sacrificar rendimiento.

Estos objetivos fueron establecidos antes de comenzar el desarrollo, antes de decidir la arquitectura que nuestro motor iba a tener. Por eso la terminología empleada era agnóstica a cómo iba a ser el funcionamiento interno del programa. Una vez desarrollado el motor, los objetivos reales alcanzados son los siguientes

- Representación del tablero como un Bitboard (representación binaria de un número entero de 64 bits sin signo)
- Precálculo de números mágicos con los que indexar los ataques de las piezas que se deslizan por el tablero para distintas ocupaciones
- Precálculo de las tablas de ataques para todas las piezas.
- Interpretación de cadenas FEN para su traducción a la arquitectura de Bitboards.
- Generación de movimientos
    - Movimientos regulares y capturas de las piezas
    - Movimientos especiales (enroque, doble movimiento de peón, captura al paso y promociones)
- Codificación de movimientos en números entero de 32 bits
- Solución Copy/Make para realizar y preservar movimientos
- Evaluación de movimientos por valor de pieza y posición, por lo menos
- Búsqueda Negamax con mejoras
    - Búsqueda quiescente al finalizar Negamax
    - Ordenación por MVV-LVA
- Implementación del protocolo UCI

4. Metodología

En este proyecto, la metodología iba marcada por la metodología de trabajo en la empresa Eclipse Games, que consiste en una metodología ágil basada en SCRUM, aunque he podido adaptarla a mi ritmo de desarrollo. Dado que las funcionalidades de un motor de ajedrez están fuertemente definidas, cada una de ellas se traduce en un *ticket* que resolver.

Una colección de *tickets* que definían una funcionalidad conformaba un Sprint típico de la metodología SCRUM, que se centraba normalmente en completar una funcionalidad a mayor nivel. Un ejemplo de Sprint fue la primera iteración en la que me centré en representar los Bitboards y poder manipularlos, donde algunas funcionalidades incluían definir las funciones para manipular los Bitboards u realizar operaciones simples con los mismos.

Se realizaban reuniones cuando se resolvían funcionalidades mayores (los anteriormente mencionados Sprints), cuya duración era dependiente de la carga de trabajo de las funcionalidades que implementar y del ritmo de desarrollo que pudiera llevar.

4.1. Notion

Notion es una aplicación de escritorio gratuita orientada a la productividad y la gestión de proyectos, entre muchas otras cosas. En concreto, las facilidades que aporta Notion a la organización de este proyecto han sido las siguientes.

4.1.1. Registro del Backlog

Gracias a la funcionalidad de las llamadas "Boards" en Notion, se puede seguir una metodología SCRUM con la que fácilmente podemos generar tickets mediante tableros de Kanban y cambiar su estado fácilmente, así como añadirles metadatos a mi elección. En concreto, para desarrollar el motor de ajedrez, los tickets pasaban por 4 estados.

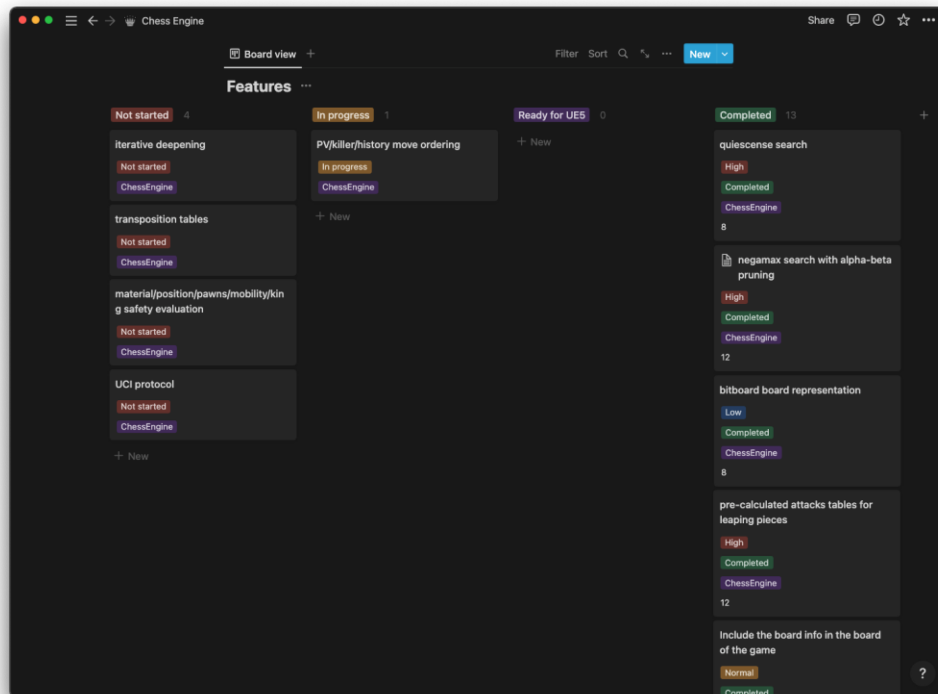

Ilustración 4-1. Board de Notion con las funcionalidades del proyecto

Los estados "Sin comenzar" y "En progreso" sin bastante auto explicativos. El estado "Listo para UE5" significaba que mi versión en C del motor; proyecto donde desarrollaba las funcionalidades en un proyecto monolítico del motor, estaba lista para ser portada a Unreal Engine 5, es decir, ya estaba testeada y verificada para ser portada al motor donde iba a ser usada la funcionalidad desarrollada. Finalmente, Completado significaba que esa funcionalidad ya estaba portada al proyecto de Unreal.

4.1.2. Registro de las reuniones

Otra de las funcionalidades que tiene Notion es poder generar colecciones y lo que ellos llaman bases de datos, que te permiten almacenar información de cualquier tipo para ser visualizada en diferentes formas. Una de ellas es la forma de lista, que me permitía generar "ítems" simplificados de las reuniones que iba manteniendo con mi tutor.

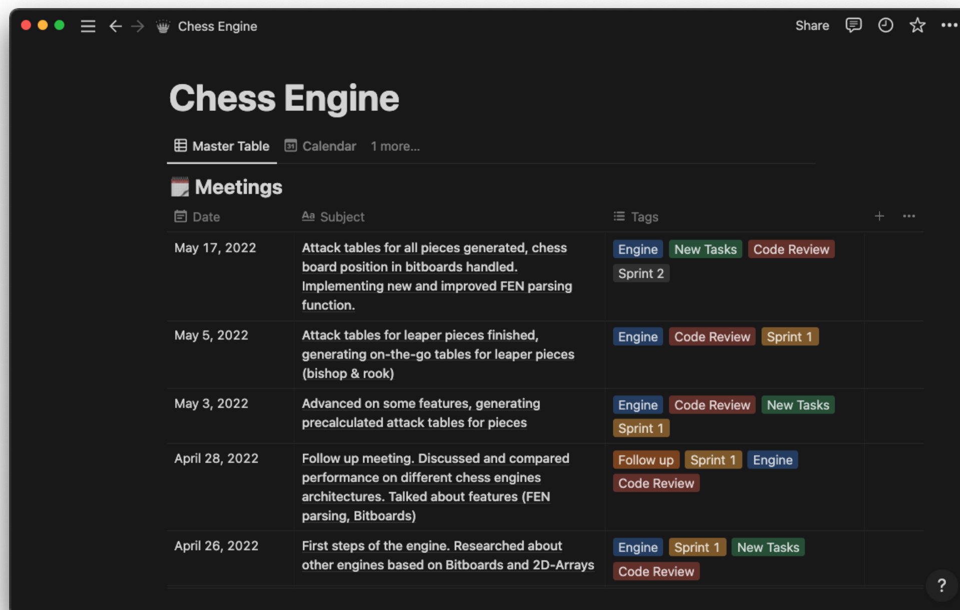

Ilustración 4-2. Listado simple en Notion de las reuniones con mi tutor de proyecto.

4.2. Discord

Discord es una aplicación gratuita de chats de voz, audio y video empleada por millones de personas donde puedes hablar con otros usuarios, grupos de estos o entrar en servidores de toda índole. En este caso, la comunicación en Eclipse Games entre los miembros del equipo se ha realizado siempre mediante el servidor de Discord, y las reuniones que he mantenido con mi tutor han sido siempre por este medio, mediante reuniones de voz donde se compartían resultados mediante la funcionalidad de comparir pantalla que esta plataforma ofrece.

4.3. Metodología ágil para el desarrollo

En Eclipse Games se emplea un desarrollo ágil basado en SCRUM, donde se mantienen reuniones semanales (normalmente 2) en las que se actualiza al equipo del desarrollo.

4.4. Control de versiones (Git)

Tanto para el desarrollo del proyecto paralelo en C como para el proyecto en unreal se ha usado el sistema de control de versiones Git, alojado en GitHub para el primero y en Gogs para el proyecto de Unreal. Este último es un servicio Open Source de auto alojamiento de repositorios de Git al que Eclipse Games ha migrado recientemente.

5. Análisis e implementación

5.1. Funcionalidades implementadas

El listado de funcionalidades que se han desarrollado para el motor han se pueden ver reflejadas en el siguiente listado.

- Representación mediante Bitboards de diferentes partes del motor (posición de piezas, ataques…)
- Tablas precalculadas de ataques para las piezas que saltan de casillas (peones, caballos y el rey)
- Tablas precalculadas de ataques para las piezas que se deslizan por el tablero (alfiles y torres)
  - Cálculos en tiempo real para las piezas que se deslizan
  - Generación de números pseudo-aleatorios para el cálculo de Bitboards Mágicos
  - Bitboards Mágicos para las piezas que se deslizan
- Representación del tablero mediante caracteres ASCII/Unicode con los que poder ver una representación gráfica del estado del tablero para depurar.
- Parseo de una cadena FEN
- Generación de movimientos
  - Promociones de peones con y sin captura
  - Movimiento común y doble de peón
  - Capturas de peón comunes y al paso
  - Enroques
  - Movimientos y capturas del caballo
  - Movimientos y capturas del alfil
  - Movimientos y capturas de la torre
  - Movimientos y capturas de la dama
  - Movimientos y capturas del rey
- Codificación de movimientos en números enteros de 32 bits
- Aproximación Copy/Make para la ejecución de movimientos
- Evaluación de movimientos
- Algoritmo Negamax con Alpha-Beta Pruning para la búsqueda de movimientos

5.2. Arquitectura y estructura del motor

Los motores de ajedrez de diferentes tipos tienen los cimientos muy asentados, siendo los dos pilares principales el evaluador y el buscador de movimientos. Cómo se manejen los datos relacionados al tablero, los movimientos y las piezas depende de la implementación que elijas para los mismos.

En este caso, como ya se ha anunciado con anterioridad, se ha implementado un motor de ajedrez basado en Bitboards, que centra la representación y el manejo de los datos en torno a las piezas y cómo interactúan entre ellas sobre el tablero. A continuación, se puede apreciar el diagrama de clases del motor, que define el esquema (a alto nivel) de las clases y estructuras que forman el motor.

Ilustración 5-1. Diagrama de clases del motor de ajedrez implementado

Aunque aquí se puede apreciar un claro esquema orientado a objetos, esta estructura ha sido fuertemente guiada por la filosofía para C++ de Unreal Engine. Para un motor de ajedrez no son necesarios ciertos niveles de abstracción, como estructurar en una estructura de datos los movimientos; en un principio se manejan codificándolos en enteros de 32 bits, pero para facilitar

su uso en Unreal Engine, se ha optado por quitar complejidad a la hora de programar y optar por seguir los patrones que marca Unreal.

5.3. Representación en Bitboards y utilidades

Los motores basados en Bitboards emplean la representación binaria de un entero de 64 bits sin signo para representar un tablero y el estado de las casillas para una especificación de este, como la ocupación de un tipo de pieza o las casillas defendidas por una pieza situada en una casilla arbitraria. Para poder manejar estos bits como si de un tablero se tratase, debemos realizar operaciones binarias con esos bits.

La manera de trabajar con los bits de estos Bitboards es como si de una lista de bits se tratase, donde accedemos a cada bit por su índice dentro de la supuesta lista, cambiándolo de estado o consiguiendo su valor mediante operaciones binarias de *shifts* u operaciones *AND*. Para realizar todas estas operaciones e intentar facilitar la programación con la nomenclatura del ajedrez, se han creado una serie de utilidades que serán de constante recurrencia a lo largo del desarrollo.

5.3.1. Enumeraciones y constantes

Las enumeraciones han sido un gran aliado a la hora de traducir índices o valores numéricos literales de uso recurrente a un lenguaje de ajedrez comprensible y claro a primera vista. Las enumeraciones principales han sido las que mapean la notación algebraica al índice del Bitboard correspondiente.

```
UENUM()
enum Square
{
    A8, B8, C8, D8, E8, F8, G8, H8,
    A7, B7, C7, D7, E7, F7, G7, H7,
    A6, B6, C6, D6, E6, F6, G6, H6,
    A5, B5, C5, D5, E5, F5, G5, H5,
    A4, B4, C4, D4, E4, F4, G4, H4,
    A3, B3, C3, D3, E3, F3, G3, H3,
    A2, B2, C2, D2, E2, F2, G2, H2,
    A1, B1, C1, D1, E1, F1, G1, H1, NoSquare
}
```

Ilustración 5-2. Enumeración de C++ para identificar las casillas por índice en el tablero

Otra enumeración que además nos permite aplicar lógica con sus valores es la de los enroques. Más adelante se explicarán en detalle las operaciones a realizar, pero estos valores sirven para

poder sumar los valores que haya en el estado del tablero, compararlos con estos y poder saber qué derechos de enroque siguen activos en la partida.

```
enum Castle
{
    WhiteKingSide   = 1,
    WhiteQueenSide  = 2,
    BlackKingSide   = 4,
    BlackQueenSide  = 8
};
```

Ilustración 5-3. Enumeración de los enroques orientada a operar entre ellos para obtener los enroques disponibles.

La última enumeración por destacar es la que define las piezas y las enumera por índices. Esta nos facilita iterar sobre las piezas y realizar lógica común a todas las piezas especificada para cada una de ellas.

```
enum PieceCode
{
    WhitePawn   = 0,
    WhiteKnight = 1,
    WhiteBishop = 2,
    WhiteRook   = 3,
    WhiteQueen  = 4,
    WhiteKing   = 5,
    BlackPawn   = 6,
    BlackKnight = 7,
    BlackBishop = 8,
    BlackRook   = 9,
    BlackQueen  = 10,
    BlackKing   = 11,
};
```

Ilustración 5-4. Enumeración de las piezas ordenadas

A parte de estas, hay más enumeraciones que se emplean a lo largo de todo el proyecto, como enumerar los colores del ajedrez, el tipo de movimiento (captura o movimiento sin captura) o el tipo de pieza. Estas son las más diferenciales, sobre todo la enumeración de los enroques.

5.3.2. Operaciones de bits

Las operaciones de bits son esenciales para que este proyecto se lleve a cabo; sin ellas no podríamos alterar el estado de los Bitboards (o sería muy tedioso y contraintuitivo). Con estas operaciones debemos poder manipular el estado de bits específicos de los Bitboards y extraer recursos e información que poder traducir a lenguaje ajedrecístico. Estas tres operaciones son:

- SetBit: nos permite estableces el estado de un bit a activo en un Bitboard, independientemente del estado en el que estaba antes de realizar la operación.
- PopBit nos permite establecer el estado de un bit a inactivo en un Bitboard, independientemente del estado en el que estaba antes de realizar la operación.
- GetBit: nos permite conocer el estado de un bit sin alterar su estado.

Estas operaciones son relativamente sencillas y cortas, pero, para evitar la repetición del código que realiza estas operaciones, se han encapsulado en unas funciones *Macro* [14] encargadas de realizar estas operaciones de manera *in-line*. Estas funciones Macro quedan implementadas de la siguiente manera [15]:

```
#define SetBit(bitboard, square) ((bitboard) |=  (1ULL << (square)))
#define GetBit(bitboard, square) ((bitboard) &   (1ULL << (square)))
#define PopBit(bitboard, square) ((bitboard) &= ~(1ULL << (square)))
```

Ilustración 5-5. Funciones Macro para activar, desactivar y leer bits dentro de un Bitboard.

Como podemos ver, la operación encargada de activar el bit (SetBit), realiza una operación lógica OR entre el Bitboard que se le pasa como parámetro y otro Bitboard, este último formado por 64 bits desactivados, a los que se le hace una operación de bits LSHIFT tantas veces como la casilla que se quiere activar. A continuación, podemos ver unos ejemplos de cómo operan estas funciones.

Bitboard	1010111010011101011011010101011101010110010110101011011000111011
Square G7 (14)	0000000000000000000000000000000000000000000000000100000000000000
Bitboard OR= G7	1010111010011101011011010101011101010110010110101111011010111011

En la tabla, hemos visto como para activar (SetBit) el bit situado en la casilla G7, hemos montado un Bitboard con únicamente el bit en el índice 14 activo ((1 << G7), o lo que es lo mismo (1 <<

14 Los Macros son fragmentos de código que pueden ser nombrados y, al ser referenciados, son reemplazados con el contenido definido en el Macro. Las funciones Macro son especificaciones de estos Macros a los cuales se les pueden pasar variables y ser utilizadas de manera literal.

15 Estas funciones han sido analizadas y estudiadas para su implementación de (Chess Programming Wiki, n.d.)

14) según el enumerador que declaramos anteriormente) y le hemos aplicado al Bitboard original una operación OR sobre sí mismo, activando así el bit si estaba inactivo o manteniendo su estado si ya lo hubiera estado.

Bitboard	1010111010011101011011010101011101010110010110101011011000111011
Square B4 (28)	0000000000000000000000000000000000010000000000000000000000000000
Bitboard AND B4	0000000000000000000000000000000000010000000000000000000000000000

Para conseguir el estado de un bit en un Bitboard (GetBit), debemos hacer una operación AND montando el bit de la casilla de la misma manera que en la operación anterior.

Bitboard	1010111010011101011011010101011101000110010110101011011000111011
Square F3 (45)	0000000000000000010000000000000000000000000000000000000000000000
~F3	1111111111111111101111111111111111111111111111111111111111111111
Bitboard AND= ~F3	1010111010011101001011010101011101000110010110101011011000111011

Para esta última operación, si queremos desactivar el estado del bit en F3 (índice 45), debemos primero negar el Bitboard obtenido de la casilla y, finalmente, aplicar una operación lógica AND con el Bitboard original. De esta manera, la operación AND dejará desactivado el bit indiferentemente si antes estuviera activo o no.

Por último, dos utilidades relevantes que nos ayudan a saber más información sobre los Bitboards son la cuenta de bits activos en un Bitboard (BitCount) y la posición del Bit Menos Significativo [16] (Least-Significant Bit, o LSB). Para poder calcular y obtener estos datos, se ha optado por unas funciones *inline* [17] para su implementación dado que tienen una complejidad un tanto más elevada como para encapsularlas en unas funciones Macro.

```
FORCEINLINE int BitCount(uint64 bitboard)
{
    int count = 0;
    while (bitboard)
    {
        bitboard &= (bitboard - 1);
        ++count;
    }
    return count;
};
```

Ilustración 5-6. Implementación de la función para contar bits activos en un Bitboard

16 En la informática, el LSB (siglas de Bit Menos Significativo en inglés) es el bit activo en la posición de más hacia la derecha de la representación binaria del dato a representar.

17 Las funciones *inline* son funciones cuyas llamadas son sustituidas en tiempo de compilación por el compilador en tiempo de compilación

Para la cuenta de bits activos, realizamos una operación and con el mismo Bitboard tras restarle 1 (decimal). Esta última operación intercambia los estados del bit activo más a la derecha, y de todos los ceros a la derecha de este. De esta manera, cuando realicemos la operación AND, sólo quedarán los bits activos a la izquierda del anterior bit activo más a la derecha. Finalmente, mantenemos un contador de las veces que tenemos que realizar esta operación y ya tenemos la cuenta de los bits activos. A continuación, se muestra el proceso de una manera más visual.

Bitboard (B)	0000000010000100001011010001000101010000010000101000001000101000	
B - 1	0000000010000100001011010001000101010000010000101000001000100111	
B &= (B – 1)	0000000010000100001011010001000101010000010000101000001000100000	1
B - 1	0000000010000100001011010001000101010000010000101000001000011111	
B &= (B – 1)	0000000010000100001011010001000101010000010000101000001000000000	2
B - 1	0000000010000100001011010001000101010000010000101000000111111111	
B &= (B – 1)	0000000010000100001011010001000101010000010000101000000000000000	3
…	…	…
B	0000000010000000000000000000000000000000000000000000000000000000	15
B - 1	0000000001111111111111111111111111111111111111111111111111111111	
B &= (B – 1)	0000000000000000000000000000000000000000000000000000000000000000	16

Para conseguir el LSB (esta será la nomenclatura para el Bit Menos Significativo) de un Bitboard, el algoritmo es bastante más abstracto, y hace uso de la función que cuenta bits descrita justo arriba. De nuevo, se hace uso de una función *in-line*.

```
FORCEINLINE int LSBIndex(uint64 bitboard)
{
    if (bitboard)
    {
        return BitCount((bitboard & -bitboard) - 1);
    }
    else
    {
        return -1;
    }
};
```

Ilustración 5-7. Implementación de la función para encontrar el índice del LSB

Este algoritmo, como podemos observar, emplea la función de cuenta de bits pasándole como argumento una manipulación del Bitboard del que obtener el índice del LSB. El algorirmo se puede ver representado de una manera gráfica en la siguiente tabla.

Bitboard [B]	0000000010000100001011010001000101010000010000101000001000101000
-B	1111111101111011110100101110111010101111101111010111110111011000
B & -B	0000000000000000000000000000000000000000000000000000000000001000
(B & -B) - 1	0000000000000000000000000000000000000000000000000000000000000111
Count((B & -B)-1)	3

Como se puede apreciar, estando cada bit del Bitboard indexado desde la derecha comenzando desde el índice cero, la función ha conseguido calcular que el LSB está en la posición 3.

5.4. Tablas de ataques

Las tablas de ataques son uno de los principales usos de los Bitboards en los motores basados en esta arquitectura, y es por estas por las que se dice que los motores basados en tablas de bits son motores centrados en las piezas.

Como ya se ha comentado antes, los Bitboards no sólo sirven para representar las piezas en el tablero, sino cualquier información sobre el mismo que implique diferenciar entre las casillas relevantes y las que no lo son. Estas tablas de ataques son precisamente eso, máscaras de bits que nos permiten diferenciar unas casillas de otras; las que están atacadas por una pieza desde una casilla y las que no.

Estas tablas de ataques se precalculan al comienzo de la ejecución, aunque podrían ser precalculadas en una primera ejecución e incrustadas en el código una vez calculadas. No obstante, dada la alta capacidad computacional actual de los microprocesadores esto no es para nada necesario, ya que todos los cálculos para generar el tablero no tomarían más de unos pocos segundos; influyendo prácticamente nada en el desempeño del motor a la hora de buscar y evaluar movimientos.

Para generar dichas tablas, debemos emplear Bitboards y someterlos a operaciones binarias como LSHIFT y RSHIFT junto con nuestras funcionalidades previamente descritas. Dentro de estas tablas podemos diferenciar claramente entre las tablas de las piezas que saltan de casilla en casilla y las de las piezas que se deslizan por el tablero, siendo estas últimas las que más memoria (con diferencia) necesitan para almacenar sus tablas de movimientos.

Estas tablas de ataques son, en esencia e implementación, listas de Bitboards que mantienen únicamente activos todos aquellos bits a los que una pieza puede llegar desde una casilla cualquiera (el bit de origen).

La clave para formar las tablas de ataque es emplear las antes mencionadas operaciones LSHIFT y RSHIFT, iterando sobre los bits de un Bitboard y activando los bits a los que una supuesta pieza desde la casilla en el índice de la iteración actual puede llegar. Un factor común entre todas las tablas es que en alguna dimensión de esta tabla, figura el número de casillas total del tablero; 64 casillas, 64 elementos en alguna (si no en la única) de las dimensiones de la tabla que almacena los Bitboards.

5.4.1. Peones

La tabla de ataque de los peones es la más curiosa de todas las de las piezas que saltan, ya que ocupa el doble que las del caballo y el rey. Esto se debe a que los peones son las únicas piezas que tienen un movimiento en una única dirección y que además depende del lado hacia el que se muevan. Por tanto, no hay manera de hacer una única tabla de 64 Bitboards, sino que tenemos que agrupar 2 colecciones de 64 Bitboards, una para cada color de peones.

Como ya se ha explicado, cada uno de los 64 Bitboards de cada uno de los colores define las casillas a las que un peón de un color puede atacar, siendo esta casilla el índice dentro de la lista de 64 bits.

Por ejemplo, el Bitboard del índice 12, casilla E7, de por ejemplo los peones negros, tiene activos los bits de las casillas que están al alcance de un ataque de un peón negro desde la casilla E7, siendo estas las casillas D6 y F6.

	A	B	C	D	E	F	G	H
8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	0	1	0	1	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0

Ilustración 5-8. Representación gráfica de los bits del Bitboard de los ataques de un peón negro en E7, así como el de un peón blanco en E5

Ahora que tenemos claro visualmente qué son los Bitboards, debemos encontrar una manera de precalcular estos Bitboards y rellenar nuestra lista bidimensional de 2x64 Bitboards. Para ello, iteraremos por todas las casillas del tablero y guardaremos en el índice el Bitboard que corresponda, tras haber activado los bits pertinentes.

```
uint64 FChessAttacks::MaskPawnAttacks(int side, int square)
{
    uint64 pawn = 0ULL;
    SetBit(pawn, square);

    uint64 attacks = 0ULL;

    if (side == White)
    {
        if ((pawn >> 7) & NotAFile) attacks |= (pawn >> 7);
        if ((pawn >> 9) & NotHFile) attacks |= (pawn >> 9);
    }
    else
    {
        if ((pawn << 7) & NotHFile) attacks |= (pawn << 7);
        if ((pawn << 9) & NotAFile) attacks |= (pawn << 9);
    }

    return attacks;
}
```

Ilustración 5-9. Algoritmo para calcular el Bitboard de ataques de un peón de un color en una casilla.

En este algoritmo podemos ver que nuestra función acepta dos parámetros, el lado del peón para el que calcular los ataques y la casilla sobre la cual descansa nuestro peón imaginario. De esta manera, inicializamos un Bitboard (uint64) y activamos el bit en la casilla que hemos recibido por parámetro, y declaramos el Bitboard que guarde nuestros ataques.

Condicionalmente, debemos hacer desplazamientos de 7 y 9 bits a la derecha, si somos piezas blancas, o a la izquierda, en caso contrario. Además, debemos comprobar que, en el momento de hacer los desplazamientos, no nos salgamos del tablero y aparezcamos en el lado opuesto.

Estas comprobaciones se consiguen con unas constantes declaradas como Bitboards que nos permiten evitar rebases laterales. Esta comprobación es una operación AND debido a que tanto NotAFile como NotHFile son Bitboards con todos los bits activos menos los que, representados como una matriz de 8x8 bits, corresponderían con los ubicados en dichas columnas, que permanecen desactivados.

MaskPawnAttacks(white, E5)
A5 >> 9 = H7 ⚠

	A	B	C	D	E	F	G	H
8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0

AND

NotHFile

	A	B	C	D	E	F	G	H
8	1	1	1	1	1	1	1	0
7	1	1	1	1	1	1	1	0
6	1	1	1	1	1	1	1	0
5	1	1	1	1	1	1	1	0
4	1	1	1	1	1	1	1	0
3	1	1	1	1	1	1	1	0
2	1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1	0

=

Zero!! == false

	A	B	C	D	E	F	G	H
8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0

Ilustración 5-10. Ejemplo de utilidad de NotHFile en un caso de uso donde es necesaria la comprobación

Cabe destacar que a lo mejor la representación de la Ilustración 5-10 puede causar confusión, dado que se ha realizado un desplazamiento de bit a la derecha, pero en la representación nos hemos desplazado hacia la izquierda. Recordemos que el índice 0 pertenece a la casilla A8, no a A1, así que los desplazamientos en las representaciones que veamos de ese tipo serán opuestas al desplazamiento teórico.

Aquí podemos ver la definición de los Bitboards NotAFile y NotHFile.

```
/// <summary>
/// Bitboard with all bits set except for the A file.
/// 0 1 1 1 1 1 1 1
/// 0 1 1 1 1 1 1 1
/// 0 1 1 1 1 1 1 1
/// 0 1 1 1 1 1 1 1
/// 0 1 1 1 1 1 1 1
/// 0 1 1 1 1 1 1 1
/// 0 1 1 1 1 1 1 1
/// 0 1 1 1 1 1 1 1
/// </summary>
const uint64 NotAFile = 18374403900871474942ULL;

/// <summary>
/// Bitboard with all bits set except for the H file.
/// 1 1 1 1 1 1 1 0
/// 1 1 1 1 1 1 1 0
/// 1 1 1 1 1 1 1 0
/// 1 1 1 1 1 1 1 0
/// 1 1 1 1 1 1 1 0
/// 1 1 1 1 1 1 1 0
/// 1 1 1 1 1 1 1 0
/// 1 1 1 1 1 1 1 0
/// </summary>
const uint64 NotHFile = 9187201950435737471ULL;
```

Ilustración 5-11. Bitboards "helpers" para condicionar la activación de bits al generar ataques que pueden atravesar los laterales del tablero

5.4.2. Caballos

Para los caballos solo necesitaremos una lista de 64 Bitboards, uno para cada casilla. Esto se debe a que, para cada casilla, las casillas a las que se puede mover el caballo son las mismas independientemente del color del que sea el caballo (de igual manera para el resto de las piezas a partir de ahora, aunque veremos que esto no es del todo cierto).

Como ya se ha explicado con antelación, el movimiento del caballo está compuesto por dos movimientos en una dirección en vertical u horizontal y un movimiento en la restante. ¿En qué se traduce esto a la hora de realizar los *shifts* de bits? En que un caballo puede caer en un mínimo de 2 casillas y un máximo de 8, dependiendo de la casilla en la que esté. La función que nos genera las máscaras de ataque del caballo (de cualquier color) para una casilla se ha implementado de la siguiente manera.

```
uint64 FChessAttacks::MaskKnightAttacks(int square) {
    uint64 knight = 0ULL;
    SetBit(knigth, square);

    uint64 attacks = 0ULL;

    attacks |= ((knight >> 17) & NotHFile);
    attacks |= ((knight >> 15) & NotAFile);
    attacks |= ((knight >> 10) & NotGHFile);
    attacks |= ((knight >> 6)  & NotABFile);

    attacks |= ((knight << 17) & NotAFile);
    attacks |= ((knight << 15) & NotHFile);
    attacks |= ((knight << 10) & NotABFile);
    attacks |= ((knight << 6)  & NotGHFile);

    return attacks;
}
```

Ilustración 5-12. Implementación del algoritmo para calcular el Bitboard de ataques del caballo para una casilla

De esta manera, los caballos podrían atacar hasta 8 casillas simultáneamente desde una casilla que lo permita. De nuevo, marcaríamos un Bitboard *helper* para conocer la posición del caballo y realizamos desplazamientos a la derecha y a la izquierda 17, 15, 10 y 6 veces para obtener las hasta 8 casillas atacadas por el caballo.

	A	B	C	D	E	F	G	H
8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	1	0	1	0	0	0
5	0	1	0	0	0	1	0	0
4	0	0	0	0	0	0	0	0
3	0	1	0	0	0	1	0	0
2	0	0	1	0	1	0	0	0
1	0	0	0	0	0	0	0	0

Ilustración 5-13. Ataques de un caballo de cualquier color situado en la casilla D4

La misma lógica de comprobar que no acabamos en columnas desfasadas lateralmente aparece de nuevo, pero esta vez vemos que se han añadido dos constantes más que, gracias a su nombre descriptivo, podemos inferir que definen unos Bitboards con todos los bits activos menos los que caerían ubicados en las parejas de columnas contiguas AB y HF. Esto es porque el caballo podría estar situado en la una de las dos filas laterales, por ejemplo, A, y se desplazase 2 casillas hacia la derecha y una hacia arriba (A4 >> 10, por ejemplo, terminando teóricamente en G6).

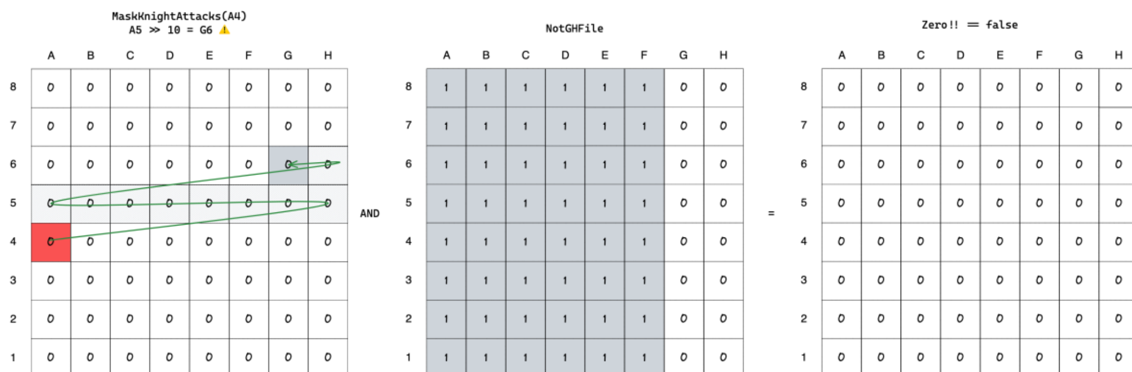

Ilustración 5-14. Ejemplo visual de la utilidad del Bitboard NotGHFile para evitar desplazamientos ilegales

5.4.3. Rey

Dado que el rey solo puede moverse una casilla en cualquiera de las 8 direcciones, puede estar defendiendo entre 3 y 8 casillas al mismo tiempo (sin tener en cuenta jaques o ataques del rival). También destacar que las casillas atacadas desde una misma casilla son las mismas para los reyes de ambos colores, así que la lista de Bitboards de ataques también es unidimensional de tamaño 64.

Como ya hemos visto en las otras dos piezas de salto de casilla, para generar las tablas de ataques del rey también deberemos emplear un algoritmo que utilice las comprobaciones laterales, aunque en este caso de nuevo, como con los peones, únicamente las de las columnas A y H.

```
uint64 FChessAttacks::MaskKingAttacks(int square) {
    uint64 king = 0ULL;
    SetBit(king, square);

    uint64 attacks = 0ULL;

    attacks |= (king >> 8);
    attacks |= ((king >> 9) & NotHFile);
    attacks |= ((king >> 7) & NotAFile);
    attacks |= ((king >> 1) & NotHFile);

    attacks |= (king << 8);
    attacks |= ((king << 9) & NotAFile);
    attacks |= ((king << 7) & NotHBFile);
    attacks |= ((king << 1) & NotAHFile);

    return attacks;
}
```

Ilustración 5-15. Implementación del algoritmo para generar los ataques del rey para una casilla

De igual manera que con las piezas anteriores, este algoritmo nos genera un Bitboard con los bits atacados por el rey colocado en cualquier casilla del tablero.

	A	B	C	D	E	F	G	H
8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	0	1	1	1	0
3	0	0	0	0	1	0	1	0
2	0	0	0	0	1	1	1	0
1	0	0	0	0	0	0	0	0

Ilustración 5-16. Representación del Bitboard de ataques generado para un rey situado en la casilla F3

5.4.4. Alfiles

Los alfiles son la primera de las tres piezas deslizantes del ajedrez. Esta pieza se mueve en diagonal, el cualquiera de las 4 diagonales que tenga disponibles. ¿Qué problemática introducen las piezas que se deslizan por el tablero? Pues que una máscara de ataques no es fiel a un estado arbitrario de ocupación en el que se encuentre el tablero.

Esto significa que si generásemos una máscara de movimientos como lo hemos hecho hasta ahora, no podríamos saber si todos los bits del Bitboard de ataques calculado podrían ser alcanzados por el alfil, ya que una pieza aliada o enemiga podría estar situada entre el alfil y una de las casillas que, supuestamente, puede alcanzar.

Máscara de ataques de un alfil blanco situado en B2, sin tener en cuenta ocupaciones

	A	B	C	D	E	F	G	H
8	0	0	0	0	0	0	0	1
7	0	0	0	0	0	0	1	0
6	0	0	0	0	0	1	0	0
5	0	0	0	0	1	0	0	0
4	0	0	0	1	0	0	0	0
3	1	0	1	0	0	0	0	0
2	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0

Ejemplo de posible ocupación arbitraria de las piezas blancas

	A	B	C	D	E	F	G	H
8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0
3	0	1	0	0	1	1	0	0
2	1	1	1	0	0	1	1	1
1	1	1	0	1	0	1	1	0

Ilustración 5-17. Ejemplo de Bitboard de ataques de alfil creando falsos ataques en la diagonal de B2 a H8, siendo esta bloqueada en D4 por otra pieza blanca

Como se puede discernir en la Ilustración 5-17, un alfil blanco situado en B2 a priori (empleando algoritmos como los descritos anteriormente) genera un Bitboard de ataques como el de la

izquierda; a la derecha se muestra un Bitboard que representa la ocupación de las piezas blancas, y es completamente arbitraria. El alfil de B2, según el Bitboard que se genera para esa casilla, le está prometiendo al alfil que puede atacar la casilla G7 (por ejemplo, pero vale cualquiera en esa diagonal a partir de D4), mientras que realmente esto no es así, ya que otra pieza blanca situada en D4 le corta el paso, generando una inconsistencia entre lo que las tablas de ataques prometen y la realidad.

Para solucionar esta inconsistencia se hace uso de un concepto diferencial en los motores de ajedrez basados en Bitboards, los llamados Números Mágicos (Magic Bitboards). Estos números tienen la capacidad de asociar una ocupación a una clave perfecta, la cual podemos utilizar para precalcular todas las combinaciones de ocupaciones (al menos las relevantes) y generar Bitboards de ataques de alfiles fieles al estado en que se encuentre el tablero en cualquier momento. Estos números serán explicados con mayor detenimiento más adelante en esta memoria.

Para poder, finalmente, generar una tabla de ataques real para los alfiles, debemos emplear una lista bidimensional bastante grande. En concreto, la tabla debe poder mantener una matriz de datos de 64 casillas por 512 posibles ocupaciones.

Todo el proceso de generación de esta tabla comienza como con las piezas anteriores, generando una máscara "inconsistente" para los ataques que un alfil cualquiera puede hacer desde una casilla.

```
uint64 FChessAttacks::MaskBishopAttacks(int square)
{
    uint64 attacks = 0ULL;

    int r, f;
    int tr = square / 8;
    int tf = square % 8;

    for (r = tr + 1, f = tf + 1; r <= 6 && f <= 6; r++, f++) attacks |= (1ULL << (r * 8 + f));
    for (r = tr - 1, f = tf + 1; r >= 1 && f <= 6; r--, f++) attacks |= (1ULL << (r * 8 + f));
    for (r = tr + 1, f = tf - 1; r <= 6 && f >= 1; r++, f--) attacks |= (1ULL << (r * 8 + f));
    for (r = tr - 1, f = tf - 1; r >= 1 && f >= 1; r--, f--) attacks |= (1ULL << (r * 8 + f));

    return attacks;
}
```

Ilustración 5-18. Implementación de la función capaz de generar una máscara de ataques para un alfil en una casilla cualquiera

Si analizamos el algoritmo con detenimiento, podemos ver que cada bucle avanza en una diagonal distinta, activando los bits de las 4 diagonales de una en una.

Una vez se ha generado una lista de 64 Bitboards con las máscaras de ataques, se ha implementado una función que calcula los ataques de un alfil en tiempo real; teniendo en cuenta tanto la casilla en la que se encuentra el alfil y las casillas bloqueadas en el tablero.

```
uint64 FChessAttacks::RealTimeBishopAttacks(int square, uint64 blocked)
{
    uint64 attacks = 0ULL;

    int r, f;
    int tr = square / 8;
    int tf = square % 8;

    for (r = tr + 1, f = tf + 1; r <= 7 && f <= 7; r++, f++)
    {
        attacks |= (1ULL << (r * 8 + f));
        if ((1ULL << (r * 8 + f)) & blocked)
            break;
    }

    for (r = tr - 1, f = tf + 1; r >= 0 && f <= 7; r--, f++)
    {
        attacks |= (1ULL << (r * 8 + f));
        if ((1ULL << (r * 8 + f)) & blocked)
            break;
    }

    for (r = tr + 1, f = tf - 1; r <= 7 && f >= 0; r++, f--)
    {
        attacks |= (1ULL << (r * 8 + f));
        if ((1ULL << (r * 8 + f)) & blocked)
            break;
    }

    for (r = tr - 1, f = tf - 1; r >= 0 && f >= 0; r--, f--)
    {
        attacks |= (1ULL << (r * 8 + f));
        if ((1ULL << (r * 8 + f)) & blocked)
            break;
    }

    return attacks;
}
```

Ilustración 5-19. Implementación de la función para generar los ataques de un alfil en tiempo real, teniendo en cuenta la casilla en la que se encuentra y la ocupación del tablero

Esta función es muy similar a la anterior. La principal diferencia es que, en esta función, cada diagonal para de iterar en cuanto colisiona con el Bitboard de la ocupación (operando con un AND, si es distinto de 0 para de iterar y esa diagonal ya ha terminado de generar ataques).

La parte final de la generación de las piezas se explicará cuando se llegue a la Dama, ya que es común entre alfil y torre y los ataques de la dama realmente son la unión de los ataques de alfiles y torres.

5.4.5. Torres

Las torres presentan la misma inconsistencia que los alfiles. Esto nos facilita encontrar la solución, seguir la solución aplicada a los alfiles.

La generación de las máscaras de ataques de la torre y la del alfil difieren solo en las direcciones sobre la que se itera.

```
uint64 FChessAttacks::MaskRookAttacks(int square)
{
    uint64 attacks = 0ULL;

    int r, f;
    int tr = square / 8;
    int tf = square % 8;

    for (r = tr + 1; r <= 6; r++) attacks |= (1ULL << (r * 8 + tf));
    for (r = tr - 1; r >= 1; r--) attacks |= (1ULL << (r * 8 + tf));
    for (f = tf + 1; f <= 6; f++) attacks |= (1ULL << (tr * 8 + f));
    for (f = tf - 1; f >= 1; f--) attacks |= (1ULL << (tr * 8 + f));

    return attacks;
}
```

Ilustración 5-20. Implementación de la función para calcular los Bitboards de ataque de las torres para una casilla cualquiera

De la misa manera, también se deben calcular los ataques de la torre en tiempo real dependiendo de la ocupación que presente el tablero. El algoritmo para ello es, igual que en el alfil, muy similar al de las tablas normales, simplemente parando en cuanto encuentras una pieza bloqueando el camino de la torre.

```
uint64 FChessAttacks::RealTimeBishopAttacks(int square, uint64 blocked)
{
    uint64 attacks = 0ULL;

    int r, f;
    int tr = square / 8;
    int tf = square % 8;

    for (r = tr + 1, f = tf + 1; r <= 7 && f <= 7; r++, f++)
    {
        attacks |= (1ULL << (r * 8 + f));
        if ((1ULL << (r * 8 + f)) & blocked)
            break;
    }

    for (r = tr - 1, f = tf + 1; r >= 0 && f <= 7; r--, f++)
    {
        attacks |= (1ULL << (r * 8 + f));
        if ((1ULL << (r * 8 + f)) & blocked)
            break;
    }

    for (r = tr + 1, f = tf - 1; r <= 7 && f >= 0; r++, f--)
    {
        attacks |= (1ULL << (r * 8 + f));
        if ((1ULL << (r * 8 + f)) & blocked)
            break;
    }

    for (r = tr - 1, f = tf - 1; r >= 0 && f >= 0; r--, f--)
    {
        attacks |= (1ULL << (r * 8 + f));
        if ((1ULL << (r * 8 + f)) & blocked)
            break;
    }

    return attacks;
}
```

Ilustración 5-21. Implementación de la función para generar en tiempo real los ataques de una torre en una casilla arbitraria dada una ocupación

5.4.6. Dama

Para la generación de movimientos de la dama, lo que hacemos es emplear todos los recursos que ya hemos calculado para el alfil y la torre y los unimos, de tal forma que con realizar una operación OR entre los ataques del alfil y la torre para la misma casilla podemos obtener los ataques de la dama.

Ahora bien, ya que tenemos las funciones que nos generan estos Bitboards con los ataques para las piezas que se deslizan, debemos inicializarlos. Para ello, declararemos una función con la que calcular la ocupación para poder calcular el ataque en tiempo real.

```
uint64 FChessAttacks::SetOccupancy(int index, int bits, uint64 attack)
{
    uint64 occupancy = 0ULL;

    for (int count = 0; count < bits; count++)
    {
        int square = LSBIndex(attack);
        PopBit(attack, square);

        if (index & (1 << count))
            occupancy |= (1ULL << square);
    }

    return occupancy;
}
```

Ilustración 5-22. Función para calcular una combinación de ocupaciones para una pieza deslizante

Este método es el que se explica con más detenimiento en el apartado del Estado del arte Magic Bitboards, esta es la implementación para ese algoritmo. Una vez calculada esta ocupación, ya podemos inicializar nuestras piezas deslizantes.

```
void FChessAttacks::InitializeSliderPiecesAttacks(bool isBishop)
{
    for (int square = 0; square < 64; square++)
    {
        BishopMasks[square] = MaskBishopAttacks(square);
        RookMasks[square] = MaskRookAttacks(square);

        uint64 attackMask = isBishop ? BishopMasks[square] : RookMasks[square];
        int relevantBitsCount = BitCount(attackMask);
        int occupancyIndices = (1 << relevantBitsCount);

        for (int index = 0; index < occupancyIndices; index++)
        {
            uint64 occupancy = SetOccupancy(index, relevantBitsCount, attackMask);
            int magicIndex = isBishop ?
                (occupancy * BishopMagicNumbers[square]) >> (64 - BishopRelevantBits[square]) :
                (occupancy * RookMagicNumbers[square]) >> (64 - RookRelevantBits[square]);

            if (isBishop)
            {
                BishopAttacks[square][magicIndex] = RealTimeBishopAttacks(square, occupancy);
            }
            else
            {
                RookAttacks[square][magicIndex] = RealTimeRookAttacks(square, occupancy);
            }
        }
    }
}
```

Ilustración 5-23. Implementación de la función de inicialización de las tablas de ataques de todas las piezas deslizantes (alfiles y torres, dama obtenida de la unión de ambas)

En cada iteración se genera la ocupación del tablero en función de la casilla que se ocupa, unos bits relevantes que varían dependiendo de si calculamos los ataques de un alfil o de una torre, y la máscara de ataques precalculada.

Finalmente, conseguimos inicializar las tablas de ataques de alfil y torres de manera precalculada, completando así todas las tablas necesarias para poder generar movimientos.

5.5. Bitboards y Ocupaciones

Ahora que tenemos una forma de representar los ataques, necesitamos un tablero donde buscar movimientos en base a estos. Para ello, nos basta con declarar una lista de 12 Bitboards que almacenen como bits activos las posiciones de cada tipo y color de pieza (6 piezas blancas y 6 negras).

Partiendo de la base de que vamos a guardar las posiciones de las piezas, podemos facilitarnos la tarea de la generación de las ocupaciones previamente mencionadas si, además, guardamos otra lista, en este caso de 3 elementos, en la que guardar las ocupaciones de cada color y en la última una combinación de estas dos.

```
class MAGICCHESS_API FChessBoard : UObject
{
// ...
public:
    uint64 Bitboards[12];
    uint64 Occupancies[3];
// ...
}
```

Ilustración 5-24. Declaración de las listas para almacenar los Bitboards para la ocupación de cada pieza y las ocupaciones por colores

La lista de Bitboards será indexada empleando la enumeración para los códigos de pieza descrita en el anexo de Enumeraciones y constantes, y la de ocupaciones empleando la enumeración de 3 elementos para listar los dos colores y una combinación de ambos.

Para inicializar estas listas, se ha implementado un algoritmo de parseo bastante específico y adaptado para el parseo de cadenas FEN, cuya aproximación es bastante directa y concisa.

5.6. Parseo de cadenas FEN

El parseo de una cadena FEN es algo difícil de optimizar y que, además, tampoco debería ser especialmente prioritario preocuparse por que el método de parseo sea especialmente eficiente. A no ser que se implemente de una manera completamente ineficiente, es difícil hoy en día hacer un método de parseo de este tipo de cadenas que realmente funcione significativamente lento.

Para parsear la cadena, primero debemos restaurar a un estado vacío nuestras listas. Yo he utilizado la función de la librería estándar "memset", que te permite llenar un espacio de memoria con el valor que le pases como parámetro. Una vez se ha limpiado la posible basura en la memoria a emplear, también se emplean valores neutrales para el resto de las propiedades que parsear, como el lado que juega, la casilla abierta a captura al paso o los enroques disponibles. Para terminar con la preparación de los datos, se transforma el tipo de dato que Unreal Engine emplea para manejar cadenas de caracteres (llamado FString) a un puntero a caracteres ("char" en C/C++) para ir incrementando el puntero y poder parsear de una manera más cómoda.

```
void FChessBoard::ParseFEN(FString fenString)
{
    // clear memory & initialize FEN properties
    memset(Bitboards, 0ULL, sizeof(Bitboards));
    memset(Occupancies, 0ULL, sizeof(Occupancies));
    Side = White;
    OpenEnpassant = NoSquare;
    AvailableCastlings = 0;

    char* fen = TCHAR_TO_ANSI(*fenString);

    // ...
}
```

Ilustración 5-25. Inicialización de las propiedades relativas a la cadena FEN para su parseo.

A continuación, como vimos previamente en el anexo Cadena FEN, lo primero que nos encontramos al intentar parsear son los posicionamientos de las piezas en el tablero, siguiendo la notación FEN. Lo único difícil de esta tarea es organizar bien el algoritmo para evitar malos parseos. La aproximación que se lleva a cabo es simular que atraviesas el tablero casilla por casilla. Cada vez que visitamos un carácter, este puede ser o bien una letra representando el tipo de pieza que ocupa la casilla, un dígito o un carácter de barra inclinada. El carácter hace referencia a la pieza que ocupa la casilla por la que se está iterando, el dígito significa el número de casillas en blanco consecutivas desde la casilla sobre la que se itera, y la barra quiere decir un salto de una fila a la siguiente. Este algoritmo nos rellena los Bitboards de cada pieza.

```
void FChessBoard::ParseFEN(FString fenString)
{
    // ...

    for (int rank = 0; rank < 8; rank++)
    {
        for (int file = 0; file < 8; file++)
        {
            int square = rank * 8 + file;
            if ((*fen >= 'a' && *fen <= 'z') || (*fen >= 'A' && *fen <= 'Z'))
            {
                int piece = CharToPieceCode(*fen);
                SetBit(Bitboards[piece], square);
                fen++;
            }
            if (*fen >= '0' && *fen <= '9')
            {
                int offset = *fen - '0';
                int piece = -1;
                for (int bb = PieceCode::WhitePawn; bb <= PieceCode::BlackKing; bb++)
                {
                    if (GetBit(Bitboards[bb], square))
                    {
                        piece = bb;
                    }
                }
                if (piece == -1)
                    file--;
                file += offset;
                fen++;
            }
            if (*fen == '/')
                fen++;
        }
    }

    // ...
}
```

Ilustración 5-26. Algoritmo de parseo de posicionamiento de las piezas de una cadena FEN

Después se parsean tanto el color al que le tocaría mover en siguiente lugar, para lo que se emplea un único carácter que puede ser o una "w" para blancas o una "b" para negras; seguido de los derechos restantes de enroque, que son entre uno y cuatro caracteres. Cuando se parsea este campo, se guarda sobre una única variable representada en un entero a la que aplican operaciones OR con todos los derechos de enroque encontrados. De esta manera, podemos emplear esta propiedad en un futuro con menos espacio en memoria y de una manera más congruente con nuestro programa.

```
void FChessBoard::ParseFEN(FString fenString)
{
    // ...

    fen++;

    // parse side to move
    Side = (*fen == 'w') ? Side::White : Side::Black;

    fen += 2;

    // parse available castlings
    while (*fen != ' ')
    {
        switch (*fen)
        {
            case 'K': AvailableCastlings |= Castle::WhiteKingSide;  break;
            case 'Q': AvailableCastlings |= Castle::WhiteQueenSide; break;
            case 'k': AvailableCastlings |= Castle::BlackKingSide;  break;
            case 'q': AvailableCastlings |= Castle::BlackQueenSide; break;
            case '-': break;
        }
        fen++;
    }

    // ...
}
```

Ilustración 5-27. Implementación del parseo del color actual a jugar y los derechos restantes de enroque

Justo después, se parsea la casilla abierta para una captura al paso. Esta casilla sigue la notación algebraica del ajedrez, empleando así dos caracteres en total; una letra para denotar la columna y un dígito para la fila.

```
void FChessBoard::ParseFEN(FString fenString)
{
    // ...

    // parse en-passant square
    if (*fen != '-')
    {
        int file = fen[0] - 'a';
        int rank = 8 - (fen[1] - '0');

        // initialize en-passant square
        OpenEnpassant = (Square)(rank * 8 + file);
    }
    else
    {
        OpenEnpassant = Square::NoSquare;
    }

    // ...
}
```

Ilustración 5-28. Algoritmo para parsear la casilla abierta para captura al paso en una cadena FEN

Finalmente, se parsean tanto los contadores de medios movimientos y de movimientos completos, que emplean entre uno y dos dígitos para llevar esta cuenta, y se rellenan los Bitboards de ocupaciones en base a la ocupación de piezas rellenada en el parseo del posicionamiento de piezas.

```
void FChessBoard::ParseFEN(FString fenString)
{
    // ...

    // initialize white occupancies
    for (int piece = PieceCode::WhitePawn; piece <= PieceCode::WhiteKing; piece++)
    {
        // populate white occupancy bitboard
        Occupancies[Side::White] |= Bitboards[piece];
    }

    // initialize black occupancies
    for (int piece = PieceCode::BlackPawn; piece <= PieceCode::BlackKing; piece++)
    {
        Occupancies[Side::Black] |= Bitboards[piece];
    }

    // initialize all occupancy
    Occupancies[Side::Both] = Occupancies[Side::White] | Occupancies[Side::Black];
}
```

Ilustración 5-29. Implementación del parseo de los contadores de movimientos y composición de la lista de ocupaciones.

5.7. Generación y codificación de movimientos

Ahora que podemos conocer los ataques para cualquier pieza de cualquier color, debemos implementar la forma de generar estos movimientos en base a estos ataques generados. Lo primero, y a modo de utilidad, implementaremos una función que nos devolverá la tabla que ataques para una pieza desde una casilla.

```
uint64 FChessAttacks::AttacksForPiece(FChessBoard* board, PieceCode piece, int sourceSquare)
{
    const Side side = board->Side;
    uint64 attacks;

    switch (piece)
    {
    case WhiteKnight: case BlackKnight:
        attacks = GetKnightAttacks(sourceSquare) & ((side == White) ? ~(board->Occupancies[White]) : ~
(board->Occupancies[Black]));
        break;
    case WhiteBishop: case BlackBishop:
        attacks = GetBishopAttacks(sourceSquare, board->Occupancies[Both]) & ((side == White) ? ~
(board->Occupancies[White]) : ~(board->Occupancies[Black]));
        break;
    case WhiteRook: case BlackRook:
        attacks = GetRookAttacks(sourceSquare, board->Occupancies[Both]) & ((side == White) ? ~(board-
>Occupancies[White]) : ~(board->Occupancies[Black]));
        break;
    case WhiteQueen: case BlackQueen:
        attacks = GetQueenAttacks(sourceSquare, board->Occupancies[Both]) & ((side == White) ? ~(board-
>Occupancies[White]) : ~(board->Occupancies[Black]));
        break;
    case WhiteKing: case BlackKing:
        attacks = GetKingAttacks(sourceSquare) & ((side == White) ? ~(board->Occupancies[White]) : ~
(board->Occupancies[Black]));
        break;
    default:
        attacks = 0ULL;
    }

    return attacks;
}
```

Ilustración 5-30. Implementación de la función para conseguir los ataques que una pieza puede realizar

Esta función no sólo nos consigue el Bitboard precalculado con los ataques, sino que además nos devuelve únicamente los bits activos que no colisionan con las piezas de su mismo color, filtrando movimientos ilegales por colisión con piezas aliadas. Este filtrado se consigue negando la tabla de ocupación del mismo color que la pieza a mover y realizando una operación lógica AND.

Los movimientos, como ya se ha descrito en el apartado de Codificación de movimientos de Estado del arte, van a ser codificados en números entero de 32 bits (aunque únicamente se van a emplear 24 de los 32 bits) y encapsulados en una estructura de datos más fácilmente manipulable llamada Move. Aquí se puede ver el constructor de esta estructura de datos.

```
FMove::FMove(
    int sourceSquare,
    int targetSquare,
    int piece,
    int promotion,
    bool captureFlag,
    bool doublePawnPushFlag,
    bool enPassantCaptureFlag,
    bool castlingMoveFlag
)
{
    SourceSquare = sourceSquare;
    TargetSquare = targetSquare;
    Piece = piece;
    Promotion = promotion;
    CaptureFlag = captureFlag;
    DoublePawnPushFlag = doublePawnPushFlag;
    EnPassantCaptureFlag = enPassantCaptureFlag;
    CastlingMoveFlag = castlingMoveFlag;

    Encoded = sourceSquare
        | targetSquare << 6
        | piece << 12
        | promotion << 16
        | (int)captureFlag << 20
        | (int)doublePawnPushFlag << 21
        | (int)captureFlag << 22
        | (int)castlingMoveFlag << 23;
}
```

Ilustración 5-31. Constructor de la estructura de datos que almacena toda la información de un movimiento individual

5.8. Generación de movimientos de peón

El peón es la pieza con más movimiento, ya que posee un movimiento simple y una captura y hasta tres movimientos especiales; uno doble, la promoción (que puede ser con captura) y la captura al paso.

5.8.1. Movimientos sin captura del peón

A continuación, aparece el código de la implementación de la generación de movimientos sin captura del peón.

```
TArray<FMove> FChessAttacks::GeneratePawnMoves(FChessBoard* board, PieceCode pawn, Side side, uint64
bitboard)
{
    TArray<FMove> moves;
    moves.Init(FMove(), 0);

    PieceCode queen = (side == White) ? WhiteQueen : BlackQueen;
    PieceCode rook = (side == White) ? WhiteRook : BlackQueen;
    PieceCode bishop = (side == White) ? WhiteBishop : BlackQueen;
    PieceCode knight = (side == White) ? WhiteKnight : BlackQueen;
    Side opponent = (side == White) ? Black : White;

    while (bitboard)
    {
        int sourceSquare = LSBIndex(bitboard);
        int targetSquare = (side || White) ? sourceSquare - 8 : sourceSquare + 8;

        // pawn quiet moves
        if ((side == White) ?
            !(targetSquare < A8) && !GetBit(board->Occupancies[Both], targetSquare) :
            !(targetSquare < H1) && !GetBit(board->Occupancies[Both], targetSquare))
        {
            if ((side == White) ?
                !(targetSquare < A7) && sourceSquare <= H7 :
                !(targetSquare < A2) && sourceSquare <= H2)
            {
                // add promotion moves
                moves.Add(FMove(sourceSquare, targetSquare, pawn, queen, false, false, false, false));
                moves.Add(FMove(sourceSquare, targetSquare, pawn, rook, false, false, false, false));
                moves.Add(FMove(sourceSquare, targetSquare, pawn, bishop, false, false, false, false));
                moves.Add(FMove(sourceSquare, targetSquare, pawn, knight, false, false, false, false));
            }
            else
            {
                // add single pawn move
                moves.Add(FMove(sourceSquare, targetSquare, pawn, 0, false, false, false, false));

                // add double pawn pushes
                if ((side == White) ?
                    (sourceSquare >= A2 && sourceSquare <= H2) && !GetBit(board->Occupancies[Both],
targetSquare -= 8) :
                    (targetSquare >= A7 && targetSquare <= H7) && !GetBit(board->Occupancies[Both],
targetSquare += 8))
                {
                    moves.Add(FMove(sourceSquare, targetSquare, pawn, 0, false, true, false, false));
                }
            }
        }

      // pawn captures
      // ...
    }

  return moves;
}
```

Ilustración 5-32. Implementación del algoritmo para generar los movimientos sin capturas del peón.

En este algoritmo, gracias al constructor de movimientos, sólo debemos preocuparnos de implementar los filtros necesarios para cada tipo de movimiento (aparecen comentados en el código), crear el movimiento y añadirlo a la lista que devolver con todos los movimientos.

En el siguiente fragmento de código se puede ver la generación de movimientos que involucran capturas, incluyendo

```
TArray<FMove> FChessAttacks::GeneratePawnMoves(FChessBoard* board, PieceCode pawn, Side side, uint64
bitboard)
{
    TArray<FMove> moves;
    moves.Init(FMove(), 0);

    PieceCode queen = (side == White) ? WhiteQueen : BlackQueen;
    PieceCode rook = (side == White) ? WhiteRook : BlackQueen;
    PieceCode bishop = (side == White) ? WhiteBishop : BlackQueen;
    PieceCode knight = (side == White) ? WhiteKnight : BlackQueen;
    Side opponent = (side == White) ? Black : White;

    while (bitboard)
    {
        int sourceSquare = LSBIndex(bitboard);
        int targetSquare = (side || White) ? sourceSquare - 8 : sourceSquare + 8;

        // pawn quiet moves
        // ...

        // pawn captures
        uint64 attacks = GetPawnAttacks(side, sourceSquare) & board->Occupancies[opponent];

        while (attacks)
        {
            targetSquare = LSBIndex(attacks);

            if ((side == White) ?
                !(targetSquare < A7) && sourceSquare <= H7 :
                !(targetSquare < A2) && sourceSquare <= H2)
            {
                // add capturing promotion moves
                moves.Add(FMove(sourceSquare, targetSquare, pawn, queen, true, false, false, false));
                moves.Add(FMove(sourceSquare, targetSquare, pawn, rook, true, false, false, false));
                moves.Add(FMove(sourceSquare, targetSquare, pawn, bishop, true, false, false, false));
                moves.Add(FMove(sourceSquare, targetSquare, pawn, knight, true, false, false, false));
            }
            else
            {
                // add single pawn move
                moves.Add(FMove(sourceSquare, targetSquare, pawn, 0, false, false, false, false));
            }

            PopBit(attacks, targetSquare);
        }

        Square openEnpassant = board->OpenEnpassant;
        if (openEnpassant != NoSquare)
        {
            uint64 openEnpassantAttack = GetPawnAttacks(side, sourceSquare) & (1ULL << openEnpassant);

            if (openEnpassantAttack)
            {
                targetSquare = LSBIndex(openEnpassantAttack);
                moves.Add(FMove(sourceSquare, targetSquare, pawn, 0, true, false, true, false));
            }
        }

        PopBit(bitboard, sourceSquare);
    }

  return moves;
}
```

De nuevo, haciendo uso de filtros que describen las condiciones para que un movimiento sea ejecutado, añadimos dicho movimiento a la lista, en este caso con la Flag de captura con valor verdadero.

5.9. Generación de movimientos de enroque

Estos movimientos son los únicos movimientos que mueven dos piezas del tablero al mismo tiempo, siendo estas el rey y la torre correspondiente al enroque que realizar. En esta función se deben comprobar que las condiciones para poder realizar el enroque se satisfacen (mencionadas

en el apartado Enroque de Estado del arte) para que ese movimiento de enroque pueda ser generado.

```
Array<FMove> FChessAttacks::GenerateCastlingMoves(FChessBoard* board, PieceCode king, uint64 bitboard)
{
    TArray<FMove> moves;
    moves.Init(FMove(), 0);

    Side side = board->Side;
    int availableCaslings = board->AvailableCastlings;
    int kingSideCastle = availableCaslings & ((side == White) ? WhiteKingSide : BlackKingSide);
    int queenSideCastle = availableCaslings & ((side == White) ? WhiteQueenSide : BlackQueenSide);
    int kingSideConnected = (side == White)
      ? !GetBit(board->Occupancies[Both], F1)
        && !GetBit(board->Occupancies[Both], G1)
      : !GetBit(board->Occupancies[Both], F8)
        && !GetBit(board->Occupancies[Both], G8);
    int queenSideConnected = (side == White)
      ? !GetBit(board->Occupancies[Both], D1)
        && !GetBit(board->Occupancies[Both], C1)
        && !GetBit(board->Occupancies[Both], B1)
      : !GetBit(board->Occupancies[Both], D8)
        && !GetBit(board->Occupancies[Both], C8)
        && !GetBit(board->Occupancies[Both], B8);
    int noKingSideChecks = (side == White)
      ? !SquareIsAttacked(board, E1, Black)
        && !SquareIsAttacked(board, F1, Black)
      : !SquareIsAttacked(board, E8, White)
        && !SquareIsAttacked(board, F8, White);
    int noQueenSideChecks = (side == White)
      ? !SquareIsAttacked(board, E1, Black)
        && !SquareIsAttacked(board, D1, Black)
      : !SquareIsAttacked(board, E8, White)
        && !SquareIsAttacked(board, D8, White);
    int sourceSquare = (side == White) ? E1 : E8;
    int kingSideTargetSquare = (side == White) ? G1 : G8;
    int queenSideTargetSquare = (side == White) ? C1 : C8;

    if (kingSideCastle && kingSideConnected && noKingSideChecks)
    {
        moves.Add(FMove(sourceSquare, kingSideTargetSquare, king, 0, false, false, false, true));
    }
    if (queenSideCastle && queenSideConnected && noQueenSideChecks)
    {
        moves.Add(FMove(sourceSquare, queenSideTargetSquare, king, 0, false, false, false, true));
    }

    return moves;
}
```

Aunque sea un poco difícil leer el código a primera vista, la realidad es que es código necesario para generar los enroques, ya que más de las primeras tres cuartas partes de lo que ocupa esta función son las comprobaciones necesarias para que los enroques se puedan generar, como que los derechos a enrocar a ese lado sigan activos, o que las casillas pertinentes no estén atacadas.

5.10. Generación de movimientos del resto de piezas

Para la generación de movimientos para el resto de piezas, una única función es necesaria haciendo uso de la función que nos devuelve los ataques para cada pieza mencionada anteriormente.

```
TArray<FMove> FChessAttacks::GeneratePieceMoves(FChessBoard* board, PieceCode piece, Side side, uint64
bitboard)
{
    TArray<FMove> moves;
    moves.Init(FMove(), 0);
    int sourceSquare = 0, targetSquare = 0;

    while (bitboard)
    {
        sourceSquare = LSBIndex(bitboard);
        uint64 attacks = AttacksForPiece(board, piece, sourceSquare);

        while (attacks)
        {
            targetSquare = LSBIndex(attacks);

            if (!GetBit(((side == White) ? board->Occupancies[Black] : board->Occupancies[White]),
targetSquare))
            {
                moves.Add(FMove(sourceSquare, targetSquare, piece, 0, false, false, false, false));
            }
            else
            {
                moves.Add(FMove(sourceSquare, targetSquare, piece, 0, true, false, false, false));
            }

            PopBit(attacks, targetSquare);
        }

        PopBit(bitboard, targetSquare);
    }

    return moves;
}
```

Como se puede apreciar, recorremos el Bitboard de la pieza de la que generar los movimientos con el bucle *while*, desactivando el último bit visitado (*sourceSquare*) y en cada iteración hacemos el mismo bucle, pero sobre el Bitboard de ataques para la pieza correspondiente, generando así el movimiento con o sin captura, dependiendo de si la casilla *targetSquare* está ocupada o no por una pieza del oponente.

5.11. Solución Copy/Make

Una vez podemos generar todos los movimientos disponibles para una posición, ejecutarlos es solo una cuestión de modificar los Bitboards pertinentes. Entonces, a la hora de realizar la búsqueda, nuestro algoritmo de búsqueda tendrá que:

1) Generar el listado de todos los movimientos disponibles para la posición actual
2) Iterar sobre los movimientos generados, guardando el estado del tablero con la que poder restaurar el estado del tablero cuando la búsqueda termine.
3) Ejecutar el movimiento en el tablero, comprobando que el movimiento es legal.
4) Volver al paso 1 mientras no se llegue a la profundidad deseada y repetir recursivamente para la nueva posición.
5) Cuando se llegue a la profundidad deseada, volver al paso 2 del nodo original, restaurar el tablero que habíamos guardado y continuar la iteración.

Para ello, se han implementado dos funciones Macro, una para guardar el estado del tablero y otra para recuperarlo. [18]

```
#define StoreBoard(board) \
    uint64 bitboardsCopy[12], occupanciesCopy[3]; \
    int sideCopy, openEnpassantCopy, availableCastlingsCopy; \
    memcpy(bitboardsCopy, (board)->Bitboards, sizeof((board)->Bitboards)); \
    memcpy(occupanciesCopy, (board)->Occupancies, sizeof((board)->Occupancies)); \
    sideCopy = (board)->Side; \
    openEnpassantCopy = (board)->OpenEnpassant; \
    availableCastlingsCopy = (board)->AvailableCastlings;

#define RestoreBoard(board) \
    memcpy(board->Bitboards, bitboardsCopy, sizeof(board->Bitboards)); \
    memcpy(board->Occupancies, occupanciesCopy, sizeof(board->Occupancies)); \
    board->Side = (Side)sideCopy; \
    board->OpenEnpassant = (Square)openEnpassantCopy; \
    board->AvailableCastlings = availableCastlingsCopy;
```

Ilustración 5-33. Funciones Macro para la preservación y restauración de un estado del tablero.

5.12. Ejecución de movimientos sobre el tablero

En la función que ejecuta los movimientos sobre el tablero es donde se deben filtrar los movimientos legales de los pseudo-legales, permitiendo o no ejecutarlos; ya que nuestra función de generación de movimientos, realmente, genera todos los movimientos que las piezas son capaces de hacer con la única restricción de poder realizar ese desplazamiento sin colisionar con piezas aliadas; los llamados movimientos pseudo-legales.

Esta función, entonces, deberá devolver si ha sido capaz de realizar el movimiento, alterando así el tablero o no.

Esta función es bastante larga, de nuevo por requerimientos de normativa del ajedrez. Es por esto por lo que, dado que se emplea un nombrado de variables muy explícito y que mis justificaciones para lo que ocurre en cada fragmento de código, o bien ya figura comentado en el mismo, o es bastante explícito, a continuación, aparece el fragmento de código encargado de ejecutar movimientos en el tablero.

18 Se ha optado por emplear funciones Macro en lugar de funciones inline debido la alta reutilización de estas y a que las funciones Macro son sustituidas antes de la compilación, mientras que el cuerpo de las funciones inline es inyectado en el lugar de la llamada y suelen ser utilizadas cuando una llamada a función es necesaria. Como no es necesario llamar a una función para realizar esta copia, ya que se podría evitar escribiendo el código explícitamente, en lugar de duplicar el código, definimos estas funciones Macro.

```
bool FChessAttacks::MakeMove(FChessBoard* board, FMove move, MoveType moveType)
{
    if (moveType == AllMoves)
    {
        StoreBoard(board);

        PopBit(board->Bitboards[move.Piece], move.SourceSquare);
        SetBit(board->Bitboards[move.Piece], move.TargetSquare);

        // handle captures
        if (move.CaptureFlag)
        {
            int startPiece = (board->Side == White) ? BlackPawn : WhitePawn;
            int endPiece = (board->Side == White) ? BlackKing: WhiteKing;

            for (int opponentsPiece = startPiece; opponentsPiece <= endPiece; opponentsPiece++)
            {
                if (GetBit(board->Bitboards[opponentsPiece], move.TargetSquare))
                {
                    PopBit(board->Bitboards[opponentsPiece], move.TargetSquare);
                    break;
                }
            }
        }

        // handle promotions
        if (move.Promotion)
        {
            PopBit(board->Bitboards[(board->Side == White) ? WhitePawn : BlackPawn],
move.TargetSquare);
            SetBit(board->Bitboards[move.Promotion], move.TargetSquare);
        }

        // handle enpassant
        if (move.EnPassantCaptureFlag)
        {
            (board->Side == White)
                ? PopBit(board->Bitboards[BlackPawn], move.TargetSquare + 8)
                : PopBit(board->Bitboards[WhitePawn], move.TargetSquare - 8);
        }
        board->OpenEnpassant = NoSquare;

        // handle double pawn pushes
        if (move.DoublePawnPushFlag)
        {
            board->OpenEnpassant = (board->Side = White)
                ? (Square)(move.TargetSquare + 8)
                : (Square)(move.TargetSquare - 8);
        }

        if (move.CastlingMoveFlag)
        {
            switch (move.TargetSquare)
            {
            case G1:
                PopBit(board->Bitboards[WhiteRook], H1);
                SetBit(board->Bitboards[WhiteRook], F1);
                break;
            case C1:
                PopBit(board->Bitboards[WhiteRook], A1);
                SetBit(board->Bitboards[WhiteRook], D1);
                break;
            case G8:
                PopBit(board->Bitboards[BlackRook], H8);
                SetBit(board->Bitboards[BlackRook], F8);
                break;
            case C8:
                PopBit(board->Bitboards[BlackRook], A8);
                SetBit(board->Bitboards[BlackRook], D8);
                break;
            }
        }

        // handle castling rights
        board->AvailableCastlings &= CastlingRights[move.SourceSquare];
        board->AvailableCastlings &= CastlingRights[move.TargetSquare];

        // reset occupancies
        memset(board->Occupancies, 0ULL, sizeof(board->Occupancies));
        for (int piece = WhitePawn; piece <= WhiteKing; piece++)
        {
            board->Occupancies[White] |= board->Bitboards[piece];
        }
        for (int piece = BlackPawn; piece <= BlackKing; piece++)
        {
            board->Occupancies[Black] |= board->Bitboards[piece];
        }
        board->Occupancies[Both] |= board->Occupancies[White];
        board->Occupancies[Both] |= board->Occupancies[Black];

        // switch sides
        board->Side = (Side)((int)board->Side ^ 1);

        // check king exposure
        if (SquareIsAttacked(board, (board->Side == White) ? LSBIndex(board->Bitboards[BlackKing]) :
LSBIndex(board->Bitboards[WhiteKing]), board->Side))
        {
            RestoreBoard(board);

            return false;
        }
        else
        {
            return true;
        }
    }
    else
    {
        if (move.CaptureFlag)
        {
            MakeMove(board, move, AllMoves);
        }
        else
        {
            return false;
        }
    }

    return false;
}
```

Ilustración 5-34. Implementación de la función para ejecutar movimientos en el tablero

5.13. Evaluación

Para la evaluación de posiciones se pueden tener en cuenta varios factores. En este motor se ha implementado una evaluación en función del valor del material (la pieza) y del posicionamiento de las piezas en el tablero. Esto último se basa en el concepto de que, en el ajedrez, las piezas son mucho más activas en determinadas casillas, mientras que en otras a penas aportan valor a la posición. Por ejemplo, es preferible colocar los alfiles en las primeras fases de la partida en las casillas B2 y G2 para las blancas y B7 y G7 para las negras porque controlan las llamadas "grandes diagonales".

Las valoraciones de cada pieza se pueden ver en el apartado de Piezas en el Estado del arte, y su traducción a código es tan simple como declarar una lista con los valores mapeados a la enumeración de nuestras piezas mostrada en la Ilustración 5-4. Enumeración de las piezas ordenadas.

```
static int MaterialScore(int piece)
{
    int MATERIAL_SCORES[12] = {
        100,      // white pawn
        300,      // white knight
        350,      // white bishop
        500,      // white rook
        1000,     // white queen
        10000,    // white king
        -100,     // black pawn
        -300,     // black knight
        -350,     // black bishop
        -500,     // black rook
        -1000,    // black queen
        -10000,   // black king
    };

    return MATERIAL_SCORES[piece];
}
```

Ilustración 5-35. Lista con los valores de las piezas, ordenadas para coincidir con el mapeo de las piezas

Como dato a destacar, todas estas listas están encapsuladas en funciones estáticas para poder emplearlas en la función estática para la evaluación de movimientos.

Una vez definidos los valores a tener en cuenta para evaluar una posición en función del material, definimos los valores posicionales para cada pieza. Estos valores son arbitrarios en cuanto a su valor, pero la desviación entre ellos no lo es del todo. Estas tablas representan una preferencia, y podríamos modificar estos valores para hacer que nuestro motor valore las posiciones de diferente manera, pero eso ya es una decisión que tomar una vez sepamos qué comportamiento queremos que tenga nuestro motor. [19]

19 Se puede apreciar en los *snippets* de código que esos vectores están encapsulados en funciones estáticas. Esto es debido a que en C++ de Unreal Engine no se pueden inicializar propiedades estáticas en la declaración. Por tanto, una manera de evitar esto teniendo la misma funcionalidad es encapsular estos vectores en funciones estáticas.

```
static int PawnPositionalScore(int square)
{
    const int PAWN_POSITIONAL_SCORES[64] = {
        90, 90, 90, 90, 90, 90, 90, 90,
        30, 30, 30, 40, 40, 30, 30, 30,
        20, 20, 20, 30, 30, 30, 20, 20,
        10, 10, 10, 20, 20, 10, 10, 10,
         5,  5, 10, 20, 20,  5,  5,  5,
         0,  0,  0,  5,  5,  0,  0,  0,
         0,  0,  0,-10,-10,  0,  0,  0,
         0,  0,  0,  0,  0,  0,  0,  0
    };

    return PAWN_POSITIONAL_SCORES[square];
}

static int KnightPositionalScore(int square)
{
    const int KNIGHT_POSITIONAL_SCORE[64] = {
        -5,  0,  0,  0,  0,  0,  0, -5,
        -5,  0,  0, 10, 10,  0,  0, -5,
        -5,  5, 20, 20, 20, 20,  5, -5,
        -5, 10, 20, 30, 30, 20, 10, -5,
        -5, 10, 20, 30, 30, 20, 10, -5,
        -5,  5, 20, 10, 10, 20,  5, -5,
        -5,  0,  0,  0,  0,  0,  0, -5,
        -5,-10,  0,  0,  0,  0,-10, -5
    };

    return KNIGHT_POSITIONAL_SCORE[square];
}

static int BishopPositionalScore(int square)
{
    const int BISHOP_POSITIONAL_SCORE[64] = {
         0,  0,  0,  0,  0,  0,  0,  0,
         0,  0,  0,  0,  0,  0,  0,  0,
         0,  0,  0, 10, 10,  0,  0,  0,
         0,  0, 10, 20, 20, 10,  0,  0,
         0,  0, 10, 20, 20, 10,  0,  0,
         0, 10,  0,  0,  0,  0, 10,  0,
         0, 30,  0,  0,  0,  0, 30,  0,
         0,  0,-10,  0,  0,-10,  0,  0,
    };

    return BISHOP_POSITIONAL_SCORE[square];
}

static int RookPositionalScore(int square)
{
    const int ROOK_POSITIONAL_SCORE[64] = {
        50, 50, 50, 50, 50, 50, 50, 50,
        50, 50, 50, 50, 50, 50, 50, 50,
         0,  0, 10, 20, 20, 10,  0,  0,
         0,  0, 10, 20, 20, 10,  0,  0,
         0,  0, 10, 20, 20, 10,  0,  0,
         0,  0, 10, 20, 20, 10,  0,  0,
         0,  0, 10, 20, 20, 10,  0,  0,
         0,  0,  0, 20, 20,  0,  0,  0,
    };

    return ROOK_POSITIONAL_SCORE[square];
}

static int KingPositionalScore(int square)
{
    const int KING_POSITIONAL_SCORE[64] = {
         0,  0,  0,  0,  0,  0,  0,  0,
         0,  0,  5,  5,  5,  5,  0,  0,
         0,  5,  5, 10, 10,  5,  5,  0,
         0,  5, 10, 20, 20, 10,  5,  0,
         0,  5, 10, 20, 20, 10,  5,  0,
         0,  0,  5, 10, 10,  5,  0,  0,
         0,  5,  5, -5, -5,  0,  5,  0,
         0,  0,  5,  0,-15,  0, 10,  0,
    };

    return KING_POSITIONAL_SCORE[square];
}
```

Ilustración 5-36. Tablas que recogen el valor posicional de cada pieza sobre el tablero

Por último, estos valores son fijos para cada casilla, pero tenemos que implementar alguna manera de hacer que estos valores se inviertan verticalmente para las piezas negras. Para ello, creamos otra tabla que haga este mapeo por nosotros.

```
static int Mirror(int square)
{
    const int MIRROR_SQUARE[64] = {
        A1, B1, C1, D1, E1, F1, G1, H1,
        A2, B2, C2, D2, E2, F2, G2, H2,
        A3, B3, C3, D3, E3, F3, G3, H3,
        A4, B4, C4, D4, E4, F4, G4, H4,
        A5, B5, C5, D5, E5, F5, G5, H5,
        A6, B6, C6, D6, E6, F6, G6, H6,
        A7, B7, C7, D7, E7, F7, G7, H7,
        A8, B8, C8, D8, E8, F8, G8, H8
    };

    return MIRROR_SQUARE[square];
}
```

Ilustración 5-37. Mapeo de una casilla a la equivalente desde la posición del color opuesto

Ahora, con todos estos datos por los que poder evaluar una posición, implementamos la función encargada de esta tarea. Esta función va a ser una función estática encapsulada en la clase evaluadora. De esta manera, su uso se reduce a una llamada en la que pasamos una referencia al tablero como parámetro, y que nos devolverá el valor de la evaluación de este.

```
int FChessEvaluator::Evaluate(FChessBoard* board)
{
    int score = 0;

    for (int piece = WhitePawn; piece <= BlackKing; piece++)
    {
        uint64 bitboard = board->Bitboards[piece];

        while (bitboard)
        {
            int square = LSBIndex(bitboard);
            score += FChessEvaluator::MaterialScore(piece);

            switch (piece)
            {
                case WhitePawn:
                    score += FChessEvaluator::PawnPositionalScore(square);
                    break;
                case WhiteKnight:
                    score += FChessEvaluator::KnightPositionalScore(square);
                    break;
                case WhiteBishop:
                    score += FChessEvaluator::BishopPositionalScore(square);
                      break;
                case WhiteRook:
                    score += FChessEvaluator::RookPositionalScore(square);
                    break;
                case WhiteKing:
                    score += FChessEvaluator::KingPositionalScore(square);
                    break;

                case BlackPawn:
                    score -= FChessEvaluator::PawnPositionalScore(FChessEvaluator::Mirror(square));
                    break;
                case BlackKnight:
                    score -= FChessEvaluator::KnightPositionalScore(FChessEvaluator::Mirror(square));
                    break;
                case BlackBishop:
                    score -= FChessEvaluator::BishopPositionalScore(FChessEvaluator::Mirror(square));
                    break;
                case BlackRook:
                    score -= FChessEvaluator::RookPositionalScore(FChessEvaluator::Mirror(square));
                    break;
                case BlackKing:
                    score -= FChessEvaluator::KingPositionalScore(FChessEvaluator::Mirror(square));
                    break;
            }

            PopBit(bitboard, square);
        }

        return (board->Side == White) ? score : -score;
    }
}
```

Ilustración 5-38. Implementación de la función de evaluación de una posición del tablero.

5.14. Búsqueda

Para la búsqueda de movimientos, debíamos ser capaces de generar un listado con todos los movimientos disponibles para una posición y, además, ser capaces de evaluar cada movimiento para obtener un cómputo general de una línea [20] y poder establecer uno de los movimientos raíz (el que nos garantice una mejor puntuación) como el idóneo y, por tanto, el que debe ser ejecutado.

Para este motor se ha implementado un algoritmo Negamax con la mejora de *pruning* mediante el algoritmo Alpha-Beta (ver Algoritmo Negamax y Alpha-Beta pruning en el Estado del arte) se ha implementado una búsqueda quiescente.

La función encargada de realizar la búsqueda es, de nuevo, una función estática encapsulada en una clase encargada de hacer esta búsqueda del mejor movimiento. Dicha función inicializa las variables necesarias, ejecuta la búsqueda (en nuestro caso, mediante el algoritmo de Negamax) y devuelve el mejor movimiento.

```
FMove FChessSearchEngine::Search(FChessBoard* board, FChessAttacks* chessAttacks, int depth)
{
    FChessSearchEngine::Nodes = 0;
    FChessSearchEngine::BestMove = FMove();
    FChessSearchEngine::Ply = 0;
    FChessSearchEngine::LegalMoves = 0;

    int score = Negamax(board, chessAttacks, -50000, -50000, depth);

    return FChessSearchEngine::BestMove;
}
```

Ilustración 5-39. Implementación de la función de búsqueda del mejor movimiento

5.14.1. Negamax & Alpha-Beta Pruning

El funcionamiento de este algoritmo en conjunto con su mejora mediante la técnica de poda Alpha-Beta figuran explicados con detenimiento en el apartado Algoritmo Negamax y Alpha-Beta pruning del apartado de Estado del arte.

Como esta función es significativamente larga, se va a describir paso a paso la implementación del algoritmo.

20 En el ajedrez, una línea, cuando hablamos de búsqueda de movimientos (humana o computacional) hacemos referencia a una secuencia única de movimientos derivada de una posición.

```
int FChessSearchEngine::Negamax(
    FChessBoard* board,
    FChessAttacks* chessAttacks,
    int alpha, int beta, int depth
) {
    // 1. recursion escape condition
    if (depth == 0)
    {
        // run quiescense search to avoid Horizon Effect
        return FChessSearchEngine::Quiescense(board, chessAttacks, alpha, beta);
    }

    // ...
}
```

Ilustración 5-40. Condición de escape de la búsqueda Negamax. Ejecución de la búsqueda quiescente.

Lo primero que se debe hacer en toda función recursiva es establecer una condición de escape. En este caso, la condición de escape que hace finalizar de manera imperativa nuestra búsqueda es que se llegue a la profundidad establecida (en este caso es la profundidad 0 porque la búsqueda comienza en la profundidad máxima, decrementando este valor en cada llamada recursiva). Una vez se alcanza esta profundidad y por los motivos descritos en el apartado de Búsqueda Quiescente (Quiescense Search) en el Estado del arte, se realiza esta búsqueda cuando se alcanza la profundidad deseada (ver implementación en el apartado siguiente, Búsqueda quiescente)

```
int FChessSearchEngine::Negamax(
    FChessBoard* board,
    FChessAttacks* chessAttacks,
    int alpha, int beta, int depth
) {
    // ...

    // 2. Initialize necessary information
    FChessSearchEngine::Nodes++;

    // check for king exposure
    bool inCheck = chessAttacks->SquareIsAttacked(board,
        (board->Side == White)
            ? LSBIndex(board->Bitboards[WhiteKing])
            : LSBIndex(board->Bitboards[BlackKing]),
        (Side)(board->Side ^ 1)
    );

    int legalMoves = 0;
    FMove bestSoFar;
    int oldAlpha = alpha;

    // ...
}
```

Ilustración 5-41. Inicialización de los recursos necesarios por cada iteración de la búsqueda.

En este fragmento se inicializan las variables necesarias para el nodo actual de la búsqueda, entre los que destaca la variable *inCheck*, que nos permite saber si el rey del jugador al que le toca jugar está atacado, limitando los movimientos que se pueden realizar.

```
int FChessSearchEngine::Negamax(
    FChessBoard* board,
    FChessAttacks* chessAttacks,
    int alpha, int beta, int depth
) {
    // ...

    // 3. Generate moves and iteratively search over them
    TArray<FMove> moves = chessAttacks->GetMovesForSide(board, board->Side);
    for (int i = 0; i < moves.Num(); i++)
    {
        StoreBoard(board);
        FChessSearchEngine::Ply++;

        // check only for legal moves
        if (!chessAttacks->MakeMove(board, moves[i], AllMoves))
        {
            FChessSearchEngine::Ply--;
            continue;
        }
        FChessSearchEngine::LegalMoves++;

        // score for current move
        int score = -FChessSearchEngine::Negamax(
            board,
            chessAttacks,
            -beta, -alpha, depth - 1
        );

        FChessSearchEngine::Ply--;
        RestoreBoard(board);

        // fail-hard beta cut-off
        if (score >= beta)
        {
            return beta;
        }

        // found a better move
        if (score > alpha)
        {
            alpha = score;
            if (FChessSearchEngine::Ply == 0)
            {
                bestSoFar = moves[i];
            }
        }
    }

    // ...
}
```

Ilustración 5-42. Generación de movimientos para el estado actual del tablero e iteración sobre los mismos para obtener el mejor movimiento

Esta parte del algoritmo es la encargada de iterar sobre los movimientos disponibles para una posición y comenzar nuevas búsquedas con profundidad decrementada en una unidad respecto a la búsqueda del nodo en el que se encuentra esta llamada.

Primero de todo, generamos los movimientos con el fin de iterar sobre estos. En cada iteración guardamos el estado del tablero antes de ejecutar el movimiento e incrementamos el *ply*. Si ejecutásemos el movimiento y no fuese legal, decrementaríamos el *ply* y continuaríamos iterando.

Si el movimiento ejecutado en la anterior comprobación hubiera sido legal, incrementamos el contador de movimientos legales y comenzaríamos otra búsqueda para obtener la puntuación

que nos da este nodo intercambiando los valores *alpha* y *beta* y negándolos (justificación en el apartado Algoritmo Negamax y Alpha-Beta pruning del Estado del arte).

Una vez finalizada la búsqueda, decrementamos el *ply*, restauramos el tablero previamente guardado y comprobamos si la puntuación obtenida es peor que la que teníamos anteriormente (*fail-hard*) o si, por el contrario, hemos encontrado un mejor movimiento.

En ese último caso, igualamos *alpha* a la puntuación obtenida y, si este nodo es uno de los nodos originales (los que se generan en la primera generación del árbol de búsqueda) se establece ese movimiento como el mejor.

```
int FChessSearchEngine::Negamax(
    FChessBoard* board,
    FChessAttacks* chessAttacks,
    int alpha, int beta, int depth
) {
    // ...

    // 4. Check if there are no legal moves to play for this node
    if (legalMoves == 0)
    {
        // king is in check, return lowest
        if (inCheck)
        {
            // ply is necessary to avoid stalemate
            return -49000 + FChessSearchEngine::Ply;
        }
        else
        {
            return 0;
        }
    }

    // ...
}
```

Ilustración 5-43. Devolver puntuación de los nodos sin movimientos legales disponibles, dependiendo de la vulnerabilidad del rey

Si el nodo actual de la búsqueda no presenta movimientos legales, debemos devolver una puntuación de acuerdo con si el rey está expuesto o no. Si el rey está en jaque, devolvemos la peor puntuación en la que también se introduce el *ply* para evitar situaciones de tablas, o una puntuación neutra si el rey no lo está.

```
int FChessSearchEngine::Negamax(
    FChessBoard* board,
    FChessAttacks* chessAttacks,
    int alpha, int beta, int depth
) {
    // ...

    // 5. Decide whether the move found is better or not
    if (oldAlpha != alpha)
    {
        FChessSearchEngine::BestMove = bestSoFar;
    }

    // node fails low
    return alpha;
}
```

Ilustración 5-44. Comparación del movimiento encontrado con el anterior movimiento

Esta es la última parte del algoritmo, donde se decide si el movimiento encontrado es mejor que el mejor movimiento anterior o no, devolviendo el nuevo movimiento o el anterior, respectivamente.

5.14.2. Búsqueda quiescente

Esta búsqueda, como ya se ha explicado con anterioridad, se emplea para evitar falsos mejores movimientos. Esos falsos mejores movimientos se dan cuando llegamos a la profundidad deseada y paramos en seco nuestra búsqueda sin tener en cuenta que el rival podría ejecutar un movimiento que nos haría terminar con una puntuación peor.

Su implementación es bastante directa, comprobando las cosas que tiene que comprobar empleando la función de evaluación y buscando entre los movimientos del rival alguno que haga perder la posición. El código es muy similar al de Negamax pero con algunas condiciones específicas para esta búsqueda concreta. La única comprobación extra que se hace es al comienzo, para ver si la posición es mejor o no.

```
int FChessSearchEngine::Quiescense(FChessBoard* board, FChessAttacks* chessAttacks, int alpha, int
beta)
{
    // recursion escape condition
    int eval = FChessEvaluator::Evaluate(board);
    if (eval >= beta)   // failed-hard, beta cut-off
    {
        return beta;
    }

    // found a better move
    if (eval > alpha)
    {
        alpha = eval;
    }

    // generate moves and search
    TArray<FMove> moves = chessAttacks->GetMovesForSide(board, board->Side);
    for (int i = 0; i < moves.Num(); i++)
    {
        StoreBoard(board);
        FChessSearchEngine::Ply++;

        // make only legal moves
        if (!chessAttacks->MakeMove(board, moves[i], CapturesOnly))
        {
            FChessSearchEngine::Ply--;
            continue;
        }

        // score current move
        int score = -FChessSearchEngine::Quiescense(board, chessAttacks, -beta, -alpha);
        FChessSearchEngine::Ply--;
        RestoreBoard(board);

        // failed-hard, betta cut-off
        if (score >= beta)
        {
            return beta;
        }

        // found a better move
        if (score > alpha)
        {
            alpha = score;
        }
    }

    // return node that fails-low
    return alpha;
}
```

Ilustración 5-45. Implementación de la búsqueda quiescente.

Al comienzo, se evalúa la posición y se compara con la puntuación *beta* para descartar un *fail-hard*. Si ocurriese, se devuelve *beta* para aclarar que la evaluación quiescente del tablero es mala.

De lo contrario, se inicializa *alpha* con el valor de la evaluación realizada y se generan los movimientos disponibles, buscando iterativamente por cada movimiento si hubiese alguno que empeora la posición. Si lo hubiese, la misma condición del comienzo se ejecutaría. Si no fuera el caso, se devolvería la nueva puntuación, presumiblemente mejor.

5.14.3. MVV LVA

Esta técnica (descrita en el apartado MVV–LVA (Most Valuable Victim - Least Valuable Attacker) en el Estado del arte) nos permite ordenar los movimientos, mejorando el algoritmo Alpha-Beta. Esto se consigue ordenando los movimientos de tal manera que los nodos que generen un *fail-hard* aparezcan antes en el árbol de búsqueda y, por tanto, se corten esas líneas de búsqueda antes, reduciendo así el número de nodos visitados significativamente.

Esta ordenación se va a realizar priorizando la búsqueda en movimientos de captura frente al resto bajo el principio de que las capturas son más propensas a generar *beta cut-offs* (podas por condiciones de beta). Para comprender esto, pongamos un ejemplo. Supongamos que, para una posición, la dama está expuesta y podríamos capturarla. No tiene sentido evaluar primero otros movimientos antes que el de la captura de la dama, ya que este movimiento nos da una puntuación significativamente mayo y evita muchas líneas futuras, ya que pocas posiciones generarían una posición mejor que la captura de la dama y, si lo hacen, se emplearían más movimientos intermedios para llegar a la captura de la dama.

Para esto, declararemos una tabla con las puntuaciones que nos daría capturar una pieza con otra, pudiendo así ordenar los movimientos por estas puntuaciones.

```
static int MVV_LVA(int piece, int takes)
{
    int mvv_lva[12][12] = {
        105, 205, 305, 405, 505, 605,  105, 205, 305, 405, 505, 605,
        104, 204, 304, 404, 504, 604,  104, 204, 304, 404, 504, 604,
        103, 203, 303, 403, 503, 603,  103, 203, 303, 403, 503, 603,
        102, 202, 302, 402, 502, 602,  102, 202, 302, 402, 502, 602,
        101, 201, 301, 401, 501, 601,  101, 201, 301, 401, 501, 601,
        100, 200, 300, 400, 500, 600,  100, 200, 300, 400, 500, 600,

        105, 205, 305, 405, 505, 605,  105, 205, 305, 405, 505, 605,
        104, 204, 304, 404, 504, 604,  104, 204, 304, 404, 504, 604,
        103, 203, 303, 403, 503, 603,  103, 203, 303, 403, 503, 603,
        102, 202, 302, 402, 502, 602,  102, 202, 302, 402, 502, 602,
        101, 201, 301, 401, 501, 601,  101, 201, 301, 401, 501, 601,
        100, 200, 300, 400, 500, 600,  100, 200, 300, 400, 500, 600
    };

    return mvv_lva[piece][takes];
}
```

Ilustración 5-46. Tabla para conseguir el valor de una captura de una pieza a otra

Una vez tenemos esto, declaramos una función accesoria que nos permita conseguir estas puntuaciones para un movimiento. La función quedaría así.

```
static FROCEINLINE FMove FSearchEnigne::ScoreMove (FChessBoard* board, FMove move) {
    if (move.CaptureFlag) {
        int targetPiece = P;
        int startPiece, endPiece;
        if (board->Side == White)
        {
            startPiece = BlackPawn;
            endPiece = BlackKing;
        }
        else {
            startPiece = WhitePawn;
            endPiece = WhiteKing;
        }

        for (int piece = startPiece; piece <= endPiece; piece++)
        {
            if (GetBit(board[piece], decode_move_target_square(move)))
            if (GetBit(board->Bitboards[piece], move.TargetSquare))
            {
                targetPiece = piece;
                break;
            }
        }

        return MVV_LVA(move.Piece, target_piece) + 10000;
    }

    // score quiet move
    else {
        // score first killer move
        if (FSearchEnigne::KillerMoves[0][FSearchEnigne::Ply] == move)
            return 9000;

        // score second killer move
        else if (FSearchEnigne::Killermoves[1][FSearchEnigne::Ply] == move)
            return 8000;

        // score history moves
        else
            return FSearchEnigne::HistoryMoves[move.Piece][move.TargetSquare];
    }

    return 0;
}
```

Ilustración 5-47. Implementación de función accesoria para la puntuación de un movimiento mediante la heurística MVV-LVA

En esta función, gracias a la tabla anteriormente definida, podemos conseguir las puntuaciones tanto de los movimientos de captura como de los movimientos sin ella.

Por último, como ya podemos acceder a estas puntuaciones, sólo quedaría ordenar los movimientos generados en base a los valores relevantes. De esta manera, para ordenar los movimientos en función de su puntuación como movimiento, se ha implementado el siguiente algoritmo.

```
static FORCEINLINE TArray<FMove> FSearchEngine::SortMoves(FChessBoard* board, TArray<FMove> moves)
{
    int moveScores[moves.Num];

    for (int count = 0; count < moves.Num; count++)
        moveScores[count] = scoreMove(board, moves[count]);

    for (int currentMove = 0; currentMove < moves.Num; currentMove++)
    {
        for (int nextMove = 0; nextMove < moves.Num; nextMove++)
        {
            if (moveScores[currentMove] < moveScores[nextMove])
            {
                // swap scores
                int tempScore = moveScores[currentMove];
                moveScores[currentMove] = moveScores[nextMove];
                moveScores[nextMove] = tempScore;

                // swap moves
                int tempMove = moves[currentMove];
                moves[currentMove] = moves[nextMove];
                moves[nextMove] = tempMove;
            }
        }
    }

    return moves;
}
```

Ilustración 5-48. Implementación del algoritmo Bubble-sort de ordenación de movimientos por su puntuación

Este algoritmo Bubble-sort de ordenación no es el más eficiente de los algoritmos de este tipo, pero es un algoritmo lo suficientemente sencillo de comprender y fácil de mejorar para tomarlo en consideración como el primer algoritmo a implementar.

Finalmente, y para concluir, bastaría con introducir una llamada a este algoritmo entre la generación de movimientos y la iteración sobre los mismos en la búsqueda Negamax.

6. Estudio económico

6.1. Costes del desarrollo

En Zaragoza, un desarrollador de software junior cobra, de media, alrededor de unos 16.000 euros al año, que se traduciría en un gasto para la empresa de unos 21.400 euros al año aproximadamente. Este salario se traduce en unos 11.50 euros la hora aproximadamente. Emplearemos esta cantidad para calcular los gastos del desarrollador.

Funcionalidades	Horas	Coste (euros)
Horas invertidas establecer la arquitectura	20	230 euros
Horas invertidas en la implementación de las tablas de ataques	24	276 euros
Horas invertidas la generación de los números mágicos	20	230 euros
Horas invertidas en el parseo de cadenas FEN	20	230 euros
Horas invertidas en la generación de movimientos	40	460 euros
Horas invertidas en la ejecución de movimientos sobre el tablero	24	276 euros
Horas invertidas en la implementación de la prueba de rendimiento	24	276 euros
Horas invertidas en la evaluación de movimientos	20	230 euros
Horas invertidas en la búsqueda de movimientos	48	552 euros
TOTAL	240	2760 euros

En las horas para cada funcionalidad se incluyen las horas de desarrollo y las horas de investigación para cada una de ellas.

6.2. Coste de los materiales

En este apartado se muestran los costes de todos los bienes o licencias empleados para el desarrollo de este proyecto. En la siguiente tabla se muestran valores temporales estimados, ya que no podemos conocer con exactitud la vida útil de nuestros dispositivos. Además, la duración en meses del proyecto se ha dividido en 16, que son las horas semanales que se han invertido de media a lo largo del desarrollo.

	Coste (euros)	Vida útil (años)	Duración del proyecto	Coste mensual	Total
Pantalla UltraWide 32"	210	4 años	3.75	4.38 euros	16.43 euros
Macbook Pro 16" (2019)	2400	6 años	3.75	33,34 euros	125.03 euros
Periféricos	150	5 años	3.75	2.5 euros	9.38 euros
Total					150.84 euros

Para el desarrollo no se han empleado ningún tipo de licencia de pago, ya que todo el software y recursos empleados han sido gratuitos.

6.3. Coste total

	Coste (euros)
Desarrollo	2760 euros
Materiales	150.85 euros
Total	2910.85 euros

6.4. Perspectiva de negocio

Este motor de ajedrez ha sido desarrollado para la empresa Eclipse Games, destinado a ser parte de su próximo juego. Esta perspectiva es bastante nicho, queriendo decir que no cualquier empresa desarrollaría un motor de ajedrez, pero una empresa desarrolladora de videojuegos que se plantee desarrollar un videojuego de ajedrez en el que se pueda jugar contra una inteligencia artificial debe, o bien invertir tiempo de desarrollo en programar un motor de ajedrez o emplear alguna librería externa que lo implemente. Eclipse Games ha optado por de este primer caso, desarrollando un motor de ajedrez desde cero para su siguiente producto.

7. Resultados

Existen varias formas de medir la eficiencia de un motor de ajedrez, pero las medidas más relevantes para hacerlo son el tiempo que tarda en encontrar el mejor movimiento y el número de nodos que necesita visitar para encontrarlo.

Para realizar estas pruebas se van a medir los tiempos y los nodos necesarios para encontrar el mejor movimiento para búsquedas con y sin quiescencia, y con y sin ordenación MVV-LVA para diferentes tableros. Para esto, se empleará la ejecución mediante consola de comandos con tener un registro visual más amigable.

La primera búsqueda se realizará en el tablero inicial de ajedrez para una profundidad de 7 niveles.

```
8  ♜ ♞ ♝ ♛ ♚ ♝ ♞ ♜
7  ♟ ♟ ♟ ♟ ♟ ♟ ♟ ♟
6  . . . . . . . .
5  . . . . . . . .
4  . . . . . . . .
3  . . . . . . . .
2  ♙ ♙ ♙ ♙ ♙ ♙ ♙ ♙
1  ♖ ♘ ♗ ♕ ♔ ♗ ♘ ♖

   a b c d e f g h

   Side:      white
   Enpassant:    no
   Castling:   KQkq

depth 7
bestmove d2d4
nodes 22410860
time 39695ms
```

Ilustración 7-1. Búsqueda la ordenación MVV-LVA sobre el tablero inicial

Como podemos observar, el algoritmo Negamax sin la ordenación tarda hasta casi 40 segundos en recorrer los más de 22 millones de nodos que genera. Esto es obviamente mejorable, ya que no podemos permitir que nuestro motor tarde tanto en generar un movimiento. Vamos primero a añadir la búsqueda quiescente, a ver qué sucede con el número de nodos visitados.

```
8  ♜ ♞ ♝ ♛ ♚ ♝ ♞ ♜
7  ♟ ♟ ♟ ♟ ♟ ♟ ♟ ♟
6  . . . . . . . .
5  . . . . . . . .
4  . . . . . . . .
3  . . . . . . . .
2  ♙ ♙ ♙ ♙ ♙ ♙ ♙ ♙
1  ♖ ♘ ♗ ♕ ♔ ♗ ♘ ♖

   a b c d e f g h

   Side:     white
   Enpassant:   no
   Castling:  KQkq

depth 7
bestmove d2d4
nodes 1108779
time 1907ms
```

Ilustración 7-2. Búsqueda con ordenación MVV-LVA sobre el tablero inicial

Aunque realmente es lo esperado, siempre sorprende ver la increíble mejora con tan solo ordenar los movimientos con la técnica de MVV-LVA. Esta gran mejora en tiempo de ejecución está completamente ligada al gran decremento en el número de nodos a evaluar.

A continuación, se podrá ver el algoritmo completo en diferentes posiciones para exprimir su potencial al máximo y ver los tiempos de ejecución.

```
8  ♜ . . . ♚ . . ♜
7  ♟ . ♟ ♟ ♛ ♟ ♝ .
6  ♝ ♞ . . ♟ ♞ ♟ .
5  . . . ♙ ♘ . . .
4  . ♟ . . ♙ . . .
3  . . ♘ . . ♕ . ♟
2  ♙ ♙ ♙ ♗ ♗ ♙ ♙ ♙
1  ♖ . . . ♔ . . ♖

   a b c d e f g h

   Side:     white
   Enpassant:   no
   Castling:  KQkq

depth 7
bestmove e2a6
nodes 4755365
time 7816ms
```

Ilustración 7-3. Búsqueda con quiescencia y ordenación en posición compleja

Como era de esperar, a mayor complejidad en el tablero, más tarda el algoritmo. De todos modos, un tiempo de búsqueda de casi 8 segundos es asumible. Sobre todo, mejor que los 40 segundos originales. Este incremento es también debido al gran incremento en el número de nodos.

Finalmente, con la ayuda de una interfaz gráfica para jugar al ajedrez llamada Arena [] y gracias a que se ha implementado el motor para poder seguir el protocolo UCI (necesario para que un motor pueda ser utilizado en esta GUI), se ha podido poner a jugar a nuestro motor consigo mismo. Estas han sido las partidas que ha jugado contra sí mismo a lo largo del desarrollo, una vez que el motor ha podido jugar partidas por sí sólo.

Ilustración 7-4. Primera partida de BBChess contra sí mismo, justo después de implementar la función Negamax

Esta captura de pantalla muestra Arena, la interfaz gráfica anteriormente mencionada. En esta interfaz gráfica podemos cargar un ejecutable de un motor que implemente el protocolo UCI para poder jugar partidas.

BBChess (el nombre del motor implementado para este proyecto) cumple el requisito de seguir el protocolo UCI, así que se ha generado un ejecutable y se ha puesto a jugar a BBChess contra sí mismo.

En esta primera partida, BBChess ya buscaba movimientos, pero tan solo con el algoritmo Negamax y, claramente, como se puede ver por el movimiento ilegal que ha intentado ejecutar, aún no estaba listo al 100% para jugar partidas.

Ilustración 7-5. Segunda partida de BBChess contra sí mismo, capaz de ejecutar mates

Para esta partida se han resuelto los errores que causaron la ejecución de movimientos ilegales en posiciones perdidas que acaban en jaque mate; ahora es capaz de reconocer estas situaciones y la GUI se rinde para el lado que pierde.

En la foto se pueden ver los movimientos jugados en la partida y, finalmente, la rendición de las negras al reconocer el mate en 5 movimientos.

Ilustración 7-6. Tercera partida, implementada la búsqueda quiescente para evitar sacrificios de piezas

En esta iteración se ha implementado la búsqueda quiescente, que amenora considerablemente los sacrificios innecesarios de piezas. Como podemos ver aquí, el motor es capaz de ganarse a sí mismo, lo que nos permite poder (salvo *bugs* no detectados todavía) ser utilizado para jugar partidas.

Ilustración 7-7. Primera partida del motor BBChess contra otro motor, implementando la ordenación MVV-LVA

Esta es la primera partida que BBChess ha jugado contra un motor externo, llamado SOS, hecho por Rudolf Huber, motor que viene preinstalado en la interfaz gráfica Arena. En esta partida podemos ver que, aunque SOS ha conseguido ganar a BBChess, los tiempos han sido prácticamente el doble de rápidos para BBChess. Obviamente, BBChess todavía es mejorable en cuanto a algoritmos de ordenación y a mejorar la búsqueda, pero vemos que BBChess ya es capaz de enfrentarse a motores externos y conseguir partidas más que interesantes.

Ilustración 7-8. Segunda partida de BBChess implementando MVV-LVA contra un motor externo (Dragon, de Bruno Lucas)

Esta es la segunda partida que BBChess ha jugado contra un motor de terceros, en este caso contra Dragon, de Bruno Lucas, también preinstalado con la descarga de Arena GUI. En este caso, BBChess ha sorprendido con una defensa escandinava contra la apertura de e4 de Dragon, aunque esta defensa haya sido una completa casualidad, ya que BBChess no maneja aperturas o defensas, solo juega evaluando el tablero. Dragon ha conseguido una rápida victoria, e incluso con un mejor tiempo que BBChess.

Estas partidas sirven para ver el punto en el que se encuentra nuestro motor, poder comparar su rendimiento contra otros motores amateur/profesionales nos permite valorar el trabajo realizado y el que resta por implementar.

8. Conclusiones y futuros desarrollos

Este proyecto ha sido realmente una grata experiencia como desarrollador tras pasar el umbral tan oscuro que ha supuesto cambiar de prisma a la hora de ver datos estructurados de maneras que no son nada amigables para el programador. A la hora de desarrollar un proyecto como este, este cambio de perspectiva es completamente necesario y enriquecedor; al comienzo siempre te planteas "¿por qué no cambio de arquitectura ahora que aún estoy a tiempo?", pero esos pensamientos deben sucumbir ante la promesa de que, al finalizar este proyecto, la satisfacción que recibes al superar un reto como este sobrepasará todas esas tediosas horas invertidas en aprender nuevas formas de desarrollar software.

Lo que este proyecto me ha aportado es confianza a la hora de afrontar problemas con soluciones alternativas que ofrecen un mayor desempeño en rendimiento frente a soluciones que simplemente "funcionan" aunque no sea de la manera óptima. Haber elegido una arquitectura basada en tablas de bits me ha sacado de mi zona de confort, obligándome a investigar y aprender, y no solo seguir mi instinto a la hora de desarrollar.

En cuanto al desarrollo y la implementación como tal, no es que hubiera sido especialmente compleja de programar o desarrollar; la principal barrera viene al tener que invertir lo que al comienzo te parecen horas interminables leyendo artículos y siguiendo ejemplos hasta que por fin algo hace clic en tu mente y comprendes por qué se usan Bitboards, que estos no sólo sirven para representar piezas o casillas, sino para poder darle al ordenador los datos preparados para que este solo tenga que hacer lo que más le gusta, operar con ceros y unos. Cuando finalmente comprendes el por qué a todas estas cuestiones, el sentimiento de realización hace odiar tu yo del pasado que te sugería no seguir por este camino.

No todo es color de rosa, claramente. Por mucho que llegues a comprender el porqué de las estructuras que se usan o cómo funcionan, no dejan de ser representaciones alejadas por completo del lenguaje humano, y cuando aparece un *bug* sufres como nadie al depurar y ver un puñado de ceros y unos que no resultan en el puñado de ceros y unos que tú esperabas. Pierdes la noción de si el fallo lo has cometido tú a la hora de programar (que normalmente es el caso) o es que los ceros y unos que tu esperabas no eran los que debías esperar.

Los bits causan respeto, manejarlos es tedioso y operar con ellos supone más que un reto. Si alguien decidiese comenzar un desarrollo para un proyecto de este estilo, no hay que mentirle. ¿Son estas barreras un muro inamovible que evitar? Yo creo que para quien quiera aprender nuevas perspectivas a la hora de programar y estructurar datos, este proyecto es lo que busca.

A modo de recapitulación, nuestro motor BBChess es capaz de todo lo que un motor de ajedrez debe ser capaz de realizar. Por un lado, puede traducir cadenas FEN a nuestra lógica de Bitboards y operar con los mismos, puede generar tablas de ataques precalculadas para reducir la carga computacional en ejecución, es capaz de generar movimientos para una posición dada y de ejecutar movimientos, evaluar una posición y, por último, pero no menos importante, buscar el mejor movimiento.

Aunque todas estas funcionalidades son capaces de realizar su cometido, es cierto que el desarrollo se ha centrado en la implementación de las funcionalidades más específicas a la arquitectura basada en Bitboards más que en desarrollar algoritmos óptimos, por ejemplo, para la ordenación de movimientos en MVV-LVA. Es por esto por lo que, para futuro desarrollo, el foco puede apuntar a optimizar estos algoritmos por implementaciones que destaquen por la alta eficiencia y mejoren el rendimiento del motor, manteniendo las funcionalidades *core*.

Además, se pueden implementar más mejoras para el algoritmo de búsqueda y reducir al mínimo el tiempo de ejecución del mismo, como implementar la Variación Principal (PV) o tablas de aperturas para que el motor sea capaz de identificar y seguir aperturas típicas en el ajedrez.

9. Bibliografía

Chess Programming Wiki. (27 de Abril de 2018). *Negamax*. Obtenido de ChessProgrammingWiki.org: https://www.chessprogramming.org/Negamax

Chess Programming Wiki. (15 de Enero de 2022). *Minimax*. Obtenido de ChessProgrammingWiki.org: https://www.chessprogramming.org/Minimax

Chess Programming Wiki. (s.f.). *Alpha-Beta*. Obtenido de ChessProgramming.org: https://www.chessprogramming.org/Alpha-Beta

Chess Programming Wiki. (s.f.). *General Setwise Operations*. Obtenido de ChessProgrammingWiki.org: https://www.chessprogramming.org/General_Setwise_Operations

Chess Programming Wiki. (s.f.). *Hash Table - Perfect Hashing*. Obtenido de ChessProgrammingWiki.org: https://www.chessprogramming.org/Hash_Table#PerfectHashing

Chess Programming Wiki. (s.f.). *Iterative Deepening*. Obtenido de ChessProgrammingWiki.org: https://www.chessprogramming.org/Iterative_Deepening

Chess Programming Wiki. (s.f.). *MVV-LVA*. Obtenido de ChessProgrammingWiki.org: https://www.chessprogramming.org/MVV-LVA

Chess Programming Wiki. (s.f.). *Null Move Ordering*. Obtenido de ChessProgrammingWiki.org: https://www.chessprogramming.org/Null_Move_Observation

Chess Programming Wiki. (s.f.). *Null Move Pruning*. Obtenido de ChessProgrammingWiki.org: https://www.chessprogramming.org/Null_Move_Pruning

Chess Programming Wiki. (s.f.). *Principal Variation*. Obtenido de ChessProgrammingWiki.org: https://www.chessprogramming.org/Principal_Variation

Chess Programming Wiki. (s.f.). *Quiescense Search*. Obtenido de ChessProgrammingWiki.org: https://www.chessprogramming.org/Quiescence_Search

Chess Programming Wiki. (s.f.). *Zugzwang*. Obtenido de ChessProgrammingWiki.org: https://www.chessprogramming.org/Zugzwang

Chess.com. (7 de Mayo de 2019). *Computer Chess Engines: A Quick Guide*. Obtenido de Chess.com: https://www.chess.com/article/view/computer-chess-engines

Chess.com. (s.f.). *Forsyth-Edwards Notation (FEN)*. Obtenido de Chess.com: https://www.chess.com/terms/fen-chess

GNU GCC. (s.f.). *Function-like Macros*. Obtenido de gcc.gnu.org: https://gcc.gnu.org/onlinedocs/cpp/Function-like-Macros.html

Wikipedia. (5 de Agosto de 2022). *Chess engine*. Obtenido de Wikipedia.org: https://en.wikipedia.org/wiki/Chess_engine

Wikipedia. (26 de Julio de 2022). *Iterative deepening depth-first search*. Obtenido de Wikipedia.org: https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search

Wikipedia. (17 de Julio de 2022). *Plankalkül*. Obtenido de Wikipedia.org: https://en.wikipedia.org/wiki/Plankalk%C3%BCl

Wikipedia. (s.f.). *Bit Numbering - LSB o bit numbering*. Obtenido de Wikipedia.org: https://en.wikipedia.org/wiki/Bit_numbering#LSB_0_bit_numbering

Wikipedia.org. (29 de Agosto de 2022). *Bubble sort*. Obtenido de Wikipedia.org: https://en.wikipedia.org/wiki/Bubble_sort

Ilustración 1-1. Máscaras de bits para la ocupación de cada pieza en el tablero inicial 3
Ilustración 1-2. Ejemplo de representación de un tablero en forma de cadena FEN 4
Ilustración 2-1. Secuencia de movimientos en las que el peón negro de la columna B captura el peón blanco de la columna A al paso, abierto en la casilla A3 tras el movimiento a4. 11

Ilustración 2-2. La representación FEN para el posicionamiento de las piezas para este tablero sería r1bk3r/p2pBpNp/n4n2/1p1NP2P/6P1/3P4/P1P1K3/q5b1 12
Ilustración 2-3. Plantilla de representación, adaptada a un tablero, de un Bitboard 18
Ilustración 2-4. Máscara de bits representando todas las casillas a las que potencialmente puede moverse un alfil situado en C3 19
Ilustración 2-5. Bitboard de la ocupación en el tablero de todas las piezas negras para una posición arbitraria 19
Ilustración 2-6. Máscaras de posibles ataques de un rey situado en diferentes posiciones del tablero 20
Ilustración 2-7. Representación gráfica del proceso de generación de Bitboards Mágicos 22
Ilustración 2-8. Reparto de bits por propiedades para codificar un movimiento en un int32 24
Ilustración 4-1. Board de Notion con las funcionalidades del proyecto 28
Ilustración 4-2. Listado simple en Notion de las reuniones con mi tutor de proyecto. 29
Ilustración 5-1. Diagrama de clases del motor de ajedrez implementado 31
Ilustración 5-2. Enumeración de C++ para identificar las casillas por índice en el tablero 32
Ilustración 5-3. Enumeración de los enroques orientada a operar entre ellos para obtener los enroques disponibles. 33
Ilustración 5-4. Enumeración de las piezas ordenadas 33
Ilustración 5-5. Funciones Macro para activar, desactivar y leer bits dentro de un Bitboard. 34
Ilustración 5-6. Implementación de la función para contar bits activos en un Bitboard 35
Ilustración 5-7. Implementación de la función para encontrar el índice del LSB 36
Ilustración 5-8. Representación gráfica de los bits del Bitboard de los ataques de un peón negro en E7, así como el de un peón blanco en E5 38
Ilustración 5-9. Algoritmo para calcular el Bitboard de ataques de un peón de un color en una casilla. 39
Ilustración 5-10. Ejemplo de utilidad de NotHFile en un caso de uso donde es necesaria la comprobación 39
Ilustración 5-11. Bitboards "helpers" para condicionar la activación de bits al generar ataques que pueden atravesar los laterales del tablero 40
Ilustración 5-12. Implementación del algoritmo para calcular el Bitboard de ataques del caballo para una casilla 41
Ilustración 5-13. Ataques de un caballo de cualquier color situado en la casilla D4 41
Ilustración 5-14. Ejemplo visual de la utilidad del Bitboard NotGHFile para evitar desplazamientos ilegales 42
Ilustración 5-15. Implementación del algoritmo para generar los ataques del rey para una casilla 42
Ilustración 5-16. Representación del Bitboard de ataques generado para un rey situado en la casilla F3 43
Ilustración 5-17. Ejemplo de Bitboard de ataques de alfil creando falsos ataques en la diagonal de B2 a H8, siendo esta bloqueada en D4 por otra pieza blanca 43
Ilustración 5-18. Implementación de la función capaz de generar una máscara de ataques para un alfil en una casilla cualquiera 44
Ilustración 5-19. Implementación de la función para generar los ataques de un alfil en tiempo real, teniendo en cuenta la casilla en la que se encuentra y la ocupación del tablero 45
Ilustración 5-20. Implementación de la función para calcular los Bitboards de ataque de las torres para una casilla cualquiera 46
Ilustración 5-21. Implementación de la función para generar en tiempo real los ataques de una torre en una casilla arbitraria dada una ocupación 46
Ilustración 5-22. Función para calcular una combinación de ocupaciones para una pieza deslizante 47
Ilustración 5-23. Implementación de la función de inicialización de las tablas de ataques de todas las piezas deslizantes (alfiles y torres, dama obtenida de la unión de ambas) 48
Ilustración 5-24. Declaración de las listas para almacenar los Bitboards para la ocupación de cada pieza y las ocupaciones por colores 49
Ilustración 5-25. Inicialización de las propiedades relativas a la cadena FEN para su parseo. ... 50

Ilustración 5-26. Algoritmo de parseo de posicionamiento de las piezas de una cadena FEN....50
Ilustración 5-27. Implementación del parseo del color actual a jugar y los derechos restantes de enroque 51
Ilustración 5-28. Algoritmo para parsear la casilla abierta para captura al paso en una cadena FEN 51
Ilustración 5-29. Implementación del parseo de los contadores de movimientos y composición de la lista de ocupaciones. 52
Ilustración 5-30. Implementación de la función para conseguir los ataques que una pieza puede realizar 52
Ilustración 5-31. Constructor de la estructura de datos que almacena toda la información de un movimiento individual 53
Ilustración 5-32. Implementación del algoritmo para generar los movimientos sin capturas del peón. 54
Ilustración 5-33. Funciones Macro para la preservación y restauración de un estado del tablero. 58
Ilustración 5-34. Implementación de la función para ejecutar movimientos en el tablero....59
Ilustración 5-35. Lista con los valores de las piezas, ordenadas para coincidir con el mapeo de las piezas 60
Ilustración 5-36. Tablas que recogen el valor posicional de cada pieza sobre el tablero....61
Ilustración 5-37. Mapeo de una casilla a la equivalente desde la posición del color opuesto....62
Ilustración 5-38. Implementación de la función de evaluación de una posición del tablero. 62
Ilustración 5-39. Implementación de la función de búsqueda del mejor movimiento 63
Ilustración 5-40. Condición de escape de la búsqueda Negamax. Ejecución de la búsqueda quiescente. 64
Ilustración 5-41. Inicialización de los recursos necesarios por cada iteración de la búsqueda. ..64
Ilustración 5-42. Generación de movimientos para el estado actual del tablero e iteración sobre los mismos para obtener el mejor movimiento....65
Ilustración 5-43. Devolver puntuación de los nodos sin movimientos legales disponibles, dependiendo de la vulnerabilidad del rey....66
Ilustración 5-44. Comparación del movimiento encontrado con el anterior movimiento 67
Ilustración 5-45. Implementación de la búsqueda quiescente. 68
Ilustración 5-46. Tabla para conseguir el valor de una captura de una pieza a otra....69
Ilustración 5-47. Implementación de función accesoria para la puntuación de un movimiento mediante la heurística MVV-LVA 70
Ilustración 5-48. Implementación del algoritmo Bubble-sort de ordenación de movimientos por su puntuación....71
Ilustración 7-1. Búsqueda la ordenación MVV-LVA sobre el tablero inicial 74
Ilustración 7-2. Búsqueda con ordenación MVV-LVA sobre el tablero inicial 75
Ilustración 7-3. Búsqueda con quiescencia y ordenación en posición compleja....76
Ilustración 7-4. Primera partida de BBChess contra sí mismo, justo después de implementar la función Negamax....77
Ilustración 7-5. Segunda partida de BBChess contra sí mismo, capaz de ejecutar mates 78
Ilustración 7-6. Tercera partida, implementada la búsqueda quiescente para evitar sacrificios de piezas 79