

Inteligencia Artificial para Videojuegos

Práctica 1 : Puzzle de 8 piezas



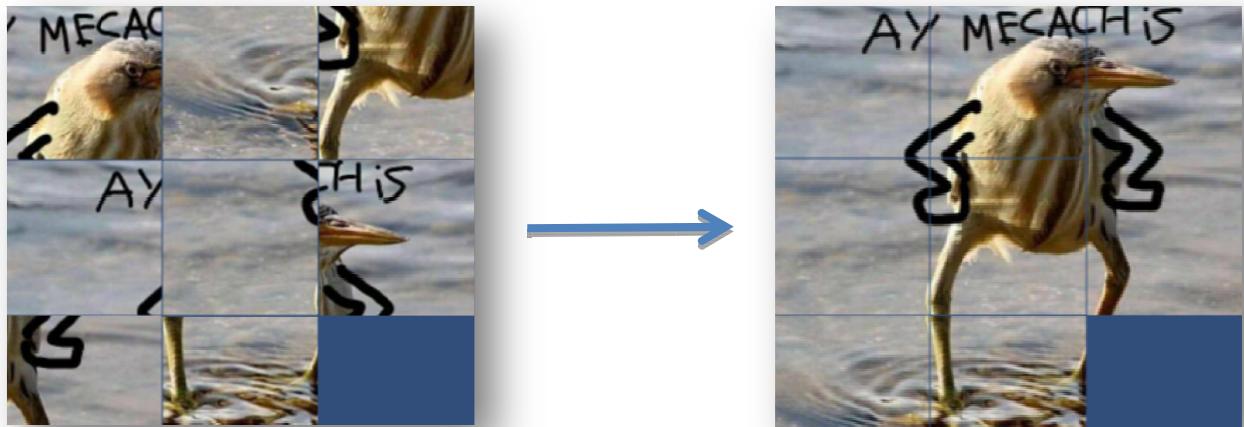
Grupo 07

Alejandro Ortega Álvarez

Borja Cano Álvarez

Problema a resolver

- La propuesta es realizar en Unity el típico juego para niños de puzzle de 9 piezas que forman una imagen o un conjunto de números, donde falta una pieza para poder mover las 8 restantes y así poder ordenar o desordenar el desafío.

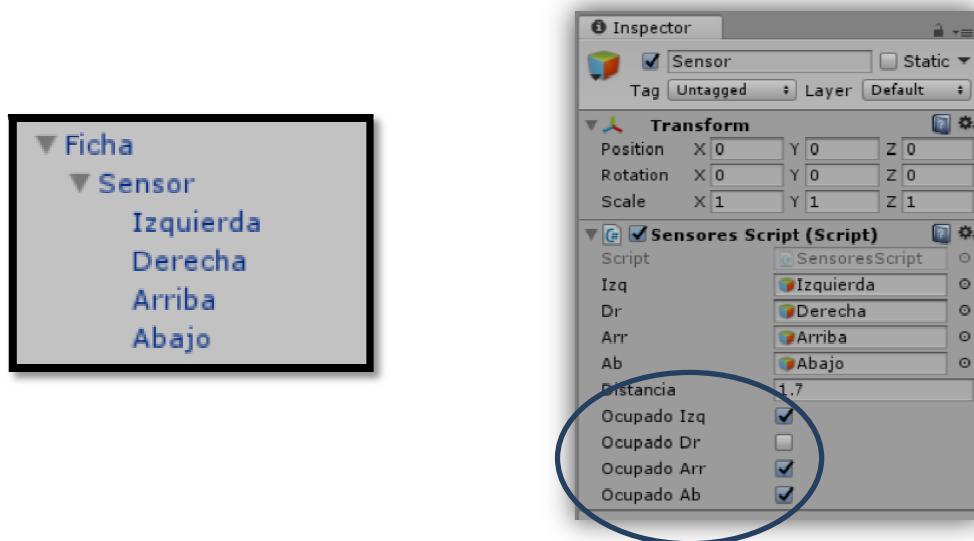


- Para ello se piden los siguientes requisitos:
 1. Generar el puzzle de 8 piezas, que se puedan mover
 2. Un contador de movimientos
 3. Un botón para reiniciar
 4. Un botón para resolver el juego automáticamente mediante 2 algoritmos

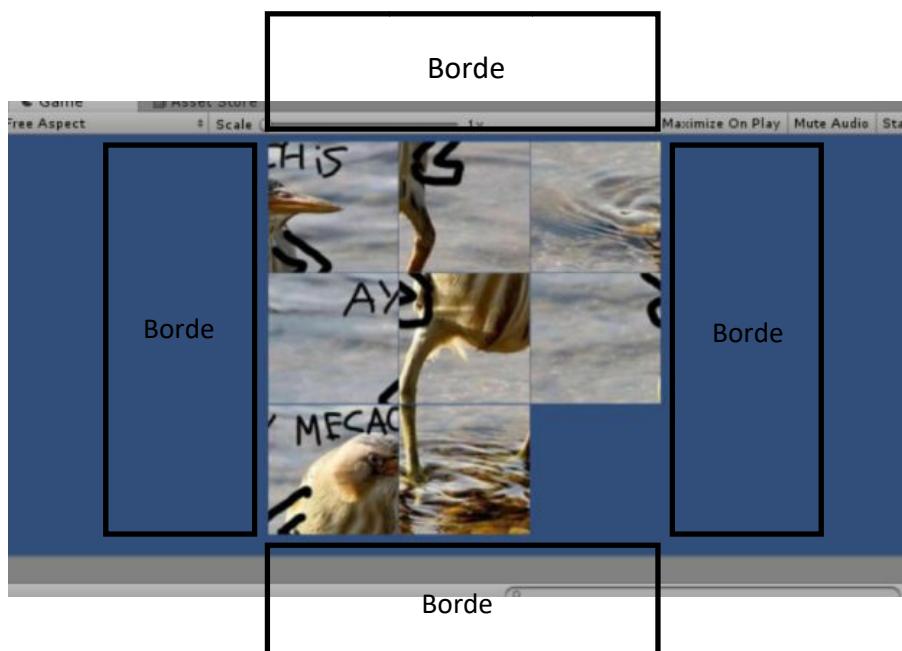
Cómo lo hemos hecho

GameObjects

- El GameObject principal del juego es la “ficha”. Consta de 4 GameObjects llamados sensores. Estos sensores se encargan de detectar en su Script con la función Overlap si un cierto radio existe la presencia o no de otra ficha (o un borde). Éstos valores los almacenamos en 4 booleanos y los podremos usar después para saber si hay hueco para mover la ficha o no.



- Los demás GameObjects del juego son los bordes (triggers invisibles con Box collider para detectarlos), y los textos y botones que se ven en el juego.



Scripts

- En el Script principal del juego es el GameManager encargado de gestionar todo el juego. Consta de los siguientes métodos:
 - o El método Start() crea el tablero de fichas (que lo almacenamos en un array) con las posiciones correctas de cada ficha, que también las guardamos para luego compararlas con las posiciones mientras se juega. Además crea los bordes y llama al método Baraja().
 - o El método Baraja inicializa todos los contadores de tiempo y movimientos a 0, y crea nuevas posiciones aleatorias para cada ficha (la forma de dar posiciones aleatorias hemos decidido que sea recorriendo el tablero y por cada ficha, intercambiar su posición por otra ficha del tablero aleatoriamente).
 - o El método Victoria() comprueba si se ha ganado o no la partida comparando las posiciones correctas con las posiciones que actualmente tiene cada ficha.
 - o El método mueveFicha() se encarga de, recibiendo el identificador de una ficha, mover dicha ficha en la dirección donde ésta pueda, si es que puede (ya que no puede encontrar más de un hueco para moverla si lo hay). Si hay hueco, mueve la ficha, suma movimientos y comprueba si se ha ganado la partida con ello (llamando a Victoria()).
- El Script Index se encarga de dar un identificador a cada ficha para que sea única y podamos saber cuál es en todo momento.
- El Script BoardPosition dota a la ficha de un Vector2Int, el cual representa su posición en el tablero (valores entre (0,0) y (2,2) siendo (0,0) la esquina superior izquierda).
- Cada ficha tiene un Script Move, que detecta si se ha hecho click sobre ella para poder moverla llamando al método mueveficha() del GameManager.
- El Script de los sensores se encarga de ver con la función Overlap, si en una distancia X entra en contacto con un box collider o no, y lo almacena en el booleano que corresponda.
- Por último, el Script de UI es el que gestiona los textos del juego, qué muestran y se reinician a 0. Hay un texto que indica los movimientos, un texto que indica el tiempo transcurrido y un texto final que muestra las estadísticas.
- SCRIPT DEL ALGORITMO, EXPLICADO EN LA SIGUIENTE PAG

Algoritmo

Cuando empezamos a estudiar el problema que se nos presentaba buscamos en internet y entramos un algoritmo que clamaba ser el más sencillo con una eficacia muy buena. Este es el algoritmo “Hill Climbing”. Pasamos a explicarlo a continuación.

Este algoritmo necesita de dos listas:

- Una lista con los estados abiertos (esta lista idealmente consiste en una cola de prioridad de mínimos).
- Una lista con los estados ya visitados.

Ahora el algoritmo actúa de la siguiente manera:

1. Añadimos el estado actual a la lista de estados abiertos.
 2. Comprobamos que la cabeza de la lista es la solución, si lo es terminar. Si la lista está vacía se reporta un error, no se ha encontrado solución.
 3. Se generan todos los estados posibles a partir del estado evaluado anteriormente.
 4. Se descartan los estados que ya hemos visitado (están en la lista de visitados)
 5. Se añaden los estados que queden a la lista de estados abiertos.
 6. Se ordena la lista de estados abiertos de acuerdo al mínimo coste*
 7. Volver al paso 2.
- El coste se calcula con “la distancia Manhattan” el cual suma las distancias en bloques entre las posiciones actuales de las fichas y las que deberían ser.

Para encontrar la secuencia de movimientos que darían al estado resuelto se hace backtracking desde el nodo final hasta el inicial.

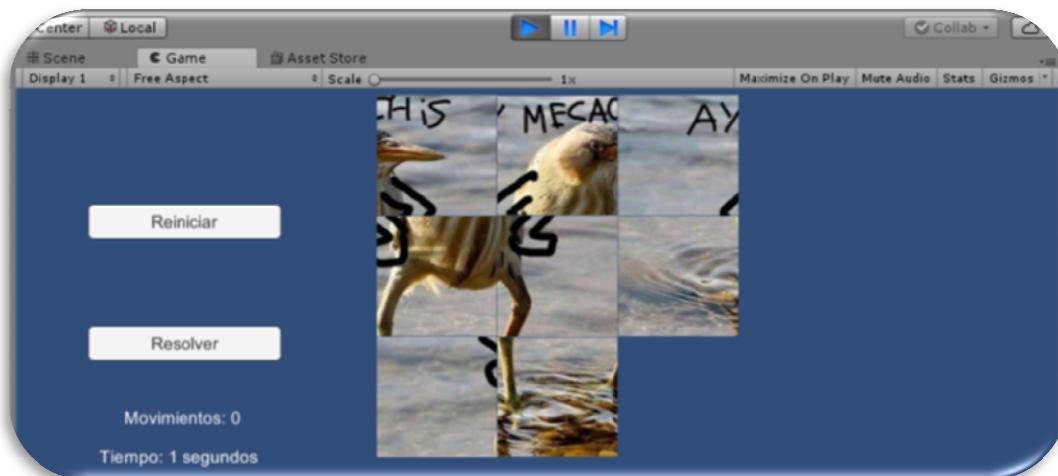
La magnitud del problema es relativamente alto ($!9/2 = 181,440$ estados posibles), por eso una de las bondades de este algoritmo es la optimización en cuanto al recorrido de nodos, ya que sólo se recorren nodos nuevos, evitando así también bucles.

Además gracias a la heurística empleada dado que, idealmente, sólo se pasan por los estados más prometedores (aunque se permite abandonar el nodo (no se trata de un algoritmo voraz))

Además está demostrado que con algoritmos muy similares a este el juego puede ser resuelto en 30 movimientos o menos desde cualquier estado inicial.

Juego final

Ésta es la pantalla que se muestra al comenzar el juego



-Si se hace click en una ficha que se pueda mover, se pondrá en la posición del hueco y sumará 1 al contador de movimientos.

-Si se pincha el botón de reiniciar, se ponen a 0 los contadores de movimiento y tiempo y se vuelven a barajar las fichas (de forma completamente distinta a la anterior y dejando el hueco en el mismo sitio, abajo a la derecha).

-Si se pincha en el botón resolver, desde el estado actual del juego y mediante el algoritmo, se moverán solas las piezas hasta completar automáticamente el puzzle (mostrando los movimientos también).

-Si se completa el puzzle, ya sea jugando o dándole al botón de resolver, sale la imagen completa con un texto que te informa de que has ganado y otro texto que muestra las estadísticas. Si se desea empezar otra partida sólo hay que darle al botón de reiniciar y se volverá todo como al principio.



Observaciones y comentarios

La observación más importante es que no hemos conseguido implementar funcionalmente el resolutor automático, entre las causas de ello podemos destacar los siguientes:

- La adaptación a C#: este es el mayor causante de todo. Tras tanto acostumbramiento a C++ se ha sentido increíblemente costoso acostumbrarse a la sintaxis y la filosofía de C#.
- Mala interpretación de la documentación consultada. Esto es que implementamos el algoritmo de forma iterativa cuando debería de haberse implementado de forma recursiva. Este error a priori no es fatal, ya que cualquier bucle iterativo tiene su traducción recursivo, pero dado a la falta de tiempo y a la poca comodidad que tenemos actualmente con el lenguaje no nos ha sido posible. Este punto es importante porque de forma recursiva el backtracking necesario para obtener la secuencia de movimientos necesaria para solucionar el problema es increíblemente natural, mientras que de manera iterativa es altamente complicado.
- En cuanto al juego “vanilla” es perfectamente funcional.

Referencias y material utilizado

Referencias y apoyos del desarrollo de la práctica:

- *Apuntes del Campus Virtual (tanto de IAV como de MARP)*
- *Tutoriales de Unity en Youtube*
- <https://towardsdatascience.com/solve-slide-puzzle-with-hill-climbing-search-algorithm-d7fb93321325>
- <https://code.msdn.microsoft.com/windowsdesktop/Graphical-A-Search-for-a-2461a8f9>
- <https://www.codeproject.com/Articles/616874/Puzzle-using-A-A-Star-Algorithm-Csharp>