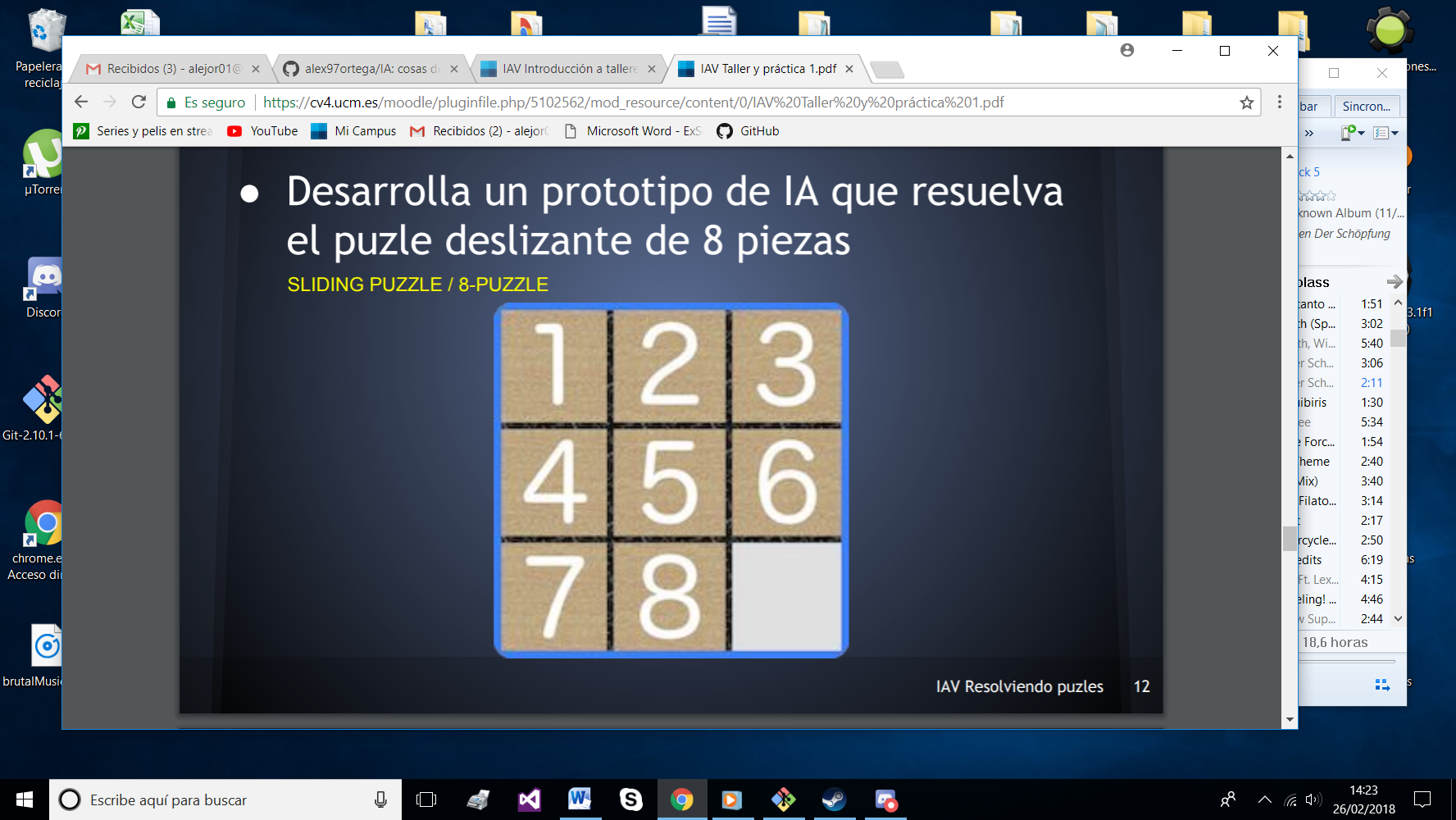
**Inteligencia Artificial para Videojuegos**

***Práctica 1 : Puzle de 8 piezas***



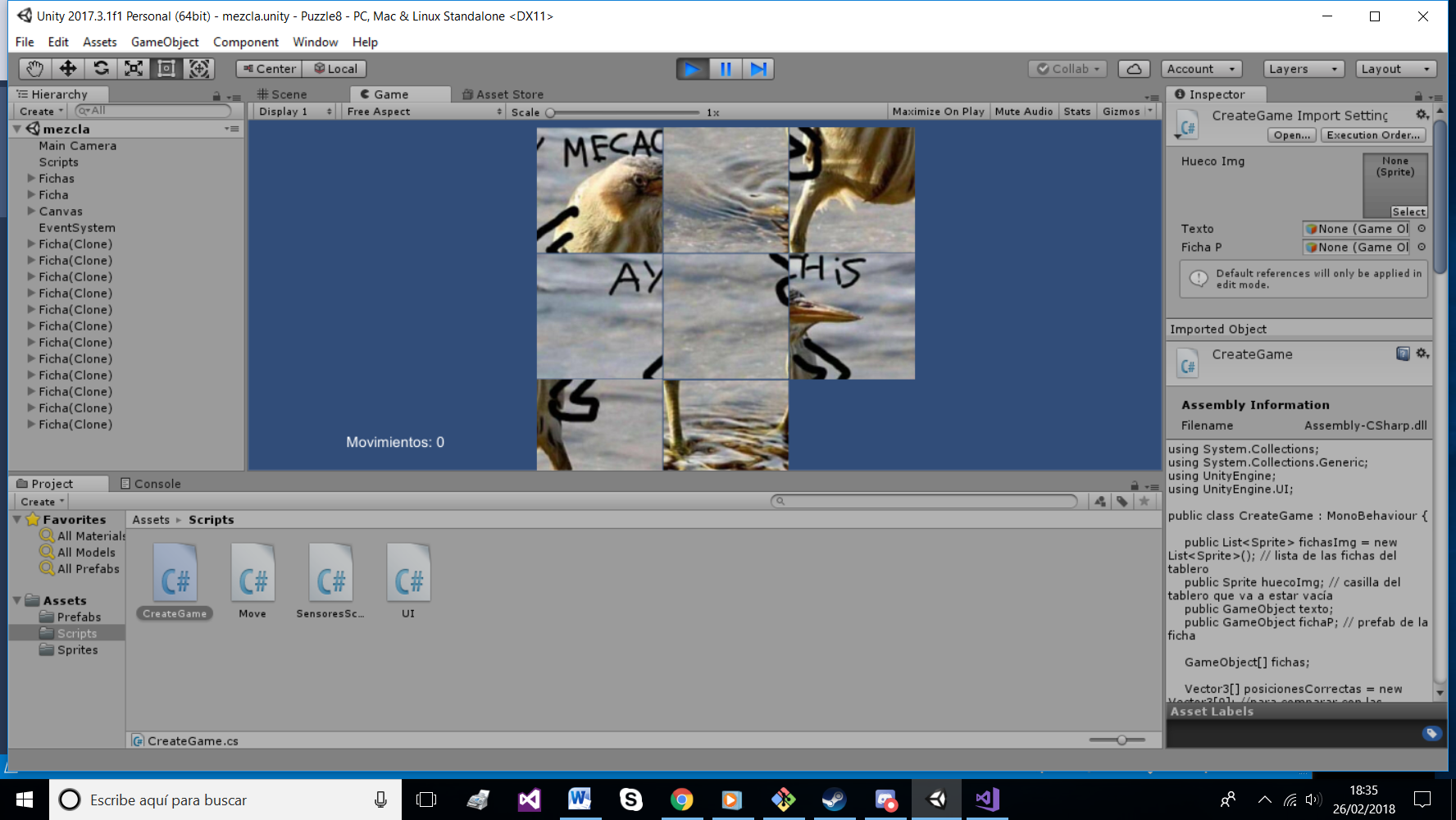
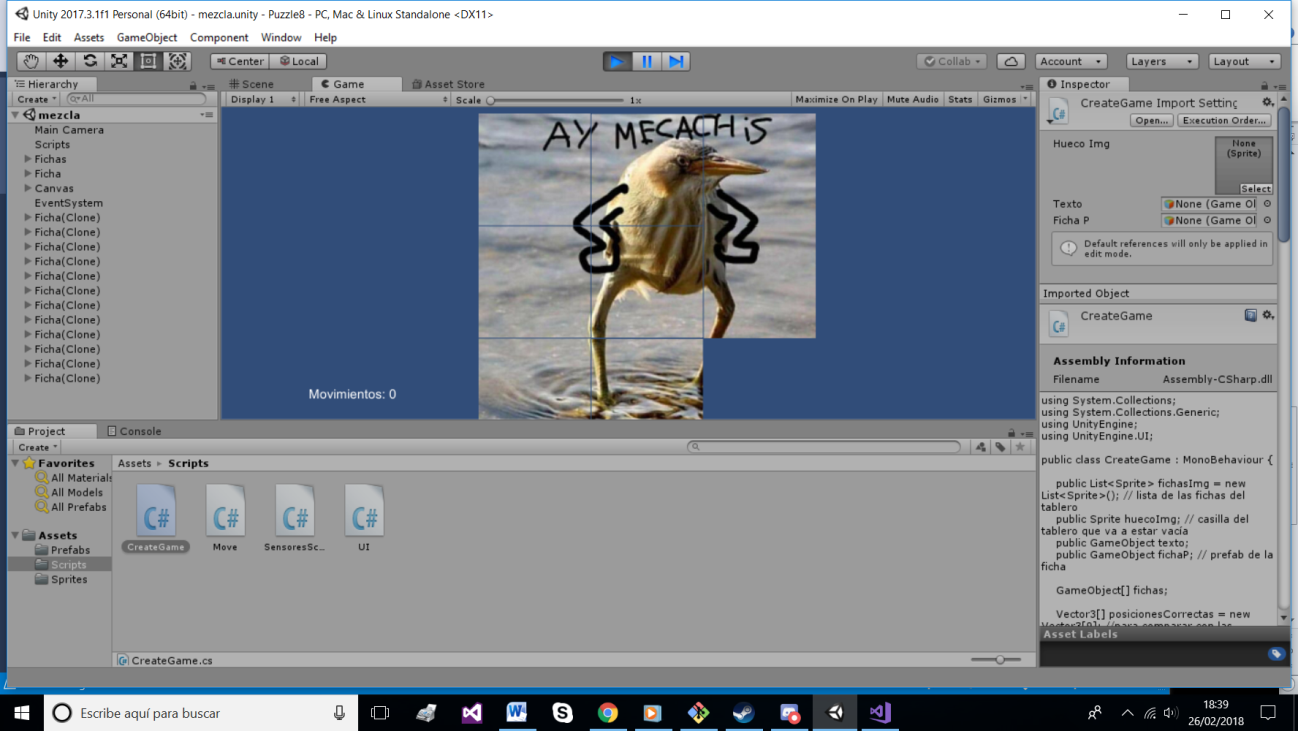
*Grupo 07*

*Alejandro Ortega Álvarez*

*Borja Cano Álvarez*

Problema a resolver

* *La propuesta es realizar en Unity el típico juego para niños de puzle 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.*



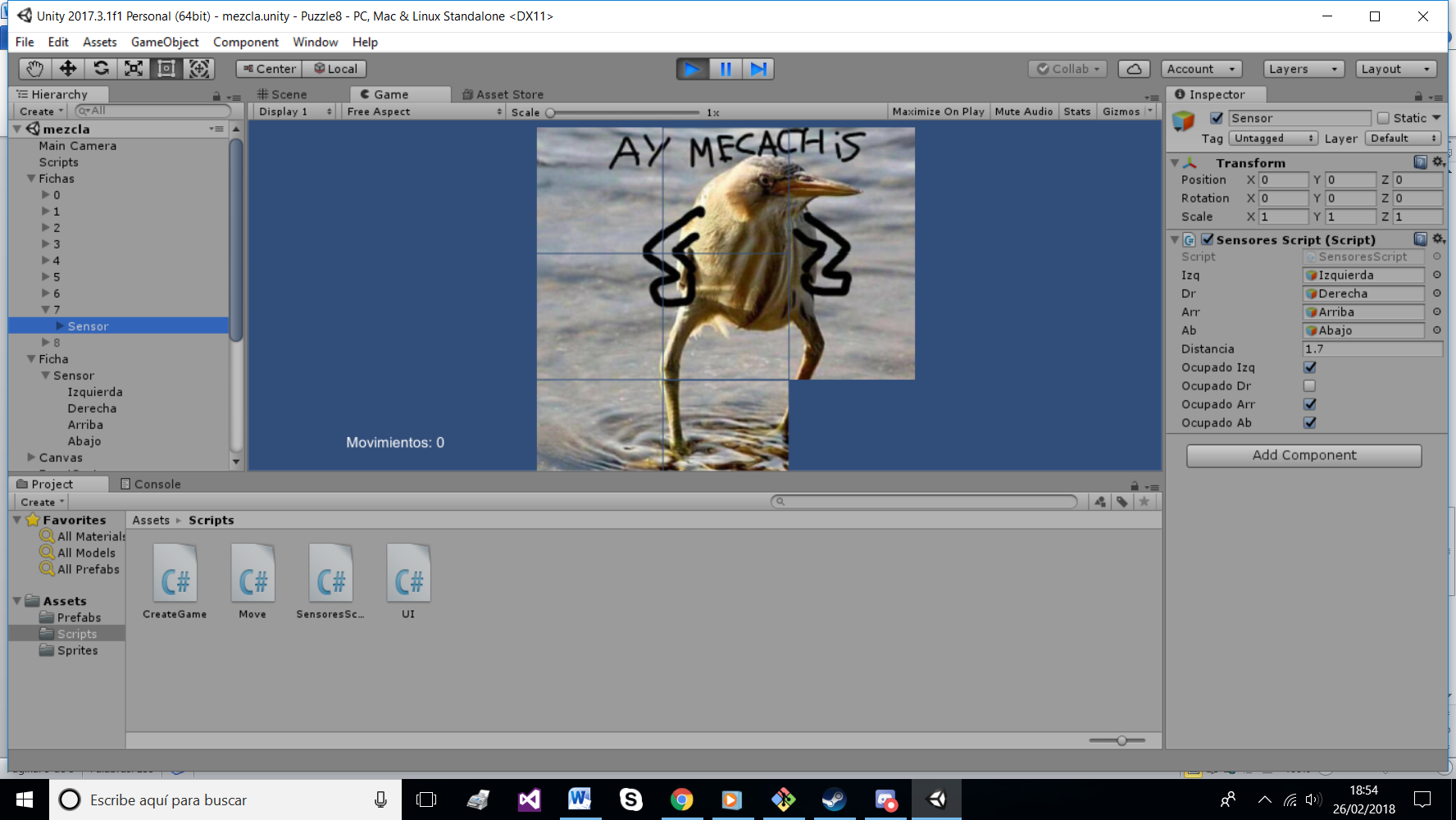
* *Para ello se piden los siguientes requisitos:*

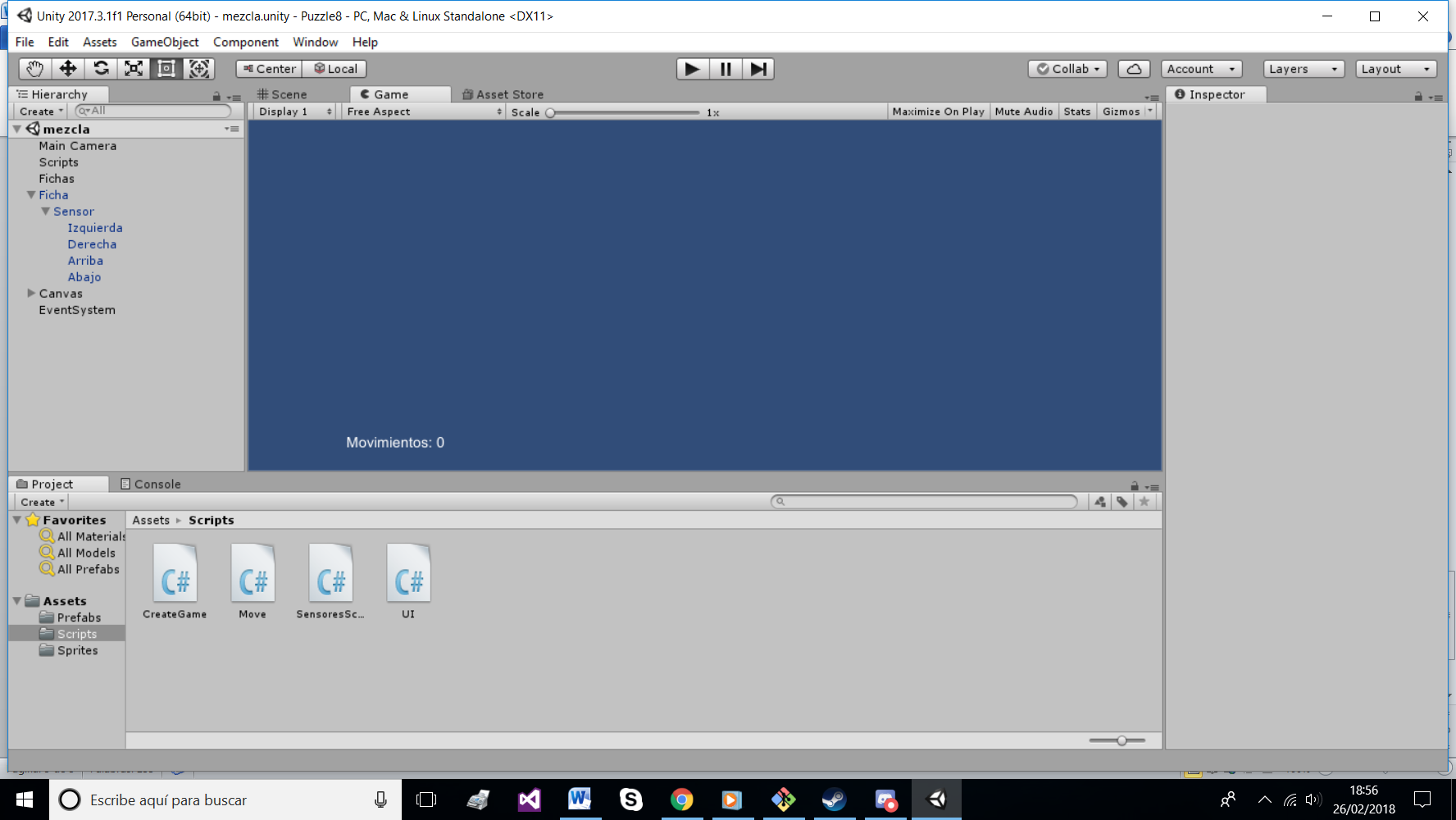
1. *Generar el puzle de 8 piezas, que se puedan mover*
2. *Un contador de movimientos*
3. *Un botón para reiniciar*
4. *Un botón para resolver el juego automáticamente mediante 2 algoritmos*

Cómo lo hemos hecho

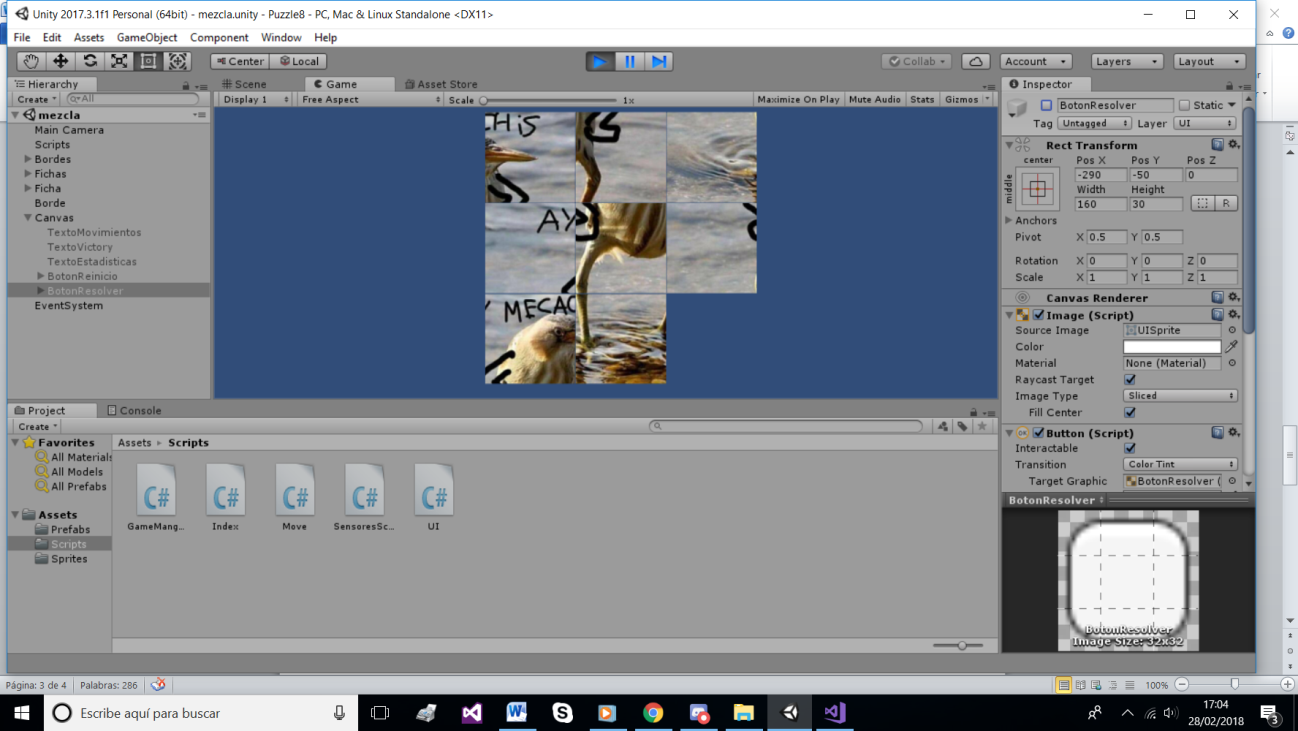
***GameObjects***

* 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.

Borde

Borde

Borde

Borde

***Scripts***

* En el Script principal del juego es el GameManager encargado de gestionar todo el juego. Consta de los siguientes métodos:
  + 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().
  + 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).
  + El método Victoria() comprueba si se ha ganado o no la partida comparando las posiciones correctas con las posiciones que actualmente tiene cada ficha.
  + 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 Victora()).
* 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

Algoritmos

Tras investigar por internet dimos con el algoritmo “Hill Climbing” el cuál clamaba ser el más sencillo por lo que decidimos utilizarlo.

El algoritmo consiste en lo siguiente:

* Una lista de estados candidatos
* Un lista de estados visitados

Pasos:

1. Se inserta el estado actual en la lista de candidatos
2. Se comprueba que el estado actual sea el resuelto, de no ser así continuar
3. Se generan todos los estados posibles a partir de la cabeza de la lista
4. Descartamos los estados que ya estén contenidos en la lista de vistos
5. Se añaden los estados resultantes a la lista de candidatos.
6. Se saca el estado a la cabeza de la lista de candidatos y se añade a la de vistos
7. Se re-ordena la lista de manera que el estado con menor coste calculado con la función m() se pone a la cabeza
8. Se vuelve a 2. Si la lista de candidatos está vacía se reporta un error

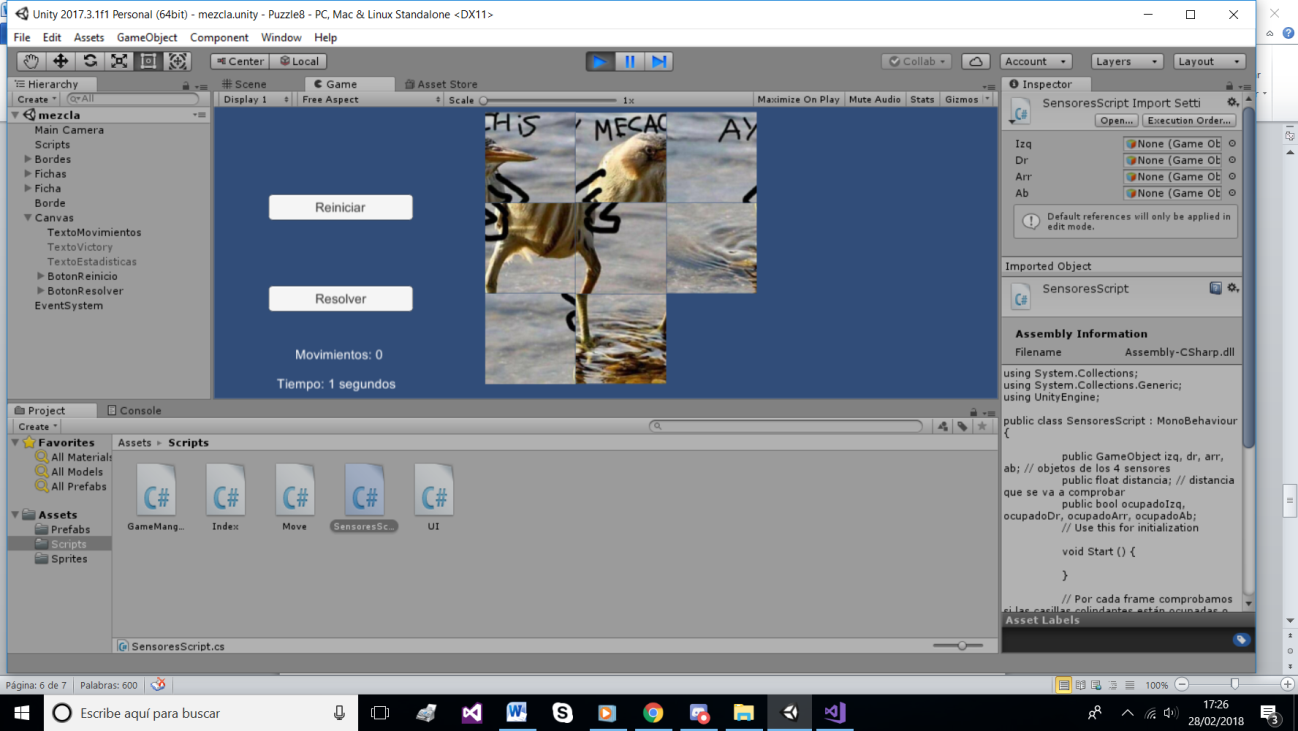
M() (Cálculo de coste)

El coste del estado se calcula midiendo la distancia en casillas desde la ficha que se ha movido para generar el estado hasta la posición final de la misma

Luego por recursión tomamos las fichas que hay que mover para llegar al estado resuelto.

Juego final

Ésta es la pantalla que se muestra al comenzar el juego

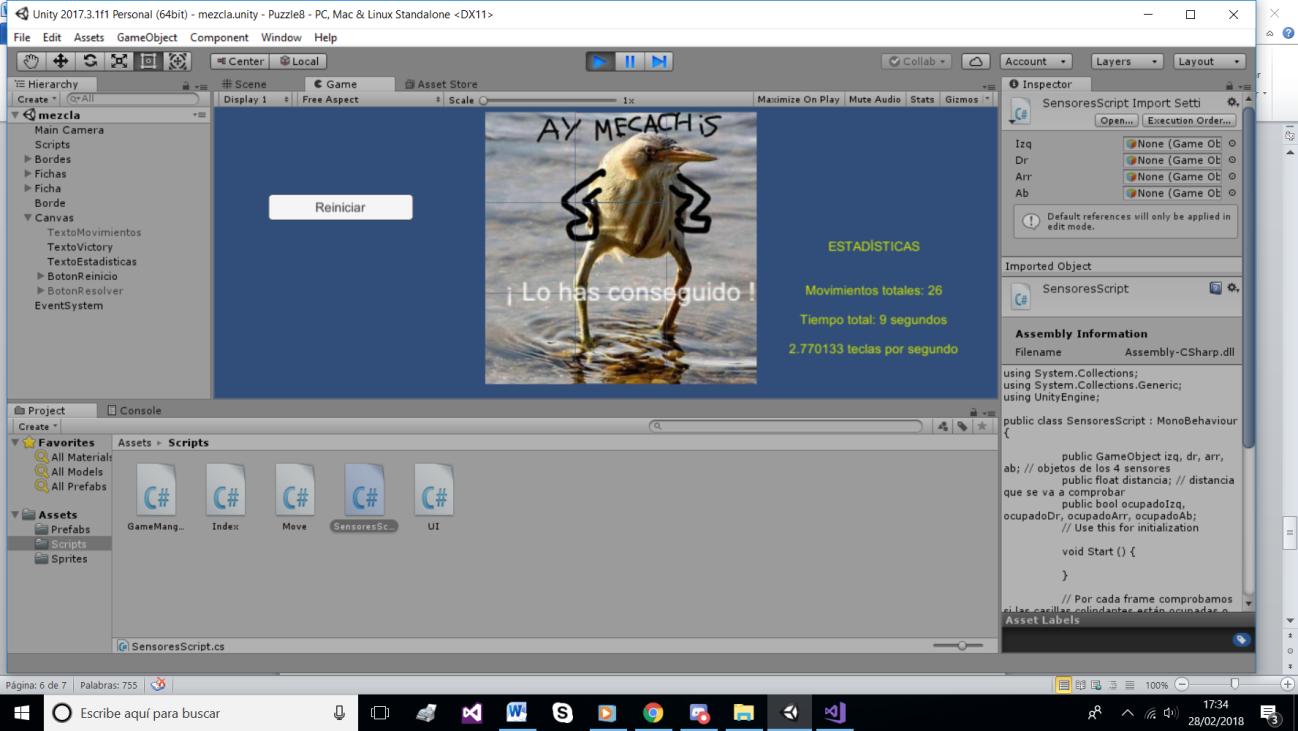


-Si se hace click en una ficha que se pueda mover, se pondrá en la posición del hueco y sumará 1 al contador de movimientos.

-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 puzle (mostrando los movimientos también).

-Si se completa el puzle, 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.



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://towardsdatascience.com/solve-slide-puzzle-with-hill-climbing-search-algorithm-d7fb93321325)