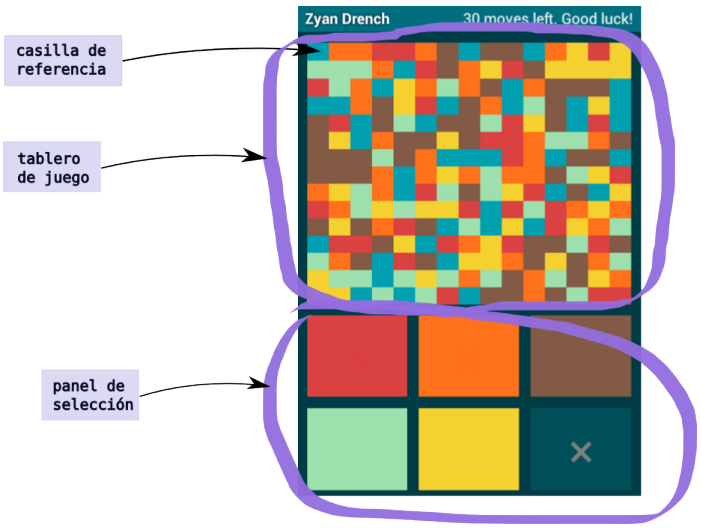
Fundamentos de la programación

Práctica 3. Zyan Drench

Indicaciones generales:

- **Lee atentamente** el enunciado e implementa el programa tal como se pide, con los métodos y requisitos que se especifican.
- El programa, además de ser correcto, debe estar bien estructurado y comentado. Se valorarán la claridad, la concisión y la eficiencia.
- La entrega se realizará a través del campus virtual, subiendo únicamente el archivo `Program.cs` (generado por MonoDevelop), con el programa completo.
- Los nombres de los participantes se pondrán como comentario al principio de dicho archivo (antes de `using System; namespace ...`, etc).
- El **plazo de entrega** finaliza el día 10 de marzo de 1016.

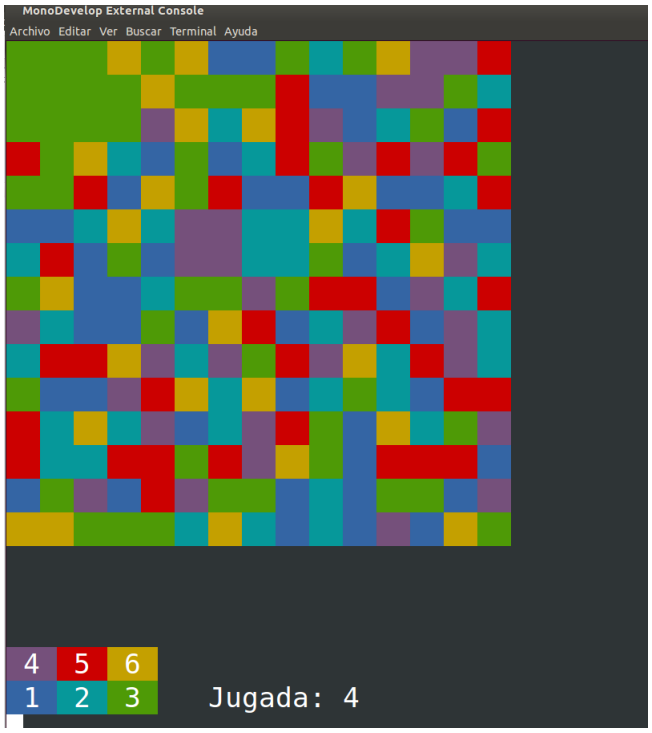
Zyan Drecnh es un juego para Android que puede encontrarse en `https://drench.codeplex.com/`. Aunque tiene unas reglas muy elementales es un juego muy entretenido y adictivo. Consta de un tablero de 15x15 casillas que pueden ser de 6 colores diferentes y un panel de selección con esos 6 colores:



1. Descripción del juego

La mecánica del juego es simple (explicaremos e implementaremos el modo *classic, single player*). La casilla superior izquierda es la **casilla de referencia**. En cada jugada, el jugador elige un color en el panel de selección y entonces todas las casillas adyacentes a la casilla de referencia que sean de su mismo color se cambian a dicho color (incluida la propia casilla de referencia). Eligiendo adecuadamente los colores, en sucesivas jugadas se va ampliando la región de casillas del mismo color, hasta que finalmente se obtiene un tablero monocromo (en el juego original se limita a 30 el número de jugadas).

En nuestra versión para consola el juego tendrá el siguiente aspecto:



El tablero es igual que en el juego original y el panel de selección se muestra abajo, con los colores numerados de 1 a 6. La forma de seleccionar un color es introduciendo el número correspondiente (sin esperar *intro*, para hacerlo más ágil).

Para representar el tablero utilizaremos un array bidimensional que detallaremos más adelante. La parte más compleja de este programa es el **algoritmo de propagación** del color desde la casilla de referencia hacia las casillas adyacentes (este problema ya se planteó en el ejercicio 4 de la sesión 9 de laboratorio). Para facilitar la explicación, en lo que sigue llamaremos *old* al color de partida de la casilla de referencia y *new* al nuevo color seleccionado por el jugador. Dada una casilla cualquiera del tablero con coordenadas (i, j) la idea general del algoritmo es la siguiente:

- Si el color de la casilla (i, j) es *old* entonces:
  - Se pone el color de la casilla a *new*.
  - Se aplica el mismo algoritmo a las casillas adyacentes: $(i-1, j), (i+1, j), (i, j-1), (i, j+1)$ que formen parte del tablero. Por ejemplo, para la casilla $(0, 0)$ serán solo $(1, 0)$ y $(0, 1)$.
- Si el color de la casilla (i, j) es otro, no hay nada que hacer.

Este algoritmo necesita un poco más de elaboración por dos razones. Primero, porque incurriría en un ciclo infinito: la exploración de la casilla $(0, 0)$ desencadena la exploración su adyacente $(0, 1)$, que a su vez desencadena nuevamente la exploración de la $(0, 0)$. Para evitar esto el algoritmo necesita llevar cuenta del **conjunto de casillas visitadas** (para no reiterar el proceso sobre ellas).

Por otro lado, la exploración de una casilla desencadena en general la exploración de *varias* adyacentes, que a su vez desencadenarán otras exploraciones. Es decir, en general el algoritmo debe mantener un **conjunto de casillas pendientes de procesar**, que son casillas ya visitadas, pero aun no procesadas. En resumen, las casillas se dividen en *visitadas* y *no visitadas*; a su vez, dentro de las visitadas hay *procesadas* y *pendientes de procesar*. Gráficamente:

Para entender mejor el papel de estos conjuntos en el algoritmo vamos a describir en detalle la ejecución del mismo. Supongamos el siguiente tablero (donde a, b y c representan colores):

	0	1	2	...
0	a	a	c	...
1	b	a	b	...
2	a	a	...	...
...	...	...	...	

Utilizaremos un conjunto *procesadas* para almacenar las casillas procesadas y otro *pendientes* de casillas pendientes de procesar. Supongamos ahora que cambiamos el color de la casilla $(0,0)$ a c, i.e., *old=a*, *new=c*. El algoritmo arranca con los conjuntos *procesadas={}* y *pendientes={(0,0)}* y da los siguientes pasos:

- Procesamos la primera de las casillas pendientes, la $(0,0)$. Se cambia su color a c y tenemos *procesadas={(0,0)}* y *pendientes={(0,1),(1,0)}*.
- Procesamos la siguiente pendiente, la $(0,1)$, de color a. Como el color de la casilla es *old*, se cambia a c (*new*) y se añade a las procesadas (*procesadas={(0,0),(0,1)}*). Se elimina del conjunto de pendientes y se añaden las adyacentes aun no visitadas: en este caso, las adyacentes son $(0,0),(0,2),(1,1)$, pero la $(0,0)$ ya está visitada (porque aparece en *procesadas*), luego *pendientes={(1,0),(0,2),(1,1)}*.
- La siguiente pendiente, la $(1,0)$, no cambia nada y se elimina del conjunto de pendientes, obteniendo *pendientes={(0,2),(1,1)}* y *procesadas={(0,0),(0,1),(1,0)}*.
- La siguiente, la $(0,2)$ se deja intacta porque no es de color *old* y obtenemos *pendientes={(1,1)}* y *procesadas={(0,0),(0,1),(1,0),(0,2)}*.
- ... el algoritmo continúa hasta que *pendientes={}*.

Se puede hacer una implementación simple y elegante de los conjuntos *procesadas* y *pendientes* **utilizando un solo array `posVis` de posiciones visitadas**, junto con dos índices `pend` y `fin`. Las casillas procesadas se almacenan en el rango [0,`pend-1`], y las pendientes en el rango [`pend`,`fin`-1]. La ventaja de esta representación es que pasar una casilla de las pendientes a las procesadas supone únicamente incrementar el índice `pend` y añadir las adyacentes en el array `posVis` a partir de la posición `fin`. En el ejemplo anterior, tras procesar la casilla $(0,0)$ la situación sería:

	0	1	2	3
`posVis`	(0,0)	(0,1)	(1,0)	...
		↑		↑
		`pend`		`fin`

Ahora, la primera pendiente es la casilla $(0,1)$ y tras procesarla tendremos:

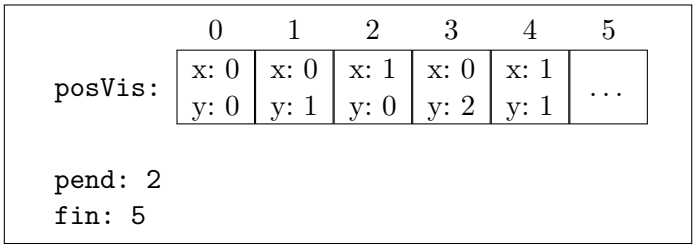
	0	1	2	3	4	5
`posVis`	(0,0)	(0,1)	(1,0)	(0,2)	(1,1)	...
			↑			↑
			`pend`			`fin`

2. Programación del juego

En nuestro programa declararemos los siguientes elementos en la clase (fuera de los métodos):

- Dos constantes `ANCHO` y `ALTO` que definen las dimensiones del tablero.
- Un tipo enumerado `Color` con los valores `rojo, amarillo,...` para representar los 6 colores del juego.
- Un tipo estructurado `Posicion` con dos campos `x` e `y` para representar posiciones en el tablero.
- Otro tipo estructurado `Visitados` para mantener la información de posiciones visitadas, con tres campos:
  - un campo `posVis` que será un array de posiciones
  - dos campos enteros `pend` y `fin` con los índices explicados arriba.

Así pues, una variable `vis` de este tipo tendrá la siguiente estructura:



El tablero se declarará y creará en el método `Main` como un array bidimensional de tipo `Color` y de tamaño `ANCHO` x `ALTO`. Los métodos a programar son los siguientes:

- `void genera (tab)`: inicializa el tablero `tab` con colores aleatorios (no escribe nada en pantalla, solo rellena el contenido de la matriz).
- `void dibuja(tab, cont)`: dibuja en pantalla el tablero `tab`, el panel de selección y el contador de jugadas, tal como se ha detallado en la segunda figura. Para que las casillas tengan aspecto cuadrado habrá que escribir 2 blancos del color correspondiente.
- `bool estaVisitado(vis, pos)`: determina si la posición `pos` está en la estrutura de posiciones visitadas `vis`.
- `void añadeVecinos (vis, pos)`: añade a la estructura `vis` las posiciones adyacentes a la posición `pos` que aun no han sido visitadas, utilizando el método anterior.
- `void expande(tab, new)`: dado el tablero `tab` y el color new, da un paso de juego, i.e., aplica el algoritmo de propagación de color a partir de la casilla de referencia $(0, 0)$. En este método se declara el array `visitados` con sus índices `pend` y `fin` explicados antes. Se inicializan los pendientes con la casilla de referencia y se hace el bucle de propagación hasta agotar las posiciones pendientes de procesar.
- `bool finalPartida(tab)`: determina si todas las casillas del tablero `tab` son del mismo color (fin del juego).
- `Color leeColor()`: lee de teclado un dígito del 1 al 6 y devuelve el color correspondiente. La conversión se puede hacer con `c = ((Console.ReadKey (true)).KeyChar - '0' - 1);` (cómo funciona esta instrucción?)
- `void Main ()`: presenta el tablero y solicita colores hasta que termina el juego.

Nota: Los únicos métodos de interacción con el usuario son `dibuja` y `leeColor` (aparte de `Main` que escribirá el número de jugadas que ha hecho el usuario para terminar el juego).