Como desarrollar un Buscaminas en Pharo Smalltalk POO

Paradigmas de Programación (Universidad Tecnológica Nacional)

Paradigmas de la Programación: Juego Buscaminas

[a]Blanco Liotti Ignacio, [b]Vilalta Tomás Federico, [c]Yaya Franco Gabriel
[a]U.T.N F.R.Re, ISI B, ignacioliotti@gmail.com
[b]U.T.N F.R.Re, ISI B, tomasfedericovilalta@gmail.com
[c]U.T.N F.R.Re, ISI A, franc095cp@gmail.com

1. Resumen

Durante este trabajo exploraremos el desarrollo del juego "Buscaminas" en el lenguaje de programación SmallTalk en Pharo 8.0. Para esto clases tales como SimpleSwitchMorph, Array2D, BorderedMorph entre varios otros. A grandes rasgos el juego es un display de dos arreglos de SimpleSwitchMorph, uno de celdas y uno de minas, ambos superpuestos, con las celdas siendo las que están en la capa delantera, y al activarlas, estas se volverán transparentes, permitiendo la visión de las minas.

Además se agregó una nueva característica al juego original: se pueden ganar puntos en el juego al hacer click en un objetivo móvil, desafiando a los jugadores a realizar dos tareas en simultáneo para intentar conseguir la mayor cantidad de puntos, en el menor tiempo posible.

2. Introducción

El *"Buscaminas"* es un juego que cuyo objetivo es descubrir la ubicación de todas las minas del tablero sin activarlas. El tablero contiene de manera aleatoria, una cantidad determinada de minas que el jugador debe desactivar. Si este llegara a tocar una mina, habrá perdido y el juego terminará. El jugador tiene la habilidad de marcar las casillas que crea contiene una mina con una bandera, y si consigue marcar correctamente todas las ubicaciones del tablero, ganará la partida. Las casillas pueden indicar con números que hay una cierta cantidad de minas junto a ella, variando desde *1 hasta 8*. Al hacer clic en una ubicación vacía, si no hay ninguna mina adyacente a ésta, todas las casillas vacías adyacentes también se desbloquearán, liberando una parte del tablero; en cambio, si hay una mina, la casilla seleccionada será marcada con el número de minas adyacentes que tenga.
A modo de ejemplo, si se revela una casilla y esta muestra el *número 1*, significa que en una de las 8 casillas adyacentes, existe una única bomba

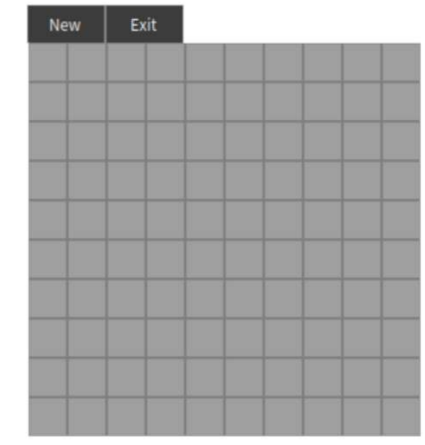

Fig. 1 Juego ejecutado

3. Desarrollo

El juego consta de tres clases principales propias del mismo, las cuales son, *"Buscaminas", "Bomba"* y *"Celda"*. Tanto *Celda* como *Bomba* son clases que se refieren a cada casilla individual del tablero, y sus formatos son muy similares.

```
SimpleSwitchMorph subclass: #Bomba
instanceVariableNames: 'cellsPerSide'
        classVariableNames: ''
        package: 'Buscaminas'
```

```
SimpleSwitchMorph subclass: #Celda
instanceVariableNames:'mouseAction'mouseAction2'
'bandera'
classVariableNames: ''
package: 'Buscaminas'
```

Ambas clases son subclases de la clase SimpleSwitchMorph y la única diferencia es que *Celda* posee variables de instancia llamadas *mouseAction, mouseAction2* y *bandera.*

Sus inicializaciones son muy similares también:

```
1 Celda>>initialize
2 super initialize.
3 self label: ''.
4 self borderWidth: 1.
5 bounds := 0 @ 0 corner: 30 @ 30.
6 offColor := Color paleGreen .
8 onColor := Color transparent.
9 self useSquareCorners.
10 bandera:= false.
```

```
1 Bomba>>initialize
2 super initialize.
3 self label: ''.
4 self borderWidth: 1.
5 bounds := 0 @ 0 corner: 30 @ 30 .
6 offColor:= Color red.
7 onColor:= Color lightGray lighter .
8 self turnOn.
9 (10 atRandom)>1
10 ifFalse:[ self turnOff ].
11 self useSquareCorners.
```

Podemos ver como la inicialización de ambas es muy similar, ambos son cuadrados de 30x30 unidades, sin ningún tipo de texto con sus respectivos colores de prendido y apagado. La gran diferencia que poseen ambos radica en las últimas líneas de ambos métodos. Para *bomba* esto radica de la lina 8 a la 10, en la que determina que todas las celdas "bomba" empezaran en su estado encendido, pero cada una de ellas, de forma aleatoria determinara si esta debe apagarse o no, utilizando el método *atRandom*, aleatoriamente en la instanciación de cada bomba tienen un 10% de probabilidades de apagarse, y este estado de apagado es el que nosotros consideraremos para el resto del juego como una *"Mina"*. Algo también importante que los diferencia que será tratado más adelante es el método de inicialización de Celda la variable booleana *"bandera"* que será la que nos permitirá determinar si el usuario marcó una celda como potencial mina o no.

Luego se encuentra la tercer clase para la ejecución del juego: *"Buscaminas"*.
Esta clase está definida como:

```
BorderedMorph subclass: #Buscaminas
        instanceVariableNames: 'cells bombs c cellsPerSide'
        classVariableNames: ''
        package: 'Buscaminas'
```

Buscaminas es la clase instanciada para ejecutar el "Buscaminas" como tal, la misma se inicializa con el siguiente método:

```
1  Buscaminas>>initialize
2  | sampleCell width height |
3  super initialize.
4  cellsPerSide:=UIManager default
5    chooseFrom: #('Chico' 'Medio' 'Grande')
6    values: #(10 15 20)
7    lines: #( 1 2 3) message: 'Elija el Tamaño' title: 'Buscaminas'.
8  sampleCell := Celda new.
9  width := sampleCell width.
10  height := sampleCell height.
11  self bounds: (100 @ 100 extent: (width * cellsPerSide ) @ (height * cellsPerSide ) + (2 * self
12  borderWidth)).
13  self borderColor: Color transparent.
```

```
14  self color: Color transparent.
15  bombs := Array2D  new: cellsPerSide  tabulate: [ :i :j | self newBombAt: i at: j ].
16  cells := Array2D  new: cellsPerSide  tabulate: [ :i :j | self newCellAt: i at: j ].
17  self controles.
18 finalScore:= 0.
19 gameState:= true.
```

Este método *initialize* es el punto de partida del juego. Podemos observar como en las líneas 4 a 7 pregunta al usuario el tamaño del tablero al que quiera jugar, y le adjudica, dependiendo la elección del usuario, los valores 10, 15 o 20 a la variable *"cellsPerSide"* respectivamente. Esta variable es, como su nombre lo indica, la cantidad de filas y columnas que tendrá el tablero. Esta variable es luego utilizada en las líneas 11 y 12 para determinar el tamaño del tablero en general, y en las líneas 15 y 16 para la instanciacion de un Array2D con esa cantidad de filas y columnas.
Otra parte crucial de este código son justamente esas dos líneas, la 15 y 16, en las mismas instancian dos Array2D, uno llamado *bombs* y otro *cells*, los cuales son justamente tableros en los cuales se hará un display de las celdas con las cuales el usuario jugará. Por último llama también al método controles, los cuales generarán los botones los cuales nos permitirán salir del juego o crear un nuevo tablero

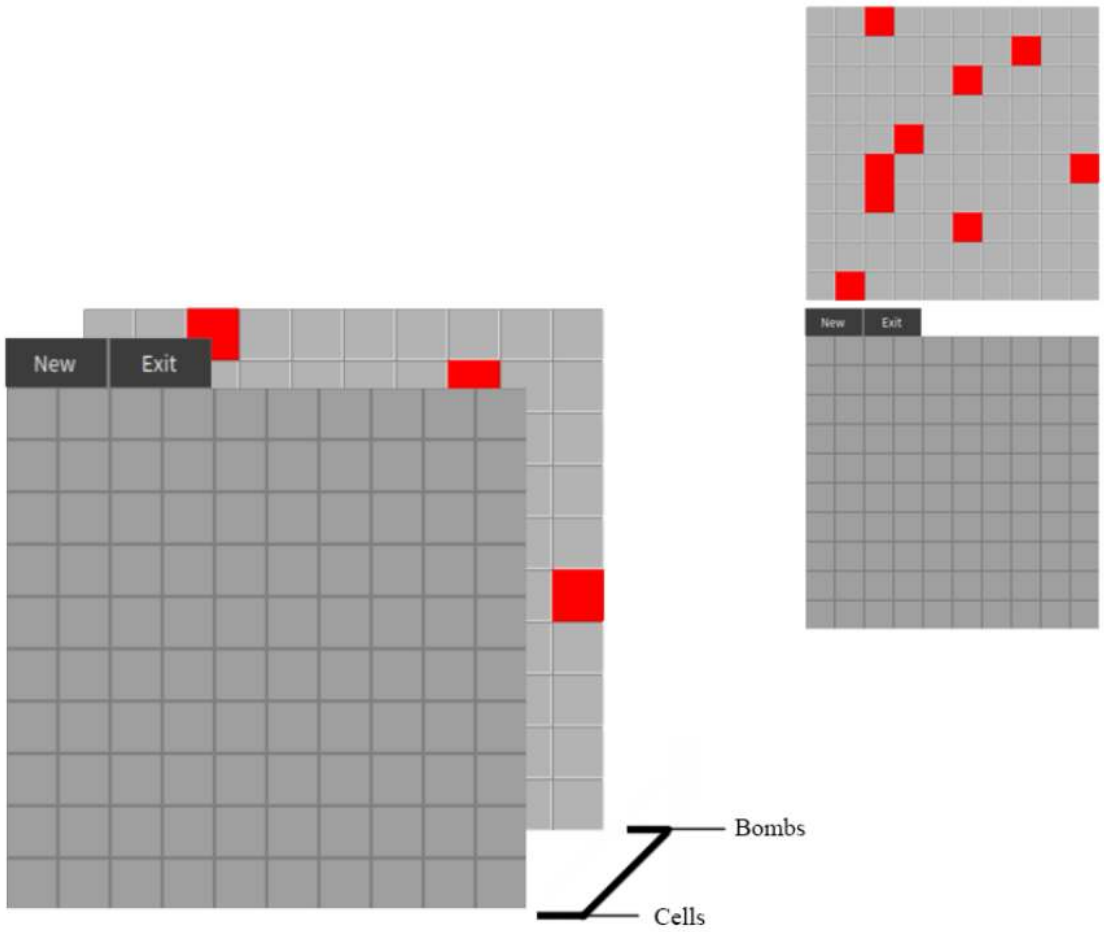

Fig. 2 Capas que conforman el tablero del juego (arreglo de celdas superpuesto al de bombas)

Siguiendo con el flujo del código, al instanciar el Array2D ejecuta en cada elemento del mismo el siguiente método:

```
Buscaminas>>newBombAt: i at: j
| origin |
```

```
c := Bomba new.
origin := self innerBounds origin.
self addMorph: c.
c position: ((i - 1) * c width) @ ((j - 1) * c height) + origin.
^ c
```

Dicho de manera simple, lo que hace es, usando una coordenada dada del arreglo (i, j), instancia en esta la clase *Bomba* generando así las casillas observadas con las posibles minas. y lo mismo su método análogo "newCellAt: at: "

```
1Buscaminas>> newCellAt: i at: j
2 | origin |
3 c := Celda new.
4 origin := self innerBounds origin.
5 self addMorph: c.
6 c position: ((i - 1) * c width) @ ((j - 1) * c height) + origin.
7 c mouseAction: [self checkNeighbour: i at: j ].
8 c mouseAction2: [ (cells at:i at:j) bandera
9                         ifTrue:[(cells at:i at:j) color: Color lightGray.
10                                (cells at:i at:j) bandera: false]
11                        ifFalse:[(cells at:i at:j) color: Color lightGreen darker.
12                                 (cells at:i at:j) bandera:true].
13                        self checkWinCondition:i at:j.].
14 ^c
```

Podemos ver una clara diferencia entre ambos métodos y es que *newCellAt:at:* tiene dos puntos fundamentales, en las líneas 8 y 9 ejecutan los métodos *mouseAction* y *mouseAction2*; para no extendernos en partes de código que no profundizan en la lógica detrás del mismo, simplemente entenderemos estos dos métodos como los que se accionan cuando se realiza el click izquierdo (mouseAction) y el derecho (mouseAction2). Al realizar el click derecho el usuario está intentando marcar que el cree que en esa celda hay una mina, por lo que debemos cambiar el valor de "bandera" de esa celda en especifico a "true", y luego verificar si es que con esta selección, el usuario logró seleccionar todas las minas y por consiguiente ganar el juego, por lo que es llamado al final del bloque el método *checkWinCondition:at:* que justamente ejecuta esta tarea. Previo a esto son chequeadas dos condiciones: ¿La bandera está levantada o no? en el caso de que sí, significa que el usuario ya marcó esta celda y se arrepintió de haberla marcado, por lo que debemos volver a marcarla de gris como estaba previamente, en caso de no estar marcada, significa que el usuario desea marcar como una casilla con una posible mina, por la que le atribuimos el color verde, luego de cada una devolvemos el valor de la bandera correspondiente, esta parte del código es exclusivamente pensada para que el usuario pueda marcar y desmarcar libremente estas casillas.

El método checkNeighbour es uno de los métodos más importantes de todo el juego, tiene tres partes importantes, la primera es mostrar la casilla seleccionada, si esta tiene una mina, indicar que se perdió el juego, en caso contrario avanza a su segunda parte, donde verifica la cantidad de minas alrededor de nuestra casilla seleccionada, si hay mínimo una mina, la casilla deberá pasar a revelarse a sí misma y mostrar el número de minas. En el caso que no haya una mina alrededor de la casilla, el método deberá realizarse nuevamente en estas casillas adyacentes.

```
1 Buscaminas>>checkNeighbour: i at: j
2 | conta ady|
3 conta:=0.
4                                                    "En el caso de haber una bomba se pierde el juego"
5 (bombs at:i at:j) isOn
6 ifFalse: [(cells at: i at: j) toggleState.
7              self loseScreen.]
```

```
8 ifTrue:[
9                                                   "Si la casilla seleccionada no es una bomba entonces contara cuantas
minas existen adyacentes a la casilla"
10 ady:=(self contarDe: i at: j).
11 (ady)>0
12 ifTrue:[
13          (cells at:i at:j) label:( '    ' , (ady)asString , '   ').
14          (cells at:i at:j) color: (ady=1
15          ifTrue:[ Color lightGray darker darker]
16          ifFalse:[
17          ady=2
18          ifTrue:[Color lightOrange]
19          ifFalse:[
20          ady=3
21          ifTrue: [ Color orange]
22          ifFalse:[
23          ady=4
24          ifTrue: [ Color magenta]
25          ifFalse: [
26          ady=5
27          ifTrue: [ Color purple lighter lighter lighter lighter]
28          ifFalse: [
29          ady=6
30          ifTrue: [ Color purple lighter lighter]
31          ifFalse:[
32          ady=7
33          ifTrue:[Color purple darker]
34          ifFalse:[Color red]]]]]]).]

35ifFalse:[(self laRecursionDe: i at: j)]].
```

El método inicia preguntando, en la posición del arreglo dada, es decir, donde el usuario hizo click, ¿Existe una bomba en estado apagado? es decir, ¿Hay una mina? De ser así simplemente llama al método *"loseScreen"* que muestra por pantalla el mensaje de que terminó el juego, caso contrario llama al método contarDe:at: que lo que hace es justamente contar cuántas minas hay en la proximidad, este método será explicado posteriormente. Luego realizado el conteo de minas, tenemos que preguntar, ¿Hay por lo menos una mina alrededor de la casilla? de ser asi simplemente se cambia el *label* de la casilla a la cantidad de minas adyacentes y se la colorea acorde a la misma. En el caso de que no haya ninguna mina alrededor de la casilla entonces se llama al método *laRecursionDe:at:* el cual, como su nombre que deja poco a la imaginación indica, es una recursión que ejecuta lo siguiente

```
1 Buscaminas>>laRecursionDe: i at: j
2 -1 to: 1 do:[:x|
3          -1 to: 1do:[:y|
4                    (1<=(i+x))&(1<=(j+y))&((i+x)<=(cellsPerSide))&((j+y)<=(cellsPerSide))
5                    ifTrue:[(cells at: i+x at: j+y)isOn
6                                        ifFalse:[
7                                                  (cells at: i+x at: j+y) toggleState.
8                                                  self checkNeighbour: (i+x) at: (j+y)].]]]
```

Este método realiza una simple tarea, que es sumar 1 o sustraer 1 del índice introducido (i, j) y analizar esas casillas, si alguna celda esta prendida, es decir ya fue seleccionada y transparentada por el usuario, no debería

hacerse nada, no queremos que alguna casilla ya descubierta se vuelva a esconder. En el caso de estas celdas estar escondidas entonces debería descubrirlas y llamar al método checkNeighbour en estas casillas adyacentes, generando así una recursión de métodos que es la que nos permite crear la situación tan conocida en la que el usuario toca una casilla y una gran parte del tablero es revelada tal como lo muestra la *Figura 5*

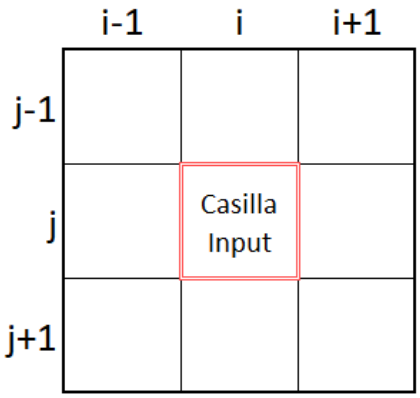

Fig. 3 Cómo navega el código a través de las casillas dependiendo de la casilla seleccionada.

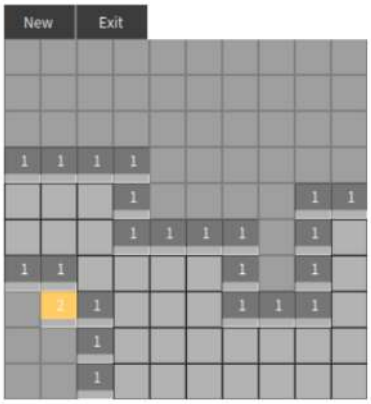

Fig. 4 Resultado de la correcta implementación de la recursión empleada para liberar grandes porciones del mapa

Una parte crucial de este código es la línea 4 del mismo, en la misma se realiza una sentencia booleana muy importante, básicamente lo que dice es que este nuevo índice adyacente a la celda que el usuario seleccionó, no puede ser menor a 1 ni mayor a cellsPerSide (cantidad máxima de celdas en el tablero), es decir no podemos pedirle a la instanciación de Array2D que busque una casilla de índice 0 ni índice mayor al máximo de casillas, ya que esto nos daría error, esta salvaguarda es la que nos permite que el usuario seleccione casillas en los bordes sin que el programa nos indique que hemos buscado una casilla inexistente.
Este método es muy similar al previamente mencionado *contarDe:at* por lo que podemos ver

```
1 Buscaminas>>contarDe: i at: j
2       |conta|
3       conta:=0.
4  -1 to: 1 do:[:x|
5  -1 to: 1 do:[:y|
6               (1<=(i+x))&(1<=(j+y))&((i+x)<=(cellsPerSide))&((j+y)<=(cellsPerSide))
7               ifTrue:[(bombs at: i+x at: j+y)isOn
8                       ifFalse:[conta:=conta+1]]]].
9       ^conta.
```

Podemos observar como tienen el mismo funcionamiento, sumar o sustraer 1 en el índice del arreglo, chequear la existencia de una bomba con estado apagado, es decir con una mina presente y simplemente contarlos y devolver el valor del contador una vez terminado el método.

Variación original: *Target Practice*

Como modificación del juego original, se discutió que sería más interesante hacer el juego más difícil, por lo que se decidió que añadiremos un objetivo móvil como si se tratara de una práctica de tiro.
La diana acumularía puntos cada vez que fuera clickeada. El desafío sería conseguir la mayor cantidad de puntos posibles pero sin distraerse de la tarea de ganar el buscaminas en el menor tiempo posible. Además se requiere de velocidad y precisión, puesto que el objetivo se mueve impredeciblemente y fallar puede ocasionar la activación de una celda con bomba.

La clase TargetMorph

Para esta implementación se creó una nueva clase *'TargetMorph'* como subclase de *'ImageMorph'* .

```
ImageMorph subclass: #TargetMorph
        instanceVariableNames: 'sprite score game xyBound'
        classVariableNames: ''
        package: 'Buscaminas'
```

'score' contendrá el puntaje actualizado del jugador.
'game' es una instancia de la clase 'Buscaminas'
'xyBound' recibirá el tamaño del arreglo de Buscaminas para limitar los saltos de la diana
'sprite' es el modelo gráfico de la diana.

Se definieron los siguientes métodos
TargetMorph >> initialize
handlesMouseDown:
mouseDown:
score
step
stepTime

TargetMorph>>initialize

```
initialize
        | gameBounds |
        super initialize.
        sprite := Form
                fromFileNamed: 'targetSprite.png'.
        self form: sprite. "Asigna el sprite al Morph"
        self position: 240 @ 240.
        score := 0.
        game := Buscaminas new openInWorld. "Crea una instancia de Buscaminas"
        gameBounds := game cellsPerSide.
        gameBounds < 15 "Checkea el tamaño de buscaminas creado"
                ifTrue: [ xyBound := 380 ]
                ifFalse: [ xyBound := gameBounds < 20
                                ifTrue: [ 525 ]
                                ifFalse: [ 690 ] ]
```

Para el sprite se creó un pequeño gráfico(Fig.5) en el programa *Aseprite* y luego se lo asigna al morph. Vale notar que targetSprite.png tiene que encontrarse dentro del directorio donde se encuentra la imagen de Pharo que se esté usando. Por ejemplo: 'C:\Users\Usuario\Documents\Pharo\images\Pharo 8.0 - 64bit (stable)' donde *Pharo 8.0 - 64bit (stable)* es la carpeta que contiene la imagen.

Fig. 5 targetSprite.png

Por último se crea una instancia de Buscaminas, con la que el objeto interactúa para determinar los puntajes, límites de juego, etc.

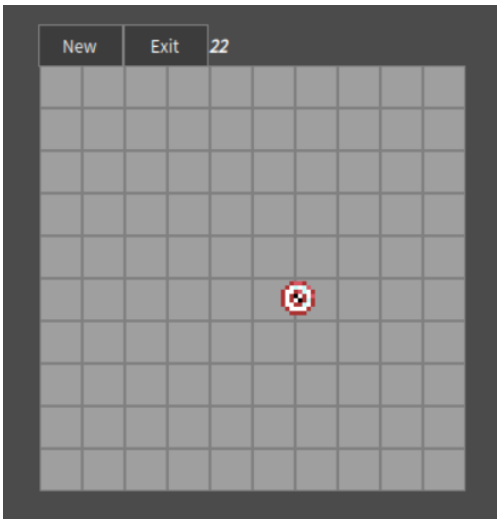

Fig.6. Juego iniciado con Target Practice

TargetMorph>> handlesMouseDown y TargetMorph>>mouseDown

```
handlesMouseDown: anEvent
 ^ true
***
mouseDown: anEvent
        anEvent redButtonPressed
                ifTrue:
                        [ self position: (100 to: xyBound) atRandom @ (100 to: xyBound) atRandom.
                        score:= score+1 .
                         game finalScore: score]fork.
        self changed
```

'*handlesMouseDown:*' simplemente habilita al objeto a reaccionar a clicks de mouse.
'mouseDown' se encarga de actuar en respuesta a un evento del mouse, moviendo a la diana a un lugar aleatorio dentro de los límites del juego, al mismo tiempo que almacena cada click como un punto en *score* y lo envía al Buscaminas para que lo muestre al finalizar el juego (Fig.7) .

Fig.7. Juego finalizado con display de puntaje

Animación de la diana

Para que la diana se mueva por la pantalla, recurrimos al sistema de animación que provee Moprhic, utilizando dos métodos: *step* y *stepTime*.

```
stepTime
^ 900
***
step
self position: (100 to: xyBound) atRandom @(100 to: xyBound) atRandom.
game gameState ifFalse: [ self delete].
self changed.
```

'step' es enviado al objeto cada intervalo de tiempo establecido por *stepTime.* Se encarga de mover la diana al hacerle click para evitar el abuso de conseguir múltiples puntos en los momentos estacionarios del objetivo. Además pergunta por el estado del juego y elimina el objeo si el juego terminó
'stepTime' simplemtente establece el tiempo en milisegundos.

Conclusión

El Buscaminas es realmente un juego simple, con escasas reglas y muy sencillas, pero a su vez estas representaron un desafío logístico a la hora de implementarlas. Como equipo enfrentamos varias dificultades no solamente con el desarrollo del código encargado de la logica detras del juego perse sino también su implementación gráfica a través de los *Morphs* en pharo, lo que nos llevó múltiples veces a reimaginar una solución ya ideada porque esta no se implementa del todo correctamente con la interfaz gráfica que podíamos generar. Uno de estos problemas fue el tener que mentalizar todo el juego como dos arreglos superpuestos entre sí, pero una vez superado este problema, en realidad este planteo del juego como tal nos trajo grandes beneficios, como por ejemplo poder realizar una gran cantidad de métodos en una misma casilla con tan solo dos datos, los índices en *"i"* y *"j"*, y no solamente dentro del mismo arreglo *Cells* o *Bombs* sino también entre ellos ya que al estar superpuestos la misma coordenada de una casilla que se verifique en *Cells* a su vez serán coordenadas análogas para la casilla que quiera tratar en Bombs. Lo cual a nuestro razonamiento y la implementación del código nos resultó de gran ayuda.

Por último, la implementación original al juego (Target Practice) fue un gran ejercicio sobre programación en bloques, puesto que debimos la característica por sí misma para luego integrarla al juego original.

Bibliografia.

-Andrew P. Black, Damien Pollet, Stéphane Ducasse, Oscar Nierstrasz (2010) Pharo by Example 5.0

-Stéphane Ducasse with Damien Pollet April 1, 2018 Learning Object-Oriented Programming, Design and TDD with Pharo