PABLO DÍAZ-HEREDERO GARCÍA - [100405973]

ÍNDICE

INTRODUCCIÓN

La práctica realizada tiene como fin la recreación del videojuego ***Galaga*** en Java a partir de la librería ***GameBoardGUI*** facilitada por el ***PLG*** *(Planning and Learning Group)* de la **Universidad Carlos III de Madrid.**
La siguiente memoria recopila los aspectos más relevantes del proyecto realizado donde se hace especial hincapié en el diseño de clases, la funcionalidad implementada y los diferentes métodos utilizados para su desarrollo. En última instancia se aporta una valoración sobre la práctica.

Figura 1. **-** *Imagen de inicio del juego*

DIAGRAMA DE CLASES

El siguiente diagrama resume el contenido de cada clase (incluyendo métodos y atributos) y la relación entre las mismas en el proyecto.

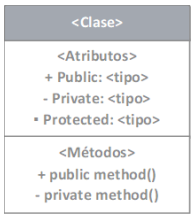


Figura 2. *– Leyenda de bloques*

Enemy

- **board**: GameBoardGUI
- **id:** int
- **type:** char
- **health:** int
- ***POINTS*:** int
- **x:** int
- **y:** int
- **formationX:** int
- **formationY**: int
- **direction:** int
- **entryPhase:** int
- **entryCircle:** int
- **sprite:** String
- **animationCounter:** int
- **InFormation:** boolean
- **counterPath:** int
- **indexPath :** int
- **randomPath**: int
- **probabilityCounter**: int
- **outBoard:** boolean
- **deadAnimation:** boolean
- **freezed:** boolean
- **shotCount:** int
- **shot[]:** Torpedo[]

+ decreaseHealth()
+ getSprite()
+ move()
+ startShot()
+ followPath()
+ resetPath()
+ upProbCounter()
+ setX()
+ getX()
+ setY()
+ getY()
+ setFormationX()
+ getFormationX()
+ setFormationY()
+ getFormationY()
+ getID()
+ setDirection()
+ getDirection()
+ setSprite()
+ setDeadAnimation
+ isDeadAnimation()
+ setAnimationCounter()
+ getAnimationCounter()
+ getType()
+ setFreezed()
+ isFreezed()
+ setInFormation()
+ isInFormation()
+ setOutPath()
+ isOutPath()
+ setRandomPath()
+ getRandomPath()
+ setProbabilityCounter()
+ getProbabilityCounter()
+ setEntryPhase
+ getEntryPhase
+ setEntryCircle
+ getEntryCircle
+ setHealth()
+ getHealth()
+ setShotCount()
+ getShotCount()

EnemyZako

Constructor de **EnemyZako** hace llamada a **constructor de la clase superior (Enemy)** otorgando los valores a los atributos en la clase super.

Valores propios de esta clase que son introducidos en super independientemente de lo establecido en el constructor de EnemyZako:
- **type**: 'Z'
+ ***POINTS***: *Constants.ZAKOPOINTS*

EnemyGoei

Constructor de **EnemyGoei** hace llamada a **constructor de la clase superior (Enemy)** otorgando los valores a los atributos en la clase super.

Valores propios de esta clase que son introducidos en super independientemente de lo establecido en el constructor de EnemyGoei:
- **type**: 'G'
+ ***POINTS***: *Constants.GOEIPOINTS*

EnemyCommander

Constructor de **EnemyCommander** hace llamada a **constructor de la clase superior (Enemy)** otorgando los valores a los atributos en la clase super.

Valores propios de esta clase que son introducidos en super independientemente de lo establecido en el constructor de EnemyCommander:
- **type**: 'C'
- ***POINTS***: *Constants.COMMANDERPOINTS*

EnemyEnterprise

Constructor de **EnemyEnterprise** hace llamada a **constructor de la clase superior (Enemy)** otorgando los valores a los atributos en la clase super.

Valores propios de esta clase que son introducidos en super independientemente de lo establecido en el constructor de EnemyEnterprise:
- type: 'E'
+ POINTS: Constants.ENTERPRISEPOINTS

Constants

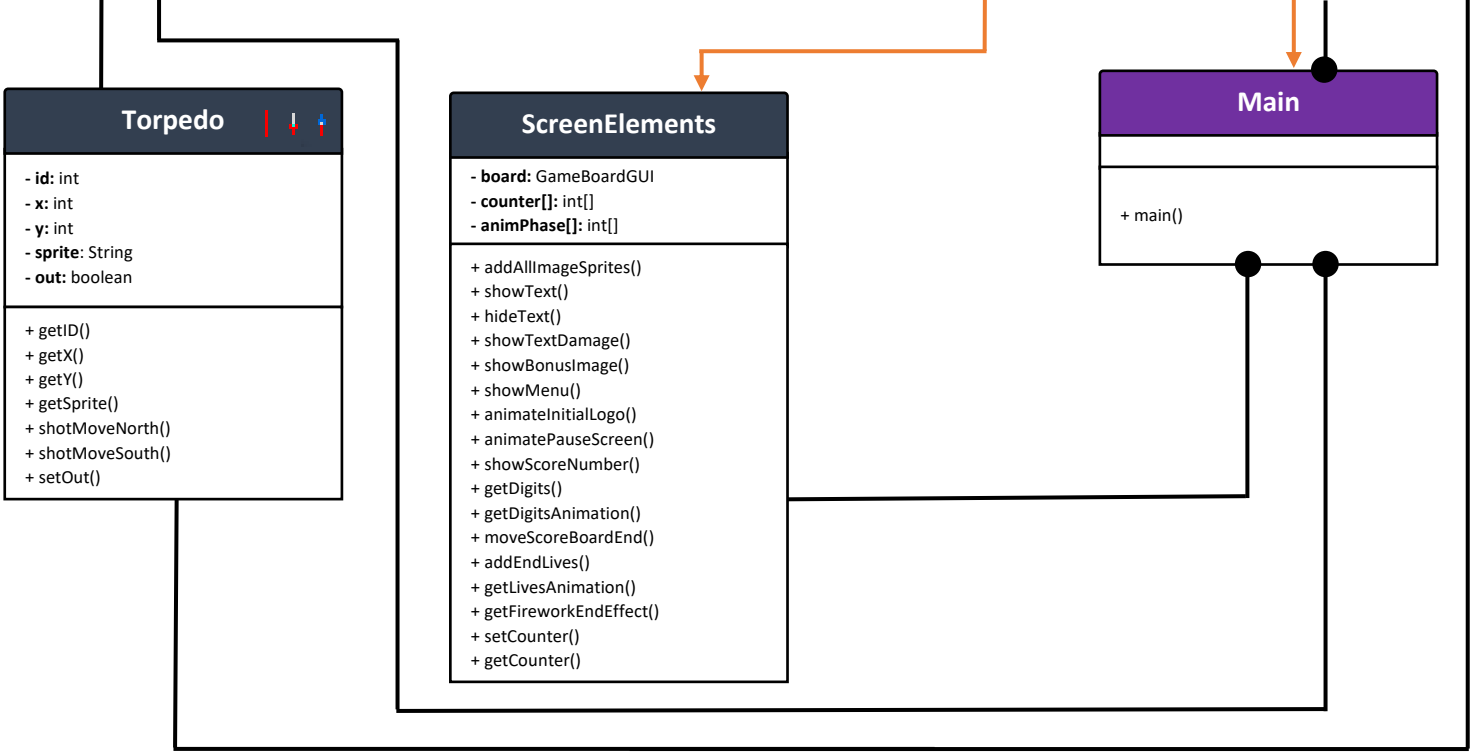
+ ***WIDTH*:** int
+ ***HEIGHT*:** int
+ ***HITBOXSIZE*:** int
+ ***MAXTORPEDOS*:** int
+ ***MAXHEALTH*:** int
+ ***ZAKOPOINTS*:** int
+ ***GOEIPOINTS*:** int
+ ***COMMANDERPOINTS*:** int
+ ***ENTERPRISEPOINTS*:** int
+ ***MEDALPOINTS*[]:** int[]
+ ***PLAYERNAME*:** String
+ ***BOARDTITLE*:** String
+ ***WELCOMEPHRASE*:** String
+ ***PORTRAIT*:** String
+ ***LANGUAGE*:** String
+ ***PROBJUMPLV*[]:** double[]
+ ***PROBSHOTLV*[]:** double[]
+ ***LV1FORMATION[][]*:** char[][]
+ ***LV2FORMATION[][]*:** char[][]
+ ***LV3FORMATION[][]*:** char[][]
+ ***LV4FORMATION[][]*:** char[][]

+ ***DIR_N*:** int
+ ***DIR_NNE*:** int
+ ***DIR_NE*:** int
+ ***DIR_ENE*:** int
+ ***DIR_ESE*:** int
+ ***DIR_SE*:** int
+ ***DIR_SSE*:** int
+ ***DIR_S*:** int
+ ***DIR_SSW*:** int
+ ***DIR_SW*:** int
+ ***DIR_WSW*:** int
+ ***DIR_W*:** int
+ ***DIR_WNW*:** int
+ ***DIR_NW*:** int
+ ***DIR_NNW*:** int
+ ***MOVES[][]*:** int[][]
+ ***PATHZIGZAGRIGHT[][]*: int[][]**
+ **PATHZIGZAGLEFT[][]: int[][]**
+ **PATHZIGZAGLEFTBIG[][]: int[][]**
+ ***PATHZIGZAGRIGHTBIG[][]*: int[][]**
+ **PATHCIRCLEANTICLOCKWISE[][]: int[][]**
+ **PATHCIRCLECLOCKWISE[][]: int[][]**
+ **PATHCIRCLE[][]: int[][]**
+ **PATHENTERPRISELEFT[][]: int[][]**
+ **PATHENTERPRISERIGHT[][]: int[][]**

Player

- **id:** int
- **health:** int
- **score:** int
- **invencible:** boolean
- **x:** int
- **y:** int
- **direction:** int
- **sprite:** String
- **deadAnimation:** boolean
- **animationCounter:** int
- **enemyHits:** int
- **shotCount:** int
- **shot[]:** Torpedo[]

+ upScore()
+ animateDamage()
+ move()
+ resetAttributes()
+ setSprite()
+ getSprite()
+ setID()
+ getX()
+ getY()
+ getDirection()
+ setHealth()
+ getHealth()
+ setScore()
+ getScore()
+ setInvencible()
+ isInvencible()
+ setDeadAnimation()
+ isDeadAnimation()
+ setAnimationCounter()
+ getAnimationCounter
+ setEnemyHit()
+ getEnemyHit()
+ setShotCount()
+ getShotCount()
+ setSpeed()
+ getSpeed



Los distintos métodos de las clases se encuentran detallados y explicados en los comentarios del código y en la funcionalidad. La clase ***Constants*** mantiene una férrea relación con el resto de clases del juego, ya que en ella se han especificado las Constantes, las cuales almacenan datos estáticos de gran importancia para el juego.

- Con el fin de permitir la configuración del juego, se han unificado en la clase ***Constants*** aquellos parámetros cuya modificación puede ser de interés (puntuación que otorgan los enemigos al ser eliminados, tamaño del tablero, formaciones por cada nivel [designación "gráfica" mediante una matriz de los enemigos en la formación], vida máxima del jugador, caminos, probabilidades de salto y disparo de los enemigos [por nivel], y atributos del tablero de juego [nombre del jugador inicial, título de la ventana, idioma...]).
- Las clases ***EnemyZako***, ***EnemyGoei***, ***EnemyCommander*** y ***EnemyEnterprise*** mantienen una relación de herencia con respecto a la clase ***Enemy***. De esta manera, los constructores de cada clase por tipo de enemigo establecen los valores a los atributos de la clase super ***Enemy*** en adición a aquellos distintivos propios de cada clase por tipo de enemigo (los cuales son añadidos independientemente de lo introducido en el constructor). Estos distintivos son ***type***, el cual determina el tipo de enemigo mediante un char (*así pues, 'Z' es el distintivo de los enemigos Zako*), y ***POINTS***, Constante que obtiene el valor de la clase ***Constants*** (*así pues, la puntuación que otorgan los Zakos es introducida en el constructor super mediante* ***Co nstants.ZAKOPOINTS***).
- La clase ***Torpedo*** es utilizada mediante la creación de los objetos torpedo en las clases ***Enemy*** y ***Player***. En ambas clases se utilizan arrays de torpedos.
- Se ha obviado la clase ***SoundLib*** ya que se trata de una clase externa obtenida de internet y ajena a la práctica cuya única función es la de añadir sonidos con el fin de ambientar el juego.

FUNCIONALIDAD

OBSERVACIONES PREVIAS

Con el fin de facilitar el entendimiento de los elementos que se han implementado debemos hacer hincapié en dos herramientas clave:

- **TRIGGERS (Gatillos)**

Al tratarse el juego de un *bucle constante*, con el fin de activar determinadas secciones del código según conveniencia, se ha diseñado un sistema de ***triggers***. Un trigger es un *boolean* cuyo objetivo es act ggivar o desactivar secciones del código de forma fácil y rápida.

- **CONTADORES (Control temporal / cuantitativo)**

En este juego, el control temporal se ha implementado mediante un *array* de ***contadores*** los cuales aumentan con el número de iteraciones del bucle del juego hasta un valor determinado. También existen contadores específicos (número de enemigos, nivel…).

TRIGGER	FUNCIÓN
trigger_levelStart	Se activa cuando el nivel comienza
trigger_enemiesReachedFormation	Se activa cuando los enemigos han completado su entrada y se han colocado en la formación
trigger_formationReachedMax	Se activa cuando la formación en movimiento ha llegado al límite lateral establecido
trigger_formationAnimated	Se activa para animar los sprites de la formación
trigger_gameRunning	Se activa cuando se comienza el juego (tras el menú de inicio) y se desactiva cuando se termina (victoria o derrota)
trigger_welcomeShowed	Se activa cuando se ha mostrado el logo de bienvenida
trigger_enemiesFreezed	Se activa cuando los enemigos están congelados
trigger_enemyDefeatedTxtShowing	Se activa cuando se ha eliminado a un *Galaga* o a un *Enterprise* y se quiere mostrar su texto característico
trigger_textShowing	Se activa cuando determinados textos están mostrándose
trigger_bonusActive	Se activa si se ha recibido un bonus hasta que este se usa
trigger_bonusInvincibleActive	Se activa si se ha utilizado el bonus de invencibilidad
trigger_spacePressed	Se activa cada vez que se pulsa el espacio. Añade un disparo y vuelve a desactivarse hasta la próxima pulsación.
trigger_soundsActive	Se mantiene activo si los sonidos del juego están activados
trigger_godModeActive	Se activa cuando se ha introducido el comando ***god***
trigger_gamePaused	Se activa cuando se ha pausado el juego con ***[TAB]*** o el comando **pause**
trigger_endFireworksActive	Se activa si se ha conseguido la primera medalla. Animación de fuegos artificiales
trigger_shotReceived	Se activa si el jugador ha recibido un disparo
trigger_allEnemyDead	Se activa si se han matado a todos los enemigos mediante el comando ***killEnemies***
trigger_levelGenerated	Se activa cuando el nivel se ha generado completamente (añadidos todos los enemigos)
trigger_levelCompleted	Se activa cuando el nivel ha sido completado
trigger_levelCommandIntroduced	Se activa cuando se introduce el comando ***level <Número de nivel>***

CONTADOR	FUNCIÓN
var_counter[0]	Contador de animación del logo de inicio
var_counter[1]	Contador de pulsaciones de la tecla *up* (menú) y activación del soundtrack en el *loop* del juego una única vez
var_counter[2]	Contador que tiene valor 1 si se está mostrando el menú *"Ayuda"*
var_counter[3]	Contador que tiene valor 1 si se está mostrando el menú *"Velocidad"*
var_counter[4]	Contador de pulsaciones de la tecla *tab*
var_counter[5]	Contador de pulsaciones de la tecla *space* en el juego. Al pulsarse tres veces, se vacía la barra de introducción de comandos.
var_counter[6]	Contador que controla el periodo de invencibilidad tras recibir daño
var_counter[7]	Contador que controla la animación de la pantalla de pausa
var_counter[8]	Contador que controla el número de imagen de la pantalla de pausa
var_counter[9]	Contador que controla la animación del *Sprite* del jugador [daño]
var_counter[10]	Contador que controla la animación de los *Sprites* de los enemigos
var_counter[11]	Contador que controla el tiempo de mostrado de los textos
var_counter[12]	Contador que añade espera antes de finalizar el juego
var_counter[13]	Contador que controla el tiempo que se muestra el logo al empezar el juego
var_counter[14]	Contador que controla el tiempo de congelación de los enemigos
var_counter[15]	Contador que controla el bonus de invencibilidad / modo dios
var_counter[16]	Contador que controla el movimiento de la formación [y límites]
var_counter[17]	Contador que controla el tiempo que se muestra el texto de nivel completado
var_counter[17]	Se activa cuando el nivel se ha generado completamente (añadidos todos los enemigos)
var_counter[18]	Contador que añade espera antes de finalizar el nivel
var_counter[19]	Contador que controla la animación del *Scoreboard* final
var_counter[20]	Contador que controla la espera de tiempo entre el *Scoreboard* final y las *vidas* finales
var_counter[21]	Contador que controla el tiempo de animación de las *vidas* finales
var_counter[22]	Contador que controla el tiempo de adición de las *medallas* finales
var_counter[23] a [25]	Contador que controla el tiempo de animación de los fuegos artificiales
var_counter[26]	Contador que controla las veces que se ha introducido el comando ***god***
var_bonusCounter	Controla el número de bonus actual
var_actualLevel	Controla el número de nivel actual
var_shotCounter	Contador utilizado para recorrer el array de disparos del jugador
*var_**<tipo de enemigo>**Counter*	Contador que almacena el número de **<tipo de enemigo>** por nivel
var_killedEnemies	Contador que almacena el número de enemigos eliminados

SPRINT 1: Enjambre, jugador básico

Todos los aspectos detallados en el ***Sprint 1*** de la práctica han sido desarrollados satisfactoriamente.

1- **Generar la estructura del enjambre con sus posiciones y diferentes disposiciones de enemigos en cada nivel.**

(Ejemplo: [Main class] líneas del código 2391 – 2508)

La **estructura del enjambre** se ha diseñado de forma que puede ser representada mediante una matriz de 7 filas por 9 columnas. La **matriz que representa el enjambre** comienza en las coordenadas *X = 40* e *Y = 10*.

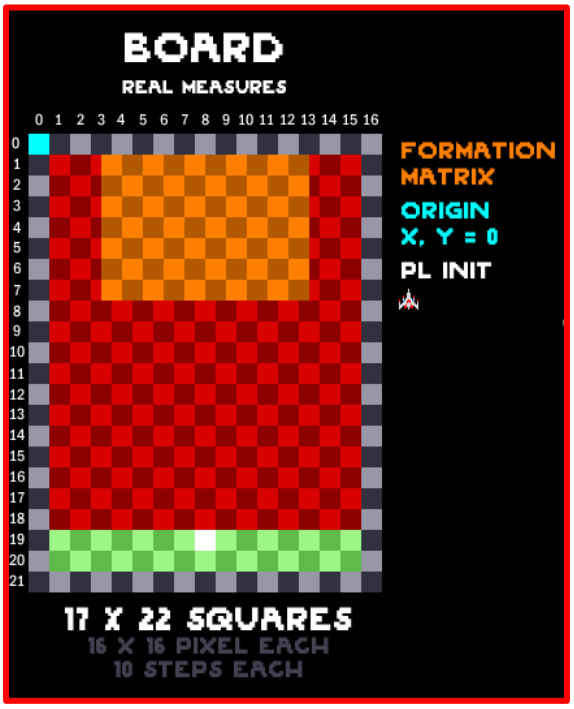


Figura 3. – *Esquema inicial 1:1 del tablero, la formación y la posición del jugador (y su región de movimiento)*

Para **establecer los enemigos en las posiciones de la matriz**, éstas se han especificados de "forma gráfica" en el código en la clase ***Constants*** de forma que la creación de nuevos niveles y disposiciones de enemigos es relativamente fácil.

La forma en la que se ha desarrollado esta idea, está basada en dos *fors* que en cada nivel recorren la matriz representativa del nivel (en la clase ***Constants***) buscando las designaciones de los enemigos (*chars*). Cuando se alcanza la designación del tipo de enemigo, se añade un enemigo de ese tipo, utilizando las ID's reservadas para cada tipo de enemigo (por centenares) y un contador por tipo de enemigo que aumenta en una unidad cada vez que se ha añadido un enemigo. Este contador aumenta la ID también.

La **adición de enemigos a la matriz de la formación** denominada ***enemies[][]*** tiene lugar cuando el par de *fors* encuentra un tipo de Enemigo distinto de '.' (el cual dejará la posición en la matriz en *null*), mediante la creación de un *objeto* del tipo ***Enemy<tipo de Enemigo>*** mediante su constructor. La concentración de todos los enemigos en una sola matriz permite simplificar el código y realizar todas las llamadas y métodos que tengan relación con los enemigos de una sola vez, sin tener que ir recorriendo diversas matrices de enemigos. De esta forma, teniendo en cuenta que la posición de la matriz no sea *null*, se tiene constancia de que en esa posición existe un enemigo.

La **distribución por centenares** de las ID's permiten la rápida identificación de los enemigos y a su vez, evita la superposición de ID's (ya que reservamos 100 ID's por tipo de enemigo).

Con respecto a las **coordenadas de la formación de cada enemigo**, son designadas mediante el mismo par de *fors y* la suma de la coordenada X del origen de la matriz de la formación (40) a la variable ***j*** que utiliza el for para recorrer las columnas de la matriz multiplicada por 10 (*ya que cada casilla son 10 pasos) y a la suma de la coordenada Y del origen de la matriz de la formación (10) a la variable* ***i*** que utiliza el *for* para recorrer las filas de la matriz multiplicada igualmente por 10.

De esta manera, de forma simple se obtiene la coordenada del enemigo (ya que si existe un enemigo de uno de los tipos designados en una casilla ***i***, ***j*** de la matriz, este enemigo obtiene las coordenadas de la formación mediante *formationX = 40 + (j*10), formationY = 10 + (i*10)* ***[Figura 4]***

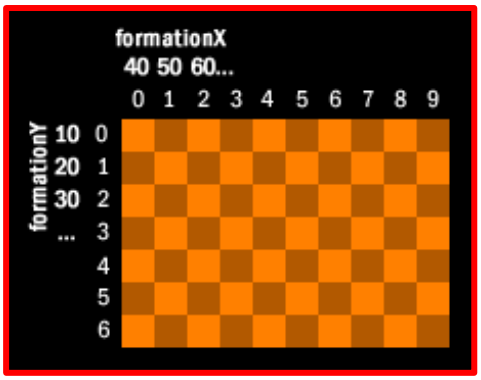


Figura 4. – *Esquema de la matriz de la formación teniendo en cuenta las coordenadas X e Y (**en negrita**) y las filas y columnas.*

Por otra parte, para las **animaciones de entrada**, los enemigos son dispuestos inicialmente mediante sus coordenadas X e Y de forma secuencial. Es decir, cuando se crean los enemigos mediante los *fors* mencionados anteriormente, la coordenada X se establece mediante un array de dos posiciones denominado (**<nombre de Enemigo>Init[]**) cuya posición 0 almacena la X inicial dotada a los enemigos de ese tipo y cuya posición 1 almacena la Y inicial.

Para distribuir de forma espaciada a los enemigos, se hace uso del contador de enemigos por tipo (**var_<nombre de Enemigo>Counter)**. Dependiendo del tipo de enemigo y de su posición inicial especificada anteriormente, se utilizará el número de enemigos del contador para añadir o restar separación en los mismos (de forma que entrarán con cierta separación y no todos al unísono), multiplicando por cierto valor de X o Y según conveniencia y sumándola a los valores iniciales determinados anteriormente.

En última instancia, se realiza mediante otro par de *fors* la **adición de los enemigos al tablero mediante la librería**, utilizando los métodos *gb_addSprite(), gb_moveSpriteCoord(), gb_setSpriteImage() y gb_setSpriteVisible()*.

2- Establecimiento del jugador y conseguir que se mueva (dentro de los límites).

(Ejemplo: [Main class] líneas del código 500 – 512 (movimiento), 45 (creación del objeto Player), 236 – 243 (adición del Sprite al tablero))

El jugador es añadido fuera del bucle principal del juego ya que se encuentra en todos los niveles. Como ya fue especificado con anterioridad, el jugador es añadido mediante los **métodos de la librería para la adición de Sprites *[ver punto 1]*** en las coordenadas X = 85 e Y = 190 *establecidos al comienzo del código en la creación del **objeto Playe***r. El movimiento del jugador está determinado por las teclas *left arrow key* y *right arrow key* del teclado. La **captura de la pulsación de esas teclas** está determinada por el *String **lastAction*** el cual obtiene la última acción de teclado mediante el método de la librería ***gb_getLastAction()**.trim* (.trim elimina los espacios).

Cada pulsación de una de las dos teclas designadas para el movimiento del jugador hace que su X incremente o disminuya en 4 pasos *(siempre y cuando el **trigger** de nivel comenzado (**trigger_levelStart**) tenga valor true, la vida del jugador no sea igual a 0, y la X del jugador sea menor a 160 (límite derecho) y mayor que 15 (límite izquierdo))* mediante el ***método move()***

3- Dotación de características del jugador, reflejarlas en el marcador.

(Ejemplo: [Main class] líneas del código 144 – 155)

El jugador tiene designados atributos en la clase ***Player***. Estos atributos tienen valor inicial 0 salvo las vidas actuales (inicialmente obtenidas de la clase ***Constants*** pero variables en el caso

de las vidas en la clase ***Player***). La vida máxima (la cual es obtenida de la clase ***Constants***) es reflejada inicialmente en el marcador del tablero. Para reflejar los valores de estos elementos se hace uso de los **métodos de la librería** entre los que destacan *gb_setValueHealthMax (cuyo valor es establecido desde* ***Constants****)* o *gb_setTextPlayerName (el cual establece el* ***nombre del jugador*** *obtenido mediante un* ***array de tres posiciones*** que realiza un *split* al comando ***new game <nombre>*** para almacenar el **<nombre>** y utilizarlo en el método anterior) entre otros. Estos valores son actualizados constantemente en el ***loop principal del juego*** mediante todos los métodos relacionados con el marcador del tablero.

SPRINT 2: Animaciones, torpedos del jugador

Todos los aspectos detallados en el ***Sprint 2*** de la práctica han sido desarrollados satisfactoriamente.

1- Mover el enjambre en conjunto.

(Ejemplo: [Main class] líneas del código 1568 – 1627)

El **movimiento de la formación** se ha realizado mediante un contador con el fin de que funcione con cualquier tipo de formación de enemigos establecida. El contador tiene que llegar a un valor inicial (ya que los enemigos se colocan tras la entrada en su posición determinada por *formationX* y *formationY* (donde la formación tiene como referencia el centro del tablero). Una vez ha llegado a este valor inicial predefinido, el contador comienza a incrementar hasta llegar hasta un valor determinado (el doble que el inicial, ya que llegando al valor inicial, encontrándose inicialmente la formación en el centro, la formación llega al borde derecho. En ese momento deberá recorrer el doble de distancia para llegar al borde izquierdo). Al llegar el contador a estos valores determinados, se activa un *trigger* denominado ***trigger_formarionReachedMax*** cuya función es aumentar en una unidad las coordenadas Y de la formación de forma que la formación se desplace lentamente hacia abajo (si y solo si la formación ha llegado a los límites preestablecidos). Si ***trigger_formationReachedMax*** se encuentra en *false* los enemigos se mueven hacia la derecha aumentando su formationX en una unidad, de la misma forma, si está en *true* se moverán hacia la derecha disminuyendo su formationX en una unidad.

Por otra parte, cuando **un enemigo se encuentra en la formación *inFormation*** de ***Enemy*** (en *true*), las coordenadas X e Y de ese enemigo son establecidas a sus coordenadas *formationX* y *formationY*. Cuando un enemigo salta de la formación, mantiene sus coordenadas X e Y y se obvian las de la formación, las cuales siguen moviéndose y cambiando con la formación al completo. *[líneas 848 – 864]*

2- Animar a los enemigos cambiando las imágenes según su dirección.

(Ejemplo: [Enemy class] líneas 79-110, [Main class] líneas 879 – 978)

El método ***getSprite()*** de la clase ***Enemy*** devuelve el nombre de archivo del *Sprite* asociado a la dirección actual del enemigo. Este método fue desarrollado en clase y se basa en la coincidencia entre el número identificador de la dirección y el número identificador del Sprite de la imagen asociada a la misma.

En cuanto a la animación de los *Sprites* en reposo (**iddle animation**), es activada por el ***trigger_formationAnimated*** y consiste en un *contador* que al llegar a cierto valor, cambia las

imágenes de todos los enemigos por la segunda imagen de su animación. Tras cambiar la imagen por la nueva, el *contador* vuelve a sumar iteraciones hasta que se reinicia, establece la primera imagen de la animación y repite el ciclo. Si los enemigos están congelados (***trigger*_enemiesFreezed**) no se producen cambios de *Sprite* y mediante el método ***getSprite()*** se obtendrán los sprites congelados directamente.

3- Disparar torpedos con la barra espaciadora. Eliminación de los enemigos si son alcanzados por un torpedo.

(Ejemplo: [Main class] líneas 525-560 (adición de disparos), 2143 – 2305 (impacto con los enemigos y movimiento de los disparos))

La **adición de los disparos del jugador** se ha basado en un *trigger* (***trigger_spacePressed***) el cual se activa si y solo si, el nivel ha comenzado (***trigger_levelStart***) y la vida del jugador no es igual a 0 cuando se pulsa el espacio. Al establecerse el *trigger* en true, un *for* recorre el *array de disparos del objeto* ***Player*** y si encuentra una posición vacía (*null*) añade un ***Torpedo*** en esa posición, utilizando como referencia las coordenadas del jugador. Ese torpedo es añadido al tablero mediante los **métodos de adición de sprites al tablero** vistos con anterioridad. Al añadir un disparo, el contador interno de disparos de la clase ***Player*** incrementa en una unidad y se establece el ***trigger_spacePressed*** en false, a la espera de otra pulsación para la adición de otro disparo. Si no hay posiciones libres en *null* no se añadirán disparos adicionales.

El **movimiento de los disparos** se ha realizado mediante un *for* que recorre el array de disparos hasta que encuentra una posición donde esté almacenado un objeto ***Torpedo*** (posición no vacía). En ese momento, mediante el método ***shotMoveNorth()*** de la clase ***Torpedo*** se incrementa la coordenada Y del torpedo en una unidad mientras el torpedo no impacte con un enemigo o se salga por el tablero (esto se consigue mediante el *boolean* de ***out*** de la clase ***Torpedo***). Cuando un torpedo se sale por los límites definidos, se establece en ***out***, al igual que si impacta con un enemigo.

El **impacto con los enemigos** se ha llevado a cabo mediante la comprobación de las coordenadas de los enemigos y las coordenadas de los torpedos, mediante tres *fors* encadenados (uno que recorre el array de disparos del jugador, otro que recorre las filas de la matriz de enemigos y otro que recorre las columnas de esa misma matriz). Cuando un disparo coincide con las coordenadas de un enemigo (más/menos un intervalo de impacto que hemos definido como ***HITBOX*** y que es modificable desde ***Constants***) el enemigo disminuye su vida en una unidad (mediante el método ***decreaseHealth()***, activando su boolean ***deadAnimation*** en caso de que su vida sea 0. Al jugador mediante el método ***upScore()*** se le suman los puntos correspondientes a ese enemigo y también se le suma uno a los aciertos. El contador ***var_killedEnemies*** aumenta en una unidad (servirá para comprobar que el número enemigos eliminados sea igual al número de enemigos en el nivel, lo que dará paso al siguiente nivel).

4- OPCIONAL: Indicación de la eliminación de enemigos por consola.

(Ejemplo: [Main class] líneas 2167 - 2168)

La **notificación por consola de la eliminación de enemigos** se ha realizado mediante el *método de la librería* ***gb_println*** utilizando un mensaje predefinido en adición a la *ID* que identifica al enemigo eliminado y a los *puntos* que otorga el enemigo al ser eliminado. Estos valores se obtienen de los objetos almacenados en ***enemies[][]*** (matriz de enemigos) mediante *getters*.

SPRINT 3: Entrada de los enemigos

Todos los aspectos detallados en el ***Sprint 3*** de la práctica han sido desarrollados satisfactoriamente.

1- Entrada de los enemigos en cada nivel, retardo (espaciado), posición inicial, ruta determinada con anterioridad en función del tipo.

(Ejemplo: [Main class] líneas 1629 – 2141 (fases de entrada)))

Las **entradas de los enemigos** se han realizado por *fases*. Dentro de la clase ***Enemy*** se define un contador de fase (el cual establece en la fase 0 inicialmente a cada enemigo) y un contador de vueltas (círculos, que cuenta los loopings que dan los enemigos). Los enemigos son establecidos como vimos en el **Sprint 1** con una separación (determinada por el número de enemigo) y unas coordenadas iniciales que se encuentran fuera del tablero. A partir de esas coordenadas iniciales, tienen lugar las entradas de los enemigos.

Al igual que hacíamos a la hora de añadir los enemigos en grupos similares mediante la paridad y la operación *módulo* los movimientos van a ser de igual manera repartidos de forma similar para aquellos enemigos que estén divididos en grupos (*En este caso, Zakos, Goeis y Enterprises*). En este caso se va a hacer uso de *el identificar de cada enemigo o ID* el cual mediante la operación *módulo* servirá para dividir los enemigos en dos grupos. Existe un *switch* inicial que mediante el atributo ***type*** (char que identifica a cada enemigo) designa la ruta a seguir según el tipo de enemigo.

La entrada tendrá lugar cuando el nivel haya comenzado (***trigger_levelStart***) y continuará siempre y cuando los enemigos no estén congelados (***trigger_enemiesFreezed***). Se han definido los movimientos mediante el método **move()** y los métodos **setX()** y **setY()** de cada enemigo.
Para realizar movimientos en círculo, se ha utilizado la dirección del enemigo actual, y se le ha sumado uno o quince (dependiendo de si nos interesa que el movimiento sea horario o antihorario) y se ha realizado el módulo por 16 (ya que existen 16 direcciones posibles). Cuando el enemigo llega a la última dirección (establecida dependiendo de si es horario o antihorario) se incrementa el contador de círculos o loopings en una unidad (***entryCircle***). Cada fase es llevada a cabo por cada enemigo de forma individual. De esta manera, cuando un enemigo llega hasta el punto designado como fin de la fase en la que se encontraba pasa a la siguiente.

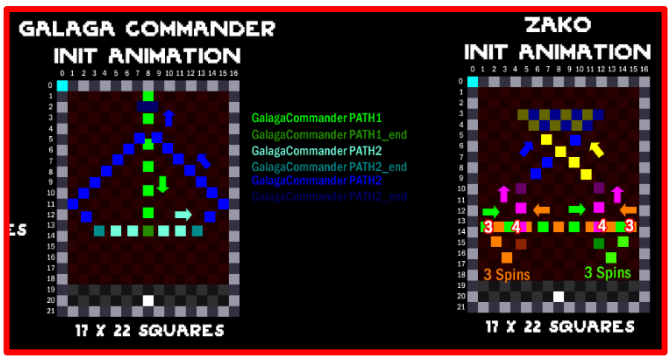


Figura 5. *– Boceto inicial a escala de las animaciones. Los colores representan las distintas fases*

Los puntos de cambio de fase se han designado mediante *ifs* con las *coordenadas X o Y* dependiendo del caso ***[Figura 5]***. Mediante un *switch* se comprueba en que fase se encuentra el enemigo y se realizan los movimientos designados para cada una en específico. Cuando el enemigo ha realizado todas las fases, se establece una penúltima fase en la cual los enemigos se colocan en la formación mediante la aproximación a la misma de forma lenta (sumando o restando 1 a su X y su Y hasta que

coincidan con *formationX* y *formationY* respectivamente). Al llegar a la posición final (en la formación) se establece la fase con el valor 8.

Cuando todos los enemigos llegan a la fase 8, se activan los *triggers*: ***trigger_formationAnimated*** y ***trigger_enemiesReachedFormation*** los cuales activan el movimiento de la formación y la animación de los sprites.

SPRINT 4: Ataques

Todos los aspectos detallados en el ***Sprint 4*** de la práctica han sido desarrollados satisfactoriamente.

1- Enemigos rompen la formación de forma aleatoria y descienden siguiendo rutas preestablecidas

(Ejemplo: [Main class] líneas 1352 – 1512)

Cuando los enemigos han llegado a la formación, si el nivel actual no es el nivel 0 (reinicio del juego), comenzarán a generarse *números aleatorios*. Si ese número aleatorio generado es menor que la *probabilidad de salto de la formación* recogida en la clase ***Constants*** y los enemigos no están congelados, el enemigo establece su *boolean* ***inFormation*** en false, lo que le permite obtener sus coordenadas X e Y en lugar de utilizar las de la formación y recorrer un camino. El camino se selecciona de manera aleatoria mediante un switch, y está implementado mediante el método ***followPath()*** el cual sigue los caminos predefinidos en la clase ***Constants***. Existe un contador propio para cada enemigo que al llegar a ciertas iteraciones, genera un nuevo número aleatorio para cambiar el camino que está siguiendo el enemigo.

Cuando esto ocurre, se reinicia el contador y los contadores propios que utiliza el método ***followPath()*** mediante los métodos ***resetPath()*** y ***resetProbCounter()***

2- Disparos enemigos de forma aleatoria con probabilidad determinada

(Ejemplo: [Main class] líneas 1352 – 1392 (comienzo del disparo aleatoriamente), líneas 2309 – 2332 (recorrer la matriz de disparos enemigos y hacer que se muevan), líneas 1164-1187 (comprobar si el disparo alcanza al jugador), [Enemy class] líneas 127 – 146)

Dentro de la clase ***Enemy*** se encuentra definido un array de torpedos.

Al igual que el salto de la formación, cuando un número aleatorio generado es menor que la probabilidad de disparo definida en ***Constants***, el enemigo comienza a disparar. Esto se realiza mediante el método ***startShot()*** el cual añade un disparo cada vez que es activado. Los disparos, con el fin de evitar la superposición de ID's, utilizan una ID que comienza en 2000 a la cual se le suma la ID del enemigo (multiplicada por 10) y el número de disparo de ese enemigo. De esta manera se evita la superposición de ID's de los torpedos ya que los enemigos tienen ID's secuenciales. En el loop del juego se recorre la matriz de disparos de cada enemigo. Cuando se encuentra una posición no vacía (no *null*), se mueve ese disparo mediante el método ***shotMoveSouth()*** de la clase ***Torpedo*** hasta que el torpedo impacta contra el jugador o sale por el tablero por la parte inferior (**out** se establece en true).

3- Disminución de la vida del jugador al ser alcanzado por un torpedo. Representación gráfica de las vidas del jugador. Muerte si un enemigo choca contra el jugador.

(Ejemplo: [Main class] líneas 1143 – 1296 (invencibilidad, daño al jugador, vidas en pantalla…)

Al igual que con los enemigos, se utiliza la ***HITBOX*** o caja de colisión, pero esta vez asociada a las coordenadas del jugador. Se recorre el array de disparos de cada enemigo, y cuando uno de estos disparos entra dentro del intervalo de coordenadas del jugador definido por la ***HITBOX***, se activa el ***trigger_shotReceived***, el cual controla que el disparo solo actúe una vez sobre el jugador. A continuación, se quita una vida al jugador mediante el método ***setHealth()*** si la vida del jugador no es igual a 0. Se activa la invencibilidad temporal y se establece ***trigger_shotReceived*** en *false*. En caso de colisión con un enemigo, la vida se establece a 0 y el jugador muere instantáneamente.

La **representación gráfica de las vidas** hace uso de la vida máxima definida en ***Constants***, añadiendo tantos *Sprites* como vidas se hayan definido. Las ID's de los sprites de las vidas están definidas a partir de 2 (ya que la ID del jugador es 1). Mediante un *for* se recorren los *Sprites* de las vidas del jugador. Si la vida del jugador no es igual a la vida máxima definida, se elimina el último corazón estableciendo su *Sprite* en el del corazón en blanco y negro.

SPRINT 5: Explosiones, comandos

Todos los aspectos detallados en el ***Sprint 5*** de la práctica han sido desarrollados satisfactoriamente.

1- Explosiones de los enemigos y del jugador

(Ejemplo: [Main class] líneas 980 – 1055)

Las **explosiones** se han implementado mediante el uso de contadores y del método ***setSprite()*** de ***Player*** y ***Enemy***. Los contadores utilizados para la animación de muerte se encuentran dentro de las clases ***Enemy*** y ***Player*** y se denominan ***animationCounter***. En el caso de los enemigos, disponer de un contador individual para cada enemigo es esencial, ya que si eliminamos a dos enemigos muy seguidamente, cada uno tendrá un ritmo distinto de explosión (puesto que uno de ellos habrá sido eliminado antes que el otro).

Cuando un enemigo o el jugador muere, se activa el *boolean* ***deadAnimation*** de su clase. Esto provocará que se active la sección del código que se encarga de cambiar el Sprite del jugador/enemigo por los respectivos a la explosión (con un intervalo de tiempo para animarlo). En caso de que los enemigos estén congelados (***trigger_enemiesFreezed***), se utilizarán explosiones de hielo. Al llegar al último cambio de Sprite de la explosión, se establece el jugador/enemigo en invisible mediante el método **setSpriteVisible()** de la librería.

2- Enemigos adicionales

Se ha **añadido un enemigo adicional** que se encuentra en el videojuego original denominado ***Enterprise***. Se ha modificado su Sprite para que sea blanco y se han añadido tres fases ya que el enemigo cuenta con 3 vidas. Se han establecido caminos de salto de formación especiales para este enemigo y se ha modificado el Sprite de su torpedo para simular que es un rayo láser. Todo esto se consigue mediante los *switches* que comprueban el tipo de enemigo.

3- <u>Comandos de consola (introducir help en el juego para visualizarlos en el mismo)</u>

(Ejemplo: [Main class] líneas 614 – 796)

Se han **<u>añadido múltiples comandos de consola</u>** con funcionalidades muy diversas. Todos ellos se han añadido mediante el *<u>switch</u>* que comprueba el valor de *<u>lastAction</u>*. Se ha distinguido entre los comandos de usuario y los comandos de desarrollador (los cuales han sido pensados para permitir una óptima y rápida valoración de esta práctica y de todas las funcionalidades).

COMANDO	FUNCIÓN
help	Muestra la lista de comandos (ventana emergente)
god	Establece / quita la invencibilidad permanente al jugador
level ***<level number>***	Finaliza el nivel actual y comienza el nivel especificado
endGame	Termina el juego actual como *victoria*
freeze	Congela todos los enemigos durante un periodo de tiempo
clearConsole	Vacía todos los mensajes de la consola de comandos
hideWindow	Oculta la ventana del marcador y solo deja visible el juego [Uso único]
pause	Pausa / continúa la ejecución del juego [Al igual que pulsar *TAB*]
stop	Pausa la ejecución del juego durante 30 segundos
playerPoints++	Añade 5000 puntos al marcador del jugador
playerPointsUp	Añade 1 punto al marcador del jugador
decreaseHealth	Quita una vida al jugador
reset stats	Reinicia los atributos y marcadores del jugador a los iniciales
killPlayer	Establece la vida del jugador a 0. Termina el juego como *derrota*
killEnemies	Elimina todos los enemigos del nivel actual y lo completa

4- <u>Nuevas acciones para teclas (up, down, tab…)</u>

- **Tecla up arrow**: en el menú inicial sirve para ocultar la ventana del marcador. En el juego sirve para activar los bonus disponibles.
- **Tecla down arrow**: en el menú inicial sirve para acceder a la ventana de ayuda
- **Tecla right arrow**: en el menú inicial sirve para acceder al selector de velocidad.
- **Tecla tab**: en el menú inicial permite la desactivación / activación de los sonidos. En el juego permite la pausa del mismo. En la pantalla final permite salir del juego.
- **Tecla space**: en el menú inicial permite empezar el juego o seleccionar la velocidad (si se encuentra el jugador en la ventana de selección de velocidad)

EXTRAS AÑADIDOS

Se han añadido múltiples elementos extra entre los cuales destacan el diseño de nuevas imágenes y la adición de sonidos.

- **Menú inicial con opciones** [ayuda, sonido, velocidad]
- **Imágenes y textos**
- **Bonus basados en la puntuación** [congelación e invencibilidad temporal]
- **Marcador gráfico de la puntuación actual**
  El cual se basa en 5 *Sprites* cuya imagen cambia según la puntuación del jugador (dividida en dígitos mediante la operación *<u>módulo</u>*)
- **Medallas en caso de victoria o derrota**
- **Comandos de usuario / desarrollador**
- **Pantalla de pausa, victoria y derrota**
- **Fondo aleatorio de estrellas**
- **Fondo de planetas superpuesto** [seleccionado entre tres imágenes prediseñadas de forma aleatoria]

Figura 6: *Diseño previo a implementación del marcador gráfico en pantalla.*

CONCLUSIONES

Conclusiones Finales

En esta práctica se ha realizado una recreación del videojuego ***Galaga***, se ha implementado mediante el uso de la librería aportada por el departamento de programación de la **uc3m** en el lenguaje de programación ***Java***. Para la reproducción fidedigna del videojuego, se han completado y desarrollado distintos Sprints, llegando finalmente a la totalidad de la funcionalidad del juego.

Con el fin de aportar mayor realismo y mejorar la experiencia a la hora de jugar el juego, se han añadido numerosos extras entre los cuales destacan los sonidos, las imágenes, el nuevo enemigo, el contador de puntuación en pantalla, los múltiples comandos que pueden ayudar a realizar determinadas acciones dentro del juego o las recompensas finales entre otras.

Problemas Encontrados

Debido a la realización de esquemas previos a la implementación de los Sprints en el código, se han realizado los diversos Sprints sin muchos problemas. Uno de los mayores problemas a la hora de trabajar con arrays han sido los errores espontáneos al recorrer determinadas posiciones. Sin embargo han sido solucionados satisfactoriamente.

La adición de extras como las imágenes y los textos han supuesto un reto a la hora de coordinar todos los extras con el juego.

Hubiese sido conveniente establecer una mayor separación entre la clase ***Main*** y los niveles, creando una clase ***Nivel***. A su vez, hubiese sido de nuestro agrado reducir la cantidad de líneas de código.

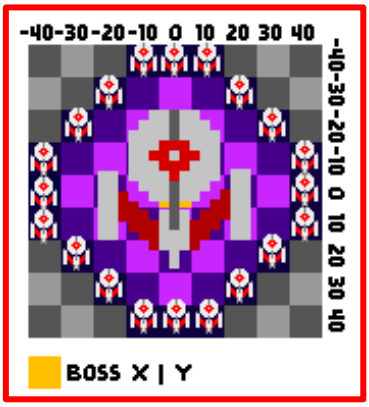


Figura 8. – *Esquema de coordenadas y formación del boss final*

En cuanto a la funcionalidad, desarrollamos un *boss final* el cual tenía enemigos orbitando alrededor de él. Sin embargo, por falta de tiempo, no pudimos implementarlo de forma óptima, por lo que fue obviado en el código final. ***[Figura 8]***

Comentarios Adicionales

La práctica ha sido una experiencia grata, la cual nos ha servido para aprender diferentes cosas entre las que destacan: entendimiento de las diferentes relaciones de herencia, comportamiento de los objetos en el código, programar con diferentes clases relacionadas entre sí, afrontar problemas y errores y tratar de solucionarlos revisando el código, entender la complejidad a la que pueden llegar ciertos juegos y su programación, y el desarrollo de ideas personales pensando la forma más óptima de implementación en el código.