uc3m | Universidad Carlos III de Madrid

PROYECTO FINAL DE PROGRAMACIÓN

Alumnos:

César Martínez Lara

Álvaro Morata Hontanaya

Índice de contenidos

1. Introducción.

Esta memoria recoge datos de la práctica final de programación del curso 2018/19, consistente en hacer un juego parecido al Galaga publicado por la empresa japonesa Namco en el año 1981 en Java.

Para navegar por este documento hay que saber lo siguiente:

- ➢ En el apartado 2., está la explicación de todo el diseño del programa. Eso incluye que hace cada método y variable de cada clase.
- ➢ En el punto 3., se explican los algoritmos que hemos creído más oportunos de explicar por ser los más complejos de todo el proyecto.
- ➢ En el punto 4., se encuentra, aparte de lo que se ha llegado a realizar de cada *sprint*, las reglas que rigen este juego. Por tanto, leérselo es necesario para saber cómo funcionan las diferentes mecánicas.

Además, para una fácil lectura de este documento se ha hecho una leyenda con ciertos colores utilizados junto a la fuente de letra `Consolas`:

- `Clase` ← Los nombres de las clases se designarán así.
- `Método()` ← Los nombres de los métodos se escribirán utilizando ese mismo color, también sus parámetros.
- `Variable` ← Los nombres de las variables se designarán con ese color.
- `Valor` ← Los valores que reciban ciertas variables se designarán con ese color.
- `Tipo` ← Los tipos de variable o método recibirán ese color.

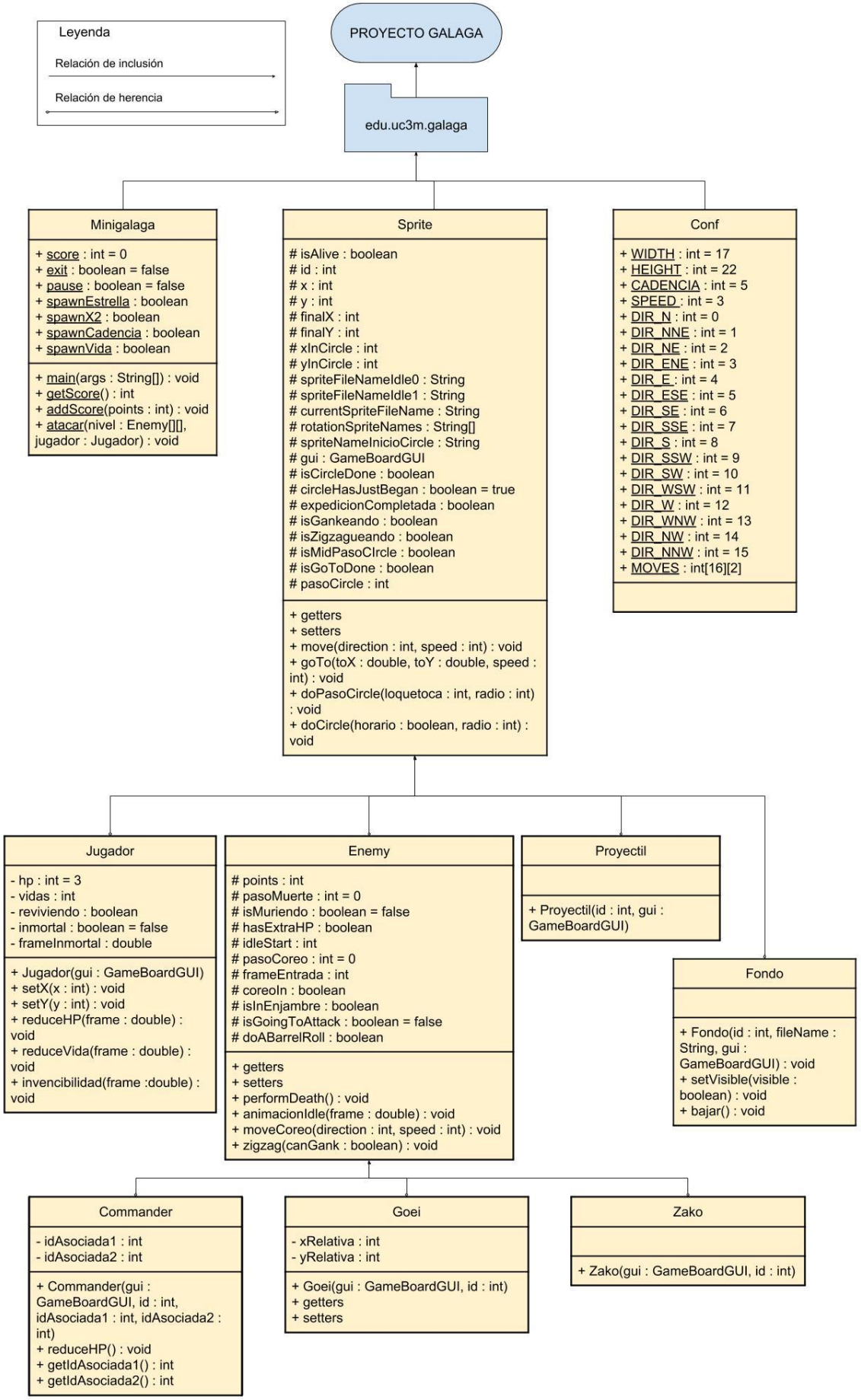
2. Diseño del programa.

En cuanto al diseño del programa, se ha optado por hacer un total de 10 clases, tal y como se verá en el punto 2.1., explicándose en el 2.2. la función de cada clase y dándose una breve explicación de cada método y variable.

2.1. Diagrama de clases.

Antes de ver el diagrama de clases, es conveniente leer lo siguiente:

- Ante la falta de recursos e inexperiencia de los creadores de este proyecto sobre los diagramas escritos en UML (Unified Modeling Language), se ha intentado aproximar lo máximo posible el diagrama siguiente a dicho estándar. Aún así, es muy posible que no se adapte totalmente a esta norma.
- Para ahorrar espacio en el diagrama, los `getters` y `setters` de ciertas clases se han escrito en conjunto. Aún así, si se desea ver cuáles son los `setters` y `getters` que han sido combinados en este diagrama, en el apartado 2.2. estarán descritas todas las clases con los `getters` y `setters` para cada variable.


Leyenda
Relación de inclusión
Relación de herencia
PROYECTO GALAGA
edu.uc3m.galaga
Minigalaga
+ score : int = 0
+ exit : boolean = false
+ pause : boolean = false
+ spawnEstrella : boolean
+ spawnX2 : boolean
+ spawnCadencia : boolean
+ spawnVida : boolean
+ main(args : String[]) : void
+ getScore() : int
+ addScore(points : int) : void
+ atacar(nivel : Enemy[][], jugador : Jugador) : void
Sprite
isAlive : boolean
id : int
x : int
y : int
finalX : int
finalY : int
xInCircle : int
yInCircle : int
spriteFileNameIdle0 : String
spriteFileNameIdle1 : String
currentSpriteFileName : String
rotationSpriteNames : String[]
spriteNameInicioCircle : String
gui : GameBoardGUI
isCircleDone : boolean
circleHasJustBegan : boolean = true
expedicionCompletada : boolean
isGankeando : boolean
isZigzagueando : boolean
isMidPasoCIrcle : boolean
isGoToDone : boolean
pasoCircle : int
+ getters
+ setters
+ move(direction : int, speed : int) : void
+ goTo(toX : double, toY : double, speed : int) : void
+ doPasoCircle(loquetoca : int, radio : int) : void
+ doCircle(horario : boolean, radio : int) : void
Conf
+ WIDTH : int = 17
+ HEIGHT : int = 22
+ CADENCIA : int = 5
+ SPEED : int = 3
+ DIR_N : int = 0
+ DIR_NNE : int = 1
+ DIR_NE : int = 2
+ DIR_ENE : int = 3
+ DIR_E : int = 4
+ DIR_ESE : int = 5
+ DIR_SE : int = 6
+ DIR_SSE : int = 7
+ DIR_S : int = 8
+ DIR_SSW : int = 9
+ DIR_SW : int = 10
+ DIR_WSW : int = 11
+ DIR_W : int = 12
+ DIR_WNW : int = 13
+ DIR_NW : int = 14
+ DIR_NNW : int = 15
+ MOVES : int[16][2]
Jugador
- hp : int = 3
- vidas : int
- reviviendo : boolean
- inmortal : boolean = false
- frameInmortal : double
+ Jugador(gui : GameBoardGUI)
+ setX(x : int) : void
+ setY(y : int) : void
+ reduceHP(frame : double) : void
+ reduceVida(frame : double) : void
+ invencibilidad(frame :double) : void
Enemy
points : int
pasoMuerte : int = 0
isMuriendo : boolean = false
hasExtraHP : boolean
idleStart : int
pasoCoreo : int = 0
frameEntrada : int
coreoIn : boolean
isInEnjambre : boolean
isGoingToAttack : boolean = false
doABarrelRoll : boolean
+ getters
+ setters
+ performDeath() : void
+ animacionIdle(frame : double) : void
+ moveCoreo(direction : int, speed : int) : void
+ zigzag(canGank : boolean) : void
Proyectil
+ Proyectil(id : int, gui : GameBoardGUI)
Fondo
+ Fondo(id : int, fileName : String, gui : GameBoardGUI) : void
+ setVisible(visible : boolean) : void
+ bajar() : void
Commander
- idAsociada1 : int
- idAsociada2 : int
+ Commander(gui : GameBoardGUI, id : int, idAsociada1 : int, idAsociada2 : int)
+ reduceHP() : void
+ getIdAsociada1() : int
+ getIdAsociada2() : int
Goei
- xRelativa : int
- yRelativa : int
+ Goei(gui : GameBoardGUI, id : int)
+ getters
+ setters
Zako
+ Zako(gui : GameBoardGUI, id : int)

2.2. Diseño de las clases.

2.2.1. Clase Sprite.

La clase abstracta `Sprite` es la clase madre de las clases `Jugador`, `Enemy` (y, por ende, de `Commander`, `Goei` y `Zako` también, como se verá posteriormente), `Torpedo` y `Fondo`.

Variables (todas ellas protegidas):

- `isAlive`: variable booleana que indica si el Sprite es visible.
- `id`: indica el número de identificación del objeto.
- `x`, `y`: indican la posición actual del objeto.
- `finalX`, `finalY`: indican la posición final de un objeto de este tipo.
- `spriteFileNameIdle0`, `spriteFileNameIdle1`: almacenan el nombre de los dos diferentes *sprites* para un objeto animado.
- `currentSpriteFileName`: almacena el nombre del *sprite* en uso.
- `gui`: objeto `GameBoardGUI` para almacenar datos de la interfaz de la clase `MiniGalaga`.
- `rotationSpriteNames[]`: almacena el nombre de los distintos *sprites* para las posibles rotaciones del objeto.
- `isGoToDone`: booleano que indicará si el método `goTo()` ha sido completado.
- `isCircleDone`: booleano que indica si se ha completado un movimiento circular.
- `pasoCircle`: indica el paso actual para el método `doCircle()`.
- `xInCircle`, `yInCircle`: indica la posición actual en el movimiento circular.
- `isMidPasoCircle`: indica si se está ejecutando un paso en el método `doCircle()`.
- `circleHasJustBegan`: inicializado en true. Indica si es la primera vez que se está ejecutando el método `doCircle()`.
- `spriteNameInicioCircle`: guarda el nombre del *sprite* en el que se empezó a ejecutar el método `doCircle()`.
- `isZigzagueando`: indica si el objeto esta realizando este tipo de ataque.
- `isGankeando`: indica si el objeto está realizando este tipo de ataque.
- `expedicionCompletada`: indica si se ha terminado de atacar

Métodos (todos ellos públicos):

- `getters` de `spriteFileNameIdle0`, `spriteFileNameIdle1`, `finalX`, `finalY`, `currentSpriteFileName`, `isZigzagueando`, `isGankeando`, `isExpedicionCompletada`, `spriteFileName`, `id`, `x`, `y`, `isAlive`.
- `setters` de `spriteFileNameIdle0`, `spriteFileNameIdle1`, `finalX`, `finalY`, `currentSpriteFileName`, `isZigzagueando`, `isGankeando`, `isExpedicionCompletada`, `spriteFileName`, `id`, `x`, `y`, `isAlive`.
- `move(int direction, int speed)`: mueve el objeto en la dirección indicada un número de veces `speed`.
- `goTo(double toX, double toY, int speed)`: cuando se ejecuta este método, el objeto se mueve hacia la posición (`toX`, `toY`) frame a frame.
- `doPasoCircle(int loquetoca, int radio)`: método para simplificar el código del método `doCircle()`. Se explicará mejor su funcionamiento en la parte de algoritmos más importantes.

- `doCircle(boolean horario, int radio)`: cuando se ejecute este método, el objeto ejecutará un movimiento circular en sentido horario o antihorario.

2.2.2. Clase Jugador.

La clase `Jugador` es una clase extendida de `Sprite`. Además de los métodos y variables de su superclase tiene las siguientes variables propias:

- `hp`: variable `int` inicializada en `3` que indica la vida del jugador.
- `vida`: variable que almacena las vidas del jugador.
- `reviviendo`: variable booleana que indica si el jugador está reviviendo.
- `inmortal`: indicará si el jugador está en el estado de invencibilidad.
- `frameInmortal`: almacena el *frame* en el que se inicia el estado de invulnerabilidad.
- `powerUpActivoX2`: informa de si este *powerUp* está activo.

Además, tiene los siguientes métodos:

- Un `constructor` que toma como parámetro un objeto `GameBoardGUI` llamado `gui`.
- `setX(int x)`: método que cambia la posición `x` a una nueva.
- `setY(int y)`: método que camba la posición `y` a una nueva dentro de unos límites.
- `reduceVida(double frame)`: método para reducir el valor de `vidas` del jugador.
- `reduceHP(double frame)`: método público para reducir el valor de `hp` del jugador.
- `invencibilidad(double frame)`: método que vuelve invulnerable al jugador durante un tiempo determinado.

2.2.3. Clase Proyectil.

La clase `Proyectil` es una clase extendida de la clase `Sprite`. Además de los métodos y variables de la clase `Sprite`, tiene el siguiente método:

- Un `constructor` que tiene como parámetros: `id` (`int`), `gui` (`GameBoardGUI`) y `sprite` (`String`).

2.2.4. Clase Enemy.

La clase abstracta `Enemy` es una clase extendida de Sprite, heredando sus métodos y variables. Además, tiene las siguientes variables y métodos propios.

Variables (todas ellas protegidas):

- `points`: puntos otorgados por matar al enemigo correspondiente.
- `pasoMuerte`: utilizada como contador para la animación de la explosión de los enemigos.
- `isMuriendo`: booleana que indica si el enemigo ha comenzado el proceso de muerte.
- `hasExtraHP`: memoria que indicará si el enemigo tiene vida extra o no.
- `idleStart`: corresponde al *frame* de comienzo de la animación de aleteo de los enemigos.
- `pasoCoreo`: indica el paso actual en la coreografía.

- `frameEntrada`: recoge el valor del *frame* en el que empezará a ejecutar la coreografía.
- `isInEnjambre`: si el enemigo está en su posición correspondiente del enjambre su valor será `true` y si no será `false`.
- `coreoIn`: indica si el objeto ha empezado la coreografía.
- `isGoingToAttack`: indica si va a atacar el enemigo.
- `doABarrelRoll`: booleana que indica si el enemigo va ha realizar en un ataque un giro circular.

Métodos:

- `performDeath()`: método para la animación de muerte de los enemigos.
- `animacionIdle(double frame)`: método de la animación pasiva de los enemigos (aleteo).
- `Getter` y `setter` para la variable `frameEntrada` y `setter` para la variable `isMuriendo`.
- `moveCoreo(int direction, int speed):` prácticamente igual que el método `move(int direction, int speed)` de la clase `Sprite`, solo que en este caso también incrementa el valor de `pasoCoreo` en `1` cada vez que se ejecuta.
- `zigzag()`: método para el movimiento o ataque en zigzag de los enemigos cuando salen de la formación.

2.2.5. Clase Commander.

La clase `Commander` es una clase extendida de la clase `Enemy`, a su vez extendida de la clase `Sprite`, por eso mismo hereda tanto las variables y métodos de Sprite como los de `Enemy`. Tiene también las siguientes variables propias:

- `idAsociada1`, `idAsociada2`: serán las id's de los Goeis que escolten a este comandante.
- El valor de `points` se cambia a `500`.

Métodos de la clase `Commander`:

- Un `constructor` que toma como parámetros un objeto `gui` (`GameBoardGUI`), una `id` (`int`), una `idAsociada1` (`int`) y una `idAsociada2` (`int`).
- Un `getter` para `idAsociada1` y otro para `idAsociada2`.
- `reduceHP()`: método que reduce la vida del comandante, cambiando su *sprite*.

2.2.6. Clase Goei.

Clase extendida de `Enemy`, del mismo modo que `Commander`. Tiene los siguientes métodos y variables:

Variables:

- `xRelativa`, `yRelativa`: posiciones relativas si son escoltas.
- El valor de `points` se cambia a `250`.

Métodos:

- Un `constructor` con los parámetros `gui` (`GameBoardGUI`) e `id` (`int`).
- `getters` y `setters` para `isEscolta1`, `isEscolta2`, `xRelativa` e `yRelativa`.

2.2.7. Clase Zako.

Subclase de `Enemy` al igual que `Commander` o `Goei`.

Variables:

- Se cambia el valor de `points` a `100`.

Métodos:

- Un `constructor` con los parámetros `gui` (`GameBoardGUI`) e `id` (`int`).

2.2.8. Clase MiniGalaga.

La clase `MiniGalaga` es la clase que contiene el método *main* de este programa, en la cual se sitúa todo el flujo del juego. A continuación, se describirán las variables declaradas en esta clase.

- `score`: variable entera, que almacena la puntuación del jugador a lo largo de la partida. Inicializada en `0`.
- `exit`: variable booleana, utilizada como variable de control para el bucle "infinito" que contiene todo el flujo del juego. Inicializada en `false`.
- `pause`: variable inicializada con valor `false` y que servirá también para el control del bucle infinito. Presionando el tabulador su valor cambiará a `true`, pausando el bucle hasta que se presione de nuevo y su valor cambie a `false`.
- `spawnEstrella`: indica si el *powerUp* de invencibilidad ha salido en pantalla.
- `spawnX2`: indica si este *powerUp* ha salido en pantalla.
- `spawnCadencia`: indica si el *powerUp* de aumento de cadencia ha salido en pantalla.
- `spawnVida`: indica si el *powerUp* de vida extra ha salido en pantalla.

Además de estas variables, la clase `MiniGalaga` contiene los siguientes métodos:

- `getScore()`: método estático y público que retornará el valor de la variable `score`.
- `addScore()`: método público y estático que sumará un valor a la variable dependiendo del tipo de enemigo eliminado `score`.
- `atacar(Enemy[][] nivel, Jugador jugador)`: ataque en zigzag y *gankeo*. Código necesario para que los enemigos realicen uno de estos dos ataques.
- `main(String[] args)`: método principal que no devuelve ningún valor (`void`). Dentro de este método se encontrará todo el control del flujo de la aplicación. Se va a proceder a explicar más en detalle sus partes más importantes.
    - En la parte inicial del código dentro del método `main()` nos podemos encontrar toda la parte de creación del objeto `gui`, la interfaz. Se cambia el color de cada cuadro a color negro con un bucle, se cambia de color la "red" a negro y se muestra como visible. Además, se efectúan ciertas configuraciones previas al inicio del juego.
    - La segunda parte presente es la creación del objeto `jugador` de clase `Jugador` y su configuración, la de los distintos niveles, los cuales son *arrays* bidimensionales de clase `Enemy`, que incluyen en cada una de sus filas el número de enemigos correspondiente a una fila del enjambre. También se encuentra en esta parte la creación del *array* `torpedos` de clase `Torpedo`.

- Además, hay un apartado en el cual se declaran todas las variables necesarias para el posterior código de este programa. Estas variables son:
  - `cooldown`: variable igualada a la constante `CADENCIA` de la clase `Conf`.
  - `frameLvl1`, `frameLvl2` y `frameLvl3`: estas variables `double` almacenarán el fotograma actual del correspondiente nivel, o lo que es lo mismo, contará las veces que se está realizando el condicional del nivel. Se inicializan en `0`.
  - `inLvl1`, `inLvl2`, `inLvl3`: variables `boolean` que se utilizarán como condición para saber cuál es el nivel actual. Se inicializan en `false` excepto `inLvl1`, que se inicializa en `true` porque es el primer nivel. Sus valores cambian según se vaya superando cada nivel.
  - `nextTorpedo` y `nextTorpedoEnemigo`: contadores `int` para los *arrays* de torpedos del jugador y de los enemigos. Se inicializan en `0`.
  - `izq` y `derecha`: variables `boolean` para controlar el desplazamiento lateral del enjambre. `izq` se inicializa en `true` y `derecha` en `false`.
  - `numTorpedosDisparados`, `numTorpedosFallados`: cuenta el número de disparos efectuados por el jugador y el número de disparos que no han impactado en un enemigo.
  - `lvl1Creado`, `lvl2Creado`, `lvl3Creado`: indica si han sido creados los distintos niveles.
  -
- A continuación, tenemos presente los bucles con condiciones `pause` y `exit`. Dentro de ellos tenemos presentes 3 condicionales que engloban la mayoría del código de esta clase. Estos condicionales tienen como condición las variables booleanas `inLvl1`, `inLvl2` o `inLvl3`. Los contenidos de estos condicionales son prácticamente iguales, así que se explicará una sola vez como generalización de los 3 niveles.
  - Inicialización de los *arrays* de enemigos y asignación de sus correspondientes *sprites*. En esta parte se declaran el tipo de enemigos de cada fila de los *arrays* `nivel1[][]`, `nivel2[][]` y `nivel3[][]`.
  - Inicialización y creación del fondo del juego.
  - Vidas en pantalla. Se muestran en la esquina inferior izquierda de la interfaz las vidas actuales del jugador.
  - Asignación de las posiciones iniciales y de la variable `frameEntrada` de cada elemento de los *arrays* de enemigos.
  - Creación de la coreografía del nivel con la utilización de los métodos de movimiento de las clases `Sprite` y `Enemy`.
  - Comprobación de que la coreografía ha sido completada y cambio del valor de `coreoCompletada`.
  - Movimiento del enjambre de enemigos. Aquí se incluye el manejo del conjunto de enemigos y ciertos cambios en las posiciones de los enemigos para su vuelta al enjambre.

        - Vuelta tras ataques. En esta zona se sitúa el código para la vuelta de los enemigos tras realizar un ataque.
        - Muerte de los enemigos. Se comprueba si la posición de alguno de los torpedos del jugador coincide con la posición de algún enemigo.
        - Animación periódica pasiva: código necesario para el aleteo de los enemigos.
        - Torpedos enemigos. Código necesario para que los enemigos lancen aleatoriamente torpedos contra el jugador, además de la comprobación de que, si la posición de uno de ellos coincide con la del jugador, se reduce su vida en 1.
        - Escolta. En esta parte se asignan los Goeis que escoltarán al comandante Galaga cuando ataque. A cada comandante se le asignarán dos Goeis escolta.
        - Colisión: se comprueba si la posición de algún enemigo coincide con la posición del jugador. Si es así, el jugador muere y acaba la partida.
    - Al final del todo se encuentra también la animación para la explosión del jugador.

2.2.9. Clase Fondo.

Clase extendida de `Sprite` creada para manejar el fondo de pantalla utilizado en el juego. Además de las variables y métodos de `Sprite` contiene los siguientes propios:

Métodos:

- Un `constructor` con los parámetros `id` (`int`), `fileName` (`String`), y `gui` (`GameBoardGUI`).
- `bajar()`: método utilizado para que el fondo vaya bajando.
- `setVisible(boolean visible)`: método para hacer que sea visible o no el fondo.

2.2.10. Clase Conf

Esta clase abstracta contiene ciertas configuraciones que utilizar en el resto de clases. Todas sus variables son finales, es decir, constantes. Las variables de la clase `Conf` son las siguientes:

- `WIDTH` y `HEIGHT`: indican la anchura y altura de la interfaz.
- `CADENCIA`: indica la velocidad de disparo del jugador. Cuanto mayor el valor de esta variable más lenta será la velocidad de disparo del jugador.
- `SPEED`: indica la velocidad de movimiento del enjambre. Cuanto más bajo el valor, mayor será la velocidad.
- `DIR_N`, `DIR_NNE`, `DIR_NE`, `DIR_ENE`, `DIR_E`, `DIR_ESE`, `DIR_SE`, `DIR_SSE`, `DIR_S`, `DIR_SSW`, `DIR_SW`, `DIR_WSW`, `DIR_W`, `DIR_WNW`, `DIR_NW`, `DIR_NNW`. Estas variables de tipo `int` indican las 16 direcciones posibles de orientación en este juego.
- `MOVES`: *array* bidimensional de tipo `int` que almacena los 16 movimientos posibles, según su dirección.

3. Algoritmos más relevantes.

3.1. Algoritmo del método goTo() de la clase Sprite.

Este método es en realidad el más sencillo, para escalas macroscópicas utiliza el ángulo que forma con la horizontal y así calcula el movimiento para recorrer el mínimo camino. Si el ángulo es menor que 11º se moverá horizontalmente 1 paso, si el ángulo es mayor de 79, se moverá verticalmente 1 paso.

Para escalas menores a 4 coordenadas de distancia el método está diseñado en base al siguiente diagrama que hemos creado:

El sprite empieza en el punto negro, en base al destino, hará diferentes movimientos:

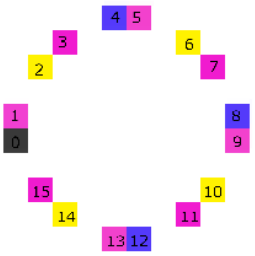
Si se tiene que mover solo horizontal o verticalmente (direcciones 0, 4, 8 y 12) hará los movimientos verdes;

en diagonal los amarillos (direcciones 2, 6, 10 y 14) ; para los azules, una combinación de los dos anteriores; los rojos son movimiento simples en base a la direcciones (1,3,5,7,9,11,13 y 15). En cuanto a los morados, son combinaciones de algunos de los anteriores.

Para hacer estos movimientos hace uso del método move() que lo mueve en en las direcciones cartesianas del 0 al 15.

3.2 Algoritmo del método doCircle() de la clase Sprite.

Es un método de movimiento que usa el goTo() para hacer un movimiento circular en 16 pasos en base a este diagrama:



Este diagrama representa los pasos que hay que seguir para realizar una circunferencia en sentido horario si el sprite está mirando a la dirección 0 (norte). Cada cuadrado representa una coordenada y el color representa el tipo de movimiento que le toca realizar al sprite. El color negro es el lugar donde empieza el sprite y debe realizar el movimiento número 0 para ir al paso 1 (es decir +1 en la vertical). Para ir al paso 2 debe hacer el movimiento 1 (es decir +1 horizontal + 2 vertical), etc. Los colores azules y el negro significa realizar un movimiento en vertical u horizontal. Los colores morados significa moverse 2 veces en una dirección y 1 en la otra. Los colores amarillos significan moverse 1 vez diagonalmente.

Este método también tiene en cuenta el radio deseado y esto lo hace simplemente multiplicando las distancias a moverse. Para un radio de 2, el paso 0 sería moverse 2 veces en la vertical, y el paso 1 sería moverse 4 veces en la vertical y 2 en la horizontal.

Cuando las distancias entre pasos son muy grandes, el sprite no puede hacerlas en un solo frame, así que guarda el destino (las posiciones del siguiente checkpoint) y vuelve a hacer un movimiento de aproximación en la siguiente iteración.

El método permite elegir si el movimiento será en sentido horario o antihorario. Para sentidos antihorarios los pasos a seguir serían al contrario de los horarios, es decir: el primer paso sería el mov. 15, el siguiente sería el mov. 14, etc.

Además tiene en cuenta la dirección inicial del sprite para que el centro de círculo sea el apropiado, para esto se toma la dirección cartesiana (0-15) inicial y ese número es el del movimiento a realizar primero, p.e.: Si su dirección inicial era la 4, primero realizará el movimiento número 4, luego el 5, … , hasta el movimiento número 3 (teniendo en cuenta que el sentido elegido era el horario)

3.3 Algoritmo del método atacar().

Este método es el encargado de hacer que los enemigos inicien aleatoriamente sus ataques. Para los enemigos de la fila 2 en adelante los pasos a seguir no pueden ser más fáciles:

Cada enemigo tiene una posibilidad de empezar un movimiento zigzagueante y una vez que lo está ejecutando tiene una probabilidad aún más pequeña de terminar de hacer zigzag y empezar un goTo() hacia la posición del jugador en ese preciso momento, una vez llega a esa posición vuelve a hacer zigzag hasta que desaparece por debajo de la pantalla. Para cada enemigo de la tercera fila los pasos que se sigue son los mismos, pero una vez ha llegado a la posición del jugador, éstos tienen un 50% de probabilidades de hacer una circunferencia en sentido horario.

Los enemigos de la 1ª fila no tienen voluntad de atacar, pues estos obedecen a su comandante, y solo empezarán a atacar cuando están en formación. Para ello previamente las formaciones se establecen así:

Al crearse los comandantes se pide quién será su escolta 1 y quién su escolta 2, para ello se guarda en el objeto comandante las posiciones de sus escoltas en su array. De modo que una formación sencilla en la que las filas son enemy[0] y enemy[1], estando el comandante en la fila 1(enemy[0]) y los 2 escoltas en la fila 2 (enemy[1]), los valores guardados en enemy[0][0] para escolta1 y escolta2 son 0 (por enemy[1][0]) y 1 (por enemy[1][1]) respectivamente. Para identificar a un escolta1 respecto a su comandante sería enemy[1][enemy[0][0].escolta1]. Con esto en mente, un comandante tiene un 1% de probabilidad de iniciar un ataque en cada frame. Una vez el comandante a dado la orden de atacar, sus escoltas empezarán un movimiento a sus posiciones respecto al comandante si no estaban ya en ese sitio. Cuando todos estén en su sitio, los 3 empezarán un movimiento zigzagueante hasta el fondo de la pantalla.

4. Partes de la práctica realizadas y funcionamiento del juego.

4.1. Sprint 1. Enjambre y movimiento básico del jugador.

Este Sprint fue realizado en su totalidad. El juego actual dispone de 3 niveles, cada uno con una formación propia y un número de enemigos diferente. En cuanto al movimiento lateral del jugador, hemos pensado que un movimiento de 1/10 cada vez que se pulsasen las flechas direccionales laterales era demasiado lento, más sabiendo que no responde del todo bien por configuración de la interfaz, por lo que las decisiones finales fueron que se moviese ½ de casilla por cada pulsación de las flechas. Se añadió también de forma adicional el movimiento del jugador hacia arriba o hacia abajo dentro de unos límites espaciales. Esta mecánica puede no utilizarse y no tendrá repercusión alguna en la experiencia de juego. En caso de utilizarse puede ayudar al jugador a esquivar ciertos ataques y, por tanto, a hacer el juego más fácil.

El jugador está dotado de los campos puntuación, vida máxima y vida actual, y se muestran en la interfaz.

Además, se muestra el número de disparos realizados y el número de disparos que no han alcanzado a un enemigo.

4.2. Sprint 2. Animaciones y torpedos del jugador.

La primera parte que se realizó fue el desplazamiento lateral conjunto del enjambre. Posteriormente se creó una animación para el “aleteo” o animación pasiva de los enemigos cuando están situados en su posición correspondiente dentro del enjambre.

Se creó también el código necesario para que el jugador pudiese disparar torpedos. La cadencia de disparo es configurable desde la clase `Conf`. La seleccionada por defecto es una cadencia bastante baja, que permite que haya unos 3 torpedos a lo largo de toda la pantalla. Se pensó así para aumentar la dificultad. Presionando espacio se puede lanzar un torpedo. Si se deja presionado se consigue la máxima cadencia posible, pero no es posible moverse y disparar manteniendo pulsadas alguna de las flechas direccionales y el espacio simultáneamente.

Más tarde se creó el método `move()` que cambia el *sprite* del enemigo correspondiente según la dirección en la que se mueva.

Se ha optado por no mostrar por la consola de la interfaz la muerte de cada enemigo, ya que se pensó que podría entorpecer la experiencia de juego.

4.3. Sprint 3. Entrada.

Se han programado 3 entradas o coreografías diferentes para los 3 distintos niveles disponibles, en las cuales los enemigos tienen tiempos de entrada diferentes, y dependiendo de la posición de la que se quisiese que saliesen se sitúan en una posición inicial o en otra. Después de terminar la entrada o coreografía correspondiente se dirigen a su posición determinada dentro del enjambre.

4.4. Sprint 4. Ataques.

Programados dos tipos de ataques: ataque en zigzag y *gankeo*:

- Ataque en zigzag: el enemigo correspondiente baja realizando un movimiento en zigzag hasta el borde inferior de la pantalla para volver a su posición por el borde superior de la pantalla. Este movimiento puede ser realizado por los comandantes galaga, los goeis o por los zakos. Además, los comandantes galaga tienen asignados dos goeis escolta. Si los dos o un goei escolta están vivos, cuando haga este ataque el comandante, estos dos saldrán con él a realizar el ataque en zigzag. Si se matan antes a los comandantes galaga que a sus goeis escolta, estos entran en modo *rampage* (aumenta su velocidad de movimiento en zigzag y no paran de atacar).
- *Gankeo*: este tipo de ataque se resume en que los enemigos van a la posición en la que estaba el jugador a modo de kamikaze. Si fallan en matar al jugador, siguen hasta el borde de la pantalla y vuelven a su lugar del enjambre una vez reaparecen por la parte superior. Este ataque puede ser realizado por los zakos o los goeis que no escoltan a ningún comandante galaga.

Los enemigos disparan de forma aleatoria torpedos. Si estos torpedos tocan el *sprite* del jugador, entonces este pierde una vida. Si solo le queda una vida muere.

Si uno de los enemigos impacta con el jugador, pierde todas las vidas y acaba la partida.

4.5. Sprint 5. Animaciones de explosiones. Comandos de la consola.

Programadas las animaciones de explosión tanto de los enemigos como del jugador.

Actualmente los comandos disponibles son:

- Si se presiona “tab” el juego se pone en pausa.
- Si se presiona “left” el jugador se mueve ½ de casilla hacia la izquierda.
- Si se presiona “right” el jugador se mueve ½ de casilla hacia la derecha.
- Si se presiona “up” el jugador se mueve ½ de casilla hacia arriba (dentro de unos límites).
- Si se presiona “down” el jugador se desplaza hacia abajo ½ de casilla (dentro de unos límites).
- Si se presiona “space” el jugador dispara un torpedo que sigue una trayectoria hacia el borde superior de la interfaz.
- Si se recibe el comando “exit game” (botón de salir) se finaliza el programa.

4.6. Extras.

Se ha animado un fondo para mejorar la estética del programa. Consiste en un fondo estrellado con 3 profundidades distintas. Cada una de estas profundidades se mueve con una velocidad diferente para dar un efecto de velocidad en el espacio.

Agregado un tiempo de invencibilidad si el jugador recibe un disparo enemigo. Durante este tiempo, la nave del jugador parpadea.

PowerUps (4 disponibles):

- Estrella: vuelve la nave del jugador dorada e invulnerable durante 10 segundos.
- X2: da el doble de puntuación al matar enemigos durante 10 segundos.
- Cadencia: aumenta la cadencia de disparo del jugador durante 10 segundos.

- Vida extra: da una vida extra al jugador.

5. Conclusiones

Álvaro Morata Hontanaya: El proyecto nos ha parecido una muy buena manera de aprender a diseñar y crear un programa de una envergadura superior a las prácticas que veníamos haciendo anteriormente. Aún así, ya sea por un mal diseño o poca experiencia haciendo un programa tan grande nos hemos encontrado con varios problemas:

• Numerosos errores, la mayoría de ellos solucionados. Aún así, es posible que haya algún bug. Estos bugs no impedirán jugar el juego o que éste presente un error de ejecución, sin embargo, puede ser que a lo mejor la posición de algún enemigo varíe y no se posicione bien en su sitio del enjambre.

• Errores a la hora de coordinarse para trabajar el proyecto. Al no poder trabajar en conjunto sobre el mismo proyecto, trabajar simultáneamente se hace bastante complejo.

Al no saber manejar ciertas clases nos hemos visto un poco atascados a la hora de implementar ciertos extras que queríamos poner, como es el caso de los sonidos y la música.

Opinión de César Martínez Lara: Yo preferiría haber tenido que hacer el de Civ que hicieron otro año porque personalmente me gusta más ese género, pero al menos los juegos arcade como este son más fáciles de testear.

Mis últimas 2 semanas han sido trabajar sin parar, sobre todo los últimos 7 días han sido desde que me despertaba hasta que me acostaba a las 3a.m., primer año de programación y ya he conocido el “crunch”, Al 4 día no sabía a qué día estábamos de la semana y a los 2 días antes de terminar había perdido la noción del tiempo completamente. Esto es culpa mía por no haber sabido planear con antelación, eso lo tengo clarísimo vamos.

Cada vez que ejecutábamos el programa después de añadir líneas de código una parte de mí quería ver qué clase de bugs divertidos ocurrían, incluso nos gustó tanto uno que lo acabamos introduciendo como ataque especial. Pero la verdad es que la mayoría del tiempo me lo he pasado corrigiendo bugs y esa parte ya no era tan divertida.

De lo que más estoy orgulloso es de los que yo llamo “mis hijos predilectos”, el método goTo() y el doCircle(), me habré tirado más de 30h entre los 2 para sacarlos adelante y quitar bugs. doCircle() tenía 4000 líneas al principio, al día siguiente lo dejé en 1500 y a última hora se me ocurren formas de reducirlo a menos de 300. También cuando programé los powerUps en una 1 hora, me impresioné a mí mismo.

Me da rabia no haber sabido solucionar los bugs de última hora, pero al menos como dicen en la industria, el juego es “gold”, estaría listo para enviar (Aunque no con mucho contenido)

A 1 hora y 10 minutos de que se cierre el plazo de entrega me encuentro super orgulloso de mi trabajo estas últimas semanas y una parte mí cree que por fin ha encontrado su vocación.