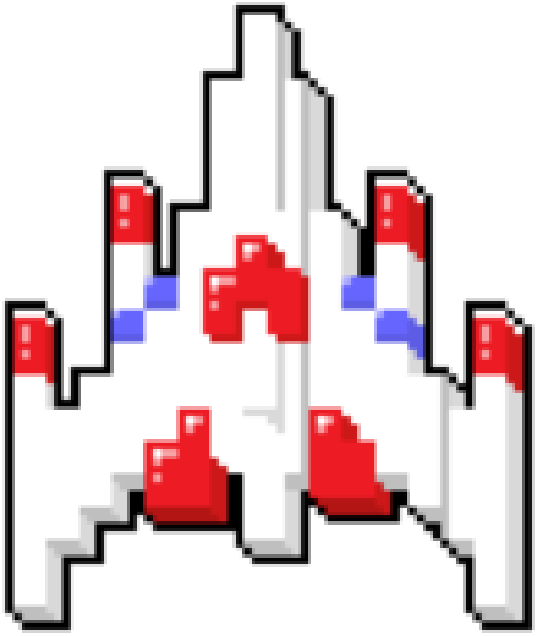
Memoria MiniGalaga



Autores

Alberto Martín Mateos

Luis Pinilla Rubio

Índice

Introducción

MiniGalaga se trata de un programa en el que el jugador, teniendo el control de una nave situada en la parte inferior de la pantalla, intenta matar a los enemigos (Goei, Comandante Galaga y Zako) a través de disparos, mientras se defiende de sus ataques.

En este documento trataremos de mostrar las principales características del juego y de su código, intentando hacer comprensibles aquellas partes que pudieran resultar difíciles de entender.

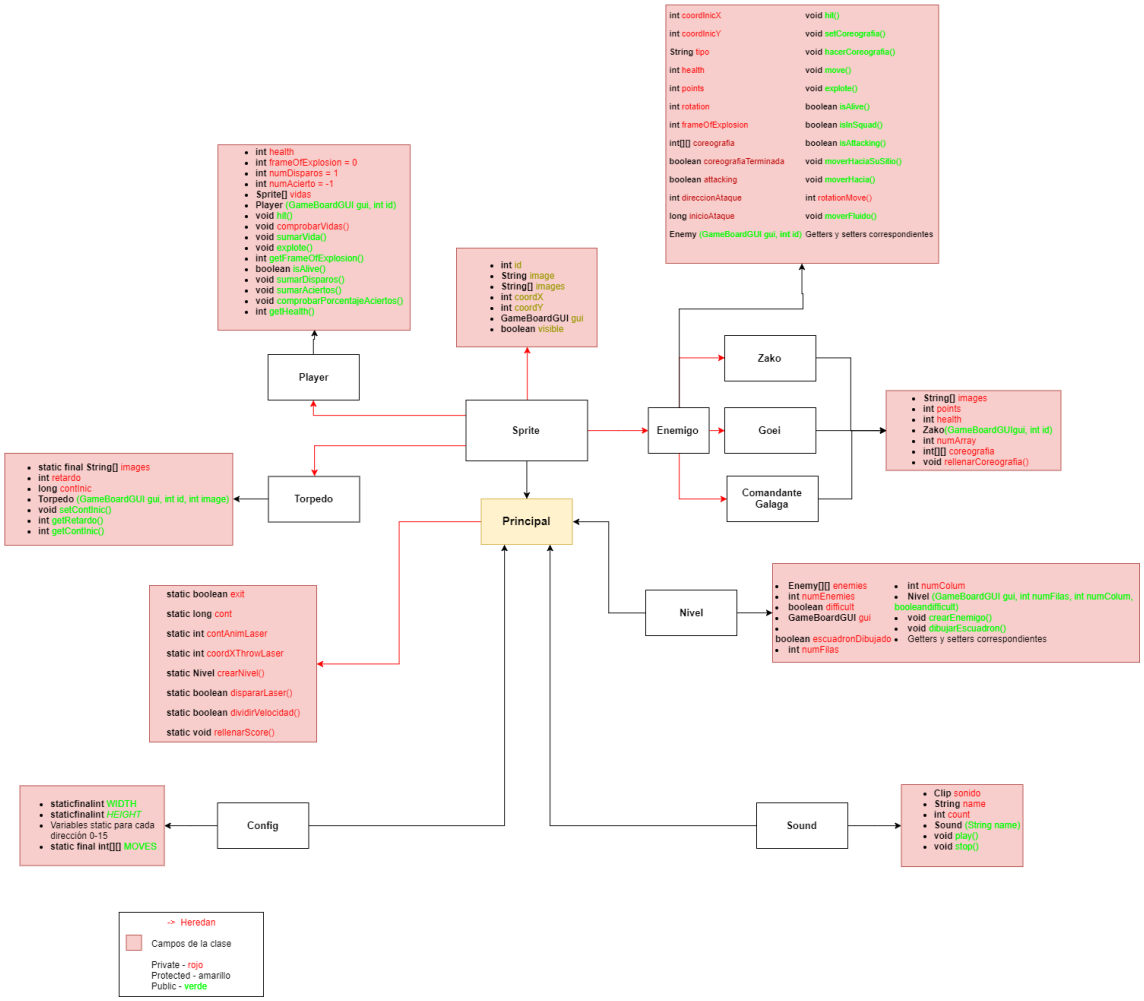
Instrucciones

Para que el juego se ejecute perfectamente deberá seguir detalladamente las siguientes instrucciones:

- El archivo JAR, la carpeta “Images”, la carpeta “Sounds” y la carpeta lib (libreria) deberán estar juntos en la misma carpeta para que el JAR pueda acceder a las imágenes y sonidos.
- El archivo ZIP, impórtalo a eclipse, importa además la librería (carpeta lib), y arrastra dentro del proyecto de java (en eclipse), la carpeta imágenes y sonidos.
- Si dispones de la carpeta imágenes que se aportaba al iniciar el proyecto, borrala e importa la nuestra. En la nuestra se encuentran las imágenes que hemos añadido además de las imágenes que ya venían, de las cuales algunas las hemos tenido que arreglar, ya que había cuatro imágenes mal colocadas: enemy2G1.png (aleteo), enemy105.png, enemy205.png, enemy305.png y enemy905.png
  La imagen enemy2G1.png le falta el aleteo y las demás están incorrectas, están invertidas verticalmente, deberían apuntar más hacia abajo

Una vez hecho esto ya podrá disfrutar del juego tranquilamente.

Diseño de clases



Métodos y algoritmos más relevantes

Clase principal

Creación de niveles aleatorios

Una de las partes malas que una gran mayoría de juegos es que al final se termina, bien sea porque ganas o porque pierdes, siempre se termina. Para que la diversión no se acabase nunca decidimos crear todos los niveles posibles para que, mientras no mueras puedas jugar todo lo que quieras. Una solución sería ir creando niveles uno a uno, pero dado que esta solución es muy laboriosa y larga, decidimos que este trabajo lo podían hacer mucho mejor el mismo ordenador, ya que es mucho más rápido y nunca se “cansa”, de ahí surgió el método:

- `crearNivel (GameBoardGui gui, int numNivel, Sprite[] proyectilesLaser, Sprite labelScore, Sprite labelLevel, Sprite spriteNumLevel, Sprite[] arrayPuntos, Sprite base, Sprite gameover, Sprite winner, Player player)`

Ciertamente para crear un nuevo nivel con este método solo haría falta la parte que está en negro, el GameBoardGui y el int, la parte en gris es únicamente para que, una vez creado el enjambre del nivel, los Sprites en gris queden por encima del enjambre, simplemente por cuestión de estética.

Dentro de este método diseñamos manualmente el nivel 1, 2 y 3, pero a partir de este se generan completamente de forma aleatoria. Se elige aleatoriamente el número de filas y de columnas del escuadrón y la dificultad del nivel, además de las posiciones y tipos de enemigos, los cuales también son aleatorios

En nuestro programa, cada vez que se pasa de nivel se borra el nivel antiguo, se crea el nivel nuevo y se sobrescribe, esto hace que disminuya bastante la memoria usada por el juego, en lugar de crear un inmenso número de niveles a la vez.

Ralentización

Este juego se ejecuta a 60 frames por segundo, una velocidad media de juego, si todos los movimiento y procesos que debe realizar los hiciera una vez por frame, el juego sería prácticamente imposible de manejar, ya que iría todo demasiado rápido. Por ello se necesita ralentizarlo, el problema es que se necesita ralentizar solo algunos procesos, porque si fuera ralentizar el juego completo, disminuiríamos los frames por segundo y estaría solucionado, pero esto no nos vale.

Para realizar dichas ralentizaciones nos basamos en dos maneras:

1. Por tiempo natural: Usamos el `System.currentTimeMillis()`(Método que te devuelve el tiempo real en milisegundos), para saber cuánto tiempo ha pasado desde su última ejecución. Esto lo usamos para controlar el tiempo que esta un enemigo fuera del escuadrón, cuando lleve 10 segundos fuera del escuadrón regrese a él.
2. Por frames: Para ello usamos un contador, que sume 1 cada vez se complete un bluce/frame. Para ralentizarlo usando esta manera creamos el método:

   - `dividirVelocidad(int num)`

   Este método recibe un entero, y realiza la operación módulo:

   <contador de bucles> % <numero entero>

   Si esta operación da 0 devuelve true, de lo contrario devuelve false. Con esto conseguimos que cuando se realice un proceso que queramos

ralentizar, la primera vez que se produzca puede haber cumplido o no la ralentización (pero esto no nos interesa saberlo, ya que lo que queremos ralentizar es el proceso, no el comienzo del proceso), pero en los pasos siguientes si la cumplirá.

Ejemplo:

Si la ralentización es `dividirVelocidad(4)` y el contador está a 175, cuando se inicie el proceso no va a esperar 4 bucles, solo 1, porque 176%4=0, pero al siguiente paso si se deberá esperar hasta el bucle 180 (que es lo que nos interesa), y así seguirá sucesivamente de 4 en 4 bucles.

Clase Enemy

Movimiento Fluido

Uno de los grandes problemas que surge cuando crear un juego con movimientos en tiempo real, es la fluidez. Es decir, puedes ordenar que un enemigo o imagen vaya a unas coordenadas X e Y exactas pero lo que no puedes ordenar es que vaya a un punto de la pantalla sin saltar directamente, es decir, que vaya poco a poco hacia ese punto. Esto lo hemos conseguido con el uso de coordenadas cartesianas y coordenadas polares que te permiten ir en 16 direcciones de manera lineal (teniendo en cuenta las diagonales), y entonces ahora debes ir paso a paso.

Con esto parece que ya está solucionado el problema por completo, pero en realidad solo se ha solucionado una parte del problema, tenemos fluidez en el desplazamiento, pero ahora faltaría la fluidez en la dirección, es decir, que no cambie de dirección bruscamente. Para ello se nos ocurrió el siguiente método:

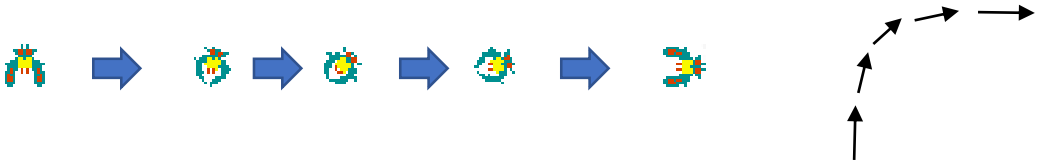
- `moverFluido(int direccion)`

Usando este método, cuando se cambia la dirección del movimiento, se le obliga a que pase por las direcciones que hay entre medias, dando la sensación de mayor naturalidad y fluidez. A continuación, con las siguientes imágenes representaremos la función del método.

Ej.: Al mover un enemigo al norte y después al este

X Sin el método

\- Con el método



Como se puede apreciar en las imágenes anteriores con nuestro método, se puede conseguir hacer curvas y darle mayor fluidez al movimiento. Ahora sí, con ambas cosas (Coordenadas cartesianas y moverFluido) se ha conseguido una fluidez óptima para el movimiento, dándole un "aire" más realista.

Mover hacia un lugar

Además de la fluidez conseguida en el apartado anterior, para realizar una gran cantidad de movimientos, sería mucho más sencillo decirle a dónde quieres que vaya el objeto, y que este vaya por el camino más corto, hasta allí; en lugar de tener que ir paso a paso, pensando en cada frame a donde le toca ir para acabar donde le has dicho que acabe. Pues exactamente esto es lo que hace nuestro método:

- `moverHacia(Sprite destino, boolean sentido)`

En este caso se ha diseñado para que vaya hacia las coordenadas de un Sprite, en lugar de unas coordenadas fijas, pero funcionaría igual. Este método calcula cuanta distancia hay desde sus coordenadas hasta las coordenadas a dónde se quiere ir, calcula que dirección polar (N,S,E,O,…) debe tomar para llegar antes (el camino más corto es en línea recta), y llamando a este método el objeto va hacia esas coordenadas de manera automática frame a frame, sin preocuparte de donde se encuentra el objeto en ese instante.

El método:

- `moverHaciaSuSitio()`

en realidad, es una variación del método `moverHacia`, en el que las coordenadas a dónde debe ir, en este caso son sus coordenadas iniciales en el escuadrón.

Funcionalidades extras

En este apartado hablaremos sobre las diversas funcionalidades extras que hemos añadido a nuestro Galaga, aportando al juego una mayor diversión y un mayor entretenimiento, evitando que la monotonía acapare el juego.

Power-Up

- **Láser**: Se trata de un disparo especial con el que hemos dotado al jugador. Cuando conseguimos coger uno de los ítems que caen (las dos barras azuladas), disponemos de un uso de láser. Para activarlo, debemos pulsar la tecla del tabulador; de esta manera empezará a cargarse y cuando termine, lanzará el disparo matando a los enemigos que alcance en su camino.

- **Llave**: Es el otro ítem que cae desde arriba. Cuando el jugador coloca la nave en el camino de la llave, al encontrarse el jugador suma una vida extra.

Sonido

El juego dispone de varios sonidos que le dan un ambiente más ameno y entretenido. De momento, como prueba solo hemos añadido sonido a los disparos de torpedos del jugador y al disparo laser, en un futuro se podría añadir más sonidos. Ej.: Sonido explosiones, sonido torpedos enemigos, sonido ambiental (de fondo), sonido pase de nivel, etc.

Varios niveles de dificultad

El juego posee dos niveles de dificultad un fácil, que es el predeterminado para los niveles 1 y 2, y otro modo difícil para el nivel 3 y algunos de los aleatorios.

El segundo nivel de dificultad (Modo Difícil), aumenta la densidad de disparos enemigos, aumenta las probabilidades de que los enemigos salgan del escuadrón, y disminuye la probabilidad de que caigan power-ups. Además, permite que al enemigo que cuando llegue al límite superior dela pantalla aparezca por debajo sorprendiendo al jugador y destruyendole, mientras que en el Modo Fácil cuando el enemigo llegaba arriba daba la vuelta para no salirse de la pantalla.

Varias velocidades

El juego posee tres velocidades, por defecto es la velocidad 2, la velocidad 1 es más lento, y la velocidad 3 es más rápido. Los diferentes modos de velocidad afectan a la velocidad de movimiento del escuadrón, a la velocidad de los torpedos enemigos, y a la velocidad de movimiento del ataque de los enemigos.

Modo “god” y “morir”

El juego dispone de un modo “god” (modo dios) (invulnerable a los ataques enemigos), mientras esté activado no recibes ningún disparo ni choque de los enemigos, por lo que no puedes morir, excepto por un comando extra que hemos añadido, el comando “morir”, en el que destruyes tu nave y pierdes.

Conclusión

La práctica ha sido resuelta totalmente según indicaba su enunciado. Además, como hemos señalado anteriormente, hemos añadido algunos elementos como el nivel aleatorio, los sonidos o los power-up.

Sin embargo, por la limitación del tiempo no hemos podido implementar varios apartados: el retardo del movimiento del jugador que trae el código del profesor, un modo multijugador para que dos personas pudiesen jugar a la vez o más sonidos, entre otros.

Para finalizar, destacar que ha sido una práctica que nos ha resultado difícil por la cantidad de tiempo que necesita, sumado a que estamos empezando a programar. En cambio, una vez superada la prueba, ha sido gratificante y divertido haber hecho un juego así.