Práctica de
Lenguajes de Programación

Junio 2008

Alumno: Antonio Rivero Cuesta
CENTRO ASOCIADO DE PALMA DE MALLORCA

Escuela Técnica Superior de
Ingeniería Informática

UNED

ÍNDICE

MEMORIA DE LA PRÁCTICA

Descripción de la práctica
En la práctica de este año se va a diseñar e implementar una versión del legendario arcade “Space Invaders”. Esto nos servirá para estudiar y practicar los mecanismos de la Programación Orientada y Objetos y el soporte a la concurrencia de Java.

Historia del juego
Space Invaders fue diseñado y programado por Toshiro Nisikado para Taito, Japón en 1978. En 1980, el juego se diseñó para la Atari 2600, y se convirtió en el juego más popular en esta plataforma y en uno de los más famosos de todos los tiempos. Además de estar en todos los salones recreativos, también se implantó para la Nintendo, consiguiendo enormes beneficios económicos.

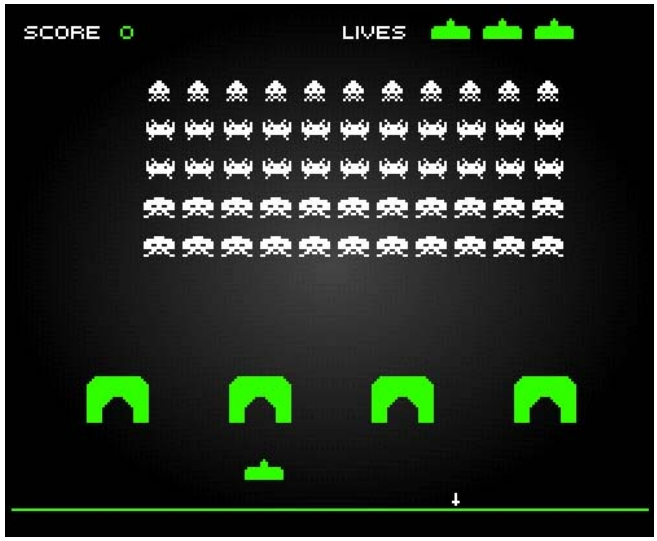


Figura 1. Imagen del juego original

Reglas del juego
El juego sigue las siguientes reglas (ver Figura 1):

1. Varias filas de naves alienígenas o UFOs avanzan hacia la base defensora, con movimientos oscilatorios de izquierda a derecha, bajando poco a poco.
2. Una nave guardián defiende la base.
3. Las naves invasoras lanzan misiles esporádicamente.
4. La nave guardián lanza disparos de uno en uno.
5. Las fortalezas son destruidas por los disparos de los invasores (y por los propios).
6. Conforme el número de invasores decrece, se mueven más rápidamente,
7. Cada UFO (Unidentified Flying Object) tiene cuatro tipos de puntuaciones (50, 100, 150 o 300 puntos), según su forma.
8. Cuando el marcador alcanza los 1500 puntos, aparece un bonus de láser.
9. El jugador tiene tres vidas.
10. El juego finaliza cuando todos los invasores han sido alcanzados o cuando los invasores llegan a la base.

Además,

El juego no disparará otra bala hasta que la última haya causado impacto. Los UFO también disparan balas, de forma esporádica. Para simplificar el enunciado de la práctica, vamos a asumir que:

- No hay fortalezas.
- Las naves alienígenas tienen todas las mismas formas.
- No llevaremos los puntos,
- No haremos el bonus de láser,
- Sólo habrá una vida.

Elementos obligatorios del juego

A continuación se muestra una propuesta del juego. En él aparecen cuatro clases de elementos (Ver Figura 2):

- naves alienígenas o UFOs, que se mueven de izda. a dcha. y van bajando hacia abajo poco a poco. Esporádicamente lanzan misiles.
- la nave guardián, que responde a los siguientes eventos:
  - Tecla O : se mueve a la izda.
  - Tecla P : se mueve a derecha.
  - Espacio : disparo de láser.
- el láser disparado por la nave guardián (trayectoria ascendente). Cuando el láser de la nave alcanza una nave enemiga, ésta desaparece del juego.
- los misiles disparados por los UFOs (trayectoria descendente). Cuando un misil alcanza a la nave, finaliza el juego.

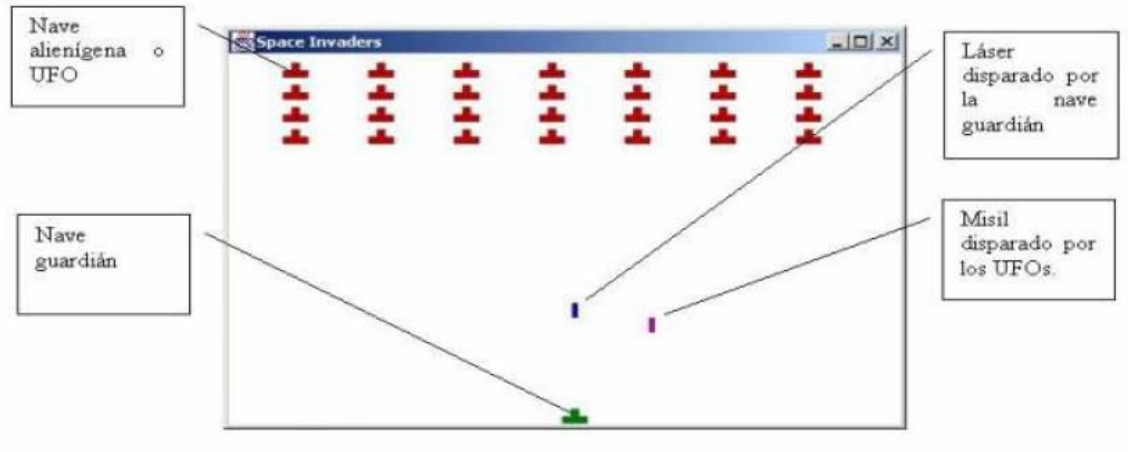


Figura 2. Versión simplificada del juego.

Conceptos Básicos

Describiré brevemente algunos de los conceptos y tecnicismos de programación que se han utilizado durante el desarrollo de la presente práctica.

Sprite

Un *bitmap* (mapa de bits) es la representación de una imagen. Un *sprite* es un *bitmap* dotado de características propias como anchura, altura, desplazamiento, etc. que necesita ser controlado.
Los *sprites* son la representación gráfica de los personajes que aparecen en primer plano. Interactúan entre ellos, el fondo, y algunas veces con los objetos en primer plano. Pueden correr, saltar, volar, nadar, patear, o disparar. Normalmente, un juego proporciona un personaje principal controlado por el jugador, mientras los otros pueden ser monstruos o insectos que persiguen al personaje principal. El objetivo en una situación como ésta sería repeler el *sprite* del personaje principal de los posibles disparos de los *sprites* de los enemigos, evitando que lleguen a colisionar.
El reto principal de la programación de los *sprites* es animarlos. Hay moverlos y que realicen acciones los más suavemente posible. Para comprobar que ocurre en la animación de un *sprite*, nos valemos de una acción de nuestro juego. Cuando se pulsa la tecla <Espacio>, el personaje principal realiza un disparo. Para que esto sea posible, se han creado diferentes *sprites.*

Nuestro juego va a consistir en lo que se ha dado en llamar shooter en el argot de los videojuegos. Para nosotros resulta más familiar decir “marcianitos”. En este tipo de juegos manejamos una nave que tiene que ir destruyendo a todos los enemigos que se pongan en su camino.

Podemos decir que un sprite es un elemento gráfico determinado (una nave, un coche, etc...) que tiene entidad propia y sobre la que podemos definir y modificar ciertos atributos, como la posición en la pantalla, si es o no visible, etc... Un sprite, pues, tiene capacidad de movimiento. Distinguimos dos tipos de movimiento en los sprites: el movimiento externo, es decir, el movimiento del sprite por la pantalla, y el movimiento interno o animación.
Estos son los sprites del juego original:

Utilizaré para la práctica cuatro sprites:

- Nave defensora: 
- Laser de la nave defensora, de color amarillo:
- Alien: 
- Misil del Alien, de color verde: 

Imaginemos ahora que jugamos a un videojuego en el que manejamos a un hombrecillo. Podremos observar cómo mueve las piernas y los brazos según avanza por la pantalla. Éste es el movimiento interno o animación. La siguiente figura muestra la animación del sprite de un gato.

La operación más importante de un sprite es el movimiento por la pantalla. Veamos los métodos que nos permitirán moverlo.

```
public void setX(int x) {
      posx=x;
}

public void setY(int y) {
      posy=y;
}

int getX() {
      return posx;
}

int getY() {
      return posy;
}
```

Los métodos `setX()` y `setY()` actualizan las variables de estado del sprite (`posx`,`posy`). Los métodos `getX()` y `getY()` realizan la operación contraria, es decir, nos devuelve la posición del sprite. Además de la posición del sprite, nos va a interesar en determinadas condiciones conocer el tamaño del mismo.

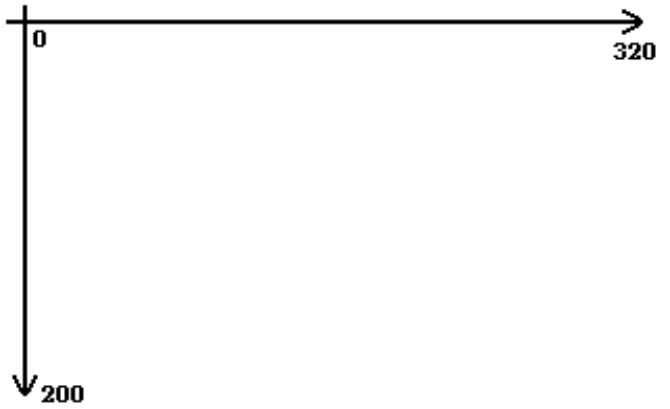
```
int getAncho() {
      return sprites[nframes].getWidth();
}

int getAlto() {
      return sprites[nframes].getHeight();
}
```

Los métodos `getAncho()` y `getAlto()` nos devuelven el ancho y el alto del sprite en píxeles. Para ello recurrimos a los métodos getWidth() y getHeigth() de la clase Image.

```
java.awt
Class Image
java.lang.Object
  └ java.awt.Image
```

Coordenadas

Las coordenadas de la pantalla no son las coordenadas cartesianas. Realmente la coordenada Y es precisamente opuesta porque aumenta a medida que se baja y disminuye a medida que subimos. Éste es un problema que tiene no sólo Java, sino que todos los lenguajes de programación y todas las bibliotecas gráficas tipo *Raster*. Puede deberse a razones históricas por la forma de dibujar los píxeles en pantalla a nivel de hardware. Hay que acostumbrarse, pero las primeras veces puede provocar problemas de razonamiento.
Para posicionar un sprite en la pantalla hay que especificar sus coordenadas. Es como el juego de los barquitos, en el que para identificar un cuadrante hay que indicar una letra para el eje vertical (lo llamaremos eje Y) y un número para el eje horizontal (al que llamaremos eje X). En un ordenador, un punto en la pantalla se representa de forma parecida. La esquina superior izquierda representa el centro de coordenadas. La figura siguiente muestra el eje de coordenadas en una pantalla con una resolución de 320 por 200 píxeles.



Un punto se identifica dando la distancia en el eje X al lateral izquierdo de la pantalla y la distancia en el eje Y a la parte superior de la pantalla. Las distancias se miden en píxeles. Si queremos indicar que un sprite está a 100 píxeles de distancia del eje vertical y 150 del eje horizontal, decimos que está en la coordenada (100,150).

Frame

Mediante este término denominamos cada instante mínimo de tiempo correspondiente al juego, a veces denominado fotograma. En los juegos, en el periodo que dura este tiempo se suelen producir una secuencia de eventos que modifican las distintas variables que lo controlan. Una secuencia común de eventos en un videojuego puede ser la siguiente:

- Recoger las pulsaciones de teclado del jugador y realizar los cálculos necesarios.
- Dibujar el *sprite* de los actores encima de estos *tiles*.
- Mostrar el resultado por pantalla.

Para que estos eventos se produzcan de forma rápida y suave tal que no sean percibidos por el ojo humano, deben producirse con una latencia de 25 veces por segundo, de aquí el término *fps* (*frames per second*).

En lo referente al estado necesitamos algún método para el control de frames, o lo que es lo mismo, de la animación interna del sprite.

Actor

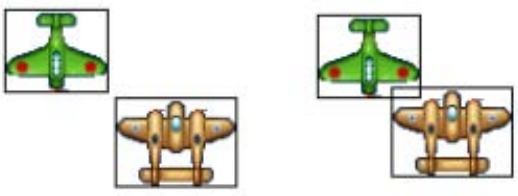
Un actor es una entidad externa al sistema que realiza algún tipo de interacción con el mismo. Se representa mediante una figura humana dibujada con palotes. Esta representación sirve tanto para actores que son personas como para otro tipo de actores (otros sistemas, sensores, etc.).
El personaje principal del juego es un actor, y cada una de sus manifestaciones: desplazarse a la izquierda o derecha, disparar son sus *sprites*.

Para dibujar el sprite, vamos a crear el método draw(). Lo único que hace este método es dibujar el frame actual del sprite en la pantalla.

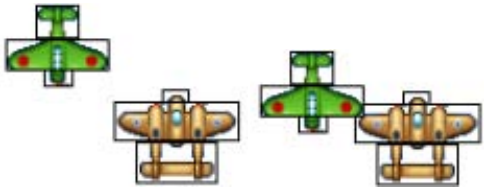
```
public void draw(Graphics g) {
      g.drawImage (sprites[frame], posx, posy,
      Graphics.HCENTER|Graphics.VCENTER);
}
```

Colisiones

Otra característica muy interesante de los sprites es que nos permiten detectar colisiones entre ellos. Esta capacidad es realmente interesante si queremos conocer cuando nuestra nave ha chocado con un enemigo o con uno de sus misiles, para ello nos resta dotar a nuestra librería con la capacidad de detectar colisiones entre sprites. Imaginemos dos sprites, un avión y un disparo enemigo. En cada vuelta del gameloop tendremos que comprobar si el disparo ha colisionado con nuestro avión. Podríamos considerar que dos sprites colisionan cuando alguno de sus píxeles visibles (es decir, no transparentes) toca con un píxel cualquiera del otro sprite. Esto es cierto al 100%, sin embargo, la única forma de hacerlo es comprobando uno por uno los píxeles de ambos sprites. Evidentemente esto requiere un gran tiempo de computación, y es inviable en la práctica. En nuestra librería hemos asumido que la parte visible de nuestro sprite coincide más o menos con las dimensiones de la superficie que lo contiene. Si aceptamos esto, y teniendo en cuenta que una superficie tiene forma cuadrangular, la detección de una colisión entre dos sprites se simplifica bastante. Sólo hemos de detectar el caso en el que dos cuadrados se solapen.



En la primera figura no existe colisión, ya que no se solapan las superficies, las superficies están representadas por el cuadrado que rodea al gráfico. La segunda figura muestra el principal problema de este método, ya que nuestra librería considerará que ha habido colisión cuando realmente no ha sido así. A pesar de este pequeño inconveniente, este método de detección de colisiones es el más rápido. Es importante que la superficie tenga el tamaño justo para albergar el gráfico. Se trata de comprobar si el cuadrado que contiene el primer sprite, se solapa con el cuadrado que contiene al segundo. Hay otros métodos más precisos que nos permiten detectar colisiones. Consiste en dividir el sprite en pequeñas superficies rectangulares tal y como muestra la próxima figura.



Se puede observar la mayor precisión de este método. El proceso de detección consiste en comprobar si hay colisión de alguno de los cuadros del primer sprite con alguno de los cuadrados del segundo utilizando la misma comprobación que hemos utilizado en el primer método para detectar si se solapan dos rectángulos. A continuación se muestra la forma de realizarlo:

```
/** Rectángulo usado por la entidad durante la colisión */
private Rectangle yo = new Rectangle();
/** Rectángulo usado por otra entidad durante la colisión */
private Rectangle el = new Rectangle();
```

```
java.awt
Class Rectangle
java.lang.Object
  └ java.awt.geom.RectangularShape
      └ java.awt.geom.Rectangle2D
          └ java.awt.Rectangle
```

Este es el aspecto que tienen los métodos de detección de colisiones en la clase movimiento:

```
public boolean colisionaCon(movimiento other) {
      yo.setBounds((int) x,(int) y,sprite.getAncho(),sprite.getAlto());
      el.setBounds((int) other.x,(int)other.y,other.sprite.getAncho(),
      other.sprite.getAlto());
      return yo.intersects(el);
}


public abstract void colisionaCon (movimiento other);
```

La clase Canvas

```
java.awt

Class Canvas
java.lang.Object
  └ java.awt.Component
      └ java.awt.Canvas
```

Canvas representa un área de trabajo para la pantalla en la que podemos dibujar los eventos que nos interesen.
Si tenemos un applet que trabaja con imágenes directamente, ya sea un applet gráfico o de dibujo, los lienzos o zonas de dibujo (Canvas) resultan muy útiles.
Los Canvas son un componente básico que captura eventos de exposición, de ratón y demás eventos relacionados. La clase Canvas no responde a estos eventos, pero se puede extender esa clase base creando subclases en las que controlemos esos eventos.

Al permitir saltarse el manejo normal de eventos, y junto con los métodos de representación gráfica, los canvas simplifican la producción de applets que necesitan una única funcionalidad para distintas áreas.

El Paquete AWT

Contiene las clases para crear interfaces de usuarios y dibujar gráficos e imágenes. **AWT** es el acrónimo del X Window Toolkit para Java, donde X puede ser cualquier cosa: Abstract, Alternative, Awkward, parece que Sun se decanta por **Abstracto**. Se trata de una biblioteca de clases Java para el desarrollo de Interfaces de Usuario Gráficas.

La estructura básica del AWT se basa en componentes y contenedores. Estos últimos contienen Componentes posicionados a su respecto y son Componentes a su vez, de forma que los eventos pueden tratarse tanto en Contenedores como en Componentes, corriendo por cuenta del programador (todavía no hay herramientas de composición visual) el encaje de todas las piezas, así como la seguridad de tratamiento de los eventos adecuados.

AWT permite hacer interfaces gráficas mediante artefactos de interacción con el usuario, como botones, menús, texto, botones para selección, barras de deslizamiento, ventanas de diálogo, selectores de archivos, etc. Y por supuesto despliegue gráfico general. Estos artefactos de interacción se denominan *widgets.*

Artefactos complejos

Los artefactos que hemos visto hasta el momento son simples porque la plataforma sabe como dibujarlos y qué tamaño deben ocupar. Es decir la aplicación no necesita especificar tamaño ni forma.
El canvas es un artefacto complicado porque AWT no conoce en absoluto el aspecto que debe tener en la ventana. Por lo tanto es la aplicación la que se debe preocupar de especificar qué tamaño debe tener y cómo se dibuja. Lo anterior se logra creando una nueva clase derivada a partir de Canvas. En ella se redefinen algunos métodos que realizan las funciones que un canvas normal no sabe hacer.

Threads

Otro detalle importante para el desarrollo de esta práctica es el uso de threads o hilos. Un hilo es un proceso que se está ejecutando en un momento determinado en nuestro sistema operativo, como cualquier otra tarea, esto se realiza directamente en el procesador. Existen los llamados 'demonios' que son los procesos que define el sistema en sí para poder funcionar y otros que llamaremos los hilos definidos por el usuario o por el programador, estos últimos son procesos a los que el programador define un comportamiento e inicia en un momento específico.

En Java, el proceso que siempre se ejecuta es el llamado ***main*** que es a partir del cual se inicia prácticamente todo el comportamiento de nuestra aplicación, y en ocasiones a la aplicación le basta con este solo proceso para funcionar de manera adecuada, sin embargo, existen algunas aplicaciones que requieren más de un proceso (o hilo) ejecutándose al mismo tiempo (multithreading), por ejemplo, se tiene una aplicación de una tienda departamental de la cual se actualizan los precios y mercancías varias veces al día a través de la red, se verifican los nuevos descuentos y demás pero que a su vez es la encargada de registrar las compras y todos movimientos que se realice con la mercancía dentro de la tienda, si se decide que dicha aplicación trabajará de la manera

simple y con un solo proceso (o hilo), el trabajo de la actualización de precios y mercancías debe de finalizar antes de que alguien pueda hacer algún movimiento con un producto dentro de la tienda, o viceversa, ya que la aplicación no es capaz de mantener el proceso de actualización en segundo plano mientras se registra un movimiento. Si se toma este modelo mono-hilo el tiempo y dinero que se perderá dentro de la tienda será muchísimo mayor comparando con un modelo multi-hilo. En un modelo multi-hilo se pueden realizar todas las actualizaciones en segundo plano mientras se registra una o más ventas o movimientos, cada proceso independiente del otro viviendo o ejecutándose al mismo tiempo dentro de la misma aplicación.

Al hablar de multi-hilo pudiera parecer que necesitamos más de un procesador para realizar dichas tareas pero no es así, el procesador mismo junto con la máquina virtual de Java gestionan el flujo de trabajo y dan la impresión de que se puede ejecutar más de algún proceso al mismo tiempo (aunque en términos estrictos eso no es posible), de cualquier manera no ahondaré en el funcionamiento del procesador, basta con entender que en Java, 2 o más procesos pueden ejecutarse al mismo tiempo dentro de una misma aplicación y para ello son necesarios los Threads o hilos.

En Java un hilo o Thread puede ser dos cosas:

- Una instancia de la clase **`java.lang.Thread`**
- Un proceso en ejecución

Una instancia de la clase java.lang.Thread, no es más que cualquier otro objeto, con variables y métodos predefinidos. Un proceso en ejecución es un proceso individual que realiza una tarea o trabajo, tiene su propia pila de información independiente a la de la aplicación principal.

Es necesario entender que el comportamiento de los hilos o threads varía de acuerdo a la máquina virtual, incluso el concepto más importante a entender con los hilos en Java es que...

"Cuando se trata de hilos, muy pocas cosas está garantizadas"

...por ello se debe ser cautelosos al momento de interpretar el comportamiento de un hilo.

Crear un hilo (Thread):
Un hilo o proceso en Java comienza con una instancia de la clase java.lang.Thread, si analizamos la estructura de dicha clase podremos encontrar bastantes métodos que nos ayudan a controlar el comportamiento de los hilos, desde crear un hilo, iniciarlo, pausar su ejecución, etc.

Métodos de uso común:
start(): usado para iniciar el cuerpo del thread definido por el método run().
sleep(): pone a dormir una thread por un tiempo mínimo especificado.
join(): usado para esperar por el término de la thread, por ejemplo por término de run().
yield(): Mueve el thread desde el estado de ejecución al final de la cola de procesos en espera por la CPU.

La acción sucede dentro del método ***`run()`***, digamos que el código que se encuentra dentro de dicho método es el trabajo por hacer, por lo tanto, si queremos realizar

diversas operaciones cada una simultánea pero de manera independiente, tendremos varias clases, cada una con su respectivo método run(). Dentro del método run() puede haber llamadas a otros métodos como en cualquier otro método común, pero la pila de ejecución del nuevo proceso siempre comenzará a partir de la llamada al método run().

Definir un nuevo hilo:

Para definir e instanciar un nuevo Thread (hilo, proceso) existen 2 formas:

- Extendiendo (o heredando) a la clase **java.lang.Thread**
- Implementando la interfaz **Runnable**

Normalmente en el trabajo diario es más recomendable el implementar la interfaz Runnable, en lugar de extender la clase java.lang.Thread debido a que una clase solamente puede heredar o extender otra sola clase pero puede implementar muchas interfaces. Si extendemos de la clase java.lang.Thread no podremos extender o heredar ninguna otra, pero si implementamos la interfaz Runnable, podremos heredar cualquier otra clase e implementar muchas otras interfaces sin perder el comportamiento de la nuestra.

En cualquiera de los dos casos, ya sea heredando de java.lang.Thread o implementando Runnable, al definir e instanciar un nuevo hilo, necesitaremos de redefinir el método run(), veamos cómo hacerlo.

Extendiendo **java.lang.Thread:**

Hereda a la clase java.lang.Thread por medio de la palabra `'extends'`.
Redefine el método ***`run()`*** .

```
class MiHilo extends Thread{
      public void run(){
      System.out.println("Trabajo por hacer dentro de MiHilo");
      }
}
```

Nuevamente, esto no es recomendable ya que al heredar la clase Thread, no se puede heredar nada más.

Hay que tener en cuenta que se puede sobrecargar el método `run()` sin ningún problema como se muestra a continuación...

```
class MiHilo extends Thread{
      public void run(){
      System.out.println("Trabajo por hacer dentro de MiHilo");
      }
      public void run(String s){
      System.out.println("La cadena ingresada es " + s);
      }
}
```

...sin embargo, al realizar esto, no se utiliza el nuevo método `public void run (String s)` en un proceso separado, es un simple método común y corriente como cualquier otro que tienes que mandar llamar de manera independiente ya que los hilos trabajan con un método `run()` sin argumentos.

Implementando la interfaz Runnable:

- Implementa a la interfaz Runnable por medio de la palabra **'implements'**.
- Redefine el método ***run()*** .

```
class MiHilo implements Runnable{
      public void run(){
      System.out.println("Trabajo por hacer dentro de MiHilo");
      }
}
```

Independientemente de cómo definas tu hilo (por medio de extends Thread o implements Runnable), el hilo tendrá el mismo comportamiento.

Instanciando un hilo o Thread:
Debemos recordar que cada hilo de ejecución es una instancia de la clase Thread, independientemente de si tu método run() está dentro de una subclase de Thread o en una implementación de la interfaz Runnable, se necesita un objeto tipo Thread para realizar el trabajo.

Si has extendido la clase Thread, el instanciar el hilo es realmente simple:

```
MiHilo h = new MiHilo();
```

Si se implementa la interfaz Runnable, es un poquito más complicado, pero solo un poco:

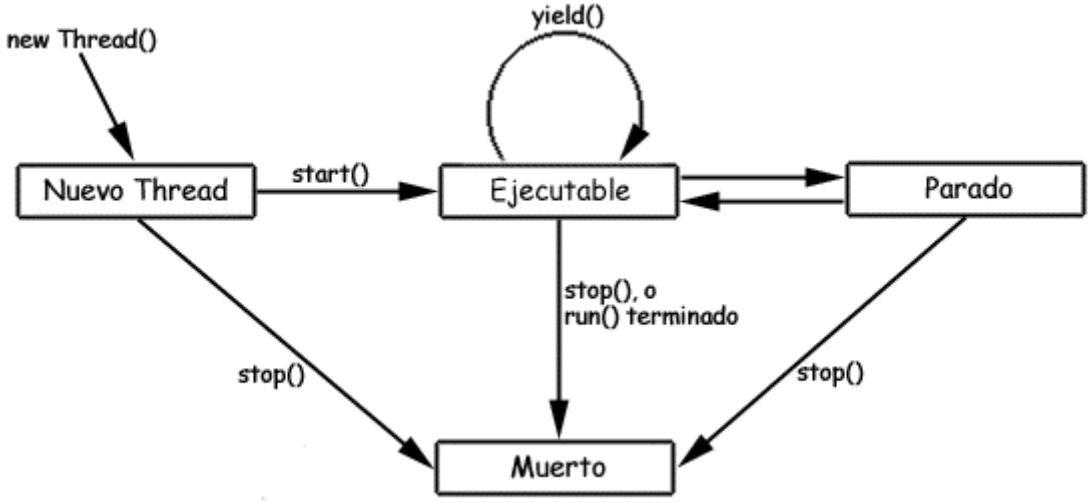
```
MiHilo h = new MiHilo();
//Pasas tu implementación de Runnable al nuevo Thread
Thread t = new Thread(h);
```

Puedes pasar una misma instancia de una clase que implementa Runnable a diversos objetos de tipo Thread...

```
public class PruebaHilos{
      public static void main (String[] args){
      MiHilo h = new MiHilo();
      Thread t1 = new Thread(h);
      Thread t2 = new Thread(h);
      Thread t3 = new Thread(h);
      }
}
```

El pasar un solo objeto tipo MiHilo a varios objetos tipo Thread significa que el trabajo que se encuentra dentro del método run() de MiHilo se realizará en diversas ocasiones, o lo que es lo mismo, varios hilos realizarán el mismo trabajo.

Ciclo de vida de un thread



Fase de Análisis

El problema fundamental con el que se enfrenta un analista de sistemas es interpretar las necesidades del usuario y las aspiraciones del producto a desarrollar, y transformar la descripción informal disponible en un conjunto de hechos, es decir, lo que se conoce como la especificación de requisitos de la aplicación. Para ello se desarrolló un Lenguaje de Modelado, como UML, de conocida eficacia en el análisis de proyectos en cualquier campo y no sólo en el ámbito de la informática. Aunque UML dispone de un número amplio de diagramas, que juntos representan la arquitectura del proyecto, me centraré en los siguientes:

- El Diagrama de Caos de Uso.
- El Diagrama de Transición de Estados.
- El Diagrama de Clases.

En estos diagramas se muestran la vista estática y dinámica de la aplicación y ayudan a modelar y organizar el comportamiento del sistema y a identificar procesos reutilizables dentro de éste. Comenzaré haciendo un análisis con el método de Abbott para definir las clases, atributos y métodos.

Análisis del juego con el Método de Abbott

La idea es identificar en el texto de la descripción aquellas palabras o términos que puedan corresponder a elementos significativos del diseño: tipos de datos, atributos y operaciones, fundamentalmente. Los tipos de datos aparecerán como sustantivos genéricos, los atributos como sustantivos, y las operaciones como verbos o como nombres de acciones. Algunos adjetivos pueden sugerir valores de los atributos.
El siguiente paso establecer dos listas, una de nombres y otra de verbos y operaciones. Después se reorganizan dichas listas extrayendo los posibles tipos de datos, y asociándoles atributos y operaciones.

Nombre	Adjetivos	Verbos
Aliens	Forma	Disparan, mueven, colisionar
Nave Guardian	Forma	Disparan, mueven, colisionar
Misil	Esporádicamente	Mueve, colisionar
Láser	De uno en uno	Mueve, colisionar
Invader		

Técnicas de Diseño Orientadas a Objetos

El diseño orientado a objetos es esencialmente igual al diseño basado en abstracciones, pero añadiendo las características de herencia y polimorfismo.
La idea global de las técnicas de diseño orientadas a objetos es que en la descomposición modular del sistema cada módulo contenga la descripción de una clase de objetos o de varias clases relacionadas entre sí.

Diseño orientado a objetos

La técnica general de diseño se basa en los siguientes pasos:

- ESTUDIAR Y COMPRENDER EL PROBLEMA
- DESARROLLAR UNA POSIBLE SOLUCIÓN
- IDENTIFICAR LAS CLASES Y OBJETOS
- IDENTIFICAR LAS OPERACIONES SOBRE LOS OBJETOS
- APLICAR LA HERENCIA
- DESCRIBIR LAS OPERACIONES
- ESTABLECER LA ESTRUCTURA MODULAR

Diagrama de Transición de Estados. DTE:

Comenzamos el juego con la pantalla inicial y esperamos a pulsar una tecla tal como nos pide el programa, una vez comenzado tenemos una matriz de aliens en la aparte superior de la ventana que se mueven de derecha a izquierda y van descendiendo cada vez que tocan el extremo de la ventana, también nos disparan esporádicamente, por otro lado la nave guardian debe esquivar los disparos y eliminar todos los aliens para alcanzar el objetivo final, a medida que vamos eliminando aliens su velocidad se incrementa en un 3%. Si los aliens nos alcanzan con un misil o llegan a descender hasta el nivel de la nave defensora entonces finaliza el juego y perdemos por derrota de los aliens.

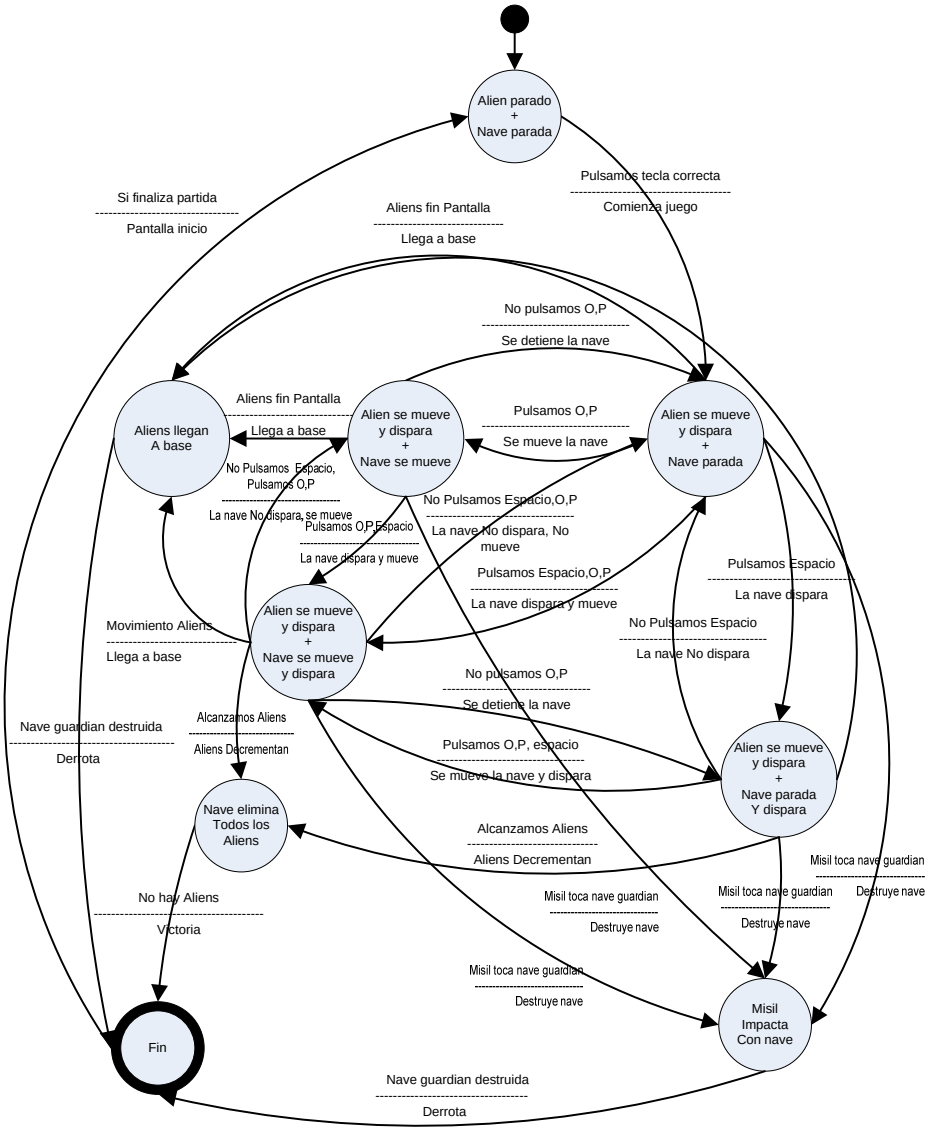


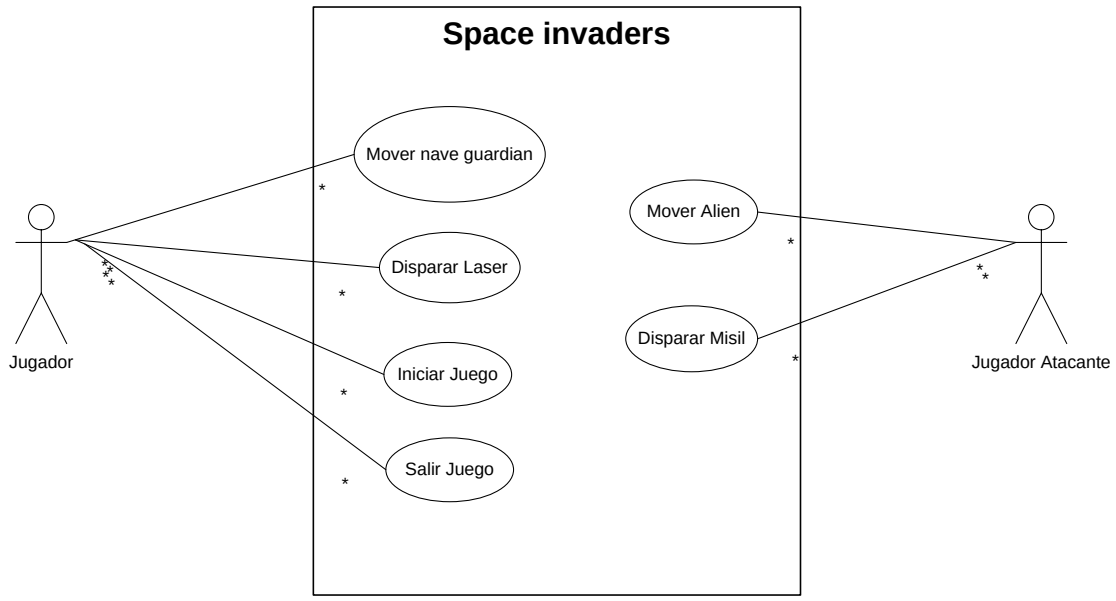
Diagrama de Casos de Uso

Un caso de uso es una descripción de la secuencia de interacciones que se producen entre un actor y el sistema, cuando el actor usa el sistema para llevar a cabo una tarea específica. Expresa una unidad coherente de funcionalidad, y se representa en el Diagrama de Casos de Uso mediante una elipse con el nombre del caso de uso en su interior. El nombre del caso de uso debe reflejar la tarea específica que el actor desea llevar a cabo usando el sistema.

En los diagramas de Casos de Uso pueden existir las siguientes relaciones:

Extiende: Cuando un caso de uso especializa a otro extendiendo su funcionalidad.
Usa: Cuando un caso de uso utiliza a otro. Se representan como una línea que une a los dos casos de uso relacionados, con una flecha en forma de triángulo y con una etiqueta *<<extends>>* o *<<uses>>* según el tipo de relación.

En el diagrama de casos de uso se representa también el sistema como una caja rectangular con el nombre en su interior. Los casos de usos están en el interior de la caja del sistema, y los actores fuera, estando unido cada actor a los casos de uso en los que participe mediante una línea.



Descripción de los Casos de Uso en Formato Expandido

Caso de Uso:	Mover la nave Guardián
Actor:	Jugador
Propósito:	Realizar un movimiento con la nave Guardián
CURSO DE EVENTOS	
Curso Normal	**Alternativas**
Pulsar O: La nave guardián se desplaza hacia la izquierda.	Cuando llega al extremo lateral izquierdo se para.
Pulsar P: La nave guardián se desplaza hacia la derecha.	Cuando llega al extremo lateral derecho se para.
Otras teclas: Las demás teclas no están definidas en este juego para realizar movimientos.	Acción predeterminada para cada tecla.

Caso de Uso:	Disparar Láser
Actor:	Jugador
Propósito:	Realizar un disparo de Láser con la nave Guardián
CURSO DE EVENTOS	
Curso Normal	**Alternativas**
Pulsar Espacio: La nave Guardián hace un disparo de Láser	Si colisiona con un alien, se eliminan las dos entidades, si no colisiona se retira el disparo Láser una vez alcanza el extremo superior de la ventana
Otras teclas: Las demás teclas no están definidas en este juego para realizar movimientos.	Acción predeterminada para cada tecla.

Caso de Uso:	Iniciar Juego
Actor:	Jugador
Propósito:	Comenzar a jugar
CURSO DE EVENTOS	
Curso Normal	**Alternativas**
Pulsar Tecla: Comienza la partida. Cualquier tecla es válida.	No debemos pulsar escape ya que abortaremos la partida.

Caso de Uso:	Salir del juego
Actor:	Jugador
Propósito:	Abortar el curso normal de la partida
CURSO DE EVENTOS	
Curso Normal	**Alternativas**
Pulsar Esc: Se aborta la partida en curso.	Cualquier otra tecla no aborta el juego.

Caso de Uso:	Mover Aliens
Actor:	Jugador Atacante
Propósito:	Movimiento de los aliens por la ventana
CURSO DE EVENTOS	
Curso Normal	**Alternativas**
La computadora se encarga de gestionar esta acción según la programación del juego.	Moverse de forma anormal o en contra de lo establecido en la práctica.

Caso de Uso:	Disparar Misil
Actor:	Jugador Atacante
Propósito:	Realizar esporadicamente disparos de Misil los Aliens
CURSO DE EVENTOS	
Curso Normal	**Alternativas**
La computadora se encarga de gestionar esta acción según la programación del juego.	Realizar los disparos de forma anormal o en contra de lo establecido en la práctica.

Fase de Diseño

En esta fase, nos centraremos en obtener un diseño que se adapte a las necesidades de la aplicación y una definición de las clases, así como su respectivo diagrama que nos sirva para ver la colaboración e interacción de éstas.

Nos centraremos en dos aspectos principales:

- Obtención de un diseño de la interfaz de usuario que se adapte a nuestras necesidades.
- Definición de las clases a utilizar y establecer diagramas entre éstas, que nos indiquen la estructura a seguir en la Fase de Implementación.

Diseño de la Interfaz de Usuario de la Aplicación

La interfaz de usuario debe adaptarse a nuestras necesidades más imperiosas, una de ellas es hacer que el manejo de la aplicación sea intuitivo y agradable a la vista, cito a continuación la propuesta de esta práctica:

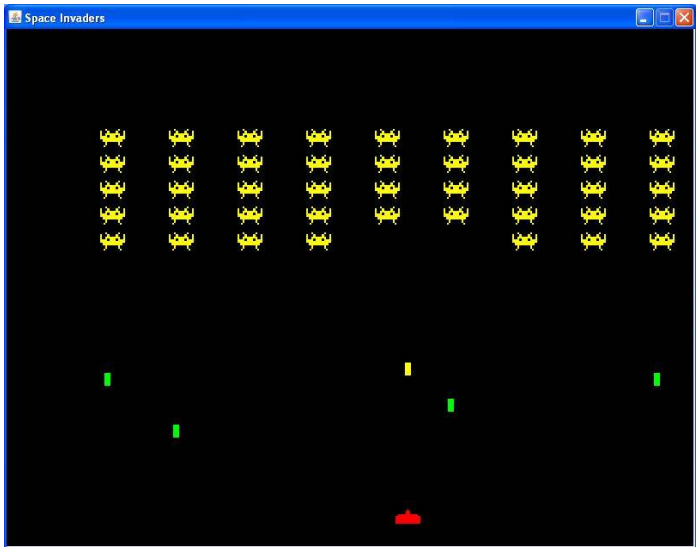
- Varias filas de naves alienígenas o UFOs avanzan hacia la base defensora, con movimientos oscilatorios de izquierda a derecha, bajando poco a poco.
- Una nave guardián defiende la base.
- Las naves invasoras lanzan misiles esporádicamente de color verde.
- La nave guardián dispara un láser de uno en uno de color amarillo.
- Conforme el número de invasores decrece, se mueven más rápidamente,
- El juego finaliza cuando todos los invasores han sido alcanzados o cuando los invasores llegan a la base.

Además,

El juego no disparará otra bala hasta que la última haya causado impacto. Los UFO también disparan balas, de forma esporádica. Para simplificar el enunciado de la práctica, vamos a asumir que:

- No hay fortalezas.
- Las naves alienígenas tienen todas las mismas formas.
- No llevaremos los puntos,
- No haremos el bonus de láser,
- Sólo habrá una vida.
- la nave guardián, que responde a los siguientes eventos:
    - Tecla O : se mueve a la izda.
    - Tecla P : se mueve a derecha.
    - Espacio : disparo de láser.
- el láser disparado por la nave guardián (trayectoria ascendente). Cuando el láser de la nave alcanza una nave enemiga, ésta desaparece del juego.

La interfaz de usuario va a tener la siguiente forma:



Package javax.swing

Proporciona un conjunto de componentes ligeros, la interfaz es una aplicación *Swing* que presenta su GUI (*Graphical User Interface*) principal dentro de un *JFrame*. Un *JFrame* es un contenedor *Swing* de alto nivel que proporciona ventanas para *applets* y aplicaciones. Un *JFrame* tiene decoraciones características como un borde, un título, y botones para cerrar y minimizar la ventana. Un ejemplo simple crea un *JFrame*, añade componentes al panel de contenido, y quizás añade una barra de menú. Sin embargo, a través de su panel raíz, *JFrame* proporciona soporte para una mayor personalización. Presento a continuación la herencia de JFrame.

```
Class JFrame
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └java.awt.Window
              └java.awt.Frame
                  └javax.swing.JFrame
```

Diseño y Diagrama de Clases

El diagrama de clases describe las clases y objetos que debe tener el sistema y las diversas relaciones de carácter estático que existen entre estas. Además, a una mayor profundidad en éstas, se mostrarán los atributos y operaciones de las distintas clases y las restricciones de visibilidad a la que se verán sujetas.

Una vez estudiado los requisitos funcionales del sistema y elegida la interfaz de usuario, estamos en disposición de comenzar a desarrollar las diferentes clases que compondrán el modelo de análisis, para tener en cuenta las necesidades y funcionalidades del software. Aún así, puede que sea necesario realizar algunos cambios en el diagrama de clases inicial, modificando las existentes o agregando otras nuevas. Este hecho se hará visible sobre todo en la Fase de Implementación, donde tendremos que tomar decisiones que afecten al diseño de las clases propuesto. El objetivo de todo buen analista es determinar las clases a implementar en el estudio de las especificaciones del

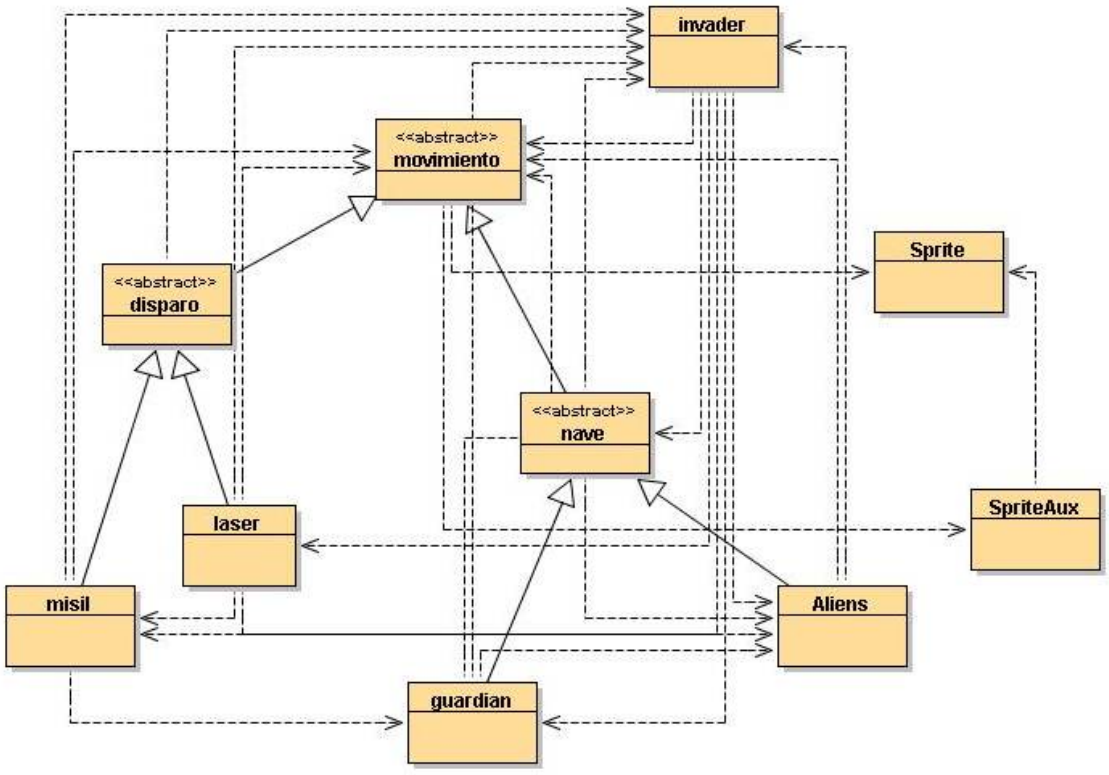
cliente y que estas no necesiten ser modificadas en ninguna otra parte del proyecto, aunque esta tarea resulta bastante difícil y requiere de una amplia experiencia.
Vamos a ver la descripción de las clases más importantes y su relación con otras clases y objetos a través de diagramas para posteriormente ir profundizando en las mismas, incluyendo los métodos y propiedades más importantes.

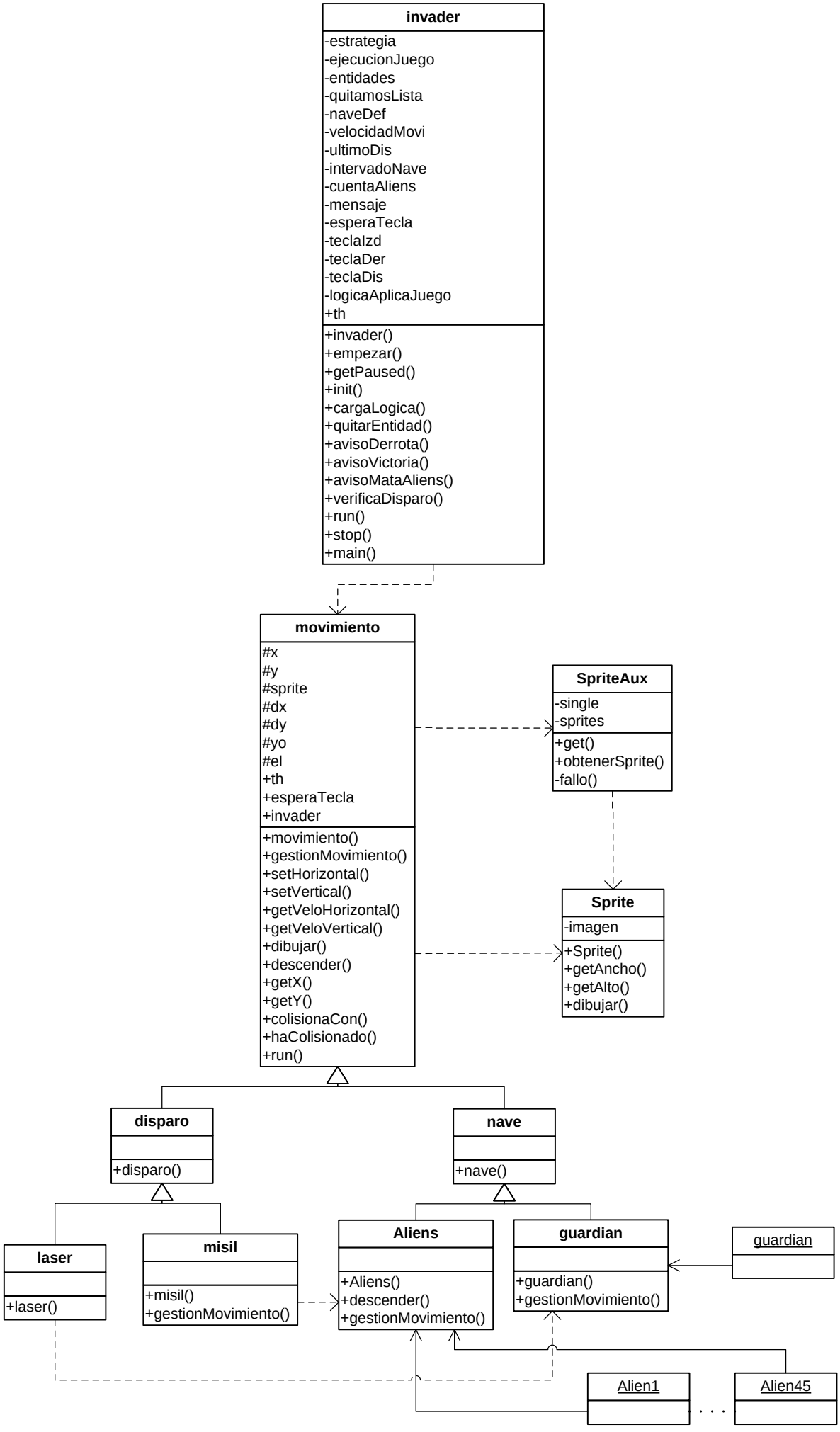
Diagrama de Clases

En donde:

- **Parte superior**: Contiene el nombre de la clase.
- **Parte intermedia**: Contiene los atributos (o variables de instancia) que caracterizan la clase.
- **Parte inferior**: Contiene los métodos u operaciones, los cuales son la forma como interactúa el objeto con su entorno.

Diagrama de clases de la aplicación:

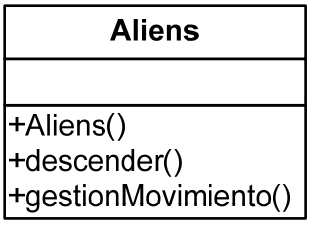



invader
-estrategia
-ejecucionJuego
-entidades
-quitamosLista
-naveDef
-velocidadMovi
-ultimoDis
-intervadoNave
-cuentaAliens
-mensaje
-esperaTecla
-teclaIzd
-teclaDer
-teclaDis
-logicaAplicaJuego
+th
+invader()
+empezar()
+getPaused()
+init()
+cargaLogica()
+quitarEntidad()
+avisoDerrota()
+avisoVictoria()
+avisoMataAliens()
+verificaDisparo()
+run()
+stop()
+main()
movimiento
#x
#y
#sprite
#dx
#dy
#yo
#el
+th
+esperaTecla
+invader
+movimiento()
+gestionMovimiento()
+setHorizontal()
+setVertical()
+getVeloHorizontal()
+getVeloVertical()
+dibujar()
+descender()
+getX()
+getY()
+colisionaCon()
+haColisionado()
+run()
SpriteAux
-single
-sprites
+get()
+obtenerSprite()
-fallo()
Sprite
-imagen
+Sprite()
+getAncho()
+getAlto()
+dibujar()
disparo
+disparo()
nave
+nave()
laser
+laser()
misil
+misil()
+gestionMovimiento()
Aliens
+Aliens()
+descender()
+gestionMovimiento()
guardian
+guardian()
+gestionMovimiento()
guardian
Alien1
Alien45

Definición de las Clases

Una vez vista la relación entre las clases, paso a describir cada una de ellas mostrando los atributos y métodos más importantes.

La clase Aliens




```
Class Aliens
java.lang.Object

      nave

          Aliens
```

```
public class Aliens extends nave
/*************************************************************
/* Esta clase representa los alien.
/* @autor Antonio Rivero
/************************************************************* /
```

Field Summary

double	**velocidad**

Constructor Summary

Constructor Summary
Aliens(invader juego, java.lang.String ref, int x, int y) Descrip: Constructor de clase.

Method Summary	
void	**descender** () Descrip: Actualización de los alien
void	**gaestionMovimiento**(long delta) Descrip: Controla el movimiento de los aliens, evita que el alien se desplace fuera de los lados de la pantalla

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Aliens
public **Aliens**(invader juego, java.lang.String ref, int x, int y)
Descrip: Constructor de clase. Crea un nuevo alien

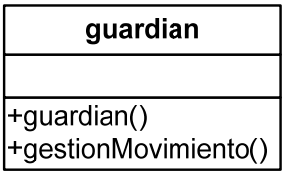
Method Detail

descend
public void **descender**()
Descrip: Actualización de los alien

---

gestionMovimiento
public void **gestionMovimiento**(long delta)
Descrip: Controla el movimiento de los aliens, evita que el alien se desplace fuera de los lados de la pantalla

La Clase guardian



Class guardian
java.lang.Object

nave

guardian

---

public class **guardian** extends nave
/***
/* Esta clase representa la nave guardian
/* @autor Antonio Rivero
/*** /

Constructor Summary

guardian(invader juego, java.lang.String ref, int x, int y)
Descrip: Constructor de clase.

Method Summary

void	gaestionMovimiento (long delta) Descrip: Controla el movimiento de la nave, evita que la nave se desplace fuera de los lados de la pantalla

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

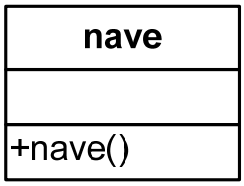
Constructor Detail

guardian
public **guardian**(invader juego, java.lang.String ref, int x,int y)
Descrip: Constructor de clase. Crear una nueva entidad para representar la nave defensora

Method Detail

gestionMovimiento
public void **gestionMovimiento** (long delta)
Descrip: Controla el movimiento de la nave, evita que la nave se desplace fuera de los lados de la pantalla

La Clase nave



Class nave
java.lang.Object

movimiento

nave

```
public abstract class nave extends movimiento
/*************************************************************
/* Esta clase representa nave
/* @autor Antonio Rivero
/************************************************************* /
```

Constructor Summary

nave(invader juego, java.lang.String ref, int x, int y)
Descrip: Constructor de clase.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

nave
public **nave**(invader juego, java.lang.String ref, int x, int y)
Descrip: Constructor de clase. Crear una nueva entidad para representar la nave defensora

La Clase misil


```
Class misil
java.lang.Object

      movimiento

          misil
```

```
public class misil extends movimiento
/*********************************************************
/* Esta clase representa un disparo del alien
/* @autor Antonio Rivero
/********************************************************* /
```

Constructor Summary
misil(invader juego, java.lang.String sprite, int x, int y) Descrip: Constructor de clase.

Method Summary	
void	gaestionMovimiento (long delta) Descrip: Controla el movimiento del misil, evita que se sature el juego de misiles que no se retiran.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

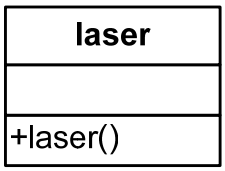
Constructor Detail

```
misil
public misil(invader juego,java.lang.String sprite, int x, int y)
Descrip: Constructor de clase. Crea un nuevo disparo del alien
```

Method Detail

```
gestionMovimiento
public void gestionMovimiento (long delta)
Descrip: Controla el movimiento del misil, evita que se sature el
juego de misiles que no se retiran.
```

La Clase láser




```
Class laser
java.lang.Object

      movimiento

          laser
```

```
public class laser extends movimiento
/*********************************************************
/* Esta clase representa un laser de la nave defensora
/* @autor Antonio Rivero
/********************************************************* /
```

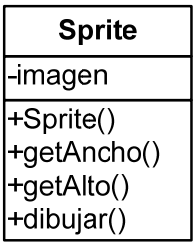
Constructor Summary
laser(invader juego, java.lang.String sprite, int x, int y) Descrip: Constructor de clase.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

```
laser
public laser(invader juego,java.lang.String sprite, int x, int y)
Descrip: Constructor de clase. Crea un nuevo laser de la nave
defensora
```

La Clase Sprite



Class Sprite
java.lang.Object

Sprite

---

public class **Sprite** extends java.lang.Object
Descrip: Clase Sprite. Un sprite se visualiza en pantalla. Un sprite no da información, únicamente la imagen y no la localización. Esto nos permite usar un sprite simple en diferentes partes del juego sin tener que almacenar muchas copias de esa imagen

---

Constructor Summary
Sprite(java.awt.Image imagen) Descrip: Constructor de la clase Se crea un nuevo sprite basado en una imagen

Method Summary	
void	**dibujar**(java.awt.Graphics g, int x, int y) Descrip: dibuja el frame en pantalla
int	**getAlto**() Descrip: getHeight determina la altura del sprite Si no se conoce devuelve -1
int	**getAncho**() Descrip: getWidth determina anchura del sprite Si no se conoce devuelve -1

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Sprite
public **Sprite**(java.awt.Image imagen)
Descrip: Constructor de la clase Se crea un nuevo sprite basado en una imagen

Method Detail

dibujar
public void **dibujar**(java.awt.Graphics g, int x, int y)
Descrip: dibuja el frame en pantalla

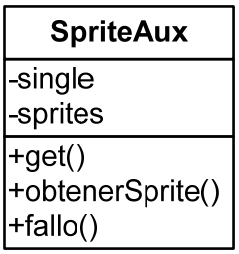
---

getAlto
public int **getAlto**()
Descrip: getHeight determina la altura del sprite Si no se conoce devuelve -1

---

getAncho
public int **getAncho**()
Descrip: getWidth determina anchura del sprite Si no se conoce devuelve -1

La Clase SpriteAux



Class SpriteAux
java.lang.Object

SpriteAux

---

public class **SpriteAux** extends java.lang.Object
Descrip: Clase SpriteAux. Se encarga de los recursos para los Sprites en el juego. Es importante el cómo y dónde utilizamos los recursos Responsable carga y cache de los sprites

---

Constructor Summary
SpriteAux()

Method Summary	
static SpriteAux	**get**() Descrip: Crea una instancia simple de la clase
Sprite	**obtenerSprite**(java.lang.String ref) Descrip: Recupera un sprite

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

SpriteAux
public **SpriteAux**()

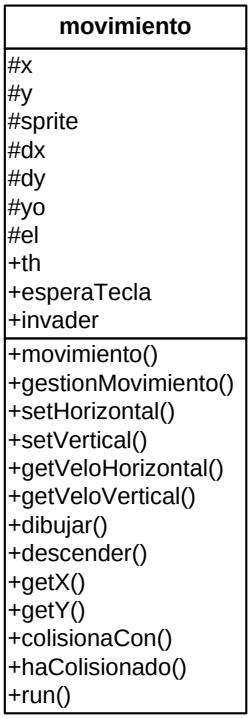
Method Detail

get
public static SpriteAux **get**()
Descrip: Crea una instancia simple de la clase

---

obtenerSprite
public Sprite **obtenerSprite**(java.lang.String ref)
Descrip: Recupera un sprite

La Clase movimiento



Class movimiento
java.lang.Object

movimiento
All Implemented Interfaces:
java.lang.Runnable

---

public abstract class **movimiento** extends java.lang.Objectimplements java.lang.Runnable
/**
/* Esta clase representa algunos elementos que aparecen en el juego, la entidad tiene la responsabilidad de resolver las colisiones y movimientos basados en un conjunto de características definidas en cualquiera de las subclases. Utilizo double para la localización de los pixels para dar más precisión.

Field Summary	
protected double	**dx** Velocidad horizontal actual de la ntidad(pixels/sec)
protected double	**dy** Velocidad vertical actual de la entidad(pixels/sec)
protected Sprite	**sprite** Sprite que representa esta entidad
java.lang.Thread	**th** Declaración del Thread
protected double	**x** Localización actual "x" de esta entidad
protected double	**y** Localización actual "y" de esta entidad

Constructor Summary
movimiento(java.lang.String ref, int x, int y) Descrip: Constructora de la clase, referenciada sobre un sprite y la localización de este.

Method Summary	
boolean	**colisionaCon**(movimiento other) Descrip: Verifica si la entidad ha colisionado con otra
double	**getVeloHorizontal**() Descrip: Devuelve la posición del sprite
double	**getVeloVertical**() Descrip: Devuelve la posición del sprite
void	**dibujar**(java.awt.Graphics g) Descrip: Dibuja los gráficos de esta entidad
int	**getX**() Descrip: Localiza la posición de x
int	**getY**() Descrip: Localiza la posición de y
abstract void	**haColisionado**(movimiento other) Descrip: Aviso a la entidad que ha colisionado con otra
void	**Logica**() Descrip: Trabaja con la lógica asociada a esta entidad Este método se llama frecuentemente
void	**mover**(long delta) Descrip: Solicita a la entidad el movimiento respecto al tiempo trancurrido
void	**setHorizontal**(double dx) Descrip: Crea la posición horizontal del sprite
void	**setVertical**(double dy) Descrip: Crea la posición vertical del sprite
void	**run**()

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail
dx protected double **dx** Velocidad horizontal actual de la entidad(pixels/sec)
dy protected double **dy** Velocidad vertical actual de la entidad(pixels/sec)
sprite protected Sprite **sprite** Sprite que representa esta entidad
th public java.lang.Thread **th** Declaración del Thread

x
protected double **x**
Localización actual "x" de esta entidad

---

y
protected double **y**
Localización actual "y" de esta entidad

Constructor Detail

movimiento
public **movimiento**(java.lang.String ref,int x,int y)
Descrip: Constructora de la clase, referenciada sobre un sprite y la localización de este.

Method Detail

colisionaCon
public boolean **colisionaCon**(movimiento other)
Descrip: Verifica si la entidad ha colisionado con otra

---

getVeloHorizontal
public double **getVeloHorizontal**()
Descrip: Devuelve la posición del sprite

---

getVeloVertical
public double **getVeloVertical**()
Descrip: Devuelve la posición del sprite

---

dibujar
public void **dibujar**(java.awt.Graphics g)
Descrip: Dibuja los gráficos de esta entidad

---

getX
public int **getX**()
Descrip: Localiza la posición de x

---

getY
public int **getY**()
Descrip: Localiza la posición de y

---

haColisionado
public abstract void **haColisionado**(movimiento other)
Descrip: Aviso a la entidad que ha colisionado con otra

Logica
public void **Logica**()
Descrip: Trabaja con la lógica asociada a esta entidad Este método se llama frecuentemente

---

mover
public void **mover**(long delta)
Descrip: Solicita a la entidad el movimiento respecto al tiempo trancurrido

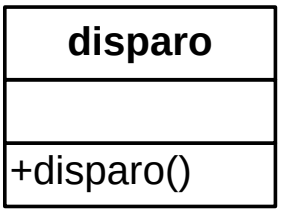
---

setHorizontal
public void **setHorizontal**(double dx)
Descrip: Crea la posición horizontal del sprite

---

setVertical
public void **setVertical**(double dy)
Descrip: Crea la posición vertical del sprite

La Clase disparo




```
Class disparo
java.lang.Object

      movimiento

          disparo
```

```
public abstract class disparo extends movimiento
/*********************************************************
/*Esta clase representa los disparos del juego
/* @autor Antonio Rivero
/********************************************************** /
```

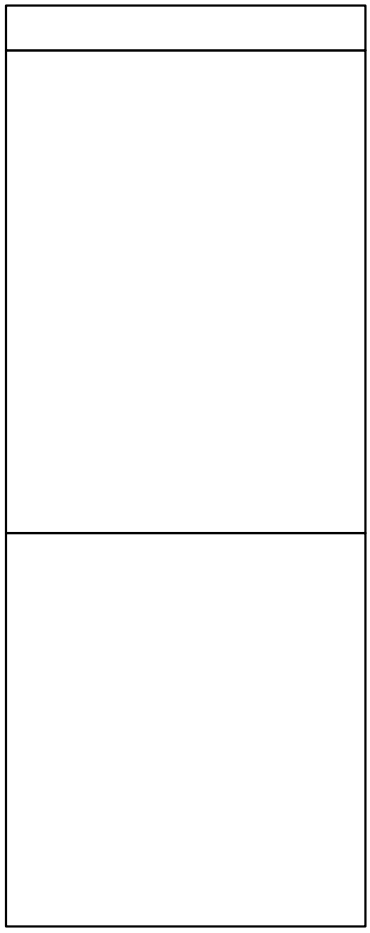
Constructor Summary
disparo(invader juego, java.lang.String sprite, int x, int y) Descrip: Constructor de clase.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

```
disparo
public disparo(invader juego,
               java.lang.String sprite, int x, int y)
Descrip: Constructor de clase. Crea un nuevo disparo
```

La Clase invader



Class invader
java.lang.Object

java.awt.Component

java.awt.Canvas

invader

All Implemented Interfaces:
java.awt.image.ImageObserver, java.awt.MenuContainer,
java.io.Serializable, java.lang.Runnable,
javax.accessibility.Accessible

```
public class invader extends java.awt.Canvasimplements
java.lang.Runnable
/*************************************************************
/* Clase principal del juego, se encarga de coordinar las funciones
/* del programa y coordinar la lógica de juego, la gestión consiste
/* en ejecutar el bucle del juego moviendo y dibujando las entidades
/* en el lugar apropiado. Será informada cuando las entidades dentro
/*del juego detecten  eventos, como deribar un alien, el derribo de la
/*nave defensora  actuando de manera adecuada.
/* @autor Antonio Rivero
/************************************************************* /
```

Nested Class Summary

Nested classes/interfaces inherited from class java.awt.Canvas
java.awt.Canvas.AccessibleAWTCanvas

Nested classes/interfaces inherited from class java.awt.Component
java.awt.Component.AccessibleAWTComponent, java.awt.Component.BltBufferStrategy, java.awt.Component.FlipBufferStrategy

Field Summary	
java.lang.Thread	**th** Declaración del Thread

Fields inherited from class java.awt.Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface java.awt.image.ImageObserver
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary
invader() Descrip: Constructor de clase y puesta en marcha del juego

Method Summary	
void	**avisoDerrota**() Descrip: Aviso que la nave defensora es destruida
void	**avisoMataAliens**() Descrip: Aviso que un alien ha sido destruido
void	**avisoVictoria**() Descrip: Aviso que la nave defensora destruye todos los aliens
void	**cargaLogica**() Descrip: Inicializa la lógica del juego, que comenzará a funcionar cuando lo requiera algún evento
boolean	**getPaused**()
static void	**main**(java.lang.String[] argv) Descrip: Punto de entrada al juego, se crea una instancia de clase que inicializa la ventana de juego y el bucle
void	**quitarEntidad**(movimiento entity) Descrip: Quita la entidad del juego
void	**run**() Descrip: El método run() es el corazón de cualquier "Thread" y contiene las tareas de ejecución, la acción sucede dentro del método run(), además es el bucle del juego, este bucle se ejecuta durante todo el juego siendo reponsable de todas las actividades Calcular el tiempo desde el último bucle Procesar las entradas del jugador Mover todo basado en el tiempo desde el último bucle Dibujar todo lo que hay en pantalla Actualizar buffers

	para que la nueva imagen sea visible Verifica entradas Actualiza los eventos
void	**stop**() Descrip: Método stop()
void	**verificaDisparo**() Descrip: laser del jugador

Methods inherited from class java.awt.Canvas
addNotify, createBufferStrategy, createBufferStrategy, getAccessibleContext, getBufferStrategy, paint, update

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

th
public java.lang.Thread **th**
Declaración del Thread

Constructor Detail

invader
public **invader**()
Descrip: Constructor de clase y puesta en marcha del juego

Method Detail

avisoDerrota
public void **avisoDerrota**()
Descrip: Aviso que la nave defensora es destruida

---

avisoMataAliens
public void **avisoMataAliens**()
Descrip: Aviso que un alien ha sido destruido

---

avisoVictoria
public void **avisoVictoria**()
Descrip: Aviso que la nave defensora destruye todos los aliens

---

cargaLogica
public void **cargaLogica**()
Descrip: Inicializa la lógica del juego, que comenzará a funcionar cuando lo requiera algún evento

---

getPaused
public boolean **getPaused**()

---

main
public static void **main**(java.lang.String[] argv)
Descrip: Punto de entrada al juego, se crea una instancia de clase que inicializa la ventana de juego y el bucle

---

quitarEntidad
public void **quitarEntidad**(movimiento entity)
Descrip: Quita la entidad del juego

---

run
public void **run**()
Descrip: El método run() es el corazón de cualquier "Thread" y contiene las tareas de ejecución, la acción sucede dentro del método run(), además es el bucle del juego, este bucle se ejecuta durante

todo el juego siendo reponsable de todas las actividades Calcular el tiempo desde el último bucle Procesar las entradas del jugador Mover todo basado en el tiempo desde el último bucle Dibujar todo lo que hay en pantalla Actualizar buffers para que la nueva imagen sea visible Verifica entradas Actualiza los eventos

Specified by:

run in interface java.lang.Runnable

---

stop

public void **stop**()

Descrip: Método stop()

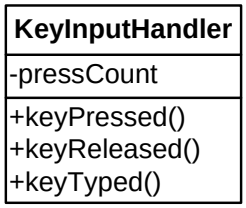
---

verificaDisparo

public void **verificaDisparo**()

Descrip: laser del jugador

La Clase KeyInputHandler



Method Summary	
void	keyPressed (KeyEvent e)
void	keyReleased (KeyEvent e)
void	keyTyped (KeyEvent e)

Fase de Implementación

En esta fase vamos a mostrar como se ha llevado a cabo la idea inicial que hemos ido definiendo en las dos fases anteriores, implantando las clases propuestas

Plataforma de Programación

Se exige utilizar Java como herramienta para la creación de la aplicación y dentro de todo el campo de opciones utilizamos *J2SE*, que se conoce como la parte básica y genérica dentro del ámbito de trabajo de Java. Hay que tener en cuenta varios factores como:

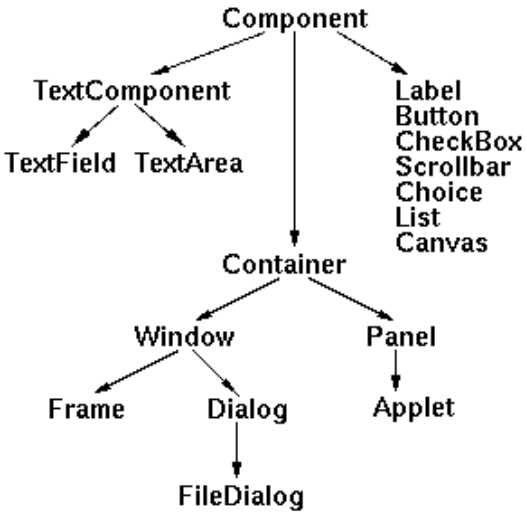
- Los mecanismos de la programación orientada a objetos, su utilización para la resolución de problemas y las características principales que nos ofrece el lenguaje de programación.
- La aceptación de la creación de un software orientado a objetos.
- En los últimos años Java ha tenido gran aceptación dentro de los campos de la telefonía móvil y de los navegadores Web, pudiendo empaquetar la aplicación una vez esté finalizada de forma que puede ser distribuida fácilmente en diversos campos.

Clases Implementadas en la Aplicación

En esta sección, vamos a ver de qué manera se ha llevado a cabo la implementación de las principales clases diseñadas. Hablaremos también de las clases proporcionadas por Java que nos han ayudado a crear estas clases y como se han solucionado los problemas planteados a la hora de implementar la solución. Para cada clase implementada en nuestro paráctica, vamos a comentar aquellos fragmentos del código fuente que consideramos importantes para la compresión del mismo.

Clase invader

Nuestra clase *invader* es heredera de la clase *Canvas*, la clase por excelencia para los componentes visuales. Para crear la asociación entre nuestra clase y la ventana que vamos a emplear, dado que Canvas desciende de Component, podemos añadirlo al panel de la ventana como un componente más. Para organizar los contenedores se le asignan un *layout manager* usando *setLayout*, que está dentro de java.lang.Object, pero en nuestro caso no es necesario y se pone a *null*.
Nuestra ventana base va a ser creada y mantenida por la clase invader. Las siguientes secciones cubren las primeras secciones de código en la clase principal.



Comienzo

En java la clase principal es "`públic estátic void main (String arg [])`".Aquí es donde comienza la aplicación. Desde aquí vamos a crear una instancia de nuestra clase principal, que llama a todo lo demás. invader será una subclase de Canvas, ya que será el principal elemento que muestra los gráficos. Hay que tener en cuenta que debe ser una subclase de Canvas que apoya el uso de aceleración de gráficos.

```
public static void main(String argv[]) {
        invader g = new invader();
                g. run();
}
```

Creación de la ventana

Primero tenemos que crear nuestra ventana y configurar su contenido Vamos a fijar nuestra resolución a 800x600.

```
//Crear un marco para contener nuestro juego
JFrame container = new JFrame("Space Invaders");
//obtener el contenido de la trama y la
//creación de la resolución del juego
JPanel panel = (JPanel) container.getContentPane();
panel.setPreferredSize(new Dimension(800,600));
panel.setLayout(null);
//configurar nuestro tamaño de la ventana
setBounds(0,0,800,600);
panel.add(this);
```

Con la ventana que estamos trabajando se va a utilizar la aceleración de gráficos. Haremos la ventana de tamaño fijo, para evitar que el usuario pueda modificarlo.
Con `public void setResizable(boolean resizable)` decidimos si es posible redimensionar la ventana.

Parametros:

`resizable` : `true` si es redimensionable, `false` queda fijado.

Con public void **`setVisible`**`(boolean b)` mostramos o ocultamos este componente dependiendo del parámetro b.

Parameteros:

b: `true`, muestra el componente, si no queda oculto.

```
setIgnoreRepaint(true);
// hacer la ventana visible
container.pack();
//fija la ventana
container.setResizable(false);
container.setVisible(true);
```

Aceleración de gráficos

```
java.awt.image
Class BufferStrategy
java.lang.Object
  └ java.awt.image.BufferStrategy
```

La clase BufferStrategy representa el mecanismo con el que organizamos la memoria en un Canvas o en una ventana. Las limitaciones del hardware y software determinan si el BufferStrategy puede ser implementado y cómo. Estas limitaciones son detectadas a través de habilidades del GraphicsConfiguration usado cuando se crea un Canvas o en una ventana.
Para administrar nuestra Zona de Dibujo (Canvas), vamos a trabajar con una clase del JDK. Se llama BufferStrategy, y su técnica para la gestión de buffers, o más bien la memoria de intercambio de buffers que nos resultará útil. Esta página nos sirve para la aceleración de gráficos y el page flipping, (consiste en tener el doble de memoria, dos paginas de memoria, de lo que la pantalla usa, es decir, si tenemos una resolución de 800x600 a 16 bpp, se usa casi 1 Mbyte de memoria de video, en este caso usaríamos casi 2 Mbytes para que mientras se dibuja en una página se muestre la otra).
La creación de un BufferStrategy no puede ser más simple. Simplemente pedir a Canvas que lo ejecute. La único que debe ser especificado es el número de buffers a utilizar para la gestión de la pantalla, en este caso vamos a usar sólo dos.

```
//Crear el BufferStrategy
//para gestionar nuestro Gráficos
createBufferStrategy(2);
strategy = getBufferStrategy();
```

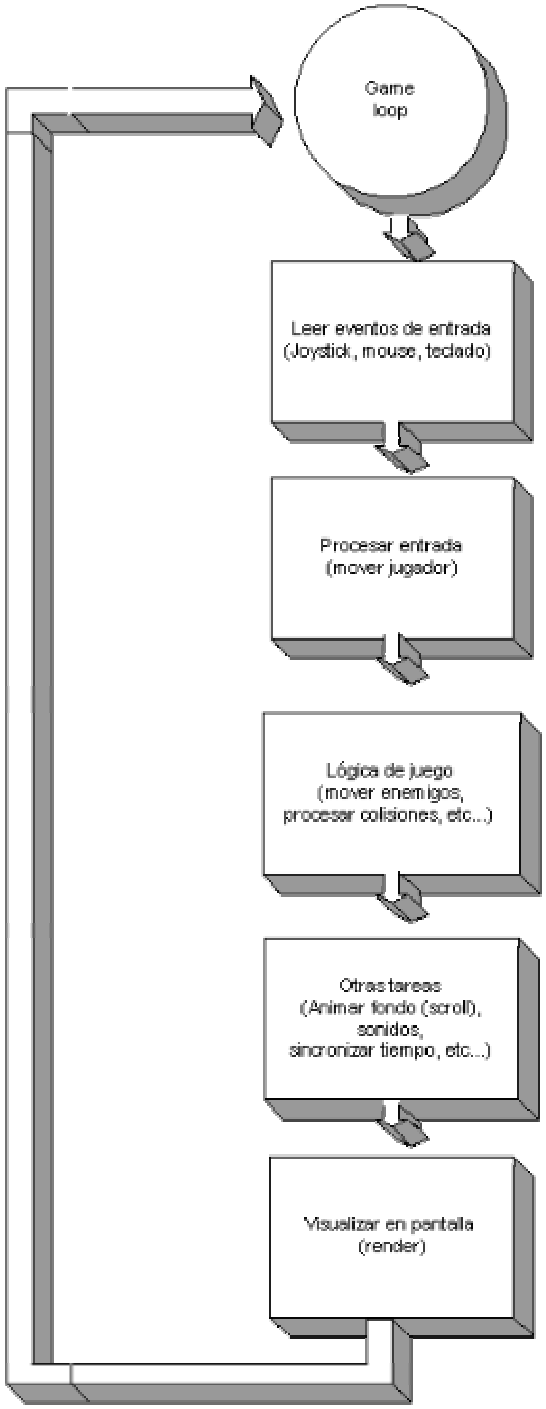
El bucle del juego, “game loop”.

El bucle del juego es muy importante en este tipo de juegos.

El bucle del juego se ejecuta así:

- Calcular el tiempo desde el último bucle
- Procesar las entradas del jugador
- Mover todo basado en el tiempo desde el último bucle
- Dibujar todo lo que hay en pantalla
- Actualizar buffers para que la nueva imagen sea visible
- Verifica entradas
- Actualiza los eventos

Cuando jugamos a un juego parece que todo pasa a la vez, en el mismo instante, sin embargo, sabemos que un procesador sólo puede realizar una acción a la vez. La clave es realizar cada una de las acciones tan rápidamente como sea posible y pasar a la siguiente, de forma que todas se completen antes de visualizar el siguiente frame del juego.
El "game loop" o bucle de juego es el encargado de "dirigir" en cada momento que tarea se está realizando. En la figura podemos ver un ejemplo de game loop, y aunque más o menos todos son similares, no tienen por que tener exactamente la misma estructura. Analicemos el ejemplo.



Lo primero que hacemos es leer los dispositivos de entrada para ver si el jugador ha realizado alguna acción. Si hubo alguna acción por parte del jugador, el siguiente paso es procesarla, esto es, actualizar su posición, disparar, etc..., dependiendo de qué acción sea. En el siguiente paso realizamos la lógica de juego, es decir, todo aquello que forma parte de la acción y que no queda bajo control del jugador, por ejemplo, el movimiento de los enemigos, cálculo de trayectoria de sus disparos, comprobación de colisiones entre la nave enemiga y la del jugador, etc... Fuera de la lógica del juego quedan otras tareas que realizamos en la siguiente fase, como son actualizar el scroll de fondo (si lo hubiera), activar sonidos (si fuera necesario), realizar trabajos de sincronización, etc.. Ya por último, nos resta volcar todo a la pantalla y mostrar el siguiente frame. Esta fase es llamada “fase de render”.

Normalmente, el game loop tendrá un aspecto similar a lo siguiente:

```
int done = 0;
while (!done) {
    // Leer entrada
    // Procesar entrada
    // Lógica de juego
    // Otras tareas
    // Mostrar frame
}
```

Antes de que entremos en el game loop, tendremos que realizar múltiples tareas, como inicializar todas las estructuras de datos, etc...
El siguiente ejemplo es mucho más realista. Está implementado en un thread.

```
public void run() {
    iniciar();
    while (true) {

        // Actualizar fondo de pantalla
        doScroll();

        // Actualizar posición del jugador
        computePlayer();

        // Actualizar pantalla
        repaint();
        serviceRepaints();

        try {
            Thread.sleep(sleepTime);
        } catch (InterruptedException e) {
            System.out.println(e.toString());
        }
    }
}
```

Lo primero que hacemos es inicializar el estado del juego. Seguidamente entramos en el bucle principal del juego o game loop propiamente dicho. En este caso, es un bucle infinito, pero en un juego real, tendríamos que poder salir usando una variable booleana que se activara al producirse la destrucción de nuestro avión o cualquier otro evento que suponga la salida del juego.

Ya dentro del bucle, lo que hacemos es actualizar el fondo de pantalla, a continuación calculamos la posición de nuestra nave para posteriormente forzar un repintado de la pantalla con una llamada a repaint() y serviceRepaints(). Por último, utilizamos el método sleep() perteneciente a la clase Thread para introducir un pequeño retardo. Este retardo habrá de ajustarse a la velocidad del dispositivo en que ejecutemos el juego.

Nuestro gameloop quedaría de esta forma:

```
while (ejecucionJuego) {
        //calcular tiempo desde la última actualización, esto
        //se utiliza para calcular en qué medida las entidades deberán
        //mover este bucle
        long delta = System.currentTimeMillis() - lastLoopTime;
        lastLoopTime = System.currentTimeMillis();
        // Encuentra el contexo para la aceleración de gráficos
        Graphics2D g = (Graphics2D) strategy.getDrawGraphics();
        g.setColor(Color.black);
        g.fillRect(0,0,800,600);
        //finalmente, hemos completado dibujo
        g.dispose();
        strategy.show();
        //finalmente detenerse
        //100 fps en las ventanas, esto podría variar debido a la cada bucle
        try { Thread.sleep(10); } catch (Exception e) {}
}
```

Por último se llama a gameLoop, método `run()`, en la parte inferior de la clase principal del juego. Esto inicia el bucle del juego.

Para la creación de threads, lo primero declaro la clase invader de la siguiente manera:

```
public class invader extends Canvas implements Runnable{
```

y utilizo los métodos `start() y run()` para el procesamiento de los hilos

`public void` **`start`**`()`
Hace que este thread comience a ejecutarse, java llama al método run() de este thread. Lo que pretendemos es que dos threads se ejecuten concurrentemente, uno es el que devuelve la llamada de start() y el otro el que ejecuta el método run()

`public void` **`run`**`()`
Usamos **`run`**`()` conjuntamente con Runnable. El método run() es el corazón de cualquier "Thread" y contiene las tareas de ejecución, la acción sucede dentro del método *run()*, digamos que el código que se encuentra dentro de dicho método es el trabajo por hacer, por lo tanto, si queremos realizar diversas operaciones cada una simultánea pero de manera independiente, tendremos varias clases, cada una con su respectivo método run(). Dentro del método run() puede haber llamadas a otros métodos como en cualquier otro método común, pero la pila de ejecución del nuevo proceso siempre comenzará a partir de la llamada al método run().

En `init` y en `verificaDisparo` utilizamos el método start.

```
	private void init() {
		// se crea la nave defensora en el centro de la pantalla
		naveDef = new nave(this,"nave.gif",370,550);
		entidades.add(naveDef);
		naveDef.th.start();

	public void verificaDisparo() {
		...
		...
		// Posicion inicial disparo de nave
		// naveDef.getX()+13,naveDef.getY()-10
		laser shot = new laser
		(this,"laser.gif",naveDef.getX()+13,naveDef.getY()-10);
		shot.th.start();
		entidades.add(shot);
```

Sprites y gestión de los recursos

La clase Sprite

La clase sprite actuará como una envoltura alrededor de java.awt.Image. Sin embargo, nos va a dar un lugar para ampliar el juego más adelante. La operación más importante de un sprite es el movimiento por la pantalla

```
public class Sprite {
	/** La imagen que hay que extraer de este sprite */
	private Image image;
	/**
	 * Crea un nuevo sprite sobre la base de una imagen
	 * @params: image La imagen que se esta sprite
	 */
	public Sprite(Image imagen) {
		this.imagen = imagen;
	}
	/**
	 * Obtener la anchura  del señalado sprite
	 * @return anchura  en píxeles de este sprite
	 */
	public int getAncho() {
		return image.getWidth(null);
	}
	/**
	 * Obtener la altura del señalado sprite
	 * @return altura en píxeles de este sprite
	 */

	public int getAlto () {
		return image.getHeight(null);
	}
	/**
	 * Dibuja el sprite en el contexto que ofrece gráficos
	 * @params:g Los gráficos contexto,en la que señalar a la sprite
	 * @params: x La ubicación en la que señalar a la sprite
	 * @params: y la ubicación y en la que señalar a la sprite
	 */
	public void draw(Graphics g,int x,int y) {
		g. dibujar (imagen,x,y,null);
	}
}
```

La clase SpriteAux

Una parte importante de cualquier juego es la gestión de los recursos que se utilizan. En nuestro caso, los buscamos en los sprites. Sin embargo, lo que queremos hacer es cargar cualquier sprite solo una vez. También sería muy útil si la recogida de los sprites existentes en una ubicación concentra. Con esto en mente vamos a poner en marcha un objeto SpriteAux, es decir una clase nueva. Esta será responsable de la carga y la caché de los sprites. Para que sea más fácil de hacerse con el SpriteAux se implementará como singleton (tipo de clase que se puede instanciar una sola vez).

La aplicación de la Singleton

Un singleton significa simplemente tener una única instancia del objeto disponible. En muchos casos esto se hace simplemente por conveniencia, sin embargo a veces hay buenas razones para la aplicación del diseño para que sólo exista una única instancia de la clase. La aplicación de este sprite en nuestra tienda es como esto:

```
private static SpriteAux single = new SpriteAux ();

/**
 * Obtener la única instancia de esta clase
*/

public static SpriteAux get() {
      return single;
}
```

El retorno single es la única instancia de la clase que se crea, el método estático get nos da acceso a la instancia.

Cargando Sprites

El primer paso es recuperar la imagen del sprite. En Java podemos utilizar ClassLoader para encontrar la imagen. Esto hace que sea mucho más fácil de desarrollar juegos con los recursos empaquetados. ya que hace posible webstart.
El primer paso es localizar el sprite:

```
URL url = this.getClass().getClassLoader().getResource(ref);
```

Es importante tener en cuenta que la cadena de las funciones utilizadas para llegar a la clase es sólo para apoyar clase especializada, en nuestro caso WebStart. URL permitirá acceder a la imagen que se especifica en la cadena "ref".
El siguiente paso es cargar la imagen. En Java se hace con la clase ImageIO. Para cargar nuestra imagen usamos este código:

```
source Image = ImageIO.read (url);
```

Tenemos que asignar memoria para guardar nuestra imagen, de esta manera la imagen no consume recursos de la CPU, sólo utiliza la tarjeta gráfica. La asignación de la memoria gráfica se logra de este modo:

```
//Crear una imagen de tamaño adecuado para almacenar nuestros sprite
GraphicsConfiguration gc = GraphicsEnvironment.getLocalGraphicsEnvironment().
getDefaultScreenDevice().getDefaultConfiguration();
Image image = gc.createCompatibleImage(sourceImage.getWidth(),
                                       sourceImage.getHeight(),
                                       Transparency.BITMASK);
```

El paso final es dibujar la imagen. Esto crea nuestro sprite:

```
//dibuja la imagen con aceleración gráfica
image.getGraphics (). DrawImage (sourceImage, 0,0, null);
```

Lo único que nos queda por hacer es crear un Sprite.

Caché de los Sprites

La misión de SpriteAux es gestionar la caché de las imágenes que se han cargado. Para ello, añadimos un HashMap. Este mapa se vinculará a la cadena de referencias de las imágenes de los sprites que se cargan. Siempre que se cargar un sprite hay que añadirlo.

```
//Crear un sprite, añadir que la caché de la devolvieran
Sprite sprite = new Sprite (imagen); sprites.put (ref, sprite);
```

Del mismo modo, cada vez que un sprite solicita verificar si ya esta en el mapa. Si es así, devolvemos nuestra copia en caché de carga en lugar de una copia nueva:

```
//Si ya tenemos el sprite en la caché
//entonces regresar si la versión existente
if (sprites.get (ref)!= Null) {
return (Sprite) sprites.get (ref);
}
```

Movimiento

Para poner todo en movimiento utilizamos la clase movimiento. En cada paso por el bucle actualizamos la información. El bucle tiene este aspecto:

```
//bucle pidiendo a cada entidad para moverse
for (int i = 0; i < entidades.size (), i + +) {
(movimiento) entity (movimiento) entidades.get (i);
entity. gestionMovimiento (delta);}
}
// bucle dibujo todas las entidades que tenemos en el juego
for (int i=0;i<entidades.size();i++) {
        movimiento entity = (movimiento) entidades.get(i);
        entity.dibujar(g);
        }
```

Recordar que se calculó el tiempo que se había tardado en realizar el bucle. Esto puede ser utilizado para calcular en qué medida una entidad debe moverse por el bucle.

La clase movimiento

La clase movimiento contendrá la ubicación y el movimiento actual y la representación visual. Cuando una entidad está construida estos valores se actualizarán. Esto incluye la recuperación de la sprite desde SpriteAux. La ubicación y el movimiento asociado a la entidad será definido y recuperado a través de los métodos establecidos.
Una vez que estas propiedades se definen podemos cubrir los dos métodos que requieren de movimiento. El método gestionMovimiento () tiene este aspecto:

```
public void gestionMovimiento (long delta) {
/ / actualizar la ubicación de la entidad en base a mover velocidades
        x += (delta * dx) / 1000;
        y += (delta * dy) / 1000;
}
```

Tomamos el tiempo transcurrido que se multiplican por el movimiento en cada sentido y lo añadimos sobre esta ubicación, la división entre 1000 es para ajustar el valor del movimiento que se especifica en píxeles por segundo, pero el tiempo se especifica en milisegundos. Cada bucle de todas las entidades que se moverán de acuerdo con sus valores de circulación actuales.
Ahora necesitamos dibujar una entidad de Aceleración de Gráficos en nuestro contexto. Se realiza de este modo:

```
public void dibujar(Graphics g) {
        sprite.dibujar(g,(int) x,(int) y);
}
```

En esencia, este sólo señala al sprite en el contexto de gráficos suministrados en su ubicación actual. Así que cada bucle, mueve a la entidad y se trazar en el lugar correcto.

Ahora que hemos definido nuestra entidad básica debemos crear unas subclases que utilizaremos más adelante. Tenemos que crear cinco subclases de la clase movimiento; nave, guardián, Aliens, láser y misil.
El último paso es crear nuestras entidades y añadirlas al entorno del juego. Si queremos añadir un método para realizar cualquier acción lo añadimos a la clase main, en este caso invader y lo ponemos en init (). Esto inicializa un conjunto de entidades cuando empieza el juego. La implementación actual es así:

```
private void init() {
        th = new Thread(this); //Crea un nuevo hilo
        (//crear la nave, aproximadamente en el centro de la pantalla
        naveDef = new guardian (this," nave.gif ",370,550);
        entidades.add(naveDef);
        //crear Un bloque de los aliens (5 filas, por 12 de alien)
        cuentaAliens = 0;
        for (int row=0;row<5;row++) {
                for (int x=0;x<9;x++) {
                movimiento alien = new Aliens (this,
                "alien.gif",+(x*50), 50)+row*30); entities.add(alien);
                cuentaAliens ++; }}}
```

Como se puede ver, el inicio toma dos pasos. La primera es la creación de la nave del jugador. Hay que crear una clase nave con la nave en el centro de la pantalla.
El segundo paso es la creación de todos los alien. Se hace con dos bucles `for` anidados. Además contamos con el número de aliens que hemos creado a fin de poder analizar si el jugador ha ganado el juego.

Suponiendo que el código para mostrar el movimiento y el apoyo a las entidades se ha añadido al juego en el bucle principal, al ejecutar el juego debe mostrar la nave defensora y una matriz de alien, en este caso cinco filas y nueve columnas.
Ahora cada tipo de entidad se mueve a su manera y con sus propias limitaciones. Veamos cada uno de ellos.

Detección de colisión

En nuestro bucle principal del juego que está en la clase invader, vamos a comprobar en cada pasada si una entidad ha colisionado con cualquier otra entidad.

```
for (int p=0;p<entidades.size();p++) {
      for (int s=p+1;s<entidades.size();s++) {
            movimiento yo = (movimiento) entidades.get(p);
            movimiento el = (movimiento) entidades.get(s);
                  if (yo.colisionaCon(el)) {
                        yo.haColisionado(el);
                        el.haColisionado(yo);
                  }
      }
}
```

En primer lugar, tendremos que saber si dos entidades han chocado. Para ello, tenemos en la clase moviento esto:

```
public boolean colisionaCon(movimiento other) {
      yo.setBounds((int) x,(int) y,sprite.getAncho(),sprite.getAlto());
      el.setBounds((int) other.x,(int) other.y,other.sprite.getAncho(),
      other.sprite.getAlto());
      return yo.intersects(el);
}
```

```
java.awt
Class Component
java.lang.Object
  └ java.awt.Component

setBounds

public void setBounds(int x,int y, int width, int height)
```

setBounds mueve y reconstruye este componente, la localización de las coordenadas x e y se dan a partir del borde superior izquierdo y la nueva forma especifica el ancho y el alto.

Parameteros:

x : nueva coordenada de este componente, variable x
y : nueva coordenada de este componente, variable y
width : nueva anchura de este componente
height : nueva altura de este componente

Este método comprueba si la propia entidad choca con la otra entidad. Vamos a basarnos en regiones rectangulares de intersección de la clase java.awt.Rectangle.
Primero configuramos los dos rectángulos para representar a las dos entidades. Ahora vamos a utilizar la funcionalidad de java.awt.Rectangle importado, para comprobar si las dos entidades se entrecruzan entre sí.
Lo siguiente que vamos a añadir es una manera de notificar a las entidades que han colisionado con otra. Para ello vamos a añadir un método como éste a la clase Entity.

```
public abstract void haColisionado(movimiento other);
```

Una vez más, el resultado de la colisión se basa en la lógica juego. Si la entidad que colisionó con la nave es un Alien entonces avisar que la partida finaliza

Cuando la nave alcanza con el laser a un alien, lo elimina a la vez que se retirar el laser que ha colisionado, de esta forma se van decrementando los aliens que es el objetivo que buscamos. El código quedaría de esta forma:

```
            //si un laser de la nave colisiona con
            //un alien se alnulan los dos.
            if (other instanceof laser) {
                  juego.quitarEntidad(this);
                  juego.quitarEntidad(other);
                  juego.avisoMataAliens();
}
```

La nave tiene que reaccionar cuando le alcanza el disparo de un alien, es decir, el jugador debe ser destruido y por exigencias del enunciado finalizar la partida. El código quedaría de esta forma:

```
      //si un misil del alien colisiona con
      //la nave guardian se eliminan los dos
      if (other instanceof guardian) {
            juego.quitarEntidad(this); // Eliminamos nave guardian
            juego.quitarEntidad(other);   // Eliminamos misil
            juego.avisoDerrota();
     }
```

La clase guardián

La nave guardián será controlada por el jugador. Queremos evitar que la nave guardián se desplace fuera de los lados de la pantalla, así que añadiré un fragmento de código al método `gestionMovimiento()` en nave:

```
      public void gestionMovimiento (long delta) {
      //si nos movemos y llegado al final de la parte la izquierda
      //no podemos seguir
            if ((dx < 0) && (x < 10)) {
                  return;
            }
            //si nos movemos y llegado al final de la parte la derecha
            //no podemos seguir
            if ((dx > 0) && (x > 750)) {
                  return;
            }
```

```
            super. gestionMovimiento (delta);
      }
```

Lo que estamos diciendo aquí es que si estamos moviéndonos y llegamos a la parte izquierda estamos a punto de pasar la pantalla entonces no permiten el movimiento en ese sentido. A la inversa, si estamos moviéndonos y llegamos a la parte derecha y está a punto de pasar fuera de la parte derecha de la pantalla entonces no se permite el movimiento en ese sentido. Lo que acabamos de describir es el movimiento normal de una entidad.

Además cuando se colisiona con un alien se retiran del juego las dos entidades.

La clase laser

La clase laser es bastante simple, el disparo sólo se tiene que desplazar por la pantalla de forma vertical hasta que encuentre un alien o salga fuera del rango que establecemos por la parte superior de la pantalla, momento en el que lo quitamos de la entidad. La velocidad del laser está definida dentro del contructor del clase y queda de esta forma:

```
      dy = -800; // Velocidad del laser
```

La clase Aliens

Los alien son la parte más difícil de nuestro juego. A medida que se desplazan debemos tener en cuenta cuando llegan al lateral de la pantalla y empiezan a moverse en la dirección opuesta. Cada vez que hay un cambio de dirección se desplazan un paso hacia abajo hasta poder llegar a colisionar con la nave defensora si no los hemos eliminado todos. Parte de esto está cubierto por la lógica del juego. Primero inicializamos el movimiento de los alien para comenzar a mover a la izquierda sobre la velocidad predefinida y vamos a poner la detección de un alien chocando con los límites de la ventana de juego.

```
public void gestionMovimiento (long delta) {
        //si hemos llegado a la parte izquierda de la pantalla y
        // solicitar una lógica actualización
        if ((dx < 0) && (x < 10)) {
                juego.cargaLogica ();
        }
        // Y viceversa, si hemos llegado a la parte derecha de
        // la pantalla, solicitar una actualización lógica
        if ((dx > 0) && (x > 750)) {
                juego.cargaLogica ();
        }
        super. gestionMovimiento (delta); // movimiento normal
}
```

De la misma manera que en nave, verificar si la entidad ha golpeado el borde de la pantalla. Sin embargo, en este caso, el juego notificará que la lógica de juego para todas las entidades que deben ser ejecutados.

La clase misil

Necesitamos una clase que haga que los alien disparen, debemos tener cuidado con la velocidad del misil, la frecuencia, que se implementa en el método run de la clase invader y cuando colisiona con otra entidad que se implementa en el método haColisionado de la clase movimiento, quedaría de esta forma:

```
dy = 100; // Velocidad del misil


//frecuancia de disparo
if (Math.random()<0.02) {


//si un misil del alien colisiona con
//la nave guardian se eliminan los dos
if (other instanceof guardian) {
      juego.quitarEntidad(this);
      juego.quitarEntidad(other);
```

También es importante retirar el misil del juego una vez desaparezca por la parte inferior de la pantalla, de esta manera hago un uso mucho más eficiente de esta entidad y puedo limitar la aparición de misil en la pantalla, como implemento en el método run de la clase invader.

```
//si sale fuera de la pantalla,
//lo quitamos
if (y > 600){
juego.quitarEntidad(this);
// Frecuencia de disparo
// limitación a 5 disparos en pantalla
if (Math.random()<0.02 && numAliensTiro<=5)
```

Para que los aliens disparen de forma esporádica tal como propone el enunciado de la práctica utilizo el siguiente código, que sitúo en la parte del bucle del juego, método `run()`, en la clase invader:

```
// Cuando comienza el juego los aliens
// empiezan a disparar aleatoriamente
if (Math.random()<0.03) {
        for (int i=0;i<entidades.size();i++) {
        Entity entity = (Entity) entidades.get(i);
        if (entity instanceof Aliens) {
        if (Math.random()<0.02 && numAliensTiro<=5) {
        misil alienTiro = new misil
(this, "tiro2.gif", entity.getX()+13, entity.getY()+25);
        entidades.add(alienTiro);
                }
        }
        }
}
```

Teclas del juego

```
java.awt.event
Class KeyAdapter
java.lang.Object
  └ java.awt.event.KeyAdapter

java.lang.Object
  └ java.util.EventObject
      └ java.awt.AWTEvent
          └ java.awt.event.ComponentEvent
              └ java.awt.event.InputEvent
                  └ java.awt.event.KeyEvent
```

Nuestro próximo paso es hacer que la nave sea controlable. Para hacer esto tenemos que añadir una clase a la clase principal de nuestro juego. El aspecto que presenta es:

```
private class KeyInputHandler extends KeyAdapter {
	public void keyPressed(KeyEvent e) {
		if (e.getKeyCode() == KeyEvent.VK_O) {
			teclaIzd= true;
		}
		if (e.getKeyCode() == KeyEvent.VK_P) {
			teclaDer= true;
		}
		if (e.getKeyCode() == KeyEvent.VK_SPACE) {
			teclaDis= true;
		}
	}

	public void keyReleased(KeyEvent e) {
		if (e.getKeyCode() == KeyEvent.VK_O) {
			teclaIzd= false;
		}
		if (e.getKeyCode() == KeyEvent.VK_P) {
			teclaDer= false;
		}
		if (e.getKeyCode() == KeyEvent.VK_SPACE) {
			teclaDis= false;
		}
	}
	public void keyTyped(KeyEvent e) {
		// si Pulsa escape, entonces si salimos del juego
		if (e.getKeyChar() == 27) {
			System.exit(0);
		}
	}
}
```

Esta clase simplemente recoge teclas que están siendo pulsadas y liberadas y los registros de los estados en un conjunto de variables booleanas en la clase principal.
Para hacer esto necesitamos para añadirlo a nuestra ventana como un "KeyListener". Esto se hace añadiendo la siguiente línea al constructor de nuestra clase de juegos:

```
addKeyListener(new KeyInputHandler());
```

Este código fuente se configurará con los valores adecuados, como son las teclas pulsadas. El siguiente paso es responder a esos valores que están en el bucle del juego. Si agregamos esto en el bucle del juego podemos agregar el control de la nave:

```
// nave no se mueve. Si una tecla de cursor se presiona entonces
// actualizar el movimiento
      naveDef.setoHorizontal(0);
      if ((teclaIzd) && (!teclaDer)) {
            naveDef. setoHorizontal (-velocidadMovi);
            } else if ((teclaDer) && (!teclaIzd)) {
            naveDef. setoHorizontal (velocidadMovi);
      }
```

Como se puede ver, verificamos que la izquierda o la derecha se presionan. Si son presionadas actualizaremos el movimiento de los valores asociados con el jugador de la nave.

Si el jugador pulsa la tecla de disparo, se efectúa un disparo vertical. Sin embargo, no queremos dejar que el jugador siga disparando. Debemos limitar la cadencia de disparado tal como nos piden en el enunciado. Pondremos esta funcionalidad en un método llamado de utilidad `" verificaDisparo()"`

```
//Si estamos presionando fuego, entonces dispara
if (teclaDis) {
       verificaDisparo();
}
```

Para limitar la cadencia de disparado hacemos lo siguiente

```
public void verificaDisparo() {
      // verifica tiempo de espera del disparo
      if (System.currentTimeMillis() - ultimoDis < intervaloNave) {
            return;
            }
      // si esperamos lo suficiente, creamos un Nuevo disparo
      // y tomamos el tiempo
      ultimoDis = System.currentTimeMillis();
      // Posicion inicial disparo de nave
      // naveDef.getX()+13,naveDef.getY()-10
      laser shot = new laser
       (this,"laser.gif",naveDef.getX()+13,naveDef.getY()-10);
      shot.th.start();
      entidades.add(shot);
      }
```

En primer lugar, verificamos si desde la última vez que el jugador efectuó un disparo ha transcurrido suficiente tiempo. Ahora vamos a crear y añadir una entidad para representar a los disparos del jugador.

Bucle del juego

El paso final para lograr que la labor de detección de colisiones sea efectiva es añadir una sección al bucle del juego en la clase invader a través de todas las entidades de control si colisionan unas con otras. Éste es el código:

```
// fuerza bruta colisiones, comparar cada entidad con las
// demás. Si cualquiera de ellas chocan notifica que
// ambas entidades han colisionado
for (int p=0;p<entidades.size();p++) {
```

```
for (int s=p+1;s<entidades.size();s++) {
      movimiento yo = (movimiento) entidades.get(p);
      movimiento el = (movimiento) entidades.get(s);
            if (yo.colisionaCon(el)) {
            yo.haColisionado(el);
            el.haColisionado(yo);
            }
      }
}
```

Para cada entidad que a través de todo comprobar si se ha producido una colisión. Si se ha producido una colisión hay que notificarlo a ambas partes.
Una forma más inteligente de hacer esto podría ser a la verificación de las colisiones sólo cada cierto tiempo.

Lógica de Juego

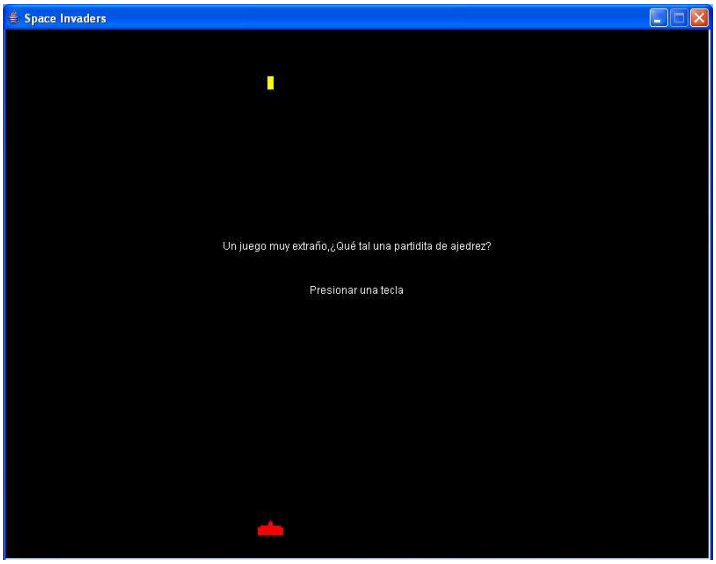
El paso final de nuestro juego es llenar el juego de partes lógicas. Esta implementado en un conjunto de métodos de la clase invader.

avisoDerrota ()

Este método se llama para indicar que el jugador ha sido derrotado. Esto puede ocurrir si un alien ha colisionado con el jugador o un alien alcanza con un disparo a la nave defensora. Como anécdota en el mensaje que nos avisa sobre la derrota quise hacer mención a la película juegos de guerra, en la que la máquina pregunta ¿Otra partidita, profesor Falken?, cuando está a punto de desencadenar la tercera guerra mundial.

avisoVictoria ()

Este método se llama para indicar que el jugador ha ganado el juego. El objetivo de este juego es matar a todos los alien.



Pantalla que nos muestra el fin de la partida con victoria.

avisoMataAliens ()

Este método se llama para indicar que un alien ha sido derrotado como resultado de una colisión registrada por un disparo. Una vez que todos los aliens han sido alcanzados por disparos de la nave defensora el jugador ha ganado. El código es así:

```
public void avisoMataAliens() {
// decrementa los alien, si no quedan más, el jugador ha ganado
      cuentaAliens--;
      if (cuentaAliens == 0) {
      avisoVictoria();
      }
      // si todavía hay algunos alien entonces
      // acelerar todos los que quedan
      for (int i=0;i<entidades.size();i++) {
      movimiento entity = (movimiento) entidades.get(i);
      if (entity instanceof Aliens) {
      // acelerar un 3%
      entity. setHorizontal (entity.getVeloHorizontal() * 1.03);
      }
      }
}
```

A medida que vamos eliminando aliens, la velocidad aumenta un 3%. Un último detalle que necesitamos es el apoyo a la entidad basada en la lógica del juego. Tenemos la obligación implícita en la construcción de la **Aliens**. Con este pequeño incremento de velocidad cada vez que se destruye un alien se consigue dar un poco de emoción al juego, al mismo tiempo que te permite terminar la partida con éxito, se podría incrementar la dificultad haciendo que acelerasen más.

Entidad Lógica

Cada entidad debe tener la posibilidad de apoyar su propia lógica. Para facilitar esto vamos a agregar un método a la clase **movimiento**. descender() permitirá a las subclases de **movimiento** definir un poco la lógica que se ejecutará cada vez que se solicita la lógica del juego. En Aliens por ejemplo pedimos que se ejecute la lógica de juego, cuando el borde de la pantalla se detecta. descender () La aplicación en Aliens tiene este aspecto:

```
public void descender() {
        //si detectamos que un alien llega al borde,
        // cambiamos el sentido
        dx = -dx;
        y += 10;
        // si hemos llegado a la parte inferior de la pantalla el
        // jugador muere
        if (y > 570) {
                juego.avisoDerrota ();
        }
}
```

Así que, cuando un alien detecta el borde de la pantalla que indica que el juego de lógica debe ejecutarse en las entidades. Las entidades del alien cambian de dirección (dx =-dx) y bajan un poco la pantalla (y + = 10). Por último, si el alien se mueve por la parte inferior de la pantalla, suficientemente bajo entonces notificar al juego que el jugador está muerto.

Entidad de la infraestructura lógica

Para completar la lógica del juego en la entidad tenemos que añadir un método en la clase invader que se indicará que la lógica del juego en la entidad debería ser ejecutada. El método es:

```
public void cargaLogica () {
        logicaAplicaJuego = true;
```

Esto indica que en el bucle del juego deberíamos ejecutar la lógica asociada a cada entidad actualmente en el juego.

```
// Si un evento ha indicado que debería ejecutarse
// la lógica del juego entonces se solicita a cada entidad
      if (logicaAplicaJuego) {
      for (int i=0;i<entidades.size();i++) {
            movimiento entity = (movimiento) entidades.get(i);
                  entity. descender();
            }
            logicaAplicaJuego = false;
      }
```

Si está activa, entonces llamamos a descender() de cada método. Por último, reiniciamos para lo que la lógica de juego se ejecute automáticamente próximo bucle.

Fase de Prueba

En este capítulo nos centraremos en las pruebas de usuario, es decir, intentaremos descubrir los problemas de menor importancia con los que el usuario se puede encontrar a la hora de utilizar la aplicación.

Nos centraremos en tres objetivos primordiales:

1. Ayudar a encontrar los errores de cualquier tipo.
2. Convencer al cliente de que no hay errores importantes.
3. Proporcionar información que ayudará en la evolución del sistema en un futuro.

Tipos de Pruebas Disponibles

- **Prueba de usabilidad.** Comprueba que el sistema sea fácil de utilizar de forma efectiva. Esto puede incluir la prueba de todo el sistema, de parte de él, o de su documentación.
- **Prueba de módulo.** Comprueba los módulos individuales del sistema; en un desarrollo Orientado a Objetos, esto significa probar los objetos.
- **Prueba de integración.** Comprueba que las partes del sistema de un cierto nivel funcionan correctamente todas juntas; por ejemplo, que los objetos implicados en una colaboración funcionen tal y como está planeado.
- **Prueba de aceptación.** Normalmente lo ejecuta el cliente, o algún grupo independiente que representa al cliente; valida que el sistema realmente cumple su propósito.
- **Prueba de ejecución.** Puede tener lugar a cualquier nivel para comprobar que la ejecución del sistema es o será satisfactoria.
- **Prueba de carga.** Es un tipo de prueba de ejecución que pone a los sistemas bajo cargas mayores de las que normalmente se esperan, para comprobar que baja el rendimiento en vez de fallar catastróficamente.

Pruebas Realizadas sobre la Aplicación

Dentro del ámbito de pruebas disponibles que se han especificado en el apartado anterior, pasamos a comentar las que han sido factibles de realizar la práctica.

Prueba de usabilidad

Al tratarse nuestra aplicación de un videojuego, cualquier usuario que se encuentre familiarizado con ellos conocerá las principales teclas de mando, tales como las de dirección, <Espacio>, <O>, <P>, etc., es posible incluir tanto métodos como clases nuevas para mejorar el videojuego tales como un contador de puntos, un fondo de pantalla que de la sensación de desplazamiento por el espacio, una ayuda para el jugador o tecla de pausa, se podría incluir una pantalla de ayuda donde se muestren el significado de estas nuevas opciones a la que se accedería mediante la pulsación de una tecla diferente de las ya asignadas.

Prueba de módulo

A medida que se iban desarrollando las clases de la aplicación, se efectuaban pruebas individuales antes de integrarlas al proyecto con el fin de que si surgía algún problema, fuera más fácil su detección y posterior solución. Dichas pruebas han sido utilizadas con mayor frecuencia para la animación de las imágenes contenidas tanto dentro los objetos *sprites*. Este proceso ha sido realizado en cada una de las clases que componen la aplicación con resultado satisfactorio.

Prueba de integración

Todas las clases interaccionan entre sí correctamente y las pulsaciones de tecla por parte del jugador ofrecen el resultado esperado, por lo que no hay nada que objetar en esta prueba.

Prueba de aceptación

Ésta es una de las dos pruebas que se ha decidido no realizar. El motivo se debe a que, como la aplicación no está destinada a ninguna clientela específica ni va a ser utilizada en un entorno empresarial, no tiene sentido que ningún cliente, en nuestro caso los jugadores, se encarguen específicamente de probar la aplicación en busca de algún error de consideración.

Prueba de ejecución

El único impedimento que puede surgir en este apartado es que el usuario no haya instalado previamente la versión JRE necesaria para la ejecución de la aplicación en su ordenador. Respecto a lo demás, no es necesario ningún software adicional que instalar.

Prueba de carga

Se corresponde con la segunda prueba descartada para nuestro proyecto. Esta práctica esta destinada al uso y disfrute del usuario, y no se trata de una herramienta de trabajo sino de diversión. No requiere un nivel alto de procesamiento de datos ya que éstos sólo son cargados al principio, por lo que se desestima dicha prueba.

Ejecución y Contenido del CD

Antes de comenzar la Ejecución

A continuación describiremos la configuración mínima del equipo para una correcta ejecución, así como la recomendada.

Herramientas Necesarias

Para la creación, compilación y ejecución de aplicaciones en el lenguaje de programación Java, nuestro requisito es la instalación del entorno de programación J2SE.

Modo consola

Desde la versión 1.2 del SDK (*Software Development Kit*), es posible la ejecución de aplicaciones empaquetadas en ficheros JAR con el interprete de Java. El comando básico es:

c:\java\bin\java -jar invader.jar

La opción *-jar* le dice al intérprete que la aplicación está empaquetada en un .jar.
Para esta práctica no entregaré un fichero .jar

También podemos llamar al archivo invader.java con el siguiente comando:

c:\java\bin\java invader

Contenido del CD

- Documentación
    - Memoria de la Práctica.
    - Código del programa en lenguaje Java, fuente y compilado.

Bibliografía

http://java.sun.com/j2se/1.4.2/docs/api/

http://www.classicgaming.cc/classics/spaceinvaders/techinfo.php

http://www.ulpgc.es/otros/tutoriales/java/

http://www.programacion.net/java/tutorial/ags_j2me/

http://www.ulpgc.es/otros/tutoriales/java/Cap4/canvas.html

http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=javadoc

http://www.el-mundo.es/navegante/2003/11/28/juegos/1070015046.html

http://monillo007.blogspot.com/2008/01/hilos-en-java-threads-parte-1.html

http://www.itox.mx/Comunidad/Librero/umlTotal.pdf

http://www.clikear.com/manuales/uml/

http://www.dandel.net/2008/01/29/30-anos-de-space-invaders/

http://www.ingenierosoftware.com/analisisydiseno/casosdeuso.php

CÓDIGO FUENTE

```
1    /*************************************************************
2    /*************************************************************
3    /*
4    /*  Esta clase representa los alien.
5    /*
6    /*  @autor Antonio Rivero
7    /*
8    /*************************************************************
9    /************************************************************/
10   public class Aliens extends nave {
11
12   //*********************************************************************
13   /** Descrip: Constructor de clase. Crea un nuevo alien
14    *
15    * @Params: juego Representa el alien que creamos
16    * @Params: ref Referencia del sprite para mostrar del alien
17    * @Params: x Localización inicial del alien, variable x
18    * @Params: y Localización inicial del alien, variable y
19    *
20    ***********************************************************************/
21       public Aliens(invader juego, String ref,int x,int y) {
22            super(juego,ref,x,y );
23           dx = -75;
24       }
25
26   //*********************************************************************
27   /** Descrip: Actualización de los alien
28    *
29    *  @Params: Nada
30    *  @Return: Nada
31    *
32    ***********************************************************************/
33       public void descender() {
34            // si detectamos que un alien llega al borde,
35            // cambiamos el sentido y bajamos y += 10;
36           dx = -dx;
37           y += 10;
38            // si hemos llegado a la parte inferior de la pantalla
39            // la nave defensora es destruida
40            if (y > 570) {
41                juego. avisoDerrota();
42            }
43       }
44
45   //*********************************************************************
46   /** Descrip: Controla el movimiento de los aliens, evita que el alien
47    *           se desplace fuera de los lados de la pantalla
48    *
49    *  @Params: delta. Tiempo transcurrido desde el último movimiento (ms)
50    *  @Return: Nada
51    *
52    ***********************************************************************/
53       public void gestionMovimiento (long delta) {
54            if ((dx < 0) && (x < 10)) { // si alcanza el lado izquierdo
55                juego. cargaLogica(); // actualizamos la logica del juego
56            }
57            if ((dx > 0) && (x > 750)) { // y viceversa para el derecho
58                juego. cargaLogica();
59            }
60            super.gestionMovimiento (delta);
61       }
62   }
```

```
1    /*************************************************************
2    /*************************************************************
3    /*
4    /*  Esta clase representa los disparos del juego
5    /*
6    /*  @autor Antonio Rivero
7    /*
8    /*************************************************************
9    /*************************************************************/
10   public abstract class disparo extends movimiento {
11
12   //*************************************************************
13   /** Descrip: Constructor de clase.
14    *           Crea un nuevo disparo
15    *
16    * @Params: juego Representa el disparo que creamos
17    * @Params: ref Referencia del sprite para mostrar la nave
18    * @Params: x Localización inicial del laser, variable x
19    * @Params: y Localización inicial del laser, variable y
20    *
21    *************************************************************/
22       public disparo(invader juego, String sprite, int x, int y) {
23            super(juego,sprite,x,y );
24       }
25   }
```

```
1    /*************************************************************
2    /*************************************************************
3    /*
4    /*  Esta clase representa la nave guardian
5    /*
6    /*  @autor Antonio Rivero
7    /*
8    /*************************************************************
9    /*************************************************************/
10   public class guardian extends nave {
11
12   //*************************************************************
13   /** Descrip: Constructor de clase.
14    *           Crear una nueva entidad para representar la nave
15    *           defensora
16    *
17    * @Params: juego Representa la nave que creamos
18    * @Params: ref Referencia del sprite para mostrar la nave
19    * @Params: x Localización inicial de la nave, variable x
20    * @Params: y Localización inicial de la nave, variable y
21    *
22    *************************************************************/

23       public guardian(invader juego, String ref,int x,int y) {
24            super(juego,ref,x,y );
25       }
26
27   //*************************************************************
28   /** Descrip: Controla el movimiento de la nave, evita que la nave
29    *           se desplace fuera de los lados de la pantalla
30    *
31    *  @Params: delta. Tiempo transcurrido desde el último movimiento (ms)
32    *  @Return: Nada
33    *
34    *************************************************************/
35       public void gestionMovimiento (long delta) {
36            //si nos movemos y llegado al final de la parte la izquierda
37            //no podemos seguir
38            if ((dx < 0) && (x < 10)) {
39                 return;
40            }
41            //si nos movemos y llegado al final de la parte la derecha
42            //no podemos seguir
43            if ((dx > 0) && (x > 750)) {
44                 return;
45            }
46            super.gestionMovimiento (delta);
47       }
48   }
```

```
1    /***********************************************************************
2    ***********************************************************************
3    **
4    **  Importaciones generales del sistema
5    **
6    ***********************************************************************
7    ***********************************************************************/
8    import java.awt.Canvas;
9    import java.awt.Color;
10   import java.awt.Dimension;
11   import java.awt.Graphics2D;
12   import java.awt.event.KeyAdapter;
13   import java.awt.event.KeyEvent;
14   import java.awt.event.WindowAdapter;
15   import java.awt.event.WindowEvent;
16   import java.awt.image.BufferStrategy;
17   import java.util.ArrayList;
18   import javax.swing.JFrame;
19   import javax.swing.JPanel;
20
21   /***********************************************************************
22   ***********************************************************************
23   /*
24   /*  Clase principal del juego, se encarga de coordinar las funciones
25   /*  del programa y coordinar la lógica de juego, la gestión consiste
26   /*  en ejecutar el bucle del juego moviendo y dibujando las entidades
27   /*  en el lugar apropiado.
28   /*  Será informada cuando las entidades dentro del juego detecten
29   /*  eventos, como deribar un alien, el derribo de la nave defensora
30   /*  actuando de manera adecuada.
31   /*
32   /*  @autor Antonio Rivero
33   /*
34   /***********************************************************************
35   ***********************************************************************/
36   public class invader extends Canvas implements Runnable{
37       /** Estrategia que nos permite utilizar page flipping */
38       private BufferStrategy estrategia;
39       /** Verdadero si el juego se está ejecutando */
40       private boolean ejecucionJuego = true;
41       /** Lista de todas las entidades que existen en el juego*/
42       private ArrayList entidades = new ArrayList();
43       /** Lista de entidades que necesitamos eliminar del juego*/
44       private ArrayList quitamosLista = new ArrayList();
45       /** movimiento de la nave defensora*/
46       private movimiento naveDef;
47       /** Velocidad a la que se mueve el jugador (pixels/sec)*/
48       private double velocidadMovi = 150;
49       /** Tiempo del último laser*/
50       private long ultimoDis = 0;
51       /** Intervalo de laser de la nave (ms) */
52       private long intervadoNave = 650;
53       /** Número de aliens que se muestran en pantalla*/
54       private int cuentaAliens;
55       /** Mensaje impreso por pantalla, esperando presionar tecla*/
56       private String mensaje = "";
57       /** Verdadero si no presionamos ninguna tecla*/
58       private boolean esperaTecla = true;
59       /** Verdadero si presionamos la tecla de izquierda*/
60       private boolean teclaIzd = false;
61       /** Verdadero si presionamos la tecla de derecha*/
62       private boolean teclaDer = false;
63       /** Verdadero si estamos disparando*/
64       private boolean teclaDis = false;
65       /** Verdadero si la lógica de juego necesita ser aplicada*/
66       /** por lo general como consecuencia de un evento*/
67       private boolean logicaAplicaJuego = false;
68       /** Declaración del Thread*/
69       public Thread th = new Thread(this);
70
71
```

```
72    //*****************************************************************
73    /** Descrip: Constructor de clase y puesta en marcha del juego
74     *
75     *  @Params: nada
76     *  @Return: nada
77     *
78     *****************************************************************/
79        public invader() {
80            // crea un frame para contener el juego
81            JFrame container = new JFrame("Space Invaders");
82            // Coje el contenido del frame y carga la resolución del juego
83            JPanel panel = (JPanel) container.getContentPane();
84            panel.setPreferredSize(new Dimension(800,600));
85            panel.setLayout(null);
86            // Carga la ventana, (Canvas) y lo pone en el frame
87            setBounds(0,0,800,600);
88            panel.add(this);
89            // Informa AWT no interrumpe a la aceleración gráfica
90            setIgnoreRepaint(true);
91            // finalmente vemos la pantalla
92            container.pack();
93            //fija la ventana
94            container.setResizable(false);
95            container.setVisible(true);
96            // información para cerrar la ventana
97            // suponiendo que queramos salir del juego
98            container.addWindowListener(new WindowAdapter() {
99            public void windowClosing(WindowEvent e) {
100           System.exit(0);}});
101           // añade las teclas del sistema, definidas más tarde
102           addKeyListener(new KeyInputHandler());
103           requestFocus();
104           // crea el BufferStrategy
105           // administrar la aceleración de gráficos
106           createBufferStrategy(2);
107           estrategia = getBufferStrategy();
108           // inicializa las entidaddes del juego
109           init();
110       }
111
112   //*****************************************************************
113   /** Descrip: Comienza el juego, debe de partir de cero con todos los
114    *           aliens y la nave defensora
115    *
116    *  @Params: nada
117    *  @Return: nada
118    *
119    *****************************************************************/
120       private void empezar() {
121           // limpiar si se ha inicializado alguna entidad
122           entidades.clear();
123           init();
124           // inicializa las teclas
125           teclaIzd = false;
126           teclaDer = false;
127           teclaDis = false;
128       }
129
130       public boolean getPaused() {
131           return esperaTecla;
132           }
133
134
135   //*****************************************************************
136   /** Descrip: Inicializa las entidades, cada entidad debe de ser añadida
137    *           en el conjunto del juego
138    *
139    *  @Params: nada
140    *  @Return: nada
141    *
142    *****************************************************************/
```

```
143         private  void  init() {
144             th  = new Thread(this); //Crea un nuevo hilo
145             // se crea la nave defensora en el centro de la pantalla
146             naveDef  = new guardian(this,"nave.gif",370,550);
147             entidades. add(naveDef);
148             naveDef.th. start();
149             // Grupo inicial de aliens en este caso 5 filas y 9 columnas
150             cuentaAliens  = 0;
151             for (int row=0;row<5;row++) {
152                 for (int x=0;x<9;x++) {
153                     // Distancia lateral entre ellos 100+(x*80),
154                     // Distancia vertical entre ellos (50)+row*30
155                    movimiento alien   = new Aliens
156                     (this,"alien.gif",100+(x*65),(50)+row*30);
157                    entidades.  add(alien);
158                    cuentaAliens  ++;
159                 }
160             }
161 
162         }
163 
164     //*****************************************************************
165     /** Descrip: Inicializa la lógica del juego, que comenzará a funcionar
166      *           cuando lo requiera algún evento
167      *
168      *  @Params: nada
169      *  @Return: nada
170      *
171      *****************************************************************/
172         public  void cargaLogica() {
173             logicaAplicaJuego   = true;
174         }
175 
176     //*****************************************************************
177     /** Descrip: Quita la entidad del juego
178      *
179      *  @Params: entity. La entidad debe ser quitada
180      *  @Return: nada
181      *
182      *****************************************************************/
183         public  void quitarEntidad (movimiento entity ) {
184             quitamosLista.  add(entity);
185         }
186 
187     //*****************************************************************
188     /** Descrip: Aviso que la nave defensora es destruida
189      *
190      *  @Params: nada.
191      *  @Return: nada
192      *
193      *****************************************************************/
194         public  void avisoDerrota() {
195             // comentario que se hace en la película juegos de guerra
196             // que tiempos!!
197             mensaje ="¿Otra partidita, profesor Falken?"  ;
198             esperaTecla  = true;
199         }
200 
201     //*****************************************************************
202     /** Descrip: Aviso que la nave defensora destruye todos los aliens
203      *
204      *  @Params: nada.
205      *  @Return: nada
206      *
207      *****************************************************************/
208         public  void avisoVictoria() {
209             // comentario que se hace en la película juegos de guerra
210             // que tiempos!!
211     mensaje ="Un juego muy extraño,¿Qué tal una partidita de ajedrez?"    ;
212             esperaTecla  = true;
213         }
```

```
214
215    //*************************************************************
216    /** Descrip: Aviso que un alien ha sido destruido
217     *
218     *  @Params: nada.
219     *  @Return: nada
220     *
221     *************************************************************/
222        public void avisoMataAliens () {
223            // decrementa los alien, si no quedan más, el jugador ha ganado
224            cuentaAliens --;
225            if (cuentaAliens  == 0) {
226                avisoVictoria ();
227            }
228            // si todavía hay algunos alien entonces
229            // acelerar todos los que quedan
230            for (int i=0;i<entidades. size();i++) {
231                movimiento entity   = (movimiento ) entidades. get(i);
232                if (entity instanceof  Aliens ) {
233                    // acelerar un 3%
234            entity. setHorizontal (entity. getVeloHorizontal () * 1.03);
235                }
236            }
237        }
238
239    //*************************************************************
240    /** Descrip: laser del jugador
241     *
242     *  @Params: nada.
243     *  @Return: nada
244     *
245     *************************************************************/
246        public void verificaDisparo () {
247            // verifica tiempo de espera del disparo
248            if (System .currentTimeMillis () - ultimoDis  < intervadoNave ) {
249                return ;
250            }
251            // si esperamos lo suficiente, creamos un Nuevo disparo
252            // y tomamos el tiempo
253            ultimoDis  = System .currentTimeMillis ();
254            // Posicion inicial disparo de nave
255            // naveDef.getX()+13,naveDef.getY()-10
256            laser shot  = new laser
257            (this,"laser.gif" ,naveDef. getX()+13,naveDef. getY()-10);
258            shot.th. start();
259            entidades. add(shot );
260        }
261
262    //*************************************************************
263    /** Descrip: El método run() es el corazón de cualquier "Thread"
264     *           y contiene las tareas de ejecución, la acción
265     *           sucede dentro del método run(), además es el
266     *           bucle del juego, este bucle se ejecuta durante todo el
267     *           juego siendo reponsable de todas las actividades
268     *           Calcular el tiempo desde el último bucle
269     *           Procesar las entradas del jugador
270     *           Mover todo basado en el tiempo desde el último bucle
271     *           Dibujar todo lo que hay en pantalla
272     *           Actualizar buffers para que la nueva imagen sea visible
273     *           Verifica entradas
274     *           Actualiza los eventos
275     *
276     *  @Params: nada.
277     *  @Return: nada.
278     *
279     *************************************************************/
280        public void run() {
281            long lastLoopTime  = System .currentTimeMillis ();
282            // Inicializa y mantiene el bucle hasta el fin del juego
283            while (ejecucionJuego ) {
284                //calcular tiempo desde la última actualización, esto
```

```
285                 //se utiliza para calcular en qué medida las entidades deberán
286                 //mover este bucle
287                 long delta = System.currentTimeMillis() - lastLoopTime;
288                 lastLoopTime = System.currentTimeMillis();
289                 // Encuentra el contexo para la aceleración de gráficos
290                 Graphics2D g = (Graphics2D) estrategia.getDrawGraphics();
291                 g.setColor(Color.black);
292                 g.fillRect(0,0,800,600);
293
294                 if (!esperaTecla) {
295                     for (int i=0;i<entidades.size();i++) {
296                         movimiento entity = (movimiento) entidades.get(i);
297                         entity.gestionMovimiento(delta);
298                     }
299                 }
300                 // Dibuja las enidades del juego
301                 for (int i=0;i<entidades.size();i++) {
302                     movimiento entity = (movimiento) entidades.get(i);
303                     entity.dibujar(g);
304                 }
305
306                 // fuerza bruta colisiones, comparar cada entidad con las
307                 // demás. Si cualquiera de ellas chocan notifica que
308                 // ambas entidades han colisionado
309                 for (int p=0;p<entidades.size();p++) {
310                     for (int s=p+1;s<entidades.size();s++) {
311                         movimiento yo = (movimiento) entidades.get(p);
312                         movimiento el = (movimiento) entidades.get(s);
313                         if (yo.colisionaCon(el)) {
314                             yo.haColisionado(el);
315                             el.haColisionado(yo);
316                         }
317                     }
318                 }
319                 // Eliminar entidades marcadas
320                 entidades.removeAll(quitamosLista);
321                 quitamosLista.clear();
322                 // Si un evento ha indicado que debería ejecutarse
323                 // la lógica del juego entonces se solicita a cada entidad
324                 if (logicaAplicaJuego) {
325                     for (int i=0;i<entidades.size();i++) {
326                         movimiento entity = (movimiento) entidades.get(i);
327                         entity.descender();
328                     }
329                     logicaAplicaJuego = false;
330                 }
331                 // Esperamos que se presione una tecla para comenzar el juego
332                 // nos lo muestra el mensaje mensaje
333                 if (esperaTecla) {
334                     g.setColor(Color.white);
335                     g.drawString(mensaje,(800-g.getFontMetrics().
336                      stringWidth(mensaje))/2,250);
337                     g.drawString("Presionar una tecla",(800-g.getFontMetrics
338                      stringWidth("Presionar una tecla"))/2,300);
339                 }
340                 g.dispose();
341                 estrategia.show();
342                 // Resolver el movimiento de la nave En primer lugar asumir
343                 // que la nave no se mueve. Si una tecla de cursor se
344                 // presiona entonces actualizar el movimiento
345                 naveDef.setHorizontal(0);
346                 if ((teclaIzd) && (!teclaDer)) {
347                     naveDef.setHorizontal(-velocidadMovi);
348                 } else if ((teclaDer) && (!teclaIzd)) {
349                     naveDef.setHorizontal(velocidadMovi);
350                 }
351                 //Si estamos presionando fuego, entoces dispara
352                 if (teclaDis) {
353                     verificaDisparo();
354                 }
355                 // Nos tomamos una pausa, debería ejecutarse a 100 fps
```

```
356                 try { Thread.sleep(10); } catch (Exception e) {}
357
358                 // Cuenta el numero de disparos de los aliens
359                 int numAliensTiro = 0;
360                 for (int i=0;i<entidades.size();i++) {
361                    movimiento entity = (movimiento) entidades.get(i);
362                     if (entity instanceof misil) {
363                      numAliensTiro ++;
364                     }
365                 }
366                 // Cuando comienza el juego los aliens
367                 // empiezan a disparar aleatoriamente
368                 if (Math.random()<0.03) {
369                     for (int i=0;i<entidades.size();i++) {
370                    movimiento entity = (movimiento) entidades.get(i);
371                     if (entity instanceof Aliens) {
372                     // Frecuencia de disparo
373                     if (Math.random()<0.02 && numAliensTiro <=5) {
374                     // Posicion inicial disparo de los aliens:
375                    misil alienTiro = new misil
376                     (this, "misil.gif", entity.getX()+13, entity.getY()+25);
377                            entidades.add(alienTiro);
378                            alienTiro.th.start();
379                              }
380                         }
381                     }
382                 }
383             }
384         }
385
386     //*****************************************************************
387     /** Descrip: Clase que recoje las pulsaciones del teclado, las
388      *           habituales izquierda, derecha y disparo.
389      *           Se impementa como una clase interna
390      *  @Params: nada.
391      *  @Return: nada.
392      *  @author Antonio Rivero
393      *
394      *****************************************************************/
395         private class KeyInputHandler extends KeyAdapter {
396             // Número de teclas presionadas mientras
397             // esperamos presionar una tecla
398             private int pressCount = 1;
399
400     //*****************************************************************
401     /** Descrip: Una tecla ha sido presionada
402      *  @Params: e. Detalles de la tecla presionada
403      *  @Return: nada.
404      *
405      *****************************************************************/
406             public void keyPressed(KeyEvent e) {
407                 // Esperamos presionar una tecla
408                 if (esperaTecla) {
409                     return;
410                 }
411                 if (e.getKeyCode() == KeyEvent.VK_O) {
412                    teclaIzd = true;
413                 }
414                 if (e.getKeyCode() == KeyEvent.VK_P) {
415                    teclaDer = true;
416                 }
417                 if (e.getKeyCode() == KeyEvent.VK_SPACE) {
418                    teclaDis = true;
419                 }
420             }
421
422     //*****************************************************************
423     /** Descrip: Una tecla ha sido liberada
424      *  @Params: e. Detalles de la tecla liberada
425      *  @Return: nada.
426      *
```

```
427     ************************************************************************/
428             public void keyReleased(KeyEvent e) {
429                 // Esperamos liberar una tecla
430                 if (esperaTecla ) {
431                     return;
432                 }
433                 if (e.getKeyCode() == KeyEvent.VK_O ) {
434                    teclaIzd   = false;
435                 }
436                 if (e.getKeyCode() == KeyEvent.VK_P ) {
437                    teclaDer   = false;
438                 }
439                 if (e.getKeyCode() == KeyEvent.VK_SPACE ) {
440                    teclaDis   = false;
441                 }
442             }
443
444
445     //************************************************************************
446     /** Descrip: Una tecla ha sido pulsada
447      *  @Params: e. Detalles de la tecla pulsada
448      *  @Return: nada.
449      *
450      ************************************************************************/
451             public void keyTyped(KeyEvent e) {
452                 if (esperaTecla ) {
453                     if (pressCount == 1) {
454                        esperaTecla   = false;
455                         empezar();
456                        pressCount   = 0;
457                     } else {
458                        pressCount  ++;
459                     }
460                 }
461                 // si pulsamos escape, salimos del juego
462                 if (e.getKeyChar() == 27) {
463                     System.exit(0);
464                 }
465             }
466         }
467
468     //************************************************************************
469     /** Descrip: Método stop()
470      *  @Params: Nada
471      *  @Return: Nada
472      *
473      ************************************************************************/
474         public void stop(){
475             if (th!=null) th.stop();
476         }
477
478     //************************************************************************
479     /** Descrip: Punto de entrada al juego, se crea una instancia de
480      *           clase que inicializa la ventana de juego y el bucle
481      *  @Params: argv. Argumentos pasados al juego
482      *  @Return: nada.
483      *
484      ************************************************************************/
485         public static void main(String argv[]) {
486             invader g  =new invader();
487             // Comienzo del juego, el método no devuelve nada hasta
488             // que finaliza el juego, de ahí que utilicemos el
489             // principal thread para ejecutar el juego
490             g.run();
491         }
492     }
```

```
1    /*************************************************************
2    /*************************************************************
3    /*
4    /*  Esta clase representa un laser de la nave defensora
5    /*
6    /*  @autor Antonio Rivero
7    /*
8    /*************************************************************
9    /*************************************************************/
10   public class laser extends disparo {
11
12   //*************************************************************
13   /** Descrip: Constructor de clase.
14    *           Crea un nuevo laser de la nave defensora
15    *
16    * @Params: juego Representa el laser que creamos
17    * @Params: ref Referencia del sprite para mostrar la nave
18    * @Params: x Localización inicial del laser, variable x
19    * @Params: y Localización inicial del laser, variable y
20    *
21    *************************************************************/
22       public laser(invader juego, String sprite, int x,int y) {
23           super(juego,sprite, x, y );
24           dy = -800; // Velocidad del laser
25       }
26   }
```

```
1    /*************************************************************
2    /*************************************************************
3    /*
4    /*  Esta clase representa un disparo del alien
5    /*
6    /*  @autor Antonio Rivero
7    /*
8    /*************************************************************
9    /*************************************************************/
10   public class misil extends disparo {
11
12    //*************************************************************
13   /** Descrip: Constructor de clase.
14    *           Crea un nuevo disparo del alien
15    *
16    * @Params: juego Representa el misil que creamos
17    * @Params: ref Referencia del sprite para mostrar del alien
18    * @Params: x Localización inicial del disparo del alien, variable x
19    * @Params: y Localización inicial del disparo del alien, variable y
20    *
21    *************************************************************/
22     public misil(invader juego,  String sprite,  int x, int y) {
23       super(juego,sprite, x, y );
24       dy = 100; // Velocidad del misil
25     }
26
27    //*************************************************************
28   /** Descrip: Controla la trayectoria del disparo
29    *
30    *  @Params: delta. Tiempo transcurrido desde el último movimiento (ms)
31    *  @Return: Nada
32    *
33    *************************************************************/
34     public void gestionMovimiento (long delta) {
35       super.gestionMovimiento (delta); // movimiento normal
36       if (y > 600) {                   // si sale fuera de la pantalla,
37         juego. quitarEntidad (this);    //lo quitamos
38       }
39     }
40   }
```

```
1    /**************************************************************************
2    **************************************************************************
3    **
4    **  Importaciones generales del sistema
5    **
6    **************************************************************************
7    **************************************************************************/
8    import java.awt.Graphics;
9    import java.awt.Rectangle; // para las colisiones
10
11   /**************************************************************************
12   /*************************************************************************
13   /*
14   /*  Esta clase representa algunos elementos que aparecen en el juego,
15    *  la entidad tiene la responsabilidad de resolver las colisiones y
16    *  movimientos basados en un conjunto de características definidas
17    *  en cualquiera de las subclases. Utilizo double para la
18    *  localización de los pixels para dar más precisión.
19    *  @autor Antonio Rivero
20   /*
21   /*************************************************************************
22   /*************************************************************************/
23   public abstract class movimiento implements Runnable{
24       /** Localización actual "x" de esta entidad*/
25       protected double x;
26       /** Localización actual "y" de esta entidad*/
27       protected double y;
28       /** Sprite que representa esta entidad */
29       protected Sprite sprite;
30       /** Velocidad horizontal actual de la entidad(pixels/sec) */
31       protected double dx;
32       /** Velocidad vertical actual de la entidad(pixels/sec) */
33       protected double dy;
34       /** Rectangulo usado por la entidad durante la colisión */
35       private Rectangle yo = new Rectangle();
36       /** Rectangulo usado por otra entidad durante la colisión */
37       private Rectangle el = new Rectangle();
38       /** Declaración del Thread*/
39       public Thread th = new Thread(this);
40       public boolean esperaTecla = true;
41       public invader juego;
42
43   //************************************************************
44   /** Descrip: Constructora de la clase, referenciada sobre un sprite y
45    *           la localización de este.
46    *  @Params: ref Referencia del sprite para mostrar por pantalla
47    *  @Params: x Localización inicial de la entidad, variable x
48    *  @Params: y Localización inicial de la entidad, variable y
49    *  @Return: nada.
50    *
51    ************************************************************/
52       public movimiento(invader juego, String ref,int x,int y) {
53           this.juego = juego;
54           this.sprite = SpriteAux.get().obtenerSprite(ref);
55           this.x = x;
56           this.y = y;
57       }
58
59   //************************************************************
60   /** Descrip: Solicita a la entidad el movimiento respecto al tiempo
61    *           trancurrido
62    *  @Params: delta. Tiempo transcurrido en ms
63    *  @Return: nada.
64    *
65    ************************************************************/
66       public void gestionMovimiento(long delta) {
67           // carga el movimiento de las entidades
68           x += (delta * dx) / 1000;
69           y += (delta * dy) / 1000;
70       }
71
```

```
72      //*****************************************************************
73      /** Descrip: Crea la posición horizontal del sprite
74       *  @Params: dy. Velocidad horizontal de la entidad (pixels/sec)
75       *  @Return: nada.
76       *
77       *****************************************************************/
78          public void setHorizontal(double dx) {
79              this.dx = dx;
80          }
81
82      //*****************************************************************
83      /** Descrip: Crea la posición vertical del sprite
84       *  @Params: dy. Velocidad vertical de la entidad (pixels/sec)
85       *  @Return: nada.
86       *
87       *****************************************************************/
88          public void setVertical(double dy) {
89              this.dy = dy;
90          }
91
92      //*****************************************************************
93      /** Descrip: Devuelve la posición del sprite
94       *  @Params: nada.
95       *  @Return: Velocidad horizontal de la entidad (pixels/sec)
96       *
97       *****************************************************************/
98          public double getVeloHorizontal() {
99              return dx;
100         }
101
102     //*****************************************************************
103     /** Descrip: Devuelve la posición del sprite
104      *  @Params: nada.
105      *  @Return: Velocidad vertical de la entidad (pixels/sec)
106      *
107      *****************************************************************/
108         public double getVeloVertical() {
109             return dy;
110         }
111
112     //*****************************************************************
113     /** Descrip: Dibuja los gráficos de esta entidad
114      *  @Params: g. Graficos del dibujo
115      *  @Return: nada.
116      *
117      *****************************************************************/
118         public void dibujar(Graphics g) {
119             sprite.dibujar(g,(int) x,(int) y);
120         }
121
122     //*****************************************************************
123     /** Descrip: Trabaja con la lógica asociada a esta entidad
124      *           Este método se llama frecuentemente
125      *  @Params: nada.
126      *  @Return: nada.
127      *
128      *****************************************************************/
129         public void descender() {
130         }
131
132     //*****************************************************************
133     /** Descrip: Localiza la posición de x
134      *  @Params: nada.
135      *  @Return: La situación de la variable x de la entidad
136      *
137      *****************************************************************/
138
139         public int getX() {
140             return (int) x;
141         }
142
```

```
143   //*****************************************************************
144   /** Descrip: Localiza la posición de y
145    *  @Params: nada.
146    *  @Return: La situación de la variable y de la entidad
147    *
148    *****************************************************************/
149       public int getY() {
150           return (int) y;
151       }
152
153   //*****************************************************************
154   /** Descrip: Verifica si la entidad ha colisionado con otra
155    *  @Params: other. Entidad que verifica la colision
156    *  @Return: Verdadero si las entidades colisionan unas con otras
157    *
158    *****************************************************************/
159       public boolean colisionaCon (movimiento other ) {
160           yo. setBounds ((int) x,(int) y,sprite. getAncho (),sprite. getAlto ())
161           el. setBounds ((int) other.x, (int) other.y,other.sprite.  getAncho ()
162           other.sprite.  getAlto ());
163           return yo.intersects (el);
164       }
165
166   //*****************************************************************
167   /** Descrip: Aviso a la entidad que ha colisionado con otra
168    *  @Params: other. entidad con la que colisionamos
169    *  @Return: Nada
170    *
171    *****************************************************************/
172       public  void haColisionado (movimiento other ){
173           //si un misil del alien colisiona con
174           //la nave guardian se eliminan los dos
175           if (other instanceof guardian ) {
176               juego. quitarEntidad (this); // Eliminamos nave guardian
177               juego. quitarEntidad (other); // Eliminamos misil
178               juego. avisoDerrota ();
179           }
180           //si un laser de la nave colisiona con
181           //un alien se alnulan los dos.
182           if (other instanceof laser ){
183               juego. quitarEntidad (this);
184               juego. quitarEntidad (other);
185               juego. avisoMataAliens ();
186           }
187       }
188
189   //*****************************************************************
190   /** Descrip: El método run() es el corazón de cualquier "Thread"
191    *           y contiene las tareas de ejecución, la acción
192    *           sucede dentro del método run().
193    *  @Params: Nada
194    *  @Return: Nada
195    *
196    *****************************************************************/
197           public void run() {
198             long lastLoopTime  = System.currentTimeMillis ();
199             while (true) {
200              esperaTecla   = juego. getPaused ();
201               long delta = System.currentTimeMillis () - lastLoopTime;
202              lastLoopTime   = System.currentTimeMillis ();
203               if (!esperaTecla ) {
204                    gestionMovimiento (delta);
205               }
206               try {
207                    Thread.sleep(10);
208                    } catch (Exception e) {
209                        break;
210                      }
211             }}}
```

```
1    /*************************************************************
2    /*************************************************************
3    /*
4    /*  Esta clase representa nave
5    /*
6    /*  @autor Antonio Rivero
7    /*
8    /*************************************************************
9    /*************************************************************/
10   public abstract class nave extends movimiento  {
11
12   //*********************************************************************
13   /** Descrip: Constructor de clase.
14    *           Crear una nueva entidad para representar las naves
15    *
16    * @Params: juego Representa la nave que creamos
17    * @Params: ref Referencia del sprite para mostrar la nave
18    * @Params: x Localización inicial de la nave, variable x
19    * @Params: y Localización inicial de la nave, variable y
20    *
21    *********************************************************************/
22       public nave(invader juego, String ref,int x,int y) {
23            super(juego,ref,x,y );
24       }
25   }
```

```
1    /*************************************************************************
2    *************************************************************************
3    **
4    **  Importaciones generales del sistema
5    **
6    *************************************************************************
7    *************************************************************************/
8    import java.awt.Graphics;
9    import java.awt.Image;
10
11   //************************************************************************
12   /** Descrip: Clase Sprite. Un sprite se visualiza en pantalla.
13    *           Un sprite no da información, únicamente la imagen
14    *           y no la localización.
15    *           Esto nos permite usar un sprite simple en diferentes
16    *           partes del juego sin tener que almacenar muchas copias
17    *           de esa imagen
18    *  @autor Antonio Rivero
19    *
20    ************************************************************************/
21   public class Sprite {
22       /** imagen dibujada por el sprite */
23       private Image imagen;
24
25   //************************************************************************
26   /** Descrip: Constructor de la clase
27    *           Se crea un nuevo sprite basado en una imagen
28    *  @Params: imagen. imagen del sprite
29    *
30    ************************************************************************/
31       public Sprite(Image imagen) {
32           this.imagen = imagen;
33       }
34
35   //************************************************************************
36   /** Descrip: getWidth determina anchura del sprite
37    *           Si no se conoce devuelve -1
38    *
39    *  @Params: nada.
40    *  @Return: anchura en pixeles de este sprite
41    *
42    ************************************************************************/
43       public int getAncho() {
44           return imagen.getWidth(null);
45       }
46
47   //************************************************************************
48   /** Descrip: getHeight determina la altura del sprite
49    *           Si no se conoce devuelve -1
50    *  @Params: nada.
51    *  @Return: altura en pixeles de este sprite
52    *
53    ************************************************************************/
54       public int getAlto() {
55           return imagen.getHeight(null);
56       }
57
58   //************************************************************************
59   /** Descrip: dibuja el frame en pantalla
60    *
61    *  @Params: g dibuja el sprite
62    *  @Params: x Localización inicial del sprite, variable x
63    *  @Params: y Localización inicial del sprite, variable y
64    *  @Return: nada.
65    *
66    ************************************************************************/
67       public void dibujar(Graphics g,int x,int y) {
68           g.drawImage(imagen,x,y, null);
69       }
70   }
```

```
1    /*****************************************************************
2    *****************************************************************
3    **
4    **  Importaciones generales del sistema
5    **
6    *****************************************************************
7    *****************************************************************/
8    import java.awt.GraphicsConfiguration ;
9    import java.awt.GraphicsEnvironment ;
10   import java.awt.Image;
11   import java.awt.Transparency ;
12   import java.awt.image.BufferedImage ;
13   import java.io.IOException ;
14   import java.net.URL;
15   import java.util.HashMap ;
16   import javax.imageio.ImageIO; // carga la imagen
17
18   //***************************************************************
19   /** Descrip: Clase SpriteAux. Se encarga de los recursos para los
20    *           Sprites en el juego. Es importante el cómo y dónde
21    *           utilizamos los recursos
22    *           Responsable carga y cache de los sprites
23    *
24    *  @autor Antonio Rivero
25    *
26    ***************************************************************/
27   public class SpriteAux {
28       /** Instancia simple de esta clase */
29       private static SpriteAux single = new SpriteAux();
30
31   //***************************************************************
32   /** Descrip: Crea una instancia simple de la clase
33    *
34    *  @Params: nada
35    *  @Return: Una instancia simple de la clase
36    *
37    ***************************************************************/
38       public static SpriteAux get() {
39           return single;
40       }
41       /** cache del sprite */
42       private HashMap sprites = new HashMap();
43
44   //***************************************************************
45   /** Descrip: Recupera un sprite
46    *
47    *  @Params: ref. Referencia a la imagen usada por el sprite
48    *  @Return: Una instancia que contiene la peticion de la
49    *           aceleracion grafica del sprite
50    *
51    ***************************************************************/
52       public Sprite obtenerSprite(String ref) {
53           // si ya hemos conseguido un sprite en la caché
54           // entonces podemos devolverlo
55           if (sprites.get(ref) != null) {
56               return (Sprite) sprites.get(ref);
57           }
58           BufferedImage sourceImage = null;
59           try {
60               // ClassLoader.getResource() nos asegura el sprite
61               // en el sitio adecuado.
62               URL url = this.getClass().getClassLoader().getResource(ref);
63               if (url == null) {
64                   fallo("No se puede encontrar la referencia: " +ref);
65               }
66               // utiliza ImageIO para leer la imagen
67              sourceImage = ImageIO.read(url);
68           } catch (IOException e) {
69               fallo("Fallo al cargar: " +ref);
70           }
71           // crea una aceleración gráfica del sprite
```

```
72            GraphicsConfiguration  gc = GraphicsEnvironment .
73            getLocalGraphicsEnvironment  ().getDefaultScreenDevice  ().
74            getDefaultConfiguration  ();
75            Image  image  = gc.createCompatibleImage  (sourceImage.
76            getWidth (),sourceImage. getHeight (),Transparency .BITMASK );
77            // dibuja la imagen con aceleración gráfica
78           image. getGraphics ().drawImage (sourceImage, 0,0,null );
79            // crear un sprite y lo añade al caché
80           Sprite sprite   = new  Sprite (image );
81           sprites. put(ref,sprite );
82            return  sprite;
83        }
84
85    //*********************************************************************
86    /** Descrip: Tratamiento de recursos
87     *
88     *  @Params: message. mensaje lanzado a la ventana
89     *  @Return: nada
90     *
91     *********************************************************************/
92        private  void  fallo(String  message ) {
93            System .err. println (message );
94            System .exit (0);
95        }
96    }
```