MOO PANG!: DOCUMENTO DE ANÁLISIS

INTRODUCCIÓN

Una pequeña empresa de videojuegos nos pide una versión sencilla del conocido videojuego *Pang!* (ver figura). En este videojuego, unas burbujas van cayendo del cielo, y el jugador debe explotarlas antes de que se acumulen demasiadas burbujas y toquen al jugador, momento en el cual se acabará la partida.

Figura 1: versión original del juego Super Pang!

Para explotar las bolas, el jugador lanzará unos ganchos atados a una cuerda. Si la bola toca la punta del gancho o la cuerda, ésta se dividirá en dos bolas más pequeñas que al ser tocadas por el gancho, se dividirán en otras bolas más pequeñas; este proceso se repetirá sucesivamente hasta que las bolas sean tan pequeñas que, al ser tocadas por el gancho, se eliminan completamente.

Los principales objetivos de este ejercicio son dos:

1. Familiarizarse con el proceso de análisis y diseño de una aplicación real.
2. Aplicar los conocimientos teóricos de diseño y programación orientada a objetos, y aprender técnicas para la creación de aplicaciones interactivas.

Entre los objetivos de este ejercicio **no** está el de producir un videojuego completo con calidad profesional, ya que para ello se requeriría un equipo de trabajo más extenso compuesto por programadores, grafistas, músicos, diseñadores de juegos, e incluso *probadores* que se dedicaran a jugar al juego para descubrir fallos a corregir.

OBJETIVOS

Creado por Mario Macías y distribuido gratuitamente bajo licencia Creative Commons.

Después de una entrevista con el cliente (ficticio), se han identificado los siguientes objetivos para la aplicación:

- **O1**: Creación de una versión sencilla del videojuego *Pang!*
- **O2**: Para ampliar al máximo la base de usuarios potenciales, la aplicación debe estar disponible para varias plataformas: PC, Mac, teléfonos móviles medios, *smartphones*, e incluso navegadores web.
- **O3**: El cliente pide que el tiempo de desarrollo de las diferentes versiones para todas las plataformas anteriormente mencionadas debe ser corto. Además, el coste de desarrollo debe ser barato.

REQUISITOS

Basándonos en los objetivos (y en nuestra amplia experiencia como desarrolladores de videojuegos), se pueden deducir los siguientes requisitos para la aplicación:

REQUISITOS FUNCIONALES

- **RF1**: La aplicación será sencilla. Unas bolas irán cayendo del techo y no pueden tocar al jugador. El jugador irá lanzando ganchos para reventarlas. Cuando una bola toque al jugador, la partida acabará.
- **RF2**: Cada partida tendrá una puntuación asociada. Al inicio será 0, y se irá incrementando en uno cada vez que un gancho (o la cuerda que lleva atada) toque una bola.
- **RF3**: Al iniciar la aplicación, antes de que el jugador empiece a jugar, deberá mostrarse una pantalla de presentación con el título del juego y un mensaje del estilo “Pulsa espacio para empezar partida”. Cuando el jugador pulse espacio, la partida empezará.
- **RF4**: Al finalizar la partida, se mostrará un mensaje de fin de juego, así como la puntuación de la partida. Al menos debe durar unos segundos para que el jugador tenga tiempo a leer su puntuación. Después del mensaje de fin de juego, se volverá a la presentación inicial.

REQUISITOS NO FUNCIONALES

- **RNF1**: La aplicación será desarrollada en Java. Esto nos permitirá ejecutar una misma aplicación en diferentes sistemas de escritorio, y adaptarla rápidamente a plataformas móviles con pocos cambios en el código.
- **RNF2**: La aplicación debe ser divertida para el usuario, por lo que se prestará especial atención en proveer una jugabilidad y dificultad ajustadas para el usuario.
- **RNF3**: Con tal de facilitar el mantenimiento y una posible actualización de la aplicación en el futuro, el código debe ser modular y tener un adecuado diseño.

Creado por Mario Macías y distribuido gratuitamente bajo licencia Creative Commons.

- **RNF4**: Con tal de cumplir los motivos académicos por los que ha sido creado este ejercicio, el código deberá ser entendible en la medida de lo posible por los alumnos de la asignatura MOO. Esto implica utilizar nombres descriptivos para clases, métodos y atributos, así como mantener el código documentado y comentado.

CASOS DE USO

CU1: PARTIDA A MOO PANG!

- **Actor**: el usuario que ejecuta *MOO Pang!* para jugar una partida
- **Precondición**: El usuario ha visto la pantalla de título y ha pulsado espacio para empezar a jugar.
- **Postcondición**: El usuario ve la pantalla de fin de juego, y a continuación vuelve a la pantalla de título.
- **Escenario Principal**

1. El personaje principal del juego aparece en el centro de la pantalla.
2. Van cayendo bolas grandes del cielo con una frecuencia baja al principio, y esta va progresivamente incrementándose para hacer más difícil la partida.
3. Las bolas tienen una velocidad horizontal constante, y una velocidad vertical que se acelera con el tiempo, simulando una atracción gravitatoria.
   3.1. Cuando una bola toca el suelo, rebota hacia arriba (su velocidad vertical, sea cual fuere, cambia de signo).
   3.2. Cuando una bola toca una de las paredes laterales, rebota hacia el otro lado (su velocidad horizontal cambia de signo).
4. El personaje principal puede moverse a izquierda o derecha, si el usuario/jugador ha apretado la tecla de izquierda o derecha, respectivamente.
5. Si el usuario/jugador pulsa la barra espaciadora, se intenta lanzar un gancho.
   5.1. Si ya hay dos ganchos visualizándose en pantalla, el jugador no podrá lanzar más.
   5.2. Si hay uno o ningún gancho en pantalla, el jugador lanzará otro.
6. Si un gancho toca el techo, éste desaparece.
7. Si un gancho toca una bola, el gancho desaparece.
   7.1. Si el radio de la bola es menor o igual a un umbral mínimo, la bola desaparece.
   7.2. Si el radio de la bola es mayor al umbral mínimo, la bola se divide en dos bolas de radio igual a un 60% de la bola anterior. Una bola seguirá la dirección de la bola previa, y la otra su dirección contraria.
8. Si una bola toca al jugador, la partida finaliza.

Creado por Mario Macías y distribuido gratuitamente bajo licencia Creative Commons.

MOO PANG!: DISEÑO TÉCNICO

CÓMO FUNCIONA UN VIDEOJUEGO

En un videojuego, diferentes objetos de diversos tipos se mueven simultáneamente en pantalla. Por ejemplo, en *MOO Pang!* se mueven continuamente por pantalla varias bolas de distintos tamaños, el personaje que representa al jugador, y los ganchos que éste lanza. Esto implica que deben hacerse muchas tareas de diversa índole, pertenecientes a objetos diversos, de manera **simultánea**. Sin embargo, con los conocimientos que hemos obtenido la asignatura MOO, no somos capaces de hacer que un programa haga varias acciones en paralelo (por ejemplo, que una bola se mueva de un lado a otro mientras también se mueve el jugador).

En realidad, un videojuego funciona en cierto modo como una película: es una sucesión visual de **fotogramas** ligeramente diferentes al fotograma anterior (ver ejemplo en Figura 2). Cada objeto móvil en pantalla tiene unas propiedades, como puede ser la posición, la forma o el color. Para cada fotograma, dicho objeto debe recalcular ligeramente su posición en función del comportamiento para el que está programado (por ejemplo, la bola de la Figura 2 recalculará continuamente su posición para ir rebotando por pantalla). Una vez ha recalculado su posición, se redibujará en la pantalla de tal manera que el nuevo dibujo refleja los cambios en su estado: posición, color, tamaño… Si actualizamos ligeramente y repintamos todos los objetos en orden para cada fotograma, el observador/jugador tendrá la sensación de que todo se está moviendo a la vez.

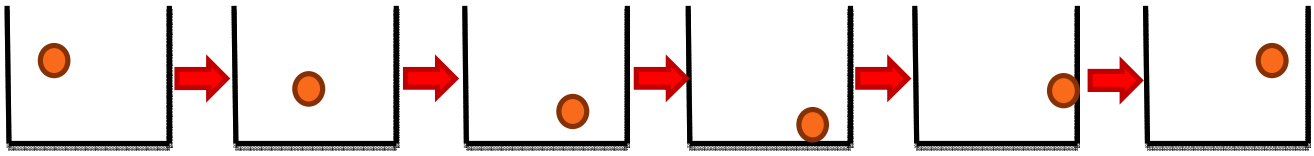
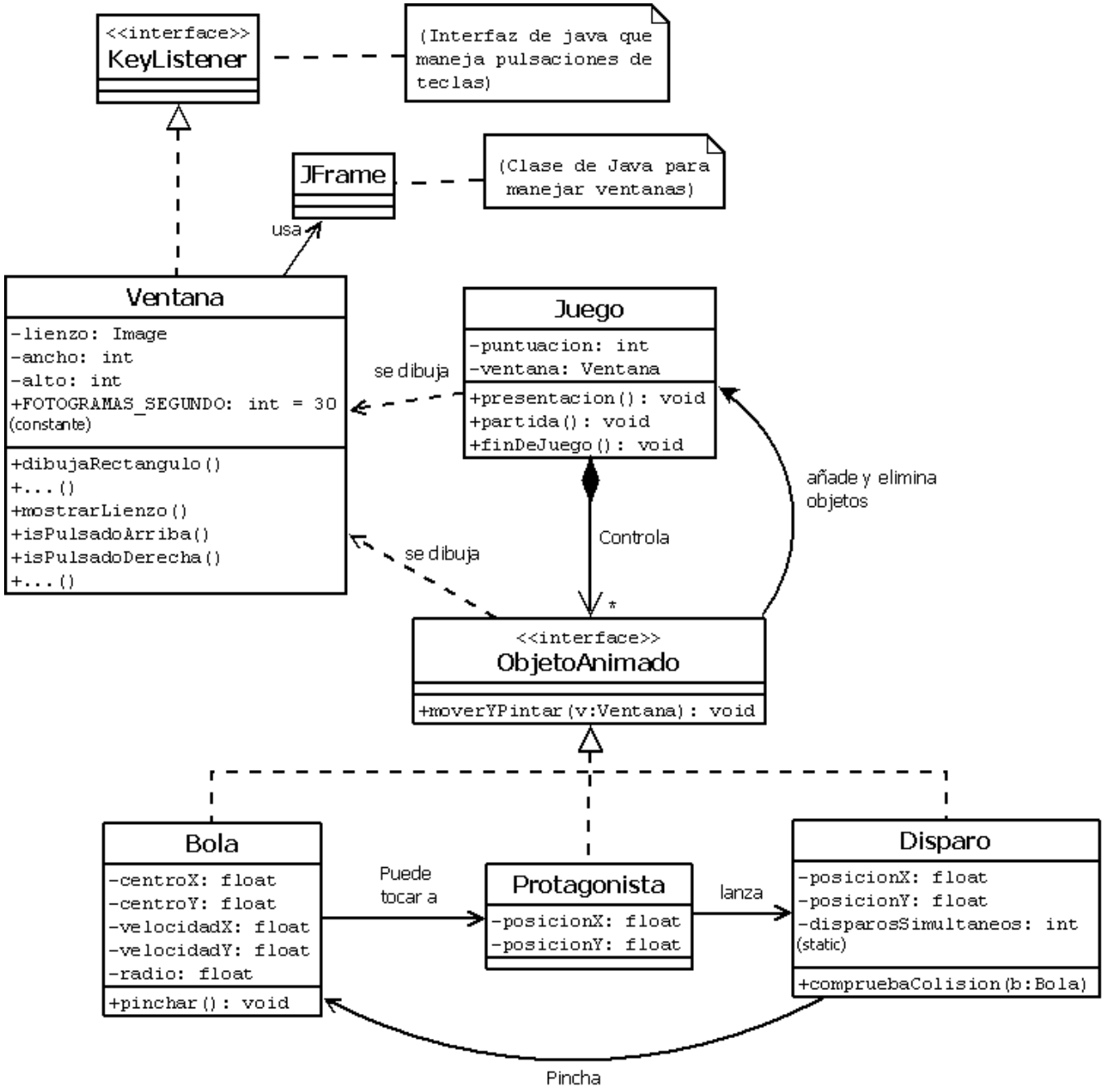


Figura 2: Ejemplo de sucesión de fotogramas para una pelota rebotando por pantalla. En cada fotograma, una bola debe recalcular su posición y repintarse en pantalla, reflejando ese cambio de posición. El observador tendrá, pues, la sensación de que esa pelota se está moviendo.

DIAGRAMA DE CLASES UML

Creado por Mario Macías y distribuido gratuitamente bajo licencia Creative Commons.



Basándonos en el documento de análisis, en lo anteriormente explicado en este documento, y en nuestra experiencia como programadores, se ha propuesto la estructura para el código de *MOO Pang!* reflejada en la Figura 3. En el diagrama de clases UML hay tres elementos principales:

- La clase **Ventana** es una clase creada especialmente para ocultar la complejidad a la hora de crear ventanas y dibujar en ellas, capturar y gestionar las pulsaciones del teclado, y controlar en qué momento se debe redibujar cada fotograma del juego. Como parte del contenido de esta clase está fuera del objetivo de este ejercicio (y de la asignatura de MOO), no se entra en detalle

Creado por Mario Macías y distribuido gratuitamente bajo licencia Creative Commons.

en cómo está implementada. La grandeza de la programación orientada a objetos es que os bastará con conocer sus métodos públicos para utilizarla (de la misma manera que una persona no necesita ser ingeniero en telecomunicaciones para hacer una llamada telefónica).

- La clase **Juego** es la encargada de controlar el estado de la partida (la puntuación, si la partida ha finalizado, etc…). Además, contiene una lista con todos los objetos que se mueven simultáneamente en pantalla. Para cada fotograma, llamará a cada uno de los objetos móviles (que implementan la interfaz ObjetoMovil) para que éstos actualicen ligeramente su posición y se redibujen en un lienzo de dibujo que está oculto a los ojos del jugador: si se dibujaran directamente en la pantalla, el jugador vería como se borran y se dibujan en todo momento, con lo que la pantalla parpadearía y sería incómodo. Cuando todos los objetos hayan acabado de moverse y dibujarse en el lienzo oculto, la clase Juego llama a la clase Ventana para que muestre en su interior el contenido del lienzo, ya actualizado completamente.
- La interfaz **ObjetoMovil** es una interfaz que debe ser implementada por cualquier objeto del juego que deba ser movido: cada una de las instancias de las clases **Protagonista, Disparo** y **Bola**. La interfaz ObjetoMovil solo define un método: **moverYPintar**. Cada clase que implementa dicha interfaz debe implementar este método, donde actualizará su posición ligeramente según el comportamiento deseado, comprobará colisiones entre objetos de otras clases (por ejemplo, un disparo comprobará si colisiona con una bola para partirla en dos), y se redibujará en pantalla. La clase Juego contiene una lista de referencias a ObjetoMovil, que recorrerá en cada fotograma y llamará al método *moverYPintar* de cada instancia.
    - **Pregunta**: ¿Por qué definir ObjetoMovil como una interfaz y no como una superclase o una clase abstracta?
    - **Respuesta**: La aplicación funcionaría igual, pero desde el punto de vista del diseño no sería demasiado correcto. En realidad, el personaje, el disparo y las bolas no tienen nada en común como para tener que pertenecer a la misma "familia" de clases. Lo único que tienen en común es un "punto de contacto", es decir, un método que se llama igual para todos, pero que hace cosas completamente diferentes en cada clase. En estos casos, siempre es mejor definir una interfaz, tanto desde el punto de vista del diseño como de la eficiencia del programa.

CONTENIDO ADICIONAL: DETECTANDO COLISIONES ENTRE LOS DIFERENTES ELEMENTOS DEL JUEGO

Para que la aplicación *MOO Pang!* sea realmente un juego, es necesario que los elementos del juego interactúen entre ellos, de tal manera que el estado de un objeto pueda cambiar el estado de otro. Por ejemplo, si un disparo toca una bola, esta bola se pincha y se dividirá en dos; si una bola toca al jugador, el juego se acaba, si una bola toca la pared o el suelo, ésta rebota. Para ello, es necesario que en cada

Figura 3: Diagrama de clases UML de MOO Pang!

Creado por Mario Macías y distribuido gratuitamente bajo licencia Creative Commons.

fotograma se detecten las **colisiones** entre los diferentes elementos.

Este apartado explica qué métodos se han utilizado para detectar las colisiones entre los diversos objetos que se mueven en el juego. La materia aquí explicada no tiene nada que ver con la programación, el diseño, o cualquier otro tema explicado en la asignatura. Es simplemente una descripción de cómo se ha resuelto un problema concreto desde un punto de vista matemático: una vez se conoce el modelo matemático, programarlo en Java es prácticamente automático.

Recuerda: siempre que te enfrentes a un problema, piensa en cómo lo resolverías tú paso a paso, y ya te preocuparás luego de cómo programarlo en una computadora.

DETECTAR UNA COLISIÓN ENTRE UNA BOLA Y LAS PAREDES

La geometría para detectar la colisión entre una bola y las paredes o el suelo está representada en la Figura 4. Sea una bola, representada por una circunferencia de centro (ox, oy) y radio r, ésta colisionará con las paredes o el suelo si:

- Sea la pared izquierda representada por la coordenada horizontal izqX, la bola colisionará con dicha pared si .
- Sea la pared derecha representada por la coordenada horizontal derX, la bola colisionará con dicha pared si .
- Sea el suelo, representado por la coordenada vertical sueloY, la bola colisionará con el suelo si

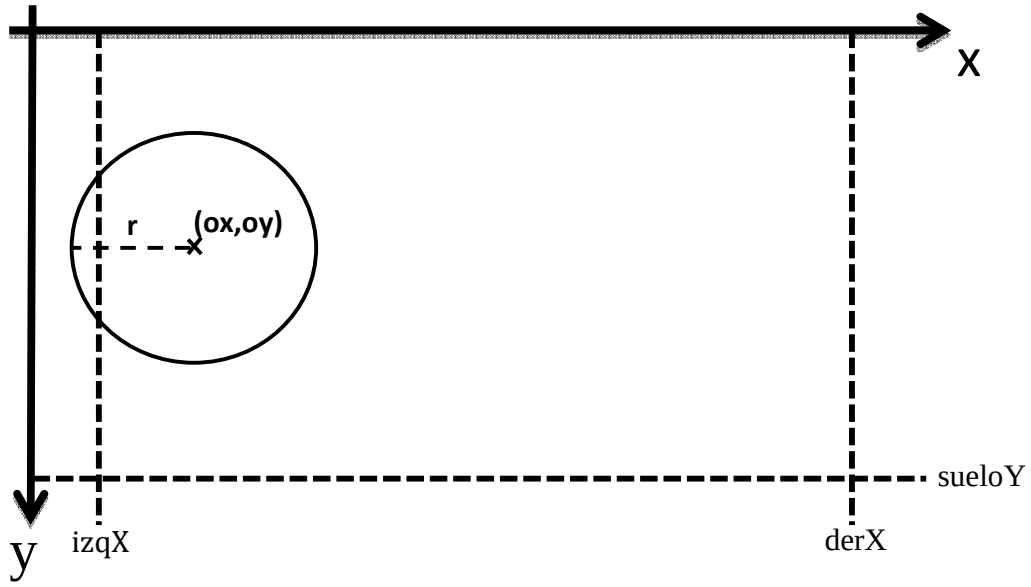


Figura 4: Geometría para la colisión entre una bola y las paredes o el suelo.

Creado por Mario Macías y distribuido gratuitamente bajo licencia Creative Commons.

Si la bola colisiona con alguna pared, se deberá invertir su velocidad horizontal (para que rebote hacia el lado contrario). Si la bola colisiona con el suelo, se deberá invertir su velocidad vertical (para que rebote hacia arriba).

DETECTAR UNA COLISIÓN ENTRE UNA BOLA Y UN DISPARO DEL JUGADOR

La geometría para detectar si una bola colisiona con el disparo de un jugador no es mucho más complicada que la del caso anterior. Para hacerlo de una manera sencilla, el gancho no es más que una línea vertical cuya posición horizontal viene determinada por la coordenada gx, y cuya punta va subiendo a cada momento, con una altura determinada por gy (Figura 5). Dada una circunferencia de radio r y cuyo centro es (ox,oy), la bola representada y el gancho colisionarán si se dan las dos siguientes condiciones:

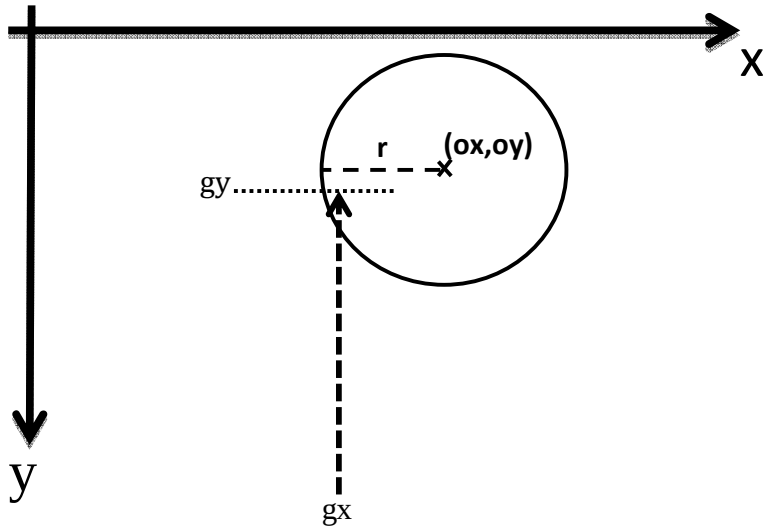


Figura 5: geometría para la collision entre una bola y un gancho.

Recuerda que el símbolo significa 'y' lógico. El código que en la práctica se ha utilizado para programar esta detección de colisión es un poco más complejo, ya que tiene en cuenta el grosor de la cuerda, y la forma triangular del gancho.

DETECTAR UNA COLISIÓN ENTRE UNA BOLA Y EL JUGADOR

Detectar la colisión entre una bola y el jugador se puede hacer de dos maneras diferentes. La manera más compleja, pero más exacta, sería comprobar si la bola colisiona con cada una de las primitivas que componen la silueta del personaje: triángulos, cuadrados, círculos…

Sin embargo, la técnica que se ha usado es más sencilla (aunque menos precisa): se ha definido un rectángulo de tamaño que envuelve parcialmente al personaje, y cuya esquina superior izquierda

Creado por Mario Macías y distribuido gratuitamente bajo licencia Creative Commons.

viene definida por el punto (px,py), tal y como aparece en la figura. La bola colisionará con el personaje si la circunferencia que envuelve a la bola colisiona con el cuadro que envuelve parcialmente al personaje, es decir, si:

Recuerda que significa 'o' lógico y significa 'y' lógico. En la función anterior se comprueba que la bola esté al menos parcialmente dentro de los dos márgenes verticales del rectángulo, pero solo se comprueba que esté por debajo del margen superior. Para este juego, no hay que comprobar que esté también por encima del margen inferior, ya que ésta nunca podrá estar completamente debajo del jugador, porque el suelo se lo impide.

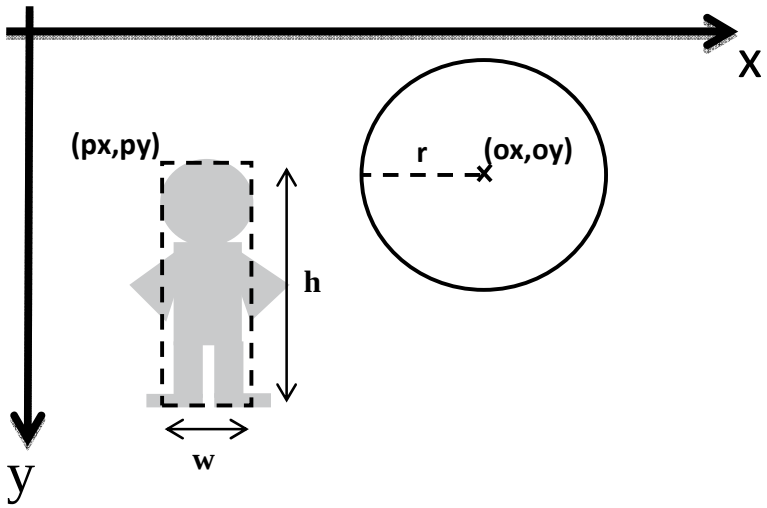


Figura 6: geometría de colisión entre el jugador, que viene enmarcado dentro de una caja para simplificar, y una bola.

PARA SABER MÁS

Si estás interesado en ampliar tus conocimientos sobre programación de videojuegos en Java, tal vez te resulte interesante mi Proyecto de fin de carrera en Ingeniería Informática: Diseño y creación de aplicaciones de entretenimiento sobre dispositivos móviles: http://mario.site.ac.upc.edu/others/proyecto_ulpgc.pdf

Creado por Mario Macías y distribuido gratuitamente bajo licencia Creative Commons.