Memoria de proyecto

Integración de funcionalidades en The Game

Alejandro Contell Fernández

acontellfernandez@cifpfbmoll.eu

01/04/2025

Palma de Mallorca

Table of Contents

The Game

The Game es un programa desarrollado en Java en el que se aplican conceptos y módulos aprendidos durante el curso de Desarrollo de Aplicaciones
Multiplataforma, como el uso de threads, sockets o interfaces gráficas.

El programa está estructurado en tres secciones principales siguiendo el modelo de desarrollo MVC (Modelo-Vista-Controlador). En este modelo, la lógica del programa se encuentra en el modelo, los gráficos y elementos visuales en la vista, y el control general del funcionamiento recae en el controlador.

A diferencia de un sistema tradicional basado en un servidor central, The Game utiliza un enfoque Peer to Peer (P2P), donde cada instancia del programa actúa simultáneamente como cliente y servidor. Esto permite que varios The Game en ejecución dentro de una red puedan conectarse entre sí e intercambiar información sin necesidad de un servidor principal.

Para implementar este funcionamiento, se ha ampliado la estructura clásica del modelo MVC añadiendo una sección extra dedicada a las comunicaciones, la cual se encarga de gestionar todo el sistema P2P a través de conexiones mediante sockets.

A continuación, se desglosa la arquitectura general de la aplicación para entender cómo se organiza internamente y cómo se dividen sus diferentes módulos.

Controlador Global

The Game separa claramente la sección de comunicaciones (P2P) del resto de la aplicación que funciona de manera local.

El controlador global (clase TG, también conocida como main) es el encargado de iniciar y ejecutar toda la aplicación. Además, actúa como punto central de coordinación entre la lógica local y el sistema de comunicaciones.

Una de sus funciones iniciales es generar automáticamente una cantidad predefinida de objetos del tipo Planeta, que forman parte de la representación visual del juego.

Peer Controller

La sección de comunicaciones se compone de un controlador, un servidor y un cliente. La idea detrás de un sistema P2P es que la aplicación actúe tanto como servidor (escuchando a otros programas The Game que intenten conectarse), como cliente (buscando activamente en la red otros The Game disponibles).

Tanto si un programa externo se conecta a nuestro servidor como si nuestro cliente local encuentra otro The Game, se establece una comunicación bidireccional que permite el intercambio de información entre ambas aplicaciones, lo que permite enviar mensajes de texto para pruebas o incluso objetos del juego, como asteroides u otras entidades visibles.

Gracias a este mecanismo, The Game puede expandirse más allá del entorno local, permitiendo que varios jugadores compartan una experiencia sincronizada entre diferentes dispositivos conectados.

El controlador de comunicaciones (PCT) se encarga de inicializar tanto el servidor como el cliente, y de gestionar su funcionamiento de forma conjunta.

Servidor

El servidor de la aplicación, una vez activado, queda a la espera de conexiones entrantes desde otros programas The Game. Si otro dispositivo lo detecta y se conecta correctamente, se crea un canal de comunicación que permite intercambiar información entre ambas aplicaciones.

Este proceso incluye una selección automática de puertos, lo cual permite que varios The Game puedan ejecutarse en la misma red (e incluso en la misma máquina, usando puertos diferentes).

Cliente

El cliente de la aplicación hace el proceso inverso al servidor. Su función es escanear la red para detectar otros The Game que tengan un servidor activo. Si encuentra uno, intenta establecer una conexión para formar el canal de comunicación.

Una vez conectado, el cliente comprueba periódicamente si la conexión sigue activa. Si el canal sigue disponible, entra en un modo de espera antes de volver a comprobar el estado. Si detecta que la conexión se ha perdido, intenta reconectarse. En caso de no lograrlo, vuelve a iniciar la búsqueda en la red desde cero.

Local Controller

El controlador local (LCT) se encarga de gestionar la parte de la aplicación que funciona en el entorno local del usuario. Aquí es donde se establece la estructura basada en el modelo MVC (Modelo-Vista-Controlador), dividiendo las responsabilidades entre la lógica del programa y su representación visual. El controlador local coordina dos componentes principales:

Local View

La vista local (LV) define la ventana principal del programa, incluyendo todos los elementos visuales que el usuario ve en pantalla. Además de mostrar los objetos gráficos, está preparada para detectar interacciones del usuario, como movimientos del ratón o pulsaciones de teclas.

Local Model

El modelo local (LM) es el encargado de gestionar todos los objetos del juego que se representan y actualizan durante la ejecución. Estos objetos pueden ser tanto estáticos como dinámicos, y se organizan siguiendo una jerarquía clara:

- **VO (Visual Object):** Es la clase base que representa cualquier objeto visual del juego. Sirve como plantilla común para elementos como muros (Wall), que actúan como límites del escenario. Los objetos VO pueden tener una imagen, posición, escala, estado y efectos visuales superpuestos.
- **VOD (Visual Object Dynamic):** Es una extensión de VO que añade comportamiento dinámico a los objetos. Esto significa que cada objeto de este tipo funciona de forma independiente gracias a su propio hilo de ejecución (thread). Gracias a esto, los VOD pueden moverse, actualizarse y detectar colisiones sin depender del hilo principal del programa. Algunos ejemplos son planetas, asteroides, satélites o armas.

Los objetos VOD contienen toda la lógica de movimiento y los métodos necesarios para ser visualizados por la vista (LV). Además, se comunican con el modelo (LM) para comprobar colisiones y reaccionar en consecuencia, como desaparecer o generar animaciones.

Sistema de físicas

Para dotar a los objetos dinámicos de un comportamiento más coherente dentro del entorno de juego, The Game incorpora un sistema básico de físicas. Este permite que ciertos objetos puedan moverse suavemente por el espacio.

Cada objeto dinámico (VOD) dispone de un modelo físico propio, representado por una estructura llamada PhysicalModel. Este modelo contiene las propiedades físicas del objeto, como su velocidad y aceleración, y la lógica necesaria para actualizar su posición a medida que avanza el tiempo.

En cada ciclo de actualización, el sistema calcula cuánto debe desplazarse un objeto en función del tiempo transcurrido desde su último movimiento, aplicando además cualquier aceleración que pudiera estar afectándolo. Esto evita que los objetos se desplacen de forma repentina y permite que sus trayectorias sean más suaves y naturales.

Para llevar a cabo todo este proceso, PhysicalModel utiliza los siguientes objetos:

- **DoubleVector:** Es la base de todo el sistema. Representa un vector en dos dimensiones, es decir, un valor con dirección y magnitud en los ejes X e Y. Se utiliza para definir posiciones, velocidades y aceleraciones.
  Esta clase permite realizar operaciones comunes como sumar vectores, escalarlos, rotarlos o calcular su módulo.
- **Position:** Extiende DoubleVector y añade un dato adicional, una marca de tiempo. Este valor indica el momento exacto en el que se actualizó por última vez la posición del objeto.
- **PhysicalVariables:** Almacena dos vectores para cada VOD, velocidad y aceleración. Ambos se actualizan constantemente y son los que el sistema usa para calcular el nuevo movimiento del objeto en cada ciclo de ejecución. • **PhysicalCharacteristics:** Contiene información sobre el comportamiento físico del objeto, como su masa o sus límites máximos de velocidad y aceleración.

Cálculo del movimiento

El movimiento de los objetos se basa en un sistema de actualización constante. En cada ciclo, el sistema calcula cuánto debe moverse un objeto en función del tiempo que ha pasado desde su último movimiento, y en qué dirección debe hacerlo. El proceso sigue una lógica sencilla:

1. **Tiempo transcurrido:** Primero, el sistema consulta cuánto tiempo ha pasado desde la última vez que se actualizó la posición del objeto. Esto se obtiene gracias a la marca de tiempo almacenada en la posición del objeto.
2. **Desplazamiento:** A partir del tiempo transcurrido y la velocidad actual, se calcula cuánto debería desplazarse el objeto. Este desplazamiento se suma a la posición anterior, generando así una nueva posición.
3. **Cambio de velocidad:** Después, si el objeto tiene aceleración, esta se aplica para modificar su velocidad. Es decir, si el objeto se estaba acelerando o frenando, su velocidad cambia ligeramente para el siguiente ciclo.

Este proceso se repite mientras el objeto esté activo, lo que permite que se mueva de forma fluida y continua por la pantalla. Aunque los cálculos en segundo plano se realizan con precisión matemática, el objetivo es lograr un comportamiento natural sin movimientos bruscos ni efectos artificiales.

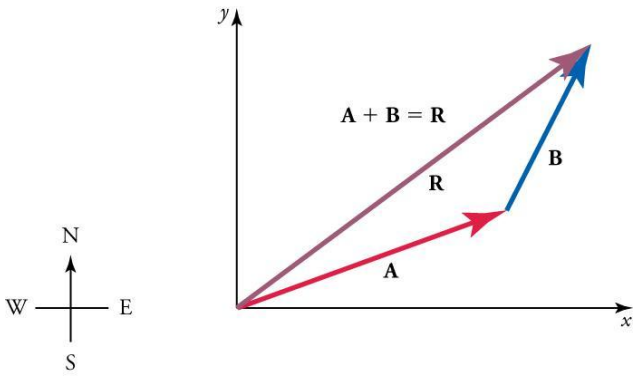
Añadidos del proyecto

Como realización del proyecto de fin de grado se han añadido diversas funcionalidades, entre ellas el cálculo de movimiento de los VODs teniendo en cuenta factores gravitatorios producidos por los diferentes elementos de la aplicación, y la inclusión de una nave controlada por el usuario con sus propias interacciones con los elementos de juego.

Fuerza gravitatoria

Por la estructura de la aplicación, todos los VODs tienen integrado un objeto llamado PhysicalModel que contiene las funciones necesarias para calcular la nueva posición a la que el VOD se tendrá que desplazar. Dicha función se encuentra en la función Run() proveído al implementar el objeto Runnable que permite el uso de threads. En la versión base el movimiento se trata de una simple línea recta con velocidad y dirección determinadas. Esto se ha cambiado para dotar de una nueva forma de calcular el movimiento basado en la Ley de Gravitación universal de Newton y la resolución de cálculo vectorial.

$$\vec{g} = -G\,\frac{M_1\,M_2}{r^2}$$



El método desarrollado se llama calcNewLocationGravity() y al igual que el anterior recibe como parámetros Position, que almacena él tiempo entre desplazamientos y un cálculo pala eliminar el desfase entre estos, y PhysicalVariables, que contiene como objetos DoubleVector la velocidad y aceleración.

Modificación del objeto Planeta

Al objeto Planeta se le ha añadido un parámetro de masa de tipo double y un objeto static de tipo List que guardará todos los planetas creados en la aplicación. Estos serán usados dentro del nuevo método para realizar el cálculo vectorial a través de la Ley de Newton. La masa se ha añadido en el constructor del Planeta y una vez creado se añade directamente a la List allPlanets. Además, se han creado los correspondientes get().

Inicio de variables

La primera parte del calcNewLocationGravity() consiste en la creación de dos objetos DoubleVector, objetcPos que almacena la posición pasada como parámetro a la función, y gravAccel que se inicia con valores $(0, 0)$. Le sigue la creación de la constante de gravitación universal G con su valor real de 6×10^{-11} y la clonación de phyVar para utilizar la velocidad y aceleración del anterior movimiento, manteniendo la correlación en el tiempo.

```
public void calcNewLocationGravity(Position pos, PhysicalVariables
phyVar) {
    DoubleVector objectPos = new DoubleVector(pos);
    DoubleVector gravAccel = new DoubleVector(0, 0);
    double G = 6 * 10e-11;

    phyVar.cloneSpeed(this.phyVariables.speed);
    phyVar.cloneAcceleration(this.phyVariables.acceleration);
```

Cálculo vectorial

Una vez iniciadas las variables se procede a realizar el cálculo del vector de aceleración mediante el recorrido de la lista de planetas. Este se realiza usando un for donde se realizan los siguientes pasos:

1. Se guarda la posición del planeta actual en planetPos como DoubleVector.
2. Se calcula la distancia al planeta mediante cálculo vectorial, si esta resulta demasiado pequeña se le asigna el valor base de 1 para evitar cálculos complejos.
3. Se calcula y almacena como double la fuerza gravitatoria del vector en forceMagnitude mediante la aplicación de la fórmula de gravitación.
4. Finalmente se calculan las componentes X e Y del vector de gravedad y se asigna a la variable anteriormente creada gravAccel.

```
for (Planet planet : Planet.getAllPlanets()) {
    DoubleVector planetPos = planet.getPosition();

    double dx = planetPos.getX() - objectPos.getX();
    double dy = planetPos.getY() - objectPos.getY();
    double r2 = dx * dx + dy * dy;
    double distance = Math.sqrt(r2);

    if (distance < 1) distance = 1;

    double forceMagnitude = G * planet.getMass() / (distance *
distance);

    double ax = forceMagnitude * dx / distance;
    double ay = forceMagnitude * dy / distance;

    gravAccel.setXY(gravAccel.getX() + ax, gravAccel.getY() + ay);
}
```

Asignación del desplazamiento

Para calcular la nueva posición se actualiza la aceleración de phyVar usando el nuevo vector de gravedad gravAccel, se calcula el offset de tiempo para dotar al movimiento de una mayor suavidad, se crea una nueva Position usando como base la que se ha pasado al método como parámetro y se le añaden el offset y el tiempo actual en milisegundos (este se usará en la siguiente llamada a calcNewLocationGravity() para calcular de nuevo el offset). Finalmente se clona en la Position pasada como parámetro la nueva posición calculada para su actualización y se realiza el mismo paso con la velocidad en la PhysicalVariable.

```
phyVar.acceleration.set(gravAccel);

long nowMillis = currentTimeMillis();
long elapsedMillis = nowMillis - pos.getPositionMillis();

DoubleVector offset = this.calcOffset(phyVar.speed, elapsedMillis);
Position newPos = new Position(pos);
newPos.add(offset);
newPos.setPositionMillis(nowMillis);

pos.clone(newPos);
DoubleVector speed = this.calcSpeed(phyVar.acceleration, phyVar.speed,
elapsedMillis);
phyVar.cloneSpeed(speed);
```

Movimiento

Para finalizar, el VOD que ejecuta el calcNewLocationGravity() usa la Position y PhysicalVariable actualizadas por este para llamar a nextMove() y desplazarse a la nueva posición calculada.

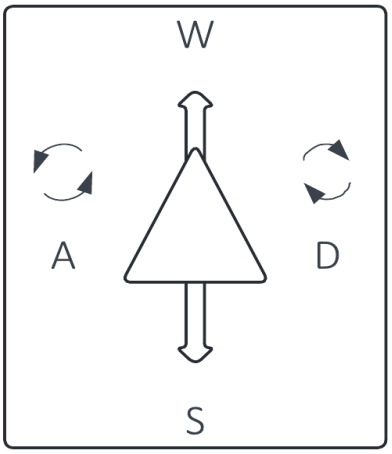
Nave

Como segundo añadido se ha creado una nave para que el usuario pueda interactuar y desplazarse por el entorno de juego. La nave extiende el objeto VOD aunque tiene un Run() único adaptado a su funcionalidad y control por el usuario.

Controles

La nave se desplaza mediante una aceleración y deceleración en el sentido al que apunta la proa de la nave. Para realizar dichas acciones se han asignado las teclas W y S respectivamente. Para cambiar el ángulo de la nave, o para girar en lengua vernácula, se usan las teclas A y D, siendo la A la rotación antihoraria y la D la rotación horaria.

Finalmente hay dos controles especiales, el primero asignado a la Q es un sistema de frenado progresivo que cancela todo el movimiento que esté sufriendo la nave hasta dejarla completamente estática en el sitio; la segunda se ha asignado a la Barra Espaciadora, esta crea una nave nueva si la primera ha sido destruida por el resultado de un impacto contra un asteroide o un planeta. La lógica del respawn se encuentra en el objeto TG en vez de la propia nave.




```
private void setupKeyBindings() {
    SwingUtilities.invokeLater(() -> {

KeyboardFocusManager.getCurrentKeyboardFocusManager().addKeyEventDispa
tcher(e -> {
            if (e.getID() == KeyEvent.KEY_PRESSED || e.getID() ==
KeyEvent.KEY_RELEASED) {
                boolean pressed = e.getID() == KeyEvent.KEY_PRESSED;
                switch (e.getKeyCode()) {
                    case KeyEvent.VK_W -> up = pressed;
                    case KeyEvent.VK_S -> down = pressed;
                    case KeyEvent.VK_A -> left = pressed;
                    case KeyEvent.VK_D -> right = pressed;
                    case KeyEvent.VK_Q -> braking = pressed;
                }
            }
            return false;
        });
    });
}
```

Parámetros de control

La nave tiene sus parámetros divididos en tres grupos principales:

1. El primero define las constantes relacionadas con el movimiento, se encuentran el tamaño, la tasa de aceleración, la velocidad de rotación y la velocidad máxima que podrá alcanzar.
2. En segundo lugar, se encuentra las direcciones up, down, left, right que se usarán para rotar y desplazar la nave, el ángulo en el cual el objeto aparecerá y el freno.
3. Como tercer y último grupo principal se encuentra el tamaño de la ventana de la aplicación.

Para finalizar se encuentra un BufferedImage que contendrá el Sprite que usará la nave.

```
private static final int SIZE = 20;
private static final double ACCELERATION = 0.0005;
private static final double ROTATION_SPEED = 0.22;
private static final double MAX_SPEED = 0.5;

private boolean up, down, left, right;
private double angle = 270;
private boolean braking = false;

private final int windowWidth = 1300;
private final int windowHeight = 700;

private final BufferedImage shipImage;
```

Representación gráfica

La nave a través del método Paint() aplicará una imagen a modo de Sprite a través del BufferedImage y Graphics, para que tenga una representación visual en el juego. Dicho Sprite, gracias a la nave que extiende VOD, se usará para controlar las colisiones con otros objetos.

```
public void paint(Graphics gr) {
    Graphics2D g2 = (Graphics2D) gr.create();
    int x = (int) (getPosition().getX());
    int y = (int) (getPosition().getY());
    if (shipImage != null) {
        int imageWidth = shipImage.getWidth();
        int imageHeight = shipImage.getHeight();

        AffineTransform transform = AffineTransform.getRotateInstance(
                Math.toRadians(angle), x, y
        );
        g2.setTransform(transform);
        g2.drawImage(shipImage, x - imageWidth / 2, y - imageHeight /
2, null);
    } else {
        super.paint(gr);
    }
    g2.dispose();
}
```

Movimiento

El movimiento está inspirado en la nave del juego arcade Asteroids, haciendo que esta se desplace mediante una aceleración o deceleración en el sentido de la nave junto a la capacidad de rotar en sentido horario y antihorario. Además, se le ha añadido la interacción gravitatoria que sufren los VODs para incrementar el realismo y dificultar la jugabilidad, por lo que la nave será atraída por las distintas gravedades de los planetas creando una resistencia al movimiento o incluso absorbiendo completamente la nave si se acerca demasiado.

Finalmente, se le ha incluido un sistema de frenado gradual, que dejará la nave estática en su posición, y un sistema de interacción con los bordes de la aplicación. Al impactar en contra uno de los bordes de la ventana del juego la nave saldrá rebotada en la dirección contraria al impacto sufriendo una pérdida en su aceleración a causa de este, simulando la pérdida de energía cinética por interactuar con el entorno.

Ciclo de movimiento

La nave podrá ser controlada mientras el estado de esta sea diferente a DEAD, esto es manejado con un bucle while con un if que además confirmará que el estado sea ALIVE, ya que existe un tercer estado llamado COLLIDED con su propia lógica.

```
while (this.getState() != src.tg.local.vo.VOState.DEAD) {
    if (this.getState() == src.tg.local.vo.VOState.ALIVE) {
```

Aceleración y rotación

Como se ha comentado en el apartado de controles, la nave cuenta con la capacidad de acelerar, decelerar y rotar. Este apartado es el primero en realizarse dentro del ciclo de movimiento.

Primero se guardan en variables los tiempos para calcular un offset que dota al movimiento de mayor suavidad (aunque no se usarán hasta mas adelante). Le sigue el control de rotación de la nave y finalmente el sistema de aceleración. Al ser un sistema inercial el cálculo dirección depende de la resolución de las funciones Seno y Coseno para el cambio de la aceleración. Finalmente se aumenta o reduce la aceleración lineal dependiendo de si se pulsa la W o la S (up y down respectivamente).

```
long nowMillis = System.currentTimeMillis();
long elapsed = nowMillis - prevMillis;
prevMillis = nowMillis;

if (left) angle -= ROTATION_SPEED * elapsed;
if (right) angle += ROTATION_SPEED * elapsed;

DoubleVector userAcc = new DoubleVector(0, 0);
if (up) {
    userAcc.setXY(
            Math.cos(Math.toRadians(angle)),
            Math.sin(Math.toRadians(angle))
    );
    userAcc.scale(ACCELERATION);
}
if (down) {
    userAcc.setXY(
            Math.cos(Math.toRadians(angle + 180)),
            Math.sin(Math.toRadians(angle + 180))
    );
    userAcc.scale(ACCELERATION);
}
```

Gravedad

Una vez establecido la acción que quiere realizar el usuario se procede a inducir la gravedad al movimiento. Este se comporta de la misma forma que el que se puede encontrar en calcNewLocationGravity() pero usando los parámetros y variables de la Nave.

```
DoubleVector gravAccel = new DoubleVector(0, 0);
double G = 6 * 10e-11;
DoubleVector shipPos = new DoubleVector(pos);
for (Planet planet : Planet.getAllPlanets()) {
    DoubleVector planetPos = planet.getPosition();
    double dx = planetPos.getX() - shipPos.getX();
    double dy = planetPos.getY() - shipPos.getY();
    double r2 = dx * dx + dy * dy;
    double distance = Math.sqrt(r2);

    if (distance < 1) distance = 1;

    double forceMagnitude = G * planet.getMass() / (distance *
distance);
    double ax = forceMagnitude * dx / distance;
    double ay = forceMagnitude * dy / distance;
    gravAccel.setXY(gravAccel.getX() + ax, gravAccel.getY() + ay);
}

DoubleVector totalAccel = new DoubleVector(userAcc);
totalAccel.add(gravAccel);
```

Frenado

Sigue el frenado de la nave. Si el usuario decide presionar la Q para iniciar el frenado, la nave empezará a cancelar su aceleración de forma gradual, haciendo que desde que se inicia hasta que se queda estática un proceso que tardará mas o menos en función de la velocidad que lleve la nave en el momento.

```
if (braking) {
    final double BRAKE_STRENGTH = 0.0005;

    DoubleVector brakeAccel = new DoubleVector(phyVars.speed);
    if (brakeAccel.getModule() > 0) {
        brakeAccel.scale(-1);
        double speedMag = phyVars.speed.getModule();
        double brakeMag = Math.min(BRAKE_STRENGTH, speedMag / (elapsed
> 0 ? elapsed : 1));
        if (speedMag > 0) {
            brakeAccel.scale(brakeMag / speedMag);
            totalAccel.add(brakeAccel);
        }
    }
}
```

Aplicación del movimiento

Después del frenado se encuentra la sección encargada de aplicar todo lo calculado en los apartados anteriores a la posición y variables físicas, en esta sección además se aplica el offset para mejorar la suavidad del movimiento.

```
DoubleVector speed = new DoubleVector(phyVars.speed);
DoubleVector speedIncrement = new DoubleVector(totalAccel);
speedIncrement.scale(elapsed);
speed.add(speedIncrement);

if (speed.getModule() > MAX_SPEED) {
    double scale = MAX_SPEED / speed.getModule();
    speed.scale(scale);
}

phyVars.speed.set(speed);
phyVars.acceleration.set(totalAccel);

DoubleVector offset = new DoubleVector(speed);
offset.scale(elapsed);
pos.add(offset);
pos.setPositionMillis(nowMillis);

double x = pos.getX();
double y = pos.getY();
double halfSize = SIZE / 2.0;
```

Rebote y colisión

Para finalizar se encuentra el código encargado de administrar la colisión con los bordes de la ventana y detectar si hay algún impacto entre la nave y otro VOD.

```
boolean bounced = false;
if (x - halfSize < 0) {
    pos.setX(halfSize);
    phyVars.speed.setX(-phyVars.speed.getX());
    bounced = true;
}
if (x + halfSize > windowWidth) {
    pos.setX(windowWidth - halfSize);
    phyVars.speed.setX(-phyVars.speed.getX());
    bounced = true;
}
if (y - halfSize < 0) {
    pos.setY(halfSize);
    phyVars.speed.setY(-phyVars.speed.getY());
    bounced = true;
}
if (y + halfSize > windowHeight) {
    pos.setY(windowHeight - halfSize);
    phyVars.speed.setY(-phyVars.speed.getY());
    bounced = true;
}
if (bounced) {
    phyVars.speed.scale(0.85);
}
this.getLocalModel().collisionDetection(this);
```

Sistema de respawn

Como se ha comentado con anterioridad, en caso de que la nave impacte contra otro VOD, cambiara el estado a DEAD y se borrará del tablero de juego, lo que deja al usuario sin forma de interactuar. Para sortear este problema se ha creado dentro del objeto TG, que actúa como main de la aplicación, una forma de crear una nueva nave.

Este método se encarga de crear una nueva instancia de nave a través le pulsar la Barra Espaciadora. Se hace una comprobación para asegurarse de que no haya una nave activa y en caso contrario crea una nueva y la activa para que el usuario pueda seguir interactuando.

```
private void setupRespawnKey() {

java.awt.KeyboardFocusManager.getCurrentKeyboardFocusManager().addKeyE
ventDispatcher(e -> {
        if (e.getID() == java.awt.event.KeyEvent.KEY_PRESSED &&
e.getKeyCode() == java.awt.event.KeyEvent.VK_SPACE) {
            boolean aliveShipExists = false;
            for (VO vo : this.peerController.getVisualObjects()) {
                if (vo instanceof Ship s && s.getState() ==
src.tg.local.vo.VOState.ALIVE) {
                    aliveShipExists = true;
                    break;
                }
            }
            if (!aliveShipExists) {
                Ship newShip = new Ship(new DoubleVector(100, 600));
                newShip.activate();
                this.peerController.addVisualObject(newShip);
            }
        }
        return false;
    });
}
```

Conclusión

A partir de un modelo de juego base se han expandido las bases del cálculo inercial de los diferentes objetos para respetar la gravitación de otros cuerpos, y se ha creado un elemento nuevo a modo de nave para que el usuario pueda interactuar con el medio proporcionado y mejorado, desarrollando un movimiento y comportamiento únicos. Finalmente se ha dotado de gráficos a este nuevo objeto y creada la capacidad de crear nuevas naves en caso de perder la actual para que el usuario no tenga que reiniciar la aplicación o quedarse impasible sin poder interactuar.

Todo el proyecto se puede encontrar en este enlace de GitHub, junto a todos los cambios realizados y las diferentes releases. En estas se puede descargar el proyecto a través de un .zip facilitando su acceso sin necesidad de instalar Git. Aún con todo, se debe tener una versión de Java compatible y un entorno o IDE donde poder ejecutar archivos java.