UNIVERSIDAD NACIONAL GENERAL SARMIENTO

Programación I
El camino de Gondolf

Integrantes:

Melanie Rosi (melanieRosi25@gmail.com)
Nicole Terracciano (nicoleterracciano28@gmail.com)

Docentes

Leonor Gutiérrez Leonardo Waingarten

El presente trabajo práctico consiste en el desarrollo de un juego en Java que tiene como protagonista a Gondolf, un mago que debe defenderse de una invasión de murciélagos enemigos que aparecen desde los bordes de la pantalla y lo persiguen constantemente.

Durante el proceso de desarrollo, fuimos implementando distintas clases para organizar mejor el código, asignando a cada una responsabilidades específicas: el personaje principal, los enemigos, los hechizos, los obstáculos y el entorno general del juego. Paso a paso, incorporamos nuevas funcionalidades como el movimiento del personaje, la lógica de colisiones, el uso de hechizos, el panel lateral de interfaz y el sistema de victoria o derrota.

En este informe vamos a detallar cómo diseñamos cada clase, qué rol cumple cada método y cómo fuimos resolviendo los distintos problemas que surgieron. También compartiremos algunas decisiones de diseño que tomamos y qué cambios realizamos.

Implementamos distintas clases que nos permitieron dividir las responsabilidades del juego de manera estructurada y funcional.

A continuación, se describen las clases principales, sus variables y métodos.

Clase Juego:

La clase Juego es la clase principal de nuestro proyecto. Se encarga de coordinar y conectar todos los elementos del juego: el personaje principal, los murciélagos (enemigos), los hechizos para defenderse de ellos, el entorno gráfico y la lógica general de victoria o derrota.

Desde esta clase controlamos el flujo del programa a través del método tick(), que se ejecuta constantemente para actualizar lo que ocurre en pantalla, procesar las acciones del mago (como moverse o lanzar hechizos), y generar respuestas inmediatas por parte del entorno y los enemigos. Toda la dinámica del juego, desde la aparición de murciélagos hasta el manejo de colisiones, energía mágica y condiciones de fin de partida, está gestionada desde esta clase, lo que la convierte en el núcleo del proyecto.

Las variables de instancia de la clase Juego permiten representar todos los elementos visibles y lógicos del juego:

- **Entorno entorno;** la ventana gráfica donde se dibujan los elementos y se detectan entradas del teclado y mouse.
- **Image fondo;** imagen de fondo del juego.
- **Image imagenVictoria, imagenDerrota;** imágenes que se muestran al finalizar el juego (según gane o pierda).
- **Murciélagos murcielagos;** murciélago individual con su posición, imagen y estado (vivo o muerto).
- **boolean inhArriba, inhAbajo, inhIzquierda, inhDerecha: v**ariables booleanas indican si el mago tiene restringido el movimiento en alguna dirección. Se activan cuando colisiona con una roca o con los bordes de la pantalla, y se usan en las condiciones del método tick() para evitar que el jugador se mueva en esa dirección.
- **Personaje mago;** personaje principal que el jugador controla.
- **Obstaculo[] rocas;** arreglo de obstáculos fijos en pantalla.
- **Murcielagos[] murcielagosEnPantalla;** arreglo de enemigos activos en pantalla.
- **int totalEnemigos=50;** cantidad total de murciélagos a generar.

- **int maxEnemigosPantalla=10;** máximo de murciélagos vivos en pantalla al mismo tiempo.
- **int enemigosVivos=0;** contador de murciélagos activos en pantalla.
- **int enemigosDerrotados=0;** cantidad de murciélagos eliminados por el mago.
- **Hechizo hechizoAgua, hechizoFuego:** representan los hechizos disponibles.
- **int energiaMagica =100;** energía disponible para lanzar hechizos.
- **boolean botonAguaSeleccionado=false;** y **boolean botonFuegoSeleccionado=false;** indican cuál hechizo está seleccionado por el jugador.
- **boolean circuloActivo=false;** indica si se está mostrando un círculo de hechizo.
- **long tiempoInicioCirculo=0;** registra cuándo se activó el círculo del hechizo.
- **int circuloX=0;** y **circuloY=0;** coordenadas donde se activó el hechizo.
- **int radioCirculo=0;** radio del efecto visual.
- **Color colorCirculo= Color.cyan;** color del hechizo (rojo o cyan según el tipo de hechizo).

El constructor **Juego()** inicializa todos los elementos fundamentales del juego.

Se crea el entorno gráfico usando la clase Entorno, definiendo un título para la ventana ("Proyecto para TP") y un tamaño de 800 x 600 píxeles.

Cargamos la imagen del fondo (fondo.png), creamos al mago en el centro de la pantalla (posición (300.0, 300.0)) y definimos un arreglo de cinco rocas (Obstaculo[5]), usando dos arreglos con coordenadas x e y.

Para cada posición, inicializamos una roca pasándole su ubicación y el entorno para poder dibujarla. A continuación, se cargan las imágenes de victoria y derrota, que se mostrarán al finalizar el juego según el resultado.

Después inicializamos el arreglo *murcielagosEnPantalla,* que almacena los enemigos activos en pantalla en cada momento. La cantidad máxima de murciélagos en pantalla, la definimos con una variable (maxEnemigosPantalla) y nos aseguramos de que no se generen más de los permitidos.

Luego declaramos los dos hechizos disponibles para el jugador: la *"Bomba de Agua"*, que no consume energía pero tiene un alcance corto, y la *"Bomba de Fuego",* que sí requiere energía pero cubre un área mayor.

Metodo Tick ():

- Al comienzo del método *tick()*, se dibuja el fondo en la pantalla . Para ello utilizamos el método *entorno.dibujarImagen(...)*, centrando la imagen del fondo en el medio del entorno, ajustado un tamaño.
- Luego verificamos si el juego ha finalizado (juegoTerminado). En caso afirmativo, mostramos la imagen correspondiente (victoria o derrota) y se termina la ejecución del método con return, evitando así cualquier otra actualización de juego.
- Si el juego no terminó, se dibuja el mago en su posición actual usando el método *mago.dibujar().*
- Si un hechizo fue lanzado recientemente *(circuloActivo*), se dibuja un círculo con el método *entorno.dibujarCirculo(...)* para representar su área de efecto. Este círculo permanece visible durante 300 milisegundos.
- Se verifican colisiones entre el mago y cada una de las rocas del arreglo rocas mediante el método *colisionMagoRoca(...)*. Además, controlamos si el mago colisiona con los bordes del entorno utilizando el método *colisionMagoPantalla(...).*
- Se restablecen las variables de inhibición de movimiento *(inhAbajo, inhArriba, inhDerecha, inhIzquierda)* a false, para permitir nuevamente el desplazamiento del mago en el siguiente ciclo, salvo que se detecte una nueva colisión.

A continuación, se evalúa si el jugador hizo clic con el botón izquierdo del mouse mediante el método *entorno.sePresionoBoton(entorno.BOTON_IZQUIERDO*). Si es así, se obtienen las coordenadas del cursor con *entorno.mouseX() y entorno.mouseY()* para determinar en qué parte de la pantalla se hizo clic.

Dependiendo de la ubicación del clic:

- Si fue sobre el botón correspondiente al hechizo de agua (coordenadas entre 615 y 785 px de ancho, y entre 165 y 215 px de alto), se selecciona ese hechizo

mediante el método *hechizoAgua.seleccionar(),* y se deselecciona el de fuego con el método *hechizoFuego.deseleccionar(*).

- Si fue sobre el botón del hechizo de fuego (coordenadas entre 615 y 785 px de ancho, y entre 220 y 270 px de alto), y además hay suficiente energía mágica, se realiza la selección inversa.

Si el clic se realizó fuera del panel lateral (es decir, con mx < 600), y había un hechizo previamente seleccionado con energía suficiente, se procede a lanzarlo. Para esto se:

- ☐ Determina qué hechizo está seleccionado.
- ☐ Registra las coordenadas del clic como centro del área de efecto *(circuloX, circuloY).*
- ☐ Configura el color del círculo (rojo para fuego, cian para agua) y activa un temporizador visual *(tiempoInicioCirculo con System.currentTimeMillis() que es un método estático de la clase System en Java).*
- ☐ Se activa *circuloActivo* para que el efecto visual se muestre durante 300 milisegundos.

En cada ejecución del método *tick()*, el juego evalúa si debe generarse un nuevo enemigo. Para ello, se verifican dos condiciones:

- → Que la cantidad actual de murciélagos vivos *(enemigosVivos)* sea menor al máximo permitido en pantalla *(maxEnemigosPantalla).*
- → Que la suma de murciélagos vivos y derrotados no supere el total permitido para la partida *(totalEnemigos).*

Si ambas condiciones se cumplen, se llama al método auxiliar *crearMurcielagoEnBorde(),* que genera un nuevo enemigo en un borde aleatorio del entorno. Si la creación tiene éxito (es decir, devuelve un objeto distinto de null), el nuevo murciélago se incorpora al arreglo *murcielagosEnPantalla* mediante a*gregarMurcielago(nuevo)* y se incrementa el contador de enemigos vivos.

Luego, el juego recorre el arreglo de murciélagos activos *(murcielagosEnPantalla).* Para cada uno:

→ Se actualiza su posición utilizando el método *moverHacia(mago.x, mago.y),* que permite simular el movimiento del enemigo en dirección al mago.

→ Se dibuja en pantalla con *dibujar(entorno).*

→ Se evalúa si se encuentra lo suficientemente cerca del mago mediante *estaCerca(...).* En caso afirmativo:

- El mago recibe daño utilizando *recibirDanio(2).*
- El murciélago se elimina del arreglo (asignándole null).
- Se decrementa el contador de enemigos vivos y se incrementa el de derrotados.

Finalmente, se evalúan las condiciones de finalización del juego. Estas son:

- *Victoria:* si el número de murciélagos derrotados *(enemigosDerrotados)* alcanza exactamente 50, se establece *juegoTerminado = true y magoGano = true.*
- *Derrota:* si el mago pierde todos sus puntos de vida *(verificado con !mago.estaVivo()),* también se finaliza el juego, pero con *magoGano = false*.

Panel lateral:

En esta parte del método *tick()*, diseñamos un panel lateral que muestra al jugador información clave del estado del juego. Dibujamos rectángulos para dividir el área en secciones: los botones de hechizos permiten elegir entre "*Bomba de Agua*" y "*Bomba de Fuego*", y cambian de color según cuál esté seleccionado.

La barra de vida muestra el porcentaje de salud actual del mago, mientras que la barra de energía mágica indica cuánta energía queda para lanzar hechizos.

Finalmente, en la parte inferior se muestra cuántos enemigos han sido derrotados, lo que ayuda a seguir el progreso del jugador durante la partida.

Método crearMurcielagoEnBorde(); tiene como objetivo generar un nuevo enemigo (murciélago) en una posición aleatoria, justo fuera de alguno de los bordes de la pantalla del juego.

Primero, se elige aleatoriamente uno de los cuatro bordes (superior, inferior, izquierdo o derecho). Luego, se le asignan coordenadas iniciales de forma que el murciélago

aparezca ligeramente fuera de los límites visibles, simulando que ingresa desde afuera del entorno.

Método agregarMurcielago(Murcielagos nuevo): tiene como objetivo incorporar un nuevo murciélago al arreglo *murcielagosEnPantalla,* que representa a todos los enemigos actualmente visibles en el entorno.

Lo que hace es recorrer el arreglo buscando la primera posición disponible (es decir, una casilla que esté en null) y, una vez que la encuentra, asigna allí el nuevo objeto Murcielagos que se recibe como parámetro. Apenas lo agrega, sale del bucle utilizando return, ya que no hace falta seguir buscando.

Método colisionMagoRoca(Personaje p, Obstaculo o); detecta si el personaje principal (el mago) está colisionando con una roca, y determina desde qué lado ocurre esa colisión. Para lograrlo, compara los bordes del mago con los bordes del obstáculo, utilizando comparaciones de distancia y superposición vertical u horizontal, según el caso.

Método colisionMagoPantalla(Personaje p); se encarga de detectar si el mago intenta salirse de los límites de la pantalla (es decir, del entorno gráfico). Para eso, verifica sus bordes (superior, inferior, izquierdo y derecho) y, si alguno supera los límites permitidos, activa las variables inhArriba, inhAbajo, inhIzquierda o inhDerecha para bloquear el movimiento en esa dirección.

Así, el mago no puede atravesar los bordes del entorno, y su movimiento queda restringido solo cuando es necesario.

Clase Personaje ;

La clase Personaje representa al mago, que es el personaje principal controlado por el jugador en el entorno gráfico. Contiene información clave sobre su posición, dirección, tamaño, imágenes, estado de vida y detección de colisiones.

Variables de instancia:

- **double x, y** : Posición central del personaje en el entorno.
- **boolean direccion** : Indica si el personaje mira a la derecha (true) o izquierda (false).
- **Image imgD, imgI** : Imágenes del personaje mirando hacia la derecha e izquierda.
- **double escala** : Escala de tamaño de la imagen del personaje.
- **double alto, ancho** : Altura y ancho del personaje (calculados según la imagen y la escala).
- **Entorno e** : Referencia al entorno gráfico para poder dibujar al personaje.
- **Double bordeDerecho, bordeIzquierdo, bordeSuperior, bordeInferior** :Límites del personaje que se usan para detectar colisiones con obstáculos o bordes.
- **int vidaMax = 100:** Vida máxima del personaje (valor inicial fijo).
- **int vidaActual = vidaMax :** Vida actual del personaje. Empieza con el valor máximo.

En el constructor de la clase Personaje, inicializamos las variables de instancia que definen la posición y apariencia del mago en el entorno. Se reciben las coordenadas iniciales x e y, y también la referencia al objeto Entorno. Luego, se cargan las imágenes correspondientes a las direcciones en las que el personaje puede mirar (derecha e izquierda), se define una escala para ajustar su tamaño en pantalla, y se calculan las dimensiones reales del personaje (alto y ancho). Además, se guarda la referencia al entorno gráfico para poder utilizarla luego al momento de dibujar o mover al personaje.

Método Dibujar() : se encarga de mostrar gráficamente al personaje en pantalla utilizando la imagen correspondiente a la dirección en la que está mirando. Si el personaje está mirando hacia la derecha (direccion == true), se dibuja con la imagen *imgD*. En caso contrario, se usa la imagen imgI. En ambos casos, la imagen se centra en las coordenadas actuales del personaje (x e y), con una escala previamente definida. Para ello, se utiliza el método *dibujarImagen(...)* proporcionado por el entorno gráfico.

Método recibirDanio(int d) : se encarga de descontar puntos de vida cuando el personaje es atacado o recibe algún tipo de daño. Este método toma como parámetro un valor entero d, que representa la cantidad de daño recibido. Luego, ese valor se resta a la variable vidaActual. En caso de que el resultado sea menor a cero, se fuerza que la vida quede en 0, evitando así valores negativos que no tendrían sentido dentro del contexto del juego.

Método estaVivo() : verifica si el personaje aún sigue con vida. Retorna true si la vida actual es mayor a cero, y false si ya no le queda energía.

Clase Murciélago ;

Representamos a los enemigos del mago, cada objeto de esta clase tiene su propia posición, velocidad y estado (vivo o muerto), puede moverse y dibujarse en la pantalla.

Variables de instancia:

- **double x, y:** representan las coordenadas actuales del murciélago en la pantalla.
- **double velocidad**: determina la velocidad con la que se mueve. Está inicializada en 1.5.
- **Image img**: es la imagen que representa visualmente al murciélago.
- **double escala:** define cuánto se reduce el tamaño de la imagen original (en este caso, al 15%).
- **boolean muerto**: indica si el murciélago fue eliminado. Inicialmente es false.

El constructor de la clase Murciélagos, *public Murcielagos(double x, double y),* permite crear un enemigo nuevo indicando su posición inicial. Utiliza los valores x e y como coordenadas y carga automáticamente su imagen ("murcielagos.png") para ser dibujada en el entorno gráfico del juego.

Método dibujar(Entorno e) : se encarga de mostrar gráficamente al murciélago en la pantalla. Para lograrlo, recibe como parámetro una referencia al entorno (Entorno e) y utiliza el método *dibujarImagen(...)* que proporciona Entorno.

Se dibuja la imagen del murciélago (img) en su posición actual (x, y), sin rotación (ángulo 0), y con una escala previamente definida (escala), lo que permite ajustar el tamaño del enemigo en pantalla. Este método se invoca dentro del método tick(), que se ejecuta en cada iteracion del juego, con el objetivo de que el murciélago se dibuje continuamente en pantalla mientras esté activo (es decir, mientras no haya sido eliminado).

Método moverHacia(double xMago, double yMago): permite que el murciélago se desplace en dirección al mago. Calcula la distancia y dirección entre ambos personajes, y ajusta la velocidad del murciélago en función de esa distancia: se mueve más rápido cuando está lejos y más lento al acercarse. Esto genera un movimiento progresivo y realista en cada iteración del juego, actualizando su posición constantemente simulando persecución.

Metodo estaCerca(double px, double py, double umbral): verifica si el murciélago está cerca de un punto dado (por ejemplo, el mago). Para eso, compara la distancia entre sus coordenadas y las del punto recibido. Si esa distancia es menor al umbral especificado, devuelve true.

Se usa en tick() para detectar si un murciélago logró acercarse lo suficiente al mago como para hacerle daño.

Metodo estaMuerto(): devuelve true si el murciélago fue marcado como muerto, y false si sigue activo. Es útil para verificar su estado antes de dibujarlo o moverlo.

Metodo morir(): Marca al murciélago como muerto cambiando el valor de su atributo muerto a true. Se llama, por ejemplo, cuando es alcanzado por un hechizo.

Clase Obstáculo

La clase Obstáculo representa elementos sólidos y estáticos del entorno, las rocas, que el personaje no puede atravesar. Su función principal es servir de barrera dentro de la zona de juego.

Variables de instancia:

- **double x, y:** coordenadas del centro del obstáculo en el entorno.
- **double escala:** factor de escala para ajustar el tamaño de la imagen.
- **Image imgO:** imagen del obstáculo, cargada desde un archivo gráfico.
- **double ancho, alto**: dimensiones visuales del obstáculo tras aplicar la escala.
- **Entorno e:** referencia al entorno gráfico, necesaria para dibujar.
- **double bordeIzquierdo, bordeDerecho, bordeSuperior, bordeInferior:** valores que definen los límites del obstáculo, utilizados para detectar colisiones con el personaje.

El constructor de la clase Obstaculo, *public Obstaculo(double x, double y, Entorno e),* se encarga de crear y posicionar una roca en el entorno del juego. Recibe las coordenadas x e y donde debe ubicarse y una referencia al Entorno para poder dibujarla.

Le asignamos una escala fija para reducir su tamaño, y a partir de eso calculamos su alto y anchos reales. También definimos los bordes (izquierdo, derecho, superior e inferior), que usamos más adelante para detectar colisiones con el mago.

Método dibujar()

Este método permite mostrar visualmente el obstáculo en pantalla. Utiliza el método *dibujarImagen(...)* del entorno gráfico, posicionando la imagen de la roca en sus coordenadas actuales (x, y) y aplicándole la escala definida.

Clase Hechizo

La clase *Hechizo* representa los diferentes poderes mágicos que el personaje principal puede utilizar dentro del juego. Está diseñada para encapsular los atributos y comportamientos esenciales de cada hechizo.

Variables de instancia:

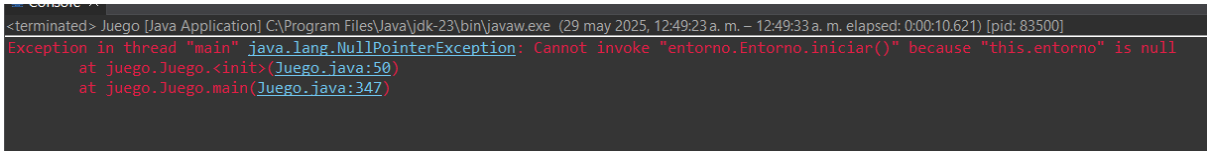
- **nombre:** Identifica el tipo de hechizo, como "Agua" o "Fuego".
- **costoEnergia**: Representa la cantidad de energía mágica que el mago necesita gastar para lanzar ese hechizo.
- **areaEfecto:** Define el radio de acción del hechizo (en píxeles), determinando cuántos enemigos puede afectar.
- **seleccionado:** Es un booleano que indica si ese hechizo está actualmente activado para ser lanzado por el jugador.
  El constructor, public Hechizo (String nombre, int costoEnergia, int areaEfecto): inicializa el hechizo con sus valores principales (nombre, costo de energía y área de efecto).
  Métodos
- *seleccionar():* Marca el hechizo como activo.
- d*eseleccionar():* Marca el hechizo como inactivo.

Cuando el jugador hace clic en uno de los botones del panel lateral, el hechizo correspondiente se selecciona. Al hacer clic dentro del área de juego, si el hechizo está activo y el mago tiene suficiente energía, se lanza el hechizo, creando un círculo visual en el entorno. Los enemigos dentro del radio definido por *areaEfecto* son eliminados, y la energía del mago se reduce según *costoEnergia.*

Durante el desarrollo, nos encontramos con un error importante al ejecutar el juego: un NullPointerException que ocurría apenas se abría la ventana. Luego de revisar el código, notamos que el problema se debía a que llamábamos al método entorno.iniciar() dentro del constructor Juego() antes de haber inicializado todos los objetos del juego, en particular el objeto mago.

El framework Entorno comienza a ejecutar el método tick() automáticamente cuando se llama a entorno.iniciar(). Como tick() intenta usar mago.dibujar(), pero mago aún no estaba creado, el programa se rompía al ejecutarse esa línea.

Reorganizamos el constructor y movimos la línea entorno.iniciar() al final, asegurándonos de que todos los elementos (como mago, rocas, hechizos, etc.) estuvieran correctamente inicializados antes de comenzar el ciclo del juego. Esto solucionó el error y permitió que el entorno se inicie con todos los objetos listos.



Durante la implementación de las colisiones entre el mago y los obstáculos, cometimos un error en las condiciones que verificaban si el mago chocaba con el límite superior o inferior de una roca. En el bloque de código correspondiente, comparamos los bordes del mismo obstáculo entre sí, como se muestra a continuación:

```
//limite con las rocas por debajo:funciona
   if (Math.abs(o.bordeInferior-o.bordeSuperior)< 0.5
     && p.bordeIzquierdo < o.bordeDerecho
     && p.bordeDerecho > o.bordeIzquierdo){
    this.inhAbajo=true;
   }
//limite con las rocas por arriba:funciona
   if (Math.abs(o.bordeSuperior-o.bordeInferior)< 0.5
     && p.bordeIzquierdo < o.bordeDerecho
     && p.bordeDerecho > o.bordeIzquierdo){
    this.inhArriba=true;
   }
```

Este tipo de comparación era incorrecta porque siempre daba una diferencia pequeña, pero no involucraba al personaje, por lo tanto el mago atravesaba las rocas verticalmente.

Reemplazamos esas líneas para que el cálculo de colisión compare correctamente los bordes del personaje y del obstáculo:

```
//limite con las rocas por debajo:funciona
  if (Math.abs(p.bordeInferior-o.bordeSuperior)< 0.5
    && p.bordeIzquierdo < o.bordeDerecho
    && p.bordeDerecho > o.bordeIzquierdo){
   this.inhAbajo=true;
  }
//limite con las rocas por arriba: funciona
  if (Math.abs(p.bordeSuperior-o.bordeInferior)< 0.5
    && p.bordeIzquierdo < o.bordeDerecho
    && p.bordeDerecho > o.bordeIzquierdo){
   this.inhArriba=true;
  }
```

Visualmente:

A lo largo de este proyecto aprendimos a trabajar por primera vez en un entorno gráfico, lo cual fue una experiencia totalmente nueva y desafiante. Logramos desarrollar un juego funcional, combinando imágenes, movimiento por teclado, interacción con el mouse y enemigos que reaccionan en tiempo real. Esta fue nuestra

primera experiencia creando un juego desde cero, y nos permitió aplicar muchos de los contenidos aprendidos en clase de una forma práctica y concreta.

Durante el desarrollo surgieron varios problemas, como errores al organizar el orden de inicialización de los objetos (por ejemplo, al llamar a entorno.iniciar() antes de tener todo preparado), o dificultades con las colisiones del personaje que permitían atravesar obstáculos. Al resolver estos errores fuimos incorporando herramientas clave para depurar código, organizar mejor nuestras clases y pensar con mayor claridad la lógica de funcionamiento del programa.

Nos llevamos como aprendizaje no solo conocimientos técnicos, sino también la experiencia de trabajar en equipo, resolver problemas, y mantener la motivación para seguir mejorando el proyecto. El trabajo con este entorno nos permitió integrar programación orientada a objetos con una aplicación visual y dinámica.