Manual del programador

“BUSCANDO A NEMO”

Autores:

Gaspar Huerta Saavedra

Mauricio Antonio Salazar

Montserrat Ximena Hernández Gallegos

Documento diseño de videojuego (GDD)

1. Visión general

Buscando a Nemo es un juego de descubrimiento en el que tenemos que buscar a un perro llamado Nemo. En el camino nos encontraremos con muchos obstáculos que nos impedirán encontrarnos con el perro. Todo comienza en el nivel 1 cuando el perro se escapa de casa y tenemos que salir a la calle a buscarlo.

La dificultad consiste en que el perro está corriendo a través de los carros y no sólo nosotros lo estamos buscando, hay una banda de ladrones de perros que también están detrás de él, y tenemos que atraparlo antes que ellos.

En el segundo nivel, el perro ha entrado a una quinta y ahora es más difícil todo ya que en esta hay un montón de obstáculos como arbustos, ríos y charcos de lodo que forman una especie de laberinto. Además de que a lo largo de todo el mapa, hay una serie de "pociones" a encontrar que pueden ser ventajas como aumentar tu velocidad. Aunque también pueden ser desventajas, tales como disminuir la velocidad, quedarse congelado, invertir el sentido de las direcciones, etc.

Por último, en el nivel 3 ha caído la noche y ahora no tenemos la visión del mapa completo, sólo contamos con una lámpara que nos permite ver únicamente en un radio específico alrededor de nosotros.

Enemigos/obstáculos: Ladrones, pociones, y carros.

¿Cómo describiría el género(s)? ¿Es para un jugador o para múltiples jugadores (y, si es el último, cooperativo o competitivo)?
El género es de descubrimiento, y es solo para un jugador.

¿Cuál es el público objetivo? Incluya los datos demográficos (la edad, los intereses y la experiencia de juego de los jugadores potenciales), la plataforma del juego (computadora de escritorio, consola o teléfono inteligente) y cualquier equipo especial requerido (como gamepads).
Nuestro juego tiene un ambiente familiar que permite que sea para todas las edades.
La plataforma de juego es ya sea una computadora de escritorio, o una laptop, y no ocupa de equipos especiales requeridos.

d) ¿Por qué la gente querrá jugar a este juego? ¿Qué características distinguen a este juego de títulos similares? ¿Cuál es el gancho que hará que la gente se interese al principio, ¿cómo el juego mantendrá a la gente interesada y qué lo hace divertido?

La gente querrá jugar nuestro juego porque es muy divertido y ameno, además de que existe el rasgo de originalidad, ya que no hay juegos famosos cuya historia sea, ir en busca de tu perro que escapó. Y es justo este nuestro gancho hacía al público, ya que ademas de ser una historia original para un videojuego, es algo que pasa en la vida real, y puede generar empatía, más si has pasado por eso.

2. Mecánica

a) ¿Cuáles son los objetivos del personaje? Estos pueden dividirse en objetivos a corto, mediano y largo plazo.
El objetivo a corto plazo, es primeramente evadir a los carros del nivel 1 para que no te atropellen.
A mediano plazo podemos decir que es encontrar y recolectar pociones que nos puedan dar una ayuda al momento de ir detrás de nuestro objetivo final (sin embargo, no hay que olvidar que estas pociones también pueden causarnos daño, solo la suerte dirá que tipo de poción resulta).
Y evidentemente, nuestro objetivo a largo plazo, es buscar a Nemo, es decir, y detrás del perro, y atraparlo antes de que los ladrones nos ganen, los carros lo atropellen, o llegue al fin del camino antes que nosotros, para así poder ganar.

b) ¿Qué habilidades tiene el personaje? Esto debe incluir cualquier acción que el personaje sea capaz de realizar, como moverse, atacar, defender, recolectar elementos, interactuar con el entorno, etc. Describa las habilidades o acciones en detalle; por ejemplo, ¿qué tan alto puede saltar el personaje? ¿Puede el personaje caminar y correr?
Habilidades:
I) Puede caminar hacia la derecha, izquierda, arriba y abajo.
II) Puede recolectar pociones, y cada una, le dará una habilidad especial, las cuales son:
   a) Ir más rápido.
   b) Ir más lento.
   c) Intercambio de direcciones en los controles.
   d) Congelarse.
   e) Disminuir el radio de visión.
III) Atrapar y salvar al perro.

c) ¿Qué obstáculos o dificultades enfrentará el personaje? Algunos obstáculos están activos (como enemigos, proyectiles o trampas) y deben describirse en detalle (cómo afectan al jugador, su ubicación, patrones de movimiento, etc.). Otros obstáculos son pasivos (como puertas que deben desbloquearse, laberintos que deben ser navegados, rompecabezas que deben resolverse o límites de tiempo que deben superarse). ¿Cómo puede el personaje superar estos obstáculos (objetos, armas, hechizos, reflejos rápidos)?
El personaje en el primer nivel, se encontrará con un primer obstáculo que son los autos que van pasando por la calle, los cuales lo pueden atropellar, ya sea a él, o al perro.
Otra de las dificultades, la cual sucede en todos los niveles, son tambien lo son las pociones, ya que aunque el personaje mismo puede ir a por ellas, para esperar obtener la ventaja de aumentar su velocidad, estas pueden resultar ser desventajas (disminuir

velocidad, intercambio de dirección, congelarse, disminuir su visión del juego(la cual esta sucede durante todo el nivel 3)).
Y pues la otra adversidad con la que se encuentra a lo largo de todo el juego, son los ladrones. Existen ladrones que quieren encontrar el perro antes que el jugador, y si lo logran, el jugador pierde.

d) ¿Qué objetos puede obtener el personaje? ¿Cuáles son sus efectos, dónde se obtienen y con qué frecuencia aparecen?
Puede obtener pociones, solo que no las carga, al tocarlas se aplican en él. Las puede encontrar en todo el momento alrededor de todo el mapa.
Y sus efectos van desde aumentar su velocidad, disminuirla, intercambiar el sentido de las direcciones, además de disminuir el radio de visión del mapa.

e) ¿Qué recursos se deben administrar (como salud, dinero, energía y experiencia)? ¿Cómo se obtienen y utilizan estos recursos? ¿Son limitados?
No cuenta con recursos de este estilo, más que la "vida", la cual solo posee 1.

f) Describa el entorno del mundo del juego. ¿Qué tan grande es el mundo (en relación con la pantalla)? ¿Hay varias habitaciones o regiones? ¿El juego es lineal o abierto? En otras palabras, ¿hay una progresión estrictamente lineal de niveles o tareas para completar, o puede el personaje seleccionar niveles, explorar el mundo y completar misiones a voluntad?
El mundo es lineal, tienes que pasar cada nivel para poder avanzar al siguiente, y siempre inicias en el 1ero.
Y en cuestión del tamaño del mundo, este se genera automáticamente cada vez que inicias un juego y avanzas con el scroll. Mientras que la pantalla mide 640 por 480, nuestro mundo mide 2624 por 640.

3. Dinámica

a) ¿Qué hardware requiere el juego (teclado, mouse, altavoces, gamepad, pantalla táctil)? ¿Qué teclas / botones se usan y cuáles son sus efectos? ¿Cómo se informa al jugador sobre el esquema de control (un documento manual separado, menús del juego, tutoriales o signos en el juego)?
Se utilizan las teclas: "a" o "flecha izquierda", para ir hacia la izquierda; "d" o "flecha derecha", para ir a la derecha; "w" o "flecha arriba", para ir arriba; "s" o "flecha abajo", para ir hacia abajo.

b) ¿Qué tipo de competencia necesitará desarrollar el jugador para dominar el juego? ¿Hay acciones complejas que se puedan crear a partir de combinaciones de mecánicas básicas del juego? ¿La mecánica del juego o el entorno del mundo del juego animan directa o indirectamente al jugador a desarrollar o desalentar cualquier estrategia de juego en particular? ¿El rendimiento del jugador afecta la mecánica del juego ?
La mecánica del juego no cambia, el jugador en si no ocupa dominar acciones complejas, son las mismas desde un inicio, solo evitar los obstáculos, e ir detrás del perro. La única

situación que si puede ser más diferente, es cuando se tome la poción que le invertirá el sentido de las direcciones de las teclas, y deberá adaptarse a esto.

c) ¿Qué datos de juego se muestran durante el juego (como puntos, salud, elementos recopilados, tiempo restante)? ¿Dónde se muestra esta información en la pantalla? ¿Cómo se transmite la información (texto, iconos, gráficos, barras de estado)?
Justo arriba de la pantalla, se muestra el tiempo que llevas en tu partida. Además de que cuando estás bajo los efectos de una poción, se muestra un cronómetro extra que va en decremento, para señalar los segundos que quedan del efecto.

d) ¿Qué menús, pantallas o superposiciones habrá (pantalla de título, ayuda / instrucciones, créditos, juego terminado)? ¿Cómo cambia el jugador entre pantallas y a qué pantallas se puede acceder entre sí?
Existe el menú inicial, en donde puedes clickear “enter” para poder avanzar a las pantallas que indican cómo se juega. Y después de estas, (las cuales pasa al igual con “enter”), inicia el juego, y para poder avanzar a las pantallas de los siguientes niveles, primero tiene que terminar con cada nivel, y automáticamente, pasará a la siguiente.

e) **¿Cómo interactúa el jugador con el juego a nivel de software (pausa, salir, reiniciar, controlar el volumen)?**
La única interacción del jugador es al ganar o perder, ya que podrá elegir si continuar o no.

4. Estética

a) Describe el estilo y la sensación del juego. ¿El juego tiene lugar en un mundo rural, tecnológico o mágico? ¿Se siente el mundo del juego desordenado o escaso, ordenado
o caótico, geométrico u orgánico? ¿El humor es alegre o serio? ¿El ritmo es relajante o
frenético? Todos los elementos estéticos discutidos aquí deberían trabajar juntos y contribuir a crear un tema coherente y cohesivo.
El juego se desarrolla en varios ambientes : la ciudad, un bosque y un bosque a oscuras, el mundo es algo caótico dado que hay varias cosas ocurriendo a la vez, hay rateros , carros, y el perro. Es un juego algo frenético dado que debes perseguir al perro sin que te lo ganen o que se escape, así como que es un poco humorístico.

b) ¿Utiliza el juego pixel art, line art o gráficos realistas? ¿Son los colores brillantes u oscuros, variados o monocromáticos, brillantes u opacos? ¿Habrá animaciones basadas
en valores o imágenes? ¿Hay algún efecto especial? Crear una lista de gráficos que utilizará.
El arte del juego es principalmente pixel art, con colores dependiendo del nivel, desde oscuros como el gris hasta claros como el verde pasando por rojos y amarillos cálidos. Así mismo, tiene animaciones basadas en los sprites de los mismos personajes.

c) ¿Qué estilo de música de fondo o sonidos ambientales usará el juego? ¿Qué efectos de sonido se utilizarán para las acciones de los personajes o para las interacciones con enemigos, objetos y el medio ambiente? ¿Habrá efectos de sonido correspondientes a las interacciones con la interfaz de usuario? Enumera toda la música y los sonidos que necesitarás.

El juego no tiene efectos de sonido como tal. sin embargo tiene música por nivel según la ambientación del mismo, así como por pantalla.

d) ¿Cuál es la historia de fondo relevante para el juego? ¿Cuál es la motivación del personaje para alcanzar su objetivo? ¿Habrá una trama o una historia que se desarrolle a medida que el jugador avance en el juego?

La historia del juego es que se te ha perdido tu perro, por lo que deberás encontrarlo. Evidentemente, al ser la mascota del protagonista, se espera atraparla dado el gran afecto que le tiene.

e) ¿Qué estado(s) emocional(es) intenta provocar el juego: felicidad, emoción, calma, sorpresa, orgullo, tristeza, tensión, miedo, frustración?

Felicidad al atrapar el perro, aunque puede llegar a frustrar dado que no es un juego sencillo del todo.

f) ¿Qué hace que el juego sea "divertido"? Algunos jugadores pueden disfrutar de los gráficos, música, historia o las emociones evocadas por el juego. Otras características que los jugadores pueden disfrutar incluyen la

Lo que lo hace divertido es la estética de 8 bits, además de lo frenético que puede ser perseguir al perrito, a pesar de poseer controles sencillo cuesta y ganar genera una satisfacción que te hace querer volverlo a jugar para superar tu tiempo.

Diagrama De Clases UML:

Dado que el diagrama de clases es muy grande, he aquí un link para visualizarlo :

https://github.com/Gspr-bit/Proyecto-final-S2023-TOO/blob/main/docs/Diagrama%20de%20clases.pdf

Documentación

```
/**
 * Carro
 *
 * @author (Montse)
 * @version (a version number or a date)
 */
public class Car extends Character {
    public int startPositionY;
```

```
    /**
     * Constructor de la clase actor
     *
     * @author Montse
     */
    public Car(int x, int y) {
        this.v = 4;
        this.posX = x;
        this.posY = y;
        this.direction = Direction.DOWN;

        prepare();
    }

    /**
     * Hace el resto de cosas necesarias para inicializar la clase.
     */
    private void prepare() {
        String[] carType = {"a", "b", "c", "d", "e"};
        Random random = new Random();
        String chosenCarType = carType[random.nextInt(carType.length)];
        setImage("Cars/car-" + chosenCarType + ".png");
    }

    /**
     * Act - do whatever the Car wants to do. This method is called whenever
     * the 'Act' or 'Run' button gets pressed in the environment.
     */
    public void act() {
        if (this.isTouchingPlayer() || isTouchingDog()) {
            WindowSwitcher.lose();
        }
    }

    /**
     * Hace que el carro regrese a su posición original cuando llega al final del mapa.
     */
    public void changeDirection() {
        if (this.posY >= WorldMap.MAP_HEIGHT * WorldMap.TILE_SIZE) {
            this.posY = startPositionY;
        }
    }
}

/**
 * Clase padre abstracta para todos los personajes móviles del juego
 *
 * @author Gaspar
```

```
 */
public abstract class Character extends Actor {
    // Update rate of the image FPS/UPDATE_RATE
    protected static final int UPDATE_RATE = 10;  // 6 images per second
    // Velocidad del personaje
    protected int v;
    protected int imageTimer;
    protected Map<Direction, ArrayList<String>> images;
    protected Direction direction;
    // Posición del character en el mapa
    protected int posX;
    protected int posY;

    /**
     * Act - do whatever the character wants to do. This method is called whenever
     * the 'Act' or 'Run' button gets pressed in the environment.
     */
    public abstract void act();

    /**
     * Regresa la posición en píxeles en X del personaje.
     * La posición (x, y) comienza en (0, 0). X se incrementa hacia la derecha.
     * Esta no es la posición en la ventana, sino en el mapa.
     *
     * @return La posición en X del personaje
     */
    public int getPosX() {
        return posX;
    }

    /**
     * Regresa la posición en píxeles en Y del personaje.
     * La posición (x, y) comienza en (0, 0). Y se incrementa hacia abajo.
     * Esta no es la posición en la ventana, sino en el mapa.
     *
     * @return La posición en Y del personaje
     */
    public int getPosY() {
        return posY;
    }

    /**
     * Cambia la dirección del personaje
     */
    public abstract void changeDirection();

    /**
     * Mueve al personaje de acuerdo a su dirección y velocidad.
```

```
 *
 * @author Montse, Gaspar
 */
public void move() {
    switch (this.direction) {
        case UP: {
            this.posY -= v;
            break;
        }
        case DOWN: {
            this.posY += v;
            break;
        }
        case LEFT: {
            this.posX -= v;
            break;
        }
        case RIGHT: {
            this.posX += v;
            break;
        }
        case NONE:
            break;
    }
}

/**
 * Método para saber si el personaje puede moverse hacia la posición dada.
 *
 * @param direction Dirección hacia donde se quiere mover el jugador.
 * @return true si el jugador se puede mover hacia allá
 * @author Mauricio, Gaspar, Montse
 */
public boolean canMoveTowards(Direction direction) {
    // Siempre se puede quedar donde ya está
    if (direction == Direction.NONE) return true;

    int dx = this.getImage().getWidth() / 2 + v;
    int dy = this.getImage().getHeight() / 2 + v;

    // UP, DOWN, LEFT, RIGHT
    int[] dxs = {0, 0, -dx, dx};
    int[] dys = {-dy, dy, 0, 0};

    Tile nextTile = (Tile) this.getOneObjectAtOffset(dxs[direction.ordinal()],
            dys[direction.ordinal()], Tile.class);
```

```
        Character nextCharacter = (Character)
this.getOneObjectAtOffset(dxs[direction.ordinal()],
                dys[direction.ordinal()], Character.class);

        return nextTile != null && nextCharacter == null && !nextTile.isCollidable();
    }

    /**
     * Cambiar la imagen del jugador para que parezca como si se estuviera moviendo
     *
     * @author Gaspar
     */
    public void updateImage() {
        if (this.direction == Direction.NONE) return;

        setImage(images.get(this.direction).get(imageTimer / UPDATE_RATE));

        this.imageTimer++;
        if (this.imageTimer / UPDATE_RATE >= images.get(this.direction).size())
            this.imageTimer = 0;
    }

    /**
     *
     * @return si el ladrón está tocando al jugador.
     */
    public boolean isTouchingPlayer() {
        return this.getOneIntersectingObject(Player.class) != null;
    }

    /**
     *
     * @return si el ladrón está tocando al perro.
     */
    public boolean isTouchingDog() {
        return this.getOneIntersectingObject(Dog.class) != null;
    }

    public void drawCharacter(MyWorld world) {
        int objectPosX = this.getPosX() - world.getPlayer().getPosX();
        int objectPosY = this.getPosY() - world.getPlayer().getPosY();

        if (objectPosX >= 0 && objectPosX < world.getWidth() && objectPosY >= 0 &&
objectPosY < world.getHeight()) {
            world.addObject(this, objectPosX + WorldMap.TILE_SIZE / 2, objectPosY +
WorldMap.TILE_SIZE / 2);
        }
    }
```

```
}
```
Direcciones del personaje
```
public enum Direction {
    UP, DOWN, LEFT, RIGHT, NONE
}

/**
 * Dog
 * Write a description of class Dog here.
 *
 * @author Gaspar
 */
public class Dog extends Character {
    private static final int MOVEMENT_DURATION = 1;
    private static final int IDLE_DURATION = 3;
    // Límite de distancia a la que se puede acercar el jugador antes de que el perro corra de
nuevo
    private static final int DISTANCE_THRESHOLD = 4 * WorldMap.TILE_SIZE;
    private final Tile[][] map;
    private final PathFinder pathFinder;
    private int movementStartTime;
    private int movementEndTime;
    private boolean hide;

    public Dog(int x, int y, Tile[][] map) {
        this.map = map;
        this.v = 2;
        this.posX = x;
        this.posY = y;
        this.pathFinder = new PathFinder(this.map);
        this.movementStartTime = 0;
        this.hide = false;
        this.direction = Direction.RIGHT;

        prepare();
    }

    /**
     * Hace el resto de cosas necesarias para inicializar la clase.
     */
    private void prepare() {
        Direction[] d = {Direction.UP, Direction.DOWN, Direction.LEFT, Direction.RIGHT};
        String[] f = {"up", "down", "left", "right"};

        this.images = new HashMap<>(4);

        //Establecer las imágenes
        for (int i = 0; i < 4; i++) {
```

```
        images.put(d[i], new ArrayList<>(3));

        for (int j = 0; j < 3; j++) {

            images.get(d[i]).add("Dogs/dog-" + f[i] + "-" + j + ".png");


        }
    }
}

public boolean isHidden() {
    return hide;
}

/**
 * Act - do whatever the Dog wants to do. This method is called whenever
 * the 'Act' or 'Run' button gets pressed in the environment.
 */
public void act() {
    changeDirection();
    move();
    updateImage();

    if (isTouchingThief()) {
        WindowSwitcher.lose();
    }

    if (isTouchingPlayer()) {
        WindowSwitcher.nextLevel(((MyWorld) this.getWorld()).getLevel());
    }
}


@Override
public void changeDirection() {
    // Solo cambia su dirección cuando está justo en medio de un Tile
    // De otra manera puede suceder un desfase extraño
    if (this.posX % WorldMap.TILE_SIZE != 0 || this.posY % WorldMap.TILE_SIZE != 0)
        return;

    int i = (this.posX / WorldMap.TILE_SIZE);
    int j = (this.posY / WorldMap.TILE_SIZE);
    try {
        this.direction = pathFinder.findDirection(i, j);
    } catch (PathEmptyException | InvalidPointException | EndOfPathException e) {
        this.direction = pathFinder.findPath(i, j);
```

```
            System.err.println("Advertencia: El camino ha sido recalculado, esto puede causar
problemas de rendimiento");
        }

        // Cambia su dirección solo cuando es momento
        int time = Timer.getTime();

        if (time == this.movementStartTime || isNearToPlayer()) {
            this.movementEndTime = time + MOVEMENT_DURATION;
        } else if (time == this.movementEndTime) {
            this.movementStartTime = time + IDLE_DURATION;
        }

        if (time > this.movementEndTime && time < this.movementStartTime) {
            this.direction = Direction.NONE;
            this.hide = canHide();
            return;
        }

        this.hide = false;
    }

    private boolean canHide() {
        int i = (this.posX / WorldMap.TILE_SIZE);
        int j = (this.posY / WorldMap.TILE_SIZE);

        int[] dis = {0, 0, 1, -1};
        int[] djs = {1, -1, 0, 0};

        for (int k = 0; k < 4; k++)
            if (map[i + dis[k]][j + djs[k]].isCollidable())
                return true;

        return false;
    }

    private boolean isNearToPlayer() {
        return !this.getObjectsInRange(DISTANCE_THRESHOLD, Player.class).isEmpty();
    }

    private boolean isTouchingThief() {
        return !this.getObjectsInRange(WorldMap.TILE_SIZE, Thief.class).isEmpty();
    }
}

/**
 *Direction
 *Cuadros por los que está formado el mapa.
```

```
 *
 * @author Gaspar
 */
public class Tile extends Actor {
    private final TileType type;
    private final boolean collidable;

    public Tile(GreenfootImage image, TileType type) {
        setImage(image);
        this.type = type;
        // Tipos de tile donde no se puede pisar.
        this.collidable =
                this.type == TileType.BUSH ||
                        this.type == TileType.WATER ||
                        this.type == TileType.TREE ||
                        this.type == TileType.ROCK ||
                        this.type == TileType.WALL ||
                        this.type == TileType.ROOF;
    }

    /**
     * Regresa el tipo de piso.
     * @return Tipo de piso
     */
    public TileType getType() {
        return type;
    }

    /**
     * Regresa si no se puede pisar sobre este Tile.
     * @return 'True' si no se puede pisar en este Tile. 'False' de otro modo
     */
    public boolean isCollidable() {
        return collidable;
    }

    public enum TileType {BUSH, DIRT, GRASS, MUD, ROCK, TREE, WATER, PAVEMENT,
WALL, ROOF}
}

/**
     * Effect
     * Regresa un efecto aleatorio, excepto NONE.
     *
     * @return un efecto aleatorio.
     * @author Gaspar
 */
public enum Effect {
```

```
    NONE, FREEZE, DIZZY, SLOW, BLIND, FAST;

    public static Effect randomEffect() {
        Random random = new Random();
        Effect[] effects = Effect.values();
        return effects[random.nextInt(effects.length - 1) + 1];
    }
}

/**
 * Esta superclase se utiliza para los objetos que deben estar fijos en el mapa.
 *
 * @author Montse
 */
public abstract class FixedObject extends Actor {

    protected int posX;
    protected int posY;

    protected FixedObject(int x, int y) {
        this.posX = x;
        this.posY = y;
        setLocation(posX, posY);
    }

    /**
     * Regresa la posición en Tiles en X del personaje.
     * La posición (x, y) comienza en (0, 0). X se incrementa hacia la derecha.
     * Esta no es la posición en la ventana, sino en el mapa.
     *
     * @return La posición en X del personaje
     */
    public int getPosX() {
        return posX;
    }

    /**
     * Regresa la posición en Tiles en Y del personaje.
     * La posición (x, y) comienza en (0, 0). Y se incrementa hacia abajo.
     * Esta no es la posición en la ventana, sino en el mapa.
     *
     * @return La posición en X del personaje
     */
    public int getPosY() {
        return posY;
    }

    /**
```

```
     * Función que hace que los items desaparezcan cuando el jugador los toca.
     * Los objetos están guardados dentro en MyWorld FixedObjects
     * Entonces si queremos eliminar un objeto, no solo lo borramos del mundo,
     * también del ArrayList.
     *
     * @author Montse
     */
    public void remove() {
        ((MyWorld) getWorld()).getFixedObjects().remove(this);
        getWorld().removeObject(this);
    }
}

/**
 * Ventana que se muestra cuando el jugador pierde.
 *
 * @author (your name)
 * @version Mau
 */
public class GameOverScreen extends World {

    /**
     * Constructor for objects of class GameOverScreen.
     */
    public GameOverScreen() {
        // Create a new world with 600x400 cells with a cell size of 1x1 pixels.
        super(600, 400, 1);
        prepare();
    }

    /**
     * Lee las opciones del jugador.
     * 'Y' para jugar de nuevo.
     * 'N' para salir del juego.
     */
    public void act() {
        if (Greenfoot.isKeyDown("Y")) {
            Greenfoot.delay(10);
            WindowSwitcher.showLevel(1);
        }
        if (Greenfoot.isKeyDown("N")) {
            System.exit(0);
        }
    }

    /**
     * Hace el resto de cosas necesarias para inicializar la clase.
     */
```

```
    private void prepare() {
        setBackground("Screens/game-over-screen.png");
    }
}

/**
 * Ventana que se muestra al iniciar el juego.
 *
 * @author Mau
 */
public class InitialScreen extends World
{

    /**
     * Constructor for objects of class InitialScreen.
     *
     */
    public InitialScreen()
    {
        // Create a new world with 600x400 cells with a cell size of 1x1 pixels.
        super(600,400,1);
        prepare();
    }

    /**
     * El jugador debe presionar 'enter' para entrar al juego.
     */
    public void act(){
        if(Greenfoot.isKeyDown("ENTER")){
            InstructionsScreen1 world = new InstructionsScreen1();
            // Esto es necesario porque Greenfoot no soporta
            // keyRelease() como cualquier framework decente haría.
            Greenfoot.delay(10);
            Greenfoot.setWorld(world);
        }

    }

    /**
     * Prepare the world for the start of the program.
     * That is: create the initial objects and add them to the world.
     */
    private void prepare() {
        setBackground("Screens/initial-screen.png");
    }
}

/**
```

```
 * Pantalla 1 de instrucciones.
 *
 * @author Mau
 */
public class InstructionsScreen1 extends World {
    /**
     * Constructor for objects of class InstructionsScreen1.
     */
    public InstructionsScreen1() {
        // Create a new world with 600x400 cells with a cell size of 1x1 pixels.
        super(600, 400, 1);
        prepare();
    }

    public void act() {

        if (Greenfoot.isKeyDown("ENTER")) {
            InstructionsScreen2 world1 = new InstructionsScreen2();
            Greenfoot.delay(10);
            Greenfoot.setWorld(world1);
        }
    }

    /**
     * Prepare the world for the start of the program.
     * That is: create the initial objects and add them to the world.
     */
    private void prepare() {
        setBackground("Screens/instructions-1.png");
    }
}

/**
 * InstructionsScreen2
 *
 * @author Mau
 */
public class InstructionsScreen2 extends World {

    /**
     * Constructor for objects of class InstructionsScreen2.
     */
    public InstructionsScreen2() {
        // Create a new world with 600x400 cells with a cell size of 1x1 pixels.
        super(600, 400, 1);
        prepare();
    }
```

```
    public void act() {
        if (Greenfoot.isKeyDown("ENTER")) {
            Greenfoot.delay(10);
            WindowSwitcher.showLevel(1);
        }
    }

    /**
     * Prepare the world for the start of the program.
     * That is: create the initial objects and add them to the world.
     */
    private void prepare() {
        setBackground("Screens/instructions-2.png");
    }
}

/**
 * Clase que representa una poción.
 *
 * @author Gaspar
 */
public class Item extends FixedObject {
    private final Effect effect;
    private final int effectDuration;

    public Item(Effect effect, int effectDuration, int x, int y) {
        super(x, y);//constructor de objeto fijo

        this.effect = effect;
        this.effectDuration = effectDuration;

        prepare();
    }

    /**
     * Prepare the world for the start of the program.
     * That is: create the initial objects and add them to the world.
     */
    private void prepare() {
        String[] colors = {"green", "purple", "yellow"};

        Random random = new Random();
        String chosenColor = colors[random.nextInt(colors.length)];
        setImage("Items/potion-" + chosenColor + ".png");
    }

    /**
     * @return El tipo de efecto.
```

```
     */
    public Effect getEffect() {
        return effect;
    }

    /**
     * @return La duración del efecto.
     */
    public int getEffectDuration() {
        return effectDuration;
    }
}


/**
 * Ventana que se muestra antes de comenzar el nivel 1.
 *
 * @author Mau
 */
public class Level1Intro extends World
{

    /**
     * Constructor for objects of class Level1Intro.
     *
     */
    public Level1Intro()
    {
        // Create a new world with 600x400 cells with a cell size of 1x1 pixels.
        super(600, 400, 1);
        Timer.reset();
        prepare();
    }

    /**
     * El jugador debe presionar 'enter' para continuar.
     */
    public void act() {
        if (Greenfoot.isKeyDown("ENTER")) {
            try {
                MyWorld world = new MyWorld(1);
                Greenfoot.delay(10);
                Greenfoot.setWorld(world);
            } catch (WorldMap.WrongGenerationPercentagesException e) {
                throw new RuntimeException(e);
            }
        }
    }
```

```
    /**
     * Prepare the world for the start of the program.
     * That is: create the initial objects and add them to the world.
     */
    private void prepare() {
        setBackground("LevelImages/level-1.png");
    }
}

/**
 * Ventana que se muestra antes de comenzar el nivel 2.
 *
 * @author Mau
 */
public class Level2Intro extends World {

    /**
     * Constructor for objects of class Level2Intro.
     */
    public Level2Intro() {
        // Create a new world with 600x400 cells with a cell size of 1x1 pixels.
        super(600, 400, 1);
        prepare();
    }

    /**
     * El jugador debe presionar 'enter' para continuar.
     */
    public void act() {
        if (Greenfoot.isKeyDown("ENTER")) {
            try {
                MyWorld world = new MyWorld(2);
                Greenfoot.delay(10);
                Greenfoot.setWorld(world);
            } catch (WorldMap.WrongGenerationPercentagesException e) {
                throw new RuntimeException(e);
            }
        }
    }

    /**
     * Prepare the world for the start of the program.
     * That is: create the initial objects and add them to the world.
     */
    private void prepare() {
        setBackground("LevelImages/level-2.png");
    }
```

```
}
/**
 * Ventana que se muestra antes de comenzar el nivel 3.
 *
 * @author Mau
 */
public class Level3Intro extends World {

    /**
     * Constructor for objects of class Level3Intro.
     */
    public Level3Intro() {
        // Create a new world with 600x400 cells with a cell size of 1x1 pixels.
        super(600, 400, 1);
        prepare();
    }

    public void act() {
        if (Greenfoot.isKeyDown("ENTER")) {
            try {
                MyWorld world = new MyWorld(3);
                Greenfoot.delay(10);
                Greenfoot.setWorld(world);
            } catch (WorldMap.WrongGenerationPercentagesException e) {
                throw new RuntimeException(e);
            }
        }
    }

    /**
     * Prepare the world for the start of the program.
     * That is: create the initial objects and add them to the world.
     */
    private void prepare() {
        setBackground("LevelImages/level-3.png");
    }
}



/**
 * Mundo principal del juego.
 *
 * @author Mau, Montse, Gaspar
 */
public class MyWorld extends World {
    public static final int WORLD_WIDTH = 40 * WorldMap.TILE_SIZE;
    public static final int WORLD_HEIGHT = 30 * WorldMap.TILE_SIZE;
```

```
    private final Player player;
    private final Dog dog;
    private final Random random;
    private final WorldMap worldMap;
    private final ArrayList<FixedObject> fixedObjects;
    private final ArrayList<Thief> thieves;
    private final ArrayList<Car> cars;
    private final int level;
    private Shadow shadow;

    /**
     * Constructor for objects of class MyWorld.
     */
    public MyWorld(int level) throws WorldMap.WrongGenerationPercentagesException {
        super(WORLD_WIDTH, WORLD_HEIGHT, 1);

        this.level = level;
        this.random = new Random(new Date().getTime());
        this.fixedObjects = new ArrayList<>();
        this.worldMap = new WorldMap(new Date().getTime(), level);
        this.dog = new Dog(this.getWidth() / 2 + 8, this.getHeight() / 2,
worldMap.getMapTiles());
        this.player = new Player();
        this.cars = new ArrayList<>();
        this.thieves = new ArrayList<>();

        prepare();
    }

    private void prepare() {
        setPaintOrder(Shadow.class, Car.class, Player.class, Dog.class, Thief.class,
FixedObject.class, Tile.class);

        this.player.setLocation(this.getWidth() / 2, this.getHeight() / 2);
        this.addObject(player, this.getWidth() / 2, this.getHeight() / 2);

        try {
            this.shadow = new Shadow(level == 3 ? 14 : 0);
            this.addObject(this.shadow, this.getWidth() / 2, this.getHeight() / 2);
        } catch (InvalidShadowSizeExceptions e) {
            // No debería entrar aquí.
            throw new RuntimeException(e);
        }

        generateItems();
        generateThieves();
        if (this.level == 1)
            generateCars();
```

```
    }

    public Player getPlayer() {
        return player;
    }

    public int getLevel() {
        return level;
    }

    public ArrayList<FixedObject> getFixedObjects() {
        return this.fixedObjects;
    }

    public Shadow getShadow() {
        return shadow;
    }

    public void act() {
        this.worldMap.drawMap(this);
        drawFixedObjects();
        drawThieves();
        drawDog();
        drawCars();
        Timer.update();
        showTimer();

        // revisar si el perro ya llegó al otro lado
        if (this.dog.getPosX() + this.dog.getImage().getWidth() >= WorldMap.MAP_WIDTH *
WorldMap.TILE_SIZE) {
            // reiniciar el nivel
            WindowSwitcher.showLevel(this.level);
        }

        this.cars.forEach(car -> {
            car.move();
            car.changeDirection();
        });
    }

    /**
     * Genera los items que serán mostrados en el mapa
     *
     * @author Gaspar
     */
    private void generateItems() {
        int count = 10;
```

```
        while (count > 0) {
            int x = random.nextInt(WorldMap.MAP_WIDTH);
            int y = random.nextInt(WorldMap.MAP_HEIGHT);

            if (worldMap.getMapTiles()[x][y].isCollidable())
                continue;

            fixedObjects.add(new Item(Effect.randomEffect(), random.nextInt(8) + 2, x, y));
            count--;
        }
    }

    /**
     * Genera a los ladrones
     *
     * @author Montse
     */
    private void generateThieves() {
        int count = 5;

        while (count > 0) {
            int x = random.nextInt(WorldMap.MAP_WIDTH);
            int y = random.nextInt(WorldMap.MAP_HEIGHT);

            if (worldMap.getMapTiles()[x][y].isCollidable())
                continue;

            thieves.add(new Thief(x * WorldMap.TILE_SIZE, y * WorldMap.TILE_SIZE));
            count--;
        }
    }

    private void generateCars() {
        int numberOfVerticalStreets = WorldMap.MAP_WIDTH / 16 + (WorldMap.MAP_WIDTH
% 16 > 0 ? 1 : 0);

        for (int i = 0; i < 5; i++) {
            int randomStreet = random.nextInt(numberOfVerticalStreets);
            int y = random.nextInt(WorldMap.MAP_HEIGHT * WorldMap.TILE_SIZE);
            int x = randomStreet * 16 * WorldMap.TILE_SIZE + 32;

            Car car = new Car(x, y);
            car.startPositionY = 0;

            this.cars.add(car);
        }
    }
```

```
    private void drawFixedObjects() {
        List<FixedObject> objectsInMap = this.getObjects(FixedObject.class);
        removeObjects(objectsInMap);

        // Pintar los objetos en el mapa
        this.fixedObjects.forEach(object -> {
            int objectPosX = object.getPosX() * WorldMap.TILE_SIZE - player.getPosX();
            int objectPosY = object.getPosY() * WorldMap.TILE_SIZE - player.getPosY();

            if (objectPosX >= 0 && objectPosX < getWidth() && objectPosY >= 0 && objectPosY
< getHeight()) {
                addObject(object, objectPosX + object.getImage().getWidth() / 2, objectPosY +
object.getImage().getHeight() / 2);
            }
        });
    }

    private void drawDog() {
        removeObject(dog);

        // No dibujar al perro si está oculto
        if (dog.isHidden()) {
            dog.setImage("Dogs/dog-invisible.png");
        } else {
            dog.setImage("Dogs/dog-right-0.png");
        }

        dog.drawCharacter(this);
    }

    private void drawCars() {
        List<Car> objectsInMap = this.getObjects(Car.class);
        removeObjects(objectsInMap);

        // Pintar los carros en el mapa
        this.cars.forEach(object -> object.drawCharacter(this));
    }

    private void drawThieves() {
        List<Thief> objectsInMap = this.getObjects(Thief.class);
        removeObjects(objectsInMap);

        // Pintar los ladrones en el mapa
        this.thieves.forEach(object -> object.drawCharacter(this));
    }

    public void showTimer() {
        String s = player.getEffect() + " " + (player.getEffectEnd() - Timer.getTime());
```

```
        if (player.getEffectEnd() - Timer.getTime() == 0) {
            s = "";
        }
        if (player.getEffect() != Effect.NONE) {
            this.showText(s, MyWorld.WORLD_WIDTH / 2, 60);
        }
        this.showText("Tiempo: " + Timer.getTime(), MyWorld.WORLD_WIDTH / 2, 30);
    }
}


/**
/**
 * Clase Jugador. Esta clase es instanciada desde el mundo.
 *
 * @author Mauricio, Montse, Gaspar
 * @version (a version number or a date)
 */

public class Player extends Character {
    // Tipo de efecto
    private Effect effect;
    // Momento en que se dejará de aplicar el efecto en segundos
    private int effectEnd;
    // Visibilidad del jugador. El jugador podrá ver en el radio especificado
    private int visibility;
    // Aquí guardamos el valor que debe tener `v` cuando no tiene ningún efecto
    private final int defaultV;

    public Player() {
        this.effect = Effect.NONE;
        // Duración del efecto
        this.visibility = 0;
        this.v = this.defaultV = 2;
        this.direction = Direction.RIGHT;
        this.posX = this.posY = 0;

        prepare();
    }

    /**
     * Prepara las imágenes del jugador.
     */
    private void prepare() {
        Direction[] d = {Direction.UP, Direction.DOWN, Direction.LEFT, Direction.RIGHT};
        String[] f = {"up", "down", "left", "right"};

        this.images = new HashMap<>(4);
```

```
        // Establecer las imágenes
        for (int i = 0; i < 4; i++) {
            images.put(d[i], new ArrayList<>(6));

            for (int j = 0; j < 6; j++) {
                images.get(d[i]).add("Player/" + f[i] + "-" + j + ".png");
            }
        }

        this.imageTimer = 0;
    }

    public Effect getEffect() {
        return effect;
    }

    public void setEffect(Effect effect) {
        this.effect = effect;
    }

    public int getVisibility() {
        return visibility;
    }

    public void setVisibility(int visibility) {
        this.visibility = visibility;
    }

    public int getEffectEnd(){
        return this.effectEnd;
    }


    public void act(){
        changeDirection();
        updateImage();
        applyEffect();
    }

    /**

     * Método para cambiar la dirección y posición del jugador.
     * @author Mauricio, Montse
     */
    public void changeDirection() {
        // Este método no mueve al jugador, solo cambia su sprite para que apunte a la
dirección correspondiente
```

```
    if (Greenfoot.isKeyDown("right") || Greenfoot.isKeyDown("D")) {
        this.direction = Direction.RIGHT;
        if (this.v < 0) {
            this.direction = Direction.LEFT;
        }
        if (canMoveTowards(this.direction)) {
            this.posX += v;
        }
    }

    if (Greenfoot.isKeyDown("left") || Greenfoot.isKeyDown("A")) {
        this.direction = Direction.LEFT;
        if (this.v < 0) {
            this.direction = Direction.RIGHT;
        }
        if (canMoveTowards(this.direction)) {
            this.posX -= v;
        }
    }
    if (Greenfoot.isKeyDown("up") || Greenfoot.isKeyDown("W")) {
        this.direction = Direction.UP;
        if (this.v < 0) {
            this.direction = Direction.DOWN;
        }
        if (canMoveTowards(this.direction)) {
            this.posY -= v;
        }
    }
    if (Greenfoot.isKeyDown("down") || Greenfoot.isKeyDown("S")) {
        this.direction = Direction.DOWN;
        if (this.v < 0) {
            this.direction = Direction.UP;
        }
        if (canMoveTowards(this.direction)) {
            this.posY += v;
        }
    }
}

public void applyEffect() {
    Item item = (Item) this.getOneIntersectingObject(Item.class);
    if (item != null) {
        // Momento en que se comenzó a aplicar el efecto en segundos
        int effectStart = Timer.getTime();
        this.effectEnd = effectStart + item.getEffectDuration();
        this.effect = item.getEffect();
        item.remove();
        if (this.effect == Effect.BLIND) {
```

```
                try {
                    ((MyWorld) getWorld()).getShadow().setSize((new Random().nextInt(6) * 2) +
2);
                } catch (InvalidShadowSizeExceptions e) {
                    throw new RuntimeException(e);
                }
            }
        } else if (Timer.getTime() < effectEnd) {
            switch (this.effect) {
                case SLOW: {
                    this.v = 1;
                    break;
                }
                case DIZZY: {
                    this.v = -defaultV;
                    break;
                }
                case FREEZE: {
                    this.v = 0;
                    break;
                }
                case FAST: {
                    this.v = 4;
                }
                case BLIND: {
                    // Ya lo hicimos arriba
                    break;
                }
                case NONE: {
                    // No hacer nada
                }
            }
        } else {
            // Regresar las cosas a la normalidad
            this.v = this.defaultV;
            ((MyWorld) getWorld()).getShadow().reset();
            this.effect=Effect.NONE;
        }
    }
}
/**
 *
 *Clase que implementa las funciones necesarias para encontrar caminos
 *
 * @author Gaspar
 */
public class PathFinder
{
```

```
    private static final int INF = (int) 10e9;
    private final ArrayList<Point> path;
    private final Tile [][] map;
    private final int [][] distances;

    public PathFinder(Tile [][] map)
    {
        this.path = new ArrayList<>();
        this.map = map;
        this.distances = new int[map.length][map[0].length];
    }

    /**
     * Regresa la dirección hacia donde debería caminar cuando está en el punto 0, 0
     * @param i Posición x del jugador en Tiles
     * @param j Posición y del jugador en Tiles
     * @return Dirección hacia donde debería caminar
     */
    public Direction findDirection(int i, int j) throws PathEmptyException,
                                                        InvalidPointException,
                                                        EndOfPathException {
        int index = this.path.indexOf(new Point(i, j));

        if (this.path.isEmpty()) {
            throw new PathEmptyException("El camino no ha sido inicializado. Por favor ejecuta
findPath()");
        }

        if (index == -1) {
            throw new InvalidPointException("No estás dentro del camino");
        }

        Point nextPoint;
        try {
            nextPoint = this.path.get(index + 1);
        } catch (IndexOutOfBoundsException e) {
            throw new EndOfPathException("Haz llegado al final del camino");
        }

        int di = nextPoint.i - i;
        int dj = nextPoint.j - j;

        if (di > 0)
            return Direction.RIGHT;

        if (di < 0)
            return Direction.LEFT;
```

```
        if (dj > 0)
            return Direction.DOWN;

        if (dj < 0)
            return Direction.UP;

        // No debería llegar así
        throw new RuntimeException("Algo extraño sucedió");
    }

    /**
     * Busca el camino desde (i, j) hasta el camino
     * punto más a la izquierda posible.
     * Regresa el camino.
     * @param i Posición x del jugador en Tiles
     * @param j Posición y del jugador en Tiles
     * @return Dirección hacia donde debería caminar.
     */
    public Direction findPath(int i, int j) {
        for (int [] row : distances)
            Arrays.fill(row, INF);

        path.clear();

        try {
            Point targetPoint = bfs(i, j);
            retrievePath(targetPoint);
            path.add(0, new Point(i, j));

            return this.findDirection(i, j);
        } catch (InvalidPointException | PathEmptyException | EndOfPathException e) {
            throw new RuntimeException(e + " (i, j) = " + i + ", " + j + " path = " + path);
        }
    }

    /**
     * Implementación de BFS para encontrar el camino más corto.
     * Nota que esta función es bastante costosa por lo que solo debería ser llamada una sola
vez.
     * @param i Posición x del jugador en Tiles
     * @param j Posición Y del jugador en Tiles
     * @return El punto más a la izquierda más cercano.
     * @throws InvalidPointException cuando queremos comenzar la búsqueda en un punto
donde no puede estar el personaje
     */
    private Point bfs(int i, int j) throws InvalidPointException {
        Queue<Point> queue = new LinkedList<>();
```

```
        if (i < 0 || i >= map.length || j < 0 || j >= map[0].length || map[i][j].isCollidable()) {
            throw new InvalidPointException("El punto (" + i + ", " + j + ") no es un punto válido.
path = " + path);
        }

        int [] dis = {0, 0, 1, -1};
        int [] djs = {1, -1, 0, 0};

        distances[i][j] = 0;

        Point rightMostPoint = new Point(i, j);

        queue.add(new Point(i, j));
        while (!queue.isEmpty()) {
            Point p = queue.poll();
            int pi = p.i;
            int pj = p.j;

            for (int k = 0; k < 4; k++) {
                int di = dis[k];
                int dj = djs[k];
                int ni = pi + di;
                int nj = pj + dj;
                if (ni >= 0 && ni < map.length && nj >= 0 && nj < map[0].length) {
                    if (!map[ni][nj].isCollidable() && distances[ni][nj] > distances[pi][pj] + 1) {
                        distances[ni][nj] = distances[pi][pj] + 1;
                        queue.add(new Point(ni, nj));

                        if (ni > rightMostPoint.i ||
                                (ni == rightMostPoint.i &&
                                        distances[ni][nj] < distances[rightMostPoint.i][rightMostPoint.j]))
                            rightMostPoint = new Point(ni, nj);
                    }
                }
            }
        }

        return rightMostPoint;
    }

    /**
     * Construye el camino a partir de la matriz `distances` que genera el BFS.
     */
    private void retrievePath(Point target) {
        int i = target.i;
        int j = target.j;
        int distance = distances[i][j];
        int [] dis = {0, 0, 1, -1};
```

```
        int [] djs = {1, -1, 0, 0};

        while (distance > 0) {
            for (int k = 0; k < 4; k++) {
                int di = dis[k];
                int dj = djs[k];
                int ni = i + di;
                int nj = j + dj;
                if (ni >= 0 && ni < distances.length && nj >= 0 && nj < distances[0].length) {
                    if (distances[ni][nj] == distance - 1) {
                        path.add(0, new Point(i, j));
                        i = ni;
                        j = nj;
                        distance--;
                        break;
                    }
                }
            }
        }
    }
}

class InvalidPointException extends Exception {
    public InvalidPointException(String message) {
        super(message);
    }
}

class PathEmptyException extends Exception {
    public PathEmptyException(String message) {
        super(message);
    }
}

class EndOfPathException extends Exception {
    public EndOfPathException(String message) {
        super(message);
    }
}
```

```
 * Representación de un punto en un plano cartesiano.
 *
 * @author Gaspar
 */
class Point {
    protected int i;
    protected int j;
```

```
    public Point(int i, int j) {
        this.i = i;
        this.j = j;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Point point = (Point) o;
        return i == point.i && j == point.j;
    }

    @Override
    public int hashCode() {
        return Objects.hash(i, j);
    }

    public String toString() {
        return "(" + i + ", " + j + ")";
    }
}
```

```
/**
 * Sombra que oscurece el mapa.
 * Esta sombra siempre debe estar en el centro de la ventana.
 *
 * @author (Gaspar)
 * @version (a version number or a date)
 */
public class Shadow extends Actor {
    private final int defaultSize;

    /**
     * @param size Tamaño de la sombra. 0 significa que no hay sombra.
     * @throws InvalidShadowSizeExceptions El tamaño especificado no es válido. Debe ser
un número par entre 0 y 14
     */
    public Shadow(int size) throws InvalidShadowSizeExceptions {
        this.defaultSize = size;
        setSize(size);
    }

    /**
     * Cambia el tamaño de la sombra.
```

```
     *
     * @param size Tamaño de la sombra. 0 significa que no hay sombra.
     * @throws InvalidShadowSizeExceptions El tamaño especificado no es válido. Debe ser
un número par entre 0 y 14
     */
    public void setSize(int size) throws InvalidShadowSizeExceptions {
        if (size < 0 || size > 14 || size % 2 != 0) {
            throw new InvalidShadowSizeExceptions("El tamaño de la sombra sólo puede ser un
número par entre 0 y 14");
        }
        setImage("Shadows/shadow-" + size + ".png");
    }

    /**
     * Regresa la sombra a su tamaño original.
     */
    public void reset() {
        try {
            setSize(defaultSize);
        } catch (InvalidShadowSizeExceptions e) {
            // No debería llegar aquí
            throw new RuntimeException(e);
        }
    }
}

class InvalidShadowSizeExceptions extends Exception {
    public InvalidShadowSizeExceptions(String message) {
        super(message);
    }
}

 * Temporizador del juego.
 *
 * @author (Gaspar)
 * @version (a version number or a date)
 */
public class Timer {
    private static final int FPS = 60;

    private static int time = 0;

    /**
     * Regresa el tiempo en segundos
     * transcurrido desde que se inició el juego
     *
     * @return tiempo en segundos
     * @author Gaspar
```

```
     */
    public static int getTime() {
        return time / FPS;
    }

    /**
     * Actualiza el temporizador
     * Este método debe ser llamado dentro de act() de un solo objeto en el juego
     *
     * @author Gaspar
     */
    public static void update() {
        time++;
    }

    /**
     * Reinicia el temporizador a 0
     */
    public static void reset()  {
        time = 0;
    }

}
/**
 * Ladrón
 *
 * @author Montse
 */
public class Thief extends Character
{
    // Momento en que el ladrón comenzó a moverse hacia una dirección
    private int movementStart;
    // Tiempo en que el ladrón continúa moviéndose hacia la misma dirección
    // antes de cambiar de dirección.
    private final static int MOVEMENT_DELAY = 2;
    private final Random random;

    /**
    *Constructor clase Thief
    *@author Montse
    */
    public Thief(int x, int y){
        this.movementStart = 0;
        this.v = 1;
        this.direction = Direction.RIGHT;
        this.posX=x;
        this.posY=y;
        this.random = new Random((long) x*y);
```

```
    prepare();
}

/**
 * Prepara las imágenes del ladrón.
 */
private void prepare() {
    Direction[] d = {Direction.UP, Direction.DOWN, Direction.LEFT, Direction.RIGHT};
    String[] f = {"up", "down", "left", "right"};

    this.images = new HashMap<>(4);

    // Tipos de ladrones
    String [] type = {"a", "b"};

    Random random = new Random();

    //Establecer las imágenes
    for (int i = 0; i < 4; i++) {
        images.put(d[i], new ArrayList<>(2));

        String chosenTipo = type[random.nextInt(type.length)];
        for (int j = 0; j < 2; j++) {

            images.get(d[i]).add("Thief/"+ chosenTipo +"-"+ f[i] + "-" + j + ".png");


        }
    }

    this.imageTimer = 0;
}

/**
 * Act - do whatever the Thief wants to do. This method is called whenever
 * the 'Act' or 'Run' button gets pressed in the environment.
 */
public void act()
{
    if (isTouchingPlayer()) {
        dropItem();
    }

    updateImage();
    changeDirection();
    move();
}
```

```
    /**
     * Función que cambia la dirección del ladrón aleatoriamente cada 2 segundos
     * o cuando choque con un objeto.
     *
     * @author Montse
     */
    @Override
    public void changeDirection() {
        // Determinar si el momento en que debe de cambiar su dirección ha llegado.
        // O si está a punto de chocar
        if (Timer.getTime() >= this.movementStart + MOVEMENT_DELAY ||
            !canMoveTowards(this.direction))
        {
            this.movementStart = Timer.getTime();
            this.direction = Direction.values()[random.nextInt(4)];
        }
    }

    /**
     * Suelta un item aleatorio en su posición actual.
     */
    private void dropItem() {
        Item item = new Item(Effect.randomEffect(), 3,
                (this.posX + this.getImage().getWidth()) / WorldMap.TILE_SIZE,
                (this.posY + this.getImage().getHeight()) / WorldMap.TILE_SIZE);
        ((MyWorld)(this.getWorld())).getFixedObjects().add(item);
    }
}
/**
/**
 * Ventana que se muestra cuando el jugador gana.
 *
 * @author Mau
 */
public class WinScreen extends World
{

    /**
     * Constructor for objects of class WinScreen.
     *
     */
    public WinScreen()
    {
        // Create a new world with 600x400 cells with a cell size of 1x1 pixels.
        super(600, 400, 1);
        prepare();
    }
```

```
    /**
     * Lee las opciones del jugador.
     * 'Y' para jugar de nuevo.
     * 'N' para salir del juego.
     */
    public void act() {
        if (Greenfoot.isKeyDown("Y")) {
            Level1Intro intro = new Level1Intro();
            Greenfoot.delay(10);
            Greenfoot.setWorld(intro);
        }
        if (Greenfoot.isKeyDown("N")) {
            System.exit(0);
        }
    }

    /**
     * Hace el resto de cosas necesarias para inicializar la clase.
     */
    private void prepare() {
        setBackground("Screens/win-screen.jpeg");
        showText("Tu tiempo: " + Timer.getTime() + " segundos", getWidth() / 2, 3 * getHeight()
/ 4);
    }
}
```

Función que cambia de nivel.

```
 *
 * @author Gaspar
 */
public class WindowSwitcher
{
    public static void nextLevel(int currentLevel) {
        switch (currentLevel) {
            case 1:
            {
                showLevel(2);
                break;
            }
            case 2:
            {
                showLevel(3);
                break;
            }
            case 3:
            {
                win();
```

```
            }
        }
    }

    public static void showLevel(int level) {
        switch (level) {
            case 1:
            {
                Level1Intro intro = new Level1Intro();
                Greenfoot.setWorld(intro);
                break;
            }
            case 2:
            {
                Level2Intro intro = new Level2Intro();
                Greenfoot.setWorld(intro);
                break;
            }
            case 3:
            {
                Level3Intro intro = new Level3Intro();
                Greenfoot.setWorld(intro);
                break;
            }
        }
    }

    /**
     * Muestra la ventana de victoria.
     */
    public static void win() {
        WinScreen winScreen = new WinScreen();
        Greenfoot.setWorld(winScreen);
    }

    /**
     * Muestra la ventana de derrota.
     */
    public static void lose() {
        GameOverScreen gameOver = new GameOverScreen();
        Greenfoot.setWorld(gameOver);
    }


}

/**
 * Representación del mapa del juego.
```

```
 * El mapa es una matriz formada por cuadrados llamados Tiles.
 * Al instanciar esta clase hay que
 *
 * @author Gaspar
 */
public class WorldMap {
    // Dimensiones de cada Tile en pixeles
    public static final int TILE_SIZE = 16;
    // Dimensiones del mapa en Tiles
    public static final int MAP_WIDTH = 164;
    public static final int MAP_HEIGHT = 40;

    private final Tile[][] mapTiles;
    private final Tile.TileType[] tileTypes = Tile.TileType.values();
    // Tipos que puede haber
    private final String[] tilePaths = {"bush", "dirt", "grass", "mud", "rock", "tree", "water",
"pavement", "wall", "roof"};
    // Total de imágenes de cada tipo
    private final int[] tileAmounts = {1, 1, 11, 1, 1, 1, 1, 1, 1, 1};
    // Porcentaje total de apariciones de cada tipo. Deben sumar 100
    // Si no suman 100 no dibuja el fondo.
    // Estos datos son utilizados en generateCountryMap()
    private final int[] percentages = {20, 2, 70, 2, 2, 2, 2, 0, 0, 0};

    private final Random random;

    /**
     * Constructor para el mapa
     */
    public WorldMap(long seed, int level) {
        this.random = new Random(seed);
        this.mapTiles = new Tile[MAP_WIDTH][MAP_HEIGHT];

        prepare(level);
    }

    /**
     * Llena el mapa de acuerdo al nivel
     *
     * @param level Nivel del juego.
     */
    private void prepare(int level) {
        if (level == 1) {
            generateCityMap();
        } else if (level == 2 || level == 3) {
            try {
                generateCountryMap();
            } catch (WrongGenerationPercentagesException e) {
```

```
                throw new RuntimeException(e);
            }
        } else {
            throw new RuntimeException("El nivel debe ser de 1 a 3");
        }
    }

    public Tile[][] getMapTiles() {
        return this.mapTiles;
    }

    /**
     * Genera un fondo aleatorio para la ciudad
     *
     * @author Gaspar, Mau
     */
    private void generateCityMap() {
        int blockWidth = 12;
        int blockHeight = 8;
        int streetWidth = 4;

        for (int x = 0; x < MAP_WIDTH; x++) {
            for (int y = 0; y < MAP_HEIGHT; y++) {
                int imageType;

                if (x % (streetWidth + blockWidth) < streetWidth || y % (streetWidth + blockHeight)
< streetWidth) {
                    imageType = Tile.TileType.PAVEMENT.ordinal();
                } else if (y % (streetWidth + blockHeight) > streetWidth + blockHeight - 2) {
                    imageType = Tile.TileType.WALL.ordinal();
                } else {
                    imageType = Tile.TileType.WALL.ordinal();
                }

                int imageIndex = random.nextInt(tileAmounts[imageType]);
                int i = x % (streetWidth + blockWidth) - streetWidth + 1;
                int j = y % (streetWidth + blockHeight) - streetWidth + 1;
                GreenfootImage image;
                if (i >= 1 && j >= 1) {
                    image = new GreenfootImage("MapTiles/House/house" + "-" + j + "," + i +
".png");
                } else {
                    image = new GreenfootImage("MapTiles/" + tilePaths[imageType] + "-" +
imageIndex + ".png");
                }

                mapTiles[x][y] = new Tile(image, Tile.TileType.values()[imageType]);
            }
```

```
        }
    }

    /**
     * Genera un fondo aleatorio para el campo
     *
     * @author Gaspar
     */
    private void generateCountryMap() throws WrongGenerationPercentagesException {
        // Calcular la sumatoria de los porcentajes
        for (int i = 0; i < percentages.length - 1; i++)
            percentages[i + 1] += percentages[i];

        // Los porcentajes deben sumar 100
        // JVM deshabilita los assert y no hay manera de habilitarlos en greenfoot !&#*"#
        // Entonces en lugar de detener el programa simplemente no dibujará ningún fondo
        // Si no está dibujando el fondo deberías checar cuánto suman los porcentajes
        // assert (percentages[percentages.length -1] == 100);
        if (percentages[percentages.length - 1] != 100)
            throw new WrongGenerationPercentagesException("Percentages don't add 100.
Please check WorldMap.percentages");

        // Llenar el mapa con valores aleatorios
        for (int x = 0; x < MAP_WIDTH; x++) {
            for (int y = 0; y < MAP_HEIGHT; y++) {
                int imageType = getRandomType();
                int imageIndex = random.nextInt(tileAmounts[imageType]);

                GreenfootImage image = new GreenfootImage("MapTiles/" + tilePaths[imageType]
+ "-" + imageIndex + ".png");

                mapTiles[x][y] = new Tile(image, tileTypes[imageType]);
            }
        }

        // Poner un cuadro de pasto 4 x 4 en el centro donde el jugador va a aparecer
        int startX = MyWorld.WORLD_WIDTH / (TILE_SIZE * 2) - 2;
        int startY = MyWorld.WORLD_HEIGHT / (TILE_SIZE * 2) - 2;
        int endX = MyWorld.WORLD_WIDTH / (TILE_SIZE * 2) + 2;
        int endY = MyWorld.WORLD_HEIGHT / (TILE_SIZE * 2) + 2;
        for (int x = startX; x < endX; x++) {
            for (int y = startY; y < endY + 2; y++) {
                int imageType = Tile.TileType.GRASS.ordinal();
                int imageIndex = random.nextInt(tileAmounts[imageType]);

                GreenfootImage image = new GreenfootImage("MapTiles/" + tilePaths[imageType]
+ "-" + imageIndex + ".png");
```

```
                mapTiles[x][y] = new Tile(image, tileTypes[imageType]);
            }
        }
    }

    /**
     * Dibuja el mapa en el fondo dependiendo de la posición del jugador
     *
     * @author Gaspar
     */
    public void drawMap(MyWorld world) {
        // Quita todos los anteriores porque los va a volver a dibujar

        // Guarda todos los objetos del tipo Tile en una lista
        List<Tile> tiles = world.getObjects(Tile.class);
        // Quita todos los tiles del mapa
        world.removeObjects(tiles);
        // Calcula el índice de la matriz a partir de donde comenzará a dibujar los tiles
        // La posición del jugador está dada en píxeles, entonces debemos dividir entre el
tamaño
        // en píxeles de cada Tile para saber la posición de la matriz.

        // Esto es, si por ejemplo el jugador se encuentra en (20, 40)
        // Dividimos entre el tamaño de cada Tile (16) j queda que deberemos comenzar a
dibujar
        // a partir del Tile (1, 2)
        int startTileX = world.getPlayer().getPosX() / TILE_SIZE;
        int startTileY = world.getPlayer().getPosY() / TILE_SIZE;

        // Para calcular en qué tile de la matriz vamos a terminar de dibujar
        // simplemente calculamos cuántos tiles caben a lo largo y lo ancho de la ventana
        // obteniendo su tamaño en píxeles y dividiendo entre el tamaño de cada tile.
        int endTileX = startTileX + world.getWidth() / TILE_SIZE;
        int endTileY = startTileY + world.getHeight() / TILE_SIZE;

        // Volviendo a startTileX j startTileY. Podemos mirar que en el ejemplo sobran 4 j 8
píxeles
        // Debemos tomarlos en cuenta porque si no se va mirar como si fuera saltando cada
16 píxeles
        // en lugar de como si se fuera recorriendo
        int offSetX = world.getPlayer().getPosX() % TILE_SIZE;
        int offSetY = world.getPlayer().getPosY() % TILE_SIZE;

        // Variables donde vamos a guardar dónde debe ir cada tile
        // Su posición debe ir guardada en píxeles.
        // Sus posiciones las vamos a ir moviendo junto a 'i', 'j'
        // Comenzamos con TILE_SIZE - 8 porque si comenzamos con 0 se tapa la mitad del
tile
```

```
        int tilePosX = TILE_SIZE / 2;
        int tilePosY;

        // Iteramos por la matriz desde startTileX hasta endTileX (Renglones)
        for (int x = startTileX; x < endTileX; tilePosX += TILE_SIZE, x++) {
            tilePosY = TILE_SIZE / 2;
            // Iteramos por la matriz desde startTileY hasta endTileY (Columnas)
            for (int y = startTileY; y < endTileY; tilePosY += TILE_SIZE, y++) {
                // Checamos que 'i', 'j' se encuentren dentro de los límites de la matriz
                if (x < 0 || x >= MAP_WIDTH || y < 0 || y >= MAP_HEIGHT)
                    continue;

                // Agregamos al tile en la posición correspondiente
                // Aquí se puede observar cómo le restamos el residuo que calculamos
anteriormente
                // (No me preguntes por qué hay que restarlo en lugar de sumarlo)
                world.addObject(mapTiles[x][y], tilePosX - offSetX, tilePosY - offSetY);
            }
        }

    }

    /**
     * Regresa un tipo de piso de acuerdo a los porcentajes de probabilidad dados
     *
     * @return int que indica qué tipo de piso deberá ser usado.
     */
    private int getRandomType() {
        int randomPercentage = random.nextInt(100);

        for (int i = 0; i < percentages.length; i++) {
            if (randomPercentage <= percentages[i]) {
                return i;
            }
        }

        // Should not get here
        return -1;
    }

    /**
     * Esta excepción se lanza cuando los porcentajes de cuántos tiles deberían ser
mostrados
     * están mal establecidos j no suman 100.
     */
    static class WrongGenerationPercentagesException extends Exception {
        public WrongGenerationPercentagesException(String message) {
            super(message);
```

```
    }
  }
}
```

Reporte

Dentro del desarrollo lo primero que tuvimos que hacer fue juntarnos para platicar acerca del tipo de juego que queríamos hacer y de ideas generales del mismo. Después de concretar más el juego (ya decididos a que iba ser de buscar al perro) se hizo la primer versión del GDD y del diagrama UML, donde se incluyó el personaje principal, el perro, los rateros y los ítems a grosso modo. Posteriormente, tuvimos que crear un repositorio en Github para poder subir lo que tenía que hacer cada quien del proyecto.

El diagrama UML fue cambiado considerablemente ya que al final surgieron un total de 29 clases dentro del juego, debido a que se tienen las clases por pantalla, clases de los cuadritos del mapa, entre otros, de manera que la funcionalidad del juego resultó ser más compleja de lo esperado.

Más adelante, se repartió mejor qué iba a hacer cada quien, de manera que al tener las clases ya diseñadas era más práctico pasarnos ahora sí directo al código. Solo que a decir verdad. Al inicio era un poco aterrador atacar el proyecto.

De primeras se esperaba que los niveles se dibujaran manualmente, es decir, se hacía una sola vez y se establecía en el fondo, pero se encontró que con base en la posición del jugador, se calcula qué parte del mapa se va a dibujar respecto a la visibilidad del jugador y, para el nivel del bosque, se dio prioridad de aparición a ciertos objetos para que sea acorde a la ambientación.

Otro punto a destacar es que de primeras el movimiento del jugador fue fácil de programar, pero surgieron problemas a partir de querer establecer las colisiones con el ambiente, dado que había que tomar en cuenta del tamaño de la imagen del jugador y de la de los cuadros.
También se tuvieron ciertas dificultades con el contador debido a que primero se intentó con una clase directamente de Greenfoot, pero al no funcionar como se esperaba optamos mejor por crear una desde cero, aunado a esto hubo que mostrarlo como texto en lugar de imagen al ser más simple de esta manera.

Junto a esto, el scrolling no mostró problemas al ser algo relativamente fácil, ya que la manera en que planteamos desde un inicio la creación del mapa, se prestó para esto del scrolling. Porque recordando justo como funciona nuestro mapa, nosotros al “mover al jugador”, no lo estamos moviendo a este en sí, movemos el mapa, y las partes que no estaban a pantalla, se generan automáticamente con nuestro algoritmo. De modo que cada que iniciemos una partida, nuestro mapa será diferente, lo que lo hace más divertido al no ser repetitivo, y que terminó justo facilitando esto del scrolling.

Por otro lado, la navegación de las pantallas de nivel y de las instrucciones , esto a consecuencia de que Greenfoot toma en el método que lee las teclas la última presionada, sin tomar en cuenta cuántas veces se hizo, motivo por el cual hubo que poner un retraso

para que se desplace correctamente a través de las pantallas en lugar de recorrer todas de golpe.

Algo interesante del proyecto es que para lograr el movimiento del perro notamos que podría ser interesante que este busque el mejor camino para llegar al otro lado del mapa, esto por medio de un BFS (Breadth Search o búsqueda por amplitud).

Conclusión

Podemos decir que en cuestión de aprendizaje obtenido, fue una gran experiencia, ya que pudimos ver en la práctica muchos de todos los conceptos vistos a lo largo del semestre, que aunque sí habíamos visto en ejercicios, la realidad es que no se compara con ya aplicarlos a un proyecto de gran escala (para lo que estamos acostumbrados).

Además, nuestros aprendizajes no sólo consistieron en quedar más claro conceptos relacionados con la programación orientada a objetos, sino que también aprendimos mucho sobre el uso de git para la creación de proyectos en equipo.

Así que podemos concluir que, a pesar de ser una travesía con muchos tropiezos (en su mayoría relacionados con el uso de la plataforma de greenfoot, el uso de git, y la coordinación de trabajar en equipo), al haber aprendido todo lo mencionado anteriormente, fue un viaje balanceado en sus momentos buenos y malos, además de ser única, y de cierta manera divertida al final, ya que después de tanto trabajo, podemos ver materializado nuestro esfuerzo en nuestro primer videojuego, así como ser rica en experiencia en el diseño de clases para trabajos más grandes.

Gameplay del juego:

https://www.youtube.com/watch?v=YL4Qs6j9BvQ