**PORTAL 2D: Informe.**

**Portales**

A la hora de implementar los portales se tuvieron en cuenta varios factores: cómo disparar los portales, cómo se diferencian los portales, cómo atravesar los portales y cómo crear los portales sobre superficies específicas. El método de creación de portales es el siguiente:

* El jugador hace click en una dirección y el controller le da esa dirección al arma de portales (Portal gun) de la entidad Player. La portal gun utiliza una especie de línea llamada Raycast que empieza en la posicion del Player, con direccion a la posicion del mouse para saber a qué parte del mapa está apuntando el jugador.
* Cuando el raycast colisiona con una entidad, se verifica si la misma es una superficie en la que se puedan crear portales (de ahora en más referidas como superficie portal-able). Si no es así, no se crea un portal.

Otras consideraciones en la implementación de los portales:

* Los portales se diferencian entre sí a través de una variable que representa su color.
* Por regla general, solo portales de distinto color pueden estar conectados entre sí.
* Cuando una entidad toca un portal se teletransporta a la ubicación del portal opuesto instantáneamente, conservando su velocidad y cambiando la dirección de su movimiento según la normal de la superficie donde se encuentra dicho portal.

**Obstáculos**

Hay dos tipos de obstáculos en el juego, aquellos que no permiten el paso del jugador de manera no letal, y aquellos que representan un riesgo, en este caso, la muerte.

*Obstáculos no letales:*

Existen puertas que bloquean el paso a ciertas áreas del juego y son abiertas por interruptores. Las puertas son sólidas y solo se puede crear portales a través de ellas si están abiertas.

*Obstáculos letales:*

Los obstáculos letales funcionan matando al jugador de alguna forma u otra. Al morir el jugador, el nivel se reinicia automáticamente. Los obstáculos letales son los siguientes:

* Torreta: Las torretas son entidades estacionarias que disparan proyectiles. Una vez que el jugador entra en su rango de tiro, la torreta dispara en dirección del jugador. Si un proyectil toca al jugador, éste muere.
* Acido: El ácido es una entidad, también estacionaria, que destruye cualquier otra entidad no estática con la que entra en contacto.

**Patrones implementados:** GameState, Observer y MVC.

* El patrón GameState se utilizó para manejar el flujo de estados del programa a partir de un stack de estados, pusheando por ejemplo el estado pausa y haciendo su pop correspondiente cuando se reanuda el juego.
* El patrón Observer se usó para notificar cuando se agrega y se remueve una entidad en el nivel. Específicamente, para agregar y remover views en el estado PlayState.
* Para respetar el MVC, se decidió que cada estado del juego (pausa, play, menu, etc) sea un controlador, y tenga una referencia a su view.

**Decisiones de diseño**

* Compromiso realizado al diseñar el sistema de interacciones:

Para explicar este compromiso, primero se tiene que explicar cómo funciona, a rasgos generales, Box2D, la librería que simula el mundo físico de libGDX. El mundo de Box2D (de ahora en más World) es el objeto que contiene a los cuerpos físicos y el que simula la física en sí. Un cuerpo (Body) está compuesto por una o más Fixtures (la forma más fácil de explicar que es una Fixture es decir que es la representación corporal del Body en el World. Cuando se realiza una colisión entre Bodies, en realidad es una colisión entre sus Fixtures).

Cuando dos Fixtures colisionan, World crea una nueva instancia de la clase Contact, la cual contiene referencias a las 2 fixtures que colisionaron, además de otros datos relevantes a la colisión. Estas colisiones pueden ser utilizadas por medio de una interfaz provista por Box2D, ContactListener. Uno de los problemas por el cual se tuvo que llegar al diseño final es el siguiente: ContactListener sólo procesa los contactos cuando empiezan y cuando terminan. Es por eso que la clase Entity posee métodos BeginInteraction y EndInteraction.

Una vez solucionado el problema de los contactos siendo procesados solo dos veces, debido a la forma en la que fue diseñada la jerarquía de entidades, se encontró un problema de Casting, el cual se puede explicar de la siguiente manera genérica:

*Sean las clases A y B, tal que B hereda de A. A está diseñada de la siguiente forma:*

*public class A{*

*public void método(A sujeto){*

*System.out.println(“Tomado como sujeto A”);*

*}*

*public void método(B sujeto){*

*System.out.println(“Tomado como sujeto B”);*

*}*

*}*

*Entonces, creamos un nuevo main y lo definimos de la siguiente forma:*

*public static void main(String[] args){*

*A var1 = new A()*

*A var2 = newB()*

*var1.método(var2)*

*}*

*Y ejecutamos dicho main. veremos que en la consola se ve “Tomado como sujeto A”..*

En el caso del proyecto, esto se produce al querer hacer interacciones, dado que el ContactListener toma a todas las instancias como Entity, en vez de castearlas a su clase correspondiente. Como resultado directo, al llamar al método BeginInteraction o EndInteraction, se tomaría al parámetro como una Entity y no como, digamos, una instancia de Box o  de Player.

La solución a este problema fue implementar los métodos BeginInteraction(Entity)/EndInteraction(Entity) en todas las subclases Entity. Dichos métodos están implementados de tal manera que al ser llamados, la instancia llama al método correspondiente de la entidad con la que interactúa, pero pasándose a sí misma como parámetro, logrando que se la tome como instancia de la clase correspondiente sin tener que realizar ningún casteo explícito.

* Compromiso en el diseño de los botones:

El diseño de los botones está directamente relacionado con el compromiso de diseño anterior. Debido a que las interacciones sólo se procesan al inicio y al final, se necesitaba una forma de llevar cuenta de la cantidad de entidades que estaban presionando un botón en un momento dado. La solución más simple (y a nuestro parecer, la mejor) fue llevar un contador de “interacciones”. Al empezar un nuevo contacto, se suma 1 a dicho contador, y al terminar el mismo contacto, se resta 1.

* Compromiso en el agregado de Queues al level:

Se tuvieron que agregar queues de remoción a la clase Level debido a un problema con Box2D. El problema es que World no permite ni crear ni remover bodies mientras se está actualizando, pero las interacciones también se procesan en dicha actualización, lo que llevó a buscar una solución para poder borrar entidades al detectar un contacto entre, por ejemplo, el ácido y una caja. La solución fue crear una lista de entidades que se tienen que eliminar al terminar la actualización de World.

* Compromiso en la separación entre Bullets y Entities:

Debido a que la forma en que fueron implementadas, las torres crean balas si detectan, al momento de actualizarse, que pueden disparar una bala. El problema que genera esto es que, por la forma en que se actualizan todas las entidades (un for each que va entidad por entidad, llamando al método update()) no se pueden crear balas, ya que al ser un loop for each, Java lanza ConcurrentModificationException, porque se agregan elementos a la lista de entities mientras se está recorriendo dicha lista. La solución fue separar a las balas de las entidades en un set aparte. En relación al uso de sets por sobre el uso de listas para representar a la colección de entidades dentro del Level, se eligió esta forma ya que evita más de una referencia a una misma entidad dentro de la colección, evitando así también actualizaciones múltiples de la misma  entidad en un solo frame.