Hitman 2D: Informe Final

Para comenzar, es necesario recordar brevemente cuál era la funcionalidad que se esperaba lograr en este trabajo. Dada la naturaleza de *Hitman*, el elemento del sigilo debe ser una parte central de un juego que intente aproximarse a la esencia de esta saga, por lo que, sin lugar a dudas, en este trabajo se ha puesto especial énfasis en este aspecto, fundamentalmente en dos áreas: el manejo de mensajes y el comportamiento de los NPCs (jugadores no humanos).

En el sistema de mensajería que hemos implementado, existen *mensajes* (Ver interface Message) que los diferentes tipos de Character (Ver clases: NPC, Player, Goon, Civil) emiten en las circunstancias apropiadas. Por ejemplo, al caminar o correr el personaje controlado por el jugador emite sonidos de diferente intensidad (alta si corre, baja si camina), los cuales deben ser escuchados por los NPCs. Además, informa al mundo de su posición en cada iteración (Ver clase Vision). Por último, cuando el personaje o los NPCs agresivos disparan, las Balas (Ver clase Bullet) también se comportan como mensajes.

Todos estos mensajes deben, de alguna forma, llegar a su objetivo. Para esto, hemos diseñado un sistema de diferentes manejadores de mensajes (Ver clase MessageManager), los cuales mantienen cada uno una lista de todos quienes pueden recibir determinado tipo de mensaje (Ver interface Listener, BulletListener, VisionListener, NoiseListener), además de todos los mensajes que fueron recibidos en la iteración actual. Cada uno de estos manejadores (uno por cada tipo de mensaje: ver NoiseManager, VisionManager, BulletManager) es una clase singleton del tipo apropiado (según tipo de mensaje y tipo de Listener: ver implementación) la cual, al ser llamada con el método update(), itera sobre los mensajes y los escuchadores, siendo los mensajes los que deciden si deben notificar a un escuchador (método notify(Listener l)). Entendemos que el hecho de que exista una clase singleton por cada tipo de mensaje no es lo más adecuado, pero frente a las distintas alternativas que estuvimos debatiendo, la limitada cantidad de tipos de mensajes y el uso que le hemos dado a los tipos genéricos de Java para la definición del MessageManager, ésta nos ha resultado la alternativa más prolija y eficaz de implementar.

Por otro lado, al llegarle un mensaje a un NPC, este debe saber qué hacer frente a dicho mensaje. Para esto, hemos implementado una máquina de estados utilizando los enums de Java, tomando la idea de esta guía de Libgdx: https://github.com/libgdx/gdx-ai/wiki/State-Machine . La idea principal es la siguiente: cada NPC tiene un estado (ver enum NPCState), una máquina de estados (Ver clase NPCStateMachine) y una serie de comportamientos (ver paquete Behaviour). Un NPC empieza con un tipo de comportamiento y, en cada iteración, solicita a la NPCStateMachine que actualice su estado, pasándole su contexto (ver clase Context), es decir, todo lo que recibió en la iteración pasada. Los diferentes tipos de estado saltan de uno a otro en función de este contexto recibido, mediante un método updateState implementado en cada uno de los tipos del enum. Los diferentes estados conllevan diferentes comportamientos: un NPC agresivo disparará al jugador si lo ve, un Civil huirá de él.

Con respecto a otros aspectos del backend: mapa, pathfinding, serialización:

- Mapa: hemos utilizado el TiledMap de libgdx, una herramienta que toma un mapa en un formato .tmx y lo interpreta. Sobre este TiledMap hemos creado nuestro mapa lógico, el cual contiene diferentes funcionalidades para verificar colisiones, casillas habilitadas, etc.
- Pathfinding: implementamos dos tipos de pathfinders: LinearPathFinder y AStarPathFinder. El primero se usará únicamente cuando la posición final deseada no se encuentra bloqueada por el mapa (para ello utilizará la funcionalidad del mapa). El segundo se utilizará en el caso de que no exista un camino lineal entre la posición del NPC y el destino. Utiliza una implementación del algoritmo A*.
- Serialización: En cuanto a este apartado, nos ha parecido que la única circunstancia en la cual ameritaba usar algún aspecto relacionado a la serialización era en el cargado de los niveles. Para proveer de flexibilidad a la hora de crear un nivel, y evitando construir toda una aplicación con tal fin, hemos considerado apropiado que la información requerida para inicializar un nivel se guarde en un archivo XML, el cual sería parseado y convertido a una clase. Para esto utilizamos el entorno JAXB (Java Architecture for XML Binding) , el cual permite, mediante el uso de anotaciones, definir una relación entre clases creadas y archivos XML que pueden ser parseados.

Con respecto a la interacción modelo-frontend:

Hemos implementado clases controladoras para cada uno de los objetos que se dibujan en pantalla (a excepción del TiledMap). Cada uno de estos controladores posee una referencia al modelo y a su correspondiente vista. En cada iteración, actualizará al modelo y, luego, enviará a la vista la información requerida para su correcta representación en el mapa. La clase GameManager, que es la que sirve de nexo principal entre los dos apartados, crea ambos y se encarga de actualizarlos.