Paradigmas Proyecto Final: Taxi Driver

Hecho por Isabel Morell y Sofía Negueruela

Enlace al repositorio de github: https://github.com/202215473/PyTP_ProyectoFinal

Introducción

Como proyecto final hemos decidido desarrollar el proyecto propuesto: Taxi Driver. El juego va a consistir en un taxista que debe llevar a los pasajeros lo más rápido posible a su destino, pero siempre yendo seguros y cómodos. Sin embargo, el taxi se enfrentará a diferentes obstáculos como la persecución de policías en el modo vs. Policías de IA y la presencia de radares, vallas de obras o *debufs* que reducirán su vida y las propinas que reciban de los pasajeros. De la misma forma, si se intenta salir del mapa también perderá puntuación y ganará menos dinero.

El objetivo del taxista será llevar al pasajero a su destino con la máxima comodidad posible para obtener el máximo dinero y poder comprar diferentes taxis y hacerles mejoras.

Instrucciones

El jugador comenzará seleccionado el modo de juego que desee. La diferencia entre ambos modos es que en el segundo (vs. Policías de IA), si los policías con su radar detectan que el taxi va más rápido de lo permitido, entonces comenzarán a perseguir al jugador y, si le pillan, acabará el juego.

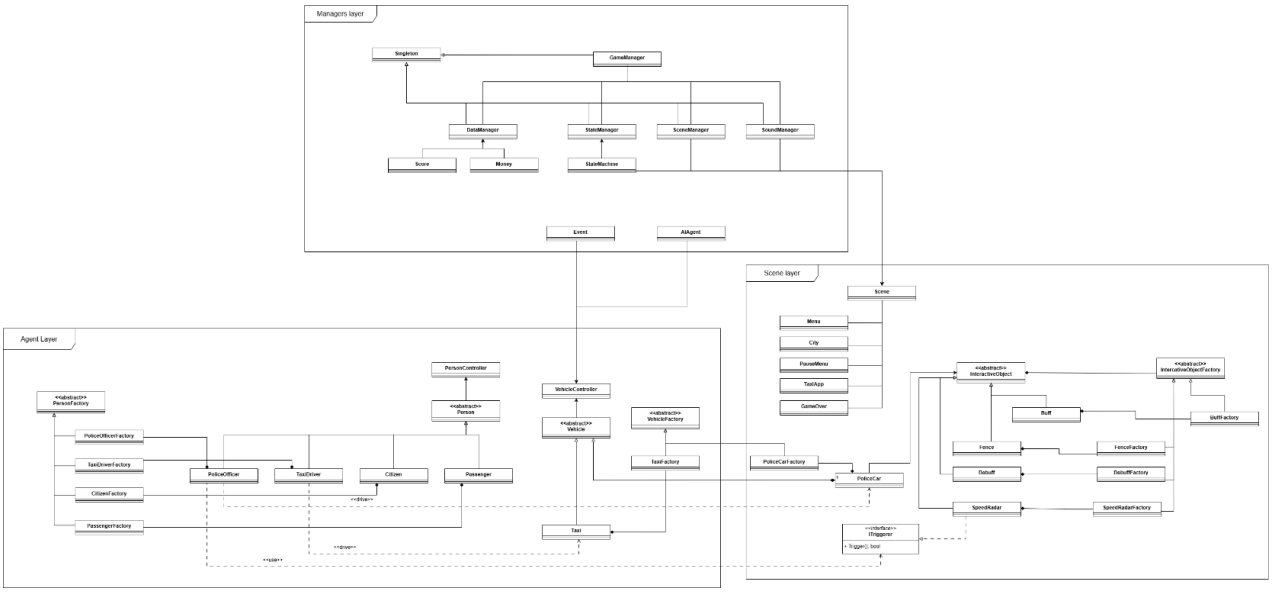
Una vez seleccionado el modo, el jugador verá arriba a su izquierda el mapa de la ciudad con su posición en blanco, las posiciones de los clientes en rosa y, cuando haya recogido a un cliente, aparecerá el destino del cliente en amarillo. Para recoger a un cliente debe acercarse lo suficiente a él y a una velocidad menor de 15m/s (54 km/h). Estas condiciones también se deben cumplir a la hora de dejar al jugador.

En la pantalla de juego también se puede observar la cantidad de dinero que ha acumulado el jugador arriba a la derecha y la barra de vida en medio. Ésta irá bajando a medida que el taxi se choque con los diferentes obstáculos y se mostrará tanto en el tamaño de la barra como en su color. Una vez el taxi haya perdido toda su vida, finalizará la partida.

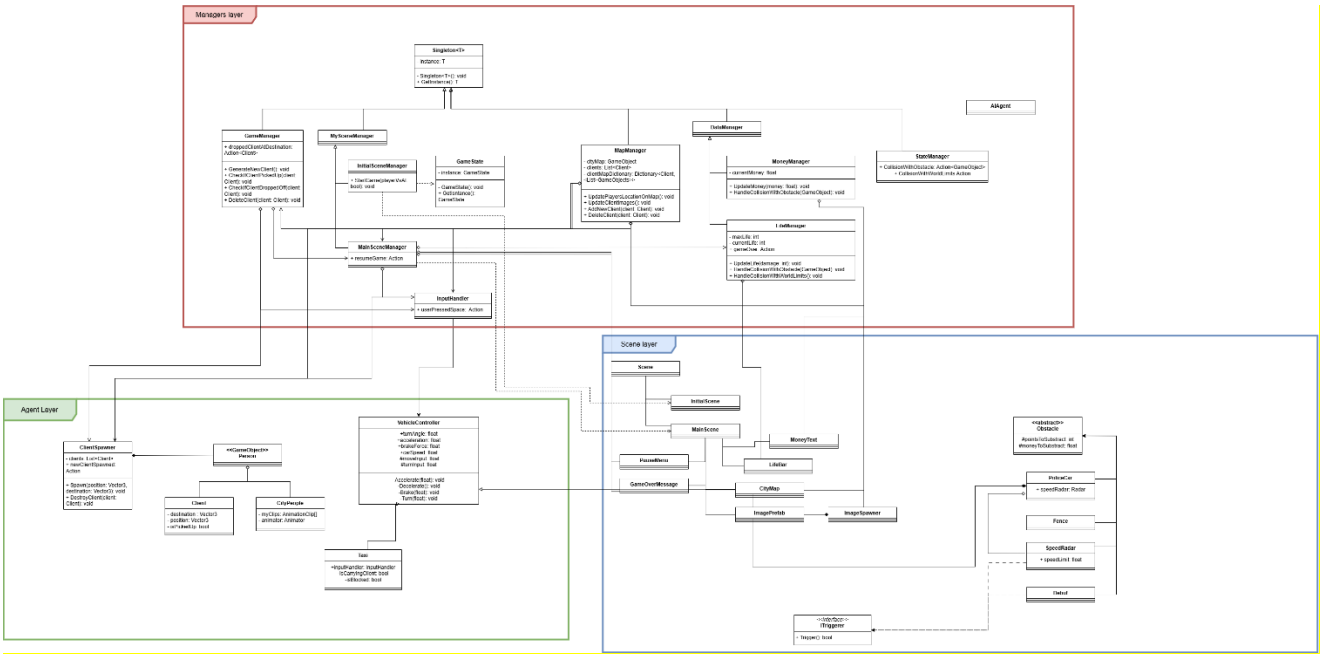
Para pausar el juego, el usuario debe pulsar la tecla espaciadora. Al hacer esto el juego se parará por completo por lo que el taxi no podrá moverse y no aparecerán más clientes. Si el jugador desea terminar la partida, puede pulsar el botón de Quit que le llevará de vuelta al menú inicial. Si desea continuar la partida, debe pulsar el botón de Resume y continuará el juego tal y como lo pausó. Por lo tanto, si cuando se paró el juego el taxi estaba en movimiento entonces, al pulsar el botón de Resume, el taxi seguirá moviéndose. Los clientes irán apareciendo cada vez más rápido a medida que el usuario vaya dejando a clientes en su destino.

Arquitectura del juego

Al comienzo del proyecto, diseñamos el siguiente diagrama UML que mostraba la arquitectura que iba a seguir el juego.



A continuación, mostramos el diagrama UML de clases que representa el proyecto finalizado:



Como se puede observar, la arquitectura está dividida en tres capas en ambas versiones: la capa de agentes, la capa de escena y la capa de más bajo nivel, la de los managers.

Los patrones principales creacionales que hemos utilizado son los patrones de Singleton en la creación de los managers, de Factoría en la creación de diferentes GameObjects a través de sus respectivos spawners y el de Bridge en la creación del jugador principal.

De patrones estructurales hemos utilizado el patrón Decorador para darle la funcionalidad extra al coche de policía en el segundo modo de juego y de patrones de comportamiento hemos aplicado principalmente el patrón Observer a través de la publicación y subscripción a eventos para comunicar las diferentes partes del juego. Además, también hemos aplicado el patrón State para determinar si el jugador está colisionando contra un objeto o uno de los bordes del mundo.

Capa de managers

La capa de Managers es la capa encargada de gestionar los elementos del juego y hacer que funcione correctamente.

En la versión inicial teníamos un GameManager, un DataManager que controlaba la puntuación y el dinero, un SceneManager, un StateManager y un SoundManager. Todos estos managers iban a heredar de una clase Singleton pues solo queremos una instancia de cada manager para que no haya varios objetos que puedan realizar las tareas de gestión del juego.

Finalmente, hemos mantenido en esta capa una arquitectura muy similar a la inicial. El patrón de creación que hemos aplicado con los managers ha sido el patrón Singleton y, para ello, hemos creado una clase Singleton de la que heredan todos los managers del juego.

El GameManager se encarga de la lógica principal del juego al ser el encargado de la creación de los clientes, tarea que hace a través del clientSpawner; y el encargado de comprobar si el taxi ha recogido a un cliente o si ha dejado en su destino al cliente que llevaba. Además, una vez detecta que el taxi ha dejado al cliente, lanza un evento al que están suscritos el MapManager y el MoneyManager para que actualicen sus valores y elimina al cliente a través del clientSpawner.

Para controlar los canvas de las escenas y los cambios entre escenas tenemos las clases InitialSceneManager y MainSceneManager que heredan de MySceneManager. El primero de los dos se encarga de controlar el menú inicial e indicar al GameState el modo en el que se va a jugar el juego. El GameState implementa el patrón Singleton por lo que no se destruirá al cambiar las escenas. El MainSceneManager controla tanto el menú de pausa así como la aparición del mensaje de game over cuando termina el juego. Para determinar cuando el juego ha terminado, está suscrito al evento de gameOver que lanza LifeManager cuando el jugador pierde toda su vida. Además, el MainSceneManager también está suscrito al InputHandler para saber cuando debe mostrar el menú de pausa y al clientSpawner para mostrar un mensaje cuando aparezca un nuevo cliente. El MainSceneManager lanza un evento de resumeGame para avisar de cuando debe continuar el juego.

El MapManager es el encargado de actualizar las posiciones de los clientes, sus destinos y del jugador en tiempo real. Este manager no lo habíamos planteado en la primera versión, pero cuando fuimos a añadir su lógica en el MainSceneManager nos dimos cuenta de que para mantener el principio de única responsabilidad era mejor crearlo en una clase distinta.

Para guardar y actualizar los diferentes datos del juego hemos implementado dos managers el MoneyManager y el LifeManager que heredan del DataManager. Ambos están suscritos al StateManager para actualizar los valores de los datos en caso de colisión. Por último, tenemos el StateManager que se encarga de comprobar si el jugador ha colisionado contra uno de los obstáculos o uno de los bordes del mundo y lanza el evento correspondiente en cada caso.

La división de tareas nos ha permitido garantizar la claridad y encapsulación del código y la gestión de tareas por elementos externos al juego permite la abstracción y encapsulación del juego.

Capa de agentes

La capa de agentes contiene a las diferentes personas que van a interactuar en el juego. Las acciones de estas estarán controladas por el controlador de personas. Los personajes son los ciudadanos de la ciudad, los pasajeros del taxi, y el taxista que será controlado por el jugador y los policías que conducirán los coches de policía. Realmente, para simplificar el código, tanto los taxistas como los policías se asume que van dentro de los respectivos vehículos, por lo que sólo tendremos Client y CityPeople. Todos serán creados por ClientSpawner, siguiendo el patrón de diseño de la factoría con el objetivo de abstraer la creación de las personas de su funcionalidad y para seguir los principios SOLID.

Esta capa también incluirá el taxi que vaya a conducirse. Todos los vehículos serán creados por la factoría asociada a ese tipo de vehículo. Hemos elegido el patrón de la factoría para separar la creación de los objetos de su funcionalidad para cumplir el principio de única responsabilidad (Single Responsability Principle) y porque puede que se creen más de un taxi o más de un coche de policía. Sin embargo, en la implementación, esta funcionalidad se ha hecho creando GameObjects en Unity directamente, por eso no está incluida en el UML final.

Capa de escena

La capa de escena incluye las diferentes escenas que se mostrarán durante el juego. Estos son elementos de Unity que no tendrán código asociado.

En esta capa también se incluyen los objetos interactivos del juego que son los Obstacles que podrán tener algún tipo de efecto sobre el taxi, interactuando así con él. Cabe mencionar que el coche de policía no hereda de la clase Obstacle pero incluye su funcionalidad mediante el patrón *Decorator* y el Radar será lanzado por el policía. Además, el Radar puede existir aunque no esté controlado por un policía.

Los obstáculos serán creados siguiendo el patrón de la factoría pues consideramos que así mantenemos la abstracción y seguimos los principios SOLID. De nuevo, esto se ha implementado desde Unity sin crear una clase explícitamente.

Historias de usuario

Para comprender qué debe hacer el código exactamente, hemos emplea la técnica “Historias de usuario”. Partimos de las siguientes premisas:

“Como jugador, quiero poder conducir un taxi y llevar a pasajeros rápidamente a su destino para ganar mucho dinero”

El objetivo principal del juego es permitir al jugador (taxista) trasladar a los pasajeros de un punto al otro del mapa (ciudad) lo más rápido posible, haciéndoles sentir seguros y cómodos durante el trayecto y sin perder puntos de vida.

Vamos a descomponer la historia en tareas. Estas engloban el desarrollo del proyecto en distintas fases que ayudan a crearlo de forma ordenada y por partes. Cuando se desarrolla un juego, es importante tener claro las tareas principales, aunque estas se puedan desglosar más adelante.

- Crear una interfaz de usuario para que el jugador pueda conducir el taxi.
- En esa interfaz, añadir elementos de la ciudad (personas, obstáculos…) descritos en la sección anterior, que sean visibles para el taxi.
- Crear funcionalidad de cada uno de dichos elementos y definir cómo interaccionan con el taxi.
- Implementar sistema de puntuación y vida para el jugador, que pueda observar cómo aumenta el dinero que tiene al recibir una propina de un viajero y, además, ver cuántos puntos de vida le quedan.

Los requisitos funcionales del proyecto engloban:

- Un menú principal, que permita iniciar el juego cuando desee el jugador.
- Un sistema de puntuaciones, tanto para la vida del jugador como para el dinero que obtiene.
- Jugadores CPU, es decir, que haya oponentes dentro del juego previamente programados con Inteligencia Artificial.
- El juego debe ofrecer la opción de multijugador en local.

Los no funcionales abarcan:

- Un *loop* de juego cerrado, es decir, que la partida acabe en algún momento (*win state*, *fail state*) y exista la opción de volver a jugar.
- Los componentes se crean y destruyen al inicio y final de cada partida respectivamente.

“Como pasajero, quiero que me lleven a mi destino de forma rápida y segura para llegar cuanto antes”

El objetivo del pasajero es tener un viaje seguro y cómodo, pero también lo más breve posible. Recordamos que en esta ciudad todos van siempre con prisa.

Vamos a descomponer la historia en tareas

- Crear una interfaz que muestre que el pasajero está dentro del taxi.
- Crear funcionalidad de cada uno de dichos elementos y definir cómo interaccionan con el taxi.
- Implementar sistema de satisfacción que sirva para determinar cuánta propina dar al taxista.

Los requisitos funcionales del proyecto engloban:

- El sistema de satisfacción del pasajero.
- Que haya otros ciudadanos que no sean pasajeros.

Los no funcionales abarcan:

- El viaje del pasajero tiene inicio y fin.

“Como policía, quiero que se respeten los límites de velocidad para que la ciudad sea un lugar seguro”

El objetivo del policía es reducir al máximo la posibilidad de accidentes. Para ello, han colocado radares por toda la ciudad, y cada coche de policía tiene un radar también.

Vamos a descomponer la historia en tareas

- Crear una interfaz que muestre los coches de policía y los radares de la ciudad.
- Crear funcionalidad de los radares y de los coches.

Los requisitos funcionales del proyecto engloban:

- El sistema de detenciones. Cuando un vehículo supere el límite de velocidad, será perseguido por la policía hasta ser detenido y llevado a la comisaría.
- Que el coche de policía y los radares seas visibles para otros vehículos.

Los no funcionales abarcan:

- El turno de patrulla del policía tiene inicio y fin.
- El radar solo mide la velocidad cuando está activado (los radares de la ciudad estarán activados siempre, los de los coches de policía estarán activados siempre que el policía esté patrullando.