Desarrollo de Aplicaciones para Dispositivos Móviles

Memoria Práctica 3

Rodrigo Martínez Sánchez
Manuel Pérez Ramil

ÍNDICE

Descripción general

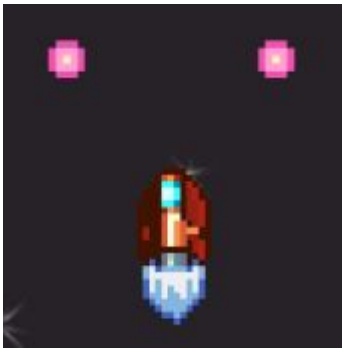
Stanford es un shoot'em up de scroll vertical de temática futurista. El jugador controla una nave espacial y debe destruir un número de naves enemigas para superar el juego. El jugador podrá escoger entre dos naves y dos tipos de proyectiles para completar su misión. Los gráficos son de estética pixel art y la música y el audio son 16 bit.

Requisitos contemplados

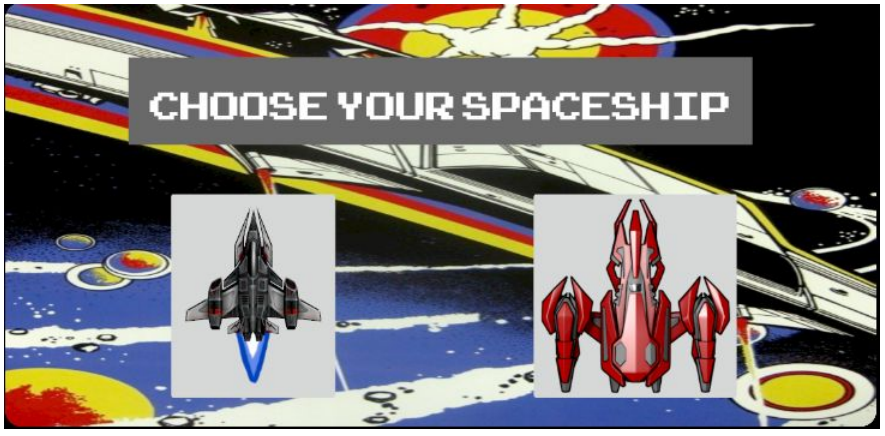
Desafíos plata:

- **Diseño**: hemos buscado nuevos assets para todos los elementos del juego y hemos diseñado la interfaz y elegido la fuente para recordar la estética de los videojuegos antiguos (gráficos pixel art y audio 8-bit).

- **Disparo múltiple**: el jugador tiene dos tipos de armas de disparo automático, que son el disparo recto y el dual. El disparo recto dispara un único proyectil hacia delante que se destruye al impactar con un enemigo. El disparo dual dispara dos proyectiles en diagonal.



- **Tipos de naves**: el jugador puede escoger entre dos tipos de nave, la lanzadera y el destructor. La lanzadera es una nave ligera, que se mueve a gran velocidad pero que será destruída con tan sólo recibir un golpe. El destructor es más lento pero podrá soportar hasta tres disparos antes de destruirse. Estas dos naves se pueden escoger antes de comenzar una partida en el menú de selección de nave.



Desafíos oro:

- **Múltiples enemigos**: existen dos tipos de naves enemigas. Los kamikazes son pequeños y veloces y la forma que tienen de atacar es impactando su nave contra la del jugador. Los bombarderos son más lentos y grandes y disparan proyectiles cada cierto tiempo.

- **Sonido y música**: contiene dos canciones, una para el menú y otra para la partida. Además, las naves reproducen sonido al disparar y al explotar. La música ha sido obtenida de freesound.org, del usuario FoolBoyMedia.

Implementación y justificación

Para la implementación del videojuego Stanford hemos utilizado el motor proporcionado en clase. Los cambios más notables han sido los siguientes:

- **GameEngine.java**: hemos hecho que el GameEngine administre las colisiones. Para ello hemos creado una nueva estructura de datos: Hitbox.
  La clase Hitbox asocia un Sprite con un Rect. Un Rect es una clase que almacena un rectángulo y permite comprobar superposiciones de rectángulos con el método Rect.Intersects(Rect r1, Rect r2).
  Entonces, el GameEngine almacena una Hitbox para cada Sprite, dividiéndolos en 5 tipos diferentes de grupos. Sólo comprobará colisiones entre Hitbox que pertenezcan a diferentes tipos de grupos. Así, almacenamos en un grupo a las naves enemigas y en otro a la nave del jugador (igual para los disparos). De este modo, no comprobará colisiones entre naves enemigas, o entre disparos, ahorrando cálculo innecesario. Cuando se produce una colisión, se le comunica a ambos objetos junto con información de con quién han colisionado. Para comprobar los FPS a los que funciona el juego basta con descomentar la línea 80 de GameFragment.java.
  También hemos migrado la administración de la Pool de Bullets al GameEngine, para facilitar que esta sea compartida entre todos las naves enemigas.
  Finalmente, GameEngine contiene también la soundPool de los sonidos más recurrentes (disparos y explosiones) que puede ser llamada por los GameObjects en el update o cuando se les comunica una colisión.

- **Añadida clase Hitbox.java**: Estructura de datos que contiene un gameObject y un Rect que representa su hitbox. Las instancias de la clase Sprite son las que pueden tener una Hitbox, ya que necesitan implementar el método getRect(), que devuelve el Rect asociado a su sprite.

- **Añadida clase AudioController.java**: controlador de música de fondo, permite reproducir las canciones sin interrupciones entre cambios de fragmentos.

- **Añadida clase ScoreView**: se encarga de pintar por pantalla la información de la puntuación y las vidas restantes.

- **Modificaciones a Sprite**: las modificaciones han sido realizadas para poder utilizar bitmaps con múltiples sprites en un único archivo y para que implemente un método getRect() que devuelve el rectángulo asociado a su sprite para el cálculo de colisiones

- **Modificaciones a controladores de input**: hemos cambiado la función del botón. En nuestro proyecto, el disparo es automático y el botón sirve para comunicar un cambio de arma. Ahora, al pulsar el botón, se marca en la variable weaponSwap de InputController, y SpaceShipPlayer se encargará de notificarle que ha recibido esa información para que pueda volver a comunicarse el evento de cambio de arma. Así nos aseguramos de que la información llega a su destinatario.

- **Modificaciones a Bullet.java**: esta clase ahora contiene información sobre cuale es el grupo de colisión al que pertenece, para únicamente afectar a gameObjects del bando contrario. Además, pueden inicializarse con 3 tipos de movimiento diferentes: recto, diagonal izquierdo y diagonal derecho.

- **Añadida clase Explosion.java**: esta clase es llamada cuando un Bullet impacta con un objetivo y crea una animación de explosión.

- **Creadas clases SpaceShipEnemyMedium y SpaceShipEnemySmall**: estas dos clases son los dos enemigos. La primera es el bombardero, que dispara proyectiles. La segunda es el kamikaze que únicamente se mueve hacia delante.

- **Modificaciones a SpaceShipPlayer**: se pueden crear dos tipos diferentes de nave, la lanzadera y el bombardero, cada uno con su velocidad, sprite y vidas particular. Ambas poseen dos tipos de disparos que se cambian al pulsar el botón de cambio de arma.

Conclusiones

Esta práctica nos ha supuesto un desafío por el hecho de tener que trabajar sobre un código de base que no habíamos implementado nosotros, además de familiarizarnos con algún elemento nuevo de Android Studio. Sin embargo creemos que ha sido interesante ya que en nuestros futuros trabajos lo más probable es que debamos partir también de proyectos ya comenzados.

También nos ha resultado interesante la implementación de las colisiones sin guía, ya que nos han comentado a lo largo de la carrera cómo hacerlas teóricamente, pero a la hora de realizar las prácticas siempre eran en entornos que ya las tenían implementadas. Hemos intentado hacer una implementación eficiente al dividir las colisiones por grupos y comprobar únicamente las que pertenezcan a grupos diferentes.