UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES

FACULTAD DE CIENCIAS EXACTAS

INGENIERÍA DE SISTEMAS

VIDEOJUEGO ORIENTADO A OBJETOS

PROGRAMACIÓN ORIENTADA A OBJETOS

Alumno: Ludmila Baliño.

Alumno: Federico Joaquín Iribarren Susino.

1. RESUMEN

El presente informe forma parte del trabajo final de la cátedra Programación Orientada a Objetos, correspondiente a la carrera Ingeniería de Sistemas de la Facultad de Ciencias Exactas de la Universidad Nacional del Centro de la Provincia de Buenos Aires (UNICEN).

El objetivo del trabajo es desarrollar un programa en C# que modela el comportamiento de un videojuego estilo *Arcade*, utilizando el paradigma de programación orientada a objetos.

El informe está dividido en tres secciones principales. En la sección “Análisis del videojuego” se brinda más detalles e información acerca del programa. En “Diseño e implementación” se abordan cuestiones tales como el diseño de clases y las estructuras de almacenamiento que fueron necesarias utilizar. En “Consideraciones” se hace referencia a las decisiones y aspectos de importancia sobre el código fuente.

Cabe aclarar que se utilizó el motor de videojuegos multiplataforma Unity para el desarrollo del programa. El motor gráfico de Unity utiliza OpenGL y el script se basa en MonoDevelop

2. ANÁLISIS DEL VIDEOJUEGO

El proyecto “Space Riders” se desarrolló con el motor de videojuegos Unity, y es compatible con la plataforma PC. Es del género Arcade, que se refiere a los videojuegos clásicos o que recuerdan a las máquinas del mismo nombre.

A continuación se detallarán las principales características del videojuego:

El videojuego se desarrolla en el espacio exterior. El jugador elige una nave con la que debe enfrentarse a diversos obstáculos para poder sumar puntos. El objetivo principal es sobrevivir el mayor tiempo posible en un escenario con el fin de eliminar enemigos y así obtener un mejor puntaje. El personaje es una nave, el jugador puede elegir diferentes naves para cada partida,las cuales tienen atributos que le ayudan a enfrentar obstáculos. (Daño de disparo, Velocidad de disparo, Vida de la nave, Escudo de la nave). El escenario no tiene fin, un escenario acaba cuando el jugador pierde la partida.

Los enemigos son los obstáculos del juego. Estos son los que le agregan dificultad. Entre ellos se encuentran:

- Asteroides (Causan daño al impactar con la nave del jugador):

- Naves enemigas (Disparan al jugador)

El juego se inicia en un Menú, donde el jugador puede elegir una Nueva Partida o puede visualizar los *Records*, antes de una partida, el jugador podrá elegir una nave para utilizar, comienza la partida y la nave debe enfrentar los obstáculos que aparecen (Naves y asteroides). Además aparecen *Power Ups* y Vida. La vida es un aumento definitivo para la vida del jugador, mientras que los power up tienen una duración limitada. Al final de cada partida, se muestra el puntaje obtenido en la partida y si es un nuevo récord, se le pide el nombre al jugador y se almacena en la tabla de Records.

3. DISEÑO E IMPLEMENTACIÓN

MODELADO

En esta fase del proyecto se buscó resolver la implementación del videojuego, reconociendo que clases se necesitan. A lo largo del trabajo se buscó respetar el paradigma de la Programación Orientada a Objetos en la mayor medida posible.

En primer lugar, cuando se diseñaron los elementos que hacían al juego, se reconoció que las naves y asteroides tienen comportamiento en común, el cual fue modelado en la clase abstracta ElementoJuego, esta solo implementa el método RestarVida(), que es común a las naves y asteroides. Estos elementos junto a los ítems son los encargados de darle el aspecto lúdico al proyecto. Por su parte, Item es una clase abstracta que identifica a aquellos elementos que pueden ser recolectados por el jugador para que este obtenga un beneficio.

Todos los GameObject tiene un método llamado OnTriggerEnter que se encarga de escuchar frame por frame que objetos en la escena entran en contacto con el “oyente”. Este método tiene como parámetro un collider (malla que define los límites de un GameObject), del cual no se puede saber a qué objeto pertenece. Por lo tanto es necesario preguntar por el tag del Gameobject. El tag es una propiedad de los Gameobjects que los identifican en la escena.

```
void OnTriggerEnter(Collider other){
    if(other.CompareTag("Jugador")){
        CausarDaño(other.gameObject);
        Morir();
    }
    else if(other.CompareTag("DisparoJugador")){
        Disparo d = other.gameObject.GetComponent<Disparo>();
        RestarVida(d.GetDaño());
        Explotar();
        Destroy(other.gameObject);
    }
}
```

Método OnTriggerEnter implementado en la clase Asteroide.

Por otro lado, para que las naves puedan disparar se precisó de una clase Disparo que es instanciada por una Nave y es la encargada de impactar en los elementos del juego.

Es importante destacar la comunicación entre PowerUp y Nave. Cuando estas colisionan, PowerUp le envía un mensaje a Nave modificando temporalmente uno de sus atributos y la Nave hace uso de este para regresar a su estado normal pasada la duración correspondiente del beneficio. Gracias a las corrutinas se logró que dado un tiempo X, la nave se encuentre en un “estado de beneficio” y luego retorne a su “estado normal”. Una corrutina es una función que tiene la habilidad de pausar su ejecución y devolver el control a Unity para luego continuar donde lo dejó.

A continuación se demuestra cómo se pensaron las colisiones de los elementos en la escena a partir de una tabla. Esta se utilizó para entender quién tiene que realizar una acción al momento de colisionar.

	Asteroide	**Nave Enemiga**	**Nave Jugador**	**Disparo Enemigo**	**Disparo Jugador**	**PowerUp**
Asteroide	- - - - - -	- - - - - -	Se causa daño al jugador y el asteroide es destruido.	- - - - - -	Se causa daño al asteroide y el disparo es destruido.	- - - - - -
Nave Enemiga	- - - - - -	- - - - - -	- - - - - -	- - - - - -	Se causa daño a la nave enemiga y el disparo es destruido.	- - - - - -
Nave Jugador	Se causa daño al jugador y el asteroide es destruido.	- - - - - -	- - - - - -	Se causa daño a la nave jugador y el disparo es destruido.	- - - - - -	Se obtiene un beneficio temporal y el PowerUp es destruido.
Disparo Enemigo	- - - - - -	- - - - - -	Se causa daño a la nave jugador y el disparo es destruido.	- - - - - -	- - - - - -	- - - - - -
Disparo Jugador	Se causa daño al asteroide y el disparo es destruido.	Se causa daño a la nave enemiga y el disparo es destruido.	- - - - - -	- - - - - -	- - - - - -	- - - - - -
PowerUp	- - - - - -	- - - - - -	Se obtiene un beneficio temporal y el PowerUp es destruido.	- - - - - -	- - - - - -	- - - - - -

Tabla de colisiones

GENERACIÓN DE ELEMENTOS

Para generar los elementos en la escena, se implementó la clase abstracta CreadorElementos() basada en el patrón *factory*. Este patrón define una clase para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar, esto permite que una clase delegue en sus subclases la creación de objetos(en el proyecto esto lo implementa la clase CreadorCompuesto). El inconveniente que conlleva esta implementación es que a la hora de agregar algún elemento al juego, es necesario definir el creador correspondiente.

Esta solución surgió ante la problemática del *Spawner* o generador de elementos en la escena. Es una clase que se encarga de modelar la aparición de enemigos, obstáculos e ítems en escena de forma aleatoria, por lo tanto fue necesario resolver cómo crear instancias de estos objetos de forma automática, delegando la responsabilidad a clases constructoras específicas.

ALMACENAMIENTO DE DATOS

Para guardar la información correspondiente al ranking, a los prefabs del juego y a la nave seleccionada del jugador, se realizó una clase FileManager. Esta clase, a partir de un *file path* permite insertar y leer una línea determinada del archivo y también leerlo completo. La ventaja de implementar un FileManager es que se delega la manipulación de archivos a una sola clase.

Con la implementación de la clase, se facilita la carga de *prefabs* ya que los *paths* están almacenados en un archivo, en el caso que se quiera agregar un elemento al juego, basta con agregar el *path* al archivo que los contiene. Sin esta clase, la única forma posible permitida por Unity era asignándolos dentro del código manualmente a una estructura estática, por lo tanto, al querer agregar un elemento, no solo se tenía que copiar el *path* en el código correspondiente, sino también cambiar el tamaño del arreglo. Como conclusión se puede observar que la clase FileManager proporciona escalabilidad al programa.

```
obstaculos = new Object[6];

obstaculos[0] = AssetDatabase.LoadAssetAtPath("Assets/Prefabs/Spawner/Asteroide_Gris01.prefab", typec
obstaculos[1] = AssetDatabase.LoadAssetAtPath("Assets/Prefabs/Spawner/Asteroide_Gris02.prefab", typec
obstaculos[2] = AssetDatabase.LoadAssetAtPath("Assets/Prefabs/Spawner/Asteroide_Gris03.prefab", typec
obstaculos[3] = AssetDatabase.LoadAssetAtPath("Assets/Prefabs/Spawner/Asteroide_Marron01.prefab", typ
obstaculos[4] = AssetDatabase.LoadAssetAtPath("Assets/Prefabs/Spawner/Asteroide_Marron02.prefab", typ
obstaculos[5] = AssetDatabase.LoadAssetAtPath("Assets/Prefabs/Spawner/Asteroide_Marron03.prefab", typ
```

Carga manual de las rutas de los prefabs

JUGADOR VS CONTROLADOR JUGADOR

Para modelar el comportamiento relacionado a la nave del jugador y el usuario en sí, se implementaron dos clases: Jugador y ControladorJugador. Por un lado, la clase ControladorJugador se encarga de controlar las físicas relacionadas al movimiento y disparo de la nave. Esta clase se encarga de formar los vectores de movimiento a partir de las entradas del teclado generadas por el usuario, mientras que el disparo se ejecuta automáticamente cuando la nave ingresa en escena. Por otro lado, la clase Jugador es la encargada de llevar la cuenta de la puntuación del usuario y de comprobar el estado de la nave, para saber cuando finaliza el juego. Esto es necesario ya que la instancia de ControladorJugador es eliminada cuando la nave muere, por lo tanto el control de puntos tiene que ser modelado en una clase externa al comportamiento de la nave.

A diferencia de la nave del jugador, la nave enemiga es controlada con una clase ControladorEnemigo, que genera un vector de movimiento vertical, igual para todas las instancias de las naves enemigas y también se encarga de que esta dispare.

4. CONSIDERACIONES

NAVE JUGADOR

En la escena de elegir una nave, surgió el siguiente problema:

¿Cómo le comunico al jugador la nave elegida si este no se encuentra en escena?

Como solución se implementó un Selector, este permite la comunicación entre las dos escenas a partir de un archivo de texto, en el cual se almacena el filepath de la nave elegida y luego es leído por el jugador.

NOMBRE ARCADE

Dado que el género del videojuego es Arcade, una vez que se logra un récord se le solicita al jugador que ingrese sus iniciales en la pantalla del juego.

La implementación de esta característica trajo un problema al momento de almacenar un nuevo récord:

La clase ranking debe esperar a que el usuario ingrese sus

iniciales, para escribir en el archivo

Este error ocurre debido a que los métodos eran sincrónicos. Por lo tanto la solución tiene que ser mediante métodos asíncronos, que se realicen de forma que no se produzcan anomalías en el guardado. Para resolverlo se agregó la librería: *System.Threading.Tasks.* Esta permite utilizar los operadores async y await. El operador await se aplica a un task asincrónico para insertar un punto de suspensión en la ejecución del método hasta que finalice la tarea esperada. En este caso la tarea representa un trabajo en curso. Un await no bloquea el hilo en el que se está ejecutando, en cambio, hace que el compilador registre el resto del método async como una continuación en la tarea esperada. Entonces el control regresa al *caller* del método async. Cuando la tarea finaliza, invoca su continuación, y la ejecución del método async se reanuda donde lo dejó.

La clase NombreArcade es la encargada de recibir las iniciales del usuario mediante una corrutina y la implementación de lo explicado en el párrafo anterior.

AUDIO

Para mejorar la experiencia del jugador, el juego cuenta con mecanismos que permiten la reproducción de audio. Por un lado, se pueden reproducir canciones y por otro sonidos de explosiones, efectos, etc.

Para reproducir las canciones se diseñó una clase abstracta ReproductorMusica que tiene un arreglo de canciones y la canción a reproducir. De esta clase surgieron dos subclases: ReproductorMusicaEscenario se utiliza en un escenario en particular. A esta clase se le deben brindar las canciones que se pueden reproducir en una única escena, una vez finalizado el juego puede reproducir una canción en particular. Se consideró que la canción en la última posición del arreglo, simboliza la canción de fin de juego. Por otro lado surgió ReproductorMusicaEscenas que permite que un conjunto de canciones o una canción en particular perdure en una determinada cantidad de escenas, esto se logra con la llamada a un método de Unity: *DontDestroyOnLoad().*

Para reproducir sonidos particulares se utilizaron los componentes que provee Unity.

BOTONES

Para comenzar a jugar, el usuario debe presionar botones en la UI. Estos son los encargados de permitir el flujo a través de las escenas. Para ello se diseñó una clase particular, que dado un número de escena al hacer click sobre el botón te dirige a ella.

5. CONCLUSIÓN

Como conclusión, se quiere hacer énfasis en los beneficios que aporta utilizar un paradigma orientado a objetos en el desarrollo de un videojuego. Los principales beneficios son:

- **Reusabilidad**. Cuando se diseña adecuadamente las clases, se pueden reutilizar fácilmente en otro proyecto.
- **Modificabilidad**. La facilidad de añadir, suprimir o modificar nuevos objetos permite hacer modificaciones de una forma muy sencilla, por lo tanto es una gran ventaja a la hora de agregar elementos nuevos al videojuego.
- **Fiabilidad**. Al dividir el problema en partes más pequeñas se puede probar de manera independiente y aislar mucho más fácilmente los posibles errores que puedan surgir. En el caso de un videojuego, se puede aislar el comportamiento de los diferentes objetos en escena y poder testearlos fácilmente.

Es importante destacar el beneficio de la utilización de archivos, ya que se ha minimizado la duplicación de datos, de manera que se optimiza el aprovechamiento de memoria, teniendo en cuenta las exigencias gráficas de un videojuego, se consideró muy importante la optimización en cuanto a duplicación de datos y la elección correcta de estructuras de almacenamiento.