UFO Defense: Desarrollo de un juego tower defense con Unity

Máster en desarrollo de aplicaciones sobre dispositivos móviles

Autor: Gilberto José Conca Pascual

Tutor: Jordi Joan Linares Pellicer

Septiembre 2017

Índice

1. Introducción

En este proyecto se desarrolla UFO Defense, un juego del género "tower defense", utilizando Unity

1.1. Objetivos

Este trabajo persigue desarrollar un juego del género *tower defense* utilizando el motor Unity, para dispositivos móviles. También se busca aprovechar algunas características que pueden ofrecer este tipo de dispositivos, como por ejemplo Google Play Games.

1.2. Motivación

La motivación que me ha llevado a plantearme y desarrollar este trabajo es el gusto que tengo por los videojuegos, especialmente los de estrategia. También el reto que suponía hacer este juego en 3D, ya que no había desarrollado antes yo solo un juego en 3D.

2. El juego

2.1. Género

El género Tower defense se basa en que el jugador debe defender un lugar de oleadas de enemigos construyendo torres. Es estratégico ya que la posibilidad de construir torres suele estar limitada ya sea por dinero o por espacio. Algunas características comunes en estos juegos es el tener a disposición del jugador varios tipos de torres, las cuales suelen poder mejorarse, como así también varios tipos de enemigos cada uno con sus peculiaridades. Los niveles normalmente constan de un punto de inicio por donde llegan los enemigos, uno o varios caminos que estos tomarán, y finalmente el punto de destino. También suele ser común en algunos juegos de este género que la cantidad de enemigos en las oleadas más avanzadas que aparecen a la vez sea enorme, para así abrumar al jugador. Sin embargo en este juego se ha optado por oleadas que produzcan enemigos de uno en uno, aún así invocando a bastantes enemigos en total, siendo las oleadas más o menos espaciadas en el tiempo mientras se evitan acumulaciones. Esta decisión se debe a que en móviles menos potentes el juego podría ralentizarse un poco.

2.2. Jugabilidad

El juego se compone de 4 niveles, antes de comenzar cada nivel hay una pantalla de preparación en la que se puede escoger el tipo de torreta a utilizar, no se pueden escoger más de tres torretas y tampoco no se puede empezar el nivel sin seleccionar ninguna, el juego avisa mediante texto y un efecto sonoro cuando no se cumplen los requisitos para empezar el nivel.

Cuando un nivel empieza, se le dejan cinco segundos de gracia al jugador para que coloque las torretas que pueda antes de empezar a generar los enemigos. Tras comenzar una oleada se inicia la cuenta atrás para la siguiente, esto sigue así hasta la última, tras la cual al acabar con todos los enemigos se completa el nivel.

Cada nivel se comienza con una cantidad de dinero predeterminada para construir torretas (siendo siempre suficiente cantidad para construir cualquier torreta), sin embargo se consigue más al destruir enemigos, cada tipo de enemigo da una cantidad distinta acorde a su dificultad y salud. También se puede aumentar la cantidad de dinero agitando el dispositivo móvil.

La vida del jugador en los tres primeros niveles es 10 y en el último 5. Si se agota se enseña la pantalla de Game Over, desde la cual se puede volver al menú principal o reiniciar el nivel. Por el contrario al conseguir sobrevivir a todas las oleadas se desbloquea el siguiente nivel y se desplegará la pantalla de nivel completado donde se puede ir al siguiente nivel o volver al menú principal.

2.2.1. Los controles

El control en este juego es muy simple, el juego se sirve de botones para seleccionar torretas y luego hay que tocar el lugar válido donde se desee colocar la torreta seleccionada.

Si se toca una torreta ya construida, se despliega un pequeño menú encima de esta, donde hay cuatro botones, uno para mejorar la torreta, el cual indica

el precio de mejora y si ésta se puede mejorar; otro para venderla, el cual indica también el dinero que se obtendrá al venderla, y otros dos que sirven para indicar el objetivo al que atacarán, el más cercano o más alejado de esta. Si se vuelve a tocar la misma torreta con el menú desplegado este desaparece, y si se toca otra para abrir el menú, aparecerá uno encima de la otra torreta y se cerrará el que ya estaba abierto.

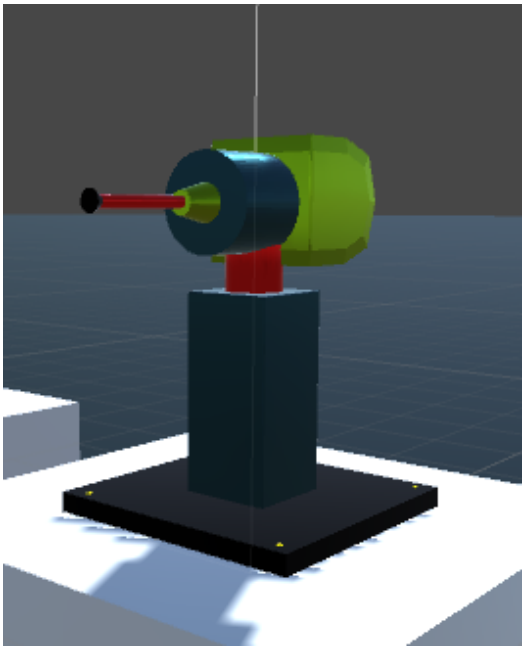
2.3. Las torres

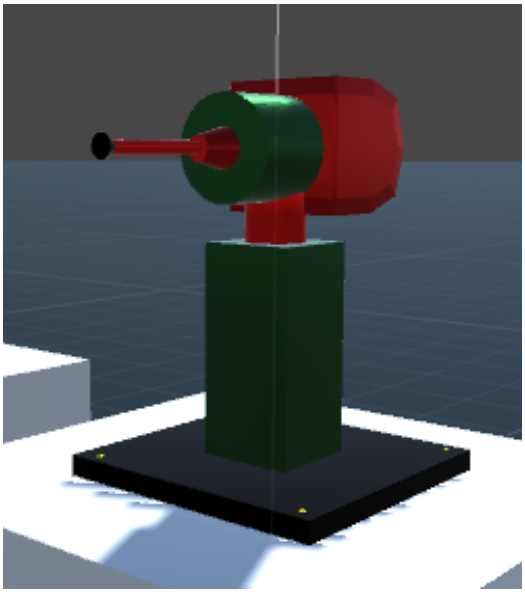
Las torres sólo se pueden construir en unas casillas predeterminadas, una vez construidas se pueden mejorar, vender, o seleccionar si disparará a los blancos más alejados en su radio de visión o a los más cercanos. Cada torreta tiene un precio de compra, uno de mejora y uno de venta, siendo este último la mitad del precio original de la torre. Las torretas mejoradas tienen mayor rango y pueden disparar balas más letales.

Hay cuatro tipos de torres:

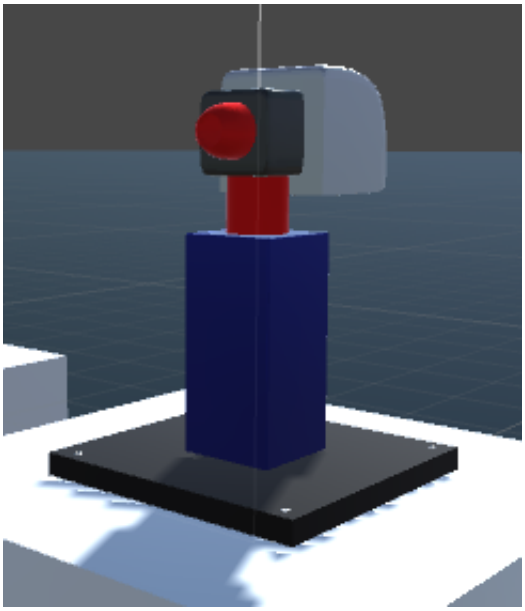
- Torreta normal: Una torreta barata con capacidad de daño moderada y rango medio.
- Lanzamisiles: Sus disparos hacen daño de área
- Láser: Daño constante que atraviesa obstáculos y ralentiza al enemigo.
- Ametralladora: Más cara con más rango y ratio de disparo que la torreta normal. No se puede mejorar

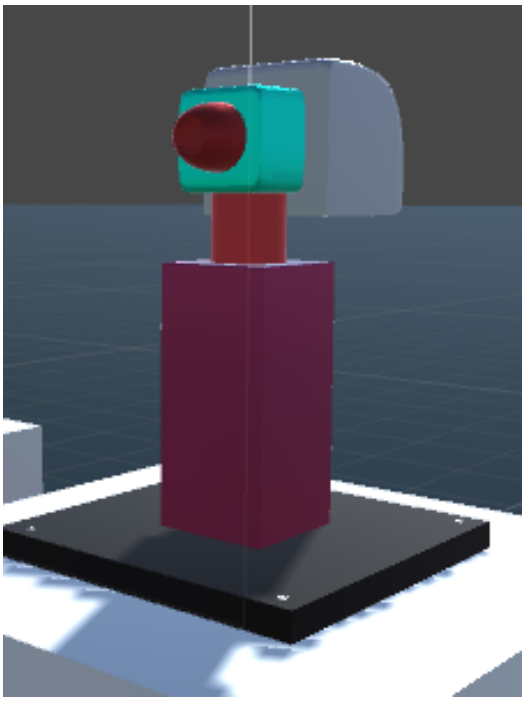
Normal:



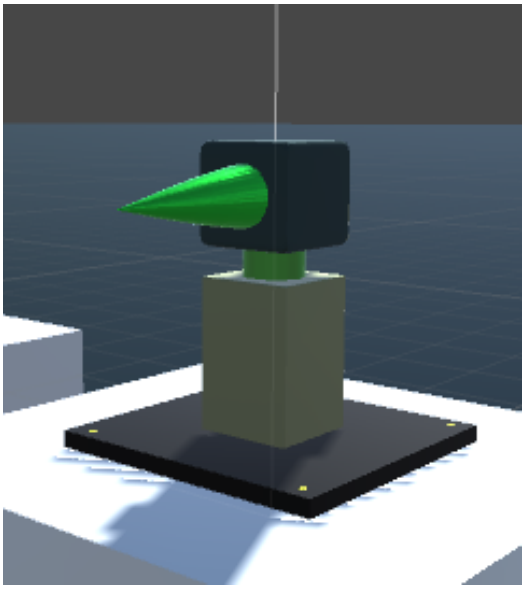


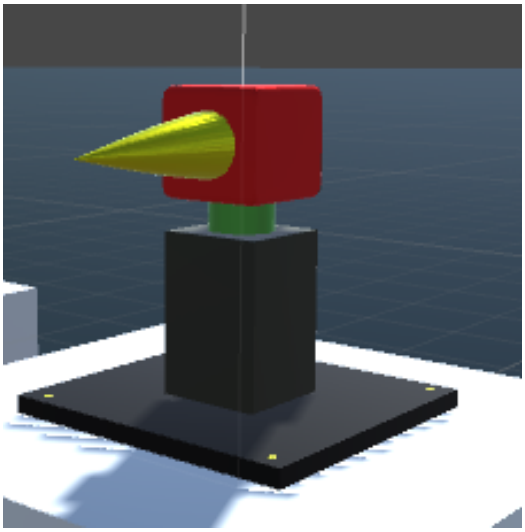
Lanzamisiles:



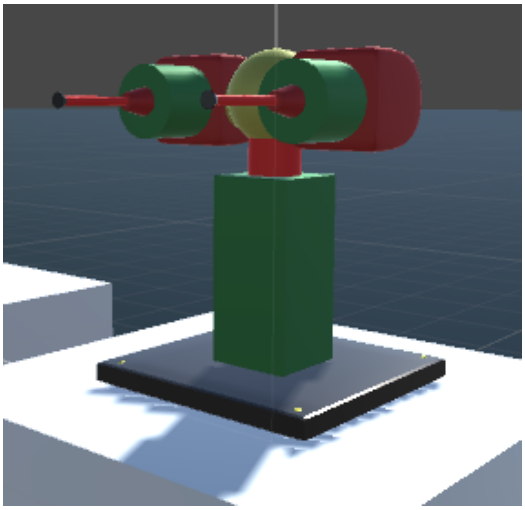


Láser:





Ametralladora:

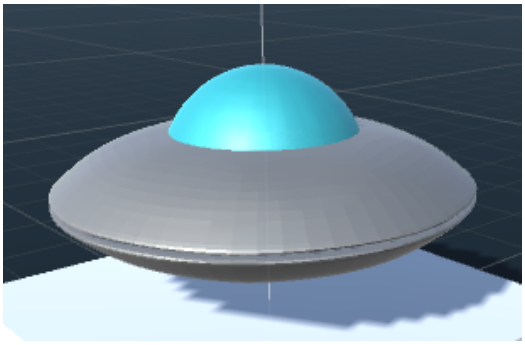


2.4. Los enemigos

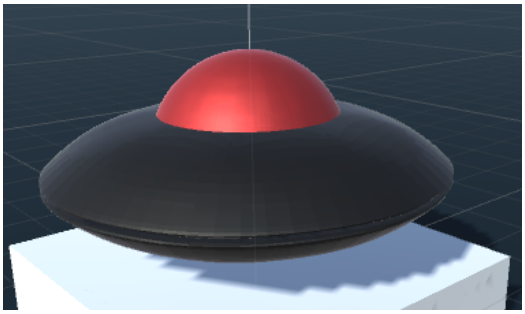
Su objetivo es llegar desde su punto de aparición hasta la base del jugador mientras cruzan un camino predeterminado. Sobre ellos hay una barra de vida para que el jugador pueda ver como afectan las torretas a los distintos tipos de enemigos, y si es necesario cambiar la estrategia y modificar las torretas. Los distintos tipos de enemigos son:

- Normal: Con una velocidad y vida moderadas simplemente se dirigen hacia la base del jugador.
- Duro: Mucha vida y poca velocidad.
- Rápido: Rápidos, pero con una cantidad de vida ridícula.
- Teleportador: Algo menos de vida que el normal, pero puede aparecer a mitad del recorrido.

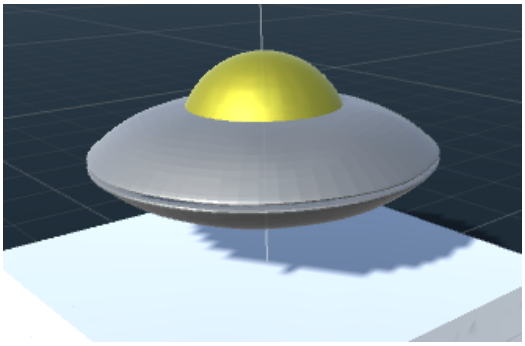
Normal:



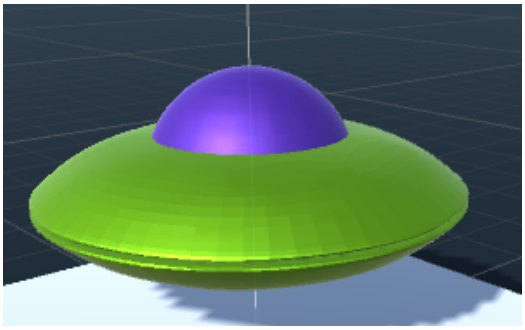
Duro:



Rápido:



Teleportador:



El movimiento de los enemigos está basado en dos scripts y unos objetos aparte llamados del propio enemigo llamados waypoints. Cada waypoint indica el camino a seguir y es hijo de un GameObject llamado Waypoints con un script con el mismo nombre, el cual guarda las posiciones de todos los waypoints ordenados en un array publico. Luego el script de movimiento del enemigo hace que vaya de uno a otro hasta llegar al último que es el que se encuentra en la base del jugador, entonces el enemigo se destruye y el jugador pierde una vida.

La barra de vida se compone de un Canvas con dos imágenes superpuestas las cuales son simples rectángulos. La que se encuentra delante es de color verde y desde el editor se configura para que se pueda reducir el tamaño horizontalmente de derecha a izquierda mediante script. Debajo de esta imagen se encuentra la otra de color negro. Esto hace el efecto de que parezca que la barra de vida se vacíe.

```
//Cada vez que el enemigo sufre daño
public void TakeDamage (float amount)
    {
        health -= amount;

        //Actualiza la barra de vida. fillAmount entre 1 y 0
        healthBar.fillAmount = health / startHealth;

        if(health <= 0 && !isDead) //Si la vida llega a 0
        {
            Die();
        }
    }
```

3. Tecnologías utilizadas

3.1. Unity

Unity es un motor para crear videojuegos desarrollado por Unity Technologies con capacidad de desarrollar videojuegos para muchas plataformas, entre ellas PC, PlayStation (3, 4), XboX (360, One), Android, IOS, e incluso algunos modelos de Smart TV. También puede desarrollar para realidad virtual.

Un punto fuerte bastante importante de Unity es su potencia al calcular físicas, colisiones, etc...

Además aunque este juego esté desarrollado en 3D, Unity ha incorporado facilidades que ya permiten el desarrollo eficaz de juegos en 2D.

Otra de las bondades de Unity es que resulta fácil de aprender a la par que potente.

3.2. Blender

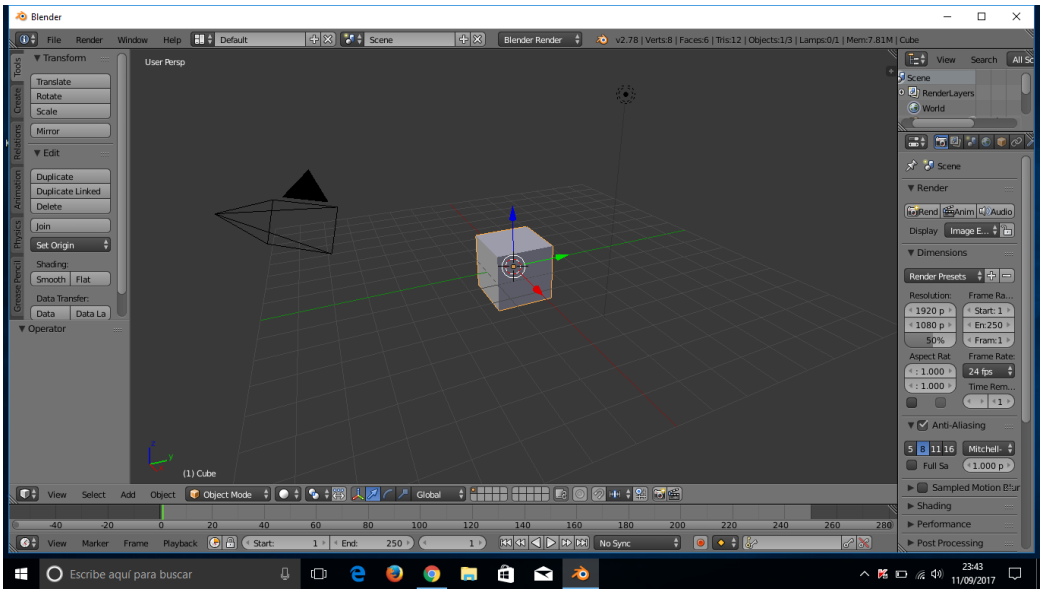
En el desarrollo del juego se ha utilizado otra tecnología a parte de Unity. Las torretas se han modelado usando Blender.

Blender es un software libre de modelado 3D bastante potente, aunque no sólo se especializa en el modelado, también se pueden producir animaciones gracias a que incorpora edición de video. Es compatible con los sistemas operativos: Windows, MacOS, Linux y Solaris entre otros.

Es tan completo y potente que, muchos estudios que utilizan software especializado de pago contratan a gente que sepa manejar Blender, pues es capaz de equipararse a estos.

Aunque las nociones básicas son relativamente fáciles de aprender, es un programa bastante difícil de dominar.

Al ser libre y gozar de una comunidad activa existen infinidad de extensiones que facilitan muchas tareas, algunas incluso se añaden oficialmente en versiones posteriores.



4. Google Play Services

Google Play Services permite añadir ciertas características al juego como: logros, tablas de clasificación, guardado en la nube, eventos, misiones, regalos *in-game*, etc... Incluso facilita poder hacer un juego multijugador (con opción que sea a tiempo real o por turnos).

Estos servicios son bastante populares, ya sea porque facilitan al jugador conservar su progreso cuando cambia de dispositivo (guardado en la nube), un reto añadido (logros), alicientes para seguir jugando o recompensas por ciertas acciones (misiones, regalos, eventos), o incluso la posibilidad de competir contra otros (multijugador, tablas de clasificaciones).

Este juego cuenta con propiedades de Google Play Services, sin embargo para añadirlas mediante Unity se requiere un plugin. UFO Defense utiliza el plugin desarroyado por Google llamado *Google Play Games plugin for Unity*, el cual permite la adición de dichos servicios.

Su instalación es sencilla, se descarga un zip desde la página de GitHub y se extrae, luego desde Unity se importa el paquete de assets, y tras un pequeño rato de instalación ya está instalado. Tras esto ya se puede añadir el código necesario para manejar los servicios.

También hay que tener una cuenta de Google y haber pagado la tasa de 25 euros para poder acceder a Google Play Conosle. Para poder empezar a configurar los servicios en un juego hay que añadir el juego en la pestaña de "Servicios de juegos". Para esto no es necesario subir aún el APK, sin embargo hay que añadir la clave SHA1, para lo cual hay que generar el keystore del juego.

Para que los servicios funcionen hay que incluir en los scripts de Unity el inicio de sesión a Google Play Games:

```
PlayGamesPlatform.Activate();

Social.localUser.Authenticate((bool success) =>
            {

            });
```

4.1. Logros

Actualmente se han popularizado los logros, que son como unos hitos que se obtienen al cumplir ciertos objetivos en el juego. Esto puede servir para mantener a los jugadores en el juego o para que intenten distintas estrategias.

Para añadir un logro al juego, primero se debe añadir en la consola de desarrollo de Google Play. Para esto una vez añadido el juego a los servicios para juegos, hay que añadir un logro en su respectiva sección. Cada logro necesita un nombre, una descripción, una imagen que lo represente, y su valor en puntos dependiendo de la dificultad del logro. También está la posibilidad de mantenerlo oculto al jugador y revelarlo cuando se cumplan ciertas condiciones en el juego.

Los logros pueden ser de dos tipos, los que se completan en un único paso y los incrementales. Los primeros de desbloquean cuando el jugador realiza una acción específica, mientras que los segundos requieren la repetición de una acción un número de veces determinado.

Cuando el logro ha sido creado y configurado en la consola de desarrollo se tienen que obtener los recursos generados por Google Play Console en los cuales están los códigos de identificación de cada logro que también han sido generados. Estos recursos son un texto xml que se debe copiar y pegar en la ventana Android setup de Unity. Un ejemplo de recursos:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
Google Play game services IDs.
Save this file as res/values/games-ids.xml in your project.
-->
<resources>
  <!-- app_id -->
  <string name="app_id" translatable="false">
    numero_id_de la_app</string>
  <!-- package_name -->
  <string name="package_name" translatable="false">
    nombre_del_paquete</string>
  <!-- achievement First of many -->
  <string name="achievement_first_of_many" translatable="false">
    id_logro</string>
  <!-- achievement Investment -->
```

```
  <string name="achievement_investment" translatable="false">
    id_logro</string>
  <!-- achievement Crusher -->
  <string name="achievement_crusher" translatable="false">
    id_logro</string>
  <!-- achievement Destroyer -->
  <string name="achievement_destroyer" translatable="false">
    id_logro</string>
  <!-- achievement Obliterator -->
  <string name="achievement_obliterator" translatable="false">
    id_logro</string>
  <!-- achievement Initiated -->
  <string name="achievement_initiated" translatable="false">
    id_logro</string>
</resources>
```

Las IDs de los logros son códigos alfanuméricos.

Una vez hecho esto se han de implementar el manejo de logros en los scripts de Unity. Para cualquier logro hay que indicar el identificador del logro correspondiente.

Para los logros únicos hay dos formas de manejarlos. Para desbloquearlo:

```
using GooglePlayGames;
using UnityEngine.SocialPlatforms;

PlayGamesPlatform.Instance.UnlockAchievement
    (ID, (bool success) =>
        {
            if(success)
                //
            else
                //
        });
```

Otra forma:

```
using GooglePlayGames;
using UnityEngine.SocialPlatforms;

Social.ReportProgress
```

```
(ID, 100.0f, (bool success) =>
    {
        if(success)
            //
        else
            //
    });
```

En esta última forma hay que tener en cuenta que si el logro está oculto y se desea que se sólo pueda mostrar al jugador y no desbloquearlo en ese momento, hay que cambiar el 100.0f que se pone al lado de la ID por 00.0f

Para desbloquear los logros incrementales el código cambia:

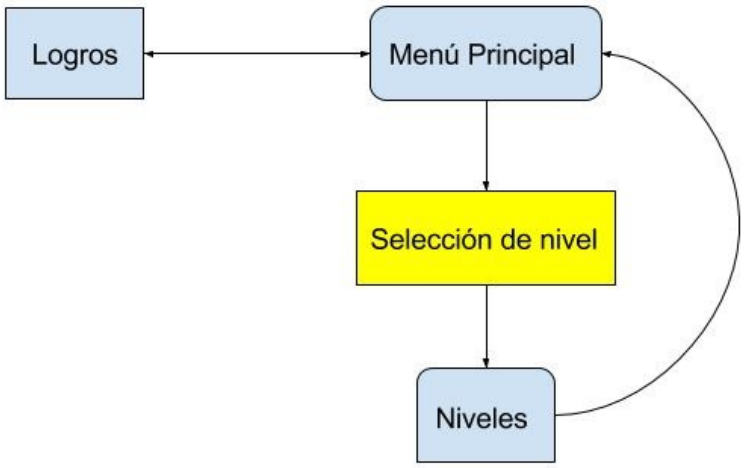
```
using GooglePlayGames;
using UnityEngine.SocialPlatforms;

PlayGamesPlatform.Instance.IncrementAchievement
(ID, numero_de_incremento, (bool success) =>
        {
            if (success)
                //
        });
```

Como se puede ver, al lado del número del identificador se añade el número de incremento del logro.

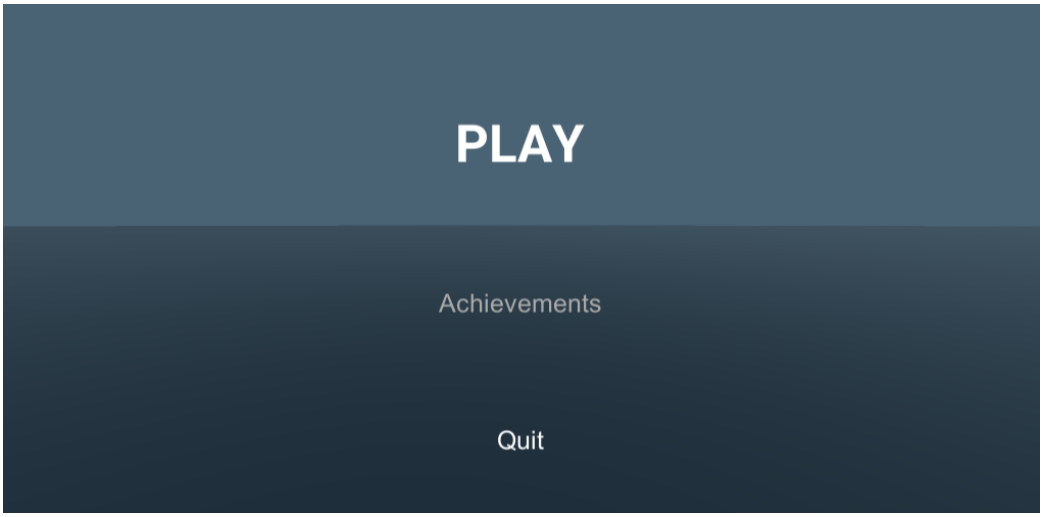
5. Arquitectura de la aplicación

5.1. Esquema de diseño

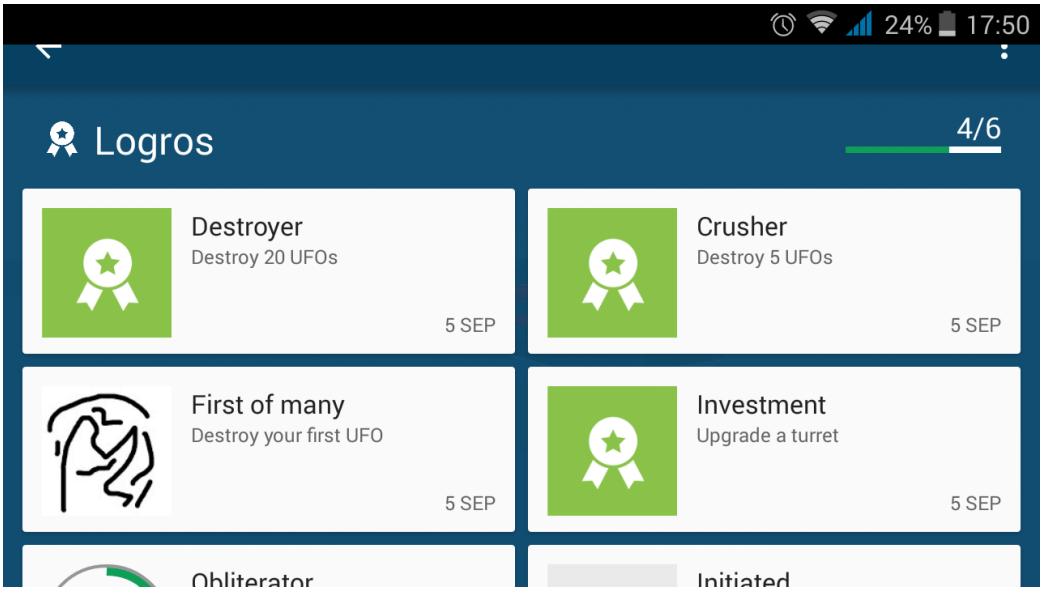


El menú principal es la primera pantalla de la aplicación, desde donde se puede acceder a la pantalla de selección de nivel, a la lista de logros, y salir del juego.

Desde el menú principal se hace login a la cuenta de Google Play del usuario, lo cual permite la consecución de logros.

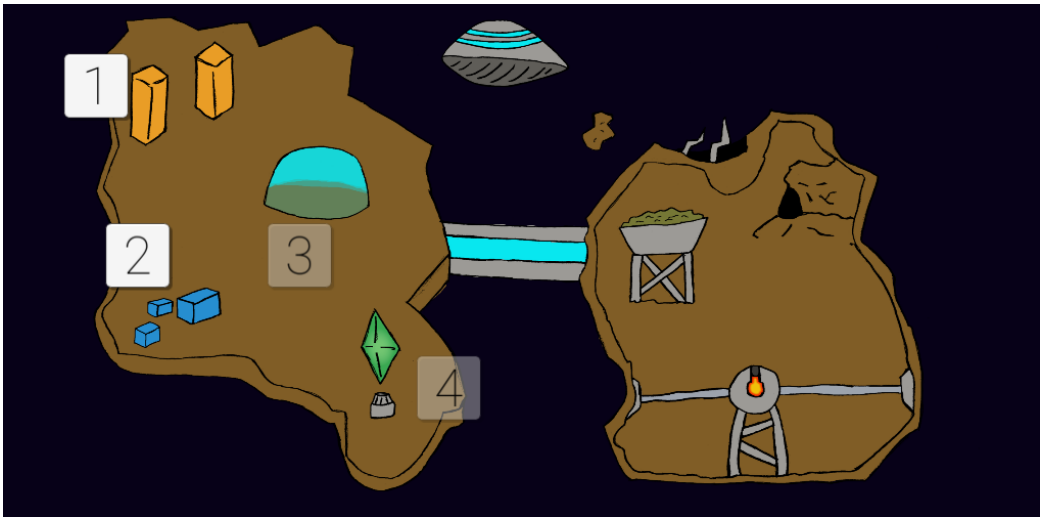


Al pulsar el botón Achievements se accede a la pantalla de logros, la cual sólo se puede acceder si se está logeado. Esta pantalla es una interfaz propia de Google Play Games, se muestran los logros que se han conseguido, y de los que aún faltan por conseguir muestra su progreso si son incrementales. También se pueden ver los puntos que vale cada logro y su descripción.

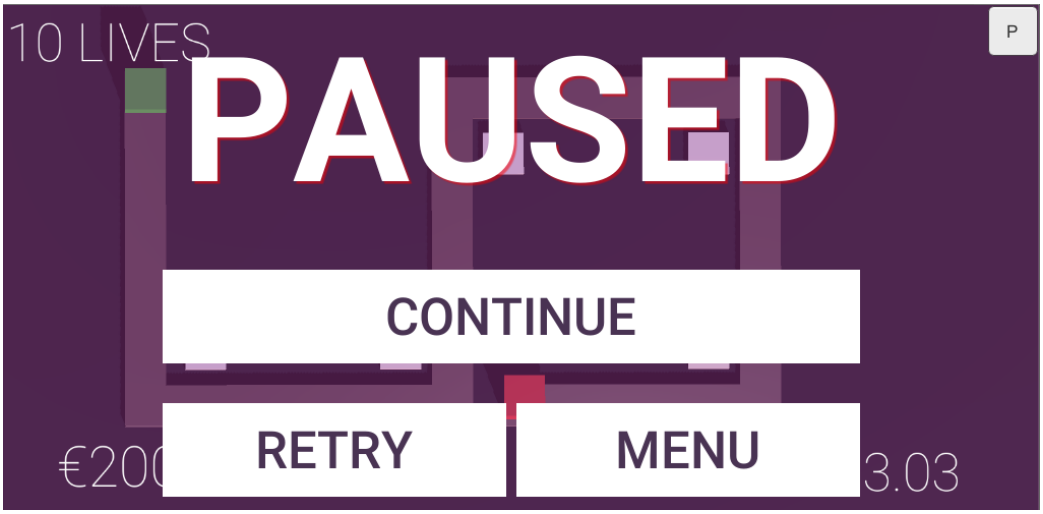


Desde la pantalla de selección de nivel se puede acceder a cualquier nivel

del juego que se haya desbloqueado. Para desbloquear un nivel se debe acabar primero el anterior.



Desde los propios niveles se puede acceder al menú de pausa el cual permite volver al menú principal o reiniciar el nivel



Los niveles tienen otra interfaz, la de Game Over. Cuando se pierde un nivel aparece esta pantalla, la cual es bastante parecida a la de pausa. Desde aquí se puede volver al menú principal y reintentar el nivel.

6. Conclusiones

En este trabajo se ha desarrollado un juego para dispositivos móviles utilizando Unity. El género del juego es un tower defense, el cual se adapta bien a estos dispositivos pese a ser generalmente popular en PC, aunque los niveles se han tenido que hacer más rápidos y sencillos que en PC debido a que en las plataformas móviles suele ser más conveniente que los niveles de los juegos no sean extremadamente largos.

Se ha descrito en profundidad la jugabilidad y los elementos más importantes del juego, como las torres y sus mejoras y los enemigos, haciendo comentarios sobre algunas partes importantes del código.

Luego se han descrito brevemente las tecnologías utilizadas, tanto Unity para el desarrollo como Blender para el modelado 3D de las torretas, obstáculos y enemigos.

A continuación se ha explicado detalladamente la inclusión de la API de Google Play Services, necesaria para implementar los logros, los cuales son una característica interesante que proporciona Google.

Finalmente se ha mostrado el flujo de escenas de la aplicación y comentado lo que sucede en cada una con detalle.

7. Anexos

A continuación se incluyen algunos scripts interesantes que forman parte esencial del juego.

7.1. Movimiento de los enemigos

7.1.1. Enemy

La clase Enemy, como se puede ver a continuación, controla las funciones generales del enemigo. Tiene variables esenciales como la velocidad, vida y valor, también un booleano que dependiendo de su estado indica si el enemigo se puede teletransportar. Esta clase controla el daño, la destrucción, mermas de velocidad y La gestión de logros relacionados con la destrucción de estos.

```
using UnityEngine;
using UnityEngine.UI;

using GooglePlayGames;
using UnityEngine.SocialPlatforms;

public class Enemy : MonoBehaviour {

    //Variables//
    public float startSpeed = 10f;

    [HideInInspector]
        public float speed;

    public float startHealth = 100;
    private float health;

    public int worth = 50;

    public GameObject deathEffect;

    [Header("Enemy Type")]
    public bool teleporter;
```

```
public GameObject teleportEffect;

[Header("Unity Stuff")]
public Image healthBar;

//Para evitar bug que hace que muera varias veces.
private bool isDead = false;

//-------------------------------------------------
private void Start()
{
    speed = startSpeed;
    health = startHealth;
}


public void TakeDamage (float amount)
{
    health -= amount;

    healthBar.fillAmount = health / startHealth;

    if(health <= 0 && !isDead)
    {
        Die();
    }
}


public void Slow(float percentage)
{
    speed = startSpeed * (1 - percentage);
}


void Die() //Muerte del enemigo
{
    isDead = true;
```

```
//Logro  First of many
PlayGamesPlatform.Instance.UnlockAchievement
    (GPGSIds.achievement_first_of_many, (bool success) =>
    {
        //TODO Algo
        if (success)
            Debug.Log("1st DED");
        else
            Debug.Log("Error");
    });


//Logros Destruir enemigos
//Crusher
PlayGamesPlatform.Instance.IncrementAchievement
    (GPGSIds.achievement_crusher, 1, (bool success) =>
    {
        if (success)
            Debug.Log("Crusher ++");
    });

//Destroyer
PlayGamesPlatform.Instance.IncrementAchievement
    (GPGSIds.achievement_destroyer, 1, (bool success) =>
    {
        if (success)
            Debug.Log("Destroyer ++");
    });

//Obliterator
PlayGamesPlatform.Instance.IncrementAchievement
    (GPGSIds.achievement_obliterator, 1, (bool success) =>
    {
        if (success)
            Debug.Log("Obliterator ++");
    });
////
```

```
        PlayerStats.Money += worth;

        GameObject effect = Instantiate(
            deathEffect, transform.position, Quaternion.identity);
        Destroy(effect, 5f);

        //Reduce el número de enemigos vivos
        WaveSpawner.EnemiesAlive--;

        Destroy(gameObject);
    }
}
```

7.1.2. EnemyMovement

EnemyMovement está directamente relacionada con Enemy y requiere que el mismo GameObject que disponga de este script tenga también el script Enemy, también hace uso de la clase Waypoints para saber cual es el siguiente destino en el mapa. El momimiento puede estar condicionado por el tipo del enemigo, si este es un teleportador aparecerá primero en una posición aleatoria hasta la mitad del mapa y luego continuará normalmente. Siempre que acaba un movimiento se restablecerá la velocidad inicial, provocando que los láseres ralenticen al enemigo sólo mientras le están disparando.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(Enemy))]
public class EnemyMovement : MonoBehaviour {

    //Variables//
    private Transform target;    //siguiente destino
    private int waypointIndex = 0; //Índice para los waypoints

    private Enemy enemy;
```

```
//Variable exclusiva de teleportador
private bool hasTeleported = false;

//-----------------------------------------------
void Start()
{
    enemy = GetComponent<Enemy>();

    //Apunta al primer waypoint
    target = Waypoints.points[0];
}


void Update()
{
    //Calculo de teleportación
    if (enemy.teleporter && !hasTeleported)
    {
        int destination = Random.Range(
            1, Waypoints.points.Length/2 + 1);

        transform.position = Waypoints.points[
            destination].transform.position;
        waypointIndex = destination + 1;
        target = Waypoints.points[waypointIndex];

        GameObject efect = Instantiate(
            enemy.teleportEffect,
            transform.position, Quaternion.identity);
        Destroy(efect, 0.05f);

        hasTeleported = true;
        return;
    }

    //Cálculos de movimiento//
    Vector3 direction = target.position - transform.position;
```

```
        transform.Translate(
            direction.normalized * enemy.speed * Time.deltaTime,
            Space.World);

        //Si estamos en el waypoint objetivo
        if (Vector3.Distance(
            transform.position, target.position) <= 0.4f)
        {
            GetNextWaypoint();
        }

        enemy.speed = enemy.startSpeed; //Restablece la velocidad
    }


    //Se aumenta el índice y se apunta al siguiente waypoint objetivo
    void GetNextWaypoint()
    {
        //Si se han acabado los waypoints
        if (waypointIndex >= Waypoints.points.Length - 1)
        {
            EndPath();
            //Evita errores por posible retraso en la destrucción
            return;
        }

        waypointIndex++;
        target = Waypoints.points[waypointIndex];
    }


    void EndPath()
    {
        PlayerStats.Lives--;
        WaveSpawner.EnemiesAlive--;
        Destroy(gameObject);
    }
}
```

7.1.3. Waypoints

Waypoints se le ha asignado a un Game Object del mismo nombre el cual tiene otros Game Objects como hijos. Este script recoge en un array las posiciones de dichos hijos.

```
using UnityEngine;

public class Waypoints : MonoBehaviour {

    public static Transform[] points;

    private void Awake()
    {
        /*Inicializa el array con el número de hijos
        * que tenga el objeto waypoints
        */
        points = new Transform[transform.childCount];
        for (int i = 0; i < points.Length; i++)
        {
            /*asigna a la posición i de points
            * el hijo i de este objeto
            */
            points[i] = transform.GetChild(i);
        }
    }
}
```

7.2. Torretas

7.2.1. Turret

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Turret : MonoBehaviour {

    //Atributos//
```

```
private Transform target;
private Enemy targetEnemy;

[Header("General")]

public float range = 15f;

[Header("Use Bullets (default")]
public GameObject bulletPrefab;
public float fireRate = 1f;
private float fireCountdown = 0f;

[Header("Use Laser")]
public bool useLaser = false;

public int damageOverTime = 30;
public float slowAmount = 0.5f;

//Una linea que hace el efecto de un láser
public LineRenderer lineRenderer;
public ParticleSystem impactEffect;
public Light impactLight;

[Header("Target Far enemy")]
private bool farEnemy = false;

[Header("Unity Setup Fields")]

public string enemyTag = "Enemy";

//Rotación
public Transform partToRotate;
public float turnSpeed = 10f;

public Transform firePoint;
```

```
//-------------------------------------
    void Start () {

    InvokeRepeating("UpdateTarget", 0f, 0.5f);

}


/*Detecta a todos los enemigos y selecciona el que está
*a menor distancia en el rango
*/
void UpdateTarget()
{
    if (farEnemy)
    {
        UpdateFarTarget();
        return;
    }

    GameObject[] enemies =
        GameObject.FindGameObjectsWithTag(enemyTag);
    float shortestDistance = Mathf.Infinity;
    GameObject nearestEnemy = null;

    foreach (GameObject enemy in enemies)
    {
        float distanceToEnemy = Vector3.Distance(
            transform.position, enemy.transform.position);
        if (distanceToEnemy < shortestDistance)
        {
            shortestDistance = distanceToEnemy;
            nearestEnemy = enemy;
        }
    }

    if (nearestEnemy != null && shortestDistance <= range)
    {
        target = nearestEnemy.transform;
```

```
            targetEnemy = nearestEnemy.GetComponent<Enemy>();
        }
        else
        {
            //Si no hay ningún enemigo en el rango
            target = null;
        }

    }


    /*Detecta a todos los enemigos y ataca al que esté
    *a mayor distancia dentro del rango
    */
    void UpdateFarTarget()
    {
        GameObject[] enemies =
            GameObject.FindGameObjectsWithTag(enemyTag);
        float farestDistance = 0;
        GameObject farestEnemy = null;

        foreach (GameObject enemy in enemies)
        {
            float distanceToEnemy = Vector3.Distance(
                transform.position, enemy.transform.position);
            if (distanceToEnemy > farestDistance)
            {
                farestDistance = distanceToEnemy;
                farestEnemy = enemy;
            }
        }

        if (farestEnemy != null && farestDistance <= range)
        {
            target = farestEnemy.transform;
            targetEnemy = farestEnemy.GetComponent<Enemy>();
        }
        else
```

```
    {
        //Si no hay ningún enemigo en el rango
        target = null;
    }
}


//--------------------
    void Update () {

    fireCountdown -= Time.deltaTime;

    if (target == null) //¿Hay objetivo?
    {
        if (useLaser) //¿Es un láser?
        {
            if (lineRenderer.enabled)
            {
                lineRenderer.enabled = false;
                impactLight.enabled = false;
                impactEffect.Stop();
            }
        }

        return;
    }


    LockOnTarget(); //Apunta al objetivo

    if (useLaser)
    {
        Laser(); //Dispara el láser
    }else
    {
        //Ratio de disparo
        if (fireCountdown <= 0f)
        {
```

```
            Shoot();
            fireCountdown = 1f / fireRate;
        }

    }

    }

//--------------------
void LockOnTarget() //Apunta al objetivo
{
    Vector3 direction = target.position - transform.position;
    Quaternion lookRotation = Quaternion.LookRotation(direction);
    //Rotación suave
    Vector3 rotation = Quaternion.Lerp(
        partToRotate.rotation, lookRotation,
        Time.deltaTime * turnSpeed).eulerAngles;
    partToRotate.rotation = Quaternion.Euler(0f, rotation.y, 0f);
}

//--------------------
void Laser() //El laser hace daño cada frame y ralentiza
{
    targetEnemy.TakeDamage(damageOverTime * Time.deltaTime);
    targetEnemy.Slow(slowAmount);

    if (!lineRenderer.enabled)
    {
        lineRenderer.enabled = true;
        impactLight.enabled = true;
        impactEffect.Play();
    }

    lineRenderer.SetPosition(0, firePoint.position);
    lineRenderer.SetPosition(1, target.position);

    Vector3 dir = firePoint.position - target.position;
```

```
            //para que no aparezca dentro del enemigo
            impactEffect.transform.position =
                target.position + dir.normalized;

            impactEffect.transform.rotation =
                Quaternion.LookRotation(dir);

        }

        //-------------------
        void Shoot() //Instancia la bala
        {
            GameObject bulletGO = (GameObject) Instantiate(
                bulletPrefab, firePoint.position, firePoint.rotation);
            Bullet bullet = bulletGO.GetComponent<Bullet>();

            if (bullet != null)
                bullet.Seek(target); //Le pasamos el objetivo a la bala
        }

        //Para el editor. Dibuja el rango-----
        private void OnDrawGizmosSelected()
        {
            Gizmos.color = Color.red;
            Gizmos.DrawWireSphere(transform.position, range);
        }
}
```