UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CAMPUS D'ALCOI

Desarrollo de una aplicación de entretenimiento con networking para dispositivos móviles utilizando Unity3D

MEMORIA PRESENTADA POR:

Domingo Sanz Marti

Tutor:

Jordi Joan Linares Pellicer

GRADO DE INGENIERIA INFORMATICA

Convocatoria de defensa: *Julio de 2017*

RESUMEN

El proyecto llevado a cabo muestra el desarrollo de una aplicación de entretenimiento para un dispositivo móvil con sistema operativo Android. Para ello es necesario conocer el motor de videojuegos Unity 3d y las librerías de Google Play Services. También se realizará un análisis de los requisitos necesarios para realizar el diseño y desarrollo de la aplicación. En esta documentación se recogen todos los pasos seguidos para desarrollar el proyecto.

Palabras clave: Unity3D, juego, Android, aplicación, desarrollo, análisis, google, play, servicios.

ABSTRACT

The Project carried out shows the development of an entertainment application for a mobile device with Android operating system. To do this it is necessary to know the Unity3D video game engine and the Google Play Services libraries. An analysis of the requeriments necesarry to carry out the design and the development of the application will also be carried out. This documentation includes all the steps taken to develop the project.

Keywords: Unity3D, game, Android, application, development, analysis, google, play, services.

Índice general

Índice de ilustraciones

Índice de tablas

INTRODUCCIÓN

1. Descripción del proyecto

En este proyecto se describe la aplicación de entretenimiento que se ha realizado, en este caso un videojuego, llamado Defend The Barricade. Se trata de un videojuego de estrategia al estilo *tower defense* realizado en el motor Unity, al que se le añade la característica de multijugador y el cual ha sido enfocado a dispositivos móviles, específicamente con sistema operativo Android.

Se ha desarrollado con fines tanto académicos como de autorrealización, aunque una meta a largo plazo podría ser competir en el mercado con otros juegos del mismo género.

2. Motivación

La industria de los videojuegos está en pleno auge desde hace un tiempo. Cada año se baten cifras récord relacionadas con las ventas, que continúan creciendo cada año. Por ello, los videojuegos se han consolidado como primera industria de ocio audiovisual e interactivo en España, donde el consumo de éste sector supera los mil millones de euros, que se obtienen de sumar tanto la venta física como la venta online.

Ilustración 1: Consumo de videojuegos en España en el año 2016 según la AEVI

Los videojuegos son un producto de gran complejidad donde mucha gente tiene a cargo labores artísticas como diseño de personajes, escenarios, ilustraciones conceptuales, modelados 3D, animaciones y programación, no obstante, actualmente existe la cultura de la empresa independiente o como se autodenominan, empresas *indie*. Existen muchas herramientas que hoy en día facilitan la creación de videojuegos a equipos pequeños y medianos, entre ellas el motor que se utiliza en este proyecto, Unity, que permite reducir muchísimo los tiempos de desarrollo y, por lo tanto, aumentar la oferta de este tipo de productos en el mercado. Muchas de las empresas se nutren de estas herramientas, incluso personas autodidactas como es mi caso.

La principal motivación en este proyecto ha sido participar en el desarrollo de un videojuego como programador y así poder poner a prueba las capacidades a partir de lo que he aprendido a lo largo de la universidad y por mi cuenta.

3. Originalidad de la idea

En la industria del videojuego las ideas originales que dan lugar a nuevos juegos no son excesivas, y en la mayoría de los casos se lanzan al mercado productos que consisten en la mejora, revisión o continuación de productos e ideas ya existentes. Por esto, este proyecto consiste en utilizar un producto ya existente (juegos de estrategia tipo *tower defense*) y darle un giro que no aparece (o aparece muy poco) en las listas de aplicaciones de la "Play Store". Esto supone un extra de motivación, ya que puede tener un gran éxito si consigue llegar al público.

4. Estado del arte

Durante estas últimas décadas, el desarrollo y éxito de los videojuegos ha crecido prácticamente de forma exponencial. Tienen su origen sobre el año 1950, donde comenzaron a aparecer las primeras computadoras electrónicas tras el fin de la Segunda Guerra Mundial, se llevaron a cabo los primeros intentos por implementar programas de carácter lúdico. Algunos de estos videojuegos fueron el *Tennis for Two* (1958) o el *Spacewar* (1971), auténticos pioneros del género. Todos ellos eran todavía prototipos, juegos muy simples y de carácter experimental que no llegaron a comercializarse, entre otras cosas porque funcionaban en unas máquinas que solo estaban disponibles en universidades o en institutos de investigación.

No sería hasta la década de los 70 donde comenzaron a descender los costes de fabricación y a salir a la luz los primeros videojuegos dirigidos al público. Títulos como *Computer Space* (1971) o *Pong* (1972), de Atari, inauguraron las primeras máquinas recreativas construidas.

En los años 80, la Atari tuvo que compartir su dominio de la industria del videojuego con dos compañías llegadas de Japón: Nintendo y SEGA. Paralelamente, surgió una generación de ordenadores personales asequibles y con capacidades gráficas que llegaron a los hogares de millones de familias. A partir de entonces, los videojuegos empezaron a convertirse en una poderosa industria.

Los años 90 traen el salto a la tecnología de 16-bit, lo que significa importantes mejoras gráficas. En 1990 aparece un estilo de juego en el género de estrategia llamado *tower defense* con el título *Rampart*. Llegando al nuevo milenio, el género comenzó a aparecer en escenarios de *StarCraft, Age of Empires II* y *Warcraft III* creados por los usuarios.

Finalmente, desarrolladores de videojuegos independientes comenzaron a utilizar Adobe Flash para crear videojuegos *tower defense* que se pudieran jugar desde el mismo navegador web, dando lugar a videojuegos como el inmensamente popular *Desktop Tower Defense.*

5. Objetivos

El objetivo de este trabajo es el desarrollo de una aplicación de entretenimiento con elementos de networking para plataformas móviles utilizando la ayuda del motor de videojuegos Unity 3D y los servicios de Google Play. Además, se pretenden añadir otras características como sistema de micropagos, tabla de clasificaciones y sistema de logros.

Para la realización de este proyecto, se ha redactado un documento de diseño (GDD). Aquí se describe el diseño de interfaz, escenarios, objetos, así como la funcionalidad de los mismos. Consta como anexo al proyecto.

Por otro lado, también es de vital importancia para mí, el proceso de aprendizaje para la realización del proyecto, ya que toda la experiencia adquirida relacionada con éste ha sido de forma autodidacta y que me será de gran ayuda de cara al futuro.

6. Metodología

Se realizará el siguiente esquema de trabajo:

- Diseño del videojuego mediante GDD.
- Creación del diagrama de Gantt para planificar el proyecto.
- Búsqueda de recursos audiovisuales.
    - Boceto de los escenarios.
    - Boceto de los menús.
    - Obtención de recursos mediante el Asset Store.
- Programación del videojuego.
    - Programación de menús e interfaces.
        - Pantalla de introducción.
        - Pantalla de carga del menú principal.
        - Menú principal.
        - Interfaz del juego.
    - Pantalla de carga de la partida
    - Programación del entorno multijugador con los servicios Google Play Services.
        - Inicio sesión.
        - Creación de partida multijugador.
    - Programación de la mecánica de las partidas.
- Testear y ajustar la mecánica del juego para que sea equitativo.
- Testeo en varios dispositivos móviles.
- Creación de la documentación.

HERRAMIENTAS DE DESARROLLO

1. Unity 3D

1.1 Introducción

Unity es un motor de videojuego multiplataforma creado por Unity Technologies. Está disponible como plataforma de desarrollo para Microsoft Windows, OS X y Linux y permite crear juegos para diversas plataformas del mercado.

A partir de su versión 5.4.0 ya no soporta el desarrollo de contenido para navegador a través de su plugin web, en su lugar utiliza WebGL.

Unity tiene dos versiones: Unity Professional (pro) y Unity Personal.

1.2 Historia

La empresa Unity Technologies fue fundada en 2004 por David Helgason (CEO), Nicholas Francis (CCO), y Joachim Ante (CTO) en Copenhague, Dinamarca después de su primer juego, GooBall, que no obtuvo éxito.

Ilustración 2: Juego GooBall

Los tres reconocieron el valor del motor y las herramientas de desarrollo y se dispuso a crear un motor que cualquiera pudiera usar a un precio accesible.

El éxito de Unity ha llegado en parte debido al enfoque en las necesidades de los desarrolladores independientes que no pueden crear ni su propio motor del juego ni las herramientas necesarias o adquirir licencias para utilizar plenamente las opciones que aparecen disponibles. El enfoque de la compañía es “democratizar el desarrollo de juegos”, y hacer el desarrollo de contenidos interactivos en 2D y 3D lo más accesible posible.

La primera versión de Unity se lanzó en la Conferencia Mundial de Desarrolladores de Apple en 2005. Fue construido exclusivamente para funcionar y generar proyectos en los equipos de la plataforma Mac y obtuvo el éxito suficiente como para continuar con el desarrollo del motor y herramientas. Unity 3 fue lanzado en septiembre de 2010 y se centró en empezar a introducir más

herramientas que los estudios de alta gama por lo general tienen a su disposición, con el fin de captar el interés de los desarrolladores más grandes, mientras que proporciona herramientas para equipos independientes y más pequeñas que normalmente serían difíciles de conseguir en un paquete asequible. La última versión de Unity, Unity 5, lanzada a principios de 2015, se anunció en Game Developers e incluye añadidos como *Mecanim animation*, soporte para DirectX 11 y soporte para juegos en Linux y arreglo de *bugs* y texturas. Desarrollado por creadores de juegos para mayor expectativa.

1.3 ¿Por qué Unity 3D?

La elección de un motor de juego a otro puede influenciar mucho a la hora de decidirse y decantarse por una plataforma u otra, puede que se tenga pensado desarrollar un juego para una plataforma en concreto y por ello la elección de un motor de juego puede ser decisiva e importante.

En este caso, se ha optado por elegir Unity 3D por la simplicidad de poder exportar el proyecto a una plataforma u otra de manera muy rápida y sencilla, ya que se puede estar desarrollando un videojuego para PC y después querer adaptarlo y exportarlo para dispositivos móviles como Android o IOS fácilmente, solo con realizar pocos cambios en el diseño y la programación.

2. El complemento de Google Play Services para Unity

Google Play Services es un proyecto de código abierto cuyo objetivo es proporcionar un complemento que permita a los desarrolladores de juegos integrarse con la API de Google Play Games a partir de un juego hecho en Unity. Sin embargo, este proyecto no está en modo alguno aprobado o supervisado por Unity.

Este complemento permite acceder a la API de Google Play Games a través de la interfaz social de Unity. Proporciona soporte para las siguientes características de Google Play Games API:

- Registrarse.
- Desbloquear/revelar/incrementar logros.
- Puntuación del puesto en tablas de clasificación.
- Guardar en la nube leer/escribir.
- Interfaces visuales de Logros/Tabla de clasificación.
- Eventos y misiones.
- Conexiones cercanas.
- Multijugador por turnos.
- Multijugador en tiempo real.

Los desarrolladores de aplicaciones pueden utilizar Google Play Game Services para permitir una experiencia más competitiva y social mediante el uso de tablas de clasificación, tanto públicas como entre amigos, logros y sesiones multijugador. La API también dispone para guardar sincronizaciones rápidas de juegos en la nube de Google.

La plataforma Google+ proporciona inicio de sesión único, lo que permite que el usuario se autentifique automáticamente en aplicaciones que ofrezcan una experiencia más personalizada y que compartan opciones con Google+.

GOOGLE PLAY GAMES: REAL- TIME MULTIPLAYER

El multijugador en tiempo real de Google Play Games es una función que permite configurar juegos en tiempo real en Internet con hasta 4 jugadores. La plataforma se ocupa de la conexión y la infraestructura de red y expone una API que permite enviar mensajes de un jugador a otro. Por lo tanto, no se necesita implementar ningún código de conectividad de red de bajo nivel. Todo lo que se tiene que hacer es implementar una lógica de juego basada en intercambios de mensajes entre jugadores.

En la terminología multijugador en tiempo real de Google Play Games, un juego se produce en una *room*. Una r*oom* es un lugar virtual donde los jugadores se reúnen para jugar en tiempo real. Estas son las formas en las que se puede iniciar o unirse a una *room* multijugador:

- Partida rápida: Esto significa establecer o unirse a una sala con oponentes aleatorios (también llamado *automatch*). Al hacerlo, especifica el número mínimo y máximo de opositores con los que jugar, y la plataforma Google Play Games coloca automáticamente al usuario en una *room* con el número dado de oponentes anónimos.

- Pantalla de invitaciones: Crea una pantalla de invitación estándar de Google Play Games al usuario, donde podrá elegir con qué amigos quiere jugar. Esta pantalla también permite al jugador añadir oponentes mediante *automatch*, por lo que incluso pueden mezclar y combinar (por ejemplo, se puede elegir jugar con dos amigos específicos y un oponente aleatorio).

- Aceptar en la bandeja: Se mostrará una bandeja de entrada de la invitación al usuario, que es una pantalla estándar de Google Play Games que contiene todas las invitaciones pendientes que el usuario ha recibido. El usuario puede aceptar una de esas invitaciones para unirse a una *room.*

- Aceptar invitación: Acepta una invitación particular cuya ID sabes. Esto normalmente se hace en respuesta a la recepción de una invitación a una *room*.

Una vez conectado correctamente a una *room*, se pueden enviar y recibir mensajes a otros participantes. Estos mensajes son simplemente arrays de bytes que el programador es responsable de codificar y decodificar en un formato a su elección. También se deben manejar los eventos de conexión/desconexión apropiadamente para el juego (por ejemplo, si un compañero se desconecta de la habitación, el jugador debe desaparecer del juego). Cuando el juego termina, todos los jugadores deben salir de la habitación, momento en el que son libres de crear o unirse a una habitación diferente.

1. Autenticación para servicios de Google Play

Para poder utilizar todos los servicios que ofrece Google Play dentro de la aplicación el usuario debe obtener un *token* de autenticación ya que se va a acceder a un servicio el cual está ligado a una cuenta Google que contiene información personal.

La aplicación conecta con Google mediante los datos de autenticación del usuario, su usuario y contraseña, y solicita acceso a un servicio específico de Google.

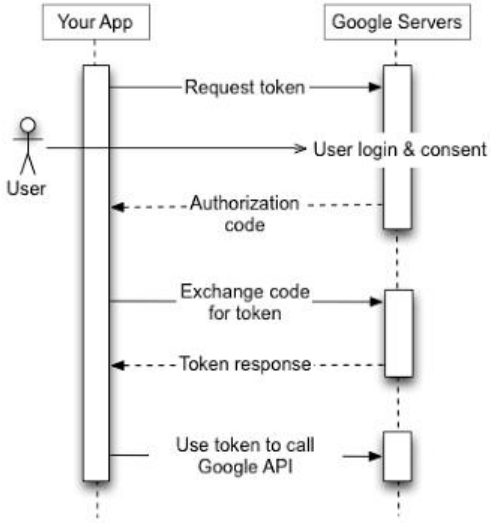

Ilustración 3: Diagrama de Autenticación

El primer paso de la aplicación es enviar la petición con el *token* y las credenciales del usuario para acceder al servicio Google. Una vez adquiridos, Google devuelve un código de autorización y el usuario intercambia el código de autentificación por el *token*.

Si todo se produce correctamente el usuario podrá utilizar ese *token* en cada petición que haga a los servicios de Google.

2. Creación/Unión a una *room*

Existen dos formas de crear o unirse a una *room* por invitación o búsqueda automática. En ambos casos se deben especificar los oponentes mínimos y máximos, además de la variante del juego (se pueden establecer varios códigos para diferenciar distintos estilos de juego como por ejemplo: cooperativo, captura la bandera, etc…).

```
public void StartMatchMaking()
{
    PlayGamesPlatform.Instance.RealTime.CreateQuickGame(minimumOpponents, maximumOpponents, gameVariation, this);
}

public void InviteToMatch()
{
    PlayGamesPlatform.Instance.RealTime.CreateWithInvitationScreen(minimumOpponents, maximumOpponents, gameVariation, this);
}
```

Ilustración 4: Código de creación/unión a una room

Para este juego se han inicializado la variable de *minimumOpponent* a 1, *maximumOpponents* a 1 y *gameVariation* a 0 (0 por defecto ya que solo existe un mapa).

3. Envío de mensajes

Para enviar un mensaje a todos los demás participantes, se utiliza la llamada al método *SendMessageToAll*. A éste método hay que pasarle una variable booleana que hace referencia al tipo de mensaje y un array de bytes de los datos que se desean enviar.

```
public void SendProgressBar(float progress)
{
    _updateProgressMessage.Clear();
    _updateProgressMessage.Add(_protocolVersion);
    _updateProgressMessage.Add((byte)'P');
    _updateProgressMessage.AddRange(System.BitConverter.GetBytes(++_myProgressMessageNum));
    _updateProgressMessage.AddRange(System.BitConverter.GetBytes(progress));
    byte[] messageToSend = _updateProgressMessage.ToArray();
    Debug.Log("MP - Send Progress Update-> " + "Progress: " + progress);
    PlayGamesPlatform.Instance.RealTime.SendMessageToAll(false, messageToSend);
}

public void SendCoreUpdate(float health)
{
    _updateCoreMessage.Clear();
    _updateCoreMessage.Add(_protocolVersion);
    _updateCoreMessage.Add((byte)'U');
    _updateCoreMessage.AddRange(System.BitConverter.GetBytes(health));
    byte[] messageToSend = _updateCoreMessage.ToArray();
    Debug.Log("MP - Send Core Update-> " + "Health: " + health);
    PlayGamesPlatform.Instance.RealTime.SendMessageToAll(true, messageToSend);
}

public void SendMinionCreated(int id)
{
    _minionCreatedMessage.Clear();
    _minionCreatedMessage.Add(_protocolVersion);
    _minionCreatedMessage.Add((byte)'C');
    _minionCreatedMessage.Add((byte)id);
    byte[] messageToSend = _minionCreatedMessage.ToArray();
    Debug.Log("MP - Send Minion Created-> " + "Id:" + id);
    PlayGamesPlatform.Instance.RealTime.SendMessageToAll(true, messageToSend);
}

public void SendMinionDestroyed(int id)
{
    _minionDestroyedMessage.Clear();
    _minionDestroyedMessage.Add(_protocolVersion);
    _minionDestroyedMessage.Add((byte)'E');
    _minionDestroyedMessage.Add((byte)id);
    byte[] messageToSend = _minionDestroyedMessage.ToArray();
    Debug.Log("MP - Send Minion Destroyed-> " + "Id:" + id);
    PlayGamesPlatform.Instance.RealTime.SendMessageToAll(false, messageToSend);
}

public void SendMinionDamageDealt(int id, float damage)
{
    _updateMinionDamageDealtMessage.Clear();
    _updateMinionDamageDealtMessage.Add(_protocolVersion);
    _updateMinionDamageDealtMessage.Add((byte)'D');
    _updateMinionDamageDealtMessage.Add((byte)id);
    _updateMinionDamageDealtMessage.AddRange(System.BitConverter.GetBytes(damage));
```

Ilustración 5: Código de envío de mensajes

Los mensajes pueden ser de dos tipos: *realiable* o *unreliable.*

Reliable: Son mensajes que utilizan el protocolo TCP/IP, con lo que garantizan la entrega al destinatario, y siempre llegan en orden.

Unreliable: Utilizan el protocolo UDP, con lo que no garantizan que lleguen al destinatario, y pueden no llegar en orden.

Normalmente los mensajes *unreliables* llegan antes que los *reliables*. Típicamente un juego debería enviar mensajes de tipo *unreliables* para actualizaciones que no afectan a la jugabilidad, y enviar importantes mensajes por vía de mensajes *reliables.*

4. Recepción de mensajes

Cuando se recibe un mensaje de otro participante, se debe llamar al método *OnRealTimeMessageReceived* para manejar toda la información.

```
public void OnRealTimeMessageReceived(bool isReliable, string senderId, byte[] data)
{

    byte messageVersion = (byte)data[0];

    char messageType = (char)data[1];
    Debug.Log("TIPO MENSAJE ENTRANTE: " + messageType);

    if (messageType == 'P' && data.Length == _updateProgressMessageLength)
    {
        int progressMessageNum = System.BitConverter.ToInt32(data, 2);
        float progress = System.BitConverter.ToSingle(data, 6);
        Debug.Log("Player " + senderId + " has " + "Progress: " + progress);
        if (updateProgressListener != null)
        {
            updateProgressListener.PorgressUpdateReceived(senderId, progressMessageNum, progress);
        }
    }
    else if (messageType == 'U' && data.Length == _updateCoreMessageLength)
    {
        float health = System.BitConverter.ToSingle(data, 2);
        Debug.Log("Player " + senderId + " has " + "Health: " + health);
        if (updateListener != null)
        {
            updateListener.CoreUpdateReceived(senderId, health);
        }
    }
    else if (messageType == 'C' && data.Length == _minionCreatedMessageLength)
    {
        int minionId = (int)data[2];

        Debug.Log("Id minion recibido: " + minionId);
        if (updateListener != null)
        {
            updateListener.MinionCreatedReceived(senderId, minionId);
        }
    }
    else if (messageType == 'E' && data.Length == _minionDestoyedMessageLength)
    {
        int minionId = (int)data[2];

        Debug.Log("Id minion recibido: " + minionId);
        if (updateListener != null)
        {
            updateListener.MinionDestroyedReceived(senderId, minionId);
        }
    }
    else if (messageType == 'D' && data.Length == _updateMinionDamageDealtLength)
    {
        int id = (int)data[2];
```

Ilustración 6: Código de recepción de mensajes

En cada mensaje se identifican varias partes:

- El primer byte es un número para controlar las versiones diferentes de los mensajes.

- El segundo byte es un carácter que identifica el tipo del mensaje.
- El resto del contenido de los mensajes son conversiones sobre los datos enviados.

5. Manejo del evento de desconexión

Cuando se finaliza una partida en tiempo real, se debe salir de la *room* llamando al método *LeaveRoom*:

```
public void LeaveGame()
{
    PlayGamesPlatform.Instance.RealTime.LeaveRoom();
}
```

Ilustración 7: Código para salir de una room

Éste método ejecuta una llamada al método *OnLeftRoom* que confirma que el jugador ha abandonado la *room*.

```
public void OnLeftRoom()
{
    ShowMPStatus("You have left the room.");
    if (updateListener != null)
    {
        updateListener.LeftRoomConfirmed();
    }
}
```

Ilustración 8: Código que confirma la salida de un jugador de la room

Una vez confirmado se llama a un método de la interfaz MPUpdateListener para que el GameController maneje esa confirmación y realice sus respectivas acciones del proceso de salida de una *room*.

6. Guardar/Cargar partidas en la nube de Google Play

El servicio de guardado permite seguir con el progreso con el que se terminó la anterior partida. Permite sincronizar los datos del juego de un jugador en varios dispositivos, por ejemplo, si tienes un juego que se ejecuta en Android, puedes usar este servicio para permitir que un jugador inicie un juego en su teléfono Android y luego continuar jugando en una tableta sin perder ningún progreso. Este servicio también se puede usar para asegurar que un jugador continúa desde donde lo dejó, incluso si su dispositivo se pierde, se destruye o se comercializa para un modelo más nuevo.

Este servicio de Google se ha diseñado para que a partir de una pantalla puedas guardar tu progreso manualmente, pero en este juego se ha utilizado una técnica para guardarlo automáticamente a partir de ciertas acciones en el juego.

```
//overwrites old file or saves a new one
public void SaveToCloud()
{
    if (MultiplayerController.Instance.IsAuthenticated())
    {
        Debug.Log("Saving progress to the cloud... filename: " + m_saveName);
        m_saving = true;

        PlayGamesPlatform.Instance.SavedGame.OpenWithAutomaticConflictResolution(
            m_saveName,
            DataSource.ReadCacheOrNetwork,
            ConflictResolutionStrategy.UseLongestPlaytime,
            SavedGameOpened);
    }
    else
    {
        Debug.Log("Not authenticated!");
    }
}
```

Ilustración 9: Código para ejecutar el proceso de guardado

Para guardar partidas primero se debe comprobar que se está conectado a los servicios de Google Play, luego se actualiza una variable booleana llamada *m_saving* a verdadero para indicar que se va a guardar (esta variable está creada porque guardar y cargar se encuentran en el mismo método, así que dependiendo del valor de *m_saving* guardará o cargará). Por último se llamará al método *OpenWitgAutomaticConflictResolution* pasándole 4 parámetros:

- Nombre del archivo.
- Fuente del archivo: Desde la cache o desde la red.
- Estrategia de resolución de conflicto: Utilizado para decidir si se quiere sobrescribir la partida guardada.
- Interfaz de la librería de Google Play Services.

Para cargar partidas se ha de llamar al mismo método pero cambiando la variable *m_saving* a falso, de esta manera se indica que se va a cargar. Una vez llamado al método Guardar/Cargar se ejecutará el método *SavedGameOpened*.

```
private void SavedGameOpened(SavedGameRequestStatus status, ISavedGameMetadata game)
{

    if (status == SavedGameRequestStatus.Success)
    {

        if (m_saving)
        {
            byte[] data = ToBytes();

            SavedGameMetadataUpdate.Builder builder = new SavedGameMetadataUpdate.Builder();
            builder = builder
                .WithUpdatedDescription("Saved game at " + DateTime.Now);
            SavedGameMetadataUpdate updatedMetadata = builder.Build();

            PlayGamesPlatform.Instance.SavedGame.CommitUpdate(game, updatedMetadata, data, SavedGameWritten);

        }
        else
        {
            PlayGamesPlatform.Instance.SavedGame.ReadBinaryData(game, SavedGameLoaded);
        }
    }
    else
    {
        Debug.LogWarning("Error opening game: " + status);
    }
}
```

Ilustración 10: Proceso para cargar/guardar una partida en la nube

Si la variable anterior *m_saving* es verdadera empezará el proceso de guardado, que consiste en convertir a un array de bytes toda la información que se quiera para posteriormente enviar una señal con esos datos mediante la llamada al método *CommitUpdate.*

```
private byte[] ToBytes()
{
    listToSave.Clear();
    listToSave.Add((byte)accountStats.hasPlayedValue);
    listToSave.Add((byte)accountStats.levelValue);
    listToSave.AddRange(System.BitConverter.GetBytes(accountStats.maxExpValue));
    listToSave.AddRange(System.BitConverter.GetBytes(accountStats.expValue));
    listToSave.AddRange(System.BitConverter.GetBytes(accountStats.gemsValue));
    listToSave.AddRange(System.BitConverter.GetBytes(accountStats.coinsValue));
    byte[] bytes = listToSave.ToArray();
    return bytes;
}
```

Ilustración 11: Conversión de datos a un array de bytes

Si la variable es falsa simplemente leerá los datos llamando al método *ReadBinary* los cuales se deberán pasar de bytes al tipo de variable adecuado.

```
private void FromBytes(byte[] bytes)
{
    int hasPlayed = (int)bytes[0];
    int level = (int)bytes[1];
    float maxExp = System.BitConverter.ToSingle(bytes, 2);
    float exp = System.BitConverter.ToSingle(bytes, 6);
    int gems = System.BitConverter.ToInt32(bytes, 10);
    int coins = System.BitConverter.ToInt32(bytes, 14);

    MergeWith(hasPlayed, level, maxExp, exp, gems, coins);
}
```

Ilustración 12: Conversión de bytes a datos

ANÁLISIS DEL PROBLEMA

1. Análisis de requisitos

La aplicación resultante tiene que satisfacer ciertos requisitos concretos al final de la misma. Estos determinarán la solución finalmente elegida para la implementación del proyecto y ellos también dependerán las conclusiones y resultados que confirmen hasta donde ha llegado el trabajo.

2. Alcance

Los requisitos especificados están dirigidos a usuarios de la aplicación Defend The Barricade. Tiene como objetivo ser un juego de estrategia en multijugador con características de Google Play Services para dar un cierto grado de competitividad.

3. Requisitos funcionales

Este apartado recoge los requisitos que afectan directamente a la funcionalidad principal de la aplicación.

Identificación del requisito	**RF01**
Nombre del requisito	Carga del menú principal
Características	• *Splash Screen.* • Inicio sesión usuarios. • Carga de datos de la última sesión.
Descripción del requisito	Se iniciará la aplicación con un logo identificativo. El sistema debe iniciar sesión de los usuarios y cargar los datos de su última partida (si ya han jugado anteriormente) automáticamente al abrir el juego.

Tabla 1: Requisito funcional 1

Identificación del requisito	**RF02**
Nombre del requisito	Visualización del menú principal

Características	• Botones del menú:
	  ○ Adquirir gemas por dinero real.
	  ○ Logros.
	  ○ Tabla de clasificaciones.
	  ○ Jugar.
	  ○ Invitaciones a amigos.
	  ○ Ajustes.
	• Panel confirmación para salir de la aplicación.

Descripción del requisito	La aplicación deberá tener en funcionamiento los botones citados anteriormente. Y se deberá pedir confirmación si se quiere salir de la aplicación.

Tabla 2: Requisito funcional 2

Identificación del requisito	**RF03**
Nombre del requisito	Acceso a la tienda gemas
Características	• Objetos para vender con sus respectivos precios.
Descripción del requisito	La aplicación ha de ofrecer una serie de productos al usuario para su posible compra.

Tabla 3: Requisito funcional 3

Identificación del requisito	**RF04**
Nombre del requisito	Consulta de logros
Características	• Visualización de la pantalla de logros por defecto.
Descripción del requisito	La aplicación deberá ofrecer una opción donde se puedan visualizar los logros.

Tabla 4: Requisito funcional 4

Identificación del requisito	**RF05**
Nombre del requisito	Configuración de la aplicación
Características	• Botón conectar/desconectar de Google Play Services. • Audio si/no.
Descripción del requisito	La aplicación deberá ofrecer la posibilidad de conectarse o desconectarse de Google Play Services, también deberá ofrecer la posibilidad de activar o desactivar el sonido.

Tabla 5: Requisito funcional 5

Identificación del requisito	**RF06**
Nombre del requisito	Envío de invitaciones a amigos
Características	• Enviar invitación a un amigo seleccionado
Descripción del requisito	La aplicación debe ofrecer la posibilidad invitar a un amigo a jugar desde un listado de amigos.

Tabla 6: Requisito funcional 6

Identificación del requisito	**RF07**
Nombre del requisito	Consulta la tabla de clasificaciones
Características	• Visualización de la tabla de clasificación por defecto.
Descripción del requisito	La aplicación deberá ofrecer una opción donde se pueda visualizar la tabla de clasificaciones.

Tabla 7: Requisito funcional 7

Identificación del requisito	**RF08**
Nombre del requisito	Sincronización del inicio de las partidas
Características	• Sincronización del comienzo de la partida entre jugadores.
Descripción del requisito	El sistema deberá enviar de cada jugador el progreso de la carga (ejecutado en segundo plano) de su partida para sincronizar el inicio.

Tabla 8: Requisito funcional 8

Identificación del requisito	**RF09**
Nombre del requisito	Configuración de las partidas
Características	• Núcleos • Oleadas de súbditos • Nodos para las torres • Torres • Interfaz de juego
Descripción del requisito	El sistema debe ser capaz de crear los núcleos (el del jugador y el oponente), crear los nodos propios para poder crear torres más adelante. También, debe ser capaz de crear oleadas de súbditos (del jugador y del oponente). Por último, el sistema debe de ser capaz de crear torres en el momento que el jugador crea más oportuno.

Tabla 9: Requisito funcional 9

Identificación del requisito	**RF10**
Nombre del requisito	Visualización de las recompensas
Características	• Título victoria/derrota • Textos de recompensa
Descripción del requisito	La aplicación deberá ofrecer una pantalla donde se podrán visualizar las recompensas después de cada partida.

Tabla 10: Requisito funcional 10

DISEÑO E IMPLEMENTACIÓN

Después de haber realizado un análisis de los requisitos y establecer las funcionalidades de la aplicación, se debe definir un diseño de la misma dentro del ciclo de desarrollo de software. Para ello se describirá de manera detallada un diseño de la arquitectura de la aplicación.

1. Diseño de la arquitectura

En este apartado se analizará la arquitectura que debe toma la aplicación así como un estudio de los componentes que la formarán. Esto permitirá trazar los requisitos descritos en la fase de análisis de tal manera que se pueda verificar que la implementación realizada cumple con todas las funcionales previamente descritas.

La aplicación utiliza una arquitectura basada en componentes. Esta arquitectura se enfoca en la descomposición del diseño en componentes funcionales o lógicos que expongan interfaces de comunicación bien definidas. Esto permite una mayor transparencia y depuración así como una mayor reutilización de componentes y facilidad de desarrollo.

La siguiente figura se corresponde con el diagrama de componentes de la aplicación. En él se representan los principales componentes del sistema así como las relaciones entre ellos.

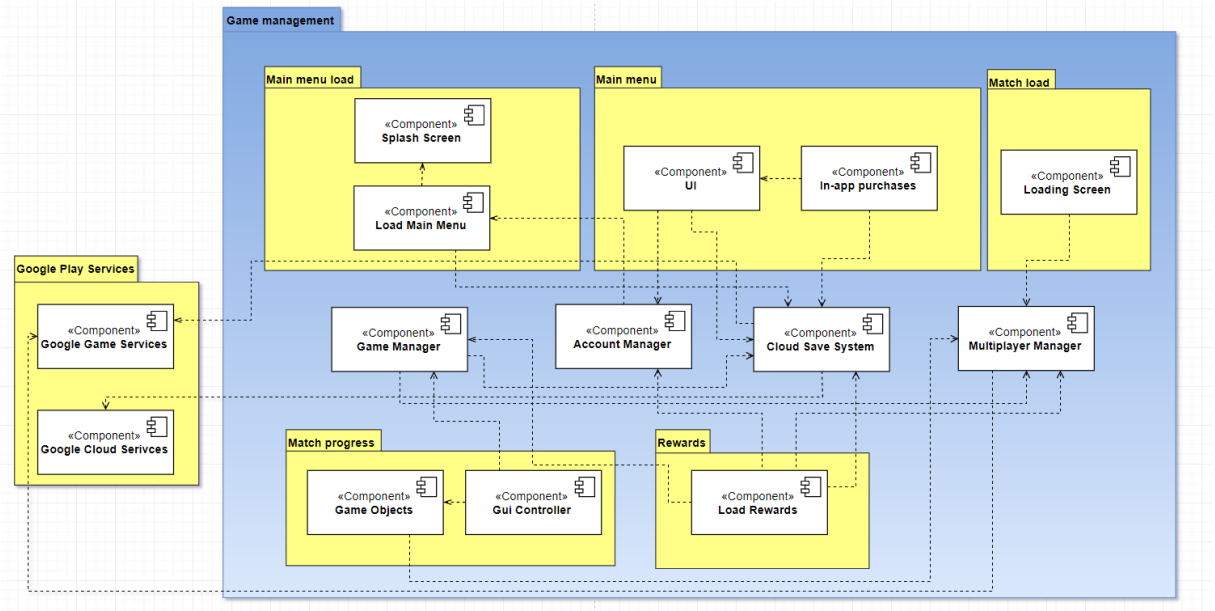

Ilustración 13: Diagrama de componentes

Identificación del componente	**COM01**
Nombre del componente	Google Play Game Services
Acciones	• Accede a la API de Google Game Services
Dependencias	**N/A**
Requisitos	RF01, RF02, RF03, RF04, RF05, RF06, RF07, RF08, RF09, RF10

Tabla 11: Componente Google Play Game Services

Identificación del componente	**COM02**
Nombre del componente	Google Cloud Services
Acciones	• Accede a la API de Google Drive
Dependencias	**N/A**
Requisitos	RF01, RF02, RF03, RF10

Tabla 12: Componente Google Cloud Services

Identificación del componente	**COM03**
Nombre del componente	Load Main Menu
Acciones	• Conecta con Google Play Services • Cargar y Guarda en datos de la nube
Dependencias	**COM09**
Requisitos	RF01

Tabla 13: Componente Load Main Menu

Identificación del componente	**COM04**
Nombre del componente	Main menu UI
Acciones	• Visualizar tabla de clasificaciones • Visualizar tienda de gemas • Visualizar logros • Buscar partida • Invitar amigos • Ajustes
Dependencias	**COM08, COM09**
Requisitos	RF02, RF03, RF04, RF05, RF06, RF07

Tabla 14: Componente Main menu UI

Identificación del componente	**COM05**
Nombre del componente	In-app Purchases
Acciones	• Compra productos por dinero real
Dependencias	**COM04, COM09**
Requisitos	RF02, RF03

Tabla 15: Componente In-app Purchases

Identificación del componente	**COM06**
Nombre del componente	Loading Screen
Acciones	• Sincroniza la carga inicial de las partidas
Dependencias	**COM10**
Requisitos	RF08

Tabla 16: Componente Loading Screen

Identificación del componente	**COM07**
Nombre del componente	Game Manager
Acciones	• Controlador de partidas
Dependencias	**COM09, COM10, COM11, COM12**
Requisitos	RF9

Tabla 17: Componente Game Manager

Identificación del componente	**COM08**
Nombre del componente	Account Manager
Acciones	• Controlador de datos de la cuenta activa
Dependencias	**COM03**
Requisitos	RF01, RF02, RF03, RF10

Tabla 18: Componente Account Manager

Identificación del componente	**COM09**
Nombre del componente	Cloud Save System
Acciones	• Guarda y carga datos en la nube
Dependencias	**COM01, COM02**
Requisitos	RF01, RF02, RF03, RF10

Tabla 19: Componente Save System

Identificación del componente	**COM10**
Nombre del componente	Multiplayer Manager
Acciones	• Envío de mensajes entre jugadores • *Matchmaking*
Dependencias	**COM01**
Requisitos	RF06, RF08, RF09

Tabla 20: Componente Multiplayer Manager

Identificación del componente	**COM11**
Nombre del componente	Game Objects
Acciones	• Envío de mensajes de minions • Envío de mensajes de núcleos • Envío de mensajes del estado de la partida (vida de cada jugador)
Dependencias	COM07, COM10
Requisitos	RF09, RF10

Tabla 21: Componente Game Objects

Identificación del componente	**COM12**
Nombre del componente	Gui Controller
Acciones	• Controla todos los atributos de los jugadores en la partida
Dependencias	COM07, COM11
Requisitos	RF09

Tabla 22: Componente Gui Controller

Identificación del componente	**COM13**
Nombre del componente	Load Rewards
Acciones	• Visualiza las recompensas obtenidas al finalizar una partida
Dependencias	COM07, COM08, COM09
Requisitos	RF10

Tabla 23: Componente Load Rewards

2. Implementación

En este apartado se van a comentar los aspectos más relevantes respecto a la implementación de la aplicación, así como las consideraciones tomadas durante el desarrollo de la misma. No se pretende crear un manual de desarrollo extenso, por lo que aspectos básicos de implantación relacionados con Unity como puede ser activar animaciones, llamadas a métodos de la librería de Unity u otros temas triviales de esta plataforma pueden encontrarse en los tutoriales ofrecidos por la página oficial de Unity.

Para exponer estos puntos se tomará como referencia el diagrama de componentes obtenido durante la fase de diseño, se dividirá la sección en 4 partes.

2.1 Carga del menú principal

Al inicializar la aplicación se ejecutará en primer lugar el *Splash Screen* donde simplemente se observará un logo con un efecto visual llamado *fade*.

```
public class SplashScreen : MonoBehaviour
{
    public Image splashImage;
    public string sceneName;

    IEnumerator Start()
    {
        splashImage.canvasRenderer.SetAlpha(0.0f);

        FadeIn();

        yield return new WaitForSeconds(2.5f);

        FadeOut();

        yield return new WaitForSeconds(1.5f);

        SceneManager.LoadScene(sceneName);
    }

    void FadeIn()
    {
        splashImage.CrossFadeAlpha(1.0f, 1.5f, false);
    }

    void FadeOut()
    {
        splashImage.CrossFadeAlpha(0.0f, 1.5f, false);
    }
}
```

Ilustración 14: Código utilizado para dar efecto fade a la splash screen

El efecto *fade* aumentará el color *alpha* de 0 a 1 y al cabo de 2,5 segundos volverá a ser 0, es decir, aparecerá y desaparecerá. Después se iniciará la pantalla de carga del menú principal.

En la pantalla de carga se realizarán una serie de procesos antes de ir al menú principal.

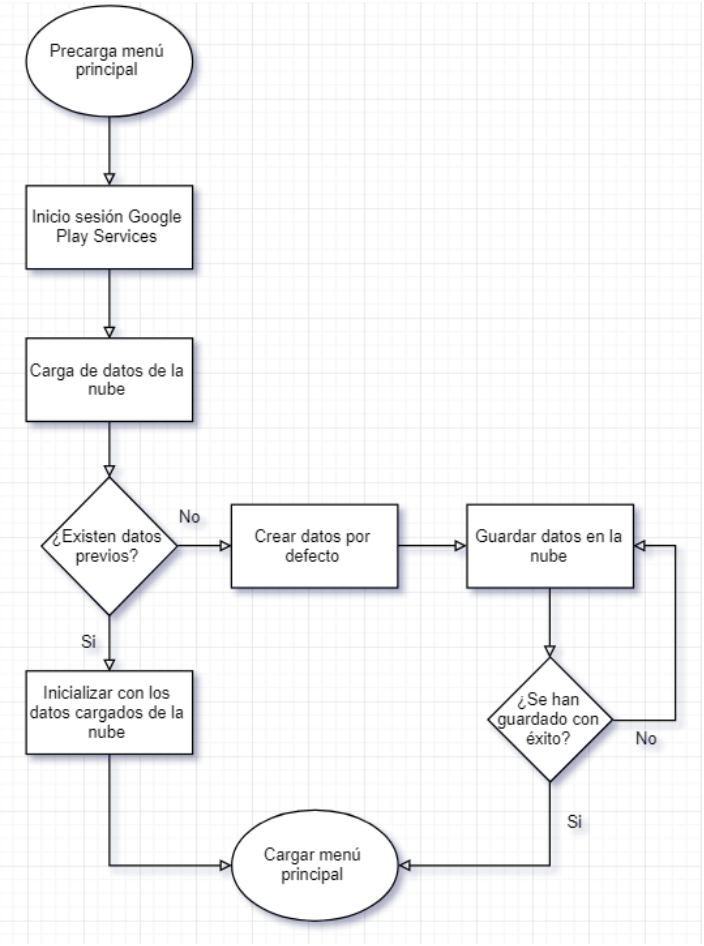

Ilustración 15: Diagrama de flujo de carga del menú principal

En primer lugar se iniciará sesión en los servicios de Google Play, una vez conectados cargará la última partida. En este punto puede haber dos opciones: que sea la primera vez que juega o que ya se haya jugado. Si ya se ha jugado anteriormente se cargarán los datos de la nube y luego se iniciará el menú principal, en cambio, si no se ha jugado nunca, se crearán unos datos por defecto y luego se guardarán en la nube, todo seguido se iniciará el menú principal.

El código y la explicación del proceso de guardar/cargar datos se puede encontrar en el punto 6 Guardar/Cargar partidas en la nube de Google Play del capítulo Google Play Games: Real-time multiplayer.

2.2 Menú principal

En la pantalla del menú principal se pueden encontrar diversas opciones las cuales se van a describir a continuación.

2.2.1 Logros

La opción de la pantalla de logros es una característica de Google Play Services donde simplemente llamando al método *ShowAchievementsUI* se mostrará la pantalla por defecto de los logros.

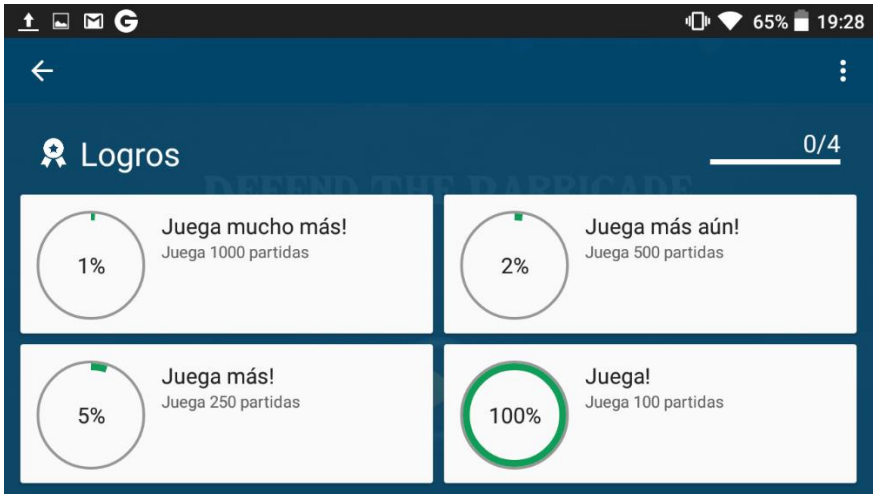

Ilustración 16: Pantalla de logros

Existen dos tipos de logros: logros incrementales y no incrementales. Los logros incrementales se desbloquean al completar una cierta cantidad de pasos (más de 1 paso) y los no incrementales se completan al momento, es decir, completando un solo paso.

Para desbloquear un logro incremental se debe llamar al método *IncrementAchievement* donde se tiene que indicar la referencia al logro y la cantidad que se desea incrementar.

Para desbloquear un logro que no es incremental se debe llamar al método *ReportProgress* donde se tiene que indicar la referencia al logro y el valor "100.0f".

```
public void Logro_JuegaI()
{
    PlayGamesPlatform.Instance.IncrementAchievement("CgkI5Oiv9_4REAIQAQ", 1, (bool success) =>
    {
    });
}

public void Logro_JuegaII()
{
    PlayGamesPlatform.Instance.IncrementAchievement("CgkI5Oiv9_4REAIQAg", 1, (bool success) =>
    {
    });
}

public void Logro_JuegaIII()
{
    PlayGamesPlatform.Instance.IncrementAchievement("CgkI5Oiv9_4REAIQAw", 1, (bool success) =>
    {
    });
}

public void Logro_JuegaIV()
{
    PlayGamesPlatform.Instance.IncrementAchievement("CgkI5Oiv9_4REAIQBA", 1, (bool success) =>
    {
    });
}
```

Ilustración 17: Fragmento de código para desbloquear logros

Estos logos se desbloquean jugando partidas.

2.2.2 Tabla de clasificaciones

La opción de la pantalla de logros es una característica de Google Play Services donde simplemente llamando al método *ShowLeaderboardUI* se mostrará la pantalla por defecto de los logros.

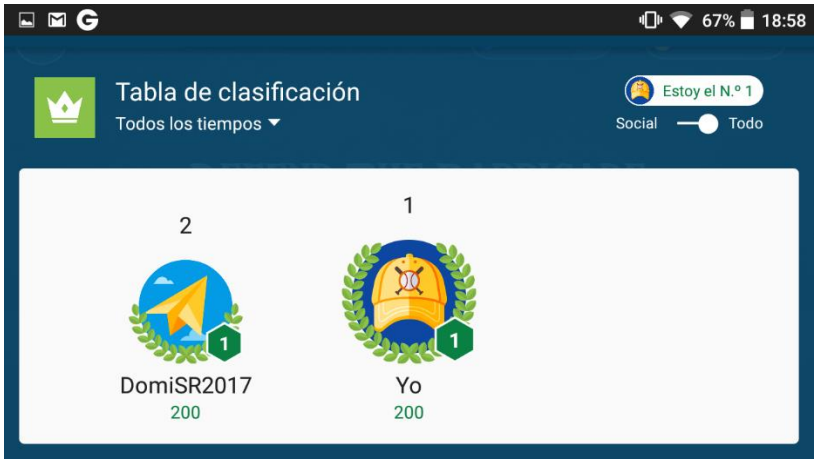

Ilustración 18: Pantalla de la tabla de clasificación

Para aumentar o disminuir la puntuación en la tabla de clasificaciones se debe llamar al método *ReportScore* e indicar los puntos que se desean aumentar o disminuir, y la referencia a la tabla.

```
public void GainScore()
{
    PlayGamesPlatform.Instance.ReportScore(200, "CgkI5Oiv9_4REAIQCQ", (bool success) => {
    });
}

public void LoseScore()
{
    PlayGamesPlatform.Instance.ReportScore(-100, "CgkI5Oiv9_4REAIQCQ", (bool success) => {
    });
}
```

Ilustración 19: Fragmento de código para aumentar/disminuir puntuación

El primer método aumenta 200 puntos en la clasificación, mientras que el segundo disminuye 100 puntos.

2.2.3 Jugar

Al pulsar el botón de jugar se activa una búsqueda de partida automáticamente llamada *automatch*. El sistema busca otros jugadores que también estén buscando una partida lo cual activa el método *OnRoomSetupProgress* donde se puede visualizar el progreso de la búsqueda y personalizar el *feedback* que se le da al jugador para que no se desespere. El *feedback* que se da en esta aplicación consiste en visualizar el tiempo que lleva un jugador buscando una partida.

Ilustración 20: Tiempo de búsqueda de una partida

Una vez encontrado un oponente, el sistema llamará al método OnRoomConnecte. Una vez iniciado este método significa que el jugador se encuentra en una *room* y que va a empezar en breve la partida.

```
public void OnRoomConnected(bool success)
{
    if (success)
    {
        ShowMPStatus("You're connected to the room! You would probably start the game now.");
        lobbyListener.HideLobby();
        lobbyListener = null;
        SceneManager.LoadScene("LoadingScreen");
        _myProgressMessageNum = 0;
    }
    else
    {
        ShowMPStatus("Uh-oh. Encountered some error connecting to the room.");
        lobbyListener.HideLobby();
    }
}
```

Ilustración 21: Fragmento de código de conexión a una room

La siguiente pantalla será la pantalla de sincronización de la partida (explicado más adelante).

2.2.4 Invitaciones

Esta opción permite invitar a algún amigo a una partida. Llamando al método *CreateWithInvitationScreen* (explicado en el punto 6 Guardar/Cargar partidas en la nube de Google Play del capítulo Google Play Games: Real-time Multiplayer) el jugador puede invitar a un amigo solamente, ya que este juego es de uno contra uno.

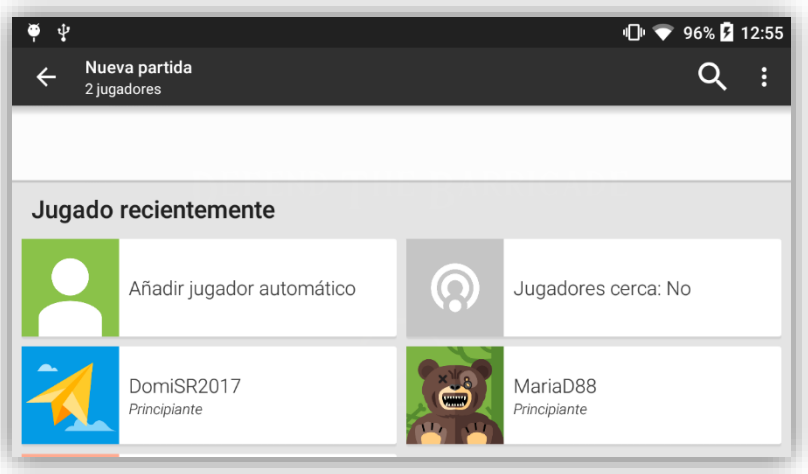

Ilustración 22: Pantalla de invitaciones

Cuando un jugador es invitado a una partida le debe aparecer un mensaje como que otro jugador le está invitando a jugar al juego, por lo que tendrá dos opciones posibles al aparecer el mensaje de invitación, rechazar o aceptar. Al aceptar se unirá a automáticamente a la partida.

2.2.5 Ajustes

La pantalla de ajuste contiene diversas opciones:

- Conectar/Desconectar de los Servicios de Google Play.
- Activar/Desactivar sonido.

2.2.6 Tienda de gemas

La tienda de gemas es la única fuente de monetización de la aplicación. Esta opción permite la compra de varias cantidades de gemas donde cada cantidad tiene un precio diferente.

Ilustración 23: Pantalla de la tienda de gemas

El proceso para la compra de gemas está definido en el siguiente diagrama:

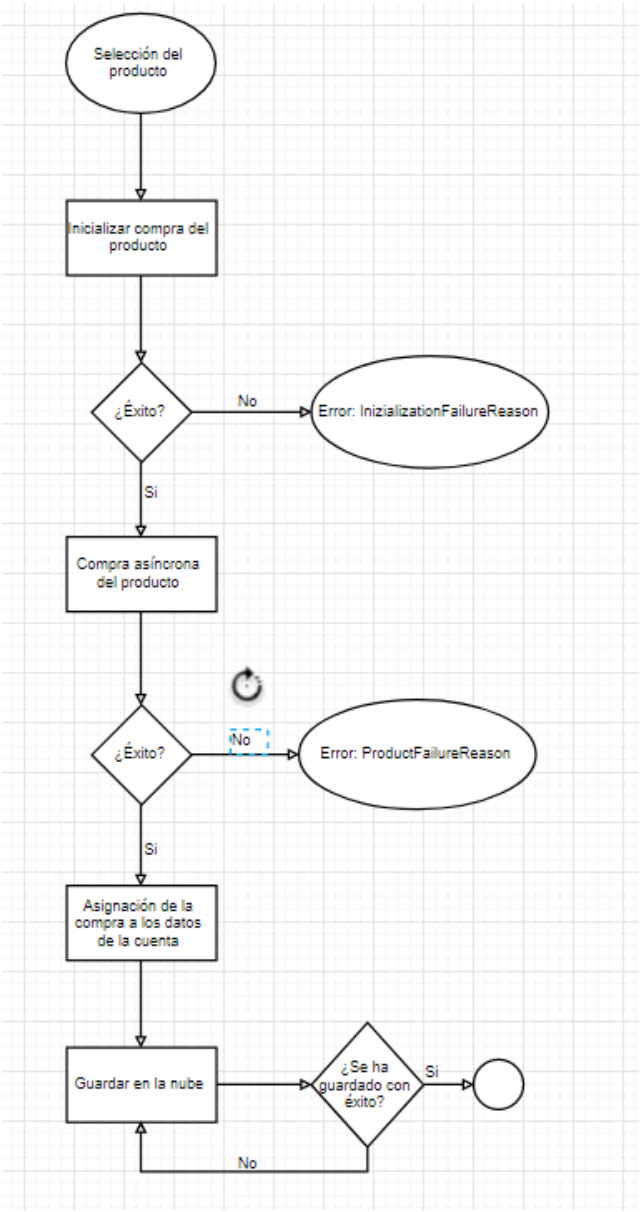

Ilustración 24: Diagrama de flujo de la compra de gemas

En primer lugar se inicializa la sesión de compra del producto, si falla no continuará y saldrá un mensaje de error, si se inicializa correctamente, se realizará la compra del producto de manera asíncrona. Si todo ha ido correcto se aumentará la cantidad de gemas compradas a la cantidad anterior y todo seguido se guardará los datos en la nube para no perder el progreso.

2.3 Carga de la partida

Esta pantalla se ha diseñado con el propósito de sincronizar la partida de los dos jugadores, ya que en las pruebas realizadas había momentos en los que un jugador entraba antes a la partida que el otro,

este proceso soluciona ese error ya que hasta que ambos jugadores no terminan de cargar su partida no pueden entrar a la partida.

```
void SetupScene()
{
    MultiplayerController.Instance.updateProgressListener = this;

    _myParticipantId = MultiplayerController.Instance.GetMyParticipantId();

    allPlayers = MultiplayerController.Instance.GetAllPlayers();

    for (int i = 0; i < allPlayers.Count; i++)
    {
        string nextParticipantId = allPlayers[i].ParticipantId;

        if (nextParticipantId == _myParticipantId)
        {
            mainPlayerSide = i;

            if (mainPlayerSide == 0)
            {
                leftPlayerName.text = allPlayers[i].DisplayName;

            }
            else
            {
                rightPlayerName.text = allPlayers[i].DisplayName;

            }

        }
        else
        {
            opponentPlayerSide = i;
            if (opponentPlayerSide == 0)
            {
                leftPlayerName.text = allPlayers[i].DisplayName;

            }
            else
            {
                rightPlayerName.text = allPlayers[i].DisplayName;

            }
        }
    }
}
```

Ilustración 25: Código que realiza la configuración inicial de la pantalla de carga

El código simplemente prepara la pantalla, sitúa a cada lado un jugador dependiendo del orden de entrada.

Todo seguido se inicia una *Coroutine*, donde cada jugador enviará su progreso y por lo que se sincronizará la partida.

```
IEnumerator LoadLevenAsync()
{
    yield return new WaitForSeconds(2f);

    async = SceneManager.LoadSceneAsync(sceneName);
    async.allowSceneActivation = false;

    while (!async.isDone)
    {
        if (mainPlayerSide == 0)
        {
            float progress = Mathf.Clamp01(async.progress / 0.9f);

            leftProgBar.value = progress;
            MultiplayerController.Instance.SendProgressBar(leftProgBar.value);
        }
        else
        {
            float progress = Mathf.Clamp01(async.progress / 0.9f);

            rightProgBar.value = progress;
            MultiplayerController.Instance.SendProgressBar(rightProgBar.value);
        }

        yield return null;
    }
}
```

Ilustración 26: Fragmento de código para visualizar el proceso de carga

Unity tiene un pequeño error al cargar una pantalla en segundo plano, por lo que solo llega al 90% de carga. El siguiente 10% inicia la escena, por eso se utiliza la operación *Mathf.Clamp01*, para que el total que se visualice en la pantalla sea del 90%.

Cada jugador recibirá el progreso del otro mediante el método *ProgressUpdateReceived*. Una vez los dos progresos lleguen al 90% de su carga se iniciará la partida.

```
void Update()
{
    if (rightProgBar.value >= 0.9f && leftProgBar.value >= 0.9f)
    {
        StartCoroutine("waitBeforeStart");
    }
}

IEnumerator waitBeforeStart()
{
    //leftProgBar.value = 1f;
    //rightProgBar.value = 1f;
    yield return new WaitForSeconds(3f);
    async.allowSceneActivation = true;
}

public void PorgressUpdateReceived(string participantID, int progressMessageNum, float progress)
{
    if (progressMessageNum <= _lastMessageNum)
    {
        return;
    }
    else
    {
        _lastMessageNum = progressMessageNum;

        Debug.Log("PROGRESSO: " + progress);
        if (opponentPlayerSide == 0)
        {
            leftProgBar.value = progress;
        }
        else
        {
            rightProgBar.value = progress;
        }
    }
}
```

Ilustración 27: Código que controla la recepción del progreso de carga del oponente

2.4 Proceso de una partida

Esta parte es la más importante ya que es la parte principal del juego.

La partida consiste en defender un castillo de las oleadas de súbditos del enemigo, para ello se permite la construcción de torres sobre unos nodos en concreto. Cada torre cuesta una cierta cantidad de dinero por lo que eliminar súbditos del oponente o hacer que los súbditos propios lleguen al núcleo enemigo ara que aumente el oro a medida que la partida avance (en el GDD (Game Design Document) de los anexos se explica en más detalle la mecánica).

Los componentes principales son:

Nombre del componente	**Main Core**
Descripción	Núcleo de cada jugador que controla la vida principal, las oleadas de los súbditos y los caminos que recorrerán los súbditos.
Scripts del componente	MainCoreController.cs WaveSpawner.cs
Funcionalidad de los script	• MainCoreController: Inicializa características propias de cada jugador como los caminos y los nodos. También realiza acciones como cambiar el rumbo de los súbditos. • WaveSpawner: Crea cada 15 segundos una oleada de 5 súbditos.

Tabla 24: Componente Main Core

Nombre del componente	**Opponent Core**
Descripción	Es un reflejo del núcleo del oponente por lo que únicamente éste lo puede controlar.
Scripts del componente	OpponentController.cs
Funcionalidad de los scripts	OpponentController únicamente sirve para poder visualizar el núcleo enemigo y su vida.

Tabla 25: Componente Opponent Core

Nombre del componente	**Main Minion**
Descripción	Súbdito del jugador y única fuente de daño para poder ganar la partida.
Scripts del componente	MainMinionController.cs
Funcionalidad de los scripts	MainMinionController sirve para controlar los estados de movimiento y destrucción del propio súbdito. En este script se controla su vida y cada segundo se envían mensajes sobre su posición para que el oponente vea reflejado en su pantalla el movimiento de éste.

Tabla 26: Componente Main Minion

Nombre del componente	**Opponent Minion**
Descripción	Es un reflejo del súbdito del oponente por lo que únicamente se puede controlar mediante la recepción de los mensajes que envía el **main minion** del otro jugador.
Scripts del componente	OpponentMinionController.cs
Funcionalidad de los scripts	Únicamente recibe datos sobre la posición del **main minion** del otro jugador.

Tabla 27: Componente Opponent Minion

Nombre del componente	**Standard Tower**
Descripción	Torre básica que dispara proyectiles.
Scripts del componente	Turret.cs
Funcionalidad de los scripts	Se utiliza para configurar el tipo de daño que realizará la torre. En este caso proyectiles.

Tabla 28: Componente Standard Tower

Nombre del componente	**Opponent Standard Tower**
Descripción	Torre creada en la pantalla del oponente cuando se crea una torre.
Scripts del componente	Turret.cs
Funcionalidad de los scripts	Se utiliza para configurar el tipo de daño que realizará la torre. Utiliza la misma configuración que la **Standard Tower.**

Tabla 29: Componente Opponent Standard Tower

Nombre del componente	**Tower Mage**
Descripción	Torre mágica que dispara un rayo que ralentiza y hace daño.
Scripts del componente	Turret.cs
Funcionalidad de los scripts	Se utiliza para configurar el tipo de daño que realizará la torre. En este caso un rayo.

Tabla 30: Componente Tower Mage

Nombre del componente	**Opponent Tower Mage**
Descripción	Torre creada en la pantalla del oponente cuando se crea una torre.
Scripts del componente	Turret.cs
Funcionalidad de los scripts	Se utiliza para configurar el tipo de daño que realizará la torre. Utiliza la misma configuración que la **Tower Mage.**

Tabla 31: Componente Opponent Tower Mage

Nombre del componente	**Main Bullet**
Descripción	Proyectil principal de las **Standard Tower**
Scripts del componente	Bullet.cs
Funcionalidad de los scripts	Traza la trayectoria de las balas.

Tabla 32: Componente Main Bullet

Nombre del componente	**Opponent Bullet**
Descripción	Proyectil que se crea en la pantalla del oponente.
Scripts del componente	Bullet.cs
Funcionalidad de los scripts	Traza la trayectoria de las balas.

Tabla 33: Componente Opponent Bullet

Nombre del componente	**Paths**
Descripción	Conjunto de puntos utilizados para trazar el camino de los súbditos.
Scripts del componente	N/A
Funcionalidad de los scripts	N/A

Tabla 34: Componente Paths

Nombre del componente	**Nodes**
Descripción	Bases de las torres.
Scripts del componente	Node.cs
Funcionalidad de los scripts	Se utiliza para comprobar si se puede construir una torre en el nodo.

Tabla 35: Componente Nodes

Nombre del componente	**Game Manager**
Descripción	Controlador principal de la partida donde se reciben la mayoría de mensajes de la red.
Scripts del componente	GameController.cs
Funcionalidad de los scripts	Configura el inicio de las partidas y maneja el flujo de mensajes de la red recibidos para poder distribuirlos de la manera más adecuada.

Tabla 36: Componente Game Manager

Nombre del componente	**Flag**
Descripción	Bandera que señala el camino actual por el que irán los próximos súbditos que salgan.
Scripts del componente	N/A
Funcionalidad de los scripts	N/A

Tabla 37: Componente Flag

Al empezar una partida se inicia un proceso de configuración del mapa, llevado a cabo por el *GameManager.*

```
void SetupMultiplayerGame()
{
    guiController = GetComponent<GUIController>();

    MultiplayerController.Instance.updateListener = this;

    _myParticipantId = MultiplayerController.Instance.GetMyParticipantId();

    allPlayers = MultiplayerController.Instance.GetAllPlayers();

    _finishMatch = new Dictionary<string, float>(allPlayers.Count);
    _playerSide = new Dictionary<string, float>(allPlayers.Count);
    _opponentScripts = new Dictionary<string, OpponentController>(allPlayers.Count - 1);
    _opponentMinionScripts = new Dictionary<string, OpponentMinionController>();

    for (int i = 0; i < allPlayers.Count; i++)
    {
        string nextParticipantId = allPlayers[i].ParticipantId;
        Debug.Log("SETTING UP PLAYER: " + allPlayers[i].DisplayName);

        _finishMatch[nextParticipantId] = -1;

        if (nextParticipantId == _myParticipantId)
        {
            mainCore = Instantiate(mainCorePrefab, Vector3.zero, Quaternion.identity);

            MainCoreController mainCoreController = mainCore.GetComponent<MainCoreController>();
            WaveSpawner waveSpawner = mainCore.GetComponent<WaveSpawner>();
            mainCoreController.SetId(i);
            mainCoreController.SetSide(i);

            _playerSide[nextParticipantId] = i;

            if (mainCoreController.GetSide() == left)
            {
                mainCore.transform.position = left_position;
                mainCore.transform.eulerAngles = new Vector3(0f, 209f, 0f);
                guiController.SetLeftHealth(25);
                waveSpawner.currentPath = waveSpawner.paths[1].leftMinionWayPoints;
            }
            else
            {
                mainCore.transform.position = right_position;
                mainCore.transform.eulerAngles = new Vector3(0f, 29f, 0f);
                guiController.SetRightName(allPlayers[i].DisplayName);
                guiController.SetRightHealth(25);
                waveSpawner.currentPath = waveSpawner.paths[0].rightMinionWayPoints;
            }
        }
        else
        {
            else
            {
                opponentCore = Instantiate(opponentCorePrefab, Vector3.zero, Quaternion.identity);

                OpponentController opponentScript = opponentCore.GetComponent<OpponentController>();
                opponentScript.SetSide(i);
                _opponentScripts[nextParticipantId] = opponentScript;
                _playerSide[nextParticipantId] = i;

                if (opponentScript.GetSide() == left)
                {
                    opponentCore.transform.position = left_position;
                    opponentCore.transform.eulerAngles = new Vector3(0f, 209f, 0f);
                    guiController.SetLeftName(allPlayers[i].DisplayName);
                    guiController.SetLeftHealth(25);
                }
                else
                {
                    opponentCore.transform.position = right_position;
                    opponentCore.transform.eulerAngles = new Vector3(0f, 29f, 0f);
                    guiController.SetRightName(allPlayers[i].DisplayName);
                    guiController.SetRightHealth(25);
                }
                Debug.Log("CORE OPPO: " + opponentCore.GetComponent<OpponentController>().GetSide() + " "
            }
        }
        Debug.Log("#### INIT STATS ####");
        Debug.Log("LEFT HEALTH: " + guiController.GetLeftHealth());
        Debug.Log("RIGHT HEALTH: " + guiController.GetRightHealth());
        Debug.Log("#### MY CORE ####");
        Debug.Log("ID: " + mainCore.GetComponent<MainCoreController>().GetId());
        Debug.Log("SIDE: " + mainCore.GetComponent<MainCoreController>().GetSide());
        Debug.Log("#### CORE OPPONET ####");
        Debug.Log("OPPONENT SIDE: " + opponentCore.GetComponent<OpponentController>().GetSide());

        guiController.SetGold(300);
        _multiplayerReady = true;
    }
```

Ilustración 28: Código de configuración inicial de la partida - GameManager

El proceso que aparece en las imágenes anteriores consiste en primer lugar en situar a cada jugador a un lado del mapa. Una vez que cada jugador esté situado se inician los valores de oro, vida y nombre de cada jugador.

A continuación, el núcleo de cada jugador creará automáticamente sus propios nodos, que más tarde le servirán para crear torres. Al mismo tiempo se crearán las rutas establecidas para los caminos.

```
void Start()
{
    if (_side == GameController.left)
    {
        Instantiate(pathFlag, buttonPaths.leftButtonPaths[1].transform.position, Quaternion.identity);
    }
    else
    {
        Instantiate(pathFlag, buttonPaths.rightButtonPaths[1].transform.position, Quaternion.identity);
    }


    if (_side == GameController.left)
    {
        for (int i = 0; i < leftNodes.Length; i++)
        {
            Instantiate(leftNodes[i]);
        }

        for (int i = 0; i < buttonPaths.leftButtonPaths.Length; i++)
        {
            Instantiate(buttonPaths.leftButtonPaths[i]);
        }
    }
    else
    {
        for (int i = 0; i < rightNodes.Length; i++)
        {
            Instantiate(rightNodes[i]);
        }

        for (int i = 0; i < buttonPaths.rightButtonPaths.Length; i++)
        {
            Instantiate(buttonPaths.rightButtonPaths[i]);
        }
    }
}
```

Ilustración 29: Código de configuración inicial del núcleo principal

Una vez terminada la configuración inicial empezará una cuenta atrás. Cuando llegue a cero saldrán oleadas de 5 súbditos cada 15 segundos.

```
IEnumerator SpawnWave()
{
    Wave wave = waves[waveIndex];

    for (int i = 0; i < wave.count; i++)
    {
        SpawnMinion(wave.enemy);
        yield return new WaitForSeconds(1f / wave.rate);
    }

    waveIndex++;

    if (waveIndex == waves.Length)
    {
        waveIndex = 0;
    }
}
```

Ilustración 30: Código para controlar las oleadas de súbditos

Al crear un súbdito se le asigna a éste un ID para poder diferenciarlo del resto, ya que cuando un súbdito envía un mensaje de su posición a través de la red se podría confundir con el mensaje de otro súbdito.

```
void SpawnMinion(GameObject minionPrefab)
{
    GameObject newMinion = null;

    if(mainCoreController.GetSide() == GameController.left)
    {
        newMinion = Instantiate(minionPrefab, spawnLeftPoint.transform.position, spawnLeftPoint.transform.rotation);
    }
    else
    {
        newMinion = Instantiate(minionPrefab, spawnRightPoint.transform.position, spawnRightPoint.transform.rotation);
    }

    newMinion.GetComponent<MainMinionController>().waypoints = currentPath;
    newMinion.GetComponent<MainMinionController>().SetId(++_minionId);

    MultiplayerController.Instance.SendMinionCreated(_minionId);

}
```

Ilustración 31: Código para crear súbditos

Cada súbdito sigue un camino que viene definido por el jugador. Por defecto los súbditos empiezan por el carril central, pero el jugador puede cambiar en cualquier momento la trayectoria del súbdito. Cabe mencionar que los súbditos solo podrán cambiar la trayectoria antes de crearse, es decir, si un súbdito empieza por un camino, este no va a poder cambiar ya su rumbo.

```
void Update() {

    if (gameController.IsGameOver)
    {
        return;
    }

    Vector3 dir = target.transform.position - transform.position;
    transform.Translate(dir.normalized * _speed * Time.deltaTime, Space.World);

    if (Vector3.Distance(transform.position, target.transform.position) <= 0.4f)
    {
        GetNextWaypoint();
    }

    _speed = startSpeed;

    if (Time.time > _nextBroadcastTime)
    {
        MultiplayerController.Instance.SendMinionUpdate(_id,
                                                transform.position.x,
                                                transform.position.y,
                                                transform.position.z);
        _nextBroadcastTime = Time.time + .16f;
    }
}

void GetNextWaypoint()
{
    if (wavepointIndex >= waypoints.Length - 1)
    {
        EndPath();
        return;
    }

    wavepointIndex++;

    target = waypoints[wavepointIndex];
}
```

Ilustración 32: Código que controla los súbditos

La trayectoria de los súbditos consiste en seguir una serie de puntos, por lo que el camino en realidad es un array de vectores de tres dimensiones (x, y, z). Al ser creados transforman su posición cada FPS (*Frame per second*) desde el primer hasta el último punto.

Llegados al último punto los súbditos desaparecen y el jugador recibe una recompensa de oro para gastarlo en la creación de torres.

Mediante el método *SendMinionUpdate* cada súbdito envía su posición a través de la red para que el oponente pueda recibir un reflejo de la posición del súbdito.

Otro punto a tratar es la creación de torres, que para poderla crear se necesita una cierta cantidad de oro que variará en función del tipo de torre seleccionado. No se podrá crear una en un nodo que ya tenga una torre asignada.

```
void Update()
{
    if (!buildManager.CanBuild)
    {
        startColor.a = 0f;
        rend.material.color = startColor;
        return;
    }

    if (turret != null)
    {
        rend.material.color = startColor;
        return;
    }

    if (buildManager.HasMoney)
    {
        hoverColor.a = 1f;
        rend.material.color = hoverColor;
    }
    else
    {
        notEnoughMoneyColor.a = 1f;
        rend.material.color = notEnoughMoneyColor;
    }
}

void OnBuildTurret()
{
    if (!buildManager.CanBuild)
    {
        return;
    }

    if(turret != null)
    {
        return;
    }

    buildManager.BuildTurretOn(this);
}
```

```
public void BuildTurretOn(Node node)
{
    if (guiController.GetGold() < turretToBuild.cost)
    {
        Debug.Log("Not enough money to build that!");
        return;
    }

    guiController.SetGold(guiController.GetGold() - turretToBuild.cost);

    GameObject turret = Instantiate(turretToBuild.prefab, node.GetBuildPosition(), Quaternion.identity);
    MultiplayerController.Instance.SendTowerCreated(node.GetBuildPosition(), turretToBuild.type);
    node.turret = turret;
    turretToBuild = null;

    Debug.Log("Turret build! Money left: " + guiController.GetGold());
}

public void SelectTurretToBuild(TurretsInShop turret)
{
    turretToBuild = turret;
}
```

Ilustración 33: Código que permite la construcción de torres en los nodos

Cuando un jugador crea una torre se envía un mensaje por la red con los datos de la posición y el tipo de torre. Existen dos tipos de torres:

- La torre estándar dispara proyectiles cada segundo causando daño individual. Su coste es de 150 monedas de oro por torre.
  Los proyectiles buscan al súbdito enemigo más cercano y trazan una trayectoria desde el cañón de la torre hasta su posición.

- La torre mágica lanza un rayo continuo sobre el súbdito enemigo causando daño cada segundo y ralentizando su movimiento. Su coste es de 300 monedas de oro por torre.

Se pueden crear tantas torres como nodos hayan.

Una vez creadas las torres, cada núcleo se podrá defender de las oleadas de súbditos. Si un núcleo llega a 0 de vida pierde la partida y ambos jugadores son llevados a la pantalla de recompensas donde se podrá visualizar la cantidad de oro y experiencia ganados por la partida.

```
void DoMultiplayerUpdate()
{
    if (IsGameOver)
    {
        return;
    }

    if (guiController.GetRightHealth() <= 0 || guiController.GetLeftHealth() <= 0)
    {
        for (int i = 0; i < allPlayers.Count; i++)
        {
            string nextParticipantId = allPlayers[i].ParticipantId;
            _finishMatch[nextParticipantId] = 0;
        }

        Debug.Log("GAME OVER");

        CheckForMPGameOver();
    }
}
```

```
void CheckForMPGameOver()
{
    Debug.Log("CHECK");
    if(mainCore.GetComponent<MainCoreController>().GetSide() == left)
    {
        if(guiController.GetLeftHealth() <= 0)
        {
            gameOverImage.sprite = defeatImage;
            gameOverEffect.sprite = defeatEffect;

            GameResult.instance.coinsValue = 20;
            GameResult.instance.expValue = 50 * AccountStats.instance.levelValue;
            MultiplayerController.Instance.LoseScore();
        }
        else
        {
            gameOverImage.sprite = victoryImage;
            gameOverEffect.sprite = victoryEffect;

            GameResult.instance.coinsValue = 75;
            GameResult.instance.expValue = 100 * AccountStats.instance.levelValue;
            MultiplayerController.Instance.GainScore();
        }
    }
    else
    {
        if(guiController.GetRightHealth() <= 0)
        {
            gameOverImage.sprite = defeatImage;
            gameOverEffect.sprite = defeatEffect;

            GameResult.instance.coinsValue = 20;
            GameResult.instance.expValue = 50 * AccountStats.instance.levelValue;
            MultiplayerController.Instance.LoseScore();
        }
        else
        {
            gameOverImage.sprite = victoryImage;
            gameOverEffect.sprite = victoryEffect;

            GameResult.instance.coinsValue = 75;
            GameResult.instance.expValue = 100 * AccountStats.instance.levelValue;
            MultiplayerController.Instance.GainScore();
        }
    }

    _showingGameOver = true;

    LeaveMPGame();
}

public void LeaveMPGame() {
    MultiplayerController.Instance.LeaveGame();
}
```

Ilustración 34: Código de comprobación de game over

3. Elementos de mayor complejidad

La configuración inicial de Google Play Services es un poco complicada de realizar ya que las guías que hay por Internet están un poco desfasadas, y hay pasos que no se explican y se tienen que averiguar "trasteando" la configuración.

Uno de los elementos con mayor complejidad ha sido el flujo de mensajes entre dispositivos. El problema de la sincronización de las oleadas de súbditos ha sido bastante difícil de abordar, ya que involucraba un gran número de mensajes y aumento del tráfico de red. Al final se ha conseguido reducir mediante técnicas de compresión de datos (explicado más adelante).

4. Retos tecnológicos

- **Reducción FPS**
  En los juegos los FPS (*Frames per second*) son bastante importantes, ya que menos de 30 FPS supone una mala calidad en el juego porque se ve a "saltos". El máximo de FPS al que se puede llegar son 60, ya que a partir de ahí no hay diferencia alguna.

  En los móviles la optimización de los FPS es difícil y requiere de mucha experiencia en temas de renderizado, poligonaje de objetos, iluminación, texturizado,… Además cada juego es diferente y requiere diferentes configuraciones para poder optimizar el juego, por lo que es difícil encontrar información respecto a ello. En mi caso he probado con diferentes dispositivos móviles el juego y la tasa variaba bastante en cada uno de ellos, desde 58 FPS a rangos bajos como 26 – 34 FPS. Al no tener un apoyo económico importante para la obtención de elementos gráficos optimizados para dispositivos móviles, resulta bastante complicado obtener una tasa de FPS constante y alta.

- **Reducción de latencia**

  La latencia es el tiempo que tarda un mensaje desde que se ha enviado hasta que se ha recibido. Ya desde un principio desarrollar un juego multijugador supone un gran reto, con lo que optimizar la latencia para que se pueda obtener un producto fluido supone un reto aún mayor.

  Podría decirse que la latencia es el mayor de los problemas que se ha tenido al desarrollar el proyecto ya que en esta aplicación hay muchos elementos involucrados que envían mensajes por la red constantemente.

  Se han encontrado algunas soluciones bastante aceptables para reducir un poco el problema de la latencia. A continuación se explican algunos ejemplos de optimización:

  - Reducción del tamaño de mensajes: Como ya se ha explicado anteriormente cada usuario envía mensajes al otro jugador para informar sobre el actual estado del juego. En estos mensajes se pasan variables de tipo "int", "float" y "byte", por lo que el tamaño de cada mensaje puede ser bastante grande ya que un "int" ocupa 4 bytes, al igual que un float. Una solución ha sido reducir el tamaño de estos mensajes convirtiendo los 4 bytes de un int a 1 byte. Mirando que el rango de tamaño de un byte va de 0 a 255 se ha optado por convertir los "int" a "byte" ya que el valor de esos "int" no iba a superar el valor de 255.

- Compresión de datos: Otra forma de reducción que se ha encontrado pero que no se ha utilizado por temas de precisión fue reducir los 4 bytes de un float a 1 byte. Por ejemplo, obteniendo la posición 11.0 en eje X (esta posición es un float) si se multiplica por 256/360, da 8, a continuación este resultado se pasa a "int", con lo que con el método anterior de pasar "int" a "byte" se reduce el float a 1 byte. Al recibir el mensaje se vuelve a convertir ese byte a float multiplicando el resultado (8) por 360/256 con lo que da 11.25.
  Este método pierde algo de precisión pero es un buen método para optimizar el paso de mensajes.

- Otra forma de reducir la latencia seria reduciendo en número de llamadas a la red, ya que por defecto Unity realiza 30 llamadas por segundo y cambiándolo a 6 es una buena forma de reducir el tráfico de red.
  Para obtener esta reducción se crea una variable para controlar el envío, entonces al tiempo actual se le suma 0.16f para reducir las llamadas a 6 por segundo. Después solo se comprueba que el tiempo actual sea mayor que el tiempo permitido para enviar un mensaje.

```
private float _nextBroadcastTime = 0;
```

```
if (Time.time > _nextBroadcastTime)
{
    MultiplayerController.Instance.SendMinionUpdate(_id,
                                                    transform.position.x,
                                                    transform.position.y,
                                                    transform.position.z);
    _nextBroadcastTime = Time.time + .16f;
}
```

Ilustración 35: Fragmento de código para reducción de mensajes por segundo

- **Sincronización multijugador**
  La sincronización multijugador ha supuesto un gran reto a la hora de abordar el proyecto. La complejidad que supone sincronizar el comienzo, el transcurso y la finalización de una partida es de grandes dimensiones.

  Para poder sincronizar una partida se deben tener en cuenta aspectos como, el ancho de banda y las características del móvil de cada jugador. Para poder sincronizar la partida se ha utilizado una pantalla de carga para que un jugador no empiece una partida antes que el otro, ya que el que empieza antes tendría ventaja sobre el otro, y además la partida no se desarrollaría igual en los dos dispositivos.

PLANIFICACIÓN

En esta sección se va a establecer la duración de las distintas tareas y actividades realizadas a lo largo del proyecto. Esta planificación se definirá en base a la herramienta del Diagrama de Gantt en la cual se representará gráficamente la duración de cada etapa o actividad a partir de una fecha comienzo y una de fin en base a un calendario. Este diagrama se representa en la imagen de la siguiente página.

Como se aprecia en el diagrama las fases de documentación sobre Unity3D y la fase de implementación han supuesto una gran carga en el desarrollo del proyecto

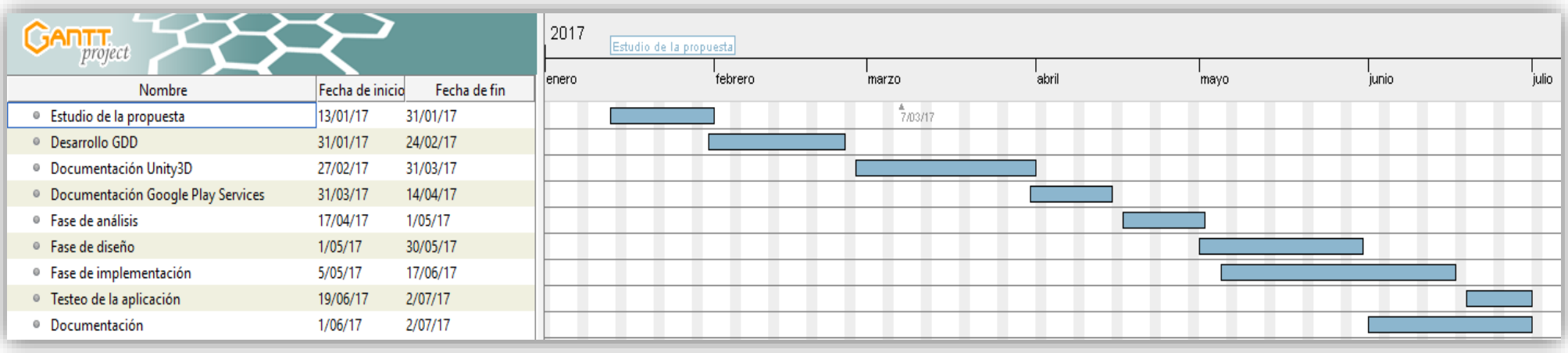

Ilustración 36: Diagrama de Gantt - Planificación del TFG

RESULTADOS

1. Capturas de la aplicación

Ilustración 37: Pantalla de carga

Ilustración 38: Menú principal

Ilustración 39: Menú principal en búsqueda de una partida rápida

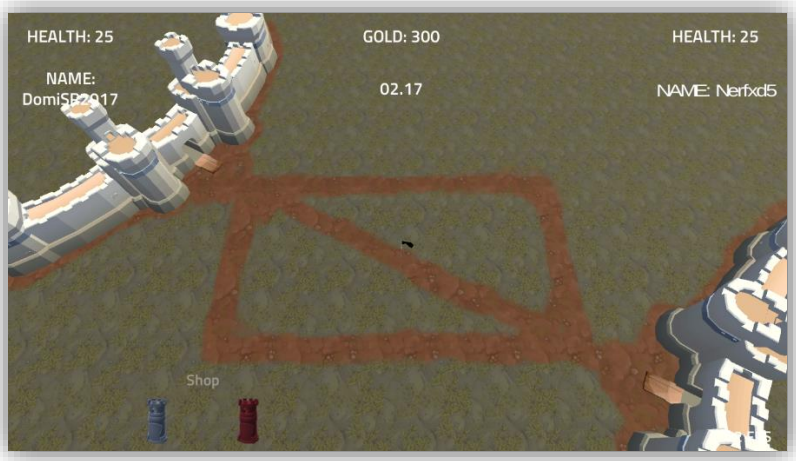

Ilustración 40: Inicio de una partida

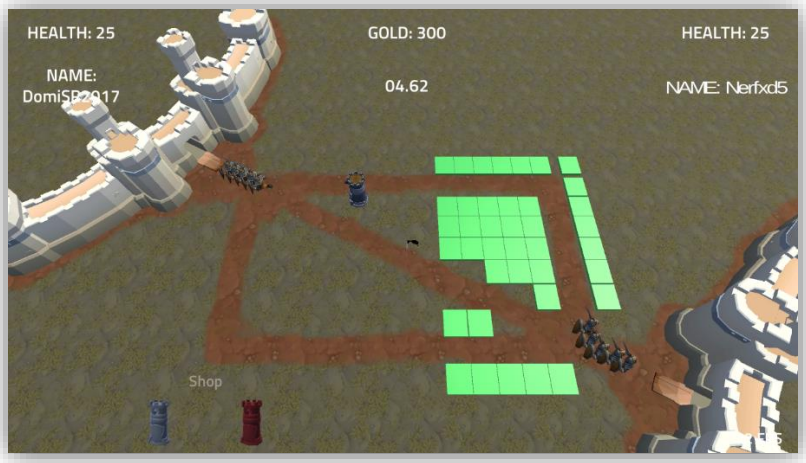

Ilustración 41: Torre seleccionada para su colocación en la partida

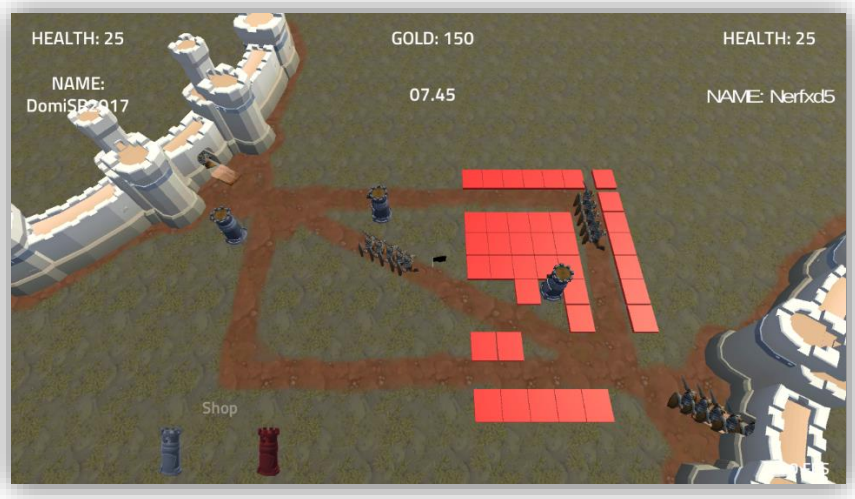

Ilustración 42: Indicativo de falta de oro para colocar torres

Ilustración 43: Pantalla de recompensas

2. Publicar en Stores

Para poder subir una aplicación o un juego a Play Store se debe de crear una cuenta de desarrollador en la que hay que pagar 25$.

Una vez registrado hay que seguir toda una serie de pasos específicos para que la aplicación pueda ser subida con éxito.

El juego se ha catalogado como juego de estrategia y apto a partir de 6 años de edad. La clasificación del contenido del juego, Google lo ha acabado catalogando de la siguiente forma:

Ilustración 44: Imagen de catalogación del juego

Todas estas imágenes representan que en todo el mundo, este juego es apto para mayores de +7 años.

Aquí podemos ver una visualización del resultado final de la aplicación subida a Play Store:

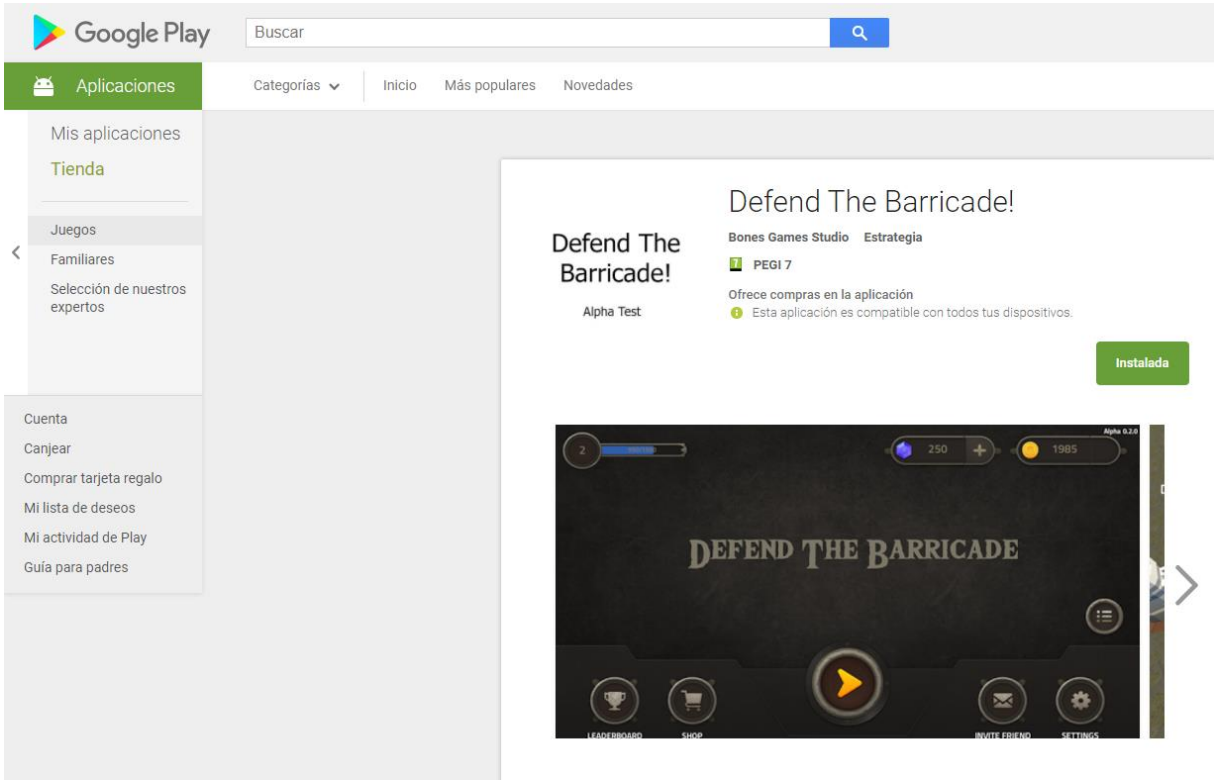

Ilustración 45: Imagen del juego publicado

TRABAJO FUTURO

Después de haber finalizado el proyecto, existen diferentes mejoras y nuevos desarrollos que podrían llevarse a cabo en el futuro. A continuación se detallan una serie de mejoras o nuevas características:

- Mejorar aspectos visuales.
- Optimización del networking.
- Más opciones de compra en la tienda.
- Más mapas.
- Añadir la aplicación al mercado de Amazon Store y Apple Store.
- Exportar el producto a otras plataformas móviles como Windows Phone o IOS.

CONCLUSIONES

El desarrollo de este proyecto ha sido complicado debido a que se han tenido que superar una serie de retos y tomar diferentes decisiones para completar su desarrollo. Además, la gran dificultad que supone hacer un videojuego por una única persona complica aún más el proyecto, ya que se han tenido que ejercer roles de todo tipo (programador, analista, diseñador, artista, *tester*,…).

La idea principal estaba clara, desarrollar una aplicación de entretenimiento con *networking* para dispositivos móviles utilizando Unity 3D. A partir de esta idea, se ha ido evolucionando el proyecto llegando a un punto en el que se han añadido múltiples funcionalidades que han enriquecido el proyecto.

Todas las nuevas ideas que iban surgiendo a lo largo del proyecto eran fruto de las muchas horas dedicadas al mismo, ya que a medida que se avanzaba en el desarrollo se aprendían más cosas que han permitido aumentar el alcance del proyecto, partiendo de un simple juego donde realizabas una partida, a una aplicación con múltiples funcionalidades como micropagos, logros, tablas de clasificaciones, etc.

Desde un punto de vista personal, ver como los conocimientos adquiridos durante la carrera en las diversas asignaturas ha permitido desarrollar una aplicación que supone una alta satisfacción personal. Si a esto se le suman los nuevos conocimientos adquiridos en la etapa de autoaprendizaje a lo largo del desarrollo de este proyecto, se ve como se cumplen con creces los objetivos propuestos y se amplían a la vez que se plasman los que ya se poseían.

BIBLIOGRAFÍA

[1] Unity Scripting API - Versión 5.6, Unity Technologies. https://docs.unity3d.com/ScriptReference/, 19 de Junio 2017.

[2] Unity Tutorials, Unity Technologies. https://unity3d.com/es/learn/tutorials, 2017.

[3] How to Create a Tower Defense Game in Unity – Part 1, Barbara Reichart. https://www.raywenderlich.com/107525/create-tower-defense-game-unity-part-1, 8 de Septiembre 2015.

[4] Creating a Cross-Platform Multiplayer Game in Unity – Part 1, Todd Kerpelman. https://www.raywenderlich.com/86040/creating-cross-platform-multiplayer-game-unity-part-1, 22 de Enero 2015.

[5] How to make a Tower Defense Game, Bracekys. https://www.youtube.com/watch?v=beuoNuK2tbk&list=PLPV2KyIb3jR4u5jX8za5iU1cqnQPmbzG0, 6 de Julio 2016.

[6] Google Play Game plugin for Unity, Google Inc. https://github.com/playgameservices/play-games-plugin-for-unity, 7 de Junio 2017.

[7] Google APIs for Android, Google Inc. https://developers.google.com/android/reference/packages, 7 de Junio 2017.

[8] How to Make a Skill Tree, Game Design HQX. https://www.youtube.com/watch?v=3FHhwfmotgs, 8 de Enero 2016.

[9] Limit camera movement, TexByte. http://answers.unity3d.com/questions/1243099/limit-camera-movement-1.html, 13 de Septiembre 2016.

[10] How to Integrate the new IAP System [Tutorial], Alzan. https://forum.unity3d.com/threads/how-to-integrate-the-new-iap-system-tutorial.374604/, 2 de Septiembre 2012.

[11] Configuring for Google Play Store, Unity Technologies. https://docs.unity3d.com/Manual/UnityIAPGoogleConfiguration.html, 19 de Junio 2017.

[12] Best Practices for Rift and Gear VR, Oculus VR LLC. https://developer.oculus.com/documentation/unity/latest/concepts/unity-best-practices-intro/, 2017.

[13] Practical Guide to Optimization for Mobiles – Rendering Optimization, Unity Technologies. https://docs.unity3d.com/430/Documentation/Manual/iphone-PracticalRenderingOptimizations.html, 16 de Julio 2013.

[14] Touch Input Tutorial, Sebastian Lague. https://www.youtube.com/watch?v=SrCUO46jcxk, 2 de Febrero 2014.

[15] How to Make a Loading Screen in Unity, Nick Pettit. http://blog.teamtreehouse.com/make-loading-screen-unity, 3 de Noviembre 2015.

[16] Real-Time Best Practices, GameSparks. https://docs.gamesparks.com/tutorials/real-time-services/real-time-best-practices.html, 28 de Junio 2017.

[17] Play Game Services in Unity, Google Inc. https://codelabs.developers.google.com/codelabs/playservices_unity/index.html?index=..%2F..%2Findex#0, 7 de Junio 2017.

ANEXOS

1. GDD (Game Design Document)

1.1 Descripción General

Juego tipo *tower defense* multijugador que consta de partidas rápidas en las que cada jugador deberá defender su núcleo utilizando el propio arsenal de torres y habilidades para mejorar las características de las oleadas de súbditos en el momento más oportuno.

Un juego tipo *cartoon* ambientado en diferentes mundos.

1.2 Plataforma dirigida

El juego estará enfocado a plataformas móviles utilizando el motor gráfico Unity 3D y los servicios de Google Play.

1.3 Modelo de monetización

Los micropagos internos en el juego será el modelo de monetización utilizado. Constará de una serie de productos cada uno de mayor precio, con ese producto se podrán obtener recompensas visuales si se desea.

1.4 Estilo visual

El estilo visual será de tipo *cartoon* como *Clash Royale* u otros juegos de tipo *tower defense* en 3D.

1.5 Estilo de audio

Dependiendo del mapa de la partida la música será diferente y enérgica para tener al usuario alerta y centrado en el juego.

1.6 Empezando a jugar

1.6.1 Menú principal

Al entrar al juego se visualizará un *logo* y todo seguido aparecerá una pantalla de carga de menú. En esta pantalla se realizarán varias acciones antes de seguir con la siguiente pantalla:

- Conectar con los servicios de Google Play.
- Cargar datos del juego si existen, sino significa que es la primera vez que se juega.
- Si es la primera vez que se juega se crearan valores por defecto y luego se guardaran.

Después de realizar todas las acciones anteriores, se cargará la pantalla del menú principal, donde se encontrarán diferentes opciones:

- Jugar.

- Tabla de clasificación.
- Logros.
- Tienda de mejoras visuales.
- Invitar amigo.
- Opciones.
- Tienda de gemas.

En el menú principal también se indicará el nivel del jugador, las gemas y las monedas que este poseerá.

1.6.2 Comienzo del juego e introducción

Para poder empezar a jugar partidas multijugador se tendrán que pasar una serie de partidas de introducción para que el jugador aprenda la mecánica del juego.

Una vez terminado estas partidas, el jugador será capaz de realizar partidas competitivas contra jugadores reales.

- Existirán dos formas de jugar en modo competitivo.
- Buscando automáticamente una partida.
- Invitando a un amigo.

Antes de empezar la partida, los jugadores serán enviados a la pantalla de carga donde se sincronizará la partida.

Una vez sincronizados cada jugador será situado a cada lado (izquierda o derecha). A partir de este momento cada jugador tendrá 300 monedas de oro para poder empezar a situar torres y defender su núcleo y habrá un contador de 5 segundos antes de que las oleadas de súbditos empiecen a aparecer.

Al matar a un súbdito del equipo contrario el jugador obtendrá monedas de oro y experiencia al igual que si un súbdito llega al núcleo enemigo. Se ganará más experiencia y monedas de oro si un súbdito llega al núcleo enemigo.

Con la experiencia, el jugador subirá de nivel hasta un máximo de 5, con lo que obtendrá mejores torres y súbditos durante la partida.

Con el oro, el jugador podrá comprar nuevas torres o mejorar las ya existentes. También se utilizará para las habilidades.

La partida se termina si el tiempo se agota o si un jugador pierde toda la vida. Si el tiempo se agota ganará el que más vida tenga. En caso de tener la misma vida se empatará.

Al finalizar la partida, cada jugador recibirá una serie de recompensas. El que gane tendrá mucha más recompensa que el que pierda.

1.6.3 HUD en la partida y Menús

En la partida aparecerán la vida y nombres de ambos jugadores y el oro del propio jugador en la parte superior de la pantalla.

En la parte inferior aparecerán las habilidades (vacías al principio ya que no se han añadido puntos de nivel al núcleo) y la tienda de torres (aparecerán las torres básicas y según el nivel del núcleo aparecerán más).

1.6.4 Multijugador

Se utilizarán los servicios de Google Play Services para el juego. De este modo se configurarán rápidamente las opciones de "matchmaking", tabla de clasificación, tienda, logros e invitaciones a amigos.

1.7 UI

1.7.1 Menú principal

El menú constará de:

- Tabla de clasificación
  - Cada jugador podrá ver en qué posición de la clasificación se encuentra.
- Logros
  - Se obtendrán logros si se realizan ciertos pasos en las partidas.
- Tienda
  - Se podrán comprar monedas para intercambiarlas por objetos estéticos en el juego.
- Botón de jugar
  - Para empezar partida rápida.
- Invitar amigos
  - Invitaciones a amigos para realizar una partida.
- Opciones
  - Conectarse/Desconectarse de los Servicios de Google Play.
- Sonido
  - Activar/Desactivar.
- Calidad de los gráficos.
- Nivel de la cuenta
  - Se aumentará a medida que se juegue.
- Tipo de moneda 1
  - Comprar objetos por menor precio. La moneda es más difícil de conseguir.
- Tipo de moneda 2
  - Comprar objetos por mayor precio. La moneda es más fácil de conseguir.

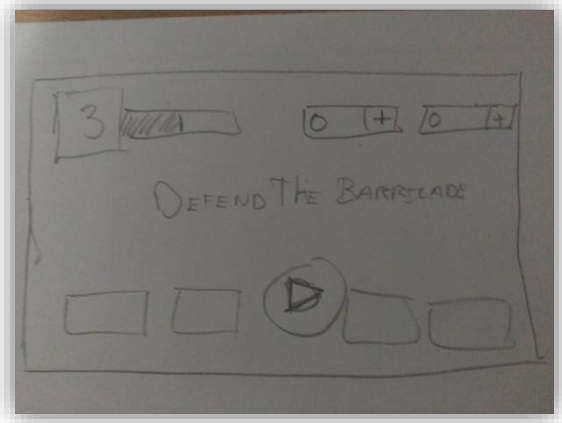

Ilustración 46: Esbozo pantalla menú principal

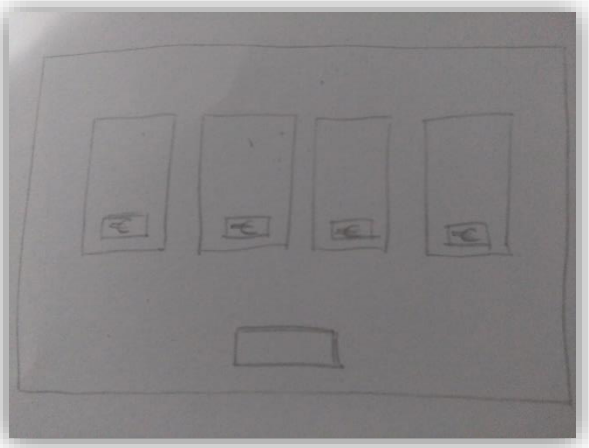

Ilustración 47: Esbozo pantalla de tienda de gemas

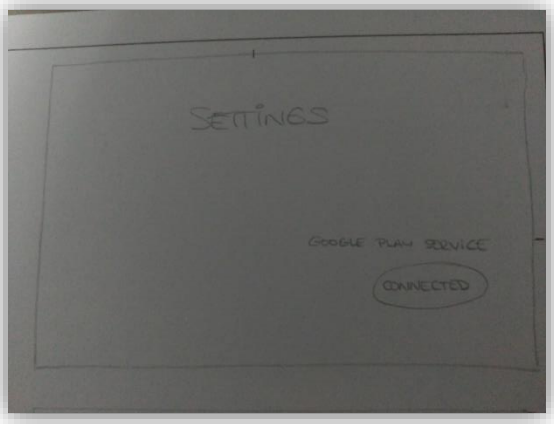

Ilustración 48: Esbozo pantalla de ajustes

1.7.2 Pantalla de configuración de la partida

Al pulsar el botón “Jugar” se buscará automáticamente una partida. Al encontrarla se conectarán los dos jugadores y ambos accederán a la pantalla de carga para sincronizar el inicio de la partida.

1.7.3 HUD en la partida

La interfaz en el juego contendrá los siguientes componentes:

- Esquina superior izquierda:
  - Vida del jugador 1
  - Nombre del jugador 1
- Parte superior al centro:
  - Oro
- Esquina superior derecha:
  - Vida del jugador 2
  - Nombre del jugador 2

- Esquina inferior izquierda:
    - Habilidades
- Esquina inferior derecha:
    - Tienda de torres

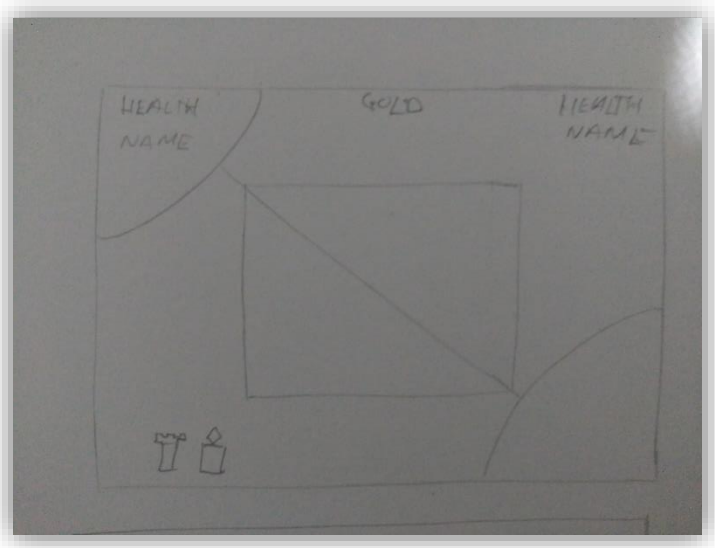

Ilustración 49: Esbozo mapa del juego

1.7.4 Pantalla de final de partida

Al finalizar la partida aparecerá un texto de victoria o derrota y todo seguido se cargará la pantalla de recompensas donde se visualizarán las mismas.

1.8 *Gameplay*

Al empezar una partida los dos jugadores (los núcleos) se situarán uno a cada lado y después de 5 segundos empezarán a salir oleadas de súbditos para atacar el núcleo enemigo.

Durante la partida cada jugador puede utilizar torres para defender su núcleo y así poder defenderse de las oleadas de enemigos.

A lo largo de la partida cada jugador recibirá experiencia para subir de nivel el núcleo y así obtener mejoras en las torres y en los súbditos. También obtendrá puntos de nivel para conseguir habilidades con lo que podría obtener una ventaja en la partida.

Si la vida del oponente llega a cero, ganará el jugador que tenga aún vida. Y si el tiempo se ha agotado y los dos jugadores tienen aún vida, ganará el jugador que más vida tenga.

1.8.1 Mecánicas

La mecánica del juego es simple y se puede enumerar de la forma siguiente:

- Matar súbditos enemigos para obtener monedas de oro y realizar acciones como compra de torres o accionar habilidades.

- Utilizar las habilidades en el momento oportuno para hacer que los súbditos lleguen al núcleo del enemigo y derrotarlo. Algunas habilidades serán:
    - Aumento temporal de la velocidad de los súbditos.
    - Aumento temporal del daño de los súbditos.
    - Reducción temporal del tiempo de aparición de los súbditos.
    - Colocar estratégicamente las torres para defenderte mejor de las oleadas de súbditos.
- Utilizar las habilidades en el momento oportuno para defenderte de los súbditos del oponente. Algunas habilidades serán:
    - Aumento temporal del daño de las torres.
    - Aumento temporal del rango de las torres.
    - Reducción temporal del coste de las torres.

1.8.2 Controles

No existen controles en el juego sino botones para realizar acciones como habilidades especiales o crear torres.

1.8.3 Ganar la partida

Para ganar el juego debe quedar la vida del oponente a cero o si el tiempo se agota y ninguno de los dos jugadores tiene la vida a cero, entonces el que más vida tenga en ese momento gana la partida.

1.9 Puntuación

La recompensa de monedas es la misma siempre, mientras la recompensa de experiencia variará según el nivel que tenga la cuenta del jugador.

1.10 Recompensas

Al ganar una partida se obtienen 75 monedas de oro y 100 de experiencia por el nivel de la cuenta del jugador.

Al perder una partida se obtienen 50 monedas de oro y 50 de experiencia por el nivel de la cuenta del jugador.

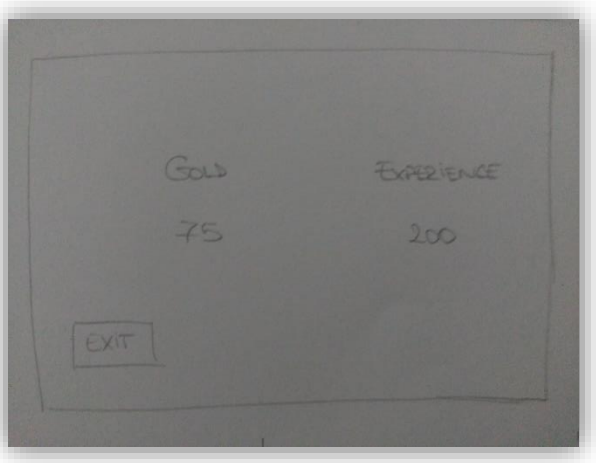

Ilustración 50: Pantalla recompensas

1.11 Misiones y Logros

Existe cierta cantidad de logros que se obtendrán al completar ciertos objetivos en el juego.

- **Juega!**: Juega 100 partidas.
- **Juega más!**: Juega 250 partidas.
- **Juega más aún!:** Juega 500 partidas.
- **Juega mucho más!**: Juega 1000 partidas.

1.12 *Assets*

Los objetos que se necesitarán para el juego deben ser *low poly* y de estilo *cartoon* ya que el juego está enfocado en plataformas móviles y por eso los requerimientos gráficos deben ser mínimos para una óptima visualización.

Ilustración 51: Torre estandar

Ilustración 52: Torre mágica

Ilustración 53: Súbditos