ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

Titulación:

GRADO EN INGENIERÍA INFORMÁTICA

Título del proyecto:

APLICACIÓN DE ENTRETENIMIENTO PARA ANDROID DESARROLLADA CON UNITY3D EN LENGUAJE JAVASCRIPT.

Autor: Javier Jiménez Revidiego

Tutor: Oscar Ardaiz Villanueva

Pamplona, 19 Noviembre 2015

Índice:

1. Resumen - Sumary

1.1. Sumary

This document contains the memory project thesis titled “Entertainment app for Android, developed with unity3d in JavaScript language.

The project has been devised, planned and developed by Javier Jimenez Revidiego. The supervisor and tutor has been Oscar Ardaiz Villanueva, teacher of the public university of Navarra.

The main goal is to create a new game app for Android devices, which will consists in trying to beat a number of levels with the same dynamics but greater difficulty each. These levels are composed of a three-dimensional interior scene, decorated with multiple objects in order to make it as realistic as possible.

The game tries to teach values such as responsibility and personal autonomy as well as entertain. The player must catch the toys or the “target objects” with a helicopter, which is the main object of the game. The items are randomly distributed before time runs out.

All objects created with the ability to move are affected by gravity, and all have their collision area defined, which means that they cannot be crossed.

Lifecycle phases of this software development have been followed: requirements specification, analysis, design, implementation and testing.

The main developing tool that I am going to use is “Unity 5”, the latest version available of Unity 3D.

The last step will be make a Google Play Store account to upload the project for easy downloading and dissemination

1.2. Resumen

En este documento se recoge la memoria del Proyecto de Fin de Estudios para la obtención del Grado de Ingeniero Informático en la Universidad Pública de Navarra con título: “Aplicación de entretenimiento para Android desarrollada con Unity3D en lenguaje Javascript".

El proyecto ha sido ideado, planificado y desarrollado por Javier Jiménez Revidiego. El supervisor ha sido Oscar Ardaiz Villanueva, profesor en la Universidad Pública de Navarra.

El objetivo principal es crear una nueva aplicación de juego para dispositivos móviles con sistema operativo Android, que consistirá en tratar de superar varios niveles con mayor dificultad cada uno, compuestos por un escenario interior tridimensional, decorado con múltiples objetos.

El juego trata de educar en valores como la responsabilidad y el orden además de divertir.

Tiene como objeto principal un helicóptero con el que el jugador tendrá que capturar los juguetes, repartidos al azar entre diferentes puntos, antes de que se agote el tiempo.

Todos los objetos con capacidad para moverse se ven afectados por la gravedad, y tienen definida su área de colisión lo que implica que no pueden ser atravesados.

Para la realización de este proyecto, se han seguido todas las fases del ciclo de vida en un producto: especificación de requisitos, análisis, diseño, implementación y pruebas.

La herramienta de desarrollo principal que voy a utilizar es "Unity 5", la última versión disponible de Unity 3D.

El último paso será crear una cuenta Google de Play Store para subir el proyecto y facilitar su descarga y difusión.

2. Introducción:

2.1. Antecedentes:

Se desea crear una aplicación para Android, con el objetivo de demostrar los conocimientos y aptitudes además de para obtener experiencia y quizá confianza o inversión de terceras personas interesadas en el negocio del mundo audiovisual en 3D-2D. Para ello, se usarán las siguientes tecnologías y lenguajes:

Adobe PhotoShop CS5 + Filter Forge 2:

Una de las mejores aplicaciones de diseño y tratamiento de imágenes del mundo, junto a un plugin especialmente útil para facilitar la realización de texturas simétricas con las que se pueda texturizar los objetos utilizando un mosaico con estas imágenes.

Autodesk 3dsMax 2011, Maya 2011:

Son 2 potentes herramientas para el modelado de gráficos en 3d, efectos especiales y animación, habitualmente orientadas al desarrollo de videojuegos, películas de animación y simulaciones.

Unity 3D:

Unity 3D es una herramienta de desarrollo y programación integrada para la creación de juegos 3D, aplicaciones interactivas 3D, simulaciones y visualizaciones de formación médica y arquitectónica, etc. Pudiendo estar dirigidas para cualquiera de las siguientes plataformas:

- Dispositivos Móviles: iOS, Android, Windows Phone 8, Blackberry 10, Tizen.
- Dispositivos de Escritorio: Windows, Mac, SO Linux/Stream.
- Consolas: PS3, PS4, PSVITA, XBOX ONE, XBOX 360, Wii U.
- VR y AR: Oculus Rift, Gear VR, Microsoft Hololens y Project Morpheus.

Para poder realizar cualquier tipo de aplicación usando Unity 3D, es necesario crear al menos una escena en la que poder ir incluyendo los objetos. Estos objetos pueden contener diferentes componentes que ayudan a definir la estructura y comportamiento del objeto en el entorno.

Existen varios tipos de objetos especiales: Texturas GUI, Cámaras, Luces, Sistemas de partículas múltiples para componer efectos gráficos tales como fuego, explosiones, arcos eléctricos, humo, tornados, tormentas de arena o cualquier cosa que imaginemos.

Android SDK:

Android SDK (Software Development Kit), es el kit de desarrollo de software con el cual es posible desarrollar aplicaciones para dispositivos con el sistema operativo Android.

Una aplicación Android está compuesta por un conjunto de ficheros empaquetados en formato .apk y guardada en el directorio /data/app del sistema operativo Android.

Para poder publicar los proyectos de destinados a esta plataforma, Unity 3D, requerirá tener instalado el Android SDK. Las aplicaciones podrán ser vendidas a través de Play Store propiedad de Google, los desarrolladores reciben el 70% de los ingresos a través de Google check out.

JavaScript y C#:

Son los principales lenguajes de desarrollo en Unity 3D.

El siguiente enlace, muestra cuáles son sus diferencias y como se usan:

https://unity3d.com/es/learn/tutorials/modules/beginner/scripting/c-sharp-vs-javascript-syntax

En resumen, con Javascript todos los scripts automáticamente derivan de “MonoBehaviour”, una clase de Unity de la cual se heredan todos los scripts, y que se explicará más adelante. Sin embargo, si se quiere utilizar C# o Boo, será necesario especificar que el script hereda o deriva de MonoBeaviour.

2.2. Objetivos:

El objetivo de este proyecto es la creación de una aplicación para el entretenimiento utilizando las ventajas ofrecidas por Unity 3D.

La programación se podrá realizar en los lenguajes JavaScript y C#, para finalmente publicarse en la tienda online de Google “Play Store”.

El resultado debe poder servir de ejemplo para demostrar las aptitudes como ingeniero informático, además de quizá en un futuro para recuperar el coste de inscripción como desarrollador de Play Store, aunque en un principio la aplicación será totalmente gratuita.

Las etapas básicas del desarrollo del proyecto serían las siguientes:

1 - Formación en herramientas y lenguajes para desarrollar la Aplicación.

2 - Análisis y Diseño de la aplicación.

3 - Implementación de la aplicación.

4 - Pruebas de la aplicación.

5 - Documentación de la aplicación para el correcto manejo de la interface.

2.3. Fases del proyecto:

Fase de Formación:

La primera fase consistirá en realizar un breve repaso de los lenguajes de programación, tras lo cual comenzará la preparación y aprendizaje en el uso del programa Unity 3D en su versión 5.

Fase de Desarrollo:

La segunda fase consistirá en satisfacer los requisitos del proyecto, para lo cual será necesario realizar iteraciones, generando un prototipo en cada una de ellas para ir probándolo y mejorándolo de manera progresiva.

2.4. Planificación:

Inicialmente se espera que la fase de formación ocupe como máximo 1 mes, tras lo cual deberá comenzar la iteración 1, la cual será más larga que el resto debido a que deberán producirse todos los recursos necesarios para acabar con un primer prototipo funcional.

Las iteraciones 2 y 3 consistirán en añadir más funcionalidad, hasta el punto en el que se hayan cumplido todos los requisitos.

La iteración 4 deberá comprender una fase de testeo más exhaustiva que en las iteraciones anteriores, además de volver a verificar que se satisfacen los requisitos correctamente.

Fase	Inicio	Fin	Duración
Formación	01/04/2015	01/05/2015	1 mes
Iteración 1	02/05/2015	05/07/2015	2 meses
Iteración 2	17/07/2015	15/08/2015	1 mes
Iteración 3	16/08/2015	15/09/2015	1 mes
Iteración 4	16/09/2015	15/10/2015	1 mes
Documentación y Memoria	16/10/2015	08/11/2015	1 mes

3. Fase de Formación:

3.1. Lenguajes C# y JavaScript:

La formación en estos lenguajes no me ocupó casi nada de tiempo, ya que ya era conocedor de su estructura y funcionamiento y los había utilizado anteriormente en diferentes proyectos, así que únicamente podría considerarse un repaso, para el cual visité las siguientes páginas de ayuda:

http://learnxinyminutes.com/docs/javascript/

http://learnxinyminutes.com/docs/csharp/

El tiempo total que invertí en esta fase fue de apenas unas horas.

3.2. Adobe PhotoShop + Filter Forge:

La formación en este programa tampoco me ocupó demasiado tiempo, ya que también había utilizado gran parte de sus herramientas en innumerables ocasiones, la parte en la que más centré mi formación para este proyecto es la que está enfocada en las texturas, que más tarde debería utilizar para dar realismo a los materiales de los que se supone deberían están compuestos los objetos del juego, las texturas más utilizadas fueron diferentes tipos de madera, para ello decidí utilizar el Plugin “FILTER FORGE 2”, a continuación en esta imagen puede verse un ejemplo de textura que imita a la madera.

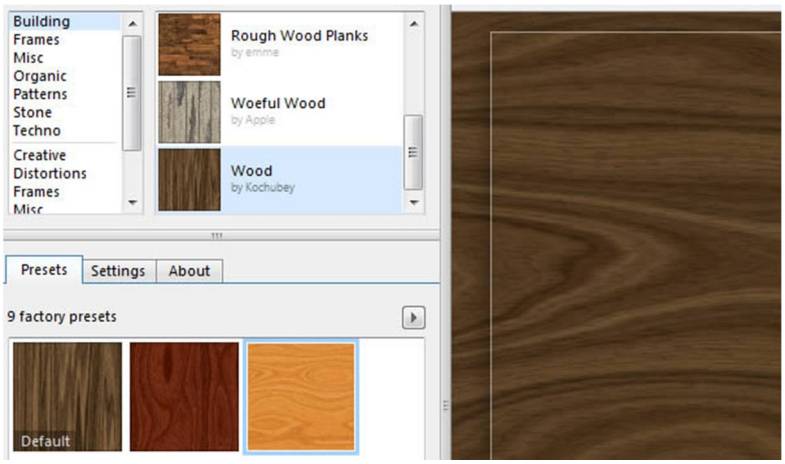

Fig 1. Vista del plugin Filter Forge 2.

3.3. Autodesk 3ds Max y Maya 2011:

En cuanto al primero de los programas también conocido como “3d Studio Max”, ya lo había utilizado antes en la una asignatura dentro del plan de estudios, así que no requería de formación complementaria.

En cuanto al segundo, sí que tuve que invertir parte de tiempo para conocer el funcionamiento de sus herramientas más interesantes, las cuales en mi opinión son bastante potentes. Para ello visité diferentes sitios aunque el principal y el que más útil me pareció fue el siguiente:

https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/ENU/123112/files/maya-tutorials-html.html

3.4. Unity 3D (Versión 5):

Aquí empieza lo realmente interesante, ya que es esta la herramienta principal donde se dará un uso final a los recursos que se han realizado con las anteriores herramientas. Existen diferentes versiones, aunque lo ideal es utilizar la última a día de hoy, la cual es llamada Unity 5.

Como dije anteriormente Unity 3D cuenta con una gran abanico de posibilidades para realizar el aprendizaje, en este link se pueden observar las múltiples opciones y los diferentes recursos disponibles facilitar el aprendizaje y posterior desarrollo a los usuarios: [https://unity3d.com/es/learn]

Comencé leyendo la documentación oficial, la cual es bastante clara además de extensa, está organizada en secciones de las cuales resaltaré las más importantes.

La documentación es accesible desde el siguiente link:

http://docs.unity3d.com/es/current/Manual/UnityManualRestructured.html

Vision general de Unity + Conceptos basicos (suficiente para comprender como funciona el programa) - Teclas de Acceso Rapido (para moverse, girar, escalar y modificar el punto de pivote) - Preferencias + Construyendo Escenas - GameObjects - Componentes - Etiquetas (Tags) - Luces - Camaras + Creacion del juego - Instanciando Prefabs en tiempo de ejecucion - Input - Input convencional del juego - Input de dispositivo movil - Teclado movil - Transforms	**Indicaciones Especificas de Plataforma** + Android **Arquitectura** + Referencias de Arquitectura **Audio** + Vista general de Audio **Graficos** + Informacion general de las graficas (Graphics Overview) -Optimizando el Rendimiento Grafico -Modelar personajes para un Rendimiento optimo + Los comos de las graficas (Graphics HowTo's) - How do I make a Skybox - How do I make a Mesh Particle Emitter - How do I fix the rotation of an imported model - Como importo modelos desde mi aplicacion 3D - Importin objects from Maya - Importin objects from 3D Studio Max
Física + Vision general de la Fisica - Rigidbodies - Colliders - Joints - Character Controllers + Referencia de fisica en 3D	**UI (Interfaz de Usuario)** + UI Overview + UI HowTo's - Designing UI for Multiple Resolutions - Making UI elements fit the size of their content - Creating UI elements from scripting - Creating Screen Transitions
Scripting + Vision general de Scripting - Creando y usando scripts - Variables y el Inspector - Controlar GameObjects utilizando	**Componentes** - Event Functions - Creando y destruyendo GameObjects + Event System Reference - Touch Input Module

Estas son las algunas de las clases más importantes de Unity 3D, aparecen en verde las utilizadas en el proyecto.

1.Collision			
2.ContactPoint			
3.Debug			
4.GUI			
5.Input			
6.Object	a.Component	i. Behaviour	1.AudioSource
			2.AudioListener
			3.GUIElement [GUIText/GUITexture]
			4.Light
			5.Monobehaviour
		ii.Collider	
		iii.ParticleAnimator	
		iv.ParticleEmitter	
		v.Rigidbody	
		vi.Transform	
	b.GameObject		
	c.Texture		
7.Ray			
8.RaycastHit			
9.Screen			
10.Time			
11.Touch			
12.Vector3			

1) Collision:

Clase destinada a detectar las colisiones entre objetos. La información de las colisiones se pasa a través de los eventos.

Collider.OnCollisionEnter:

Este evento es llamado cuando el collider/rigidbody ha comenzado a tocar a otro rigidbody/collider. A diferencia de OnTriggerEnter, en OnCollisionEnter es pasado como parámetro el Collision class y no un Collider. La Collision class contiene información sobre puntos de contacto (contact points), velocidad de impacto (impact velocity), etc.

Collider.OnCollisionStay:

Es llamado una vez por frame para todo collider/rigidbody que está tocando otro rigidbody/collider. A diferencia de OnTriggerStay, OnCollisionStay pasa la clase Collision y no un Collider.

Collider.OnColliderExit:

Se llama cuando el collider/rigidbody deja de tocar otro rigidbody/collider. Como en los anteriores, también pasa la clase Collision en vez de un Collider.

2) ContactPoint:

Describe el punto de contacto cuando sucede una colisión. Los puntos de contacto son almacenados en la estructura Collision.

3) Debug:

Contiene los métodos necesarios para realizar un debug de la forma más sencilla desarrollando un juego, las más utilizadas han sido Break() y Log().

4) GUI:

La clase GUI es el interfaz para los GUI de Unity 3D cuyo posicionamiento debe realizarse de forma manual, indicando el punto de origen y el tamaño para poder ser mostrado en pantalla.

5) Input:

Esta clase se usa para poder obtener información sobre las entradas del dispositivo en el que se esté ejecutando la aplicación, para aplicaciones destinadas a dispositivos de escritorio se establecen los controles de los ejes en el Input Manager, con 17 funciones asignadas en total, pero pueden añadirse más modificando el valor de “size” el cual indicará el tamaño de la lista que contiene las funciones.

Input en dispositivos móviles (Pantalla Multi-Touch):

Los dispositivos iOS son capaces de rastrear hasta cinco dedos tocando la pantalla al mismo tiempo. Los dispositivos Android no tienen un límite unificado en cuántos dedos pueden rastrear. Esto varía de dispositivo en dispositivo y puede ser cualquiera desde dos dedos en dispositivos viejos hasta 5 dedos en algunos nuevos dispositivos.

Cada toque de dedo es representado por una estructura de dato Input.Touch:

Como un dispositivo se mueve, su hardware acelerómetro envía una señal lineal de aceleración que cambia en los tres ejes primarios de las tres dimensiones del espacio. Se puede utilizar esa información para detectar la orientación actual del dispositivo y si esta orientación cambia.

El hardware de aceleración envía los valores de orientación de los ejes del dispositivo en forma de fuerzas G, de tal forma que un valor de 1.0 representa una fuerza de +1G en ese eje, mientras que un valor de -1.0 representa una fuerza de -1G. Para obtener los datos del acelerómetro se puede leer la propiedad Input.acceleration en cualquiera de sus tres ejes, también se puede utilizar Input.deviceOrientation para saber la orientación del dispositivo en el eje deseado.

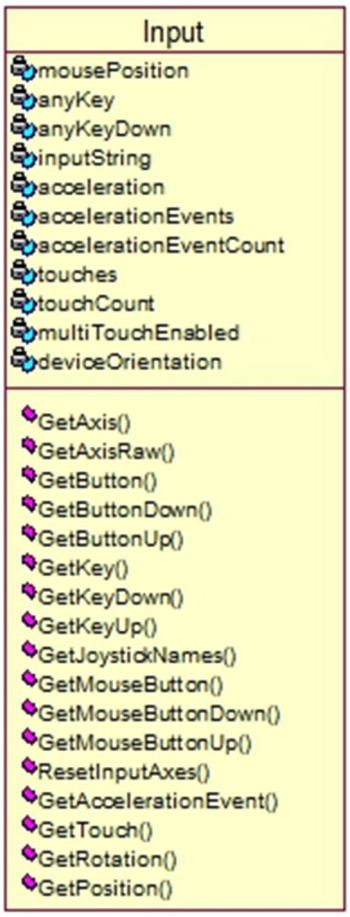

Fig 2. Clase Input

6) Object:

Es la clase base de todos los objetos que Unity 3D puede hacer referencia. Cualquier variable que derive de un objeto podrá verse en el inspector permitiendo darle un valor en el interfaz, sin tener que modificar el script.

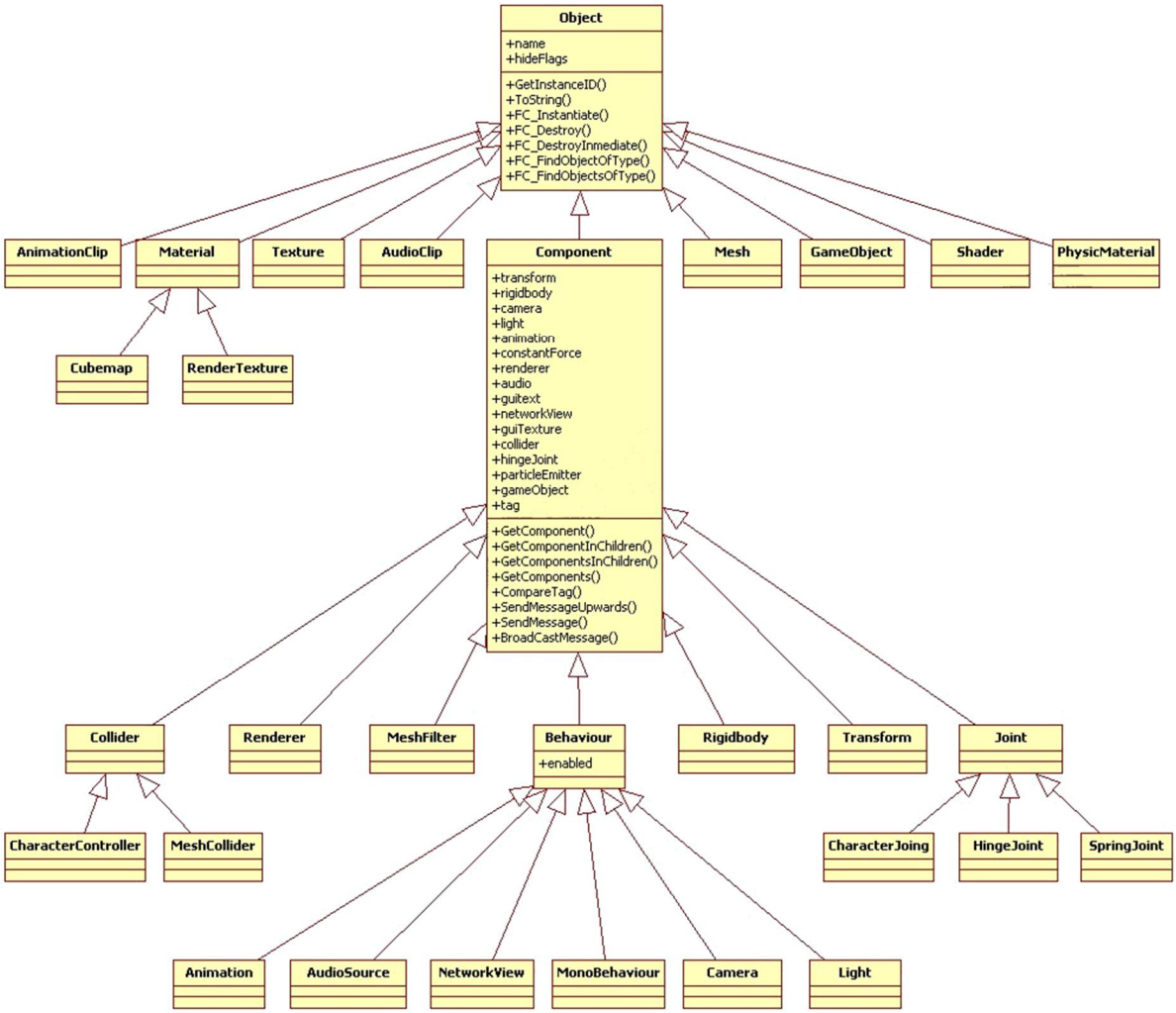

Fig 3. Clases heredadas de la clase Object.

En esta imagen se puede ver el diagrama de clases de la clase Object, en el cual además se ven las relaciones de herencia con las clases más importantes y utilizadas en el proyecto, como son Rigidbody, Transform, GameObject, Texture, Component, y muchas más. La API de Unity está orientada a objetos y se compone de algo más de un centenar de clases, de las cuales una tercera parte más o menos están relacionadas entre sí en términos de herencia y el resto o son conjuntos de funciones auxiliares (clase tiempo, de funciones matematicas, etc) o bien meramente permiten funcionalidades especiales.

Las funciones de la clase (class functions) que se heredan de la clase Object son estas: AnimationClip, AssetBundle, AudioClip, Component, Flare, Font, GameObject, Material, Mesh, PhysicMaterial, ScriptableObject, Shader, TerrainData, TextAsset y Texture.

Component (hereda de Object):

Esta es la clase base para todos los componentes que pueden ir unidos a un GameObject.

Hereda a la mayoría de los miembros de la clase Object anteriormente nombrados: “Behaviour”, “Cloth”, “Collider”, “Joint”, “Mesh Filter”, “Occlusion Area”, “Particle Animator”, Particle Emitter”, “Renderer”, “Rigidbody”, “TextMesh”, “Transform” y “Tree”.

Este es el diagrama de la clase Component:

Fig 4. Clase Component

i) Behaviour (hereda de Component):

Behaviour son componentes que se pueden habilitar o deshabilitar. Solo contienen una variable, enabled, y con ella se pueden habilitar o deshabilitar los componentes.Tiene los miembros heredados tanto de la clase “Components” como de la clase “Object”.

AudioSource (hereda de Behaviour):

A un GameObject se le puede unir un componente AudioSource para así poder reproducir sonidos desde el lugar donde se encuentre el propio objeto, pudiéndose reproducir un único AudioClip en cada momento, el cual podrá reproducirse, pausarse o detenerse usando las funciones audio.Play(), audio.Pause(), y audio.Stop(). También se podrá ajustar el volumen y muchas otras propiedades tanto en el Inspector como en el propio código.

PlayOneShot se utilizará cuando se quiera reproducir una única vez el sonido.

PlayClipAtPoint se utiliza para reproducir sonidos en una posición estática.

AudioListener (hereda de Behaviour):

Representa a un listener en el espacio 3D. Este componente graba todos los sonidos que se reproducen a su alrededor y los vuelve a reproducir para que el usuario los escuche. Solo se deberá tener un listener en cada escena. La variable velocityUpdateMode permite al audio listener variar el modo de actualizarse. Con volume se puede controlar el volumen del juego y con pause puedes pausar el estado del audio.

GUIElement (hereda de Behaviour):

Es la clase básica para imágenes y texto mostrados en la interfaz de usuario (GUI). Esta clase contiene todas las funcionalidades de cualquier elemento GUI, y hereda dos clases las cuales serán importantes ya que permitirán mostrar tanto texturas como texto en el la interfaz de usuario:

GUIText (hereda de GUIElement):

Es un texto para mostrar en la interfaz (GUI) la información dinámica que se desee en cada momento, como por ejemplo los números que pueden indicar el tiempo restante en una partida o el número de objetos recogidos como es el caso de este proyecto.

GUITexture (hereda de GUIElement):

Sirve para poder mostrar una textura 2D en la GUI, y es capaz de detectar si se hace un Input.Touch sobre ella.

Light (hereda de Behaviour):

La clase light se usa para controlar el aspecto de las luces en Unity 3D.

Las propiedades se muestran también en el inspector, y lo más usual es cambiar y controlar los valores de la luz ahí mismo, aunque otras veces puede requerirse modificar estos valores en el código de forma dinámica a lo largo de la ejecución de la aplicación.

MonoBehaviour (hereda de Behaviour):

Esta es la clase de la que derivan o heredan todos los script de la aplicación. Como ya se comentó en la fase de formación, con Javascript todos los scripts automáticamente derivan de MonoBehaviour, sin embargo, si se quiere utilizar C# o Boo, será necesario especificar que el script hereda o deriva de MonoBeaviour. La única variable que maneja esta clase es useGUILayout, que se usa para saltar la fase de diseño de la GUI, deshabilitando esta variable.

Existen unas funciones “Overridable” ya predefinidas, cada una de las cuales tienen un comportamiento característico.

Estas son las más importantes.

Update(): Es la función llamada cada frame, siempre y cuando MonoBehaviour esté habilitado, y si uso más común es para implementar el comportamiento de la aplicación.

FixedUpdate(): Esta función se llama repetidamente, al igual que Update pero a cada fotograma.

LateUpdate(): Esta función se llama repetidamente, al igual que Update pero la primera llamada se realizará cuando el resto de funciones Update ya se haya llamado.

Awake(): Esta función se llama cuando la instancia del script se está cargando. Suele utilizarse para inicializar variables o el estado del juego antes de que el juego comience.

Start(): Se llama una única vez antes de que cualquier método Update sea lanzado, a diferencia de Awake(), Start() se llama solamente si la instancia del script está habilitada.

El orden de ejecución puede verse claramente aqui en la figura 5:

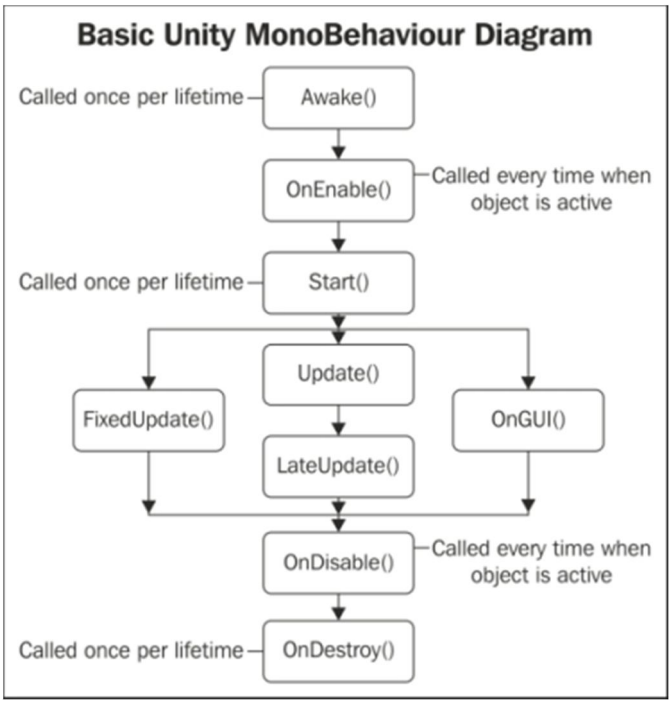

Fig 5. Orden de llamada de las funciones en Unity 3D.

ii) Collider (hereda de Component):

Es la clase fundamental de los componentes que se pueden asociar a los para detectar colisiones con otros colliders. Hay 4 tipos de colliders:

- BoxCollider: Es un collider con forma de cubo que tiene 2 variables “center” y “size” para configurar su tamaño y posición.
- SphereCollider: Es un collider con forma esférica que tiene 2 variables “center” y “radius” para configurar su tamaño y posición.
- CapsuleCollider: Es un collider con forma de capsula que tiene como variables “center”, “radius”, “height” y “direction” para configurar su tamaño, posición y forma.
- MeshCollider: Es un collider que cuya forma se adapta a la malla que define el objeto. En la versión 5 de Unity 3D, este tipo de collider deberá ser de tipo convex si se quiere marcar también como IsTrigger

iii) ParticleEmitter (hereda de component):

Es un script de interfaz para emitir partículas. Se puede configurar indicando el número de partículas, su tamaño, su energía (duración), la posición original de las partículas al emitirse, el rango de emisión… etc.

iv) ParticleAnimator (hereda de component):

Este componente es el encargado de mover las partículas que se crean en un ParticleEmitter, se usa para aplicar fuerzas de arrastre y color a los sistemas de partículas. Tanto ParticleEmitter como ParticleAnimator son scripts que se pueden modificar en el interfaz de Unity 3D en la ventana de inspector o en el propio script.

v) Rigidbody (herada de component):

La clase Rigidbody controla la posición de los objetos en una simulación física. Los componentes Rigidbody tienen la función de controlar la posición de los objetos, de controlar si al objeto le afectará o no la fuerza gravitatoria, y calcular como responden los objetos ante colisiones.

Para manipular los parámetros de un Rigidbody, es recomendable trabajar sobre la función FixedUpdate como ya se comentó anteriormente, esta clase tiene las funciones OnCollisionEnter, OnCollisionExit y OnCollisionStay, para enviar mensajes a otros objetos sobre las colisiones que hayan tenido.

vi) Transform (hereda de Component y IEnumerable):

Todo objeto en una escena tiene un componente Transform. Se usa para almacenar y manipular la posición, rotación y escala de los objetos. Todo Transform puede tener un parent (padre), que permite cambiar y aplicar la posición, rotación y la escala jerárquicamente, en el panel Hierarchy..

b) GameObject (hereda de Object):

Esta clase es la clase plantilla para todas las entidades u objetos de las escenas en Unity 3D. El constructor de la clase es GameObject, que crea un nuevo objeto llamado name (nombre pasado por parámetro).

c) Texture (hereda de Object):

Es la clase base para el manejo de las texturas. Contiene funcionalidades que son comunes en las clases Texture2D y RenderTexture.

7) Ray (estructura):

Es la representación de los rays. Un ray es una línea infinita que comienza en origin (origen) y va en cualquier dirección. Las variables son origin (el punto de origen del ray) y direction (la dirección del ray). El constructor es Ray, crea un ray que comienza en origin hacia direction. Las funciones son GetPoint, que devuelve el punto en las unidades de distancia a lo largo del rayo, y ToString que devuelve una cadena (string) sobre el rayo con un formato entendible.

8) RaycastHit (estructura):

Es una estructura usada para obtener información sobre un raycast, la cual es muy útil para detectar si hay algún collider interponiéndose en la línea visual imaginaria que uno a un objeto con otros.

9) Screen:

Se usa para obtener información sobre la pantalla, una lista de resoluciones soportadas, la resolución actual. SetResolution permite cambiar de resolución.

10) Time:

Es una clase que permite detener o ralentizar el tiempo entre otras cosas.

11) Touch (estructura):

Es una estructura que describe el estado del toque de un dedo en la pantalla.

12) Vector3 (estructura):

Es la representación de vectores 3D y puntos. Esta estructura se usa para indicar la posición 3D y la dirección en la que está o debe estar orientado un objeto. El constructor de la clase es Vector3, crea un nuevo vector con las componentes x, y, z.

Compilación y ejecución de la aplicación de Unity en Android.

El proceso de pasar la aplicación de Unity 3D al propio móvil con Android es tremendamente sencillo, se requiere de una licencia de Unity 3D para Android para poder crear el apk.

Con el SDK de Android instalado, en Unity 3D en la pestaña File, hay que seleccionar la opción “build settings”, en la cual se selecciona como plataforma “Android”, tras esto será necesario insertar los datos que se requieren en la sección de “player settings”, los cuales explicare más detalladamente.

Será obligatorio dotar de un nombre al producto que se desea producir, además de una compañía desarrolladora, en este apartado se puede ingresar la palabra “None” si el desarrollador no pertenece a ninguna compañía.

Además, desde de la versión 4.6 es posible definir el tipo de arquitectura para la que estará destinado el instalador apk.

Si se selecciona “ARM7” la aplicación estará dirigida a la mayoría de dispositivos, con la excepción de lo de arquitectura x86, si se selecciona “X86” la aplicación funcionará solo en algunos de los nuevos dispositivos, si se selecciona la opción que combina ambas propiedades, la aplicación funcionará en todos los dispositivos, pero tendrá un coste en memoria, llegando a requerirse el doble de espacio de almacenamiento para instalar la aplicación.

Si se desea publicar en Play Store habrá que rellenar más apartados los cuales explicaré más adelante en la sección correspondiente.

Una vez adquiridos los conocimientos necesarios sobre la herramienta principal resulta totalmente necesario estudiar los principios básicos de la física que son necesarios para realizar la aplicación.

3.4. Física del vuelo de Helicópteros:

Para diseñar el movimiento de un helicóptero de forma similar al que tendría uno real hay que procurar estudiar los principios básicos de la física que permiten el vuelo de estos aparatos.

En primer lugar los helicópteros deben lograr levantar el vuelo, para ello es necesario vencer a la fuerza de gravedad, el valor resultante de multiplicar la masa de un objeto por la gravedad es lo que conocemos como peso. Para elevar un objeto de 1 kg de peso (considerando la gravedad de la tierra 9.8 m/s^2) se debería aplicar una fuerza igual o superior en sentido opuesto, para lo cual, los helicópteros utilizan lo que llamamos “rotor”. Un rotor es un motor que produce rotación de unas palas externas, colocadas en un ángulo que facilita el desplazamiento del aire en la dirección deseada, en el caso de un helicóptero, el rotor principal que otorga la sustentación está colocado en el centro del fuselaje sobre el eje vertical, produciendo el movimiento de un volumen de aire suficiente para generar la fuerza necesaria para levantar el vuelo.

En la vida real todo motor que gira en una dirección provoca un efecto de rotación en sentido opuesto, este efecto es llamado, par de motor, par de torsión, o esfuerzo de torsión, también conocido como torque, y obliga a que sea absolutamente necesario aplicar una segunda fuerza para compensar el efecto. Para ello se utiliza un segundo rotor u otro tipo de sistema de propulsión. Actualmente existen diversas soluciones para poder contrarrestar el par de torsión y evitar que el cuerpo del helicóptero gire en sentido contrario del giro del rotor principal, además de permitir el poder girar, usar 2 rotores sobre el mismo eje vertical girando en sentidos contrarios, usar 2 rotores sobre diferentes ejes verticales, o usar un segundo sistema de propulsión en la cola, por ejemplo otro rotor, el rotor de cola será el que se utilice para este proyecto ya que es el más conocido por el público en general, este rotor no necesita levantar el peso del helicóptero, así que puede ser más pequeño. En la realidad este rotor únicamente necesita girar en un sentido para poder controlar el giro del helicóptero en ambas sentidos, dado que el rotor 1 ya produce un giro sobre el eje "y" en un sentido, de acuerdo al efecto de par de motor, no hace falta más que emplear una fuerza igual en sentido contrario para equilibrarlo y detener el giro.

Este dibujo podría aclarar el concepto del cual estoy hablando.

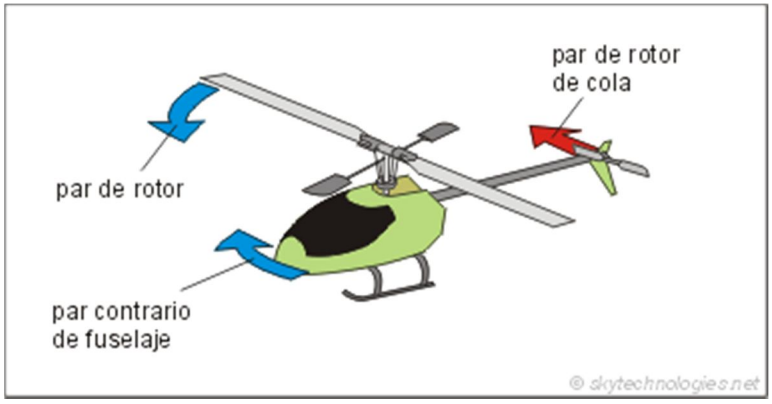

Fig 6. Dibujo del par de rotor.

Pero para simular el vuelo de un helicóptero dentro de un ambiente creado a partir de unas especificaciones, considero que no es necesario recrear este efecto, ya que únicamente sería útil si se diese el caso de la destrucción del segundo rotor o rotor de cola, lo cual debería producir el efecto anteriormente comentado en el helicóptero e impedir su manejo, y el cual podría ser simulado desde de ese momento. Dado que este requisito no está incluido, la simulación de este efecto se dejará para futuras versiones en caso de que quiera añadirse tal funcionalidad.

Este proyecto no trata de realizar un simulador de helicóptero, simplemente se desea utilizar el aparato para desplazarse por el escenario, por lo tanto, la aceleración vertical y el giro podrán aplicarse y controlarse directamente de forma que algunas leyes de la física puedan ser ignoradas, eligiendo en cada momento la dirección en la que deseemos rotar el helicóptero.

El rotor 2 únicamente realiza una función de decoración, reproduciendo una rotación en un único sentido, la velocidad de giro aumentará o se verá reducida de acuerdo a la potencia aplicada al helicóptero en cada instante.

3.3. Análisis:

Análisis de requisitos:

Los requisitos propios de cada escena serán los siguientes:

Escena del menú:

o Las opciones deberán poder elegirse presionando de forma táctil, mediante un simple toque con el dedo en la pantalla sobre la opción deseada.

o Las opciones disponibles permitirán jugar, ver las instrucciones de juego, y salir de la aplicación, al elegir la opción de jugar, se cargará el escenario 1, mostrando mientras tanto los controles.

Escenas de Juego:

o Las escenas comenzarán con un mensaje en gran tamaño indicando el nivel en el que se encuentra el jugador.

o Tras el mensaje que muestra el nivel, se debe mostrar un cuenta atrás de 3 segundos que incluya sonidos.

o El jugador no debe tener posibilidad de movimiento hasta que la cuenta atrás no termine y se considere la partida empezada..

o El jugador dispondrá de un helicóptero, cuyo estado inicial será apagado.

o En cuanto comience la partida, el helicóptero adquirirá potencia por si solo aunque el usuario no toque los controles.

o El helicóptero debe tratar de equilibrar su posición y su potencia en todo momento, logrando estabilidad si se sueltan los controles.

o Para superar el cada nivel habrá que conseguir sumar al marcador un número igual al número de nivel en el que se encuentre el jugador.

o Para incrementar el valor del marcador de objetos, habrá que atraparlos y llevarlos a donde se indique antes de que se agote el tiempo.

o El jugador debe disponer de un control táctil en forma de barra vertical, con la que poder ajustar la potencia.

o El jugador debe disponer de un joystick digital con el que poder indicar el sentido de giro en los ejes “x” e “y”. Con movimientos verticales, dirigirá la inclinación frontal-trasera gracias a una rotación en el eje “x”, y con movimientos horizontales la rotación sobre el eje “y”.

o El jugador debe poder realizar una rotación en el eje z mediante el acelerómetro, el cual deberá usar la posición o ángulo del dispositivo, sin que sea necesario rotar el dispositivo por completo, al mismo tiempo se requiere de una baja sensibilidad inicial para evitar inclinaciones indeseadas por el jugador.

o La potencia debe poder consultarse mediante un para permitir su consulta aunque el sonido esté desactivado.

o El tiempo restante y el número de objetos recogidos se deberán mostrar para permitir su consulta en todo momento.

o Cada vez que se inicie un escenario los objetos aparecerán de forma aleatoria entre una serie de puntos, siendo posible que en algunas ocasiones no se encuentren objetos en algunos de ellos, y con la condición de que únicamente aparecerá un objeto en cada punto.

o Para capturar los objetos habrá que pulsar un botón que activará un efecto visual, creando un rayo que culmina en un campo de atracción, el cual permitirá a los objetos separarse del suelo y aproximarse al origen mediante una fuerza de atracción que se detendrá en cuanto uno de los objetos quede atrapado.

o Cuando alguno de los objetos objetivo este atrapado, se moverá allá donde vaya el jugador sin separarse de la posición en la que se encuentra el efecto.

o Si dos o más objetos objetivo, son atraídos hacia el rayo de atracción, únicamente quedará atrapado el primero que llegue.

o Al transportar un objeto objetivo, si éste entra en contacto con cualquier parte del escenario, incluyendo el suelo, (a excepción del resto de objetos móviles que falten por capturar) irremediablemente se soltará, rompiéndose el enlace que mantenía atrapado al objeto en el rayo de atracción.

o Un objeto objetivo debe poder soltarse mediante el mismo botón que permitió activar la atracción para atraparlo, al romper el enlace, el objeto objetivo debe caer siguiendo la ley de la gravedad.

o Si un objeto objetivo entra en la zona objetivo, se soltará del rayo de atracción automáticamente, sumando la puntuación y el tiempo extra ganado a los marcadores, a partir de aquí dejará de ser considerado objetivo.

o El jugador debe poder contar con una flecha o indicador que señale la dirección que hay que moverse para encontrar el objeto objetivo mas cercano a la posición del jugador, este indicador debe ser de color verde, y una vez capturado un objeto, la flecha debe cambiar su color a rojo, e indicar el lugar donde hay que depositar el objeto ya atrapado.

o El helicóptero debe poder estar visible en todo momento, para ello se debe controlar si algún objeto se interpone entre el helicóptero y el lugar desde donde se sitúa la cámara que ve la escena de juego, realizando las modificaciones necesarias a la posición de la cámara en cada momento.

o El jugador debe poder se disponer de un botón para pausar la escena, el cual al accionarse deberá abrir un menú con 3 opciones: volver al juego, reiniciar el escenario, o salir del juego, dentro del menú, el sonido estará desactivado y los controles inhabilitados.

o Si el tiempo finaliza sin haber logrado el objetivo, la partida habrá finalizado, se mostrará la puntuación respecto al tiempo y los objetos recogidos y se permitirá mediante un botón reiniciar la partida en el mismo nivel.

o La partida también debe finalizar automáticamente si el helicóptero vuelca o llega a colocarse en alguna posición de no retorno en la que irremediablemente se vea incapaz de recuperar su posición de vuelo.

o Si se supera el nivel, se mostrará la puntuación respecto al tiempo y los objetos recogidos y se permitirá avanzar al siguiente nivel o reiniciar la partida en el mismo nivel.

Una vez analizados los requisitos se realizarán los diagramas de estado para representar como sería el funcionamiento de cada escena.

 Diagramas de clases de la aplicación:

Diagrama de clases de las escenas de juego:

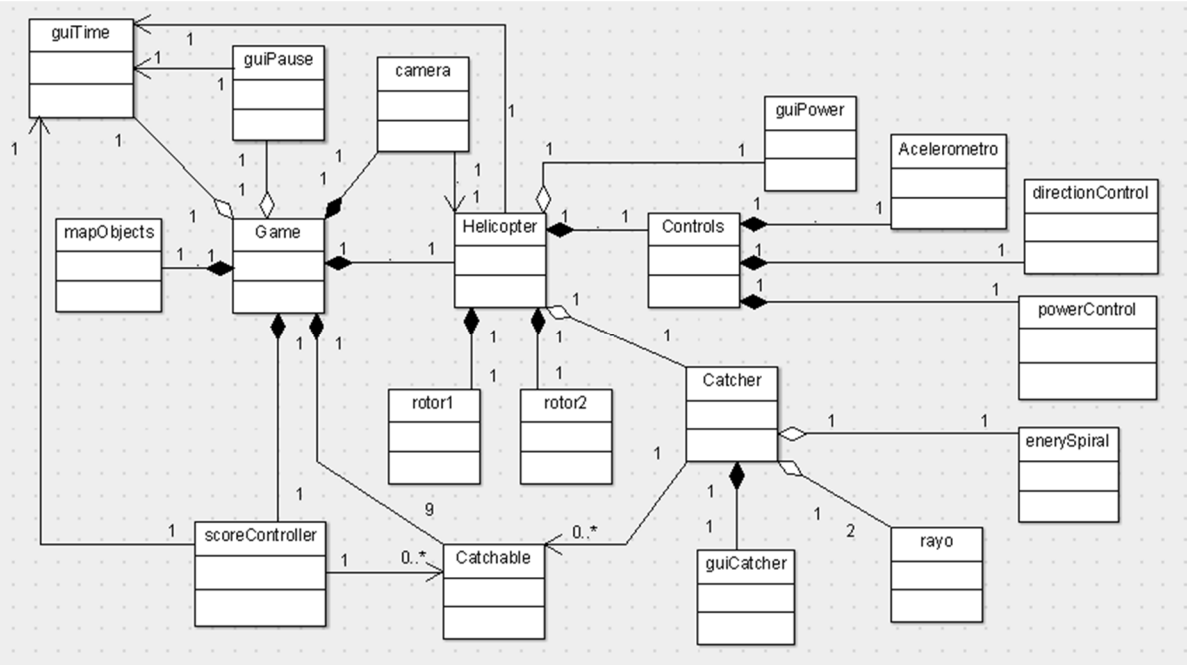

Fig 7. Diagrama de clases.

El juego contiene una serie de objetos los cuales estarán compuestos de otros objetos y asociados a la vez con otros.

Todos los objetos se heredan de la clase Object o de alguna de sus clases heredadas.

El objeto principal es el helicóptero, que está compuesto de dos rotores y sus correspondientes controles, este objeto tiene otras partes cuya importancia es menos relevante para el vuelo, tiene un marcador donde será visible la potencia en cada instante, además de un objeto llamado “catcher”, que le permitirá capturar los objetos llamados “catchables”.

El catcher podrá crear relaciones de asociación con los diferentes objetos objetivos a los cuales se les llama con el tag “catchable”, aunque también es posible que el cátcher no este asociado con ninguno, por lo que esta asociación tendrá una cardinalidad [0..N].

Los objetos de clases “energySpiral” y “rayo” son efectos visuales que se agregaran al catcher cuando esté activado, lo cual se hará desde un objeto de la clase “guiCatcher”, el cual es un botón que podrá ser pulsado por el jugador, y resultará imprescindible tanto para comenzar a atraer los objetos, como para soltar al objeto atrapado en el catcher.

El objeto de la clase scoreController, se encargara de contabilizar los objetos objetivo “catchables” que entren en su zona de colisión, mostrando al jugador el número que indica cuántos de estos han sido recogidos con éxito. Este objeto necesita comunicarse con el objeto de la clase “guiTime” el cual es el encargado de gestionar el tiempo y las fases de juego.

El objeto de la clase “guiTime”, estará relacionado con otros objetos para poder permitir que estos trabajen adecuadamente según la fase en la que se encuentre el juego o el valor actual del tiempo restante, un ejemplo de esto es la comunicación que habrá con el helicóptero en el inicio de cada partida para impedir que se pueda despegar antes de que se dé la salida.

Finalmente el objeto de la clase “guiPause” será el encargado de detener el tiempo gestionado en el objeto de clase “guiTime”, cuando el jugador se lo indique a través de una pulsación del botón.

Hay que tener en cuenta que existen multitud de objetos y componentes en el juego, los cuales todos podrían representarse en un diagrama de clases, pero debido a la extrema complejidad que resultaría, se ha optado por detallar en el diagrama únicamente los esenciales y más importantes.

 Diagramas de estado de la aplicación:

Diagrama de estado de la escena del menú:

Los dispositivos Android no tienen un límite unificado en cuanto al número de dedos que pueden rastrear, varía de dispositivo en dispositivo y puede ser cualquiera desde dos dedos en dispositivos viejos hasta cinco dedos en los más modernos. Cada toque de dedo es representado por una estructura de dato Input.Touch:

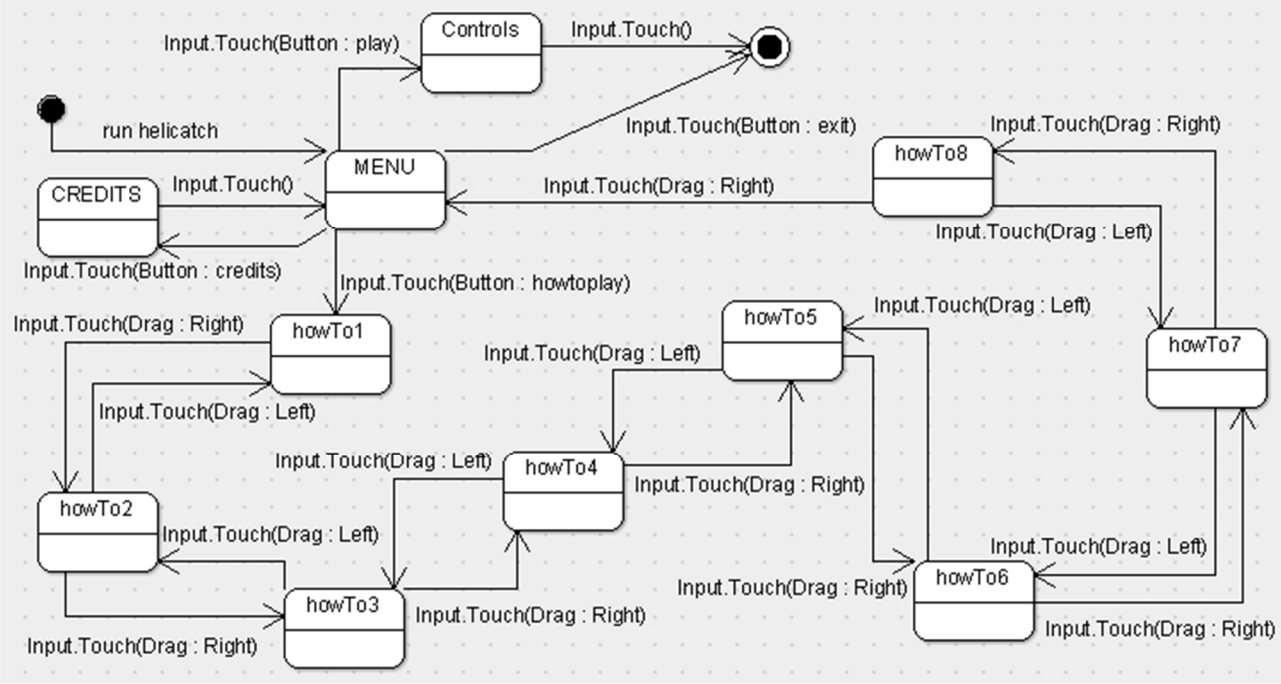

Fig 8. Diagrama de Estado para la escena del menu.

Al arrancar la aplicación aparecerá un menú con las siguientes opciones:

- PLAY (Jugar): Al tocar el botón en la pantalla, se mostrará un panel con los controles de juego.

- HOW TO PLAY (Como jugar): Al tocar el botón en la pantalla, aparecerá la primera imagen mostrando las instrucciones de juego, el modo de avanzar o retroceder en el conjunto de imágenes de instrucciones será tocando y arrastrando el dedo hacia la derecha o hacia la izquierda de la pantalla.

- CREDITS: Al tocar el botón en la pantalla, se abrirá un panel con la información del desarrollador, pudiendo volver al menú tocando otra vez la pantalla.

- EXIT: Al tocar el botón en la pantalla, la aplicación se cerrará.

Diagrama de estado de cualquiera de las escenas de juego:

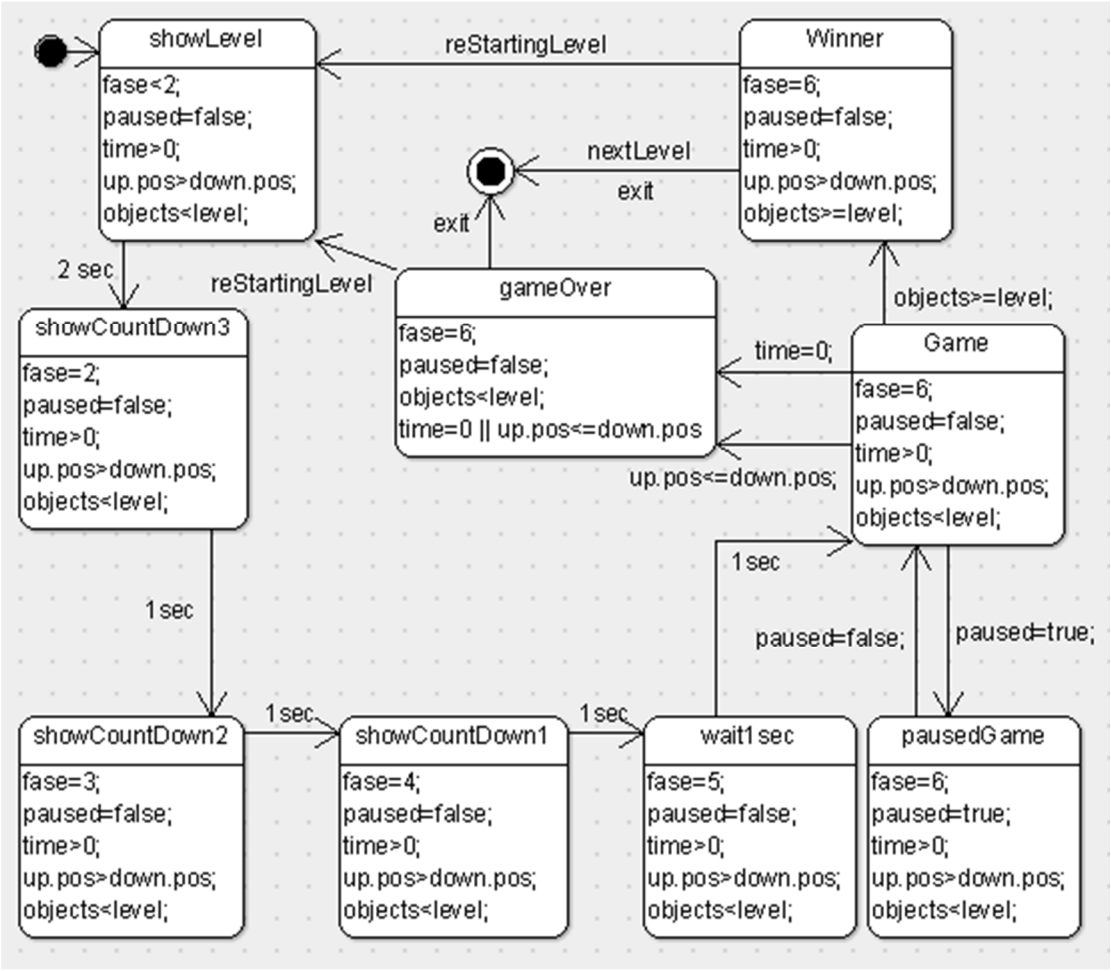

Fig 9. Diagrama de Estado para la escena de juego

Todas las escenas de juego siguen la misma dinámica, la única diferencia radica en el número de objetos objetivo que se exige recoger para superar el nivel.

Para diferenciar los posibles estados en los que podría encontrarse el juego, hay que diferenciar los posibles estados de 7 variables. La variable “fase” es la principal, la cual comienza con valor cero. Durante los seis primeros segundos se modifica su valor aumentándola en una unidad cada segundo.

El estado showLevel se mantendrá durante los dos primeros segundos de cada partida, teniendo la variable fase un valor menor de dos, en este estado se mostrará el nivel en el que se encuentra el jugador.

Los estados showCountDown3, showCountDown2, y showCountDown1, serán los siguientes, durante un segundo cada uno, indicarán una cuenta atrás incluyéndose sonidos además del número que indicará los segundos restantes para dar comienzo a la partida.

Tras otro segundo, la variable fase aumenta su valor a 5, donde el juego se encontrará en un estado llamado “wait1sec”, en este estado no se mostrará ninguna imagen, únicamente deberá sonar el sonido que indica el final de la cuenta atrás, únicamente servirá para esperar durante un segundo más, dando comienzo a la fase 6, en este punto se empieza a descontar el tiempo, y se permitirá empezar a moverse al jugador, es lo que aparece en el diagrama como estado “Game”.

Si el tiempo se agota llegando al valor cero, irremediablemente el juego cambiará a un estado comúnmente conocido como “Game Over”, el cual indicará que la partida ha finalizado sin haber cumplido el objetivo. A este mismo estado se llegará si se vuelca el helicóptero, o lo que es lo mismo, si la parte superior del helicóptero se encuentra por debajo de la parte inferior.

Desde el estado de juego (fase=6) se puede ir a un estado conocido como “Pausa”, que consiste en detener el avance del tiempo en el juego para permitir que el jugador pueda dirigir su atención hacia otras tareas, este estado se activará mediante un toque con el dedo sobre el botón de pausa, lo que provocará que se modifique el valor de la variable “paused” haciéndola “true”.

Por último, existe un último estado al que se llegará superando el objetivo del nivel, el número de objetos atrapados debe ser mayor o igual al nivel del juego (objects >= level), este estado permite repetir el escenario, salir, o avanzar de nivel en caso de no ser el último.

Diagrama de estado del helicóptero en cualquiera de las escenas de juego:

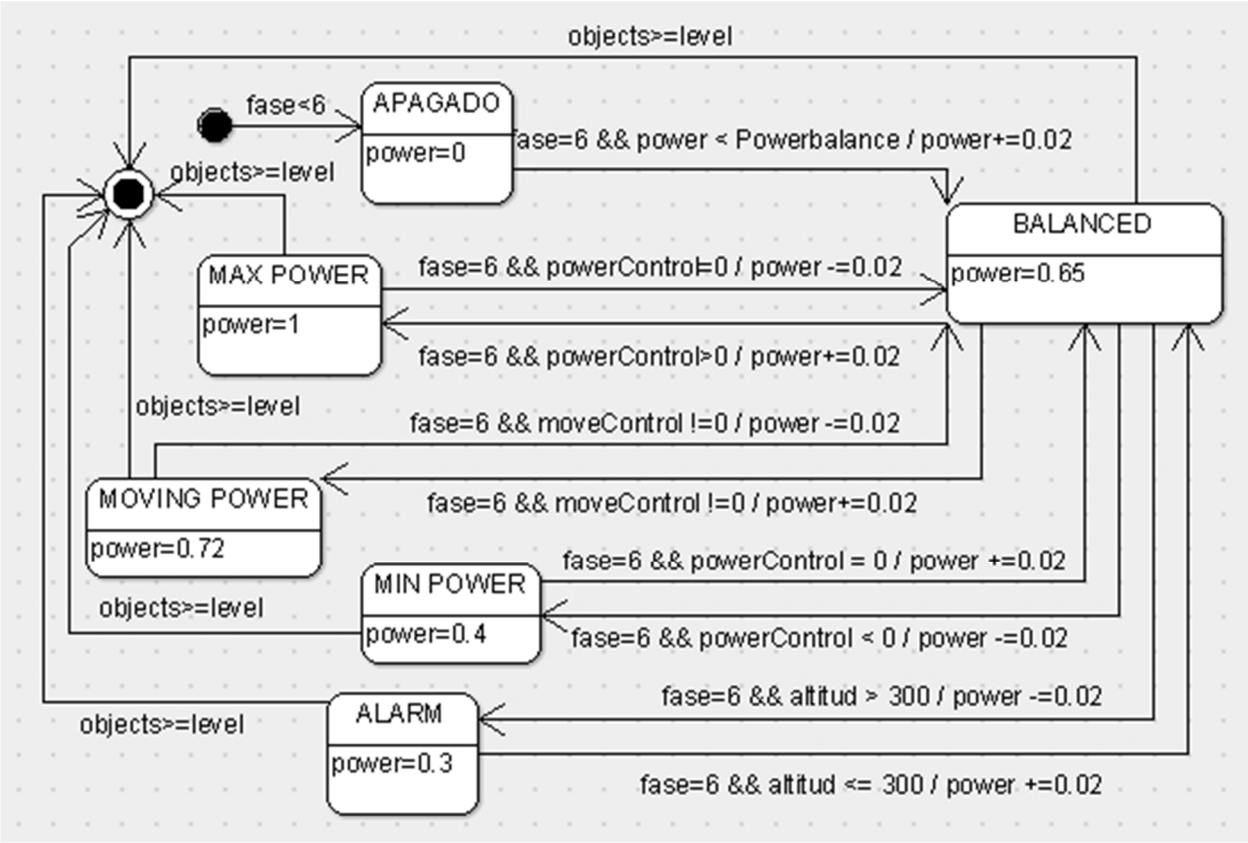

Fig 10. Diagrama de Estado del helicóptero.

En primer lugar el helicóptero comenzará en estado apagado, tras comenzar la fase de juego, adquirirá potencia hasta equilibrarse con su propio peso y la gravedad. Se dispondrá de 2 controles táctiles, y el acelerómetro, para poder manejar el helicóptero, uno de los controles táctiles será el encargado de modificar la potencia, el cual estará compuesto por una barra vertical que detectará si el jugador arrastra el dedo hacia arriba o hacia abajo. El otro control será el típico joystick de un gamepad con el cual el helicóptero podrá rotar sobre el eje vertical, o inclinarse hacia adelante o hacia atrás.

El control para seleccionar la potencia deseada detectará como decía únicamente movimientos verticales, lo que permitirá variar la potencia según el caso. Cuando se mueva hacia arriba, el helicóptero estará a máxima potencia, en caso de moverse hacia abajo, la potencia se reducirá al 40%, si se suelta el control de potencia, ésta se auto-ajustará al nivel que permite la estabilidad.

El control para seleccionar la dirección se podrá mover los ejes vertical y horizontal, adquiriéndose un nivel de potencia superior al nivel de estabilidad del estado “Balanced”.

Se considerará únicamente la potencia como variable indicadora del estado del helicóptero, ya que el resto de controles solo permiten modificar la orientación en caso de que la potencia sea superior al 0%. El acelerómetro será capaz de detectar la posición del dispositivo para aplicarla al helicóptero a través de una función que facilitará el manejo en gran medida, la cual explicare más adelante.

Diagrama de estado del catcher en cualquiera de las escenas de juego:

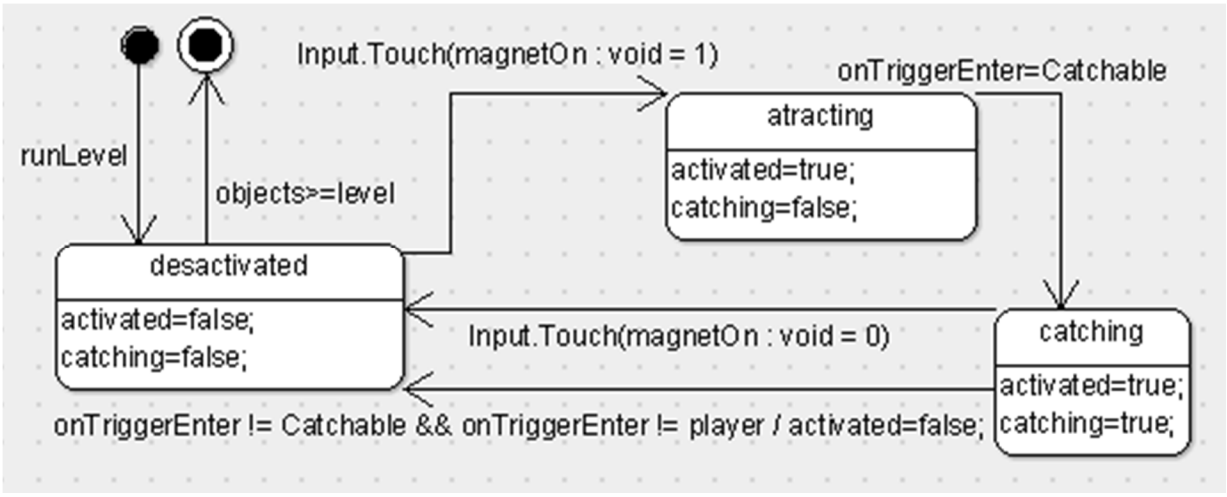

Fig 11. Diagrama de Estado del catcher.

Se dispone de un botón táctil para activar la atracción, al presionarse, la variable activated cambiará su valor a “true”, indicando que los objetos (catchables) que estén en el radio de atracción comenzaran a acercarse al cátcher o rayo de atracción, en cuanto el primero de estos entre en la zona de enlace, el catcher cambiará su estado a “catching”, asignando el correspondiente valor “true” a la variable que indicará dicho estado. El rayo de atracción o “Catcher” atrapará los objetos únicamente si la atracción está activada y no está portando todavía ningún objeto.

Un objeto objetivo “catchable” dejará de estar atrapado si el jugador presiona el botón para soltarlo, la variable activated asociada al botón que controla el catcher volverá a su valor inicial “false”.

Los objetos dejarán de estar atrapados si tocan cualquier otro collider de la escena, excluyendo otros objetos catchables y el propio helicóptero del jugador, esto modificará automáticamente el valor activated del cátcher.

Diagrama de estado de los objetos en cualquiera de las escenas de juego:

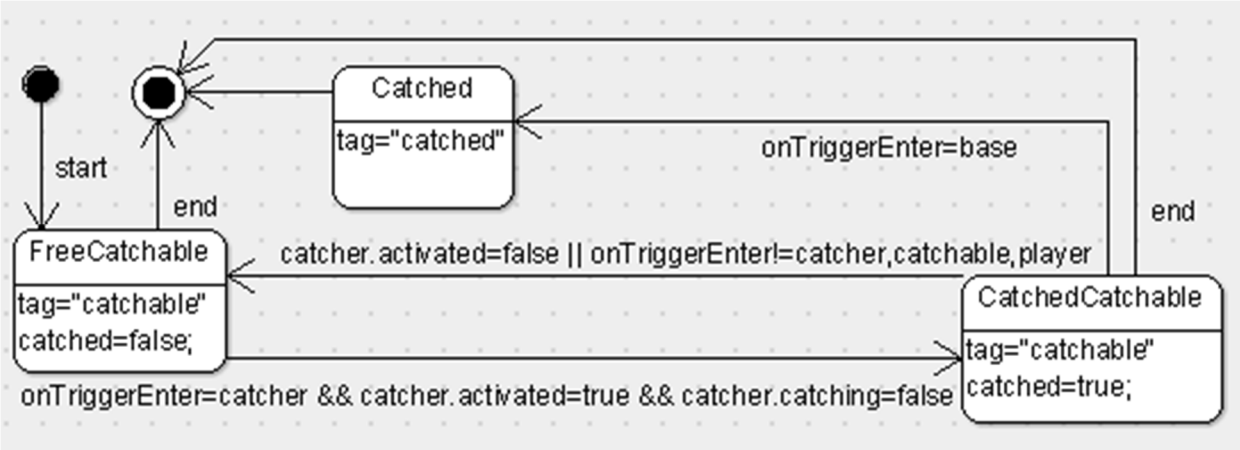

Fig 12. Diagrama de Estado de los objetos

Los objetos comienzan la partida con el tag “catchable”, y su variable catched con un valor “false”, lo que sería el estado “FreeCatchable”. Cuando un objeto en este estado entra dentro de la zona correspondiente al collider del catcher, se comprueba si el catcher tiene un valor “false” en su variable catching junto a un valor “true” en su variable activated, si se cumplen las dos condiciones, el objeto cambia al estado “CatchedCatchable” lo que significa que está atrapado y que imitara al catcher copiando sus movimientos.

Desde el estado “CatchedCatchable”, el objeto puede volver al estado inicial o bien porque el usuario lo ha decidido mediante el botón del cátcher, o bien porque se ha detectado una colisión con algún objeto con un tag diferente a “catcher”, “catchable”, “player” o “base”, provocando que el cátcher se desactive.

En caso de que la colisión sea con el objeto cuyo tag es “base”, se modificará el tag del objeto objetivo haciendo que su nuevo valor sea “catched”, lo cual implicará que dejará de verse afectado por la atracción, haciendo también imposible que vuelva a contabilizarse en el marcador de objetos recogidos

3.4. Diseño e Implementación:

En esta sección se explicará con detalle todo proceso de diseño e implementación.

Materiales y recursos necesarios para construir las escenas:

Las escenas principales son todas exactamente iguales, con la salvedad del lugar donde aparecen el helicóptero y los objetos objetivo.

Para poder producir la escena es necesario en primer lugar acudir a los programas de diseño y modelado 3d, con los que se crearán todos los modelos, las imágenes y las texturas necesarias. El primer paso era crear la biblioteca de objetos que iban a ser utilizados, el requisito indispensable era un bajo número de polígonos, para ello se realizaron gran parte de objetos desde cero, otros fueron casi completamente modificados eliminando partes de ellos y reduciéndolos con la herramienta “Maya 2011”, utilizando también el 3dsMax para los objetos que descargue en formato .3ds, tras las modificaciones todos fueron exportados en formato .fbx (un formato de archivo 3D independiente de la plataforma y gratuito, que proporciona acceso al contenido creado en cualquier paquete de software).

A continuación, unas imágenes de ejemplo sobre este proceso, acabando con el resultado final de la habitación que será utilizada como escenario, decorada al completo para hacer la experiencia de juego agradable, amena y divertida, además de realista.

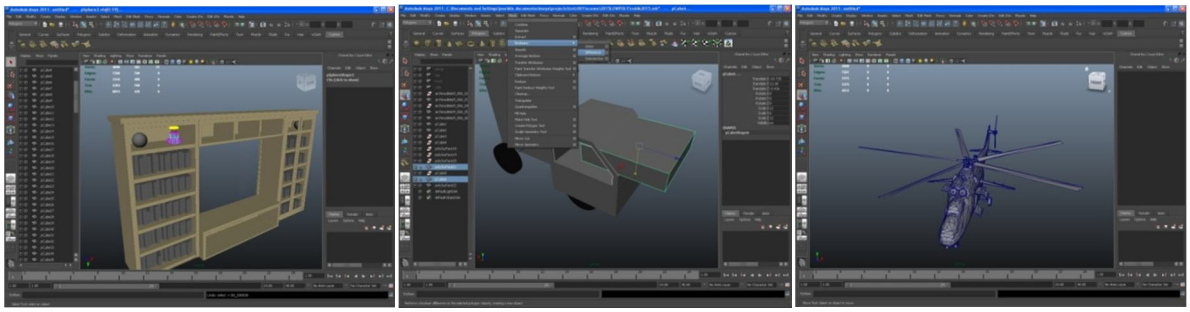

Fig 13. Capturas de pantalla del programa Maya 2011.

Fig 14. Helicoptero, juguetes capturables y vistas del escenario.

Tras el modelado se realizaron las texturas necesarias con el programa PhotoShop CS5, utilizando el plugin FilterForge, como se ve el resultado general es más que aceptable.

Tras esa fase y antes de comenzar con el desarrollo de la implementación, habrá que seguir utilizando el programa de imagen para realizar los botones y cuadros de mando que van a ser necesarios en el juego, incluidas todo tipo de todo tipo de imágenes que vayan a ser mostradas a lo largo de la partida.

En las siguientes imágenes se ven los resultados de este proceso, incluyendo las imágenes no seleccionadas, que aparecen bajo la equis roja, el resto se utilizarán en el juego.

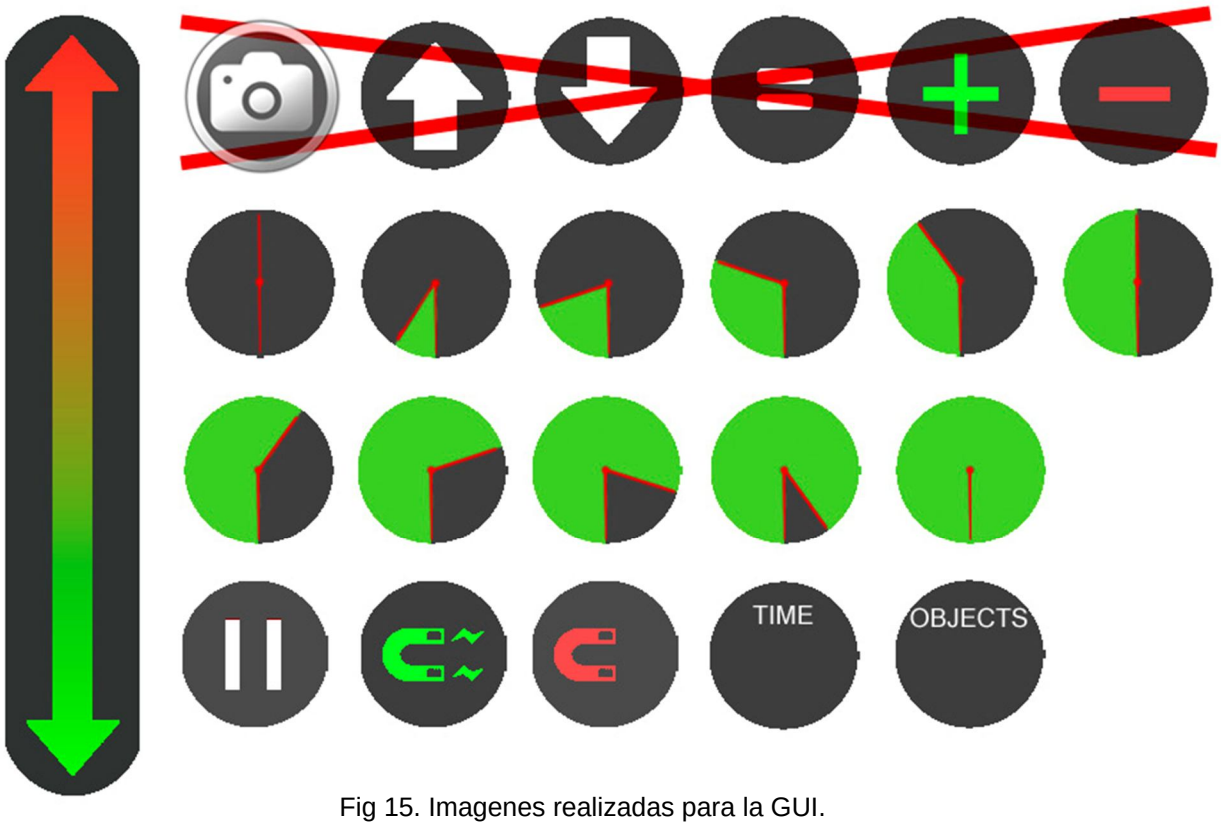

Fig 15. Imagenes realizadas para la GUI.

La barra vertical será la que permita modificar la potencia del helicóptero, cuyo estado podrá ser observable con el indicador de potencia.

Este indicador estará formado por las imágenes que aparecen en las 2 siguientes filas 2 y 3, después de la fila tachada con una equis roja.

La primera imagen de la tercera fila estará destinada a ser el botón de pausa, los dos siguientes serán para, capturar objeto/soltar objeto, y por último quedan las imágenes para el marcador de tiempo, y para marcador de objetos ya recogidos.

Algunos ejemplos de las imágenes que serán mostradas a lo largo de la partida

Fig 16. Imagenes mostrando el nivel.

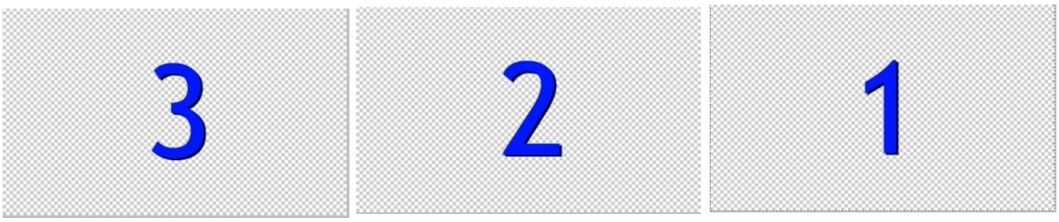

Fig 17. Imagenes de la cuenta atras.

Fig 17. Imagen al superar un nivel

Fig 19. Imagen al superar el ultimo nivel

Fig 20. Imagen de creditos.

Fig 21. Instrucciones del juego.

Una vez finalizados los recursos necesarios para implementar el juego, comenzaré a detallar las características de las escenas, con los objetos que las componen y los scripts asociados a estos.

Escena del menú principal:

El menú principal está compuesto de una serie de objetos contenidos en la vista de inicio, incluyendo el texto con el título del juego y los botones necesarios para ir a otras secciones.

Fig 22. Vista del menú principal.

Al tocar con el dedo el botón “play” se abrirá una ventana en la que podrá verse una imagen con los controles del juego, el objetivo es que todo jugador pueda ver las instrucciones aunque sus instintos le lleven a presionar el botón de jugar sin conocerlas, esta imagen dará paso al inicio del juego tras unos segundos en los cuales se estará cargando el nivel 1.

Al tocar el botón “credits”, se mostrará la sección correspondiente a los créditos, al tocar una segunda vez la pantalla se vuelve a la sección principal.

El botón “how to play” funciona de la misma manera que el resto, pero una vez tocado nos lleva a un conjunto de imágenes mostrando las instrucciones del juego, la forma de avanzar o retroceder entre estas imágenes es tocando con el dedo y arrastrando a la derecha o a la izquierda según se quiera la siguiente o la anterior imagen.

Finalmente el botón “exit” permite al jugador salir del juego mediante un simple contacto con el dedo.

A continuación se mostrarán los objetos y scripts que hacen posible este funcionamiento.

El menú está compuesto por objetos de diferente tipo:

En primer lugar, la cámara que permite al usuario ver la escena, esta cámara tendrá asociado un fondo de pantalla “background”, y un objeto “buttonsound” con el componente AudioSource.

El script encargado de controlar la escena irá indicándole en cada momento cual es la posición y dirección de la cámara, valiéndose de 2 objetos más “Camera Position” y “Camera Target”.

Los botones “PLAY”, “HOWTO”, “CREDITSLINK”, “EXIT”, tendrán asociado un TextMesh en el cual irá el texto a mostrar en cada botón.

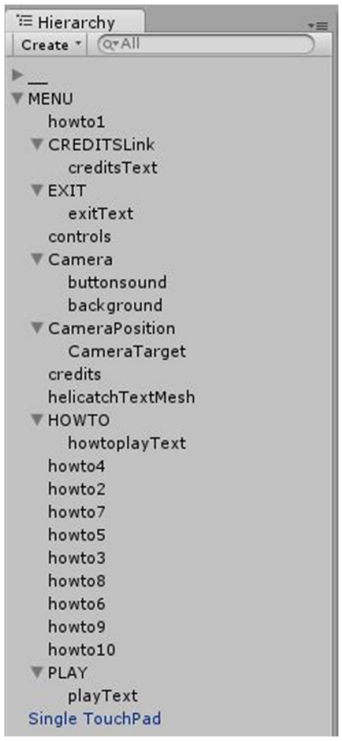

Fig 23. Objetos el menú.

Para detectar el contacto de un dedo, es necesario que los botones tengan como componente un Collider.

El objeto “Single TouchPad” es un objeto prefabricado, que puede obtenerse del package “Standard Mobile Assets”, se le ha colocado una textura completamente transparente, para que el jugador tenga la sensación de no estar ante ningún tipo de botón, simplemente tendrá que realizar movimientos de arrastre de dedos horizontalmente en la pantalla para poder interactuar, hacia la derecha para avanzar o izquierda para retroceder.

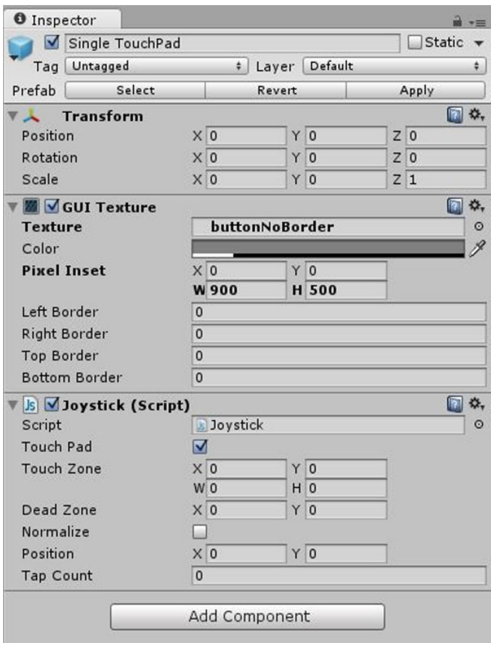

Fig 24. Objeto SingleTochPad.

El objeto “Single Touch” será utilizado únicamente para la sección de instrucciones tras haber hecho contacto en el botón: “how to play”, el script encargado de definir este comportamiento, es el mismo desde el cual se manejaran todas las secciones y objetos del menú, sea cual sea su tipo.

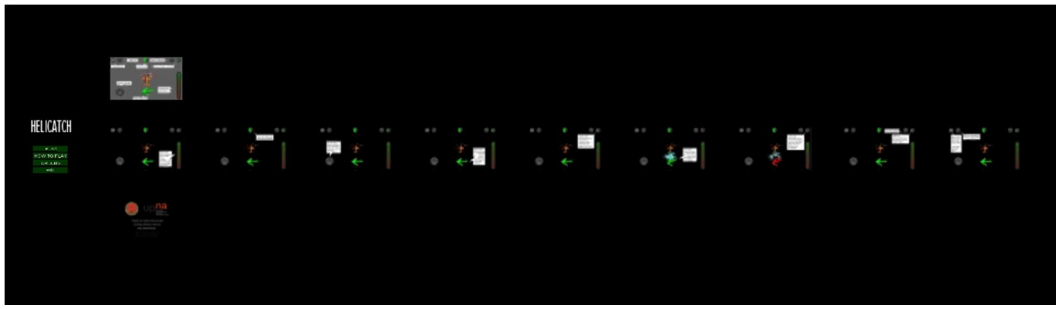

Fig 25. Perspectiva de los objetos del menu principal.

Este script, lo he llamado con el nombre de “menu.js”, y estará incluido como componente en la mayoría de los objetos que aparecen en esta escena y que serán descritos a partir de ahora, estará encargado de mover la cámara entre las diferentes secciones, cargar el nivel 1 o cerrar la aplicación dependiendo del botón que se pulse. El script en cuestión es el siguiente:

```
var touchControl: Joystick;     //objeto que permite detectar el arrastre de los dedos en la pantalla
var cameraPosition:GameObject;  // objeto que indica la posicion de la camara
var x:float;                    // valor para el eje x
var y:float;                    // valor para el eje y
var z:float;                    // valor para el eje z
var menuType:String;            //TYPES: PLAY, EXIT, LINK, HOWTOBEGIN
var guiStart: int;              //guiStart=0 (menu) | guiStart=1 (cargar escena 1)
var fase: int;                  //fase=0 (menu) | fase=1( cargar mensaje de espera)
var loadingSkin: GUISkin;       //letras y fondo de pantalla
var customGuiStyle : GUIStyle; //Estilo de letras
private var howtofase : int; //variable que indica la pagina de instrucciones actual y permite arrastar dedos
private var moving = false; //variable que indica si se esta cambiando de howtofase

function Start() {;

    Time.timeScale = 1;//Se activa el transito de tiempo a velocidad normal
    AudioListener.volume = 1;//Se activa el volumen
    howtofase = 0; //howtofase = 0 indica que se esta visionando la sección principal
}

//funcion que mueve la camara a las coordenadas indicadas
function linker(){
    yield WaitForSeconds(1); //Se espera 1 segundo
    //se asignan las coordenadas al objeto cameraPosition que indicara la posicion de la camara.
    cameraPosition.transform.position.x = x;
    cameraPosition.transform.position.y = y;
    cameraPosition.transform.position.z = z;
}

function startLevel(){
    yield WaitForSeconds(1);//Se espera 1 segundo
    Application.LoadLevel (1);//carga el nivel 1
    fase = 1;//cambia el valor de la variable fase a 1, para indicar que ya debe mostrarse el mensaje de espera
}

function OnGUI() {
    if  (guiStart == 1){  //Si guiStart == 1 -> la partida debe comenzar ya
        GUI.skin=loadingSkin; //se carga la skin que indica que se esta cargando el nivel
        if (fase == 0) {
            startLevel(); //startLevel() es la funcion que carga el nivel y cambia la variable fase a 1
        }
        if (fase == 1) {    //Si fase ha sido cambiada ya a 1.
            //Se muestra un mensaje por pantalla con el estilo "customGuiStyle" predefinido anteriormente
            GUI.Box(Rect (0,0,Screen.width,Screen.height), "WAIT A FEW SECONDS PLEASE", customGuiStyle);
        }
    }
}
```

```
//la funcion sera llamada al arrastrar el dedo hacia la derecha
function howtonext() {
    //Si la howtofase es < 10, y no se esta cambiando de howtofase (moving == false), se permite el cambio
    if ((moving==false) && (howtofase < 10)) {
        moving = true;//se activa mientras se realiza el cambio
        yield WaitForSeconds(0.5);//se espera medio segundo
        if (howtofase == 9){//si la howtofase es la ultima
            howtofase=0;    //se indica que la howtofase = 0 --> se estará visionando la seccion principal
            cameraPosition.transform.position.x = 327.9769; //se modifica la posicion de la camara respecto al eje X
        }
        else {//Si la howtofase no es la ultima, al arrastrar el dedo se avanza de howtofase
            howtofase += 1; //se aumenta de fase
            //se modifica la posicion de la camara respecto al eje X añadiendole 30 unidades
            cameraPosition.transform.position.x = cameraPosition.transform.position.x + 30;
        }
        moving = false;//finalmente se vuelve a reestablecer el valor original de moving, indicando que el cambio ha terminado
    }
}

//la funcion sera llamada al arrastrar el dedo hacia la izquierda
function howtoback() {
    //Si la howtofase es > 0, y no se esta cambiando de howtofase (moving == false), se permite el cambio
    if ((moving==false) && (howtofase > 0)) {
        moving = true;//se activa mientras se realiza el cambio
        yield WaitForSeconds(0.5); //se espera medio segundo
        if (howtofase == 1){//si la howtofase es la primera
            howtofase=0; //se indica que la howtofase = 0 --> se estará visionando la seccion principal
            cameraPosition.transform.position.x = 327.9769; //se modifica la posicion de la camara respecto al eje X
        }
        else {//Si la howtofase no es la primera, al arrastrar el dedo se vuelve a la howtofase anterior
            howtofase -= 1; //se reduce de fase
            //se modifica la posicion de la camara respecto al eje X restandole 30 unidades
            cameraPosition.transform.position.x = cameraPosition.transform.position.x - 30;
        }
        moving = false;//finalmente se vuelve a reestablecer el valor original de moving, indicando que el cambio ha terminado
    }
}

//la funcion de actualizacion se repite continuamente para actualizar el objeto en el que el script está como componente
function Update () {
    //Detecta y almacena la cantidad de toques detectados
    for (var i=0; i < Input.touchCount; ++i) {
        if (Input.GetTouch(i).phase == TouchPhase.Began) {
            var hit : RaycastHit; //Se prepara un rayo
            var ray = Camera.current.ScreenPointToRay(Input.GetTouch(i).position); //Se lanza desde la posicion tocada
            if (Physics.Raycast(ray, hit, Mathf.Infinity)) { //Si el rayo lanzado hasta el infinito detecta una colisión
                //Si esta ha sido contra un collider con menuType = EXIT
                if ((menuType == "EXIT") && (hit.collider.gameObject == this.gameObject))       {
                    Application.Quit();
                }
                //Si esta ha sido contra un collider con menuType = PLAY
                else if ((menuType == "PLAY") && (hit.collider.gameObject == this.gameObject))      {
                    guiStart = 1; //Se activa la variable que indicara a la funcion OnGUI que debe cargarse el nivel 1
                }
                //Si esta ha sido contra un collider con menuType = LINK
                else if ((menuType == "LINK") && (hit.collider.gameObject == this.gameObject))      {
                    linker(); //se mueve la camara a las coordenadas con las que linkea el objeto
                }
                //Si esta ha sido contra un collider con menuType = HOWTOBEGIN
                else if ((menuType == "HOWTOBEGIN") && (hit.collider.gameObject == this.gameObject))        {
                    linker(); //se mueve la camara a las coordenadas con las que linkea el objeto (página 1 de instrucciones)
                    howtofase=1; //esta variable indica que se esta visionando la página 1 de instrucciones
                }
            }
        }
    }
    //detecta si se ha arrastrado el dedo hacia la izquierda
    if ((touchControl.position.x < 0) && ((howtofase>0) && (howtofase<10))) {
        howtoback(); //Se lanza la funcion para ir a la página anterior de las instrucciones
    }
    //detecta si se ha arrastrado el dedo hacia la derecha
    else if ((touchControl.position.x > 0) && ((howtofase>0) && (howtofase<10))){
        howtonext(); //Se lanza la funcion para ir a la página siguiente de las instrucciones
    }
}
```

Con esto se puede dar por finalizada la explicación de la escena del menú que permitirá a los jugadores acceder a las escenas de juego.

Escenas de juego:

Las escenas de juego como se comentó anteriormente son todas exactamente iguales con la diferencia del lugar donde aparecen los objetos con capacidad de moverse por el espacio.

Esta escena lleva una gran cantidad de objetos y scripts que realizaran las funciones necesarias para comunicarse entre ellos a fin de garantizar que el juego funcione correctamente.

En primer lugar hay que posicionar los objetos estáticos en el mapa, creados anteriormente con los programas de diseño y modelado 3d, tras esto se le aplican las texturas realizadas en el programa de diseño de imágenes, y por último se añaden 2 tipos de Colliders, los normales que impedirán que un objeto normal traspase a otro, y los Triggered, cuya función será detectar colisiones mediante scripts, a fin de diferenciar entre los diferentes tipos de superficies ya que según se ha planeado los objetos objetivos deberán soltarse del cátcher cuando estos toquen cualquier otra superficie que no pertenezca a otro de ellos o al helicóptero en cuestión que los transporta.

Por ejemplo, observando este objeto (una cama litera), se puede observar de color verde la malla que indica las zonas de colisión o colliders, que pueden agregarse a cualquier objeto que se desee de la siguiente forma:

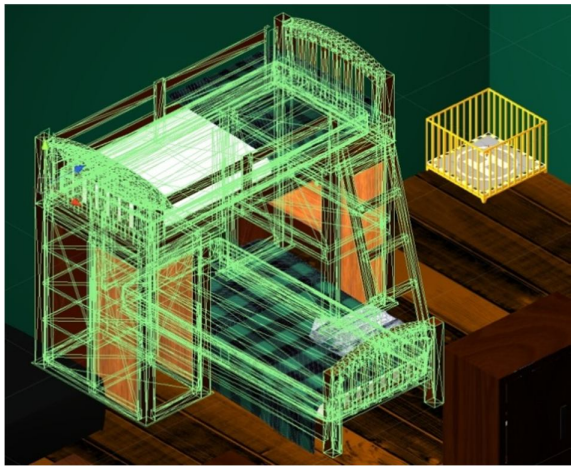

Fig 26. Objeto resaltando los colliders.

Se abre la pestaña de componentes, se selecciona la sección de componentes de física (physics), y se elige entre uno de los Colliders disponibles.

El box collider es el más componente más simple de este tipo y el que más utilizado, con la excepción de objetos complejos para los cuales se requiere del Mesh Collider, el cual se adapta la malla.

Como último paso, los colliders deben duplicarse, indicando "isTrigger = true".

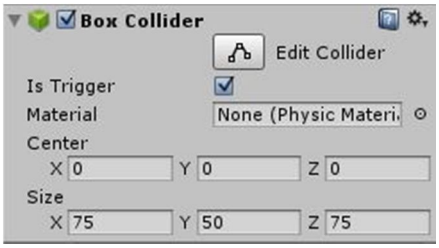

Fig 27. Propiedades de box collider.

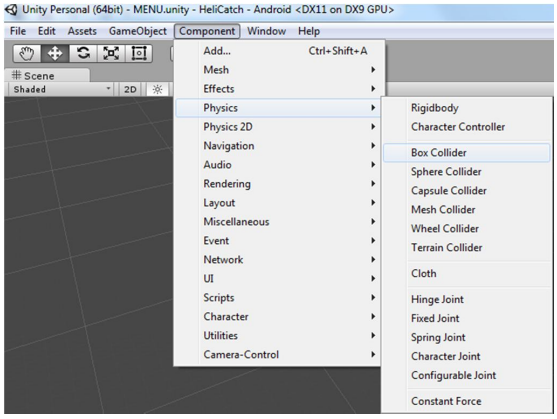

Fig 28. Ruta en Unity 3D para el box collider.

La habitación donde se desarrolla el juego está compuesta de múltiples objetos decorativos los cuales serán todos estáticos, y estarán englobados dentro de mapObjects1, mapObjects2, y mapbase el cual contiene el suelo, techo, y paredes.

Fig 29. Objetos del mapa.

Además de este tipo de objetos, son necesarias luces que iluminen la escena, incluidos efectos para crear un brillo indicador de la luminosidad.

Fig 30. Objeto con iluminación.

ObjectSpawn es un objeto que contiene a otros objetos, los cuales estarán repartidos a lo largo de toda la escena en diferentes posiciones, en total hay 19 posiciones, que serán los lugares donde aparecerán los objetos objetivo o “catchables” al iniciar cada partida. Esto se realiza mediante un script llamado “ObjectSpawn.js” que tendrá como variables las siguientes listas de objetos.

ObjectsToSpawn y SpawnLocations serán las encargadas de almacenar, en el primer caso los objetos a instanciar y en el segundo los lugares donde se instanciarán, uno en cada localización.

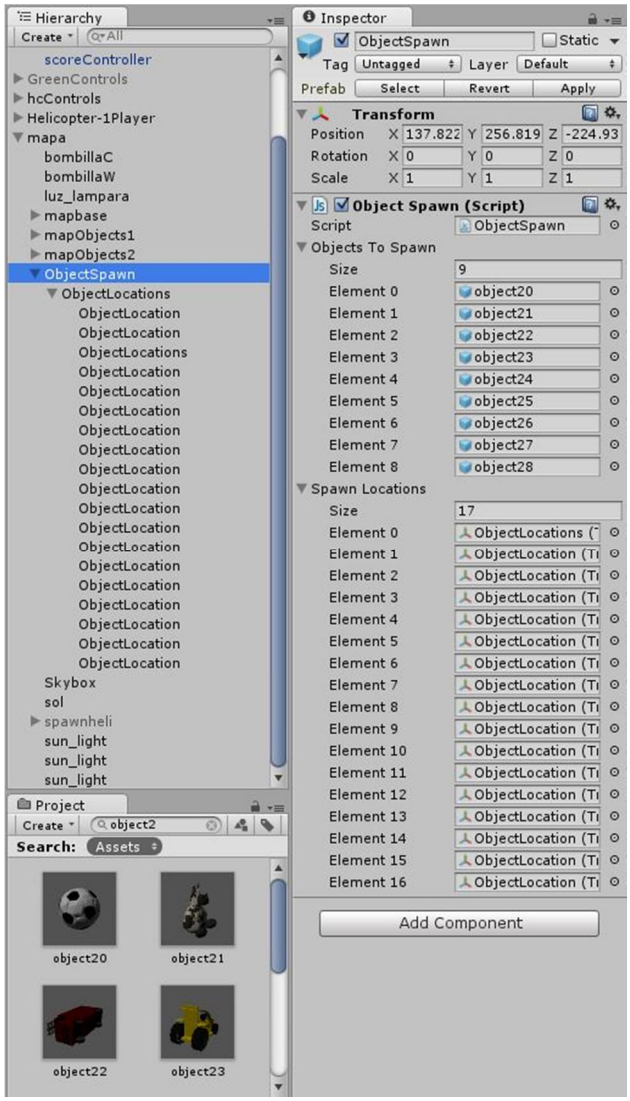

Fig 31. Objeto y script ObjectSpawn.

Este es el script: ”ObjectSpawn.js”

```
var objectsToSpawn : GameObject[]; //Objectos para instanciar
var spawnLocations : Transform[]; //lugares donde instanciar los objetos
private var selectedLocations : Transform[];
static var objectspawncomplete : boolean = false; //variable que indicará el fin del proceso

//Al comenzar la partida
function Start() {
    SpawnAllObjects(); //se lanza la función que instancia los objetos en lugares al azar
}

function SpawnAllObjects() {
    //se crea un numero aleatorio que definirá el lugar donde se instanciará el 1º objeto
    var placeToSpawn : int = Random.Range( 0, spawnLocations.length );
    //mientras no haya recorrido todos los objetos en lugares al azar
    for (var i=0; i < objectsToSpawn.length; i++){
        //a cada uno de los objetos
        var spawn = placeToSpawn + i; //se le da un lugar diferente para instanciarse
        //esto se hace mediante el modulo del número obtenido anteriormente con la longitud de la tabla
        //esto evitará que varios objetos aparezcan en el mismo punto ocasionando problemas de rendimiento
        //ya que se producían colisiones y movimientos extraños muy costosos de calcular.
        var elemento = modulo(spawn, spawnLocations.length);
        //Se instancia dicho objeto en el lugar donde haya decidido el azar
        Instantiate( objectsToSpawn[i], spawnLocations[elemento].position, transform.rotation );
    }
}

//devuelve el modulo de los valores que obtiene en la entrada de la función
function modulo(dividendo : int, divisor : int) : int {
     return (((dividendo) % divisor) + divisor) % divisor;
}

function Update () {

}
```

Los objetos a instanciar, llevarán un tag que los identificará como objetos de un mismo tipo, el nombre para identificarlos inicialmente es “catchable” como he dicho ya tantas veces, cambiando de tag a “catched” una vez hayan sido llevados hasta el lugar objetivo donde hay que depositarlos.

Para hacer esta labor es primordial dotar al helicóptero de capacidad para moverse por el espacio tridimensional para ello, tras situar el objeto del helicóptero que se ha importado de Maya 2011, se situará en un lugar cualquiera de la escena, y habrá que asegurarse de que sus ejes y punto de pivote están bien situados y orientados, tras esto habrá que dotarlo de controles. La orientación y posición del pivote es esencial como podrá verse claramente con el script.

Los controles como ya dije en una sección anterior, van a ser un Joystick digital para indicar la dirección de inclinación frontal y trasera, además de la rotación y una barra táctil para modificar la potencia en caso de querer subir o bajar a una altura diferente.

Fig 32. Controles táctiles del helicóptero.

Ambos controles llevarán el mismo script asociado como componente con la singularidad de que la barra de potencia tiene un valor “true” para la variable que indica que es un TouchPad.

El script encargado de manejar todo esto es “Joystick.js”

```
#pragma strict

@script RequireComponent( GUITexture )

// A simple class for bounding how far the GUITexture will move
class Boundary
{
    var min : Vector2 = Vector2.zero;
    var max : Vector2 = Vector2.zero;
}

static private var joysticks : Joystick[];                  // A static collection of all joysticks
static private var enumeratedJoysticks : boolean = false;
static private var tapTimeDelta : float = 0.3;              // Time allowed between taps

var touchPad : boolean;                                     // Is this a TouchPad?
var touchZone : Rect;
var deadZone : Vector2 = Vector2.zero;                      // Control when position is output
var normalize : boolean = false;                            // Normalize output after the dead-zone?
var position : Vector2;                                     // [-1, 1] in x,y
var tapCount : int;                                         // Current tap count

private var lastFingerId = -1;                              // Finger last used for this joystick
private var tapTimeWindow : float;                          // How much time there is left for a tap to occur
private var fingerDownPos : Vector2;
private var fingerDownTime : float;
private var firstDeltaTime : float = 0.5;

private var gui : GUITexture;                               // Joystick graphic
private var defaultRect : Rect;                             // Default position / extents of the joystick graphic
private var guiBoundary : Boundary = Boundary();            // Boundary for joystick graphic
private var guiTouchOffset : Vector2;                       // Offset to apply to touch input
private var guiCenter : Vector2;                            // Center of joystick

function Start(){
    // Cache this component at startup instead of looking up every frame
    gui = GetComponent( GUITexture );

    // Store the default rect for the gui, so we can snap back to it
    defaultRect = gui.pixelInset;

    defaultRect.x += transform.position.x * Screen.width;// + gui.pixelInset.x; // -  Screen.width * 0.5;
    defaultRect.y += transform.position.y * Screen.height;// - Screen.height * 0.5;

    transform.position.x = 0.0;
    transform.position.y = 0.0;

    if ( touchPad ) {
        // If a texture has been assigned, then use the rect ferom the gui as our touchZone
        if ( gui.texture )
            touchZone = defaultRect;
    }
    else    {
        // This is an offset for touch input to match with the top left
        // corner of the GUI
        guiTouchOffset.x = defaultRect.width * 0.5;
        guiTouchOffset.y = defaultRect.height * 0.5;
        // Cache the center of the GUI, since it doesn't change
        guiCenter.x = defaultRect.x + guiTouchOffset.x;
        guiCenter.y = defaultRect.y + guiTouchOffset.y;
        // Let's build the GUI boundary, so we can clamp joystick movement
        guiBoundary.min.x = defaultRect.x - guiTouchOffset.x;
        guiBoundary.max.x = defaultRect.x + guiTouchOffset.x;
        guiBoundary.min.y = defaultRect.y - guiTouchOffset.y;
        guiBoundary.max.y = defaultRect.y + guiTouchOffset.y;
    }
}

function Disable(){
    gameObject.active = false;
    enumeratedJoysticks = false;
}

function ResetJoystick(){
    // Release the finger control and set the joystick back to the default position
    gui.pixelInset = defaultRect;
    lastFingerId = -1;
    position = Vector2.zero;
    fingerDownPos = Vector2.zero;
    if ( touchPad )
        gui.color.a = 0.325;
}

function IsFingerDown() : boolean {
    return (lastFingerId != -1);
}

function LatchedFinger( fingerId : int ) {
    // If another joystick has latched this finger, then we must release it
    if ( lastFingerId == fingerId )
        ResetJoystick();
}
```

```
function Update(){
    if ( !enumeratedJoysticks ) {
        // Collect all joysticks in the game, so we can relay finger latching messages
        joysticks = FindObjectsOfType( Joystick ) as Joystick[];
        enumeratedJoysticks = true;
    }
    var count = Input.touchCount;
    // Adjust the tap time window while it still available
    if ( tapTimeWindow > 0 )
        tapTimeWindow -= Time.deltaTime;
    else
        tapCount = 0;

    if ( count == 0 )
        ResetJoystick();
    else    {
        for(var i : int = 0;i < count; i++)     {
            var touch : Touch = Input.GetTouch(i);
            var guiTouchPos : Vector2 = touch.position - guiTouchOffset;
            var shouldLatchFinger = false;
            if ( touchPad )         {
                if ( touchZone.Contains( touch.position ) )
                    shouldLatchFinger = true;
            }
            else if ( gui.HitTest( touch.position ) )   {
                shouldLatchFinger = true;
            }
            // Latch the finger if this is a new touch
            if ( shouldLatchFinger && ( lastFingerId == -1 || lastFingerId != touch.fingerId ) )            {
                if ( touchPad )             {
                    gui.color.a = 0.15;
                    lastFingerId = touch.fingerId;
                    fingerDownPos = touch.position;
                    fingerDownTime = Time.time;
                }
                lastFingerId = touch.fingerId;
                // Accumulate taps if it is within the time window
                if ( tapTimeWindow > 0 )
                    tapCount++;
                else {
                    tapCount = 1;
                    tapTimeWindow = tapTimeDelta;
                }
                // Tell other joysticks we've latched this finger
                for ( var j : Joystick in joysticks )   {
                    if ( j != this )
                        j.LatchedFinger( touch.fingerId );
                }
            }

            if ( lastFingerId == touch.fingerId )   {
                // Override the tap count with what the iPhone SDK reports if it is greater
                // This is a workaround, since the iPhone SDK does not currently track taps
                // for multiple touches
                if ( touch.tapCount > tapCount )
                    tapCount = touch.tapCount;
                if ( touchPad )     {
                    // For a touchpad, let's just set the position directly based on distance from initial touchdown
                    position.x = Mathf.Clamp( ( touch.position.x - fingerDownPos.x ) / ( touchZone.width / 2 ), -1, 1 );
                    position.y = Mathf.Clamp( ( touch.position.y - fingerDownPos.y ) / ( touchZone.height / 2 ), -1, 1 );
                }
                else        {
                    // Change the location of the joystick graphic to match where the touch is
                    gui.pixelInset.x =  Mathf.Clamp( guiTouchPos.x, guiBoundary.min.x, guiBoundary.max.x );
                    gui.pixelInset.y =  Mathf.Clamp( guiTouchPos.y, guiBoundary.min.y, guiBoundary.max.y );
                }
                if ( touch.phase == TouchPhase.Ended || touch.phase == TouchPhase.Canceled )
                    ResetJoystick();
            }
        }
    }
    if ( !touchPad ) {
        // Get a value between -1 and 1 based on the joystick graphic location
        position.x = ( gui.pixelInset.x + guiTouchOffset.x - guiCenter.x ) / guiTouchOffset.x;
        position.y = ( gui.pixelInset.y + guiTouchOffset.y - guiCenter.y ) / guiTouchOffset.y;
    }
    // Adjust for dead zone
    var absoluteX = Mathf.Abs( position.x );
    var absoluteY = Mathf.Abs( position.y );
    if ( absoluteX < deadZone.x )   {
        // Report the joystick as being at the center if it is within the dead zone
        position.x = 0;
    }
    else if ( normalize )   {
        // Rescale the output after taking the dead zone into account
        position.x = Mathf.Sign( position.x ) * ( absoluteX - deadZone.x ) / ( 1 - deadZone.x );
    }
    if ( absoluteY < deadZone.y )   {
        // Report the joystick as being at the center if it is within the dead zone
        position.y = 0;
    }
    else if ( normalize )   {
        // Rescale the output after taking the dead zone into account
        position.y = Mathf.Sign( position.y ) * ( absoluteY - deadZone.y ) / ( 1 - deadZone.y );
    }
}
```

Con estos controles ya en funcionamiento el siguiente paso es hacer volar al objeto que se ha diseñado para este propósito, como decía antes, es esencial tener bien posicionados sus ejes y el punto de pivote, además será necesario tener por separado 2 objetos que serán los encargados de girar simulando a los rotores, por supuesto que la orientación de los ejes y del punto de pivote de estos objetos también es esencial, ya que sino al girar provocarían un efecto no deseado y muy alejado de lo que sería la realidad.

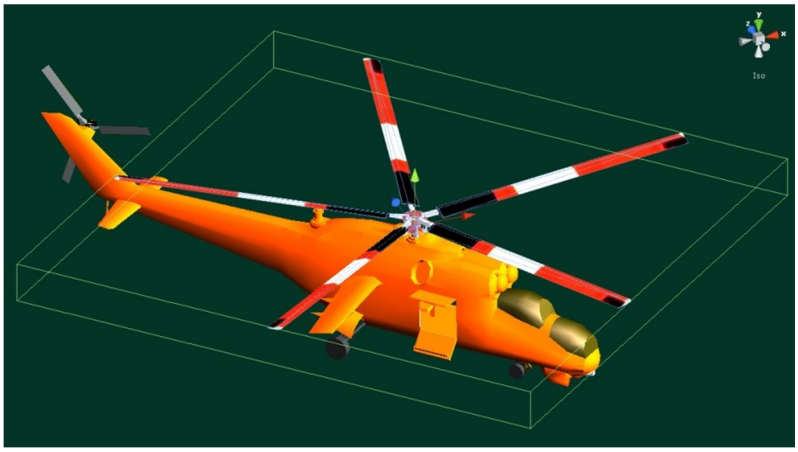

Fig 33. Vista del helicoptero con el rotor principal seleccionado.

El rotor 1 debe tener el eje “y” (verde) bien orientado y posicionado en el centro mientras que la orientación de los otros 2 ejes depende del ángulo de giro en cada instante sobre el eje vertical.

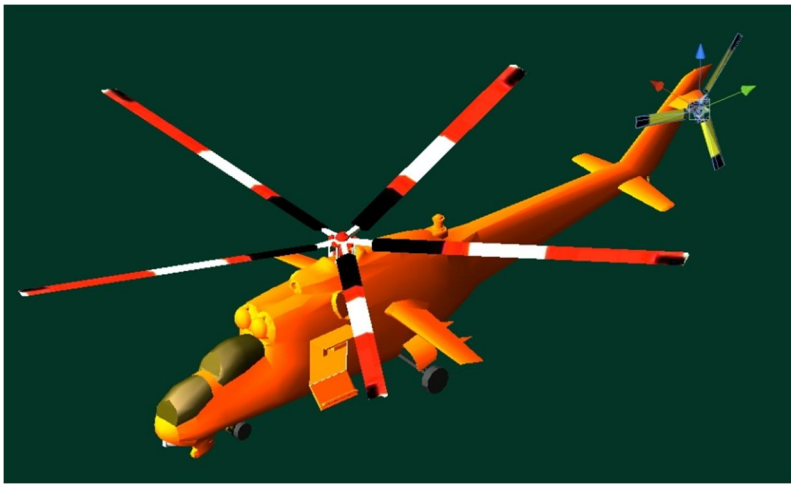

Fig 34. Vista del helicoptero con el rotor trasero seleccionado.

En cambio el rotor 2 debe tener el eje “x” (rojo) bien orientado y posicionado en el centro.

Una vez explicado esto se puede avanzar con el contenido del script que permitirá seguir aportando realismo al juego.

El script, llamado “helicopter1player.js”, irá asociado como componente al objeto principal que contiene a todos los demás que forman el helicóptero, incluido el AudioSource que permitirá producir el sonido de las hélices.

Este script es largo y complejo así que lo explicaré por partes.

```
var             directionControl    : greenControl; //control de direccion para pc (etapa de desarrollo)
var             powerControl        : greenControl; //control de potencia para pc (etapa de desarrollo)
//var           directionControl    : Joystick;     //control de direccion para dispositivos tactiles
//var           powerControl        : Joystick;     //control de potencia para dispositivos tactiles
public var      power               : float;        //porcentaje de potencia (rotor1) [0..1] controlada por jugador

private var     maxReachableForce   : float;        //maxima potencia alcanzable con el rotor1 al 100%
private var     powerToBalanceV     : float;        //potencia necesaria para estabilizarse verticalmente
private var     rotationSpeed       : float;        //velocidad de rotación de la animacion para los rotores

private var     rotor1_gameobject   : GameObject;   //Rotor principal que girara sobre el eje Y
private var     rotor2_gameobject   : GameObject;   //Rotor principal que girara sobre el eje Z
private var     guiTime_gameobject  : GameObject;   //Contiene el componente que gestiona el tiempo

function Start()
{
    //Anula el sonido del helicoptero
    audio.Stop();
    //Se busca en la escena el objeto llamado guiTime y este se asigna a la variable guiTime_gameobject
    guiTime_gameobject = GameObject.Find("guiTime");
    //Se busca en la escena el objeto llamado rotor1 y este se asigna a la variable rotor1_gameobject
    rotor1_gameobject = GameObject.Find("rotor1");
    //Se busca en la escena el objeto llamado rotor2 y este se asigna a la variable rotor2_gameobject
    rotor2_gameobject = GameObject.Find("rotor2");
    //Se establece la potencia inicial a 0
    power = 0;
    //la velocidad de giro de los rotores a 0
    rotationSpeed = 0;
    //  Un rigidbody de masa = 1kg, requiere de al menos una aceleración vertical de 9.81 m/s²
    // por cada kg para no caer por el efecto de la gravedad.
    //  Las especificaciones de un helicóptero civil real indican que la velocidad de ascenso sería
    // como minimo de 500 m/min a nivel del mar. En funcion del tamaño del escenario y de los objetos y de
    // una velocidad de ascenso aceptable se determinó que la potencia necesaria para la realizacion de un
    // vuelo axial ascendente seria entorno a 5 m/s² (considerando un rigidbody standar con masa = 1kg)
    // el valor se guardará en la variable = "maxReachableForce" (14.81 newtons, si power=1)
    maxReachableForce = 14.81;
    Time.timeScale = 1;
    AudioListener.volume = 1;
}
```

Inicialmente se crean las variables y se les da un valor inicial para permitir el funcionamiento correcto del script.

- directionControl y powerControl son controles que servirán para manejar ciertos movimientos del helicóptero,
- power: será la potencia aplicada según se manejen estos controles.
- maxReachableForce es la máxima fuerza vertical que se producirá cuando el valor de power sea máximo (1).
- powerToBalanceV es la potencia necesaria para mantener el helicóptero estable manteniendo la altura, se obtiene a partir del valor de la gravedad y del valor de maxReachableForce.
- rotationSpeed será la velocidad de giro para los rotores (rotor1 y rotor2)

```
// La función FixedUpdate es consistente sin importar el numero de fps del juego.
// Sirve para mantener estable el helicóptero en el aire evitando movimientos irregulares a causa de los fps
function FixedUpdate () {
    //Si la fase 6 no esta activa nose permite aplicar potencia a los rotores ni manejar la direccion
    if (guiTime_gameobject.GetComponent( guiTime ).fase != 6) {
        audio.Stop();
    }
    //Si la fase 6 esta activa (fase de juego "en marcha") se permite aplicar potencia a los rotores
    else if (guiTime_gameobject.GetComponent( guiTime ).fase == 6) {
        if (audio.isPlaying == false) {audio.Play();}
        //La potencia estará siempre comprendida entre 0 y 1 (siendo 1: el 100% de la potencia maxima aplicable)
        if ( power > 1.0 ) {
            power = 1.0;
        }else if ( power < 0.0 ) {
            power = 0.0;
        }
        //Velocidad de rotación de los rotores depende de la potencia aplicada y del deltaTime
        rotationSpeed += 1400 * power * Time.deltaTime;
        // Se aplica rotación a los gameobjects que representan los rotores sobre el eje y rotara el rotor1, y
        // sobre el eje x el rotor2 la velocidad de rotación será la indicada en la variable "rotationSpeed"
        rotor1_gameobject.transform.rotation = transform.rotation * Quaternion.Euler( 0, rotationSpeed, 0 );
        rotor2_gameobject.transform.rotation = transform.rotation * Quaternion.Euler( rotationSpeed, 0, 0 );
        // se equipara el tono del AudioSource al nivel de potencia actual (el AudioClip es el sonido producido el helicóptero)
        audio.pitch = power;

        //Si el jugador esta situado por encima de la altitud permitida
        if (this.transform.position.y  > 85)
        {   //Se reduce la potencia gradualmente hasta que llegue al 20% impidiendo que se aplique potencia suficiente
            //a los rotores como para permanecer en el aire (esto impide que el jugador choque contra el techo)
            if (power > 0.2){
                power -= 0.01;
            }
        }
```

Se ha usado una función de actualización Overridable llamada FixedUpdate que permitirá crear un movimiento fluido del helicóptero.

En primer lugar, se comprueba si ha comenzado ya fase de juego, si está todavía no ha comenzado el sonido estará apagado, esperando a que el juego comience, una vez la fase de juego esté activada, el sonido comenzara a escucharse.

En todo momento la potencia del helicóptero debe estar comprendida entre el 0% y el 100%, la velocidad de rotación de los rotores será una fracción de las 1400 revoluciones por minuto que se obtienen a potencia máxima. El valor de la fracción dependerá del porcentaje de potencia elegido por el usuario en cada momento. Esta rotación se aplica en igual medida al rotor 1 y al rotor 2 ya que no es indispensable que el funcionamiento sea exacto a un helicóptero en el mundo real, el sonido también ajustará su velocidad respecto a la potencia para dar la impresión de aumento o disminución de la potencia del motor. Si el jugador supera una altitud límite, la potencia se verá reducida al 20%, si esto sucede sonará una alarma como explicare más adelante en el objeto encargado de hacerla sonar.

```
//Si el jugador esta situado por debajo de la altitud permitida se permite aplicar potencia a los rotores
else {
    //Segun la forma en la que se presionen los controles se aplicara un nivel de potencia diferente
    //Si se presiona el control de dirección y de potencia simultaneamente hacia arriba
    if ((directionControl.position.y > 0.0) && (powerControl.position.y > 0.0)) {
        if (power < 1)  {//Si la potencia no es la maxima
            power += 0.2;//se aumenta hasta que sea maxima
        }
    }
    //Si se presiona el control de dirección hacia arriba y el de potencia hacia abajo
    else if ((directionControl.position.y > 0.0) && (powerControl.position.y < 0.0))  {
        power = 0.4;//potencia al 40%
    }
    //Si se presiona el control de dirección hacia abajo y el de potencia hacia arriba
    else if ((directionControl.position.y < 0.0) && (powerControl.position.y > 0.0)) {
        if (power < 1)  {//Si la potencia no es la maxima
            power += 0.2;//se aumenta hasta que sea maxima
        }
    }
    //Si se presiona el control de movimiento hacia abajo y el de potencia hacia abajo
    else if ((directionControl.position.y < 0.0) && (powerControl.position.y < 0.0))  {
        power = 0.4;//potencia al 40%
    }
    //Si se presiona el control de movimiento hacia arriba sin presionar el de potencia
    else if ( directionControl.position.y > 0.0 ) {

        power = 0.72;//potencia al 72%
    }
    //Si se presiona el control de movimiento hacia abajo sin presionar el de potencia
    else if ( directionControl.position.y < 0.0 )  {

        power = 0.72;//potencia al 72%
    }
    //Si se presiona solo el control de potencia hacia arriba
    else if ( powerControl.position.y > 0.0 ) {
        //Potencia maxima
        if (power < 1)//Si la potencia no es la maxima
        {
            //se aumenta
            power += 0.4;
        }
    }
    //Si se presiona solo el control de potencia hacia abajo
    else if ( powerControl.position.y < 0.0 )  {
        power = 0.4;//potencia al 40%
    }
```

Si el jugador está por debajo de la altura límite, se le permitirá controlar el helicóptero, haciendo que la variable power encargada de controlar la potencia, tenga diferentes valores según las indicaciones recibidas en los controles.

```
    //en caso de que el jugador no toque los controles (se estabiliza)
    else {
        if (power == 1) {power=0.3;}
        if (power == 0.4) {power=0.99;}
        //Si la potencia es mayor de la necesaria para estar estable
        if (power > powerToBalanceV-0.02){
            power -= 0.005;//se reduce
        }
        //Si por el contrario, la potencia es menor de la necesaria para estar estable
        else {
            power += 0.005;//se aumenta
        }
    }
}
//  Se aplica la fuerza vertical de 14.81 newtons/power. "maxReachableForce"= 14.81, "power"= [0..1]
rigidbody.AddRelativeForce( Vector3.up * maxReachableForce * power);

// Fracción de la potencia que es necesario aplicar al rotor 1 para mantener estabilidad vertical
// (será el porcentaje que permite aplicar 9.8 m/s² de aceleración vertical)
// La facilidad de acelerar un cuerpo varia dependiendo de la masa y de la gravedad
// El porcentaje necesario depende del máximo de fuerza aplicable con la potencia a 1
powerToBalanceV = (rigidbody.mass * Mathf.Abs( Physics.gravity.y ) / maxReachableForce);
// Con esto ya se podria ascender, descender y permanecer estable variando la potencia segun desee el jugador.
```

Si el jugador no toca los controles la potencia se debe estabilizar, aplicándose una fuerza opuesta a la gravedad del mismo valor, para lo cual será necesario utilizar un porcentaje de la fuerza máxima vertical que puede producir el helicóptero, para ello la potencia se regulará en el punto adecuado según el peso del aparato y la gravedad de la escena.

Tras esto solo queda indicar el sentido de giro en cada eje:

-Eje X: Moviendo el joystick hacia arriba, el helicóptero se inclina hacia adelante gracias a una rotación en este eje, al igual que si se mueve hacia abajo, la inclinación será hacia atrás.

- Eje Y: Moviendo hacia la izquierda o hacia la derecha, se aplica una rotación en el helicóptero sin que se incline en ninguna dirección del plano "XZ".

- Eje Z: Utilizando el acelerómetro del dispositivo, se rota sobre el eje z, lo cual provoca una inclinación lateral.

```
// Los rotores reales pueden variar el ángulo con el que estan enganchados al fuselaje pero por no añadir
// mas complejidad de la necesaria el fuselaje ira acoplado de forma estatica a los rotores
// La posicion con orientación en un espacio 3d en unity se representa mediante el tipo Vector3,
// (vector direccional: x,y,z) "rotorVector" será el vector direccional que indica cuanto y como
//variara la rotación del helicóptero en todos sus ejes (inclinaciones y rotaciones)
var rotorVector : Vector3 = Vector3(
    (5000000 * (directionControl.position.y * directionControl.position.y * directionControl.position.y)),
    (2500000 * (directionControl.position.x * directionControl.position.x * directionControl.position.x)),
    (5000000 * (Mathf.Exp((2.5 * Input.acceleration.y)*(2.5 * Input.acceleration.y)*(2.5 * Input.acceleration.y))
                - Mathf.Exp(-((2.5 * Input.acceleration.y)*(2.5 * Input.acceleration.y)*(2.5 * Input.acceleration.y))))
              /(Mathf.Exp((2.5 * Input.acceleration.y)*(2.5 * Input.acceleration.y)*(2.5 * Input.acceleration.y))
                + Mathf.Exp(-((2.5 * Input.acceleration.y)*(2.5 * Input.acceleration.y)*(2.5 * Input.acceleration.y)))))
);
//(exp((2.5*x)^3)-exp(-((2.5*x)^3)))/(exp((2.5*x)^3)+exp(-((2.5*x)^3))) = tanh ((2.5*x)^3)
var rotationVector : Vector3;
//se actualiza el vector direccional que indicara la rotación del helicóptero en el espacio
rotationVector = rotationVector + rotorVector;
//se aplica la rotación indicada por el jugador, en otras palabras (le dice al rigidbody hacia donde debe rotar)
rigidbody.AddRelativeTorque( rotationVector );
```

Para calibrar los controles e intentar llegar al ajuste perfecto, necesitaba funciones que se adaptaran lo máximo posible al tipo de movimiento que debería realizarse para manejar de forma cómoda el helicóptero.

En primer lugar el joystick resultaba tremendamente pequeño para manejarlo con precisión, la cuestión está en que un pequeño arrastre del controlador no provoque un gran giro del helicóptero, intentando lograr al mismo tiempo un giro considerable del helicóptero, en caso de que el usuario mueva totalmente el joystick.

Para ello usé la función x^3, la cual puede compararse con la función x, en el rango [-1,1]

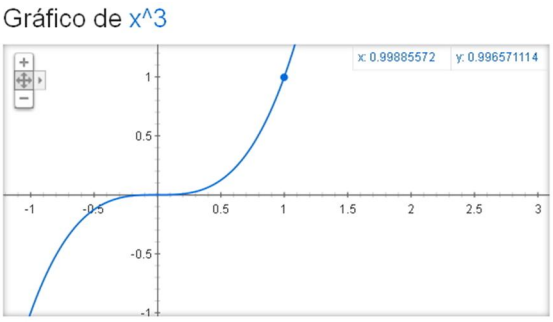

Fig 35. Grafica de la funcion y = x^3

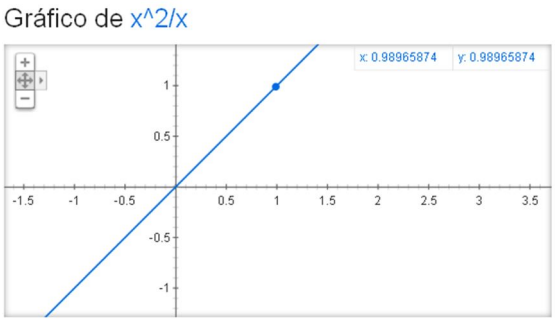

Fig 36. Grafica de la funcion y = x

Lo mismo sucede con el acelerómetro, sería deseable evitar que el jugador de forma inconsciente provoque pequeños giros del dispositivo de apenas unos grados, dando una inclinación no deseada del helicóptero, aunque en este caso también se desearía evitar tener que rotar el dispositivo 90 grados para poder llegar al máximo de inclinación con el helicóptero, para ello necesitaba una función concreta que se adaptase a mis necesidades. La función tangente hiperbólica, mediante unos pequeños ajustes se adapta de forma perfecta a lo que necesitaba.

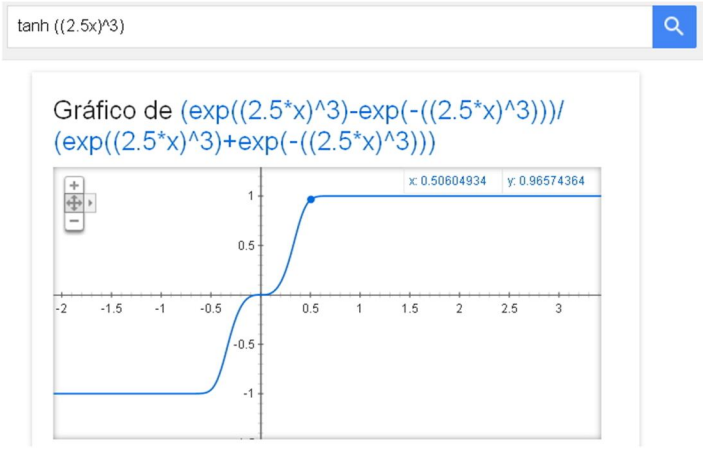

Fig 37. Grafica de la tangente hiperboidal.

Como se puede ver en la gráfica, esta función crece lentamente al encontrarse cerca del origen de coordenadas, aumentando su crecimiento hasta acercarse al punto 0.5, lo cual se vería traducido en 45º de rotación del dispositivo, a partir de este momento la función vuelve a crecer de forma muy reducida casi nula, permitiendo que aunque rotemos el móvil hasta los 90º, la fuerza de inclinación del helicóptero aumente en igual medida, esto ha sido esencial para poder utilizar el acelerómetro para realizar este tipo de movimiento en el juego,

lo cual además ha eliminado la necesidad de incluir los 2 tipos de movimientos en un único eje del joystick, lo cual no era deseable ya que en algunos casos se desearía aplicar rotación sin inclinación lateral o viceversa.

Además de aportar control sobre el helicóptero, el movimiento mediante el acelerómetro resulta atractivo y llamativo, y es un incentivo más para probar las habilidades del jugador.Las variables que almacenan el estado del acelerometro son: Input.acceleration.x, Input.acceleration.y, e Input.acceleration.z, perteneciendo estas a los correspondientes ejes según se ve en estas imágenes.

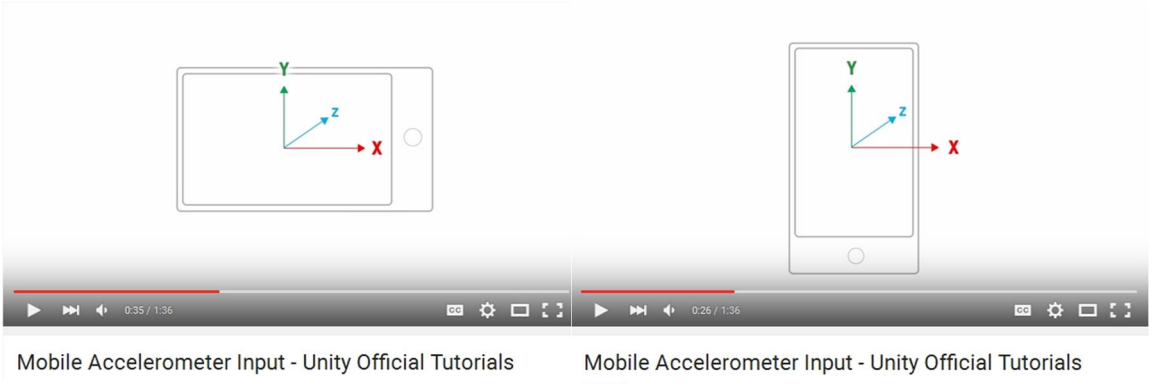

Fig 38. Ejes del acelerometro en un dispositivo Android.

Por último se comprueba el ángulo en el que se encuentra el helicóptero, para realizar las correcciones necesarias mediante rotaciones de los ejes que volverán a colocar al helicóptero en una posición sin inclinación en ninguno de los ejes “x” y “z”.

```
        //Comprueba el ángulo de inclinacion vertical del helicóptero
        if (Vector3.Angle( Vector3.up, transform.up ) < 60)
        {
            //Modifica el ángulo del helicóptero automaticamente haciendolo volver a la posición de estabilidad
            //Quaternion.Slerp: Interpola esféricamente del primer al segundo parámetro durante el tercero.
            //  Entradas: (from : Quaternion, to : Quaternion, t : float)
            //  from(origen): transform.rotation (rotación actual)
            //  to(objetivo): Quaternion (salida de la funcion Quaternion.Euler)
            //  t: 3 * Time.deltaTime, es la velocidad de recuperacion del ángulo,
            //      Time.deltaTime varia en funcion del lapse entre cada frame, asi el movimiento sera fluido
            //      el 2 implica que la recuperacion del ángulo sea mas rapida.
            //
            //  Salida: Quaternion -> que será el nuevo valor de posicion (rotación del objeto)
            ////////////////////////////////////////////////////////////////////////////////////////////
            //Quaternion.Euler: Devuelve en quaterniones una rotación pasada como parámetro en grados euler.
            //  Entradas: coordenadas (x, y, z) o un Vector3
            //      x -> 0  --> ángulo que garantiza la estabilidad en el eje x
            //      y -> transform.eulerAngles.y --> rotación actual del eje "y" (no se modifica ya que origen = objetivo )
            //      z -> 0  --> ángulo que garantiza la estabilidad en el eje z
            //  Salida:Quaternion
            transform.rotation = Quaternion.Slerp(
                                    transform.rotation,
                                    Quaternion.Euler(
                                            0,
                                            transform.rotation.eulerAngles.y,
                                            0
                                    ),
                                    3 * Time.deltaTime
                                );
        }
    }
}
```

Además de este script, y de los objetos para los rotores, rotores, el helicóptero requiere de otros objetos, como el objeto “guiPower” que será el encargado de mostrar la potencia en cada momento al jugador, mediante un marcador situado en el centro de la zona superior de la pantalla, además de hacer sonar una alarma en caso de que se supere una altura ya predefinina, con el objetivo de indicar al jugador que el techo está demasiado cerca y que se va a proceder a anular la potencia para evitar la colisión.

El objeto “guiPower” llevará asociado como componente otro AudioSource para reproducir la alarma, y un script para indicar el funcionamiento, además del AudioListener encargado de escuchar y grabar los sonidos del escenario para reproducirlos de nuevo al usuario, este componente suele ir asociado a una cámara, y debe ser único en la escena, pero para lograr un volumen máximo de la alarma junto al sonido del helicóptero, lo ideal es evitar tener el AudioListener en la cámara, ya que estará alejada de estos objetos y la distancia también será variable como se explicará más adelante. Por ello, la solución ideal es colocarlo en uno de estos objetos, por ejemplo dentro del objeto “guiPower” del que trata este apartado.

Este es el script que aportará la funcionalidad necesaria a este objeto, cuyo nombre es también “guiPower.js”.

```
//Este Script va asociado a un objeto vacio que contiene un Audio Source,
//El objeto se mueve junto al helicoptero cerca del AudioListener de la Camara.
//El Script controlara las texturas que seran mostradas al usuario para indicarle la potencia del helicoptero.

var power_texture : Texture[];  //Texturas para mostrar el nivel de potencia
private var player_gameobject : GameObject;     //Helicopter-1Player
private var guiTime_gameobject : GameObject;    //guiTime
private var currentPower : float;    //nivel de potencia


function Start()
{   //Se busca en la escena el objeto llamado Helicopter-1Player y este se asigna a la variable player_gameobject
    player_gameobject = GameObject.Find("Helicopter-1Player");
    //Se busca en la escena el objeto llamado guiTime y este se asigna a la variable guiTime_gameobject
    guiTime_gameobject = GameObject.Find("guiTime");
}

function OnGUI () {
    //Si la fase = 6 (fase de juego)
    if (guiTime_gameobject.GetComponent( guiTime ).fase == 6)
    {   //potencia actual: (se obtiene de la variable "power" perteneciente al script "helicopter1player.JS"
        currentPower = player_gameobject.GetComponent( helicopter1player ).power;
        //10 Texturas para mostrarnos la potencia actual mientras jugamos
        GUI.Label( Rect((Screen.width /2) - 35, 0, 70, 70), power_texture[ currentPower * 10 ] );
    }
}

function Update () {
    //Si el helicoptero no esta superando la altitud indicada y la alarma esta sonando
    if ((player_gameobject.transform.position.y < 85) && (audio.isPlaying))
    {
        //Apaga la alarma de altitud
        audio.Stop();
    }
    //en caso de que el helicoptero supere la altitud indicada y la alarma no este sonando
    else if ((player_gameobject.transform.position.y > 85) && (audio.isPlaying == false))
    {
        //Enciende la alarma de altitud
        audio.Play();
    }
}
```

Como puede verse, player_gameobject es el objeto que representa al helicóptero, al comenzar la partida, el marcador con la potencia obtiene las imágenes de una lista de texturas incluidas a mano desde la ventana del inspector en la variable power_texture[], dependiendo de la potencia en cada momento se mostrará la imagen correspondiente en el marcador. Si la altura supera el límite establecido sonará el AudioSource que contiene este objeto.

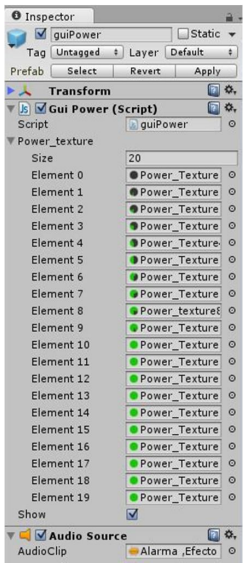

Fig 39. Objeto guiPower

Otro objeto del helicóptero indispensable para el juego, es el llamado “Catcher”, este objeto será el encargado de atraer y atrapar los objetos objetivo, estará posicionado frente al helicóptero y tendrá asociados 2 objetos con efectos de partículas, que se mostrarán cuando el jugador active la atracción, para lo cual se requieren 2 scripts que gestionen sus estados en función de las órdenes recibidas por el usuario y de lo sucedido en la escena.

El 1º efecto de partículas simula el campo de energía que provoca la atracción, requiere de tres componentes, Ellipsoid Particle Emiter, Particle Animator y Particle Renderer, que deben configurarse como se en las figuras 41, 42 y 43.

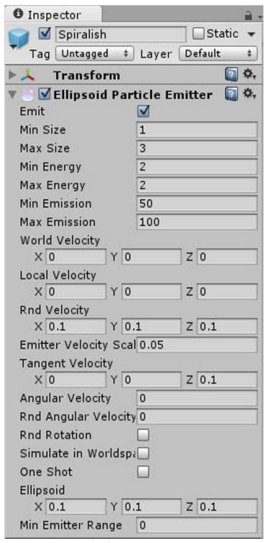

Fig 40. Ellipsoid Particle Emiter.

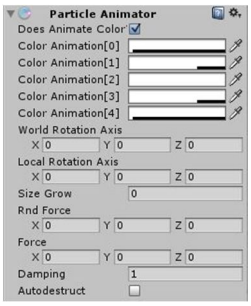

Fig 41. Particle Animator.

Fig 42. Particle Renderer.

El 2º efecto es el rayo que une al helicóptero con el campo de energía, y para el cual es necesario utilizar un “Line Renderer” además de cierta funcionalidad la cual vendrá definina en el script “rayoController.js”:

```
var target : Transform;      //objetivo del rayo
var boltWidth : float = 1;   //anchura del arco
var noise = 30.0;            //ruido
public var maxLength = 0.1; // longitud maxima del rayo
var color = Color.blue;      //color del rayo

private var lineRenderer : LineRenderer;
private var length : int; //variable que se actualiza con la longitud del rayo
private var pos : Vector3[];
private var lengthBeforeLightDisable = 10; // length of beam before mid and end light get disabled
public var activated:boolean=false;

function Start() {
    lineRenderer = GetComponent(LineRenderer); //Se guarda el componente LineRenderer en la variable
    lineRenderer.SetWidth(boltWidth, boltWidth); //se configura anchura
    lineRenderer.SetVertexCount(length); //se configura longitud
    activated = false; //se inicializa a falso
}

function Update() {
    RenderLightning(); //se lanza la funcion que muestra el rayo
}

function RenderLightning() {
    UpdateLength(); //se actualiza la longitud por si el objetivo se aleja
    if(target) transform.LookAt(target); // se mira constantemente al objetivo
    lineRenderer.SetColors(color, color); //  se modifica el color en tiempo real

    // reposiciona los elementos con un ligero desplazamiento
    for(var i=0; i<length; i++) {
        //noise será el ruido (o distorsion en la linea)
        var offset = Vector3(Random.Range(-noise, noise) * Time.deltaTime, Random.Range(-noise, noise) * Time.deltaTime, i);
        pos[i] = offset;
        pos[0] = Vector3.zero;
        pos[length - 1] = Vector3(0, 0, length - 1);//Se calcula la nueva posicion
        lineRenderer.SetPosition(i, pos[i]); //se aplica
    }
}

// actualiza la longitud del arco y la posicion del objetivo hacia donde se
function UpdateLength() {
    var hit : RaycastHit;
    var fwd = transform.TransformDirection(Vector3.forward);

    if(Physics.Raycast (transform.position, fwd, hit, maxLength)) length = Mathf.Round(hit.distance) + 1;
    else length = maxLength;

    pos = new Vector3[length];
    lineRenderer.SetVertexCount(length);
}
```

Se utilizan 2 rayos de diferentes colores con material “laser” que contiene una textura en escala de grises que otorga al rayo un aspecto de brillo interior.

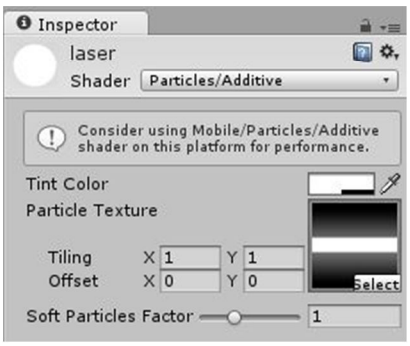

Fig 43. Material con textura laser.

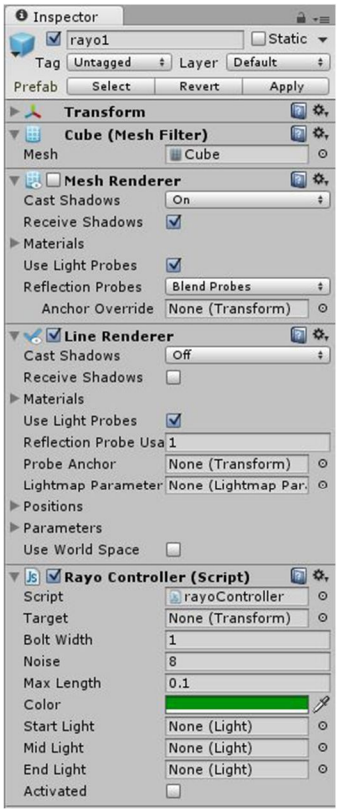

Fig 44. Rayo color verde.

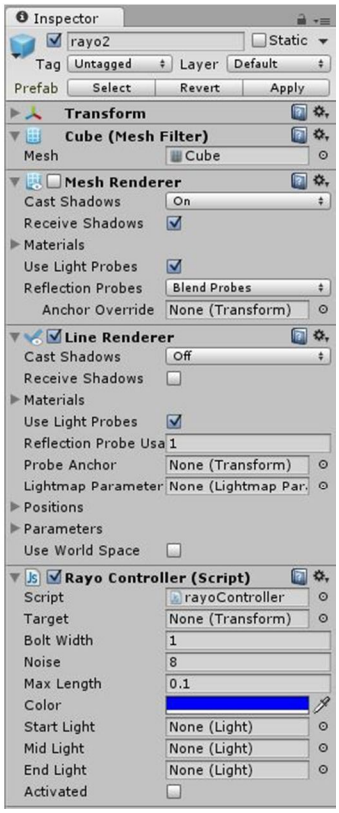

Fig 45. Rayo color azul.

El script asociado al cátcher se llama “catcherController.js”:

```
var         effect_gameobject   :GameObject;//objeto con emisor de particulas (efecto atracción activada)
public  var catching    : boolean = false;  //variable que indica si se esta transportando algun objeto atrapado
public  var effect      : boolean = false;  //variable que indicara si el efecto del catcher esta activado o desactivado
public var activated : boolean = false;     //variable que indicara si el catcher esta activado o desactivado
public var atracting    = false;            //variable que indica si la atracción esta activada
private var radius      : float = 100;      //variable que indica el radio de atracción
private var power       : float = -60;      //variable que indica la fuerza de atracción ()
private var effectposition   : Vector3;     //variable que indica la posición donde debe mostrarse el efecto visual
private var guiCatcher_gameobject : GameObject;  //objeto que contiene el boton para activar/desactivar la atracción

function Start() {
    //se fija la posición donde debe mostrarse el efecto visual
    effectposition = transform.position + Vector3(0,0,0);
    effect_gameobject = Instantiate (effect_gameobject, effectposition, transform.rotation);//se instancia el efecto
    //se desactiva el emisor de particulas para poder activarlo cuando se active la atracción
    effect_gameobject.GetComponent(ParticleRenderer).enabled = false;
    //se almacena el objeto "guiCatcher" en una variable para su uso posterior
    guiCatcher_gameobject = GameObject.Find("guiCatcher");
    catching  = false;          //variable que indica si se esta transportando algun objeto atrapado
    effect = false;
    atracting = false;
    activated = false;
}

//Mientras el collider del objeto esté en la zona que define la posicion del catcher
function OnTriggerEnter(other : Collider){
    //Si objeto es un "catchable" y esta atrapado
    if ((other.tag == "catchable") && (catching == false) && (activated)) {
        other.GetComponent( objectController ).catched=true;
        catching = true;
        atracting = false;
    }
}

//Mientras el collider del objeto esté en la zona que define la posicion del catcher
function OnTriggerStay(other : Collider){
    //Si objeto es un "catchable" y esta atrapado
    if ((other.tag == "catchable") && (catching == false) && (activated)) {
        other.GetComponent( objectController ).catched=true;
        catching = true;
        atracting = false;
    }
    if ((other.tag == "catchable") && (activated==false)) {
        other.GetComponent( objectController ).catched=false;
        catching = false;
        atracting = false;
    }
}

//La función FixedUpdate es consistente sin importar el numero de fps del juego.
//Sirve para dotar de atraccion a los objetos evitando movimientos irregulares a causa de los fps variables
function FixedUpdate () {
    if (activated) {
        if (effect == false) effect = true;
        if ((catching) && (atracting)){
            atracting =false;
        }
        else if ((catching==false) && (atracting==false)) atracting=true;
    }
    else {
        if (atracting == true) atracting = false;
        if (effect == true) effect = false;
        if (catching == true) catching = false;
    }
    //Se actualiza la posicion del efecto de atraccion respecto al movimiento del helicoptero
    effectposition = transform.position + Vector3(0,0,0);
    effect_gameobject.transform.position=effectposition;
    if (effect){
        if (effect_gameobject.GetComponent(ParticleRenderer).enabled==false)
            effect_gameobject.GetComponent(ParticleRenderer).enabled = true;
    }
    else {
        if (effect_gameobject.GetComponent(ParticleRenderer).enabled)
        effect_gameobject.GetComponent(ParticleRenderer).enabled = false;
    }

    if (catching) {
        if (effect == false) effect = true;
        if (atracting == true) atracting = false;
    }
    if (atracting) {
        var catcher = transform.position; //se almacena la posicición del catcher
        //se guardan en un array los colliders de todos los objetos en un radio
        var colliders : Collider[] = Physics.OverlapSphere (catcher, radius);
        //en la variable dist se irá almacenando la distancia del objeto con el que se trate
        var dist : float;
        for (var hit in colliders) {
            if (!hit)
                continue; //Si no hay objetos en el radio se sale del bucleç
            //en caso de que: hay objetos en el radio, éstos son "catchables" y la atraccion está activada
            if (hit.rigidbody && hit.tag == "catchable" && atracting)
            {
                //Se obtiene la distancia entre el objeto y el catcher
                dist = Vector3.Distance(hit.transform.position, catcher);
                //Se define una fuerza de atracción dependiendo de la distancia del objeto (cuanto más cerca mas fuerte)
                power = Mathf.Abs( 1/(dist/16) ) * (-500);
                //Se aplica la fuerza a los rigidbody que estaban en el radio de acción
                hit.rigidbody.AddExplosionForce(power, catcher, radius, 0.0);
            }
        }
    }
}
```

La variable “guiCatcher_gameobject” sirve para identificar al objeto encargado de mostrar el botón, el cual servirá para activar y desactivar la atracción. FixedUpdate será la función Overridable encargada de actualizar el estado de las variables con el paso del tiempo. Si la variable publica “activated” se activa con el valor true, y el catcher no tiene atrapado ningún objeto objetivo, la atracción afectará a todos los objetos objetivo que se encuentren en un radio menos al indicado (60 unidades) mostrándose el efecto de partículas, el efecto de atracción en el objeto será menor cuanto mayor sea la distancia que lo separa del helicóptero.

Si algún objeto se acerca lo suficiente como para entrar en la zona del collider asociado al cátcher y la atracción sigue activa, como se comprueba en la función onTriggerEnter, el catcher cambiará su estado a “catching”, y el objeto objetivo que haya entrado en la zona del catcher cambiará su estado a “catched”. Mientras el objeto permanezca en la zona del catcher, la función onTriggerStay se asegurará de mantener el valor correcto de las variables que definen el estado del cátcher. La fuerza de atracción producida en los objetos será la resultante de aplicar la función y=1/(x/16). Siendo x la distancia entre el cátcher y el objeto que se ve atraído.

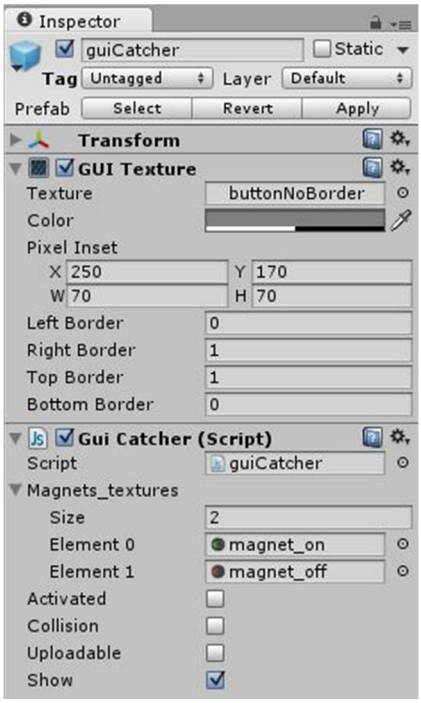

Fig 46. Objeto guiCatcher.

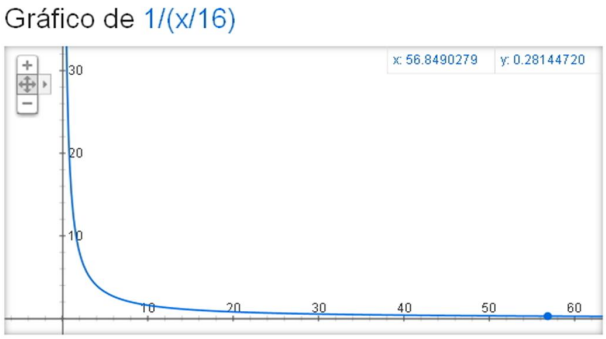

Fig 47. Grafico de la funcion y = 1/(x/16)

Como comentaba, “activated” es una variable pública, que sirve para activar la atracción cuando no se está transportando ya ningún objeto, esto se realiza a través del componente GUITexture de un objeto llamado guiCatcher, el cual antes se había asociado con la variable guiCatcher_gameobject.

Como decía, este componente GUITexture asociado al guiCatcher, es una imagen transparente que servirá para detectar el contacto del dedo al pulsar el botón, dependiendo del estado del catcher anteriormente comentado, se mostrará una de las texturas guardadas de la lista Magnets_textures que contiene el script “guiCatcher.js”, junto al resto del código necesario para hacer funcionar los botones y el cátcher de forma correcta. El script es el siguiente:

```
var magnets_textures    : Texture[]; //texturas que serviran de botones para activar/desactivar la atracción
private var catcher_gameobject  : GameObject; //catcher: objeto que atrae y atrapa los objetos objetivo "catchables"
private var rayo1_gameobject : GameObject;  //objeto asociado al rayo1 del catcher
private var rayo2_gameobject : GameObject;  //objeto asociado al rayo1 del catcher
private var guiTime_gameobject  : GameObject; //catcher: objeto que atrae y atrapa los objetos objetivo "catchables"
public var activated : boolean = false; //variable que indica a la textura si esta activado el boton (textura verde)
public var collision : boolean = false; //variable que indica al boton que ha habido una colision
private var uploadable : boolean = false; //variable que indica al boton si puede pulsarse
public var show : boolean = true;       //variable que indica si la textura debe mostrarse

function Start() {
    //Se busca en la escena el objeto llamado catcher y este se asigna a la variable catcher_gameobject
    catcher_gameobject = GameObject.Find("Catcher");
    guiTime_gameobject = GameObject.Find("guiTime");
    //Se buscan en la escena los objetos llamados rayo1 y rayo2 y se asignan a sus variables gameobject
    rayo1_gameobject = GameObject.Find("rayo1");
    rayo2_gameobject = GameObject.Find("rayo2");
    //Inicialmente la atracción estará desactivada y se mostrará el botón para poder activarla
    activated = false;
    uploadable = true;
    show = true;
}

function OnGUI () {
    if (show) {
        if (activated)
        {//Se muestra el botón para poder desactivarla
            GUI.Label( Rect( Screen.width-70, 0, 70, 70), magnets_textures[1] );
        }//en caso contrario, la atracción esta desactivada
        else
        {//se muestra el botón para poder activarla
            GUI.Label( Rect( Screen.width-70, 0, 70, 70), magnets_textures[0] );
        }
    }
}

function magnetChanger () {
    if  ((activated) && (uploadable)) {
        uploadable = false;
        catcher_gameobject.GetComponent( catcherController ).activated=false;
        rayo1_gameobject.GetComponent( rayoController ).maxLength=0.1;
        rayo2_gameobject.GetComponent( rayoController ).maxLength=0.1;
        activated = false;
    }
    else if ((activated==false) && (uploadable)) { //Si la atracción esta desactivada
        uploadable = false;
        catcher_gameobject.GetComponent( catcherController ).activated=true;
        rayo1_gameobject.GetComponent( rayoController ).maxLength=10;
        rayo2_gameobject.GetComponent( rayoController ).maxLength=10;
        activated = true;
    }
    yield WaitForSeconds(1);
    uploadable = true;
}

function objectCollision () {
    if  (activated) {
        catcher_gameobject.GetComponent( catcherController ).activated=false;
        rayo1_gameobject.GetComponent( rayoController ).maxLength=0.1;
        rayo2_gameobject.GetComponent( rayoController ).maxLength=0.1;
    }
    yield WaitForSeconds(1);
    activated = false;
}

function Update () {
        //Si la plataforma donde se ejecuta el juego es de Android o IOs
        if ((Application.platform == RuntimePlatform.Android) || (Application.platform == RuntimePlatform.IPhonePlayer))
        {  //Si se recibe un toque en la pantalla tactil sobre el componente GuiTexture adjunto al objeto
           if(Input.touchCount > 0) {
               for (var i=0; i < Input.touchCount; i++) {
                   var touch : Touch = Input.GetTouch(i);
                   if (touch.phase == TouchPhase.Began && guiTexture.HitTest(touch.position))
                       if (guiTime_gameobject.GetComponent( guiTime ).fase==6)
                           if (uploadable) magnetChanger(); //se llama a la funcion que activa/desactiva la atracción
               }
           }
        }
        else //Si la plataforma donde se ejecuta el juego es un PC
        {  //Si se hace click sobre el componente GuiTexture adjunto al objeto
            if(Input.GetMouseButton(0) && guiTexture.HitTest(Input.mousePosition))
              if (guiTime_gameobject.GetComponent( guiTime ).fase==6)
                  if (uploadable) magnetChanger(); //se llama a la funcion que activa/desactiva la atracción
        }
        if (collision) {
           collision=false;
           objectCollision();//se llama a la funcion que activa/desactiva la atracción
        }
}
```

El script crea variables para la lista de texturas, los objetos a los que necesitará acceder y para indicar si hay que mostrar una textura u otra, si se hace un Input.Touch sobre la GUITexture asociada en el objeto, se lanza la función magnetChanger, que dependiendo del estado de la variables “activated” y “uploadable”, realiza las acciones oportunas para activar o desactivar la atracción producida por el script del cátcher.

Ahora que ya se han explicado todos los controles que permiten interactuar con el entorno, solo faltaría comentar el objeto encargado de mostrar el botón pausa para poder detener el tiempo. Al pulsarlo, aparecerá un menú desde el cual se puede reiniciar el escenario o salir del juego, además de poder continuar anulando la pausa.

El objeto funciona con un GUITexture transparente para detectar el Input.Touch, asociado al objeto al igual que el objeto guiCatcher, la imagen del botón se mostrará como una textura de fondo y se encuentra almacenada en la variable pause_texture del script “guiPause.js” que explicaré a continuación.

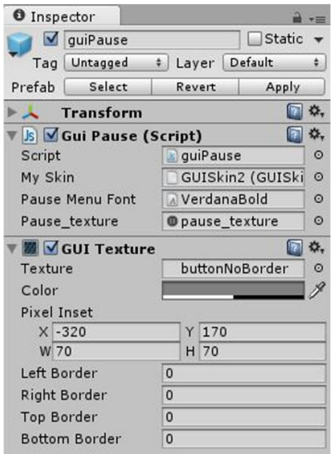

Fig 48. Objeto guiPause.

```
var mySkin : GUISkin;   //skin para el fondo del menu
var pauseMenuFont : Font;   //fuente de las letras para el menu
var pause_texture   : Texture; //textura para mostrar como imagen del boton de pausa
private var paused = false; //variable que indica si el juego esta pausado
private var guiTime_gameobject : GameObject; //objeto que gestiona el tiempo

function Start () {
    //Se busca en la escena el objeto llamado guiTime y este se asigna a la variable guiTime_gameobject
    guiTime_gameobject = GameObject.Find("guiTime");
}

function OnGUI(){
    //Si la fase = 6 (fase de juego)
    if (guiTime_gameobject.GetComponent( guiTime ).fase == 6) {
        GUI.skin.box.font = pauseMenuFont;
        GUI.skin.button.font = pauseMenuFont;
        GUI.Label( Rect(0, 0, 70, 70), pause_texture );
        if(paused == true)//Si se activa la pausa (paused == true)
        {   //Titulo del menu "Pause Menu"
            GUI.Box(Rect(Screen.width /2 - 200,Screen.height /2 - 150,400,350), "Pause Menu");
            //Boton 1: Continuar
            if(GUI.Button(Rect(Screen.width /2 - 200,Screen.height /2 - 100,400,100), "Continue")){
                //Se reestablecen los valores de sonido y de escala de tiempo a 1, y se almacena el valor false en la variable
                paused = false;
                AudioListener.volume = 1;
                Time.timeScale = 1;
            }
            //Boton 2: Reiniciar
            if(GUI.Button(Rect(Screen.width /2 - 200,Screen.height /2 ,400,100), "Restart")){
                //se almacena el valor false en la variable y se carga de nuevo el nivel actual
                paused = false;
                Application.LoadLevel(Application.loadedLevel);
            }
            //Boton 3: Salir del juego
            if (GUI.Button (Rect (Screen.width /2 - 200,Screen.height /2 + 100,400,100), "Quit Game")){
                Application.Quit();
            }
        }
    }
}
```

```
function Update () {
     //Si la aplicacion se esta ejecutando en un dispositivo Android o IOs
     if ((Application.platform == RuntimePlatform.Android) || (Application.platform == RuntimePlatform.IPhonePlayer))
     {  //Si detecta contacto en la pantalla
        if(Input.touchCount > 0) {
            //Cuenta el tiempo que se esta haciendo contacto con el dedo sobre la pantalla sin soltarlo
            for (var i=0; i < Input.touchCount; i++) {
                var touch : Touch = Input.GetTouch(i);
                //Si el contacto se hace sobre el componente "GUITexture" del objeto
                if (touch.phase == TouchPhase.Began && guiTexture.HitTest(touch.position)) {
                    //Se activa la pausa deteniendo el avance del tiempo y el sonido, ademas de almacenando el valor true en una variable
                    paused = true;
                    AudioListener.volume = 0;
                    Time.timeScale = 0;
                }
            }
        }
     }
     else //si por el contrario es un PC
     {   //Al hacer click sobre el componente "GUITexture" del objeto
         if(Input.GetMouseButton(0) && guiTexture.HitTest(Input.mousePosition))
         {
            //Se activa la pausa deteniendo el avance del tiempo y el sonido, ademas de almacenando el valor true en una variable
            paused = true;
            AudioListener.volume = 0;
            Time.timeScale = 0;
         }
     }
}
```

La función Update se lanza continuamente buscando detectar si se ha pulsado el GUITexture, en ese caso se activa la variable paused dándole un valor true. La función OnGui que también se lanza continuamente detecta el cambio en la variable paused y muestra el menú con las 3 opciones anteriormente mencionadas.

El tiempo es esencial en el juego, es el que marca el momento en el que la partida habrá terminado sin éxito si es que llega a 0, por eso es importante pausarlo si el jugador piensa desatender la partida. El objeto que gestiona el tiempo se llama “guiTime”, y lleva un script asociado de nombre: “guiTime.js” , el cual gestionará las fases de juego y reproducirá los sonidos de inicio y fin de partida

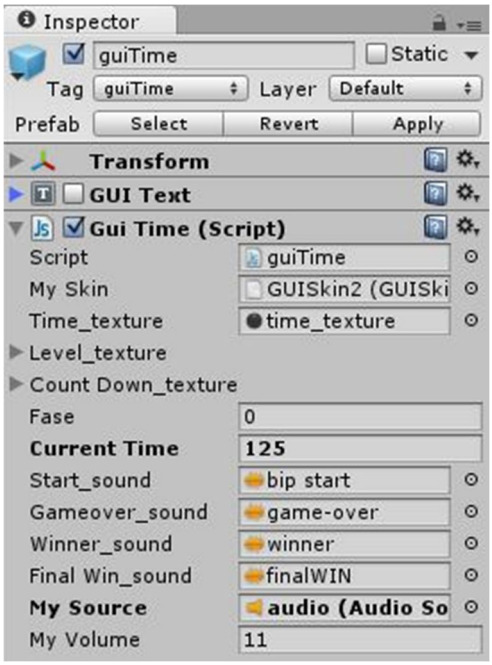

Fig 49. Objeto guiTime.

```
var mySkin : GUISkin; //GuiSkin donde se indica tipo de letra tamaño color de fondo.. etc
var time_texture : Texture; //Variable donde ira asociada la textura que se mostrara como reloj
var level_texture : Texture[]; //Texturas para mostrar el nivel en letras gigantes
var countDown_texture : Texture[]; //Texturas para mostrar la cuenta atras en letras gigantes
//fase: Variable que sirve para indicar la fase en la que se encuentra el juego //fases [0,1]: mostrar el nivel,
//fase [2,3,4]: mostrar una cuenta atras, //fase [5]: 1 segundo de espera con pitido incluido para comenzar la partida
//fase [6]: partida en curso //fase [7]: game over //fase [8]: level superado //fase [9]: juego completo superado
public  var fase : int = 0; //Siendo 0 la fase inicial
public  var currentTime : int; //Variable que mantendra almacenado el tiempo que resta para que acabe el juego
private var textfield:GUIText; //Objeto que ira asociado a la componente GuiText del objeto
private var timeRunning : boolean = false;  //variable que indica si el tiempo esta en marcha, su valor inicial es "false"
private var currentScore : int;             //variable que indica el numero de objetos atrapados
private var scoreController_gameobject : GameObject;  //Variable donde ira asociado el objeto que gestiona la puntuacion en el juego
private var guiCatcher_gameobject : GameObject;  //Variable donde ira asociado el objeto catcher
private var guiPower_gameobject : GameObject;  //Variable donde ira asociado el objeto guiPower
private var upx_gameobject : GameObject;  //Variable donde ira asociado el objeto upx
private var downx_gameobject : GameObject;  //Variable donde ira asociado el objeto downx
public var start_sound : AudioClip;     //Sonido que pronuncia 3 pitidos
public var gameover_sound : AudioClip;  //Sonido que pronuncia "game over"
public var winner_sound : AudioClip;    //Sonido que pronuncia "winner"
public var finalWin_sound : AudioClip;  //Sonido que indica el final del juego
public var mySource : AudioSource;  //AudioSource que reproduce el AudioClip correspondiente
public var myVolume : float = 1.0f; //Volumen al que se reproduce el sonido
private var start_played : boolean = false; //una vez el sonido se haya reproducido se modifica su valor a true;
private var winner_played : boolean = false; //una vez el sonido se haya reproducido se modifica su valor a true;
private var gameover_played : boolean = false; //una vez el sonido se haya reproducido se modifica su valor a true;
private var finalWin_played : boolean = false; //una vez el sonido se haya reproducido se modifica su valor a true;

//Cuando Se carga el escenario
function Start(){
    AudioListener.volume = 1; //Se activa el volumen general
    mySource.pitch = 1; //se ajusta la velocidad del sonido a 1 (máxima)
    //Se busca en la escena el objeto llamado scoreController y este se asigna a la variable scoreController_gameobject
    scoreController_gameobject = GameObject.Find("scoreController"); //objeto que gestiona la puntuacion
    guiCatcher_gameobject = GameObject.Find("guiCatcher"); //objeto asociado al boton del catcher
    guiPower_gameobject = GameObject.Find("guiPower"); //objeto asociado al potenciometro guiPower
    upx_gameobject = GameObject.Find("upx"); //objeto asociado a la parte superior del helicoptero
    downx_gameobject = GameObject.Find("downx");    //objeto asociado a la parte inferior del helicoptero
    yield WaitForSeconds(1);//Se espera 1 segundo
}

function Update () {
    if (fase != 7) {//Si todavia no s esta en la fase 7
        if (timeRunning == false) {//Si el tiempo esta parado
            timeRunning=true;//Se pone en marcha el tiempo
            textfield = GetComponent(GUIText); //Se guarda en la variable textfield el componente GuiText asociado al objeto
            UpdateTimerText(); //Se lanza la funcion que actualiza el GuiText con el valor de tiempo restante actualizado
            TimerTick(); //lanza la funcion que avanza un segundo el tiempo
        }
    }
}
```

Las variables más importantes que habría que comentar son las siguientes:

- currentTime es la que lleva el valor de tiempo restante como numero entero.

- textfield es el propio texto con el valor que se mostrará al usuario por pantalla.

- time_texture es donde se almacena la imagen que hará de marcador.

- currentScore es el valor que representa el número de objetos recogidos, para poder calcular la puntuación junto al tiempo restante al finalizar la partida.

- level_texture será una lista de imágenes para mostrar el nivel actual.

- countDown_texture, serán las imágenes para mostrar la cuenta atrás inicial

También se almacenan aquí en variables los sonidos para indicar el inicio y final de la partida.

```
//Función que se lanza 1 vez cada segundo (actualiza el marcador de tiempo y reproduce los sonidos de la cuenta atras inicial)
function TimerTick() {
    while (currentTime > (0)) {
        yield WaitForSeconds(1);//Se espera 1 segundo
        currentTime--;//Se reduce el tiempo restante en 1 unidad
        //Se comprueba si el helicóptero no ha volcado
        if (upx_gameobject.transform.position.y < (downx_gameobject.transform.position.y) + 2) {
            currentTime = 0; //Si el objeto upx esta por debajo del objeto (downx+ 2 unidades de altura) -> GameOver
        }
        UpdateTimerText();//Se lanza la funcion que actualiza el GuiText con el valor de tiempo restante actualizado
    }
}
```

En la función TimerTick se lanza una vez por segundo, además de contabilizar el tiempo, se encarga de comprobar que el helicóptero no ha volcado comparando la componente “y” de la parte superior con la inferior.

```
function UpdateTimerText() {
    //actualizar el campo de texto para mostrar la cuenta regresiva del juego
    textfield.text = currentTime.ToString();
    //Si el tiempo se ha acabado
    if (currentTime<=0){
        //Se obtiene el numero de objetos capturados para poder calcular la puntuacion final.
        currentScore = scoreController_gameobject.GetComponent( scoreController ).currentScore;
        //establece la variable fase con el valor "7" (fase fin de juego)
        fase = 7;
    }
}

function Fase(n:int) {
    yield WaitForSeconds(1);
    if (n == 1) {
        fase = n;//se inscribe el valor en la variable de fase
    }
    //Si la entrada es la fase 6, fase de juego
    else if (n == 6) {
        fase = n;//se inscribe el valor en la variable de fase
        Time.timeScale = 1; //Se activa el avance del tiempo
        AudioListener.volume = 1; //y se activa el volumen
    }
    else {
        fase = n;//se inscribe el valor en la variable de fase
    }
}

function OnGUI() {
    var myStyle : GUIStyle = new GUIStyle();//se crea un nuevo estilo para la cuenta atras durante el juego
    myStyle.fontSize = 15; //Se proporciona tamaño
    myStyle.normal.textColor = Color.white; // color para el texto
    if (fase < 7) GUI.Label( Rect( 75, 0, 70, 70 ), time_texture );//Se muestra el marcador de Tiempo (Textura)
    if (fase == 0) { //Si la fase = 0
        //Muestra el level
        GUI.Label( Rect( 0, 0, Screen.width, Screen.height), level_texture[ Application.loadedLevel-1 ] );
        Fase(1); //Cambia a fase 1
    }
    else if (fase == 1) { //Si la fase = 1
        //Muestra el level
        GUI.Label( Rect( 0, 0, Screen.width, Screen.height), level_texture[ Application.loadedLevel-1 ] );
        Fase(2); //Cambia a fase 2
    }
    else if (fase == 2) { //Si la fase = 2
        if (start_played == false) {
            mySource.PlayOneShot(start_sound, myVolume);
            start_played = true;
        }
        GUI.Label( Rect( 0, 0, Screen.width, Screen.height), countDown_texture[ 0 ] ); //Muestra el 3
        Fase(3); //Cambia a fase 3
    }
    else if (fase == 3) { //Si la fase = 3
        GUI.Label( Rect( 0, 0, Screen.width, Screen.height), countDown_texture[ 1 ] ); //Muestra el 2
        Fase(4); //Cambia a fase 4
    }
    else if (fase == 4) { //Si la fase = 4
        GUI.Label( Rect( 0, 0, Screen.width, Screen.height), countDown_texture[ 2 ] ); //Muestra el 1
        Fase(5); //Cambia a fase 5
    }
    else if (fase == 5) { //Si la fase = 5 No muestra nada
        Fase(6); //Cambia a fase 6
        scoreController_gameobject.GetComponent( scoreController ).showNumber=true;
    }
    else if (fase == 6) { //Si fase = 6, se comienza a descontar el tiempo del juego
        GUI.Label( Rect( 95, 35, Screen.height, 100 ), currentTime + " " , myStyle );//Tiempo (valor numerico)
    }
    else if (fase == 7){  //Si fase = 7, el tiempo se ha acabado
        guiCatcher_gameobject.GetComponent( guiCatcher ).show = false; //se oculta el boton catcher
        guiPower_gameobject.GetComponent( guiPower ).show = false;//se oculta el potenciometro guiPower
        scoreController_gameobject.GetComponent( scoreController ).showTexture=false;//se oculta el marcador
        if (gameover_played == false){
            mySource.PlayOneShot(gameover_sound, myVolume);
            gameover_played = true;
        }
        GUI.skin = mySkin;//Se utiliza la skin "mySkin"
        //Mitad superior de la pantalla con fondo negro y el mensaje de GameOver + Puntuacion obtenida
        GUI.Box(
            Rect(0,0,Screen.width,Screen.height/2),
            "GAME OVER: YOUR SCORE IS " + ((currentScore*currentScore)*(currentTime+9)) + ". LETS TRY AGAIN?"
        );
        //Mitad inferior de la pantalla con fondo negro y el mensaje de GameOver + Puntuacion obtenida
        GUI.Box(Rect(0,Screen.height/2,Screen.width,Screen.height), "");
        //Botones con las posibles opciones "Jugar de nuevo" o "salir"
        if (GUI.Button(Rect(Screen.width /2 - 200,Screen.height /2 - 100,400,100), "PLAY AGAIN")) {
            Time.timeScale = 1;//Se recupera la velocidad de avance de tiempo normal
            Application.LoadLevel(Application.loadedLevel);//Se carga el nivel 1 de la aplicacion
        }
        if (GUI.Button(Rect(Screen.width /2 - 200,Screen.height /2,400,100), "EXIT")) {
            Time.timeScale=1;//Se recupera la velocidad de avance de tiempo normal
            Application.Quit();//Se cierra la aplicacion
        }
    }
}
```

La variable fase comienza valiendo 0, y se incrementa una vez cada segundo hasta llegar a valer 6, lo que indicará que la partida ha comenzado. Durante los dos primeros segundos se mostrará la textura que indica el nivel, durante los 3 segundos siguientes se mostrará uno a uno los números con sonidos simultáneos para alertar al jugador de que la partida comienza en apenas unos segundos, en la fase 5 no se muestra nada ya que un segundo después la partida se dará por empezada.

La función Update es la encargada de lanzar el resto de funciones mientras el tiempo siga en marcha, una para contabilizar el tiempo, y otra para actualizar su valor en el marcador.

Cuando el valor de la variable que indica el tiempo valga 0, se mostrarán las texturas y botones que indicarán el fin de la partida o Game Over, permitiendo reiniciarla o salir del juego. La puntuación se obtendrá a partir de una operación matemática usando el tiempo restante y el número de objetos atrapados y llevados con éxito a su destino.

El marcador que muestra el número de objetos recogidos funciona de la misma manera que el marcador que muestra el tiempo restante, con la salvedad de que el número aumentará cuando se detecte que un objeto se ha recogido. El marcador estará en el objeto “scoreController”, el cual tendrá una variable Score_texture que guardará la imagen para el marcador, siendo currentScore el valor que indicará en cada momento el número de objetos que hayan recogido con éxito, cuando un objeto objetivo cuyo tag sea “catchable” entre en el Box Collider, se hará sonar el sonido “catched_sound” a través de un AudioSource llamado audio, que irá acoplado al helicóptero junto al AudioListerner del objeto “guiPower”.

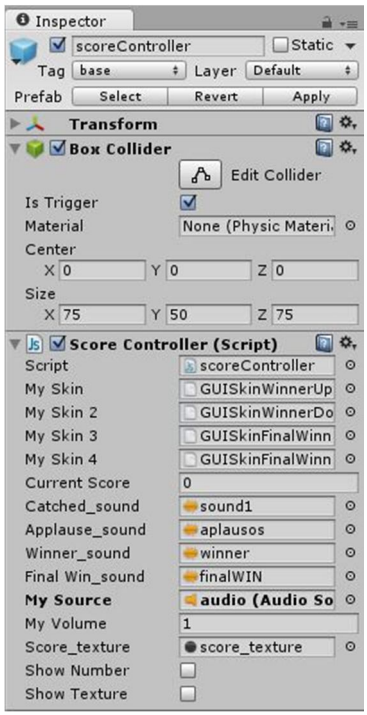

Fig 50. Objeto scoreController.

También hay 4 GUISkins para mostrar como fondo cuando se consiga el objetivo, junto a los sonidos applauses y winner cuando se haya superado un nivel inferior a 9, o los sonidos que indican el final del juego, applauses y finalwin. La variable winner, indicará como se ha logrado la victoria, con el tiempo justo, o con tiempo de sobra. Este es el script “scoreController.js”:

```
var mySkin : GUISkin;   //Skin para la mitad superior de la pantalla con globos y confeti signo de celebracion al superar un nivel
var mySkin2 : GUISkin;  //skin para la mitad inferior de la pantalla con confetis signo de celebracion al superar un nivel
var mySkin3 : GUISkin;  //Skin para la mitad superior de la pantalla con copas y estrellas de oro de celebracion al completar el juego
var mySkin4 : GUISkin;  //Skin para la mitad superior de la pantalla con estrellas de oro de celebracion al completar el juego
private var guiTime_gameobject : GameObject;//Variable para acceder al objeto donde se gestionan el tiempo y las fases en el juego
private var winner : int = 0;  //Tipo de victoria obtenida, (1: con el tiempo limite) (2:tiempo sobrado normal) (3:tiempo sobrado alto)
private var currentTime : int = 0;//variable que recoge el tiempo restante actual del objeto que gestiona el tiempo(guiTime_gameobject)
public var currentScore : int = 0;//variable que almacena la cuenta de objetos catchables recogidos
public var catched_sound : AudioClip;   //Sonido que se reproduce al entrar un catchable en la zona de colision del scoreController
public var applause_sound : AudioClip;  //Sonido que de aplausos para reproducir al superar un nivel normal
public var winner_sound : AudioClip;    //Sonido que pronuncia "winner" para reproducir al superar un nivel normal
public var finalWin_sound : AudioClip;  //Sonido que indica el final del juego
public var mySource : AudioSource;  //AudioSource que reproduce el AudioClip correspondiente
public var myVolume : float = 1.0f; //Volumen al que se reproduce el sonido

private var winner_played : boolean;    //variable que indicara si se ha reproducido el sonido
private var finalWin_played : boolean;  //variable que indicara si se ha reproducido el sonido
var score_texture : Texture; //Textura de fondo (panel donde aparece el numero)

function Start(){
    winner_played = false;
    finalWin_played = false;
    //Se busca en la escena el objeto llamado guiTime y este se asigna a la variable guiTime_gameobject
    guiTime_gameobject = GameObject.Find("guiTime");
    var mySource_gameobject : GameObject;
    mySource_gameobject = GameObject.Find("audio");
    mySource = mySource_gameobject.GetComponent( AudioSource );
}


function OnTriggerEnter(other : Collider){
    if (other.tag == "catchable") {//Si el objeto que ha entrado tiene el tag "catchable"
        mySource.PlayOneShot( catched_sound, myVolume ); //se reproduce el sonido que indica la captura del objeto con exito
        other.rigidbody.isKinematic = false;//se elimina la propiedad que evita que las fuerzas afecten al objeto
        other.rigidbody.useGravity = true;//se recupera el efecto de la gravedad sobre el objeto
        currentScore += 1;//Se suma 1 punto a la variable que se muestra en el marcador
        guiTime_gameobject.GetComponent( guiTime ).currentTime += 60;//Se añaden 60 segundos al tiempo restante
        other.tag = "catched";//se modifica el tag del objeto para que no pueda volver a interactuar con la puntuacion ni con el catcher.
    }
}
function OnGUI() {
    if (guiTime_gameobject.GetComponent( guiTime ).fase == 6) {//Si la fase = 6 (fase de juego)
        var myStyle0 : GUIStyle = new GUIStyle();//Se configura la letra con un nuevo estilo
        myStyle0.fontSize = 15; //tamaño = 15
        myStyle0.normal.textColor = Color.white; //color = blanco
        //Se dibuja el contador de objetos recogidos
        GUI.Label( Rect(Screen.width-145, 0, 70, 70), score_texture);
        //Se actualiza el numero de objetos recogidos
        GUI.Label( Rect( Screen.width-115, 35, Screen.height, 100 ), currentScore + " ", myStyle0 );
    }
    //se configura el estilo de letra (tamaño y color) para producir un mensaje de victoria
    var myStyle : GUIStyle = new GUIStyle();
    myStyle.fontSize = 30;
    myStyle.normal.textColor = Color.white;
    //si el numero de objetos recogidos es igual al nivel en el que se esta jugando
    if (currentScore >= Application.loadedLevel) {
        if (Application.loadedLevel == 9) {//Si se ha superado el ultimo nivel(9)
            GUI.skin = mySkin2;
            //Se muestra un mensaje de victoria dependiendo del valor de "winner"
            switch(winner) {
                case 1:
                    GUI.Box(Rect(0,0,Screen.width,Screen.height/2),
                        "YOU ARE GOOD.. YOUR SCORE IS " + ((currentScore*currentScore)*(currentTime+9))  + ". LETS TRY AGAIN?");
                    break;
                case 2:
                    GUI.Box(Rect(0,0,Screen.width,Screen.height/2),
                        "VERY GOOD.. YOUR SCORE IS " + ((currentScore*currentScore)*(currentTime+9)) + ". LETS TRY AGAIN?");
                    break;
                case 3:
                    GUI.Box(Rect(0,0,Screen.width,Screen.height/2),
                        "YOU ARE THE BEST!.. YOUR SCORE IS " + ((currentScore*currentScore)*(currentTime+9)) + ". LETS TRY AGAIN?");
                    break;
                default:
                    GUI.Box(Rect(0,0,Screen.width,Screen.height/2), "WINNER");
            }
            GUI.skin = mySkin3;
            GUI.Box(Rect(0,(Screen.height/2)-1,Screen.width,Screen.height), " ");
            guiTime_gameobject.GetComponent( guiTime ).fase = 9;
            if (finalWin_played == false)
            {
                mySource.PlayOneShot(applause_sound, myVolume);
                mySource.PlayOneShot(finalWin_sound, myVolume);
                finalWin_played = true;
            }
            //Muestra 2 botones, Jugar de nuevo, Salir
            if (GUI.Button(Rect(Screen.width /2 - 200,Screen.height /2 - 100,400,75), "PLAY AGAIN")) {
                Time.timeScale=1;
                Application.LoadLevel(9);
            }
```

```
            if (GUI.Button(Rect(Screen.width /2 - 200,Screen.height /2-25,400,75), "EXIT")) {
                Time.timeScale=1;
                Application.Quit();
            }
        }
        else {//si el nivel superado no es el ultimo nivel
            GUI.skin = mySkin;
            switch(winner) {//Se muestra un mensaje de victoria dependiendo del valor de "winner"
                case 1:
                    GUI.Box(Rect(0,0,Screen.width,Screen.height/2),
                        "YOU ARE GOOD.. YOUR SCORE IS " + ((currentScore*currentScore)*(currentTime+9))  + ". LETS TRY AGAIN?");
                    break;
                case 2:
                    GUI.Box(Rect(0,0,Screen.width,Screen.height/2),
                        "VERY GOOD.. YOUR SCORE IS " + ((currentScore*currentScore)*(currentTime+9)) + ". LETS TRY AGAIN?");
                    break;
                case 3:
                    GUI.Box(Rect(0,0,Screen.width,Screen.height/2),
                        "YOU ARE THE BEST!.. YOUR SCORE IS " + ((currentScore*currentScore)*(currentTime+9)) + ". LETS TRY AGAIN?");
                    break;
                default:
                    GUI.Box(Rect(0,0,Screen.width,Screen.height/2), "WINNER");
            }
            GUI.skin = mySkin2;
            GUI.Box(Rect(0,(Screen.height/2)-1,Screen.width,Screen.height), " ");
            guiTime_gameobject.GetComponent( guiTime ).fase = 8;
            if (winner_played == false) {
                mySource.PlayOneShot(applause_sound, myVolume);
                mySource.PlayOneShot(winner_sound, myVolume);
                winner_played = true;
            }
            //Muestra 3 botones, Siguiente nivel, Jugar de nuevo, Salir
            if (GUI.Button(Rect(Screen.width /2 - 200,Screen.height /2 - 100,400,75), "NEXT LEVEL")) {
                Time.timeScale=1;
                Application.LoadLevel(Application.loadedLevel + 1);
            }
            if (GUI.Button(Rect(Screen.width /2 - 200,Screen.height /2-25,400,75), "PLAY AGAIN")) {
                Time.timeScale=1;
                Application.LoadLevel(Application.loadedLevel);
            }
            if (GUI.Button(Rect(Screen.width /2 - 200,Screen.height /2 + 50,400,75), "EXIT")) {
                Time.timeScale=1;
                Application.Quit();
            }
        }
    }
}

function Update () {
    //Se actualizara el valor de puntuacion actual continuamente hasta que currentScore
    //que almacena el numero de objetos recogidos, sea igual al nivel en el que se esta jugando
    if (currentScore >= Application.loadedLevel) {
        if (currentTime == 0) //Si la variable currentTime no ha modificado su valor inicial
        {   //Se le asigna un valor nuevo para poder calcular la puntuacion apartir de el
            currentTime = guiTime_gameobject.GetComponent( guiTime ).currentTime;
        }
        if (currentTime < 40){  //mensaje en caso de que el tiempo sea menor de 40
            winner = 1;
        }
        else if ((currentTime > 39) && (currentTime < 120))
        {   //mensaje en caso de que el tiempo restante este en el siguiente rango [40..119]
            winner = 2;
        }
        else {  //mensaje en caso de que el tiempo restante sea 120 segundos o superior
            winner = 3;
        }
    }
}
```

En la función OnTriggerEnter, se detecta si un objeto objetivo “catchable” ha entrado en el collider del scoreController, y en tal caso, se suma el punto y el tiempo correspondiente y se cambia el tag del objeto objetivo a “catched” lo cual servirá para evitar la interactuación de los objetos ya recogidos satisfactoriamente.

Estos objetos representan a juguetes que estarán repartidos por la habitación, en lugares elegidos al azar en la fase de inicio del escenario mediante el script “ObjectSpawn.js” comentado al inicio de la sección de implementación en esta memoria.

Los objetos tienen un script asociado que gestiona las colisiones del objeto contra otras superficies, reproduciendo el sonido “colliSound”, o en caso de ser atrapado de ser atrapado por el cátcher reproduciendo el sonido “catchingSound”.

Fig 51. Vista con los objetos catchables.

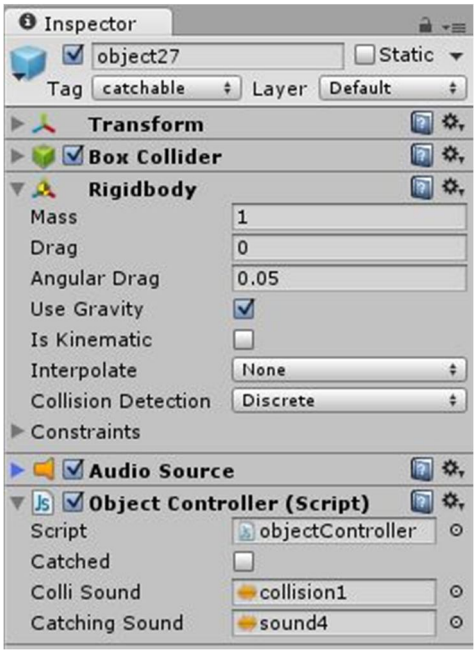

Fig 52. Detalles de un objeto catchable.

Este es el script “objectController.js” asociado a los objetos:

```
public var catched : boolean;//variable que indicara si el objeto esta atrapado o no
private var heightDamping = 10.0; //velocidad con la que el objeto actualizara su posicion mientras este atrapado
private var rotationDamping = 3.0;//velocidad con la que el objeto actualizara su rotacion mientras este atrapado
private var catcher_gameobject : GameObject; //objeto encargado de atrapar a los catchables (objetos objetivo)
private var guiCatcher_gameobject : GameObject; //objeto encargado de gestionar si el usuario desea atrapar/soltar objetos
public var colliSound : AudioClip; //sonido que reproduce una colision
public var catchingSound : AudioClip; //sonido que reproduce un sonido que indicara haber atrapado un objeto
private var myVolume : float = 1.0f;     //volumen al que se reproduciran los sonidos
private var collision : boolean = false; //variable para enviar como maximo un unico mensaje de colision

function Start() {
    catcher_gameobject = GameObject.Find("Catcher");
    guiCatcher_gameobject = GameObject.Find("guiCatcher");
    catched = false;
}

//funcion que se activa si un objeto cualquiera entra en la zona de colision de un objeto objetivo "catchable"
function OnTriggerEnter(other : Collider) {
    if (other.tag != "catcher") { //Si la colision no es con el catcher
        audio.PlayOneShot( colliSound, myVolume ); //reproduce sonido de colision
        if (catched) { //Si el objeto es catched
            //si la colision no es con el jugador ni con otro objeto y ademas no se ha enviado ya el aviso de colision
            if ((other.tag != "player") && (other.tag != "catchable") && (collision==false)) {
                SendCollision(); //Se envia el aviso de colision
            }
        }
    }
}
//Envia el aviso de colision al guiCatcher
function SendCollision(){
    catched=false;
    guiCatcher_gameobject.GetComponent( guiCatcher ).collision = true;
    yield WaitForSeconds(1);
    collision = false;
}
```

Los objetos comienzan teniendo el valor “false” para la variable catched, si se lanza la función OnTriggeredEnter con un collider de tag diferente a “catcher”, “player” o “catchable” se lanzara la función SendCollision() para soltar al objeto.

```
function LateUpdate () {
    if (catched) { //Si el objeto esta atrapado, se movera siguiendo al catcher
        var currentRotationAngle = transform.eulerAngles.y;//vector rotacion actual
        var currentHeight = transform.position.y;//posicion actual del objeto
        var wantedRotationAngle = catcher_gameobject.transform.eulerAngles.y; //futuro vector de rotacion del objeto
        var wantedHeight = catcher_gameobject.transform.position.y - 1;//futura posicion del objeto
        //apartir de las 4 variables anteriores, se actualizan los vectores de posicion y rotacion suavizadamente
        currentRotationAngle = Mathf.LerpAngle (currentRotationAngle, wantedRotationAngle, rotationDamping * Time.deltaTime);
        currentHeight = Mathf.Lerp (currentHeight, wantedHeight, heightDamping * Time.deltaTime);

        var currentRotation = Quaternion.Euler (0, currentRotationAngle, 0); //se aplica el vector de rotacion
        transform.position = catcher_gameobject.transform.position; //siguiendo la posicion del catcher
        transform.position -= currentRotation * Vector3.forward * 0.1; //a una distancia de 0.1
        transform.position.y = currentHeight; //y se actualiza la altura siendo esta la posicion del catcher
        if (this.rigidbody.isKinematic == false) { //si tiene desactiva la variable que lo convierte en kinematico
            this.rigidbody.isKinematic = true; //se activa esta variable para que las fuerzas vuelvan a afectarle
        }
        if (this.rigidbody.useGravity)  {//si tiene activa la variable que anula el efecto de la gravedad
            this.rigidbody.useGravity = false; //se desactiva esta variable para que las fuerzas vuelvan a afectarle
        }
        //renderer.material.color = Color.red;(usado en testing)
    }
    else { //en el caso de que el objeto no esté atrapado
        if (this.rigidbody.isKinematic) { //si tiene activa la variable que lo convierte en kinematico
            this.rigidbody.isKinematic = false; //se desactiva esta variable para que las fuerzas vuelvan a afectarle
        }
        if (this.rigidbody.useGravity == false) {//si tiene desactiva la variable que anula el efecto de la gravedad
            this.rigidbody.useGravity = true; //se sactiva esta variable para que las fuerzas vuelvan a afectarle
        }
        //renderer.material.color = Color.green; (usado en testing)
    }
}
```

La función LateUpdate se lanza después del haber lanzado el resto de funciones Update existentes en el proyecto. En ella mientras se tenga un valor de catched = “true”, se modificará la posición del objeto haciendo que esta sea igual a la de objeto asociado a la variable catcher_gameobject, además se modificaran sus propiedades volviéndolo un objeto kinemático al que no le afectará ni la gravedad ni ninguna otra fuerza, al restaurarse el valor “false” de catched, se restaurarán sus propiedades que harán que vuelva a verse afectado por la gravedad.

La siguiente parte a explicar del helicóptero será la cámara que permite visualizarlo en todo momento, esta cámara no contiene el típico AudioListener ya que lo traspasé a otro objeto, esto permite que pueda alejarla o acercarla sin que el sonido se vea afectado ya que si el AudioListener estuviera aquí, el sonido sería diferente dependiendo de la posición de la cámara, también varía el “field of view” (campo de vista) respecto a una cámara normal, el valor de near será 0.3 y el de far será 100000.

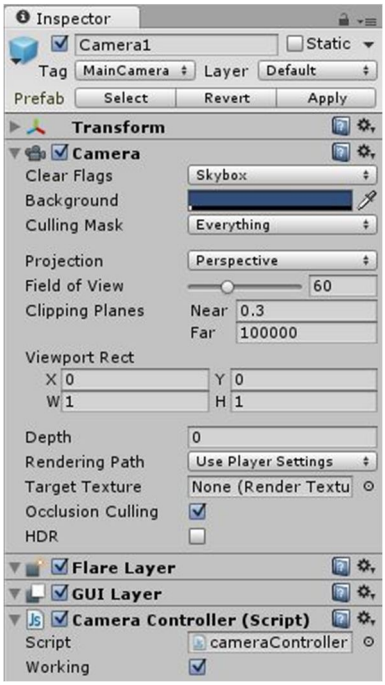

Fig 53. Objeto con la Camara.

La posición depende de si la cámara está visualizando al objeto o existe algún objeto en medio interponiéndose en la visión del jugador, inicialmente comienza a una distancia de 25 unidades, acercándose rápidamente mientras sea necesario para no perder de vista al helicóptero, y volviendo a su posición original lentamente en caso de poder visionar al objeto. Este es el script asociado como componente a la cámara: “cameraController.js”.

```
public var working : boolean = true; //variable que indica si la camara se esta controlando
private var distance = 24.0; // distancia en el plano x-z hasta el helicoptero
private var height = 10.0; // altura sobre la que la camara estará
private var heightDamping = 5.0; //velocidad de cambio de posicion de la camara en la altura
private var rotationDamping = 1.0; //velocidad de cambio de posicion de la camara en las rotaciones
private var player : Transform; //posicion del helicoptero

function Start () {
    player = GameObject.Find("Helicopter-1Player").transform; //guarda la posicion del helicoptero
    this.camera.enabled = true;//activa la camara a la que esta asociada el script
}

function LateUpdate () {
    if (working){ //Si funciona (juego no esta pausado)
        var hit : RaycastHit; //crea una variable de colision de rayo
        var rayDirection = player.position - transform.position; //se indica la direccion del helicoptero
        if (Physics.Raycast (transform.position, rayDirection, hit))
        {   //Si el primer objeto con el que colisiona es el jugador (esta siendo visible)
            if (hit.transform == player)
            {   //si la distancia es menor a 24 unidades, se sube lentamente
                if (distance < 24)
                    distance = distance + 0.02;
                if (height < 10)//Si la altura es menor a 10 unidades, se sube lentamente
                    height = height + 0.02;
            }
            else //Si el objeto con el que colisiona el rayo no es el helicoptero
            {   //si la distancia es mayor del limite (5 unidades) se reduce rapidamente, de unidad en unidad
                if (distance > 7)
                    distance = distance - 1;
                if (height > 4)//Si la altura es mayor del limite (2 unidades) se reduce rapidamente
                    height = height - 1;    //de unidad en unidad
            }
        }
        var wantedRotationAngle = player.eulerAngles.y; // Calcula la rotacion deseada apartir de la del jugador en el eje y
        var wantedHeight = player.position.y + height; // calcula la altura deseada
        var currentRotationAngle = transform.eulerAngles.y; //almacena la rotacion actual de la camara
        var currentHeight = transform.position.y; //almacena la altura actual de la camara
        // Realiza la rotacion en el eje y de forma gradual desde el origen: "currentRotationAngle", al destino "wantedRotationAngle"
        //la velocidad de modificacion del angulo será la obtenida apartir de multiplicar la variable "rotationDamping" por el deltatime
        //siendo deltatime el tiempo entre cada frame (el cual es variable dependiando de la capacidad de procesamiento del dispositivo
        //esto es necesario para poder ver el helicoptero desde la misma posición (desde atras) pese a que este pueda rotar en el espacio
        currentRotationAngle = Mathf.LerpAngle (currentRotationAngle, wantedRotationAngle, rotationDamping * Time.deltaTime);
        // realiza el movimiento de una posicion a otra de forma suave como en la linea anterior se ha modificado el ángulo
        currentHeight = Mathf.Lerp (currentHeight, wantedHeight, heightDamping * Time.deltaTime);
        var currentRotation = Quaternion.Euler (0, currentRotationAngle, 0);// convierte el angulo en rotacion
        transform.position = player.position;// Configura la posicion de la camara respecto al jugador
        transform.position -= currentRotation * Vector3.forward * distance;//a la distania deseada
        transform.position.y = currentHeight;// Configura la posicion de la camara a la altura
        transform.LookAt (player); //función que obliga a modificar el angulo de la camarapara tener siempre al helicoptero en vista
    }
}
```

A la cámara va asociado un objeto que estará encargado de indicar la dirección en la que hay que desplazarse para poder llegar al lugar donde está el objeto objetivo más próximo, o en caso de ya haber atrapado alguno, el lugar objetivo donde hay que depositarlo.

Este componente llamado “arrow” tendrá forma de flecha, y tendrá asociado como componente un script donde se habrá indicado como y cuando debe apuntarse a un objeto o a otroaunque primero explicare otro script que va asociado a los cubos que componen la flecha, el script en cuestión se llama “arrowColor.js”.

Este script modificará el color del objeto en el que esté como componente, haciéndolo verde si el catcher no tiene ningún objeto atrapado o rojo en caso contrario, mientras la flecha esté indicando el lugar donde depositar el objeto.

```
private var catcher_gameobject: GameObject; //objeto catcher

function Start () {
    catcher_gameobject = GameObject.Find("Catcher"); //se guarda el catcher en la variable
    renderer.material.color = Color.green;//si inicia dotando de color verde a los componentes de la flecha (3 cubos)
}

function LateUpdate () {
    //Si el catcher no lleva ningun objeto
    if (catcher_gameobject.GetComponent( iman ).catching == false) {
        renderer.material.color = Color.green; //la flecha sera verde (ya que apunta al catchable mas cercano)

    }
    //Si el catcher lleva algun objeto
    else if (catcher_gameobject.GetComponent( iman ).catching == true) {
        renderer.material.color = Color.red;//la flecha sera roja (ya que apunta al lugar donde debe dejarlo)
    }
}
```

Ahora solo queda orientar la flecha mediante el script “arrow.js”

```
private var catcher_gameobject: GameObject; //catcher
private var target : Transform; //objetivo donde apuntara la flecha

function Start () {
    catcher_gameobject = GameObject.Find("Catcher"); //se guarda el catcher en su variable
    target = null; //se inicia el objetivo por defecto
    if (rigidbody)  //si es un rigigbody
        rigidbody.freezeRotation = true; //se impide la libre rotacion del objeto
}

//Devuelve el objeto mas cercano con un tag usado como entrada de la funcion
function GetClosestObject(tag:String) : GameObject {
 var objectsWithTag : GameObject[]; //se crea una lista de game_objetcs
 objectsWithTag = GameObject.FindGameObjectsWithTag(tag); //se meten los objetos en los que coincida el tag con la entrada de la función

 if (objectsWithTag.Length == 0) {//Si no se ha encontrado ningun objeto
    return null;//devuelve null
 }

 var closestObject : GameObject; //se crea una variable para ir almacenando el objeto mas cercano dentro del bucle siguiente
 for (var obj : GameObject in objectsWithTag) { //para cada objeto de la lista de objetos anteriormente poblada
     if(!closestObject)  { //Si se acaba de lanzar la funcion (closestObject es null)
        closestObject = obj;//se asigna el primer objeto a la variable
     }
     //compara la distancia entre el objeto que se esta tratando y el mas cercano almacenado en la variable closetObject
     if(Vector3.Distance(transform.position, obj.transform.position) <= Vector3.Distance(transform.position, closestObject.transform.position))     {
        closestObject = obj; //guardando el objeto que tenga una distancia menor en el closestObject
     }
 }
 return closestObject; //Se devuelve el objeto que tenga la menor distancia de todas respecto a la flecha que indicará su posición
}

//Update se lanza continuamente para actualizar el estado del objeto
function Update () {
    if (catcher_gameobject.GetComponent( iman ).catching == false) {
        target = GetClosestObject("catchable").transform; //se guarda el objeto "catchable" mas cercano en la variable
    }
    else if (catcher_gameobject.GetComponent( iman ).catching == true) {
        target = GetClosestObject("base").transform; //se guarda el objeto "base" mas cercano en la variable
    }
    if (target) {
        transform.LookAt(target); //Se manda la orden de mirar al objetivo
    }
}
```

En este script, se lanza continuamente una función que almacenará en la variable “target” la posición del objeto con tag “catchable” que se encuentre a menor distancia, siendo el target el objeto con tag “base” en caso de que el jugador ya este portando algún objeto.

Con esto el jugador ya puede encontrar los objetos objetivo y el lugar donde depositarlos, lo cual significa que ya están explicados todos los objetos y losscripts que componen las escenas, únicamente habría que colocar al helicóptero en un lugar diferente en cada escena e ir guardándolas para después añadirlas a la lista de niveles para publicar en “build settings” .

Aún queda explicar a la configuración de Unity3d la cual es la siguiente:

Fig 54. Configuración del Audio.

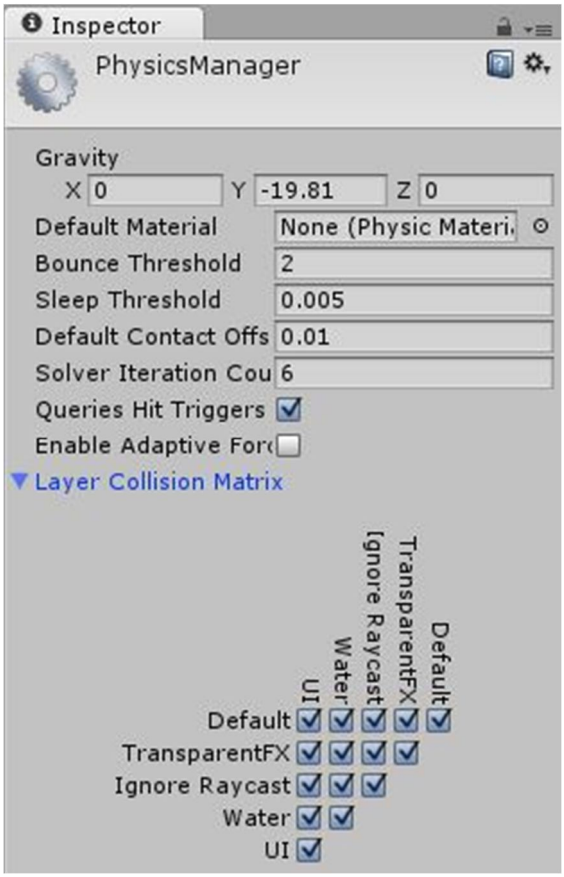

Fig 55. Configuración de la física.

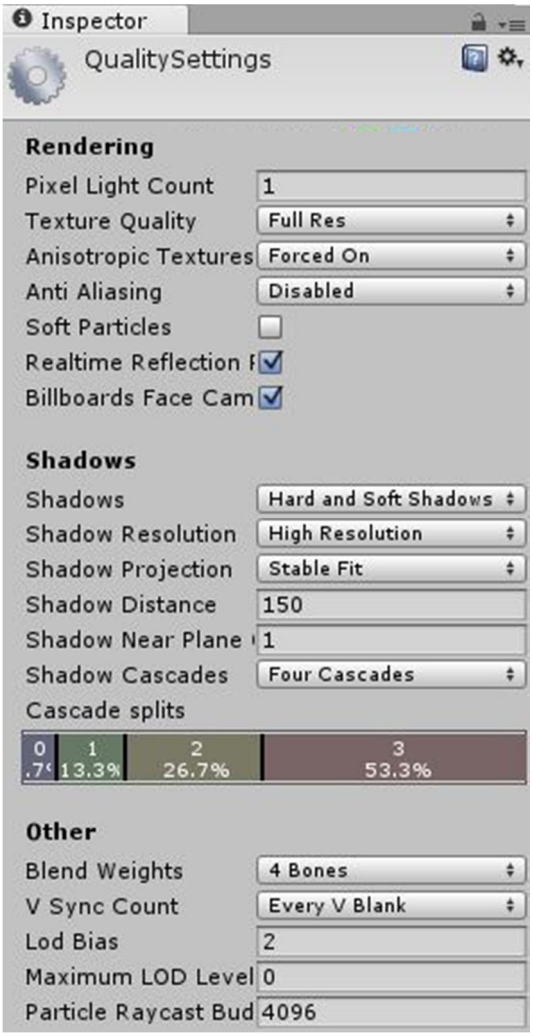

Fig 56. Configuración de la calidad grafica.

Fig 57. Configuración de la cámara.

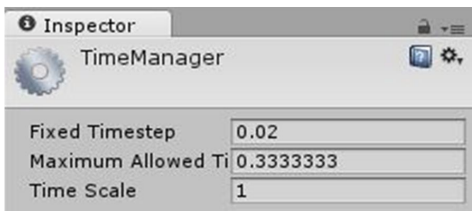

Fig 58. Configuración del tiempo.

La sección “player settings”, es la más importante de la configuración, es la que permite construir el instalable apk, para lo cual será estrictamente necesario rellenar los campos que aparecen en este desplegable.

Fig 59. Configuración al publicar (1)

En primer lugar hay que insertar el nombre de la compañía y del producto, además del icono que tendrá la aplicación.

Tras esto se configuran las propiedades de la plataforma, la orientación por defecto será, en este caso Landscape con Auto Rotation.

Una vez seleccionada la orientación deseada se despliega la siguiente pestaña para poder configurar el icono que llevará la aplicación en dispositivos de mayor tamaño que puedan usar Android.

En la pestaña de “Splash image”, se puede colocar una imagen para mostrar al cargar la aplicación.

La sección de player settings continua: la siguiente pestaña lleva el nombre de “Other Settings”, es donde se selecciona la versión correspondiente de la aplicación el código bundleIdentifier, creado a partir del nombre de la compañía y del producto para identificar de forma univoca cada aplicación, así como el nivel mínimo de api necesario para ejecutarla.

En publishing settings están las opciones que se deben Rellenar correctamente para poder publicar la aplicación en el PlayStore, primero hay que crear una new keystore, para lo cual se selecciona la opción correspondiente, se introduce la contraseña, la cual será necesario recordar ya que sin ella no se podrá actualizar una vez haya sido subida a PlayStore.

La public key o Android Market Licensing (LVL) es la clave que identificará al desarrollador y para obtenerla es necesario registrarse, el precio es de 25$.

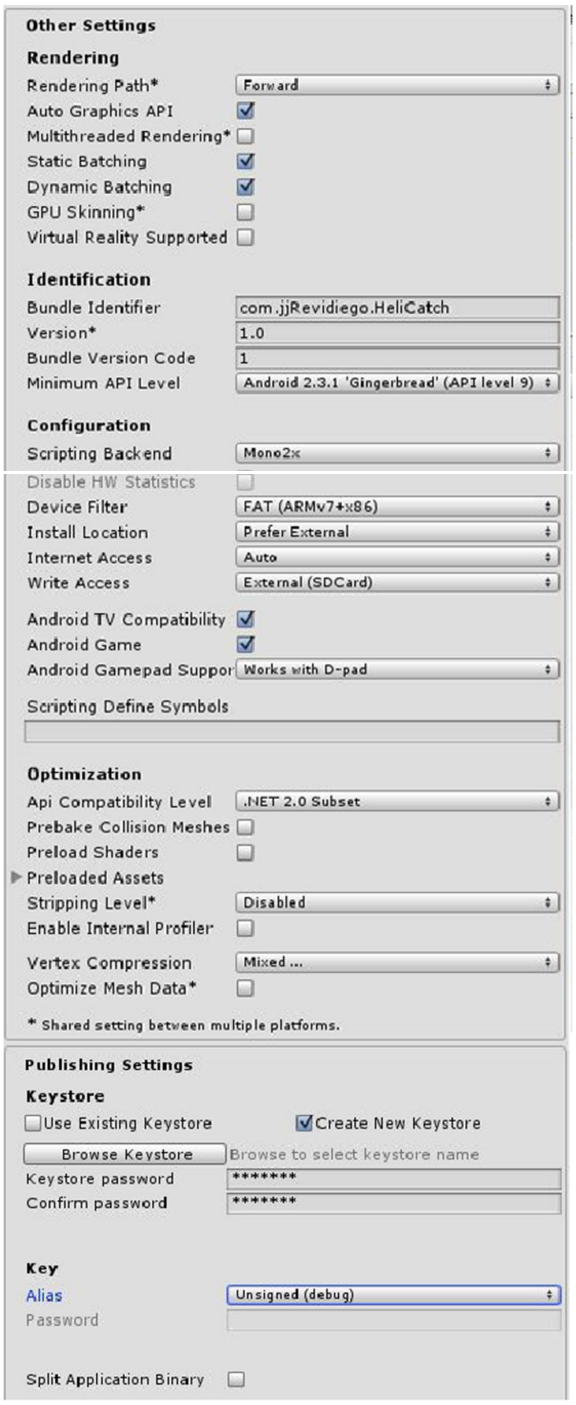

Fig 60. Configuración al publicar (2).

Una vez ingresados los datos, se le da a “Browse keystore”, para elegir el lugar donde se depositará el fichero que almacena la contraseña y la identificación del desarrollador. Para crear siguientes versiones si habrá que seleccionar “Use existing KeyStore”, elegir el fichero en el menú desplegable al hacer click en “Browse keystore”, e introducir el alias y la contraseña.

3.4. Publicación en PlayStore:

Ahora solo quedaría abrir la consola de desarrollador de Google Play, y crear una nueva aplicación con el botón azul de arriba a la derecha, una vez creada aparecerá en estado draft (borrador), esperando que se complete el resto del proceso.

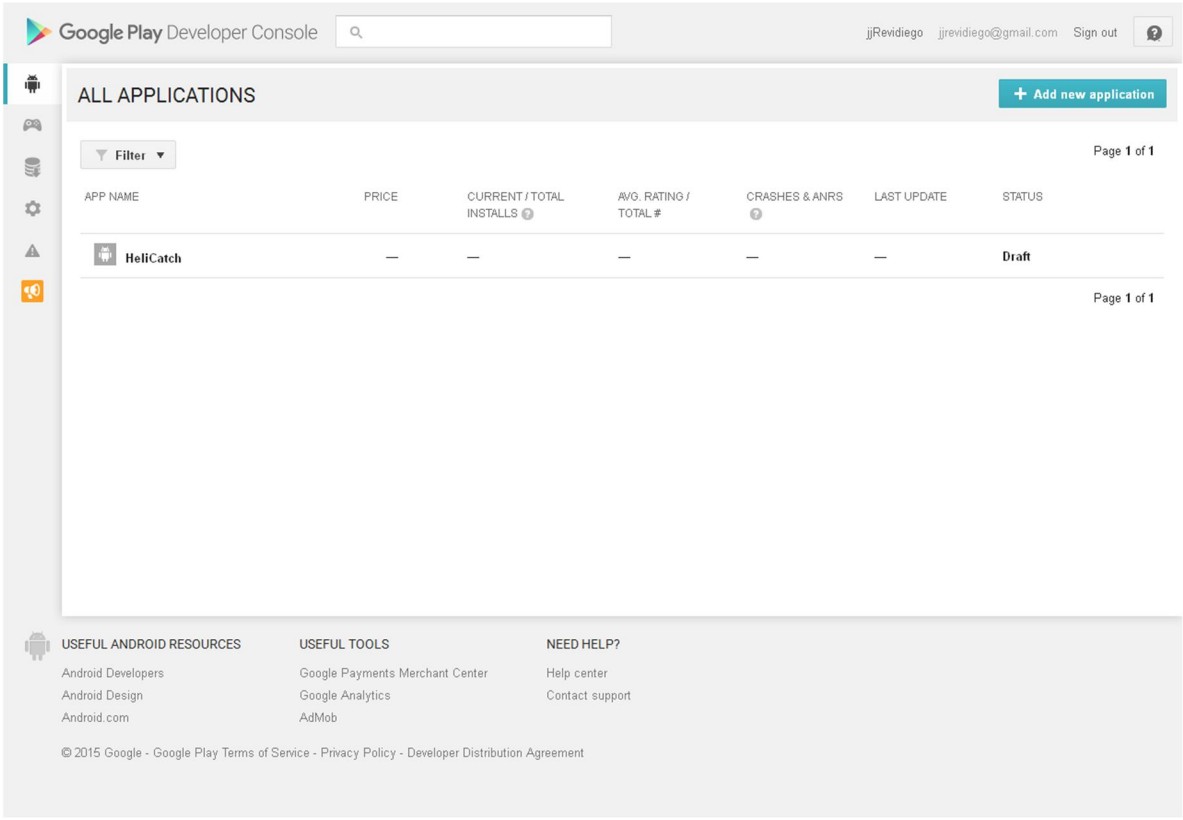

Fig 61. Vista Inicial de la consola PlayStore al crear el proyecto HeliCatch.

Este proceso consiste en añadir la información requerida sobre los detalles del producto, titulo, descripción, idioma… etc.

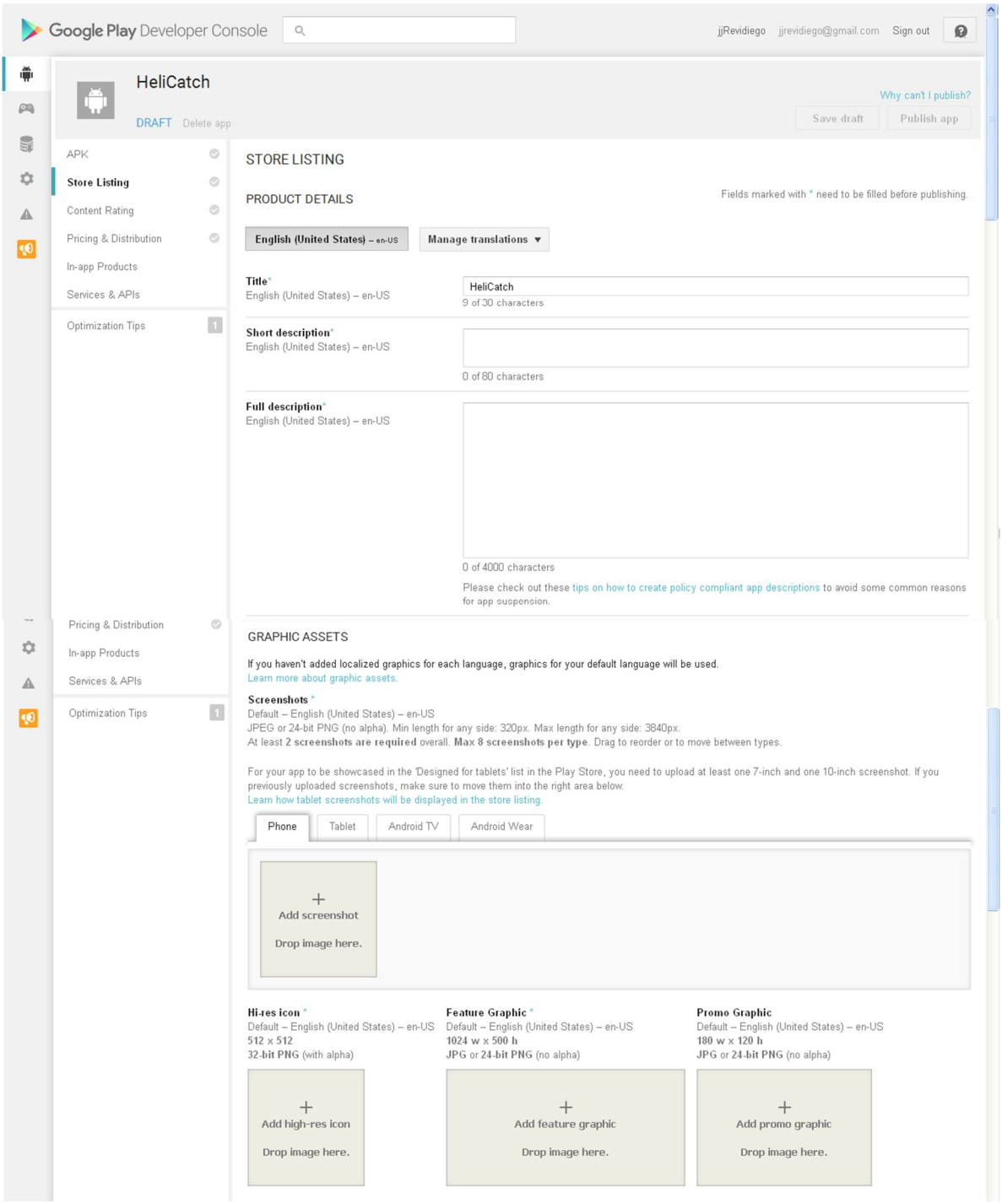

Fig 62. Vista de la seccion Store Listing en la consola PlayStore.

Para poder publicar una aplicación habrá que completar todos los pasos necesarios, los cuales aparecerán resaltados en rojo al intentar publicar la aplicación sin haberlos rellenado, como se muestra en la siguiente imagen:

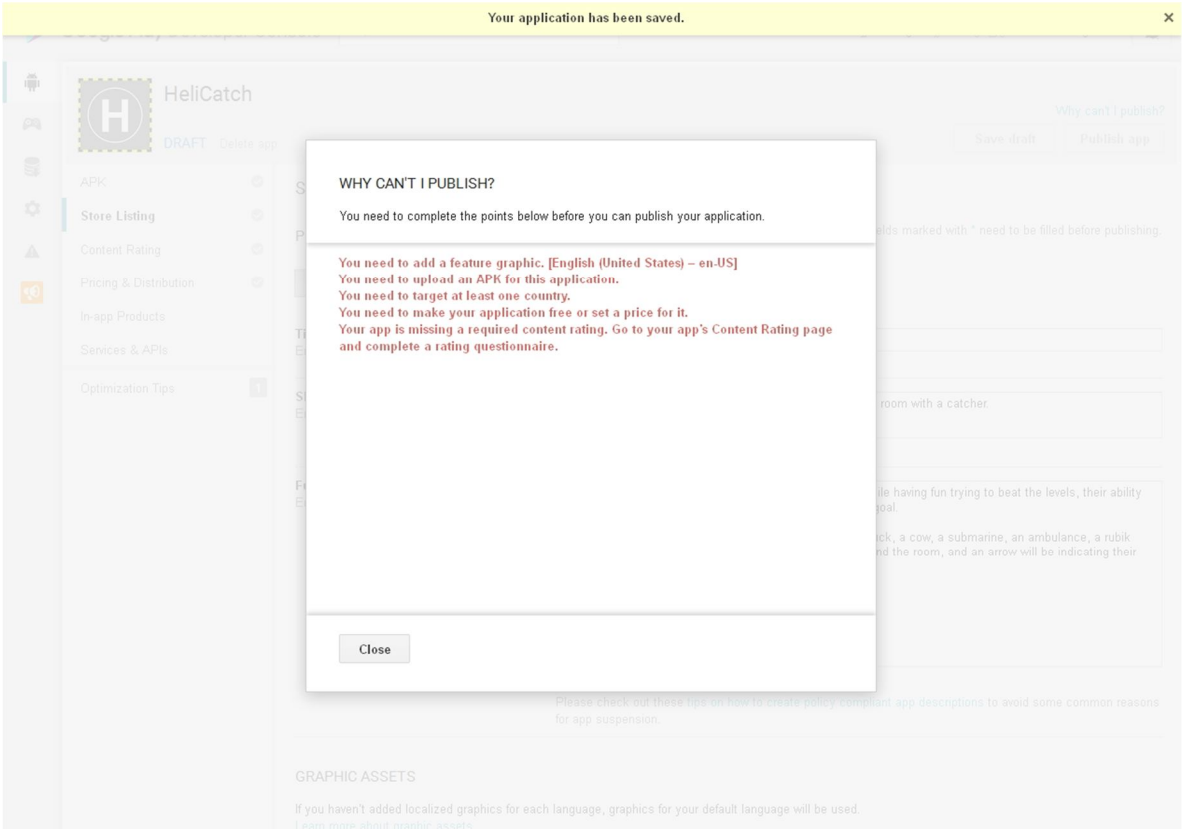

Fig 63. Indicaciones informando de los pasos necesarios para poder publicar la aplicación.

Habrá que añadir los graphic assets: (feature y promo), subir la aplicación en formato apk, decidir en qué países estará disponible para su descarga, seleccionar el precio al que se desea proveer y obtener el certificado que indique el tipo de contenido y su aptitud para el público según edades.

Tras haber elegido el idioma y haber rellenado la descripción, se añaden los Graphic Assets, capturas de pantalla tanto para dispositivos telefónicos como para tabletas, el icono y las imágenes que se mostrarán a los posibles usuarios.

El objetivo es indicar visualmente el contenido de la aplicación, tratando que esto sea hecho de la forma más atractiva posible.

Estas son las capturas de pantalla correspondientes:

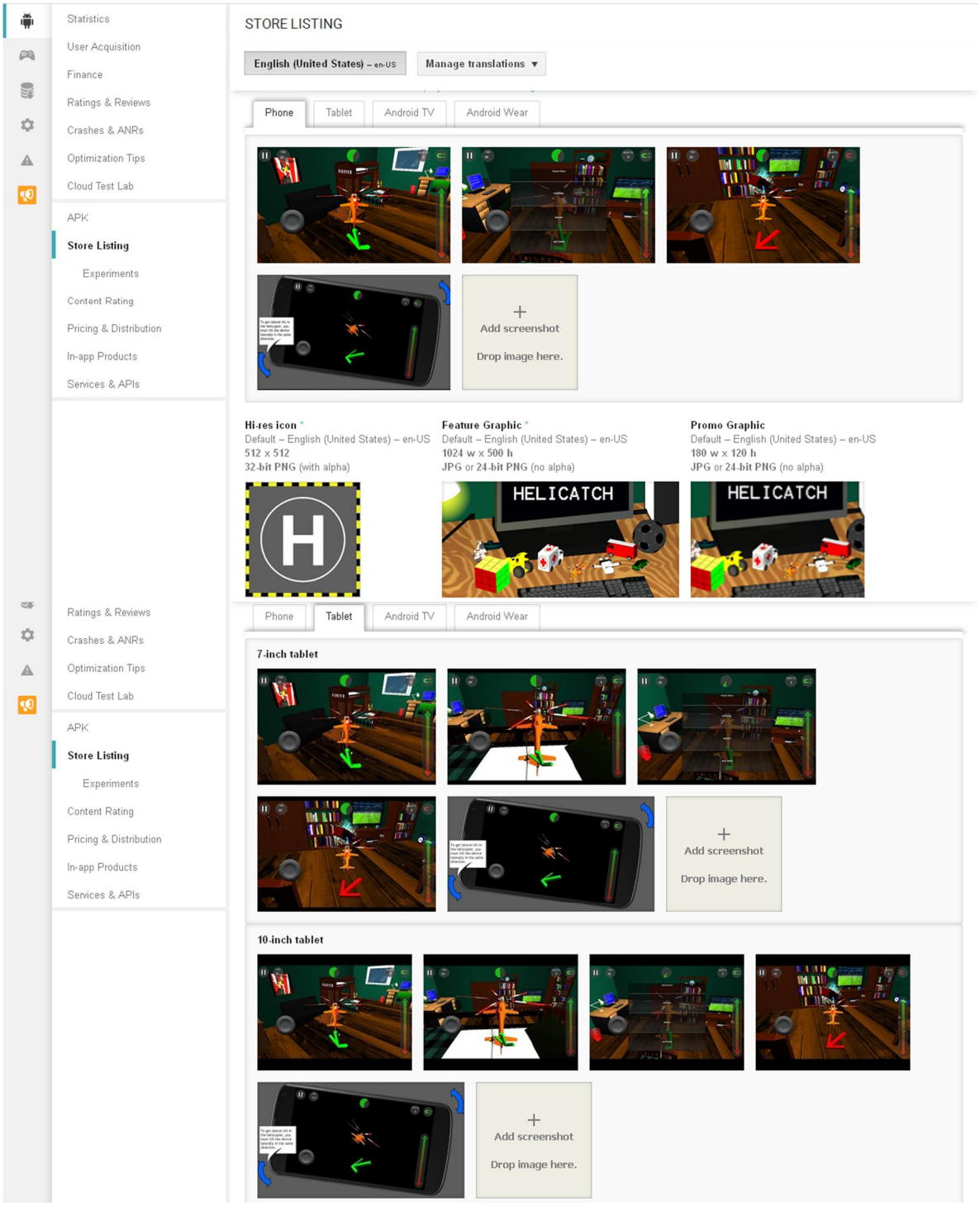

Fig 64. Captura de pantalla: requisitos para publicar la aplicación (1)

Después de eso, se debe continuar completando la lista de requisitos para poder publicar la aplicación.

El apk se puede subir fácilmente arrastrándolo al lugar indicado desde la pestaña APK, tras haber completado el envío, se podrá modificar el estado a versión Alpha, Beta o en Producción.

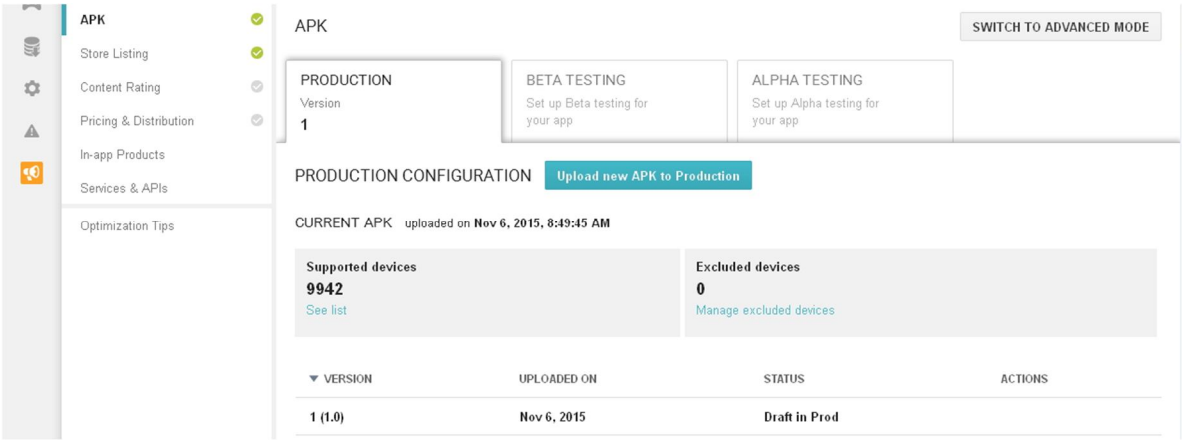

Fig 65. Captura de pantalla: requisitos para publicar la aplicación (2)

Además hay una sección de modo avanzado, que permitirá subir diferentes instaladores apks para diferentes arquitecturas, dado que algunos de los dispositivos más modernos pueden llevar la cpu x86, que provocará un fallo si el instalador apk no es el adecuado.

El instalador compatible con x86 no se puede producir a partir de versiones antiguas de Unity 3d, obligatoriamente se debe publicar el proyecto en una versión superior a Unity 4.6. Para ello habrá que seleccionar la opción:

FAT (ARM7 + x86) o (x86) en la casilla Device Filter, que aparece en la sección

“Other Settings” dentro de la pestaña de configuración “Player Settings” anteriormente comentada.

Las posibles opciones son:

FAT (ARM7 + x86): (compatible con ambos dispositivos)

ARM7: (solo dispositivos antiguos)

x86: (solo dispositivos con arquitectura x86)

El siguiente paso podría ser rellenar el cuestionario para crear un certificado que califique el contenido de la aplicación con un rango aptitud según edades.

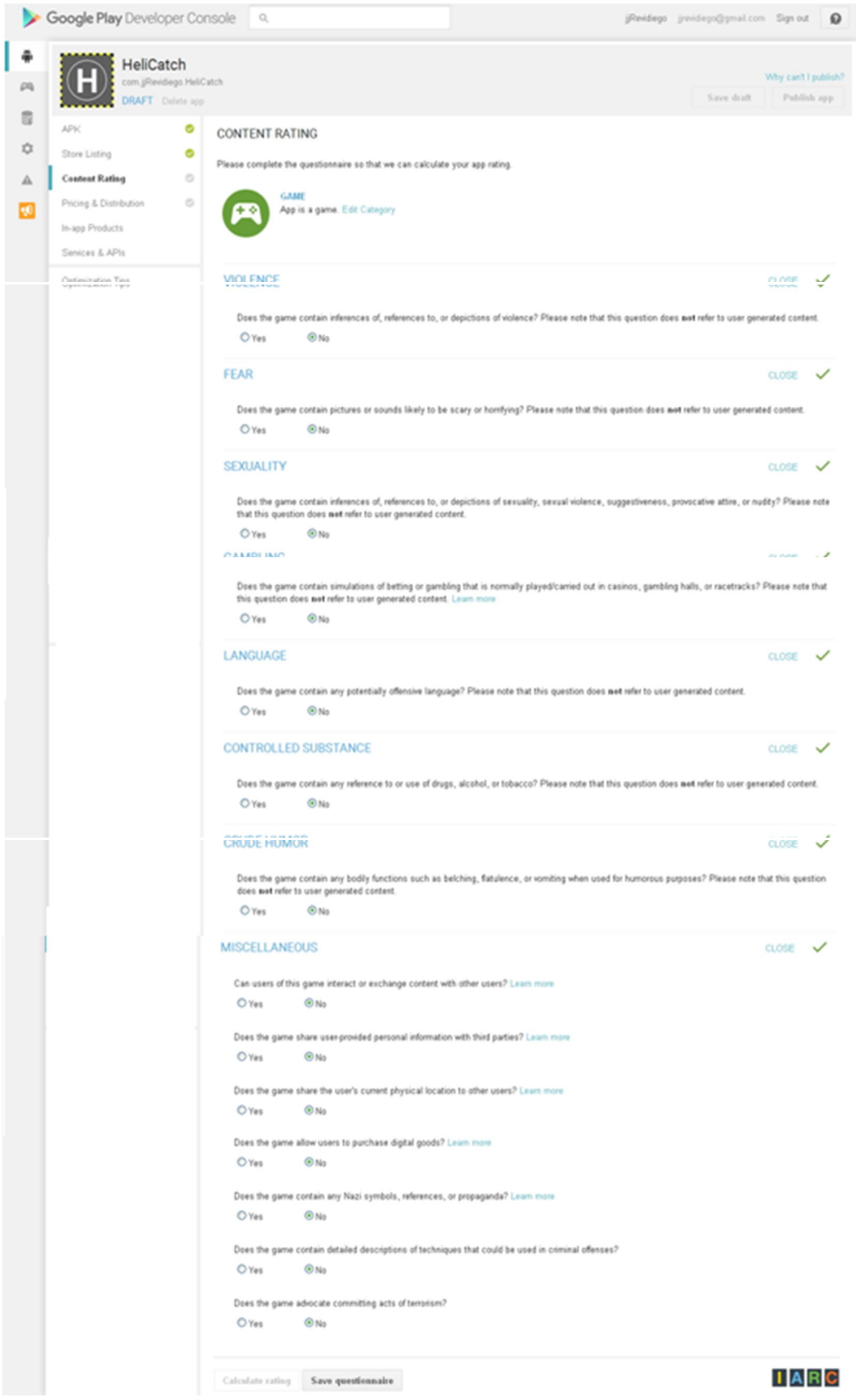

Fig 66. Captura de pantalla: requisitos para publicar la aplicación (3)

Siendo este el resultado:

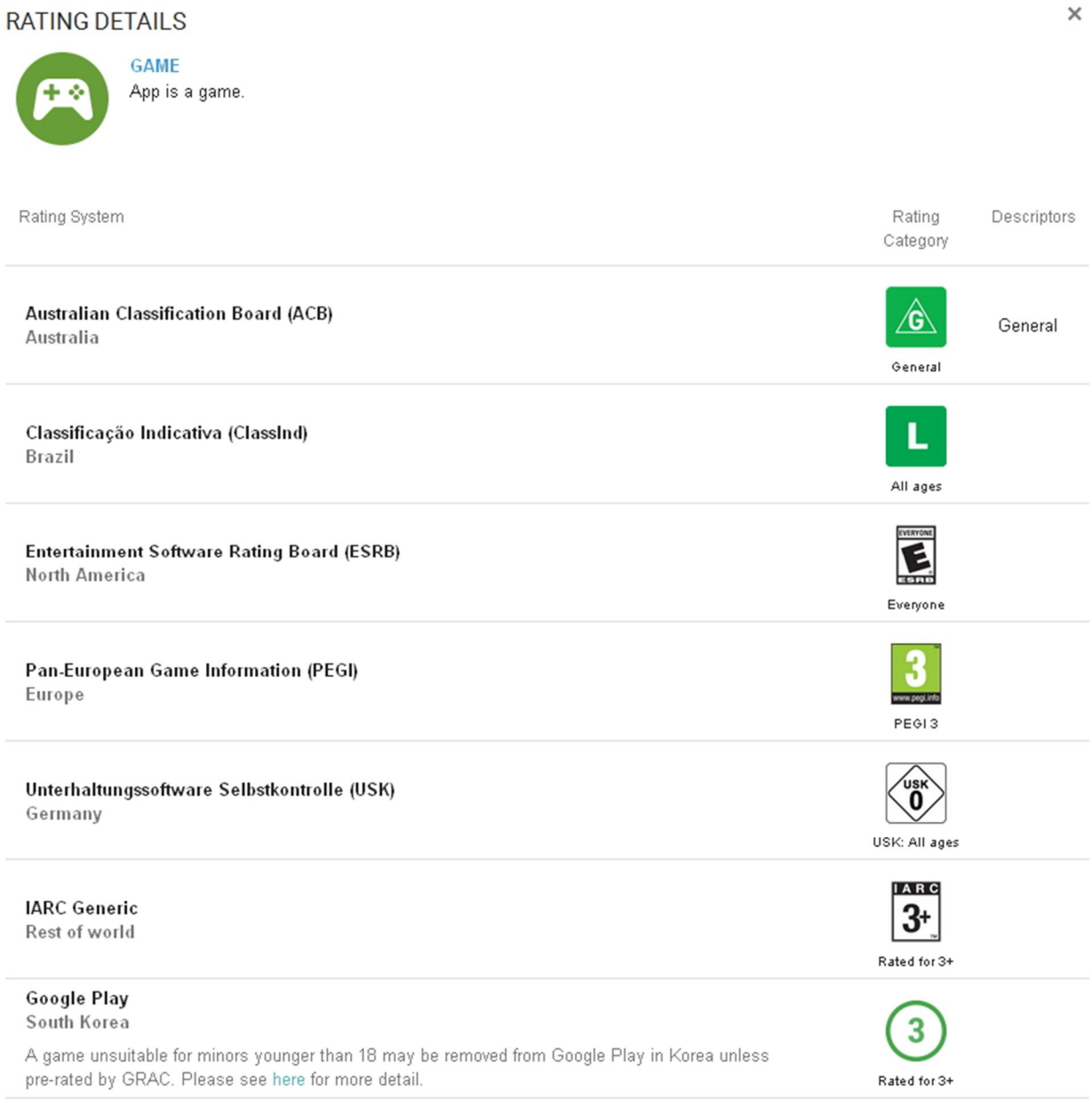

Fig 67. Certificado resultado tras realizar la encuesta.

Este certificado indicará a los posibles usuarios de la aplicación que no aparecen escenas violentas, ni sexuales, ni ningún contenido ofensivo.

La aplicación estará destinada para todos los públicos con el logotipo de Pegi 3 a nivel europeo.

Únicamente queda elegir el precio y los países en los que estará disponible.

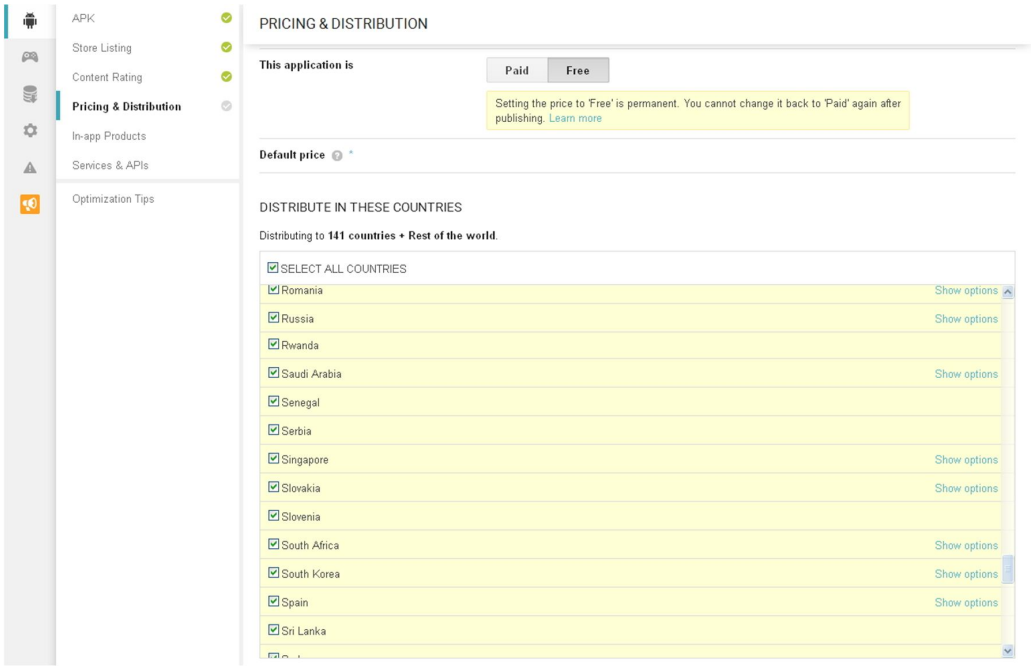

Fig 68. Captura de pantalla: requisitos para publicar la aplicación (4)

Y aceptar las condiciones de Google.

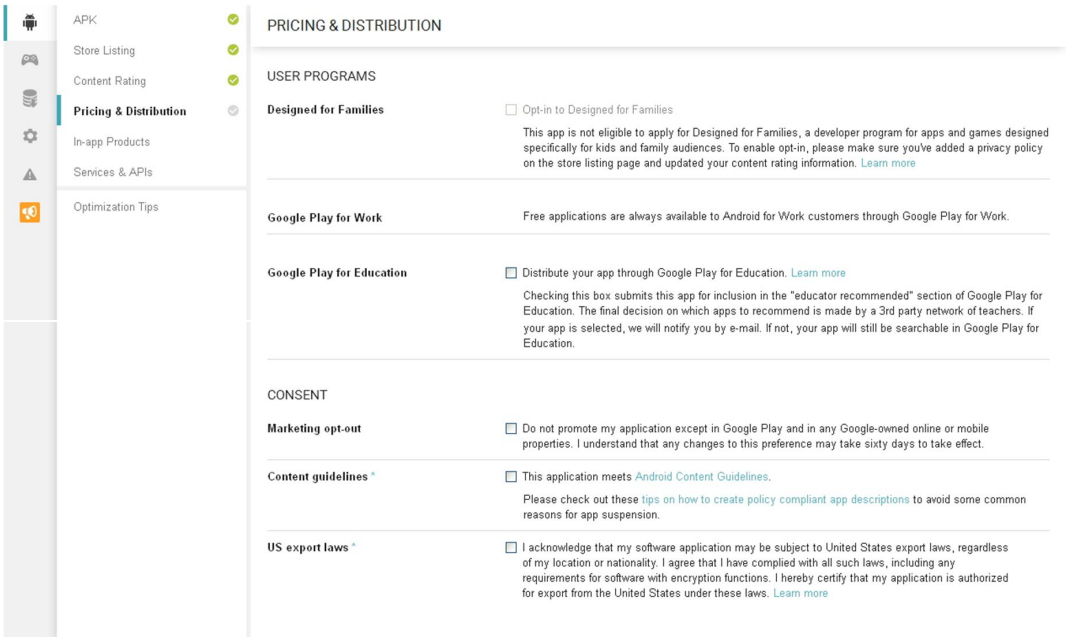

Fig 69. Captura de pantalla: requisitos para publicar la aplicación (5)

En concreto son necesarias las 2 últimas: Content guideLines, y US export laws, las cuales se deberán marcar para poder finalizar el proceso.

Tras esto, ya se podrá publicar la aplicación, lo cual actualizará el perfil del producto que en pocas horas estará disponible para su descarga.

3.4. Resultado:

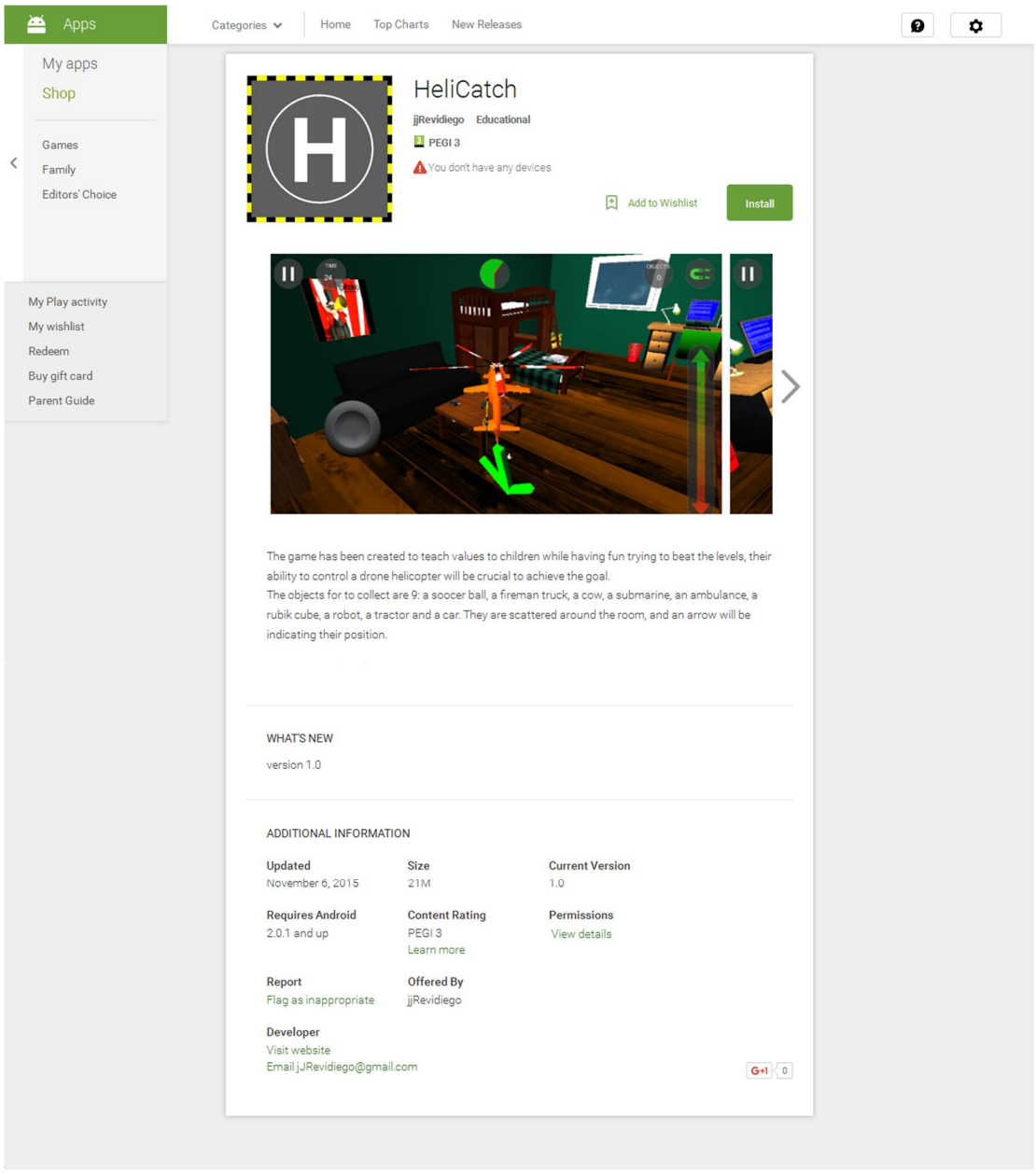

Fig 70. Captura de pantalla tras publicar la aplicación.

El resultado tras estos meses de trabajo, es una aplicación educativa y divertida, apta para todos los públicos, ejecutable en la plataforma Android. La aplicación es multi-táctil, contiene varios botones y controles, además se sirve del acelerómetro para poder indicar cierto tipo de movimientos.

Se incentiva para los niños, la buena costumbre de recoger los juguetes, y además permite mejorar los procesos cognitivos que permiten desenvolverse y situarse en un espacio tridimensional.

A continuación algunas capturas de pantalla del juego:

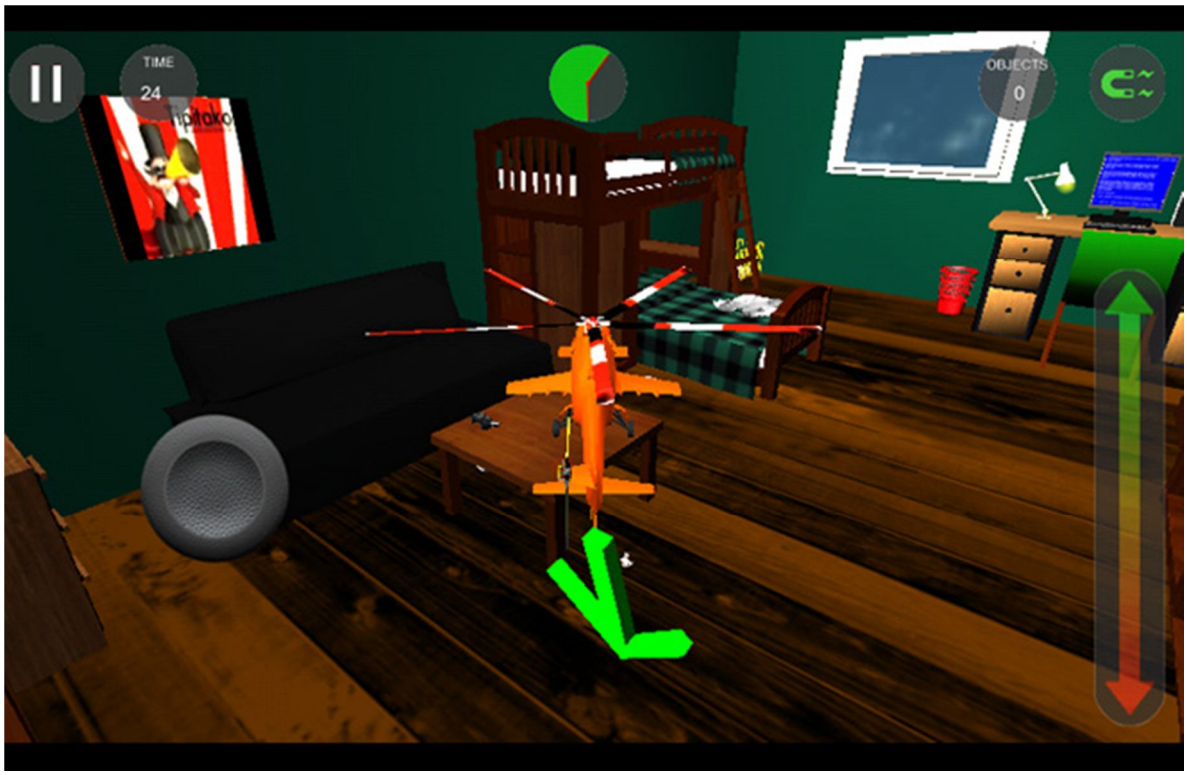

Fig 71. Captura del juego (1).

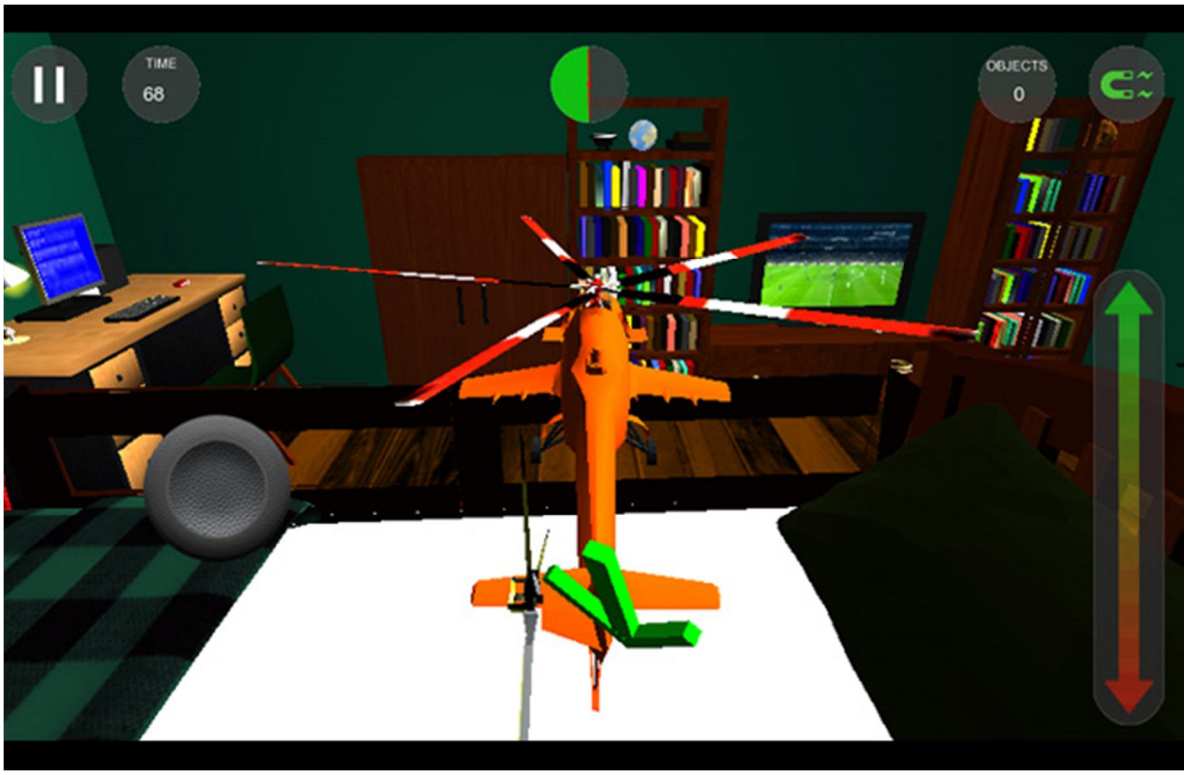

Fig 72. Captura del juego (2).

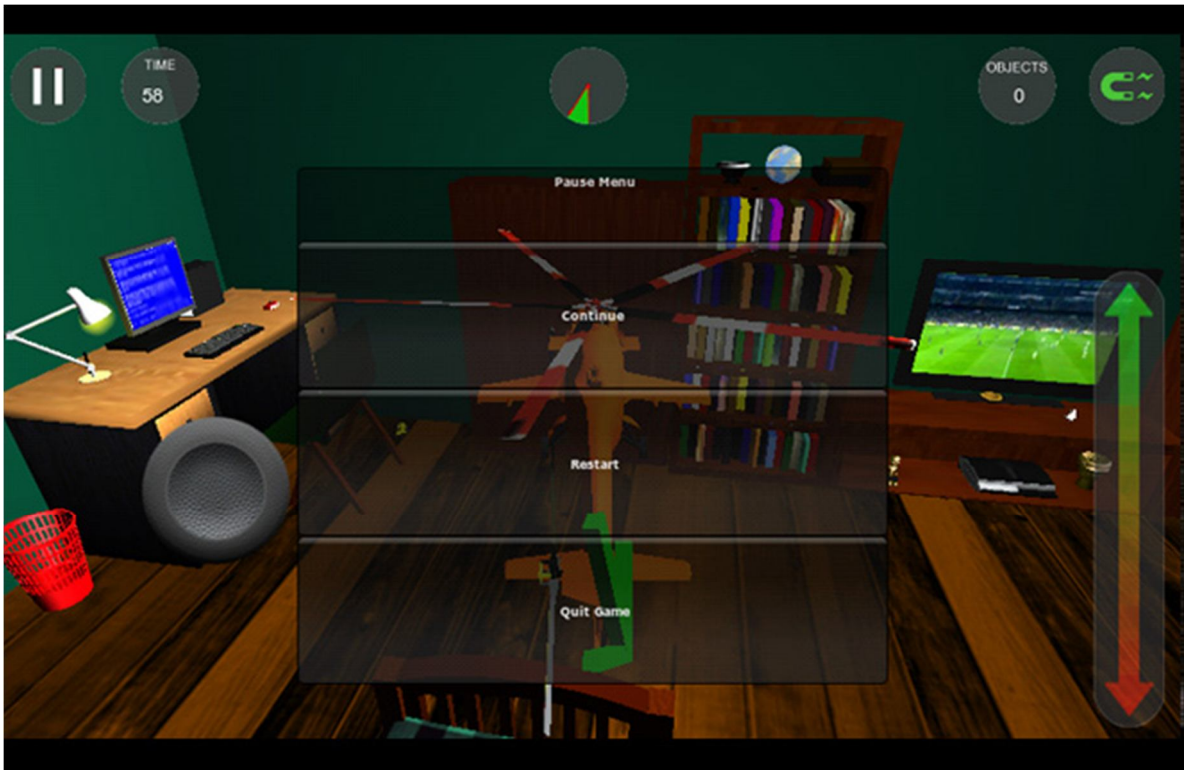

Fig 73. Captura del juego (3). Menú de pausa con las tres posibles opciones.

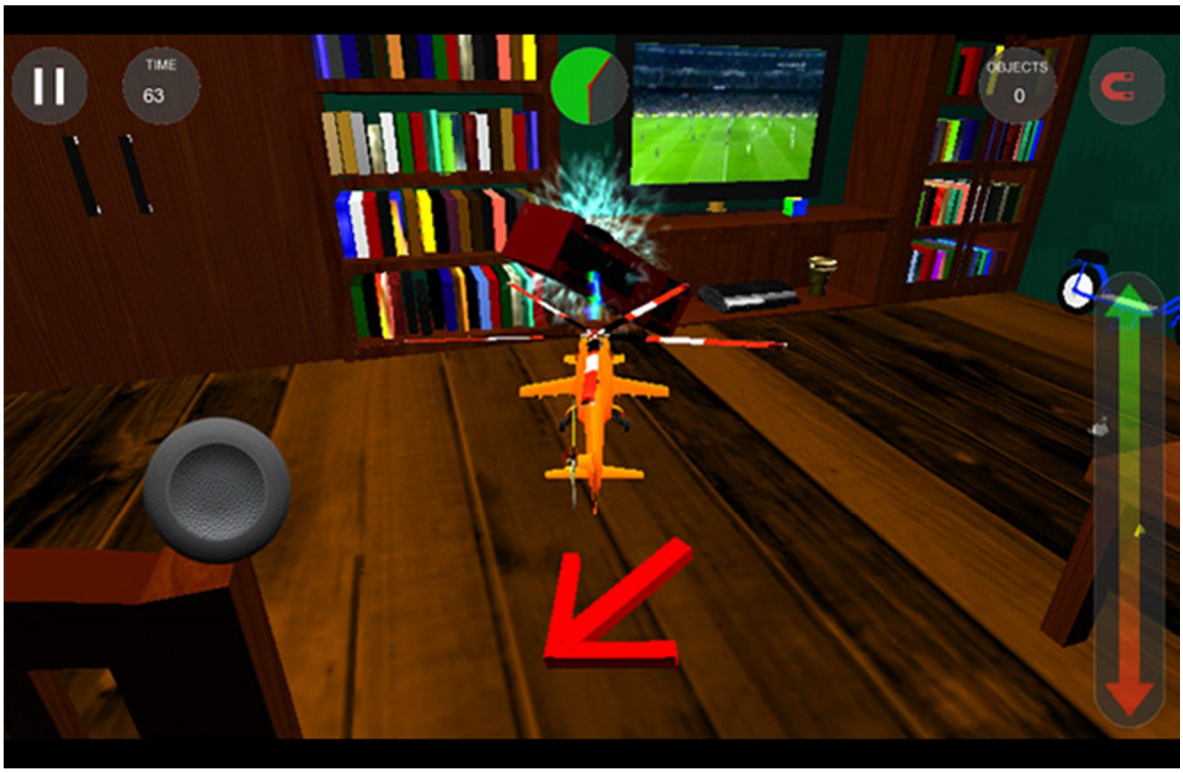

Fig 74. Captura del juego (4): Flecha roja indicando el lugar donde depositar el objeto.

3.6. Problemas encontrados:

A lo largo del desarrollo han surgido numerosos problemas y defectos que han tenido que ir solucionándose con el objetivo de que todo funcionase correctamente de acuerdo a las especificaciones.

En primer lugar era necesario poblar la escena con un numero de objetos lo suficientemente grande como para no dar sensación de vacío, pero tratando de no incluir demasiados polígonos como para que el juego se vuelva demasiado pesado a la hora de procesarlo y almacenarlo.

Para ello se tuvo que realizar una búsqueda de múltiples objetos con licencia para ser usados de forma gratuita, y tras esto realizar una selección de los más idóneos para modificar, finalmente siendo necesario crear muchos de ellos desde cero, ya que en su mayoría los objetos descargables contenían un numero de polígonos extremadamente alto e inútil a la hora de observar el objeto desde largas distancias.

Una vez hecho el decorado surgía el problema de dar movimiento al objeto principal, el helicóptero. Por una parte se deseaba lograr un movimiento suave y realista del helicóptero, para ello fue necesario investigar hasta dar con la solución, el uso de los valores DeltaTime y la función Slerp para la modificación de Quaterniones, cuyo resultado permite al helicóptero recuperar su posición de estabilidad.

Además de reproducir el vuelo del helicóptero, había que capturar los objetos. La forma de hacerlo finalmente fue mediante un objeto “cátcher” que al activarse comenzaba a generar una fuerza de atracción a sobre los múltiples objetos “catchables”, haciendo que estos se moviesen hacia el origen de tal fuerza, quedando el atrapado el primero en llegar. Hubo que estudiar la forma de trabajar de este objeto ya que en cuanto uno de los “catchables” entrase en el cátcher, este debía quedarse atrapado, haciendo que la atracción hacia el resto de objetos se detuviera, para atrapar el objeto lo ideal era eliminar el efecto de la gravedad sobre este y se activaba su propiedad kinematic, tras esto solo quedaba aplicar el mismo tipo de movimiento del helicóptero al objeto.

Dado que el juego se desarrolla en una habitación cerrada, eran comunes las colisiones con paredes y con el techo, esto ocasionaba varios problemas. En primer lugar era necesario evitar que el jugador chocase con el techo siendo habitual que un jugador inexperto presione el acelerador sin ser consciente del rápido ascenso hacia el techo. También había que garantizar un estado de "Game Over" si el helicóptero volcaba quedando bloqueado contra alguna pared o suelo.

Pero los problemas más importantes respecto a las colisiones, se originaban cuando un objeto atrapado por el cátcher entraba en la zona de colisión con otro objeto, este seguía las órdenes de permanecer en su posición, lo que ocasionaba que unos objetos atravesasen la pared, el suelo y otros objetos decorativos. La solución ideal era soltarlos, desactivando el catcher si estos colisionaban con cualquier otro objeto del decorado.

Teniendo en cuenta que originalmente los objetos "catchables" se encuentran posicionados sobre alguna superficie, o lo que es lo mismo, tocando la zona de colisión de otros objetos, la atracción generada por el catcher para atraer los objetos era totalmente necesaria, aun así persiste el problema, obligando al jugador a posicionar el catcher lo suficientemente distante como para evitar que al intentar atraparse este tenga que soltarse por seguir en contacto con alguna otra superficie o "collider".

Otro problema aparte era calibrar la sensibilidad de los controles para poder ser manejados adecuadamente, siendo esto requisito imprescindible para el uso del acelerómetro, ya que nadie querría tener que girar el dispositivo 90º para poder realizar un movimiento, y a la vez se deseaba que este fuese un movimiento suavizado aplicando la cantidad de fuerza correspondiente de acuerdo al valor de movimiento en los controles.

Las ecuaciones que se usaron para solucionar este problema fueron:

"$y = x^3$" para el joystick, siendo $x \in \{-1,1\}$

"$y = \tanh (2.5x)^3$" para el acelerómetro, siendo $x \in \{-1,1\}$

Aparte de esto, ha sido necesario posicionar correctamente las texturas y colliders, tratando de utilizar formas simples para evitar, encontrar los archivos de sonido para dar más vida al juego y mejorar la interactuación de los jugadores, y finalmente probar todo en su conjunto tratando de detectar y eliminar todo tipo de comportamientos indeseados, además de recoger las opiniones y propuestas de las personas que me han ayudado a probarlo, a las cuales agradezco su interés y paciencia con cada nueva versión que les llevaba.

Como último problema, cabría mencionar la falta de recursos y financiación, la cual hubiera acelerado y mejorado el proceso de desarrollo tanto en coste de tiempo como en calidad, ya que el proyecto ha sido realizado en ordenadores con una potencia bastante limitada, y mediante software en versiones de prueba gratuitas.

4. Conclusiones y líneas Futuras.

4.1. Conclusiones Técnicas

Como conclusión desde el punto de vista técnico, he de decir que Unity 3D es una estupenda herramienta para el desarrollo de cualquier aplicaciones tanto en 2d como en 3d, soporta 2 estupendos lenguajes como son C# y JavaScript, además de facilitar enormemente el desarrollo permitiendo realizar pruebas en tiempo real sin falta de utilizar un simulador. He utilizado JavaScript en todos los scripts por comodidad debido que en la guía o referencia de código que viene con Unity 3D, aparece todo bien ejemplificado con JavaScript como predeterminado, también permite cambiarlo a ya sea C# o Boo, pero ya más profundamente en la referencia hay algunas cosas que solo vienen documentadas en JavaScript.

Porque Unity 5? Unity 3D cuenta múltiples versiones, comencé utilizando la versión 3.4, pero finalmente me vi obligado a migrar el proyecto a la versión 5, debido a que solo es posible generar el instalador apk compatible con arquitecturas x86, en versiones superiores a la 4.5. La migración fue tremendamente fácil y cómoda pese a que cambiaba bastante sintaxis en el código, Unity 5 permite realizar las modificaciones automáticamente. Además, aunque no la utilicé, Unity 3D cuenta con una tienda online donde sería posible comprar objetos con comportamientos ya definidos, con scripts y plugins configurables, para descargarlos de forma directa y gratuitamente si su licencia lo permite, siendo mucho más completa para la versión 5. También hay multitud de proyectos y tutoriales online para facilitar el aprendizaje, y una gran comunidad dispuesta a ayudarte a resolver las dudas mediante el uso de múltiples foros.

Cabe destacar la gran compatibilidad de este software con todo tipo de formatos y programas de apoyo, necesarios para crear los contenidos que se usaran aquí.

Sobre PlayStore es realmente simple, permite monetizar los proyectos con un 70% del beneficio obtenido tras las ventas, lo cual no está nada mal.

4.1. Conclusiones Personales

Como conclusión desde el punto de vista personal, he de decir que el desarrollo de aplicaciones para entretenimiento es algo que considero de poca relevancia para el futuro de la humanidad, pero dado el creciente mercado existente a nivel mundial, debe de considerarse seriamente la realización de proyectos de este tipo, ya que pueden otorgar grandes ingresos.

El proceso de desarrollo incluyendo las fases de pruebas puede resultar incluso divertido, ya que tras codificar el comportamiento deseado, se comprobaba que este se realizase correctamente según las especificaciones, convirtiéndose cada prueba en un pequeño juego donde el objetivo era detectar los posibles defectos, ir probando y calibrando la sensibilidad de los controles resultaba en parte divertido.

Durante parte del desarrollo, el juego en si no tenía demasiado aspecto de producir diversión, pero finalmente tras completarse todas las fases del ciclo de vida del proyecto puedo decir que el reto de manejar el helicóptero correctamente puede resultar incluso adictivo, el acelerómetro ha sido esencial en este ámbito.

Como conclusión personal final, podría decir que el desarrollo de aplicaciones de entretenimiento para Android, será casi con total seguridad algo en lo que ocupe parte de mi tiempo libre.

4.3. Líneas futuras

Este proyecto en sí, podría continuarse ampliando el número de mapas, aeronaves y objetivos.

Con unos pequeños cambios en el script principal podría realizarse el vuelo de un drone quadricóptero o cualquier otro tipo de aparato volador o vehículo, ya sea real o fantástico.

He de decir que ya he realizado esos cambios y probado como sería el vuelo del quadricóptero, tenía buen aspecto, pero me llevaría más tiempo calibrarlo adecuadamente, finalmente opté por dejar el proyecto con únicamente el helicóptero como aeronave y quizá utilizar el código realizado para el quadricóptero en futuros proyectos, ya sea en otra aplicación de entretenimiento diferente o en esta misma actualizándola a la versión 2.0.

Aquí una imagen del quadricóptero en pleno vuelo dentro del escenario principal de este proyecto (el modelo es de escasa calidad gráfica ya que fue realizado únicamente para pruebas):

Como decía, podrían realizarse otros modos de juego, carreras a través de un escenario boscoso donde el objetivo sea llegar el primero a la meta esquivando los árboles, o disparar proyectiles contra algún enemigo, aunque esto último lo considero menos apropiado para el público general hacia el cual irían dirigidas estas aplicaciones.

5. Bibliografía

5.1. Libros

Getting started with Unity (Patrick Felici)

Creating games with Unity and Maya (Adam Watkins)

5.1. Webs

Física y vuelo de helicópteros:

http://www.aero.us.es/heli/downloads/0809/actuacionesAterrizajeDespegue.pdf

http://www.aero.us.es/heli/downloads/0809/actuacionesVueloAvanceInclinado.pdf

https://upload.wikimedia.org/wikipedia/commons/1/12/Dise%C3%B1o_de_helic%C3%B3pteros_y_otras_aeronaves_diversas_-_Ingenier%C3%ADa_aeron%C3%A1utica_-_ETSEIAT_-_UPC.pdf

http://upcommons.upc.edu/bitstream/handle/2099.1/9728/Documento%20-%20Memoria%20y%20Presupuesto.pdf?sequence=1

Unity 5:

https://unity3d.com/es/learn

http://docs.unity3d.com/es/current/Manual/UnityManualRestructured.html

http://unityspain.com

http://unityscripts.blogspot.com.es

http://stackoverflow.com/questions/tagged/unity3d

Software Complementario:

https://images.autodesk.com/adsk/files/gettingstartedmaya2011.pdf