Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Creación de un juego de gamificación doméstica en Unity

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Jesús Juan González Nieto

Tutor: Ramón Pascual Mollá Vayá

Curso 2014-2015

Resumen

La realización de las tareas de casa es algo difícil de gestionar, tanto para niños como para jóvenes o adultos. En un hogar medio es común que se produzcan discusiones porque alguien no ha hecho lo que debía o lo ha hecho mal. Y una de las principales causas de todo esto es la falta de motivación y organización.

El objetivo de este proyecto es desarrollar una aplicación que ayude a un núcleo familiar a gestionar las tareas domésticas. De modo que se obtenga una motivación mediante una recompensa virtual (y más adelante una recompensa real) cada vez que se realicen o se intenten realizar. Debido la inclusión de objetivos o metas se puede conseguir que las tareas se lleven a cabo en el periodo esperado. Esta aplicación está orientada a un soporte de dispositivo móvil ya que da mayor facilidad a la participación de toda la familia y mayor comodidad por la capacidad táctil de este tipo de soportes

Palabras clave: hogar, gamificación, videojuego, Android, Unity3D.

Tabla de contenidos

Índice de ilustraciones ... 8
1. Introducción ... 10
   1.1. Motivación ... 10
   1.2. Objetivos ... 10
   1.3. Estructura de la obra ... 11
   1.4. Concepto de gamificación o juguetización ... 12
   1.5. Concepto de conductismo y su relación con la aplicación ... 13
      1.5.1. Condicionamiento Clásico ... 14
      1.5.2. Condicionamiento Operante ... 15
2. Estado del arte ... 16
   2.1. Crítica al estado del arte ... 16
   2.2. Propuesta ... 17
3. Herramientas a utilizar ... 17
      3.1. Motor de juego ... 17
      3.2. Lenguaje de programación ... 19
      3.3. Entorno de desarrollo ... 20
4. Diseño de la solución ... 21
   4.1. Introducción al desarrollo en Unity ... 21
   4.2. Escenas ... 27
   4.3. Objetos comunes ... 42
5. Implementación ... 44
   5.1. General ... 44
   5.2. Formularios ... 48
   5.3. Listas ... 50
   5.4. Botones ... 52

6. Conclusiones 55
6.1. Análisis de los resultados 55
6.2. Relación con los estudios cursados 55
6.3. Trabajo futuro 56
7. Agradecimientos 57
8. Bibliografía 58

Índice de ilustraciones

Ilustración 1 - Captura de MonoDevelop 20
Ilustración 2 - Captura de Visual Studio 21
Ilustración 3 - GameObject botón y sus componentes 22
Ilustración 4 - Captura de la lista de prefabs 23
Ilustración 5 - Diagrama de flujo de ejecución por Unity3D 26
Ilustración 6 - Escena Home 28
Ilustración 7 - Casos de uso de Home 28
Ilustración 8 - Escena Login 29
Ilustración 9 - Casos de uso de Login 29
Ilustración 10 - Escena UserMan 30
Ilustración 11 - Casos de uso de UserMan 30
Ilustración 12 - Escena MainMenu 31
Ilustración 13 - Casos de uso de MainMenu 31
Ilustración 14 - Escena SettingsMenu 32
Ilustración 15 - Casos de uso de SettingsMenu 32
Ilustración 16 - Escena TaskMan 33
Ilustración 17 - Casos de uso de TaskMan 33
Ilustración 18 - Escena TaskToDo 34
Ilustración 19 - Casos de uso de TaskToDo 34
Ilustración 20 - Escena TaskInfo 35
Ilustración 21 - Casos de uso de TaskInfo 35
Ilustración 22 - Escena TaskToVerify 36
Ilustración 23 - Casos de uso de TaskToVerify 36
Ilustración 24 - Escena Leaderboards 37
Ilustración 25 - Casos de uso de Leaderboards 37
Ilustración 26 - Escena RewardsMan 38
Ilustración 27 – Casos de uso de RewardsMan 38
Ilustración 28 - Escena RewardsMenu 39
Ilustración 29 - Casos de uso de RewardsMenu 39
Ilustración 30 - Escena RewardsAvailable 40
Ilustración 31 - Casos de uso de RewardsAvailable 40
Ilustración 32 - Escena MyRewards 41
Ilustración 33 - Casos de uso de MyRewards 41

Ilustración 34 - Diagrama de flujo de la aplicación 42
Ilustración 35 - Leyenda de los mapas de código 44
Ilustración 36 - Mapa de código de User.cs 45
Ilustración 37 - Mapa de código de Task.cs 45
Ilustración 38 - Mapa de código de Reward.cs 46
Ilustración 39 - Mapa de código de GameManager.cs 47
Ilustración 40 - Mapa de código de ScreenManager.cs 47
Ilustración 41 - Mapa de código de TaskInformation.cs 49
Ilustración 42 - Mapa de código de TaskForm.cs 50
Ilustración 43 - Mapa de código de RewardList.cs 51
Ilustración 44 - Mapa de código de TaskListButtonCheck.cs 54

1. Introducción

1.1. Motivación

La motivación para la realización de este proyecto viene de juntarse dos cosas importantes de mi vida. Una es que desde hace varios años soy monitor de tiempo libre en una parroquia y me gusta lo que hago, enseñar a los niños a hacer las cosas bien es gratificante. Y yo también fui niño y hacer las tareas de casa no era algo con lo que disfrutara. Pero en esos tiempos lo más avanzado era una hoja con una tabla en la que cada día una persona tenía una tarea. Me habría gustado tener una aplicación como esta en mi infancia, eso es un punto importante de los que me hicieron escoger este proyecto.

Y la segunda es que esta primera razón se junta con la Informática, otro gran pilar de mi vida. Al juntarse estas dos resulto una importante llamada de atención ante este proyecto. Y además de estas dos, resulta que yo desde pequeño siempre he sido un gran jugador de videojuegos. Y educar con la ayuda de los videojuegos me parece una técnica maravillosa, porque a la vez que aprenden se los pasan bien.

De este modo, vi en este proyecto reflejados a niños que me recuerdan a mí con ganas de aprender y de jugar, y que mejor que dedicarle tiempo a mi pasión para ayudar a los niños a que aprendan a realizar las tareas de casa bien y ordenadamente con el aliciente de que están jugando a un videojuego. Pero no solo niños ya que también puede ayudar a adolescentes, jóvenes e incluso a los propios padres, tíos, abuelos, etc.

Todos estos motivos han despertado gran interés en mí respecto al hecho de desarrollar una aplicación de gamificación doméstica.

1.2. Objetivos

El objetivo del proyecto es el de diseñar e implementar una aplicación mediante la cual un núcleo familiar común pueda gestionar las tareas domésticas. Estas deben ser verificables por un usuario designado para ello y dependiendo de la realización, otorgaran una puntuación. De ese modo se ayuda al aprendizaje de la realización de tareas en casa trabajando sobre el concepto del conductismo (véase sección 1.5).

Un objetivo importante es la adaptabilidad de la aplicación a la diversidad de núcleos familiares. De modo que las tareas serán totalmente personalizables, así como la recompensa por estas y su puntuación.

Otro objetivo a tratar es la gestión de usuarios. Ya que existirán usuarios que realizaran tareas, usuarios que verificaran tareas, y también usuarios que realicen ambas cosas.

1.3. Estructura de la obra

La memoria se estructura en 5 bloques principales:

Introducción, donde se explica la justificación de realizar dicho proyecto, junto con los objetivos puestos en este. Después se introduce el concepto de juguetización, base principal del proyecto y punto muy importante. A continuación se enlazara con la parte de psicología que incluye la juguetización para dejar claro el concepto y el proyecto en general.

Análisis, en esta parte se realiza un pequeño repaso de las ideas similares al proyecto que existen en este momento en el mercado, se revisan ventajas y desventajas para terminar con la presentación del proyecto como solución a algunas desventajas existentes en el mercado actual. También se analizan las herramientas a utilizar y el porqué de la elección de estas.

Diseño, sección en la que se comienza con una pequeña introducción al diseño en Unity y luego se sigue con el diseño realizado en dicho motor para la aplicación con la explicación de cada parte.

Implementación, en esta sección se repasa la aplicación desarrollada desde el punto de vista del código, la estructuración de las clases, el uso de los *scripts* y las relaciones entre estos y el diseño de la aplicación.

Conclusiones, última parte en la que se repasan los resultados obtenidos, la relación del trabajo realizado con los estudios cursados y se plantean puntos para un futuro trabajo en el proyecto.

1.4. Concepto de gamificación o juguetización

El termino Gamificación [1] (o correctamente dicho en español, juguetización[1]) hace referencia al empleo de técnicas, herramientas y mecánicas, comúnmente utilizadas en videojuegos, aplicadas a entornos no lúdicos con el fin de crear un entorno mucho más motivador para el usuario. Se busca crear un enlace de modo que el usuario se comprometa con ese entorno y no solo se implique sino que reciba estímulos que le hagan mantener fuerte ese compromiso.

Con el avance de las tecnologías y de la informática en concreto, se ha conseguido controlar todo mediante ordenadores, smartphones, pulseras cuantificadoras, etc. Y eso es uno de los motivos por el que la juguetización es un concepto que coge fuerza y se va estableciendo en nuestra cultura. Además el crecimiento de los videojuegos en la sociedad, que ha pasado de ser un hobbie de un sector pequeño de la sociedad a tener juegos adaptados a todos los sectores, desde niños, a jóvenes e incluso a gente mayor. Y ese avance ha despertado el interés por los videojuegos en muchos otros campos como la comunicación, la educación, la psicología, la productividad o la salud. Y la juguetización es otro modo de explotar la gran evolución de los videojuegos.

Los desarrolladores de videojuegos son conocedores desde hace años de las necesidades humanas, como son los logros, el deseo de recompensa, el deseo de la expresión libre y el deseo de competitividad entre otros. Mediante la juguetización podemos aplicar esos conocimientos a un gran abanico de entornos y actividades.

Para poder conseguir que un entorno no lúdico consiga captar la atención de los usuarios como lo hace un juego hay que utilizar las herramientas de estos. Estas herramientas son cosas como los puntos, niveles y clasificaciones, de modo que el usuario vea su progreso plasmado y pueda ver que sus acciones no pasan desapercibidas. Otras herramientas serían las recompensas, los regalos o premios virtuales y físicos, una herramienta clave en la motivación de las personas ya que es un principal estimulo ante el que el usuario responde con un interés que ayuda a mantener la atención y el esfuerzo. Otra última herramienta serían los desafíos, retos o duelos, en los que se conseguiría captar la atención del usuario gracias al afán de competitividad y a la curiosidad.

---

[1] Gamificación según la RAE http://www.gamkt.com/2011/12/09/gamificacion-segun-la-rae/

Para aclarar el término juguetización voy a poner algunos ejemplos en diversos entornos. El primero, el entorno comercial. Un claro ejemplo es BBVA Game[2], una aplicación que ha desarrollado BBVA en la que por hacer tus operaciones en la web de BBVA puedes obtener puntos y canjearlos por participaciones en sorteos, regalos directos, etc. Y además puedes obtener premios si estas en puestos altos de la clasificación al terminar el mes. Se puede apreciar claramente como han usado múltiples técnicas y herramientas características de los videojuegos para que los usuarios creen un compromiso con la nombrada aplicación y como beneficios para la empresa se aumente el uso de la plataforma online de operaciones.

En otro ámbito, el educativo, tenemos el ejemplo de Duolingo[3], una aplicación gratuita que te ayuda a aprender idiomas a base de realizar traducciones que van de un nivel simple a algo más complejo. Y esas traducciones luego la empresa las utiliza para ofrecer a los usuarios un sistema de traducción muy efectivo llevado a cabo por miles de personas alrededor del mundo. En este ejemplo se obtiene una recompensa conforme realizas traducciones y lecciones a modo de puntos de experiencia que como se ha comentado anteriormente es algo que atrae al usuario y lo mantiene comprometido con la aplicación.

Como último ejemplo tenemos el ámbito sanitario, con la aplicación Pixelate[4] como muestra. Es una aplicación en la que se dispone de una mesa con una pantalla y unos tenedores que se conectan a esta y en la pantalla van apareciendo frutas y verduras que hay que comer en el orden correcto. De este modo se puede ayudar a los niños y jóvenes a que lleven una dieta más equilibrada impulsando la ingesta de frutas y verduras mediante el uso de técnicas como la puntuación o los duelos, creando una competitividad que al fin y al cabo es buena tanto para los usuarios como para la empresa.

1.5. Concepto de conductismo y su relación con la aplicación

Para entender mejor nuestro trabajo, a continuación hablaremos del concepto de aprendizaje [2] y de lo que esto conlleva. El aprendizaje podemos describirlo como un

---

[2] El juego como estrategia: BBVA Game: https://www.youtube.com/watch?v=8GhwvGfJlak

[3] https://www.duolingo.com/

[4] Pixelate http://sureskumar.com/?p=589

cambio más o menos permanente de conducta que ocurre como resultado de la experiencia. La mayor parte de la conducta es adquirida o aprendida.

Desde la psicología del aprendizaje diferenciamos dos perspectivas dominantes en el estudio de dicha área: el estructuralismo y el funcionalismo. Aunque ambas perspectivas se diferenciaban en sus supuestos, las dos compartían una debilidad en común: carecían de una metodología de investigación precisa y definida. Gracias a los esfuerzos del fisiólogo ruso Iván Pavlov, entre otros, fue apareciendo una estrategia más objetiva, con lo que nació en movimiento conductista.

1.5.1. Condicionamiento Clásico

Al iniciarse el siglo XX, Iván Pavlov, un fisiólogo ruso cuyos estudios sobre la digestión le llevaron a ganar el Premio Nobel en 1904, estaba realizando una serie de experimentos relacionados con la salivación de los perros. Para estudiar las respuestas de salivación, realizó una incisión quirúrgica en las fauces de los perros para poder recoger y medir la saliva que producían. Pavlov se dio cuenta de que tras algunas repeticiones de esta experiencia, los perros empezaban a salivar antes de ver u oler la comida; de hecho empezaban a salivar cuando el ayudante entraba en la sala, así que decidió empezar a estudiar este proceso de aprendizaje, al que llamamos hoy en día Condicionamiento Clásico.

Pavlov comenzó observando si el perro salivaba en respuesta a un estímulo determinado (campana). Como era de esperar, el perro no salivaba con el sonido de la campana, por ello Pavlov pasó al siguiente punto. El investigador tocaba la campana e inmediatamente enseñaba comida al perro, y éste empezaba a salivar; repitió el proceso varias veces y observó que el perro salivaba en cada una de ellas. Al final Pavlov tocaba la campana pero sin ofrecer comida, pero a pesar de ello el producía saliva. La campana, ante la cual el perro previamente no había respondido, provocaba ahora una respuesta de salivación. Así pues, se había producido un cambio de conducta como resultado de la experiencia y desde la perspectiva conductista, un aprendizaje.

Ahora explicaremos el aprendizaje en tres pasos o fases:

1. Un estímulo neutro es un estímulo ante el cual el organismo no responde. En el caso del perro, la campana.
2. El estímulo neutro se presenta inmediatamente antes que otro estímulo que si provoca respuesta. El segundo estímulo se denomina estímulo

incondicionado, ya que el organismo siempre responde a él sin necesidad de haber aprendido a hacerlo. En el caso del perro, la comida suponía un estímulo incondicionado ante el cual respondía con una respuesta incondicionada, la salivación.

3. Tras emparejarse con un estímulo incondicionado, el estímulo previamente neutro empieza a provocar una respuesta, de manera que deja de ser "neutro". Se convierte entonces en un estímulo incondicionado ante el cual el organismo ha aprendido a emitir una respuesta condicionada. En el experimento del perro, la campana una vez que se emparejó con la comida, se convirtió en un estímulo condicionado que provocaba la respuesta condicionada de salivación.

1.5.2. Condicionamiento Operante

Por otra parte, diferenciamos también el Condicionamiento Operante de Skinner. Este teórico del aprendizaje propuso que adquirimos aquellas conductas que van seguidas de ciertas consecuencias. Skinner afirmaba que "una respuesta que va seguida por un esfuerzo se fortalece y, por lo tanto, tiene más probabilidad de volver a producirse". Así, aquellas respuestas que son reforzadas tienden a incrementar su frecuencia. Esto podemos relacionarlo con el proyecto, ya que en la aplicación, los niños tendrán que realizar tareas diarias tales como hacerse la cama, poner la mesa, etc. (las tareas las establecerán los padres), para recibir puntos por parte de sus padres (o personas que verifiquen sus tareas) según hayan realizado dicha tarea. Estos puntos se canjearán por recompensas que los padres habrán acordado anteriormente con los menores (comprar un juego, ir al cine, jugar todos en casa, etc.). Al reforzar las respuestas "buenas" de los niños, estos tenderán a repetirlas.

Seguido a esto, diferenciamos diferentes tipos de reforzadores:

- Reforzamiento positivo: supone la presentación de un estímulo bueno después de la respuesta. Por ejemplo una sonrisa, alabanzas, regalos…
- Reforzamiento negativo: incrementa una respuesta mediante la retirada de un estímulo, generalmente de carácter aversivo o desagradable. Por ejemplo quitar tareas del hogar, disminuir los deberes, etc.

En nuestro proyecto, con la aplicación desarrollada, lo que pretendemos es que el niño adquiera el hábito de realizar las tareas en casa; por lo que usaremos el reforzamiento positivo.

2. Estado del arte

2.1. Crítica al estado del arte

Actualmente, el número de aplicaciones para gestionar las tareas es muy alto. Pero se reduce mucho si lo enfocamos a los niños y en general a la participación de toda la familia. No existen muchas aplicaciones con ese planteamiento, pero de las que existen, se pueden encontrar diversas carencias que el proyecto a trabajar intenta solventar.

Las aplicaciones dentro de este campo generalmente tratan las tareas con una verificación simple, o está hecha, o no. Un ejemplo de este problema sería la aplicación *HomeRoutines*[5] en la que se verifica que se haya completado una tarea con una estrella. Esto es un problema ya que, en caso de que una persona, bien sea niño o mayor intente hacer una tarea y no logre completarla o bien por no saber hacerla o bien por cualquier otro factor, haría que la tarea se verificase como no realizada. Este problema quedaría solventado con la propuesta de que las tareas sean evaluables en medida del desarrollo de las mismas. Si se ha hecho con ayuda puntuara menos, pero puntuara. De modo que todo esfuerzo tendrá su recompensa en términos de puntuación virtual.

En algunas de las aplicaciones de este tipo, como *OurHome*[6], nos encontramos con aplicaciones en las que los usuarios se pueden verificar sus propias tareas, lo cual puede dar lugar a que el niño o el integrante de la familia en cuestión, se verifique una tarea que realmente no ha completado. Esto lo trataremos de solventar mediante la inclusión de usuarios con capacidad de verificar las tareas, de este modo obtendremos una verificación más objetiva.

Además, en aplicaciones como *iRewardChart*[7] el niño completa sus tareas, gestiona sus recompensas pero no sabe del resto de personas de la casa. Ahí es donde entra una de las ideas que aporta este proyecto. Habrá un ranking general en el que aparecerán todos los miembros de la familia. Eso les ayudara a los niños (y también al resto de la familia) a darse cuenta de que los demás también trabajan y dará la posibilidad de obtener motivación comparando las tareas de varios miembros de la unidad familiar.

Por último, en las aplicaciones que tratan de tareas de casa no siempre queda claro en que consiste cada tarea ni donde se lleva a cabo. Por ejemplo, en la aplicación *You Rule*

---

[5] http://www.homeroutines.com/
[6] http://www.ourhomeapp.com/?lang=es
[7] http://www.irewardchart.com

Chores[8], y también en más aplicaciones, la tarea se escribe en texto pero se da por hecho que la persona encargada de hacerla sabe cómo se hace, algo que puede llevar a tareas mal realizadas o a dejadas a medias. Para ello se incluye en el proyecto la posibilidad de identificar la tarea mediante una foto, o video, aprovechando las capacidades técnicas de los dispositivos móviles y tabletas.

2.2. Propuesta

Este proyecto aporta al mercado de aplicaciones una solución para la gestión de tareas enfocada sobre todo a los núcleos familiares, esto se logra con la ayuda de entre otras herramientas, la creación de múltiples usuarios y asignación de diferentes permisos para cada uno. Añade una motivación a la realización de tareas con métodos como la verificación de tareas gradual, en la que dependiendo de cómo se haya hecho la tarea se obtendrá más o menos recompensa. Y también aporta seguridad a la hora de puntuar las tareas mediante la posibilidad de tener usuarios que verifiquen las tareas y no poder hacerlo uno mismo. Por último, brinda también la posibilidad de concienciar a los integrantes de la vivienda del trabajo realizado por el resto gracias a la inclusión de marcadores en los que podrán ver quien ha trabajado más o menos, Por lo tanto no estamos ante una simple aplicación de gestión de tareas, sino ante una aplicación que motiva a los miembros de la familia a hacer sus tareas y además, a hacerlas correctamente.

3. Herramientas a utilizar

Para la realización de un videojuego es muy importante la elección de las herramientas con las que se va a implementar puesto que esta elección determinara el camino que seguirá el desarrollo.

3.1. Motor de juego

El motor de juego es la base de todo videojuego, depende de este el rendimiento que luego se obtenga del producto final y también dependen de este las herramientas disponibles tales como *plugins*[9], manuales, soporte de la comunidad, etc.

[8] http://www.yourulechores.com/
[9] Complementos que aportan funcionalidades extra

Existen diversos factores que influyen a la hora de elegir un motor de juego, tales como el tipo de juego a desarrollar o la plataforma para la que ira destinado el juego. Esto es debido a que hay motores de juego especiales para algunos tipos de juegos, como por ejemplo, *GameMaker*[10] de la empresa *YoYo Games* es un motor de juego para la realización de videojuegos en 2D únicamente, y que usa un lenguaje de programación propio para los *scripts* (GML). Y otro tipo de motor centrado en una plataforma seria por ejemplo *jPCT-AE*, un motor de juego orientado a la plataforma de dispositivos con Android. Algunos de los motores mediante los que se podría desarrollar este proyecto serían los siguientes.

Cocos2d-x[11] es un motor de juego enfocado a los videojuegos 2D, de código abierto y soporta C++ y *scripts* en Lua o JavaScript. Tiene importantes videojuegos como prueba de su rendimiento, tales como *Badland*, *Geometry Dash* o *Plants vs. Zombies*.

Project Anarchy[12] de *Havok* es un motor de juego realizado en C++ y con soporte para *scripting* en Lua. Dispone de múltiples herramientas, una muy útil para este proyecto sería la inclusión de un editor *WYSIWYG*[13] que a la hora de diseñar interfaces sería un punto a favor.

SIO2[14] es un motor de juego orientado sobre todo a plataformas móviles y con control tactil. Basado en C++, y utiliza Lua para *scripts*. Ofrece herramientas para un fácil desarrollo de proyectos de Android de cara a integrarlos en el *Google Play*.

Finalmente, para la realización de este proyecto, se ha escogido el motor de juego *Unity3D*[15] de la empresa *Unity Technologies*. Como motivos, entre otros (véase *Tabla 1*) tenemos la gran comunidad que tiene detrás, que le aporta una magnifica documentación, gran cantidad de tutoriales [3] y videos que comprenden todas las partes del diseño y el desarrollo de un videojuego en este motor. Además de esto, existe una tienda llamada *Asset Store*[16] en la que se incluyen muchos *plugins* con los que mejorar el desarrollo del proyecto.

[10] http://www.yoyogames.com/studio
[11] http://www.cocos2d-x.org/
[12] https://www.projectanarchy.com/
[13] What You See Is What You Get. Referencia al tipo de diseño en el que ves en el editor el aspecto que tendrá el producto final.
[14] http://www.sio2interactive.com/
[15] https://unity3d.com/es
[16] https://www.assetstore.unity3d.com/

Motor	GameMaker	jPCT-AE	Cocos2d-x	Project Anarchy	SIO2	Unity3d
Plataformas	Windows, Linux, OS X, Android, iOS, Xbox, PlayStation , HTML5 y Tizen	Java y Android	Windows, Linux, OS X, Android, iOS y Windows Phone	Windows, Linux, OS X, Android, iOS, Xbox y PlayStation	iOS, Android, Bada, WebOS y Windows	Windows, Linux, OS X, Android, iOS, Xbox, PlayStation, Wii y Web.
Lenguaje de programación / *Scripting*	Delphi / GML	Java / -	C++ / Javascript	C++ / Lua	C-C++ / Lua	C# / C#, Javascript y Boo
3D / 2D	2D	3D	2D	3D	3D	3D
Licencias	Gratuita y de pago	Gratuita	Gratuita	Gratuita y de pago	Gratuita y de pago	Gratuita y de pago

Tabla 1 - Comparación de los motores de juego

3.2. Lenguaje de programación

El lenguaje de programación es una elección importante porque dependiendo del lenguaje puede haber más o menos soporte y cada lenguaje tiene sus diferentes funcionalidades. Esta elección va muy determinada por el motor de juego, puesto que los motores no son compatibles con todos los lenguajes, sino que suelen ser compatibles con uno o dos lenguajes.

En este caso, Unity3D es compatible con C#, Javascript y Boo. En mi caso escogeré C# debido a que es el lenguaje con el que más estoy familiarizado. Javascript es un lenguaje que he usado pero del que tengo un muy bajo nivel. Y Boo es un lenguaje que además de no haber utilizado nunca, posee un bajo soporte por ser el menos común en el desarrollo con este motor de juego.

C# es un lenguaje de programación orientado a objetos desarrollado por *Microsoft*. Tiene una sintaxis derivada de C/C++ y un modelado de objetos similar al de Java. Ahora mismo se usa mucho puesto que a pesar de ser desarrollado para trabajar con la plataforma .NET, creada por la misma empresa, se puede compilar para funcionar en otros sistemas además de *Windows*, como *Linux*, *Android*, *iOS* o *Windows Phone*. Estos últimos, junto con el auge de los dispositivos móviles han hecho que C# sea un lenguaje comúnmente usado para el desarrollo de aplicaciones móviles.

3.3. Entorno de desarrollo

Unity viene con MonoDevelop[17] (véase *Ilustración 1*) incluido, que es un entorno de desarrollo de código. Pero también da soporte al uso de Visual Studio[18] (véase *Ilustración 2*), el cual fue el elegido para desarrollar los *scripts*. Esto es debido a la gran familiarización de este entorno por su uso en múltiples asignaturas de la carrera, en prácticas realizadas en empresas y en su utilización para proyectos personales.

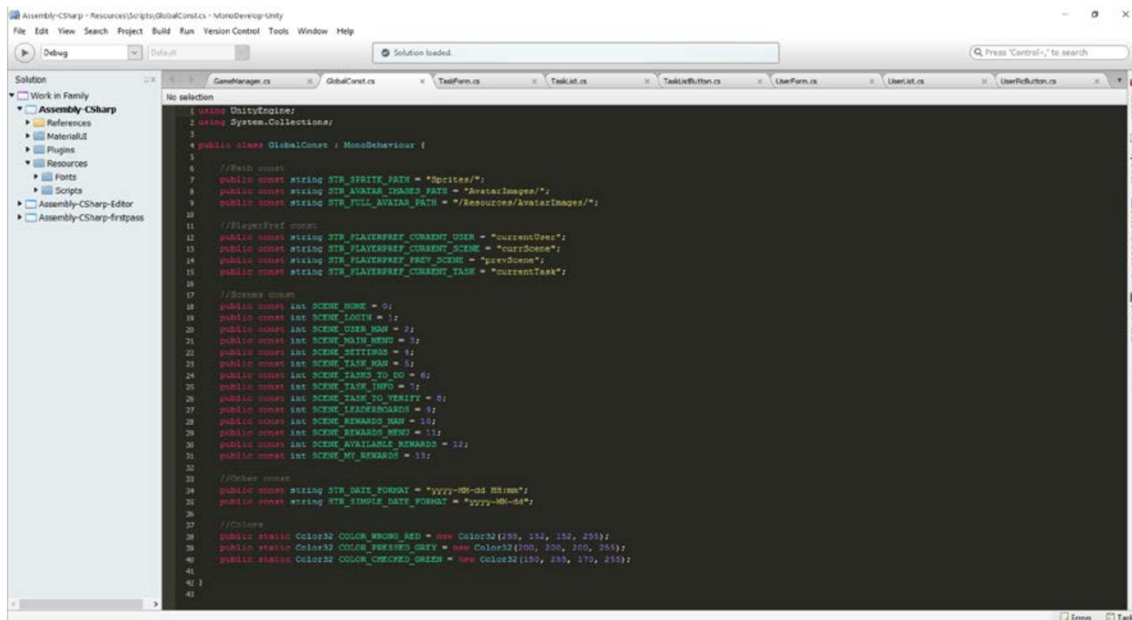

Ilustración 1 - Captura de MonoDevelop

Visual Studio es un entorno de programación desarrollado por Microsoft para plataformas Windows. Este da soporte a múltiples lenguajes de programación: C++, C#, Visual Basic .NET, F#, Java, Python, Ruby, PHP. Visual Studio proporciona un entorno de desarrollo en el que se pueden desarrollar aplicaciones para ordenadores, dispositivos móviles e incluso aplicaciones web. Todo esto respaldado por una gran comunidad de usuarios y una gran cantidad de cursos y tutoriales que se pueden encontrar en la web de Microsoft Developer Network[19]. Además de un gran soporte, cuenta con cantidad de *plugins* y mejoras para los múltiples lenguajes de programación soportados.

[17] http://www.monodevelop.com/
[18] https://www.visualstudio.com/
[19] https://msdn.microsoft.com/

Es un entorno de desarrollo que ofrece múltiples opciones de licencias, desde una opción gratuita muy completa, hasta las opciones más complejas orientadas sobre todo al sector profesional.

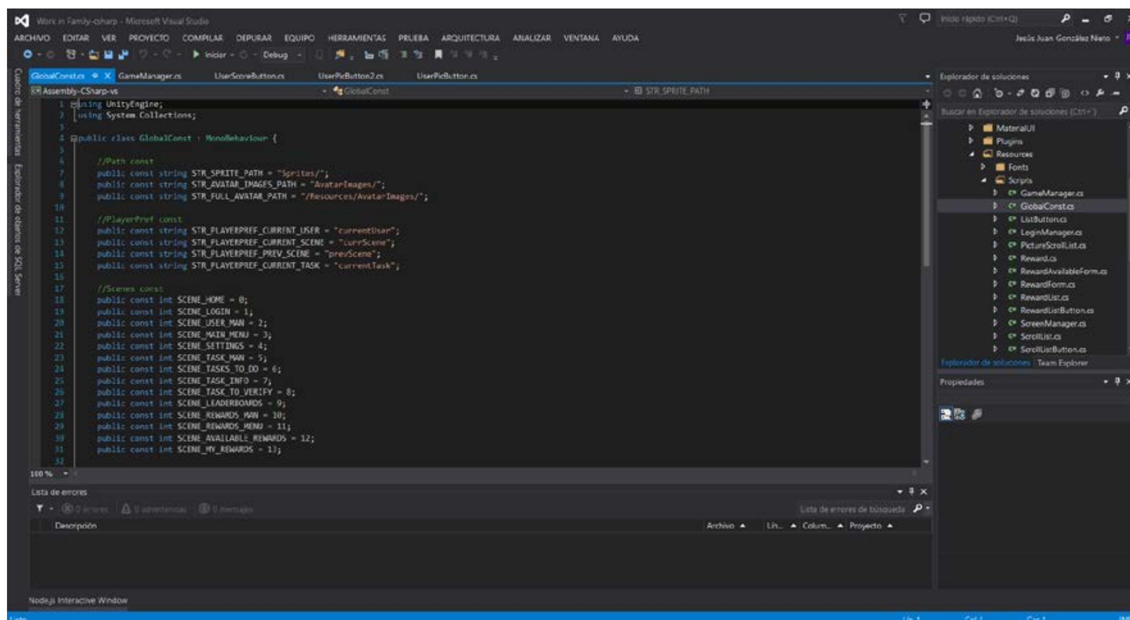

Ilustración 2 - Captura de Visual Studio

4. Diseño de la solución

4.1. *Introducción al desarrollo en Unity*

Para comenzar hablando sobre el desarrollo y diseño de la aplicación antes hay que dejar claro cómo se desarrolla en el motor elegido.

En Unity se trabaja por escenas, esto son pantallas en las que tendremos una pantalla de inicio, un menú de acceso, un menú principal, etc. Dentro de cada escena se trabaja con *GameObjects* que son contenedores de información (véase *Ilustración 3*) en los que podremos añadir o quitar componentes y que representaran desde un gestor de escenas, hasta un campo de texto o una imagen. En lo que respecta al código trabajaremos con *scripts* en C#. Estos *scripts* se añadirán como componentes a los *GameObjects* (a partir de ahora objetos) dándoles una funcionalidad más controlada ya que desde el editor de Unity podemos asignar componentes de objetos a variables públicas de un *script*. De este modo los *scripts* que gestionen un formulario y todos sus

componentes tendrán variables públicas referidas a todos los campos del formulario y así se podrá trabajar desde el código editando los campos u obteniendo su valor.

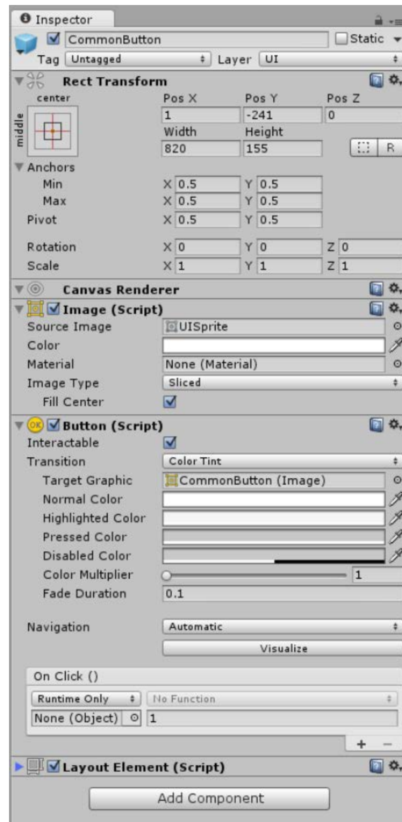

Ilustración 3 - GameObject botón y sus componentes

Otra herramienta importante de trabajo en Unity son los *prefabs* que se crean a partir de objetos. Sería una manera de guardar un objeto con todos sus componentes para de esa forma poder instanciarlo las veces que se quiera y de ese modo poder estandarizar un objeto en el proyecto. En este proyecto se han creado múltiples *prefabs* para el desarrollo de las interfaces (véase *ilustración 4*).

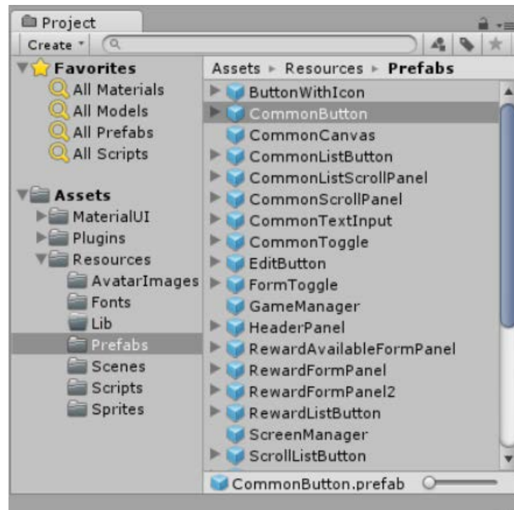

Ilustración 4 - Captura de la lista de prefabs

La vida de los objetos se define por un ciclo de vida (véase *Ilustración 5*) el cual se debe conocer ya que en este ciclo de vida encontramos métodos muy útiles para trabajar con los objetos, como el método que se llama al inicio del objeto, el que se llama cuando se actualiza, o el método que se llama justo antes de ser destruido.

Para poder entender mejor el desarrollo en Unity voy a pasar a describir brevemente estos métodos del ciclo de vida de los objetos

Editor de Unity

- *Reset*: método que inicializa un *script* cuando este es adjuntado a un objeto.

Cuando carga la primera escena

- *Awake*: función llamada después de cualquier *Start* y antes de la instanciación de un prefab.
- *OnEnable*: función que se llama, si el objeto está activo, justo cuando es habilitado. Esto ocurre cuando se ha creado una instancia de *MonoBehaviour*, se ha cargado un nivel o un objeto con el *script* como componente es instanciado.

Antes de la actualización del primer *frame*

- *Start*: método al que se llama justo antes de la primera actualización de *frame*, pero solo si el componente *script* está activo.

Entre *frames*

- *OnApplicationPause*: este método será lanzado en la pausa detectada al final del *frame*. Después de este método se lanzara un *frame* extra para poder trabajar con el estado de pausa.

Orden de actualización

- *FixedUpdate*: es un método que puede llegar a ser más frecuente que *Update* y que dependiendo de la velocidad de *frames* puede ser llamado múltiples veces en un *frame* o ninguna. Es donde se realizan los cálculos de físicas debido a que es independiente de la velocidad de *frames*.
- *Update*: función que se llama una vez por *frame*.
- *LateUpdate*: método llamado en cada *frame* una vez termina la función *Update*. Se usa para procesar cosas como la cámara, que debe calcularse una vez se hayan completado todos los cálculos de *Update*.

Renderización

- *OnPreCull*: método que es llamado justo antes de que se proceda a determinar que objetos son visibles a la cámara.
- *OnBecameVisible/OnBecameInvisible*: función que es llamada cuando un objeto se vuelve visible/invisible a alguna cámara.
- *OnWillRenderObject*: método llamado una vez por cada cámara para la que un objeto es visible.
- *OnPreRender*: llamada una vez antes de que la cámara comience el renderizado.
- *OnRenderObject*: método llamado después de que se complete todo el renderizado.
- *OnPostRender*: método llamado por una cámara al terminar el renderizado de una escena.
- *OnRenderImage*: llamado después del renderizado para procesar la imagen resultante de este.
- *OnGUI*: método lanzado multiples veces por *frame* como respuesta a los eventos GUI[20].

[20] *Graphical User Interface*: interfaz gráfica de usuario

- *OnDrawGizmos*: método utilizado para dibujar *gizmos* [21]en la escena por motivos de visualización.

Co-rutinas

- *yield*: las co-rutinas se ejecutan tras la finalización de todas las funciones de actualización.
- *yield WaitForSeconds*: continúa como la función anterior pero sumándole un tiempo especificado.
- *yield WaitForFixedUpdate*: continua una vez todos los FixedUpdate hayan sido llamados en todos los *scripts*.
- *yield WWW*: se ejecuta una vez una descarga haya sido completada.
- *yield StartCoroutine*: encadena las co-rutinas esperando hasta que termine la anterior.

Cuando el objeto es destruido

- *OnDestroy*: función que se llama tras la actualización del ultimo *frame* en la vida de un objeto, justo antes de su destrucción.

Cuando se sale de la aplicación

- *OnApplicationQuit*: esta función es llamada en todos los objetos cuando la aplicación va a cerrarse. También funciona en el editor de Unity cuando se para la reproducción.
- *OnDisable*: método que se llama en un objeto cuando pasa a estar deshabilitado o inactivo.

Debido a este tipo de desarrollo nos encontramos con puntos a favor, como el hecho de que puedes trabajar con objetos independientemente añadiéndoles un *script* y de ese modo la organización es más sencilla. O el hecho de que las variables de un *script* se pueden ver y editar desde el editor de Unity que ayuda para la depuración de la aplicación,

[21] Estructura grafica que se usa en diseño con 3 líneas en la que cada una marca un eje.

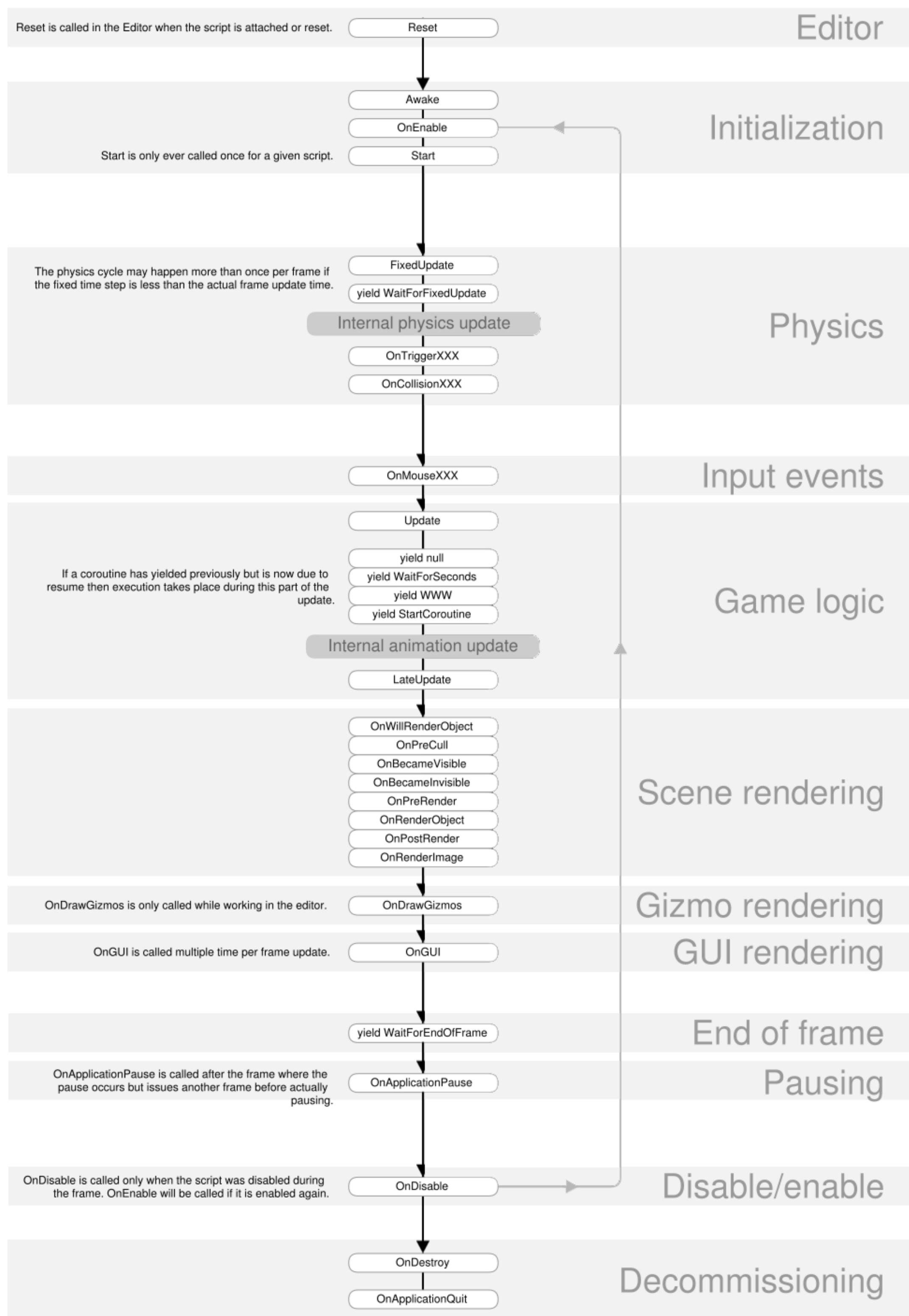

Ilustración 5 - Diagrama de flujo de ejecución por Unity3D

También nos encontramos con puntos en contra, como el hecho de que al cambiar de escena los objetos son destruidos. Esto se puede solventar con el uso de patrones *singleton*[22], además de la utilización de un archivo persistente llamado *PlayerPrefs* donde se pueden almacenar variables numéricas o de cadenas de texto pero cuyo uso es bastante limitado ya que solo almacena variables del tipo *int*, *float* y *string*.

Y otra desventaja es que al ser un motor orientado al desarrollo de videojuegos, el soporte para desarrollo de formularios es escaso, solo existen botones, campos de texto, etiquetas y varios campos básicos más. Algo que puesto que estamos desarrollando una aplicación que a pesar de usar mecánicas y herramientas de videojuegos no llega a ser un videojuego sino que es una aplicación, hace difícil el desarrollo de algunas partes.

4.2. Escenas

A continuación vamos a desarrollar las escenas que conforman la aplicación mediante una breve descripción de la escena, seguida de una captura con el diseño de la interfaz (o varias en caso de que la interfaz disponga de algún formulario o ventana emergente) y el diagrama de casos de uso relacionados con esta escena.

A continuación del listado de escenas podremos ver el diagrama de flujo de la aplicación (véase *ilustración 34*) en el que se pueden apreciar los cambios de una escena a otra y todo el árbol del escenas que empieza en la pantalla inicial y se centra en el menú principal.

[22] http://unitypatterns.com/singletons/

- Home: esta escena es la inicial, a la que accedemos nada más abrimos la aplicación. El diseño de esta escena se compone de una imagen principal de la aplicación, un botón que nos abrirá un panel de autentificación con las imágenes de los usuarios, y un botón que nos llevara a la escena de gestión de usuarios.

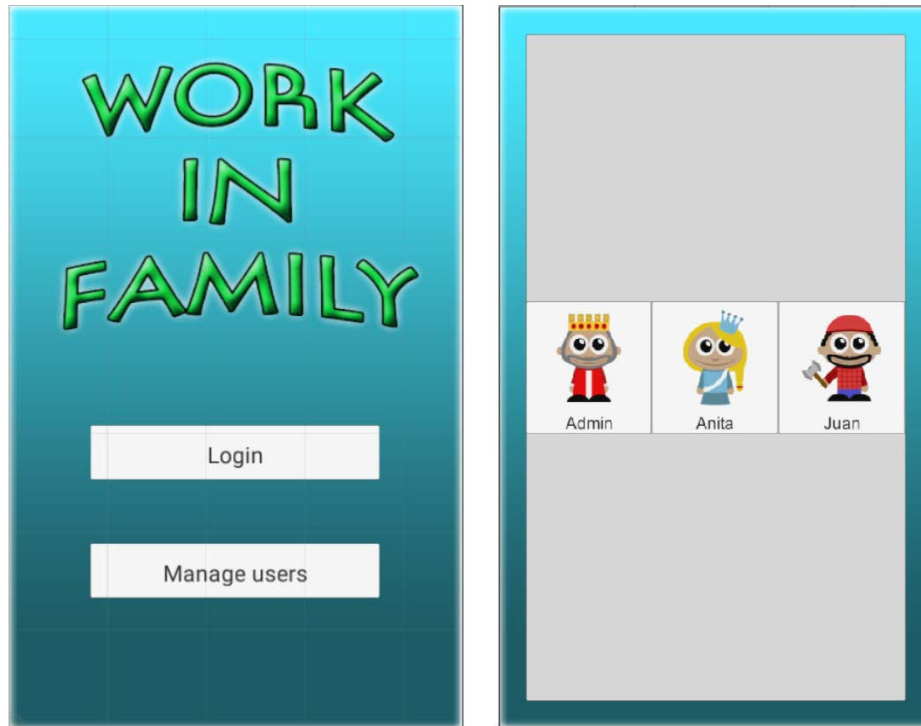

Ilustración 6 - Escena Home

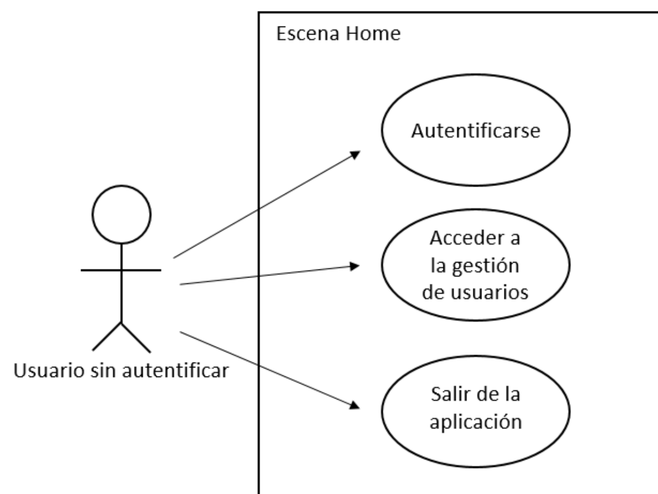

Ilustración 7 - Casos de uso de Home

- Login: aquí encontraremos el formulario de acceso al que solo se deberá de acceder en caso de haber elegido al administrador como usuario. Consta de dos campos para el usuario y contraseña, un botón de aceptar y un botón de cancelar.

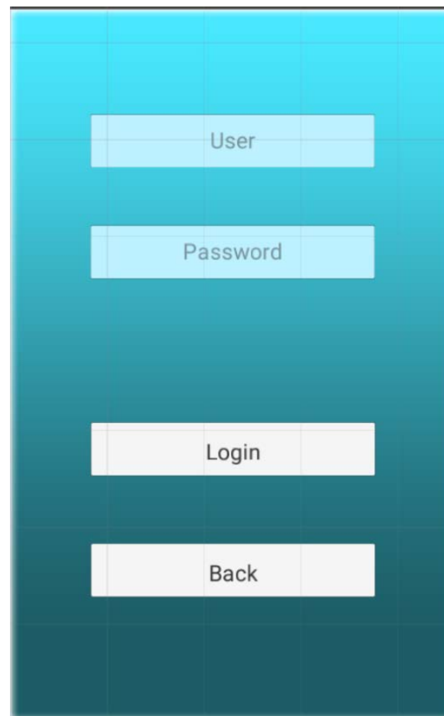

Ilustración 8 - Escena Login

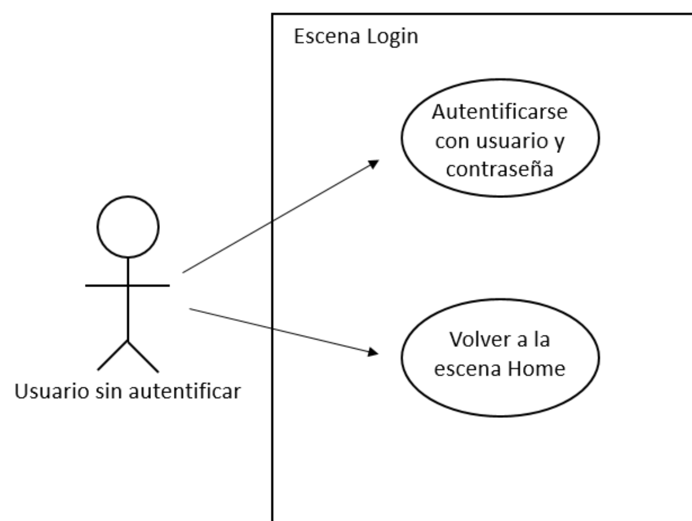

Ilustración 9 - Casos de uso de Login

- UserMan: como abreviatura de *User management* es la escena que representa el menú de gestión de usuarios. Está compuesta por una cabecera de texto, una lista que se genera automáticamente con los usuarios que existan y unos botones para añadir, editar o eliminar usuarios.

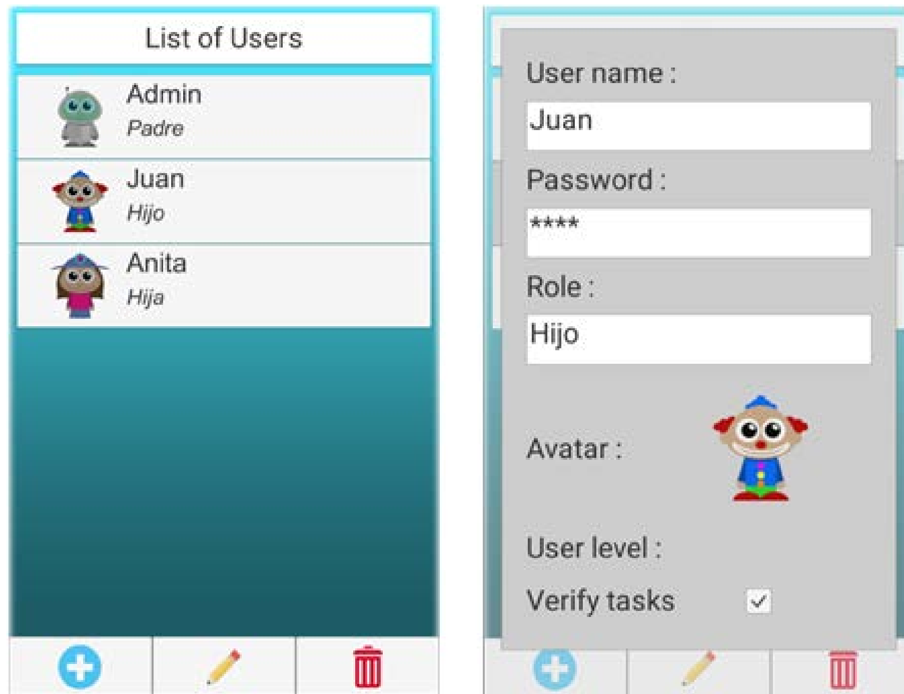

Ilustración 10 - Escena UserMan

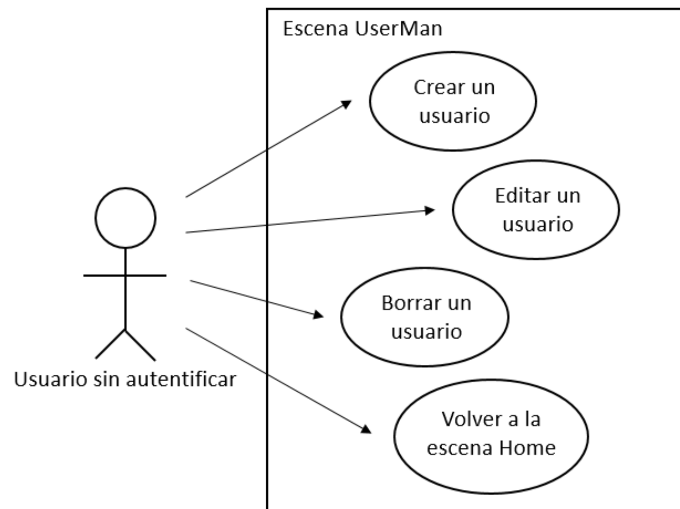

Ilustración 11 - Casos de uso de UserMan

- MainMenu: escena principal de la aplicación, el menú principal desde donde se accede a las funciones principales de esta. Dependiendo de los permisos del usuario tendrá unos botones u otros. Desde este menú podemos acceder a las funciones de chequear tareas, verificar tareas chequeadas, gestionar tareas, ver los marcadores, ver las recompensas y acceder a las opciones de la aplicación.

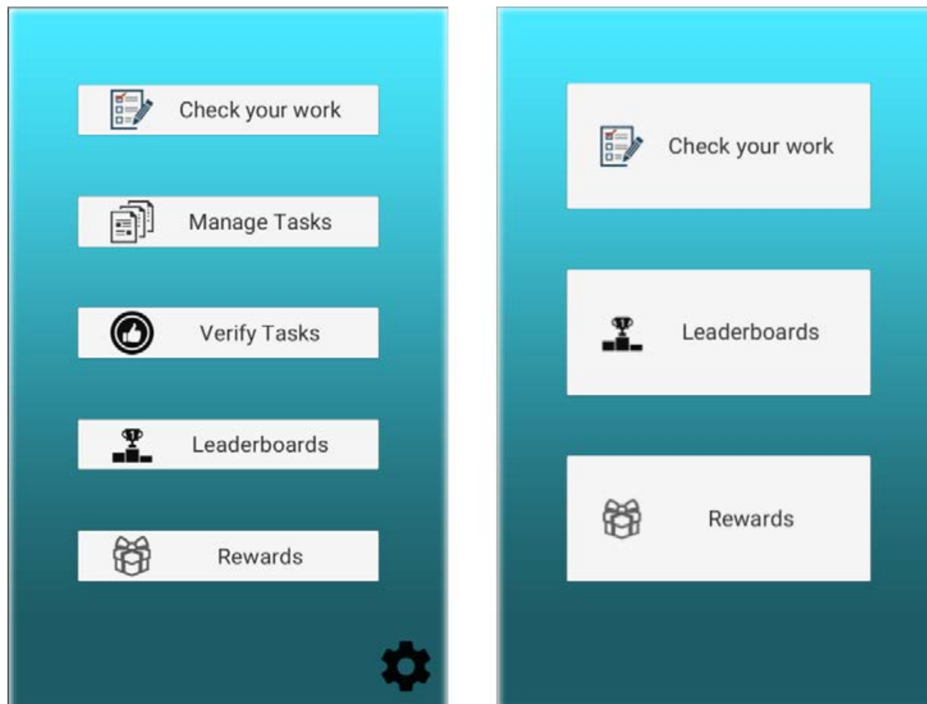

Ilustración 12 - Escena MainMenu

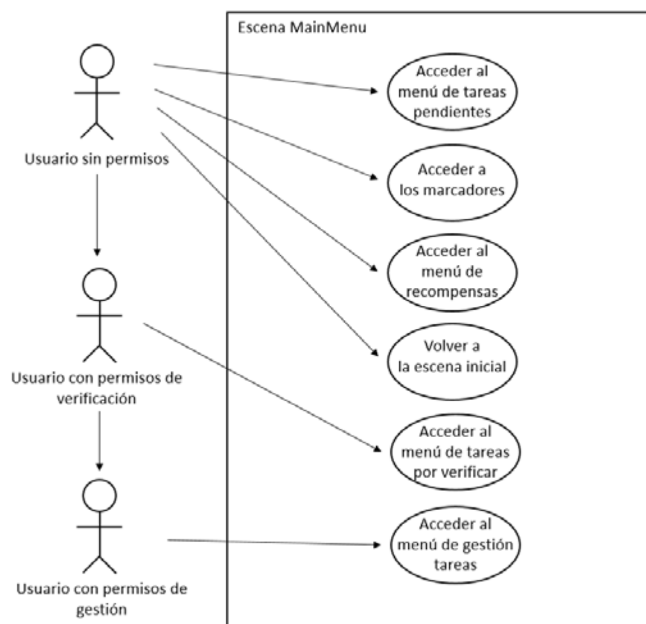

Ilustración 13 - Casos de uso de MainMenu

- SettingsMenu: menú de opciones de la aplicación. Puesto que es una aplicación orientada además de a mayores y jóvenes, a niños, el menú de opciones solo será visible para el administrador y solo dispone de una opción para borrar todos los datos y así empezar de nuevo.

Ilustración 14 - Escena SettingsMenu

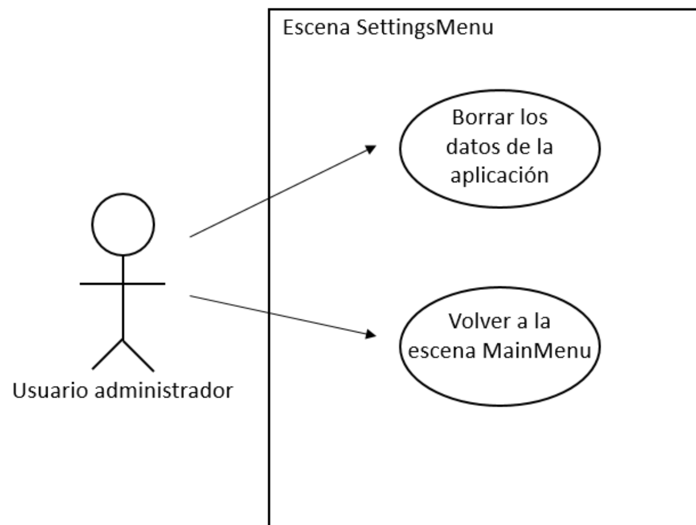

Ilustración 15 - Casos de uso de SettingsMenu

- TaskMan: abreviatura de *Task management*, escena en la que se podrán ver las tareas creadas, y además, se podrán crear nuevas, así como editar o borrar las existentes. Para el proceso de creación o editado de una tarea se invocara un formulario que en el segundo caso contendrá los datos de la tarea.

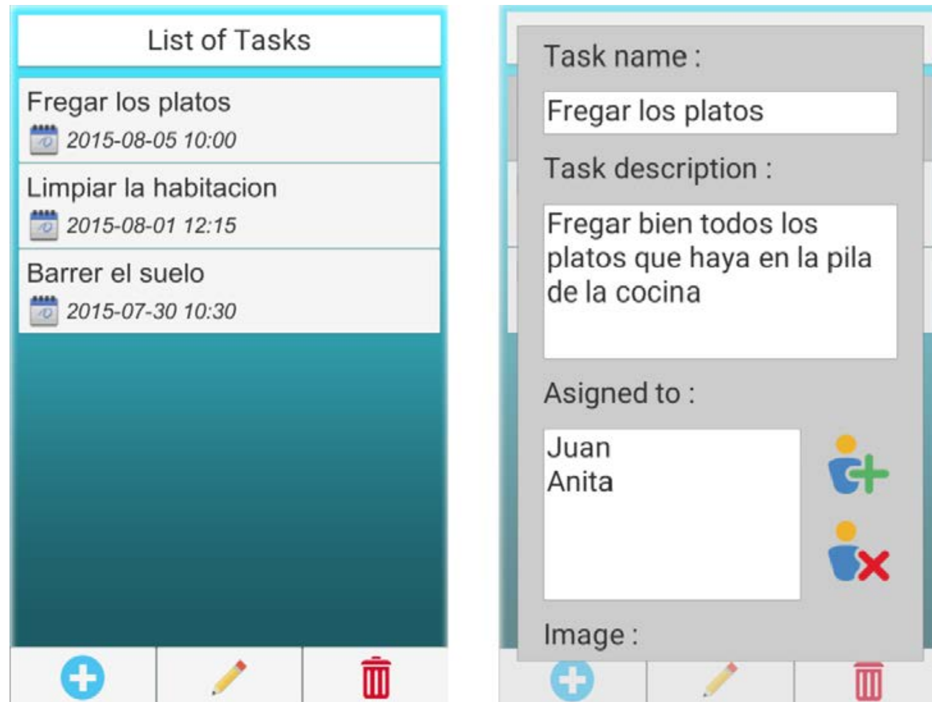

Ilustración 16 - Escena TaskMan

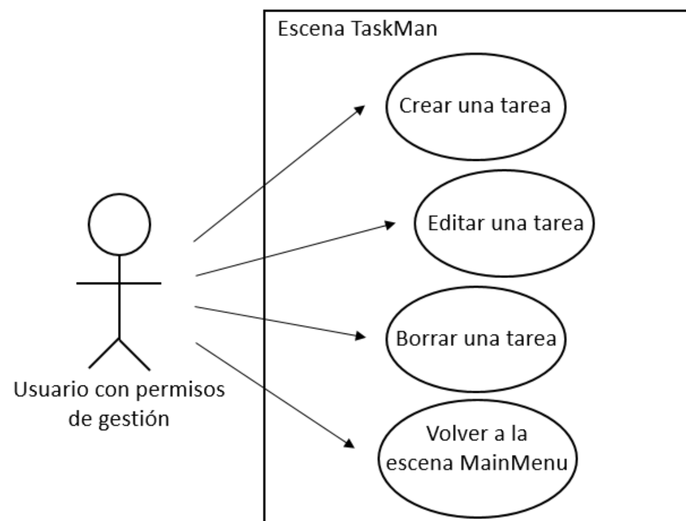

Ilustración 17 - Casos de uso de TaskMan

- TaskToDo: aquí es donde aparecerán las tareas que le han sido asignadas al usuario y podrá ver las que tiene hechas y las que tiene pendientes, así como chequear una tarea cuando haya sido completada. Además también se puede acceder a una descripción detallada de la tarea pulsando sobre ella.

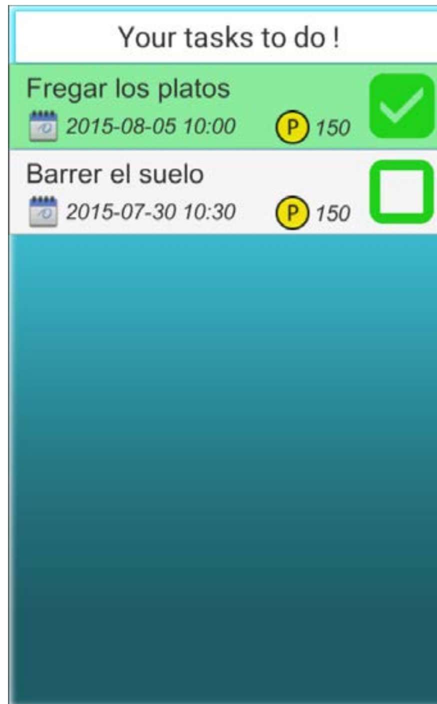

Ilustración 18 - Escena TaskToDo

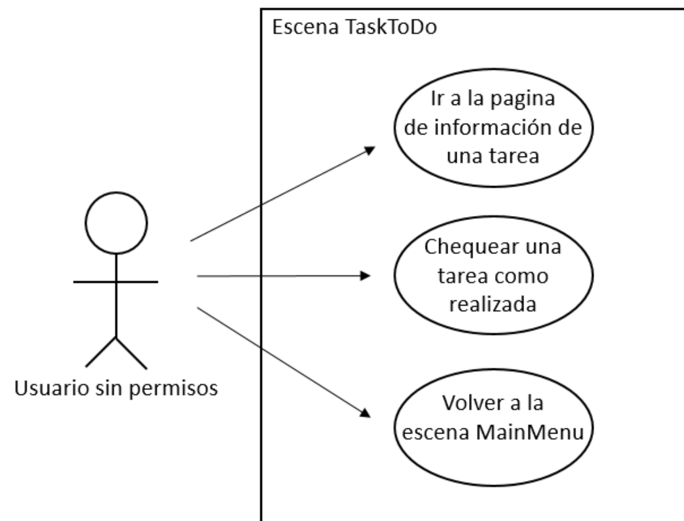

Ilustración 19 - Casos de uso de TaskToDo

- TaskInfo: escena en la que se presenta la información detallada de una tarea. Se utiliza por las escenas TaskToDo y TaskToVerify para mostrar al usuario todos los datos de una tarea en concreto.

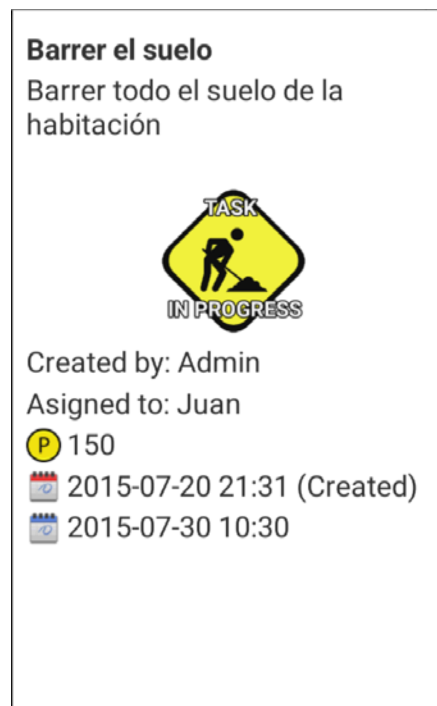

Ilustración 20 - Escena TaskInfo

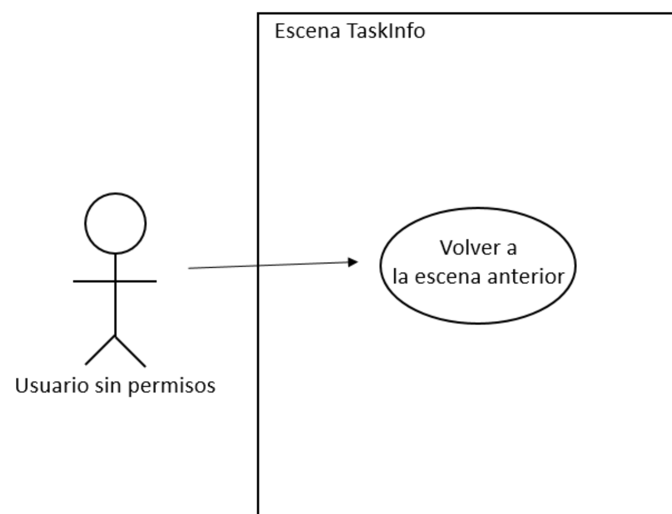

Ilustración 21 - Casos de uso de TaskInfo

- TaskToVerify: Muy similar a TaskToDo pero en esta escena entraran los usuarios con permisos de verificación tras haber visto que una tarea ha sido realizada para verificarla y añadirle una puntuación según como se haya hecho.

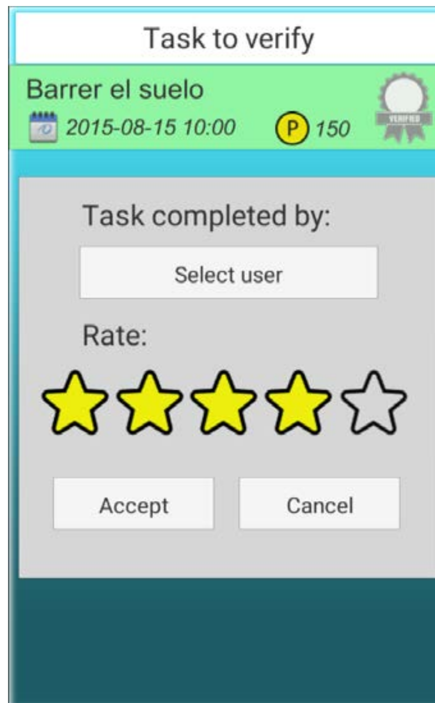

Ilustración 22 - Escena TaskToVerify

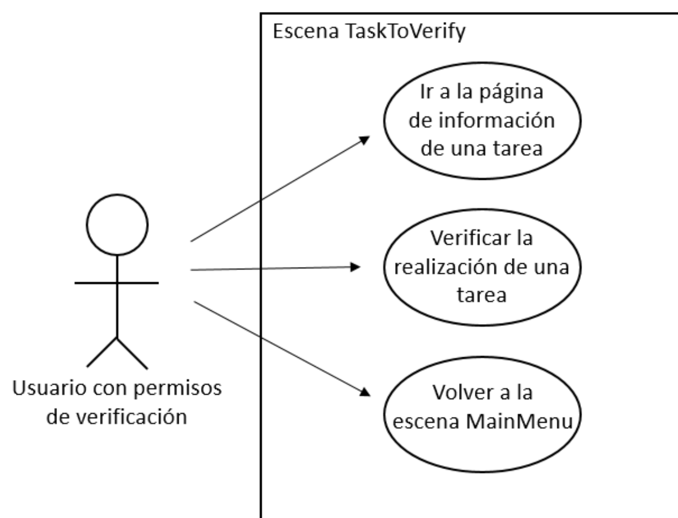

Ilustración 23 - Casos de uso de TaskToVerify

- Leaderboards: con esta escena mostramos una lista de todos los usuarios ordenada por los puntos con datos de información como los puntos de cada uno y las tareas completadas por cada uno, además de una imagen que deja ver claramente quien va primero, segundo o tercero.

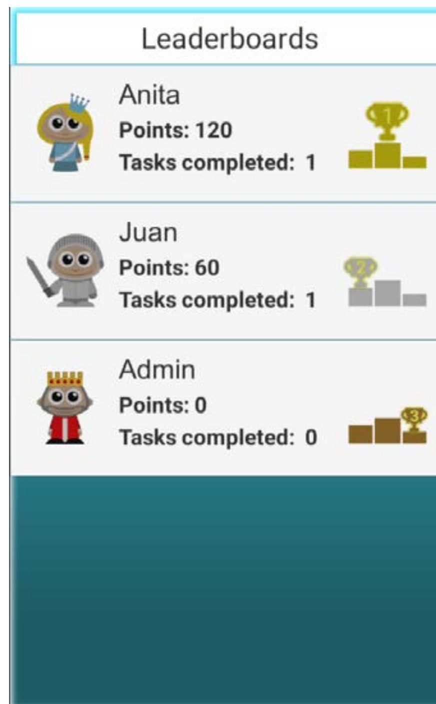

Ilustración 24 - Escena Leaderboards

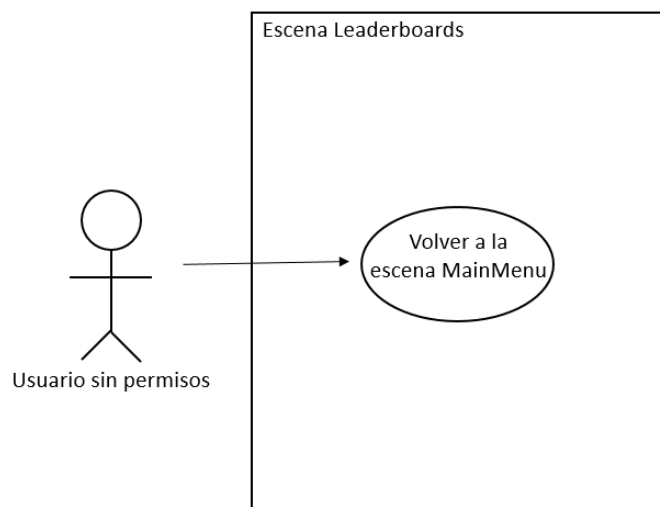

Ilustración 25 - Casos de uso de Leaderboards

- RewardsMan: abreviatura de *Rewards management*, escena similar a UserMan o TaskMan en la que el usuario podrá gestionar las recompensas, pudiendo añadir nuevas o editar las existentes.

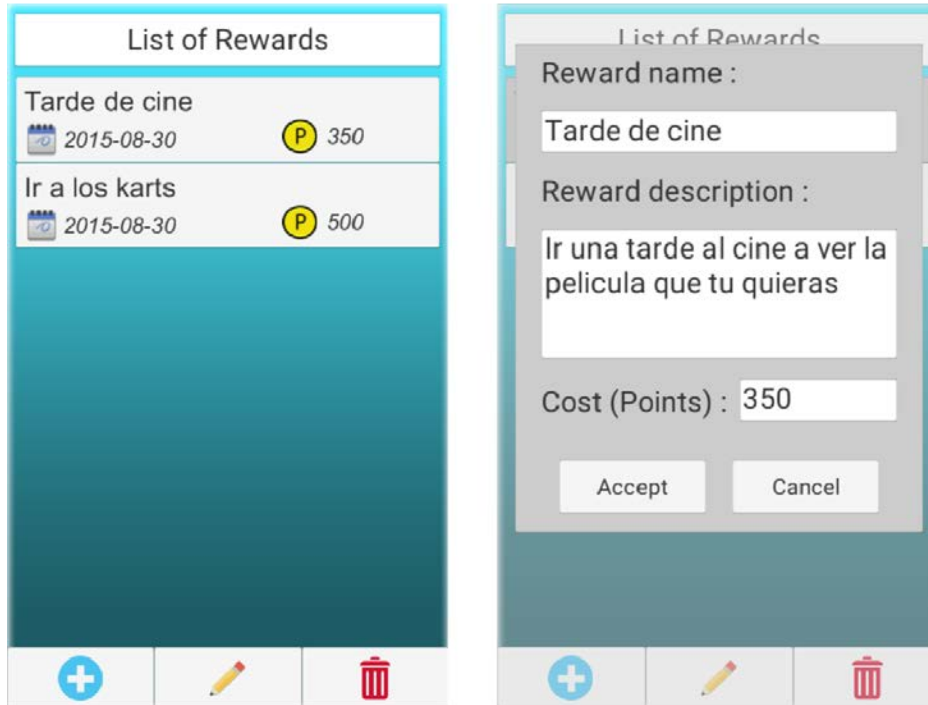

Ilustración 26 - Escena RewardsMan

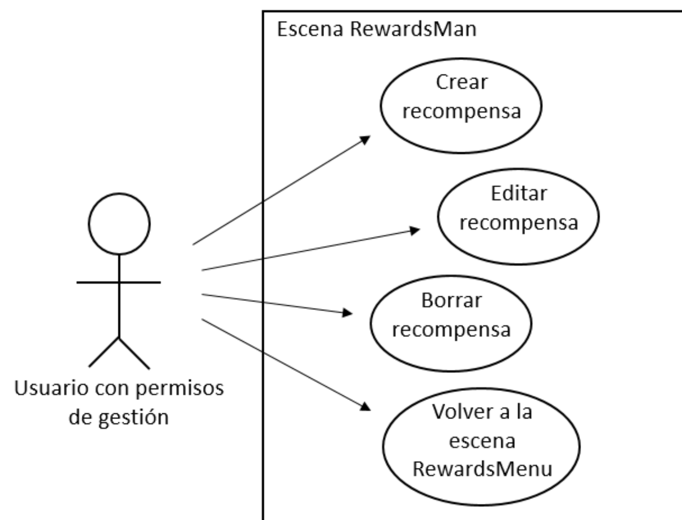

Ilustración 27 – Casos de uso de RewardsMan

- RewardsMenu: el menú al que se accede desde el botón *Rewards* del menú principal. En este dependiendo del nivel de permisos del usuario se podrá ver o no el botón de acceso a la gestión de recompensas. Y luego tenemos los botones de recompensas disponibles y de las recompensas canjeadas por el usuario.

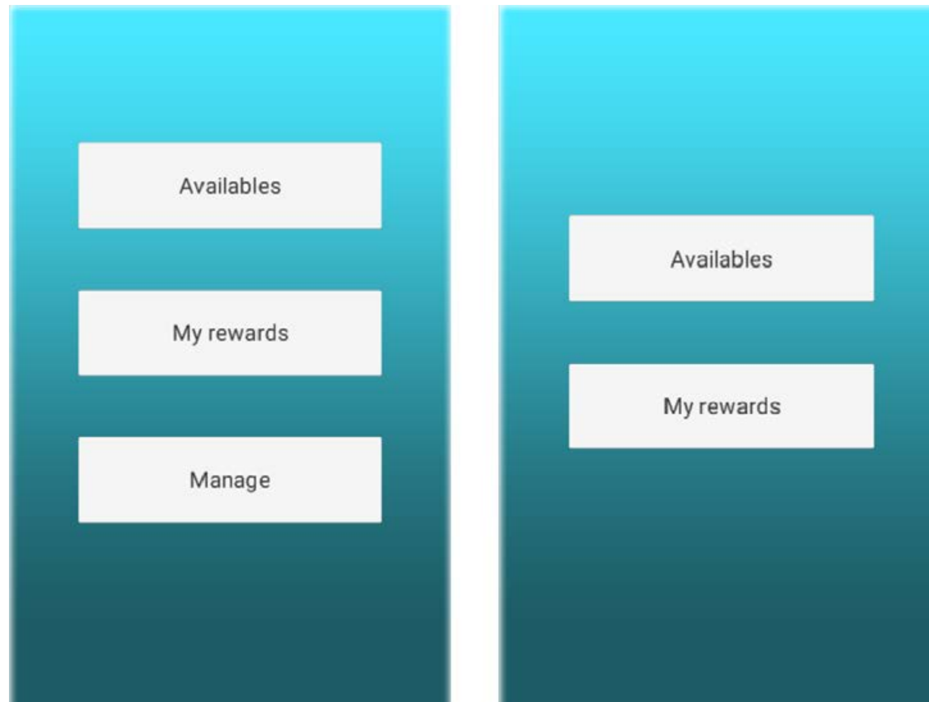

Ilustración 28 - Escena RewardsMenu

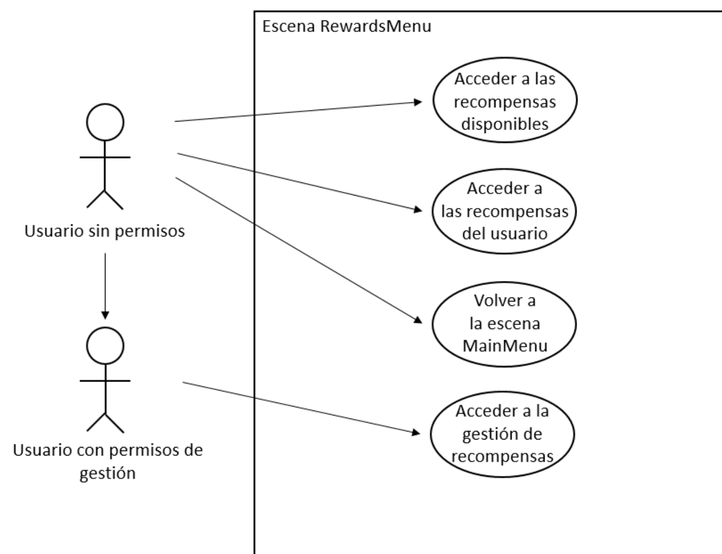

Ilustración 29 - Casos de uso de RewardsMenu

- RewardsAvailable: en esta escena aparecerá una lista con todas las recompensas, y dependiendo de la puntuación del usuario, las que no pueda canjear por falta de puntos aparecerán en otro color y no se podrá interactuar con ellas. Aquí el usuario puede hacer clic en una recompensa y ver una descripción y canjearla.

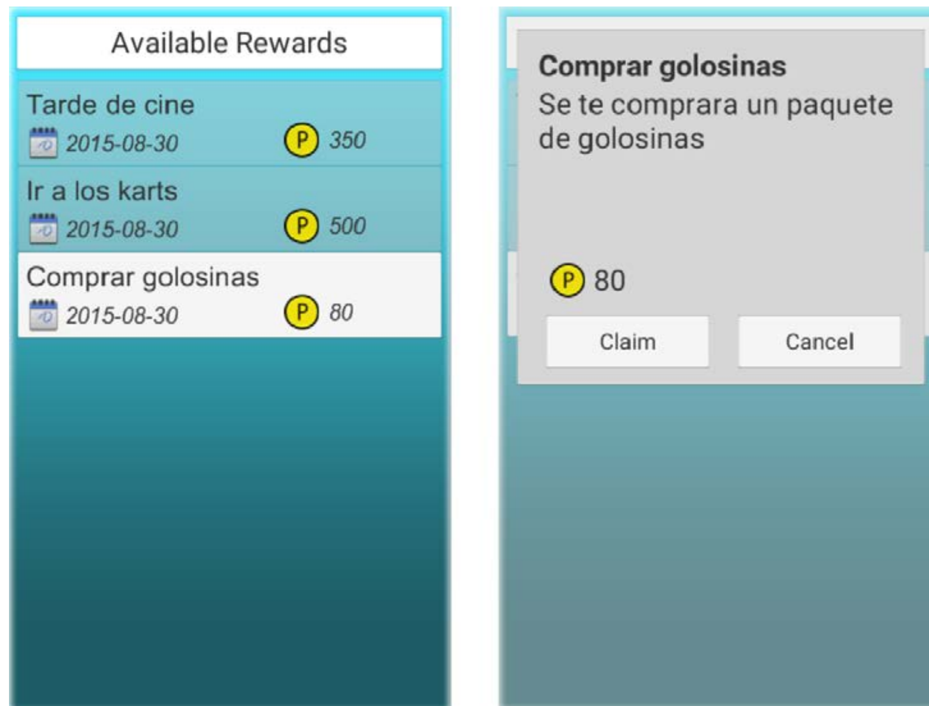

Ilustración 30 - Escena RewardsAvailable

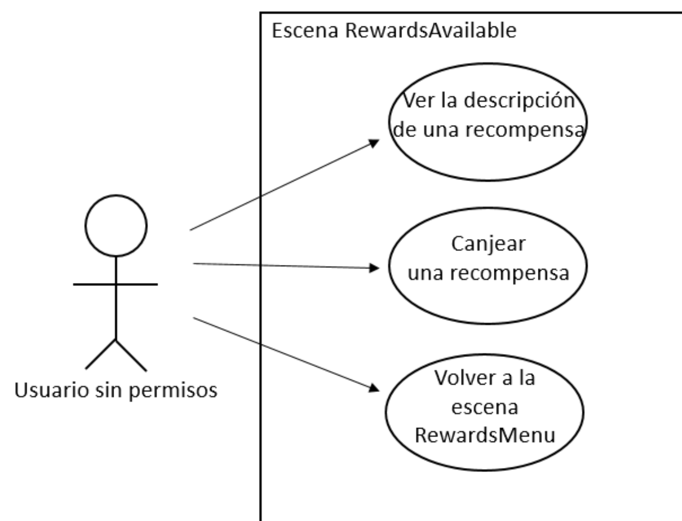

Ilustración 31 - Casos de uso de RewardsAvailable

- MyRewards: aquí el usuario dispondrá de una lista de todas las recompensas que haya canjeado con datos como el coste de la recompensa o la fecha en la que se canjeo esta recompensa.

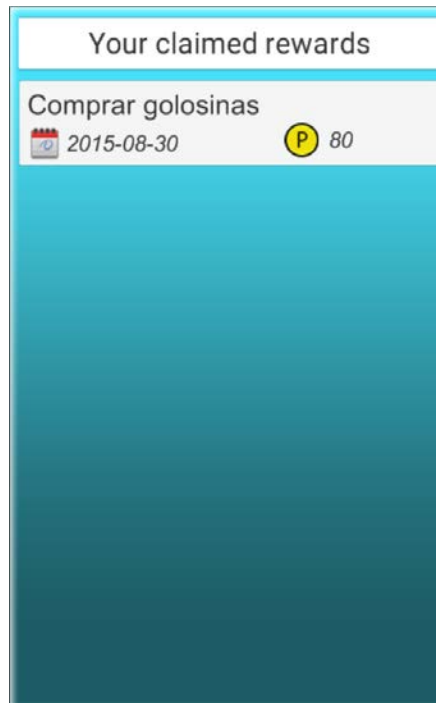

Ilustración 32 - Escena MyRewards

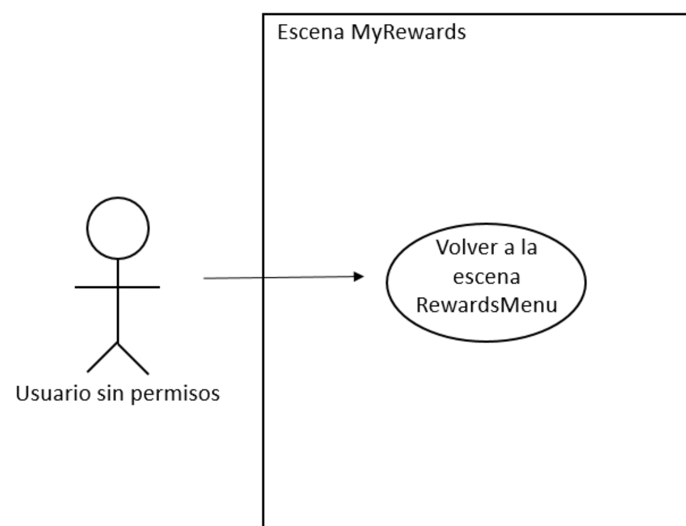

Ilustración 33 - Casos de uso de MyRewards

- Diagrama de flujo de la aplicación: en el diagrama se pueden apreciar las escenas con el mismo nombre que tienen en la aplicación, y las transiciones mediantes acceso por botones con el nombre original de estos.

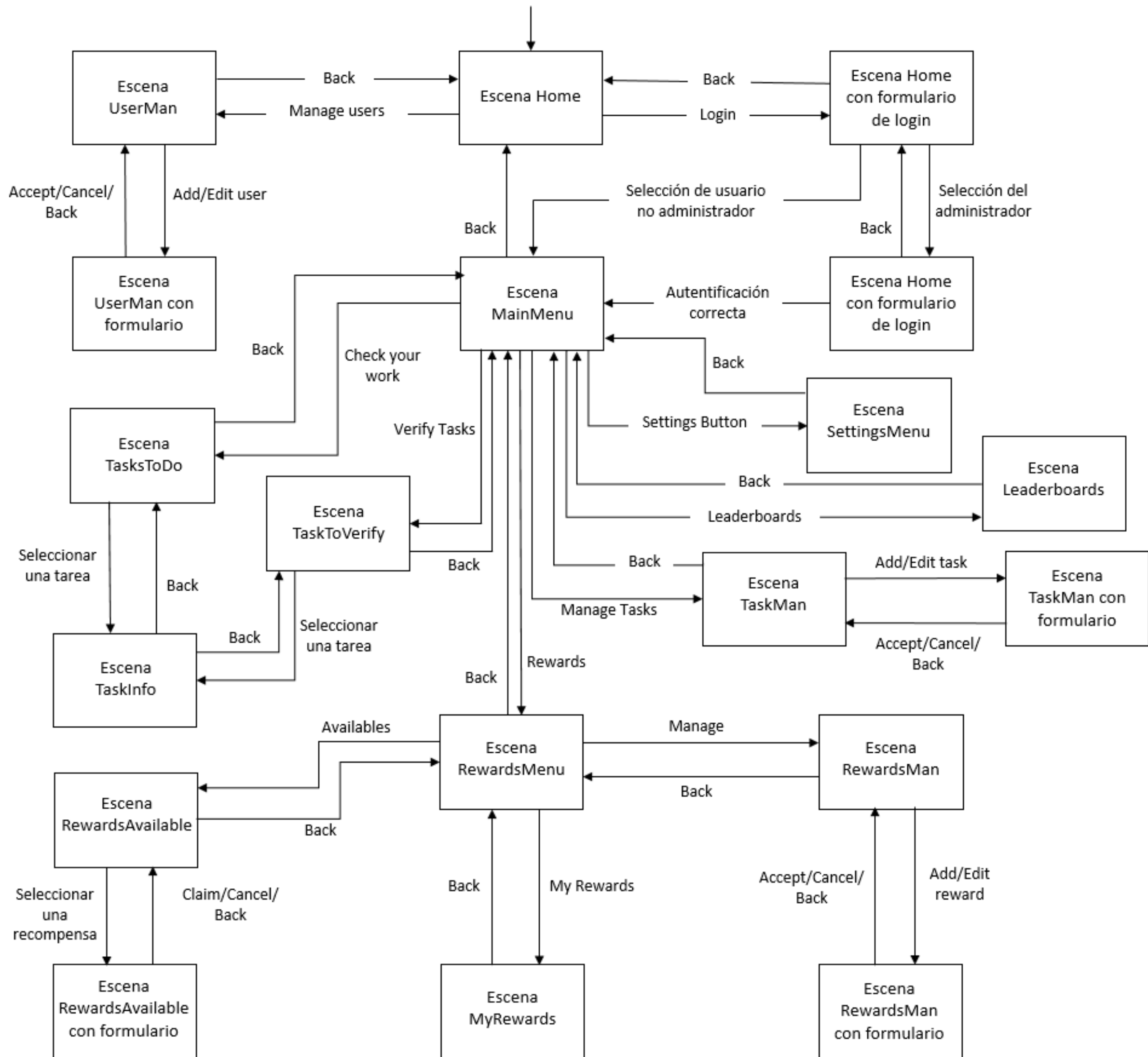

Ilustración 34 - Diagrama de flujo de la aplicación

4.3. Objetos comunes

Aquí se van a enumerar algunos de los objetos de Unity más comunes utilizados en el proyecto y una explicación de su uso y el contexto en el que son utilizados.

- GameManager: creado a partir de un objeto vacío y que dispone de un solo componente, el *script GameManager.cs*. Este objeto aparece en la escena inicial de la aplicación y no es destruido en ningún cambio de escena. Se encarga de

gestionar toda la información de la aplicación (listas de usuarios, tareas y recompensas), de guardarla en un archivo codificado en binario y de cargarlo al iniciar la aplicación. También se encarga de interactuar con el archivo persistente de *PlayerPrefs*, y contiene métodos de uso general en la aplicación (obtención de un usuario por nombre, de una tarea por id único, etc.).

- ScreenManager: este objeto también es creado a partir de un objeto vacio y un *script*. En este caso se usa el *script* con el mismo nombre *ScreenManager.cs* y puesto que este objeto está presente en todas las escenas (no es persistente, sino que en todas las escenas existe uno) gestiona eventos como el cambio de escena o las acciones derivadas de pulsaciones.

- ScrollController: objeto creado a partir de un objeto vacío al que se le añade un compenente *script* que variara dependiendo de la escena en la que nos encontremos. Este objeto está presente en todas las escenas en las que se cree alguna lista de objetos. De modo que el *script* que gestione la lista deberá de ir añadido a este objeto. En este proyecto existen *scripts* que son añadidos al ScrollController como *ScrollList* o *PictureScrollList* que son *scripts* que se definirán más adelante en el punto correspondiente a la implementación.

- Main Camera: objeto de tipo *Camera* que está presente en todas las escenas y que define el área que se verá en la aplicación. Si el proyecto fuera un juego o algo relacionado con el 3D se trabajaría más con este objeto pero para el desarrollo de una aplicación en 2D no se interactúa con este objeto.

- CommonCanvas: objeto del tipo *Canvas* que es un objeto de la interfaz que está estrechamente relacionado con la cámara. Este objeto engloba todos los elementos de la interfaz que van a ser representados, por lo que todos los objetos de la escena que aparecerán como puedan ser botones, etiquetas, etc. Deben ser hijos de este objeto. Los objetos que no aparecen sino que son objetos que gestionan variables como son el *GameManager* y el *ScreenManager* no están dentro del *Canvas*, puesto que no deben ser representados.

- EventSystem: este objeto se crea automáticamente en la escena y es el encargado de gestionar las entradas táctiles y los eventos asociados a estas.

Dispone de muchas funcionalidades pero puesto que en este proyecto no se hace uso no serán desarrolladas en este apartado.

5. Implementación

Para organizar de una manera más coherente la presentación de la implementación de la aplicación, estarán organizadas todas las clases en 4 categorías: general, formularios, listas y botones. Esto es debido a que las categorías de formularios, listas y botones contienen clases que tienen una estructura similar aunque cada una tiene una función y unos métodos. Y para aclarar el diseño de los mapas de código se puede ver la leyenda a continuación (véase *Ilustración 35*).

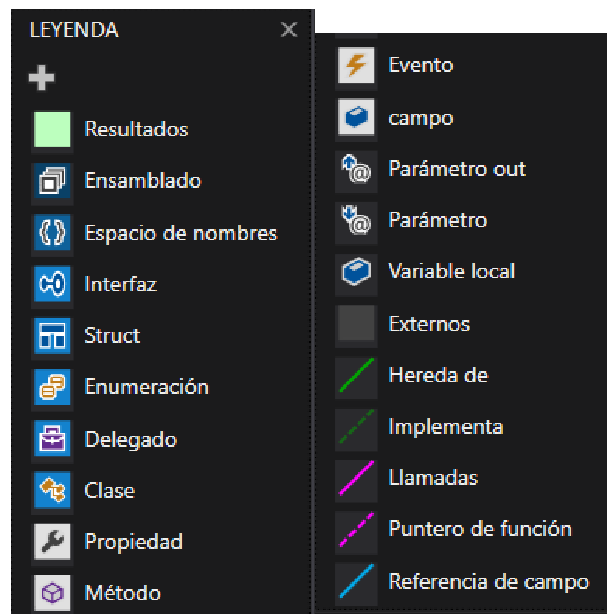

Ilustración 35 - Leyenda de los mapas de código

5.1. General

- User.cs: la clase con la que gestionamos la información de los usuarios. Consta de variables para el nombre (*string*), contraseña (*string*), rol de usuario (*string*), nivel de privilegios (*int*), icono (*string*), puntos acumulados (*int*), tareas chequeadas (lista de *int*), tareas completadas (lista de *int*) y recompensas adquiridas (lista de *reward*).

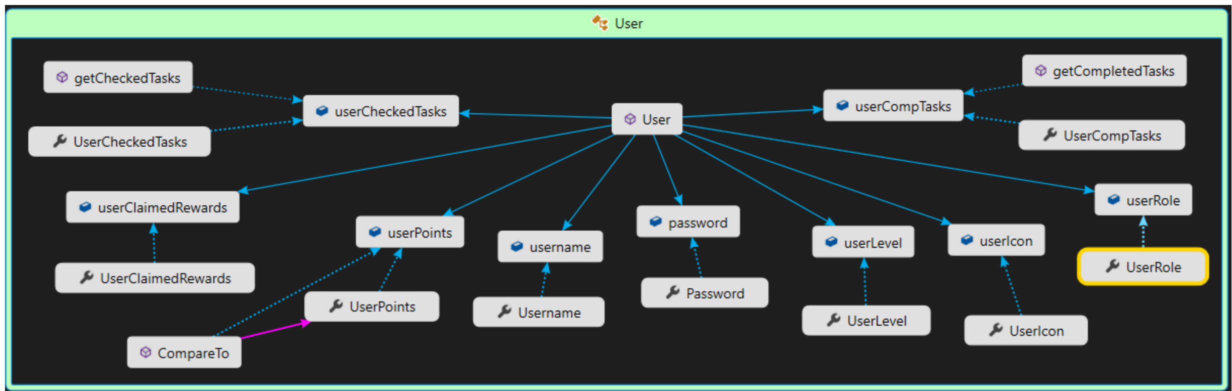

Ilustración 36 - Mapa de código de User.cs

- Task.cs: mediante esta clase gestionaremos las tareas y toda la información relacionada con estas: id único (*int*), nombre (*string*), descripción (*string*), creador (*string*), usuarios asignados (*string*), imagen (*string*), si esta completada (*bool*), si esta chequeada (*bool*), los puntos que otorga (*int*), la fecha de creación (*DateTime*) y la fecha tope para completarla (*DateTime*). Cuando se crea una tarea hay campos que son asignados por el propio sistema, como si esta completada, que se pone en *false* o el creador que se pone analizando la variable que guarda el usuario activo en el momento de la creación (variable almacenada en el fichero *PlayerPrefs*) y también la fecha de creación que se asigna utilizando la del sistema.

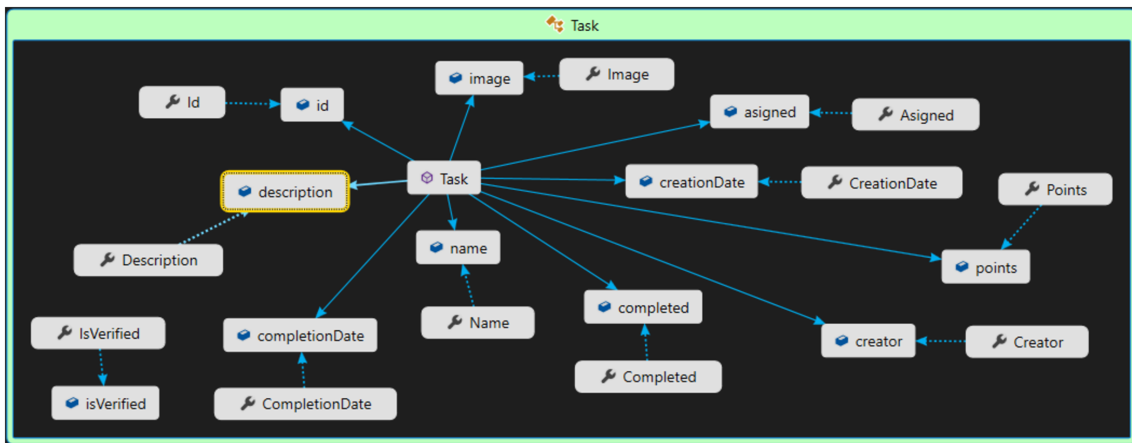

Ilustración 37 - Mapa de código de Task.cs

- Reward.cs: clase mediante la cual se definen las tareas con su constructor y todas sus variables: id único (*int*), nombre (*string*), descripción (*string*), coste (*int*), fecha de creación (*DateTime*) y fecha en la que se canjeo (*DateTime*) (este dato

lo utilizamos a la hora de listar las tareas canjeadas por cada usuario). Al igual que en la clase anterior, la fecha de creación la pone el sistema en el constructor.

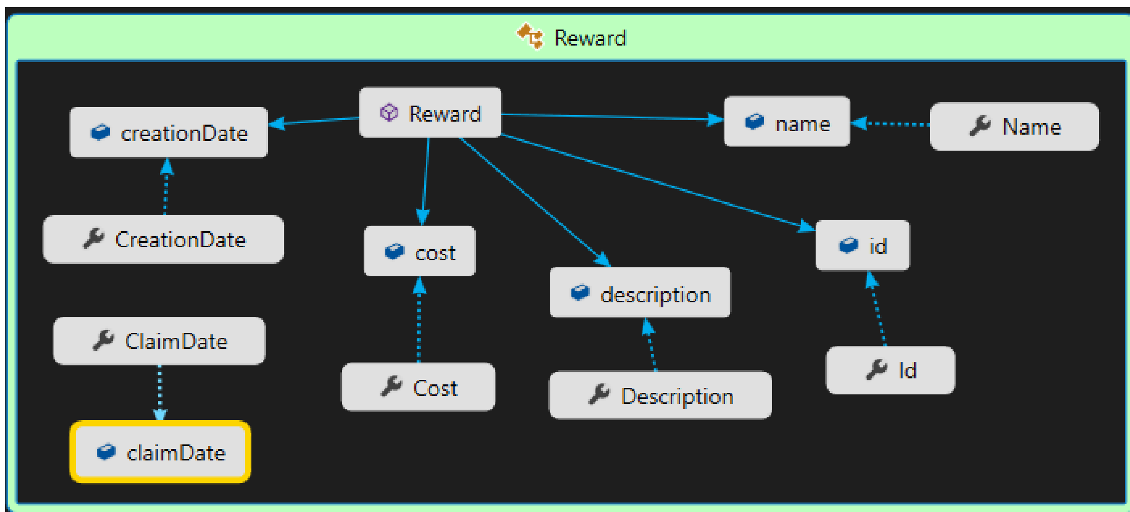

Ilustración 38 - Mapa de código de Reward.cs

- GameManager.cs: esta clase se añade a un objeto que no es destruido con el cambio de escenas y gestiona las bases de datos de usuarios y tareas de la aplicación. Con métodos como obtener usuario, obtener tarea, etc. Estos métodos se apoyan de las listas de usuarios, tareas y recompensas que se obtienen del archivo de datos que es cargado cada vez que se inicia la aplicación y que se guarda cada vez que se cierra esta.

  Dentro de esta clase también se define la clase *GameData*, marcada como *Serializable* para poder guardar una instancia de esta en el archivo codificado en binario que contiene los datos de la aplicación. Esta clase por lo tanto contiene los datos de la aplicación que serán guardados: lista de usuarios, de tareas y de recompensas. Los otros datos de la aplicación como puedan ser el último usuario autentificado o demás variables que se pueden guardar en una cadena de texto son almacenadas en el archivo *PlayerPrefs*.

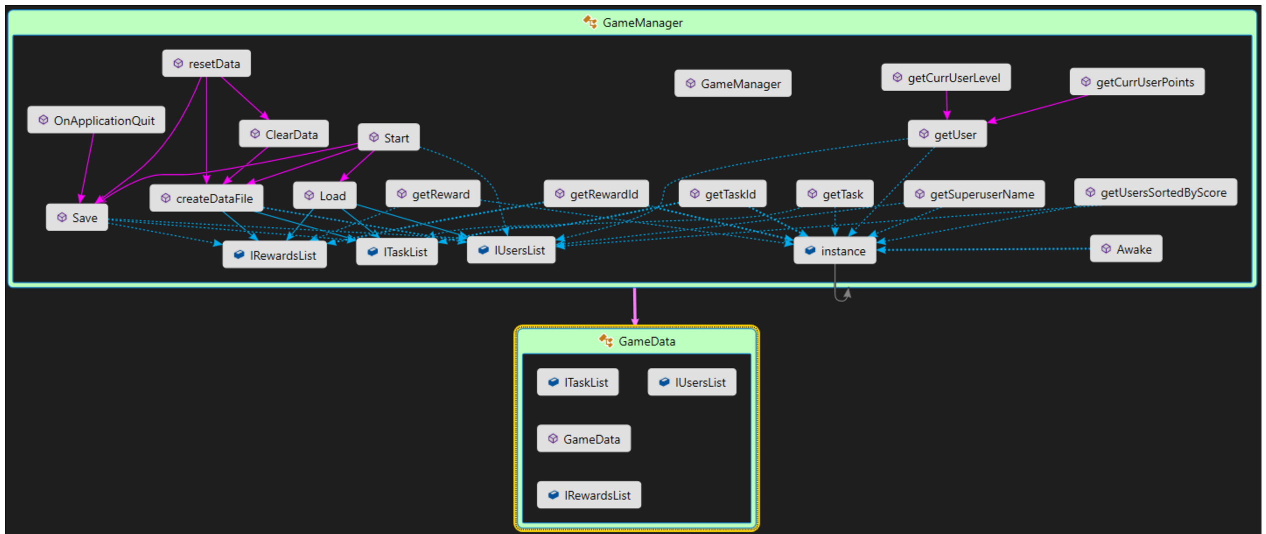

Ilustración 39 - Mapa de código de GameManager.cs

- ScreenManager.cs: clase que se le añade a un objeto del mismo nombre que se encuentra presente en todas las escenas y que gestiona los cambios de escena y la entrada de teclas como *back*[23], que dependiendo de la escena te puede llevar a la anterior o cerrar un formulario, etc.

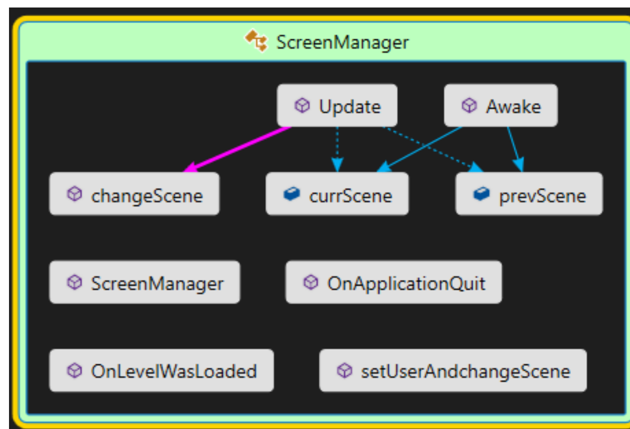

Ilustración 40 - Mapa de código de ScreenManager.cs

- LoginManager.cs: esta es una clase muy simple en la que solo hay un método llamado *checkInfo* que comprueba que la información de acceso (usuario y contraseña) es correcta. Se usa en la escena de Login.

[23] Tecla de uso común en el sistema operativo Android.

- GlobalConst.cs: aquí están almacenadas variables que se usan a nivel global. De ese modo en caso de necesitar hacer algún cambio en una de las variables solo haría falta cambiarlas en esta clase y no ir cambiándolas en todas las apariciones. Estas son:
  - Rutas de almacenamiento de la aplicación (*String*): que se utilizan cuando hay que cargar recursos o archivos en localizaciones que son estáticas.
  - Campos que se usan en el archivo persistente *PlayerPrefs* (*String*): como son *currentUser* o *currScene* ya que el valor de la variable a la que estos campos hacen referencia cambia pero el nombre de los campos no
  - Los números que se refieren a las escenas de la aplicación (*int*): ya que Unity guarda las escenas por orden y les asigna un número, y este número es que ha de añadirse como argumento al método *changeScene* de *ScreenManager* para poder cambiar de escena.
  - Los formatos de fecha utilizados (*String*): ya que de ese modo siempre que un *DateTime* se convierta en cadena de texto, se usara el mismo patrón de conversión y no habrá diversidad de formatos. Esto también da lugar a la posibilidad de si fuera necesario establecer diferentes formatos de fechas para diferentes zonas, ya que por ejemplo el formato de fecha que se usa en España es diferente al que se usa en Estados Unidos.
  - Colores personalizados (*Color32*): Esto es debido a que la clase *Color* viene con unos colores por defecto pero si quieres un color personalizado has de crearlo, y de este modo se asegura el uso del mismo color en toda la aplicación.

5.2. Formularios

El uso de formularios ha estado presente en la aplicación a la hora de gestionar usuarios, tareas y recompensas. Es un modo fácil y simple de poder crear o editar algunos de estos objetos que hasta los niños pueden usar. Los formularios generalmente comparten algunas características: la comprobación de campos al aceptar el formulario y un método que tras comprobar que todos los campos sean correctos actualiza la lista que invoco el formulario y cierra esta para ver los resultados. Y en todos se han creado métodos que se han relacionado con los eventos de clic de los botones correspondientes del formulario.

- RewardForm.cs: formulario con el que podemos crear o editar una recompensa, asignándole un nombre, una descripción y un coste en puntos. El sistema le asignara luego la fecha de creación al objeto *Reward* generado.

- RewardAvailableForm.cs: con este podemos ver una pequeña descripción de la recompensa y si al usuario le interesa puede canjearla por el valor en puntos especificado.

- TaskInformation.cs: esta clase a pesar de ser como un formulario es usada en una escena a parte para mayor comodidad del usuario. En esta se ofrece una completa descripción de la tarea obteniendo la información de esta de la base de datos.

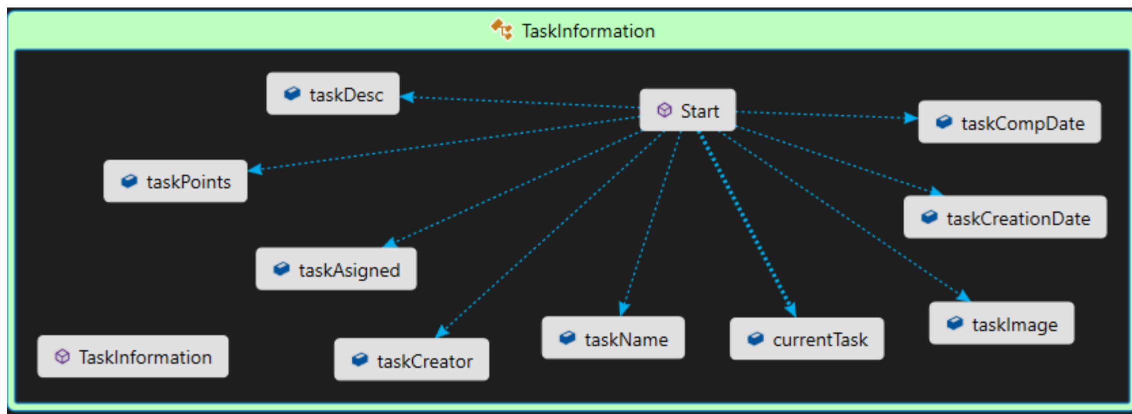

Ilustración 41 - Mapa de código de TaskInformation.cs

- TaskForm.cs: encargado de crear/editar las tareas, ponerles nombre, descripción, usuarios a los que se les asigna la tarea, imagen, puntos de recompensa por completarla y fecha límite para completarla.

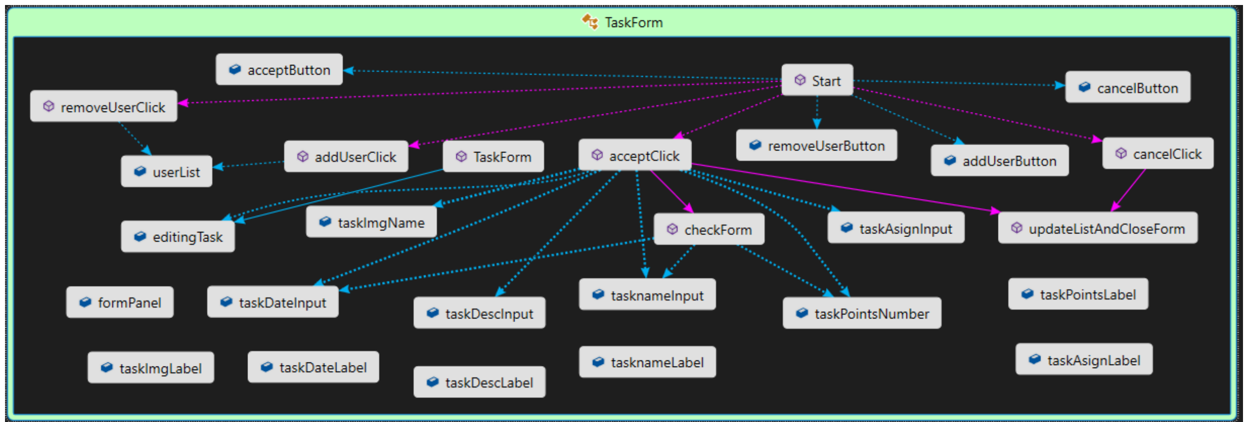

Ilustración 42 - Mapa de código de TaskForm.cs

- TaskVerifyPanel.cs: formulario simple para verificar que alguien ha completado una tarea seleccionando el usuario que la ha completado y dándole una puntuación en estrellas (de 1 a 5 estrellas) depende de lo bien que lo haya hecho. Dependiendo de esta puntuación se le asignaran puntos. Cada estrella son 0,2 más a multiplicar por la puntuación de la tarea, siendo la puntuación completa de la tarea en caso de ser realizada perfectamente.

- UserForm.cs: mediante este formulario se gestionan los usuarios y sus campos: nombre, contraseña, rol en la familia, imagen de usuario y permisos de usuario (permisos de verificación y de gestión de tareas/recompensas).

5.3. Listas

Las clases referidas a listas tienen unas funciones generales: generar una lista de objetos a partir de una base de datos (que puede ser la lista de usuarios, tareas, etc.) y dotar de funcionalidad a esta para poder seleccionar uno de ellos y editarlo o borrarlos. Llamando de este modo a los formularios. Y es la clase que hace de puente con los formularios a la hora de crear o editar un registro en la base de datos, ya que los propios formularios no son los que acceden a las listas de usuarios, tareas o recompensas. También disponen de un control para ver si está activo o no el formulario, para que si el usuario presionara el botón *Atrás* se cerrara el formulario o se volviera a la anterior escena dependiendo de esto. Para no crear clases muy similares, hay varias de las clases de listas que se usan en diferentes escenas pero que disponen de un valor booleano que determina en qué modo funciona. En el caso de las listas todas tienen una o dos variables que se establecen en el editor de Unity, el botón u objeto a instanciar

en la lista, que se le asigna con un *prefab*. Y en algunos casos el formulario a instanciar en caso de necesitar añadir o editar algo desde un formulario. Este último también se le añade como un *prefab*.

- PictureScrollList.cs: esta clase se encarga de crear la lista de usuarios con la imagen correspondiente que se usa para acceder con un usuario a la aplicación. También se usar para crear la lista de imágenes de la galería preestablecida de imágenes de usuario.

- RewardList.cs: mediante el uso de esta clase generamos 3 tipos de listas: la lista de recompensas para el apartado de gestión de recompensas, en la cual se podrán editar y se podrá invocar al formulario que realiza dicha tarea. Otro tipo de lista es la de las recompensas disponibles, en la que dependiendo de los puntos del usuario que este manejando la aplicación se deshabilitaran o no los botones de las recompensas. Y el último tipo de lista sería el de las recompensas asociadas al usuario, que serían todas las recompensas que el usuario ha canjeado en algún momento. Para el funcionamiento en un modo o en otro, la clase dispone de dos variables booleanas que marcan el funcionamiento en los métodos de la clase.

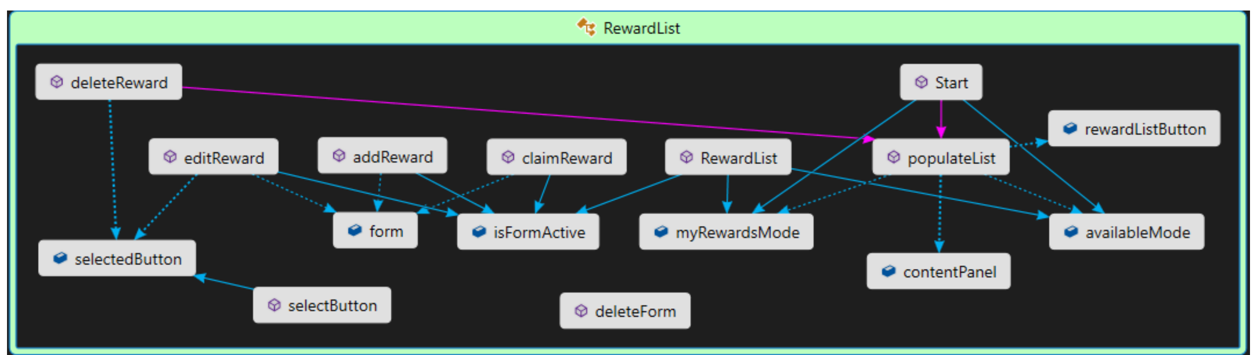

Ilustración 43 - Mapa de código de RewardList.cs

- ScrollList.cs: clase que se encarga de llenar la lista de usuarios en la escena *UserMan* para poder gestionarlos, instanciando el formulario de gestión de usuarios así como los botones diseñados para esta lista.

- TaskList.cs: lista de tareas, con una estructura muy similar a la anterior pero desarrollada para crear o editar los objetos *Task* y sus campos correspondientes.

Dispone también de un botón y un formulario de edición preestablecidos desde el editor de Unity y que serán instanciado mediante código.

- TaskCheckList.cs: lista de tareas en las que aparecerán las tareas asignadas al usuario y se podrán chequear en caso de haberlas completado. También se podrá ver una descripción completa haciendo clic a las tareas.

- TaskVerifyList.cs: similar a las dos clases anteriores pero separada igual que la anterior para dejar un código más limpio y robusto. Con esta clase se genera la lista de tareas que se han chequeado por los usuarios como hechas pero que no se han verificado. Aquí pueden acceder los usuarios con permisos de verificación y puntuar las tareas dependiendo de lo bien o mal hechas que estén.

- UserList.cs: esta clase se usa para crear una lista de nombre de usuario que se usa en dos escenas diferentes. Primero, a la hora de crear una tarea, al pulsar el botón de asignar un usuario, se crea la lista con todos los usuarios para asignarles la tarea creada o editada. Y segundo en el formulario de verificación de tareas para asignar a que usuario le estas verificando la tarea, para que en caso de haber varios no haya confusión.

- UserListByScore.cs: clase usada para generar la lista de usuarios con sus puntuaciones y posiciones en la escena *Leaderboards*. Utiliza la función del *GameManager* que genera una lista con todos los usuarios ordenados por puntuación y luego instancia el botón asignado desde el editor de Unity para creas dicha lista.

5.4. Botones

Como en Unity se trabaja con *scripts* asignados a los objetos, para gestionar la información de los botones que forman las listas generadas automáticamente por las clases de la sección anterior se utilizan *scripts* que van asignados a cada botón, y que gestionan todos los componentes de dichos botones. Además, como los botones tienen que tener una acción asignada al evento de pulsarlo, se debe hacer eso por código, puesto que el objeto al que llama no existe hasta que no está instanciado y de ese modo no se puede asignar desde el editor de Unity. De ese modo los scripts referentes a

botones suelen tener en común el hecho de relacionar en el método *Start* la acción de clic del botón con un método que llamara al script de la lista que gestiona los botones y de ese modo se aplicara la acción correspondiente, bien sea dejar el botón como seleccionado o abrir un formulario o cambiar de escena.

- ListButton.cs: clase encargada de gestionar los botones de las listas de usuarios creadas por la clase *UserList,cs*. Es un botón simple con el nombre del usuario centrado.

- RewardListButton.cs: gestióna el botón que representa las recompensas en las 3 escenas de recompensas (*RewardsMan*, *RewardsAvailable* y *MyRewards*) dependiendo de la escena le añade un tipo u otro de información al botón. En este caso se usa la misma clase puesto que el botón al que se le atañe es el mismo, solo cambia la información que se le pone.

- ScrollListButton.cs: esta clase gestiona los datos de los botones generados en la lista de la escena de gestión de usuarios. Este es uno de los botones que gestiona también el cambio de color al ser seleccionado y la llamada a la clase encargada de gestionar la lista para marcarlo como botón seleccionado. Estas últimas características son generales a todos los botones de listas de gestión (tanto de usuarios como de tareas o recompensas).

- TaskListButton.cs: clase similar a la anterior pero trabajando con tareas en lugar de usuarios. Dispone de las variables referidas al nombre, a la fecha, y también al id único a pesar de no ser mostrado, ya que se usa para luego trabajar con la tarea a la que se refiere cada botón. También dispone de las características generales de los botones pertenecientes a listas de gestión.

- TaskListButtonCheck.cs: este *script* es asignado a los botones de la lista de tareas pendientes de cada usuario y dispone de un método para cuando es pulsado cambiar a la escena de *TaskInfo*, y otro método para cuando se marca como realizada la tarea.

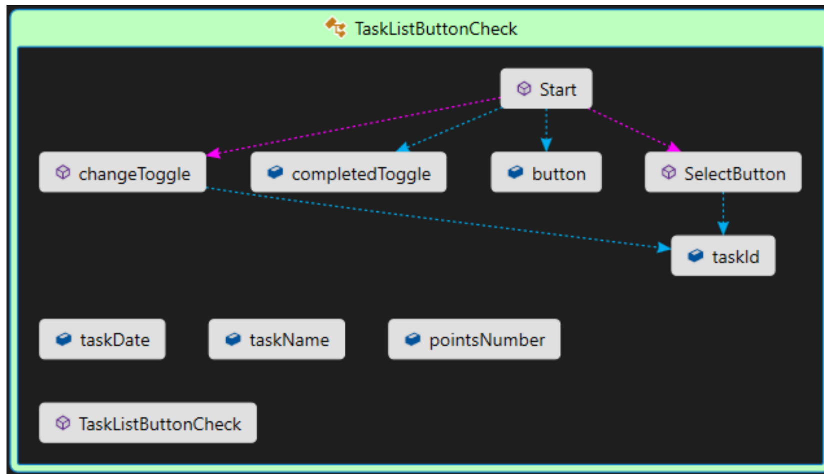

Ilustración 44 - Mapa de código de TaskListButtonCheck.cs

- TaskListButtonVerify.cs: clase similar a las dos anteriores, que gestiona la información de los botones de la lista de tareas por verificar. Estas últimas tres clases están separadas debido a que las funciones que hacían eran totalmente diferentes y usar la misma clase para los 3 botones acabaría con un código “sucio” y poco robusto.

- UserPicButton.cs: con este *script* se gestionan los botones que representan a los usuarios en el formulario de acceso. Este botón se compone de la imagen del usuario y de su nombre justo debajo. De modo que al seleccionar uno se cambie al menú principal o a la pantalla de acceso por usuario y contraseña dependiendo del usuario seleccionado.

- UserPicButton2.cs: clase que se aplica a los botones que representan las imágenes en la selección de imágenes del formulario de creación/edición de usuarios. Se genera automáticamente con las imágenes encontradas en la carpeta de imágenes de usuario preestablecida. Y al pulsar sobre esta se modificaría la imagen del formulario de usuario y se cerraría el formulario con la lista de imágenes.

  UserScoreButton.cs: este *script* es el que se usa en cada uno de los botones que forman los marcadores en la escena *Leaderboards* y que gestiona la información representada de cada usuario. Puesto que el marcador es puramente informativo no dispone de una acción relacionada con el evento de clicar el botón.

6. Conclusiones

6.1. *Análisis de los resultados*

El resultado ha sido bastante satisfactorio, habiendo desarrollado una aplicación funcional y habiendo interiorizado y aplicado el concepto de juguetización a un paradigma tan común como es la gestión de tareas del hogar. Personalmente he aprendido mucho trabajando en este proyecto, mis conocimientos sobre Unity han aumentado muchísimo y ahora entiendo mucho mejor el funcionamiento interno de un videojuego, algo que seguro me ayudara en mi futuro. Sobre el lenguaje de programación C# también he obtenido buenos resultados, puesto que aun conociéndolo y habiendo trabajado múltiples veces con él, el desarrollo de un proyecto desde cero con este lenguaje me ha ayudado a conocerlo mejor.

Además, he podido adquirir experiencia en la gestión de proyectos que siempre es bueno de cara al mundo laboral. El hecho de gestionarme yo solo todas las tareas que han conformado el proyecto y organizarme, me ha enriquecido mucho como futuro ingeniero.

6.2. *Relación con los estudios cursados*

La realización del proyecto se ha visto ayudada en múltiples ocasiones por conocimientos adquiridos en diversas asignaturas de la carrera. En la parte del diseño de la aplicación en Unity, el trabajo diseñando interfaces en la asignatura de segundo *Interfaces persona computador* ha sido de gran ayuda ya que fue la primera toma de contacto con el diseño de interfaces y sin ella el desarrollo habría sido más costoso.

A la hora de realizar la implementación de la aplicación en código, el diseño de las clases y las estructuras de datos, se ha visto la importancia de asignaturas como *Programación*, *Algorítmica* o *Estructuras de datos y algoritmos* en las que se aprende a codificar de una manera ordenada, limpia y eficiente. Y a establecer unas estructuras de datos que sean funcionales a la par que lógicas y fáciles de interpretar.

Englobando todo el desarrollo del proyecto han tenido importancia conocimientos obtenidos en *Gestión de proyectos* e *Ingeniería del Software* para la correcta

estructuración del trabajo así como el desarrollo de unos requisitos para la aplicación y los diagramas de casos de uso de cada una de las escenas.

Además de todo lo aprendido en la carrera también ha sido necesario obtener formación en múltiples campos por cuenta propia. Sobre todo en el desarrollo en Unity y el trabajo con *scripts* que aun conociendo el lenguaje era un diferente enfoque que ha habido que trabajar.

6.3. Trabajo futuro

Al comienzo del proyecto se plantearon muchas ideas pero por falta de tiempo no todas se han llevado a cabo, se ha obtenido lo que se planteó en los objetivos y en los requisitos de la aplicación pero hay mejoras que estaría bien trabajar más adelante.

Una de las mejoras importantes sería un apartado grafico nuevo enfocado sobre todo a los niños, imágenes, botones, fondos, todo muy colorido y de modo que sea mucho más amigable para los más pequeños, que es a los que más ayudara esta aplicación. Puesto que la actual selección de botones esta sacada de un *plugin* [4] para Unity de elementos para la interfaz que siguen las líneas de diseño de *Material Design*[24].

Otro punto importante sería el de añadir múltiples idiomas, ya que la aplicación está desarrollada enteramente en inglés, y se podría añadir perfectamente la opción de español, e igual en un futuro más idiomas. Para esto habría que añadir un objeto que estuviera en cada escena y se encargara de establecer en cada objeto con una etiqueta el nombre de la etiqueta dependiendo del idioma elegido. Se podría elegir uno de los idiomas disponibles mirando el idioma que hay establecido en el sistema operativo.

Como último punto seria que una vez trabajados los dos puntos anteriores la aplicación estaría totalmente preparada para comercializarse. Puesto que es una aplicación que serviría para ayudar a los núcleos familiares, y por decisión personal, no seguiría un modelo de aplicación de pago. Podría ser puesta en el mercado de aplicaciones de Android bajo un modelo gratuito y de ese modo se obtendría un *feedback* que ayudaría a mejorar más puntos y a seguir desarrollando la aplicación.

[24] https://www.google.com/design/spec/material-design/introduction.html

7. Agradecimientos

Durante el desarrollo del proyecto he recibido el apoyo de mucha gente a la que se lo he agradecido en cada momento pero me gustaría recalcar las tres figuras más importantes.

La primera es mi familia, que ha estado conmigo en todo momento y me ha apoyado y ayudado cuando lo he necesitado. La segunda es mi novia, que no paraba de darme ánimos en todo momento y de alentarme cuando pensaba que no terminaría. Y por último mi tutor del proyecto, que me ayudado en todas las dudas que le he planteado y que ha llevado un seguimiento de mi proyecto digno de un tutor ejemplar.

8. Bibliografía

[1] Gamificación S.L., *Qué es la Gamificación*, http://www.gamificacion.com/que-es-la-gamificacion. [Último acceso: 29 Julio 2015]

[2] Ormrod, Jeanne E. y Esteban C., *Psicología del aprendizaje*, Ed. Pearson, 2012.

[3] Unity3D, *Tutoriales de Unity*, http://unity3d.com/es/learn/tutorials [Último acceso: 31 Agosto 2015]

[4] InvexGames, *Material UI Kit for Unity*, http://invexgames.com/materialUI.html [Ultimo acceso: 31 Agosto 2015]

Thomas Finnegan, *Unity Android Game Development by Example Beginner's Guide*, Packt Publishing, 2013.

Unity3D, *Unity Manual*, http://docs.unity3d.com/Manual/index.html [Último acceso: 31 Agosto 2015]

Unity3D, *Unity Scripting API*, http://docs.unity3d.com/ScriptReference/index.html [Último acceso: 31 Agosto 2015]