Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

UNIVERSIDAD DE JAÉN
ESCUELA POLITÉCNICA SUPERIOR DE JAÉN

Trabajo Fin de Grado

PROTOTIPO DE VIDEOJUEGO PARA NO OCIO ("SERIOUS GAME")

Alumno: Cristóbal Mata Castro

Tutor: Prof.D. Francisco Ramón Feito Higueruela
Tutor: Prof.D. Juan Roberto Jiménez Pérez
Dpto: Ingeniería Informática

Septiembre, 2015

Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Informática

D. Francisco Ramón Feito Higueruela y D. Juan Roberto Jiménez Pérez, tutores del Trabajo Fin de Grado con título: Prototipo de serious game: Marketing del aceite de oliva, que presenta Cristóbal Mata Castro, autorizan su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, 1 de septiembre de 2015

El alumno:	**Tutor:**	**Tutor:**
Cristóbal Mata Castro	Francisco R. Feito Higueruela	J. Roberto Jiménez Pérez

Índice

Capítulo I INTRODUCCIÓN

La importancia del sector de los videojuegos en la sociedad actual está en crecimiento desde hace más de una década y todo apunta a que seguirá este camino.

Esta importancia queda patente en sectores como la educación y el marketing, entre otros. Son sectores en los que se utilizan los llamados “*Serious Games*” o “*Juegos de No-Ocio*” para alcanzar ciertos objetivos de una forma más adaptada a la sociedad actual.

Aprender jugando es mucho más fácil gracias a que el usuario está divirtiéndose. De igual forma una campaña de marketing puede ser mucho más efectiva si utilizamos un videojuego de forma que el jugador recuerde el producto.

Éste es el motivo principal del atractivo que tienen los “*Serious Games*”, un enorme potencial que puede ser explotado en una gran cantidad de áreas.

Además, la tecnología en la que ejecutar los videojuegos está al alcance de millones de personas, más teniendo en cuenta que este tipo de juegos no suele requerir una gran potencia de procesamiento y por tanto no se necesitan equipos de última generación.

En este trabajo hablaremos sobre este mercado en auge y estudiaremos algunos de los motores de videojuegos más importantes del mercado para encontrar el que nos brinde más beneficios teniendo en cuenta el proyecto que vamos a diseñar para finalmente construir el prototipo de un “*Serious game*” orientado al marketing de aceite de oliva.

1.1. Propósito general

Este trabajo surge tras la necesidad de explorar el campo de los “*Serious Games*” para utilizarlo tanto en el campo del marketing como en el de la educación aprovechando el actual auge del mismo.

Se pretende realizar un breve **análisis de algunos de los motores de videojuegos** más utilizados actualmente y más interesantes **para construir un prototipo de un videojuego para no ocio basado en el aceite de oliva**.

1.2. Objetivos

Los objetivos de este trabajo son los siguientes:

- Estudio y comparativa de algunos motores de videojuegos actuales.
- Diseño de un videojuego para no ocio relacionado con el aceite de oliva y orientado al marketing principalmente, aunque se añadirán algunos matices relacionados con la educación (aprendizaje sobre su fabricación).
- Estudio del público óptimo al que hacer objetivo.
- Construcción de un prototipo funcional para plataformas Android utilizando uno de los motores estudiados.
- Diseño de la forma de publicitar el aceite de oliva integrado en la jugabilidad del prototipo.
- Dotar al prototipo de un servicio propio apoyado en una arquitectura muy similar a REST que utiliza el protocolo HTTP para la comunicación entre dos sistemas, y así añadir características de red tales como un ranking de jugadores.
- Memoria detallada sobre el desarrollo del trabajo indicado.

1.3. Motivación

Como hemos comentado anteriormente, los “*Serious Games*” pueden ayudar a cumplir ciertos objetivos en diversos campos como la educación y el marketing, a los que vamos a orientar este trabajo.

Actualmente es el mejor momento para explorar este tipo de videojuegos por dos factores:

- **Los videojuegos son cada vez más importantes en la sociedad actual.** Hoy todo el mundo quiere jugar, la percepción de los videojuegos entre la población ha cambiado desde hace unos años cuyo público principal eran niños. Tal es el cambio que actualmente la edad media del jugador en Europa ronda los 35 años y está formado prácticamente por el mismo número de hombres que de mujeres [1].
- **La tecnología móvil está totalmente extendida en nuestra sociedad.** Esta tecnología nos permite que cualquiera de los millones de usuarios de los llamados Smartphones puedan jugar en cualquier momento. Prácticamente todo el mundo a partir de cierta edad utiliza uno de estos dispositivos.

Por estos motivos parece obvio que los “*Serious Games*” son una oportunidad de negocio actual que merece, al menos, ser explorada.

Además, en la actualidad hay una gran cantidad de motores de videojuegos en el mercado[2] e investigar sobre ellos es crucial para encontrar la forma de construir estos proyectos lo más eficientemente posible para ahorrar recursos y así minimizar riesgos en una situación económica como la actual.

Capítulo II TECNOLOGÍAS USADAS

A continuación se justifican las tecnologías, en este caso software exclusivamente, que se van a utilizar para el desarrollo de este trabajo incluyendo la construcción del prototipo del videojuego que se va a construir.

2.1. Elección del motor de videojuegos

Actualmente existe una gran variedad de motores de videojuegos, que simplifican el desarrollo de los mismos, en el mercado [2[. Por ello, de acorde al género y a los objetivos del juego que se va a diseñar debemos estudiar los diferentes motores y elegir el que consideremos óptimo y así ahorrar recursos.

Antes de comenzar con el diseño y análisis real del juego, conocemos algunos requisitos sobre el mismo:

- Estará enfocado a plataformas móviles.
- No será un juego complejo: más público.
- El principal objetivo del mismo debe ser el marketing.

Teniendo en cuenta los puntos anteriores, se han estudiado tres motores de videojuegos para el desarrollo del prototipo.

2.1.1. Unity3D

Unity3D es **posiblemente el motor más importante en el desarrollo de videojuegos**[3] **de la actualidad**, sobre todo indies. Su competencia más directa es *Unreal Engine*[4].

Este motor de videojuegos cuenta con una gran comunidad de usuarios junto a una extensa documentación, lo que hará que en caso de necesitar soporte será relativamente sencillo solucionar nuestros problemas.

Permite usar varios lenguajes de programación como JavaScript o C# y está basado en componentes.

Unity3D contiene un completo motor de físicas 2D y 3D que nos facilitará mucho trabajo. Nosotros definiremos qué hacer una vez suceda interacción y definiremos los volúmenes de colisión. Además podemos simular fuerzas automáticamente con dicho sistema de físicas.

Además es líder en la exportación a varias plataformas. Un mismo proyecto puede ser construido para prácticamente cualquier plataforma actual del mercado; desde *Microsoft Windows* hasta *Sony PlayStation 4™*, pasando por *Android*, *iOs* y otras consolas como *Nintendo Wii U™*.

Es una opción **muy potente para juegos en 3D**, **pero no así para juegos en dos dimensiones** que aunque es posible su desarrollo, no es todo lo cómodo ni eficiente que debiese.

Este motor era de pago hasta la versión 5, que se ha lanzado totalmente gratuita para usuarios con un margen de beneficios anuales fijado por la empresa responsable del motor.

Antes de esta versión se podía desarrollar gratuitamente utilizando este motor, pero había bastantes funcionalidades limitadas.

Unity3D							
3D	**2D**	**PC**	**Consola**	**Android**	**iOS**	**Precio**	**Lenguaje**
Sí	Sí	Sí	Sí	Sí	Sí	Gratis	C#, JS

Tabla 2.1. Características de Unity3D [5]

2.1.2. WaveEngine

Otra opción que se decidió estudiar fue este motor relativamente nuevo en el mercado y que está creado en España.

Éste permite el desarrollo utilizando lenguajes .NET como C# o VB, y del mismo modo que *Unity3D,* está basado en componentes.

Sobre las plataformas de exportación, *WaveEngine* es mucho más básico que *Unity3D*: sólo permite exportación a *Android*, *iOS*, *Microsoft Windows*, *Linux* y *Mac*. Pero podría ser interesante ya que las plataformas móviles, que son la que nos interesan, están entre ellas.

WaveEngine puede usarse tanto para **desarrollo de videojuegos en dos dimensiones como en 3D**. Aunque la versión con la que se experimentó no gozaba de una interfaz muy potente a la hora de diseñar los escenarios en ninguna de las dos opciones y no era muy robusto, especialmente el gestor de *Assets* del juego que mostraba errores continuamente hasta que dejó de funcionar.

WaveEngine también dispone de sistema de físicas que nos facilitarán muchos cálculos en el juego y dispone de otra gran ventaja: es totalmente gratuito, sin versiones de pago o suscripciones.

Pero quizá el problema más importante que hará que no se elija este motor como herramienta es el soporte actual: el motor es bastante novedoso y no goza de una comunidad ni documentación como la de otros motores que se han estudiado.

WaveEngine							
3D	**2D**	**PC**	**Consola**	**Android**	**iOS**	**Precio**	**Lenguaje**
Sí	Sí	Sí	No	Sí	Sí	Gratis	C#, VB

Tabla 2.2. Características de WaveEngine [6]

2.1.3. GameMaker Studio

2.1.3.1. Definición

La última opción que se ha estudiado es el motor de videojuegos creado inicialmente por *Mark Overmars*, profesor de la *universidad de Utrecht*, como herramienta para aprender a crear videojuegos[7]. Finalmente éste evolucionó hasta convertirse hoy en día en uno de los **motores de videojuegos 2D** más interesantes y utilizados del mercado.

GameMaker Studio no está basado en componentes, su arquitectura es algo más especial: pinceladas de objetos, eventos y recursos. Sus principales propiedades son:

- **Scripts, sonidos, sprites, escenas, objetos**, todo son recursos que internamente están identificados con un código numérico.
- Los objetos pueden usar **herencia simple**, pueden tener atributos de una forma muy especial que se comentará a continuación, pero **no pueden tener funciones**.
- Los objetos están **compuestos de eventos** que ejecutarán los scripts cuando estos se disparen.

Contiene un gran motor de físicas para entornos 2D, aunque el de *Unity3D* parece un poco más completo gracias a la aplicación interna de fuerzas que añade este último de forma transparente al desarrollador.

GameMaker Studio está prácticamente al nivel de *Unity3D* si de módulos de exportación hablamos: la única plataforma a la que no se puede exportar respecto a este último es *Nintendo Wii U™*.

Dispone de una versión gratuita llamada Standard que tiene bastantes limitaciones y con la que se puede exportar a *Microsoft Windows* exclusivamente. Hay que comprar cada módulo de exportación y pasarse a la versión profesional.

Éste motor no utiliza lenguajes de programación estándar, sino que es necesario conocer **GML** *(GameMaker Language),* **un lenguaje de**

programación propio de muy alto nivel orientado a gráficos y videojuegos que nos hará todo muy sencillo. Pero toda facilidad tiene un problema colateral y *GameMaker Studio* no iba ser la excepción.

La principal facilidad a la hora de programar es la libertad declarativa total de los atributos y variables que ofrece GML. Esto significa que podemos crear un atributo del objeto en cualquier parte así como editarlo o cambiar su tipo en tiempo de ejecución.

¿Dónde está el problema? Podemos pensar que no difiere mucho de lenguajes como *JavaScript* o *PHP*, pero el problema está en la arquitectura del sistema: objetos que contienen eventos.

Pensemos un objeto que en un evento específico, como puede ser una alarma, crea una variable, y en otro evento del mismo se utiliza la misma. **Es imposible que el editor de código conozca el orden de ejecución de los eventos a priori**, toda la responsabilidad de conocer el ciclo de ejecución cae sobre el desarrollador. El juego compilará, pero puede cerrarse en la ejecución.

Éste es solo uno de los problemas de la libertad que ofrece el motor a la hora de programar. Sin lugar a dudas hay otro no menos importante heredado: si **el editor no puede conocer el orden de ejecución, no puede saber qué variables pueden usarse en cada momento y no podrá recordarnos las que tenemos definidas**. Además no hay atributos privados, todos son públicos y accesibles desde cualquier objeto siempre y cuando la instancia esté activa en la escena actual.

Esto vuelve a significar que todo recae en el desarrollador. Será el encargado de memorizar todas las variables de todos los objetos y las que no forman parte de los mismos (variables globales) y saber cuándo podrá utilizar cada una de ellas y cuando no. Documentar todo esto para su memorización es una tarea complicada.

Para finalizar, cuando un objeto hereda de otro en este entorno de desarrollo, hereda sus eventos así como puede usarse para generalizar y por

ejemplo eliminar todos los objetos “enemigo” aunque sean diferentes tipos de objetos lanzando la orden de destruir al padre de la escena.

Esto hace que la documentación cuando se usa *GameMaker Studio* sea complicada de generar y entender si usamos diagramas clásicos. Su metodología óptima es *Extreme Programming.*

Como nota adicional, **en *GameMaker Studio* no existe el concepto de *clase*** como tal, **sino** que se crean ***objetos* que pasan a ser *instancias* en la escena**. Es por esto que a lo largo de todo el documento se llamará *objeto* a lo que en otros entornos se hace llamar *clase*.

GameMaker Studio							
3D	**2D**	**PC**	**Consola**	**Android**	**iOS**	**Precio**	**Lenguaje**
NR	Sí	Sí	Sí*	Sí	Sí	150$**	C#, JS

NR: *No recomendado* **Excepto Wii U* ***Más precio de módulos (100-299 cada uno)*

Tabla 2.3. Características de GameMaker Studio [8]

2.1.3.2. Justificación de uso

Aún con los problemas comentados anteriormente, se ha decidido utilizar *GameMaker Studio* al considerarse la opción que más se ajusta al tipo de videojuegos que queremos construir, además de por el conocimiento adquirido tras diez años de uso del motor.

Debemos tener en cuenta que las licencias del motor no son caras y que se especializa en el desarrollo de videojuegos en dos dimensiones gracias a herramientas mucho más potentes que las que disponen Unity3d o WaveEngine para dicho estilo.

Todo esto permitirá agilizar mucho el desarrollo del prototipo o futuro videojuego que se traducirá como menos costes.

2.2. GIMP

2.2.1. Justificación de uso

Para desarrollar un videojuego es necesario obtener una gran cantidad de contenido gráfico.

Éste se va a generar de forma totalmente original por el responsable del proyecto para el prototipo. Por tanto se debe utilizar una herramienta de edición gráfica que permita generar *sprites* con un mínimo de calidad.

Finalmente se ha elegido *GIMP* frente a herramientas como *Adobe PhotoShop* debido a su licencia y a la experiencia con la misma.

GIMP es una herramienta de manipulación de imágenes y diseño gráfico gratuito (licencia *GNU*) disponible para las principales plataformas de escritorio.

Permite exportar a una gran cantidad de formatos y dispone de las principales herramientas similares a las de *Adobe PhotoShop*; es su competencia más directa.

2.3. Android SDK

2.3.1. Justificación de uso

El videojuego está enfocado en plataformas móviles, pero para el prototipo nos centraremos en Android debido a su coste más bajo. Para poder construir una aplicación de cualquier tipo para este sistema necesitamos instalar el *Android SDK*.

Android SDK es un conjunto de librerías y herramientas *Java* que utilizan las aplicaciones para *Android* en dicho sistema operativo.

Contiene las diferentes versiones del sistema y además una máquina virtual en la que emular cada una de ellas en nuestro PC y probar las aplicaciones que

construimos directamente. Evitando la necesidad de probarlas en sistemas físicos para cada versión de *Android*.

Además contiene las herramientas necesarias para construir la aplicación y obtener un archivo *APK*. Es una herramienta esencial para el desarrollo de software en *Android*.

Hay que configurar esta herramienta en la configuración de *GameMaker Studio*[9].

2.3.2. Plataformas Java

Se ha utilizado *JDK 1.7* para la construcción del videojuego, pero podría utilizarse cualquier versión superior a la 1.6 (incluida). **YoYoGames no recomienda usar la 1.8**, aunque se ha probado y funciona correctamente.

Utilizar *JDK* es totalmente necesario ya que es el entorno que utiliza Android para sus aplicaciones. Hay que configurar la plataforma Java que vamos a utilizar en *GameMaker Studio* o no podremos construir un *APK*[9].

2.3.3. Plataformas Android instaladas

Todas las plataformas de este sistema operativo han sido instaladas a través del *SDK manager*. Si queremos llegar al mayor público posible es necesario que el *APK* sea compatible con la mayor cantidad de terminales ya que el mercado de *Android* está muy segmentado en lo que a versiones de sistema se refiere [10].

Además es conveniente instalar todas las herramientas de construcción y librerías como la de *Google Play Services* que contienen una gran cantidad de funcionalidades que es muy posible que usemos como la publicidad o conectar con la cuenta de *Google* del usuario.

2.4. PHP y mySQL

2.4.1. Justificación de uso

Se quiere dotar al juego de un servicio online con un protocolo propio pero muy similar a *REST* con el que hacer un ranking de puntuaciones. Para ello es necesario un servidor que soporte *PHP* y otro *mySQL*.

2.4.2. Definición

Para montar este servicio web se ha decidido utilizar *PHP* y *mySQL* por lo que usando *Apache* como servidor el problema estaría resuelto.

Usando un servicio *REST* se consigue la comunicación entre dos sistemas separados de forma sencilla. Éste se apoya en el protocolo *HTTP* de forma que se crea una interfaz independiente de cualquier lenguaje de programación y accesible desde cualquier aplicación que pueda usar dicho protocolo.

Son servicios muy flexibles al devolver información plana en vez de tipificada. Puede ser leída directamente desde un navegador.

Es por eso que se utilizan codificaciones para poder enviar objetos entre los sistemas, y uno de las más utilizadas es ***JSON*** (*JavaScript Object Notation*). Ésta convierte los objetos en texto plano y viceversa con el formato indicado en la tabla 2.4.

Clase “coche” en JSON

```
{ coche: [
     { “volante” : 5, “propietario” : “Ruperto” }
     { “volante” : 3, “propietario” : “Antonia” }
]}
```

Tabla 2.4. Ejemplo de la codificación JSON

Capítulo III DESARROLLO

La metodología que se va a utilizar para la gestión y desarrollo del proyecto está basada ***eXtreme Programming*** (*XP*), clasificada como ágil. No se va a poder seguir al completo debido a la naturaleza del proyecto pero sí que se van a utilizar todas las propiedades con sentido en el mismo; *XP* es muy adaptable.

Uno de los problemas que se tiene es sobre los roles: una misma persona se va a encargar de ser todos ellos. También la programación en parejas. Elementos[11] que se obviarán o minimizarán su importancia en este trabajo.

3.1. Definición del sistema

3.1.1. Alcance

Se va a diseñar un videojuego centrado en el marketing del aceite de oliva y construir su correspondiente prototipo con el que se podrán jugar a algunos minijuegos y probar todas las funcionalidades que incluirá el proyecto.

El prototipo debe ser completamente funcional e integrar alguna forma de promocionar el aceite de oliva.

3.1.2. Público objetivo

Antes de diseñar un videojuego de cualquier tipo, lo ideal es tener claro un público que será el principal objetivo al que convencer.

Por ello, y más teniendo en cuenta que el videojuego estará enfocado principalmente al marketing, lo primero que se ha investigado es sobre este tema para obtener un diseño atractivo que haga que el marketing sea efectivo.

Podemos pensar que un buen público serían los niños ya que son grandes consumidores de videojuegos, pero realmente, ¿servirá como campaña de marketing del aceite de oliva para este público? **Los niños no compran estos productos**, todo sería diferente si se tratasen de juguetes.

Además, en general el público infantil (cinco años) no tiene habilidades ni conocimientos suficientes para juegos con nivel de reto moderado según se ha comprobado con la experiencia personal en otros proyectos. Éste público sería más óptimo en caso de ser un juego más enfocado al sector educativo.

Por este mismo motivo se ha seleccionado **un público más adolescente sin cerrarse al adulto** temprano ni a niños con más de siete años. Se intentará enfocar a todo el público posible: el juego será muy casual, **con partidas breves** para que todo tipo de usuarios puedan jugar ya sea en el transporte público, después de comer o cualquier rato libre por breve que éste sea pero debe ser desafiante.

Todos los públicos buscan algo en común al jugar a videojuegos: divertirse. Ésta debe ser la premisa principal a la hora de diseñar el juego.

3.2. Planificación

3.2.1. Actividades

Para obtener una visión general de las actividades del proyecto, se ha diseñado el mapa mental de la figura 3.1 con la ayuda de *XMind* [20].

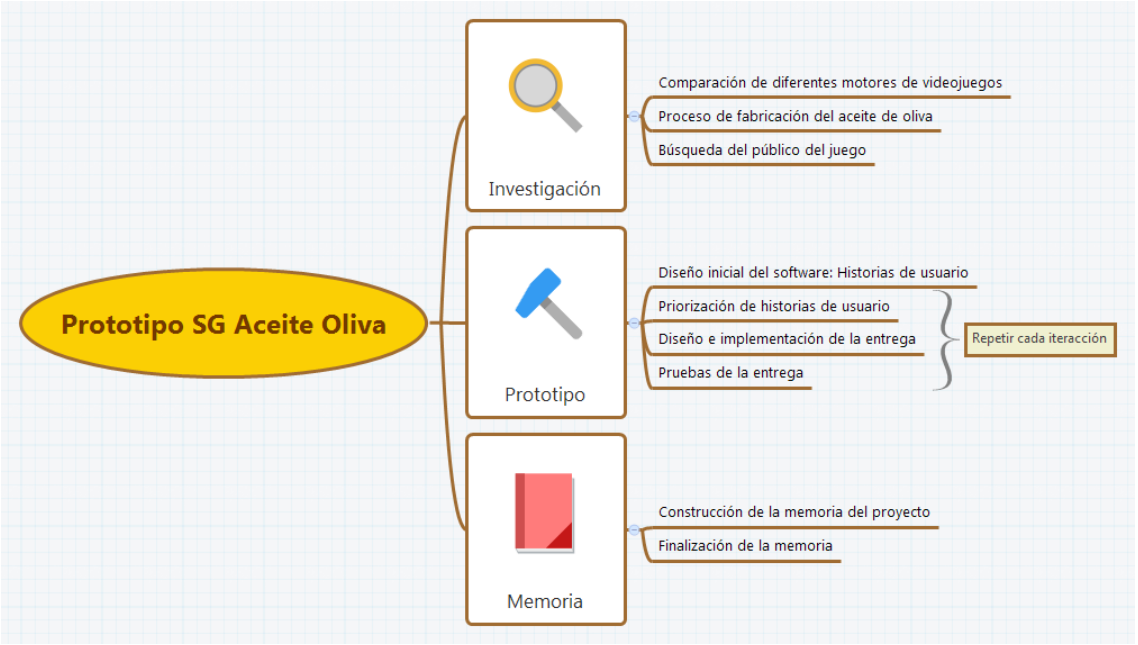


Figura 3.1. Mapa mental de actividades

3.2.2. Estimación de tiempo del proyecto

En cada una de las tareas del proyecto se incluye su propia parte relativa a la construcción de la memoria. Aun así, hay que tener en cuenta la fase de finalización de la misma que no se hará al mismo tiempo.

El desarrollo del prototipo se va a realizar basándose en metodologías ágiles, concretamente *XP*, con varias iteraciones. Por ello, y teniendo en cuenta los créditos *ECTS* del *Trabajo Fin de Grado* y que se va a construir un prototipo, se han añadirán solo tres minijuegos basados en las etapas de fabricación del aceite de oliva.

A continuación se muestra una estimación de cada una de las tareas indicadas en el apartado anterior.

ID	Nombre	Horas
INV-MOT	Comparación de motores de videojuegos	24
INV-FAB	Investigación sobre la fabricación de aceite de oliva	20
INV-PUB	Búsqueda del público del juego	20
PRO-HIS	Historias de usuario	24
PRO-PRI	Posibles nuevas historias de usuario, priorización de HU, definición de iteración	16
PRO-IMP	Implementación de la iteración	160
PRO-ENT	Finalizar iteración y pruebas	48
MEM-FIN	Finalización de la memoria	56
Total		368

Tabla 3.1. Tareas del proyecto junto a la estimación de tiempo

Las etapas de creación del prototipo de la tabla son la estimación total de todas las iteraciones, por lo que la tarea *PRO-HIS* sólo se realiza una vez.

En total, se ha estimado 360 horas, o lo que es lo mismo si se trabajan ocho horas diarias: 46 días. Dato que se ve de forma más clara en el siguiente diagrama.

3.2.3. Diagrama de Gantt

Para obtener una representación visual de las tareas a realizar en el proyecto se ha creado un diagrama de *Gantt* usando *Microsoft Project*.

El resultado ha sido el que se muestra en la figura 3.2.

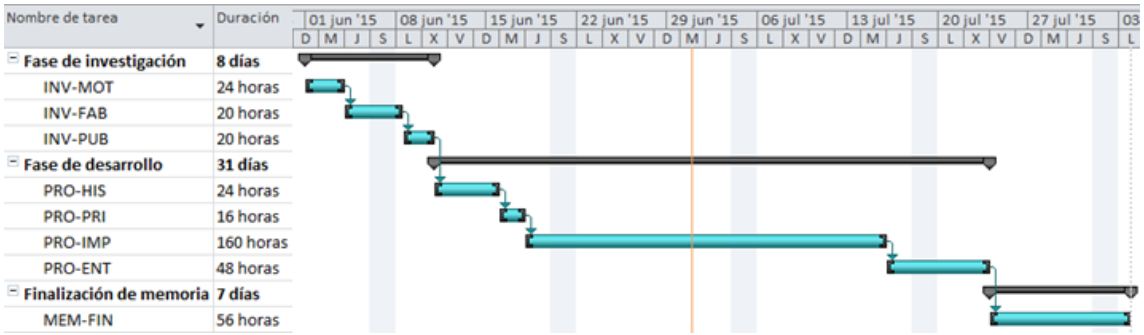


Figura 3.2. Diagrama de Gantt de la planificación

Las tareas que se repiten durante el desarrollo no se muestran correctamente organizadas en el gráfico debido al no conocimiento previo de las iteraciones (se definen tras las historias de usuario), pero el tiempo total sí es correcto. Consideremos toda la fase de desarrollo como un mismo paquete (tarea resumen del gráfico).

Se ha utilizado una jornada completa de ocho horas de lunes a viernes para la estimación, obteniendo como resultado 46 días o 2 meses aproximadamente sin contar fines de semana.

El trabajo comenzó el 1 de junio y debería estar acabado sobre el 3 de agosto.

3.3. Análisis y diseño inicial

Antes de comenzar con la construcción del prototipo debemos tener claro qué queremos hacer. Por ello el primer paso es **diseñar los minijuegos y** tras esto **crear las historias de usuario que representarán los requisitos** del proyecto.

Al ser un proyecto no muy grande, estos requisitos se complementan con las tablas de diseño del videojuego de las que vamos a hablar en este apartado. Esto quiere decir que solo utilizaremos estas dos herramientas para el análisis y diseño del prototipo en lugar de usar otras más clásicas como los diagramas de casos de uso o secuencia que sólo ralentizarán el desarrollo del mismo.

3.3.1. Diseño de los minijuegos

En esta etapa se han diseñado los minijuegos que estarán incluidos en el prototipo desde la perspectiva del “*game designer*”, no desde la del desarrollador. En otras palabras: se van a definir las mecánicas y objetivos de cada uno de los minijuegos que componen el prototipo. A raíz de este diseño, se construirán las historias de usuario que serán las que indiquen al desarrollador cómo construir el sistema.

Para diseñar los mismos se ha decidido orientarlos al **proceso de fabricación del aceite de oliva** y para ello se ha investigado sobre el mismo.

Resumiendo, hay varias etapas para la fabricación de este fluido[12]:

1. **Clasificación de las aceitunas recibidas.** Se separan en sanas y defectuosas (cogidas del suelo, picadas o enfermas) para garantizar la mejor calidad del aceite.
2. **Lavado del fruto.** Se limpian las aceitunas para eliminar residuos como barro, hojas y tallos.
3. **Molienda de la aceituna.** Se obtiene una masa tras moler el fruto.
4. **Batido de la masa.** De esta forma se favorece la separación del aceite del resto de materia.

5. **Separación de los componentes:** Normalmente se centrifuga el batido anterior para separar aceite, alpechín y orujo debido a las distintas densidades que tiene cada uno de ellos.
6. **Conservación.** Se conserva de tal forma que no pierda propiedades.

La versión final del juego debería de incluir cada una de las etapas comentadas. Pero para el prototipo vamos a implementar únicamente las tres que se detallan en los siguientes apartados.

3.3.1.1. Primer minijuego

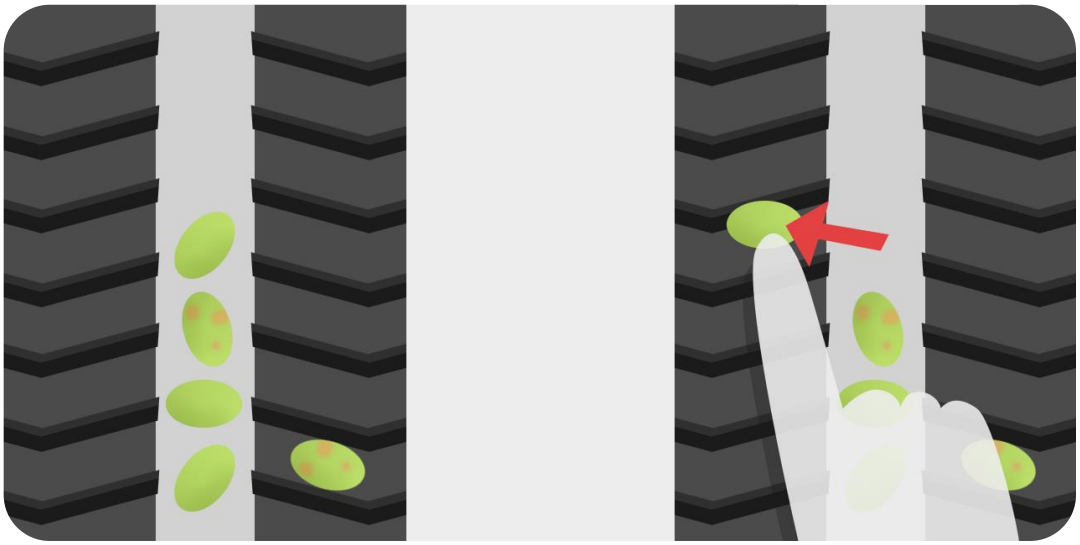


Figura 3.3. Boceto del primer minijuego

El primero de los minijuegos se basa en la primera etapa del proceso de fabricación del aceite de oliva, en la que se separan las aceitunas defectuosas de las sanas. El nivel se resume brevemente de forma visual en la figura 3.3.

El reto del jugador arrastrar las aceitunas hacia las cintas transportadoras para clasificarlas. Éstas aparecerán desde la zona superior de la pantalla y la cola no deberá colapsarse.

En la tabla 3.2 se indica toda la información de diseño del mismo.

Minijuego	**Etapa**	**Primera etapa**
	Nombre	Separación de aceituna sana
Mecánicas	**Reglas**	• Hay dos tipos de aceitunas: sanas y defectuosas. • El jugador debe arrastrar las aceitunas con el dedo a su correspondiente canal. • El jugador no puede clasificar mal las aceitunas. • El jugador no debe permitir que la cola de aceitunas rebose. • El ritmo del juego debe crecer en intensidad a medida que el jugador progrese. • La correcta clasificación de las aceitunas sumará puntos, en caso contrario terminará el minijuego con los puntos obtenidos.
	Objetivos	• Clasificar las aceitunas en su canal correspondiente dependiendo de su estado.
	Criterios de éxito	No procede.
	Criterios de fracaso	• Aceituna mal clasificada. • Colapso de aceitunas en la cola.
	Habilidades	• Clasificar aceitunas arrastrándolas.
Sonido		Se añadirá música de fondo para colaborar al estado de ánimo del jugador y efectos sonoros a los elementos con interacción.
Efectos visuales		Efectos de los mensajes al iniciar y acabar la partida.

Tabla 3.2. Diseño del primer minijuego

3.3.1.2. Segundo minijuego

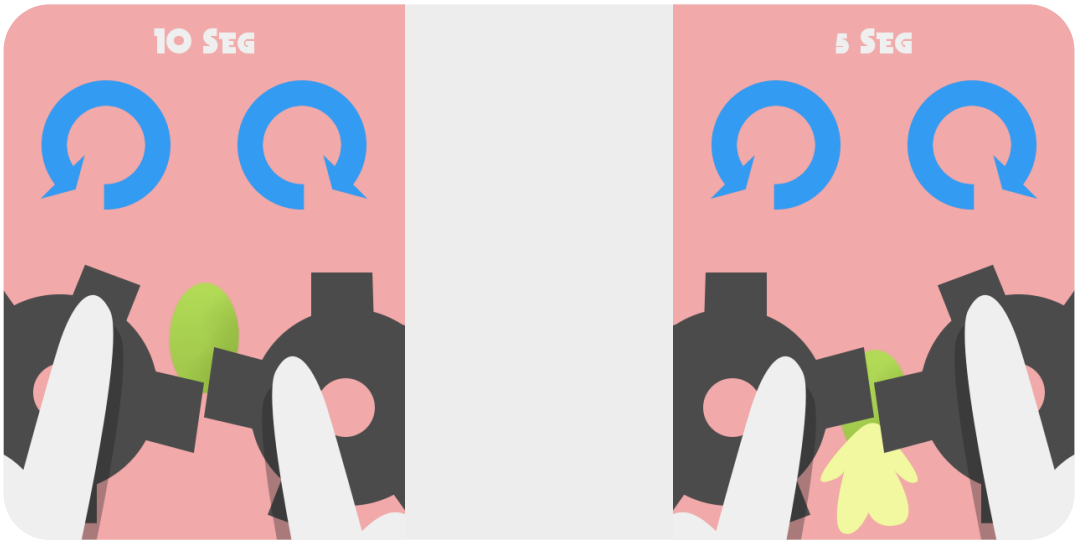


Figura 3.4. Boceto del segundo minijuego

Para el siguiente nivel se ha dejado a un lado la segunda fase del proceso de fabricación del aceite de oliva para centrarse en la tercera: la molienda de la aceituna. El nivel se resume brevemente en la figura 3.4.

En este nivel el usuario tendrá como reto girar dos rodillos/engranajes en las direcciones indicadas por el juego para moler la máxima cantidad de aceituna posible durante un periodo de tiempo fijado. Una vez éste termina, se acaba el nivel.

Toda la información de diseño del mismo se encuentra en la tabla 3.3.

Minijuego	**Etapa**	**Tercera etapa**
	Nombre	Molienda de la aceituna
Mecánicas	**Reglas**	• Girar los dos engranajes en el sentido correcto para moler la aceituna antes de que acabe el tiempo • Sólo contará en la puntuación las aceitunas correctamente molidas. • La dirección de giro cambiará arbitrariamente.
	Objetivos	• Moler las aceitunas.
	Criterios de éxito	No procede.
	Criterios de fracaso	Fin del tiempo.
	Habilidades	• Triturar las aceitunas usando los engranajes.
Sonido		Se añadirá música de fondo para colaborar al estado de ánimo del jugador y se añadirán efectos sonoros divertidos al moler aceitunas, girar los engranajes, comenzar la partida, acabarla y resto de elementos con interacción.
Efectos visuales		Efectos de los mensajes al iniciar y acabar la partida. Manchar la pantalla con el batido de la aceituna.

Tabla 3.3. Diseño del segundo minijuego

3.3.1.3. Tercer minijuego

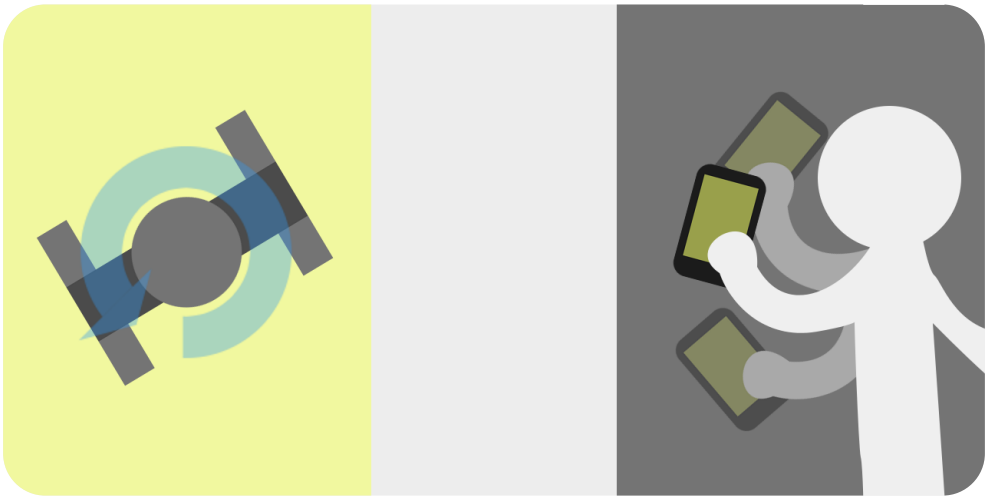


Figura 3.5. Boceto del tercer minijuego

Finalmente, el tercer y último minijuego del prototipo se basa en la cuarta etapa del proceso de fabricación: batir la masa de la aceituna como se muestra en la figura 3.5.

En esta etapa el usuario deberá agitar su dispositivo tan rápido como le sea posible durante un tiempo fijado para obtener puntos.

En la tabla 3.4 se indican todos los detalles relativos al diseño de este nivel.

Minijuego	**Etapa**	**Cuarta etapa**
	Nombre	Batido de la masa de la aceituna
Mecánicas	**Reglas**	• Agitar el dispositivo para batir la masa de la aceituna. • La cantidad de movimiento se verá reflejada en la puntuación.
	Objetivos	• Favorecer la salida del aceite de la masa.
	Criterios de éxito	No procede.
	Criterios de fracaso	Fin del tiempo.
	Habilidades	• Batir la masa moviendo el dispositivo.
Sonido		Se añadirá música de fondo para colaborar al estado de ánimo del jugador y se añadirán efectos sonoros divertidos batir la masa, comenzar la partida, acabarla y resto de elementos con interacción.
Efectos visuales		Efectos de los mensajes al iniciar y acabar la partida.

		Habrá una batidora que girará según movamos el dispositivo. Efecto de manchar la pantalla con el líquido.

Tabla 3.4. Diseño del tercer minijuego

3.3.2. Historias de usuario

En primer lugar, se muestran las historias de usuario relativas al juego en general, para después seguir hablando sobre las de cada uno de los minijuegos diseñados.

Historia 1	**Clasificación de partidas**
Como	Jugador
Quiero	Clasificar mi partida una vez terminada en los diferentes tipos de aceite de oliva
Para	Tener una idea rápida de mi actuación

Historia 2	**Ranking de puntuaciones**
Como	Jugador
Quiero	Que mi puntuación se comparta en un ranking
Para	Competir con otros jugadores

Historia 3	**Registro de usuarios**
Como	Jugador
Quiero	Compartir puntuaciones sin necesidad de registro complejo
Para	No restar tiempo de juego

Historia 4	**Interfaz y sistema del servidor**
Como	Futuro desarrollador
Quiero	Una interfaz de comunicación servidor-juego muy similar a REST y una base de datos para el ranking
Para	Que sea muy flexible

Historia 5	**Promoción de productos**
Como	Comerciante de aceite de oliva
Quiero	Promocionar mis productos usando códigos de descuento
Para	Captar nuevos clientes

Historia 6	**Efecto manchado**
Como	Jugador
Quiero	Ver mancharse la pantalla
Para	Ver más dinamismo

Historia 7	**Menú principal**
Como	Jugador
Quiero	Tener un menú de opciones
Para	Salir del juego, ver rankings, desactivar sonido y jugar

3.3.2.1. Primer minijuego

Historia 8	**Clasificación de la aceituna**
Como	Jugador
Quiero	Clasificar la aceituna arrastrándola hasta una zona
Para	Obtener puntos

Historia 9	**Dificultad en J1**
Como	Jugador
Quiero	Que la velocidad de clasificación aumente con el tiempo
Para	Hacer el juego más desafiante

Historia 10	**Fin de J1**
Como	Jugador
Quiero	Jugar sin final hasta que pierda la partida al acumular demasiada aceituna o fallar en la clasificación de la misma
Para	Hacer el juego más desafiante

3.3.2.2. Segundo minijuego

Historia 11	**Girar engranajes**
Como	Jugador
Quiero	Girar los engranajes con mis dedos simultáneamente haciendo círculos durante un tiempo determinado
Para	Moler las aceitunas antes de que acabe el tiempo

Historia 12	**Sistema de direcciones**
Como	Jugador
Quiero	Que se muestren las direcciones a las que he de girar cada uno de los engranajes
Para	Moler las aceitunas correctamente

3.3.2.3. Tercer minijuego

Historia 13	**Batir masa**
Como	Jugador
Quiero	Batir la masa de la aceituna agitando mi dispositivo
Para	Obtener puntos antes de que acabe el tiempo

3.3.3. Estimación de las historias de usuario

El siguiente paso es estimar la dificultad de cada una de las historias de usuario para gestionar mejor el progreso del proyecto mediante la asignación de puntos de historia.

Si el equipo estuviese formado por varias personas, una buena técnica para la estimación es “*Planning Poker*” [13], muy particular en *XP*: en primer lugar alguien explica la historia de usuario y cada miembro asigna un número de puntos de historia utilizando cartas con ciertos valores que pueden ser personalizados, aunque lo más estándar es utilizar 0, ½, 1, 2, 3, 5, 8, 13, 20, 40, y 100. En caso de no haber consenso se debaten las puntuaciones más extremistas y se repite la votación hasta que lo haya.

Historia	PH
Clasificación de partidas	1
Ranking de puntuaciones	13
Registro de usuarios	8
Interfaz y sistema del servidor	8
Promoción de productos	5
Efecto manchado	2
Menú principal	13
Clasificación de la aceituna	5
Dificultad en J1	1
Fin de J1	1
Giro de engranajes	3
Sistema de direcciones	3
Batir masa	3

Tabla 3.5. Estimación del coste de las historias de usuario

Debido a que este trabajo es unipersonal, no tiene mucho sentido utilizar esta técnica en su completitud. En su caso, utilizando la escala numérica mencionada, se ha estimado como se muestra en la tabla 3.5 obteniendo un total de 66 puntos de historia para la construcción del prototipo.

El siguiente paso es calcular la velocidad de desarrollo: ya estimamos anteriormente que la parte de desarrollo del prototipo duraría aproximadamente un mes, confiando en que así será y con iteraciones semanales (común en *XP*) la velocidad estimada es 66 puntos divididos por las cuatro semanas del mes: aproximadamente **17 puntos de historia por semana/iteración**.

Utilizaremos este valor para calcular las historias de usuario de cada iteración una vez estén priorizadas y para construir el *Burndown Chart*[14], un gráfico más característico de *SCRUM* pero que puede utilizarse en otras metodologías y que nos ayudará a conocer el estado actual del proyecto y si nuestras previsiones se cumplirán o tendremos que darnos prisa.

Figura 3.6. Burndown Chart inicial

3.4. Primera iteración

El desarrollo del prototipo se divide en iteraciones en las que se repite el mismo proceso. Recordemos que la metodología de trabajo es *Extreme Programming* (*XP*), una metodología ágil.

3.4.1. Priorización de historias de usuario

Una vez debatidas las historias y sus estimaciones, con el trabajo que queremos construir bastante más claro, vamos a ordenar las historias de usuario por prioridad según nuestro propio criterio (debería hacerlo el cliente, pero su rol también es nuestro). La lista de historias se indica en la tabla 3.6.

Historia	PH
Menú principal	13
Interfaz y sistema del servidor	8
Registro de usuarios	8
Ranking de puntuaciones	13
Clasificación de la aceituna	5
Dificultad en J1	1
Fin de J1	1
Efecto manchado	2
Sistema de direcciones	3
Giro de engranajes	3
Batir masa	3
Clasificación de partidas	1
Promoción de productos	5

Tabla 3.6. Historias de usuario priorizadas para la primera iteración

Teniendo en cuenta que la velocidad que hemos estimado era de unos 17 puntos de historia por iteración, tiene sentido que la primera de ellas esté formada por las dos primeras historias de usuario priorizadas (marcadas en amarillo en la tabla anterior).

3.4.2. Diseño de la iteración

El primer lugar dividimos las dos historias de usuario en tareas que debemos realizar para completar la iteración obteniendo el resultado indicado en las tablas 3.7 y 3.8.

Menú principal
Diseño gráfico de los diferentes botones
Implementación del botón
Implementación del controlador
Diseño gráfico de las notificaciones
Implementación de notificaciones
Diseño gráfico de la pantalla de título
Scripts como interfaz para gestión del sonido
Añadir sonidos a la interfaz
Implementación funcional del menú principal
Preparación para distintos idiomas

Tabla 3.7. Tareas de la historia “Menú principal”

Interfaz y sistema del servidor
Diseño de la base de datos del sistema
Diseño del protocolo de comunicación propio similar a REST

Tabla 3.8. Tareas de la historia “Interfaz y sistema del servidor”

Una vez las historias de usuario están divididas en tareas, se han llevado a cabo en el orden indicado en la misma división.

Dejando a un lado las tareas gráficas y los efectos visuales de los botones y notificaciones, en los siguientes apartados se encuentra todo el proceso de análisis y desarrollo de las tareas comentadas.

3.4.2.1. Menú principal

El sistema cuenta con controladores cuya interfaz de entrada con el usuario son objetos que notifican al controlador cuando se activan (interfaces activas) o los mismos pueden chequear los estados de las interfaces (pasivas).

Un ejemplo de interfaz pasiva es el engranaje del nivel dos que se comentará en su correspondiente sección. Un ejemplo de interfaz activa son los botones o las notificaciones.

Pero no todas las interfaces activas son iguales; un botón no es lo mismo que una notificación o mensaje pero tienen algunas propiedades en común y, aún más importante, en ocasiones es posible que necesitemos usar polimorfismo para simplificar algunas tareas.

Teniendo esto en cuenta, y sabiendo que no se permite herencia múltiple en GameMaker Studio, parece que lo más lógico es tener un objeto del que hereden todas estas interfaces de entrada y así controlar mejor las mismas con una llamada a la superclase en caso de ser necesario (tabla 3.7).

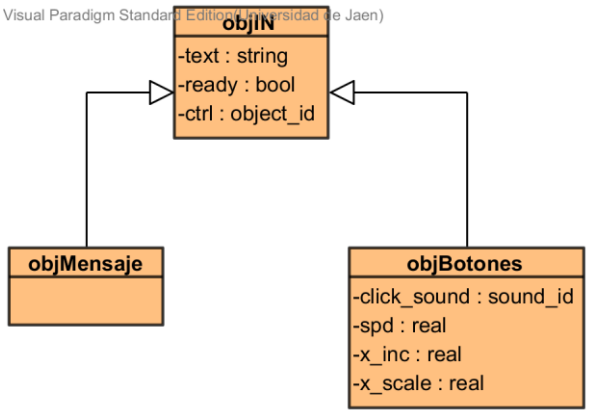


Figura 3.7. Relaciones entre los objetos de entrada

3.4.2.1.1. Implementación del botón

La interfaz se diseñó con varios estilos de botones que gozaban de diferentes formas, por lo que sería ideal tener varios objetos que hereden el comportamiento principal de objBotones, pero con ajuste de texto, visualización y efectos diferentes.

Pero además en el menú principal habrá un botón para activar y desactivar el sonido que, aunque tenga apariencia de botón pequeño, no necesita mostrar un efecto al pulsarse. Como no se ha estimado que haya más botones con esa función además del mencionado, éste se ha diseñado como en tal situación.

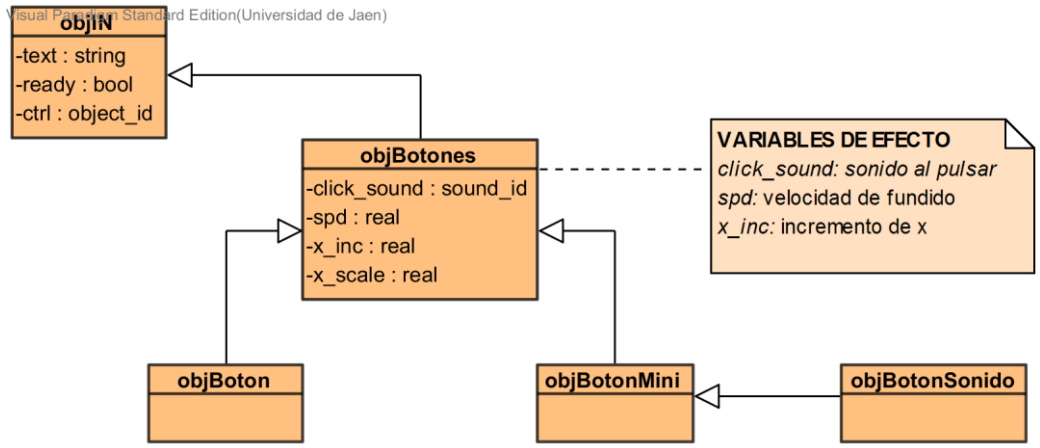


Figura 3.8. Relaciones entre los botones

Para ver la especificación completa de los objetos junto a todos sus eventos y la interacción entre los mismos, véase el *Anexo I*.

Los botones necesitan ajustar el texto que muestran dependiendo del tipo y hacerlo a tiempo real supone un coste computacional que, aunque los procesadores de la actualidad no tendrán problema en cargar, podemos evitar fácilmente si se calcula el ancho del texto una única vez.

Además, en cada botón hay que indicar el controlador que lo llamó al crearlo, un código numérico de acción y el texto que mostrará. Creando un script para instanciar los botones tendremos el código más limpio y flexible para modificaciones posteriores al hacer una interfaz única que crea los botones en lugar de hacer las asignaciones en los distintos controladores u objetos.

Se ha definido un script para cada tipo de botón que se encargará de crear la instancia, asignar los datos necesarios y llamar a la acción de calcular el ajuste del texto según el tipo de botón (especificación en *Anexo II*).

Para finalizar, estos botones deben llamar al controlador que los creó indicándole un código asignado para que este último se encargue de realizar las operaciones oportunas. Se ha decidido que los botones notifiquen al controlador después de realizar el efecto de fundido de salida por cuestiones meramente estéticas.

3.4.2.1.2. Implementación del controlador

Los controladores son los elementos principales del juego: se encargan del flujo de acciones del mismo. La idea es que, además de actuar como el patrón de diseño *Fachada*, dicho objeto actúe como *Singleton* ya que es un objeto con una definición muy particular y carece de sentido tener dos instancias diferentes (hay que tener en cuenta que *GameMaker Studio* hace difícil limitar que se crea un único objeto de su clase, la responsabilidad recae en nosotros).

Todos los controladores tienen cosas en común, como el ajuste de la vista a las diferentes resoluciones de los posibles dispositivos. Por ello, se ha considerado que lo ideal es una superclase de la que los controladores específicos hereden comportamientos y atributos.

Aun así, cada objeto podría ser uno independiente si no fuese por el evento de creación que realiza las operaciones comentadas anteriormente y por el polimorfismo: cada uno tiene sus operaciones muy concretas y poco relacionadas con los demás y por ello carece de sentido un diagrama de clases para indicar las relaciones entre controladores.

Los controladores mapean las acciones en el evento EventUser10. Cuando reciben el aviso de que una interfaz de entrada se ha activado por el usuario; estos mapean el código de acción en dicho evento y ejecutan las operaciones correspondientes.

Para ello se ha creado un script que haga como interfaz de modo que simplifique todo el proceso que suponen estas llamadas en *GameMaker Studio* y aporte flexibilidad a futuros cambios. Se trata de *mao_controller_call* (especificación en *Anexo II*).

La especificación del controlador y el caso concreto del encargado del menú principal, *objControllerInit*, se puede consultar en el *Anexo I*.

3.4.2.1.3. Implementación de notificaciones

Las notificaciones o mensajes se encargarán de mostrar información importante al usuario, pero se han considerado sin opciones, es decir, sólo se podrán aceptar y continuar con la ejecución.

Por ello no utilizan botones, sino que se aceptan al tocar cualquier lugar de la pantalla.

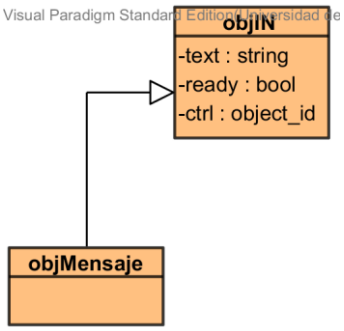


Figura 3.9. Especialización del objMensaje

Cuando una notificación aparece en la pantalla, el fondo debería llamar la atención menos que el propio mensaje y desactivar las interfaces de entrada que haya para evitar conflictos.

Al igual que los botones, los mensajes deben avisar al controlador mediante un código de acción mediante el script *mao_controller_call* (ver *Anexo II*).

No obstante, los eventos y por tanto, su forma de trabajar son diferentes (ver *Anexo I*).

3.4.2.1.4. Scripts para gestión de sonido

Crear una interfaz en lugar de utilizar de forma directa las funciones nativas de *GameMaker Studio* es una ventaja segura: en caso de necesitar modificar, por ejemplo, el motor de sonido de nuestro juego sólo tendremos que modificar la interfaz que se encarga de él.

Como ventaja añadida esta práctica limpia mucho el código evitando hacer y repetir ciertos elementos de control.

Por estos motivos se han creado scripts para gestionar la música que suena en el fondo, que se repetirá continuamente, y los efectos de sonido, que solo sonarán una vez); *mao_sound_play_background* y *mao_sound_play* respectivamente (ver especificación en *Anexo II*).

3.4.2.1.5. Implementación funcional del menú principal

Esta tarea se centra básicamente en aunar todo lo que se ha completado en tareas anteriores y dotar así de un ciclo de ejecución al menú principal.

Se han conectado los botones con el controlador y se han mapeado unas operaciones para los códigos de los mismos que, excepto el de cerrar y configuración de sonido que están operativos, nos notificarán de que la funcionalidad no está implementada aún.

Del mismo modo se han añadido detalles como burbujas que se crean aleatoriamente en el fondo para dotar de dinamismo al menú principal.

3.4.2.1.6. Implementación funcional del menú principal

Para hacer más simple una futura ampliación con otros idiomas se han creado los scripts *mao_texts_load* y *mao_texts_get*. El primero crea un mapa que relaciona una clave con las cadenas de texto y el segundo la recupera al pasarle dicha clave.

La idea es obtener los mapas desde un fichero diferente dependiendo del idioma detectado en el dispositivo, pero se considera fuera del alcance prototipo.

La función *mao_texts_load* sólo debe llamarse una vez u ocurrirán pérdidas de memoria (*memory leaks*) cada vez que se ejecute a no ser que el mapa sea destruido manualmente. Por esto nace la necesidad de crear una nueva escena que sea la primera del juego, y sólo se visite una vez, en la que un controlador cargue las configuraciones y evitar estos problemas.

De este modo se ha creado el *objControllerLoad* (ver *Anexo I*), con objetivo de cargar una vez y pasar a la siguiente escena sin que el usuario se dé cuenta.

3.4.2.2. Interfaz y sistema del servidor

Se pretende dotar al juego de un servicio en línea construido con *PHP* y *mySQL* que almacene los **usuarios, puntuaciones máximas, códigos promocionales** del juego además de generar estos según distintas **campañas**.

3.4.2.2.1. Diseño de la base de datos

Teniendo en cuenta las tres entidades que se han nombrado en la descripción anterior, se han diseñado las tablas para la base de datos como se indica en las tablas 3.9, 3.10 y 3.11.

Usuarios		
id 🔑	INT	Identificador único.
user_key	CHAR(40)	Clave privada para seguridad.
max_score	INT	Puntuación máxima.
register	TIMESTAMP	Fecha de registro.
last_connection	TIMESTAMP	Fecha de última conexión.
IP	CHAR(15)	Dirección IP.

Tabla 3.9. Diseño de la tabla “Usuarios”

Campanas		
id 🔑	INT	Identificador único.
mark	CHAR(32)	Nombre de la marca de aceite.
points	INT	Puntos a batir.
Init_date	DATE	Fecha de inicio.
end_date	DATE	Fecha de finalización.

Tabla 3.10. Diseño de la tabla “Campanas” referente a las campañas

Codigos		
id 🔑	INT	Identificador único.
id_user	INT	Usuario que lo ha obtenido.
consumed	BOOLEAN	Utilizado.
id_campana	INT	Campaña a la que pertenece.

Tabla 3.11. Diseño de la tabla “Codigos”

La base de datos se ha diseñado teniendo como objetivo ser simple y rápida y por ello contiene exclusivamente los campos que se han considerado necesarios para el prototipo.

3.4.2.2.2. Diseño del protocolo propio similar a REST

Sin lugar a dudas, un modelo de comunicación entre aplicaciones distribuidas basado en *REST* será flexible. Pero, ¿realmente necesitamos un protocolo así?

Si simplificamos éste y creamos un protocolo privado que esté basado en él, ahorraremos tiempo, al menos para el prototipo.

Además, los servicios del sistema deberán ser exclusivos para el juego excepto para gestionar los códigos promocionales, tarea de la que se debería

encargar otra aplicación diferente que utilice otro servicio del sistema que podía estar, por ejemplo, en el supermercado.

El protocolo será el siguiente:

1. Comunicación mediante **protocolo *HTTP***.
2. Utilización de métodos: ***GET* para obtener información y *POST* para crear y actualizar** (olvidamos el método *PUT*, al menos en el prototipo).
3. Como medida de **autenticación**, el paso de un mapa llamado “data” en formato JSON al solicitar un servicio que contenga un dato aleatorio en crudo y éste mismo cifrado con la clave privada del usuario o de la aplicación dependiendo del servicio usando el algoritmo sha1 (*ver Anexo II*).
4. **No se usan cabeceras** del protocolo, ni se piden datos directamente por consulta: **las variables se añaden al mapa data**.
5. Los datos son **cifrados con el algoritmo base64**.
6. El resultado de una petición es un mapa que contiene diferentes entradas, pero al menos las de control del servicio:
    a. gid: Identificador de control del juego. En este caso MAO.
    b. sec_alert: TRUE si ocurrió un problema o FALSE.
    c. err_code: Código del error (sólo si ocurre un problema).

Otra diferencia respecto a *REST* es que no sigue el estándar indicado para la llamada de los servicios y recursos, sino que llama a scripts específicos del servidor. Esto es cuestión de mapear los scripts en el servidor para que respondan al estilo de ruta, pero se ha considera fuera del ámbito del prototipo.

Para simplificar un poco todo este proceso desde el cliente se han creado varios scripts; *mao_services_load, mao_services_get, mao_services_method, mao_service_map, mao_service_map_init, mao_service_call* (ver *Anexo II*).

La definición de rutas y métodos de todos los servicios disponibles se carga al llamar a *mao_services_load*. Este script crea un mapa con la información y **debe llamarse una única vez en todo el juego** al igual que ocurría con

mao_texts_load. Por ello se llama en el *objControlledLoad* nombrado en el apartado anterior.

Con *mao_services_get* y *mao_services_method* recuperamos la *URI* y el método el servicio que indiquemos como argumento.

Finalmente usaremos *mao_services_call* para solicitar un servicio, pero este requiere que creemos el mapa *data* comentado en la definición del protocolo.

Para ello podemos utilizar *mao_service_map* y así generar el mapa con la información básica ya añadida. El script *mao_service_map_init* se usa para generar el mapa requerido en el registro del usuario; al no tener clave privada aún, necesitamos otro método diferente de control de seguridad. No deberíamos aceptar las peticiones sin filtro y registrar usuarios sin comprobaciones.

Por ello se usa la clave privada de la aplicación en este caso que solo conocen el servidor y la aplicación (constante *APP_KEY*).

Tras diseñar el protocolo y la base de datos se ha creado un script en *PHP* que se encarga de conectar con la base de datos para los servicios, *connect.php*.

3.4.3. Final de la iteración

El resultado es un menú principal dinámico con un fondo de burbujas que se mueven y cambian de forma junto a los botones y las notificaciones del juego.

Figura 3.10. Capturas de pantalla reales del menú del prototipo

Finalmente se han terminado todas las tareas de las dos historias de usuario previstas para la primera iteración. Pero la estimación de la velocidad ha sido errónea, pues se estimó en 17 y se han alcanzado los 21 puntos de historia por iteración.

Una vez refactorizado el código y solucionados todos los errores, es hora de continuar. Pero antes completamos el *Burndown Chart* añadiendo nuestro progreso real y la estimación con la nueva velocidad que se deberá ajustar al final de cada iteración:

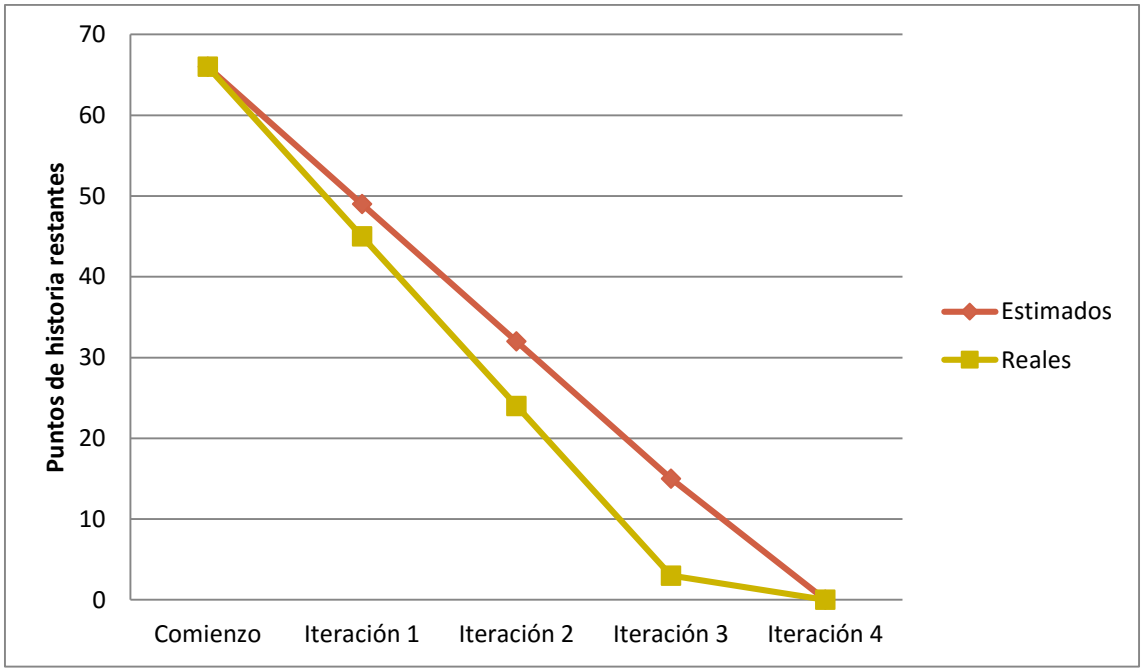


Figura 3.11. Burndown chart al terminar la primera iteración

3.5. Segunda iteración

En la segunda iteración se construirá el primer nivel al completo, de forma que será totalmente funcional.

3.5.1. Priorización de historias de usuario

En esta etapa se ha considerado que el ranking online de usuarios era menos importante para el desarrollo y se ha llevado al final del mismo, a la última iteración.

El resultado final de la lista priorizada se indica en la tabla 3.12.

Historia	PH
Menú principal	*13*
Interfaz y sistema del servidor	*8*
Registro de usuarios	8
Clasificación de la aceituna	5
Dificultad en J1	1
Fin de J1	1
Efecto manchado	2
Sistema de direcciones	3
Giro de engranajes	3
Batir masa	3
Clasificación de partidas	1
Promoción de productos	5
Ranking de puntuaciones	13

Tabla 3.12. Historias de usuario priorizadas para la segunda iteración

Anteriormente estimamos la velocidad como 17 puntos de historia, justo los puntos de historia que componen esta iteración.

Teniendo en cuenta que en la anterior conseguimos 21 puntos de historia de velocidad parece posible la realización de la iteración tal y como está planificada.

3.5.2. Diseño de la iteración

El primer paso vuelve a ser la división de las historias de usuario en tareas más pequeñas y concretas, pero las historias de usuario denominadas *Dificultad en J1* y *Fin de J1* no se han dividido al considerarse tareas en sí obteniendo el resultado de las tablas 3.13, 3.14 y 3.15.

Registro de usuarios
Aviso al iniciar sistema (Registro obligatorio)
Diseño de sistema de registro sin intervención del usuario
Servicio de registro en el servidor
Persistencia de datos de registro y configuración en local

Tabla 3.13. Tareas de la historia de usuario “Registro de usuarios”

Clasificación de la aceituna
Diseño gráfico de la escena: sprites y backgrounds
Sistema de puntuación
Mecánica del minijuego
Añadir sonidos
Transición entre escenas
Dificultad de J1
Fin de J1

Tabla 3.14. Tareas de la historia de usuario “Clasificación de la aceituna”

Efecto manchado
Gráfico de estrujado
Efecto en la pantalla
Añadir sonido al efecto

Tabla 3.15. División de tareas de “Efecto manchado”

3.5.2.1. Registro de usuarios

Una vez tenemos scripts para simplificar la tarea de la conexión al servidor como los *mao_service*, las tareas de esta historia de usuario deberían ser relativamente sencillas.

3.5.2.1.1. Aviso al iniciar sistema

Se ha modificado el controlador principal para que al cargar (evento *Create*) compruebe que haya un usuario registrado en el sistema y si no lance un *objMensaje* con el aviso para registrarlo.

3.5.2.1.2. Diseño del sistema de registro

Como el juego tiene como principal objetivo el marketing del aceite de oliva, **se ha pensado en regalar códigos promocionales dependiendo de ciertas condiciones para obtener botellas de aceite de regalo o descuentos** en las compras de ciertos supermercados.

Por eso el registro de usuarios es obligatorio, pero no significa que tenga que ser molesto para el usuario: un registro automático sin información personal en el que el usuario no pierda el tiempo rellenando datos ni confirmando cuentas.

Para ello se asigna un identificador de usuario proporcionado por el servidor y una clave privada para comprobar la identidad del usuario cuando use los servicios.

Ambas se guardan tanto en local como en el servidor, pero el usuario utilizará su clave privada para cifrar información y el servidor comprobará que el

cifrado es correcto, de esta forma no pone en riesgo las credenciales del usuario al no enviar la clave en las peticiones.

3.5.2.1.3. Sistema de registro en el servidor

Se ha creado un script en *PHP* llamado *user.php* que bajo el método *POST* y según el protocolo propio explicado en el *apartado 3.4.2.2* añade un nuevo usuario a la base de datos y devuelve el ID y la clave privada asignada para su uso futuro (ver *Anexo III*).

Para realizar la petición desde el cliente se han utilizado los scripts *mao_service_map_init* y *mao_service_call* (ver *Anexo II*). Una vez el servicio responde, si la respuesta es correcta se guarda el usuario y la clave en memoria para su posterior uso.

Finalmente, se ha creado el objeto *objViewConnection*. Éste se encarga de mostrar un gráfico animado mientras se realiza el servicio online para que el usuario tenga la impresión de que el sistema está trabajando. Una vez la aplicación tiene respuesta, esta instancia es eliminada.

Desde que se crea el *objViewConnection* hasta que se lanza la petición del servicio hay un retardo de un segundo para que se pueda ver dicho objeto en pantalla. Es por ello que las peticiones al servicio de los controladores se realizan en el evento *Alarm11* mientras que la espera sin respuesta está en *Alarm10* a la espera de un tiempo personalizado (constante *WAIT_TIME*).

3.5.2.1.4. Persistencia de datos

Para terminar esta historia de usuario, se han creado los scripts *mao_config_save* y *mao_config_load* para guardar y cargar la configuración (sonido, usuario y clave privada), hacerla persistente aunque el usuario cierre la aplicación.

3.5.2.2. Clasificación de la aceituna

El sistema se ha diseñado para aprovechar las pantallas táctiles capacitivas multi-toque de forma que el usuario pueda utilizar tantos dedos como le permita su dispositivo al mismo tiempo para clasificar la aceituna. No tiene por qué utilizar un único dedo.

Esto le proporciona un extra al reto del juego al poder sumar más puntos, pero se necesitará mucha más habilidad.

3.5.2.2.1. Sistema de puntuación

En primer lugar se ha creado una variable global denominada *total_score*. Estas variables tienen la particularidad de permanecer durante todo el ciclo de ejecución del juego una vez son creadas y son siempre accesibles (en *GameMaker Studio* las variables locales de los objetos son accesibles públicamente pero la instancia debe existir en la escena).

Se han diseñado unas cajas para mostrar la puntuación actual y el tiempo en caso de ser necesario en otro minijuego en el que sea importante.

Se ha decidido que el lugar donde el usuario deja las aceitunas sea importante, en otras palabras: si no las introduce bien en las cintas transportadoras perderá puntos. Además ganará puntos extra si las coge “al vuelo”, cuando están cayendo rápido.

3.5.2.2.2. Mecánica del minijuego

El juego dispone de un controlador, denominado *objController1*, que se encarga de crear las aceitunas así como de comprobar que se están clasificando correctamente, sumar puntos o terminar la partida (ver *Anexo I*).

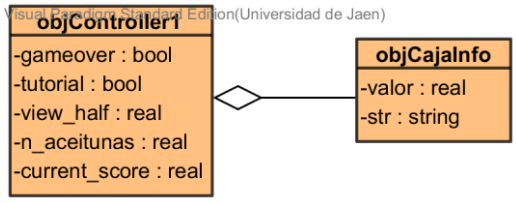


Figura 3.12. Relaciones del objController1

El *objController1* guarda una referencia a *objCajaInfo* durante todo su ciclo de vida para actualizar los datos que se muestran en la pantalla. En este caso hablamos de la caja de puntuación comentada en el apartado anterior. Para simplificar la creación y asignación de este objeto se ha creado el script *new_CajaInfo* que funciona de forma similar a la creación de botones y mensajes (ver *Anexo II*).

Al comienzo, el controlador fija su variable local de puntos a cero y crea el *objTimer*, que se encargará de realizarla cuenta atrás mediante una alarma y la creación de *objNumber* cuya única función es dibujar un *sprite*, en este caso números, con un efecto visual.

Una vez el *objTimer* llega a cero, avisa al controlador de que puede empezar la partida.

Las aceitunas son las encargadas de avisar al controlador cuando son soltadas por el usuario, de esta forma aprovechamos la arquitectura del sistema usando los eventos en lugar de que el controlador compruebe todas las instancias (interfaces activas).

3.5.2.2.3. Transición entre escenas

Se ha creado *objTransicion* para que indicándole una escena ejecute un fundido de la pantalla y al terminar cambie la escena. Pero para que al introducir la nueva no aparezca como un flash, también se ha creado *objIntro* que es creado por *objController* y hace el efecto contrario.

3.5.2.2.4. Dificultad de J1

En esta tarea se ha ajustado la velocidad con la que las aceitunas hacen aparición en la escena. Lo ideal es que comiencen apareciendo con poca frecuencia y que al usuario le dé tiempo a reaccionar la primera vez, pero conforme pasa el tiempo debe ajustarse y ser todo un reto.

Excepto para las tres primeras aceitunas, se ha utilizado la siguiente fórmula para calcular los pasos del juego que debe esperar para crear una nueva aceituna:

$$alarm[0] = \max((speed_{room} \times 3) - (n_{aceitunas} \times 7), 25)$$

Donde $speed_{room}$ = pasos por segundo de la escena (*60*)
$n_{aceitunas}$ = el número de aceitunas creadas

3.5.2.2.5. Fin de J1

El controlador comprobará cada vez que el usuario clasifique una aceituna si lo ha realizado correctamente así como cuando una nueva aceituna es creada no esté fuera de la cola. Si no es así, éste termina la partida llamando a su evento *UserDefined2*.

Una vez ha terminado la partida al comprobar que el usuario ha clasificado mal una aceituna o que la pila de aceitunas está llena, se muestra la animación de fin del nivel, se establecen los puntos obtenidos del nivel con una nueva variable global con forma de Array denominada *level_score* y se suman los puntos del controlador a la variable global *total_score*.

3.5.2.3. Efecto de manchado

Esta historia de usuario no tiene relevancia más allá de su apartado artístico.

3.5.3. Final de la iteración

El resultado es el primero de los minijuegos totalmente implementado junto al registro automático de usuarios que quedaría como se muestra la figura 3.13.

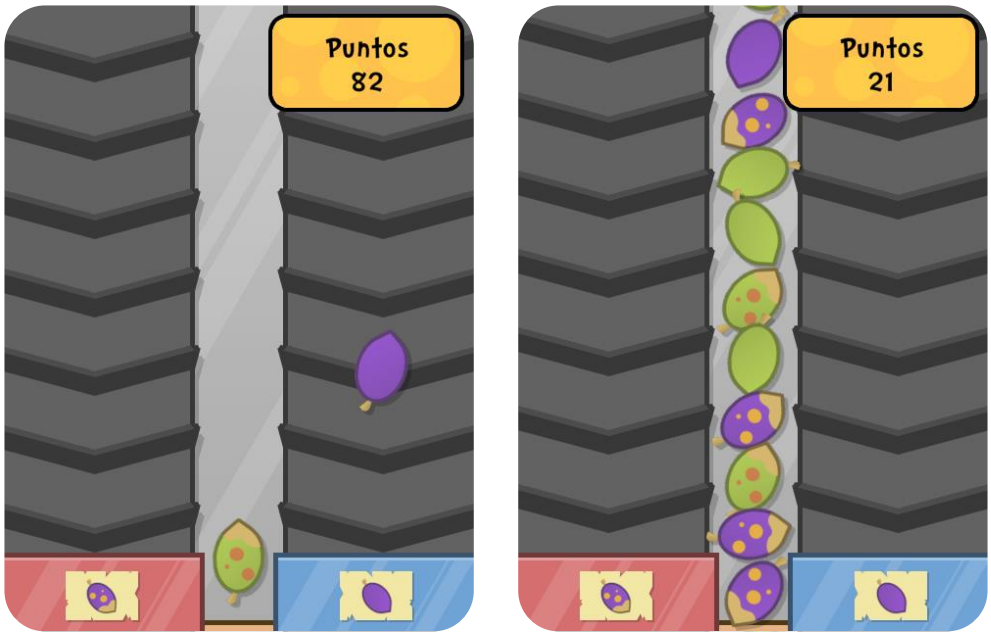


Figura 3.13. Capturas de pantalla del primer minijuego del prototipo

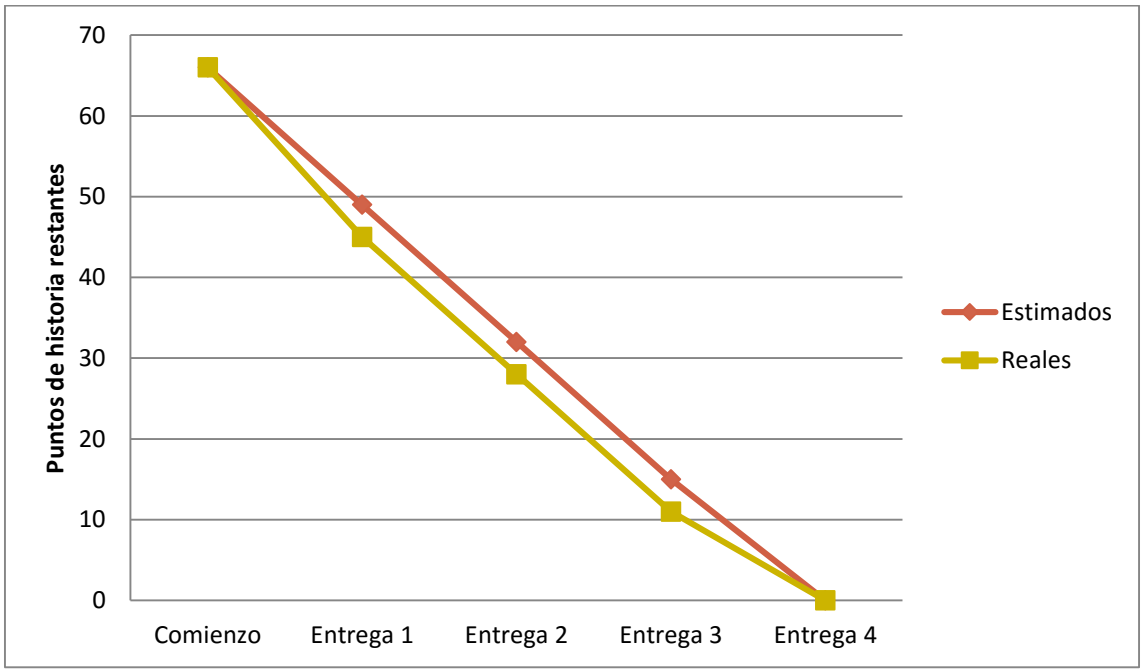


Figura 3.14. Burndown chart al terminar la segunda iteración

3.6. Tercera iteración

3.6.1. Priorización de historias de usuario

Para esta iteración se ha modificado levemente el orden de historias de usuario; únicamente se han intercambiado “*Giro de engranajes”* y “*Sistema de direcciones”* al considerarse necesaria la primera para realizar la segunda correctamente.

El resultado final de la lista priorizada cambiando las historias ya aparece en la tabla 3.16.

Historia	PH
Menú principal	*13*
Interfaz y sistema del servidor	*8*
Registro de usuario	*8*
Clasificación de la aceituna	*5*
Dificultad en J1	*1*
Fin de J1	*1*
Efecto manchado	2
Giro de engranajes	3
Sistema de direcciones	3
Batir masa	3
Clasificación de partidas	1
Promoción de productos	5
Ranking de puntuaciones	13

Tabla 3.16. Historias de usuario priorizadas para la tercera iteración

Al principio estimamos la velocidad como 17 puntos de historia. En esta iteración nos hemos fijado 15 ya que añadir la última hará un total de 28 y parece excesivo.

3.6.2. Diseño de la iteración

Tras hacer la división de tareas obtenemos el resultado de las tablas 3.17, 3.18, 3.19 y 3.20.

Giro de engranajes
Gráfico de los engranajes y trituración
Captación del giro del usuario
Aplicación de estado de giro
Añadir sonido al giro

Tabla 3.17. División de tareas de “Giro de engranajes”

Sistema de direcciones
Diseño gráfico del resto del nivel
Controlador del juego
Ajuste de puntuación y tiempo

Tabla 3.18. División de tareas de “Sistema de direcciones”

Batir masa
Gráfico de mancha en pantalla
Resto de gráficos de la escena
Shader para simular líquido en el fondo
Cálculo de agitación del dispositivo (acelerómetro)
Controlador del juego
Efecto de giro del mecanismo del centro según agitación
Añadir sonidos
Ajuste de puntuación y tiempo

Tabla 3.19. División de tareas de “Batir masa”

Clasificación de partidas
Equilibro de puntos en las fases
Gráficos de clasificación
Escena con clasificación

Tabla 3.20. División de tareas de “Clasificación de partidas”

3.6.2.1. Giro de engranajes

Los engranajes deben seguir el movimiento que hace el usuario con los dedos a su alrededor e imitarlo. Para ello se almacena la posición x e y de los dedos del usuario en la escena cuando están dentro de un cuadrado que envuelve al engranaje. Para comprobar esto se ha creado el script *mao_mouse_in* (ver *Anexo II*).

Este valor se actualiza en cada paso, pero antes se compara con la posición actual del dedo y si hay diferencia se calcula dos veces el arco tangente: usando el vector de dirección el que va desde el centro del engranaje hasta el dedo del jugador actual y desde el centro al previo.

La diferencia de los dos resultados es el ángulo que se ha movido el dedo y debe sumarse a la rotación del mismo además de almacenarse para que el controlador pueda leer el estado del engranaje.

3.6.2.2. Sistema de direcciones

Una vez se han diseñado los elementos gráficos que componen el nivel es hora de completar el resto de tareas de la mecánica del segundo minijuego.

3.6.2.2.1. Controlador del juego

El controlador de juego del segundo nivel, *objController2*, actúa de forma parecida al del primero (cuenta atrás, cajas con puntos, etc.), pero además debe crear otra caja de visualización de datos que mostrará el tiempo restante para acabar dicho nivel (*ver Anexo I*).

Además, posee unas mecánicas totalmente diferentes así como una interfaz de entrada pasiva; en esta ocasión el rodillo o engranaje no avisa al controlador cuando gira, sino que es este último el que se encarga de chequear el estado de sus interfaces.

Esto se debe a que en *GameMaker Studio* la concurrencia real de eventos no existe. ¿Qué ocurre si los dos engranajes avisan al mismo tiempo al controlador de que están girando en cierta dirección? Al ejecutarse de forma secuencial el controlador no se dará cuenta nunca.

El controlador crea una aceituna y no vuelve a crear otra hasta que ésta se ha molido. Al crearla define una dirección para cada engranaje que no cambiará hasta la próxima y asigna un valor para cada uno de ellos que decrecerá con el giro correcto de los mismos.

La cantidad de decrecimiento se basa directamente en la menor velocidad angular en radianes de los dos engranajes.

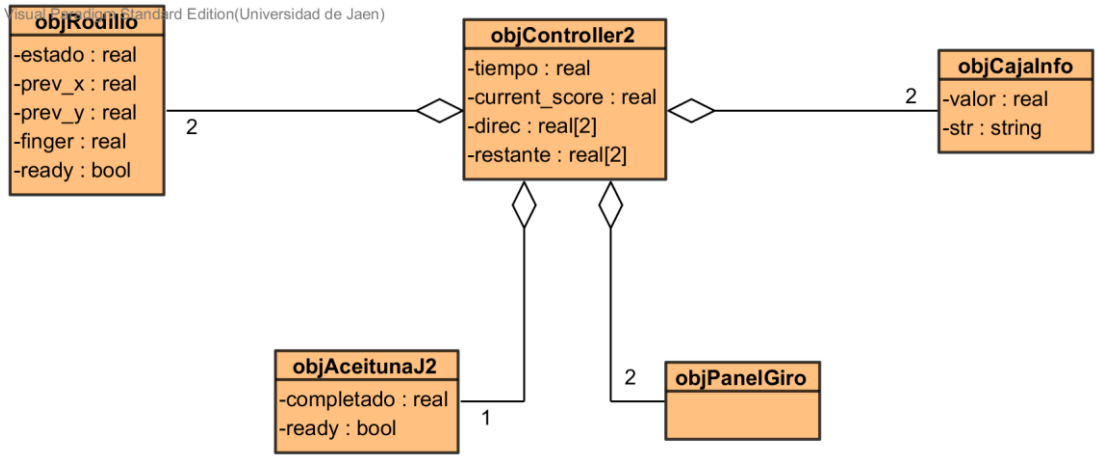


Figura 3.15. Relaciones del objController2

El controlador también se encarga de actualizar el factor de erosión de la aceituna que ha creado así como de instanciar el efecto gráfico del líquido de la aceituna correspondiente a la molienda.

La aceituna que crea este controlador contiene un factor entre cero y uno que indica el porcentaje de completitud de la misma. Esto repercute directamente en la representación gráfica de la aceituna ya que a menor factor, más hueso estará a la vista.

Una vez se acaba el tiempo del controlador, éste finaliza el minijuego como ocurría en el primero de ellos; el efecto en pantalla de terminado.

3.6.2.2.2. Ajuste de puntuación y tiempo

Una vez probado y verificado el funcionamiento del segundo nivel, se han hecho varias pruebas para encontrar el tiempo óptimo que permita que la puntuación varíe entre usuarios sin llegar a hacerse largo.

3.6.2.3. Batir masa

Esta historia de usuario es la que resume al tercer y último minijuego de los que se han diseñado para este prototipo. Utiliza el acelerómetro del dispositivo para calcular los puntos según el nivel de agitación durante unos segundos.

Dejando de lado las tareas puramente artísticas de esta historia de usuario, el proceso de ejecución de las diferentes tareas en la que se ha descompuesto es el descrito en los apartados siguientes.

3.6.2.3.1. Shader para simular líquido

La planificación original era utilizar un *shader* para simular un efecto de distorsión utilizando *GLSL ES*, pero tras hacer varias pruebas el juego bajaba la velocidad de ejecución justamente a la mitad mientras que los FPS no bajaban.

No se consiguió solucionar y finalmente se eliminó del prototipo. A cambio, se ha creado un efecto usando *sprites* que simula la masa de la aceituna en movimiento y que gira más rápido según agitamos el dispositivo. De este efecto se encarga el objeto denominado *objWave*.

3.6.2.3.2. Cálculo de la agitación del dispositivo

Al comienzo, se diseñó e implementó un sistema que sumase la cantidad total de movimiento sumando la diferencia de los valores del acelerómetro para cada eje y luego se le aplicaba una operación matemática para calcular los puntos.

Pero resultó que dependía en exceso de cada dispositivo; no todos los acelerómetros tienen la misma precisión. De modo que para calcular la agitación del dispositivo se ha decidido contar el número de veces que el valor máximo de la diferencia de valor de los tres sensores supera la barrera de cierto valor.

Se ha experimentado con los valores que reportan los sensores del acelerómetro de dos dispositivos diferentes hasta que se ha encontrado un buen valor de respuesta en ambos que ha resultado ser 1,7.

3.6.2.3.3. Controlador del juego

Para el tercer nivel hemos obtenido el controlador más simple de los tres encargados de las mecánicas. El *objController3* sólo debe encargarse de contar la agitación del dispositivo como se ha comentado en el apartado anterior y de terminar el minijuego cuando el tiempo se haya terminado (ver *Anexo I*).

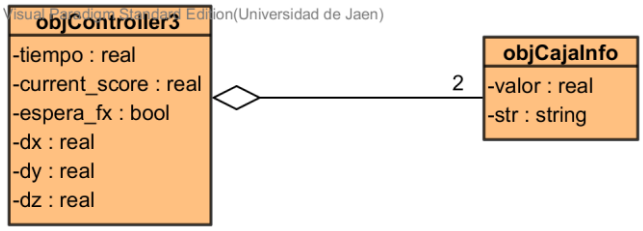


Figura 3.16. Relaciones del objCrontroller3

Este controlador guarda referencias a dos *objCajaInfo*, uno para el tiempo restante y otro para la puntuación de forma similar al del nivel dos.

3.6.2.4. Clasificación de partidas

La última historia de usuario tiene como objetivo clasificar la partida del usuario en los diferentes tipos de aceite de oliva. Pero para ello es necesario que la puntuación sea equilibrada entre los diferentes niveles.

3.6.2.4.1. Equilibrio de puntos en las fases

Para equilibrar las puntuaciones que se obtienen en las diferentes fases se ha tomado como referencia mil puntos en cada uno para una jugada normal-buena.

Se han probado los diferentes niveles para comprobar el número de aceitunas y veces que se agitaba el dispositivo varias veces y se han adaptado los puntos de cada uno de estas interacciones dependiendo de esto.

Para la clasificación final se han establecido los rangos de la tabla 3.21.

Puntos	Clasificación
Menor que 1150	Aceite refinado
Entre 1150 y 2149	Aceite mezcla
Entre 2150 y 3149	Aceite virgen
3150 o más	Aceite virgen extra

Tabla 3.21. Clasificación del aceite por puntos

Teniendo esto en cuenta, se han diseñado los gráficos necesarios y se ha pasado a la siguiente tarea.

3.6.2.4.2. Escena de clasificación

En primer lugar se ha implementado el *objBarraResultados*, un objeto que simplemente muestra dos valores en un pequeño cajón horizontal; a la izquierda muestra un texto y a la derecha un valor. Este objeto puede crearse de forma más sencilla usando el script *new_BarraResultados* que funciona similar a la creación de botones y mensajes (ver *Anexo II*).

Para esta escena se ha creado un nuevo controlador, el *objControllerResultados* (ver *Anexo I*). Éste crea un resumen de nuestra puntuación en los distintos niveles del juego junto a la puntuación final para, en un segundo paso, mostrar la clasificación de nuestra partida actual según se ha comentado en el apartado anterior.

Una vez el usuario toca la pantalla, el controlador llama al servicio de sincronización de puntos con las funciones *mao_service_call* al igual que el resto de objetos que usan servicios.

3.6.2.5. Promoción de productos

Desgraciadamente, debido a los problemas que han surgido en las historias de usuario previas y a la mala estimación de algunas de ellas, no ha quedado tiempo suficiente para esta historia de usuario. Esto quiere decir que pasará a la siguiente iteración.

3.6.3. Final de la iteración

La historia de usuario “*Promoción de productos*” no se ha completado completamente en esta iteración, de modo que pasa por completo a la siguiente.

Esto significa que en esta iteración se ha conseguido sólo una velocidad de 10 puntos de historia, debido a que la estimación de algunas de ellas era incorrecta y a los errores que han surgido.

Figura 3.17. Capturas del prototipo tras la tercera iteración

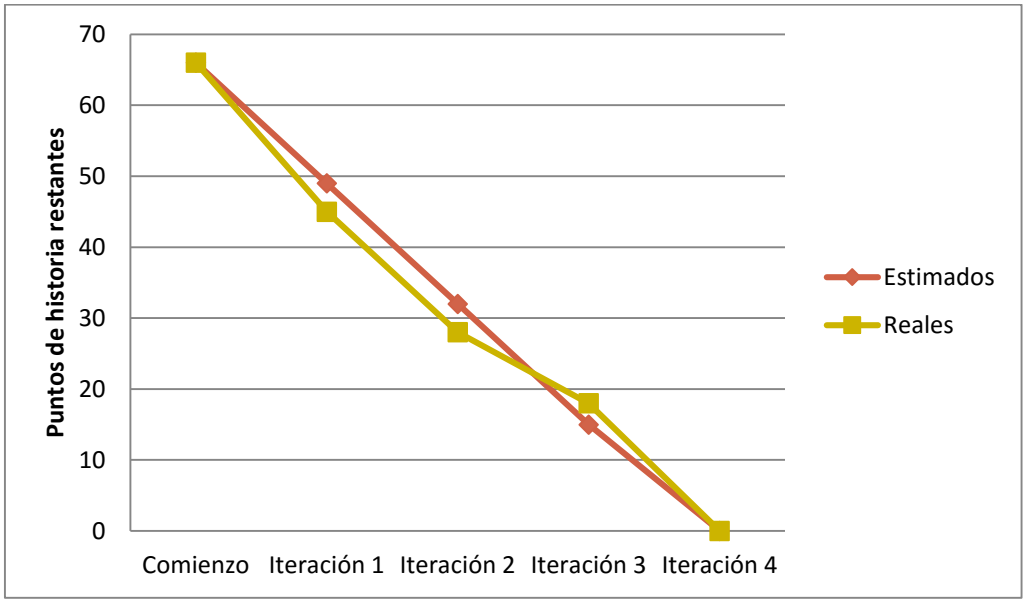


Gráfico 3.18. Burndown Chart al terminar la tercera iteración

En el *Burndown chart* se puede apreciar que vamos con **retraso para la siguiente iteración**, de modo que tendremos que darnos prisa para completar el prototipo a tiempo.

3.7. Cuarta iteración

Antes de comenzar con esta iteración se ha decidido distribuir una versión del prototipo en pruebas a modo de beta cerrada a varios usuarios para comprobar, entre otras cosas, la usabilidad del producto y los elementos a mejorar aunque estos últimos serán de cara a una versión completa.

3.7.1. Pruebas con usuarios reales

En primer lugar, se han subido los scripts de los servicios a un servidor propio y se ha creado la base de datos para comprobar por completo el prototipo. Tras esto, se ha generado un archivo APK para Android y se ha distribuido a siete usuarios que no estaban juntos para recibir sus opiniones sobre el juego.

Estas opiniones se han recogido en formato libre; en lugar de repartir cuestionarios, se han recogido en crudo. Se ha decidido hacerse así porque este método da mucha más información que un cuestionario, incluso elementos que no teníamos en mente y que con un cuestionario tampoco obtendríamos. La lista de usuarios es pequeña y pueden sacarse conclusiones fácilmente de las opiniones en este formato, sería más difícil con un gran volumen de usuarios para las pruebas.

Cinco de los siete usuarios que hicieron la prueba son o están estudiando para ser ingenieros informáticos. Además, los usuarios de las tablas 3.27 y 3.28 jugaron mientras se observaba su comportamiento.

El resultado del estudio, incluyendo el perfil de cada usuario que ha opinado, se indica en las tablas 3.22, 3.23, 3.24, 3.25, 3.26, 3.27 y 3.28.

Usuario 1	
Edad	29
Sexo	Mujer
Dedicación	Estudiante Ing. Informática
Opinión	“No sé qué hay que hacer en el nivel uno, en el dos me ha costado un poco y el tercero no sabía si batir el móvil o la rueda del centro de la pantalla y la he liado. He conseguido una catástrofe de puntos.”

Tabla 3.22. Opinión del usuario 1

Usuario 2	
Edad	24
Sexo	Hombre
Dedicación	Estudiante Ing. Informática
Opinión	“La interfaz me gusta mucho, pero debería haber un tutorial antes de cada minijuego. He fallado en todos menos en el segundo. También me gustaría que se pudiese elegir el minijuego en lugar de tener que jugar a todos.”

Tabla 3.23. Opinión del usuario 2

Usuario 3	
Edad	24
Sexo	Hombre
Dedicación	Ingeniero informático
Opinión	“Ha sido divertido pero no le veo mucho sentido a la tercera fase. Entiendo que no se quieren repetir mecánicas anteriores, pero pierdes la vista de la pantalla. ¿Qué ocurre si juegas sin sonido? ¿Cómo sabes cuándo parar? El icono es un poco raro.”

Tabla 3.24. Opinión del usuario 3

Usuario 4	
Edad	25
Sexo	Hombre
Dedicación	Estudiante Ing. Informática
Opinión	“Me ha gustado mucho, no puedo parar de jugar. Me parece un juego muy original. Estaría bien poder reiniciar el minijuego antes de pasar al siguiente si se hace mal y así poder conseguir más puntos.”

Tabla 3.25. Opinión del usuario 4

Usuario 5	
Edad	30
Sexo	Hombre
Dedicación	Estudiante Ing. Informática
Opinión	“Realmente es muy divertido, aunque frustra un poco intentar superar la puntuación. Sobre todo en el primer nivel en el que es muy difícil coger la aceituna cuando el ritmo de juego se acelera.”

Tabla 3.26. Opinión del usuario 5

Usuario 6	
Edad	16
Sexo	Mujer
Dedicación	Estudiante
Opinión	“Se necesita demasiada precisión para coger la aceituna en el primer nivel. Al principio me ha costado bastante entender qué había que hacer, sobre todo en el primero de los niveles. La interfaz es muy colorida y me gusta. Es muy divertido una vez que lo entiendes, pero creo que el primer nivel tendría que arreglarse un poco.”

Tabla 3.27. Opinión del usuario 6

Usuario 7	
Edad	27
Sexo	Hombre
Dedicación	Ingeniero de minas
Opinión	“Me ha costado mucho entender qué había que hacer en el primer nivel. Sabía clasificar las aceitunas negras porque está marcado en la zona inferior de la pantalla, el problema que he tenido es no saber dónde debía dejar las de color verde.”

Tabla 3.28. Opinión del usuario 7

Uno de los objetivos en el diseño del juego era no molestar al usuario con tutoriales que le hagan perder tiempo, pero en las opiniones se encontraron muchos problemas con esto: la mayoría de usuarios reportaron no entender la misión de cada minijuego exceptuando el segundo de ellos y especialmente en el primero.

De este último también se opinó que la precisión era demasiado alta y se hacía difícil de jugar, sobretodo en pantallas pequeñas.

También se reportó la difícil forma de jugar al tercer minijuego si el sonido del dispositivo se desactiva: no sabemos cuándo parar de agitar el dispositivo al no ver bien la pantalla.

Todos los demás elementos que se obtienen de las conclusiones, como el botón para reiniciar el nivel, quedan fuera de este prototipo pero deberían tenerse en cuenta para una versión final del producto.

3.7.2. Nuevas historias de usuario

Tras las opiniones recibidas de los usuarios que probaron la beta, nacen nuevas historias de usuario. Pero únicamente basadas en mejorar la usabilidad y experiencia de juego del prototipo.

La primera de ellas es relativa al no entendimiento de los minijuegos:

Historia	**Mejora de usabilidad**
Como	Jugador
Quiero	Recibir más información sobre la mecánica de los juegos
Para	Entender su funcionamiento

Esta historia de usuario se ha estimado con un coste de tres puntos de historia.

Finalmente, otro elemento reportado fue la excesiva alta precisión al coger la aceituna del primer minijuego. Cuando la velocidad aumenta, la jugabilidad se convierte en mala. De este modo obtenemos otra nueva historia de usuario para esta última iteración:

Historia	**Ajuste de precisión en J1**
Como	Jugador
Quiero	Una mejor respuesta al coger la aceituna
Para	Jugar cómodamente

Esta última se ha estimado con un coste de un punto de historia.

Por lo tanto, lo primero que vamos a hacer es actualizar el *Burndown chart* y así ver el estado actual del proyecto de un vistazo rápido. El resultado del nuevo gráfico se indica en la figura 3.19.

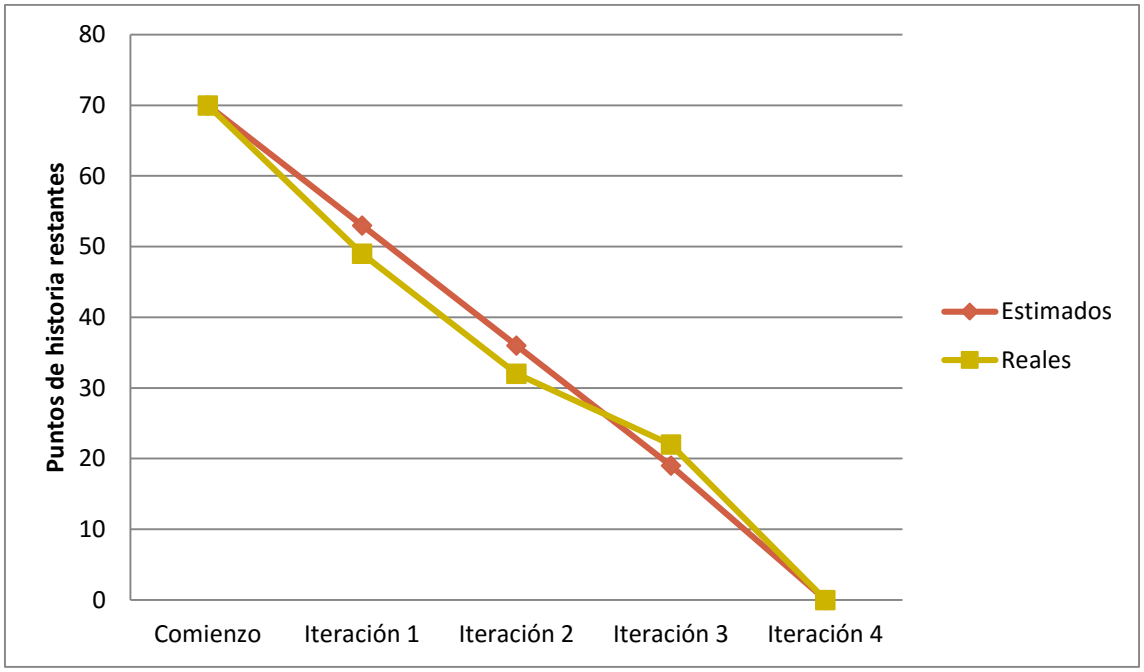


Gráfico 3.19. Burndown chart al comenzar la cuarta iteración

3.7.3. Priorización de historias de usuario

Para esta iteración se han añadido las dos historias de usuario nuevas previamente comentadas pero se les ha asignado más prioridad que a la promoción de productos y al ranking. Este último sería el menos prioritario de todos ellos al tener más importancia la promoción ya que es el objetivo principal de este prototipo.

El resultado final de la lista priorizada cambiado las historias se indica en la tabla 3.29.

Historia	PH
Menú principal	*13*
Interfaz y sistema del servidor	*8*
Registro de usuario	*8*
Clasificación de la aceituna	*5*
Dificultad en J1	*1*
Fin de J1	*1*
Efecto manchado	2
Giro de engranajes	3
Sistema de direcciones	3
Batir masa	3
Clasificación de partidas	1
Mejora de usabilidad	3
Ajuste de precisión J1	1
Promoción de productos	5
Ranking de puntuaciones	13

Tabla 3.29. Historias de usuario priorizadas para la tercera iteración

Restan sólo 22 puntos de historia para terminar completamente el prototipo y cumplir con la planificación. Si la estimación es correcta, en esta iteración debemos darnos prisa o, en el peor de los casos, la historia menos importante, Ranking de usuarios, quedaría fuera.

3.7.4. Diseño de la iteración

Tras hacer la división de tareas obtenemos el resultado de las tablas siguientes.

Mejora de usabilidad
Gráfico de flecha para indicar el arrastre de la aceituna
Añadir el efecto a las tres primeras en el nivel 1
Aceitunas de ejemplo en las cintas cada cierto tiempo en nivel 1
Gráfico para agitar el dispositivo
Hacer parpadear en el nivel tres
Cambiar pantalla a rojo cuando el nivel termine

Tabla 3.30. División de tareas de “Mejoras de usabilidad”

Ajuste de precisión J1
Usar caja ampliada en lugar de máscara de colisión
Limitar el número de aceitunas por dedo

Tabla 3.31. División de tareas de “Ajuste de precisión J1”

Promoción de productos
Servicio de actualización de puntos y comprobación de ofertas
Muestra de los códigos de descuento obtenidos
Servicio de recuperación de códigos
Implementación de la lista de códigos

Tabla 3.32. División de tareas de “Promoción de productos”

Ranking de puntuaciones
Diseño de la interfaz gráfica
Servicio de recuperación de puntos
Implementación del ranking

Tabla 3.33. División de tareas de “Promoción de productos”

3.7.4.1. Mejora de usabilidad

Especialmente en el primer minijuego se han detectado la mayoría de los problemas de usabilidad del prototipo. Por ello se ha pensado en añadir a la cinta transportadora aceitunas que sirvan como ejemplo cada cierto tiempo.

Además de esto, las tres primeras aceitunas mostrarían una flecha indicando la dirección hacia la que las debemos arrastrar. Medidas para no hacer una explicación que interrumpa al jugador.

Del mismo modo se han diseñado los gráficos y añadido los efectos de la lista de tareas al nivel uno y tres del prototipo. Es este último el fondo de la pantalla se torna en rojo para llamar la atención y que el jugador sepa que el juego ha terminado mientras agita su dispositivo.

3.7.4.2. Ajuste de precisión J1

El segundo problema detectado viene del diferente tamaño de las pantallas de los dispositivos móviles: no es igualmente fácil coger una aceituna en una pantalla de 5 pulgadas que en una de 10.

3.7.2.2.1. Usar caja ampliada en lugar de máscara

Una posible solución a este problema es ampliar la máscara de colisión del *objAceitunaJ1*, pero eso acarrearía problemas al crear la lista de aceitunas, pues se detectaría la colisión antes de que visualmente ocurra.

Por ello se ha optado por utilizar el ya creado script *mao_mouse_in* que simulará el test de colisión de una caja envolvente tipo AABB con el ratón/dedo

del usuario. Además otra ventaja es que la máscara de colisión gira con el objeto mientras que nuestro test no lo hace.

Lógicamente se ha establecido un tamaño mayor que el de la máscara de colisión del *sprite*, pero esto también crea un nuevo problema: Coger varias aceitunas de una vez con el mismo dedo.

3.7.4.2.2. Limitar el número de aceitunas por dedo

Para solucionar este problema se ha creado un *array* de cinco elementos booleanos que indica si un dedo está en uso: *objAceitunaJ1* comprueba que el dedo que la está tocando no está en uso en el momento de cogerla. Si lo está lo ignora.

3.7.4.3. Promoción de productos

El objetivo principal del juego es el marketing, elemento que se ha decidido introducir en forma de código promocional por el que se obtienen descuentos en la compra de aceite de oliva de marcas que pueden personalizarse.

Cuando el usuario cumpla ciertas condiciones obtendrá un código dependiendo de cada campaña. En el prototipo sólo hablamos de superar ciertas cantidades de puntos.

Se ha diseñado de forma que puedan promocionarse varias marcas o productos, aunque puede modificarse fácilmente.

3.7.4.3.1. Servicio de puntos y comprobación de ofertas

Se ha creado el script en *PHP* llamado *score.php* que utilizando el protocolo explicado en el *apartado 3.4.2.2* de este documento guarda la mayor de las puntuaciones enviadas por el usuario (ver *Anexo III*).

Además, comprueba las campañas activas de las que el usuario no tiene código promocional y si cumple los requisitos crea uno nuevo para dichas campañas. Estos códigos recién creados son devueltos por el servicio.

El formato del código puede y debe modificarse sin problemas en caso de construcción de una versión para producción. En el prototipo el servicio los crea con un identificador autoincrementado, al igual que el resto de entidades de la base de datos, pero almacena también el usuario al que pertenece. Este código es devuelto en formato:

$$codigo = \mathrm{ID}_{\mathrm{user}} \times 10000000 + \mathrm{ID}_{\mathrm{código}}$$

Donde $\mathrm{ID}_{\mathrm{user}}$ = Identificador numérico del usuario
$\mathrm{ID}_{\mathrm{códigp}}$ = Identificador numérico del código

El motivo de esta codificación es no usar el ID del código en bruto ya que al ser un índice autoincrementado/autogenerado es fácilmente calculable y cualquier usuario podría usar un código que no le pertenece de forma sencilla. De esta forma es necesario saber tanto el ID del código como el usuario al que pertenece: muchas menos posibilidades de acertar con fuerza bruta.

Aun así, para una versión en producción debería estudiarse un cifrado de forma para que no sea tan legible por el usuario y sea una representación única.

El cliente no guarda los códigos en local. Se ha decidido así ya que la carga de datos es muy pequeña y los códigos se usados desaparecerán de la lista de códigos obtenidos.

3.7.4.3.2. Muestra de los códigos de descuento obtenidos

Si el servicio anterior devuelve algún código, el *objControllerResultados* nos mostrará un *objMensaje* para cada uno notificándonos de que hemos obtenido un descuento para la marca concreta y el respectivo código.

Una vez ha mostrado todo los que hemos obtenido, el controlador hace volver a la pantalla principal del juego.

3.7.4.3.3. Servicio de recuperación de códigos

Debe existir alguna forma de recuperar los códigos que hemos obtenido para poder usarlos. Para ello existe en el menú principal el botón Promos.

Al pulsar este botón se llama al servicio del que se encarga el script *PHP codes.php.* Básicamente éste busca en la base de datos los códigos activos (no usados) del usuario y los devuelve (ver *Anexo III*).

3.7.4.3.4. Implementación de la lista de códigos

Al pulsar el botón Promos del menú principal, se instancia un controlador llamado *objIndiceCodigos* que solicita el servicio anterior de recuperación de códigos. Una vez obtenido, los muestra en una lista usando *objBarraResultados* y crea un botón para volver al menú principal (ver *Anexo I*).

Cuando el *objControllerInit*, controlador encargado de la escena del menú, instancia al encargado de la lista de códigos, hace no visible el objeto que representa el logo del juego y elimina todos sus botones hasta que éste termine.

3.7.4.4. Ranking de puntuaciones

La última historia de usuario que forma parte del prototipo es el ranking de puntuaciones. La idea original era hacer la clásica lista con los mejores puntos, pero, además de que el prototipo no guarda ningún nombre de usuario, ¿qué sentido tiene ver usuarios que no conoces en el ranking? Además, cuando el juego comenzase a llenarse de usuarios y por tanto de puntuaciones habrá muchos similares y éste ofrecerá poca información/utilidad.

Por este motivo se ha decidido construir un ranking diferente al estándar. En lugar de usar una lista de puntuaciones sólo se recuperará la máxima puntuación global de los usuarios y la nuestra propia como se explica a continuación.

3.7.4.4.1. Servicio de recuperación de puntos

Para recuperar la mejor puntuación global de los usuarios y la personal se ha creado el script *PHP ranking.php*. La única función de este servicio es la comentada (ver *Anexo III*).

3.7.4.4.2. Implementación del ranking

En el cliente se instancia a *objIndiceRanking*, un controlador muy similar al de recuperación de códigos pero que en su lugar llama al servicio de recuperación de puntos.

Cuando éste obtiene la respuesta del servicio instancia un *objBotella*, objeto con una representación puramente gráfica: dibuja una botella en pantalla y la rellena de aceite con un factor que oscila entre cero y uno, donde cero es vacía y uno, llena.

Esta botella representa cuán lejos está el usuario de la mejor puntuación registrada. Si la botella está llena, el usuario es el que tiene la mejor puntuación. Se podría decir que el objetivo del usuario es llenar esta botella.

Para hacerlo más entendible y preciso también se muestra un indicador marcando el máximo global, que independientemente del valor será la botella llena, y la posición del usuario.

Además se crean dos *objBarraResultados* para mostrar el valor numérico del máximo global y el del usuario.

3.7.5. Final de la iteración

Finalmente se ha conseguido completar todas las historias de usuario. Esto significa que el prototipo está acabado por completo.

Figura 3.20. Capturas del prototipo al final de la última iteración

Para terminar, una vez ha terminado el prototipo el *Burndown Chart* ha quedado de la siguiente forma:

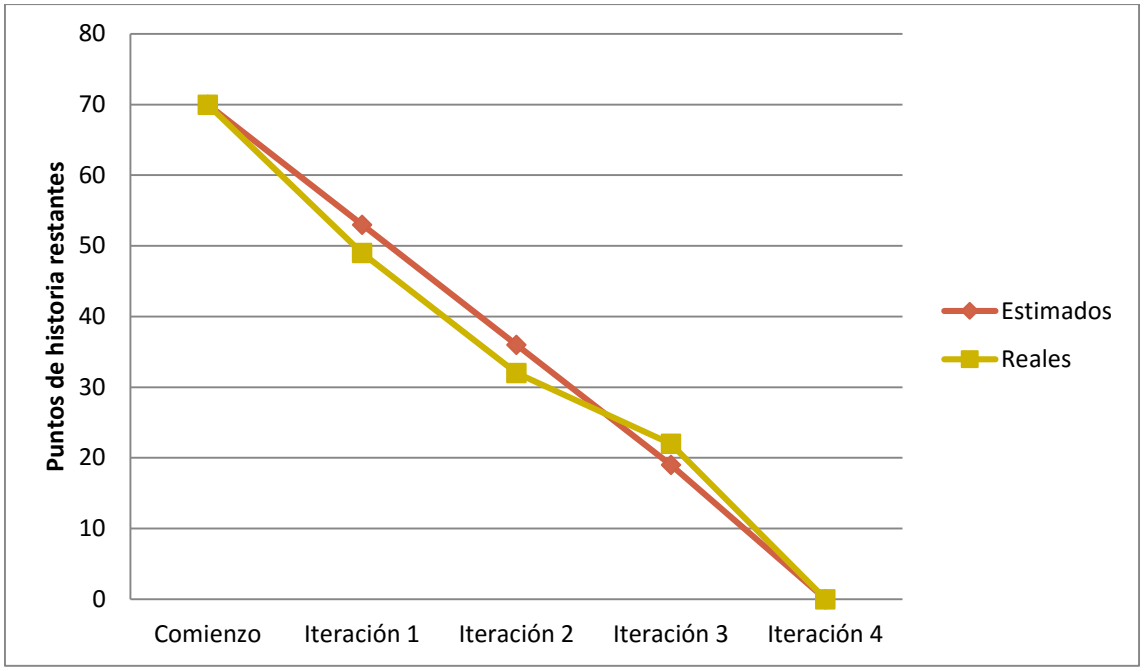


Figura 3.21. Burndown Chart al final de la construcción del prototipo

En él se puede apreciar que conseguimos cierta ventaja en la realización de las tareas al principio pero ocurrió un problema en la tercera iteración y a partir de ésta fuimos con retraso.

Capítulo IV PRESUPUESTO

A continuación se resume el presupuesto calculado para la construcción del videojuego finalizado en lugar del prototipo contando su desarrollo desde cero teniendo en cuenta que se ha desarrollado aproximadamente la mitad.

Para ello vamos a tener en cuenta que el desarrollo del prototipo para una persona se estimó en 31 días. Debido a que **el presupuesto está orientado al producto final**, como las tareas se dividen entre tres personas, vamos a considerar que **se necesitan aproximadamente 20 días de trabajo**, aunque se necesitará personal con experiencia avanzada.

4.1. Recursos humanos

Para realizar este cálculo se van a contratar varios perfiles de personas necesarias para el desarrollo de videojuegos.

Presupuesto del personal			
Elemento	Coste/Día	Días	Total
Artista	50€	15	750€
Game designer/ Big boss/ programador	75€	20	1.500€
Programador	50€	20	1.000€
Total			**3.250€**

Tabla 4.1. Presupuesto en recursos humanos

El *game designer* seguirá trabajando durante todo el proyecto como programador y jefe encargado de coordinar al resto del equipo. Sin embargo el artista no estará en la etapa final de cada iteración.

El salario por horas del grafista se ha estimado como 1.500€ al mes, el salario medio mensual de un diseñador gráfico en España[15].

El salario del programador para plataformas móviles se ha calculado en base a diferentes ofertas de trabajo publicadas en la red[16], mientras que el del game designer, que además trabajará en otros ámbitos del proyecto, se ha incrementado un 50% debido a la responsabilidad.

El problema principal para calcular el salario del game designer es que el mundo de los videojuegos en España es un mercado que aún está creciendo y no hay datos fiables de salarios de personal más especializado. Los estudios incluyen a otros países y en España el salario es menor como demuestra la diferencia de presupuestos [17].

4.2. Licencias

Para la realización del videojuego utilizando GameMaker Studio hay que comprar las licencias de exportación a plataformas móviles[18]. También hay que considerar comprar la música[19] para que sea exclusiva del juego.

Presupuesto de licencias			
Elemento	Coste	Cantidad	Total
GM Professional	160€	1	160€
Android Export	300€	1	300€
iOS Export	300€	1	300€
Música	100€	2	200€
Total			**960€**

Tabla 4.2. Presupuesto en licencias

Se han añadido solo dos canciones al presupuesto ya que son las mínimas que debería tener: pantalla de título y jugar nivel.

4.3. Presupuesto final

Para terminar, sumamos ambos y obtendremos el presupuesto final de la aplicación para producción.

Presupuesto final	
Personal	3.250€
Licencias	960€
Total	**4.210€**

Tabla 4.3. Presupuesto total

En este presupuesto no se incluye el coste de implantación del sistema de canjeado de códigos promocionales en los diferentes establecimientos para que el usuario obtenga un código de descuento en su compra de productos.

Capítulo V CONCLUSIONES

Gracias a los motores de videojuegos de la actualidad, como *GameMaker Studio* o *Unity3D*, el desarrollo de videojuegos es una tarea relativamente sencilla y barata. No se necesitan grandes presupuestos para hacer videojuegos clasificados como “*serious games*”.

En especial *GameMaker Studio* ofrece una velocidad de desarrollo extremadamente alta para juegos en dos dimensiones. Concretamente y según *YoYoGames*, responsable del motor, se puede desarrollar hasta un 80% más rápido que con lenguajes nativos[8].

Esto queda demostrado tras construir un prototipo totalmente funcional del videojuego que hemos diseñado teniendo en cuenta la pequeña cantidad de tiempo que se ha dedicado al desarrollo del mismo. Además, el motor de videojuegos utilizado prácticamente obliga a utilizar una metodología de desarrollo ágil como Extreme Programming y esto también repercute en la velocidad de desarrollo del producto además de otra serie de ventajas como la adaptabilidad a cambios que necesitan proyectos con un alto nivel de investigación o aprendizaje.

El uso de patrones de diseño en GameMaker Studio se hace relativamente diferente a otros paradigmas de programación más comunes como la Programación Orientada a Objetos. Esto se debe a la naturaleza arquitectónica del motor, pero muchos solo necesitan ser adaptados. Durante el desarrollo se han añadido algunos de ellos de forma explícita.

Es posible que otros motores como *Unity3D* también ofrezcan un buen rendimiento a la hora de desarrollar videojuegos de este estilo ya que, aunque carece de ciertas ventajas que sí posee *GameMaker Studio* como el editor de escenas 2D mejorado, este motor de videojuegos tiene otra serie de ventajas de las que no dispone el que hemos utilizado y que también aligerarán el desarrollo. Estas ventajas son, por ejemplo los *assets* descargables o sistemas ya integrados en el motor que nos ahorrarán creación de código o recursos y que, sin embargo, con *GameMaker Studio* debemos construir por nuestra cuenta.

De todo esto se concluye que, junto a lo comentado en el primer capítulo de este documento, la actualidad **es el momento óptimo para producir videojuegos de no ocio** para promocionar productos **de una forma eficaz en el objetivo y eficiente en los recursos** utilizados para ello. La elección del motor de videojuegos que se utilice pasa a un segundo plano en la prioridad ya que los motores de videojuegos de la actualidad son muy semejantes; el impacto sobre la duración del proyecto será mayor dependiendo más del diseño y de la experiencia que de la herramienta utilizada de las comparadas en este estudio.

Este prototipo es extensible hacia una versión completa en la que restaría añadir el resto de las fases del proceso de fabricación del aceite de oliva como nuevos niveles y minijuegos. Además, de la implementación de un sistema de códigos más seguro y completo así como la implantación del sistema software de canjeado de códigos en las tiendas o supermercados que acepten la promoción, una ampliación de los servicios del juego.

De este modo se pueden promocionar productos de varias empresas relacionadas con el aceite de oliva gracias al reparto de códigos de descuento promocionales de una forma divertida.

Anexo I ESPECIFICACIÓN DE OBJETOS

A continuación se detallan ordenados alfabéticamente todos los objetos que se han creado para el prototipo con las acciones que realizan cada uno de sus eventos.

Se han eliminado de esta especificación todos aquellos objetos que tienen un carácter exclusivamente visual para el sistema como puede ser el contador de tiempo o de puntos o las aceitunas de ejemplo que se muestran en el primer nivel debido a que prácticamente sólo están formados por el evento Create, Draw y las alarmas para el efecto que suelen ser Alarm0 y Alram1.

Índice

Objetos

Objeto	**objAceitunaJ1**	
Hereda	No	
Eventos	Create	Declaración y asignación de atributos
	Step	Comprobación del proceso de coger la aceituna (coger, arrastrar y soltar)
	Draw	Dibujar aceituna y su sombra

Objeto	**objBoton**	
Hereda	objBotones	
Eventos	Create	Heredado Sobre escritura de atributos para efectos
	UserDefined0	Cálculo de ajuste del texto
	Draw	Dibujar botón en la escena

Objeto	**objBotones**	
Hereda	objIN	
Eventos	Create	Heredado Declaración de atributos Llamada a Alarm0
	Alarm0	Efecto de entrada
	Alarm1	Efecto de salida Destrucción Llamada al controlador
	LeftPressed	Selección del botón Llamada a UserDefined2
	UserDefined2	Preparación para el efecto de salida Llamada a Alarm1

Objeto	**objBotonMini**	
Hereda	objBotones	
Eventos	Create	Heredado Sobre escritura de atributos para efectos
	UserDefined0	Cálculo de ajuste del texto
	Draw	Dibujar botón en la escena

Objeto	**objBotonSonido**	
Hereda	objBotonMini	
Eventos	Create	Heredado Sobre escritura de atributos para efectos Cálculo de ajuste del texto
	LeftPressed	Llamada a controlador sin efectos
	Draw	Dibujar botón en la escena

Objeto	**objController**	
Hereda	No	
Eventos	Create	Llamada a room_config_start()

Objeto	**objController1**	
Hereda	objController	
Eventos	Create	Heredado Generación de aceitunas de ejemplo, paneles y contador para habilitar
	Alarm0	Creación de aceitunas
	Alarm1	Mensaje de final
	Alarm2	Cambio de escena con objTransicion
	Alarm3	Creación de aceitunas de ejemplo
	UserDefined0	Clasificar aceituna y calcular puntos
	UserDefined1	Comprobar si la cola está llena
	UserDefined2	Terminar minijuego
	UserDefined11	Habilitar juego

Objeto	**objController2**	
Hereda	objController	
Eventos	Create	Heredado Generación de rodillos, paneles y contador para habilitar
	Alarm1	Control de tiempo restante
	Alarm3	Cambio de escena
	Step	Comprobar giro y sumar puntos
	UserDefined0	Activar engranajes y paneles
	UserDefined1	Nueva aceituna
	UserDefined11	Habilitar juego

Objeto	**objController3**	
Hereda	objController	
Eventos	Create	Heredado Generación de elementos gráficos, paneles y contador para habilitar
	Alarm0	Control de tiempo restante
	Alarm1	Creación de indicador de acción
	Alarm2	Cambio de escena
	Step	Cálculo de bonificación por movimiento
	UserDefined11	Habilitar juego

Objeto	**objControllerInit**	
Hereda	objController	
Eventos	Create	Heredado Llamada a mao_config_load() Comprobación de registro
	Alarm10	Abortar petición (tiempo agotado)
	Alarm11	Llamada al servicio de registro
	UserDefined10	Mapeado de acciones
	HTTP	Gestión del valor de vuelta del servicio

Objeto	**objControllerLoad**	
Hereda	objController	
Eventos	Create	Heredado Carga de mapas Paso a la siguiente escena

Objeto	**objControllerResultados**	
Hereda	objController	
Eventos	Create	Instanciación de título Llamada a Alarm0
	Alarm0	Creación de las cajas resumen de puntos de cada nivel
	Alarm1	Creación del gráfico de clasificación del aceite
	Alarm2	Habilitar
	Alarm10	Abortar petición (tiempo agotado)
	Alarm11	Llamada a servicio de actualización de puntos y comprobación de códigos
	UserDefined10	Mapeado de acciones
	HTTP	Gestión del valor de vuelta del servicio

Objeto	**objIN**	
Hereda	No	
Eventos	Create	Declaración de atributos

Objeto	**objIndiceCodigos**	
Hereda	objController	
Eventos	Create	Llamada a Alarm11
	Alarm10	Abortar petición (tiempo agotado)
	Alarm11	Llamada a servicio de recuperación de códigos promocionales
	UserDefined10	Mapeado de acciones
	HTTP	Gestión del valor de vuelta del servicio

Objeto	**objIndiceRanking**	
Hereda	objController	
Eventos	Create	Llamada a Alarm11
	Alarm10	Abortar petición (tiempo agotado)
	Alarm11	Llamada a servicio de recuperación de puntos/ranking
	UserDefined10	Mapeado de acciones
	HTTP	Gestión del valor de vuelta del servicio

Objeto	**objMensaje**	
Hereda	objIN	
Eventos	Create	Heredado Llamada a Alarm0
	Alarm0	Efecto de entrada
	Alarm1	Efecto de salida Destrucción Llamada al controlador
	GlobalMouse	Aceptar mensaje
	DrawGUI	Dibujar mensaje en pantalla

Objeto	**objRodillo**	
Hereda	No	
Eventos	Create	Declaración y asignación de atributos
	Step	Comprobación y aplicación del giro

Anexo II ESPECIFICACIÓN DE FUNCIONES

A continuación se muestran todas los scripts creados para el prototipo que ordenados alfabéticamente y por categoría.

Nota: Todos los códigos son numéricos. GameMaker Studio sólo diferencia dos tipos de datos: reales y strings.

Índice

Funciones

Config
mao_config_load ( )
Guarda la configuración en un archivo cifrado ya prefijado.
mao_config_save ( )
Carga la configuración desde un archivo cifrado ya prefijado.

Controller
mao_controller_call (*id, código_acción*)
Llama al evento que gestiona los códigos de las operaciones de los controladores. Suele usarse en los botones o mensajes para indicar al controlador de que el usuario está solicitando una acción, aunque se puede utilizar internamente en el propio controlador o desde cualquier objeto.

FX
mao_fx_salpicadura (*x, y*)
Crea un *objSalpicadura* en (*x*, *y*) para imitar la salida del jugo de la aceituna.
mao_fx_splash ( )
Crea un *objSplash* en una posición aleatoria para imitar manchas en la pantalla.

Map

mao_map_get (*mapa, clave*)

Devuelve el valor asignado a una clave en el mapa indicado.

Si no existe dicha clave, devuelve un String vacío.

La finalidad de esta función es evitar errores y hacer el código más legible evitando comprobar la existencia de la clave con la función is_undefined() cuando se recupere información de un mapa.

mao_map_set (*mapa, clave, valor*)

Asigna un valor a la clave del mapa indicado.

El juego comenzará a no funcionar correctamente si se usan las funciones nativas como ds_map_add() y ya existe dicha clave, en lugar de utilizar ds_map_replace().

La finalidad de esta función es evitar errores y hacer el código más legible evitando comprobar la existencia de la clave con la función ds_map_exists() para saber si hay que reemplazar el valor o añadirlo nuevo.

Mouse

mao_mouse_in (*x1, y1, x2, y2, id_mouse*)

Devuelve el test pertenencia del punto del ratón *id_mouse* en la caja (*x1*, *y1*), (*x2*, *y2*).

GameMaker Studio usa las funciones device_mouse que utilizan un id de ratón para identificar el dedo que se ha pulsado en pantallas multi-táctiles.

Service
mao_service_call (*clave, data_map*)
Solicita el servicio asignado a *clave* indicándole como datos *data_map*. Devuelve el identificador de la petición, necesario para reconocer el servicio en el evento HTTP. La finalidad de esta función es hacer la llamada al servicio de forma más limpia, sin usar las URL repartidas por el código y sin preocuparse del método. Una vez se llama a la petición data_map es destruido para evitar memory leaks.
mao_service_init_map ( )
Devuelve el mapa requerido para la petición inicial del servicio de registro de usuarios. Pueden añadirse nuevas entradas al mapa con mao_map_set. Los servicios requieren un mapa en el que, además de los datos personalizados, existan una serie de entradas destinadas a la seguridad e identificación. Este mapa crea automáticamente el mapa con las mínimas claves necesarias.
mao_service_init ( )
Devuelve el mapa requerido para la petición estándar de servicios. Pueden añadirse nuevas entradas al mapa con mao_map_set. Los servicios requieren un mapa en el que, además de los datos personalizados, existan una serie de entradas destinadas a la seguridad e identificación. Este mapa crea automáticamente el mapa con las mínimas claves necesarias.

Services
mao_services_get (*clave*)
Devuelve la URL del servicio asignado a *clave*. Si no existe devuelve una URL vacía. La finalidad de esta función es hacer independiente los servicios declarados en el código del juego de las URL del servidor, siendo más fácil la modificación futura y obteniendo más flexibilidad al hacer de fachada.
mao_services_load ( )
Carga el mapa de los servicios con URL y métodos que utiliza cada uno. Esta función debe ser llamada una única vez en el juego para evitar memory leaks. Gracias a esta función se hace mucho más sencilla la modificación de las URL y métodos que usan los servicios dentro del juego y la llamada a los mismos al tener que recordar solo la clave del servicio.
mao_services_method (*clave*)
Devuelve el método del servicio asignado a *clave*. Si no existe devuelve una cadena vacía. La finalidad de esta función es hacer independiente los servicios declarados en el código del juego de las URL del servidor, siendo más fácil la modificación futura y obteniendo más flexibilidad al hacer de fachada.

Sound
mao_sound_play (*sonido*)
Reproduce una única vez el sonido indicado si está activo en la configuración. Con esta función nos ahorramos comprobar si el sonido está activado de forma explícita.
mao_sound_play_background (*sonido*)
Reproduce en bucle el sonido indicado si está activo en la configuración. Se diferencia con mao_play_sound() en que además de repetirse en bucle, la prioridad es más alta. Además de esto, en el futuro se puede usar para que si se llama dos veces con diferentes sonidos, la detener el primero para hacer sonar el segundo, mientras que si ambos son el mismo sonido no hacer nada o gestionar como hacer cambiar la música de fondo con fundidos.

Texts
mao_texts_get (*clave*)
Devuelve el valor asignado a una clave en el mapa de los textos. Si no existe dicha clave, devuelve un la misma clave. La finalidad de esta función es la futura localización de la aplicación a otros idiomas. En lugar de introducir los textos en el código, estarán en un mapa para su uso.
mao_texts_load ( )
Carga el mapa de textos. En el prototipo los textos y palabras se definen en este script, pero la idea en el futuro es que esta función cree el mapa desde un archivo dependiendo del idioma detectado. Esta función debe llamarse una única vez en todo el juego o destruir el mapa global.textos antes de volver a realizar la llamada.

Creation
new_BarraResultados (*y, texto, valor*)
Crea un *objBarraResultados* en la posición *y* que mostrará *texto* a la y *valor*.
new_Boton (*x, y, sprite, texto, código_acción*)
Crea un botón grande en la posición (*x,y*), *sprite* y *texto* indicados. Si es pulsado llama al controlador que lo creó con el código indicado.
new_BotonMini (*x, y, sprite, texto, código_acción*)
Crea un botón pequeño en la posición (*x,y*), *sprite* y *texto* indicados. Si es pulsado llama al controlador que lo creó con el código indicado.
new_CajaInfo (*x, texto, valor*)
Crea un *objCajaInfo* en la posición *x* que muestra *texto* y *valor*.
new_Mensaje (*texto, código_acción*)
Crea un mensaje de notificación con el *texto* indicado. Si es pulsado llama al controlador que lo creó con el código indicado.

Anexo III ESPECIFICACIÓN DE SERVICIOS

Tanto los datos de entrada como los de salida deben estar/están codificados usando base64.

Todos los servicios devuelven un pequeño código de la aplicación, en este caso es “MAO”, para que el cliente pueda comprobar que el servicio que solicita es el que realmente está solicitando. Igualmente usan una codificación para notificar de errores.

A continuación se encuentra el índice con toda la especificación de los servicios detallada.

Índice

Lista de servicios

Para la llamada del servicio es necesario un mapa en formato JSON que contenga ciertas claves dependiendo del servicio. Este mapa se llama data y en las siguientes tablas aparecen las claves necesarias para que el servicio funcione.

No obstante, se puede simplificar su uso usando los scripts como mao_service_map (ver Anexo II).

En el futuro se pueden añadir más claves como el código de versión del juego para garantizar obligar a que el usuario actualice a la última versión.

codes.php	
Método	GET
Clave cliente	*GetCodes*
Busca y devuelve los códigos promocionales activos para el usuario. Los códigos marcados como usados no se muestran.	
Claves necesarias en *data*	
id_user	Identificador del usuario.
raw	Cadena aleatoria.
sec_user	Cadena formada por la concatenación de *raw* y la clave privada del usuario. Tras la concatenación, se cifra con el algoritmo sha1.
Función data	*mao_service_map*
Claves devueltas	
n_promo_codes	Número de códigos activos encontrados.
promo_codeX	Código, donde X es el número de 0 a n_promo_codes.
promo_markX	Marca de la promoción, donde X es el número de 0 a n_promo_codes.

ranking.php	
Método	GET
Clave cliente	*GetRanking*
Devuelve la mayor puntuación de todos los usuarios y la del que la solicita.	
Claves necesarias en *data*	
id_user	Identificador del usuario.
raw	Cadena aleatoria.
sec_user	Cadena formada por la concatenación de *raw* y la clave privada del usuario. Tras la concatenación, se cifra con el algoritmo sha1.
Función data	*mao_service_map*
Claves devueltas	
total_max	Máxima puntuación global.
user_score	Máxima puntuación del usuario.

score.php	
Método	POST
Clave cliente	*PutScore*
Guarda la puntuación máxima del usuario que la solicita. Busca las campañas activas en las que se supera la puntuación requerida y el usuario no dispone de código y las devuelve. Si no existen, no devuelve nada. En caso de no conseguir actualizar los puntos devolverá los errores estándar.	
Claves necesarias en *data*	
id_user	Identificador del usuario.
raw	Cadena aleatoria.
sec_user	Cadena formada por la concatenación de *raw* y la clave privada del usuario. Tras la concatenación, se cifra con el algoritmo sha1.
score	Puntos obtenidos.
Función data	*mao_service_map + score*
Claves devueltas	
n_promo_codes	Número de códigos nuevos.
promo_codeX	Código, donde X es el número de 0 a n_promo_codes.
promo_markX	Marca de la promoción, donde X es el número de 0 a n_promo_codes.

user.php	
Método	POST
Clave cliente	*PostUser*
Crea un nuevo usuario y devuelve un mapa con el identificador de usuario creado y su clave privada. Estos datos solo se devuelven una vez por el sistema, de forma que se deben almacenar en local.	
Claves necesarias en *data*	
cod_user	Cadena aleatoria generada por el cliente.
sec_app	Cadena sha1 formada por la concatenación de *cod_user* y la clave privada del servicio. Tras la concatenación, se cifra con el algoritmo sha1.
Función data	*mao_service_map_init*
Claves devueltas	
user	Identificador del usuario.
key	Clave privada generada para el usuario.

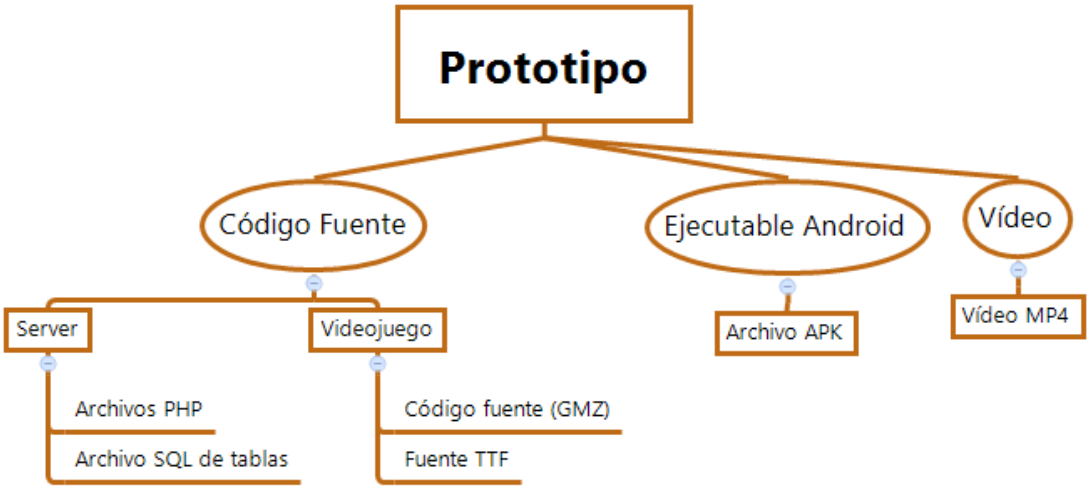
Códigos de error

Código	Significado
MAO001	Consulta vacía/método incorrecto.
MAO002	El mapa data está incompleto.
MAO005	Comprobación de seguridad no superada.
MAO101	Problema del servicio (Base de datos).

Anexo IV ARCHIVOS ADJUNTOS Y LICENCIAS

Junto a este documento se adjuntan varios archivos del prototipo, tanto ejecutables o muestras como código fuente. En la carpeta denominada “Prototipo” se encuentran repartidos en carpetas según categoría cada uno de los ficheros.

Entre ellos se incluye un vídeo de demostración con una partida de juego completa grabada desde el mismo dispositivo Android. La estructura de directorios es la siguiente:



Importante: La única forma de ver y editar el código fuente del videojuego es importando el proyecto desde GameMaker Studio. Al iniciar el IDE,en la ventana de bienvenida, hay una pestaña “Import” para ello.

Cuando se edita el código del juego, es necesario instalar la fuente incluida junto al código fuente del mismo. **Esta fuente no dispone de licencia para uso comercial**: en caso de producción del juego, es necesario sustituirla.

Finalmente, a parte del contenido generado por Cristóbal Mata, este prototipo contiene recursos de terceros que se usan bajo licencia Creative Commons bajo atribución que deberán tenerse en cuenta.

- Música por Kevin MacLeod (incompetech.com)

BIBLIOGRAFÍA

[1] **Video games in Europe** (ISFE). http://www.isfe.eu/sites/isfe.eu/files/attachments/euro_summary_-_isfe_consumer_study.pdf [Mayo 2015]

[2] **List of game engines** (VentureBeat). http://venturebeat.com/2014/08/20/the-top-10-engines-that-can-help-you-make-your-game/view-all/ [Junio 2015]

[3] **Featured Unity games** (Unity3d). https://unity3d.com/es/showcase/gallery [Julio 2015]

[4] **Top 5 game engines for developers** (Framebench). http://blog.framebench.com/top-5-game-engines-developers/ [Julio 2015]

[5] **Características de Unity3d** (Unity3d). https://unity3d.com/es/unity [Junio 2015]

[6] **Características de WaveEngine** (WaveEngine). http://waveengine.net/Engine/Overview [Junio 2015]

[7] **Game Maker Studio** (GameMaker Blog). http://www.gamemakerblog.net/the-history-of-yoyo-games/ [Junio 2015]

[8] **Game Maker Studio features** (YoYoGames). http://www.yoyogames.com/studio [Julio 205]

[9] **Android SDK and NDK setup** (YoYoGames). http://help.yoyogames.com/entries/23363366-GameMaker-Studio-Android-SDK-and-NDK-setup- [Agosto 2015]

[10] **Estadisticas oficiales sobre dispositivos Android** (Android). http://developer.android.com/about/dashboards/index.html [Junio 2015]

[11] **Rules of Extreme Programming** (ExtremeProgramming). http://www.extremeprogramming.org/rules.html [Julio 2015]

[12] **Proceso de fabricación del aceite** (CereSpain). http://www.cerespain.com/almazara.html [Junio 2015]

[13] **What is Planning Poker?** (MountainGoatSoftware). https://www.mountaingoatsoftware.com/agile/planning-poker [Junio 2015]

[14] **Burndown Charts** (ProyectosAgiles). http://www.proyectosagiles.org/graficos-trabajo-pendiente-burndown-charts [Julio 2015]

[15] **Salario medio del diseñador gráfico en España** (CuantoGanaa). http://www.cuantoganaa.com/cuanto-gana-un-disenador-grafico/ [Agosto 2015]

[16] **Oferta de trabajo desarrollador IOS** (InfoJobs). http://www.infojobs.net/madrid/analista-desarrollador-ios-madrid/of-i420f5794ad4f8aac20e64083d5db8b [Agosto 2015]

[17] **Artículo de El PAIS** (El PAIS). http://economia.elpais.com/economia/2014/09/15/actualidad/1410801712_858986.html [Agosto 2015]

[18] **GameMaker Studio Licenses** (YoYoGames). http://yoyogames.com/studio/buy [Agosto 2015]

[19] **Music Licenses** (PremiumBeat). http://www.premiumbeat.com/license [Agosto 2015]

[20] **XMind Website** (XMind): https://www.xmind.net/ [Agosto 2015]