Universidad Zaragoza
1542

Escuela de Ingeniería y Arquitectura Universidad Zaragoza
1542

Proyecto Fin de Carrera

MEMORIA

Curso de introducción a la programación de videojuegos para dispositivos móviles en Corona SDK

AUTOR

Carlos Lorenzo Paricio

DIRECTOR

Ramón Piedrafita Moreno

ESPECIALIDAD

Electrónica Industrial

CONVOCATORIA

Marzo 2012

RESUMEN

Desde la aparición en el mercado de dispositivos móviles con altas prestaciones asociados a la evolución de la tecnologia, estos han abierto un amplio mercado para el desarrollo de videojuegos y dotan de una potencia grafica similar a la que posee un entorno de juegos portátil tradicional como pueden ser la consolas portátiles Sony PSP o Nintendo DS.

El objetivo de este proyecto es desarrollar una aplicación genérica multiplataforma optimizada para Ipad, ya sea para su primera o segunda versión, que nos permita aprender a programar videojuegos 2D para dispositivos móviles que aunque no exploten todo el potencial que posee el dispositivo, es el genero mas descargado de las tiendas de aplicaciones móviles dada su adicción y dado que el consumo de batería es notablemente inferior que en los videojuegos en 3D, ya que estos últimos necesitan mas prestaciones.

Para este fin, se ha generado una aplicación didáctica que nos permite introducirnos en el mundo de la programación de videojuegos en este tipo de dispositivos , en ella se distinguen diferentes secciones. Por una parte se muestran capitulos relacionados con el lenguaje de programación de Corona SDK, por otro lado otros donde se desarrollan videojuegos por completo y se muestra su programación mediante ejemplos y también aparece otra sección con juegos totalmente funcionales.

En este proyecto también se realiza un estudio para conocer los entornos de desarrollo de los diferentes sistemas operativos móviles y varias opciones de desarrollo multiplataforma que existen en la actualidad.

Al final se ha elegido dadas sus caracteristicas un software multiplataforma llamado Corona SDK, permite la programación de dispositivos con iOS (Iphone, IPod y IPad), Android(numerosos dispositivos en el mercado) y recientemente para Amazon/KIndle. Tiene un alto rendimiento y permite la utilización de sus componentes nativos. Recalcar que varias aplicaciones desarrolladas con Corona SDK han sido lideres en descargas en la “AppStore” y “Android Market” por lo que se demuestra su gran potencial y se puede considerar como una opción alternativa a otros entornos de programación.

Aunque esta aplicación seria posible instalarla en cualquier dispositivo IOS o Android dado su código comun, es necesario poseer una gran pantalla para ejecutarla. Esta es la razón principal por la que hemos elegido una tablet y en nuestro caso un producto comercializado por Apple, el Ipad. Ya que parte del contenido de la aplicacion es texto y dado que nuestro dispositivo posee una gran pantalla de 9,7 pulgadas podremos verlo perfectamente en este dispositivo.

ÍNDICE DE CONTENIDO

1 Objetivo del proyecto

1.1 Introducción

El crecimiento y popularización de los dispositivos móviles durante los últimos años, ha generado un gran desarrollo de estos ,que, además de permitir la tradicional comunicación entre redes fijas y móviles, poseen la capacidad de ejecutar todo tipo de aplicaciones desarrolladas específicamente para estos dispositivos.

Las prestaciones de los dispositivos móviles aumentan dia día, posibilitando así la implementación de aplicaciones muy potentes e interesantes. Los denominados "Smartphones" o teléfonos de ultima generación y las "Tablets", presentan una serie de características de procesamiento, memoria y capacidad de conexión que está llevando a los grandes fabricantes, operadores y desarrolladores a una auténtica carrera por lograr las mejores aplicaciones y estar a la cabeza de ventas.

Google[1] ha hecho públicos los resultados de un estudio llevado a cabo el pasado mes de marzo en Estados Unidos, en el que se entrevistó a un total de 1.430 usuarios poseedores de un moderno Tablet.

El perfil tipo de poseedor de un tablet utiliza este dispositivo principalmente en el domicilio particular (82%) antes que en el trabajo (7%), e incluso antes que en movilidad (11%), aunque curiosamente un aparato ligero de estas características está más pensado, precisamente, para entornos móviles.

Aunque el tablet no es considerado aún como el principal dispositivo informático, está robando horas de uso al pc o al portátil.

Entre los principales usos que se dan al tablet destaca su vertiente lúdica, con un 84% de los encuestados utilizando su dispositivo para videojuegos. La lectura de noticias se encontraría en cuarta posición con un 61%, práctica que más terreno podría estar quitándole a los periódicos en papel, mientras que el 56% de los usuarios se conectan a redes sociales, el 46% consumen e-books, y el 74% gestiona su correo electrónico.

Digamos que si resumimos todos los datos que ha arrojado el estudio, tendríamos que el poseedor de un tablet lo utiliza cuando llega a su casa después del trabajo para navegar por Internet, gestionar su correo electrónico, acceder a las redes sociales a través de apps específicas y además como entorno de aprendizaje o lector de libros.

Este estudio conviene a los desarrolladores de tablets, terreno en el cual ha presentado sus credenciales Samsung con Android 3.0 , Apple con su IOS 5 para el Ipad u otras plataformas menos conocidas y por tanto menos utilizadas.

[1] http://www.imatica.org/bloges/2011/04/190486382011.html

Los tablets se distinguen por muchas características, entre las que destacan las distintas pantallas táctiles, un sistema operativo propio, la conectividad a internet y el acceso al correo electrónico. Otras aplicaciones que suelen estar presentes son las cámaras integradas, acelerómetro, la reproducción de música y visualización de fotos y videos y algunos programas de navegación así como aplicaciones de lectura de documentos en distintos formatos electrónicos. La mayoría de dispositivos también permiten al usuario instalar programas adicionales para dar un valor añadido al dispositivo y así poder competir con sus rivales.

En función de los distintos fabricantes, cada uno de los dispositivos móviles posee un sistema operativo que determina las capacidades multimedia de los aparatos, y la forma de interactuar con el usuario.

1.2 Objetivo del proyecto

En este entorno, el objetivo del proyecto es el desarrollo de una aplicación desarrollada específicamente para IPad que permita al usuario aprender interactuando con ésta de forma sencilla y sin apenas conocimientos previos de programación de videojuegos. Se pretende la realización de una herramienta sencilla y que haga posible la distribución de conocimientos de programación hacia personas que quieren iniciarse en esta disciplina.

Para este fin, después de conocer distintos entornos de desarrollo para varios sistemas operativos móviles, se utilizará una plataforma de software multiplataforma que permite la construcción para dispositivos con iOS, Android y Kindle.

El objetivo es tener un código de aplicación común que sumado a unos contenidos configurables nos permitan la construcción de aplicaciones nativas para varios dispositivos aunque en este caso la aplicación ha sido gráficamente diseñada para la tablet de Apple, el Ipad.

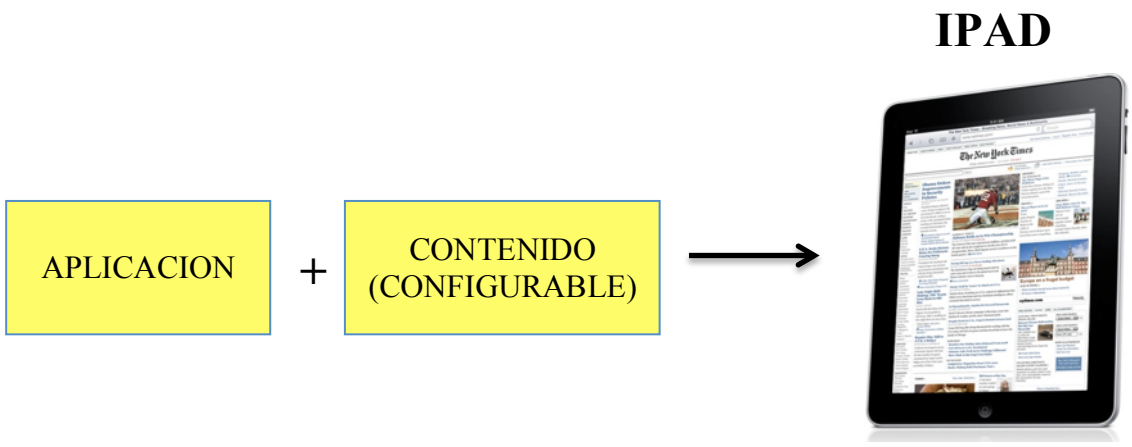

Figura 1.1 . Esquema del objetivo del proyecto

1.3 Organización de la memoria

La memoria del proyecto esta organizada por capitulos, se enumeran a continuación incluyendo una pequeña descripción del mismo.

En el capitulo 2 se especifica la metodología en el desarrollo del proyecto y la planificación del mismo.

En el capitulo 3 se realiza el diseño e implementecion de la aplicación. Todo lo incluido a la programcion de la misma.

En el capitulo 4 se recopilan las conclusiones extraídas en el desarrollo del proyecto y las posibilidades de actualización en un futuro.

En el capitulo 5 se realiza un estudio de las plataformas para la programación de dispositivos móviles incidiendo en aquellas que permiten la generación de aplicaciones multiplataforma.

En el capitulo 6 se analiza la opción elegida en este proyecto, Corona SDK de “Ansca mobile”. Se realiza una detallada descripción de sus características que han llevado a la elección de esta para el desarrollo del proyecto.

En el capitulo 7 se enumera toda la bibiografia utilizada en la memoria asi como la posibles referencias.

En el capitulo 8 se puede aceder al Anexo I donde podemos consultar detalles de las funciones mas especificas de Corona SDK.

2 Metodología y planificación del proyecto

En el desarrollo de este proyecto se ha seguido una metodología basada en un ciclo de vida evolutivo. El desarrollo evolutivo se basa en la idea de realizar una implementación inicial e ir modificando la aplicación hasta que se obtiene un sistema que se considere adecuado. Las actividades de desarrollo y verificación se entrelazan en lugar de separarse, y a su vez, existe una rápida retroalimentación entre ambas. Este modelo acepta que los requerimientos del usuario puedan modificarse e incorporarse nuevos.

La práctica demuestra que obtener todos los requerimientos al comienzo del proyecto es complejo, no sólo por la dificultad de definir la totalidad del producto, sino porque estos requerimientos evolucionan durante el desarrollo y de esta manera, surgen nuevos requerimientos a cumplir. El modelo de ciclo de vida evolutivo afronta este problema mediante una iteración de ciclos en los cuales se incluyen posibles nuevos requisitos, posteriormente se mejora o cambia el desarrollo y por último se lleva a cabo una nueva evaluación.

En la siguiente figura se puede ver de forma clara el esquema del ciclo de vida del proyecto.

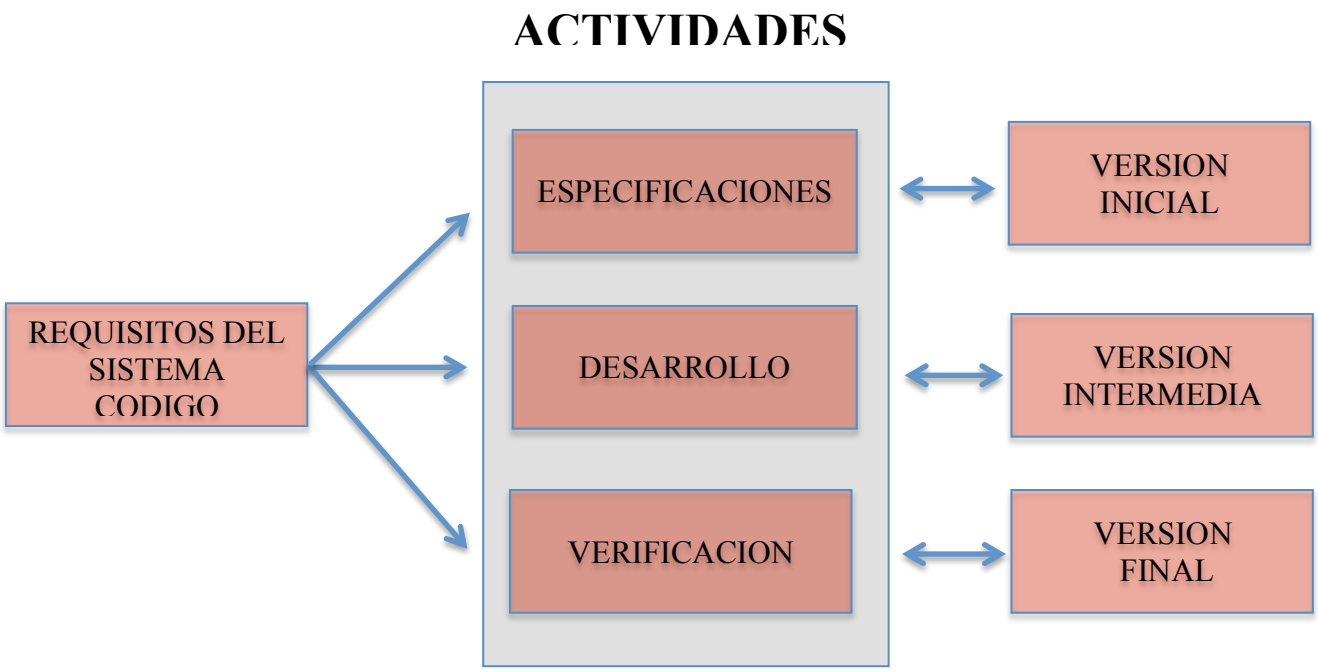

Figura 2.1 . Esquema de realización del proyecto basado en ciclo de vida evolutivo

Previamente al desarrollo evolutivo, se debe realizar la tarea de descripción del proyecto. Esta actividad incluye la definición del objetivo del proyecto, el estudio y elección de la plataforma de desarrollo y el esbozo de los requisitos de la aplicación. Para esta tarea se dedican aproximadamente 5 semanas que incluye el estudio y conocimiento de la plataforma seleccionada.

Durante la especificación, se ha profundizado en el análisis y la definición de los requisitos de la aplicación adaptándolos tanto a las necesidades surgidas en las otras fases como a las características de la plataforma de desarrollo seleccionada.

El primer desarrollo del proyecto, se centró en obtener un primer prototipo que permitiera la navegación entre los distintos apartados de la aplicación, sin apenas contenidos ni elementos gráficos finales y verificando que era aceptablemente visible en el simulador.

Previo a la segunda etapa de desarrollo en la que se implementó toda la funcionalidad de generación de contenidos , se realizó la especificación y análisis del formato que deben tener los contenidos. Solo en las versiones más evolucionadas, se ha realizado la construcción de la aplicación y la prueba en el dispositivo.

Finalmente se han desarrollado las pantallas y funciones auxiliares, así como el diseño gráfico final. Una vez realizadas las pruebas y verificaciones completas de la aplicación, se procede a la construcción de la versión final y se instala en el dispositivo para su testeo.

Como se ha indicado anteriormente, seguiremos un ciclo de vida evolutivo, por lo que todas estas fases se realizan en paralelo. Para todas estas tareas se dedican unas 14 semanas

Posteriormente, se requiere 2 semanas de documentación y revisión previas a la finalización del proyecto.

En el siguiente diagrama de Gantt pueden verse la planificación.

Id.	Nombre de tarea	Comienzo	Fin	Duración
1	Definicion Objetivo Proyecto	26/09/2011	30/09/2011	1s
2	Estudio PItaforma Desarrollo	03/10/2011	14/10/2011	2s
3	Estudio Plataforma Seleccionada	17/10/2011	28/10/2011	2s
4	Especificacion y analisis requerido	31/10/2011	27/01/2012	13s
5	Diseño	31/10/2011	27/01/2012	13s
6	Implementacion	31/10/2011	27/01/2012	13s
7	Verificacion	31/10/2011	27/01/2012	13s
8	Documentacion	03/02/2012	16/02/2012	2s
9	Finalizacion	03/02/2012	16/02/2012	2s

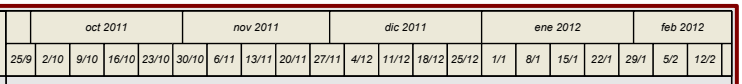

Figura 2.2 . Diagrama de Gantt en el cual puede verse la planificación del proyecto

3 Diseño e implementación de la aplicacion

3.1 Requisitos de la aplicación

Como se ha comentado al principio de la memoria, el objetivo del proyecto es el desarrollo de una aplicación didáctica espedifica para la tablet de Apple, el Ipad.

Debe ser un desarrollo que permita generar una aplicación capaz de ser portada a otro dispositivo sin apenas conocimientos sobre la misma. Cambiando ciertos parámetros, sobre todo la adaptación del contenido, podremos ser capazes de utilizar nuestra aplicación en otros dispositivos que funcionan con otro sistema operativo móvil.

De acuerdo a esta descripción del sistema se realiza la especificación de requisitos que describe las funcionalidades que debe tener la aplicación.

3.1.1 Requisitos funcionales

- Aplicación adaptada especificamente para Ipad.
- La aplicación debe ser multi-idoma(español e ingles).
- Debe ser posible cambiar ciertos paremetros de configuración.
- Al salir de la aplicación, ésta debe ser capaz de guardar la posición en la cual nos encontramos en ese momento.
- La estructura de la aplicación debe construirse para una posible actualización futura.
- La aplicación permitirá enviar el código de los juegos que aparecen en ella para que los usuarios puedan disponer de el para su estudio detallado.

3.1.2 Requisitos no funcionales

- La plataforma de desarrollo será Corona SDK de Ansca Mobile
- La aplicación deberá estar disponible para la plataforma iOS.
- El usuario debe ser capaz de interactuar con la aplicación intuitivamente y sin apenas dificultad.

3.2 Diseño

Una vez se ha estudiado la finalidad del sistema y su viabilidad, se procede a realizar el diseño de la aplicación de cara a realizar su implementación.

Se busca que la aplicacion se caracterice por ser una aplicación sencilla, fácil de manejar y que se adapte al dispositivo y a los cambios de configuración.

Al ser una aplicación didáctica no es necesario que presente un aspecto refinado, elegante y formal sino que admite que las formas de los objetos puedan ser todas distintas, las asimetrías puedan ser comunes y los colores sin sentido.

Para el diseño de la aplicación hemos considerado una opción que nos ha proporcionado Corona SDK a través de un desarrollador que ha ofrecido su trabajo para todo el mundo

de forma libre y gratuita. Es un modulo externo o librería que nos permite hacer funcionar aplicaciones a base de escenas. El nombre del modulo es "DirectorClass" y su autor es un programador brasileño llamado Ricardo Rauber. Desde que publicó su trabajo la ha ido actualizando para corregir errores y mejorarla y actualmente nos encontramos por la versión 1.4.

Recientemente Corona SDK ha publicado en su lista de API'S una nueva forma de utilizar escenas pero esta vez de forma oficial. Lo han llamado "Storyboard" y permite el movimiento entre escenas como lo hace *directorClass*.

La razón por la cual vamos a utilizar *directorClass* y no *Storyboard* es que en el inicio de la programación de la aplicación esta era la única opción que presentaba Corona SDK y se decidio en su momento utilizarla. Ademas posee alguna característica que la librería propia de Corona SDK no, como por ejemplo que lleva incorporado un recolector de objetos y de listeners que los elimina cuando cambiamos de escena. En cambio "Storyboard" no ofrece esa posibilidad.

A continuación se muestra la imagen del árbol de opciones de la aplicación.

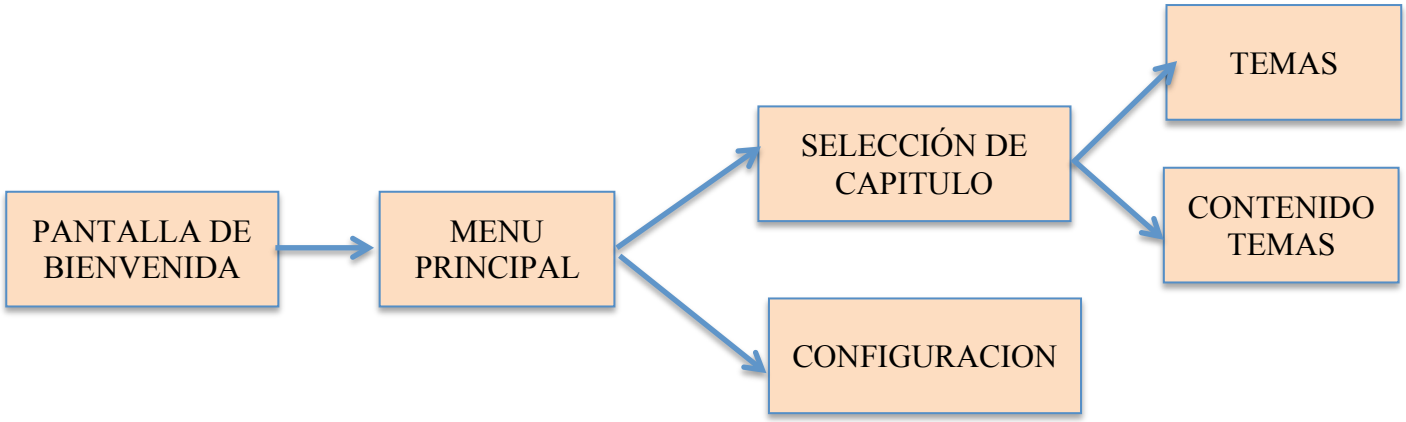

Figura 3.1. Opciones de la aplicación

La aplicación constará de una pantalla de bienvenida que nos dará paso a la pantalla del menú principal o a la ultima pantalla que teníamos cuando cerramos la aplicación en el caso de ser un capitulo de ejemplos. En la pantalla del menú principal aparecerá una lista con todos los capítulos disponibles, separados por colores y botón en la parte superior derecha que al pulsarlo aparece la pantalla de configuración.

Las opciones, como se puede apreciar en el esquema anterior, son:

- **Seleccionar Capitulo:** Dentro de la selección de capitulo existen varios tipo de ellos. Los 3 primeros son capitulos teoricos, los 4 siguientes son capítulos que están basados en ejemplos y los 4 ultimos son capítulos son juegos. Los 3 tipos de capitulos están diferenciados por colores.

- **Configuración**: Muestra la pantalla de configuración de la aplicación. Se pueden modificar varias o

- pciones.

Cuando un usuario se introduce en un capitulo la forma del contenido del mismo depende del tipo de capitulo que ha pulsado. Como se ha comentado anteriomente existen 3 tipos de capítulos distintos y pasamos a explicar mas en detalle cada uno de de ellos. Los vamos a separar por escenas y están explicacos a continuación.

3.2.1 Escenas

3.2.1.1 Tipo 1

Este tipo de escena es utilizado en los tres primeros capítulos. Estos capítulos digamos que son teoria. No podemos interactuar en ellos mas que para pasar las diapostivivas. En ellos podermos ver informarcion relacionada con el titulo del capitulo pulsado. Se puede decir que es un pase de diapositivas porque el contenido del mismo son imágenes que podemos ir cambiando con los botones que están situdados en la parte inferior de la pantalla o deslizando con el dedo hacia la derecha o hacia la izquierda dependiendo si queremos avanzar o retrasar la diapositiva.

3.2.1.2 Tipo 2

Esta escena es la que mas complejidad de las 3 lleva. Es la escena que utilizamos en los capítulos que aparecen los ejemplos de programación. Es una escena con la estructura común para los 4 capitulos pero que dependiendo de las variables que se cargan al inicio del mismo el capitulo se configura de forma diferente y es posible utilizarlo.

3.2.1.3 Tipo 3

Es la única escena que no sigue una patrón común para el mismo tipo de capítulos. Son capitulos tipo “juegos”, en ellos aparecen juegos completos que han sido creados anteriormente en los capitulos de ejemplos. Al ser juegos diferentes su programación no se parece aunque se ha intentado que la estructura de los mismos sea parecida para mayor facilidad de programación y llevar un orden concreto y poder utilizarlo en todos los juegos.

3.2.2 Diagramas de actividad

Se muestran a continuación los diagramas de actividad de carga del contenido de las escenas principales de la aplicación.

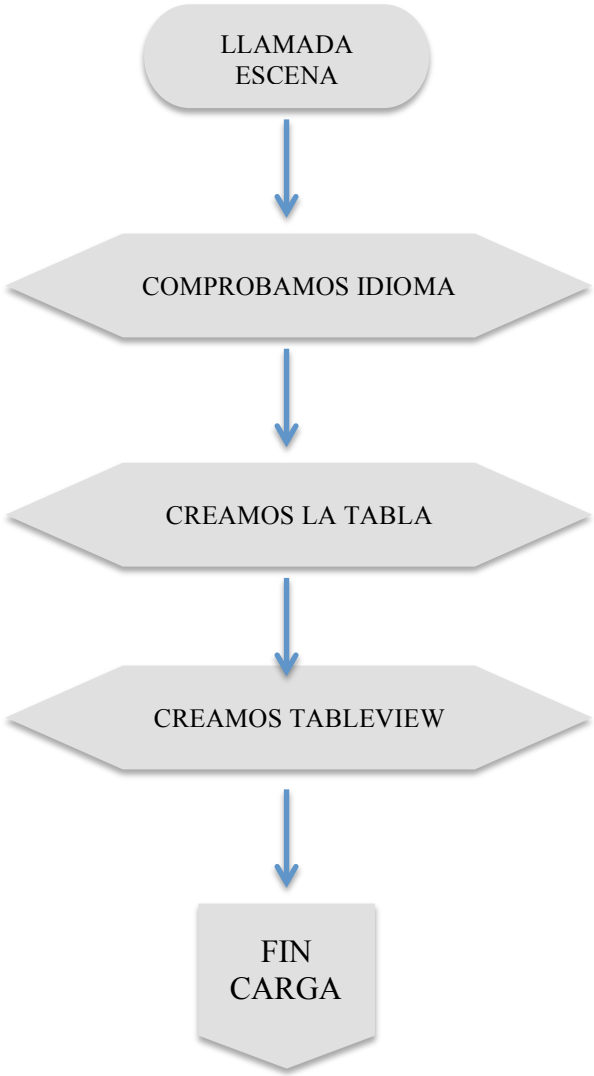

Figura 3.2. Diagrama de actividad de la carga de la pantalla principal

En este diagrama se muestra el flujo de carga de datos desde el modulo externo que contiene la tabla de capítulos del menú principal de la aplicación.

Esta carga viene condicionada mayormente por el idioma de la apliacacion y por el tipo de capitulo.

Según el estado de la variable *language_state* la tabla que pasamos a la tableView para que muestre la lista de capitulos en la pantalla del menú principal contiene un idioma u otro.

Para determinar el color de la fuente lo conseguimos con la variable *id*, nos permite seleccionar el color dependiendo de su valor. Es posible configurar la tableView a nuestro antojo y muestra de ello es que seria posible elegir un color para cada fuente de la misma.

En la siguiente figura se muestra el diagrama de actividad de la carga de la escena Tipo 1 que esta directamente relacionada con los 3 primeros capitulos.

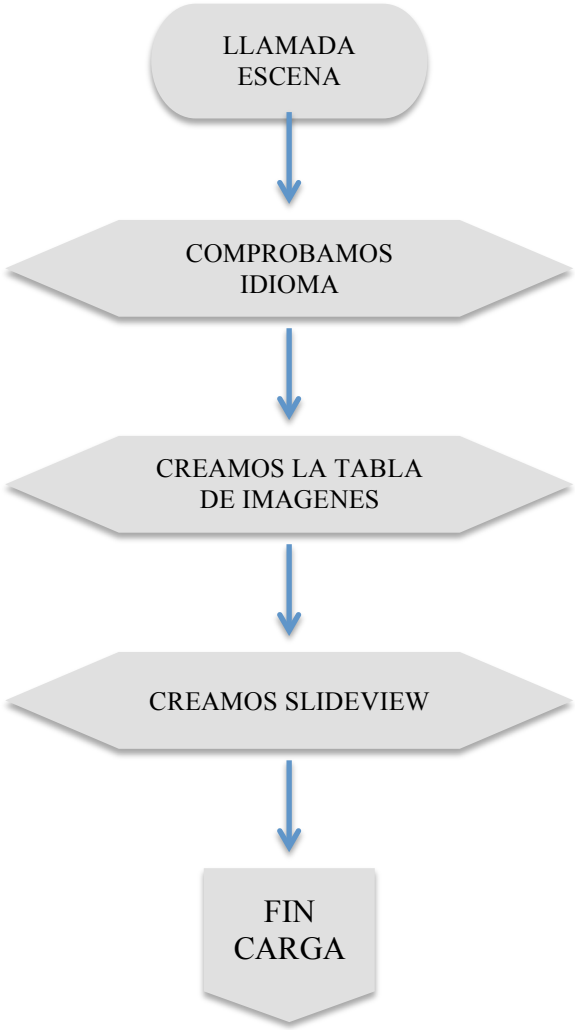

Figura 3.3. Diagrama de actividad de la carga del contenido de la escena Tipo 1

Como podemos observar el primer paso realizado es comprobar el idioma actual de la aplicación. Cada vez que se llama a la escena se hace este chequeo.

Despues de comprobar el idioma, creamos dentro de la escena la variable local *myImages* que no es mas que una tabla en la cual están incluidas las rutas de las imágenes que serán mostradas dentro del capitulo.

Estas imágenes son diferentes para cada idioma por lo cual es necesario incluir dentro de la carpeta los 2 idiomas.

Por ultimo creamos un objeto slideView, le pasamos como parámetro la tabla de ubicaciones de imágenes que hemos creado al principio dependiendo del idioma. Las imágenes que mostramos tienen unas dimensiones espeficicas para poder ocupar todo el espacio libre disponible. Estas dimensiones son 900 x 600 pixeles

A continuación se muestra el diagrama de actividad de la escena Tipo 2.

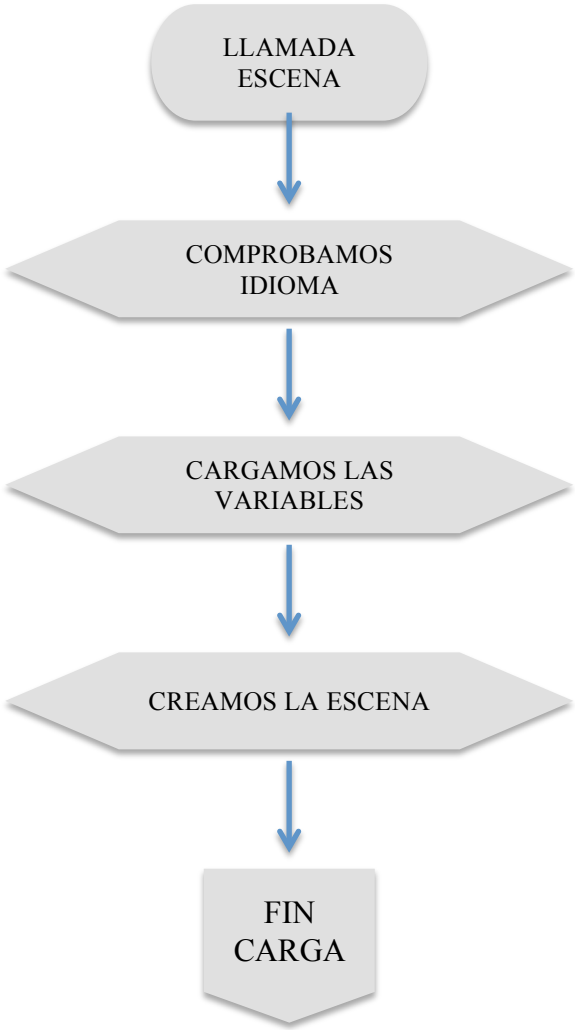

Figura 3.4. Diagrama de actividad de la carga del contenido Tipo 2

El primer paso de la escena es comprobar el estado de la variable local *language_state*. Según su estado cargamos las variables que usaremos para crear la escena. Para cargar el contenido de esta escena llamamos a la función *new* del modulo externo *variables*. Al llamar a esta función le pasamos los parámetros para configurar un capitulo especifico. Esos parámetros son el capitulo y tema que queremos visualizar. Internamente la función *new* se encarga de seleccionar las variables correspondientes dependiendo del idioma seleccionado.

Tambien hacemos uso de la función *content* que se encarga de gestionar la barra de contenido dentro de los temas de los capitulos.

Cuando se crean los objetos de la escena, estos utilizan las variables cargadas anteriormente por lo que podemos usar una misma escena para ver todos los capitulos cuya escena es del Tipo 2, los capitulos con ejemplos.

Se debe mencionar que no es posible utilizar este tipo de escena para los 3 primeros capitulos aunque la similitud de la escena sea evidente. La razón es simple, para cargar el slideView es necesario utilizar un fondo de pantalla diferente para poder ver la sensación de transición del slideView dentro del recuadro gris.

3.2.3 Diseño de pantallas de la aplicacion

3.2.3.1 Pantalla de carga

Figura 3.5. Pantalla de carga de la aplicacion

La pantalla de carga contiene una imagen con el logotipo de la Universidad de Zaragoza. Los elementos de la vista son:

- Una imagen cargada desde un fichero denominado splash.png que se incluye en el directorio de imagenes de la aplicación.

En esta pantalla no hay opción de realizar acciones, existe un temporizador con una duración de 2sg que se encarga de cambiar de escena automáticamente tras pasar el tiempo al que esta programado.

Esta pantalla da paso a la pantalla de bienvenida de la aplicación.

3.2.3.2 Pantalla bienvenida

Figura 3.6. Pantalla de bienvenida de la aplicacion

La pantalla de bienvenida contiene el titulo de la aplicación. Los elementos que contiene la vista son:

- La vista contiene un fondo de pantalla negro.
- Una imagen cargada desde fichero que es las encargada de mostras el titulo de la aplicación, dependiendo del idioma el archivo cambia.
- Si la última vez que se ejecutó la aplicación se estaba visualizando un contenido de un capítulo de ejemplos o de diapositivas, pregunta si se desea volver a la última posición. Esto es útil si se ha suspendido la aplicación por una llamada u otro motivo. El archivo posición.dat es el encargado de almacenar esta información.

La pantalla a paso al Menú principal de la aplicación o a la última posición visualizada dentro de un capítulo.

3.2.3.3 Pantalla Menú Principal

Figura 3.7. Pantalla del Menu principal de la aplicacion

La pantalla del Menu Principal contiene el listado de los capítulos seleccionables asi como los accesos a los juegos.

El listado de los capítulos esta incluido en el modulo externo “variables.lua”, el formato es una tabla en la cual se incluyen todos los parámetros referentes a la tableView, que es el objeto encargado de mostrar la tabla. Los elementos que presenta la vista son:

- Listado de capítulos .
- Titulo de la aplicación.
- Imágenes de juegos en la parte derecha de la pantalla.
- Boton de Configuración.

Desde esta pantalla es posible acceder a cualquier capitulo de la aplicación asi como al menú de configuración.

3.2.3.4 Pantalla Capitulo Diapositivas

Figura 3.8. Pantalla de presentación de diapositivas

En la pantalla de presentación de diapositivas se puede visualizar uno de los 3 primeros capitulos seleccionables en el Menu Principal. Los elementos que contiene la vista son:

- Titulo de la aplicación(Can you….)
- Titulo del capitulo.
- Objeto slideViev(incluye eventos y funciones)
- Boton *Menu*.
- Botón *Previus*.
- Boton *Next*.

Desde esta pantalla mediante la utilización de los diferentes botones es posible o dirigirse al menú principal para seleccionar otro capitulo o bien ir cambiando la diapositiva con los botones de *next* y *previus*.

Existen 3 capitulos que utlizan esta misma forma y la escena utilizada es la misma para todos y cada uno de ellos, la Tipo 1.

3.2.3.5 Pantalla Introduccion Capítulo Ejemplos

Figura 3.9. Pantalla de introducción a un capitulo de ejemplos

En la pantalla de introducción a un capitulo de ejemplos se visualiza un pequeño párrafo que nos detalla que nos vamos a encontrar en el capitulo seleccionado. El contenido de la vista es:

- Titulo de la aplicación.
- Titulo del capitulo.
- Contenido de la introduccion.
- Dispositivo sobre el cual se van a visualizar los diferentes ejemplos.
- Botón *Play*.
- Boton *Menu*.

Cuando nos encontramos en esta pantalla podemos dirigirnos hacia varias escenas. Podemos volver al menú principal o podemos pulsar el botón *play* e iniciar el capitulo.

3.2.3.6 Pantalla Visualizacion Tema

Figura 3.10. Pantalla de visualización de un capitulo de ejemplos

La pantalla de visualización de un capitulo de ejemplos es similar a la anterior con la única diferencia que el espacio ocupado por la introducción al capitulo ahora esta reservado para ver las explicaciones sobre lo que aparece en el simulador. Esta pantalla es común a los 4 capitulos de ejemplos. Cada capitulo de ejemplos contiene en su interior diferentes temas que son tratados de forma ordenada respecto a la creación del videojuego de ejemplo. Enumeramos a continuación los elementos que contiene la vista:

- Titulo de la aplicación.
- Titulo del tema que estamos visualizando
- Espacio reservado para la explicación del tema.
- Dispositivo simulado sobre el cual se visualizan los diferentes ejemplos.
- Boton *menú*.
- Boton *configuracion*.
- Botones *next* y *previus*.
- Boton *play* y *pause*.
- Boton *content*.

Existen varias posibilidades en esta pantalla a la hora de realizar acciones. Los botones con flechas de dirección se utlizan para avanzar o retroceder en el tema.

Existe un botón *play* que al pulsarlo ejerce la misma función que la flecha *next* pero de forma automática, sin necesidad de estar pulsando cada vez que avanze el tema. Se detiene cuando llega a la ultima línea de las explicaciones. Al pulsar *play* este botón desaparece y el boton *pause* aparece para detener las explicaciones cuando precise. En ese instante se alternan los botones para reanudar la marcha al volver a pulsar *play.*

El botón *content* hace aparecer una barra con el contenido del capitulo distingido por temas. A continuación se muestra el contenido de los temas de cada capitulo.

Capitulo 4 – Visualizacion y eventos :

- Introduccion I
- Introduccion II
- Introduccion III
- Introduccion IV
- Formas
- Imagenes
- *Texto*
- Grupos
- Propiedades
- Eventos 1
- Eventos 2
- Eventos 3

Capitulo 5– Animacion y movimiento :

- Transition I
- Transition II
- MovieClip I
- MovieClip II

Capitulo 6 – Motor Fisico :

- Fisica I
- FIsica II
- Escenario
- Cuerpos
- Pelota
- Juego

Capitulo 7 – Audio y video :

- Audio I
- Audio II
- Video I
- Video II

A continuacio se muestra una imagen de pantalla cuando aparece la barra oculta de contenido.

Figura 3.11. Pantalla con la barra de contenido

La forma que se gestiona la barra es la siguente, hay tantos temas que no caben todos dentro de la pantalla por lo cual se ha incluido un slide sobre los temas que delizando el dedo movemos la barra de contenido hacia ambos lados. Al pulsar sobre un tema en concreto se carga la escena pertinente y se visualiza de la misma forma.

Para ocultar la barra de contenido sin cambiar de escena es tan fácil como tocar la pantalla por encima de la barra y esta desaparece para poder continuar viendo el capitulo.

3.2.3.7 Pantalla Configuración

Figura 3.12. Pantalla de configuración de la aplicacion

En esta pantalla se muestran las opciones de configuración de la aplicación. Los elementos que contiene la vista son:

- En la parte superior aparece el titulo de la escena.
- Las opciones que se permiten modificar son el idoma, el sonido y la animación.
- Todas las opciones se cambian pulsando sobre el valor. Se guardan automáticamente en el modulo externo “variables.lua”.

Las opciones que pueden modificarse son simples y sencillas. Cuando pulsamos sobre las selección de idioma este nos pregunta si estamos de acuerdo con cambiar de idioma y nos advierte que es necesario reinicia la aplicación, como vemos a continuación.

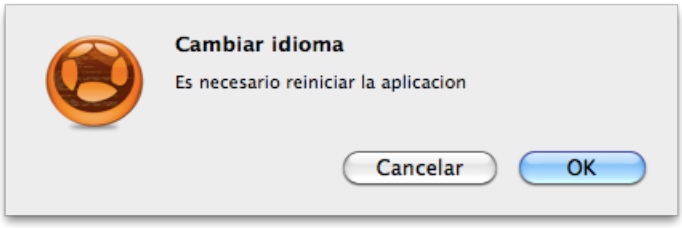

Figura 3.13. Pantalla de reinicio de aplicación.

La imagen no corresponde con la advertencia que nos muestra el dispositivo sino la que vemos en el simulador aunque solo cambia el diseño.

Para ocultar la pantalla de configuración arrastramos con el dedo hacia arriba y al soltarlo automáticamente la pantalla se oculta.

3.2.3.8 Pantalla de Menu Juegos

Figura 3.14. Pantalla de inicio del juego Bubble Ball

En esta escena podemos ver la pantalla de introducción de un juego, en este caso la del *Bubble Ball.* Para cargar un juego es necesario cargar una escena diferente por lo tanto existen tantas escenas como juegos. La estructura que mantienen los juegos es similar para todos aunque evidentemente la diferencia de contenido es grande. Es posible aprovechar el orden de carga de variables, de funciones y de eventos.

Los elementos de la vista que aparecen en la pantalla principal de los juegos son:

- Titulo del juego.
- Botones de *play*, *options* y *exit*.

3.2.3.9 Pantalla de Juegos

Figura 3.15. Pantalla del juego Space Shooter

En la pantalla de juegos podemos observar que aparecen 2 botones en las esquinas superiores. El botón de la esquina derecha nos permite volver al menú principal del juego.

Cuando pulsamos el botón superior izquierdo aparece la aplicación nativa del dispositivo para mandar un e-mail con un archivo adjunto en formato zip incluendo el juego que estamos visualizando para su posterior estudio.

3.3 Implementación

En esta sección se detalla cómo se ha implementado la aplicación.

Como se ha comentado anteriormente el diseño de la aplicación respecto a la transición de escenas se realiza mediante el modulo externo *DirectorClass*. A continuación mostramos la estructura de la aplicación y comentamos el sistema de archivos y directorios que rige la aplicación.

3.3.1 Estructura del programa

En el siguiente esquema se puede visualizar la estructura completa del programa y los ficheros externos que se utilizan en el.

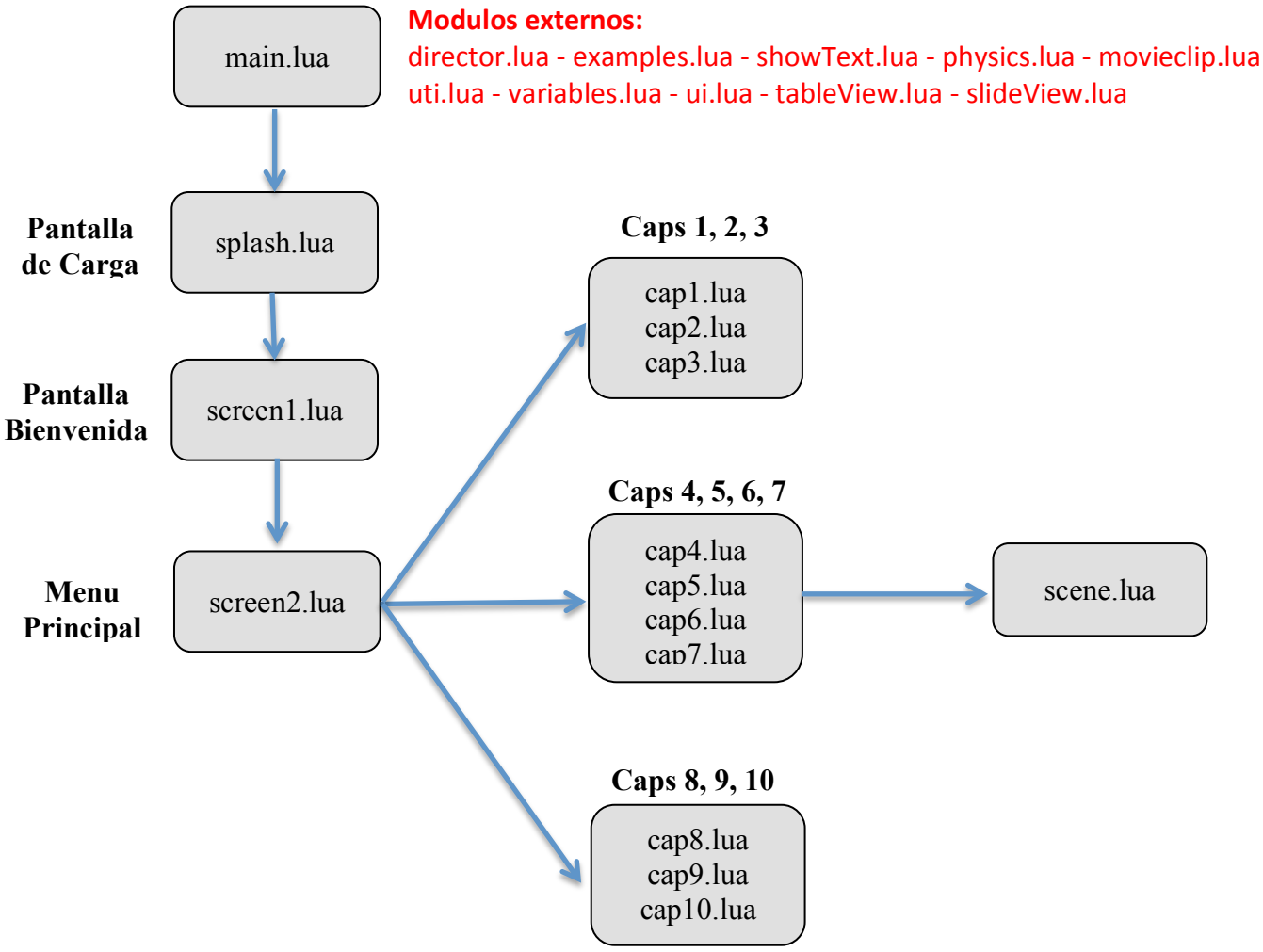

Figura 3.16 Estructura de escenas de la aplicacion

En los siguientes apartados se irán desglosando los módulos que componen la estructura del programa y los ficheros que se utilizan para su configuración.

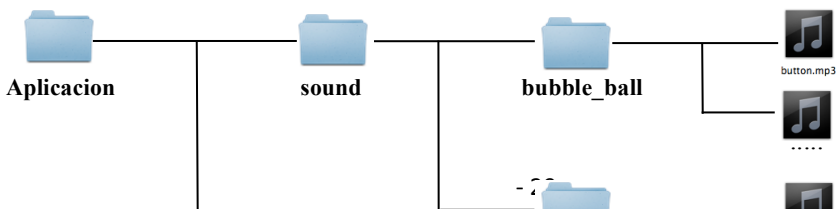

Figura 3.17. Estructura de directorios del proyecto en Corona SDK

La organización de los ficheros se localiza en una carpeta de proyecto principal, de acuerdo a la estructura establecida por Corona SDK, donde se encuentra el archivo principal de la aplicación *main.lua*, las escenas, módulos externos de código y directorios de imágenes y sonidos.

Los archivos de audio y sonidos están almacenados dentro de un directorio denominado *sound*.

Las imágenes se encuentran en el directorio *images*, dentro de este directorio las imágenes se encuentra clasificadas dependiendo la escena donde aparecen . Existen imágenes que son utilizadas por varias escenas como son los botones, fondos de pantalla y el titulo de la aplicación que aparece en todas las escenas.

Con respecto al fichero auxiliar, es creado dinámicamente por la aplicación en el directorio de documentos y definido en la constante *system.DocumentsDirectory* y que es usado para ficheros que necesitan permanecer entre varias sesiones de la aplicación.

3.3.2 Ficheros de configuración

Para la construcción de la aplicación es necesario un fichero de configuración que pasamos a explicar a continuación. Debe ser generado por el administrador y es necesaria su existecia para que la aplicación funcione con normalidad.

3.3.2.1 posicion.dat

Guarda la posición dentro de la escena que estamos visualizando. Se va actualizando al cambiar de escena. En caso de interrupción de la aplicación permite volver al punto previo al iniciar la aplicación de nuevo, en el caso de que estemos visualizando un tema dentro de un capitulo de ejemplos o dentro de unos de los capitulos de diapositivas. Contiene el nombre de la escena en la cual estamos situados. Este dato es pasado a nuestra función de chequeo de posición y este determina en que posición nos habíamos quedado anteriomente. Existen 2 posibilidades a la hora de guardar en el archivo. Si en la escena en la que estamos es necesario guardar la posición, escribimos en el archivo el nombre de la escena, por ejemplo *formas*. En el caso de que no sea necesario escribimos un “0”. Cuando la función chequea el archivo *posición.dat* sabe que si obtiene un “0” no es necesario preguntar la ultima posición conocida, en el caso que aparezca una cadena pregunta si queremos volver a esa escena. A continuación se muestra una pantall con la imagen que nos ofrece el simulador. No es la misma que aparece en el dispositivo, solo cambia gráficamente la forma de mostrarse.

Figura 3.18 Pantalla de acceso a la ultima posición guardada.

3.3.3 Ficheros de contenido

Los ficheros de contenido de la aplicación almacenan los datos para la generación de las escenas que contienen texto. En nuestro caso todos esos datos se encuentran dentro de la librería *variables.lua* y la explicamos a continuación.

3.3.3.1 variables.lua

El texto esta almacenado dentro del modulo externo *variables.lua*. La forma de selecionarlo es simple. Mediante la función *new(cap,tema)* selecionamos las cadenas de texto a mostrar y las rutas de los títulos de las escena que luego esta gestióna para que aparezcan correctamente.

Según el estado de la variable global *language_state* se carga el idioma de las variables. Podemos ver a continuación una fragmento del código de la librería *variables.lua,* exactamente la función *new(cap,tema)*

variables.lua

```
function new(cap,tema)

	if language_state == false then

	if cap == 1 then
		tema_tit = "images/titulos/spanish/introduccion.png"
		myImages = {"images/cap1/1.jpg", "images/cap1/2.jpg", "images/cap1/3.jpg", "images/cap1/4.jpg",
	"images/cap1/5.jpg", "images/cap1/6.jpg", "images/cap1/7.jpg", "images/cap1/8.jpg"}

	end

	if cap == 2 then
		tema_tit = "images/titulos/spanish/lua.png"
		myImages = {"images/cap2/1.jpg", "images/cap2/2.jpg", "images/cap2/3.jpg", "images/cap2/4.jpg"}
	end

	if cap == 3 then

		tema_tit = "images/titulos/spanish/coronaSDK.png"
		myImages = {"images/cap3/1.jpg", "images/cap3/2.jpg", "images/cap3/3.jpg", "images/cap3/4.jpg"}


	end


	if cap == 4 then

		titulo = {"Introduccion I","Introduccion II","Introduccion III","Introduccion
IV","Formas","Imagenes","Texto ","Grupos","Propiedades","Eventos 1", "Eventos 2", "Eventos 3"}
		numero_temas = 12

		if tema == 0 then

			tema_tit = "images/titulos/spanish/visualizacion.png"
			string_introduccion = "	Este capitulo nos introduce en la visualizacion de objetos asi
como los eventos que los controlan. Hemos elegido un juego standard conocido como ''Space Shooter'' para mostrar
las posibilidades que nos ofrece Corona SDK. Vamos a aprovechar el siguiente capitulo, Animacion y Movimiento,
para terminar de rematar el juego y darle una aparencia real de funcionamiento. Antes de comenzar a construir el
juego se incluye una introduccion sobre la creacion de objetos y eventos."

		end

		if tema == 1  then

			strings = {}
		strings[1] = "display.newRect(x,y,width,height)"
		strings[2] = "display.newRoundedRect(x,y,w,h,radius)"
```

```
                        strings[3] = "display.newCircle(x,y,radius)"
                        strings[4] = "display.newLine(x1,y1,x2,y3)"
                        strings[5] = "line:append(x3,y3)"
                        strings[6] = "object:setFillColor(R,G,B,transparency)"
                        strings[7] = "line:setColor(R,G,B,transparency)"
                        strings2 = {}
                        strings2[1] = "Ahora un rectangulo con los bordes redondeados:"
                        strings2[2] = "El siguiente es un circulo:"
                        strings2[3] = "Tambien es posible dibujar una linea:"
                        strings2[4] = "Con la siguiente instruccion podemos incluir otra linea al ultimo punto del
objeto anterior:"
                        strings2[5] = "Para cambiar el color de un objeto:"
                        strings2[6] = "El metodo de la linea es distinto:"
                        y = {180,255,325,395,505,575,645}
                        y2 = {120,225,295,365,445,545,615}
                        tema_tit = "images/titulos/spanish/introduccion1.png"
                        string_introduccion = "Comenzamos la introduccion creando objetos de pantalla, en
primer lugar un rectangulo:"
                        escena_anterior = "cap4"
                        escena_siguiente = introduccion2
                        fin = 7
……….……
……..……..
……..……..
```

3.3.4 Módulos o librerías

Como se ha comentado en la estructura del programa, a partir del módulo principal, main.lua, se han cargado una serie de módulos externos o librerías que nos permiten acceder a funciones y a utliidades que a priori Corona SDK no permite y que nos da un plus de comodidad a la hora de programas ya que podemoa adaptar librerías especificas para otros proyectos a cualquier otro.

La forma de incluir un modulo en un proyecto es la siguiente:

```
local ejemplo = require(“ejemplo”)
```

Dentro del modulo es necesario incluir la siguiente línea para que Corona SDK entienda lo que es.

```
module(..., package.seeall)
```

El archivo principal, *main.lua,* contiene la llamada a todos esas librerías pero la librería imprescindible para nuestra aplicación es *director.lua*. Vamos a pasar a explicar como funcionan los modulos y en primer lugar comenzamos con el mas importante.

3.3.4.1 director.lua

El funcionamiento de este modulo es muy sencillo. Es necesario que el main.lua contenga la siguente parte de código para añadir la librería. Y cada escena es necesario que siga un patrón de código para que todo valla perfecto.

Este es el código que presenta el archivo principal *main.lua*:

```
local mainGroup = display.newGroup()

local main = function ()

	mainGroup:insert( director.directorView )

	director:changeScene("splash")

	return true
end
main()
```

En este caso llamamos a la escena *splash.lua*.

Lo primero que hace el codigo es crear un grupo de objetos. En el incluiremos los objetos de pantalla que creemos y que serán llamados “hijos”.

Luego creamos la función principal de la aplicación que mas tarde llamaremos para iniciarla.

Para cambiar de escena simplemente utilizamos la siguiente instrucción:

```
director:changeScene(escena)
```

Donde escena es un archivo como por ejemplo: *scene.lua*

El formato de *scene.lua* es diferente al *main.lua* a continuación se explica con mas detalle tras mostrar un ejemplo de escena:

```
new = function ()
	local localGroup = display.newGroup()
	-- Codigo escena
	return localGroup
end
```

Creamos un grupo local que solo esta dentro de la escena. Es necesario incluir cada objetos de pantalla que se cree dentro del grupo porque de cara a una gestión correcta de la memoria en la aplicación, que como se ha comentado, es importante en dispositivos móviles, se ha utilizado una estructura basada en objetos de visualización o *display objects*. Al cambiar de escena cada uno de los objetos incluidos son eliminados y resolvemos del problema de la memoria de un plumazo. Tambien son eliminados los posible eventos asociados a los objetos.

Las posibilidades que nos ofrece esta librería para cambiar de escena son las siguientes:

```
director:changeScene(params, escena, efecto)
```

params = estos parámetros estarán disponibles en la siguiente escene. Su formato solo puede ser el de una tabla. Ha ido predeterminado por el autor.

escena = archivo con extensión punto lua.

efecto = ha sido preconfigurado por el autor, como ejemplos *downFlip, fade….*

3.3.4.2 util.lua

En ella estan incluidas ciertas herramientas que utilizamos como complemento de las librerias de lua y las propias de Corona SDK. A continuacion enumeramos todas las funciones incluidas en este librería:

function ParseCSVLine (line,sep)

Esta función lo que hace es separar el contenido de una cadena que esta separada por un carácter especifico en varias cadenas dentro de una tabla.
Parametros de entrada:

line = linea con valores separados con un caracter
sep = caracter separador. Si no se especifica se usa la coma ','

Parametro de salida:

Devuelve una tabla de strings con los parámetros que antes estaban separados.

loadValue = function(strFilename)

Esta función permite cargar en una cadena el contenido de un archivo.

Parametros de entrada:

srtFilename = archivo que va a ser procesado

Parametro de salida:

Devuelve un string con el contenido del archivo

saveValue = function(strFilename, strValue)

Esta función guarda el contenido de una string dentro de un archivo

Parametros de entrada:

srtFilename = archivo que va a ser procesado

srtValue = string que va a ser guardada

Parametro de salida:

No tiene.

getPosicionGuardada = function ()

La función se encarga de comprobar si hay alguna posición guardada al entrar en la aplicación de nuevo.

Parametros de entrada:

No tiene.

Parametro de salida:

Devuelve si hay capitulo guardado y en su caso un string con el nombre del capitulo guardado.

crearConfiguracion = function (language,grupo)

Esta función crea el menú de configuración de la aplicación.

Parametros de entrada:

language = idioma que debe mostrar el menú
grupo = el grupo local de la escena

Parametro de salida:

Devuelve un grupo, que no es mas que el propio menú. Con su eventos, botones…

3.3.4.3 movieClip.lua

Este modulo es propio de Corona SDK pero no viene incluido en la aplicación y es necesario hacerlo manualmente. Es útil para crear animaciones. Hemos explicado su funcionamiento en el tema de animación y movimiento y lo hemos utilizado en algún juego. Para agregarla al proyecto simplemente basta con copiar el archivo movieClip.lua dentro de nuestra carpeta de aplicación.

3.3.4.4 ui.lua

Otro modulo propio de Corona SDK no incluido inicialmente. Sirve para crear botones con animación. Es fácil de usar. Simplemente permite crear el botón al dale como parámetros las imágenes y la funciona asociada. Aquí un ejemplo:

```
local buttonMenu = function ( event )

        -- Accion a realizar

end

btmenu = ui.newButton{
                default = “menu.png",
                over = "menu_over.png",
                onEvent = buttonMmenu,
                id = "btmenu"
                }
```

3.3.4.5 showText

Este modulo ha sido creado especificacmente para este proyecto. Lo usamos para darle la animación a una cadena dentro de los capitulos de ejemplo. Su funcionamiento es sencillo. Le puedes pasar como parámetro un string y te carga cada letra del string individualmente. Es posible configurar parámetros como el tiempo entre letras, longitud del texto, posiciones. La dificultad de este modulo radica en que ha sido necesario individualizar el tamaño de cada letra para que al mostrarlas todas juntas den una sensación de coesion y parezca que es un string y una suma suma de varios. A continuación se detalle la función.

```
function draw( string , x , y , color , tamaño, anchura , tiempo , sonido , cap, tema1, id , tope, flag)
```

Parametros de entrada:

string = cadena a mostrar
x, y = posiciones del comienzo del string
color = los colores posibles son “blanco”, “gris” y “negro”
tamaño = de la fuente
anchura = el texto cambia de línea
tiempo = entre letras
sonido = se reproduce cuando imprime una letra
cap , tema1, id = esta relacionado con el tema que visualizamos
tope = cada tema visualiza un numero de veces este complemento
flag = depende de esta variable para comportarse de una forma u otra

Parametro de salida:

Devuelve 2 grupos. El primero con el string cargado y el segundo que devuelve es el ejemplo que se carga dentro de la funcion

3.3.4.6 tableView.lua

Este es necesario para que funcione la tabla de capitulos del menú principal. Es un widget que proporciona Corona SDK y va muy bien para este tipo de objetos. Se agrega mediante el archivo en la carpeta. Le pasamos los parámetros de configuración mediante una tabla de strings donde se incluyen los títulos y subtitulos de la misma. Es nuestro caso se ha utilizado una tableView modificada para mostrar una imagen en el lado izquierdo de cada línea.

3.3.4.7 slideView.lua

Otro widget que nos facilita Corona SDK. En nuestro caso se ha modificado internamente para adaptarlo a nuestras necesidades. Lo hemos utilizado en los capitulos de diapositivas. Nos permite deslizar con el dedo entre las diferentes imágenes hacia un lado como para otro. Es necesario incluir el archivo para agregarlo al proyecto.

3.3.4.8 examples.lua

Esta librería es la responsable de cargar los ejemplos de los capitulo que se usan para aprender a programas. Incluye una función que al pasarle unos parámetros nos devuelve un grupo con el contenido de lo que le estamos solicitando. A continuación se muestra la función que usamos:

```
function new(cap,tema,id)
```

A través de los parámetros la función sabe que tiene que mostrar. En su interior hay un entramado de condicionales que son las encargadas de elegir el objeto a mostrar.

3.3.4.9 variables.lua

La ultima librería que pasamos a explicar es sin duda es la mas importante después de la *directoraClass* dado su contenido. Anteriomente la hemos comentado como librería de contenido. En ella se almacenan todas las variables globales que aparecen en la aplicación asi como la funciones que cargan el contenido en la mayoría de las escenas. Ademas están guardadas también las tablas que pasamos como parámetro a una escena cuando la cambiamos. Vamos a enumerar la funciones que la ocupan y para que sirven cada una de ellas asi como todas las variables globales que aparecen.

Variables Globales:

```
language_state = false  -- Estado del idioma
sound_state = true  -- Estado del sonido
animation_state = true  -- Estado de la animacion
estado_temasGroup = false  -- Estado del grupo temasGroup
```

Funciones:

```
new = function (cap, tema)
```

Es la encargada de cargar el contenido a mostrar en una escena. Como vemos los parámetros de carga son necesarios para que la función sepa que debe mostrar. Esta función también detecta el idioma del string y devuelve el correcto.

```
new2 = function ( )
```

Se ocupa de cargar la pantalla de configuración. Dependiendo de estado de la variable *language_state* carga un idioma u otro.

```
new2 = función ()
```

En ella esta incluida la tabla necesaria para el objeto tableView. Tambien diferencia entre el idioma que necesitemos.

```
new4 = función ()
```

Esta función también se utilizar en la pantalla de configuración y es la encargada de determinar en que posición debe situarse los botónes de activado de sonido y de animación.

```
new5 = function ()
```

Esta función la utilizamos en la pantalla de bienvenida. Determina que idioma debemos elegir a la hora de mostrar el titulo y los textos.

```
content = function(cap,tema,tema_seleccionado)
```

Crea todos los eventos relacionados con la barra Content. Le damos parámetros dado que cada capitulo es distinto y es necesario caracterizarlo individualmente.

Tablas de parámetros:

Como hemos comentado anteriomente podemos pasar parámetros entre escenas pero solo mediante tablas. Pues eso es lo que hacemos con estas tablas. Estan configuradas de tal forma que cuando llamas a la escena *scene.lua* le pasas los paremetros necesarios para configurar esa escena. La escena es común para todos los capitlos pero su contenido no, por lo que después de pasar estos parámetros llamamos a la function de la librería variables con esos parámetros y asi conseguimos cargar lo que nos interesa.

3.3.5 Titulos de la aplicacion

Para realizar los títulos de la aplicación se ha utlizado un recurso Web gratuito y que aparece en http://cooltext.com/ Es posible generar miles tipos de fuentes para títulos. A continuación se muestra una pantalla de lo pasos para genenar un titulo de la aplicación.

Figura 3.19. Primer paso para creación de titulo

Lo primero es seleccionar el estilo del titulo que queremos. Pulsamos encima del que mas nos guste.

Figura 3.20. Segundo paso para creación de titulo

En el siguiente paso introducimos el titulo en el cuadro de texto. Podemos cambiar la fuente del titulo asi como el color.
Elegimos el formato que queremos de la imagen y pulsamo el botón render Logo para generar el titulo.

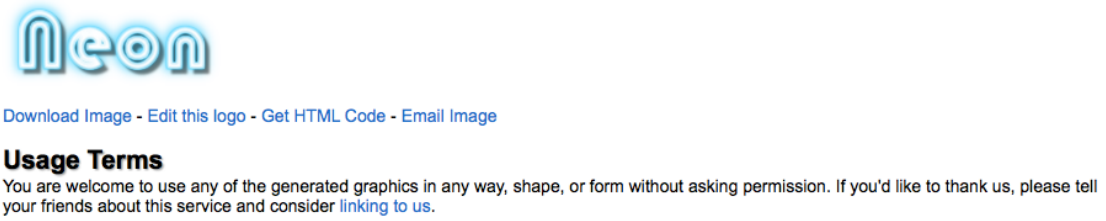

Figura 3.21. Aspecto del titulo definitivo

3.3.6 Adaptación dinámica de contenido

Figura 3.22. Dispositivo móvil Ipad

Para comenzar es necesario decidir el tamaño de contenido, con independencia de las dimensiones de la pantalla. Este método proporciona el sistema de coordenadas para el código de las aplicaciones realizadas con Corona SDK, que será independiente de la cantidad real de píxeles en la pantalla del dispositivo.

Para esto se define el tamaño que queremos al que se adapte nuestra aplicación, el tamaño del contenido, en el archivo de la aplicación ubicado *config.lua*. El código no tiene por qué conocer el tamaño real de la pantalla.

Vamos a configurar nuestro archivo *config.lua*, este se encarga de escalar el contenido de la pantalla del dispositivo que visualiza la aplicación debido al modo de escalado (parámetro *scale* definido en *config.lua*)

3.3.6.1 config.lua

Hay varios modos:

- *letterbox:* es el modo más común, ya que no se recorta el contenido aunque por el contrario puede mostrar zonas negras. En la aplicación se pueden detectar esas zonas y definir un contenido genérico, por ejemplo.

- *ZoomEven*: En este modo, el contenido se escala para llenar completamente la pantalla, aunque puede que alguna parte de la pantalla vea recortados los bordes. Esto significa que nunca habrá zonas, pero puede quedar parte del contenido fuera de la pantalla.

- *ZoomStretch*: El contenido se escala para completar totalmente la anchura y altura de la pantalla. No se pierde contenido pero se puede distorsionar si el dispositivo tiene una relación de aspecto diferente de su contenido. También puede parecer extraña cuando se gira, ya que la distorsión es determinada por la orientación inicial.

- *sin parámetro scale*: Esto desactiva el modo de escalado de contenidos, y produce el mismo resultado que no tener un archivo config.lua.

El fichero *config.lua*, permite personalizar la configuración por dispositivos.

config.lua

```
-- config.lua
-----------------------------------
-- Autor: CARLOS LORENZO PARICIO --
--         Version: 1.0          --
-----------------------------------

application =
{
	content =
	{
		width = 768,
		height = 1024,
		scale = "letterbox",
		fps = 30,
		antialias = true,

	},
}
```

Como podemos observar hemos utilizado el modo normal o *letterbox.* Configuramos las dimensiones de la pantalla para nuestro dispositivo. La pantalla del Ipad tiene una resolución de 768 por 1024 pixeles. El parámetro *fps* establece que cada 30 frames por segundo se actualiza el entrerframe de la aplicación.

Respecto al paremtetro de configuración *antialias*, Corona SDK utiliza un software anti-aliasing para los objetos vectoriales. Anteriormente este sistema estaba implementado por defecto pero ahora aparece deshabilitado y es necesario activar, este mejorará considerablemente el rendimiento de objetos vectoriales y debería haber poca diferencia visual en los dispositivos más actuales que presentan pantallas de alta definición.

3.3.6.2 build.settings

Este archive es necesario que aparezca en la raiz de nuestro proyecto si queremos configurar ciertos parametros de nuestra aplicacion.

build.settings

```
--build.settings--
-----------------------------------
-- Autor: CARLOS LORENZO PARICIO --
--              Version: 1.0          --
-----------------------------------

settings = {
       orientation =
       {
              default = "landscapeRight",

              supported = { "landscapeLeft", "landscapeRight" },

       },
}
```

Como puede observarse nuestra orientacion por defecto esta configurada como *landscaprRight*. Esto quiere decir la aplicacion esta diseñada para funcionar horizontalmente y que ademas admite girarla como se observa en el parametro *supported*. Es posible incluir opciones *portrait,* portraitUpsideDown o *landscape*.

3.3.7 Pruebas y verificación

Antes de decir que la aplicación esta terminada es necesario que esta sea sometida a un periodo de prueba y comprobación de su funcionamiento.

Para ello hemos elegido la forma de tradicional de hacerlo y es ir probando todas las opciones posibles que admite cada escena de la aplicación.

Dado que hay escenas que son similares el trabajo se simplifica y es posible reducir bastante el tiempo de verificación.

Tras someter a prueba la aplicación se detectan fallos de funcionamiento que son anotados en un blog de ensayos y que mas tarde son analizados mediante el simulador corrigiéndolos a medida que van aparenciendo.

En nuestro caso ha aparecido un fallo bastante grave y para el que ha sido necesario bastante tiempo para encontrar su solución. A continuación se explica de que fallo se trata y se comenta por encima los fallos comunes y de fácil solución.

El fallo al que nos referimos nos ha costado detectarlo porque en el simulador no aparece pero en el dispositivo si. Asi es difícil detectar fallos y solo haciendo pruebas detalladas es posible corregirlos. En este caso el problema procede de las imágenes que mostrarmos en en pantalla, particularmente en los botones. Tras diseñar los botones con un programa de

diseño conocido nos dimos cuenta que al construir la aplicación aparecia un error que no conseguíamos corregir. Nos dimos cuenta que eliminando el objeto el error desaparecia y al proceder a cambiar la imagen a mostrar el fallo ya no aparecia mas.

Otros fallos comunes son las rutas de imágenes. Suelen esta equivocadas y simplemente basta con corregirlas o colocar los archivos imagen en la ruta correcta.

Tambien son comunes los errores tipográficos. Al cometer un error en la programación detecta que algo no va bien y precisa corrección.

3.4 Tareas de administración

Una vez finalizadas las tareas de desarrollo del código de la aplicación, se deben realizar una serie de tareas de administración para poder completar la funcionalidad.

3.4.1 Distribuir aplicaciones

Una vez que se tienen los elementos para construir la aplicación y ha sido probada, se deben realizar las tareas para construir la aplicación.

En el fichero build.settings se pueden añadir una serie de valores opcionales para definir aspectos de la construcción. Se utiliza para establecer la orientación de la aplicación y el comportamiento de auto-rotación, junto con una variedad de plataformas específicas parámetros de construcción.

Para el caso de dispositivos con iOS la tareas necesarias para obtener la aplicacion:

1 - Developer account* y *Developer Certificate
Lo primero es inscribirse en el iPhone Developer Program de Apple en la siguiente dirección:

http://developer.apple.com/iphone/program/
Luego es necesario solicitar un certificado de desarrollador (Developer Certificate)

2 - Keychain certificate
Una vez que se han inscrito en el programa de desarrolladores, se debe utilizar la herramienta de "*Keychain access*", ubicado en la carpeta de servicios con el fin de crear una solicitud de certificado. Esto se utiliza para autenticar su equipo.

3 - Añadir un dispositivo
Se debe registrar un dispositivo para el que se va a construer la aplicación por lo que se necesita el número *Uniques Device Identification* (UDID).

4 - App IDs

Con el fin de obtener perfiles de aprovisionamiento, primero se debe crear una ID de la aplicación. El ID de la aplicación permite a una aplicación comunicarse con el servicio de notificaciones push y / o cualquier otro hardware externo que tiene la aplicación. Un ID de la aplicación consiste en un 10 caracteres " Bundle Seed ID" como prefijo generado por Apple, y un sufijo " Bundle Identifier" creado por el administrador del equipo en el portal del programa.

5 - Provisioning profiles

Hay tres tipos de perfiles de aprovisionamiento para el programa de iPhone: Ad Hoc, desarrollo y distribución. El perfil de distribución es lo que se utiliza para crear una aplicación con el propósito expreso de ponerlo en la App Store.

6 - Construir la aplicación

La construcción de su aplicación utilizando Corona es un proceso sencillo una vez que haya perfiles de aprovisionamiento en su lugar. Para construir la aplicación, abra el simulador de Corona y abrir un proyecto (seleccione File> Open ... para abrir el proyecto). A continuación, seleccione File> Build> IOS ... Aparecerá el siguiente diálogo:

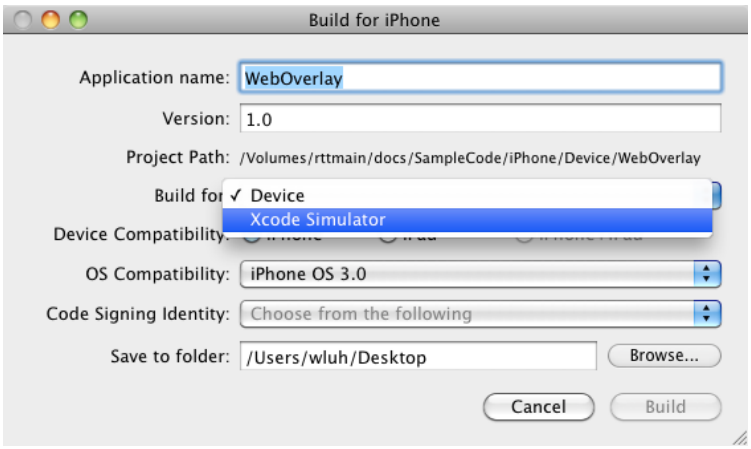

Figura 3.23. Construcción aplicación en iOS

Una vez que haya introducido toda la información pertinente, se pulsa el botón 'Build'. Una vez que Corona ha completado la construcción, la salida será una aplicación que se guarda en un directorio.

7 - Subir la aplicación a la App Store

Una vez que se han construido y probado la aplicación con Corona, es hora de subirlo a la *App Store*. Para ello, se tiene que acceder a "iTunes Connect" en el *iPhone Dev Center*. Si la aplicación va a ser de pago, se tienen que aceptar los contratos de Apple.
Después de rellenar toda la información necesaria, subir la aplicación, al menos un pantallazo, etc, la aflicción entra en el proceso de revisión por parte de Apple.

4 Conclusiones

En este capítulo se ofrecen una serie de conclusiones acerca del trabajo realizado para la consecución de los objetivos que se habían marcado en el Proyecto fin de carrera, tanto en el aspecto técnico de la aplicación desarrollada como en lo personal. Además, se proponen una serie de posibles mejoras de cara a futuras ampliaciones de la aplicación.

En un futuro la actualización de esta aplicación podría ser una nueva propuesta para un proyecto Fin de Carrera para que otro alumno que pueda estar interesado la retome y complete una serie de mejoras y actualizaciones que mas tarde enumeraremos.
Los objetivos planteados en el análisis de requisitos del proyecto han sido cumplidos con éxito.

Entre las dificultades del proyecto ha estado el aprendizaje de la programación en Corona SDK. Se desconocia totalmente la aplicación y se ha necesitado un curso de formación impartido por la Universidad de Zaragoza para iniciarse en la plataforma.

Finalmente en este proyecto se ha optado por obtener un resultado final optimizado para el iPad dado que este es el dispotivo que el departamento posee para realizar las pruebas de testeo.

La realización de este proyecto ha supuesto una ampliación de mis conocimientos de programación sobre dispositivos móviles, sobre todo en cuanto al desarrollo de aplicaciones multiplataforma, una tendencia que actualmente se está imponiendo en el mercado.

Me ha permitido conocer distintas plataformas para el desarrollo de aplicaciones, cada una con sus características y particularidades así como profundizar en Corona SDK, una herramienta, todavía en evolución, que en un futuro próximo será un referente en este campo.

En definitiva, el desarrollo del proyecto además de incrementar el conocimiento sobre herramientas y tecnologías, ha supuesto un reto personal importante una vez superadas todas las asignaturas de la carrera.

4.1 Mejoras y ampliaciones

Desde un principio, el diseño del sistema ha sido orientado a futuras ampliaciones, no solo de los contenidos como ya se puede realizar, sino de la aplicación en si.

Así pues, algunas de las posibles mejoras serían:

- Mejorar la visualización del contenido.
- Optimizar la visualización para todo tipo de dispositivos móviles y tabletas.
- Actualizaciones del contenido desde la propia aplicacion.
- Los temas para una posible actualización del contenido pueden ser:

    - Scrolling lateral.
    - Niveles en los juegos construidos.
    - Efectos graficos.

- Multijugador.
- Openfeint.
- Publicidad.

5 Bibliografía y referencias

- 'Corona SDK. Language and API Reference'. Ansca mobile, 2011
- Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes. 'Lua 5.1 Reference Manual' . Lua.org, 2007
- Pagina de recursos de documentación de corona SDK: http://developer.anscamobile.com/resources
- Índice TIOBE de utilización de lenguajes de programación, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
- http://www.imatica.org/bloges/2011/04/190486382011.html

6 Plataformas de programación en dispositivos móviles

6.1 Introducción

La primera cuestión en este proyecto es determinar las distintas alternativas y herramientas para el desarrollo de aplicaciones nativas para móviles. Cada plataforma tiene su propio lenguaje, herramientas de desarrollo y APIs con los que crear aplicaciones.

Otra opción es la creación de aplicaciones Web para móvil frente a aplicaciones nativas de los dispositivos.

La solución más óptima debe pasar por una herramienta "write once, run everywhere", es decir, un software con el que sea posible programar con un lenguaje determinado y que, además, permita que la aplicación funcione en varios dispositivos.

Este tipo de herramientas se están popularizando y se han comparado cuatro de ellas:

- Adobe Air Mobile (Adobe)
- PhoneGap (Open source)
- Appcelerator (Titanium)
- Corona SDK (Ansca mobile)

6.2 Dispositivos y sistemas operativos

El mercado de los dispositivos móviles tiene una gran variabilidad y necesita de una constante y lógica adaptación. Las características que lo definen son:

- Existencia de una gran heterogeneidad, en el que prácticamente para cada una de las principales marcas de dispositivos dispone de su propio sistema.

- Una visión de un mercado altamente volátil con cambios muy significativos en periodos muy cortos. Las cifras disponibles en cualquier "time to market", resulta difícilmente comparable o extrapolar con el de desarrollos realizados en otras épocas.

Los sistemas operativos móviles más utilizados actualmente son:

Android : es un sistema operativo basado en Linux diseñado originalmente para dispositivos móviles, tales como teléfonos inteligentes, pero que posteriormente ha expandido su desarrollo para soportar otros como tablets, reproductores MP3, netbooks, PCs e incluso televisores. Android, al contrario que otros sistemas operativos para dispositivos móviles como iOS o Windows Phone, se desarrolla de forma abierta y se puede acceder tanto al código fuente como al listado de incidencias donde se pueden ver problemas aún no resueltos y reportar problemas nuevos.

iOS : iOS (anteriormente denominado iPhone OS) es un sistema operativo móvil de Apple desarrollado originalmente para el iPhone, siendo después usado en todos los dispositivos iPhone, iPod Touch e iPad. Es un derivado de Mac OS X. La interfaz de usuario de iOS se basa en con el concepto de manipulación mediante gestos multitáctil. Los elementos de la interfaz se componen por deslizadores, interruptores y botones. La respuesta es inmediata y se provee de una interfaz fluida.

BlackBerry OS: es un sistema operativo móvil desarrollado por Research In Motion (RIM) para sus dispositivos BlackBerry. El sistema permite multitarea y tiene soporte para diferentes métodos de entrada adoptados por RIM para su uso en computadoras de mano, particularmente la trackwheel, trackball, touchpad y pantallas táctiles.

Windows Phone: es un sistema operativo móvil compacto desarrollado por Microsoft, y pensado para su uso en dispositivos móviles. Se basa en el núcleo del sistema operativo Windows CE y cuenta con un conjunto de aplicaciones básicas utilizando las API de Microsoft Windows. Está diseñado para ser similar a las versiones de escritorio de Windows estéticamente.

Symbian: es un sistema operativo que fue producto de la alianza de varias empresas de telefonía móvil, entre las que se encuentran Nokia, Sony Ericsson, Psion, Samsung, Siemens, Arima, Benq, Fujitsu, Lenovo, LG, Motorola, Mitsubishi Electric, Panasonic, Sharp, etc. El objetivo de Symbian fue crear un sistema operativo para terminales móviles que pudiera competir con los existentes en su momento. Se está dejando de utilizar aunque su presencia es todavía importante.

Java ME: La plataforma Java Micro Edition, o anteriormente Java 2 Micro Edition(J2ME), es una especificación de un subconjunto de la plataforma Java orientada a proveer una colección certificada de APIs de desarrollo de software para dispositivos con recursos restringidos. Está orientado a productos de consumo como PDAs, teléfonos móviles o electrodomésticos.

MeeGo: es una plataforma basada en Linux resultado de la unión de los sistemas operativos Maemo y Moblin, con el que Intel y Nokia pretendían competir con el sistema Android de Google. El proyecto está auspiciado por la Linux Foundation.

Mobile web (HTML and JavaScript) : desarrollo de aplicaciones web para móviles.

Qt: es una biblioteca multiplataforma ampliamente usada para desarrollar aplicaciones con una interfaz gráfica o para programas sin interfaz gráfica como herramientas para la línea de comandos y consolas para servidores. Es producido por la división de software Qt de Nokia. Qt es utilizada en KDE, un entorno de escritorio para sistemas como GNU/Linux o FreeBSD, entre otros. Qt utiliza el lenguaje de programación C++ de forma nativa, adicionalmente puede ser utilizado en varios otros lenguajes de programación a través de bindings.

En la siguiente figura se puede apreciar la evolución del porcentaje de utilización de cada sistema operativo. Este estudio se ha realizado en los últimos 4 años y esta basado en encuestas online y entrevistas a desarrolladores de 75 países y a ejecutivos que trabajan en la industria móvil en organizaciones comerciales y agencias digitales.

Se observa que Android e iOS son los más utilizados con un 53% y un 15% respectivamente. El incremento del sistema operativo Android de Google es debido al crecimiento exponencial de las ventas de dispositivos con ese sistema operativo.

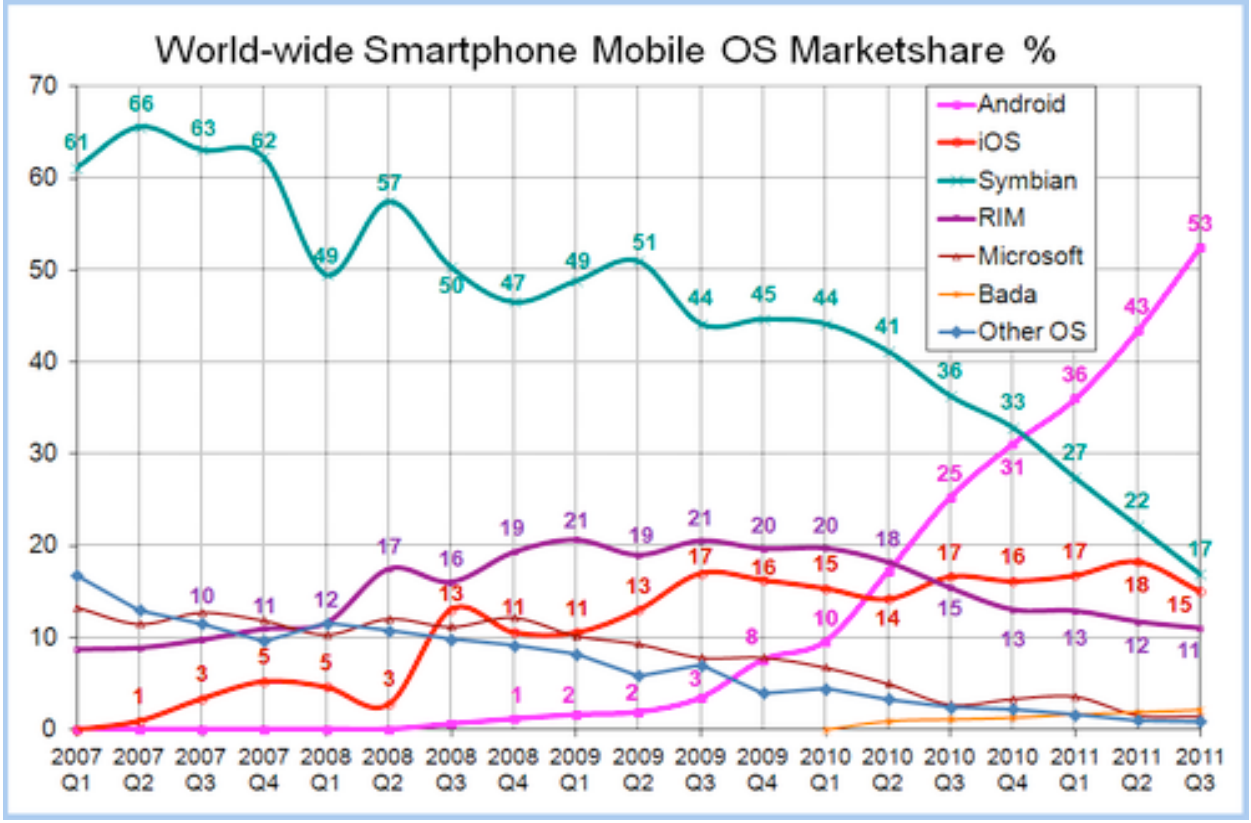

Figura 3.1. Evolucion de la utilización de S.O. móviles.

Este mismo estudio muestra que los sistemas operativos que los desarrolladores están pensando en utilizar para sus desarrollos futuros son Android e iOS. Así mismo, Symbian es la plataforma con las mayor tasa de abandono por parte de los desarrolladores. Casi el 60% de los desarrolladores utilizando actualmente Symbian está pensando en cambiar a otros. Tambien cabe destacar el descenso que la compañía RIM ha sufrido probablemente por los fallos que sus servidores sufrieron inexplicablemente a lo largo del año 2011.

6.3 Aplicaciones web versus aplicaciones nativas

Todos los dispositivos móviles tienen uno o varios navegadores y muchos de ellos ya soportan HTML5, por lo que una opción de desarrollo de aplicaciones es simplemente, crear una aplicación web y usarla desde el navegador de uno de estos dispositivos.

Ventajas:

- ***Diseño más sencillo***: Es suficiente con solo hacer un diseño adaptado a una pantalla y resolución pequeñas, simplemente adaptando un CSS por cada dispositivo. Además, las aplicaciones web se pueden "tunear" para que parezcan aplicaciones nativas: icono de aplicación, pantalla completa, splash screen, barra de estado, etc.

- ***Implementación***: las aplicaciones web pueden ser desarrolladas en cualquier tecnología de servidor, así que podemos usar cualquier lenguaje que ya se conozca (Java, Grails, Php, Ruby, Python,…) con la seguridad de que la aplicación se verá prácticamente igual en todos los terminales.

- ***Seguridad***: Se controla el acceso a la aplicación y se puedes actualizar sin necesidad de acción del usuario.

Desventajas:

- ***Acceso APIs del dispositivo***: no hay acceso completo a todas las APIs nativas del móvil. Aunque la cámara y el micro son accesibles con Flash, todos sabemos que esa tecnología está vetada en iOS. Desde HTML5 y Javascript, es posible acceder a las coordenadas del GPS, pero no en tiempo real ni de la misma manera que si pudiéramos acceder a la API del móvil directamente. Y lo mismo con otras funciones de los dispositivos.

- ***Mayor implicación de usuario***: para usar una aplicación web en un móvil, es necesario que el usuario abra el navegador y teclee la dirección, ya sea porque la sepa, la haya encontrado en Google. Una vez abierta la aplicación, debe añadirla a favoritos o, mejor todavía, crear un icono de acceso directo en el móvil para acceder a ella más tarde. Es mucho más fácil descargar una aplicación y que aparezca directamente como un icono en nuestro móvil.

- ***Velocidad de ejecución y conectividad***: Ejecutar una aplicación nativa es menos costoso que renderizar HTML e interpretar JavaScript. Además cada

petición desde nuestra aplicación implicará un acceso contra nuestro servidor. Una aplicación nativa tiene todos los recursos y procesos guardados en local, y solo accede al servidor para obtener o enviar datos si es que los necesita. Por tanto, una aplicación web no tiene la fluidez y velocidad de manejo que una aplicación nativa.

- ***Monetización***: es más fácil que un usuario pague por nuestros servicios si simplemente cobramos por nuestra aplicación al descargarla del App Store, que no hacer que el usuario se tenga que registrar y efectuar el pago en nuestra web, introduciendo manualmente todos sus datos como el número tarjeta, dirección, etc. Apple tiene su App Store para IOS y MacOS, Google su Chrome Web Store, Google Apps Marketplace y el Android Market, Amazon su Amazon Appstore, incluso hay markets alternativos como OpenAppMkt. Cada vez más empresas invierten en crear un entorno fácil y cómodo para que el usuario pueda descargar, probar y comprar aplicaciones, repartiéndose los beneficios.

6.4 Desarrollo aplicaciones móviles nativas para iOS y Android

Como se ha indicado anteriormente, cada plataforma tiene su propio lenguaje, herramientas de desarrollo y Apis con los que crear aplicaciones. Se detallan los dos más importantes, iOS y Android.

IOS

Utiliza el lenguaje Objective-C, aunque también se puede utilizar C/C++. Con este lenguaje podemos crear aplicaciones para Iphone, Ipad y Ipod Couch en sus distintas versiones. Hay también distintas versiones de IOS pero todas ellas se programan usando el mismo lenguaje. Objective-C tiene una sintaxis un tanto compleja de escribir y de leer.

Xcode es el entorno de desarrollo oficial de Apple. Con él, podemos crear aplicaciones de escritorio para Mac y para IOS. También se pueden utilizar editores de texto plano y compilar las aplicaciones “a mano”, es una tarea casi imposible.Se necesita de un ordenador Mac con el IPhone SDK y para distribuir aplicaciones en el App Store y para poder probar las aplicaciones desarrolladas en nuestro propio Iphone/Ipad, es necesario adquirir una licencia de desarrollador.

Android

Android es menos restrictivo. El lenguaje que se utiliza para programar aplicaciones es Java y tiene un SDK multiplataforma que funciona en Windows, Linux y Mac.
Se puede utilizar como entorno de desarrollo un plugin ADT para Eclipse que incluye un simulador, que también es multiplataforma, libre y gratuito

6.5 Desarrollo móvil multiplataforma

Existen diversas opciones que permiten desarrollar aplicaciones multiplataforma. Un software con el que es posible programar con un lenguaje determinado y que, además, permite que la aplicación funcione en varios dispositivos con un mismo código.

6.5.1 PhoneGap

PhoneGap es un sistema para crear aplicaciones usando HTML5, CSS3 y Javascript, ejecutadas dentro en un componente WebKit del móvil. Provee una serie de librerías Javascript desarrolladas en el lenguaje específico de cada plataforma (Objetctive-C para IOS, Java para Android, etc) que permiten acceder a las características del móvil como GPS, acelerómetro, cámara, contactos, base de datos, filesystem, etc.

Al ser una página web, se tiene acceso al DOM y se puede usar *frameworks web* como *jQuery* o cualquier otro. Requiere diseñar la aplicación web con los componentes visuales típicos del HTML, etc o usar un *framework web mobile* como *jQuery Mobile* o *Sencha Touch* entre otros. Tiene la ventaja de que se puede definir la navegación inicial de la aplicación usando un navegador en un ordenador, sin tener que ejecutarla en el simulador.

Se puede ver una aplicación PhoneGap como una serie de páginas web que están almacenadas y empaquetadas dentro de una aplicación móvil, visualizadas con un navegador web, con acceso a la mayoría de APIs del móvil, lo cual lo convierte en una alternativa muy sencilla para crear aplicaciones.

Para trabajar con cada plataforma hay que usar un sistema distinto: para Iphone/Ipad es necesario usar Xcode (solo disponible en Mac) y una plantilla de proyecto que proporciona PhoneGap. Para Android se debe usar Eclipse (Windows, Mac y Linux) y otra plantilla de proyecto específica. Y para Blackberry no hay entorno: solo Java SDK, BlackBerry SDK y Apache Ant.

Ventajas:

- Es la solución que más plataformas móviles soporta, ya que corre dentro de un navegador web. Además de Iphone/Ipad y Android, funciona también en Palm, Symbian, WebOS, Windows mobile 7 y BlackBerry,
- Es muy fácil de desarrollar y proporciona una gran libertad a los que tienen conocimientos de HTML y Javascript.
- Hay buena documentación y bastantes ejemplos.
- Es gratis, soporte de pago. Licencia BSD.

Inconvenientes:

- Requiere Mac con Xcode para empaquetar aplicaciones IOS.
- La aplicación no es más que una página web, por lo que el aspecto dependerá del framework web utilizado. Necesitaremos el uso de frameworks HTML móviles si queremos que parezca una aplicación nativa.
- No llega al rendimiento de una aplicación nativa, pues el HTML, CSS y Javascript debe ser leído e interpretado por el motor del navegador cada vez arranca.

6.5.2 Titanium Appcelerator

Con Appcelerator es posible crear aplicaciones para Android, Iphone y de escritorio, usando exclusivamente Javascript (el soporte para Blackberry está en fase beta).

Para programar proporciona Titanium Studio, un IDE basado en Eclipse con el que crear los proyectos y editar los ficheros Javascript y el resto de recursos y lanzar los scripts de creación.

Las aplicaciones se programan íntegramente con Javascript, creando y colocando "a mano" todos los controles, usando para ello una librería que hace de puente entre la aplicación Javascript y los controles del sistema. Esto significa que las ventanas y demás controles visuales (botones, listas, menús, etc) son nativos: cuando se añade un botón, se crea un botón del sistema y se añade a la vista, lo que lo hace más rápido de renderizar y la respuesta del usuario es también rápida.

Una de las características más interesantes de Appcelerator es que al empaquetar la aplicación, el Javascript es transformado y compilado. Después, cuando se arranca la aplicación en el móvil, el código se ejecuta dentro de un *engine Javascript*, tal y como dice la documentación oficial, que será *JavaScriptCore* en IOS (el intérprete de Webkit, el motor de Safari y Chrome) y *Mozilla Rhino* en Android/Blackberry.

El hecho de que el Javascript esté compilado y que los controles creados sean nativos, le hace tener mejor rendimiento posible en comparación con PhoneGap o Adobe Air para móviles y similar a Corona SDK.

Con Appcelerator es complicado maquetar, pues no existe un HTML inicial donde añadir los controles, sino que hay que crear las ventanas y componentes "a mano" con Javascript.

Los desarrollos de las librerías Javascript para cada sistema operativo evolucionan por separado por lo que es posible que no funcionen de la misma manera. A diferencia de PhoneGap, que solo tiene una librería Javascript para acceder a las características especiales del sistema, Appcelerator necesita además librerías para manejar los controles nativos y su disposición en la pantalla, por lo que el desarrollo en general es más costoso.

Para iOS, Titanium Studio genera un proyecto Xcode con el Javascript transformado junto con todas las librerías necesarias. Después es posible lanzar el simulador con la aplicación en Xcode sin salir de Titanium Studio. Una vez generado el proyecto, éste se puede abrir con Xcode y continuar empaquetándolo y configurándolo para su distribución (certificados, provisioning, logos, splash screen, etc). Desde Xcode no se puede editar el JavaScript, se debe volver a editar en Titanium Studio y regenerar el proyecto Xcode otra vez.

Sobre el soporte Android, tanto para probar en el simulador como para empaquetar la aplicación, solo hay que tener el SDK de Android instalado.

Ventajas:

- Multiplataforma móvil y también de escritorio.
- Aspecto y controles nativos. Buen rendimiento.
- Buenos ejemplos
- Gratis, soporte de pago. Licencia Apache.

Desventajas:

- Definición de componentes visuales y ubicación de controles compleja
- Mucha documentación pero poco actualizada
- Requiere Mac y Xcode para empaquetar aplicaciones IOS.

6.5.3 Adobe Air Mobile

Adobe Air mobile funciona con Flex 4 y soporta las plataformas IOS, Android y BlackBerry Tablet, además de los sistemas de escritorio Windows, Mac y Linux (a través de un runtime). Flex 4 utiliza el lenguaje de programación ActionScript, de tipado fuerte y con clases, interfaces, herencia y paquetes muy parecido a Java con el que poder hacer complejos desarrollos.

El IDE oficial, Flash Builder 4.5 es un IDE muy potente y es de pago. Es posible compilar y empaquetar las aplicaciones con el *Flex SDK opensource* y gratuito pero es más complejo. Los controles visuales usados durante el desarrollo y ejecución no son los originales de cada plataforma, sino que son específicos de Flex 4. Esto garantiza que todas las aplicaciones tendrán exactamente el mismo aspecto y comportamiento.

Se pueden depurar aplicaciones en remoto. Una de las peculiaridades es que es la única herramienta que no requiere ni el Android SDK ni el Xcode para Mac para ejecutar y crear las aplicaciones.

Ventajas:

- Multiplataforma móvil y también de escritorio.
- ActionScript es un lenguaje muy potente que permite el uso de patrones y estructuras complejas en los desarrollos.
- Desarrollo y definición de las vistas con el editor visual de MXML con Flash Builder. El IDE y Flex 4 son muy potentes, y la documentación buena.
- Flash Builder 4.5 no requiere el uso de Xcode ni Mac.
- Depuración remota.

Desventajas:

- El precio de Flash Builder 4.5. Aunque hay otras herramientas y se puede usar el SDK gratuito.
- No funciona en todos los Android, solo en los de gama alta que tengan arquitectura Arm7.
- Rendimiento es regular y la renderización no es suave en IOS.
- Aspecto no nativo (aunque homogéneo entre todas las plataformas).

6.5.4 Corona SDK

Corona es un *framework* para el desarrollo de aplicaciones gráficas para iOS/Android de la compañía Ansca Mobile. Se desarrolla en Lua y no tiene IDE, aunque si viene con un interprete-emulador y multitud de ejemplos.

Contiene varios simuladores para cada uno de los dispositivos para los que se puede desarrollar y un depurador por línea de comandos. El emulador dispone así mismo de un terminal que permite mostrar mensajes de trazas durante la ejecución.

No son necesarios conocimientos de Objective-C/Cocoa, C + + o Java. Lua es un lenguaje de programación imperativo, estructurado y bastante ligero que fue diseñado como un lenguaje interpretado con una semántica extendible.

Contiene un conjunto limitado de librerías propias de Lua así como otras propias del SDK que aportan la funcionalidad y apariencia propia de estos dispositivos móviles, en especial de los de Apple.

Hay que pagar una licencia para cada plataforma o una conjunta para IOS/Android.

Ventajas:

- Alto rendimiento en la ejecución de aplicaciones.
- Motor gráfico y físico ideal para juegos.
- Lua es un lenguaje bastante sencillo y potente.
- Buena documentación, ejemplos y plantillas. Amplia comunidad.
- Incorpora elementos nativos, sobre todo de iOS muy sencillos de implementar.

Desventajas:

- El precio de la licencia anual.
- Aunque se puede usar para cualquier tipo de aplicación, realmente es ideal para aplicaciones gráficas y juegos.
- Está en evolución.

6.5.5 Plataforma seleccionada

Después del estudio de las características de los distintos entornos multiplataforma, estás son las conclusiones.

En todos los casos, los lenguajes de las plataformas, Lua, JavaScript y ActionScipt, son suficientemente potentes y además, sencillos de implementar.

Para el diseño de la aplicación, el más avanzado es el de adobe que tiene su propio editor. El resto necesitan de una colocación un tanto manual, aunque en Corona SDK se pueden crear grupos que facilitan el diseño.

Appcelerator y Corona son los únicos que permiten crear controles nativos de cada plataforma aunque la utilización y ubicación gráfica de estos en Appcelerator es más compleja. Adobe permite usar sus propios componentes con resultado homogéneo en todas las plataformas.

PhoneGap es el que más sistemas operativos soporta. El resto cubren las más importantes Android e iOS.

El rendimiento de la aplicación es muy bueno en Corona SDK así como también en Appcelerator. Las otras dos opciones son menos rápidas aunque con un rendimiento aceptable.

Con respecto a la documentación disponible, Corona SDK es la que tiene más disponible y con una amplia comunidad en la página web de la compañía. Es una herramienta en evolución pero la documentación está siempre actualizada. La documentación en Adobe también es bastante completa. Las otras dos son herramientas jóvenes que todavía les falta para estar acabadas: sus APIs cambian, están incompletas y a veces fallan y la documentación es regular.

En cuanto a la distribución, la plataforma de Adobe es la única que no necesita tener un Mac para desarrollar para iOS.

Appcelerator y PhoneGap son gratuitos, solo hay que pagar para el soporte. Flash Builder 4.5 Premium tiene un precio muy elevado, aunque se puede usar la versión de prueba durante 30 días y hay SDK libres. Corona SDK es gratis para desarrollo, pero requiere pagar una licencia anual de $199 si quieres subir tus aplicaciones al App Store o Market de Android o $349 para ambos.

Con estos criterios, las dos mejores opciones por rendimiento y por posibilidad de utilizar componentes nativos son Corona SDK y Appcelerator. Eliminando la restricción económica asociada a las licencias de Corona SDK, y dado que la gestión gráfica es mejor y la documentación más completa en esta plataforma, se selecciona ***Corona SDK*** como base para el desarrollo de la aplicación móvil de este proyecto.

7 Plataforma Corona SDK. Características

7.1 Introducción a la Plataforma Corona SDK

La plataforma Corona SDK es un entorno de desarrollo de aplicaciones móviles para la creación de aplicaciones de altas prestaciones, aplicaciones multimedia y juegos para dispositivos iOS, Android y Kindle.

Corona SDK contiene un simulador para cada uno de los dipositivos con sistema operativo móvil de Apple, y para varios modelos de dispositivos con sistema operativo Android. Tambien contiene un depurador por línea de comandos así como aplicaciones de ejemplo y documentación.

Permite a los desarrolladores usar Lua, un lenguaje de scrip de alto rendimiento construido sobre un motor de Objective-C/C++. No son necesarios conocimientos de Objective-C/Cocoa, C + + o Java. Lua es un lenguaje de programación imperativo, estructurado y bastante ligero que fue diseñado como un lenguaje interpretado con una semántica extendible.

Contiene un conjunto limitado de librerías propias de Lua así como otras propias del SDK que aportan la funcionalidad y apariencia propia de estos dispositivos móviles, en especial de los de Apple.

7.2 Gestión de proyectos en Corona SDK

Para crear aplicaciones en Corona SDK se necesita instalar la aplicación Corona SDK en entorno Mac o Windows y utilizar un editor de texto para generar los archivos con el código. La versión de prueba de Corona SDK nos permite realizar pruebas con el simulador en dispositivos Android o iOS (solo en equipos Mac). Para construir aplicaciones y distribuirlas en la AppStore o el android Market es necesario comprar una licencia.

Para crear un proyecto de Corona es necesario como minimo una carpeta que contenga un archivo de texto llamado "main.lua". Este archivo, "main.lua", es el primer archivo que lee Corona SDK por lo que si no esta disponible la aplicación no puede iniciarse. Este archivo principal, a su vez, puede cargar otros archivos de código externo, o recursos de otros programas, tales como sonidos,imágenes o videos. La extensión de archivo ". lua" indica que el archivo está escrito en lenguaje en ese lenguaje, que es el que se usa para crear aplicaciones en Corona SDK.

De cara a la construcción de la aplicación, en la carpeta del proyecto existen 2 posibilidades de configuración mediante ficheros. El primero es el fichero config.lua con las dimensiones del contenido visible y el modo de escalado de la pantalla. Este archivo se comenta más adelante en la implementación del proyecto al tratar la adaptación dinámica de contenido en la pantalla. También se debe añadir un archivo Icon.png que será el icono de la aplicación al instalarse en el dispositivo final . El segundo fichero que podemos incluir es el denominado build.settings que describe las propiedades en tiempo de construcción. Más adelante, en las tareas de administración se comentarán las opciones de este archivo.

7.3 Lenguaje de Corona SDK: Lua

7.3.1 Generalidades

Lua es un lenguaje de programación imperativo, estructurado y bastante ligero que fue diseñado como un lenguaje interpretado con una semántica extendible.. Fue creado en 1993 en la universidad católica de Rio de Janeiro y cuyo nombre significa “Luna” en portugués. Es muy utilizado en programación de videojuegos asi como en aplicaciones para videoconsolas como PSP y Wii.

A continuación se detalla el uso del lenguaje asi como las librerías propias que posee Lua y se incluyen algunos ejemplos para poder comprender como funciona.

Identificadores

Puede ser cualquier cadena de caracteres que incluya letras, dígitos y guiones bajos. Los identificadores se utilizan como nombres de variables y campos de tablas. Se distinguen las mayúsculas y minúsculas

Palabras clave

Como cualquier otro lenguaje de programación, Lua utiliza una serie de palabras para crear las instrucciones que forman cada programa. Por este motivo, estas palabras se consideran reservadas y no se pueden utilizar como nombre de una variable o función.

and	break	do	else	elseif	
end	false	for	function	if	
in	local	nil	not	or	
repeat	return	then	true	until	while

Comentarios

Los comentarios comienzan con un guión doble (--) siempre que no esté incluido dentro de una cadena. Los bloques de comentarios están delimitados por corchetes.

```
--[[
  Comentario
]]
```

Tipos y valores

Lua es un lenguaje tipado dinámicamente por lo que no es necesario declarar el tipo de variables. Una vez asignado el valor, este define el tipo de la variable.

Los tipos básicos son:

- **nil** – este tipo tiene un sólo valor, *nil*, que representa la ausencia de valor. Es análogo al valor *null* utilizado en otros lenguajes.
- **boolean** – tiene 2 valores: *false* y *true. En expresiones condicionales, false y nil son evaluados como false mientras que cualquier otro valor es evaluado como true.*
- **number** - representa números reales (en coma flotante y doble precisión).
- **string** - representa un *array* de caracteres. Lua trabaja con 8 bits: los *strings* pueden contener cualquier carácter de 8 bits, incluyendo el carácter cero ('\0').
- **function** – representa una función dentro del script Lua.
- **table** - implementa arrays asociativos, esto es, arrays que pueden ser indexados no sólo con números, sino también con cualquier valor (excepto nil). Las tablas pueden ser heterogéneas, ya que pueden contener valores de todos los tipos (excepto nil). Las tablas son el mecanismo de estructuración de datos en Lua. Pueden ser usadas para representar *arrays* ordinarios, tablas de símbolos, conjuntos, registros, grafos, árboles, etc. Para representar registros, Lua usa el nombre del campo como índice. El lenguaje soporta esta representación haciendo la notación b.nombre equivalente a b["nombre"].

Los valores de las tablas y las funciones son *objetos*: las variables no *contienen* realmente esos valores, sino que sólo los referencian. La asignación, el paso de argumentos y el retorno de las funciones siempre manejan referencias a esos valores; esas operaciones no implican ningún tipo de copia.

La función *type* retorna un *string* que describe el tipo de un valor dado.

7.3.2 Variables

Las variables son lugares donde se almacenan valores. Existen tres tipos de variables en Lua: globales, locales y campos de tabla. Antes de la primera asignación, el valor de una variable es nil.

Variables globales

Lua asume que las variables son globales, a no ser que sean declaradas explícitamente como locales.

```
print( a ) --> nil
a = "this is one pound"
print( a ) --> "this is one pound”
```

Variables globales perduran mientras la aplicación está funcionando. Para eliminar una variable global de la memoria debe asignarse el valor nil, comportándose como si esa variable no hubiera sido inicializada.

Variables locales

Las variables locales se definen usando la palabra ***local***. Al contrario que las variables globales, las variables locales tienen un ámbito definido léxicamente: pueden ser accedidas libremente desde dentro de las funciones definidas en su mismo ámbito, que empieza con la declaración de la variable y termina con el final del bloque en el que se encuentra.

```
a = 10
local i = 1

while i <= 10 do
    local a = i*i – la variable a es diferente a la de fuera del bloque
    print( a ) --> 1, 4, 9, 16, 25, ...
    i = i + 1
end

print( a ) --> 10 (la variable local a)
```

Campos de Tabla

Los campos de tabla son los elementos que componen la tabla. Los valores se asignan a los campos que están indexados en un array. Cuando el índice es un *string*, el campo es conocido como propiedad.

```
t = { foo="hello" } – creamos una table con una propiedad individual "foo"
print( t.foo ) --> "hello"
t.foo = "bye" – asignamos un Nuevo valor a la propiedad "foo"
print( t.foo ) --> "bye"
t.bar = 10 --> creamos una nueva propiedad llamada “bar”
print( t.bar ) --> 10
print( t["bar"] ) --> 10
```

7.3.3 Expresiones

Operadores aritméticos

Lua tiene los operadores aritméticos comunes:

+ (adición)
- (substracción)
* (multiplicación)
/ (división)
% (módulo)
^ (exponenciación)
- (negación)

Si los operandos son números o strings que se convierten a números, entonces todas las operaciones tienen el significado corriente. La exponenciación trabaja con cualquier exponente. Por ejemplo, x^(-0.5) calcula la inversa de la raíz cuadrada de x.

El módulo se define como

```
a % b == a - math.floor(a/b)*b
```

Operadores relacionales

Los operadores relacionales en Lua son :

== (igualdad)
~= (no igualdad)
< (menor que)
> (mayor que)
<= (menor o igual que)
>= (mayor o igual que)

Devuelven siempre un resultado **false** o **true**.

La igualdad (==) primero compara el tipo de los operandos. Si son diferentes entonces el resultado es **false**. En otro caso se comparan los valores de los operandos. Los números y las cadenas se comparan de la manera usual. Los objetos (tablas y funciones) se comparan por *referencia*: dos objetos se consideran iguales sólo si son el *mismo* objeto. Cada vez que se crea un nuevo objeto (una tabla o función) este nuevo objeto es diferente de todos los demás objetos preexistentes.

El operador ~= es exactamente la negación de la igualdad (==).

Operadores lógicos

Los operadores lógicos en Lua son :

and (conjunción)
or (disyunción)
not (negación)

Como las estructuras de control, todos los operadores lógicos consideran *false* y nil como falso y todo lo demás como verdadero.

El operador negación *not* siempre retorna *false* o *true*.

El operador conjunción *and* retorna su primer operando si su valor es false o nil; en caso contrario *and* retorna su segundo operando.

El operador disyunción *or* retorna su primer operando si su valor es diferente de nil y *false*; en caso contrario *or* retorna su segundo argumento.

Tanto and como or usan evaluación de cortocircuito; esto es, su segundo operando se evalúa sólo si es necesario. He aquí varios ejemplos:

```
10 or 20            --> 10
10 or error()       --> 10
nil or "a"          --> "a"
nil and 10          --> nil
false and error()   --> false
false and nil       --> false
false or nil        --> nil
10 and 20           --> 20
```

Concatenación

El operador de concatenación de strings en Lua se denota mediante dos puntos seguidos ('..'). Si ambos operandos son números entonces se convierten a strings.

Operador de longitud

El operador longitud se denota mediante #. La longitud de un string es su número de bytes (significado normal de la longitud de un string cuando cada carácter ocupa un byte).

La longitud de una tabla t se define como un índice entero n tal que t[n] no es nil y t[n+1] es nil; además, si t[1] es nil entonces n puede ser cero. Para un array regular, con valores no nil desde 1 hasta un n dado, la longitud es exactamente n, el índice es su último valor. Si el array tiene "agujeros" (esto es, valores nil entre otros valores que no lo son), entonces #t puede ser cualquiera de los índices que preceden a un valor nil (esto es, Lua puede considerar ese valor nil como el final del array).

Precedencia de los operadores

La precedencia de los operadores en Lua sigue lo expuesto en la tabla siguiente de menor a mayor prioridad:

```
or
and
<     >     <=    >=    ~=    ==
..
+     -
*     /     %
Not   #     - (unario)
^
```

Se pueden usar paréntesis para cambiar la precedencia en una expresión. Los operadores de concatenación ('..') y de exponenciación ('^') son asociativos por la derecha. Todos los demás operadores son asociativos por la izquierda.

7.4 Estructuras de control

Estructura condicional, if … else
La sintaxis de la estructura condicional es:

```
if exp then
        bloque
elseif exp then
        bloque
else
        bloque
end
```

Tanto else como elseif son opcionales.

La condición de una expresión (*exp*) de una estructura de control puede retornar cualquier valor. Como se ha indicado anteriormente, tanto false como nil se consideran falsos. Todos los valores diferentes de nil y false se consideran verdaderos (en particular, el número 0 y el string vacío son también verdaderos).

Estructura repetitivas, while y until
La sintaxis de estas estructuras es:

```
while exp do
        bloque
end

repeat
        bloque
until exp
```

En ambos casos se ejecuta el bloque de forma repetitiva. En el primer caso, la expresión se evalúa al principio y en el segundo al final. En el bucle repeat–until el bloque interno no acaba en la palabra clave *until* sino detrás de la condición. De esta manera la condición puede referirse a variables locales declaradas dentro del bloque del bucle.

Estructura for
La sentencia for tiene dos formas: una numérica y otra genérica.

La forma numérica del bucle for repite un bloque mientras una variable de control sigue una progresión aritmética. Tiene la sintaxis siguiente:

```
for nombre '=' exp1 ',' exp2 [',' exp3] do
        bloque
end
```

El bloque se repite para los valores de nombre comenzando en exp1 hasta que sobrepasa exp2 usando como paso exp3.

```
for i=1, 25, 5 do
     print (i)
end
```

La sentencia for genérica trabaja con funciones, denominadas iteradores. En cada iteración se invoca a la función iterador que produce un nuevo valor, parándose la iteración cuando el nuevo valor es nil. El bucle for genérico tiene la siguiente sintaxis:

for lista_de_nombres **in** explist **do**
bloque
end

for lista_de_nombres in explist do bloque end
lista_de_nombres ::= nombre {',' nombre}

Por ejemplo, este bucle recorre las líneas de un fichero

```
for line in io.lines(filename) do
     print (line)
end
```

break
La orden break se usa para terminar la ejecución de los bucles while, repeat y for, saltando a la sentencia que sigue después del bucle.

Un break finaliza el bucle más interno que esté activo.

7.4.1 Funciones

La sintaxis para la definición de funciones es :
function nombre_de_func ([lista_de_argumentos]) bloque **end**

Así mismo se puede utilizar.
nombre_de_func = **function** ([lista_de_argumentos]) bloque **end**

Una definición de función es una expresión ejecutable, cuyo valor tiene el tipo *function*.

Los argumentos formales de una función actúan como variables locales que son inicializadas con los valores actuales de los argumentos.

La orden **return** se usa para devolver valores desde una función. Las funciones pueden retornar más de un valor.

7.4.2 Objetos, propiedades y funciones

Muchas de las API's de Corona SDK devuelven objetos. Las propiedades de esos objetos son manipulables (datos, posición, visibilidad, escala…) y se pueden añadir nuevas como si tratara de una tabla.

Las nuevas propiedades no pueden comenzar con el carácter guión bajo ("_"), ya que está reservado para el sistema.Todas las propiedades se puede acceder a través de la cadena que representa el índice, *tabla[“propiedad”]* o del operador punto ("."), *tabla.propiedad* .

Dado que las funciones pueden ser variables, una tabla puede almacenar estas como propiedades. Esto permite que una tabla pueda ser utilizada para agrupar lógicamente una familia de funciones, por ejemplo, la biblioteca matemática (*math*). Las funciones pueden utilizarse también como los métodos asociados al objeto.

Una diferencia clave entre una función almacenada como propiedad o un método de un objeto, es la sintaxis. Se necesita decirle a Lua que tiene la intención de llamar a esta función como un método de objeto, no sólo una función normal. Para ello, es necesario utilizar el operador dos puntos (":") en lugar del operador punto (".").

```
Objeto:funcion( arg1, arg2 )
```

El operador dos puntos es en realidad un acceso directo. En la mayoría de lenguajes, una llamada al método objeto es como una llamada a la función normal, excepto que hay un argumento oculto para la función que es el objeto mismo. Este argumento oculto que se conoce como this en Javascript y self en Lua. Igualmente se podría llamar un método de objeto mediante el operador punto, si se pasa el objeto como primer argumento:

```
Objeto.funcion( Objeto, arg1, arg2 )
```

7.5 Librerías estándar de Lua

Corona SDK incluye las mismas bibliotecas Lua que son parte del estándar. Estas bibliotecas proporcionan una funcionalidad útil y básica. Se agrupan en las siguientes categorías:

- Biblioteca básica: proporciona algunas funciones del núcleo de Lua
- Manipulación de cadenas: proporciona funciones para tratamiento de cadenas, búsquedas y detección de patrones.
- Manipulación de tablas: proporciona funciones genéricas para manejo de tablas
- Funciones matemáticas: Esta biblioteca es una interfaz a la biblioteca matemática estándar de C.
- Funciones de entrada y salida: proporciona dos estilos diferentes de manejo de ficheros. El primero de ellos usa descriptores de fichero implícitos; esto es, existen dos ficheros por defecto, uno de entrada y otro de salida, y las operaciones se realizan sobre éstos. El segundo estilo usa descriptores de fichero explícitos.

- Funciones de sistema operativo: proporciona funciones asociadas al sistema operativo del dispositivo.

A excepción de la biblioteca básica, cada biblioteca ofrece todas sus funciones como propiedades de una tabla o como los métodos de sus objetos. Esto crea una agrupación lógica de funciones y es la manera de crear en Lua un espacio de nombres de diferente funcionalidad.

Los detalles de las funciones de cada biblioteca se especifican en el Anexo I de esta memoria.

7.6 Librerías de Corona SDK

Corona SDK posee su propio conjunto de bibliotecas sobre las bibliotecas estándar de Lua. Algunas bibliotecas están incorporadas internamente, mientras que otras deben ser cargadas explícitamente.

Las siguientes son las bibliotecas centrales de Corona SDK y se cargan automáticamente cuando se inicia la aplicación:

- display - proporciona todas las rutinas para la creación de objetos de visualización.
- transition - funciones para la animación de objetos de visualización, lo que simplifica el proceso de creación de movimientos básicos.
- timer - ofrece funciones básicas de tiempo.
- media - permite el acceso a las capacidades multimedia del dispositivo.
- native - proporciona acceso a los elementos de la interfaz nativa de los dispositivos.
- system - es un conjunto de funciones de sistema.

Los detalles de las funciones de cada una de estas bibliotecas se especifican en el Anexo I de esta memoria.

7.7 Visualización de objetos en pantalla

Como se ha comentado, la biblioteca display contiene las funciones para crear los objetos gráficos, tanto imágenes, texto o formas geométricas.

Para gestionar el orden en que se dibujan los objetos gráficos, están organizados en una jerarquía que determina que objetos aparecen por encima de otros. La jerarquía es posible gracias a la existencia de objetos de grupo, *GroupObjects*. Son un tipo especial de

DisplayObject que pueden contener otros objetos hijos. Permiten organizar los objetos gráficos para poder establecer relaciones entre ellos.

Los objetos pertenecientes a un grupo, es decir, los hijos, están organizados en un array por lo que el primer hijo (índice 1) está por debajo del siguiente hijo, y así sucesivamente, el último hijo está siempre por encima de todos los demás.

group:insert() es el método para insertar objetos dentro de un determinado grupo existente y el acceso a los objetos se puede realizar con el índice correspondiente, *group[1]* .

```
-- se crean objetos
local square = display.newRect( 0, 0, 100, 100 )
local rect = display.newRect( 0, 0, 100, 100 )

-- se crea el grupo
local group = display.newGroup()

-- se insertan objetos
group:insert( square )
group:insert( rect )

-- acceso indexado a los objetos
assert( (group[1] == square) and (group[2] == rect) )
```

Para mover objetos hacia adelante y hacia atrás se puede cambiar el índice asociado a los mismos o bien utilizar los *métodos object:toBack()* y *object:toFront()*

Las modificaciones realizadas en las propiedades de los objetos son actualizadas en la pantalla de acuerdo al ciclo definido en el sistema. Este se ejecuta 30 o 60 veces por segundo de acuerdo al valor de velocidad de actualización de la pantalla (*frame rate*) establecido en el fichero config.lua. Cada ciclo genera un evento "*enterframe*" que es capturable por un *listener* del código Lua.

La pantalla representa el sistema de coordenadas base para los objetos. Cada objeto o grupo de visualización opera con su propio sistema de coordenadas local y se debe relacionar con las coordenadas de la pantalla.

Para definir la posición se utiliza el sistema de coordenadas cartesianas. El origen de la pantalla se encuentra en la esquina superior izquierda, como se observa en la figura 4.2, de modo que valores positivos de y, desplazan la posición hacia abajo y valores positivos de x, la desplazan a la derecha. Todas las coordenadas de la pantalla se definen en relación con este origen.

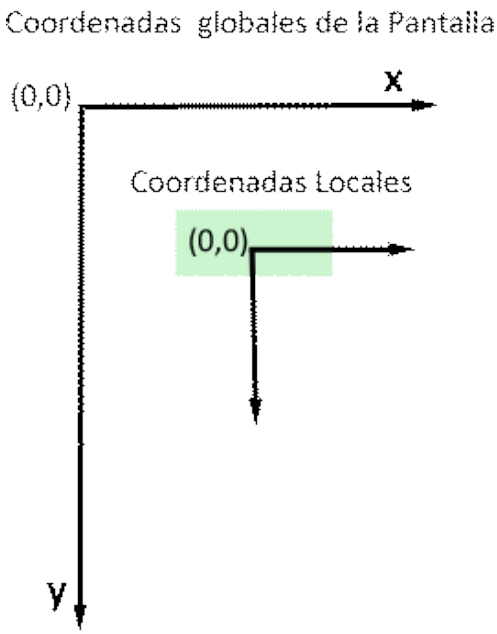

Figura 4.1. Origen de coordenadas en las pantallas de los dispositivos

Cuando un objeto es creado, sus coordenadas son relativas a la pantalla principal. Cuando se añade un *listener* para eventos táctiles (t*ouch events*) al objeto, este devuelve la posición en la pantalla. Cuando el objeto es añadido a un grupo, las coordenadas de este objeto son relativas a las del grupo y no a las de la pantalla. Para poder obtener las coordenadas de pantalla podemos llamar al método *object.contentBounds*()

Dado que los dispositivos tienen recursos limitados, es importante eliminar los objetos de visualización de la jerarquía de la pantalla cuando ya no se usan. Esto ayuda al rendimiento general del sistema al reducir el consumo de memoria (especialmente las imágenes). En el apartado de *gestión de la memoria* de este capítulo se explica como eliminar los objetos de visualización de la forma más adecuada.

Los objetos gráficos son instancias de la clase *DisplayObject* que posee propiedades y métodos comunes para todos ellos y como todos los objetos en Lua, puede ser tratado como una tabla añadiendo propiedades nuevas.

Estas son las propiedades comunes de los objetos:

object.alpha	Opacidad del objeto. Con valor 0 el objeto es transparente y con valor 1 es opaco. Por defecto valor 1.
object.height	Altura del objeto (coordenadas locales)
object.isVisible	Controla si el objeto es visible. Valor booleano (*true* es visible, *false* no es visible).
object.isHitTestable	Permite recibir eventos aun cuando el objeto no es visible. Valor booleano. Por defecto false
object.parent	Devuelve el grupo padre del objeto.
object.rotation	Angulo de rotación del objeto (grados)
object.contentBounds	Tabla que contiene las propiedades xMin, xMax, yMin, yMax en coordenadas de pantalla.
object.contentHeight	Altura (coordenadas de pantalla)
object.contentWidth	Anchura (coordenadas de pantalla)

object.width	Anchura (coordenadas locales)
object.x	Especifica la posición x (en coordenadas locales) del objeto relativa al origen del padre. Específicamente, proporciona la posición x del punto de referencia del objeto relativa al padre.
object.xOrigin	Especifica la posición x de origen del objeto relativa al origen del padre. En coordenadas locales.
object.xReference	Define la posición x de referencia con respecto a la posición x de origen. Es relativa a otro punto del objeto, no del padre. Conceptualmente, el punto de referencia es aquel sobre el que se suceden las rotaciones y los cambios de escala. Cambiando este valor no se modifica la posición del objeto. Únicamente es un punto de referencia.
object.xScale	Valor del factor de escalado en la dirección x. Un valor 0.5 escalará el objeto un 50% en la dirección x.
object.y	Especifica la posición y (en coordenadas locales) del objeto relativa al origen del padre. Específicamente, proporciona la posición x del punto de referencia del objeto relativa al padre.
object.yOrigin	Especifica la posición y de origen del objeto relativa al origen del padre. En coordenadas locales.
object.yReference	Define la posición y de referencia con respecto a la posición y de origen. Es relativa a otro punto del objeto, no del padre. Conceptualmente, el punto de referencia es aquel sobre el que se suceden las rotaciones y los cambios de escala. Cambiando este valor no se modifica la posición del objeto. Únicamente es un punto de referencia.
object.yScale	Valor del factor de escalado en la dirección y. Un valor 0.5 escalará el objeto un 50% en la dirección y.

Estas son los métodos comunes de los objetos:

object:rotate(deltaAngle)	Añade el valor deltaAngle (en grados) a la actual propiedad de rotación.
object:scale(x, y)	Multiplica las propiedades de escala xScale y yScale por los valores sx y sy respectivamente.
object:setReferencePoint(referencePoint)	Establece la referencia del objeto en el punto especificado en el parámetro. El argumento puede ser: ▪ display.CenterReferencePoint ▪ display.TopLeftReferencePoint ▪ display.TopCenterReferencePoint ▪ display.TopRightReferencePoint ▪ display.CenterRightReferencePoint ▪ display.BottomRightReferencePoint ▪ display.BottomCenterReferencePoint ▪ display.BottomLeftReferencePoint ▪ display.CenterLeftReferencePoint Cambiando la referencia del objeto se cambiarán los valores de x e y sin mover el objeto.
object:translate(deltaX, deltaY)	Añade los valores deltaX y deltaY a las propiedades x e y del objeto desplazando el objeto de su posición actual.
object:removeSelf( )	Borra el objeto y libera su memoria, asumiendo que no hay otras referencias a él.

Para objetos de texto, hay algunas propiedades y métodos específicos:

object.size	Tamaño del texto
object.text	Cadena que contiene el texto del campo de texto. Es usado para actualizar la cadena de texto del objeto.
object:setTextColor(r, g, b [, a])	Establece el color de un objeto de texto. Todos los valores deben estar entre 0 y 255. El valor a es opcional y representa Alpha (opacidad); por defecto es 255 (opaco).

7.8 Gestión de eventos

La manera de crear aplicaciones interactivas en Corona SDK es mediante la gestión de eventos. Estos desencadenan acciones que provocan una respuesta del programa. Por ejemplo, cualquier objeto que se muestre en la pantalla, puede convertirse en un botón interactivo. Esta flexibilidad es uno de las ventajas más importantes que tiene esta plataforma de desarrollo.

Hay eventos globales que no están asociados a ningún objeto en particular y que son difundidos a todos los *listeners* integrados. Se entiende por *listener*, aquellas funciones capaces de recibir y tratar los eventos. Estos eventos globales son:

enterFrame

Eventos que ocurren en el intervalo de actualización de la aplicación. Este se ejecuta 30 o 60 veces por segundo de acuerdo al valor de velocidad de actualización de la pantalla (*frame rate*) establecido en el fichero config.lua. Se genera en el objeto *Runtime*:

```
Runtime:addEventListener( "enterFrame", myObject )
```

Las propiedades asociadas a este evento son:

- event.name : cadena "enterFrame".
- event.time : tiempo en milisegundos desde el inicio de la aplicación.

system

Los eventos de sistema son distribuidos para notificar a la aplicación de acciones externas como la interrupción del programa por una llamada entrante al dispositivo. Son generados también por el objeto *Runtime*.

Las propiedades asociadas a este evento son:

- event.name: cadena "system".
- event.type: cadena que identifica el tipo de evento. Los posibles valores son:
  - "applicationStart": ocurre cuando la aplicación es lanzada y se ejecuta el código de main.lua
  - "applicationExit": ocurre cuando el usuario cierra la aplicación.
  - "applicationSuspend": ocurre cuando el dispositivo necesita suspender la aplicación por entrada de una llamada o por inactividad.
  - "applicationResume": ocurre cuando una aplicación se reanuda tras una suspensión.

orientation

Los eventos de orientación se producen cuando se cambia la orientación del dispositivo en aquellos que disponen de acelerómetro que permite detectarlos. Son generados también por el objeto *Runtime*.

Las propiedades asociadas a este evento son:

- event.name: cadena "orientation".
- event.type: cadena que representa la orientación del dispositivo:
  - "portrait"
  - "landscapeLeft"
  - "portraitUpsideDown"
  - "landscapeRight"
  - "faceUp"
  - "faceDown"
- event.delta: Es el ángulo de diferencia entre el fin y el inicio del cambio de orientación.

Hay también otros eventos globales asociados al acelerómetro, al GPS, a la brújula y a niveles bajos de memoria que dependen también de si los dispositivos tienen esas funcionalidades.

El resto de eventos se consideran locales ya que son gestionados por un *listener* concreto.

Cuando el usuario toca la pantalla con el dedo, se genera un evento de golpeo (*hit event*) que se distribuye a todos los objetos de la jerarquía de visualización que intersectan con el punto de toque de la pantalla. Este evento es propagado desde el objeto que está más arriba en la jerarquía hasta el que está más abajo.

Esa propagación se puede parar indicándole al sistema que el evento está tratado, devolviendo *true* en la función *listener* (*return true*). Si el evento no es tratado por ninguno de los objetos transversales, es difundido como un evento global al objeto *Runtime*.

Los eventos táctiles (*touch events*) son otro tipo especial de los *"hit events"* que desencadenan una secuencia de eventos con diferentes fases.

Las propiedades asociadas a estos eventos son:

- event.name: Cadena "touch".
- event.x: posición x del toque en las coordenadas de la pantalla.
- event.y: posición y del toque en las coordenadas de la pantalla.
- event.xStart: posición x en el inicio del toque.
- event.yStart: posición y en el inicio del toque.
- event.phase: cadena que identifica las fases de la secuencia de toque. Los valores son:
  - "began": un dedo toca la pantalla, inicia el evento.
  - "moved": un dedo se mueve por la pantalla.
  - "ended": un dedo es levantado de la pantalla, finaliza el evento.
  - "cancelled": el sistema cancela la secuencia del evento.

7.9 Animación y movimiento

Además de aplicaciones, Corona SDK es una herramienta muy potente para el desarrollo de juegos y una de sus grandes ventajas es que cualquier objeto que se muestra en la pantalla se puede animar y dotarlo de movimiento.

La biblioteca de transiciones contiene las funciones para animar un objeto de visualización por interpolación de una o más propiedades durante un tiempo determinado.

La manera más simple es utilizar el método *transition.to* cuyo primer argumento es el objeto sobre el que se van a modificar las propiedades y el segundo es una tabla con los parámetros de control. Estos determinan, entre otros, la duración de la animación y los valores finales de las propiedades del objeto. Los valores intermedios de las propiedades son determinados por unas funciones de ajuste que se especifican también como parámetro de control. Se pueden ver en la siguiente tabla:

easing.linear	Define un movimiento constante sin aceleración.
easing.inExpo	Inicia el movimiento desde la velocidad cero y, a continuación, lo acelera conforme se ejecuta.
easing.inOutExpo	Inicia el movimiento desde una velocidad cero, acelera y luego desacelera de nuevo hasta cero utilizando una ecuación de aceleración exponencial.
easing.inOutQuad	Inicia la animación desde una velocidad cero, se acelera, y entonces desacelera de nuevo hasta cero.
easing.inQuad	Realiza una interpolación cuadrática de los valores de la propiedad de animación en una transición empezando desde cero.
easing.outExpo	Inicia movimiento rápido y luego desacelera hasta la velocidad cero conforme se ejecuta.
easing.outQuad	Inicia movimiento rápido y desacelera mientras realiza una interpolación cuadrática de los valores de la propiedad de animación

Estos son algunos de los parámetros que se pueden definir en la tabla de control:

- params.time – duración de la transición en milisegundos.
- params.transition – funciones de ajuste de la transición. Por defecto es *easing.linear*
- params.delay – especifica el retraso desde el comienzo de la transición.
- params.onStart - es una función o *listener* que se invoca al comenzar la transición.
- params.onComplete - es una función o *listener* que se invoca al finalizar la transición.

Este es un ejemplo de transición de un objeto en la dirección vertical durante 3 segundos con un ajuste lineal:

```
circle2 = display.newCircle(65, 60, 10 )
circle2:setFillColor(255,0,255,255)
transition.to(circle2, {time=3000, y=460, transition = easing.linear})
```

Para cancelar una transición se utiliza la función transition.cancel(tween).

Hay otra biblioteca externa, *movieclip*, que permite crear *sprites animados* o *movieclips*, a partir de secuencias de imágenes, que luego se puede mover por la pantalla usando las mismas técnicas que cualquier objeto de visualización de Corona.

Las funciones de esta biblioteca están disponibles para reproducir estas animaciones de forma total o parcial, en dirección hacia adelante o hacia atrás, saltando a ciertos fotogramas, eliminando automáticamente la animación en el término de una secuencia e incluso permitiendo que se pueda arrastrar la animación mediante eventos de arrastrar y soltar. Este marco ofrece una manera rápida y ligera para crear animaciones.

La biblioteca no viene precargada en Corona, por lo que se debe cargar el módulo externo movieclip.lua en aquellas aplicaciones que lo precisen.

Para crear un nueva animación, se resalía con la función *movieclip.newAnim(frames)* donde se pasa como parámetros las distintas imágenes que forman la animación.

```
myAnim = movieclip.newAnim{ "img1.png", "img2.png", "img3.png", "img4.png" }
```

Esta animación se puede reproducir hacia adelante con la función *object:play()* de forma cíclica hasta que es detenida por la función *object:stop()*. Para reproducir de manera más particularizada, se pueden establecer los siguientes parámetros

object:play{ startFrame=a, endFrame=b, loop=c, remove=shouldRemove }

Con *loop* se indica el número de ciclos de repetición siendo el valor 0 para un ciclo sin fin. El parámetro *remove* es un valor booleano que si tienen el valor *true*, elimina el objeto cuando la secuencia se ha completado. El valor por defecto es *false*.
Similar a estas funciones, existe object:reverse() para reproducir la animación en sentido contrario al definido. Así mismo se pueden definir parámetros en esta función:

object:reverse{ startFrame=a, endFrame=b, loop=c, remove=shouldRemove }

Se reproducen los ciclos desde b (endFrame) hasta a (startFrame).

Para acceder a **frames** concretos de los definidos en la animación se utilizan las funciones *object:nextFrame()* y *object:previousFrame()*. También es posible definir un punto de ruptura en un **frame** en concreto con *object:stopAtFrame(frame).*

Para hacer que un objeto de animación se pueda desplazar por el usuario, se usa la función *object:setDrag*. Con el parámetro booleano *drag*, se indica si se puede desplazar o no. Con *limitX* y *limitY* se limita el desplazamiento en los ejes x e y o con una tabla de puntos {*left, top, width, height*} se especifica una determinada superficie de desplazamiento.

De cara a controlar la animación y el desplazamiento, los parámetros *onPress*, *onDrag* y *onRelease* permiten definir funciones o *listeners* para controlar los eventos con el mismo nombre.

```
myAnim:setDrag{ drag=true, limitX=false, limitY=false,
 onPress=myPressFunction, onDrag=myDragFunction,
          onRelease=myReleaseFunction, bounds={ 10, 10, 200, 50 }}
```

Es posible definir también animaciones personalizadas, definiendo las trayectorias y movimientos de los objetos. Se deben generar llamando de forma repetida a un listener que gestione los eventos “enterframe” del sistema, es decir, el evento que se genera por el numero de fotogramas por segundos definidos en la aplicación (con un valor por defecto de 30 modificable a 60).

Este es un ejemplo de una animación personalizada:

```
local xdirection,ydirection = 1,1
local xpos,ypos = display.contentWidth*0.5,display.contentHeight*0.5
local circle = display.newCircle( xpos, ypos, 20 );
circle:setFillColor(255,0,0,255);

local function animate(event)
        xpos = xpos + ( 2.8 * xdirection );
        ypos = ypos + ( 2.2 * ydirection );

        if (xpos > display.contentWidth - 20 or xpos < 20) then
                xdirection = xdirection * -1;
        end
        if (ypos > display.contentHeight - 20 or ypos < 20) then
                ydirection = ydirection * -1;
        end

        circle:translate( xpos - circle.x, ypos - circle.y)
end

Runtime:addEventListener( "enterFrame", animate );
```

7.10 Motor físico

Como se ha comentado en el apartado anterior, Corona SDK permite realizar de forma sencilla y potente la animación de objetos. Igual de simple es la capacidad de dotar a cualquier objeto de un motor físico que le permite interactuar con otros objetos de la aplicación.

Corona traduce automáticamente desde las unidades establecidas en la pantalla hasta las unidades internas métricas de la simulación física. Todos los valores de posición son

declarados en píxeles, que se convierten internamente en metros a una ratio de 30 píxeles por metro (esta proporción es configurable por el usuario en physics.setScale).

Para utilizar las características del motor físico de Corona SDK, se debe cargar el módulo "*physics*"

```
local physics = require "physics"
```

Para controlar la simulación física dentro de la aplicación están las funciones *physics.start(), physics.pause()* y *physics.stop().* La función *start* crea una instancia o reanuda el mundo físico, *pause* realiza una parada momentánea de la simulación y *stop* destruye el mundo físico.

El motor físico permite definir el valor de la gravedad que aplica tanto en la componente horizontal como en la vertical de los objetos. La función es *physics.setGravity*(x, y). Por defecto (0, 9.8). *physics.getGravity* devuelve los valores usados en x e y respectivamente.

Usando esta propiedad y la API del acelerómetro de Corona, se puede hacer una function basada en la gravedad dinámica de acuerdo al valor de inclinación del dispositivo:

```
local function onTilt( event )
        physics.setGravity( 10 * event.xGravity, -10 * event.yGravity )
end

Runtime:addEventListener( "accelerometer", onTilt )
```

Cuerpos físicos

El mundo de la física se basa en las interacciones de los cuerpos rígidos. Los objetos creados con corona SDK pueden ser dotados de ciertas características físicas que permiten modelar su comportamiento y su relación con otros objetos. Es por esto, que al invocar al constructor *physics.addBody* para definir las características físicas de un objeto no se crea un objeto nuevo, sino que se extienden las propiedades del mismo.

Las propiedades del objeto como posición x,y o rotación siguen trabajando normalmente auque se implemente el cuerpo físico pero los movimientos se pueden ver condicionados por la fuerza de la gravedad u otro tipo de interacciones físicas.
Un objeto de visualización con atributos físicos puede ser eliminado de la forma habitual, con object:removeSelf(), eliminándolo de la pantalla visible y de la simulación física. Una vez asignadas propiedades físicas, no se pueden eliminar de forma independiente.

Los cuerpos físicos tienen tres propiedades principales:

- density (densidad) – valor que permite calcular la masa al multiplicarlo por la superficie. Este parámetro se basa en un valor de 1,0 para el agua, es decir, los materiales más ligeros que el agua (como la madera) tienen una densidad inferior a 1,0, y los materiales más pesados (como las piedras) tienen una densidad superior a 1,0. Sin embargo, el comportamiento del objeto global dependerá

también de la gravedad y de la escala píxeles-metros (pixels-to-meter) definida en el motor físico. El valor por defecto es de 1,0.

- friction (fricción) - puede ser cualquier valor no negativo, un valor de 0 significa que no hay fricción y un valor 1 se corresponde con una fricción fuerte. El valor por defecto es de 0,3.

- bounce (rebote) - determina la velocidad con que un objeto es devuelto después de una colisión. Los valores mayores que 0,3 tienen un rebote alto. Un rebote superior a 1,0 es válido, pero produce comportamientos extraños. El valor por defecto es de 0,2.

Por defecto, el constructor *physics.addBody* supone que el objeto físico es rectangular, con los límites de colisión que se ajustan automáticamente a los bordes de la imagen asociada o un objeto vectorial. Esto funciona bien para las cajas, plataformas, grandes masas de tierra, y otros *sprites* de forma rectangular. Todos los parámetros de la tabla son opcionales, y si no se indican se toman los valores por defecto. Para varios objetos con las mismas propiedades es recomendable el uso de una tabla común como se ve en los ejemplos:

```
local crate = display.newImage( "crate.png", 100, 200 )
physics.addBody( crate, { density = 1.0, friction = 0.3, bounce = 0.2 } )

----------------------------

local crate1 = display.newImage( "crate.png", 100, 200 )
local crate2 = display.newImage( "crate.png", 180, 280 )

local crateMaterial = { density = 1.0, friction = 0.3, bounce = 0.2 }

physics.addBody( crate1, crateMaterial )
physics.addBody( crate2, crateMaterial )
```

Cuando el cuerpo físico no se ajusta a un contenido rectangular se pueden usar las estructuras circulares o poligonales.

Los cuerpos circulares requieren un parámetro radio (*radius*) adicional. Esto funciona bien para bolas, piedras y otros objetos que pueden ser tratados como perfectamente redondos en el cálculo de colisiones. Para objetos redondos “irregulares” es recomendable utilizar un radio menor al objeto o la definición de una forma poligonal.

```
local ball = display.newImage( "ball.png", 100, 200 )
physics.addBody( ball, { density = 1.0, friction = 0.3, bounce = 0.2,
                         radius = 25 } )
```

Para aquellos casos en los que los cuerpos rectangulares y circulares no encajan con la forma del objeto, se deben utilizar los cuerpos poligonales. Estos tienen un parámetro *shape* que es una tabla de coordenadas de puntos que definen la silueta de la forma. Estas coordenadas son relativas al objeto cuyo origen es situado por Corona SDK en el centro de la imagen.

Por ejemplo, para dibujar una forma rectangular de 20 píxeles de altura y 40 de anchura con origen en el centro, se debería usar la siguiente forma:

```
squareShape = { -20,-10, 20,-10, 20,10, -20,10 }
```

El número máximo de puntos (y por lo tanto, de lados del polígono) permitido por la forma de colisión es de ocho. Una definición de una forma se puede volver a utilizar varias veces. Las coordenadas del polígono debe estar definido en sentido horario, y la forma resultante debe ser convexa.

También es posible construir un cuerpo de múltiples elementos. En este contexto, cada elemento del cuerpo se especifica como una forma de polígono por separado con sus propiedades físicas. La estructura sería:

```
physics.addBody( displayObject, [bodyType,] bodyElement1, [bodyElement2, ...] )
```

Hay un caso especial de cuerpos físicos denominados sensores, que no se relacionan físicamente con otros objetos físicos, sino que solo producen los eventos de colisión cuando otros objetos pasan por ellos. Este sería un ejemplo de un sensor invisible en la pantalla:

```
local rect = display.newRect( 50, 50, 100, 100 )
rect:setFillColor( 255, 255, 255, 100 )
rect.isVisible = false  -- optional
physics.addBody( rect, { isSensor = true } )
```

Estas son las propiedades de los cuerpos físicos:

- body.isAwake – (booleano) estado “despierto” del objeto. Por defecto, todos los cuerpos se van de forma automática "a dormir" cuando no se interactúa con ellos durante un par de segundos hasta que algo (por ejemplo, una colisión) los despierta.

- body.isBodyActive – (booleano) estado de actividad actual. Cuerpos inactivos no se destruyen, sino que se retiran de la simulación y dejar de interactuar con otros organismos.

- body.isBullet – (booleano) determina si el cuerpo debe ser entendido como una "bala". Las balas están sujetas a la detección de colisiones continua, en lugar de la detección de colisiones periódicas en *timesteps*. Evita que objetos en movimiento rápido pasen por barreras sólidas. El valor predeterminado es falso.

- body.isSensor– (booleano) propiedad de solo escritura. Un sensor pasa a través de otros objetos sin rebotar pero también desencadena eventos de colisión.

- body.isSleepingAllowed – (booleano) permite que los objetos se “duerman”. Valor por defecto es verdadero.

- body.isFixedRotation – (booleano) rotación del cuerpo está bloqueada. El valor predeterminado es falso.

- body.angularVelocity - valor numérico de la velocidad actual angular (rotación), en grados por segundo.

- body.linearDamping - valor numérico de la cantidad de movimiento lineal del cuerpo que es amortiguado es amortiguado. El valor predeterminado es cero.

- body.angularDamping - valor numérico de cuanta rotación del cuerpo debe ser amortiguada. El valor predeterminado es cero.

- body.bodyType – cadena que identifica el tipo de objeto. Los posibles valores son "static" (por defecto), "dynamic"(por defecto) y "kinematic".

    - "static" - no se mueven, y no interactúan unos con otros. Ejemplos de objetos estáticos incluyen suelo, o paredes de una máquina de pinball.

    - "dynamic" - se ven afectados por la gravedad y las colisiones con otros cuerpos.

    - "kinematic" - se ven afectados por fuerzas, pero no por la gravedad. Se establece generalmente para objetos que se pueden arrastrar, al menos durante la duración del evento de arrastre.

Estos son los métodos de los cuerpos físicos:

- body:setLinearVelocity – función que establece en los componentes x e y, la velocidad lineal del objeto en píxeles por segundo.

- body:getLinearVelocity - función que devuelve la velocidad lineal del objeto en x e y en píxeles por segundo

- body:applyForce - función que se le pasa como parámetro los valores de fuerza lineal aplicada en los ejes x e y las coordenadas del objeto en las que se aplica la fuerza. Si el punto de destino es el centro de masas del cuerpo, el cuerpo será empujado en una línea recta; si el objetivo es un punto desplazado del centro de masas, el cuerpo girará alrededor de este centro. Para objetos simétricos el centro de masas coincide con el centro del objeto (object.x, object.y). Ejemplo: myBody:applyForce(500, 2000, myBody.x, myBody.y)

- body:applyTorque - un valor numérico para la fuerza aplicada de rotación. El cuerpo rotará sobre su centro de masas.

- body:applyLinearImpulse - igual que función *applyForce*, excepto que da solo un impulso en un choque único y momentáneo.

- body:applyAngularImpulse - igual que función *applyTorque*, excepto que da solo un impulso angular en un choque único y momentáneo.
- body:resetMassData - Si los datos de masa por defecto para el cuerpo han sido anulados, esta función establece la masa calculada a partir de las formas.

Colisiones

La collision entre objetos que propicia el motor físico, es tratado a través del modelo estándar de eventos – *listeners* de Corona, con tres nuevos tipos de eventos específicos.

Para detección de colisiones generales se debe usar el evento "*collision*" que incluye dos fases, "*began*" y "*ended*" para el momento inicial y final del contacto. Estas fases existen para colisiones normales entre dos cuerpos y como sensor de colisiones. Para que pueda ser tratado se debe implementar un *"collision" listener.*

Hay otros dos tipos de eventos que se usan para el choque entre dos cuerpo (no en sensores), "preCollision" y "postCollision". El primero es un tipo de evento que se dispara antes de que los objetos comiencen a interactuar y el segundo es un tipo de evento que se activa inmediatamente después que los objetos que han interactuado.

Las colisiones se transmiten entre pares de objetos, y pueden ser detectados de forma global, usando un detector de tiempo de ejecución (*Runtime listener*), o localmente en cada uno de los objetos, utilizando una tabla de *listeners*.

7.11 Conectividad

Corona incluye la última versión (v2.02) de las bibliotecas *LuaSocket*. Estos módulos Lua permiten aplicar los protocolos de red comunes como SMTP (envío de mensajes de correo electrónico), HTTP (acceso a la WWW) y FTP (carga y descarga de archivos). También se incluyen funciones de apoyo MIME (codificación común), la manipulación de URL y LTN12 (transferencia y filtrado de datos).

Luasocket es una colección de bibliotecas externas que son preinstaladas en aplicaciones de Corona y no se cargan automáticamente por defecto. Para utilizarlas, debe cargarse de forma explícita cada uno de ellos para que las funciones de las bibliotecas estén disponibles para la aplicación:

```
local socket = require ("socket") o local http = require ("http")
```

Corona permite realizar conexiones http asíncronas. Esta característica le permite realizar llamadas HTTP y HTTPS/SSL asíncronas, utilizando cualquier método válido HTTP ("GET", "POST", etc), así como la adición de encabezados y contenidos. No hace falta dejar el programa a la espera de respuesta desde el servidor sino que una vez se obtiene esta, se genera un evento que nos permite tratar la respuesta.

Para enviar una petición a un servidor, se debe especificar una URL, un método y un detector para el resultado:

network.request(url, method, listener [, params])

El método por defecto es GET y las propiedades del evento respuesta son dos:

- event.response – Una cadena que contiene la respuesta desde el servidor.
- event.isError – valor booleano que devuelve *true* en caso de error de red o *false* en caso contrario.

Este es un ejemplo petición http:

```
local function networkListener( event )
        if ( event.isError ) then
                print( "Network error!")
        else
                print ( "RESPONSE: " .. event.response )
        end
end

-- Access Google over SSL:
network.request( "https://encrypted.google.com", "GET", networkListener)
```

En ciertas ocasiones es necesario especificar cabeceras adicionales en la petición. Ambos pueden especificarse en la tabla de parámetros opcional:

- params.headers – Tabla especificando los valores de cabecera con claves de tipo cadena.
- params.body – Una cadena conteniendo el cuerpo (*body)* HTTP.

```
headers = {}
headers["Content-Type"] = "application/json"
headers["Accept-Language"] = "en-US"
headers.body = "This is an example request body."
```

Para realizar una descarga, se utiliza una función similar a la anterior salvo que se descarga la respuesta a un archivo local en la ruta que se especifique, en lugar de en la memoria caché. Esto se recomienda para respuestas de gran tamaño (por ejemplo, documentos XML), y también puede ser utilizado para la descarga de imágenes a distancia. Este es el formato:

network.download(url, method, listener [, params], destFilename [, baseDir])

El parámetro opcional *baseDir* puede ser *system.DocumentsDirectory* (por defecto) o *system.TemporaryDirectory*.

```
local function networkListener( event )
    if ( event.isError ) then
            print ( "Network error - download failed" )
    else
            myImage = display.newImage( "helloCopy.png",
  system.TemporaryDirectory, 60, 40 )
            myImage.alpha = 0
```

```
                transition.to( myImage, { alpha = 1.0 } )
        end

        print ( "RESPONSE: " .. event.response )
    end

     network.download( "http://developer.anscamobile.com/demo/hello.png", "GET",
       networkListener, "helloCopy.png", system.TemporaryDirectory )
```

Si lo que se desea es descargar específicamente imágenes para mostrarlas en pantalla, existe la siguiente función:

display.loadRemoteImage(url, method, listener [, params], destFilename [, baseDir] [, x, y])

Contiene las coordenadas x e y para situar la imagen en la pantalla.

Las propiedades del evento respuesta son las mismas que los casos anteriores y además tiene una tercera:

- event.target – representa el Nuevo objeto creado después de que la imagen es descargada.

Corona SDK permite mostrar contenido web directamente a través de los denominados *web popup,* que carga local o remotamente páginas web.

La función es:

native.showWebPopup(url [, options])
native.showWebPopup(x, y, width, height, url [, options])

y los parámetros son:

- url – URL de la página web local o remota. Por defecto, la URL es una ruta absoluta a un servidor remoto.

- x, y, width, height – parámetros de posición y tamaño de la ventana del *popup*. Si no se especifica se asume que ocupa toda la pantalla.

- Options – tabla que contiene valores opcionales. options.baseUrl para determinar si la url es relativa o absoluta, options.hasBackground para indicar si tiene un fondo opaco o no y options.urlRequest que designa una función para interceptar los eventos generados en el *web popup*.

7.12 Gestión de la memoria

Los dispositivos móviles tienen una memoria limitada disponible para su uso, por lo que se debe tener cuidado para asegurarse de que el uso total de la memoria de su aplicación se mantiene al mínimo y se gestiona de forma correcta.

Lua realiza automáticamente la gestión de la memoria. Esto significa que no debemos preocuparnos ni de asignar (o reservar) memoria para nuevos objetos ni de liberarla cuando los objetos dejan de ser necesarios. Lua gestiona la memoria automáticamente

ejecutando un recolector de basura (*garbage collector*) de cuando en cuando para eliminar todos los objetos muertos (esos objetos que ya no son accesibles desde Lua). Todos los objetos en Lua son susceptibles de gestión automática: tablas, funciones y cadenas.

Sin embargo, hay ciertos puntos que se deben tener en cuenta ya que se debe indicar al sistema que se debe considerar como basura, es decir, hay que preocuparse por la gestión lógica de la memoria. Por ejemplo, cualquier valor almacenado en una variable global no se considera basura, aunque la aplicación no lo vuelva a utilizar de nuevo.

La gestión de memoria en Corona SDK se debe aplicar sobre los siguientes cinco aspectos:

- Objetos de visualización
- Variables globales
- Runtime listeners
- Temporizadores (Timers)
- Transiciones

La forma de monitorizar la utilización de la memoria en la aplicación es añadir en la aplicación, en el fichero main.lua, el siguiente código que visualiza su utilización

```
local monitorMem = function()

    collectgarbage()
    print( "MemUsage: " .. collectgarbage("count") )

    local textMem = system.getInfo( "textureMemoryUsed" ) / 1000000
    print( "TexMem:   " .. textMem )
end

Runtime:addEventListener( "enterFrame", monitorMem )
```

Objetos de visualización

Estos son los más fáciles de gestionar, ya que son directamente responsables de la creación de objetos de visualización. La forma en que puede evitar las pérdidas de memoria al mostrar objetos es asegurarse de que se quitan aquellos que ya no son necesarios.

Dado un objeto círculo, lo podemos eliminar de la siguiente manera:

```
-- Creación del objeto
local redBall = display.newCircle( 100, 100, 25 )

-- eliminacion del objeto
redBall:removeSelf()
        o
display.remove( redBall )

-- elminación de la referencia
redBall=nil
```

La diferencia entre *display.remove()* y el método de objeto *removeSelf()* es que en primer caso se comprueba que el objeto no ha sido eliminado previamente. Es sólo una manera más segura de eliminar los objetos en Corona.

Las fugas en este caso se presentan con mayor frecuencia cuando los objetos se crean dentro de un bucle, o en alguna parte que es difícil hacer un seguimiento de cada objeto individual. Se debe asegurar que los objetos de visualización son liberados de la memoria cuando ya no son necesarios, especialmente al cambiar de imagen o pantalla.

Variables globales

En Corona SDK se recomienda que se usen la menor cantidad de variables globales (aquellas que se declaran sin usar la palabra local). En el momento que esas variables haya dejado de ser útiles o no vaya a usarse más, se debe asegurar que se les asigna el valor nil para que el recolector pueda liberar su memoria.

Runtime listeners

Cuando se elimina un objeto de visualización, los *listeners* para captar los eventos que están asociados, también se liberan de la memoria. Sin embargo, cuando se agregan *listeners* al objeto Runtime (tiempo de ejecución), por ejemplo, el *listener enterFrame*, nunca se liberan hasta que se eliminan manualmente.

Una pérdida de memoria común que ocurre con los *listeners* en el *Runtime* es cuando un desarrollador añade en una pantalla en particular, pero que se olvida de quitarlo cuando el usuario sale de la pantalla. Cuando vuelve a ella, hay dos *listeners* en el *Runtime* que se ejecutan en uno encima del otro.

Además de poder provocar un error por falta de memoria, se pueden producir errores en la gestión de los eventos con resultados inesperados.

Temporizadores y transiciones

Temporizadores y transiciones son probablemente una de las causas más comunes de errores por memoria.
Un método para manejar estas es almacenar todos los temporizadores y las transiciones en una tabla, de modo que cuando se sabe que han terminado, se puedan cancelar todos ellos a la vez.

Si se agrega el siguiente código al archivo main.lua (u otro módulo), se puede hacer fácilmente un seguimiento de los temporizadores / transiciones y cancelarlos todos a la vez, siempre y cuando sea necesario:

```
timerStash = {}
transitionStash = {}

function cancelAllTimers()
    local k, v

    for k,v in pairs(timerStash) do
        timer.cancel( v )
        v = nil; k = nil
```

```
        end

        timerStash = nil
        timerStash = {}
    end

    --

    function cancelAllTransitions()
        local k, v

        for k,v in pairs(transitionStash) do
            transition.cancel( v )
            v = nil; k = nil
        end

        transitionStash = nil
        transitionStash = {}
    end
```

Y entonces cuando se crea un nuevo temporizador o una transición:

```
    timerStash.newTimer = timer.performWithDelay( ...

    transitionStash.newTransition = transition.to( myObject { ...
```

Entonces se puede llamar a las funciones cancelAllTimers() y cancelAllTransitions() para pararlos de una vez.

8 ANEXO I – Detalle de funciones en Corona SDK

8.1 Librerías estándar de Lua

Corona incluye las mismas bibliotecas estándar Lua que son parte del estándar de Lua. Estas bibliotecas proporcionan una funcionalidad útil y básica. Se agrupan en las siguientes categorías:

- Biblioteca básica
- Módulos (bibliotecas externas)
- Manipulación de cadenas
- Manipulación de tablas
- Funciones matemáticas
- Funciones de entrada y salida
- Funciones de sistema operativo

A excepción de la biblioteca básica, cada biblioteca ofrece todas sus funciones como propiedades de una tabla o como los métodos de sus objetos. Esto crea una agrupación lógica de funciones y es la manera de crear Lua un espacio de nombres de conjuntos diferentes de funcionalidad.

8.1.1 Biblioteca básica

La biblioteca básica proporciona algunas funciones del núcleo de Lua.

assert (v [, mensaje])
Activa un error cuando el valor de su argumento v es falso (por ejemplo, nil o false); en otro caso retorna todos sus argumentos. mensaje es un mensaje de error; cuando está ausente se utiliza por defecto "assertion failed!".

error (mensaje [, nivel])
Termina la última función protegida llamada, estableciendo mensaje como mensaje de error. La función error nunca retorna.
Normalmente error añade, al comienzo del mensaje, cierta información acerca de la posición del error. El argumento nivel especifica cómo obtener la posición del error. Con nivel 1 (por defecto) la posición del error es donde fue invocada la función error. Nivel 2 apunta el error hacia el lugar en que fue invocada la función que llamó a error; y así sucesivamente. Pasar un valor 0 como nivel evita la adición de la información de la posición al mensaje.

_G
Una variable global (no una función) que almacena el entorno global (o sea, _G._G = _G). Lua mismo no usa esta variable; cambiar su valor no afecta ningún entorno, ni viceversa. (Se usa setfenv para cambiar entornos.)

getfenv ([f])
Retorna el entorno actualmente en uso por la función. f puede ser una función Lua o un número que especifica la función a ese nivel de la pila: nivel 1 es la función que invoca a

getfenv. Si la función dada no es una función Lua o si f es 0, getfenv retorna el entorno global. El valor por defecto de f es 1.

getmetatable (objeto)
Si objeto no tiene una metatabla devuelve nil. En otro caso, si la metatabla del objeto tiene un campo "__metatable" retorna el valor asociado, o si no es así retorna la metatabla del objeto dado.

ipairs (t)
Retorna tres valores: una función iteradora, la tabla t, y 0, de tal modo que la construcción

```
for i,v in ipairs(t) do bloque end
```

iterará sobre los pares (1,t[1]), (2,t[2]), ···, hasta la primera clave entera con un valor nil en la tabla.

next (tabla [, índice])
Permite al programa recorrer todos los campos de una tabla. Su primer argumento es una tabla y su segundo argumento es un índice en esta tabla. next retorna el siguiente índice de la tabla y su valor asociado. Cuando se invoca con nil como segundo argumento next retorna un índice inicial y su valor asociado. Cuando se invoca con el último índice o con nil en una tabla vacía next retorna nil. Si el segundo argumento está ausente entonces se interpreta como nil. En particular se puede usar next(t) para comprobar si una tabla está vacía.

El orden en que se enumeran los índices no está especificado, incluso para índices numéricos. (Para recorrer una tabla en orden numérico úsese el for numérico o la función ipairs.)

El comportamiento de next es indefinido si durante el recorrido se asigna un valor a un campo no existente previamente en la tabla. No obstante se pueden modificar campos existentes. En particular se pueden borrar campos existentes.

pairs (t)
Retorna tres valores: la función next, la tabla t, y nil, por lo que la construcción

```
for k,v in pairs(t) do bloque end
```

iterará sobre todas las parejas clave-valor de la tabla t.

Véase next para las precauciones a tomar cuando se modifica la tabla durante las iteraciones.

pcall (f, arg1, ···)
Invoca la función f con los argumentos dados en modo protegido. Esto significa que ningún error dentro de f se propaga; en su lugar pcall captura el error y retorna un código de estatus. Su primer resultado es el código de estatus (booleano), el cual es verdadero si

la llamada tiene éxito sin errores. En ese caso pcall también devuelve todos los resultados de la llamada después del primer resultado. En caso de error pcall retorna false más un mensaje de error.

print (···)
Recibe cualquier número de argumentos e imprime sus valores en el fichero estándar de salida (stdout), usando tostring como función para convertir los argumentos a strings. print no está diseñada para salida formateada sino sólo como una manera rápida de mostrar valores, típicamente para la depuración del código. Para salida formateada úsese string.format.

rawequal (v1, v2)
Verifica si v1 es igual a v2, sin invocar ningún metamétodo. Devuelve un booleano.

rawget (tabla, índice)
Obtiene el valor real de tabla[índice] sin invocar ningún metamétodo. tabla debe ser una tabla e índice cualquier valor diferente de nil.

rawset (tabla, índice, valor)
Asigna valor a tabla[índice] sin invocar ningún metamétodo. tabla debe ser una tabla, índice cualquier valor diferente de nil y valor un valor cualquiera de Lua.

select (índice, ···)
Si índice es un número retorna todos los argumentos después del número índice. En otro caso índice debe ser el string "#", y select retorna el número total de argumentos extra que recibe.

setfenv (f, tabla)
Establece el entorno que va a ser usado por una función. f puede ser una función Lua o un número que especifica la función al nivel de pila: nivel 1 es la función que invoca a setfenv. setfenv retorna la función dada.

Como caso especial, cuando f es 0 setfenv cambia el entorno del proceso que está en ejecución. En este caso setfenv no retorna valores.

setmetatable (tabla, metatabla)
Establece la metatabla de una tabla dada. (No se puede cambiar la metatabla de otros tipos desde Lua, sino sólo desde C.) Si metatabla es nil entonces se elimina la metatabla de la tabla dada. Si la metatabla original tiene un campo "__metatable" se activa un error.

Esta función retorna tabla.

tonumber (e [, base])

Intenta convertir su argumento en un número. Si el argumento es ya un número o un string convertible a un número entonces tonumber retorna este número; en otro caso devuelve nil.

Un argumento opcional especifica la base para interpretar el número. La base puede ser cualquier entero entre 2 y 36, ambos inclusive. En bases por encima de 10 la letra 'A' (en mayúscula o minúscula) representa 10, 'B' representa 11, y así sucesivamente, con 'Z' representando 35. En base 10 (por defecto), el número puede tener parte decimal, así como un exponente opcional (véase §2.1). En otras bases sólo se aceptan enteros sin signo.

tostring (e)
Recibe un argumento de cualquier tipo y lo convierte en un string con un formato razonable. Para un control completo de cómo se convierten los números, úsese string.format.

Si la metatabla de e tiene un campo "__tostring" entonces tostring invoca al correspondiente valor con e como argumento y usa el resultado de la llamada como su propio resultado.

type (v)
Retorna el tipo de su único argumento, codificado como string. Los posibles resultados de esta función son "nil" (un string, no el valor nil), "number", "string", "boolean, "table", "function", "thread" y "userdata".

unpack (lista [, i [, j]])
Retorna los elementos de una tabla dada. Esta función equivale a

```
return lista[i], lista[i+1], ···, lista[j]
```

excepto que este código puede ser escrito sólo para un número fijo de elementos. Por defecto i es 1 y j es la longitud de la lista, como se define a través del operador longitud .

8.1.2 Módulos (bibliotecas externas)

Corona admite la funcionalidad de módulos de Lua para crear y cargar bibliotecas externas.
En la actualidad, el SDK incluye varias bibliotecas externas como "ui.lua" (para la creación de los botones con *rollover*) y "sprite.lua" (para crear *sprites* animados, o "clips de película").

Estas librerías se pueden encontrar en los proyectos de muestra "Button" y "clip de película", ubicado en el directorio del código de ejemplo del SDK.

Crear bibliotecas externas

Se pueden crear módulos externos de Lua, lo cual es útil para la organización de grandes proyectos en varios archivos, o la creación de bibliotecas reutilizables de código para futuros proyectos.

La forma más fácil de crear un módulo es utilizar el siguiente formato y guardarlo en un fichero con extensión .lua en el mismo directorio que el archivo del proyecto main.lua:

```
module(..., package.seeall)

-- Declare the functions you want in your module
function hello()
     print ("Hello, module")
end
```

Cargar bibliotecas externas
Para cargar un módulo de directorio de su proyecto, se usa require(NombreModulo) al comienzo del archivo main.lua. Las funciones en el módulo, estarán disponibles con el formato NombreModulo.nombreFuncion ().

```
-- Carga biblioteca
local testlib = require("testlib")

-- llamada a función hello().
testlib.hello()

-- la misma function en el cache same function
local hello = testlib.hello()

-- invocaciones futuras a la función son más rápidas
hello()
```

8.1.3 Manipulación de cadenas

Esta biblioteca proporciona funciones genéricas de manejo de strings, tales como encontrar y extraer substrings y detectar patrones. Cuando se indexa un string en Lua el primer carácter está en la posición 1 (no en 0 como en C). Se permite el uso de índices negativos que se interpretan como indexado hacia atrás, desde el final del string. Por tanto el último carácter del string está en la posición -1, y así sucesivamente.

La biblioteca de strings proporciona todas sus funciones en la tabla string. También establece una metatabla para string donde el campo __index apunta a la misma metatabla. Por tanto, se pueden usar las funciones de manejo de string en un estilo orientado a objetos. Por ejemplo, string.byte(s, i) puede ponerse s:byte(i).

string.byte (s [, i [, j]])
Devuelve los códigos numéricos internos de los caracteres s[i], s[i+1], ···, s[j]. El valor por defecto de i es 1; el valor por defecto de j es i.

Téngase en cuenta que los códigos numéricos no son necesariamente portables de unas plataformas a otras.

string.char (···)
Recibe cero o más enteros. Devuelve un string con igual longitud que el número de argumentos, en el que cada carácter tiene un código numérico interno igual a su correspondiente argumento.

Téngase en cuenta que los códigos numéricos no son necesariamente portables de unas plataformas a otras.

string.dump (function)
Devuelve un string que contiene la representación binaria de la función dada, de tal manera que una llamada posterior a loadstring con este string devuelve una copia de la función. func debe ser una función Lua sin upvalues.

string.find (s, patrón [, inicio [, básica]])
Busca la primera aparición de patrón en el string s. Si la encuentra, find devuelve los índices de s donde comienza y acaba la aparación; en caso contrario retorna nil. Un tercer argumento numérico opcional inicio especifica dónde comenzar la búsqueda; su valor por defecto es 1 y puede ser negativo. Un valor true como cuarto argumento opcional básica desactiva las utilidades de detección de patrones, realizando entonces la función una operación de "búsqueda básica de substring", sin caracteres "mágicos" en el patrón. Téngase en cuenta que si se proporciona el argumento básica también debe proporcionarse el argumento inicio.

Si el patrón tiene capturas entonces en una detección con éxito se devuelven los valores capturados, después de los dos índices.

string.format (formato, ···)
Devuelve una versión formateada de sus argumentos (en número variable) siguiendo la descripción dada en su primer argumento (formato, que debe ser un string). El string de formato sigue las mismas reglas que la familia de funciones C estándar printf. Las únicas diferencias son que las opciones/modificadores *, l, L, n, p, y h no están soportadas, y que existe una opción extra q. Esta última opción da formato a un string en una forma adecuada para ser leída de manera segura de nuevo por el intérprete de Lua: el string es escrito entre dobles comillas, y todas las dobles comillas, nuevas líneas, ceros y barras inversas del string se sustituyen por las secuencias de escape adecuadas en la escritura. Por ejemplo, la llamada

```
string.format('%q', 'un string con "comillas" y \n nueva línea')
```

producirá el string:
"un string con \"comillas\" y \
 nueva línea"

Las opciones c, d, E, e, f, g, G, i, o, u, X y x esperan un número como argumento, mientras que q y s esperan un string.

Esta función no acepta valores de string que contengan caracteres cero, excepto como argumentos de la opción q.

string.gmatch (s, patrón)
Devuelve una función iteradora que, cada vez que se invoca, retorna las siguientes capturas del patrón en el string s.

Si el patrón no produce capturas entonces la coincidencia completa se devuelve en cada llamada.

Como ejemplo, el siguiente bucle

```
s = "hola mundo desde Lua"

for w in string.gmatch(s, "%a+") do
  print(w)
end
```

iterará sobre todas las palabras del string s, imprimiendo una por línea. El siguiente ejemplo devuelve en forma de tabla todos los pares clave=valor del string dado:

```
t = {}
s = "desde=mundo, a=Lua"
for k, v in string.gmatch(s, "(%w+)=(%w+)") do
  t[k] = v
end
```

Para esta función, un '^' al principio de un patrón no funciona como un ancla, sino que previene la iteración.
string.gsub (s, patrón, reemplazamiento [, n])
Devuelve una copia de s en la que todas (o las n primeras, si se especifica el argumento opcional) las apariciones del patrón han sido reemplazadas por el reemplazamiento especificado, que puede ser un string, una tabla o una función. gsub también devuelve, como segundo valor, el número total de coincidencias detectadas.

Si reemplazamiento es un string entonces su valor se usa en la sustitución. El carácter % funciona como un carácter de escape: cualquier secuencia en reemplazamiento de la forma %n, con n entre 1 y 9, significa el valor de la captura número n en el substring (véase más abajo). La secuencia %0 significa toda la coincidencia. La secuencia %% significa un carácter porcentaje %.

Si reemplazamiento es una tabla entonces en cada captura se devuelve el elemento de la tabla que tiene por clave la primera captura; si el patrón no proporciona ninguna captura entonce toda la coincidencia se utiliza como clave.

Si reemplazamiento es una función entonces la misma es invocada cada vez que exista una captura con todos los substrings capturados pasados como argumentos en el mismo orden; si no existen capturas entonces toda la coincidencia se pasa como un único argumento.

Si el valor devuelto por la tabla o por la llamada a la función es un string o un número, entonces se usa como string de reemplazamiento; en caso contrario si es false o nil, entonces no se realiza ninguna sustitución (esto es, la coincidencia original se mantiene en el string).

He aquí algunos ejemplos:

```
x = string.gsub("hola mundo", "(%w+)", "%1 %1")
--> x="hola hola mundo mundo"

x = string.gsub("hola mundo", "%w+", "%0 %0", 1)
--> x="hola hola mundo"

x = string.gsub("hola mundo desde Lua", "(%w+)%s*(%w+)", "%2 %1")
--> x="mundo hola Lua desde"

x = string.gsub("casa = $HOME, usuario = $USER", "%$(%w+)", os.getenv)
--> x="casa = /home/roberto, usuario = roberto"

x = string.gsub("4+5 = $return 4+5$", "%$(.-)%$", function (s)
      return loadstring(s)()
    end)
--> x="4+5 = 9"

local t = {nombre="lua", versión="5.1"}
x = string.gsub("$nombre-$versión.tar.gz", "%$(%w+)", t)
--> x="lua-5.1.tar.gz"
```

string.len (s)
Recibe un string y devuelve su longitud. El string vacío "" tiene longitud 0. Los caracteres cero dentro del string también se cuentan, por lo que "a\000bc\000" tiene longitud 5.

string.lower (s)
Recibe un string y devuelve una copia del mismo con todas las letras mayúsculas cambiadas a minúsculas. El resto de los caracteres permanece sin cambios. La definición de letra mayúscula depende del sistema local.

string.match (s, patrón [, inicio])
Busca la primera aparición del patrón en el string s. Si encuentra una, entonces match retorna la captura del patrón; en caso contrario devuelve nil. Si el patrón no produce ninguna captura entonces se devuelve la coincidencia completa. Un tercer y opcional argumento numérico inicio especifica dónde comenzar la búsqueda; su valor por defecto es 1 y puede ser negativo.

string.rep (s, n)
Devuelve un string que es la concatenación de n copias del string s.

string.reverse (s)
Devuelve un string que es el original s invertido.

string.sub (s, i [, j])
Retorna el substring de s que comienza en i y continúa hasta j; i y j pueden ser negativos. Si j está ausente entonces se asume que vale -1 (equivalente a la longitud del string). En

particular, la llamada string.sub(s,1,j) retorna un prefijo de s con longitud j, y string.sub(s, -i) retorna un sufijo de s con longitud i.

string.upper (s)
Recibe un string y devuelve una copia del mismo con todas las letras minúsculas cambiadas a mayúsculas. El resto de los caracteres permanece sin cambios. La definición de letra minúscula depende del sistema local.

Patrones
Clases de caracteres:
Se usan clases de caracteres para representar conjuntos de caracteres. Están permitidas las siguientes combinaciones para describir una clase de caracteres:

- x: (donde x no es uno de los caracteres mágicos ^$()%.[]*+-?) representa el propio caracter x.
- .: (un punto) representa cualquier carácter.
- %a: representa cualquier letra.
- %c: representa cualquier carácter de control.
- %d: representa cualquier dígito.
- %l: representa cualquier letra minúscula.
- %p: representa cualquier carácter de puntuación.
- %s: representa cualquier carácter de espacio.
- %u: representa cualquier letra mayúscula.
- %w: representa cualquier carácter alfanumérico.
- %x: representa cualquier dígito hexadecimal.
- %z: representa el carácter con valor interno 0 (cero).
- %x: (donde x es cualquier carácter no alfanumérico) representa el carácter x. Ésta es la manera estándar de "escapar" los caracteres mágicos. Cualquier caracter de puntuación (incluso los no mágicos) pueden ser precedidos por un signo de porcentaje '%' cuando se quieran representarse a sí mismos en el patrón.

[conjunto]: representa la clase que es la unión de todos los caracteres en el conjunto. Un rango de caracteres puede ser especificado separando el carácter del principio y del final mediante un guión '-'. Todas las clases del tipo %x descritas más arriba pueden ser también utilizadas como componentes del conjunto. Todos los otros caracteres en el conjunto se representan a sí mismos. Por ejemplo, [%w_] (o [_%w]) representa cualquier carácter alfanumérico o el subrayado, [0-7] representa un dígito octal, y [0-7%l%-] representa un dígito octal, una letra minúscula o el carácter '-'.
La interacción entre los rangos y las clases no está definida. Por tanto, patrones como [%a-z] o [a-%%] carecen de significado.

[^conjunto]: representa el complemento de conjunto, donde conjunto se interpreta como se ha indicado más arriba.
Para todas las clases representadas por letras simples (%a, %c, etc.) las correspondientes letras mayúsculas representan la clase complementaria. Por ejemplo, %S representa cualquier carácter no espacio.

Las definiciones de letra, espacio y otros grupos de caracteres dependen del sistema local. En particular, la clase [a-z] puede no ser equivalente a %l.

Cada elemento de un patrón puede ser :

- una clase de carácter simple, que equivale a cualquier carácter simple de la clase;
- una clase de carácter simple seguida por '*', que equivale a 0 ó más repeticiones de los caracteres de la clase. Estos elementos de repetición siempre equivaldrán a la secuencia de caracteres más larga posible;
- un clase de carácter simple seguida por '+', que equivale a 1 ó más repeticiones de los caracteres de la clase. Estos elementos de repetición siempre equivaldrán a la secuencia de caracteres más larga posible;
- un clase de carácter simple seguida por '-', que también equivale a 0 ó más repeticiones de los caracteres de la clase. Al contrario que '*', Estos elementos de repetición siempre equivaldrán a la secuencia de caracteres más corta posible;
- una clase de carácter simple seguida por '?', que equivale a 0 ó 1 apariciones de un carácter de la clase;
- %n, para n entre 1 y 9; este elemento equivale a un substring igual a la captura número n;
- %bxy, donde x e y son dos caracteres diferentes; este elemento equivale a strings que comienzan con x, finalizan con y, estando equilibrados x e y. Esto significa que, iniciando un contador a 0, si se lee el string de izquierda a derecha, sumando +1 por cada x que aparezca y -1 por cada y, el y final es el primero donde el contador alcanza 0. Por ejemplo, el elemento %b() equivale a una expresión con paréntesis emparejados.

Patrón:
Un patrón es una secuencia de elementos de patrón. Un '^' al comienzo de un patrón ancla la búsqueda del patrón al comienzo del string en el que se produce la búsqueda. Un '$' al final de un patrón ancla la búsqueda del patrón al final del string en el que se produce la búsqueda. En otras posiciones '^' y '$' no poseen un significado especial y se representan a sí mismos.

Capturas:
Un patrón puede contener subpatrones encerrados entre paréntesis que describen capturas. Cuando sucede una coincidencia entre un patrón y un string dado, los substrings que concuerdan con lo indicado entre paréntesis en el patrón, son almacenados (capturados) para uso futuro. Las capturas son numeradas de acuerdo a sus paréntesis izquierdos. Por ejemplo, en el patrón "(a*(.)%w(%s*))", la parte del string que concuerda con "a*(.)%w(%s*)" se guarda en la primera captura (y por tanto tiene número 1); el carácter que concuerda con "." se captura con el número 2, y la parte que concuerda con "%s*" tiene el número 3.

Como caso especial, la captura vacía () retorna la posición actual en el string (un número). Por ejemplo, si se aplica el patrón "()aa()" al string "flaaap", dará dos capturas: 3 y 5.

Un patrón no puede contener caracteres cero. Se debe usar %z en su lugar

8.1.4 Manipulación de tablas

Esta biblioteca proporciona funciones genéricas para manejo de tablas. Todas estas funciones están definidas dentro de la tabla table.
La mayoría de las funciones en la biblioteca de tablas asume que las mismas representan arrays o listas (o sea, están indexadas numéricamente). Para estas funciones, cuando hablamos de la "longitud" de una tabla queremos decir el resultado del operador longitud (#).

table.concat (tabla [, separador [, i [, j]]])
Dado una table donde todos sus elementos son strings o números devuelve tabla[i]..separador..tabla[i+1] ··· separador..tabla[j]. El valor por defecto de separador es el string vacío, el valor por defecto de i es 1 y el valor por defecto de j es la longitud de la tabla. Si i es mayor que j, la función devuelve un string vacío.

table.insert (tabla, [posición,] valor)
Inserta el elemento valor en la posición dada en la tabla, desplazando hacia adelante otros elementos para abrir hueco, si es necesario. El valor por defecto de posición es n+1, donde n = #tabla es la longitud de la tabla (véase §2.5.5), de tal manera que table.insert(t,x) inserta x al final de la tabla t.

table.maxn (tabla)
Devuelve el mayor índice numérico positivo de una tabla dada o cero si la tabla no tiene índices numéricos positivos. (Para hacer su trabajo esta función realiza un barrido lineal de la tabla completa.)

table.remove (tabla [, posición])
Elimina de tabla el elemento situado en la posición dada, desplazando hacia atrás otros elementos para cerrar espacio, si es necesario. Devuelve el valor del elemento eliminado. El valor por defecto de posición es n, donde n es la longitud de la tabla, por lo que la llamada table.remove(t) elimina el último elemento de la tabla t.

table.sort (tabla [, comparador])
Ordena los elementos de la tabla en un orden dado modificando la propia tabla, desde table[1] hasta table[n], donde n es la longitud de la tabla. Si se proporciona el argumento comparador éste debe ser una función que recibe dos elementos de la tabla y devuelve verdadero cuando el primero es menor que el segundo (por lo que not comparador(a[i+1],a[i]) será verdadero después de la ordenación). Si no se proporciona una función comparador entonces se usa el operador estándar < de Lua.

El algoritmo de ordenación no es estable; esto es, los elementos considerados iguales por la ordenación dada pueden sufrir cambios de orden relativos después de la ordenación.

8.1.5 Funciones matemáticas

Esta biblioteca es una interface a la biblioteca matemática estándar de C. Proporciona todas sus funciones dentro de la tabla math.

math.abs (x)
Devuelve el valor absoluto de x.

math.acos (x)
Devuelve el arco coseno de x (en radianes).

math.asin (x)
Devuelve el arco seno de x (en radianes).

math.atan (x)
Devuelve el arco tangente de x (en radianes).

math.atan2 (y, x)
Devuelve el arco tangente de y/x (en radianes), pero usa los signos de ambos argumentos para determinar el cuadrante del resultado. (También maneja correctamente el caso en que x es cero.)

math.ceil (x)
Devuelve el menor entero mayor o igual que x.

math.cos (x)
Devuelve el coseno de x (se asume que está en radianes).

math.cosh (x)
Devuelve el coseno hiperbólico de x.

math.deg (x)
Devuelve en grados sexagesimales el valor de x (dado en radianes).

math.exp (x)
Devuelve el valor de ex.

math.floor (x)
Devuelve el mayor entero menor o igual que x.

math.fmod (x, y)
Devuelve el resto de la división de x por y.

math.frexp (x)
Devuelve m y e tales que $x = m\ 2^e$, e es un entero y el valor absoluto de m está en el intervalo [0.5, 1) (o cero cuando x es cero).

math.huge
El valor HUGE_VAL, un valor más grande o igual que otro valor numérico cualquiera.

math.ldexp (m, e)
Devuelve $m\ 2^e$ (e debe ser un entero).

math.log (x)
Devuelve el logaritmo natural de x.

math.log10 (x)
Devuelve el logaritmo decimal (base 10) de x.

math.max (x, ···)
Devuelve el mayor valor de entre sus argumentos.

math.min (x, ···)
Devuelve el menor valor de entre sus argumentos.

math.modf (x)
Devuelve dos números, las partes entera y fraccional de x .

math.pi
El valor de pi.

math.pow (x, y)
Devuelve xy. (Se puede también usar la expresión x^y para calcular este valor.)

math.rad (x)
Devuelve en radianes el valor del ángulo x (dado en grados sexagesimales).

math.random ([m [, n]])
Esta función es un interface a rand, generador simple de números pseudo-aleatorios proporcionado por el ANSI C. (Sin garantías de sus propiedades estadísticas.)

Cuando se invoca sin argumentos devuelve un número pseudoaleatorio real uniforme en el rango [0,1). Cuando se invoca con un número entero m, math.random devuelve un número pseudoaleatorio entero uniforme en el rango [1, m]. Cuando se invoca con dos argumentos m y n enteros, math.random devuelve un número pseudoaleatorio entero uniforme en el rango [m, n].

math.randomseed (x)
Establece x como "semilla" para el generador de números pseudoaleatorios: iguales semillas producen iguales secuencias de números.

math.sin (x)
Devuelve el seno de x (se asume que está en radianes).

math.sinh (x)
Devuelve el seno hiperbólico de x.

math.sqrt (x)
Devuelve la raiz cuadrada de x. (Se puede usar también la expresión x^0.5 para calcular este valor.)

math.tan (x)
Devuelve la tangente de x (se asume que está en radianes).

math.tanh (x)
Devuelve la tangente hiperbólica de x.

8.1.6 Funciones de entrada y salida

La biblioteca de entrada/salida (I/O de sus siglas en inglés) proporciona dos estilos diferentes de manejo de ficheros. El primero de ellos usa descriptores de fichero implícitos; esto es, existen dos ficheros por defecto, uno de entrada y otro de salida, y las operaciones se realizan sobre éstos. El segundo estilo usa descriptores de fichero explícitos.

Cuando se usan descriptores implícitos todas las operaciones soportadas están en la tabla io. Cuando se usan descriptores explícitos, la operación io.open devuelve un descriptor de fichero y todas las operaciones se proporcionan como métodos asociados al descriptor.

La tabla io también proporciona tres descriptores de fichero predefinidos con sus significados usuales en C: io.stdin, io.stdout e io.stderr. La biblioteca de entrada/salida nunca cierra esos ficheros.

A no ser que se especifique, todas las funciones de entrada/salida devuelven nil en caso de fallo (más un mensaje de error como segundo resultado y un código de error dependiente del sistema como un tercer resultado) y valores diferentes de nil si hay éxito.

Manipulación de ficheros implícita

io.close ([descriptor_de_fichero])
Equivalente a descriptor_de_fichero:close(). Sin argumento cierra el fichero de salida por defecto.

io.flush ()
Equivalente a descriptor_de_fichero:flush aplicado al fichero de salida por defecto.

io.input ([descriptor_de_fichero | nombre_de_fichero])
Cuando se invoca con un nombre de fichero entonces lo abre (en modo texto), y establece su manejador de fichero como fichero de entrada por defecto. Cuando se llama con un descriptor de fichero simplemente lo establece como manejador para el fichero de entrada por defecto. Cuando se invoca sin argumento devuelve el fichero por defecto actual.

En caso de errores esta función activa error en lugar de devolver un código de error.

io.lines ([nombre_de_fichero])
Abre el fichero de nombre dado en modo lectura y devuelve una función iteradora que, cada vez que es invocada, devuelve una nueva línea del fichero. Por tanto, la construcción

```
for linea in io.lines(nombre_de_fichero) do bloque end
```

iterará sobre todas las líneas del fichero. Cuando la función iteradora detecta el final del fichero devuelve nil (para acabar el bucle) y cierra automáticamente el fichero.
La llamada a io.lines() (sin nombre de fichero) equivale a io.input():lines(); esto es, itera sobre todas las líneas del fichero por defecto de entrada. En ese caso no cierra el fichero cuando acaba el bucle.

io.open (nombre_de_fichero [, modo])
Esta función abre un fichero, en el modo especificado en el string mode. Devuelve un descriptor de fichero o, en caso de error, nil además de un mensaje de error.

El string que indica modo puede ser uno de los siguientes:

- "r": modo lectura (por defecto);
- "w": modo escritura;
- "a": modo adición;
- "r+": modo actualización, todos los datos preexistentes se mantienen;
- "w+": modo actualización, todos los datos preexistentes se borran;
- "a+": modo adición con actualización, todos los datos preexistentes se mantienen, y la escritura se permite sólo al final del fichero.

El string que indica el modo puede contener también 'b' al final, lo que es necesario en algunos sistemas para abrir el fichero en modo binario. Este string es exactamente el que se usa en la función estándar de C fopen.

io.output ([descriptor_de_fichero | nombre_de_fichero])
Similar a io.input, pero operando sobre el fichero por defecto de salida.

io.popen (prog [, modo])
Comienza a ejecutar el programa prog en un proceso separado y retorna un descriptor de fichero que se puede usar para leer datos que escribe prog (si modo es "r", el valor por defecto) o para escribir datos que lee prog (si modo es "w").

Esta función depende del sistema operativo y no está disponible en todas las plataformas.

io.read (···)
Equivalente a io.input():read.

io.tmpfile ()
Devuelve un descriptor de fichero para un fichero temporal. Éste se abre en modo actualización y se elimina automáticamente cuando acaba el programa.

io.type (objeto)
Verifica si objeto es un descriptor válido de fichero. Devuelve el string "file" si objeto es un descriptor de fichero abierto, "closed file" si objeto es un descriptor de fichero cerrado, o nil si objeto no es un descriptor de fichero.

io.write (···)
Equivalente a io.output():write.

Manipulación de ficheros explícita
fichero:close ()
Cierra el descriptor de fichero. Téngase en cuenta que los ficheros son cerrados automáticamente cuando sus descriptores se eliminan en un ciclo de liberación de memoria, pero que esto toma un tiempo impredecible de ejecución.

fichero:flush ()
Salva cualquier dato escrito en fichero.

fichero:lines ()
Devuelve una función iteradora que, cada vez que es invocada, devuelve una nueva línea leída del fichero. Por tanto, la construcción

```
for linea in fichero:lines() do bloque end
```

iterará sobre todas las líneas del fichero. (A diferencia de io.lines, esta función no cierra el fichero cuando acaba el bucle.)

fichero:read (···)
Lee en el fichero, de acuerdo el formato proporcionado, el cual especifica qué leer. Para cada formato, la función devuelve un string (o un número) con los caracteres leídos, o nil si no pudo leer los datos con el formato especificado. Cuando se invoca sin formato se usa uno por defecto que lee la próxima línea completa (véase más abajo).

Los formatos disponibles son

- "*n": lee un número; éste es el único formato que devuelve un número en lugar de un string.
- "*a": lee el resto del fichero completo, empezando en la posición actual. Al final del fichero devuelve un string vacío.
- "*l": lee la próxima línea (saltándose el final de línea), retornando nil al final del fichero. Éste es el formato por defecto.
- un número: lee un string con como máximo este número de caracteres, devolviendo nil si se llega al final del fichero. Si el número es cero no lee nada y devuelve un string vacío, o nil si se alcanza el final del fichero.

fichero:seek ([de_dónde] [, desplazamiento])
Establece (o solicita) la posición actual (del puntero de lectura/escritura) en el fichero, medida desde el principio del fichero, en la posición dada por desplazamiento más la base especificada por el string dónde, como se especifica a continuación:
- "set": sitúa la posición base en 0 (comienzo del fichero);
- "cur": sitúa la posición base en la actual;
- "end": sitúa la posición base al final del fichero.

En caso de éxito la función seek retorna la posición final (del puntero de lectura/escritura) en el fichero medida en bytes desde el principio del fichero. Si la llamada falla retorna nil, más un string describiendo el error.

El valor por defecto de dónde es "cur", y para desplazamiento es 0. Por tanto, la llamada fichero:seek() devuelve la posición actual, sin cambiarla; la llamada fichero:seek("set") establece la posición al principio del fichero (y devuelve 0); y la llamada fichero:seek("end") establece la posición al final del fichero y devuelve su tamaño.

fichero:setvbuf (modo [, tamaño])
Establece un modo buffer para un fichero de salida. El argumento modo puede ser uno de estos tres:

- "no": sin buffer; el resultado de cualquier operación de salida se produce inmediatamente.
- "full": con buffer completo; la operación de salida se realiza sólo cuando el buffer está lleno o cuando se invoca explícitamente la función flush en el descriptor del fichero.
- "line": con buffer de línea; la salida se demora hasta que se produce una nueva línea en la salida o existe una entrada de algún fichero especial (como una terminal).

Para los dos últimos casos, tamaño especifica el tamaño del buffer, en bytes. El valor por defecto es un tamaño adecuado.

fichero:write (···)
Escribe el valor de sus argumentos en el fichero dado por su fichero. Los argumentos pueden ser strings o números. Para escribir otros valores úsese tostring o string.format antes que write.

8.1.7 Funciones de sistema operativo

Esta biblioteca está implementada a través de la tabla os.

os.clock ()
Devuelve una aproximación al total de segundos de CPU usados por el programa.

os.date ([formato [, tiempo]])
Devuelve un string o una tabla conteniendo la fecha y hora, formateada de acuerdo con el string dado en formato.

Si el argumento tiempo está presente entonces ese tiempo concreto es el que se formatea (véase la función os.time para una descripción de este valor). En caso contrario, date formatea el tiempo actual.

Si formato comienza con '!' entonces el tiempo se formatea de acuerdo al Tiempo Universal Coordinado. Después de este carácter opcional, si formato es *t entonces date devuelve una tabla con los siguientes campos:

- year (cuatro dígitos),
- month (1--12),
- day (1--31),
- hour (0--23),
- min (0--59),
- sec (0--61),
- wday (día de la semana, el domingo es 1),

- yday (día dentro del año),
- e isdst (booleano, verdadero si es horario de verano).

Si formato no es *t entonces date devuelve el tiempo como un string, formateado de acuerdo con las mismas reglas que la función strftime de C.

Cuando se invoca sin argumentos date devuelve una representación razonable de la fecha y la hora que depende de la máquina y del sistema local (esto es, os.date() equivale a os.date("%c")).

os.difftime (t2, t1)
Devuelve el número de segundos desde el instante t1 hasta el t2. En POSIX, Windows y algunos otros sistemas este valor es exactamente t2-t1.

os.exit ([código])
Invoca la función exit de C, con un código entero opcional, para terminar el programa anfitrión. El valor por defecto de código es el valor correspondiente a éxito.

os.remove (nombre_de_fichero)
Elimina el fichero o directorio dado. Los directorios deben estar vacíos para poder ser eliminados. Si la función falla retorna nil, más un string describiendo el error.

os.rename (nombre_viejo, nombre_nuevo)
Renombra un fichero o directorio de nombre_viejo a nombre_nuevo. Si la función falla retorna nil, más un string describiendo el error.

os.time ([tabla])
Devuelve el tiempo actual cuando se llama sin argumentos, o un tiempo representando la fecha y hora especificadas en la tabla dada. Ésta debe tener los campos year, month y day, y puede tener los campos hour, min, sec e isdst (para una descripción de esos campos, véase la función os.date).

El valor retornado es un número, cuyo significado depende del sistema. En POSIX, Windows y algunos otros sistemas este número cuenta el número de segundos desde alguna fecha inicial dada (la "época"). En otros sistemas el significado no está especificado, y el número retornado por time puede ser usado sólo como argumento de las funciones date y difftime.

8.2 Librerías de Corona SDK

Corona SDK posee su propio conjunto de bibliotecas sobre las bibliotecas estándar de Lua. Algunas bibliotecas están incorporadas internamente, mientras que otras deben ser cargadas explícitamente.

Las siguientes son las bibliotecas centrales de Corona y se cargan automáticamente cuando se inicia la aplicación:

- display - proporciona todas las rutinas para la creación de objetos de visualización.
- transition - funciones para la animación de objetos de visualización, lo que simplifica el proceso de creación de movimientos básicos.
- timer - ofrece funciones básicas de tiempo.
- media - permite el acceso a las capacidades multimedia del dispositivo.
- native - proporciona acceso a los elementos de la interfaz nativa de los dispositivos.
- system - es un conjunto de funciones de sistema.

8.2.1 Biblioteca display

La biblioteca ***display*** contiene todas las funciones relacionadas con la creación de objetos de visualización.

Esta biblioteca está organizada en 3 grupos de funciones que permiten crear objetos en pantalla, la modificación de las propiedades de la pantalla en sí, y otras funciones funciones de utilidad.

Crear objetos de visualización
Todo lo que se dibuja en la pantalla es un *DisplayObject*. Todos tienen propiedades y métodos comunes.

display.newGroup()
Crea un grupo en el que se pueden anadir y borrar objetos hijos. Devuelve un objeto grupo (GroupObject) que lo representa.

display.newImage(filename [, baseDirectory] [, left, top])
Devuelve un objeto con la imagen cargada desde el fichero especificado en *filename*.

display.newImageRect([parentGroup,] filename [, baseDirectory] width, height)
Devuelve un objeto con la imagen cargada desde el fichero especificado en filename utilizando aquella cuya resolución está de acuerdo a la escala de contenido actual, determinada por Corona, que es la relación entre la pantalla actual y las dimensiones de contenido de la base se define en config.lua. En base a esta escala, Corona utiliza la tabla imageSuffix (también definida en config.lua), que enumera los sufijos de la misma familia de las imágenes, para encontrar la mejor combinación de las opciones de imagen disponibles.

display.loadRemoteImage(url, method, listener [, params], destFilename [, baseDir] [, x, y])
Este método devuelve un objeto con la imagen que se obtiene de forma remota de acuerdo a los parámetros.

display.newCircle(xCenter, yCenter, radius)
Crea un círculo con radio *radius* centrado en (xCenter, yCenter). Devuelve un objeto vector (vectorObject) que lo representa.

display.newRect(left, top, width, height)
Crea un rectángulo con las dimensiones especificadas (width, height) con la esquina superior situada en las coordenadas (left, top) . Devuelve un objeto vector (vectorObject) que lo representa.

display.newRoundedRect(left, top, width, height, cornerRadius)
Es similar al caso anterior pero devuelve el rectangulo con las esqueinas redondeadas de acuerdo al valor *cornerRadius*.

display.newLine([parent,] x1,y1, x2,y2)
Dibuja una linea desde un punto a otro. Opcionalmente se pueden añadir nuevos puntos al final de la línea (metodo object:append)

display.newText(string, x, y, font, size)
Crea un objeto de texto con la cadena *string* con su esquina superior izquierda en (x, y). Devuelve un TextObject que lo representa.
Se debe especificar el nombre de la fuente y el tamaño.

Propiedades de la pantalla

Propiedad	**Descripción**
display.contentCenterX	Equivalente a display.contentWidth/2
display.contentCenterY	Equivalente a display.contentHeight/2
display.contentHeight	Altura original del contenido en pixels. Este valor por defecto coincide con la altura de la pantalla, pero puede ser otro valor si se utiliza la escala de contenido en config.lua.
display.contentWidth	Ancho original del contenido en pixels. Este valor por defecto coincide con el ancho de la pantalla, pero puede ser otro valor si se utiliza la escala de contenido en config.lua.
display.viewableContentWidth	Una propiedad de sólo lectura que contiene la anchura en píxeles de la zona visible de la pantalla, en el sistema de coordenadas del contenido original. Esto es útil ya que dependiendo del modo de adaptación dinámica de la escala que se utiliza, y la relación de aspecto del dispositivo que se utiliza, algunos de los contenidos originales se pueden escalar de manera que las porciones queden fuera de la pantalla.
display.viewableContentHeight	Una propiedad de sólo lectura que contiene la altura en píxeles de la zona visible de la pantalla, en el sistema de coordenadas del contenido original.
display.contentScaleX	Es la proporción entre la anchura del contenido y de la pantalla en pixels. Este valor por defecto es 1, pero

	puede ser otro valor si se utiliza la escala de contenido en config.lua.
display.contentScaleY	Es la proporción entre la altura del contenido y de la pantalla en pixels. Este valor por defecto es 1, pero puede ser otro valor si se utiliza la escala de contenido en config.lua.
display.screenOriginX	Devuelve la distancia x desde la izquierda de la pantalla de referencia a la izquierda de la pantalla actual, en unidades de pantalla de referencia. En los modos "letterbox" o "zoomEven" la escala hace que se puedan modificar la superficie visible en la pantalla del dispositivo actual. Estos métodos permiten averiguar la cantidad de área visible se ha añadido o eliminado en el dispositivo actual.
display.screenOriginY	Devuelve la distancia y desde la parte superior de la pantalla de referencia a la parte superior de la pantalla actual, en unidades de pantalla de referencia.
display.statusBarHeight	Altura de la barra de estado.

Otras funciones

display.save(displayObject, filename [, baseDirectory])
Hace que el objeto de visualización referencia DisplayObject se guarde en una imagen JPEG y lo guarda en el nombre de archivo *filename*.

display.captureScreen(saveToAlbum)
Captura el contenido de la pantalla y lo devuelve como un objeto de imagen con origen en la parte superior izquierda de la pantalla.

display.setStatusBar(mode)
cambia la apariencia de la barra de esado del iPhone y iPod Touch. El valor *mode* puede ser:

- display.HiddenStatusBar
- display.DefaultStatusBar
- display.TranslucentStatusBar
- display.DarkStatusBar

8.2.2 Biblioteca transition

La biblioteca de transiciones permite animar un objeto de visualización por interpolación de una o más propiedades durante un tiempo determinado.

transition.to(target, params)
Devuelve una transición que anima las propiedades de un objeto de visualización durante un tiempo. El valor de las propiedades se especifica en la tabla de parámetros *params*. Para personalizar la transición se pueden especificar las siguientes propiedades:

- params.time – duración de la transición expresada en milisegundos. Por defecto la duración es de 500 ms (0.5 segundos).
- params.transition – define la transición. Utiliza la biblioteca easing . Por defecto es easing.linear . Otras opciones son:
  - easing.inQuad
  - easing.outQuad
  - easing.inOutQuad
  - easing.inExpo
  - easing.outExpo
  - easing.inOutExpo

- params.delay – especifica el retraso desde el comienzo de la transición.
- params.onStart - es una función o listener que se invoca al comenzar la transición.
- params.onComplete - es una función o listener que se invoca al finalizar la transición.

transition.cancel(tween)
Cancela la transición *tween*

Biblioteca timer
Esta biblioteca ofrece las funciones básicas para ejecutar acciones con cierto tiempo de retraso.

timer.performWithDelay(delay, listener [, iterations])
Invoca a la function listener despues del tiempo especificado en *delay* (milisegundos). Existe un parámetro opcional que indica el número de iteraciones que se invocará la función listener. Por defecto es 1 y el valor 0 representa un número infinito de invocaciones por lo que esta función debe tener la condición de cancelación.

timer.cancel(timerId)
Cancela el temporizador asociado con timerId.

Ejemplo:

```
local t = {}
function t:timer( event )
      local count = event.count
      print( "Table listener called " .. count .. " time(s)" )
      if count >= 3 then
       timer.cancel( self.source ) -- after 3rd invocation, cancel timer
      end
end

-- Invoca metodo timer de t un número infinito de veces
timer.performWithDelay( 1000, t, 0 )
```

8.2.3 Biblioteca media

La biblioteca *media* proporciona acceso a las funciones multimedia del dispositivo

Audio

Para reproducir sonidos cortos (1 a 3 segundos)

```
media.newEventSound( soundFile )
media.playEventSound( sound )
```

Para sonidos más largos

```
media.playSound( soundFile )
media.pauseSound()
media.stopSound()
```

Vídeo

Para reproducir un vídeo:

```
media.playVideo( path [, baseSource ], showControls, listener )
```

Cámara y biblioteca de fotos

Muestra las funciones de cámara o las bibliotecas de fotos del dispositivo.

```
media.show( imageSource, listener )
```

imageSource puede ser uno de estos valores

- media.PhotoLibrary
- media.Camera
- media.SavedPhotosAlbum

8.2.4 Biblioteca native

La biblioteca native proporciona acceso a varias funciones de la interfaz nativa de usuario en el dispositivo.

native.setActivityIndicator(visible)

Muestra u oculta el indicador de actividad de la plataforma.

native.showAlert(title, message [, buttonLabels [, listener]])

Muestra un cuadro de alerta emergente con uno o más botones, utilizando un control nativo de alerta.

native.cancelAlert(alert)

Cierra el cuadro de alerta mediante programación.

native.newFont(name [, size])

Crea un objeto de fuente que se puede utilizar para especificar la fuente de texto nativo en campos y marcos de texto.

native.systemFont y native.systemFontBold representan las fuentes del sistema (normal y negrita).

native.getFontNames() – devuelve un array con los nombres de todas las fuentes disponibles en el sistema.

native.newTextField(left, top, width, height [, listener])
Crea un campo de entrada de texto de una línea.

native.setKeyboardFocus(textField)
Pone el foco sobre un campo de texto y muestra el teclado del dispositivo. Pasando el valor *false*, se oculta el teclado.

native.newTextBox(left, top, width, height)
Crea un cuadro de texto de varias líneas con desplazamiento.

native.showWebPopup(url [, options])
Crea una ventana emergente con una pantalla completa para la carga local o remota de páginas web.

native.cancelWebPopup()
Oculta la ventana emergente para paginas web.

native.showPopup(name, options)
Muestra las aplicaciones nativas de envio de e-mail o sms.
name: “sms” o “mail”
options: tabla con las opciones de configuracion

8.2.5 Biblioteca system

system.getInfo(param)
Devuelve información sobre el sistema en el que se ejecuta la aplicación. El argumento es una cadena que determina la información que se devuelve. Estos son los valores válidos para el parámetro.

Parámetro	**Información**
name	Nombre legible del dispositivo
model	Modelo del dispositivo: “Iphone”, “Ipad”….
deviceID	devuelve el identificador único del dispositivo (IMEI)
environment	Entorno en el que está corriendo la aplicación: "simulator" - Simulador de Corona "device" - dispositivo
platformName	Nombre de la plataforma o SO: "Mac OS X" "iPhone OS" "Android"
platformVersion	Versión de la plataforma o SO
version	Versión del corona SDK
textureMemoryUsed	Uso de memoria en bytes
maxTextureSize	Tamaño máximo de memoria del dispositivo

system.getPreference(category, name)
Devuelve el valor una preferencia de sistema como una cadena. Las categorías son:

- Categoría "ui" – preferencias de las aplicaciones generales. En esta categoría, con el nombre "language" se puede obtener el idioma preferido en el dispositivo.
- Categoría " locale" – preferncias locales. Podemos obtener los nombres "country", "identifier" y "language".

system.getTimer()
Devuelve el tiempo en milisegundos desde que se lanzo la apliacción.

system.pathForFile(filename [, baseDirectory])
Genera una ruta absoluta usando como base el directorio de la aplicación. Un segundo parámetro opcional especifica que directorio utilizar para construir la ruta completa, y su valor por defecto es system.ResourceDirectory. Si el directorio base es system.ResourceDirectory y ruta generada apunta a un archivo que no existe, se devuelve nil.

Los directorios que pueden ser usados como base se definen en las siguientes constantes:

- system.ResourceDirectory – es el directorio donde están las herramientas de la aplicación. En el simulador se corresponde con la carpeta de proyecto.

- system.DocumentsDirectory – debe ser usado para ficheros que precisan persistencia entre varias sesiones de la aplicación.

- system.TemporaryDirectory – es un directorio temporal. Los ficheros creados en este directorio no tienen asegurada su permanencia entre sesiones.

8.2.6 Biblioteca widget

La biblioteca widget proporciona herramientas útiles para dar a nuestra aplicación un aspecto mas dinamico. Dentro de la biblioteca podemos diferencias varios objetos, a continuación los enumeramos:

widget.setTheme(themeFilename)

Nos permite dar un aspecto IOS a la aplicación. El tema grafico que tiene un dispositivo con IOS no se puede modifcar, con esta opción la aplicación parece mas integrada en el sistema operativo móvil.

widget.newButton([options])

Para agregar botones animados con aspecto IOS. Es similar a la librería *ui*.

Figura 8.1. Boton de la biblioteca *widget*

widget.newSlider([options])

Introduce un objeto slider deslizable con el dedo. Suele usarse para ampliar/reducir objetos, sonido…

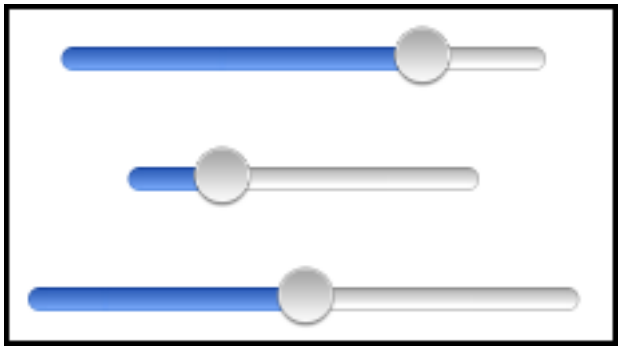

Figura 8.2. Slider de la biblioteca *widget*

widget.newScrollView([options])

Se crea un objeto mas grande que la pantalla y es necesario deslizarlo. Este widget nos permite hacer esa función de forma fácil y sencilla.

Figura 8.3. ScrollView de la biblioteca *widget*

widget.newTableView([options])

Se crea una tabla en forma de lista. Puede ocupar bastante espacio y nos permite bajar o subir entre sus líneas asi como eliminarlas.

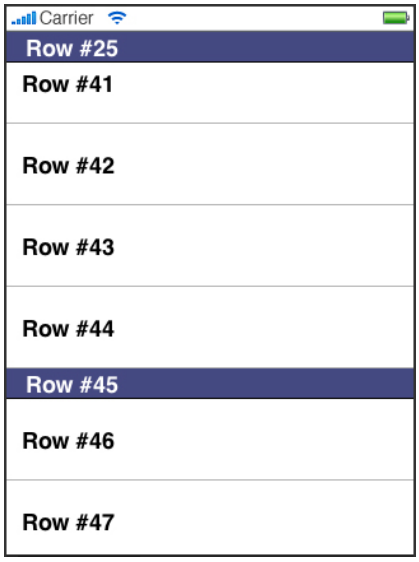

Figura 8.4. TableView de la biblioteca *widget*

widget.newTabBar([options])

Si queremos que nuestra aplicación tenga pestañas esta es una opción muy útil. Nos permite cambiar entre ellas con solo pulsar el botón correspondiente.

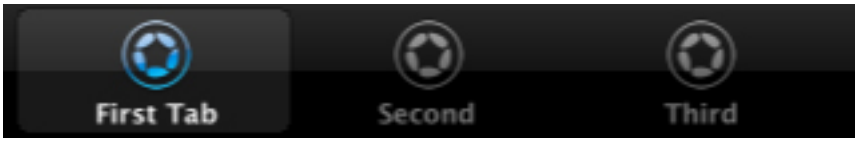

Figura 8.5. TabBar de la biblioteca *widget*

widget.newPickerWheel([options])

Crea un objeto que puede servirnos para seleccionar una fecha. Deslizando entre sus cifras seleccionamos el dia , mes y año que estamos. Puede adaptarse para otros menesteres como por ejemplo seleccionar peso, altura , edad. Es muy configurable.

Figura 8.6. Picker Wheel de la biblioteca *widget*

8.2.7 Bibioteca StoryBoard

Es la nueva librería nativa que Corona SDK ha implementado recientemente. Ha incluido esta aplicación tras la versión no oficial de gestión de escenas, la *directorClass* que lleva utilizándose prácticamente desde que apareció Corona SDK. A continuación se explica en detalle su funcionamiento.

Lo primero es añadir la librería a nuestra aplicación:

```
local storyboard = require "storyboard"
```

El siguiente paso es cargar la primera escena de la aplicación:

```
storyboard.gotoScene(escena,efecto,tiempo)
```

La estructura que presenta una escena es la siguiente:

```
local storyboard = require( "storyboard" )
local scene = storyboard.newScene()

function scene:createScene( event )
	local group = self.view

end

function scene:enterScene( event )
	local group = self.vie

end

function scene:exitScene( event )
	local group = self.view

end

function scene:destroyScene( event )
	local group = self.view

end

scene:addEventListener( "createScene", scene )
scene:addEventListener( "enterScene", scene )
scene:addEventListener( "exitScene", scene )
scene:addEventListener( "destroyScene", scene )

return scene
```

Las escenas son archivos que están dentro del directorio principal de la aplicación y cuya extensión es la misma que el archivo *main.lua*.

Como vemos la estructura de las escenas esta definida. En la función *createScene* se introduce lo referido a la creación de la escena.

En la función *enterScene* colocamos el código que queremos que se ejecute al entrar en la escena.

En la función *exitScene* es necesario eliminar todos los objetos creados para las escena asi como los listeners.

Para finalizar la función *detroyScene* nos permite ejecutar código cuando la escena es eliminada.

ÍNDICE DE FIGURAS