UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València


QT MOBILE: COMPILACIÓN CRUZADA PARA DISTINTOS SISTEMAS OPERATIVOS MÓVILES

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Andrés Piccininno Zugasti

Tutor: Antonio Cano Gómez

2013-2014

Resumen

En este proyecto se ha llevado a cabo el diseño y la implementación del tradicional juego de chapas de fútbol para dispositivos móviles. Es un proyecto orientado a más de un sistema operativo, a múltiples plataformas, y es que es ahí donde reside la dificultad e innovación que este supone.

Se ha desarrollado un juego que permite a dos usuarios, a través de un único dispositivo móvil, pasar un buen rato y recordar este antiguo pero entretenido juego.

Palabras clave: Multiplataforma, sistema operativo, compilación cruzada, aplicación móvil, *Qt*.

Tabla de contenidos

Índice de ilustraciones

1. INTRODUCCIÓN

1.1 Juegos en dispositivos móviles

Es difícil conocer a alguien en estos días que no haya jugado al conocido juego *Candy crush*. Es incluso igual de difícil ir por la calle, el metro o estar en casa y no ver a alguien que esté juntando esas famosas frutas en su dispositivo móvil. Como ocurre con *Candy crush*, ocurre también con cientos de juegos para móviles hoy en día, y es que se estima que cada año el número de consumidores de juegos móviles aumenta en un 130 por ciento.

Siendo más precisos, el consumo de juegos móviles está doblando cada año el dinero que los usuarios gastan, por ejemplo, en música digital. De entre los países más consumidores de contenido digital se encuentran los Estados Unidos, los grandes mercados de Europa como Alemania o Francia, Japón y Corea del Sur.

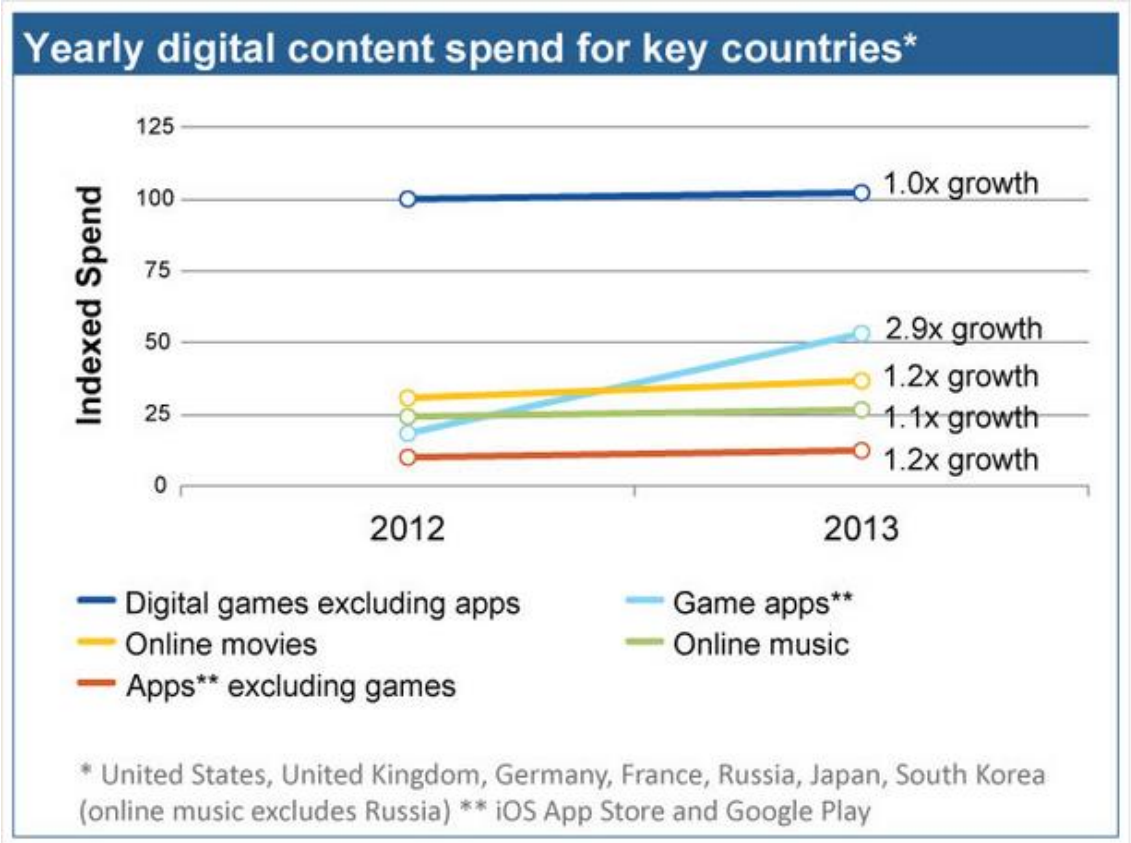


Ilustración 1 – Las aplicaciones móviles lideraron el crecimiento de contenido digital en 2013, donde los juegos destacan con una tasa de crecimiento de más de 2.9 cada año.

Para 2017, se estima que la industria de los juegos móviles podría dirigir a la industria **entera** de los videojuegos hacia unos ingresos de más de cien billones de dólares. La industria de los juegos móviles, pues, está sobrepasando los límites de la industria de los videojuegos tradicionales. En este sentido, en la industria de los juegos móviles los ingresos provienen de diferentes fuentes: la propia compra de dichos juegos, cuyos precios pueden variar desde 0.99 euros o menos hasta precios de doscientos euros como ocurre con el juego *Deduce – Grandmaster edition*, o incluso por la compra de contenido extra y adicional que ofrecen los juegos como puedan ser nuevas vidas, nuevas armas para un juego de guerra, etc.

1.2 Objetivos y motivación

El objetivo fundamental que se persigue al elaborar este proyecto, es el de poder desarrollar y ejecutar un juego en múltiples plataformas, en **diferentes sistemas operativos**. En un principio, el proyecto estará más centrado y orientado a sistemas operativos móviles, pero también se tendrá en cuenta la posibilidad de llevar el proyecto hacia sistemas operativos de escritorio. Es ahí, en la compilación cruzada, donde reside la dificultad y lo llamativo de este proyecto.

De una manera más específica, se han perseguido también, en orden de importancia, los siguientes objetivos:

- Poder portar el juego a diferentes plataformas móviles como *Android* o *Sailfish* gracias a la compilación cruzada. Intentar también desplegar el juego en sistemas operativos *iOS*, analizando la viabilidad de esto.

- Intentar llevar el proyecto a sistemas operativos de escritorio como *Windows*, *Mac* o *Linux*.

- Definir y describir el concepto de compilación cruzada y las dificultades que este conlleva.

- Dotar al juego de una interfaz amigable y atractiva para el usuario, la cual debe permitir acceder rápidamente a la pantalla de juego.

- Facilitar el uso del juego. La lógica del juego que se desarrolla debe resultar intuitiva y fácil de aprender para el usuario.

- Aprender a usar la tecnología *Qt* así como su conocido metalenguaje *Qml*, las cuales eran nuevas tecnologías para nosotros.

- Mejorar y evolucionar en el uso de la tecnología *JavaScript*, la cual sí era conocida por nuestra parte pero que no habíamos experimentado con tanta profundidad en situaciones anteriores a este proyecto.

1.3 Metodología del proyecto

Para completar la realización de este proyecto hemos ido avanzando a lo largo de cuatro fases: análisis, diseño, implementación y pruebas. Cada una de estas fases se describe con más profundidad en sus respectivos apartados de este documento.

- Análisis: esta fase consistió, fundamentalmente, en una fase de **planificación**. En primer lugar, nos dedicamos al principio del proyecto a analizar juegos similares que también habían sido desarrollados gracias a la tecnología *Qt*. Con el análisis de dichas aplicaciones, y gracias a la documentación que hay en la *web* oficial de *Qt* (http://*Qt*-project.org/doc/) comenzamos a aprender sobre el uso del lenguaje *Qml*. Además, en esta fase también se analizaron detenidamente diferentes aspectos del juego que se iba a desarrollar.

- Diseño: en esta etapa nos dedicamos a hacer bocetos intentando perfilar cómo sería la interfaz del juego que se iba a desarrollar. Para esto, se analizó qué componentes de *Qml* podríamos necesitar, cómo sería más o menos el aspecto de los diálogos, etc.

- Implementación: una vez seleccionados los componentes *Qml* que conformarían la interfaz, procedimos a comenzar con el desarrollo de la propia aplicación. Siempre contábamos con la flexibilidad de poder volver a la fase de diseño en el caso de que durante la fase de implementación no se pudiera avanzar en algún punto. Así, en esta fase se lleva a cabo la implementación tanto de la interfaz del juego como de la lógica, intentando dotar al juego de actividad.

- Pruebas: esta fase iba muy ligada tanto a la fase de diseño como a la de implementación. Cada vez que hacíamos cambios importantes en el código del juego, iniciábamos pruebas de ejecución, intentando verificar que lo nuevo que se había introducido no era erróneo. Esta fase, al igual que las otras, se comenta más detenidamente en su respectivo apartado.

2. HERRAMIENTAS Y TECNOLOGÍA

2.1 Qt

La tecnología *Qt* consiste en un *framework* multiplataforma ampliamente usado en el desarrollo de *software*. El término multiplataforma (*cross-platform*), hace referencia a la habilidad que posee un elemento *software* para poder ser ejecutado en **diferentes** arquitecturas de computadores o sistemas operativos, como ocurre en el caso de este proyecto.

Qt es actualmente desarrollado por *Digia*, la cual es a su vez la empresa propietaria de la marca *Qt*. Sin embargo, si echamos la vista atrás, *Qt* ha estado en manos de diferentes propietarios. En primer lugar, hay que decir que *Qt* fue creada por la compañía *Trolltech* en 1991. Esto seguiría así hasta 2008, cuando *Nokia* se hizo propietaria de *Trolltech* ASA, cambiando el nombre primero a *Qt Software* y, finalmente, a *Qt Development Frameworks*. Ya para Marzo de 2011, *Nokia* anunciaba la venta de la licencia comercial de la marca *Qt* y de sus servicios profesionales a *Digia*, la cual, como dijimos en líneas anteriores, es la actual propietaria de la tecnología.

Tras la adquisición de la marca, *Digia* se puso los objetivos **inmediatos** de que *Qt* debía dar soporte a las plataformas *Android*, *iOS* y *Windows* 8. No se iban a abandonar, además, otras plataformas como *Sailfish*, *VxWorks* o *Blackberry*.

Qt hace uso del estándar C++, pero también puede ser usado en otros lenguajes de programación haciendo uso del API que sea necesario según las circunstancias. También es destacable el metalenguaje *Qml*, o lenguaje de marcado (*markup language*) empleado para el diseño de interfaces de usuario sobre el que daremos más detalles en próximos apartados.

2.1.1 Versiones

Han sido numerosas las versiones que ha habido de *Qt* desde el momento de su creación. Las dos primeras versiones eran las conocidas como *Qt*/X11, la cual estaba dedicada a sistemas *Unix*, y la *Qt*/*Windows* dedicada, como su propio nombre indica, a sistemas operativos *Windows*. Ya para finales del año 2001, la primera empresa propietaria de la marca *Qt* que nombramos anteriormente, *Trolltech*, lanzaría la versión *Qt* 3.0, que añadió soporte para sistemas operativos *Mac OS X*. Para Junio de 2005, *Trolltech* era también la encargada de lanzar la versión *Qt* 4.0.

Esta versión sería destacable gracias a las diferentes tecnologías que introdujo en el *framework*, como lo eran *Tulip*, *Interview*, *Arthur* o *MainWindow*. Pero sin duda, la mayor novedad que acompañó esta nueva versión fue la presencia de la herramienta conocida como *Qt designer*, la cual permitía llevar a cabo el diseño de interfaces de usuario a través de acciones *drag & drop* (arrastrando y soltando).

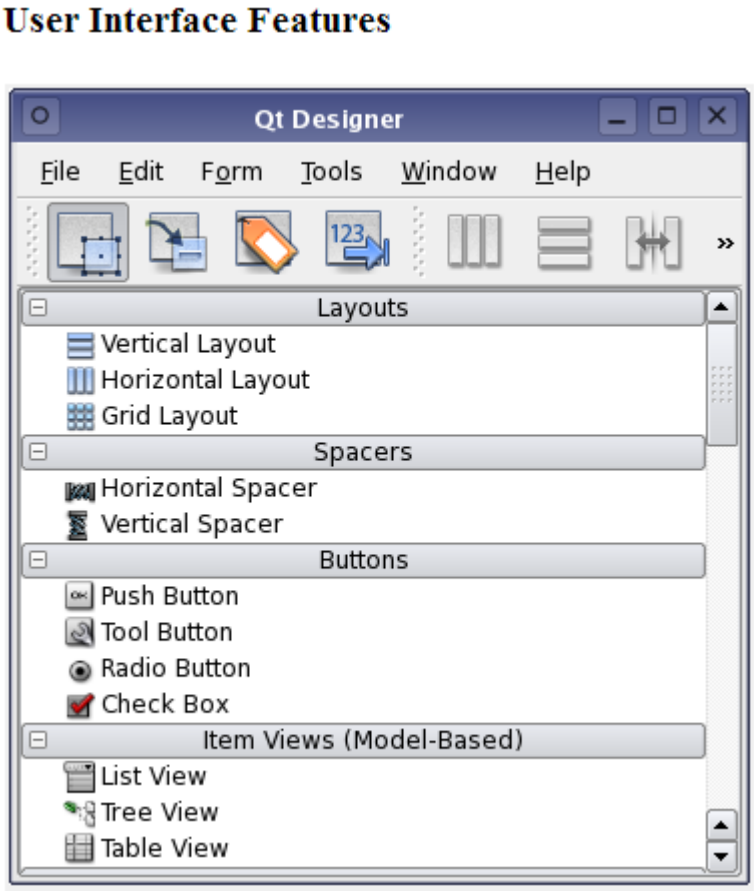


Ilustración 2 – Aspecto de la herramienta Qt Designer para el diseño de UI.

Sería en el 19 de Diciembre de 2005 cuando la versión 5 de *Qt* era lanzada. Esta nueva versión introdujo grandes cambios en la plataforma, donde los lenguajes *QML* y *JavaScript*, los cuales hemos empleado para el desarrollo de este proyecto, pasaron a tener un mayor peso. Además, con *Qt* 5 se aportaron significantes mejoras en la velocidad y facilidad de desarrollo de las interfaces de usuario. Desde el lanzamiento de esta versión, han sido cuatro las versiones que se han ido incorporando. Dentro de éstas, era en la versión *Qt* 5.2 cuando se confirmaba un soporte oficial a las tecnologías *Android* e *iOS*. Esto hacía dar un paso de gigante a la marca *Qt*, ya que de esta manera se podía desarrollar una aplicación que podía ser ejecutada en **múltiples** plataformas:

- Escritorio: *Windows*, Mac OS X, *Linux*, Solaris
- Sistemas empotrados: *Linux* (DirectFB, KMS, etc.), *Windows* (*Compact* y *Standard*), *Android*, etc.
- Móviles: *Android*, *iOS*, BlackBerry, *Sailfish*, etc.

Además, la versión 5.2 de *Qt* venía acompañada de una nueva versión de la herramienta *Qt creator*, la 3.0, la cual comentaremos más adelante.

La última versión conocida de la marca ha sido la 5.3, la cual fue lanzada el 20 de Mayo de 2014. Según los propios directores de esta tecnología multiplataforma, se trata de una versión con la que se quiere mejorar el rendimiento, la estabilidad y la usabilidad de las versiones previas.

2.1.2 *Arquitectura software*

Desde su primer lanzamiento, *Qt* se fundamentó en una serie de conceptos:

- Abstracción completa de la interfaz gráfica de usuario. Desde su primera versión, *Qt* usaba su propio motor de dibujo y sus controles intentando emular el aspecto que tendría la interfaz y sus *widgets* en las diferentes plataformas en las que se usuaria el *software* que se estaba creando. Eran pocas las clases de *Qt* que dependían de la plataforma destino.
- Comunicación entre objetos: los diferentes *widgets* de la interfaz que se diseñan con *Qt* podían enviar señales (*signals*) que contenían datos sobre el evento producido, los cuales podían ser recibidos por otros controles de la interfaz haciendo uso de unas funciones especiales conocidas como *slots*.
- Compilador de metaobjetos (*moc*): consiste en una herramienta que podía interpretar partes del código C++ como anotaciones, para así usarlas y generar código C++ adicional con metainformación sobre las clases usadas en la aplicación, que era, a su vez, usada por *Qt* para proveer de funcionalidades no disponibles en el código nativo de C++.

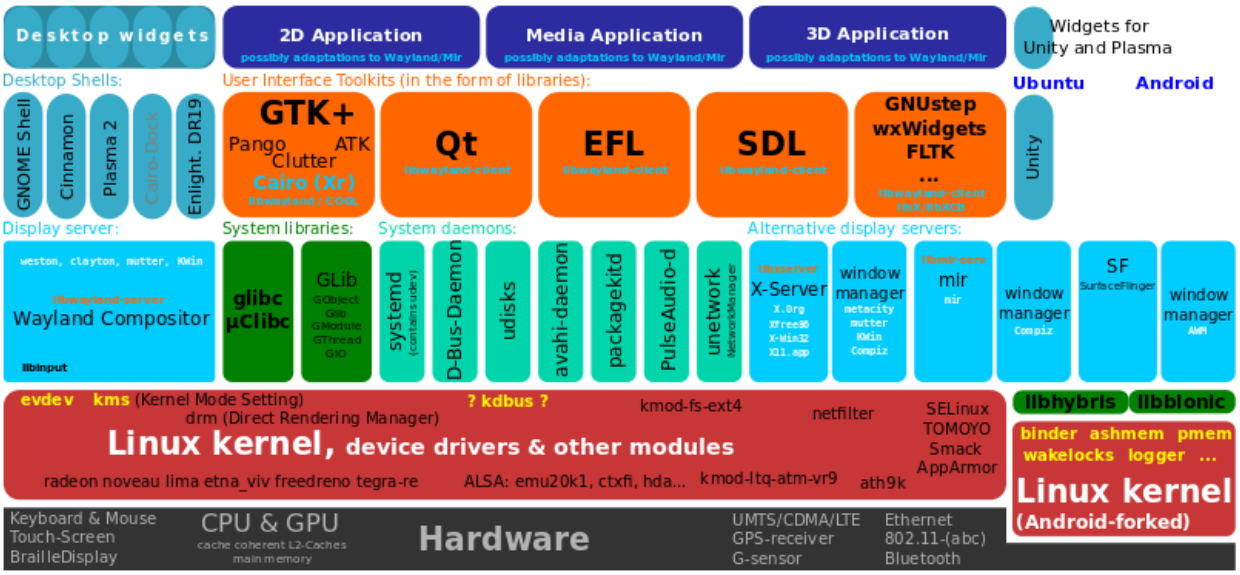


Ilustración 3 – Uso de Qt en un sistema de arquitectura Linux

2.1.3 Impacto en el mercado

La tecnología *Qt* es conocida internacionalmente. Es usada por miles de compañías líderes en todo el mundo, estando presente en alrededor de más de 70 industrias. Esta marca está presente actualmente en millones de dispositivos como puedan ser, por ejemplo, los móviles, y en aplicaciones con las que interactuamos en nuestra vida diaria.

Muchos productos de las compañías más innovadoras y con visión de futuro del mundo, han seleccionado a *Qt* como la plataforma para el desarrollo de sus productos con una garantía de futuro. Tales compañías pueden ser *Jolla*, *Michellin*, *ABB*, *BlackBerry*, *Magneti Marelli*, *Panasonic*.

Según el que es en estos momentos cofundador de la compañía *Qt*, Dynami Caliri: "*Sin Qt, estaríamos estancados en el pasado sin posibilidad de avanzar. Qt nos ha otorgado una solución eficiente para desarrollar, mantener y mejorar nuestras aplicaciones del futuro*".

2.1.4 Qt Creator

Es necesario hablar de *Qt Creator*. Es necesario hablar de esta herramienta ya que ha sido la que hemos utilizado a lo largo del desarrollo del proyecto. *Qt Creator* consiste en un entorno IDE para el desarrollo de *software*. Concretamente, es un entorno **multiplataforma** de desarrollo C++, *Qml* y *JavaScript*. Esta herramienta es **parte** del SDK que viene incluido en el *framework* de *Qt*.

Qt Creator incluye un gestor de proyectos que hace uso de un formato de proyectos multiplataforma, el formato ".pro". Este tipo de ficheros incluye información sobre qué archivos están incluidos en el proyecto, ajustes particulares para que las aplicaciones puedan ser ejecutadas, etc. Se detallará más sobre este tipo de archivos en el apartado de "Implementación", donde hablaremos, entre otras cosas, del fichero ".pro" de nuestro proyecto.

Este IDE también incorpora un avanzado editor de código, intentando aumentar la productividad del programador:

- Editor de código C++, *Qml*, y *scripts*.
- Herramientas para una rápida navegación de código.
- Subrayado de sintaxis y autocompletado de código (*completion*).
- Chequeo estático de código, así como consejos de estilo mientras el programador escribe.
- Comprobación de paréntesis.
- Soporte para aplicar formato rápido al código.

También hay que destacar el hecho de que *Qt Creator* está integrado con otra serie de herramientas orientadas a los sistemas de control de versiones como *Git*, *Subversion* o *CVS*.

Qt Creator no incluye un depurador para la ejecución de aplicaciones (*debugger*). Lo que sí se puede hacer es, a través de un *plugin* que actúa como una interfaz entre el núcleo de *Qt Creator* y depuradores nativos externos, hacer depuración (*debugging*) del lenguaje C++.

2.1.5 QML

Qml consiste en un lenguaje de programación de marcado (*markup language*), basado en *JavaScript*, con una sintaxis de aspecto muy similar a la de *JSON*, que se emplea para el **diseño** de interfaces de usuario. Forma parte del *framework Qt* del que hemos estado hablando y se emplea fundamentalmente para el diseño de aplicaciones móviles. Este lenguaje permite, principalmente, describir interfaces de usuario en términos de qué componentes las forman y cómo estos interactúan entre sí para llevar a cabo las acciones pertinentes.

Para hacer uso de este lenguaje, los programadores deben emplear el módulo *Qt Quick* de *Qt*. Este módulo consiste en una librería estándar de **tipos** y de **funcionalidad** para *Qml*. Para el acceso a toda la funcionalidad, el desarrollador necesita hacer uso de una simple sentencia *import*: "import QtQuick". Así pues, es a través de los tipos con lo que se definen los objetos o componentes que forman la interfaz del usuario. Los objetos siempre se declaran con su primera letra en mayúscula, al contrario de los datos o componentes básicos como puedan ser "date", "point", "size" o "rect". A continuación podemos ver la declaración de dos objetos, uno de tipo "Rectangle" y otro, su hijo, de tipo "Image".

```
import QtQuick 1.0

 Rectangle {
     id: canvas
     width: 200
     height: 200
     color: "blue"

     Image {
         id: logo
         source: "pics/logo.png"
         anchors.centerIn: parent
         x: canvas.height / 5
     }
 }
```

Ilustración 4 – Ejemplo de declaración de objetos en QML

Como se puede ver en el ejemplo de arriba, dentro de los objetos se pueden especificar lo que se llaman **propiedades** del objeto. Las propiedades permiten determinar cómo será el aspecto de un objeto en la interfaz que se está definiendo. La sintaxis de las propiedades es simple: "propiedad: valor". Dentro de las propiedades, la que se puede considerar más importante y que resulta de gran utilidad para cualquier desarrollador es la propiedad de "id". Cada objeto puede tener asignada una propiedad **única** que se denomina "id", por medio de la cual el objeto puede ser referenciado por otros objetos o por otros scripts. Según el objeto del que estemos hablando, las propiedades asignables a dicho objeto pueden ser bien diferentes. Así, por ejemplo, para un objeto de tipo "Rectangle" hay propiedades del tipo "border", "color", "radius"… además de aquellas propiedades que **hereda** de otros objetos como "Item". Así es, en *Qml* existe la herencia entre objetos, por medio de la cual un objeto posee no sólo los atributos (propiedades) propios de dicho objeto, sino también los del objeto del que hereda.

No vamos a comentar ni a entrar en mayor detalle en cada uno de los objetos o componentes de los que disponemos con *Qml*, pero si pensamos que es importante profundizar un poco en aquellos objetos que sí hemos usado en nuestro proyecto y que nos han permitido definir la interfaz de nuestro juego, así que estos componentes los comentamos a continuación.

- **Rectangle**: permite dibujar una figura rectangular pudiendo añadirle un borde si así se desea.

```
import QtQuick 2.0

Rectangle {
    width: 100
    height: 100
    color: "red"
    border.color: "black"
    border.width: 5
    radius: 10
}
```

Ilustración 5 – Ejemplo del objeto "Rectangle" en QML

- **Row**: posiciona a sus hijos en una única fila.

```
import QtQuick 2.0

Row {
    spacing: 2
    Rectangle { color: "red"; width: 50; height: 50 }
    Rectangle { color: "green"; width: 20; height: 50 }
    Rectangle { color: "blue"; width: 50; height: 20 }
}
```

Ilustración 6 – Ejemplo del objeto "Row" en QML

- **Item**: es un tipo básico visual de *Qml*. Es heredado por un gran número de objetos que hemos usado durante el proyecto y que iremos comentando como lo son "Rectangle", "Row", "MouseArea", "Image", "Text", etc.

- **Image**: permite visualizar una imagen en la interfaz. La imagen que se mostrará se puede indicar a través de la propiedad "source".

```
import QtQuick 2.0

Image {
    source: "pics/qtlogo.png"
}
```

Ilustración 7 – Ejemplo del objeto "Image" en QML

- **SoundEffect**: es el elemento que permite reproducir sonidos en una aplicación, concretamente archivos de formato ".wav". Al igual que ocurría con "Image", se puede indicar la fuente de sonido a través de la propiedad "source".

- **Timer**: este objeto lanza un capturador de eventos siguiendo un intervalo especificado en milisegundos. Este objeto tendrá un peso importante en distintas funcionalidades de nuestro proyecto.

- **Text**: se emplea para añadir texto con formato a la interfaz.

Hello World!

```
Text {
    text: "Hello World!"
    font.family: "Helvetica"
    font.pointSize: 24
    color: "red"
}
```

Ilustración 8 – Ejemplo del objeto "Text" en QML

- **MouseArea**: consiste en un ítem invisible que suele usarse de manera conjunta con un objeto que sí es visible, de manera que permite capturar eventos de *mouse* para dicho objeto. Así, un objeto de tipo "MouseArea" suele ir declarado **dentro** de otro objeto, como ocurre en el siguiente caso en el que se declara dentro de un "Rectangle".

```
import QtQuick 1.0

Rectangle {
    width: 100; height: 100
    color: "green"

    MouseArea {
        anchors.fill: parent
        onClicked: { parent.color = 'red' }
    }
}
```

Ilustración 9 – Ejemplo del objeto "MouseArea" en QML

- **State**: es un elemento muy útil cuando se quiere cambiar la interfaz según el estado en el que se encuentra la aplicación. Estos objetos se definen dentro de la propiedad conocida como "states" de un objeto, la cual mantiene una lista de estados en forma de *array*. Así pues, podremos controlar, de esta manera, los diferentes estados en los que se encuentra un determinado objeto de la interfaz.

```
states: [
    State {
        name: ""; // when: gameState.gameOver == false && passedSplash
        PropertyChanges { target: mainbar; x: 0 }
        StateChangeScript { script : {
                sound_home.play()
                Logic.stopGame()
            }
        }
    },
```

Ilustración 10 – Ejemplo del objeto "State" en QML

Como se puede ver en el ejemplo anterior, hay definidos dentro de "State" otros dos objetos: "PropertyChanges" y "StateChangeScript". Pasamos a comentarlos.

- **PropertyChanges**: se usa para definir cambios en los valores de propiedades que nosotros determinemos. De esta manera, cuando un objeto cambia de estado podemos hacer que algunas de sus propiedades cambien. Es necesario especificar una propiedad "target" que indique el ítem para el que queremos cambiar las propiedades.

- **StateChangeScript**: este objeto será ejecutado cuando un cambio de estado en un objeto tenga lugar. Nos permite, de esta manera, ejecutar un *script*, como se ve en el ejemplo anterior.

- **Transition**: es un elemento que define transiciones animadas cuando un cambio de estado para cierto objeto tiene lugar. Se incluyen dentro de una propiedad de tipo "transitions", la cual consiste en un *array* de objetos de tipo "Transition". Este tipo de objeto suele ir acompañado de animaciones como pueda ser la de "NumberAnimation" o "SequentialAnimation".

- **NumberAnimation**: esta animación se aplicará cuando un valor numérico cambia. Esto permite, por ejemplo, animar una propiedad cambiando sus valores progresivamente.

- **SequentialAnimation**: permite ejecutar múltiples animaciones **juntas**. En este caso, al tratarse de una animación secuencial, las animaciones incluidas dentro de "SequentialAnimation" se ejecutarán una después de otra. Así,

animaciones de tipo “NumberAnimation” se pueden incluir dentro de este tipo de animación.

- **MessageDialog**: se ha empleado este tipo de objeto para poder crear diálogos con pequeños mensajes que podrán aparecer en la interfaz. Se pueden considerar como mensajes de alerta que también permiten al usuario responder ante ellos y tomar la iniciativa.

```
import QtQuick 2.2
import QtQuick.Dialogs 1.1

MessageDialog {
    title: "Overwrite?"
    icon: StandardIcon.Question
    text: "file.txt already exists.  Replace?"
    detailedText: "To replace a file means that its existing contents will be lost. " +
        "The file that you are copying now will be copied over it instead."
    standardButtons: StandardButton.Yes | StandardButton.YesToAll |
        StandardButton.No | StandardButton.NoToAll | StandardButton.Abort
    Component.onCompleted: visible = true
    onYes: console.log("copied")
    onNo: console.log("didn't copy")
    onRejected: console.log("aborted")
}
```

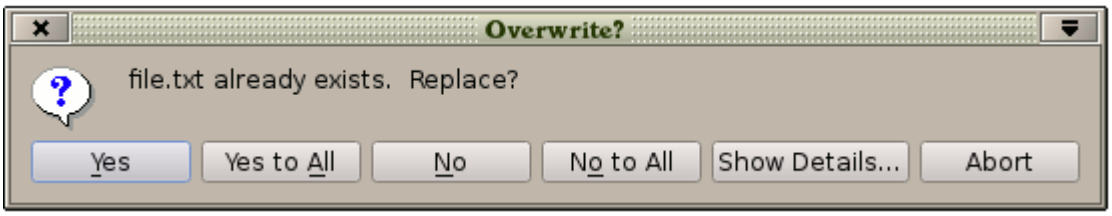


Ilustración 11 – Ejemplo del objeto “MessageDialog” en QML

2.2 JavaScript

JavaScript es un lenguaje de programación dinámico empleado ampliamente en el desarrollo de páginas *web*. Aun así, se dice de *JavaScript* que es un lenguaje no sólo para HTML o para la *web*, es, en general, un lenguaje para los ordenadores, para servidores, para *laptops*, para *tablets*, para *smartphones*, y más. En lo que respecta a la programación *web*, *JavaScript* permite hacer dinámicas a las páginas *web* que los clientes visitan, pudiendo interactuar con ellas libremente. En la siguiente gráfica se puede ver la importante presencia que JavaScript tiene en los sitios *web* que accedemos diariamente, y es que este conocido lenguaje está presente en alrededor del 90% de páginas *web*.

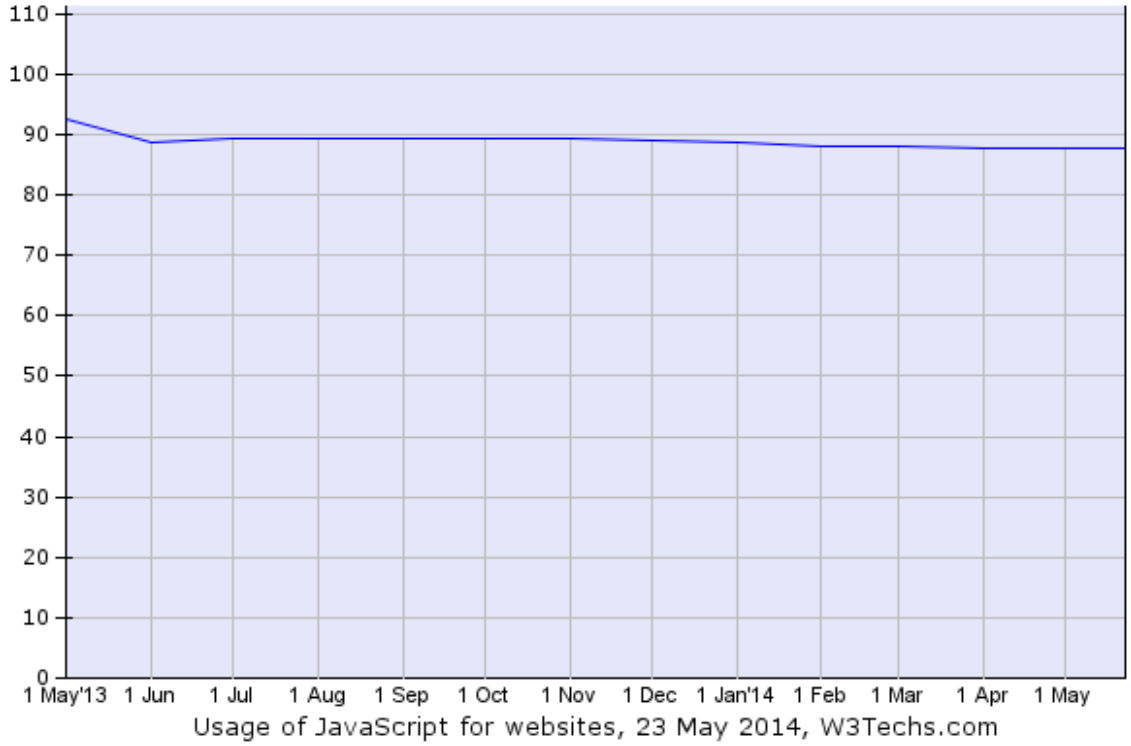


Ilustración 12 – Uso de JavaScript en la industria Web actual

La gran mayoría de sitios *web* de hoy en día usan JavaScript, y todos los navegadores *web* modernos (de escritorio, de *smartphones*, de las consolas de videojuegos, etc.) incluyen intérpretes de *JavaScript*. Se trata, en líneas generales, de un lenguaje de programación dinámico de alto nivel, ajustándose cómodamente a estilos de programación orientada a objetos y también a los de programación funcional. Su sintaxis deriva de *Java*, sus funciones de primera clase de *Scheme*, y su herencia basada en prototipos, de *Self*.

Este lenguaje de *scripting* de tipado dinámico (una variable puede tomar valores de distintos tipos) se emplea también de manera extensa en el desarrollo de **juegos** y en la creación de **aplicaciones** para escritorio y para **dispositivos móviles**. Es aquí en este apartado, en el de juegos y aplicaciones para dispositivos móviles, donde reside el interés que ha habido en nuestro proyecto para hacer uso de *JavaScript* y también es el interés que, en su momento, tuvo la marca *Qt* al apostar por este lenguaje.

De entre las diferentes características que definen la sintaxis y semántica que definen *JavaScript*, podemos destacar las siguientes:

- Variables: para asignar un valor a una variable se hace uso del operador "="

- Operadores: se usan para computar valores. Según el tipo que tengan las variables, la operación que se efectuará será una u otra. Por ejemplo, el operador

"+" puede servir para sumar dos valores, o bien para concatenar dos cadenas de caracteres.

- Sentencias: se separan por el símbolo ";".

- Uso de palabras reservadas (*keywords*): palabras como "var", que se emplean para crear una nueva variable, están reservadas. Otras palabras reservadas pueden ser "for", "while", "boolean", "function", "else", "if", "true", "false", "return", "null", "Array", "String", "length", "import", "package", "break", etc.

- Tipos de datos: las variables en *JavaScript* pueden almacenar distintos tipos de datos como números, *Strings* (cadenas de texto), *arrays*, objetos, etc. Véase el siguiente ejemplo.

```
var length = 16;
var lastName = "Johnson";
var cars = ["Saab", "Volvo", "BMW"];
var person = {firstName:John, lastName:Doe};
```

Ilustración 13 – Ejemplo de creación de variables en JavaScript

- Funciones: se declaran a través de la palabra reservada "function", pudiendo tener un número y tipo de argumentos a petición del desarrollador. Pueden ser referenciadas a través de su nombre. En nuestro proyecto, las funciones son realmente las encargadas de llevar a cabo la lógica del juego, es decir, de realizar las acciones que el usuario indica.

- Comentarios: En *JavaScript* también se pueden añadir líneas en forma de comentario gracias a los caracteres especiales de "//" o también "/* … */"

- Es *case sensitive*: *JavaScript* sí distingue entre mayúsculas y minúsculas, de manera que dos variables como puedan ser "lastName" y "lastname" son **diferentes**. Esto ocurre igual con los nombres de las funciones.

En lo que respecta a nuestro proyecto, si antes decíamos que por medio de *Qml* definíamos los objetos o componentes que describían la interfaz del juego, con *JavaScript* lo que hemos conseguido ha sido dotar de vida al juego, que el usuario pueda interactuar con esos diferentes componentes que están presentes en la interfaz. Así, para que los diferentes elementos definidos por *Qml* sean realzados y no sean simplemente algo estático, hemos incorporado la presencia de *JavaScript*, bien a través de instrucciones *inline* que se pueden encajar en el propio código *Qml* o bien a través de archivos *JavaScript* externos a los archivos *Qml*. Se puede ver, en el siguiente ejemplo, cómo se incluye en nuestro proyecto el archivo *JavaScript* necesario para poder acceder a las distintas funcionalidades que en él se hallan.

```
1  import QtQuick 2.0
2  import "content"
3  import "content/logica.js" as Logic
```

Ilustración 14 – Inclusión de archivos JavaScript. En este caso el nombre del archivo es "logica.js", el cual se encuentra en el directorio "content" de nuestro proyecto.

```
356     Timer {
357         interval: 16
358         running: true
359         repeat: true
360         onTriggered: Logic.tick()
361     }
```

Ilustración 15 – Invocación de la función "tick()" que está definida en el archivo JavaScript que hemos importado como "Logic".

2.3 VMware Workstation

Para el desarrollo del proyecto se ha hecho uso de un entorno de máquina virtual. Concretamente, hemos creado e instalado la distribución *Ubuntu* 12.04 LTS del sistema operativo *Linux*, y también el sistema operativo desarrollado por la marca *Apple*, *Mac OS X*.

Así, se ha hecho uso del entorno de virtualización conocido como "*VMware Workstation*" en su versión número 10. Esta herramienta nos permite ejecutar en **una misma** máquina física diferentes sistemas operativos como puedan ser *Windows*, *Linux*, *MAC OS*, etc.

Desde la versión 8.0 de *VMware Workstation* es necesario tener un ordenador con una arquitectura x64, es decir, tener un procesador de 64 bits. Era nuestro el caso, así que pudimos descargar e instalar este entorno siguiendo el enlace - https://my.vmware.com/web/vmware/details?downloadGroup=WKST-1002-WIN&productId=362&rPId=5403#product_downloads -. Una vez descargado el entorno de virtualización, procedimos a añadir e instalar nuestra máquina virtual de *Ubuntu* y nuestra máquina virtual *Mac*.

- Primero fue necesario descargar la versión de *Ubuntu* que antes hemos mencionado. Para ello accedimos a la página web oficial de *Ubuntu* y nos descargamos la versión de 64 bits. Hicimos esta misma operación para obtener una versión del sistema operativo *Mac OS X*, pero en este caso el sistema se obtuvo de otras fuentes, ya que al tratarse de un *software* propietario no se puede obtener gratuitamente desde la página oficial de *Apple*.

- Con eso pudimos obtener una imagen de CD ".iso" que emplearíamos más tarde al crear las máquinas virtuales.

- Para crear una máquina virtual, simplemente hay que acceder a la opción "New virtual machine…" del menú "File" en el entorno *VMware Workstation*. Tras hacer esto, se selecciona una opción de instalación: típica o personalizada.

- Lo siguiente es indicar dónde se encuentran en nuestro ordenador las imágenes de CD ".iso" que acabamos de descargar y con las que se procederá a instalar el sistema operativo *Ubuntu* y el sistema *Mac OS X*.

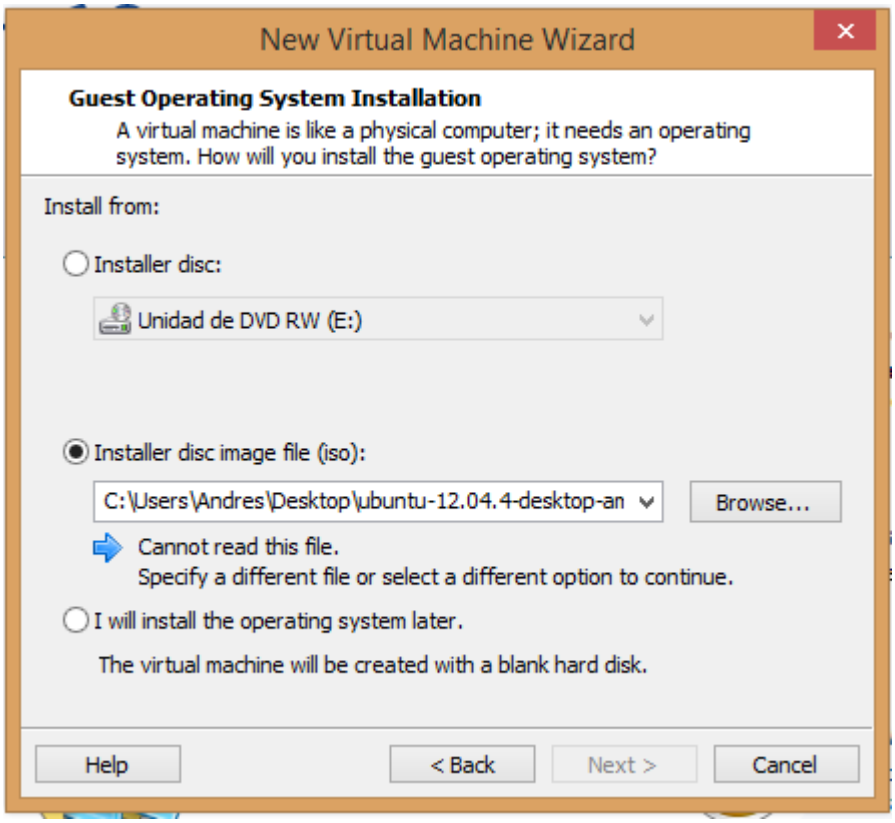


Ilustración 16 – Paso de instalación de una máquina virtual.

- Tras esto, será necesario indicar cuánta memoria y disco duro queremos dedicar a la máquina virtual. En nuestro caso asignamos 4 *gigabytes* de memoria RAM y 40 *gigabytes* de disco duro, no hacía falta más para tener un buen rendimiento.

- Finalmente, tras realizar esta serie de pasos basta con arrancar la máquina virtual de un sistema operativo y de otro para comenzar la instalación de dichos sistemas.

3. ANDROID

Como ya se ha dicho en secciones anteriores, nuestro proyecto iba a ir orientado a más de un sistema operativo móvil. Pues bien, uno de esos sistemas operativos es el sistema *Android*. A lo largo de esta sección, iremos comentando diferentes detalles sobre este sistema operativo, desde sus características más relevantes hasta aspectos relacionados con su amplia presencia en el mercado de los dispositivos móviles.

3.1 Historia

El sistema operativo *Android* era fundado en el año 2003 en Palo Alto, California. Desde sus inicios, la idea de *Android* surgió con la intención de desarrollar una plataforma *software* que permitiera abstraer el *hardware* y facilitar el desarrollo de aplicaciones para dispositivos móviles con recursos limitados.

Esta plataforma es *open source*, es decir, de código abierto, y se define sobre un núcleo (*kernel*) de *Linux*. Está diseñada, principalmente, para dispositivos con pantallas táctiles como puedan ser los *smartphones* o las *Tablets*, pudiendo usar así los dedos como mecanismo de entrada de datos y de interacción a través de diferentes gestos.

La compañía que iba a dar a conocer a este famoso sistema operativo, *Android* Inc., viviría un antes y un después tras la sucesión de un importante evento en el año 2005. Era en ese año cuando una empresa de enorme volumen llamada *Google* se convertía en la propietaria de *Android*. Así, desde entonces han sido unas once las versiones que se han diseñado de este sistema operativo.

La historia de las versiones del sistema operativo *Android* empezó en Noviembre de 2007 con el lanzamiento de la versión *Android* "beta". Sería en ese mismo año pero un poco más tarde cuando el SDK de esta plataforma *software* era lanzado, el cual comentamos con mejor detalle más adelante. Sin embargo, no era hasta el 23 de Septiembre de 2008 cuando la primera versión de *Android* se comercializaba, la conocida como *Android* 1.0. Cada una de las versiones que se han ido incorporando era acompañada por un nuevo nivel en el API de *Android*, quedando de la manera que se ve en la siguiente ilustración.

Platform Version	API Level	VERSION_CODE
Android 4.4	19	KITKAT
Android 4.3	18	JELLY_BEAN_MR2
Android 4.2, 4.2.2	17	JELLY_BEAN_MR1
Android 4.1, 4.1.1	16	JELLY_BEAN
Android 4.0.3, 4.0.4	15	ICE_CREAM_SANDWICH_MR1
Android 4.0, 4.0.1, 4.0.2	14	ICE_CREAM_SANDWICH
Android 3.2	13	HONEYCOMB_MR2
Android 3.1.x	12	HONEYCOMB_MR1
Android 3.0.x	11	HONEYCOMB
Android 2.3.4 Android 2.3.3	10	GINGERBREAD_MR1
Android 2.3.2 Android 2.3.1 Android 2.3	9	GINGERBREAD
Android 2.2.x	8	FROYO
Android 2.1.x	7	ECLAIR_MR1
Android 2.0.1	6	ECLAIR_0_1
Android 2.0	5	ECLAIR
Android 1.6	4	DONUT
Android 1.5	3	CUPCAKE
Android 1.1	2	BASE_1_1
Android 1.0	1	BASE

Ilustración 17 – Versiones de Android con sus respectivos niveles de API.

Desde 2009 se han desarrollado las diferentes versiones que se ven en la figura anterior (Ilustración 17), y desde entonces se estima que la versión más usada de este sistema a lo largo del mundo es la 4.x "Jelly Bean". Hay que decir que para 2013 *Google* estimó que alrededor de un billón de dispositivos hacían uso del sistema operativo *Android*. Reside aquí, pues, una razón de mucho peso para que nuestro proyecto fuera desarrollado también para sistemas *Android*.

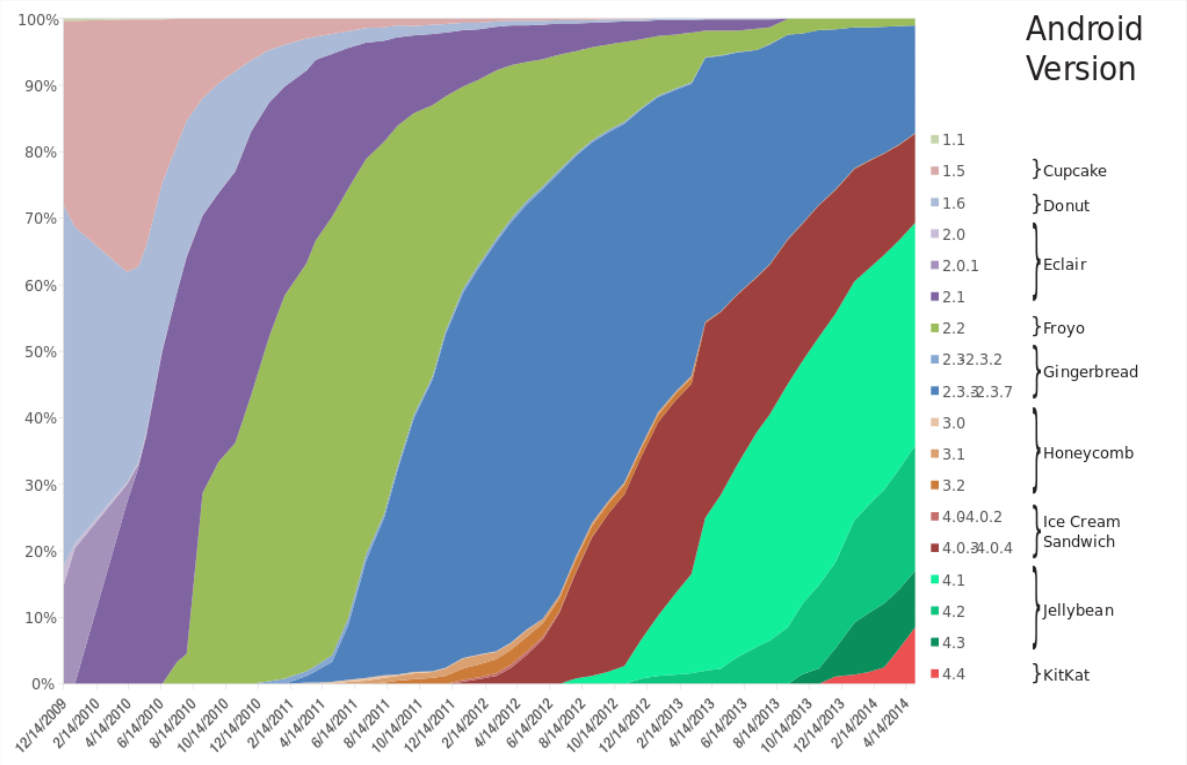


Ilustración 18 – Presencia en el mercado de las diferentes versiones de Android.

3.2 Arquitectura

Como se dijo anteriormente, *Android* contiene un núcleo o *kernel* basado en el núcleo del sistema operativo *Linux*. Se tomó esta decisión en el diseño de *Android* por el hecho de usar un núcleo que ya estaba probado y que era fiable. Es, dentro de esta arquitectura basada en capas (véase la ilustración 19), la única parte que **depende** del *hardware*, es decir que cada fabricante puede añadir aquí partes de *software* propietario. Así, *Android* aprovecha del *kernel*:

- Seguridad
- Gestión de memoria
- Gestión de procesos
- Red, gestión de alimentación
- Modelo de controladores, pudiendo soportar nuevos accesorios a través de nuevos controladores. Se desacopla el *hardware* del *software* a través de una capa de HAL (*hardware abstraction layer*).

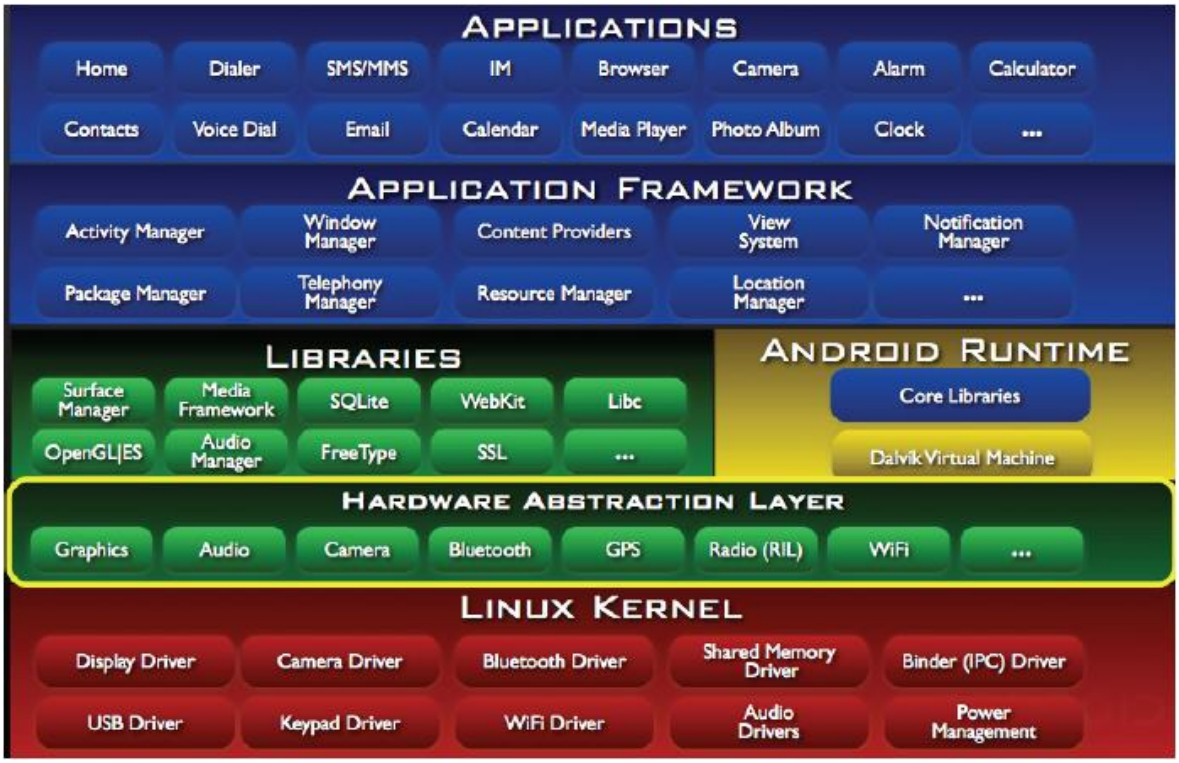


Ilustración 19 – Arquitectura en capas del sistema operativo Android.

3.3 SDK Android

El SDK (*software development kit* – conjunto de herramientas de desarrollo *software* para la creación de aplicaciones para cierto paquete o *framework*) de *Android* es lo que nos ha otorgado las librerías API y las herramientas de desarrollo necesarias para la construcción y las pruebas de ejecución sobre el juego que hemos desarrollado en el proyecto.

El SDK de *Android* está compuesto por paquetes modulares que se pueden descargar de manera separada a través de la herramienta conocida como "*Android* SDK Manager". De esta manera, cuando las herramientas del SDK son actualizadas o una nueva versión de la plataforma *Android* es lanzada, podemos hacer uso de la herramienta mencionada anteriormente y descargarnos rápidamente lo necesario para nuestro entorno de desarrollo.

Para obtener estas herramientas de desarrollo se puede descargar el SDK de *Android* desde su página oficial para desarrolladores (http://developer.*Android*.com/sdk/index.html).

3.4 Qt for Android

Se ha mencionado más de una vez en este documento el hecho de que el juego que se ha desarrollado estaba orientado, desde un principio, a más de un sistema operativo móvil. Pues bien, para poder desarrollar y testear nuestro juego para dispositivos ***Android*** tuvimos que hacer uso de la tecnología *Qt* para *Android* (*Qt for Android*). Este *framework* de desarrollo de *software* es lo que nos permite ejecutar aplicaciones *Qt* 5 en dispositivos *Android* que constan, como mínimo, de la versión 2.3.3 (nivel 10 del API) de este sistema operativo. Gracias a *Qt for Android* se puede conseguir, entre otras cosas:

- Ejecución de aplicaciones *Qml* en un dispositivo real
- Soporte de contenido multimedia para las aplicaciones *Qt Quick* 2 desarrolladas
- Compatibilidad con los sensores para reaccionar ante cambios
- Conseguir la posición actual usando información de satélites y de la red
- Establecer conexión en serie con otros dispositivos *Bluetooth*
- Desarrollar aplicaciones seguras usando librerías OpenSSL
- Crear y desplegar APK (el paquete de una aplicación) usando el entorno de desarrollo *Qt creator*

Por otra parte, para hacer uso de *Qt for Android* serán necesarios disponer de los siguientes requisitos:

- El SDK de *Android* que antes se ha explicado
- El NDK de *Android*
- Apache Ant 1.8 o superior
- Kit de desarrollador de Java SE versión 6 o superior, o usar OpenJDK en el caso de *Linux*

Procedemos a comentar cada uno de los requisitos que se han listado.

3.4.1 Android NDK

Consiste en un conjunto de herramientas que nos permite implementar partes de una aplicación usando código **nativo** como C o C++. En el caso de nuestra aplicación, sí hay algún fichero desarrollado con C++ como se comentará en el apartado de

"Implementación", pero son pocas líneas de código que no tienen mayor trascendencia en nuestro juego.

Para obtener el *Android* NDK también se puede hacer uso de la página de desarrolladores de *Android*, pudiendo descargar la versión de *Linux* de 64 bits en nuestro caso.

3.4.2 Apache Ant

Esta herramienta de línea de comandos consiste en una librería *Java* que pretende, fundamentalmente, suplir una serie de tareas ya incorporadas como la compilación, el ensamblado o el testeo de aplicaciones *Java*, aunque también puede ser usada para la construcción de aplicaciones C o C++.

En este proyecto se ha hecho uso de la versión 1.9.3 de *Apache Ant*, pudiendo descargarla a través de la página oficial de esta tecnología (http://archive.apache.org/dist/ant/binaries/).

3.4.3 OpenJDK

OpenJDK consiste en una versión en código abierto de la plataforma *Java SE*. Es, realmente, la única implementación en código abierto de *Java SE* para la que contribuye *Oracle*, empresa desarrolladora de la tecnología *Java*, entre otras.

De manera más precisa, para nuestro proyecto hemos contado con la versión 1.6 de *Java*, la cual instalamos en nuestro sistema operativo *Ubuntu* a través de la línea de comandos.

- `sudo apt-get install openjdk-6-jre`
- `sudo apt-get install openjdk-6-jdk`

Con esto, la versión 6 de *OpenJDK* quedaba instalada en nuestro sistema, cumpliendo así todos los requisitos necesarios para comenzar el desarrollo con *Qt for Android*.

Aun así, tras la instalación de la herramienta *OpenJDK* fue necesario configurar la variable de entorno JAVA_HOME para que apuntara al directorio en el que el JDK (*Java development kit*) de *Java* había sido instalado. Esto permitiría más tarde a la herramienta *Qt Creator* encontrar los archivos binarios necesarios para construir el juego que íbamos a desarrollar. Así pues, para realizar la configuración de la variable de entorno en el sistema operativo *Ubuntu* procedimos de la siguiente manera:

- Ejecutar en un terminal la instrucción: sudo su
- Tras la introducción del *password* con el que nos identificamos en el sistema hay que editar al archivo *bash.bashrc*, pudiendo para ello ejecutar una instrucción como: gedit .bashrc

- Introducir al final de dicho archivo las siguientes líneas (véase que al definir la variable JAVA_HOME se ha indicado el directorio en el que se instaló en **nuestro** sistema)

    - JAVA_HOME=/usr/lib/jvm/java-6-openjdk-amd64
    - export JAVA_HOME
    - PATH=$PATH:$JAVA_HOME
    - export PATH

Después de haber instalado estas herramientas que son necesarias y que son parte de los requisitos que se necesitan para poder hacer uso de la herramienta *Qt for Android*¸ hay que hacer una **actualización** del SDK de *Android* para obtener el API y los paquetes que contienen las herramientas necesarias para el desarrollo de nuestro proyecto. Para proceder con esta actualización, se hace uso de la herramienta *Android SDK Manager*, de manera que a través de la herramienta "terminal" podemos ejecutar la siguiente instrucción dentro, ubicándonos en el directorio en donde hayamos instalado el SDK:

- ./Android update sdk

3.4.4 *Instalación y configuración de Qt for Android*

Para que nuestro proyecto pudiera ser desarrollado y ejecutado en sistemas operativos *Android*, fue necesario que instalásemos y configurásemos la herramienta *Qt for Android*. A su vez, esta herramienta hará uso de los varios requisitos que estuvimos comentando anteriormente, por lo que, llegados a este punto, estos deberán estar correctamente instalados en nuestro sistema.

En nuestro caso, decidimos instalar la versión *Qt for Android 5.2.1*, la cual contenía la versión 3.0.1 de *Qt creator*, cómo se puede ver en la siguiente ilustración.

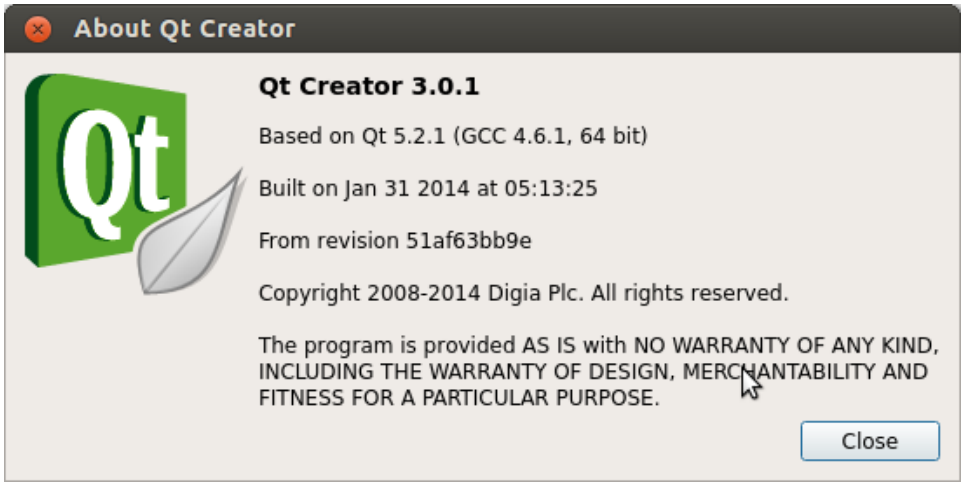


Ilustración 20 – Información de Qt creator, el cual venía incorporado en la versión 5.2.1 de Qt for Android.

Accediendo al sitio oficial de descargas de *Qt*, pudimos obtener la versión de *Qt for Android* antes mencionada. Una vez descargada, el siguiente paso era instalar este entorno con el que íbamos a trabajar.

- Concretamente, la herramienta *Qt for Android* iba a ser instalada en el entorno *Ubuntu* que se configuró como máquina virtual, así que descargamos el paquete "*Qt*-opensource-*Linux*-x64-*Android*-5.2.1.run" que era la que correspondía con dicho sistema. Una vez obtenido el archivo ".run" que se descarga, simplemente hay que ejecutarlo.

Ilustración 21 – Primera ventana del procedimiento de instalación de Qt for Android.

- Lo siguiente consiste en indicar el directorio en el que queremos que se instale todo lo necesario en lo que respecta a la tecnología *Qt*.

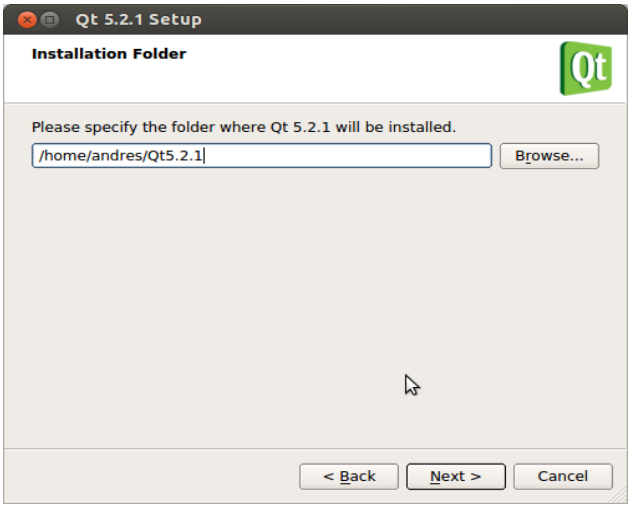


Ilustración 22 – Segunda ventana del procedimiento de instalación de Qt for Android.

- Finalmente, hay que indicar los componentes que deseamos que se instalen. Nosotros instalamos los componentes que ya venían seleccionados por defecto y que ya de por sí aglutinan todo lo necesario para el desarrollo de la parte de *Android* que nos tocaba en nuestro proyecto.

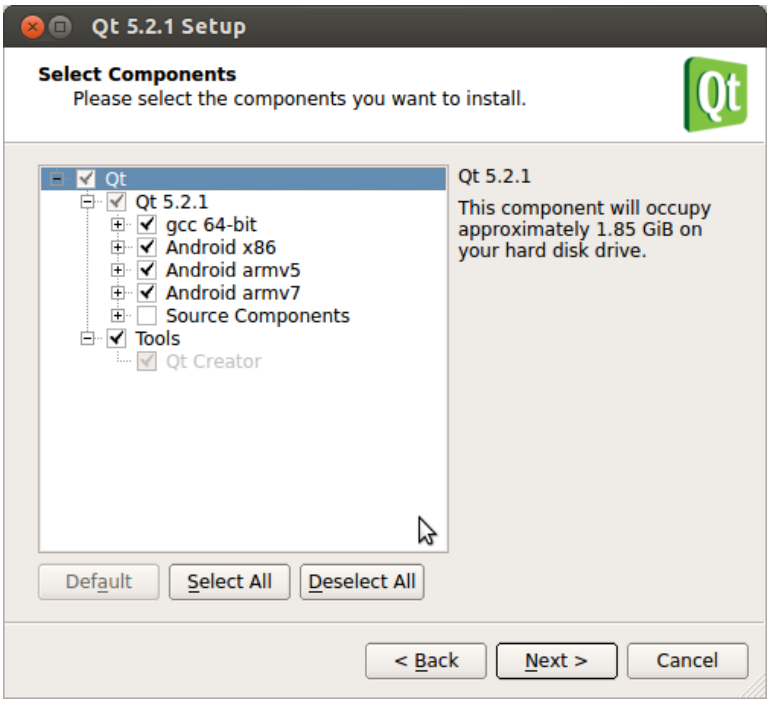


Ilustración 23 – Tercera ventana del procedimiento de instalación de Qt for Android.

Una vez se ha completado la instalación, hay que realizar una serie de pasos de **configuración** del entorno *Qt creator*. Estos pasos nos van a poder permitir comenzar a desarrollar nuestro proyecto para sistemas *Android*, pero sobretodo nos permitirá poder conectar dispositivos *Android* reales para que podamos construir y ejecutar el juego que resultará de nuestro proyecto.

- Dentro de *Qt creator* ir al menú "Tools > Options > Android". Aquí podremos especificar los directorios de SDK, NDK y de JDK que antes habíamos instalado, como se puede ver en la ilustración 23. También se especifica el directorio donde se instaló la tecnología *Apache Ant*.

- Hay que seleccionar la opción que indica "**Automatically create kits for *Android* tool chains**", la cual permitirá a *Qt creator* crear los kit de construcción y de ejecución de aplicaciones necesarios cuando se está creando un nuevo proyecto en esta herramienta. Esto se ve con más detalle en el apartado de "Implementación".

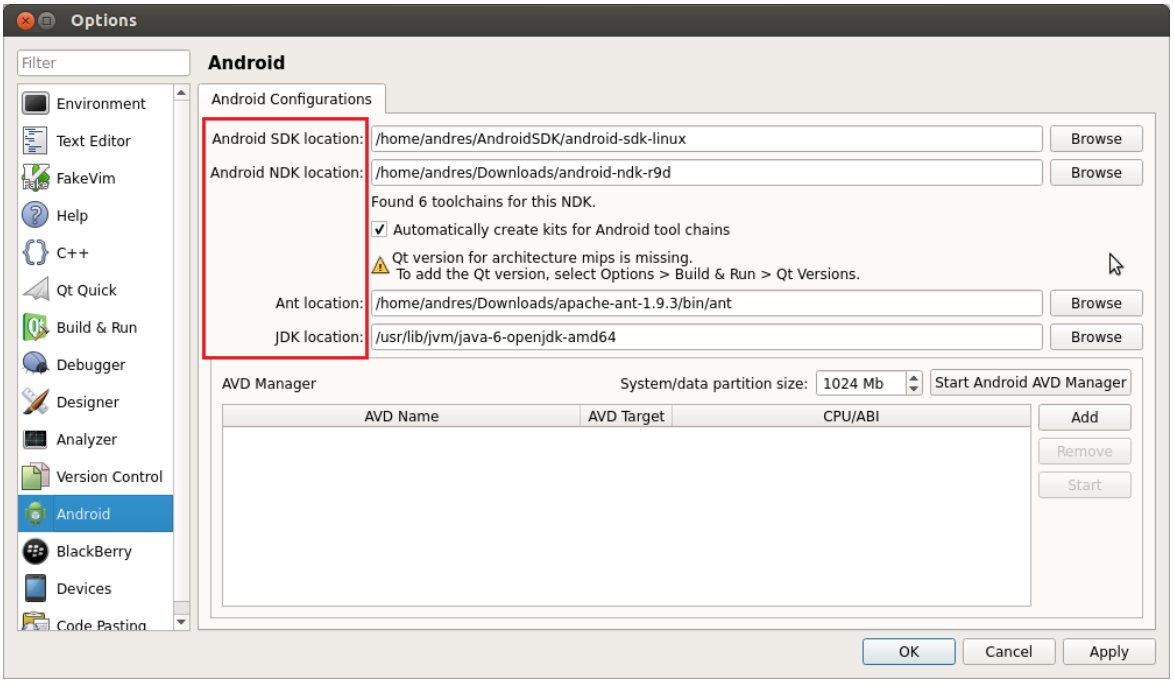


Ilustración 24 – Especificación de los directorios de las herramientas necesarias para el desarrollo de aplicaciones para sistemas Android.

- Es también importante habilitar la depuración USB en el dispositivo o dispositivos reales en los que vayamos a hacer test de nuestra aplicación. Esta opción se encuentra en diferentes partes según el dispositivo del que estemos hablando, pero suele estar en el apartado de "opciones de desarrollador".

4. SAILFISH

Otro de los sistemas operativos para los que iba dirigido este proyecto es el conocido como *Sailfish* OS. *Sailfish* consiste en un sistema operativo para dispositivos móviles. Está basado en el sistema operativo *Linux* y ha sido desarrollado por la compañía ***Jolla*** en cooperación con el proyecto conocido como **Mer**. *Jolla* es una compañía dedicada estrictamente al diseño y al desarrollo de dispositivos móviles, los cuales, por tanto, son equipados con el sistema operativo que desarrolla la propia compañía, *Sailfish*.

A pesar de que los dispositivos móviles de *Jolla* estén equipados con la tecnología *Sailfish*, existen también otras marcas y dispositivos para los que se ha portabilizado el sistema operativo, como puedan ser: *Acer* Iconia Tab W500, *Google* Nexus One, *Google* Nexus 4, 5 y 7, *Samsung* Galaxy S3, *Hp* Mini, etc.

4.1 Historia

Desde su aparición, han sido varias las versiones que han sido lanzadas de este sistema operativo de **código libre**. Se resume en forma de tabla cada una de ellas, con sus respectivos nombres y con sus fechas de lanzamiento.

Versión	**Nombre**	**Fecha lanzamiento**
Inicial	Kaajanlampi	28 Noviembre 2013
v1.0.1.10	Laadunjärvi	9 Diciembre 2013
v1.0.2.5	Maadajävri	27 Diciembre 2013
v1.0.3.8	Naamankajärvi	31 Enero 2014
v1.0.4.20	Ohijarvi	17 Marzo 2014
v1.0.5.16	Paarlampi	11 Abril 2014
v1.0.5.19	Paarlampi	24 Abril 2014

4.2 Arquitectura

Tanto el sistema operativo *Sailfish* como su SDK, del que también hablaremos, están basados en el *kernel* de *Linux* pero también en la distribución *software Mer*.

Desde *Sailfish* creen que existe hoy en día una necesidad de un sistema operativo para dispositivos móviles que sea **moderno** e **independiente**, permitiendo una competición libre e innovación. De esta manera, con *Sailfish* OS se pretende ofrecer una serie de ventajas competitivas respecto a las que puedan ofrecer los dispositivos de *Android* o los *iOS* de *Apple*. *Sailfish* se quiere plantear como una alternativa real.

La arquitectura es:

- *Hardware*: el sistema operativo de *Sailfish* está diseñado para ser ejecutado en entornos embedidos como puedan ser *smartphones* o *tablets*.

- *Kernel*: este sistema operativo presenta en su arquitectura un núcleo basado en *Linux*, con algunas modificaciones.

- Componentes principales: para los componentes principales *Sailfish* hace uso de del proyecto *Mer*. Esto incluye: multimedia, gráficos, tecnologías de comunicación, gestión de *software*, seguridad, etc.

- Interfaz de usuario (UI) y *middleware*.

- Aplicaciones.

Sailfish ha sido diseñado como una extensión o una herencia de la tecnología conocida como *MeeGo* (un sistema operativo libre para móviles que resultó de la fusión entre *Moblin* de *Intel* y *Maemo* de *Nokia*). Esto permite asegurar que las funcionalidades principales del sistema operativo como la gestión de la potencia y de la energía, o la gestión de la conectividad, ya están incorporadas y optimizadas para sistemas embedidos. La interfaz de usuario (UI), por su parte, está construida gracias a *Qml* y *Qt Quick*, tecnologías que ya se han comentado detenidamente en este proyecto. Estas dos tecnologías permiten una fácil y rápida personalización de la UI, algo bastante destacable.

Ilustración 25 – Arquitectura de Sailfish OS.

4.3 Sailfish SDK

El SDK del sistema operativo *Sailfish* era anunciado en el año 2012 en Helsinki, y al tratarse de un *software* de código abierto, su descarga está disponible de manera gratuita a través de la página oficial de *Sailfish*.

Es de remarcar el hecho de que este SDK hace uso de la tecnología *Qt* pero también de una tecnología de virtualización como lo es *Virtual Box* de *Oracle*, la cual es muy similar a la tecnología *VMware Workstation*. Con estas dos tecnologías se consigue el desarrollo, la compilación y la emulación necesarias a la hora de desarrollar el juego. En nuestro caso, se hicieron pruebas tanto en el emulador de *Sailfish* que se configura en la tecnología *Virtual Box* como en un dispositivo real *Jolla* que tenía como soporte el sistema *Sailfish*.

Desde *Jolla* se dice que el desarrollo a través del *Sailfish* SDK es muy similar al propio desarrollo del *Sailfish* OS en sí mismo, no hay diferencias entre el *software* que se desarrolla gracias al SDK respecto al *software* que puede usar un usuario en un dispositivo equipado con *Sailfish* OS. Esto lo afirman así desde *Jolla* porque el propio SDK de *Sailfish* contiene *Sailfish* OS en su totalidad, y emula así el sistema operativo *Sailfish* sobre el computador o sistema que esté ejecutando dicho SDK, lo que significa, en pocas palabras, que el sistema entero de *Sailfish* OS, incluyendo su *kernel*, está siendo ejecutado en el SDK de la máquina virtual. Aquí reside el motivo por el cual el desarrollo y la portabilidad de cualquier *software* son posibles con el SDK de *Sailfish*.

Es la naturaleza de código abierto del SDK de *Sailfish* lo que permite configurarlo y reconstruirlo para las necesidades propias y específicas de una compañía o de cualquier organización. A esto se añade la característica de que el SDK soporta una gran variedad de sistemas operativos como *Android*, *Linux* en sus distribuciones de 32 y 64 bits, el sistema operativo *OS X* de *Apple*, y *Windows*.

4.4 Tecnología

Resulta que uno de los pilares tecnológicos del sistema operativo *Sailfish*, consiste en una tecnología que ya se ha comentado de forma amplia a lo largo de este documento, y esa es el *framework Qt* para el desarrollo de aplicaciones. Como ya se sabe, esta es una tecnología multiplataforma que consta de un conjunto de librerías para el desarrollo de aplicaciones centradas en los usuarios. Así, el *framework Qt* otorga al sistema *Sailfish* OS la posibilidad de desarrollar y diseñar una experiencia de usuario muy poderosa gracias al uso del lenguaje *Qml*.

El lenguaje *Qml* y sus características dan a *Sailfish* OS la habilidad de contener una serie de elementos para el diseño de interfaces de usuario, y así poder crear aplicaciones con interfaces animadas y en las que el usuario puede interactuar a través de pulsaciones táctiles.

Para un mejor rendimiento y para una mejor experiencia de usuario, aquellas aplicaciones que sean desarrolladas, además, con la tecnología *Sailfish Silica*, unos componentes personalizados diseñados gracias a estilos de UI que *Jolla* ha creado, podrán aprovechar la potencia total de la que constan dispositivos que están equipados con sistemas *Sailfish* OS.

4.5 Qt for Sailfish

Nuestro proyecto también iba a ir destinado a dispositivos que se ejecutan sobre sistemas operativos *Sailfish*, así que era necesario tener instalado y configurado lo necesario para poder desarrollar para este tipo de plataforma. Para ello lo primero y necesario que tuvimos que hacer fue descargar el SDK con su conjunto de herramientas

de desarrollo desde la página oficial de *Sailfish*. En nuestro caso, tuvimos que descargar la versión de 64 bits para distribuciones *Linux*.

4.5.1 Instalación del SDK de Sailfish

Una vez descargado el archivo necesario, se puede proceder con la instalación del SDK con el que podremos desarrollar nuestro proyecto. Para ello, en nuestro caso, y al tratarse de un sistema *Ubuntu*, bastó con simplemente ejecutar el archivo ".run" que se obtiene tras la descarga, con lo que apareció el marco de instalación habitual.

Ilustración 26 – Primer paso en la instalación de Sailfish OS SDK.

El siguiente paso consistía en indicar el directorio en el que se instalaría el SDK con todos sus componentes. En nuestro caso lo dejamos al que salía por defecto (véase la ilustración 26). Más tarde, en el siguiente diálogo de la instalación se podrá indicar un directorio opcional o alternativo en el que se puedan guardar los proyectos que se creen, ya que, por defecto, los proyectos serán creados en el directorio *home* o *user* (véase la ilustración 27).

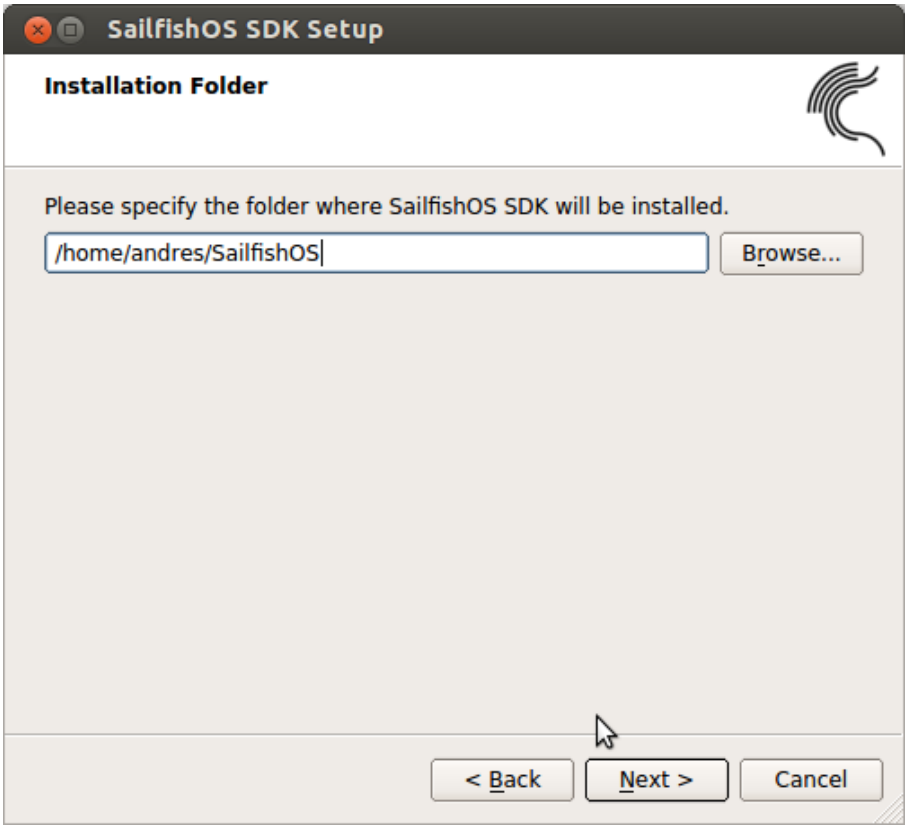


Ilustración 27 – Segundo paso en la instalación de Sailfish OS SDK.

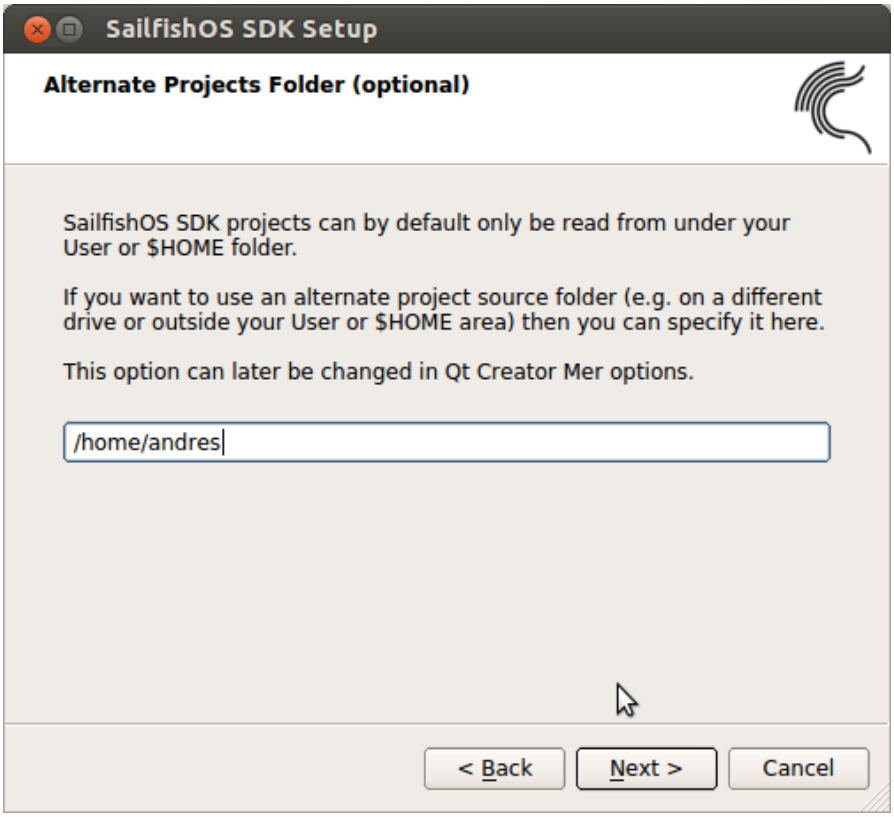


Ilustración 28 – Tercer paso en la instalación de Sailfish OS SDK.

En última instancia, habrá que especificar qué componentes del *Sailfish* SDK deseamos que se instalen. Se pueden dejar e instalar los que ya vienen indicados por defecto (véase la ilustración 28). Tras esto, simplemente habrá que indicar que se aceptan los acuerdos de la licencia del producto y, finalmente, comenzar la instalación.

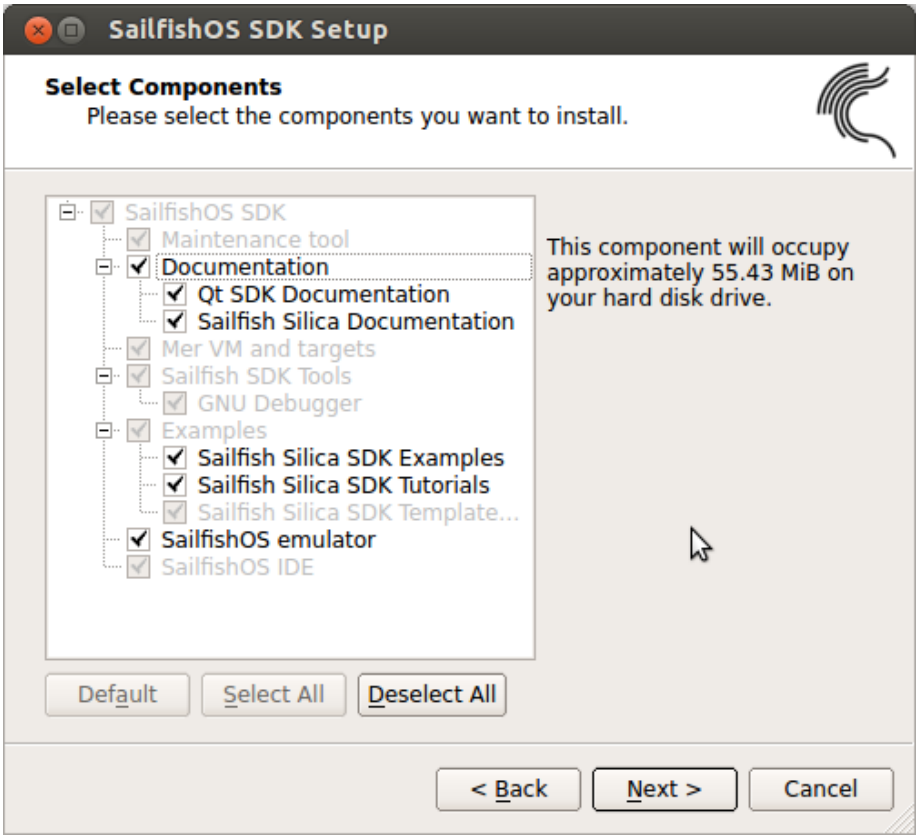


Ilustración 29 – Cuarto paso en la instalación de Sailfish OS SDK.

5. OTRAS PLATAFORMAS

Además de ejecutar el proyecto en sistemas operativos móviles como *Android* o *Sailfish*, otros objetivos que se buscaban conseguir en el proyecto eran:

- Intentar ejecutar el juego en dispositivos móviles con sistema operativo *iOS*.
- Poder llevar el proyecto también a sistemas operativos de escritorio como *Windows*, *Mac* o *Linux*.

5.1 Plataforma iOS

iOS consiste en otro sistema operativo móvil. En este caso, se trata de un sistema operativo diseñado y desarrollado por la marca *Apple* para sus dispositivos móviles, tales como el *iPad* o el *iPhone*. Domina actualmente, junto con *Android*, el mercado de dispositivos móviles, donde alrededor de un noventa por ciento de los dispositivos *Apple* se encuentra bajo la versión última de *iOS*, la siete.

Debido al gran peso que tiene *iOS* en la industria actual de los dispositivos móviles, nos veíamos en la obligación de, al menos, **intentar** ejecutar el juego que estábamos desarrollando en un dispositivo *iOS* real. Esto, tal y como se detalla en el apartado de "Problemas en el desarrollo", no iba a ser posible, y fue algo que descubrimos al principio del proyecto. Sin embargo, existía la posibilidad, al menos, de ejecutar nuestra aplicación en **simuladores** de dispositivos *iOS*, lo cual nos **garantizaba** que el proyecto era, en realidad, portable para esta plataforma. Para conseguir todo esto era necesario:

- Contar con el sistema operativo *Mac OS* ya que era necesaria la herramienta *XCode*, la cual sólo puede instalarse para este sistema.
- El entorno de desarrollo *Qt* para sistemas *Mac*.

5.2 XCode

Fue necesario instalar el sistema operativo *Mac OS* en nuestro entorno de máquina virtual. Esto nos permitiría contar con la herramienta de desarrollo de *software* conocida como *XCode*. Esta herramienta consiste en un entorno de desarrollo de aplicaciones más, la cual fue ideada por *Apple* y, a su vez, destinada para dispositivos *Apple* como *Mac*, *iPhone* y *iPad*. De esta manera, fue necesario combinar el uso de esta herramienta con un entorno *Qt* para *Mac*, ya que queríamos testear el proyecto que estábamos haciendo en simuladores de dispositivos *iOS*.

Para obtener esta herramienta de desarrollo se puede acceder a la página de desarrolladores de *Apple* (https://developer.*Apple*.com/xcode/downloads/) y descargar la versión deseada. Es necesario descargar una versión de *XCode* que sea compatible con la versión del sistema *Mac* sobre el que se instalará. Como en nuestro caso contábamos con un sistema operativo *Mac OS X 10.8*, la versión de *XCode* que descargamos era la 4.6.1. Según la versión de *XCode*, las versiones de *SDK* de *iOS*, las arquitecturas en las que se puede desplegar la aplicación, o los dispositivos sobre los que esta se puede ejecutar, son diferentes. En este caso, la versión de *XCode* que se instaló establecía lo siguiente:

- Arquitectura estándar armv7, armv7s: se trata de una arquitectura *ARM* de procesadores.

- *SDK* de *iOS* base: 6.1

- Versiones en las que se puede desplegar la aplicación desarrollada: desde *iOS* 4.3 a *iOS* 6.1. Esto no nos interesaba realmente porque no íbamos a testear la aplicación en un dispositivo real.

- Simuladores que incluye: *iPad* 6.1, *iPhone* 6.1

Para el desarrollo de la aplicación se podía hacer uso tanto de la herramienta *XCode* como del entorno de desarrollo *Qt* que se instaló para el sistema *Mac*. El esquema de trabajo sería el que sigue: es necesario tener un fichero ".pro" del proyecto, el cual se obtiene a través de la herramienta *Qt*, y a partir de ahí, exportar dicho archivo a la herramienta *XCode*. Esto nos permitirá abrir el proyecto que estamos haciendo con *Qt* en el entorno *XCode*, para poder así emplear los simuladores de dispositivos *iOS* que incluye dicho entorno. Es, como se ve, un trabajo conjunto el que tiene lugar entre ambas tecnologías.

5.3 Entorno Qt en Mac

En nuestro caso, como no íbamos a testear la aplicación en dispositivos *iOS* reales, la versión *Qt* que empleásemos era indiferente, es decir que podíamos emplear el entorno *Qt for iOS* o simplemente un entorno *Qt* normal en su versión para *Mac*. Dicho esto, para la herramienta *Qt* que necesitábamos en *Mac* descargamos la versión 5.2.1, la cual incluye el compilador *Clang*, el cual se detalla más adelante.

Este entorno es el que podemos usar, al igual que ocurría con los otros sistemas operativos, para el desarrollo de nuestro proyecto. Es el que incorpora automáticamente, tras su instalación, los kit de construcción y de aplicación necesarios para testear en dispositivos o en los simuladores de *XCode*. Se explica en qué consisten los kit del entorno *Qt* en el apartado de "Implementación". Sin entrar en más detalle, seleccionando el kit etiquetado como "iphonesimulator-clang" se crea un fichero de

extensión ".xcodeproj", el cual nos permitirá abrir el mismo proyecto en la herramienta *XCode*. Este es el paso equivalente a exportar el fichero ".pro" a la herramienta *XCode*. Esto es, en sí, lo que nos permite evaluar nuestro juego en un simulador de dispositivos *iOS*. Para verlo más detalladamente se puede consultar el apartado 10.2 de "Pruebas en *iOS*".

5.4 Sistemas operativos de escritorio

Para sacar aún más provecho de las labores de compilación cruzada, el proyecto podía ser también orientado a sistemas operativos de escritorio. Esta no consistía en una labor mucho más compleja de la que habíamos hecho para conseguir ejecutar el proyecto en sistemas operativos móviles, ya que contábamos con los entornos de desarrollo adecuados que permitían afrontar esta situación.

En efecto, para cada sistema operativo de escritorio para el que decidiésemos ejecutar el proyecto, fue necesario contar con un entorno *Qt* de desarrollo orientado a dicho sistema operativo. Así, se tomó la decisión de poder llevar el juego desarrollado a sistemas operativos *Windows*, *Mac* y *Linux* (*Ubuntu*).

5.4.1 Windows

El ordenador en el que se iba a desarrollar el proyecto tenía como sistema operativo *Windows*, en su versión 8.1, sistema desarrollado y vendido por la empresa *Microsoft*. Son numerosas las novedades que esta versión del sistema operativo más distribuido en los ordenadores del mundo trae, pero no es objeto de este proyecto comentar cada una de ellas.

Así, el sistema operativo del ordenador de trabajo era un sistema de 64 bits, arquitectura que permitiría, posteriormente, contar con la instalación de un sistema *Mac OS* en un entorno de máquina virtual. En cuanto a características a nivel de *hardware*, el ordenador contaba con:

- Procesador *Intel Core* i7-3537U a 2.00 GHz de frecuencia.
- Memoria RAM instalada de 8 *Gigabytes*.

Dicho esto, y teniendo en cuenta las características del sistema que se tenía en manos, se instaló una versión 5.2.1 del entorno *Qt* de programación. Si el proyecto que se iba a desarrollar se había creado en un entorno *Qt* sobre otro sistema, como por ejemplo *Mac*, desde la herramienta *Qt* del sistema *Windows* simplemente había que realizar la tarea de **abrir** el proyecto. Tras abrir el proyecto, era necesario configurar el kit de construcción y de ejecución que automáticamente se añaden al instalar la herramienta *Qt*. En este caso, se trata de un kit denominado *Desktop* que permite crear

un archivo ".exe" del juego que se está desarrollando, lo que permitirá ejecutar dicha aplicación sobre el sistema *Windows*.

Así, al intentar abrir el proyecto, aparece un diálogo en el que el usuario configura los kit necesarios para construir los ejecutables de la aplicación. En el caso de *Windows* aparece algo así:

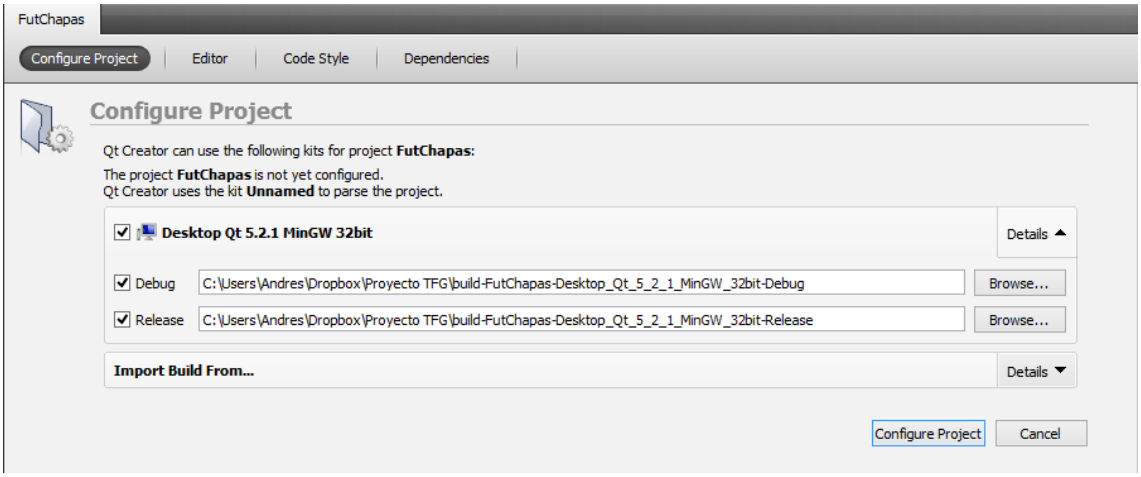


Ilustración 30 – Elección de kit Desktop al abrir un proyecto en el entorno Qt de Windows.

5.4.2 Linux (Ubuntu)

Como se indicó en el apartado de "Herramientas y tecnología", se contó con un entorno de máquina virtual con el que poder instalar y configurar sistemas operativos diferentes del nativo que tenía el ordenador de trabajo. Uno de estos sistemas operativos fue la conocida distribución del sistema operativo *Linux*, *Ubuntu*, en su versión 12.04 LTS.

Con *Ubuntu* se tiene un sistema operativo de *software* libre cuyo núcleo está basado en el sistema operativo *Linux*, de ahí la relación entre ambos. No hubo ninguna razón en particular que nos obligará a instalar la versión 12.04 en concreto, se puede hacer uso de otra versión si se desea.

En este caso, para poder ejecutar el juego en este tipo de sistema, se contó con el mismo entorno *Qt* de desarrollo que el que se empleó para sistemas *Android*, es decir, el entorno *Qt for Android*. Esto fue así porque dicho *framework* contaba tanto con un kit para la construcción de código ejecutable para sistemas móviles *Android*, como de un kit para la construcción de código ejecutable para el sistema *Ubuntu*, siendo una vez más un kit *Desktop*. Empleando este kit se podrá crear un archivo de tipo *executable* que permitirá ejecutar la aplicación en el sistema *Ubuntu*.

5.4.3 Mac OS X

El sistema operativo *OS X*, también conocido como *Mac OS X*, refleja la última versión del sistema operativo que la empresa *Apple* desarrolla para ordenadores personales. Se trata de un sistema basado en *Unix*, como ocurre en el caso de las distribuciones de *Linux* como pueda ser *Ubuntu*, y en él se intenta exprimir al máximo las capacidades *hardware* del ordenador sobre el que se ejecuta. El interés de poder ejecutar el juego creado en este tipo de sistemas operativos, reside en el protagonismo que este sistema ha ido adquiriendo en el mercado de los ordenadores a lo largo de los últimos años (véase la ilustración 30).

En esta nueva ocasión, para crear el código ejecutable de la aplicación para sistemas de escritorio *Mac*, se emplea un entorno de desarrollo *Qt* que se descarga e instala para dicho sistema. Es, otra vez, la versión 5.2.1 de esta tecnología la que se utiliza, pero con algunas modificaciones que están orientadas al sistema *Mac*, como pueda ser la incorporación de un compilador *Clang*. Como ocurre con el resto de sistemas operativos, se puede abrir el proyecto en el que se encuentra el código de la aplicación que se está desarrollando y seleccionar, seguidamente, los kit de construcción y de ejecución que se requieran. Para poder ejecutar sobre el entorno de escritorio en este sistema, se hará uso, otra vez, de un kit *Desktop* que genera un archivo de tipo *application*. Como se ve, este entorno es el mismo que se detalla en el apartado 5.3, el cual se emplea tanto para generar código ejecutable para sistemas *iOS* como para sistemas *OS X*.

Una vez comentados cada uno de los sistemas de escritorio, sería importante ver el peso que estos tienen en el mercado actual de los ordenadores personales, aunque esto no sea un objeto de estudio de nuestro proyecto. Para esto, se puede apreciar la siguiente gráfica, donde se ve que el sistema operativo *Windows* predomina fuertemente en el mercado, donde ha ido perdiendo, no obstante, cada vez más protagonismo.

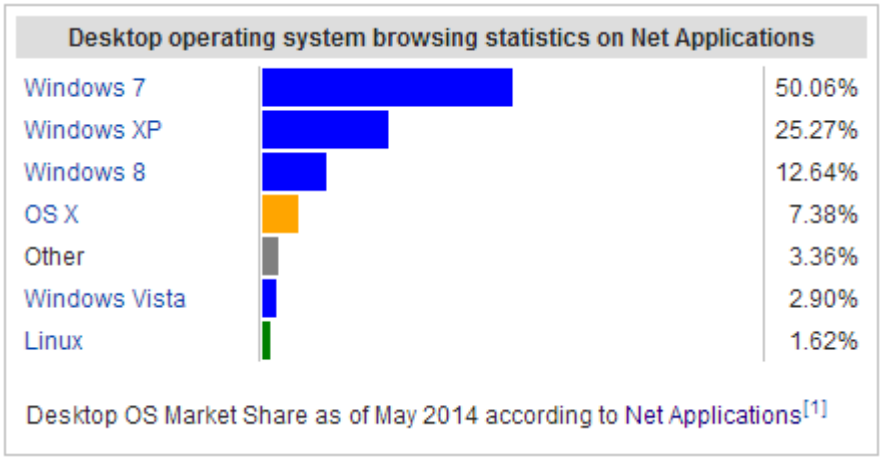


Ilustración 31 – Comparativa del dominio actual del mercado de sistemas operativos de escritorio (Mayo 2014).

6. COMPILACIÓN CRUZADA

Por medio de la compilación cruzada, somos capaces de **crear un código ejecutable** para nuestra aplicación de forma que este código se pueda ejecutar en una plataforma distinta a la que se utilizó durante la compilación. Así, esto nos ha permitido en nuestro proyecto poder compilar el código de nuestra aplicación en una plataforma o sistema operativo determinado, para producir entonces un código que se pudiera ejecutar en una plataforma o sistema operativo **diferente**.

En términos generales, un compilador cruzado es un compilador que se ejecuta en una máquina o plataforma A pero que genera un código ejecutable para una máquina o plataforma B, distinta. Esto significa que las dos plataformas, A y B, pueden ser diferentes en cuanto a arquitectura de procesador, sistema operativo y/o formato de ejecutable.

Un ejemplo de compilación cruzada es el que se tiene con el compilador *Netwide Assembler*, un ensamblador libre destinado a la plataforma *Intel x86*. Puede ser usado para programas de 16 bits y de 32, y, en este último caso, haciendo uso de las librerías pertinentes, los programas se pueden escribir de una forma en la que se garantiza la portabilidad de estos a **cualquier** sistema operativo de 32 bits, manifestándose aquí la potencia de la compilación cruzada al poder abarcar múltiples plataformas. Este compilador, conocido también como *NASM*, puede generar varios formatos binarios, código objeto y bibliotecas compartidas en cualquier máquina, incluyendo *COFF* (usada en sistemas *Unix*), *a.out* (usada en versiones antiguas del sistema *Unix* y también para sistemas *Linux*), *ELF* (también ampliamente usado en sistemas *Unix* y *Linux*), entre otros. Esta variedad de formatos de salida es lo que permite portar los programas a cualquier sistema operativo *x86*. Además, este compilador puede crear archivos binarios planos, usables para escribir cargadores de arranque, imágenes *ROM*, y varias facetas del desarrollo de sistemas operativos. Es un compilador que, además, puede correr en otras plataformas a parte de la *x86* como lo son *SPARC* y *PowerPC*. El *NASM* tiene como objetivo producir archivos objeto, es decir, los ejecutables que se obtienen al compilar el código fuente. La única excepción a esto son los binarios planos *.COM* (otro formato de código ejecutable, en este caso sin metadatos, sólo código y datos) que son limitados en el uso moderno.

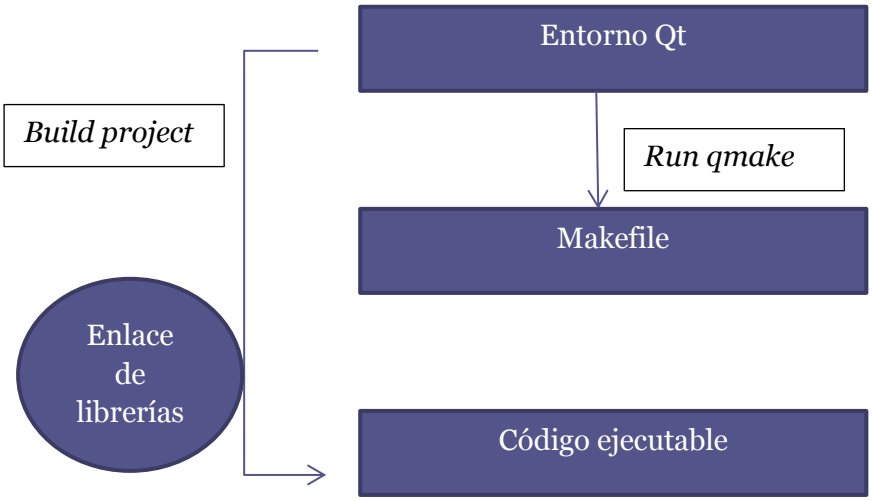
En lo que al proyecto se refiere, para poder llevar a cabo este proceso, nos hemos valido de diferentes entornos *Qt* de programación, los cuales nos han permitido crear un código ejecutable para varios sistemas operativos. Con estos *frameworks* de desarrollo junto con sus componentes, pudimos llevar a cabo el juego que teníamos que elaborar para el proyecto. Ellos, realmente, han sido los que nos han facilitado el trabajo de tener que elaborar un **mismo** código que sería destinado para plataformas que eran **diferentes**.

Al margen de estos *frameworks* que han facilitado las labores de compilación cruzada, la posibilidad de construir un compilador cruzado existe, pero era algo que no estaba dentro del ámbito de este proyecto. Gracias a la tecnología *Qt* de la que se hizo uso, el hecho de tener que desarrollar un compilador cruzado a medida fue algo que se pudo descartar. Cuando se habla de construir un compilador cruzado, se habla de tener que añadir una serie de dependencias a este. Por ejemplo, si se desea construir un compilador cruzado *GCC* será necesario, primero, instalar dependencias como *GNU make*, *GNU bison*, *flex*, y, obviamente, el compilador del sistema de trabajo que se desea reemplazar o modificar. Además, se necesitarían una serie de paquetes como *GNU GMP*, *GNU MPFR*, y *MPC*, los cuales utiliza *GCC* para operaciones en coma flotante. Por otro lado, se necesitaría un sistema en el que hubiera una instalación de *GCC* en funcionamiento, como era de esperar. Es importante también, en la construcción de un compilador cruzado como este de *GCC*, descargar lo que se conoce como *GNU binutils*, con los cuales se tiene una serie de herramientas de programación a modo de comandos para manipular código de objeto en varios formatos de archivos objeto. De entre los comandos más destacables se pueden considerar:

- as: ensamblador
- ld: enlazador
- ar: crea, modifica y extrae desde archivos
- c++filt: filtro para símbolos C++

Siguiendo con este ejemplo, al ya tener los elementos necesarios, lo que resta consistiría en construir las herramientas que en nuestro sistema de trabajo permiten convertir el código fuente en archivos objeto para la plataforma que se desee. Primero, habría que decidir dónde instalar el nuevo compilador, para luego poder añadir y compilar en el directorio de instalación los *binutils* que se habían descargado.

Dejando de lado el ejemplo, y como ya se ha dicho, fue la tecnología *Qt* la que facilitaría esto en el proyecto. Los entornos *Qt* de programación con los que se fueron contando, eran los encargados de añadir las librerías necesarias atendiendo a la plataforma para la que se estaba compilando el código. A modo de diagrama, puede observarse cómo es el procedimiento que se sigue al obtener el código ejecutable apoyándonos en la tecnología *Qt*.



Como se ve, en el entorno *Qt* en el que se desarrolla el proyecto primero se debe ejecutar una orden *qmake*. Esto se puede hacer a través de una herramienta de línea de comandos o bien, aprovechando los propios entornos *Qt*, a través de una opción de "Run qmake" que se puede pulsar en la herramienta de trabajo. Este paso es importante porque genera un archivo conocido como *Makefile*. Efectivamente, con la herramienta *qmake* se facilita el poder construir el código ejecutable para diferentes plataformas, con lo que se obtendrán distintos archivos M*akefile* para cada plataforma. El archivo *Makefile* está basado en el fichero de extensión ".pro" del proyecto, archivo que se explica con más detalle en el apartado de "Implementación". Con esto, lo que falta para obtener el código ejecutable destinado para cierta plataforma, es la ejecución de la orden "Build", la cual también se accede fácilmente desde el entorno *Qt* de trabajo. Esto permitirá transformar el código fuente en un código binario que sea ejecutable para una plataforma destino. También se comenta en más detalle las opciones de "Build" que permiten obtener código para distintas plataformas en el apartado de "Implementación".

Así, cada uno de los sistemas operativos para los que queríamos orientar el proyecto nos obligó a configurar un entorno *Qt* diferente en cada uno de ellos, lo que nos permitiría, al final, obtener un código ejecutable para cada sistema. Esto, sin embargo, ya nos hacía vislumbrar una de las primeras dificultades de la compilación cruzada.

Por un lado están los sistemas operativos de escritorio como *Windows*, *Mac* o las distribuciones de *Linux* como pueda ser *Ubuntu*. En el caso de *Windows*, fue necesario configurar un entorno *Qt* en el que venía incluido un compilador de tipo *MinGW* en su versión 4.8. Este, como otros tantos componentes de dicho entorno, se añade automáticamente al instalar el propio entorno. El compilador ***MinGW*** consiste en una implementación del compilador GCC, pudiendo migrar así este compilador para sistemas operativos *Windows*. Como se ve, esta herramienta ha sido fundamental para poder obtener el código ejecutable para sistemas *Windows*. Si hablamos de sistemas *Linux*, donde se ha utilizado una distribución *Ubuntu*, es de esperar el uso de un compilador del proyecto GNU como lo es **GCC**. Para la obtención del código ejecutable para este sistema de escritorio, se tendrá que emplear un kit *Desktop* en el entorno *Qt*. El concepto de kit en los entornos *Qt* se explica más detalladamente en el apartado de "Implementación". En el caso de sistemas operativos *Mac*, su entorno *Qt* incluye un compilador ***Clang*** para poder crear el código ejecutable orientado a este sistema de escritorio. *Clang* consiste en un compilador de código C/C++ y de Objective-C que forma parte del proyecto *LLVM*, el cual es una colección, al igual que GNU, de compiladores de alta velocidad.

Por otro lado, la dificultad de la compilación cruzada se sigue haciendo presente al querer llevar el proyecto a sistemas operativos móviles como *Android*, *Sailfish* o *iOS*. En el caso de *Android*, a través de un entorno *Qt for Android* se incluyen, en su instalación, los componentes necesarios para poder desplegar el proyecto en dispositivos de este tipo. Se incluyen, pues, una serie de kit de construcción y de ejecución para las aplicaciones, de forma que se tendrá que emplear el kit adecuado

teniendo en cuenta la arquitectura del dispositivo en el que se quiere desplegar la aplicación. En nuestro caso, fue necesario emplear un compilador ***Android* GCC** (arm - 4.8). Se trata de un compilador GCC, una vez más, orientado a dispositivos con arquitectura de procesador ARM, como era el caso de nuestro dispositivo. Aquí se ve, pues, otra dificultad de la compilación cruzada, ya que habrá que emplear el compilador adecuado atendiendo a la arquitectura del dispositivo al que se destine la aplicación.

6. 1 Compilación cruzada con GCC

Se observa que en todos los casos descritos anteriormente se ha nombrado el compilador GCC. Esto es así porque con GCC se tiene un compilador multiplataforma (*cross-platform*) en el que se soportan una gran variedad de lenguajes de programación. A lo largo de su historia, el compilador GCC se ha ido llevando a una amplia variedad de arquitecturas de procesador, siendo esta adaptabilidad un motivo más por el que emplear este tipo de compilador.

¿Por qué llevar a cabo una labor de **compilación cruzada**? Como se ha dicho, es necesario realizar estas tareas a menos que se vaya a desplegar la aplicación en el propio sistema operativo en el que se está desarrollando. En el caso de este proyecto, se ha podido obtener, por ejemplo, un código ejecutable **diferente** para sistemas *Mac OS* o *iOS* a través de una **misma** plataforma *Mac OS* y de un **mismo** entorno de desarrollo *Qt*, al igual que se ha podido obtener un código ejecutable diferente para sistemas *Linux* y *Android* a través de un sistema operativo *Ubuntu*.

En la ilustración de más abajo se puede ver la ejecución del juego desarrollado en diferentes plataformas, lo que demuestra que se pudo obtener un código ejecutable para cada una de ellas. De entre las plataformas se puede ver *Windows* (imagen de fondo), *Android*, *Mac OS X*, *Sailfish* (imagen del emulador de *Sailfish*), *Ubuntu* y, finalmente, *iOS* (imagen del emulador de *iPhone*).

Ilustración 32 – Ejecución del juego desarrollado en distintas plataformas.

7. ANÁLISIS

La fase de análisis del proyecto comenzó, precisamente, con la realización de un análisis sobre juegos que ya se habían desarrollado de una manera similar a la que íbamos a emplear para nuestro juego. Esa manera consistía en utilizar la tecnología *Qt*, haciendo uso de los lenguajes *Qml* y *JavaScript*.

Para llevar esto a cabo, vimos primero lo que tenía que ver, en cuanto a código se refiere, con la interfaz y los diferentes diálogos que podían tener los juegos que estábamos analizando. Para ello vimos los diferentes archivos "*.Qml*", y pudimos ir aprendiendo así la estructura y semántica de este lenguaje de diseño de interfaces. En esos momentos, fue muy útil la documentación que se recoge sobre el lenguaje *Qml* en el sitio oficial de *Qt*. En este lugar se explica más detenidamente y a modo de ejemplos, los diferentes componentes u objetos de los que consta el lenguaje *Qml*, algo que nos fue de gran utilidad.

Por otro parte, estaba la lógica de los juegos que se estaban analizando. Esta parte, sin embargo, no se analizó tan detenidamente ya que teníamos una cierta experiencia previa con el lenguaje *JavaScript*, y además consideramos que la parte de la lógica es algo que varía mucho dependiendo del juego del que se esté hablando y de cómo sea este, así que decidimos no perder tanto tiempo con este apartado.

Entonces, una vez analizados los aspectos de interfaz y de lógica de un par de juegos que nos sirvieron como ejemplos, procedimos a pensar en cómo podía ser el juego que íbamos a desarrollar para nuestro proyecto. Decidimos que íbamos a implementar un juego que fuera tanto tradicional como popular, que resultara entretenido. Fue entonces cuando pensamos que podíamos implementar el juego tradicional de las chapas de fútbol. Lo llamaríamos "FutChapas".

7.1 Interfaz

No pensamos en ningún momento en diseñar una interfaz que resultara compleja y difícil de entender para el usuario, teniendo en cuenta que estábamos hablando de un juego tradicional y popular. Simplemente pensamos que podía haber una **pantalla principal**, con una música para animar el juego, que diera acceso a este a través de un botón, por ejemplo, y otra **pantalla de juego** en la que se colocaría el terreno de juego, se distribuirían las chapas obteniendo una posición inicial, y además se añadiría la presencia de un marcador de goles. A pesar de que la interfaz no resultara compleja, no quiere decir que no pensásemos en hacerla atractiva para el usuario.

7.2 Lógica

Cuando decidimos que el juego que íbamos a desarrollar sería el de las chapas de fútbol, pensamos inmediatamente que el juego sería un juego **multijugador** para dos personas, y que estas jugarían a través de un **único** dispositivo móvil. Así, para llevar a cabo esto, pensamos en establecer **turnos** de juego de manera que cuando una persona ha terminado de lanzar una chapa con su equipo, el turno pasa al siguiente equipo de manera que la otra persona es la poseedora del turno actual y podrá lanzar la chapa que elija. Hay libertad a la hora de elegir la chapa que se desea lanzar en el momento en el que la persona es la propietaria del turno.

Al seleccionar una chapa, pensamos que podía aparecer un diálogo en el que se indicara hacia dónde quería dirigir esa chapa el usuario. Una vez seleccionados esos valores, la chapa comenzaría a moverse hasta que, o bien se chocara con otra chapa, o bien haya marcado un gol. En caso de que se haya chocado con otra chapa, la chapa con la que se chocó comenzaría a moverse. En caso de que la chapa se haya metido en la portería rival, es decir, haya metido un gol, se reproduciría un sonido de gol y las chapas, todas, volverían a su posición inicial. Como se ve, no se ha dicho en ningún momento que haya presencia de balón, sino que son las propias chapas las que protagonizan los goles, tal y como ocurre en el juego original de las chapas. Además, al meter un gol se añadiría este al marcador de juego de su equipo y, en caso de que ese sea el tercer gol para el equipo, se indicaría al usuario, por medio de un diálogo, si desea empezar una nueva partida o no. Asimismo, el usuario podría abandonar la partida actual sin tener que acabar el partido, para ello puede pulsar una flecha de vuelta atrás o similar que aparecería en la pantalla de juego.

En caso de que las chapas se chocaran con los límites del campo, no pararían su recorrido, sino que simplemente lo modificarían según como sea el choque, es decir, teniendo en cuenta la dirección de las chapas.

8. DISEÑO

Una vez que teníamos más o menos determinada cómo iba a ser la dinámica del juego, pensamos que teníamos que continuar por la fase de diseño. En caso de que algo de lo que habíamos planteado en la lógica del juego cambiara de manera sustancial, la fase de diseño se volvería a realizar con los cambios pertinentes. Para esta fase empleamos una serie de bocetos (*mockup*) con los que poder definir cómo sería el aspecto de la interfaz, haciendo uso de una herramienta *online* conocida como *Balsamiq mockup*, la cual se puede usar de manera gratuita (http://webdemo.balsamiq.com/).

Un aspecto ya a destacar y que formará parte tanto de la pantalla de inicio como de la pantalla de juego, es el que tiene que ver con la disposición de los elementos en la interfaz. Decidimos que la interfaz de nuestro juego, sea cual fuera el dispositivo en el que se estuviera jugando, tendría una orientación **horizontal**, y esto sería algo permanente aunque el usuario cambiara la orientación de su dispositivo. Esto lo decidimos así porque creíamos que era lo más lógico, ya que estábamos hablando de un juego de fútbol. Al tratarse de un juego de fútbol, donde un campo de fútbol es mucho más ancho que largo, pensamos que lo más cómodo para el usuario sería jugar con su móvil dispuesto horizontalmente. También pensamos que existen hoy en día juegos para dispositivos móviles que tienen también una orientación horizontal fija, como ocurre con *Angry Birds*, uno de los juegos más descargados de entre las aplicaciones móviles, y pensamos por lo tanto que esto no supondría un gran problema para el usuario, sino más bien una comodidad.

8.1 Pantalla de inicio

En primer lugar, decidimos comenzar con el diseño de la **pantalla de inicio**. Sin entrar en más detalles, comenzamos a usar la herramienta de bocetos antes mencionada y decidimos que el aspecto de la pantalla de inicio sería algo parecido a lo que sigue aquí:

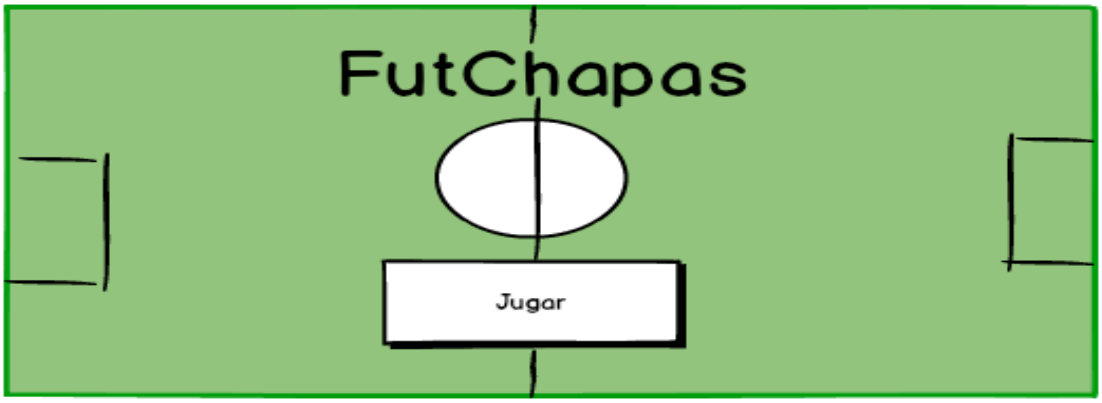


Ilustración 33 – Boceto de la pantalla de inicio del juego.

Para el diseño de esta pantalla de inicio también hay que destacar una serie de aspectos. Por un lado, se hizo uso del programa *Adobe Illustrator CC* para realizar un diseño del título "FutChapas", intentando que resultara más atractivo. No éramos grandes conocedores de esta herramienta pero resultó más o menos intuitiva a la hora de usarla y nos permitió hacer un diseño de la primera pantalla del juego mucho más atractivo. Como se puede ver en la ilustración 30, se añade una figura en sombra negra intentando representar a un jugador de fútbol.

Ilustración 34 – Diseño de la pantalla de inicio del juego con la herramienta Adobe Illustrator.

Como se puede ver, la pantalla de inicio consiste, fundamentalmente, en un conjunto de imágenes que forman parte de una misma pantalla. Por un lado, están las imágenes de dos porterías de fútbol. Por otro lado, existe otra imagen que es la que se corresponde a un jugador de fútbol, la cual viene dada en forma de sombra o algo similar. Sin embargo, en lo que respecta al título del juego, "FutChapas", se hizo uso de herramientas de texto que están incorporadas en la herramienta *Adobe Illustrator*. Esto permitió crear un título para la pantalla de inicio que fuera más personalizado, variando aspectos que tenían que ver con el posicionamiento, la opacidad, el tamaño y el trazo del texto.

Por otro lado, otra parte a destacar en el diseño de esta pantalla de inicio, es la que tiene que ver con el botón de "Jugar", el cual nos permite movernos hacia la pantalla de juego tras su pulsación. En este caso, quisimos hacer un botón que tuviera algo de atractivo para el usuario y que no resultara ser un botón común y corriente. De este modo, hicimos uso de una herramienta que también es *online* (http://buttonoptimizer.com/) que permite crear botones y que, además, permite descargar esos botones que creamos como imágenes ".png". El resultado fue un botón que era llamativo, donde había un texto con un icono de un balón de fútbol.

Ilustración 35 – Diseño del botón de la pantalla de inicio.

Esta imagen que representaba el botón de juego, simplemente se tenía que añadir junto con lo que habíamos obtenido anteriormente con *Adobe Illustrator*: el título del juego, la figura del jugador, y el terreno de juego de fondo. Con esto, dábamos, para entonces, por finalizado el diseño de la pantalla de inicio del juego.

8.2 Pantalla de juego

En segundo lugar, pasamos a comenzar a diseñar la pantalla en la que tendría lugar la mayor parte de la acción del juego. Sería en la pantalla de juego en donde los usuarios participantes podrían jugar las partidas, mover las chapas y marcar los goles. Teniendo en cuenta todo esto, pensamos que los elementos necesarios para conformar esta pantalla serían: el **terreno** de juego junto con las **porterías**, constituyendo así el fondo de esta pantalla, las **chapas** de fútbol de los dos equipos, las cuales se distribuirían a lo largo del terreno de juego, una imagen representando una **flecha** que permitiera abandonar la partida cuando se desee, y, finalmente, un **marcador** para ambos equipos junto con los **escudos** de estos. Más adelante, podrían añadirse nuevos elementos sobre la interfaz de juego como podía ser, por ejemplo, un **reloj** indicando el tiempo de juego que ha transcurrido.

Una vez más, para el diseño de las chapas de fútbol de un equipo y de otro, se volvió a hacer uso de la herramienta de diseño gráfico *Adobe Illustrator*. El trabajo, en esta ocasión, consistió, en primer lugar, en obtener una imagen, a través de internet por ejemplo, que fuera una chapa. Una vez obtenida la chapa que nos parecía más ideal para el juego por su forma y aspecto, comenzamos a hacer uso de la herramienta antes citada para personalizarla, de manera que pudimos obtener el diseño de dos chapas distintas para los dos equipos. En este caso, decidimos elegir dos equipos bastante conocidos como lo son el fútbol club Barcelona y el Real Madrid club de fútbol. Para personalizar estas chapas, primero tuvimos que recortar el área de la chapa original que no deseábamos, ya que contenía un título y una imagen que no era representativo de los equipos de fútbol. Una vez hechos estos recortes, simplemente se siguieron personalizando las chapas pintándolas, teniendo en cuenta los colores de un equipo y de otro. Véase la ilustración 33 en la que se puede ver claramente la evolución desde la chapa original, obteniendo así, gracias a *Adobe Illustrator*, las chapas de ambos equipos personalizadas.

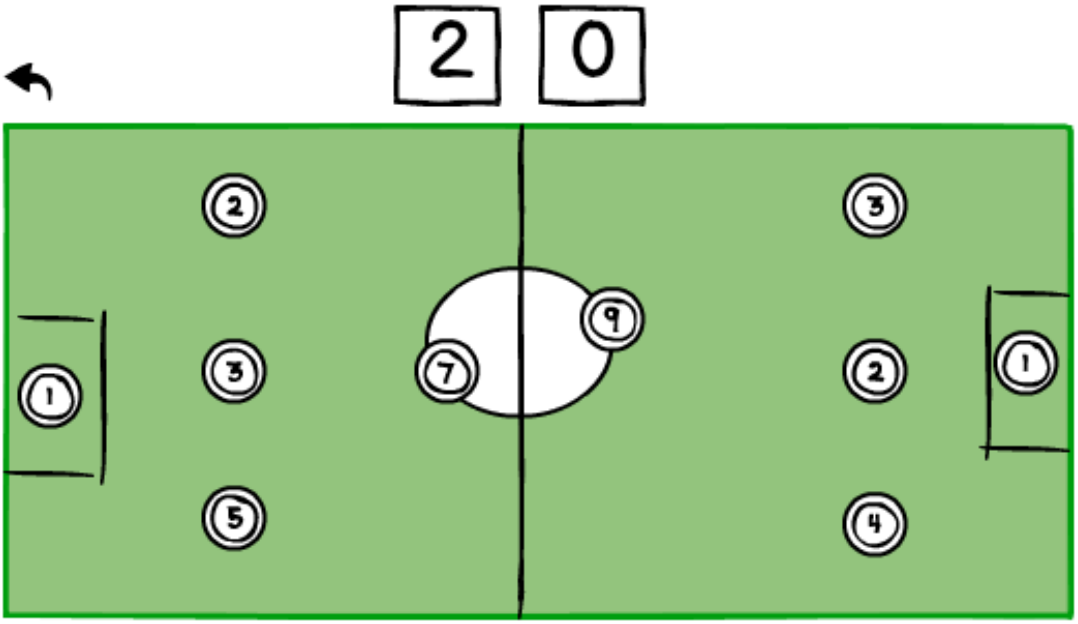


Ilustración 36 – Diseño de la pantalla de juego.

Ilustración 37 – Diseño de las chapas de juego a partir de una chapa original.

Por otra parte, también fue necesario conseguir imágenes de los escudos de los dos equipos de fútbol que habíamos elegido. Estas dos imágenes, aunque no se vean reflejadas en el boceto de la pantalla de juego antes ilustrado, se colocarían al costado de un marcador y otro, según el equipo. También para la flecha de vuelta atrás se hizo uso de una nueva imagen.

Ilustración 38 – Escudos de ambos equipos.

Ilustración 39 – Flecha de vuelta atrás para abandonar una partida.

Con todo esto, lo siguiente que tuvimos que pensar en la fase de diseño fue cómo capturar datos de los usuarios una vez que pulsaban una chapa, de manera que estos pudiesen indicar hacia dónde dirigir la chapa. Pensamos que lo más cómodo para el usuario sería que, tras la pulsación de una chapa, apareciera un diálogo en el que de una manera más o menos intuitiva pudiera indicar la dirección de la chapa en términos de altura (movimiento en eje *y*) y de profundidad (movimiento en eje *x*).

8.2.1 Diálogo tras la pulsación de una chapa

Para que el usuario pudiera indicar hacia dónde quería, aproximadamente, que se dirigiera la chapa, pensamos que podía haber un diálogo en el que a través de dos barras, una horizontal y otra vertical, se pudiera indicar un movimiento en eje *x*, a través de la barra horizontal, y otro en eje *y*, por medio de la barra vertical. Esto permitiría al usuario conseguir una gran libertad de movimiento, es decir, podría indicar, pongamos el caso, una gran profundidad a nivel horizontal, cuando por ejemplo quiere mover una chapa mucho a la derecha, pero podría también indicar, al mismo tiempo, que la chapa se moviera poco en altura, respecto a su posición actual.

De esta manera, la barra horizontal y también la vertical estarían divididas en dos mitades. Así, la mitad izquierda de la barra horizontal le permitiría indicar al usuario una mayor o menor profundidad de movimiento hacia la izquierda, según qué parte de la barra eligiese el usuario. Lo mismo ocurriría con la parte derecha de la barra horizontal, siendo en este caso un movimiento a derechas. Con la barra vertical ocurriría algo parecido. La parte superior de la barra vertical permitiría indicar una mayor o

menor altura o movimiento hacia arriba de la chapa. Mientras que la mitad de abajo permitiría al usuario indicar una mayor o menor profundidad de movimiento hacia abajo. Todo esto se ve mejor con la siguiente ilustración, donde se refleja el boceto del diálogo que aparece al pulsar una chapa.

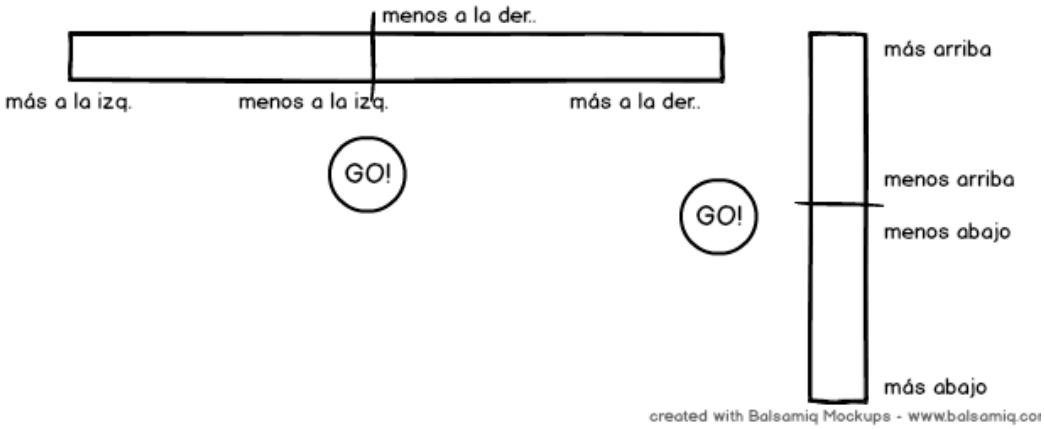


Ilustración 40 – Diálogo que aparecería al pulsar una chapa.

Como se puede ver en el diálogo, las barras horizontal y vertical son acompañadas por dos botones que tienen un texto de "GO!". Estos botones se han diseñado de la misma manera con la que diseñamos el botón de "Jugar" de la pantalla de inicio. El resultado que se obtuvo puede verse más detalladamente en la ilustración 37. La finalidad de estos botones es simple. Para cada una de las barras habrá una **animación** que consistirá en un eje que se irá moviendo a lo largo de cada una de ellas, rebotando cuando este llegue a alguno de los límites de cada una de las barras. De esta manera, el eje es lo que indicará al usuario en todo momento a qué dirección y a qué profundidad quiere mandar la chapa. Cuando el usuario percibe que el eje se encuentra en la posición que desea, pulsará el botón "GO!" de una de las barras, para, seguidamente, ir a la otra barra y pulsar el otro botón cuando el otro eje se encuentre donde el usuario desea. Así, una vez que los dos botones "GO!" han sido pulsados se habrán captado datos que tiene que ver con la dirección a la que el usuario quiere mandar la chapa que ha elegido. La animación que tiene lugar en este diálogo se comenta más detalladamente en el apartado de "Implementación".

Ilustración 41 – Aspecto del botón "GO!" en el diálogo de la chapa.

9. IMPLEMENTACIÓN

En la fase de implementación comenzaban las labores de programación del juego de chapas de fútbol que habíamos decidido realizar y que ya habíamos analizado detenidamente. Para llevar a cabo la implementación del proyecto, se hizo uso de los entornos de programación *Qt creator* que ya se han comentado en los apartados de "*Android*" y "*Sailfish*". Es cierto que el proyecto iba a estar destinado a más de un sistema operativo, pero el hecho de implementar el código del proyecto en un entorno u otro, sea el *Qt for Android*, el *SailfishOS IDE*, o los entornos *Qt* para *Windows*, *Mac* o *Linux*, no era determinante, así que podíamos emplear cualquiera de estos entornos para comenzar la implementación. El hecho de utilizar más de un entorno, sí sería determinante para poder crear el código ejecutable para cada sistema operativo al que iba dirigido el proyecto, es decir, para llevar a cabo las labores de **compilación cruzada**.

Sin más, a través de cualquiera de los entornos *Qt creator* de programación pudimos **crear el proyecto** de la siguiente manera:

- Menú "File" > "New File or Project…"
- En el siguiente diálogo seleccionar *Qt Quick Application*. Pulsar "Choose…" (véase la ilustración 38).
- Asignamos, en la siguiente ventana, un nombre y un directorio de creación para el proyecto. En nuestro caso el nombre iba a ser: "FutChapas"
- Hay que seleccionar los kit de construcción y de ejecución necesarios para la aplicación. Estos kit ya vienen incorporados automáticamente cuando instalamos los entornos de programación *Qt* que se usan. En la ilustración 39 se puede ver, a modo de ejemplo, cómo se seleccionan los kit para poder crear el código ejecutable en el caso de sistemas *Android*. Estos kit, como se ha dicho, son los que vendrán incorporados en la versión de *Qt for Android* que ya habremos descargado e instalado. Para otros sistemas, los kit son diferentes.
- Pulsar "Finish".

Una vez creado el proyecto, se crea una estructura de archivos en el entorno de programación. La estructura generada se puede apreciar en la ilustración 40.

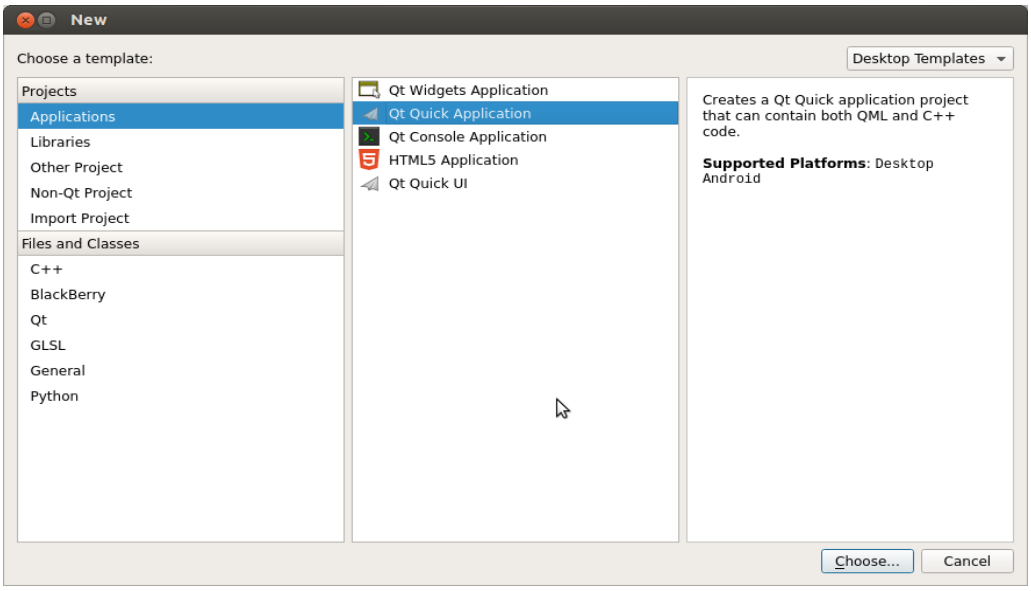


Ilustración 42 – Creación del proyecto como Qt Quick Application.

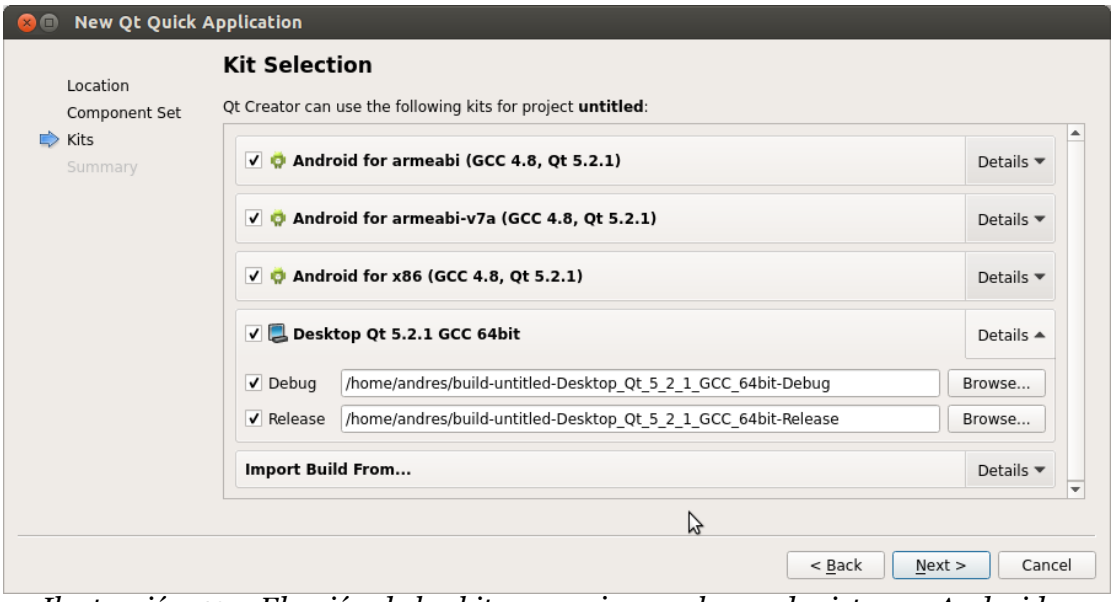


Ilustración 43 – Elección de los kit necesarios en el caso de sistemas Android.

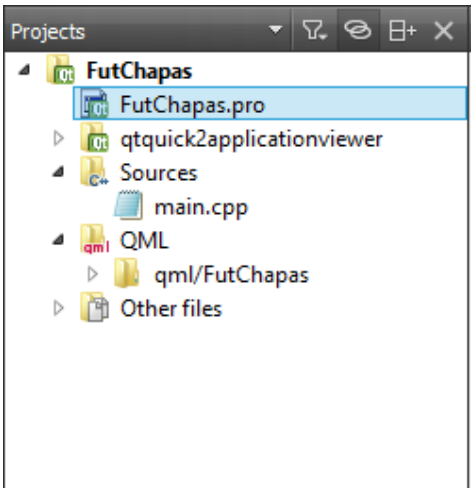


Ilustración 44 – Estructura de directorios originada al crear el proyecto.

9.1 Abrir un proyecto ya creado

Tras lo explicado en el apartado anterior, se concluye que el proyecto es necesario **crearlo** una única vez, es decir, no se tiene que crear un proyecto nuevo cuando se use otro entorno *Qt* que no fuera el que estaba destinado para dispositivos *Android*, por ejemplo. Esto quiere decir que si ahora queremos seguir implementando el proyecto usando, por ejemplo, el entorno *SailfisOS IDE* no hay ningún problema, simplemente tenemos que **abrir** un proyecto ya creado. Para ello basta con hacer lo siguiente en el entorno que se desee:

- Ir al menú "File" > "Open File or Project…"

- Buscamos el directorio en el que se encuentra el proyecto y tenemos que abrir el fichero que se llama igual que nuestro proyecto y que tiene una extensión ".pro".

- Tras esto, aparecerá un diálogo en el que se nos obliga a indicar los kit de construcción y de ejecución que emplearemos para el proyecto en el nuevo entorno. En la ilustración 41, se puede ver cómo se indican y seleccionan en el entorno *SailfishOS IDE* los kit necesarios para el sistema *Sailfish*, los cuales fueron añadidos automáticamente cuando se instaló el entorno.

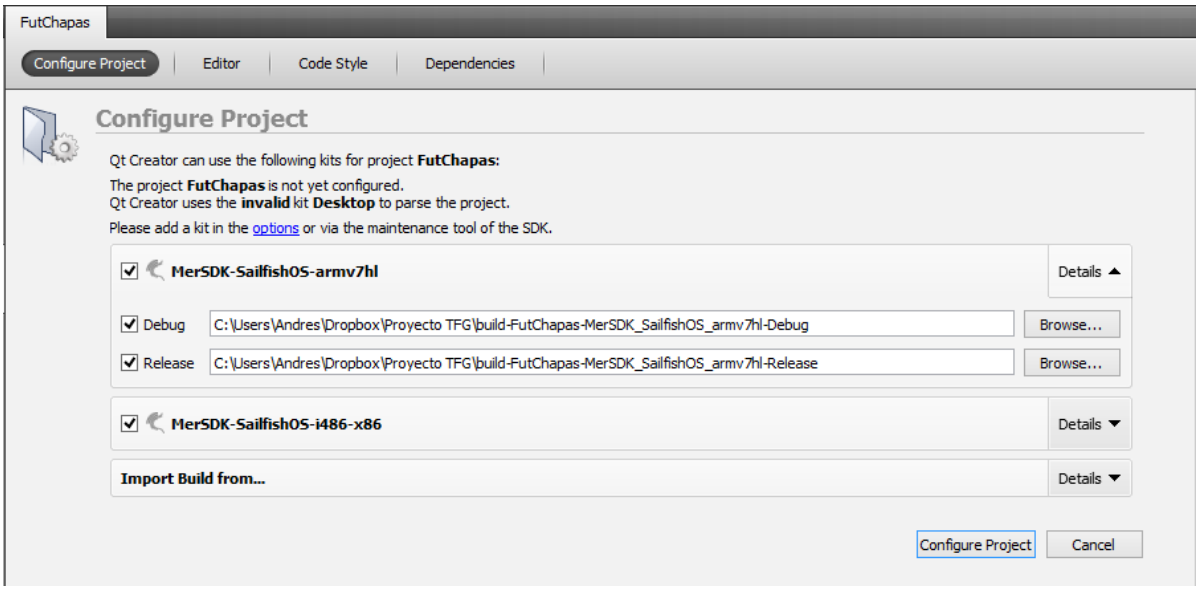


Ilustración 45 – Configuración de un proyecto ya creado en un entorno diferente del que se creó.

Antes de entrar en detalles de implementación que tienen que ver con la lógica y el diseño de la interfaz del juego, es necesario comentar una serie de archivos aparte que se encuentran inicialmente al crear un proyecto y que también se ven al abrirlo.

- Archivo ".pro", en nuestro caso **FutChapas.pro**: este tipo de archivos contiene toda la información necesaria para poder construir la aplicación. Los **recursos** que nuestra aplicación utilice, como puedan ser imágenes o sonidos, también se indican en este archivo, usando una serie de declaraciones. Estas declaraciones se construyen a través de variables, las cuales almacenan listas de *Strings*. En su forma más simple, las variables almacenan información sobre aspectos de configuración, o indican nombres de archivos o de directorios. Las variables que se pueden apreciar en nuestro fichero ".pro" son:
  - *QT*: para especificar configuraciones propias de *Qt*.
  - SOURCES: para listar archivos fuente que se emplearán al construir la aplicación.
  - OTHER_FILES: para indicar recursos externos que se emplean en el proyecto.

```
# Add more folders to ship with the application, here
folder_01.source = Qml/FutChapas
folder_01.target = Qml
DEPLOYMENTFOLDERS = folder_01
QT += multimedia

# Additional import path used to resolve QML modules in Creator's
codemodel
QML_IMPORT_PATH =

# The .cpp file which was generated for your project. Feel free to
hack it.
SOURCES += main.cpp

# Installation path
# target.path =

# Please do not modify the following two lines. Required for
deployment.
include(Qtquick2applicationviewer/Qtquick2applicationviewer.pri)
QtcAddDeployment()

OTHER_FILES += \
    Qml/FutChapas/content/arco.png \
    Qml/FutChapas/content/btn_jugar.png \
    Qml/FutChapas/content/chapa_barcelona.png \
    Qml/FutChapas/content/chapa_madrid.png \
    Qml/FutChapas/content/HomeScreen.png \
    Qml/FutChapas/content/sound_gol.wav \
    Qml/FutChapas/content/sound_home.png \
    Android/AndroidManifest.xml

ANDROID_PACKAGE_SOURCE_DIR = $$PWD/Android
```

Fichero ".pro" de nuestro proyecto

Como se puede ver en el archivo, a través de la variable "*QT*" se añade una librería **multimedia**, la cual hace falta si se quieren reproducir sonidos en dispositivos reales. Además, con la variable "OTHER_FILES" se indican los archivos o recursos que se han empleado en la aplicación como puedan ser imágenes o sonidos.

- Archivo **main.cpp**: se autogenera al crear el proyecto, como el archivo anterior. En él se indica el archivo "*.Qml*" con el que arrancar y visualizar la interfaz de nuestra aplicación, siendo en este caso el archivo "main.qml".

```
#include <QtGui/QGuiApplication>
#include "Qtquick2applicationviewer.h"

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);

    QtQuick2ApplicationViewer viewer;

viewer.setMainQmlFile(QStringLiteral("Qml/FutChapas/main.Qml"));
viewer.showExpanded();
    return app.exec();
}
```

Fichero "main.cpp" de nuestro proyecto.

9.2 Kit de construcción y ejecución en Qt creator (build & run)

A lo largo de los apartados anteriores, se han nombrado en varias ocasiones los kit de construcción y ejecución de aplicaciones que vienen incorporados con los diferentes entornos *Qt* de programación. Por medio de los kit, la herramienta *Qt creator* agrupa configuraciones y ajustes de construcción y ejecución de aplicaciones, facilitando el desarrollo **multiplataforma**. Cada kit consiste en un conjunto de valores que definen un **entorno**, pudiendo ser un entorno un dispositivo móvil, un escritorio en un sistema operativo, una máquina virtual, etc.

Así, en nuestro caso, donde definimos un juego para varios sistemas operativos (*Android*, *Sailfish*, *iOS*, *Windows*, Mac, *Linux*) fueron necesarios varios kit. Estos kit, como ya se ha dicho anteriormente, ya vienen ajustados por la herramienta *Qt creator*, así que nosotros simplemente tenemos que elegir los kit que consideremos teniendo en cuenta la plataforma para la que queremos que se ejecute el juego.

Para poder desplegar nuestro proyecto en sistemas operativos de escritorio como *Windows*, *Linux* o *Mac OS*, simplemente tuvimos que descargar la versión de *Qt creator* 5.2.1, a través de la página oficial de *Qt*, en cada uno de los sistemas. Una vez hecho esto, el entorno *Qt* en cada sistema ya dispone de los kit necesarios para la construcción y ejecución de nuestro proyecto, así que simplemente teníamos que abrir el proyecto que estaba ya creado. Para cada sistema de escritorio habrá un kit de escritorio (*Desktop*) que permite crear un ejecutable de la aplicación que se desarrolla.

Para sistemas operativos móviles, como *Android*, *Sailfish* o *iOS*, se hizo uso de los entornos comentados en los apartados de "*Android*", "*Sailfish*" y, en el caso de *iOS*, el entorno *Qt* que está orientado para sistemas *Mac*. En el caso de *Android*, la versión de

Qt for Android trae incorporados tres kit que permiten la ejecución en dispositivos reales. La elección del kit depende de la arquitectura del dispositivo sobre el que ejecutemos la aplicación. Esto se ve con más detalle en el apartado de "Pruebas". En el caso de *Sailfish*, por su parte, se cuenta con un kit que permite obtener un archivo ejecutable con el que desplegar el juego en un emulador de *Sailfish* que se configuró, en su momento, como una máquina virtual en un entorno *Virtual Box Oracle*. Mientras que para *iOS*, se puede emplear un kit que incorpora el entorno *Qt* de sistemas *Mac* con el que se genera un código ejecutable que se puede testear en el emulador de dispositivos *iOS* del entorno *XCode*.

9.3 Implementación de la interfaz

Para definir los diferentes elementos u objetos de nuestra interfaz, hemos implementado, concretamente, cinco ficheros de extensión ".*Qml*". En estos ficheros se definen los componentes *Qml* que consideramos necesarios teniendo en cuenta lo que habíamos especificado en el apartado de "Diseño". A lo largo de estos ficheros, se hace uso de diferentes recursos externos como imágenes o sonidos que fueron añadidos a un directorio llamado "content" dentro del directorio "*Qml*", siendo este último creado al empezar un nuevo proyecto.

9.3.1 main.Qml

Este archivo es el principal en lo que respecta a la definición de la interfaz del juego. En él, se definen **todos** los componentes que se pueden visualizar tanto en la pantalla de inicio del juego como en la pantalla de juego: las chapas, el terreno de juego con las porterías, los marcadores, un reloj indicando el tiempo de juego, los escudos, la flecha de abandonar partida y el botón de jugar. Las primeras líneas de la implementación de este archivo están compuestas por unas sentencias de importación de librerías y de archivos externos que son necesarios para el funcionamiento de la aplicación.

```
import QtQuick 2.0
import "content"
import "content/logica.js" as Logic
```

Como se ve, es necesario hacer una importación del *framework QTQuick 2.0*, el cual incluye una serie de librerías, ya que en caso contrario no se podrían utilizar los diferentes componentes que se definen a través del lenguaje *Qml* como puedan ser *Rectangle, Image, SoundEffect*, etc. Además, con las otras dos líneas se incluye un directorio "content", en el que se almacenan diferentes recursos externos como imágenes o sonidos, y un archivo "logica.js", el cual constituye la lógica de la aplicación. Es necesario incluir este archivo para poder **invocar** desde los archivos *Qml* las funciones que residen en él.

A continuación, comienza verdaderamente la definición de los componentes que el usuario visualizará en la interfaz del juego. Así, el primer elemento que se aprecia en la implementación es un componente de tipo *Row*. Este componente contendrá **todos** los restantes elementos que se encuentran en este archivo. Será, pues, el elemento raíz.

```
Row {
    id:mainbar
    property Row mainbar: mainbar
    property bool gameRunning: false
    width: newGameScreen.width
    height: newGameScreen.height

    property var gameState: Logic.newGameState(mainbar);

    property Image chapa2: chapa2
    property Image chapa1: chapa1
    property Image chapa3: chapa3
    property Image chapa4: chapa4
    property Image chapa5: chapa5
    property Image chapa8: chapa8
    property Image chapa9: chapa9
    property Image chapa10: chapa10
    property Image chapa11: chapa11
    property Image chapa12: chapa12

    property Image imgGameOn: imgGameOn

    property Text txtMarcador1: txtMarcador1
    property Text txtMarcador2: txtMarcador2

    property Text txtCrono: txtCrono

    property SoundEffect sound_gol: sound_gol
    …
    …
```

En este componente, lo más destacable reside en la definición de una serie de propiedades *property*. A partir de las *property*, se pueden referenciar los componentes que deseemos del archivo "main.qml" y, de esta manera, se pueden **consultar** o **modificar** los valores de las propiedades de estos componentes. También se puede ver otra sentencia destacable en este objeto, y es aquella en la que se incrusta código *JavaScript*, algo que veíamos que era posible con la tecnología *Qt*. Con ella, estamos llamando a una función "newGameState()" de nuestro archivo "logica.js" para inicializar las variables del juego.

Los elementos que siguen, por tanto, se encuentran incluidos dentro del componente *Row*. Esto lo hemos hecho así porque hemos incluido tanto la pantalla de inicio como la pantalla de juego en el mismo archivo "main.qml". Es decir, no se tienen dos ventanas diferentes para cada una de las pantallas. Al hacerlo así, pasaremos de una pantalla a otra a través de una animación, la cual se comenta más adelante, que tiene lugar en la misma fila o *Row*. El primero de los elementos que sigue es un elemento de

tipo *Item*, y en él se define la pantalla de inicio en su totalidad, incluyendo los componentes que sean necesarios atendiendo a lo definido en el diseño.

```
Item {
        id: newGameScreen
        width:800
        height:470

        Image {
            source: "content/HomeScreen.png"
            anchors.fill: parent
        }

        Image {
            source: "content/btn_jugar.png"
            anchors.bottom: parent.bottom
            anchors.bottomMargin: 60
            anchors.horizontalCenter: parent.horizontalCenter
            MouseArea {
                anchors.fill: parent
                onClicked: {
                    sound_home.stop()
                    mainbar.state = "stateGameBegins"
                }
            }
        }
        SoundEffect {
            id: sound_home
            source: "content/sound_home.wav"
        }

        SoundEffect {
            id: sound_gol
            source: "content/sound_gol.wav"
            loops: 1
        }

        Timer {
            interval: 1
            running: true
            onTriggered: sound_home.play()
        }
        //SIGUE
```

Este elemento *Item* tiene definido un ancho y alto a través de sus propiedades *width* y *height*. Los primeros elementos que contiene son dos imágenes. La primera de ellas, es la imagen de un campo de fútbol con la figura de un jugador que se obtuvo como resultado del trabajo realizado en *Adobe Illustrator*. Como se ve, esta imagen ha sido acompañada de otra, siendo esta otra el botón de jugar que en su momento se diseñó. Seguidamente hay dos componentes de sonido *SoundEffect*, uno de ellos representando un sonido que se reproducirá en la pantalla de inicio al comenzar el juego y al volver a esta pantalla, y el otro representando al sonido de un gol, cuando este tenga lugar. Lo que controlará que estos sonidos se reproduzcan o no será la lógica del juego. Sin embargo, gracias a la presencia de un componente *Timer* se consigue que al inicio del juego el sonido de la pantalla de inicio ya se reproduzca. Este sonido de la pantalla de

inicio se parará de forma inmediata cuando el usuario pulse el botón de jugar. Esto, fijándonos detenidamente, se ha implementado haciendo uso de un componente *MouseArea* alrededor del botón, de forma que cuando se registre un evento de tipo *click* sobre dicho botón, se parará el sonido y se cambiará el estado (*state*) de la fila, moviéndonos a la pantalla de juego. La definición de esta pantalla de inicio sigue a través de la inclusión del marcador de goles y de los escudos de ambos equipos.

```
        //MARCADOR 1
        Rectangle{
            id: marcador1
            y: 0
            x: newGameScreen.width*1.43
            width: 50
            height: 50
            color: "red"
            border.color: "black"
            border.width: 5
            Text {
                id: txtMarcador1
                text: "0"
                font.bold: true
                font.family: "Helvetica"
                font.pointSize: 32
                verticalAlignment: Text.AlignVCenter
                style: Text.Outline
                styleColor: "#AAAAAA"
                x: 12
            }
        }

        //MARCADOR 2
        Rectangle{
            id: marcador2
            y: 0
            x: newGameScreen.width*1.51
            width: 50
            height: 50
            color: "red"
            border.color: "black"
            border.width: 5

            Text {
                id: txtMarcador2
                text: "0"
                font.bold: true
                font.family: "Helvetica"
                font.pointSize: 32
                verticalAlignment: Text.AlignVCenter
                style: Text.Outline
                styleColor: "#AAAAAA"
                x: 12
            }
        }
        //SIGUE
```

Como se ve en estas líneas, los marcadores se han definido como objetos tipo *Rectangle* que incluyen componentes de tipo *Text*. A través de las propiedades *x* e *y*, se

posicionaron los elementos hasta que quedaron de una forma muy parecida a la que se había planificado en la fase de diseño. Al texto que indica los marcadores de ambos equipos, se le ha dado algo de formato a través de las propiedades *bold*, *family*, *pointSize*. Inicialmente, el marcador de ambos equipos, como es obvio, será de cero, y esto se indica a través de la propiedad *text*.

```
        //ESCUDOS
         Image{
             id: escudo1
             source: "content/bcn.png"
             y: 0
             x: newGameScreen.width*1.33
             width: 50
             height: 50
         }
         Image{
             id: escudo2
             source:  "content/rea.png"
             y: 0
             x: newGameScreen.width*1.61
             width: 50
             height: 50
         }

        //CRONO
          Rectangle{
              id: crono
              y: 0
              x: newGameScreen.width*1.82
              width: 120
              height: 50
              color: "blue"
              border.color: "black"
              border.width: 5
              Text {
                  id: txtCrono
                  text: "00:00"
                  font.bold: true
                  font.family: "Helvetica"
                  font.pointSize: 28
                  verticalAlignment: Text.AlignVCenter
                  style: Text.Outline
                  styleColor: "#AAAAAA"
                  x: 12
              }
          }

         //FLECHA DE VUELTA ATRÁS
         Image {
             id:flecha
             x: newGameScreen.width*1.005
             y:(2*imgGameOn.height)/120
             width: 40
             height: 40

             source:"content/btn_flecha.png"
             MouseArea {
                 anchors.fill: flecha
```

```
                onClicked: {
        Qt.createComponent("goBackDialog.Qml").createObject(flecha);
                }
            }
        }
    }//FIN de la pantalla de inicio y del Item
```

Los escudos se definen como imágenes, tal y como se ve. A través de las propiedades que se ven en ellos se posicionan y se les da un tamaño adecuado. Se define también un crono o reloj, con el que contabilizar e indicar el tiempo de juego a los jugadores, de la misma forma con la que se definieron los marcadores. Por último, también se incluye como elemento principal de la ventana de inicio la flecha que permite abandonar una partida. Lo más destacable de este elemento es el componente *MouseArea* que se ha definido alrededor de él. Esto permite detectar eventos del *mouse* que tengan lugar en dicha flecha, en dicho componente. Concretamente, cuando tenga lugar un evento *click* sobre la flecha, aparecerá un diálogo sobre el usuario. Este diálogo se ha definido en el archivo "goBackDialog.*Qml*", y como se ve se crea una instancia de este gracias a una instrucción *JavaScript* que se ha insertado en el código *Qml*. Con todo esto, el resultado final de la ventana de inicio se puede ver en la siguiente ilustración, en la cual se aprecia esta pantalla cuando la ejecución tiene lugar en un sistema operativo *Windows*.

Ilustración 46 – Aspecto de la pantalla de inicio al ejecutar el juego en un sistema Windows.

Se puede apreciar en esta pantalla de inicio que, tanto los marcadores como los escudos de los equipos como la flecha de vuelta atrás no aparecen, y sin embargo sí estaban definidos dentro del *Item* de pantalla de inicio. Esto es así porque, como se comentó antes, se han implementado las pantallas en una única ventana a través de una fila *Row*. Con esto, lo que hicimos fue posicionar los elementos que aparecen en la pantalla de juego respecto a la pantalla de inicio, y es por eso que los escudos, los marcadores y la flecha de abandonar partida se encuentran definidos, en el código, dentro de la pantalla de inicio.

Tras implementar la pantalla de inicio fue necesario comenzar a implementar lo que constituiría la **pantalla de juego**, donde tendría lugar realmente la acción de la aplicación. En este caso, la implementación también comienza con la definición de un elemento que contendrá a otros veinte. Este elemento es la imagen que define el terreno de juego con sus porterías. Sobre esta imagen se colocarán las chapas que se distribuyen a lo largo del campo.

```
//PANTALLA DE JUEGO
    Image {
        id: imgGameOn
        source: "content/campo.png"
        height:0.87*(newGameScreen.height)
        width:newGameScreen.width
        y: 50
       //SIGUE
```

Como se ve, la pantalla de juego se ha hecho un poco menos grande, en cuanto a altura, que la pantalla de inicio. Esto se ha hecho así porque se ha querido dejar un espacio arriba en el que poder poner los marcadores, los escudos de los equipos y la flecha de vuelta atrás, y así no estorbar el terreno de juego. La implementación sigue a través de la inclusión de todas las chapas en la pantalla de juego, concretamente son diez chapas, cinco por cada equipo.

```
//CHAPAS DEL FÚTBOL CLUB BARCELONA
        Image {
            id: chapa2
            width: imgGameOn.width*0.05
            height: imgGameOn.height*0.1
            property int idEquipo: 1
            y: imgGameOn.height*0.45
            enabled: true
            x: imgGameOn.width*0.05
            source: "content/chapa_barcelona.png"

            MouseArea {
                anchors.fill: parent
                onClicked: {

Qt.createComponent("chapaDialog.Qml").createObject(imgGameOn);
                    Logic.chapaPulsada(chapa2)
                }
            }
        }
        Image {
            id: chapa1
            width: imgGameOn.width*0.05
            property int idEquipo: 1
            height: imgGameOn.height*0.1
            enabled: true
            y: imgGameOn.height*0.10
            x: imgGameOn.width*0.2
            source: "content/chapa_barcelona.png"

            MouseArea {
                anchors.fill: parent
                onClicked: {

Qt.createComponent("chapaDialog.Qml").createObject(imgGameOn);
                    Logic.chapaPulsada(chapa1)
                }
            }
        }
        //SIGUE
```

```
Image {
            id: chapa3
            width: imgGameOn.width*0.05
            property int idEquipo: 1
            height: imgGameOn.height*0.1
            enabled: true
            y: imgGameOn.height*0.45
            x: imgGameOn.width*0.2
            source: "content/chapa_barcelona.png"

            MouseArea {
                anchors.fill: parent
                onClicked: {

Qt.createComponent("chapaDialog.Qml").createObject(imgGameOn);
                    Logic.chapaPulsada(chapa3)
                }
            }
        }
        Image {
            id: chapa4
            width: imgGameOn.width*0.05
            property int idEquipo: 1
            height: imgGameOn.height*0.1
            enabled: true
            y: imgGameOn.height*0.75
            x: imgGameOn.width*0.2
            source: "content/chapa_barcelona.png"

            MouseArea {
                anchors.fill: parent
                onClicked: {

Qt.createComponent("chapaDialog.Qml").createObject(imgGameOn);
                    Logic.chapaPulsada(chapa4)
                }
            }
        }
        Image {
            id: chapa5
            width: imgGameOn.width*0.05
            property int idEquipo: 1
            height: imgGameOn.height*0.1
            enabled: true
            y: imgGameOn.height*0.45
            x: imgGameOn.width*0.35
            source: "content/chapa_barcelona.png"

            MouseArea {
                anchors.fill: parent
                onClicked: {

Qt.createComponent("chapaDialog.Qml").createObject(imgGameOn);
                    Logic.chapaPulsada(chapa5)
                }
            }
        }
        //FIN de chapas de Barcelona
```

```
//CHAPAS DEL REAL MADRID
        Image {
            id: chapa8
            width: imgGameOn.width*0.05
            property int idEquipo: 2
            height: imgGameOn.height*0.1
            enabled: false
            y: imgGameOn.height*0.45
            x: imgGameOn.width*0.9
            source: "content/chapa_madrid.png"

            MouseArea {
                anchors.fill: parent
                onClicked: {

Qt.createComponent("chapaDialog.Qml").createObject(imgGameOn);
                    Logic.chapaPulsada(chapa8)
                }
            }
        }
        Image {
            id: chapa9
            width: imgGameOn.width*0.05
            property int idEquipo: 2
            height: imgGameOn.height*0.1
            enabled: false
            y: imgGameOn.height*0.10
            x: imgGameOn.width*0.75
            source: "content/chapa_madrid.png"

            MouseArea {
                anchors.fill: parent
                onClicked: {

Qt.createComponent("chapaDialog.Qml").createObject(imgGameOn);
                    Logic.chapaPulsada(chapa9)
                }
            }
        }
        Image {
            id: chapa10
            width: imgGameOn.width*0.05
            property int idEquipo: 2
            height: imgGameOn.height*0.1
            enabled: false
            y: imgGameOn.height*0.45
            x: imgGameOn.width*0.75
            source: "content/chapa_madrid.png"

            MouseArea {
                anchors.fill: parent
                onClicked: {

Qt.createComponent("chapaDialog.Qml").createObject(imgGameOn);
                    Logic.chapaPulsada(chapa10)
                }
            }
        }
        //SIGUE
```

```
Image {
            id: chapa11
            width: imgGameOn.width*0.05
            property int idEquipo: 2
            height: imgGameOn.height*0.1
            enabled: false
            y: imgGameOn.height*0.75
            x: imgGameOn.width*0.75
            source: "content/chapa_madrid.png"

            MouseArea {
                anchors.fill: parent
                onClicked: {

Qt.createComponent("chapaDialog.Qml").createObject(imgGameOn);
                    Logic.chapaPulsada(chapa11)
                }
            }
        }
        Image {
            id: chapa12
            width: imgGameOn.width*0.05
            property int idEquipo: 2
            height: imgGameOn.height*0.1
            enabled: false
            y: imgGameOn.height*0.45
            x: imgGameOn.width*0.6
            source: "content/chapa_madrid.png"

            MouseArea {
                anchors.fill: parent
                onClicked: {

Qt.createComponent("chapaDialog.Qml").createObject(imgGameOn);
                    Logic.chapaPulsada(chapa12)
                }
            }
        } // FIN de las chapas de Real Madrid
    }//FIN de la pantalla de juego
```

Todas las chapas se han definido siguiendo una estructura común. Cada una tiene un identificador propio, lo que nos ha permitido desde la lógica acceder a las propiedades de estas chapas para poder modificarlas o consultarlas. Inicialmente, tienen una posición en el campo que es la indicada por sus propiedades *x* e *y*, las cuales irán cambiando conforme se muevan. Para cada una se ha definido una propiedad *idEquipo*, la cual se emplea en la lógica de la aplicación para detectar los goles, es decir, si una chapa se mete en su propia portería no debe contabilizar como gol, y esto es algo que implementamos haciendo uso de esta propiedad. Además, inicialmente las chapas que se pueden tocar o pulsar son las del fútbol club Barcelona, es decir que es este equipo el que tiene el turno inicial. Esto se ha implementado gracias a la propiedad *enabled* que tienen los componentes de tipo *Image*. Aunque el usuario pulsara las chapas del Real Madrid en primera instancia, nada ocurriría. Finalmente, cada una de las chapas consta también de un *MouseArea* que permite detectar cuándo esta ha sido pulsada. En efecto, cuando una chapa sea pulsada, lo cual sólo será posible si está habilitada (*enabled* tiene

que ser *true*), aparecerá un diálogo sobre el usuario que le permitirá indicar hacia dónde dirigir la chapa. Con todo esto, la implementación de la pantalla de juego tendría un aspecto como el que sigue, donde se ve la ejecución de la aplicación en un sistema *Ubuntu*.

Ilustración 47 – Aspecto de la pantalla de juego al ejecutar el juego en un sistema Ubuntu.

La implementación del archivo "main.qml" no acaba aquí. Es cierto que, hasta el momento, se tienen ya implementadas las pantallas de inicio y de juego de nuestro proyecto con sus respectivos componentes, pero hay otros componentes que también son necesarios para el funcionamiento del juego.

Sin duda, uno de los componentes que ha sido fundamental para el funcionamiento de la aplicación ha sido el de tipo *Timer*. Con este componente, como se verá a continuación, se han definido temporizadores que permiten ejecutar funciones de nuestro archivo de lógica (logica.js) cada cierto intervalo, de manera periódica.

```
    Timer {
        interval: 16
        running: true
        repeat: true
        onTriggered: Logic.tick()
    }

    Timer {
        interval: 16
        running: true
        repeat: true
        onTriggered: Logic.tickDetectarColisiones()
    }

    Timer {
        interval: 16
        running: true
        repeat: true
        onTriggered: Logic.tickDetectarGol()
    }

    Timer {
        interval: 16
        running: true
        repeat: true
        onTriggered: Logic.tickDetectarLimites()
    }

     Timer {
        interval: 1000
        running: true
        repeat: true
        onTriggered: Logic.actualizarCrono()
    }
```

De esta forma, cada dieciséis milisegundos se están invocando las funciones "tick()","tickDetectarColisiones()","tickDetectarGol()" y "tickDetectarLimites()" del archivo de lógica, mientras que la función "actualizarCrono()" se ejecuta cada un segundo, actualizando así el reloj que indica el tiempo de juego. La funcionalidad de cada una de ellas se explica más detalladamente en el apartado 8.4. Estas funciones, pues, se están ejecutando durante toda la aplicación, desde que arranca hasta que para de ejecutarse.

Por último, el archivo "main.qml" finaliza con la definición de una serie de objetos *State* y de uno de tipo *Transition*. Se definen **dos estados** para la aplicación a través de objetos *State*: el primero es el estado inicial de la aplicación, el que toma al arrancarla y al visualizarse la pantalla de inicio. Cuando el usuario pulsa el botón de jugar, dicho estado cambia y toma el valor "stateGameBeggins", en cuyo caso se deberá pasar a la pantalla de juego, algo que se consigue modificando el valor de la propiedad *x* tal y como se ve en el *script* que se define dentro de *PropertyChanges*. Además, en dicho *script* también se indica que el sonido de la pantalla de inicio debe parar de reproducirse, y, finalmente, se invoca a la función del archivo de lógica "startGame()".

Estos cambios de estados en los que se pasa de una pantalla a otra vienen acompañados de una **animación** en forma de **transición**. Así, con el objeto *Transition* se ha definido una animación que consiste en rebotar durante un segundo la pantalla que ha sido cambiada. Todo esto se puede ver mejor en el siguiente cuadro.

```
states: [
        State {
            name: "";
            PropertyChanges { target: mainbar; x: 0 }
            StateChangeScript { script : {
                    sound_home.play()
                    Logic.stopGame()
                }
            }
        },
        State {
            name: "stateGameBegins";
            PropertyChanges { target: mainbar; x: -
newGameScreen.width}
            StateChangeScript { script : {
                    sound_home.stop()
                    Logic.startGame()
                }
            }
        }
    ]
    transitions: Transition {
        NumberAnimation { properties: "x,y"; duration: 1000;
easing.type: Easing.OutBounce }
    }
}
//FIN del componente raíz Row y del archivo "main.Qml"
```

9.3.2 goBackDialog.Qml

A través de este fichero, pudimos definir el diálogo que visualizará el usuario en el caso de que haya pulsado la flecha de vuelta atrás sobre la pantalla de juego. Se trata de un diálogo **modal** en el que por medio de dos botones, aceptar y cancelar, el usuario podrá indicar si desea o no finalizar la partida, respectivamente. Al tratarse de un diálogo, fue necesario incluir en el proyecto una librería adicional *QtQuick.Dialogs 1.1*, de manera que las sentencias *import* de este archivo quedaban de la siguiente manera.

```
import QtQuick 2.0
import QtQuick.Dialogs 1.1
import "content/logica.js" as Logic
```

El archivo continúa con la definición de un único elemento: *MessageDialog*. Este elemento permite definir de una manera rápida y sencilla un diálogo que resulte

simple para el usuario. Consta de una serie de manejadores de eventos (*handlers*) que permiten especificar qué hacer en caso de que se haya pulsado el botón de cancelar (*onRejected*) o el de aceptar (*onAccepted*).

```
MessageDialog {
    id: messageDialog
    title: "ATENCIÓN"
    text: "¿Terminar partida?"
    icon: StandardIcon.Warning
    standardButtons: StandardButton.Ok | StandardButton.Cancel

    onAccepted: {
        Logic.goBack()
        Logic.resetearJuego()
    }
    onRejected: {
        messageDialog.close()
    }

    Component.onCompleted: visible = true
}
```

Gracias a las propiedades *title*, *text* y *icon*, se especifica un título para el diálogo, un texto que verá el usuario y un icono, respectivamente. De esta forma, el *handler onAccepted* se ejecutaría en caso de que se haya pulsado el *StandardButton.Ok*, ejecutando entonces instrucciones *JavaScript* que se encuentran implementadas en funciones de nuestro archivo logica.js, mientras que el *handler onRejected* está ligado al botón *Cancel*, y en él simplemente se cierra el diálogo que se acaba de abrir. Con la última instrucción, simplemente se indica que el diálogo se haga visible al crear un objeto de tipo "goBackDialog". El aspecto de este diálogo se puede apreciar en esta ilustración.

Ilustración 48 – Diálogo modal que aparece al pulsar la flecha de vuelta hacia atrás.

9.3.3 finishDialog.Qml

Cuando uno de los equipos ha llegado al límite de su marcador, tres goles, es necesario indicarle al usuario que la partida ha terminado, ya que ya existe un equipo ganador. Esto se consigue a través de un diálogo, el cual, como ocurría con el diálogo anterior, es un diálogo modal simple en el que se despliega un mensaje al usuario. Además de indicarle al usuario que el partido ha finalizado, se le preguntará si desea jugar una nueva partida o no. En caso afirmativo, es decir que el usuario ha pulsado el botón *Ok*, las chapas se recolocarán en su posición inicial y los marcadores de ambos equipos volverán a cero. En caso negativo, es decir que se ha pulsado el botón de cancelar, el usuario será devuelto a la pantalla de inicio de la aplicación. La implementación de este diálogo es la que sigue:

```
import QtQuick 2.0
import QtQuick.Dialogs 1.1
import "content/logica.js" as Logic

MessageDialog {
    id: messageDialog
    title: "PARTIDO ACABADO"
    text: "¿Desea jugar otro partido?"
    icon: StandardIcon.Warning
    standardButtons: StandardButton.Ok | StandardButton.Cancel

    onAccepted: {
        Logic.resetearJuego()
    }
    onRejected: {
        messageDialog.close()
        Logic.goBack()
        Logic.resetearJuego()
    }

    Component.onCompleted: visible = true
}
```

Como se ve, en este caso también se hace uso de la lógica que se ha implementado para el juego invocando a las funciones “resetearJuego()” o “goBack()” según el botón que haya pulsado el usuario en el diálogo. En caso de que pulse aceptar, el usuario verá cómo se resetea el juego, de manera que las chapas vuelven a su posición original inicial y los marcadores a cero, encargándose de todo esto las propias funciones de la lógica. Si, sin embargo, pulsa el botón cancelar, estará indicando que no desea jugar otra nueva partida, por lo que el dialogo se cerrará, se reseteará el juego y se volverá atrás a la pantalla de inicio. Véase la ilustración 45.

9.3.4 chapaDialog.Qml

Es a través del diálogo denominado “chapaDialog.qml” por el que se captan datos del usuario que tienen que ver con la dirección a la que este quiere mandar la

chapa que ha seleccionado sobre el campo. Este diálogo se ha implementado siguiendo las pautas que se especificaron en la fase de diseño. Se han incorporado dos barras, una vertical y otra horizontal, y dos botones que acompañan sendas barras. Se trata, otra vez, de un diálogo modal pero, en esta ocasión, **personalizado**. No es un *MessageDialog* como ocurría en los dos casos anteriores. Esta vez decidimos implementar un diálogo personalizado que resultara intuitivo para el usuario a la hora de determinar a dónde dirigir la chapa que había seleccionado.

Ilustración 49 – Diálogo modal que aparece al acabar un partido.

La implementación de este diálogo comienza con la definición de un elemento de tipo *Item* que será raíz y que contendrá el resto de elementos.

```
Item {
        id: window
        visible: true
        width: 500
        height: 220
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        property Image imgGo1: imgGo1
        property Image imgGo2: imgGo2
        property Image imgDetectorH: imgDetectorH
        property Image imgDetectorV: imgDetectorV

      //SIGUE
```

Se han definido una serie de propiedades que servirán para referenciar, desde la lógica, los botones de “GO!” que aparecen en el diálogo y también los elementos que constituyen los ejes que se mueven en las barras en forma de animación. La referencia a los botones “GO!”, se podrá emplear para ocultarlos cuando hayan sido pulsados, evitando así que el usuario no pueda pulsar un botón “GO!” otra vez si ya ha sido pulsado. Las referencias de los ejes que se mueven en las barras, se emplearan para acceder a sus propiedades *x* e *y* y poder determinar, según esos valores, a dónde dirigir la chapa que fue seleccionada.

```
Rectangle {
            id: rectBorder
            width: window.width
            height: window.height
            border.color: "#800000FF"
            border.width: 10
            radius: 10
            smooth: true

            //IMAGEN DE FONDO
            Image {
                x: rectBorder.x + 5
                y: rectBorder.y + 5
                source: "content/dialog_back.jpg"
                width: rectBorder.width - 10
                height: rectBorder.height - 10
            }
        }

        //BARRA HORIZONTAL Y UN DETECTOR O EJE QUE SE MUEVE
        Image {
            id: imgBarraH
            source: "content/barra_horizontal.png"
            width: window.width*0.7
            y: 20
            x: 30

            Image {
                id: imgDetectorH
                source: "content/detector_posH.png"
                anchors.horizontalCenter: parent.verticalCenter
                x: 0

                SequentialAnimation on x {
                    id: animationX
                    loops: Animation.Infinite
                    NumberAnimation { to: imgBarraH.width -
imgDetectorH.width; duration: 1800 }
                    NumberAnimation { to: 0; duration: 1800 }
                }
            }  //FIN detector horizontal
        } //FIN barra horizontal
```

Con el elemento *Rectangle*, se ha definido simplemente un borde alrededor del diálogo, para darle un toque más atractivo y para que el usuario percibiera bien que una nueva ventana se había abierto. Viendo las propiedades, se observa que se trata de un borde con un acabado circular en las esquinas y al que se la ha intentado dar un color llamativo. Este borde rectangular contiene, a su vez, una imagen de fondo. Con esta imagen se ha intentado que el diálogo no tenga un fondo de color común o simple, sino que el fondo se trata de una imagen en tono de grises que resulta en un fondo más llamativo, que permite, además, ver con mejor claridad las barras y los botones que se presencian también en el diálogo. En el fragmento de código anterior, también se puede ver que se define una imagen que representa la barra horizontal que habrá en el diálogo, la cual, a su vez, contiene un detector o eje que se moverá a lo largo de esta barra. Así es, el eje o detector de la barra horizontal consiste en una imagen, la de *id* "imgDetectorH", que se va moviendo siguiendo una animación. En este caso, hay una animación que se aplica sobre la propiedad *x* del elemento y que viene constituida, a su vez, por dos animaciones de tipo *Number*: una para mover el eje de izquierda a derecha, y otra para moverlo de derecha a izquierda. Estas dos animaciones, se repiten de forma infinita hasta que el usuario pulsa el botón "GO!" asociado a esta barra. Dicho botón se implementa de esta manera:

```
Image {
            id: imgGo1
            source: "content/btn_go.png"
            width: 50
            height: 50
            x: window.width/2.8
            y: 60
            MouseArea {
                anchors.fill: parent
                onClicked: {
                    imgGo1.visible = false
                    animationX.running = false
                    Logic.closeDialog(window, "go1")
                }
            }
        }
```

Prestando atención, se ve que en caso de que se pulse esta imagen que representa uno de los botones "GO!", el botón desaparecerá al modificar su propiedad *visible*, y además la animación antes descrita, *animationX*, parará de ejecutarse, por lo que el eje de la barra horizontal se pararía en el punto en el que estuviera cuando el usuario pulsa este botón. Además, se hace una llamada a una función implementada en nuestro fichero logica.js, en la cual se tendrá que comprobar si los dos botones "GO!" han sido pulsados ya, para que, en dicho caso, el diálogo se cierre. Una implementación similar a esta, ocurre con la barra vertical de este diálogo, salvo que en la animación de su eje sobre la barra vertical la propiedad que se verá modificada no es la *x* sino la *y*.

```
//BARRA VERTICAL Y UN DETECTOR
        Image {
            id: imgBarraV
            source: "content/barra_vertical.png"
            height: window.height*0.8
            x: window.width*0.92
            y: 20
        //SIGUE
```

```
Image {
                id: imgDetectorV
                source: "content/detector_posV.png"
                anchors.horizontalCenter: parent.horizontalCenter
                y: 0

                SequentialAnimation on y {
                    id: animationY
                    loops: Animation.Infinite
                    NumberAnimation { to: imgBarraV.height -
imgDetectorV.height; duration: 1500 }
                    NumberAnimation { to: 0; duration: 1500 }
                }
            } //FIN detector vertical
        }  //FIN barra vertical

        Image {
            id: imgGo2
            source: "content/btn_go.png"
            width: 50
            height: 50
            x: window.width*0.8
            y: 100

            MouseArea{
                anchors.fill: parent
                onClicked: {
                    imgGo2.visible = false
                    animationY.running = false
                    Logic.closeDialog(window, "go2")
                }
            }
        }  //FIN del otro botón "GO!"
}  //FIN del Item raíz y del archivo "chapaDialog.Qml"
```

Con todo esto, el aspecto de este diálogo se puede ver en la siguiente ilustración. En este caso, se puede ver el resultado que se tendría al ejecutar el juego en un sistema *Android*.

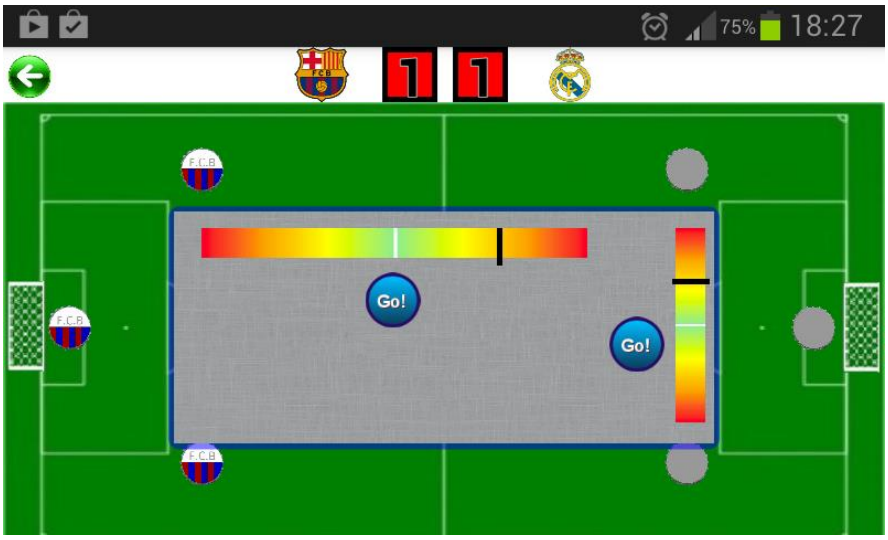


Ilustración 50 – Diálogo modal que aparece al pulsar una chapa.

9.3.5 SoundEffect.Qml

En el juego que se ha desarrollado, se reproducen sonidos en diferentes momentos y partes de este. Para poder reproducir estos sonidos sabíamos que era necesario hacer uso del componente *SoundEffect* de *Qml*, lo cual nos obligaba a importar una librería conocida como *QtMultimedia 5.0*. Era precisamente esta librería la que daba problemas de compatibilidad con ciertos dispositivos, *Android* por ejemplo, en los que ejecutábamos el juego, lo que resultaba en una ejecución en la que no había ningún tipo de sonido sobre el móvil, por lo que se perdían características del juego que se estaba desarrollando.

Para solucionar este inconveniente, decidimos crear nuestro propio componente *SoundEffect*, y fue por eso que creamos nuestro propio archivo *SoundEffect.Qml*. De esta forma, los diferentes componentes de tipo *SoundEffect* que se han visto en anteriores archivos ya comentados, son, realmente, del tipo del objeto que nosotros creamos. Así, en este nuevo objeto se encuentran las funciones de "play()" y "pause()" pertinentes para reproducir o parar el sonido.

La implementación de este componente, consta de un simple objeto *Item* en el que se han desarrollado las funciones de reproducir y de parar la música. Para la definición de este componente, se ha creado una propiedad *QtObject* en la que sí tiene lugar la importación de las librerías y directorios pertinentes. De esta manera, los objetos *SoundEffect* de otros archivos añaden estas librerías dinámicamente y se soluciona el problema que teníamos al ejecutar en dispositivos móviles.

```
import QtQuick 2.0

Item {
    id: container
    property QtObject effect: Qt.createQmlObject("import
QtMultimedia 5.0; import '../'; SoundEffect{ source: '" +
container.source + "' }",container);
    property url source: ""
    property int loops: 0
    onSourceChanged: if (effect != null) effect.source = source;
    function play() {
        if (effect != null) {
            effect.loops= 1
            effect.play()
        }
    }
    function stop() {
        if (effect != null) {
            effect.loops= 1
            effect.stop()
        }
    }
}
```

9.4 Implementación de la lógica

Para dotar de vida a la interfaz del juego que se estaba desarrollando, era necesario implementar la lógica de la aplicación, la cual permitiera al usuario interactuar con las diferentes pantallas del juego. Toda la lógica de la aplicación sería implementada en un solo archivo, el cual fue nombrado como "logica.js". Dicho archivo, como se vio en líneas anteriores, fue objeto de importación en los archivos con los que se describe la interfaz, los de extensión "*.qml*", y es que esto permitió a los componentes de la interfaz tener la capacidad de interactuar con el usuario. A lo largo de este apartado, se comentan cada una de las funciones de las que consta el archivo de lógica, qué hacen dichas funciones y cómo esto se lleva a cabo.

9.4.1 Declaración de variables globales y de constantes

Las numerosas funciones que se implementan en el archivo de lógica, hacen uso, en ocasiones, de una serie de variables **globales** y valores **constantes** que son compartidas y compartidos por las demás funciones del archivo. Las constantes, como su propio nombre indica, almacenan un valor que permanece constante a lo largo de la ejecución del juego, al contrario de lo que ocurre con las variables globales. Se comenta, a continuación, la finalidad y el propósito de cada una de estas variables y constantes:

- gameState: variable con la que se puede acceder a las diferentes propiedades que se definen en los componentes *Qml* de la interfaz del juego. Al arrancar un nuevo juego, desde el archivo "main.qml" se hace una invocación a la función "newGameState()", en la cual se pasa como argumento el id del elemento raíz de dicho archivo. La variable gameState, se inicializará en dicha función y tomará como valor el argumento que se le pasa a esta, permitiendo así acceder a los componentes, y propiedades de estos, del archivo "main.qml".

- chapaActual: variable que representa la chapa actual que se está moviendo. Al pulsar una chapa, dicha chapa representará la chapa actual, pero si sufre un choque, la variable "chapaActual" tomará como valor la chapa que ha sido chocada, ya que esta nueva será la que empiece a moverse.

- chapaElegida: variable que representa la chapa que el usuario ha pulsado. Solo cambia de valor tras haber un cambio de turno y tras haber elegido el usuario una nueva chapa.

- chapaDialog: variable que permite acceder a los diferentes componentes, y propiedades de estos, del archivo "chapaDialog.qml". Esta variable tomará el valor del id del elemento raíz del diálogo que se visualiza tras pulsar una chapa. Es importante porque es la variable que permite acceder a los valores que indica el usuario sobre hacia dónde mover la chapa que ha pulsado.

- posicionesDetectores: *array* de cuatro elementos que almacena los valores *x* e *y* que ha introducido el usuario sobre el diálogo que aparece al pulsar una chapa. Determinará, para la barra horizontal del diálogo, si el usuario ha elegido mover a izquierdas o a derechas (posición tres de este *array*), y para la barra vertical, si ha elegido mover arriba o abajo (posición uno del *array*).

- signoXDetectado, signoYDetectado: ambas variables almacenarán un valor en forma de *String* que indique "-" o "+". La variable "signoXDetectado" indicará cuál ha sido la dirección que el usuario ha indicado sobre la barra horizontal del diálogo "chapaDialog.qml", de forma que la dirección izquierda tomará el valor "-", mientras que la derecha el valor "+". Algo similar ocurre con la variable "signoYDetectado", donde se almacena la dirección que el usuario ha indicado sobre la barra vertical del diálogo de una chapa, de forma que el signo "-" indica un movimiento hacia arriba, mientras que el signo "+" hacia abajo.

- factorMovimientoAlto: constante que se emplea a la hora de determinar el valor de las variables "destinoX" y "destinoY".

- factorMovimientoBajo: constante que se emplea al determinar el valor de la variable “destinoX”.

- newX: variable que almacena el valor que el usuario ha introducido sobre la barra **horizontal** del diálogo que aparece al pulsar una chapa. Su valor es el que estará almacenado en la posición tres del *array* “posicionesDetectores”.

- newY: variable que tiene la misma finalidad que la anterior, pero que en este caso almacena el valor que el usuario ha introducido sobre la barra **vertical** del diálogo. Su valor viene dado por la posición uno del *array* “posicionesDetectores”.

- destinoX, destinoY: variables que almacenan, en formato decimal, hasta dónde se moverá, si no se choca, una chapa en su desplazamiento sobre el eje de las *x* y sobre el eje de las *y*, respectivamente. El valor de ambas, se calcula en función de la posición en la que el usuario ha parado las animaciones que se mueven sobre las barras horizontales y verticales del diálogo de pulsar una chapa, es decir, se basa en los valores de “newX” y “newY”, en la posición *x* e *y* de la chapa actualmente, y en los valores de algunas de las constantes antes citadas, según sea el caso.

- angulo: variable que determina cuál será el ángulo de movimiento de la chapa teniendo en cuenta la posición actual de esta en *x* e *y*, y los valores que se han calculado para “destinoX” y “destinoY”. Se emplea, como se verá en el código, un cálculo trigonométrico.

- recto: variable que almacena un valor booleano. En ella se indicará si el usuario ha deseado hacer un movimiento en recto o no, teniendo en cuenta los valores captados en el diálogo de una chapa. Para esto se comprueba el valor de la variable “newY”, de forma que si el usuario se para, aproximadamente, sobre el centro de la barra vertical, la variable tomará un valor igual a *true* y el movimiento será recto, o sea que no hay desplazamiento sobre el eje de las *y*.

- gol: variable que, nuevamente, almacena un valor booleano. Cuando esta variable sea igual a *true* deberá reproducirse un sonido de gol. Para que la variable tome dicho valor, se comprobará si la chapa actual que se está moviendo, está sobre ciertos límites del campo, en pocas palabras, dentro de la portería rival.

- marcador1, marcador2: variables que almacenan un valor entero y que representan el marcador de goles de un equipo y otro. Se emplearán para

comprobar si alguna de ellas ha llegado en algún momento a ser tres, ya que en dicho momento habrá que comenzar una nueva partida.

- ArrayDeChapas: variable que representa un *array* en el que se almacenan objetos de la clase "Chapa". Cada chapa que hay almacenada en este *array* tiene un id propio y único que la identifica, además de una posición *x* e *y* que refleja el punto actual en el que se encuentra la chapa en el campo. Este *array* será el que se recorrerá siempre que se quiera detectar las posibles **colisiones** que pueda haber entre la chapa actual que se está moviendo y las demás.

- limitesCampoX: constante que refleja, en un *array* de dos valores, las dimensiones del terreno de juego en el que se mueven las chapas. Permitirá detectar si una chapa intenta sobrepasar alguno de estos límites para que, en dicho caso, rebote y no se salga del campo.

- limitesPorteriasY: constante en forma de *array*, de dos valores, que indica a qué altura del campo aparecen las porterías de fútbol, es decir, a qué valor del eje *y*. Esta constante será importante para poder determinar, según la posición de la chapa actual, si ha habido gol o no.

- chapaAux: variable de tipo "Chapa" que se emplea de forma auxiliar para poder acceder a los métodos que se definen en la clase "Chapa".

- factorPonderacionAlto, factorPonderacionBajo: constantes que, al igual que ocurría con otras como "factorMovimientoAlto", influyen en el cálculo de las variables "destinoX" y "destinoY", es decir, influyen en dónde se intentará colocar una chapa tras pulsarla el usuario.

- anguloX: variable que se calcula a partir de la variable "angulo", a través de operaciones trigonométricas de senos y cosenos. Su valor indicará en qué factor se irá moviendo sobre el eje *x* la chapa que se ha pulsado.

- anguloY: esta variable también se calcula a partir de la variable "angulo", como ocurría con la variable anterior. En este caso, afectará a cómo va cambiando la posición en el eje *y* sobre la chapa actual que se está moviendo.

- empezarCrono: variable que toma un valor booleano para indicar si el crono del tiempo de juego ha empezado o no. Toma como valor inicial *false*, de manera que al pulsar el usuario el botón de jugar pasa a tomar el valor *true*.

- segundos: variable con la que se irán contabilizando los segundos del crono de la partida cuando esta dé comienzo.

- minutos: variable con la que se irán contabilizando los minutos del crono de la partida cuando esta dé comienzo.

```
var gameState;
var chapaActual;
var chapaElegida;
var chapaDialog;
var posicionesDetectores;
var signoXDetectado;
var signoYDetectado;
var factorMovimientoAlto = 0.95;
var factorMovimientoBajo = 0.5;
var newX;
var newY;
var destinoX;
var destinoY;
var angulo;
var recto;
var gol;
var marcador1;
var marcador2;
var ArrayDeChapas;
var limitesCampoX = [30, 725];
var limitesPorteriasY = [163, 204];
var chapaAux;
var factorPonderacionAlto = 1000;
var factorPonderacionBajo = 10000;
var anguloX;
var anguloY;
var empezarCrono;
var segundos;
var minutos;
```

9.4.2 Clase "Chapa"

En el lenguaje de programación *JavaScript* se pueden definir objetos que no vienen predefinidos en el propio lenguaje, objetos que tienen sus propiedades y métodos. Esto es lo que hemos hecho con las chapas que hay sobre el terreno de juego, es decir, hemos creado una serie de objetos de tipo "Chapa" para poder manipularlos durante la lógica de la aplicación. Para ello, fue necesario definir una función **constructora** y otra serie de métodos con los que realizar operaciones de **consulta** y de **modificación** sobre las chapas.

Con la función constructora podíamos crear objetos de tipo "Chapa", siendo necesario la indicación de tres parámetros o propiedades: un identificador *id* que sería único para cada chapa, un valor que indicase la posición de la chapa sobre el eje *x*, y un valor que indicase la posición de la chapa sobre el eje *y*.

Por otro lado, se implementaron métodos que permitieran realizar consultas de las propiedades antes citadas, así como también se implementaron métodos que permitieran modificarlas. También se desarrolló un método llamado "getChapa()", el cual recibía como argumento un identificador de chapa, y donde se recorría la variable "ArrayDeChapas" en busca de la chapa que tuviera el *id* pasado como argumento, devolviendo dicha chapa cuando se encontrase en este *array*. La implementación de estos métodos y de las propiedades se puede ver aquí:

```
var Chapa = function(idChapa, x, y){
    this.idChapa = idChapa;
    this.x = x;
    this.y = y;

    this.getX = function(){
        return this.x;
    }
    this.getY = function(){
        return this.y;
    }

    this.setX = function(xDestino){
        this.x = xDestino;
    }
    this.setY = function(yDestino){
        this.y = yDestino;
    }

    this.getID = function() {
        return this.idChapa;
    }

    this.getChapa = function(id) {
        var i = 0;
        while (ArrayDeChapas[i].getID() != id)
            i++;
        return ArrayDeChapas[i];
    }
}
```

9.4.3 Función "newGameState()"

Esta función se ejecuta una única vez en todo el juego, cuando arranca. A través del archivo "main.qml" se ve que se realiza una invocación a esta función, la cual acepta un argumento. En este caso, lo que ocurrirá en esta función, entre otras cosas, es que la variable "gameState" tomará como valor el argumento que se le ha pasado a la función, y este paso permitirá poder acceder desde la lógica a todos los componentes y propiedades del archivo ".*Qml*" antes nombrado, haciendo para ello uso de la variable "gameState". Es una función que sirve como paso inicializador, es decir, muchas de las variables globales tomarán un valor inicial en este paso. De entre las variables

inicializadas se encuentra la variable "ArrayDeChapas". Se creará en este paso, pues, un nuevo objeto de tipo *Array*, en el cual se añadirán objetos de la clase "Chapa". Concretamente se añadirán las diez chapas, las cinco de cada equipo, indicando para cada chapa un identificador *id*, y una posición inicial en el eje *x* e *y*.

```
function newGameState(mainbar)
{
    gameState = mainbar;
    chapaActual = null;
    chapaElegida = null;
    gol = false;
    marcador1 = 0;
    marcador2 = 0;
    posicionesDetectores = [];
    recto = false;
    empezarCrono = false;
    segundos = 0;
    minutos = 0;
    ArrayDeChapas = new Array();
    var chapa1 = new Chapa(gameState.chapa1,
gameState.imgGameOn.width*0.2, gameState.imgGameOn.height*0.10);
    var chapa2 = new Chapa(gameState.chapa2,
gameState.imgGameOn.width*0.05, gameState.imgGameOn.height*0.45);
    var chapa3 = new Chapa(gameState.chapa3,
gameState.imgGameOn.width*0.2, gameState.imgGameOn.height*0.45);
    var chapa4 = new Chapa(gameState.chapa4,
gameState.imgGameOn.width*0.2, gameState.imgGameOn.height*0.75);
    var chapa5 = new Chapa(gameState.chapa5,
gameState.imgGameOn.width*0.35, gameState.imgGameOn.height*0.45);
    var chapa8 = new Chapa(gameState.chapa8,
gameState.imgGameOn.width*0.9, gameState.imgGameOn.height*0.45);
    var chapa9 = new Chapa(gameState.chapa9,
gameState.imgGameOn.width*0.75, gameState.imgGameOn.height*0.10);
    var chapa10 = new Chapa(gameState.chapa10,
gameState.imgGameOn.width*0.75, gameState.imgGameOn.height*0.45);
    var chapa11 = new Chapa(gameState.chapa11,
gameState.imgGameOn.width*0.75, gameState.imgGameOn.height*0.75);
    var chapa12 = new Chapa(gameState.chapa12,
gameState.imgGameOn.width*0.6, gameState.imgGameOn.height*0.45);
    ArrayDeChapas.push(chapa1, chapa2, chapa3, chapa4, chapa5,
chapa8,chapa9, chapa10, chapa11, chapa12);

    return gameState;
}
```

9.4.4 Funciones "startGame()" y "stopGame()"

A través de estas dos funciones, se controla el movimiento que hay entre las pantallas que ve el usuario según las acciones que lleve a cabo. Cuando el usuario pulsa el botón de jugar en la pantalla de inicio, se invocará desde el archivo "main.*Qml*" la función "startGame()" del archivo de lógica, la cual simplemente cambiará una de las propiedades del archivo "main.qml", la denominada "gameRunning", para indicar que el juego se encuentra ahora en una nueva pantalla, la de juego, lo que significa, además,

que el crono de juego debe empezar a correr. Al estar en la pantalla de juego, el usuario puede querer abandonarla en cierto momento, pulsando para ello la flecha de vuelta atrás. En ese caso, será necesario llamar a la función "stopGame()", donde, entre otras cosas, se invocará a una función que resetea el juego y donde se vuelve a la pantalla de inicio.

```
function startGame() {
    gameState.gameRunning=true;
    empezarCrono = true;
}

function stopGame() {
    gameState.gameRunning=false;
    resetearJuego();
}
```

9.4.5 Función "chapaPulsada()"

Con esta función, las variables globales "chapaActual" y "chapaElegida" toman un nuevo valor, el que viene pasado como argumento de la función. La función será invocada cuando una chapa sea pulsada, es decir que será invocada desde el archivo "main.qml" ya que ahí se encuentran definidas las chapas y es donde el usuario puede pulsarlas. La chapa que es pulsada es la que se pasa como argumento. Por otro lado, en esta función también se deshabilitan todas las chapas, es decir, las chapas no se podrán pulsar, ya que al pulsar una chapa se abre un diálogo sobre el usuario, por lo que se quiere evitar, de esta manera, que el usuario pulse otras chapas y abra nuevos diálogos innecesariamente, algo que afectaría a la experiencia de este. También se deshabilita la flecha de vuelta atrás, la cual si se podrá volver a pulsar cuando el diálogo de la chapa se cierre.

```
function chapaPulsada (id) {
   //DESHABILITAR LAS CHAPAS
   gameState.chapa1.enabled = false;
   gameState.chapa2.enabled = false;
   gameState.chapa3.enabled = false;
   gameState.chapa4.enabled = false;
   gameState.chapa5.enabled = false;
   gameState.chapa8.enabled = false;
   gameState.chapa9.enabled = false;
   gameState.chapa10.enabled = false;
   gameState.chapa11.enabled = false;
   gameState.chapa12.enabled = false;

    //MODIFICAR VARIABLES GLOBALES
    chapaActual = id;
    chapaElegida = id;

    //DESHABILITAR FLECHA DE VUELTA ATRÁS
    gameState.flecha.enabled = false;
}   //FIN
```

9.4.6 Función "closeDialog()"

Esta función se invoca desde el diálogo en el que se captan los datos del usuario para mover una chapa, es decir, desde el archivo "chapaDialog.qml", y sirve para cerrar dicho diálogo una vez que el usuario ha introducido los datos necesarios para mover una chapa. En este caso, la función también cambiará el valor de la variable "chapaDialog", asignándole, entonces, el que viene dado por el primer argumento de la función, lo que permitirá emplear la variable "chapaDialog" para acceder a todas las propiedades y componentes del archivo "chapaDialog.qml". Además, en esta función se comprueba si el usuario ha introducido todos los datos necesarios para mover una chapa, es decir, se comprueba si los dos botones "GO!" del diálogo han sido pulsados para que, en dicho caso, el diálogo se cierre automáticamente invocando a una función "destroy()". Para ello será necesario emplear el segundo argumento de la función, el que sostiene el identificador del botón pulsado. Tras cerrar el diálogo, se invocará la función "getValoresCaptados()".

```
function closeDialog(window, idBtn) {
    chapaDialog = window;
    if (chapaDialog.imgGo2.visible == false && idBtn == "go1") {
        chapaDialog.destroy();
        getValoresCaptados();
    }
    if (chapaDialog.imgGo1.visible == false && idBtn == "go2") {
        chapaDialog.destroy();
        getValoresCaptados();
    }
}
```

9.4.7 Función "getValoresCaptados()"

Esta función realiza una tarea determinante en el desarrollo del juego. Se encarga, principalmente, de almacenar los valores que el usuario ha introducido en el diálogo que se abre al pulsar una chapa, de manera que guarda, en las variables que ahora se verán, los valores que tienen que ver con el movimiento de la chapa sobre el eje *x* y el eje *y*.

Para determinar cómo se mueve una chapa, es necesario saber qué ha indicado el usuario sobre las barras horizontal y vertical del diálogo de la chapa que ha pulsado. Los valores que ahí indique, como se vio antes, se almacenan en un *array* denominado "posicionesDetectores", así que lo primero que se hace en esta función es vaciar dicho *array* (operación "pop()") ya que la función puede ser llamada más de una vez, por lo que se irían acumulando valores en el *array* y eso es algo que no interesa. Tras vaciar el *array*, se insertarán en él (operación "push()") los valores que ha introducido el usuario en las barras, siendo los realmente importantes los de la posición uno y tres del *array*.

```
function getValoresCaptados(){
    if (posicionesDetectores != null) {
        while (posicionesDetectores.pop() != null)
            posicionesDetectores.pop();
    }

    posicionesDetectores.push(chapaDialog.imgDetectorV.x);
    posicionesDetectores.push(chapaDialog.imgDetectorV.y);
    posicionesDetectores.push(chapaDialog.imgDetectorH.Y);
    posicionesDetectores.push(chapaDialog.imgDetectorH.x);
    …
    //SIGUE
```

Una vez se tienen valores en el *array* "posicionesDetectores", estos se analizan. Para ello, primero se tiene en cuenta la dirección que ha indicado el usuario: **izquierda** o **derecha**, lo cual viene indicado por la posición tres del *array*, y **arriba** o **abajo**, lo cual viene indicado por la posición uno del *array*. Analizado esto, según sus valores se modifican las variables "signoYDetectado" y "signoXDetectado" pertinentemente. Esto se puede ver aquí:

```
if (posicionesDetectores[1] <= 88)
        signoYDetectado = "-";
    else signoYDetectado = "+";
    if (posicionesDetectores[3] <= 175)
        signoXDetectado = "-";
    else signoXDetectado = "+";
    //SIGUE
```

Conseguido esto, para trabajar de una manera más cómoda se asigna a la variable "newX" el valor contenido en la posición tres del *array* "posicionesDetectores", el cual indica cuánto quiere desplazarse en el eje *x* la chapa seleccionada, y se asigna a la variable "newY" el valor contenido en la posición uno, indicando la cantidad de desplazamiento sobre el eje *y*. Tras modificar los valores de ambas variables, se procede a determinar los valores de las variables "destinoX" y "destinoY". Efectivamente, se analizarán las variables "newX" y "newY" para que, según esos valores, se determine **cuánto** se moverá la chapa, ya que la **dirección** del movimiento ya se determinó antes.

En el análisis de "newX" y "newY" se emplean ciertos valores constantes que se comparan con dichas variables, para determinar **en qué punto** de las barras horizontal y vertical, del diálogo de una chapa, se ha parado el usuario. Es importante saber en qué punto se ha parado el usuario para saber si hay que mover la chapa con mayor o menor profundidad.

```
newX = posicionesDetectores[3]; //VALOR BARRA HORIZONTAL
newY = posicionesDetectores[1]; //VALOR BARRA VERTICAL

    if (newY <= 64) {
        if (newY <= 2)
            newY = 2;
        destinoY = chapaActual.y - (1/newY)*factorPonderacionAlto;
    }
    else if (newY > 64 && newY <= 100) {
        recto = true;
    }
    else destinoY = chapaActual.y + newY*factorMovimientoAlto;

    if (newX <= 78) {
        if (newX <= 2)
            newX = 2;
        destinoX = chapaActual.x - ((1/newX)*factorPonderacionAlto +
150);
    }
    else if (newX > 78 && newX <= 175) {
        destinoX = chapaActual.x - ((1/newX)*factorPonderacionBajo);
    }
    else {
        if (newX > 175 && newX < 310)
            destinoX = chapaActual.x + newX*factorMovimientoBajo;
        if (newX > 310)
            destinoX = chapaActual.x + newX*factorMovimientoAlto;
    }

    if (destinoY < 0)
        destinoY = chapaActual.y - (destinoY*(-1));
    if (destinoY > 365)
        destinoY = chapaActual.y - destinoY/2;
    //SIGUE
```

Aquí se ve, pues, la importancia de las **constantes** que antes se detallaron. En el caso de la barra horizontal, hay que tener en cuenta que si el usuario se para muy a la izquierda de dicha barra, el valor en el eje *x* de la barra será bastante pequeño, sin embargo al pararse en esa posición el usuario quiere indicar que quiere moverse **mucho** a la izquierda, es por eso que se tiene que hacer uso de una constante que aumente ese valor. Al indicar una posición más cercana al centro de la barra, el valor en el eje *x* será mayor, sin embargo eso significaría un menor desplazamiento hacia la izquierda, es por eso que se hace uso de una operación de división (1/newX), ya que eso permite que para valores **más pequeños** en el eje *x* el desplazamiento sea **mayor**, ya que es lo que realmente está indicando el usuario. En el caso de la barra vertical ocurre algo similar con los valores de "newY" y "destinoY". Una parte destacable del análisis de estas variables, es el que permite determinar cuándo el movimiento de una chapa será recto. Para ello, el usuario se parará en posiciones aproximadas al centro de la barra vertical, lo que hará que la variable "recto" sea *true*, por lo que la chapa, como se verá más tarde, sólo se moverá sobre el eje *x*.

Finalmente, una vez determinada la dirección y profundidad con la que se moverá una chapa, habrá que determinar cuál será el ángulo de movimiento de esta,

teniendo en cuenta, nuevamente, lo que el usuario ha indicado sobre las barras horizontal y vertical del diálogo de una chapa seleccionada. Para esto, se tendrá en cuenta la posición actual en *x* e *y* de la chapa y, además, los valores recientemente calculados de las variables "destinoX" y "destinoY". Entre estos valores, se calculará la **diferencia** que existe entre la posición actual de la chapa y la posición que, se supone, sería su destino, obteniendo las variables "deltaX" y "deltaY". Con esto, se realizan unos cálculos trigonométricos (véase la ilustración 47) con los que determinar en qué unidades incrementar la *x* (variable "anguloX") de la chapa cuando se va moviendo, y en qué unidades incrementar la *y* de la chapa (variable "anguloY").

```
    var deltaX;
    var deltaY;
    deltaX = destinoX - chapaActual.x;
    deltaY = destinoY - chapaActual.y;
    if (deltaX < deltaY) {
        angulo = Math.atan(deltaX/deltaY) ;
        anguloX = Math.sin(angulo);
        anguloY = Math.cos(angulo);
    }
    else {
        angulo = Math.atan(deltaY/deltaX) ;
        anguloX = Math.cos(angulo);
        anguloY = Math.sin(angulo);
    }

    moverChapa(chapaActual, signoXDetectado, signoYDetectado);
    gameState.flecha.enabled = true;

} //FIN de la función "getValoresCaptados()"
```

Al final de la función, se hace una invocación de la función "moverChapa()" para que, una vez determinados todos los valores necesarios, comience la chapa elegida a moverse. Además, el usuario podrá, a partir de ese momento, abandonar la partida pulsando la flecha de vuelta atrás, una flecha que no podía pulsar mientras estaba introduciendo los datos con los que mover la chapa.

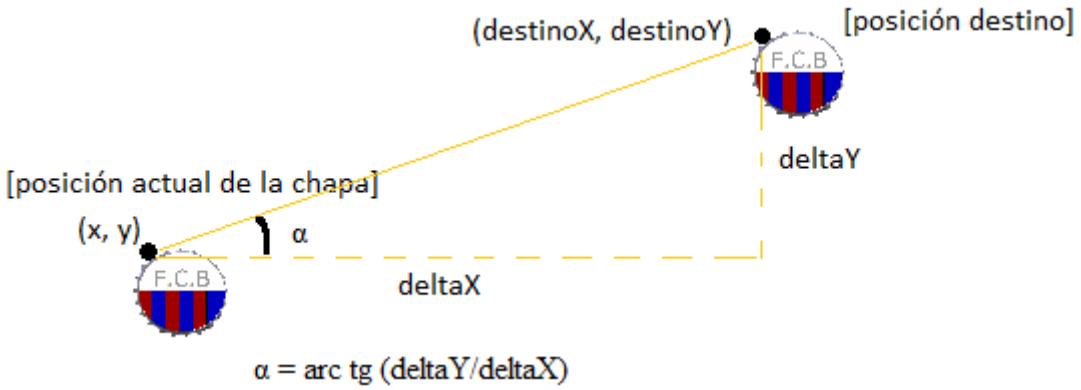


*Ilustración 51 – Demostración del cálculo del ángulo "**α**" de movimiento de una chapa.*

9.4.8 Función "moverChapa()"

En esta función se ejecuta una única instrucción. Dicha instrucción, permitirá que la chapa que pulsó el usuario y para la cual indicó unos valores de movimiento comience a moverse. La instrucción consiste, sin ir más lejos, en una modificación de la variable "chapaActual". La función que efectúa, realmente, el movimiento en sí es otra ("tick()"), y para llevar a cabo el movimiento comprueba una propiedad de la variable "chapaActual" que es modificada por la función "moverChapa()".

```
function moverChapa(idChapa, signoX, signoY) {
    chapaActual.state = "moviendo";
}
```

9.4.9 Función "cambiarTurno()"

Por medio de la función "cambiarTurno()", se lleva a cabo el cambio de turno entre equipos a la hora de tirar las chapas. Para esto, la función tomará un argumento que le indicará cuál es el identificador de la chapa que invoca la función, lo que permitirá acceder a la propiedad "idEquipo" que indica a qué equipo pertenece la chapa. Con esto, las chapas del otro equipo, es decir, las que tienen un identificador de equipo diferente, podrán ser pulsadas (propiedad "enabled" será true) para que alguna de ellas sea lanzada, mientras que las que pertenecen al equipo de la chapa que se acaba de lanzar no se podrán pulsar, incluida la chapa que se lanzó. Como se ve en este código, las chapas numeradas de uno a cinco pertenecen al equipo uno ("idEquipo" es igual a uno), que sería el fútbol club Barcelona, y las chapas numeradas de ocho a doce pertenecen al equipo dos, es decir, al Real Madrid ("idEquipo es igual a dos").

```
function cambiarTurno(idChapa) {
    if (idChapa.idEquipo == 1) {
        gameState.chapa1.enabled = false;
        gameState.chapa2.enabled = false;
        gameState.chapa3.enabled = false;
        gameState.chapa4.enabled = false;
        gameState.chapa5.enabled = false;

        gameState.chapa8.enabled = true;
        gameState.chapa9.enabled = true;
        gameState.chapa10.enabled = true;
        gameState.chapa11.enabled = true;
        gameState.chapa12.enabled = true;
    }

    else if (idChapa.idEquipo == 2) {
        gameState.chapa1.enabled = true;
        gameState.chapa2.enabled = true;
        gameState.chapa3.enabled = true;
        gameState.chapa4.enabled = true;
        gameState.chapa5.enabled = true;

        gameState.chapa8.enabled = false;
        gameState.chapa9.enabled = false;
        gameState.chapa10.enabled = false;
        gameState.chapa11.enabled = false;
```

```
            gameState.chapa12.enabled = false;
        }
    }   //FIN de la función "cambiarTurno()"
```

9.4.10 Función "resetearJuego()"

La siguiente función es la encargada de restablecer los componentes del juego cuando esta tarea sea necesaria. Su trabajo consiste en: **situar** las chapas en el lugar en el que estaban colocadas inicialmente cuando el usuario comienza el juego, llamando para ello a la función "resetearPosiciones()", volver a poner los **marcadores** de los equipos **a cero**, y volver a **establecer** el **turno** de chapas original, en el que primero puede jugar el fútbol club Barcelona y seguidamente el Real Madrid, tal y como se establece al iniciar el juego. Esta función puede ser invocada en dos momentos distintos: cuando el usuario pulsa la flecha de vuelta atrás y confirma que desea abandonar la partida, o cuando el usuario ha marcado para uno de los equipos tres goles y desea iniciar una nueva partida.

```
function resetearJuego() {
    gameState.txtMarcador1.text = "0"
    gameState.txtMarcador2.text = "0"
    marcador1 = 0;
    marcador2 = 0;

    gameState.sound_gol.stop();

    empezarCrono = false;

    resetearPosiciones();

    gameState.chapa1.enabled = true;
    gameState.chapa2.enabled = true;
    gameState.chapa3.enabled = true;
    gameState.chapa4.enabled = true;
    gameState.chapa5.enabled = true;
    gameState.chapa8.enabled = false;
    gameState.chapa9.enabled = false;
    gameState.chapa10.enabled = false;
    gameState.chapa11.enabled = false;
    gameState.chapa12.enabled = false;
}
```

9.4.11 Función "resetearPosiciones()"

Este método, como se ha visto, es empleado por la función "resetearJuego()", ya que es el método que permite establecer todas las chapas en la posición del campo en las que se encontraban al empezar la partida. Para ello, la función hace un doble trabajo. Por un lado, coloca cada una de las chapas físicamente en el propio terreno de juego, haciendo uso de la variable "gameState" que permite acceder a los componentes *Qml* con los que se definen las chapas y cambiarles las propiedades *x* e *y*. Y, por otro lado, también es necesario modificar las posiciones de las chapas en el *array* explicado con

anterioridad, el *array* "ArrayDeChapas". Para realizar esto último, se hace uso de la variable "chapaAux". Como último detalle, el método también comprueba si alguno de los marcadores de un equipo u otro está a tres, ya que, en dicho caso, se deberá abrir el diálogo definido por el archivo "finishDialog.*Qml*" en el que se indica al usuario si desea comenzar un nuevo partido.

```
function resetearPosiciones(){
    gameState.chapa2.x = gameState.imgGameOn.width*0.05;
    gameState.chapa2.y = gameState.imgGameOn.height*0.45;
    gameState.chapa1.x = gameState.imgGameOn.width*0.2;
    gameState.chapa1.y = gameState.imgGameOn.height*0.10;
    gameState.chapa3.x = gameState.imgGameOn.width*0.2;
    gameState.chapa3.y = gameState.imgGameOn.height*0.45;
    gameState.chapa4.x = gameState.imgGameOn.width*0.2;
    gameState.chapa4.y = gameState.imgGameOn.height*0.75;
    gameState.chapa5.x = gameState.imgGameOn.width*0.35;
    gameState.chapa5.y = gameState.imgGameOn.height*0.45;
    gameState.chapa8.x = gameState.imgGameOn.width*0.9;
    gameState.chapa8.y = gameState.imgGameOn.height*0.45;
    gameState.chapa9.x = gameState.imgGameOn.width*0.75;
    gameState.chapa9.y = gameState.imgGameOn.height*0.10;
    gameState.chapa10.x = gameState.imgGameOn.width*0.75;
    gameState.chapa10.y = gameState.imgGameOn.height*0.45;
    gameState.chapa11.x = gameState.imgGameOn.width*0.75;
    gameState.chapa11.y = gameState.imgGameOn.height*0.75;
    gameState.chapa12.x = gameState.imgGameOn.width*0.6;
    gameState.chapa12.y = gameState.imgGameOn.height*0.45;

    //ACTUALIZACIÓN DE LAS CHAPAS EN EL ARRAY

chapaAux.getChapa(gameState.chapa1).setX(gameState.imgGameOn.width*0.2);
chapaAux.getChapa(gameState.chapa1).setY(gameState.imgGameOn.height*0.10;
chapaAux.getChapa(gameState.chapa2).setX(gameState.imgGameOn.width*0.05);
chapaAux.getChapa(gameState.chapa2).setY(gameState.imgGameOn.height*0.45;
chapaAux.getChapa(gameState.chapa3).setX(gameState.imgGameOn.width*0.2);
chapaAux.getChapa(gameState.chapa3).setY(gameState.imgGameOn.height*0.45;
chapaAux.getChapa(gameState.chapa4).setX(gameState.imgGameOn.width*0.2);
chapaAux.getChapa(gameState.chapa4).setY(gameState.imgGameOn.height*0.75;
chapaAux.getChapa(gameState.chapa5).setX(gameState.imgGameOn.width*0.35);
chapaAux.getChapa(gameState.chapa5).setY(gameState.imgGameOn.height*0.45;
chapaAux.getChapa(gameState.chapa8).setX(gameState.imgGameOn.width*0.9);
chapaAux.getChapa(gameState.chapa8).setY(gameState.imgGameOn.height*0.45;
chapaAux.getChapa(gameState.chapa9).setX(gameState.imgGameOn.width*0.75);
chapaAux.getChapa(gameState.chapa9).setY(gameState.imgGameOn.height*0.10;
chapaAux.getChapa(gameState.chapa10).setX(gameState.imgGameOn.width*0.75;
chapaAux.getChapa(gameState.chapa10).setY(gameState.imgGameOn.height*0.4;
chapaAux.getChapa(gameState.chapa11).setX(gameState.imgGameOn.width*0.75;
chapaAux.getChapa(gameState.chapa11).setY(gameState.imgGameOn.height*0.7;
chapaAux.getChapa(gameState.chapa12).setX(gameState.imgGameOn.width*0.6);
chapaAux.getChapa(gameState.chapa12).setY(gameState.imgGameOn.height*0.4;
    if (marcador1 == 3 || marcador2 == 3) {
        Qt.createComponent("../finishDialog.Qml").createObject(gameState);
}
}
```

9.4.12 Función "goBack ()"

Cuando los jugadores pulsan la flecha de vuelta atrás en el juego y, a continuación, indican en el diálogo que les aparece que desean finalizar la partida, o bien cuando han llegado al límite de goles en uno de los dos equipos e indican que ya no desean jugar más, se hace una invocación a la función "goBack()". En esta función, únicamente se hace una modificación sobre el estado del juego, indicando que ahora el juego se encuentra en la pantalla de inicio y no en la de juego, y también se modifica el estado de la variable "chapaActual", ya que a partir de entonces no habrá ninguna chapa que se esté moviendo.

```
function goBack() {
    gameState.state = "";
    chapaActual.state = "";
}
```

9.4.13 Función "tick ()"

Esta es una de las funciones que, como se vio en el archivo donde se define parte de la interfaz del juego "main.qml", se ejecuta periódicamente cada dieciséis milisegundos desde que el juego da comienzo. Esta función, de manera resumida, es la que se encarga de que la chapa que ha pulsado el usuario, y que ha decidido mover, se **mueva** progresivamente a lo largo del eje *x* y del eje *y*, moviéndose con un cierto ángulo que ha sido determinado previamente. Así pues, lo que ocurre es que cada dieciséis milisegundos la chapa designada como "chapaActual" cambia el valor de su propiedad *x* y el de su propiedad *y*.

En primer lugar, por razones de eficiencia, se comprobará si existe, en cada intervalo en el que se ejecuta la función, si hay en estos momentos una "chapaActual" en el campo, de manera que si no la hay la función finalizará su ejecución inmediatamente. Es importante recordar que la variable "chapaActual" tomará como valor, en principio, la chapa que ha pulsado el usuario, pero si esta chapa se chocara con alguna en su recorrido, el valor de la "chapaActual" pasaría a ser el de la chapa chocada. Además, en estas primeras instrucciones de la función también se analizarán las variables "anguloX" y "anguloY" antes calculadas, para que se conviertan a valores positivos en caso de que su resultado haya sido negativo, ya que sino se tendrían movimientos no deseados.

```
function tick(){
    if (chapaActual == null || chapaActual.state == "") {
        gol = false;
        return;
    }
    chapaAux = new Chapa(chapaActual, null, null, signoXDetectado,
signoYDetectado);
    if (anguloX < 0)  anguloX *= (-1);
    if (anguloY < 0)  anguloY *= (-1);
    chapaActual.enabled = false;
    //SIGUE
```

A continuación se comienza a llevar a cabo el movimiento que ha indicado el usuario sobre la chapa. Para esto, primero se determina si el usuario ha decidido hacer un movimiento en **recto** con la chapa **o no**, analizando para ello la variable global "recto". Sea cual sea el movimiento, el desplazamiento de la chapa actual, representado por la variable "chapaActual", tendrá lugar hasta que:

- Ha llegado a su destino en el eje *x* y a su destino en el eje *y*, en el caso de un movimiento no recto. O ha llegado a su destino en el eje *x* en el caso de un movimiento recto.

- O bien, la chapa se ha metido en la portería contraria marcando un gol.

```
if (recto != true) {
        if (signoXDetectado == "-" && signoYDetectado == "-") {
            chapaActual.x -= anguloX*5;
            chapaActual.y -= anguloY*5;
            if ((chapaActual.x < destinoX && chapaActual.y <
destinoY)|| gol == true) {
                chapaActual.state = "";
                cambiarTurno(chapaElegida);
            }
        }
        else if (signoXDetectado == "+" && signoYDetectado == "-") {
            chapaActual.x += anguloX*5;
            chapaActual.y -= anguloY*5;
            if ((chapaActual.x >= destinoX && chapaActual.y <=
destinoY) || gol == true) {
                chapaActual.state = "";
                cambiarTurno(chapaElegida);
            }
        }
        else if (signoXDetectado == "-" && signoYDetectado == "+") {
            chapaActual.x -= anguloX*5;
            chapaActual.y += anguloY*5;
            if ((chapaActual.x <= destinoX && chapaActual.y >=
destinoY) || gol == true) {
                chapaActual.state = "";
                cambiarTurno(chapaElegida);
            }
        }
        else if (signoXDetectado == "+" && signoYDetectado == "+") {
            chapaActual.x += anguloX*5;
            chapaActual.y += anguloY*5;
            if ((chapaActual.x >= destinoX && chapaActual.y >=
destinoY) || gol == true) {
                chapaActual.state = "";
                cambiarTurno(chapaElegida);
            }
        }
    } //FIN de movimiento no recto
```

En caso de que el usuario haya indicado un movimiento recto para la chapa, solo la propiedad *x* de la chapa actual se verá afectada. Para ambos tipos de movimiento, rectos o en diagonal, se hace una comprobación de las variables "signoXDetectado" y

"signoYDetectado" y así, en función de los valores de estas, la chapa toma la dirección que ha indicado el usuario. Como últimas instrucciones de la función, es fundamental **actualizar** los valores de la chapa que se ha movido en el *array* "ArrayDeChapas", así que sus propiedades *x* e *y* son actualizadas gracias a los métodos "set" de la clase "Chapa".

```
else {
        if (signoXDetectado == "+") {
            chapaActual.x += 5;
            if (chapaActual.x >= destinoX || gol == true) {
                chapaActual.state = "";
                recto = false;
                cambiarTurno(chapaElegida);
            }
        }
        else {
            chapaActual.x -= 5;
            if (chapaActual.x <= destinoX || gol == true) {
                chapaActual.state = "";
                recto = false;
                cambiarTurno(chapaElegida);
            }
        }
    }  //FIN del movimiento en recto
    if (chapaActual != null) {   //Actulización en el array
        chapaAux.getChapa(chapaActual).setX(chapaActual.x);
        chapaAux.getChapa(chapaActual).setY(chapaActual.y);
    }
}   //FIN de la función "tick()"
```

9.4.14 Función "tickDetectarColisiones ()"

Esta función, como ocurre con la anterior, también se ejecuta mediante intervalos de dieciséis milisegundos. En esta ocasión, se trata de una función que pretende detectar las posibles colisiones entre las chapas. Para ello, comprobará si la chapa que representa la "chapaActual" ha colisionado con alguna de las otras chapas, es decir, comprobará si hay conflicto entre las propiedades *x* e *y* de la variable "chapaActual" y el resto de chapas del terreno de juego.

En la detección de colisiones de las chapas es necesario, como era de esperar, recorrer la variable "ArrayDeChapas", donde se almacena cada una de las chapas del terreno de juego, y donde se puede consultar para cada una de ellas sus propiedades *x* e *y* que indican en qué posición están. Es necesario también tener en cuenta el ancho y alto que ocupan las chapas, y es por eso que se emplean dos variables "height" y "width", representando el ancho y alto respectivamente. En caso de que se produzca una colisión entre la chapa que representa la "chapaActual" y alguna chapa del *array*, excluyendo la propia "chapaActual" ya que eso siempre produciría colisión, la que había sido definida hasta entonces como "chapaActual" dejará de moverse, por lo que se modificará su propiedad "*state*" a vacío. Consecuentemente, la variable "chapaActual" tomará un nuevo valor, y ese será el que viene dado por la chapa del *array* con la que se

chocó la anterior chapa, empezando a moverse la nueva. Además, es necesario **recalcular** las variables que representaban el **destino** en eje *x* y en eje *y* de la que era "chapaActual" hasta el choque, ya que al haberse producido este no podrá llegar al destino que tenía previsto.

```
function tickDetectarColisiones() {
    if (chapaActual == null || chapaActual.state == "")
        return;
    var i = 0;
    var chapaArray = null;
    var x, y;
    var height = gameState.chapa2.height;
    var width = gameState.chapa2.width;
    while (i < ArrayDeChapas.length) {
        chapaArray = ArrayDeChapas[i];
        x = chapaArray.getX();
        y = chapaArray.getY();
        if (chapaArray.getID() != chapaActual ) {
            if (chapaActual.x < x + width && chapaActual.x + width >
x && chapaActual.y < y + height && chapaActual.y + height > y) {
     chapaActual.state = "";
     destinoX *= 0.85;
     destinoY *= 0.85;
     chapaActual = chapaArray.getID();
     chapaActual.state = "moviendo";  //chapa nueva se mueve
                 break;
             }
         }
         i++;
    }
}
```

9.4.15 Función "tickDetectarGol ()"

A la vez que las chapas se están moviendo por el terreno de juego, es necesario determinar si en algún momento alguna de ellas ha dado lugar a un gol. Es por ello que existe otra función que se ejecuta, una vez más, periódicamente. En este caso, se tendrá que comprobar primero cuál es el equipo de la chapa que se ha metido en alguna portería, comprobando para ello la propiedad "idEquipo". Una vez hecho esto, se comprobará si la chapa que viene dada como "chapaActual" se encuentra en ciertos límites del campo, concretamente en los límites que marcan una portería y otra, y es por eso que se usan algunos valores finales representando estos límites (747, 162, 200, 9). Debido a eso, se hace uso de las propiedades *x* e *y* de la chapa. Dicho esto, en caso de que haya existido gol, será necesario reproducir un sonido, se subirá el marcador del equipo que ha marcado y, al final, se resetearán las posiciones de las chapas de ambos equipos.

```
function tickDetectarGol() {
    if (chapaActual == null || chapaActual.state == "")
        return;
    if (chapaActual != null) {
        if (chapaActual.idEquipo == 1) {
            if (chapaActual.x >= 747 && chapaActual.y >= 162 &&
chapaActual.y <= 200 && gol != true) {
                gameState.sound_gol.play();
                gol = true;
                marcador1++;
                gameState.txtMarcador1.text = marcador1;
                resetearPosiciones();
            }
        }
        if (chapaActual.idEquipo == 2) {
            if (chapaActual.x <= 9 && chapaActual.y >= 162 &&
chapaActual.y <= 200 && gol != true) {
                gameState.sound_gol.play();
                gol = true;
                marcador2++;
                gameState.txtMarcador2.text = marcador2;
                resetearPosiciones();
            }
        }
    }
}
```

9.4.16 Función "tickDetectarLimites ()"

En el terreno de juego por el que se mueven las chapas también juegan un papel importante los límites del mismo. Las chapas nunca deben salirse del terreno de juego ya que así ocurre en el juego tradicional de las chapas, es decir que los límites del campo son un factor a tener en cuenta. Para comprobar esto, se hace uso de algunos valores finales y de constantes como puedan ser los *arrays* de "limitesPorteriasY" o "limitesCampoX" que representan los propios límites del campo a través de valores enteros. Con esto, en caso de que un choque con algún límite tenga lugar habrá que tener varias cosas en cuenta:

- Las variables "signoXDetectado" y "signoYDetectado", ya que la chapa tendrá que **cambiar de dirección** según el límite en el que se haya chocado y la dirección con la que venía.

- El destino de la chapa también se verá afectado al haberse chocado la chapa con algún límite.

```
function tickDetectarLimites() {
    if (chapaActual == null || chapaActual.state == "")
        return;
    //Límite superior
    if (chapaActual.y <= 1){
        signoYDetectado = "+";
        return;
    }
    //SIGUE
```

```
//Límite inferior
    if (chapaActual.y >= 365){
        signoYDetectado = "-";
        return;
    }

    //Límite portería izquierda
    if (chapaActual.x <= limitesCampoX[0]){
        if (chapaActual.y <= limitesPorteriasY[0] || chapaActual.y >
limitesPorteriasY[1]) {
            signoXDetectado = "+";
            destinoX *= (-0.5);
            return;
        }
    }

    //Límite portería derecha
    if (chapaActual.x >= limitesCampoX[1]){
        if (chapaActual.y <= limitesPorteriasY[0] || chapaActual.y >
limitesPorteriasY[1]) {
            signoXDetectado = "-";
            destinoX /= 2;
            return;
        }
    }

    //Límite detrás portería izquierda
    if (chapaActual.x <= 0)
        signoXDetectado = "+";

    //Límite detrás portería derecha
    if (chapaActual.x >= 760)
        signoXDetectado = "-";

}   //FIN de la función "tickDetectarLimites"
```

9.4.17 Función "actualizarCrono ()"

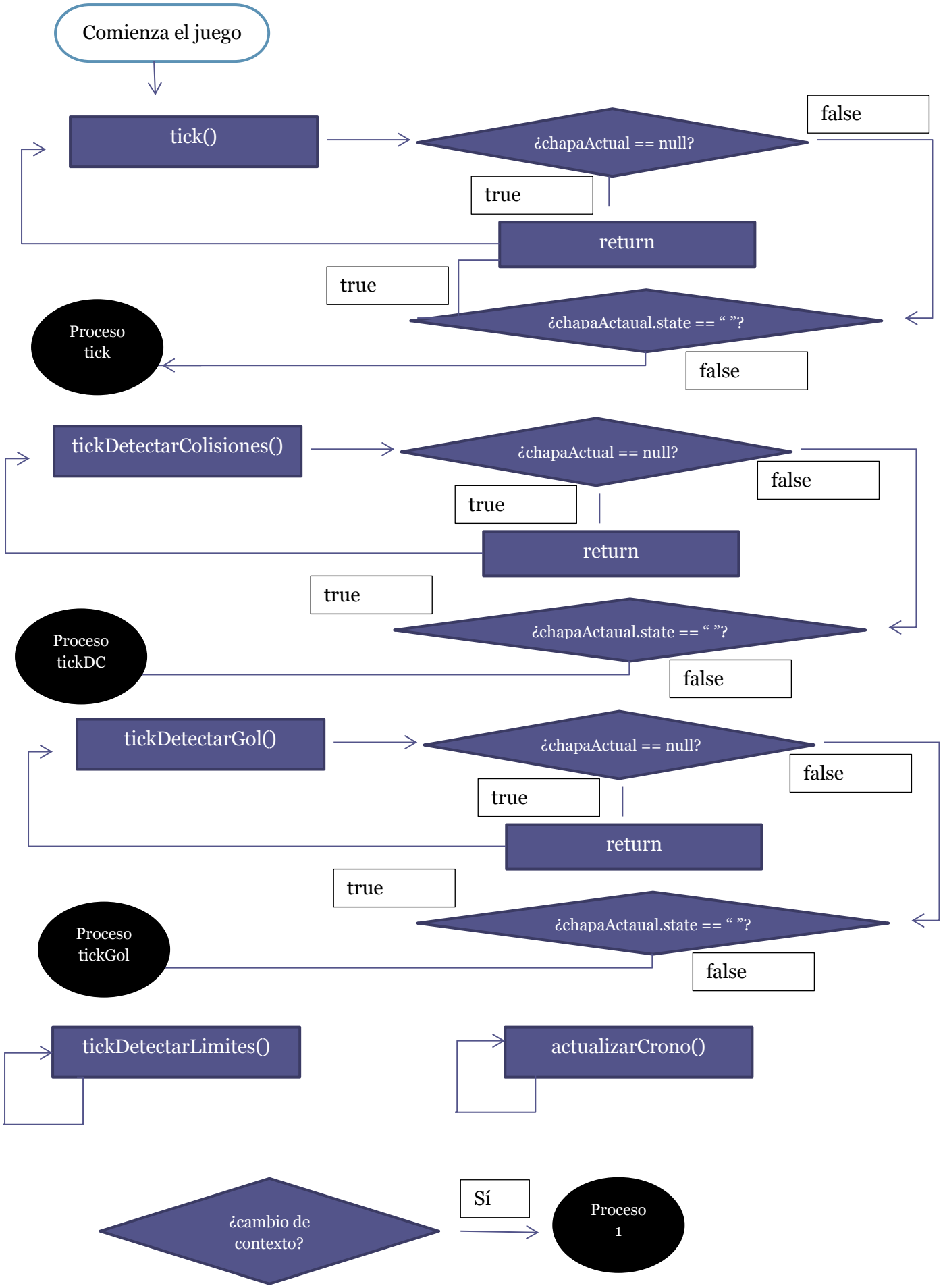
Esta función es la encargada de llevar la cuenta del reloj del partido. Cuando el usuario pulsa el botón de jugar en la pantalla de inicio, será movido a la pantalla de juego, donde verá que un reloj, arriba a la derecha de esta pantalla, le irá indicando cuánto tiempo lleva de juego.

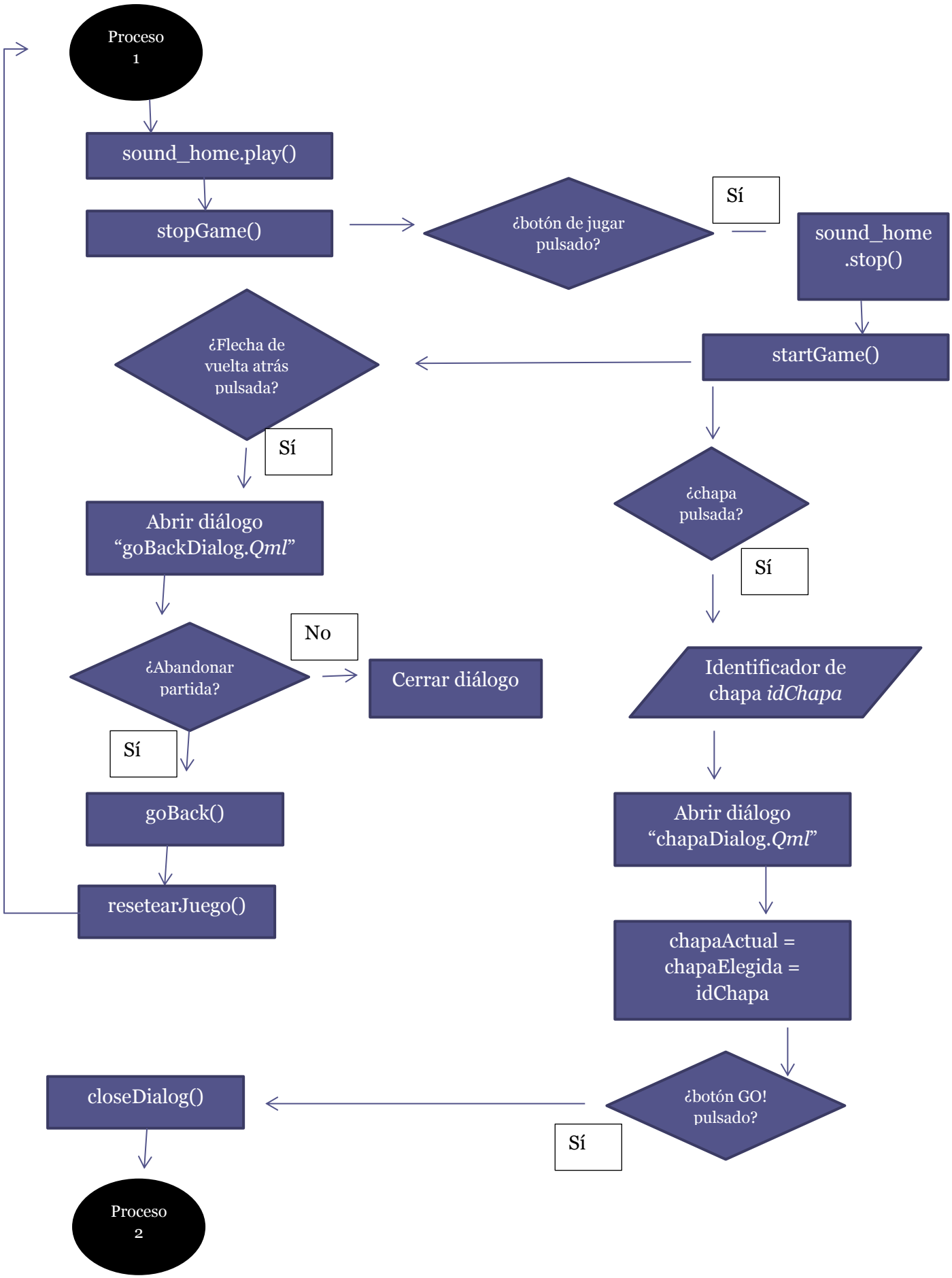
Para implementar esta funcionalidad, fue necesario emplear la variable "empezarCrono". En realidad, la función se está ejecutando durante todo el juego a intervalos de un segundo, pero es la anterior variable booleana la que indica cuándo el cronometro se empieza a modificar. Además, se utilizan también las variables "segundos" y "minutos", ya que son las que permiten llevar la cuenta del tiempo actual y las que, atendiendo a sus valores, reflejan cómo se modificará el crono de la interfaz de juego. Con esto, para modificar el crono de partido se tuvo que acceder a la propiedad "txtCrono" del archivo "main.qml".

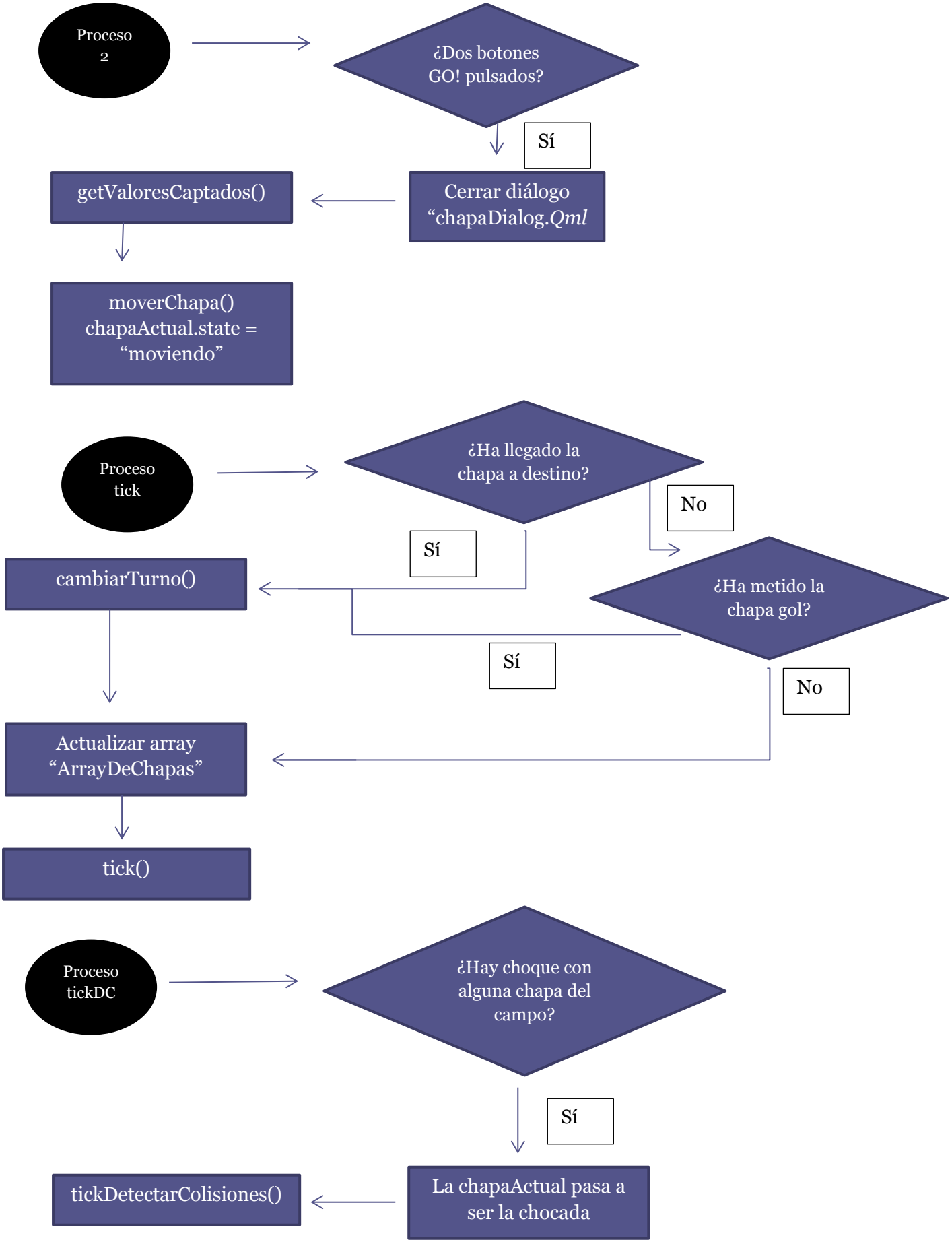
```
function actualizarCrono(){
    if (!empezarCrono) {
        gameState.txtCrono.text = "00:00";
        segundos = 0;
        minutos = 0;
        return;
    }

    if (segundos < 59) {
        segundos++;
        if (segundos.toString().length < 2) {
            gameState.txtCrono.text =
gameState.txtCrono.text.charAt(0)+

gameState.txtCrono.text.charAt(1)+":"+"0"+segundos.toString();
        }
        else {
            gameState.txtCrono.text =
gameState.txtCrono.text.charAt(0)+

gameState.txtCrono.text.charAt(1)+":"+segundos.toString();
        }
    }
    else {
        segundos = 0;
        minutos++;
        if (minutos.toString().length < 2) {
            gameState.txtCrono.text = "0"+minutos.toString()+":"+"0"+
                    segundos.toString();
        }
        else {
            gameState.txtCrono.text = minutos.toString()+":"+
                    segundos.toString();
        }
    }
}
```

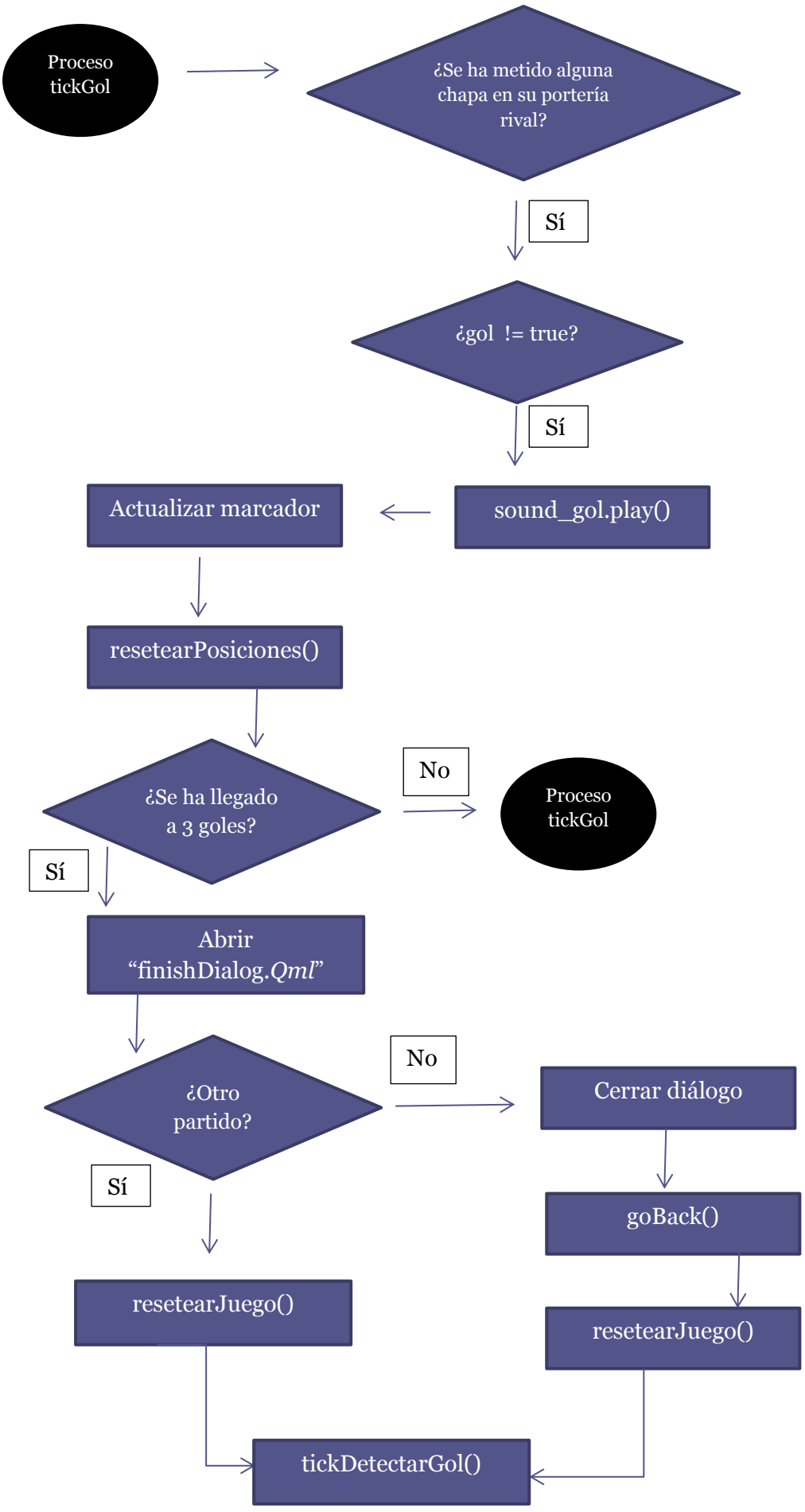
9.5 Diagrama de flujo

Resulta útil reflejar, a través de lo que se conoce como diagrama de flujo, cómo es la dinámica del juego, es decir, cómo se van ejecutando las diferentes funciones que se han ido explicando, en qué momentos se ejecutan, cuándo finalizan, etc. Esto permitirá distinguir con mejor claridad el sentido que tiene cada función que se ha implementado en el juego desarrollado. Un diagrama de flujo no consiste más que en un esquema con el que representar gráficamente un algoritmo. Permite esquematizar las distintas operaciones que tienen lugar en un trabajo o función mediante diversos símbolos, indicando así cómo es el propio flujo del trabajo.


Comienza el juego
tick()
¿chapaActual == null?
false
true
return
true
¿chapaActaual.state == “ ”?
Proceso tick
false
tickDetectarColisiones()
¿chapaActual == null?
false
true
return
true
¿chapaActaual.state == “ ”?
Proceso tickDC
false
tickDetectarGol()
¿chapaActual == null?
false
true
return
true
¿chapaActaual.state == “ ”?
Proceso tickGol
false
tickDetectarLimites()
actualizarCrono()
¿cambio de contexto?
Sí
Proceso 1


Proceso 1
sound_home.play()
stopGame()
¿botón de jugar pulsado?
Sí
sound_home .stop()
startGame()
¿Flecha de vuelta atrás pulsada?
Sí
Abrir diálogo "goBackDialog.Qml"
¿Abandonar partida?
No
Cerrar diálogo
Sí
goBack()
resetearJuego()
¿chapa pulsada?
Sí
Identificador de chapa idChapa
Abrir diálogo "chapaDialog.Qml"
chapaActual = chapaElegida = idChapa
¿botón GO! pulsado?
Sí
closeDialog()
Proceso 2


Proceso 2
¿Dos botones GO! pulsados?
Sí
Cerrar diálogo "chapaDialog.Qml
getValoresCaptados()
moverChapa()
chapaActual.state = "moviendo"
Proceso tick
¿Ha llegado la chapa a destino?
No
Sí
cambiarTurno()
¿Ha metido la chapa gol?
Sí
No
Actualizar array "ArrayDeChapas"
tick()
Proceso tickDC
¿Hay choque con alguna chapa del campo?
Sí
La chapaActual pasa a ser la chocada
tickDetectarColisiones()


Proceso tickGol
¿Se ha metido alguna chapa en su portería rival?
Sí
¿gol != true?
Sí
sound_gol.play()
Actualizar marcador
resetearPosiciones()
¿Se ha llegado a 3 goles?
No
Proceso tickGol
Sí
Abrir "finishDialog.Qml"
¿Otro partido?
No
Cerrar diálogo
Sí
goBack()
resetearJuego()
resetearJuego()
tickDetectarGol()

10. PRUEBAS

Sin lugar a dudas, la fase de realización de pruebas fue fundamental en la consecución y completitud del proyecto. Cuando hacíamos cambios importantes que afectaban de manera radical la lógica de la aplicación, era muy importante realizar test que garantizaran que dichos cambios habían sido satisfactorios. De la misma manera, durante la implementación de la interfaz del juego también tenían lugar pruebas que permitieran ver que el aspecto del juego era el deseado, que los componentes de la interfaz estaban colocados donde se pretendía, etc.

Las pruebas que se iban realizando en el proyecto tuvieron lugar en dos niveles distintos. Por un lado, a través de un emulador que incorpora el entorno de desarrollo *Qt*, podíamos comprobar de una manera rápida y directa cómo era la ejecución de la aplicación en un sistema de **escritorio**. En este caso el emulador hace uso del kit *Desktop* teniendo en cuenta el sistema operativo de escritorio en el que se está ejecutando la aplicación. Estas pruebas realizadas a nivel de escritorio eran, ciertamente, más rápidas que las que se realizaban sobre dispositivos móviles, así que nos permitían realizar rápidas comprobaciones cuando los cambios en el código no eran tan significativos. En la siguiente ilustración, se puede ver cómo se ejecuta el juego desarrollado en el emulador para escritorio que incorpora *Qt*, empleando en este caso un entorno para sistemas Mac.

Ilustración 52 – Ejecución del proyecto en un sistema Mac OS X.

De esta manera, las pruebas que se realizaron sobre los distintos sistemas operativos de escritorio como *Windows*, *Mac OS X* o *Ubuntu*, tuvieron un

procedimiento idéntico. Cuando se quería probar la aplicación para un sistema determinado se hacía uso del entorno *Qt* para, a través de los kit que el entorno incluía, generar los archivos ejecutables necesar*ios* para el sistema operativo en el que se quería ejecutar el juego. Para todos estos sistemas de escritorio, el entorno *Qt* siempre empleaba un kit de tipo *Desktop*.

Un caso aparte es el que tenía lugar con la realización de pruebas en dispositivos **móviles**. Para ello, contamos con la posibilidad de tener en nuestras manos tanto un dispositivo con sistema *Android* como un dispositivo *Jolla* cuyo soporte era un sistema operativo *Sailfish*, aunque también se podía emplear, en esta plataforma, un emulador configurado como una máquina virtual de *Sailfish* sobre *Virtual Box Oracle*. También contamos con la posibilidad de ver si el juego funcionaría, y cómo lo haría, en dispositivos *iOS* gracias a un emulador.

10.1 Pruebas en Android

En el primer caso, pudimos ejecutar nuestra aplicación sobre un móvil Samsung Galaxy Express GT – I8730 que estaba dotado de un sistema operativo *Android* 4.1.2. En esta ilustración se puede ver la ejecución en este dispositivo.

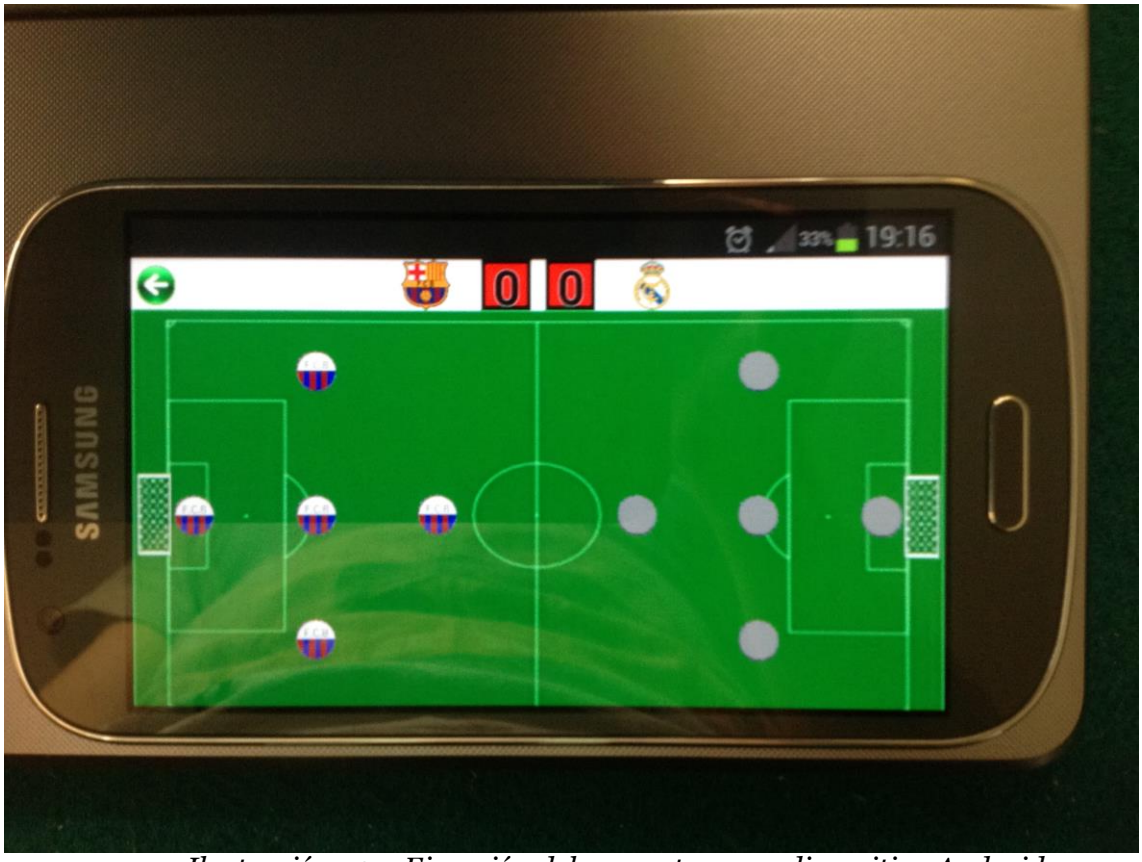


Ilustración 53 – Ejecución del proyecto en un dispositivo Android.

Las maneras de afrontar la ejecución de la aplicación sobre un dispositivo real son diversas. En el caso de desplegar dicha aplicación sobre un sistema *Android*, una de

las maneras puede consistir en desplegar el fichero ".apk", el cual se genera al compilar y construir la aplicación, sobre nuestro dispositivo. Para ello será necesario conectar nuestro dispositivo al ordenador, buscar el directorio del proyecto *Qt* en el que se ha generado el archivo ".apk" y copiarlo a nuestro móvil. Sin embargo, otra de las maneras de afrontar esto puede llevarse a cabo a través del entorno *Qt*. En este caso se debe conectar el dispositivo al ordenador, teniendo el entorno *Qt* en marcha. El siguiente paso consiste en indicar el kit de construcción y de ejecución adecuado a dicho dispositivo para que se genere el ejecutable y se pueda ejecutar en este. En el caso de *Qt for Android* son tres los kit que vienen incorporados para generar código ejecutable para sistemas *Android*, y es el propio entorno *Qt* quien nos indica si el dispositivo es compatible o no para un kit seleccionado. Cuando el dispositivo sea compatible, lo cual se determina atendiendo a la arquitectura de su procesador, se podrá generar el código pertinente y ejecutar sobre el dispositivo. En la siguiente ilustración, se aprecia el diálogo que aparece tras seleccionar un kit compatible con nuestro dispositivo tras conectarlo al ordenador. Después de pulsar "OK" comenzaría la instalación y ejecución en nuestro dispositivo.

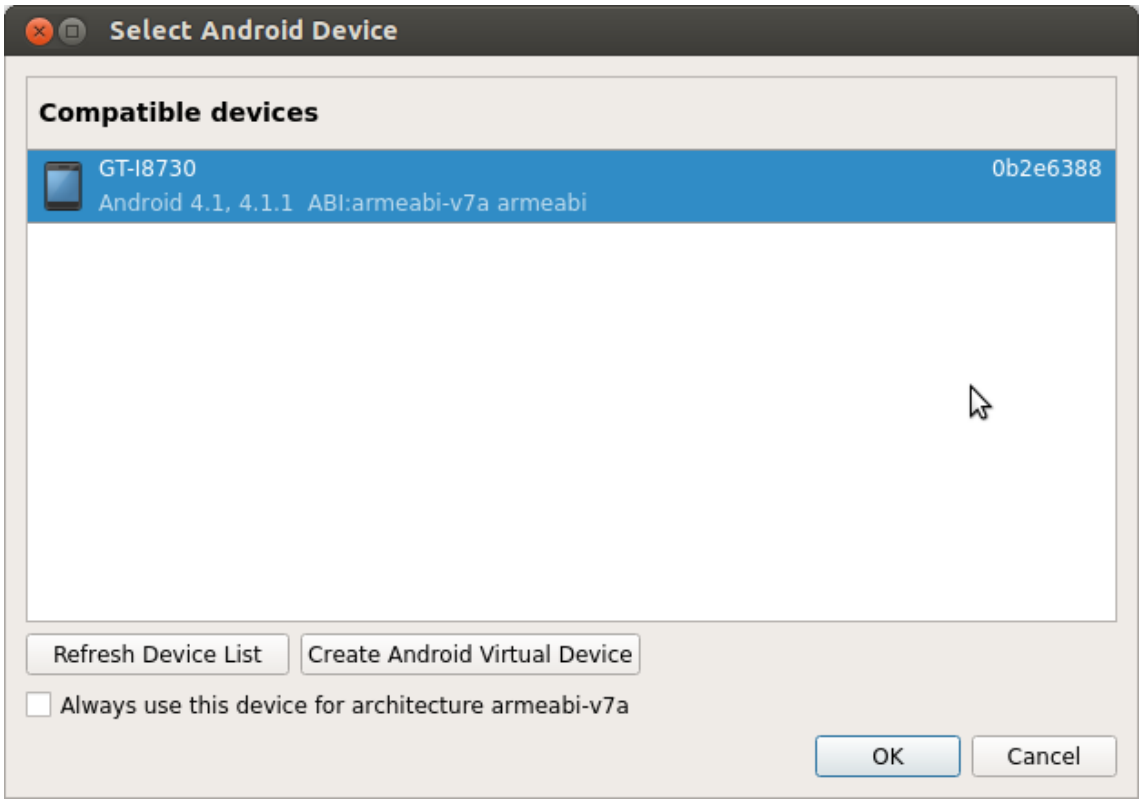


Ilustración 54 – Diálogo del entorno Qt al intentar ejecutar sobre un dispositivo real.

10.2 Pruebas en Sailfish

Para afrontar la ejecución de la aplicación que estábamos desarrollando, podíamos seguir dos caminos diferentes en este tipo de plataforma. Por un lado, existía la alternativa de ejecutar el juego en un dispositivo que tuviera como soporte el sistema operativo *Sailfish*, como ocurre, por ejemplo, en el caso de los móviles de la marca

Jolla. Pero por otro lado, también podíamos comprobar que nuestro juego funcionaba en la plataforma *Sailfish*, y ver además cómo lo hacía, gracias a un simulador que viene incorporado en el *Sailfish* IDE. Se comentará ahora, pues, cómo fueron las pruebas realizadas en el simulador de *Sailfish*.

Para configurar un simulador de dispositivo con soporte *Sailfish*, el entorno de desarrollo *Sailfish* IDE exige contar con un entorno de máquina virtual *Virtual Box*, el cual está desarrollado por la compañía *Oracle*. Con esto, se creará una máquina virtual de *Sailfish* en el entorno *Virtual Box* de *Oracle*. Para ello, el SDK de *Sailfish* emplea, en el entorno de programación *Sailfish IDE*, una máquina *Mer*, una distribución de *software* libre en la que está basada el sistema operativo *Sailfish* OS. La máquina *Mer* es, precisamente, la encargada de compilar el código del programa que se ha desarrollado, y obtener de esta manera el ejecutable que permite probar nuestra aplicación en un sistema *Sailfish*. Así, antes de lanzar a ejecución el proyecto, el usuario deberá arrancar esta máquina de construcción *Mer* en el entorno *Sailfish* IDE, véase la siguiente ilustración.

Ilustración 55 – Botón que se debe pulsar para arrancar la máquina Mer.

Hecho esto, el siguiente paso que se debe hacer es arrancar la máquina virtual de *Sailfish*. La primera vez que esta máquina se arranca permitirá que se cree y configure, automáticamente, una máquina virtual *Sailfish* sobre el entorno *Virtual Box Oracle*, siempre que lo tengamos instalado en nuestro ordenador de trabajo. Con esto, para arrancar la máquina virtual en la que se simula un dispositivo con sistema *Sailfish*, tanto la primera vez como el resto de las veces, hay que pulsar el botón que se indica en la ilustración 51.

Una vez arrancadas estas dos partes, ya se puede ejecutar el juego en sí para testear que este puede ser desplegado en sistemas *Sailfish* y también para comprobar

que todo funciona correctamente y como en el resto de plataformas para las que se probó el juego.

.

Ilustración 56 – Botón que se debe pulsar para arrancar la máquina virtual de Sailfish que se configura en Virtual Box Oracle.

*Ilustración 57 – Ejecución del juego en el **SailfishOS Emulator** que se configura en el entorno Virtual Box Oracle.*

10.3 Pruebas en iOS

Para probar el juego desarrollado en el sistema operativo móvil *iOS*, se ha trabajado con simuladores, una vez más, ya que no se pudo desplegar el proyecto en dispositivos reales por razones que se describen en el apartado "Problemas en el desarrollo". Dichos simuladores se encuentran incorporados en la herramienta de desarrollo conocida como *XCode*, *software* propietario de la marca *Apple*.

La versión de *XCode* con la que trabajamos para el proyecto constaba de un simulador de *iPad* en su versión de sistema operativo *iOS* 6.1, y de un *iPhone*, también en la misma versión.

Para realizar las pruebas, en este caso no hace falta arrancar una máquina virtual previamente o acciones similares, sino que basta con pulsar el botón de "Run" sobre la herramienta *XCode* una vez que hemos abierto nuestro proyecto *Qt* sobre dicha herramienta. Véase la ilustración 53.

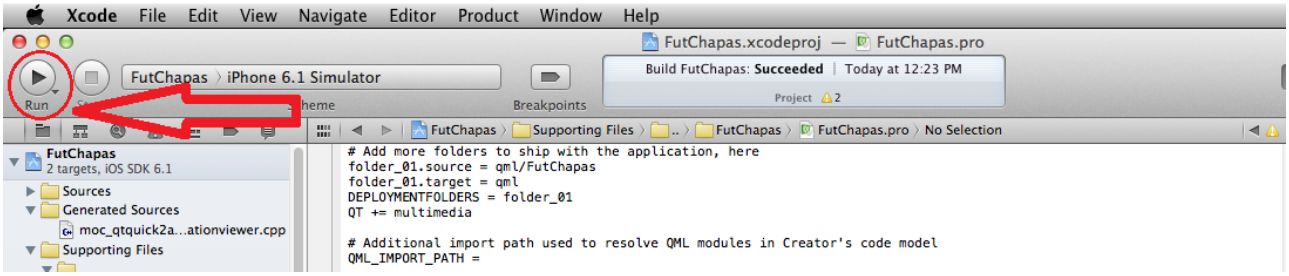


Ilustración 58 – Arrancando el juego en el entorno de programación XCode.

Al arrancar la aplicación, el simulador seleccionado, en este caso un *iPhone* 6.1, comenzará a desplegarse, mostrando a su vez la aplicación que en él se ha ejecutado.

Ilustración 59 – Ejecución del proyecto en un simulador de dispositivo con sistema operativo iOS.

11. PROBLEMAS EN EL DESARROLLO

A lo largo del proyecto, y a medida que avanzábamos en él, surgían diferentes contratiempos y dificultades que teníamos que afrontar. Hubo problemas relacionados con aspectos de implementación, problemas en el despliegue y ejecución del juego en dispositivos reales, problemas al inicio del proyecto con ciertos entornos de trabajo, etc. Por esto, todos estos detalles se comentan de forma detenida en esta sección.

11.1 Problemas iniciales

Desde el principio del proyecto, teníamos en mente la posibilidad de portarlo a varias plataformas. Como el título de este proyecto indica, estábamos, sobretodo, interesados en poder llevar el proyecto a sistemas operativos **móviles**, pero cuando vimos las facilidades que nos ofrecía la tecnología *Qt*, decidimos que también podía ser una buena opción portar el proyecto a sistemas operativos de **escritorio**.

Los sistemas operativos móviles que nos interesaron desde un principio eran bien diferentes y cada uno tenía sus aspectos característicos. Llegado el momento de comenzar el proyecto, sabíamos con certeza que la tecnología *Qt* estaba, para entonces, dando soporte tanto a los sistemas operativos *Android* como *Sailfish*. Sin embargo, no estábamos seguros, en ese entonces, de que el soporte para plataformas *iOS* fuera total. Así pues, fue con esta plataforma, *iOS*, con la que tuvimos algunos problemas iniciales. Queríamos, desde el principio, ejecutar nuestro juego en dispositivos **reales** *iOS*. Esto no pudo ser así.

Estos problemas no venían dados por la tecnología *Qt* sino que provenían de la tecnología *Apple*. Para poder ejecutar aplicaciones en un dispositivo *iOS* **real**, es necesario estar unido al programa de desarrolladores de *iOS*, el cual tiene un precio de cien dólares al año. Unirse a dicho programa es obligatorio ya que eso nos permite configurar un certificado de desarrolladores, el cual es necesario a la hora de desplegar las aplicaciones que hacemos en un dispositivo real. No estábamos dispuestos a pagar ese precio, ya que estábamos realizando un único proyecto y no sabíamos si íbamos a seguir desarrollando para la tecnología *iOS* en un futuro. Además, el proyecto no iba destinado a esa única plataforma. Sin embargo, seguíamos contando con la posibilidad de poder utilizar los **simuladores** de dispositivos *iOS* para testear nuestra aplicación, sin necesidad de contar con ningún certificado o similares. Por ello, esto nos obligó a no descartar la tecnología *iOS* en nuestro proyecto, ya que al menos podríamos utilizar simuladores de esta y garantizar así que el proyecto podía ejecutarse en este tipo de plataforma.

11.2 Otros problemas

Otra serie de problemas venían dados por aspectos que tenían que ver con la implementación del juego, o por detalles que hacían que ciertas características del juego se perdieran cuando ejecutábamos sobre un dispositivo real y no sobre un sistema de escritorio.

Una vez que tuvimos implementados los aspectos de la interfaz del juego, habiendo definido todos los componentes que se especificaron en la fase de diseño y viendo que el cambio de pantallas se hacía sin problemas, pudimos ver que había problemas de sonido cuando el juego se ejecutaba en un dispositivo móvil. Lo que ocurría es que el sonido de la pantalla de inicio no se reproducía en dispositivos móviles, y, por el contrario, cuando ejecutábamos el juego en un entorno de escritorio o sobre el emulador incorporado en *Qt*, el sonido se reproducía sin problemas. Tuvimos que atacar este problema desde dos partes. Parecía que el problema venía dado por la librería *QtMultimedia 5.0*, la cual, como ya se comentó en el apartado de "Implementación", se tenía que añadir dinámicamente al crear objetos de tipo *SoundEffect*. Así pues, tuvimos que añadir una sentencia de tipo "*QT* += multimedia" en la parte de arriba del fichero ".pro" de nuestro proyecto o después de algún comando del tipo "*QT* +=". Y, por otra parte, también se tuvo que modificar el archivo *SoundEffect.qml* que teníamos desde el principio. Era necesario **importar** el directorio desde el cual se invocaba a este tipo de componente. En efecto, era desde el archivo "main.*qml*" desde donde se creaban componentes de tipo *SoundEffect.qml*, así que debíamos importar también el directorio en el que se encontraba contenido este archivo. De esta forma, el archivo *SoundEffect.qml* vio cómo se modificaba una de sus líneas de esta manera:

```
property QtObject effect: Qt.createQmlObject("import
QtMultimedia 5.0; import '../'; SoundEffect{ source: '" +
container.source + "' }", container);
```

Surgieron más problemas que también tenían que ver con la pérdida de características del juego cuando este se ejecutaba en un dispositivo real. En esta ocasión, los problemas venían dados por los diálogos que habíamos ido creando para el juego. Como se vio en el apartado de "Implementación", el juego constaba de una serie de diálogos, algunos eran diálogos simples de tipo *MessageDialog* y otros, como ocurría con el diálogo que aparecía al pulsar una chapa, eran diálogos personalizados. Con los diálogos de tipo *MessageDialog* no había ningún problema. Este tipo de diálogos funcionaba de manera correcta tanto en sistemas de escritorio como cuando la ejecución se hacía en dispositivos móviles.

Fue al terminar de implementar el diálogo de "chapaDialog.*qml*", cuando detectamos un problema al ejecutar el juego en un móvil. En este caso, lo que pasaba era que cuando el usuario pulsaba una chapa que deseaba mover, el diálogo no aparecía y, además, la aplicación generaba una excepción y paraba de ejecutarse. Así pues,

comenzamos en ese momento a hacer un tratamiento de dicha excepción para intentar solucionar el problema. En un principio se intentó modificar el fichero ".pro" del proyecto porque pensamos que algo podía faltar, pero ninguna de estas modificaciones resultó ser la solución del problema. También pensamos que, en el caso de dispositivos *Android*, podía faltar algo en el archivo "AndroidManifest.xml" que impedía que el diálogo se visualizase y que hacía que se parase de ejecutar la aplicación. Tampoco arregló nada. Tras varios intentos, mirando de forma más detenida la documentación oficial de *Qt* pudimos encontrar la solución. Al principio, el archivo "chapaDialog.q*ml*" que estábamos implementando tenía un aspecto como este:

```
Item {
    Window {
        id: window
        visible: true
        modality: Qt.ApplicationModal
        width: 500
        height: 220
        flags: Qt.AlignCenter
        flags: Qt.Dialog
        //SIGUE
        …
       }  //FIN de Window
    } //FIN Item y del archivo
```

Según vimos en la documentación de *Qt*, un elemento de tipo *Window* se puede declarar dentro de un *Item* o dentro de otro elemento *Window*. En esto, nuestro proyecto no parecía tener ningún problema ya que se cumplía dicho requisito tal y como se ve en el código. Sin embargo, cuando veíamos ejemplos de código en la página de *Qt* en los que se hacía uso de *Window* siempre veíamos que un elemento *Window* era un elemento raíz, un elemento padre de otros. Lo que decidimos entonces, pues, fue que el elemento *Window* no nos era necesario, así que decidimos removerlo a él y a algunas propiedades que, según *Qt*, eran necesarias para determinar que la ventana *Window* fuera un diálogo y para que, además, fuera modal. Los elementos que removimos son los que se han marcado en amarillo en el código, y gracias a esto el diálogo ya pudo ser visible en dispositivos móviles.

12. CONCLUSIONES

Al comenzar este proyecto, eran varios los objetivos que se pretendían alcanzar, siendo algunos de ellos más generales y otros más específicos. Al encontrarnos en la fase final de nuestro proyecto, se puede echar la vista atrás hacia todo lo que se ha ido realizando a lo largo de este trabajo para analizar qué nos ha aportado, qué hemos conseguido aprender, qué cosas hemos completado satisfactoriamente y, como también debería hacerse, pensar en qué se podría mejorar este proyecto en el futuro y cómo se podría conseguir esto, teniendo en cuenta algunas limitaciones o impedimentos que hayan podido surgir durante su desarrollo.

El objetivo principal que se buscaba alcanzar con este proyecto era el de poder realizar un juego que se pudiera desplegar en **distintas** plataformas. Se empezó, desde un principio, a orientar el proyecto a plataformas móviles como *Android*, *Sailfish* o *iOS*, pero más adelante se determinó que el proyecto también podía ser viable para plataformas de escritorio como *Windows*, *Mac* o *Linux*. Pues bien, en lo que a sistemas operativos móviles se refiere, el proyecto ha podido ser ejecutado en dispositivos **reales** que contaban con un sistema operativo *Android* y otros que contaban con un sistema *Sailfish*. A pesar de que el proyecto **no pudo** ser probado en dispositivos que contaban con el sistema operativo *iOS*, se puede **garantizar**, una vez finalizado el proyecto, que el juego también se puede portar a este tipo de plataformas que ocupa una gran cuota en el mercado de dispositivos móviles. Esto se puede garantizar porque contamos con simuladores de dispositivos *iOS* que nos aseguraron la viabilidad del proyecto para este tipo de sistemas.

En lo que respecta a sistemas de escritorio, se puede concluir también que el proyecto ha podido ser desplegado en este tipo de sistemas. Tanto para *Windows*, como para *Mac* como para distribuciones del sistema *Linux* como *Ubuntu*, se ha podido generar un archivo ejecutable que ha permitido jugar con la aplicación desarrollada sobre estas plataformas, las cuales ocupan la mayor parte del mercado en lo que a sistemas operativos de escritorio se refiere.

Sin duda, el poder llevar el proyecto para sistemas y plataformas tan distintas es lo que más llama la atención de este proyecto. Hemos podido comprobar las grandes ventajas y también las dificultades que conlleva la **compilación cruzada**. Nos vimos en la necesidad de emplear variados entornos *Qt* de programación, dependiendo de la plataforma para la que quisiésemos desplegar el proyecto. Estos entornos eran los que permitían obtener el código ejecutable para las distintas plataformas. Pudimos entender, a lo largo del proyecto, la necesidad de la compilación cruzada en un proyecto como este, en el que se quería, desde un principio, desplegar un juego en múltiples plataformas, teniendo en cuenta que el juego sería desarrollado en otra plataforma distinta para la que se desplegaría. Pero al final, y lo que es **más importante**, hemos podido comprobar que **un mismo** proyecto, haciendo uso de las herramientas necesarias

y sin tener que afectar a la implementación de este, ha podido ser ejecutado en un número importante de plataformas, concretamente en seis, donde tres de ellas eran plataformas móviles y las otras tres plataformas de escritorio.

Cuando estábamos en alguna iteración de la fase de diseño del proyecto, siempre pensábamos que la interfaz debía resultar, como mínimo, cómoda y amigable para el usuario, que este pudiera entender rápidamente cómo era el funcionamiento del juego. A pesar de no ser uno de los objetivos principales que se buscaba en el desarrollo del proyecto, era un aspecto a tener en cuenta ya que, en caso contrario, el juego podría llegar a resultar ser un quebradero de cabeza para los jugadores. Teniendo en cuenta que se iba a programar un juego que era multijugador, y que, además, era un juego que se compartiría a través de un único dispositivo, pensamos que lo más cómodo para los jugadores sería poder ver el juego en horizontal, ya que así los diferentes componentes se verían mejor. Llegados al final del proyecto, podemos concluir que la interfaz es la que puede resultar más cómoda y atractiva para los jugadores. Pero además, cuando pensamos que íbamos a desarrollar un juego que resulta ser tradicional y que, además, es simple de jugar, decidimos que esto debíamos mantenerlo. El juego debía ser fácil de jugar para los usuarios, y es algo que hemos conseguido con la lógica implementada. Para evaluar estos aspectos de diseño y de la lógica implementada, pudimos observar a un número reducido de usuarios probando el juego, y pudimos así comprobar que entendían la dinámica de este rápidamente, y que la disposición en horizontal de los distintos componentes era la que más cómoda resultaba para ellos.

El desarrollo del proyecto ha resultado ser entretenido. Pero esta diversión ha sido acompañada de mucho aprendizaje. Hemos podido comprobar la potencia y flexibilidad de la que consta la tecnología *Qt*, la cual ha sido la protagonista a la hora de poder llevar el proyecto a múltiples plataformas. Tras finalizar el proyecto, estamos, sin duda, más familiarizados con esta tecnología y con los lenguajes con los que en ella se trabaja como *JavaScript* y *Qml*. *JavaScript* era, en el momento de comenzar el proyecto, un lenguaje de programación ya conocido por nosotros, pero gracias al desarrollo del proyecto hemos podido aprender más de este lenguaje y de las muchas cosas que se pueden conseguir con él, pudiendo ver que no se trata solo de un lenguaje destinado a la programación *web*. Sin embargo, con *Qml* no estábamos nada familiarizados, era algo totalmente nuevo y que nos ha llamado mucho la atención. Ciertamente, creemos que no hemos exprimido al máximo este lenguaje ya que son muchos los componentes que se pueden definir y, también, muchas las propiedades que estos tienen. Es un lenguaje con el que se pueden hacer interfaces gráficas muy poderosas. Por esta parte, podemos decir que, llegados a este punto del proyecto, conocemos cómo funciona la tecnología *Qml* y las grandes posibilidades que ella ofrece. Es una tecnología que hizo que la fase de implementación de la interfaz del juego resultará bastante cómoda y fácil de llevar.

En cuanto a **líneas futuras**, se pueden abrir varias para este proyecto. Por un lado, nos hubiera gustado haber ejecutado el proyecto en dispositivos *iOS* reales, algo que no ha podido conseguirse debido a la obligación que existe de pagarle un certificado de desarrolladores a *Apple* de forma anual. Pues bien, en un futuro podría contarse con

dicho certificado y así comprobar la ejecución del proyecto en dispositivos *iOS*. Además, a lo largo del proyecto también comprobamos que el uso de simuladores como los que empleamos para *Sailfish* o para *iOS* resultaba, a veces, algo lento y algo con lo que se perdía bastante tiempo ya que eran herramientas que, o bien necesitaban arrancar una máquina virtual aparte, o bien necesitaban arrancar otros entornos externos a los propios de la programación. Por otro lado, se podrían cambiar algunos detalles de la implementación del proyecto, sobretodo los que tienen que ver con la interfaz de juego. Esto, adquiriendo un poco más de experiencia en el lenguaje *Qml* sería perfectamente factible. Conociendo un poco más este lenguaje, se podría hacer una interfaz más rica y atractiva para los jugadores, sin tener que perder la comodidad y usabilidad con la que ya cuenta.

13. BIBLIOGRAFÍA

- APACHE, Ant. Welcome [en línea]. Internet: s.d. [Consulta: 11 Mayo 2014]. Disponible: http://ant.apache.org/index.html
- APP ANNIE, Blog. Digital Content Spend Accelerates, With Apps Leading Growth [en línea]. Internet: 19 Febrero 2014. Disponible: http://blog.appannie.com/app-annie-and-ihs-digital-content-report-2013/
- APPLE. OS X. Lo que hace que un Mac sea un Mac [en línea]. Internet: s.d. [Consulta: 23 Mayo 2014]. Disponible: https://www.*Apple*.com/es/osx/what-is/
- APPLE. Qué es iOS [en línea]. Internet: s.d. [Consulta: 25 Mayo 2014]. Disponible: http://www.*Apple*.com/es/*iOS*/what-is/
- BGR. Mobile games have become ridiculously lucrative over the past year [en línea]. Internet: Tero Kuittinen, 19 Febrero 2014. Disponible: http://bgr.com/2014/02/19/mobile-games-spending-growth/
- DE ANDRÉS, David y RUIZ, Juan Carlos. Fundamentos del desarrollo con Android [Material gráfico proyectable]. Valencia, 2014. 58 diapositivas.
- DEVELOPER, Apple. XCode, IDE [en línea]. Internet: s.d. [Consulta: 27 Mayo 2014]. Disponible: https://developer.*Apple*.com/xcode/ide/
- DEVELOPERS, Android. API Guides, What is API Level? [en línea]. Internet: s.d. [Consulta: 11 Mayo 2014]. Disponible: http://developer.*Android*.com/guide/topics/manifest/uses-sdk-element.html#ApiLevels
- DEVELOPERS, Android. Tools, Download the NDK [en línea]. Internet: s.d. [Consulta: 12 Mayo 2014]. Diponible: http://developer.*Android*.com/tools/sdk/ndk/index.html
- DEVELOPERS, Android. Tools, Exploring the SDK [en línea]. Internet: s.d. [Consulta: 11 Mayo 2014]. Disponible: http://developer.*Android*.com/sdk/exploring.html#Packages
- DIGIA. Qt in Use [en línea]. Internet: s.d. [Consulta: 15 Mayo 2014]. Disponible: http://*Qt*.digia.com/*Qt*-in-Use/
- DIGIA. The New Qt Designer [en línea]. Internet: s.d. [Consulta: 15 Mayo 2014]. Disponible: http://doc.*Qt*.digia.com/4.0/*Qt*4-designer.html
- DIGIA. What's New in Qt 4 [en línea]. Internet: s.d. [Consulta: 15 Mayo 2014]. Disponible: http://doc.*Qt*.digia.com/4.0/*Qt*4-intro.html
- FLANAGAN, David. JavaScript: The Definitive Guide. Loukides, Mike (editor). 6a ed. Gravenstein Highway North, Sebastopol: O'Reilly Media, Inc., 2011. 1098 p. ISBN: 978-0-596-80552-4
- LLVM. LLVM Overview, Clang [en línea]. Internet: s.d. [Consulta: 25 Mayo 2014]. Disponible: http://llvm.org/
- MINGW. Welcome to MinGW.org [en línea]. Internet: s.d. [Consulta: 23 Mayo 2014]. Disponible: http://www.mingw.org/

- OS DEV, wiki. GCC Cross-Compiler, Why do I need a Cross Compiler? [en línea]. Internet: 30 Noviembre 2006, [ref. de 17 Mayo 2014]. Disponible: http://wiki.osdev.org/GCC_Cross-Compiler#Why_do_I_need_a_Cross_Compiler.3F
- POWERS, Shelley. JavaScript Cookbook. St. Laurent, Simon (editor). 1a ed. Gravenstein Highway North, Sebastopol: O'Reilly Media, Inc., 2010. 554 p. ISBN: 978-0-596-80613-2
- QT PROJECT. [en línea]. Internet: s.d. [Consulta: 24 Mayo 2014]. Disponible: http://*Qt*-project.org/
- QT PROJECT. Building and Running [en línea]. Internet: s.d. [Consulta: 22 Mayo 2014]. Disponible: http://*Qt*-project.org/doc/*Qt*creator-2.6/creator-building-running.html
- QT PROJECT. Connecting Android Devices [en línea]. Internet: s.d. [Consulta: 12 Mayo 2014]. Disponible: http://*Qt*-project.org/doc/*Qt*creator-3.1/creator-developing-*Android*.html
- QT PROJECT. Connecting iOS Devices [en línea]. Internet: s.d. [Consulta: 26 Abril 2014]. Disponible: http://*Qt*-project.org/doc/*Qt*creator-3.1/creator-developing-*iOS*.html
- QT PROJECT. Cross-Compiling Qt for Embedded Linux Applications [en línea]. Internet: s.d. [Consulta: 22 Mayo 2014]. Disponible: http://*Qt*-project.org/doc/*Qt*-4.8/*Qt*-embedded-crosscompiling.html
- QT PROJECT. Getting Started with Qt for Android [en línea]. Internet: s.d. [Consulta: 12 Mayo 2014]. Disponible: http://*Qt*-project.org/doc/*Qt*-5/*Android*gs.html
- QT PROJECT. Glossary, kit definition [en línea]. Internet: s.d. [Consulta: 19 Mayo 2014]. Disponible: http://*Qt*-project.org/doc/*Qt*creator-2.6/creator-glossary.html#glossary-buildandrun-kit
- QT PROJECT. qmake Manual [en línea]. Internet: s.d [Consulta: 19 Mayo 2014]. Disponible: http://qt-project.org/doc/qt-4.8/qmake-manual.html
- QT PROJECT. qmake Projecty Files [en línea]. Internet: s.d. [Consulta: 19 Mayo 2014]. Disponible: http://*Qt*-project.org/doc/*Qt*-4.8/qmake-project-files.html
- QT PROJECT. Qt Creator [en línea]. Internet: 22 Enero 2014. Disponible: https://*Qt*-project.org/wiki/Category:Tools::*Qt*Creator
- QT PROJECT. Qt for Android [en línea]. Internet: s.d. [Consulta: 12 Mayo 2014]. Disponible: http://*Qt*-project.org/doc/*Qt*-5/*Android*-support.html
- QT PROJECT. Qt QML *QML* Types [en línea]. Internet: s.d. [Consulta: 13 Mayo 2014]. Disponible: http://*Qt*-project.org/doc/*Qt*-5/*QtQml*-*Qml*module.html
- QT PROJECT. Qt 5.2 is Here! [en línea]. Internet: s.d. [Consulta: 13 Mayo 2014]. Disponible: http://*Qt*-project.org/*Qt*5/*Qt*52
- SAILFISH. About, Architecture [en línea]. Internet: s.d. [Consulta: 20 Mayo 2014]. Disponible: https://*Sailfish*os.org/about-architecture.html
- SAILFISH. About, Technology [en línea]. Internet: s.d. [Consulta: 20 Mayo 2014]. Disponible: https://*Sailfish*os.org/about-technology.html

- WIKIPEDIA. a.out [en línea]. Internet: 29 Diciembre 2012, [ref. de 7 Marzo 2013]. Disponible: http://es.wikipedia.org/wiki/A.out
- WIKIPEDIA. ARM Architecture [en línea]. Internet: 5 Julio 2002, [ref. de 27 Mayo 2014]. Disponible: http://en.wikipedia.org/wiki/ARM_architecture
- WIKIPEDIA. Common Object File Format [en línea]. Internet: 29 Diciembre 2012, [ref. de 20 Agosto 2013]. Disponible: http://es.wikipedia.org/wiki/Common_Object_File_Format
- WIKIPEDIA. Executable and Linkable Format [en línea]. Internet: 12 Febrero 2006, [ref. de 11 Marzo 2013]. Disponible: http://es.wikipedia.org/wiki/Executable_and_Linkable_Format
- WIKIPEDIA. GNU Binutils [en línea]. Internet: 24 Mayo 2006, [ref. de 12 Octubre 2013]. Disponible: http://es.wikipedia.org/wiki/GNU_Binutils
- WIKIPEDIA. JavaScript [en línea]. Internet: 19 Noviembre 2001, [ref. de 20 Mayo 2014]. Disponible: http://en.wikipedia.org/wiki/JavaScript
- WIKIPEDIA. Jolla [en línea]. Internet: 9 Julio 2012, [ref. de 9 Mayo 2014]. Disponible: http://en.wikipedia.org/wiki/*Jolla*
- WIKIPEDIA. MeeGo [en línea]. Internet: 15 Febrero 2010, [ref. de 16 Mayo 2014]. Disponible: http://en.wikipedia.org/wiki/MeeGo
- WIKIPEDIA. Netwide Assembler [en línea]. Internet: 13 Julio 2006, [ref. de 4 Febrero 2014]. Disponible: http://es.wikipedia.org/wiki/Netwide_Assembler
- WIKIPEDIA. QML [en línea]. Internet: 22 Julio 2010, [ref. de 30 Marzo 2014]. Disponible: http://en.wikipedia.org/wiki/*QML*
- WIKIPEDIA. Qt (software) [en línea]. Internet: 24 Agosto 2001, [ref. de 26 Mayo 2014]. Disponible: http://en.wikipedia.org/wiki/*Qt*_(software)
- WIKIPEDIA. Sailfish OS [en línea]. Internet: 23 Julio 2012, [ref. de 20 Mayo 2014]. Disponible: http://en.wikipedia.org/wiki/*Sailfish*_OS
- WIKIPEDIA. Software Development Kit [en línea]. Internet: 6 Enero 2003, [ref. de 20 Mayo 2014]. Disponible: http://en.wikipedia.org/wiki/Software_development_kit
- WIKIPEDIA. Ubuntu [en línea]. Internet: 1 Noviembre 2004, [ref. de 24 Mayo 2014]. Disponible: http://es.wikipedia.org/wiki/*Ubuntu*
- WIKIPEDIA. Usage share of operating systems [en línea]. Internet: 23 Mayo 2007, [ref. de 23 Mayo 2014]. Disponible: http://en.wikipedia.org/wiki/Usage_share_of_operating_systems
- WIKIPEDIA. VMware Workstation [en línea]. Internet: 29 Noviembre 2006 [ref. de 19 Mayo 2014]. Disponible: http://en.wikipedia.org/wiki/VMware_Workstation
- W3SCHOOLS. JavaScript Tutorial [en línea]. Internet: s.d. [Consulta: 18 Mayo 2014]. Disponible: http://www.w3schools.com/js/default.asp
- W3TECHS. Usage of JavaScript for websites [en línea]. Internet: s.d. [Consulta: 19 Mayo 2014]. Disponible: http://w3techs.com/technologies/details/cp-javascript/all/all

- XYO, Blog. The Most Expensive Android Apps and Games [en línea]. Internet: Liam, 7 Octubre 2013. Disponible: http://xyo.net/blog/most-expensive-apps-for-*Android*