SCORE FASTER: Aplicación multijugador en línea para fomentar la destreza y competitividad de los jugadores

Christian Barquilla-Gómez

Resumen— Este proyecto presenta como principal objetivo desarrollar una aplicación multijugador en línea que fomente la habilidad de los jugadores desde un modo competitivo. En ella, los jugadores de una misma partida compiten simultáneamente por obtener la mayor puntuación encestando una pelota en una canasta en un período de tiempo finito. *Score Faster* surge por motivación personal y con la finalidad de adentrarse en el desarrollo de un videojuego en *Android* con una arquitectura *cliente – servidor*, un gran desafío para todo programador. No obstante, gracias a la librería *libGDX* se consigue dar un soporte multiplataforma y mediante su módulo *Box2d*, se gestionan todas las físicas que actúan dentro del videojuego. *Score Faster* afronta otro gran reto con éxito como es el de ofrecer compatibilidad en múltiples dispositivos con resoluciones diferentes entre ellos aplicando la técnica de generación de texturas en función de la densidad de pantalla del dispositivo. A todas estas características se le añade un sistema de personalización que permite seleccionar entre trece apariencias diferentes de pelota, utilizar un alias en cada partida o elegir el peso de la pelota, convirtiendo así a *Score Faster* en un videojuego más completo.

Paraules clau— Videojuego, multijugador, soporte multiplataforma, personalización del jugador, baloncesto, libGDX, KryoNet, arquitectura cliente-servidor, Box2D, OpenGL.

Abstract—The main objective of this project is to develop an online multiplayer application which encourages player's skill in a competitive way. In it, the players of the same match competes simultaneously for the highest score by throwing a ball inside of a basketball basket in a finite period of time. *Score Faster* arises from personal motivation and with the purpose of introduce me into *Android* game development with a *client – server* architecture, a great challenge for any programmer. However, the *libGDX library* provides cross-platform support and his *Box2d* module manages all of physics that acts inside the video game. *Score Faster* faces another great challenge successfully such as to offer compatibility in multiple devices with different resolutions beetween them applying the strategy of textures generation according to the screen density of the device. A mechanism of customization is added with the others game features that allows to select beetween thirteen ball appearances, to use an alias in each *Match* or choose the weight of the ball, converting *Score Faster* in a more complete game.

Index Terms— Video game, multiplayer, cross-platform support, player customization, basketball, libGDX, KryoNet, client-server architecture, Box2D, OpenGl.

1 Introducción

A día de hoy, se hace difícil imaginar una vida sin videojuegos, los cuales son una de las opciones más buscadas y utilizadas en el sector de entretenimiento. Sus orígenes se remontan a la década de 1950, cuando por aquel entonces se implementaron los primeros prototipos de juegos electrónicos. Desde el punto de vista histórico, la evolución de los videojuegos demuestra una clara evidencia en el apartado audiovisual y en la fuerte apuesta por la inmersión de los jugadores en la historia del videojuego.

Por un lado, uno de los principales motivos por los cuales el sector de los videojuegos ha generado tal impacto social ha sido la cada vez mayor facilidad para acceder a éstos. Si nos situamos en el inicio de su historia, visualizamos grandes y costosas máquinas recreativas a las cuales únicamente podíamos acceder visitando nuestro centro de ocio más cercano. El descenso de los costes de fabricación de los videojuegos y las máquinas *hardware* capaces de ejecutarlos ha permitido incluso incorporarlos en nuestro dispositivo móvil personal capaz de ejecutar una variedad de éstos con un solo *click*.

Por otro lado, no se puede descuidar otro de los puntos clave causantes del auge que está presentando los videojuegos desde mediados de los noventa: el apartado multijugador en línea. Observando los grandes y exitosos títulos actuales, encontrar un videojuego que no haya apostado por esta modalidad de juego es una tarea prácticamente imposible.

Este análisis ha propiciado y motivado la propuesta de este proyecto, una aplicación con gráficos 2D (pero con una perspectiva que simula el efecto 3D) capaz de ser ejecutada

- ***Correo electrónico de contacto:*** *christian.barquilla@e-campus.uab.cat*
- ***Mención realizada:*** *Ingeniería del Software*
- ***Trabajo tutorizado por:*** *Yolanda Benítez Fernández (CVC)*
- ***Curs 2016/17***

en múltiples plataformas, incidiendo en la plataforma *Android* y que incorpora un sistema multijugador en línea que busca potenciar la destreza de los jugadores desde un modo competitivo.

Por último, se presentará la estructura de este artículo el cual está compuesto por seis secciones. En este primero se expondrán los objetivos y el estado del arte de este proyecto. A continuación, se pueden consultar los apartados siguientes: metodología, resultados, conclusiones, agradecimientos y bibliografía.

1.1 Objetivos del proyecto

El desarrollo de este proyecto presenta como principal objetivo implementar una aplicación con soporte en los Sistemas Operativos *Android* y *Windows* que deriva a un videojuego que incorpora una arquitectura *cliente – servidor* encargada de gestionar el apartado multijugador en línea de éste.

Este videojuego es capaz de interconectar dos dispositivos diferentes (bajo cualquiera de los dos Sistemas Operativos citados anteriormente) por cada partida de juego. Los jugadores de una misma partida competirán simultáneamente por encestar una pelota el mayor número de veces posibles dentro de un período de tiempo finito dentro de una canasta.

Para ello, *Score Faster* proporciona un escenario de juego con una ambientación de baloncesto para que los jugadores desarrollen sus partidas dentro de él. Para añadir un grado mayor de dificultad, este escenario de juego incluye obstáculos que tratarán de poner a prueba las habilidades de los jugadores por alzarse con la victoria.

Otros factores del juego que serán determinantes durante el transcurso de la partida son los siguientes:

- **Tipo de canasta:** El jugador que realice un lanzamiento limpio (tocando únicamente la canasta) recibirá 3 puntos. En caso contrario, recibirá 2 puntos.
- **Sistema de personalización:** Cada jugador podrá seleccionar entre diversas apariencias para su pelota, escribir un nombre de jugador y modificar el peso de ésta. Los jugadores más exigentes podrán optar en utilizar una pelota más pesada o más ligera que influirá notablemente en la fuerza que deben ejercer cuando realicen el lanzamiento sobre ella.
- **Canasta de oro:** Si al finalizar el cronómetro de la partida existe un caso de empate (misma puntuación entre jugadores), se procederá a entrar en el modo "canasta de oro". En este modo, el primer jugador que acierte con su lanzamiento dentro de la canasta se proclamará ganador de la partida.

1.2 Estado del arte

Actualmente, las grandes empresas del sector de los videojuegos combinan múltiples estrategias de juego para provocar admiración y convencer al jugador con sus productos. Desde una historia conmovedora hasta una jugabilidad capaz de sumergir al jugador en ella gracias a la tecnología de realidad virtual.

No obstante, *Score Faster* se centra en el mercado de los dispositivos móviles, incidiendo en la plataforma *Android,* aunque sin dejar de lado la compatibilidad con el sistema operativo *Windows*.

Aclarado el punto anterior, antes de desarrollar este proyecto se realizó un análisis exhaustivo de las aplicaciones publicadas en la plataforma oficial de aplicaciones de *Android* (*Google Play*) en busca de contenido similar a este proyecto. Debido al bajo coste de la licencia de publicación de aplicaciones en *Android*, una gran cantidad de desarrolladores apuestan por esta plataforma y, por ende, se hallaron una multitud de aplicaciones que podrían convertirse en una clara competencia para *Score Faster*.

Sin embargo, aplicaciones como *Tiger Ball – 2D, Tiger Ball 2 – Physics Puzzles* y sobretodo la aplicación *TigerBall* se postulaban como la competencia más directa por su similar mecánica de juego, su ambientación (de baloncesto también) y la gran cantidad de descargas que presenta a día de hoy la última de estas tres aplicaciones.

Más tarde, se analizaron los puntos fuertes y débiles de estas aplicaciones. En líneas generales, estas aplicaciones destacan por su sencilla y adictiva mecánica de juego y por su apartado visual. Por otro lado, atendiendo a los comentarios de los usuarios de éstas, muchos de los jugadores estaban de acuerdo en que la monotonía acababa manifestándose con el paso del tiempo.

Es por ello, que con el desarrollo de *Score Faster* se ha procurado aprovechar los puntos fuertes de estas aplicaciones y al mismo tiempo intentar minimizar o incluso evitar la aparición de la monotonía en los futuros jugadores. Para ello, se ha incorporado una arquitectura multijugador de la cual carecen las aplicaciones anteriores para así fomentar la competitividad entre jugadores y hacer de cada partida una experiencia grata para ellos.

Por último y no menos importante, cabe mencionar que *Score Faster* ha apostado por ofrecer a los jugadores instantes previos al comienzo de una partida un sistema de personalización que les permite escoger entre una diversidad de texturas para su pelota, así como introducir un nombre de jugador o modificar el peso de ésta como elementos diferenciadores del resto de la competencia.

2 METODOLOGÍA

En el desarrollo de este proyecto se ha utilizado una metodología ágil basada en *SCRUM* [1] aunque con ligeras variaciones puesto que el equipo de desarrollo está formado por una única persona.

El proyecto se ha dividido a lo largo de su ciclo de desarrollo en *Sprints* o entregables de una semana de duración en los que se han generado versiones estables del sistema. En cada *Sprint* se han realizado tareas tanto de documentación, diseño, desarrollo y *testing* para no descuidar ninguno de estos cuatro pilares claves en un proyecto software.

Antes de definir las tareas que se han realizado en cada uno de los *Sprints* del proyecto, se ha generado previamente una lista con todos los requisitos que debía satisfacer *Score Faster*. De esta lista se seleccionaban al principio de cada *Sprint* aquellos que resultaban más prioritarios y en caso de no finalizar alguno de éstos, se aplazaban e implementaban en los siguientes *Sprints*.

Cabe destacar que al principio de cada uno de los días que conforman un *Sprint* se ha realizado un breve análisis de una duración entre 5 – 10 minutos antes de comenzar la sesión de trabajo para realizar un seguimiento del progreso alcanzando hasta el momento.

Finalmente, al final de cada *Sprint* se ha comprobado y evaluado el resultado obtenido durante esa semana de desarrollo para planificar correctamente el próximo *Sprint*.

Por último, se detallará gráficamente esta metodología de desarrollo para ilustrar su ciclo iterativo:

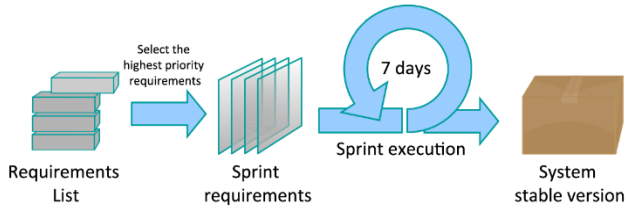


Figura 1: Metodología ágil de desarrollo de Score Faster

3 TECNOLOGÍAS DE DESARROLLO

Con el contenido de este apartado se pretende detallar las dos principales tecnologías de desarrollo que se han utilizado para implementar *Score Faster*, las cuales serán analizadas individualmente en los apartados de esta sección.

En primer lugar, es necesario poner en contexto la estructura de este proyecto que, a su vez, se ha divido en dos proyectos implementados mediante el lenguaje de programación *Java*. Es necesario recordar que *Score Faster* incorpora una arquitectura *cliente – servidor*.

Por un lado, la implementación del lado del *Cliente* (código ejecutado en el dispositivo con el que se pretende jugar) ha dado lugar a una aplicación desarrollada a partir de un proyecto de *Android Studio IDE* y que utiliza la librería *libGDX* [2].

Por otro lado, la librería *KryoNet* [3] ha facilitado el desarrollo del lado del *Servidor* el cual ha sido implementado utilizando un proyecto *Java* de *Eclipse IDE*.

Por último, cabe resaltar que tanto el proyecto *Cliente* como el proyecto *Servidor* necesitan conocer los paquetes de red que se envían entre ellos, así como la clase que representa al *Jugador y* una clase que se encarga de definir los parámetros de red comunes entre los dos proyectos. Como la lógica del multijugador está implementada únicamente en el proyecto *Servidor,* se ha exportado en formato *jar* a modo de librería (bajo el nombre de *ScoreFasterNetworking.java*) los ficheros de código fuente que necesita conocer el proyecto *Cliente* para importarlos en éste. Si se desea obtener una visión más ilustrativa de la estructura de estos dos proyectos, se puede consultar el *Apéndice A1* y *A2*.

3.1 libGDX

libGDX es una librería *open-source* que permite desarrollar videojuegos en el lenguaje *Java* y ejecutarlos en *Windows, Mac, Linux, Android, iOS, BlackBerry* y *HTML5* sin necesidad de modificar el código [4]. *libGDX* utiliza *OpenGL* [5], una especificación estándar multiplataforma que define una API la cual permite producir y manipular gráficos en 2D y 3D a partir de primitivas geométricas simples como un cuadrado o un triángulo. Uno de los principales motivos por los cuales *Score Faster* se ha desarrollado bajo esta librería es su excelente y detallada documentación, facilitando el aprendizaje a los más novatos. Otros motivos a destacar son la incorporación de otras librerías que añaden más funcionalidades y de herramientas que solucionan algunos de los problemas más comunes que un desarrollador debe afrontar en *libGDX*.

Para crear un proyecto con esta librería, en primer lugar, debemos descargar la *libGDX Project setup tool* [6]. Una vez ejecutada esta herramienta observaremos la siguiente ventana:

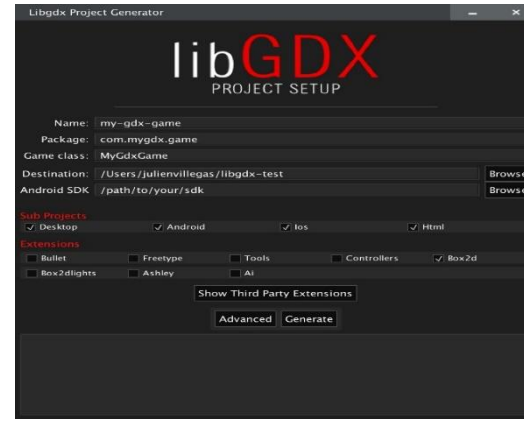


Figura 2: libGDX project setup tool

Esta herramienta nos permite escoger en primera instancia el nombre de la aplicación, del *package* del proyecto, de la clase principal del juego y la destinación del proyecto que se generará posteriormente. A continuación, en el apartado *Sub Projects* hallamos cuatro posibles plataformas de destino del juego que marcaremos o no en función de si deseamos dar compatibilidad o no. Por último, como se ha comentado anteriormente, *libGDX* incorpora dentro de ella más librerías o módulos que podemos seleccionar dentro del apartado *Extensions*. Los módulos extra que esta librería nos permite añadir son los siguientes:

- **Bullet:** Implementa un sistema de físicas en 3D.
- **Freetype:** Permite cargar archivos de fuentes de texto para poder visualizarlas en la aplicación.
- **Tools:** Conjunto de herramientas para facilitar algunas tareas. Las herramientas utilizadas se explicarán con detalle en el apartado *4.3 Implementación* de la sección *CICLO DE DESARROLLO*.
- **Controllers:** Librería para gestionar controladores de entrada como por ejemplo un mando de la *Xbox 360*.
- **Box2d:** Librería de físicas en 2D. Esta librería se analizará más exhaustivamente en el apartado *4.3 Implementación* de siguiente sección.
- **Box2dlights:** Librería para añadir iluminación en Box2d.
- **Ashley:** Pequeño *framework* para desarrollar videojuegos.
- **Ai:** Módulo de inteligencia artificial.

En *ScoreFaster* se han utilizado los módulos *Freetype, Tools* y *Box2d*.

3.2 KryoNet

KryoNet es una librería desarrollada en *Java* que proporciona una *API* para gestionar eficientemente comunicaciones *ciente-servidor* mediante los protocolos *TCP* y *UDP*. Las diferencias entre estos protocolos son las siguientes:

- El protocolo *TCP* garantiza que los datos enviados por la red serán entregados a su destino sin errores y en el mismo orden en el que se transmitieron.
- En cambio, en el protocolo *UDP* no hay confirmación de llegada de los datos enviados ni control de flujo, por lo que los mensajes de red pueden adelantarse unos a otros o incluso no llegar a su destino correctamente. Sin embargo, la transmisión de los paquetes es más rápida y eficiente que en el protocolo *TCP*.

Esta librería es especialmente adecuada para juegos por su eficiencia y como característica principal permite enviar objetos complejos desde un punto a otro. Los paquetes de red representan objetos de clases que pueden contener múltiples atributos y métodos. Sin embargo, si se desea que tanto el servidor como el cliente conozcan y procesen estos mensajes, es necesario registrar estos paquetes con el método *register* [7].

Otra de las grandes ventajas que ofrece *KryoNet* es que es necesario pocas líneas de código para configurar y lanzar un servidor [8] y los clientes que se conectarán a él [9]. No obstante, la implementación del procesamiento de los mensajes recibidos no es tan trivial.

Para incluir la librería al proyecto *Servidor* de *Eclipse IDE*, tan solo es necesario importar la librería en formato *jar* desde la opción "*Add External JARs*" del *Java Build Path*.

4 CICLO DE DESARROLLO

Este apartado tratará de explicar detalladamente la fase técnica presente a lo largo del ciclo de desarrollo de *Score Faster*. Esta sección se dividirá en cuatro apartados que representan las etapas principales de todo proyecto software: Obtención de requisitos y documentación, diseño, implementación y *testing*.

4.1 Obtención de requisitos y documentación

Antes de de dar paso a las etapas de diseño e implementación, es necesario realizar un análisis previo y exhaustivo para determinar los requisitos que debe satisfacer el proyecto. Así pues, las técnicas de elicitación de requisitos empleadas en este proyecto han sido dos: análisis de aplicaciones existentes (apartado *1.2 Estado del arte*) y la técnica *Brainstorming* (lluvia de ideas).

Con esta etapa se consigue reflejar con claridad y precisión las características del sistema (requisitos) clasificadas en funcionales (qué debe incluir el sistema) y no funcionales (restricciones del sistema, requisitos de rendimiento, etc..). El documento especializado en reunir y estructurar esta información es el documento *SRS* (*Software Requirements Specification*) cuyo contenido, además, incluye los objetivos del proyecto junto con un diagrama *AND-OR* (diagrama de objetivos), la definición de los casos de uso y sus diagramas y, por último, la trazabilidad o relación que existe entre todos ellos.

Durante el desarrollo de este proyecto se han generado otros documentos de interés como el documento que reúne los errores y dificultades surgidos, un diagrama de *Gantt* para planificar las tareas a realizar en función del tiempo, documento de *testing*, documento de diagramas *UML* y un documento de análisis que incluye los resultados de la encuesta realizada, el diseño de las pantallas de la aplicación y el análisis de la gestión de configuración y de versiones.

4.2 Diseño

La fase de diseño es una de las más importantes considerando la temática del proyecto: un videojuego. La primera impresión que todo jugador presenta sobre un videojuego queda muy influenciada por la calidad del diseño.

Es por este motivo que se han combinado las herramientas *Adobe Photoshop* [10] y *Adobe Illustrator* [11] para trabajar en el diseño de *Score Faster* y procurar alcanzar un resultado profesional y de calidad. En el caso del diseño de los elementos del escenario de juego, se ha utilizado una perspectiva que simula un efecto 3D sobre ellos, añadiéndoles una

sensación de profundidad necesaria para visualizar la acción de encestar.

Cabe destacar que la fase de diseño de *Score Faster* se divide en tres partes: diseño de las pantallas del juego, diseño de los *sprites* y otros diseños como por ejemplo el logo de la aplicación.

4.2.1 Pantallas del juego

Antes de comenzar a implementar las pantallas del juego es de vital importancia generar prototipos de diseño de éstas para facilitar la labor de su implementación. Los prototipos generados han sido de alta fidelidad para conseguir una representación lo más cercana al diseño real de la aplicación y así aprovechar al máximo el trabajo realizado en éstos.

Las pantallas del juego se componen de diversos elementos que han sido diseñados o seleccionados previamente a la generación de los prototipos. Entre estos elementos podemos encontrar la elección de las fuentes de texto utilizadas, los botones, los diversos fondos utilizados dentro de la aplicación, entre otros. A continuación, se proporcionará una ilustración de uno de los prototipos de diseño para la representación de la pantalla de elección de escenario:

Figura 3: Prototipo de diseño de la pantalla de elección de escenario

4.2.2 Sprites

En el apartado *1.1 Objetivos del proyecto* se ha informado de que *Score Faster* incorpora un sistema de personalización el cual permite al jugador elegir entre una variedad de trece apariencias diferentes para su *sprite* (en este caso, una pelota). Estas apariencias se han diseñado mediante la aplicación de determinadas técnicas utilizando *Adobe Illustrator*. En las siguientes ilustraciones se puede observar cuatro de las trece apariencias disponibles dentro de *Score Faster*:

Figura 4: Pelota naranja

Figura 5: Pelota púrpura

Figura 6: Pelota azul

Figura 7: Pelota verde

4.2.3 Otros diseños

En una aplicación *Android* es esencial diseñar un logo que la identifique en su totalidad. El logo diseñado para *Score Faster* es el siguiente:

Figura 8: Logo de Score Faster

Dentro de este apartado se podría destacar también el muro presente en el escenario de juego:

Figura 9: Diseño del muro del escenario de juego

4.3 Implementación

Este apartado tratará de explicar con el máximo nivel de detalle, claridad y exactitud, toda la fase de implementación de este proyecto. Es por ello que se estructurará en más apartados para definir los diversos procesos de implementación aplicados. A continuación, se procederá a exponer todos ellos.

4.3.1 Soporte a múltiples resoluciones y dispositivos

Con *Score Faster* se ha apostado por desarrollar una aplicación compatible en más del 95% de dispositivos *Android* actuales según la distribución de usuarios representada en el *Apéndice A3* del final de este documento. Por este motivo, *Score Faster* ofrece compatibilidad en el intervalo de versiones *Android* que comienza en la versión *4.1.x Jelly Bean* (API 16) y finaliza en la versión (incluida) *7.1 Nougat* (API 25).

Sin embargo, desarrollar una aplicación compatible en los dispositivos más antiguos conlleva afrontar más de un rompecabezas. Estos dispositivos carecen de la potencia gráfica presentes en los dispositivos actuales y, por lo tanto, no pueden procesar texturas con unas determinadas resoluciones. La técnica que se ha empleado para solucionar este gran inconveniente será detallada en el siguiente apartado.

Otro gran reto que se afrontó en el principio de la fase de desarrollo de este proyecto fue el de gestionar que diferentes dispositivos con diferentes resoluciones entre ellos visualizaran correctamente la aplicación. Para solucionar esto, *libGDX* ofrece una estrategia basada en el uso de *Viewports* [12] que permiten asignar una resolución virtual sobre la cual se trabajará internamente en la aplicación para posicionar todos los elementos dentro de la pantalla del dispositivo. Los *Viewports* además incluyen una cámara que permite gestionar el punto de vista del escenario. En el caso de *Score Faster*, se ha utilizado un *Viewport* de tipo *StrechViewport* [13] ya que este tipo de *Viewport* siempre ajusta a pantalla completa el contenido de la aplicación cuando el dispositivo que la ejecuta utiliza una resolución diferente a la definida internamente en el proyecto.

4.3.2 Carga y uso de texturas en la aplicación

En el apartado anterior se ha planteado el problema que presentan los dispositivos más antiguos al trabajar con texturas con resoluciones superiores a las soportadas por sus unidades de procesamiento. La estrategia a seguir para solucionar este problema consiste en generar la textura original en una resolución no superior a la resolución virtual de la aplicación y cargarla dentro de ésta en resoluciones diferentes para utilizar aquella que sea compatible con el dispositivo. Para ello, es necesario obtener la densidad de la pantalla del dispositivo (puntos por pulgada) y calcular el factor de escala correspondiente a la densidad. De esta operación se encarga *libGDX*.

Por lo tanto, para cada textura se generará seis versiones [14] con diferente resolución aplicando a la textura original el factor de escala correspondiente tal y como muestra la tabla proporcionada en el *Apéndice A4*.

Más tarde, después de realizar el paso anterior, entra en escena otro inconveniente derivado de una particularidad de *OpenGL*: las dimensiones de las texturas deben ser siempre una potencia de dos, tanto el ancho como la altura. De no ser así, cuando ejecutaramos la aplicación se verían texturas de color negro. En este punto, nace la pregunta clave, ¿Qué debo hacer para trabajar con texturas con dimensiones que no están en función de una potencia de 2? Para la tranquilidad de los desarrolladores de *libGDX*, existe una herramienta llamada *Texture Packer* [15] dentro del módulo *Tools* (definido en el apartado *3.1 libGDX*) que permite agrupar múltiples texturas con una resolución independiente entre ellas en un único archivo cuyas dimensiones sí que deben cumplir esta particularidad de *OpenGL*. Para utilizar dentro de la aplicación una textura presente en un archivo de estas condiciones, simplemente se debe indicar el nombre de ésta utilizando el método *findRegion*.

Para finalizar este apartado, se proporcionará una ilustración que muestra las opciones de la herramienta *Texture Packer*:

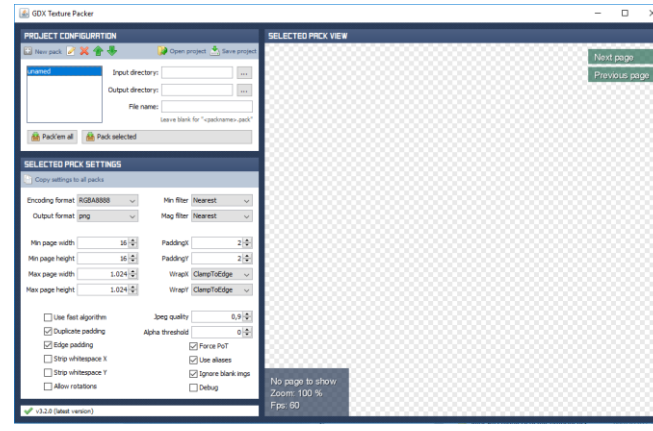


Figura 10: Texture Packer

4.3.3 Módulos del proyecto Cliente

El proyecto *Cliente* desarrollado en *Android Studio IDE* es el encargado de implementar y reunir toda la lógica de la aplicación excepto la del servidor. El código fuente de este proyecto se divide en cinco módulos más una clase principal tal y como muestra la siguiente ilustración:

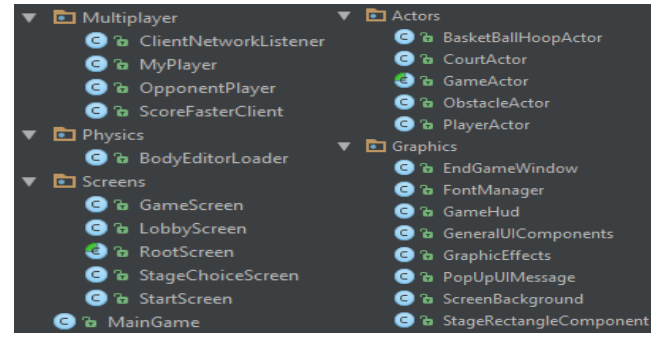


Figura 11: Módulos del proyecto Cliente

A continuación, se procederá a analizar cada uno de estos módulos:

- ***MainGame:*** Es la clase principal de la aplicación (hereda de la clase *Game*). Contiene el objeto que representa la pantalla de juego que se está ejecutando en cada momento y permite la transición entre pantallas. En esta clase se cargan y procesan todos los recursos del juego (texturas, música y

fuentes de texto). Además, cuenta con un método que es capaz de detectar la densidad de la pantalla del dispositivo y cargar la textura compatible con éste.

- ***Screens*:** Este módulo reúne todas las pantallas del juego. La aplicación está compuesta por cinco: *RootScreen* (de la que heredan el resto de pantallas), pantalla principal (*StartScreen*), pantalla de elección de escenario (*StageChoiceScreen*), pantalla de la sala de espera (*LobbyScreen*) y pantalla del desarrollo de la partida (*GameScreen*). Este módulo se analizará con más precisión en el siguiente apartado.
- ***Actors*:** Representa todos los actores con una física asociada que intervienen en una partida. Estos actores son: *GameActor* (clase padre), *BasketBallHoopActor* (canasta), *CourtActor* (campo de juego), *PlayerActor* (jugador) y *ObstacleActor* (obstáculo del escenario).
- ***Graphics*:** Este módulo contiene todas las clases relacionadas con los gráficos de la aplicación y con los elementos complejos de la interfaz de usuario.
- ***Physics*:** Únicamente contiene la clase *BodyEditorLoader* para gestionar la física de la canasta y el obstáculo del juego (por contener formas personalizadas). La física del jugador se gestiona con la primitiva simple del círculo.
- ***Multiplayer***. En este módulo se implementa la lógica del multijugador del *Cliente*. Por un lado, la clase *ScoreFasterClient* define el *Cliente* del juego y lo comunica con el *Servidor*. Por otro lado, *ClientNetworkListener* es la clase encargada de definir las operaciones a realizar cuando se reciben los paquetes enviados por el *Servidor*. En tercer lugar, la clase *MyPlayer* contiene y define todos los parámetros y métodos del *sprite* que controla cada jugador. Contiene a su vez un objeto de clase *PlayerActor* para visualizar la apariencia del *sprite* y gestionar la física de éste. Por último, la clase *OpponentPlayer* implementa toda la lógica del jugador oponente.

Finalmente, cabe mencionar que en el *Apéndice A2* se puede observar el diagrama de clases del proyecto *Cliente*.

4.3.4 Pantallas de juego: Clase Screen

En *libGDX*, una pantalla de juego es representada con un objeto de la clase *Screen* [16]. Todas las clases localizadas en el módulo *Screens* descrito en el apartado anterior heredan de esta clase. La clase *Screen* proporciona los siguientes métodos para implementar toda la lógica de una pantalla de juego:

- ***show()*:** Método que se invoca cuando la clase *MainGame* establece esta pantalla como pantalla actual con el método *setScreen*(*Screen screen*).
- ***hide()*:** Entra en ejecución cuando se procede a ejecutar otra pantalla o la aplicación se cierra.
- ***dispose():*** Método encargado de liberar los recursos de la pantalla. Se ejecuta al cerrar la aplicación.
- ***render()*:** Invocado cuando la pantalla se renderiza. Contiene el bucle principal de la pantalla.
- ***resize()*:** Entra en acción cuando la pantalla del dispositivo se redimensiona.
- ***pause():*** Entra en ejecución cuando se pierde el foco de la aplicación o se cierra.
- ***resume():*** Se invoca cuando la pantalla recupera el foco.

Para facilitar la comprensión del ciclo de vida de una *Screen*, se proporciona el siguiente diagrama de estado:

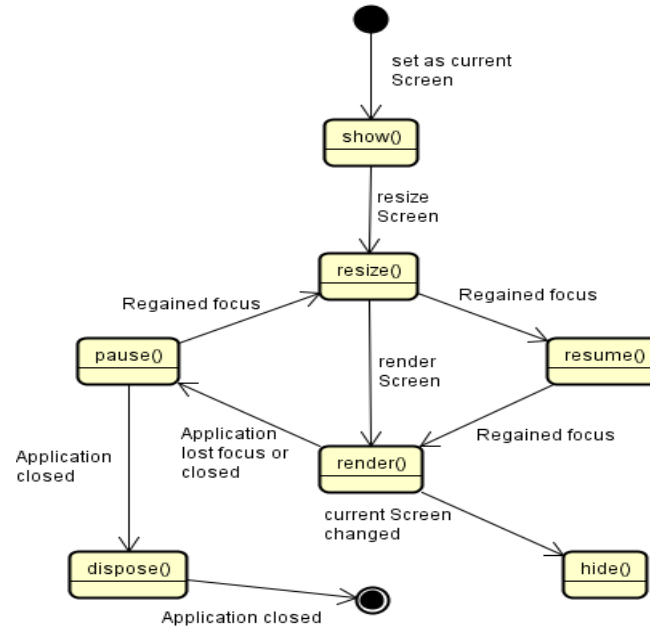


Figura 12: Diagrama de estado de una Screen

Por último, cabe resaltar que se ha diseñado un diagrama de flujo para describir el funcionamiento del proyecto *Cliente* y las transiciones entre sus pantallas de juego. Dicho diagrama se encuentra disponible en el *Apéndice A5* del final de este documento.

4.3.5 Concepto Escena: Scene2d y Scene2d.ui

libGDX proporciona por defecto una librería llamada *Scene2d* [17] que permite construir grafos de escena facilitando así, la implementación de los componentes de la interfaz de usuario. El concepto escena propuesto por esta librería presenta tres grandes pilares estrechamente relacionados y representados en forma de clases *Java*:

- ***Actor*:** Un actor es una clase que representa un nodo en el grafo de escena que contiene una posición, unas dimensiones, un origen, un escalado, rotación y color. Ejemplos de actores son una imagen (que contiene una textura), un botón, un campo de texto o cualquier elemento gráfico presente en las pantallas de la aplicación.
- ***Group*:** La clase *Group* simplemente es un conjunto de *Actores*. El uso de esta clase está especialmente recomendado en los casos en los que es necesario combinar múltiples actores para generar componentes de interfaz de usuario más complejos.

- ***Stage*:** La funcionalidad de un *Stage* es la de gestionar los diversos actores que pueden estar presentes en un escenario. Se encarga de añadir los actores o grupos de actores al escenario para visualizarlos o ejecutar determinadas acciones sobre ellos.

Para complementar el módulo anterior, *libGDX* cuenta con un *package* llamado *Scene2d.ui* [18] cuya definición se basa en una extensión de *Scene2d* que incluye actores ya definidos y otras utilidades con la finalidad de construir interfaces de usuario. Durante el proceso de desarrollo de *Score Faster* se han utilizado las siguientes clases de este *package:*

- ***Label***: Diseñada para gestionar y visualizar texto.
- ***TextButton, ImageButton* y *ImageTextButton*:** Estas tres clases definen tres tipos de botones. La primera de ellas acepta botones con texto, la segunda botones con una imagen dentro de ellos y la tercera es una combinación de las dos anteriores.
- ***TextField*:** Como su nombre especifica, representa un campo de texto que acepta las entradas por teclado. Este control se ha empleado para permitir a los jugadores modificar su alias en cada partida.
- ***Image*:** Clase utilizada para gestionar cualquier textura individual.

Este *package* permite personalizar cómodamente el diseño de todos estos elementos asignándoles un estilo con el cual se puede definir la textura de fondo, tipo de fuente de texto y su color, como opciones más utilizadas en este proyecto.

4.3.6 Box2d: Definición de un World y Bodies

Box2d [19] es una de las más famosas librerias de físicas en 2D para juegos y que ha sido traducida a múltiples lenguajes y motores de videojuegos. A diferencia de *Scene2d* y *Scene2d.ui*, no viene incluida por defecto en un proyecto *libGDX*, no obstante, tan solo es necesario marcar su *checkbox* en el apartado *extensions* de la *libGDX Project setup tool*, tal y como se describió en el apartado 3.1 libGDX.

Una particularidad muy significativa en el desarrollo con la librería *Box2D* es que las físicas generadas con ésta trabajan en metros y no en píxeles como en *Scene2D*. La estrategia utilizada en la unificación de los gráficos definidos en *Scene2D* con sus físicas generadas en *box2D* ha sido la de transformar las posiciones en metros a posiciones en píxeles antes de proceder a pintar los gráficos del juego.

Para inicializar correctamente *Box2D* en el proyecto, se debe empezar por la definición de un objeto *World* [20]. Este objeto ejerce una gravedad horizontal y vertical sobre el escenario de juego que podemos ajustar a nuestro gusto y, además, es el encargado de crear los cuerpos de las físicas de los objetos. En el caso de *ScoreFaster*, se ha utilizado una gravedad 0 en el eje horizontal y una gravedad de -10 al eje vertical simulando la gravedad de la tierra (se ha asignado el valor de 10 en vez del de 9,8 porque el uso de decimales en la gravedad de *Box2D* realentiza el tiempo de ejecución).

En segundo lugar, es necesario definir los cuerpos de las físicas asociadas a algunos de los elementos presentes en el escenario. Estos elementos son la canasta, el campo de juego, el obstáculo, los limites de la pantalla y los *sprites* de los jugadores. La definición y creación de estos cuerpos se resumen en los siguientes tres pasos:

- **Creación de un *BodyDef*:** El objeto *BodyDef* contiene los parámetros de configuración de un *Body*, como, por ejemplo, la posición y el tipo de cuerpo de éste. Existen tres tipos de cuerpos: estáticos (objetos sin movimiento), dinámicos (objetos a los cuales se aplica una fuerza como a los *sprites* de los jugadores), y cinemáticos (una combinación de los dos anteriores).
- **Creación del objeto *Body*:** Como he dicho al principio de este apartado, la creación y asignación de un objeto *Body* queda bajo la autoridad del objeto *World*, el cual invoca al método *createBody* enviándole como parámetro los parámetros del cuerpo (*BodyDef*).
- **Asignación de la *Fixture*:** Por último y no menos importante en este proceso, se encuentra el objeto *Fixture*. Este objeto es el encargado de definir principalmente, la forma geométrica del *Body*. Un *Body* puede adoptar una forma primitiva simple como un cubo, un círculo u otro polígono formado por múltiples vértices. La *Fixture,* además, es capaz de configurar parámetros como la densidad o la fricción de un cuerpo.

Incidiendo en el objeto *Fixture*, cabe resaltar que *libGDX* proporciona una magnífica herramienta bajo el nombre de *Physics Body Editor* [21] para generar fácilmente la forma poligonal de elementos complejos, calculando automáticamente el conjunto de vértices que los forman. En el *Apéndice A6* se halla una ilustración de esta herramienta.

4.3.7 Detección de la acción de encestar

En primer lugar, es necesario recordar que durante una partida de *Score Faster*, cada jugador puede recibir dos puntuaciones diferentes en función del tipo de canasta que haya realizado:

- **Canasta limpia:** El jugador recibirá tres puntos si su pelota no ha colisionado con elementos del escenario excepto con la propia canasta y su aro.
- **Canasta obstaculizada:** El jugador recibirá únicamente dos puntos si su *sprite* ha colisionado con cualquier otro elemento del escenario no incluido en una canasta limpia.

Para diferenciar y clasificar los lanzamientos satisfactorios de los jugadores, se ha empleado la clase *ContactListener* que implementa dos métodos esenciales para esta labor: *beginContact* (se ejecuta cuando dos cuerpos de *Box2d* colisionan) y *endContact* (se invoca cuando estos dos cuerpos dejan de colisionar). En el primer método se detecta que los dos cuerpos correspondan al *sprite* del jugador y a la canasta. Más tarde, se comprueba que la posición del *sprite* esté entre los límites de la canasta para dar por válido el

lanzamiento. En caso de que en este método se detecte una colisión del *sprite* con otro elemento del escenario, la canasta se convertirá a tipo *obstaculizada*. Por otro lado, en el método *endContact* se comprueba que el *sprite* haya entrado realmente dentro de la canasta.

4.3.8 *Módulos del proyecto* Servidor

El proyecto *Servidor* desarrollado en *Eclipse IDE* ha necesitado la implementación de menos clases que en el proyecto *Cliente*, a pesar de ello, su complejidad ha estado a la altura de éste. Este proyecto está formado por tres módulos claramente diferenciados e ilustrados en la captura siguiente:

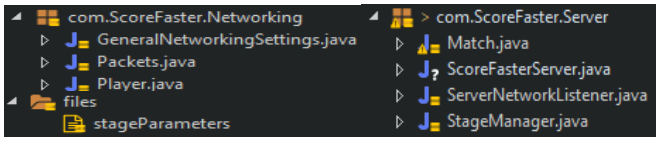


Figura 13: Módulos del proyecto Servidor

- **Módulo *Networking*:** Este módulo contiene tres clases que deben concocer tanto el *Cliente* como el *Servidor*. La clase *GeneralNetworkingSettings* contiene la definición de parámetros de red comunes como los puertos para los protocolos *TCP y UDP*, el *timeout* de la conexión o la *IP* pública de la máquina que actúa como servidor. Por otro lado, dentro de la clase *Packets* se encuentra la definición de todos los mensajes de red que envían, reciben y procesan los dos proyectos. Finalmente, en la clase *Player* se definen todos los métodos asociados a los jugadores.
- **Módulo *files:*** Files es un módulo muy sencillo que únicamente incluye un fichero *stageParameters* que contiene los valores de los diferentes parámetros identificativos del escenario de juego como, por ejemplo, el nombre del escenario o el tiempo de juego.
- **Módulo *Server*:** Este módulo implementa toda la lógica del multijugador de la parte del *Servidor*. La clase *Match* representa el objeto que relaciona a los jugadores de una misma partida e implementa los métodos para iniciar la partida y el crónometro de ésta, avisar a los jugadores cuando el tiempo se haya agotado e identificar al ganador de la partida. En segundo lugar, se encuentra la clase *ScoreFasterServer*. Esta clase configura e inicia el servidor y además gestiona la lista de jugadores y partidas activas. Más tarde, en tercer lugar, se halla la clase *ServerNetworkListener* cuyos métodos procesan los paquetes de red recibidos por los clientes y ejecuta las operaciones lógicas de respuesta correspondientes. Por último, la clase *StageManager* implementa un método para la lectura del fichero *stageParameters*.

4.4 *Testing*

La fase de *Testing*, aunque se sitúe en el último lugar de la sección del ciclo de desarrollo, ha jugado un papel relevante en este proyecto. Las pruebas realizadas en *Score Faster* se han focalizado en el apartado multijugador y en la fluidez de la jugabilidad. En este sentido, la librería *KryoNet* ha facilitado la validación de estas pruebas gracias al sistema de *logs* que incorpora con el que se ha podido observar con precisión el envío y recepción de los paquetes de red.

Por otro lado, se han realizado pruebas de usabilidad con usuarios reales para tratar de identificar errores existentes en la interfaz de usuario o durante el transcurso de la partida. A su vez, se han podido obtener algunos aspectos a mejorar de la aplicación.

Un elemento clave en la detección de errores ha sido el análisis exhaustivo de las partes más críticas del proyecto como, por ejemplo, la creación de una partida en el servidor, los posibles problemas de conexión y desconexiones de los clientes o la coordinación del cliente y del servidor para la detección del final de la partida.

Para finalizar, se debe subrayar que todas las pruebas de validación seguidas se han planificado y documentado en el *Documento de testing del sistema* junto con el resultado obtenido.

5 RESULTADOS

Finalizada en este punto la etapa de desarrollo se procederá a evaluar los resultados obtenidos. En primer lugar, la aplicación ha sido testeada combinando diversos dispositivos reales y virtuales gracias a la *Android Virtual Machine* incluida en *Android Studio IDE*. Entre estos dispositivos existe una variedad de versiones *Android* probadas. A continuación, se proporciona una ilustración con los dispositivos virtuales probados. Todos ellos han podido ejecutar correctamente la aplicación sin ningún tipo de inconveniente:

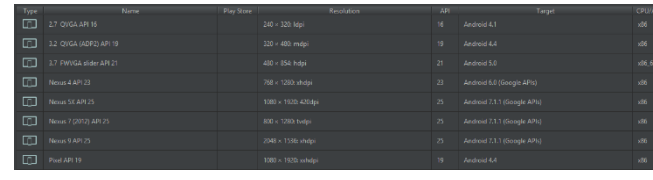


Figura 14: Dispositivos virtuales testeados con la aplicación

El trabajo dedicado al contenido audiovisual de la aplicación ha derivado en la incorporación de dos canciones (una para el transcurso de la partida y la otra para el resto de pantallas) y en la implementación de cuatro pantallas diferentes.

En el *Apéndice A7* se puede observar la transformación que ha sufrido el diseño inicial de *GameScreen* hasta adoptar la apariencia final.

6 CONCLUSIONES

Al principio de este proyecto se propuso como objetivo desarrollar una aplicación multijugador en línea y multiplataforma que incorporará una arquitectura *cliente-servidor* y a su vez implementará una idea de juego capaz de desarrollar la habilidad de los jugadores de forma competitiva. No obstante, las continuas dificultades surgidas en el apartado multijugador han supuesto un verdero reto que se ha debido aceptar y afrontar para finalizar con éxito este proyecto. Concretamente, la gestión y procesamiento de los mensajes de red enviados y recibidos entre el *Cliente* y el *Servidor* han constituido un auténtico rompecabezas que se ha resuelto con paciencia y mucha dedicación.

A pesar de ello, se ha conseguido presentar un videojuego con una jugabilidad fluida capaz de interconectar a dos jugadores en una misma partida en la cual se enfrentarán por obtener la mayor puntuación de la partida. Cada jugador es capaz de seleccionar entre trece apariencias diferentes para su *sprite,* así como utilizar un alias en cada partida o escoger el peso de su pelota.

En un futuro cercano, me gustaría implementar los requisitos secundarios que a día de hoy no se encuentran disponibles en *Score Faster*. Entre ellos, se encuentra lo siguiente:

- Una herramienta de creación de escenarios disponible para los jugadores a través de la cual puedan generar sus propios escenarios y compartirlos con otros jugadores.
- Implementar objetos utilizables en las partidas para entorpecer la jugabilidad del jugador oponente o mejorar la del propio jugador.
- Implementar un chat textual y de *emojis* que los jugadores puedan usar a lo largo de una partida.
- Ampliar el sistema de personalización del que dispone *Score Faster* añadiendo objetos que puedan ser comprados con las recompensas ofrecidas al ganar una partida y que los jugadores puedan equipar en sus *sprites*.

AGRADECIMIENTOS

Quisiera agradecer a las siguientes personas la ayuda que me han prestado en la realización de este trabajo de final de grado y en toda mi trayectoria académica en la UAB:

A mi tutora Yolanda Benítez Fernández por dirigir mi TFG y destinar su tiempo en aconsejarme y evaluarme.

A todos mis compañeros y amigos, por hacer más livianas y llevaderas las horas de clase y los exámenes.

A Natalia Salvador Téllez, por apoyarme en esta larga y sufrida lucha de cuatro años y por prestarme ayuda en el diseño de mi proyecto.

Y finalmente, a mi familia por confiar plenamente en mi y en mis capacidades.

BIBLIOGRAFÍA

[1] SCRUM (2017) [En línea}. *Proyectos Agiles Org*. [Consultado: 26 de junio de 2017], Disponible en Internet: https://proyectosagiles.org/que-es-scrum/

[2] CROSS-PLATFORM GAME DEVELOPMENT (2017) [En línea}. *libGDX*. [Consultado: 26 de junio de 2017], Disponible en Internet: https://libgdx.badlogicgames.com/

[3] TCP/UDP CLIENT/SERVER LIBRARY FOR JAVA, BASED ON KRYO (2017) [En línea}. *KryoNet.* [Consultado: 26 de junio de 2017], Disponible en Internet: https://github.com/EsotericSoftware/kryonet

[4] CROSS-PLATFORM (2017) [En línea}. *libGDX*. [Consultado: 26 de junio de 2017], Disponible en Internet: https://libgdx.badlogicgames.com/

[5] *OPENGL ES SUPPORT* (2017) [En línea}. *libGDX*. [Consultado: 26 de junio de 2017], Disponible en Internet: https://github.com/libgdx/libgdx/wiki/Opengl-es-support

[6] *LIBGDX* PROJECT SETUP TOOL "GDX-SETUP.JAR" (2017) [En línea}. *libGDX*. [Consultado: 26 de junio de 2017], Disponible en Internet: https://bitly.com/1i3C7i3

[7] REGISTERING CLASSES (2017) [En línea}. *KryoNet.* [Consultado: 26 de junio de 2017], Disponible en Internet: https://github.com/EsotericSoftware/kryonet#registering-classes

[8] RUNNING A SERVER (2017) [En línea}. *KryoNet.* [Consultado: 26 de junio de 2017], Disponible en Internet: https://github.com/EsotericSoftware/kryonet#running-a-server

[9] CONNECTING A CLIENT (2017) [En línea}. *KryoNet.* [Consultado: 26 de junio de 2017], Disponible en Internet: https://github.com/EsotericSoftware/kryonet#connecting-a-client

[10] ADOBE PHOTOSHOP CC (2017) [En línea}. Adobe. [Consultado: 26 de junio de 2017], Disponible en Internet: https://www.adobe.com/es/products/photoshop.html

[11] ADOBE ILLUSTRATOR CC (2017) [En línea}. *Adobe.* [Consultado: 26 de junio de 2017], Disponible en Internet: https://www.adobe.com/es/products/illustrator.html

[12] *VIEWPORTS* (2017) [En línea}. *libGDX*. [Consultado: 26 de junio de 2017], Disponible en Internet: https://github.com/libgdx/libgdx/wiki/Viewports

[13] *STRETCH VIEWPORT* (2017) [En línea}. *libGDX*. [Consultado: 26 de junio de 2017], Disponible en Internet: https://github.com/libgdx/libgdx/wiki/Viewports#stretchviewport

[14] SEIS DENSIDADES GENERALIZADAS (2017) [En línea}. *Android Developer*. [Consultado: 26 de junio de 2017], Disponible en Internet: https://developer.android.com/guide/practices/screens_support.html

[15] TEXTURE PACKER GUI (2017) [En línea}. *Aurelien Ribon's dev blog*. [Consultado: 26 de junio de 2017], Disponible en Internet: http://www.aurelienribon.com/blog/category/desktop-applications/libgdx-texturepacker-gui/

[16] EXTENDING THE SIMPLE GAME [En línea}. *libGDX*. [Consultado: 26 de junio de 2017], Disponible en Internet: https://github.com/libgdx/libgdx/wiki/Extending-the-simple-game

[17] SCENE2D [En línea}. *libGDX*. [Consultado: 27 de junio de 2017], Disponible en Internet: https://github.com/libgdx/libgdx/wiki/Scene2d

[18] SCENE2D.UI [En línea}. *libGDX*. [Consultado: 27 de junio de 2017], Disponible en Internet: https://github.com/libgdx/libgdx/wiki/Scene2d.ui

[19] BOX2D [En línea}. *libGDX*. [Consultado: 27 de junio de 2017], Disponible en Internet: https://github.com/libgdx/libgdx/wiki/box2d

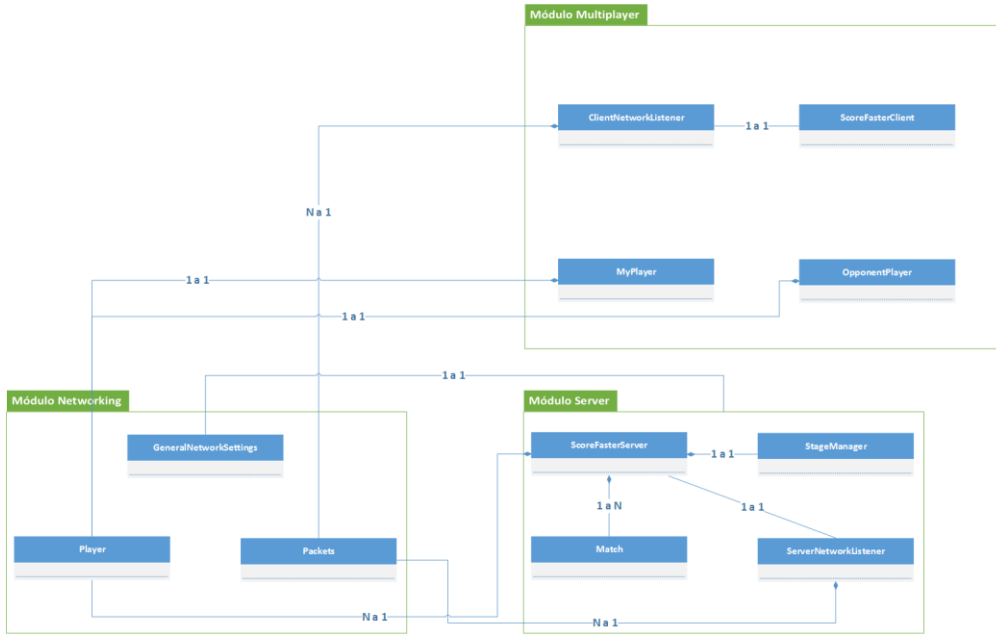
[20] CREATING A WORLD [En línea}. *libGDX*. [Consultado: 27 de junio de 2017], Disponible en Internet: https://github.com/libgdx/libgdx/wiki/box2d#creating-a-world

[21] PHYSICS BODY EDITOR [En línea}. *Aurelien Ribon's dev Blog*. [Consultado: 27 de junio de 2017], Disponible en Internet: http://www.aurelienribon.com/blog/projects/physics-body-editor/

APÉNDICE

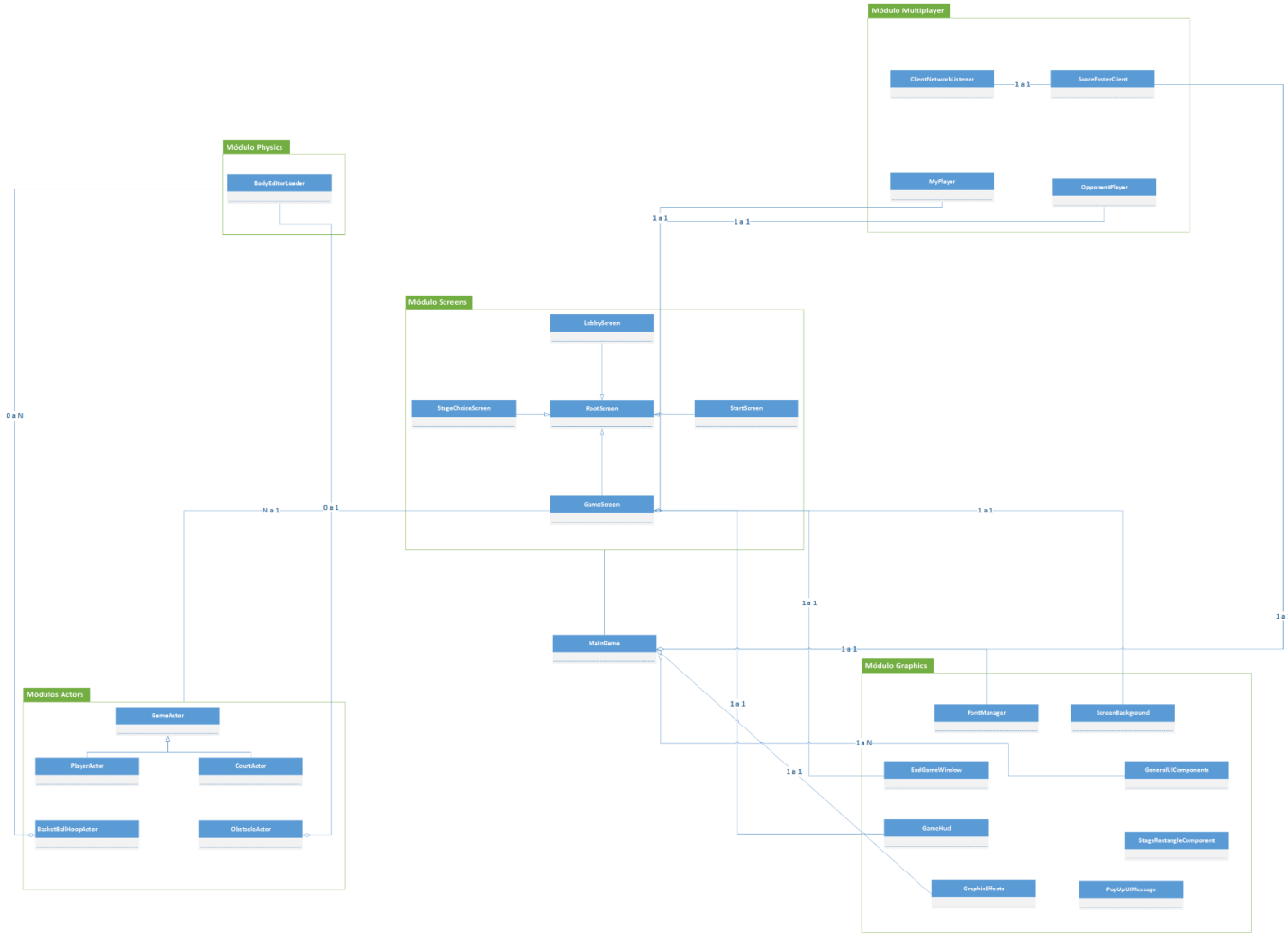
A1. DIAGRAMA DE CLASES DE LOS MÓDULOS DE MULTIJUGADOR

El siguiente diagrama representa las relaciones existentes en los tres módulos de multijugador de este proyecto:



A2. DIAGRAMA DE CLASES DEL CLIENTE

El diagrama de clases del proyecto *Cliente* muestra todas las relaciones entre los cinco módulos que contiene:



A3. DISTRIBUCIÓN DE USUARIOS POR VERSIONES DE ANDROID

La distribución de usuarios por versiones de *Android* actualizada en este mes de junio es la siguiente:

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.8%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.8%
4.1.x	Jelly Bean	16	3.1%
4.2.x		17	4.4%
4.3		18	1.3%
4.4	KitKat	19	18.1%
5.0	Lollipop	21	8.2%
5.1		22	22.6%
6.0	Marshmallow	23	31.2%
7.0	Nougat	24	8.9%
7.1		25	0.6%

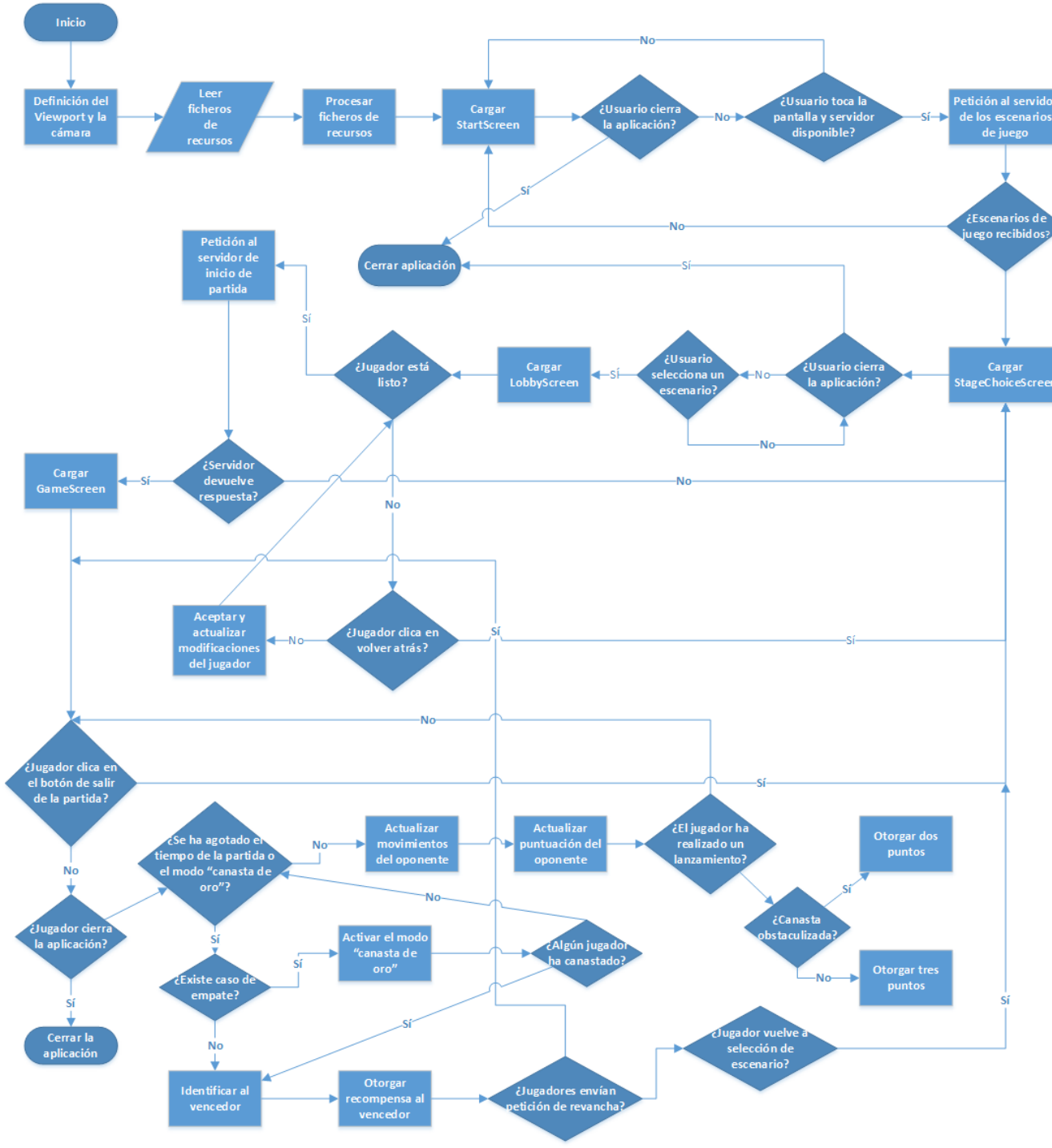
A4. SEIS DENSIDADES GENERALIZADAS DE PANTALLA Y SU FACTOR DE ESCALA

En *Android*, existe un conjunto de seis densidades generalizadas cada una de las cuales tiene asociada un factor de escala que se debe utilizar para transformar la resolución de las texturas originales:

Densidad	DPI (Puntos por pulgada)	Factor de escala
ldpi (baja)	120	0,75x
mdpi (media)	160	1,0x
hdpi (alta)	240	1,5x
xhdpi (extraalta)	320	2,0x
xxhdpi (extra extraalta)	480	3,0x
xxxhdpi (extra extra extraalta)	640	4,0x

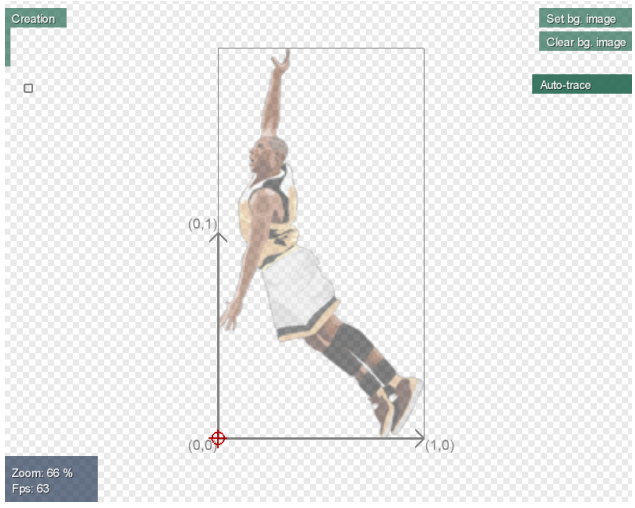
A5. DIAGRAMA DE FLUJO DEL FUNCIONAMIENTO DEL PROYECTO *CLIENTE*

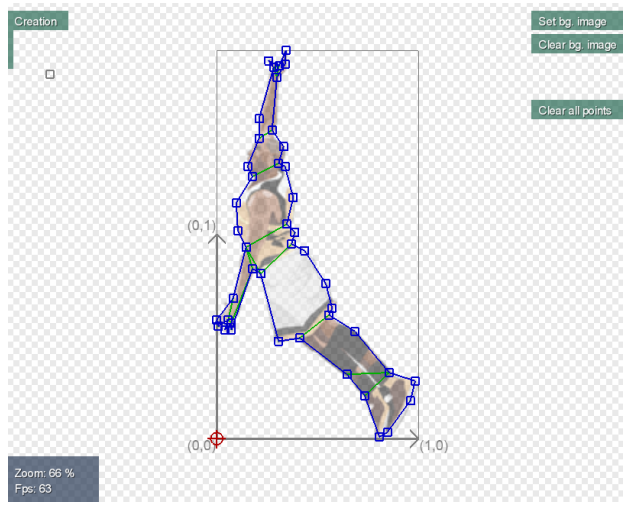
Este diagrama de flujo es la representación gráfica de todo el proceso de ejecución del proyecto *Cliente* de *Score Faster*.



A6. ILUSTRACIÓN DE LA HERRAMIENTA *PHYSICS BODY EDITOR*

La herramienta *Physics Body Editor* ha sido diseñada para generar fácilmente el conjunto de vértices que definen la física de una textura. Cuenta con una opción llamada *Auto Trace* que traza automáticamente los vértices de la textura. A continuación, se muestra el resultado obtenido por esta herramienta aplicando la técnica *Auto Trace* sobre la textura del obstáculo del escenario de juego:





A7. COMPARACIÓN DEL DISEÑO DE *GAMESCREEN* CON EL RESULTADO REAL

El diseño de la pantalla *GameScreen* probablemente es el que más cambios ha sufrido en comparación con el resultado final. A continuación, se puede visualizar esta evolución:

Diseño inicial

Diseño final