Universidad de Barcelona

Trabajo de Fin de Grado

MrRobotto 3D Engine

Autor:
Aarón Negrín Santamaría

Supervisor:
Anna Puig Puig

Facultad de Matemáticas

junio de 2015

Declaración de Autoría

Yo, Aarón Negrín Santamaría, declaro que el siguiente documeto titulado, 'MrRobotto 3D Engine' y el trabajo presentado en él es trabajo propio.

Firma:

Fecha:

"La mayoría de los buenos programadores programan no porque esperan que les paguen o que el público los adore, sino porque programar es divertido."

Linus Torvalds

UNIVERSIDAD DE BARCELONA

Abstract

Facultad de Matemáticas

Grado en Ingeniería Informática y Matemáticas

MrRobotto 3D Engine

by Aarón Negrín Santamaría

TODO

Índice general

Índice de figuras

Índice de cuadros

A mis padres, familia y amigos

Capítulo 1

Introducción

MrRobotto 3D Engine es un motor de juegos de código libre para la plataforma Android escrito íntegramente en el lenguaje Java.
El hecho de usar Java como lenguaje principal asegura una total y sencilla integración dentro de aplicaciones que hagan uso del 3D tanto nuevas como ya existentes.
Por otra parte, y para simplificar el proceso de desarrollo de software, se incluye la herramienta MrRobotto Studio, herramienta que permite que desde una sencilla y práctica interfaz web pueda editarse la escena, permitiendo ver los resultados en un dispositivo Android instantáneamente y sin necesidad de pasar por lentas etapas de compilación de código.
Por último, y además de todo esto, se ha diseñado un formato de fichero propio para la representación de la escena tridimensional, dicho fichero puede generarse a partir de una escena del software Blender mediante el uso de scripting o bien mediante las herramientas proporcionadas en MrRobotto Studio.

1.1. Historia de MrRobotto 3D Engine

La idea de crear MrRobotto 3D Engine nace a razón de un proyecto desarrollado en una de las asignaturas del Grado de Ingeniería Informática, es en ese momento donde se fragua la idea de crear un motor de juegos mucho más sofisticado, extensible y robusto, pero sobretodo, de código libre.
El nombre del proyecto surge del primer día en el que empecé a codificar las primeras líneas de código, día en el cuál, mientras visionaba una de mis series favoritas y en la que la canción de apertura se podía escuchar un estribillo pegajoso el cuál decía *"Doumo arigatou Mr. Roboto"*(Muchas gracias Sr. Robot) pues me pareció buena idea tomar de

ahí el nombre a forma de homenaje a dicha serie, a la propia casualidad, y al hecho de que era un proyecto pensado para la plataforma Android.

1.2. Objetivos y Motivación

TODO:

1.3. Tecnologías utilizadas

1.3.1. MrRobotto 3D Engine

Las distintas tecnologías utilizadas para el desarrollo de MrRobotto 3D Engine constituyen las mismas que las utilizadas en cualquier aplicación Android.

- Android SDK.
- Android Studio como IDE de desarrollo.
- Gradle como sistema de construcción, herramienta por defecto en Android Studio.
- GitHub como repositiorio de software.

A la hora de escoger la tecnología a usar se planteó la opción de implementar MrRobotto 3D Engine haciendo uso de la API nativa de Android, sin embargo se desestimó la idea por cuestiones de velocidad en el desarrollo y calidad del software generado.
Además, y tras la realización de múltiples pruebas se comprobó que aunque el rendimiento resultaba algo menor haciendo uso de código no nativo, si este era optimizado teniendo en mente los consejos de Android para aplicaciones que hacen uso de OpenGL y un tratamiento minucioso de la gestión de la memoria se podían conseguir resultados similares, incluso al tratar con escenas complejas.
Por otra parte, y como ya se comentó, uno de los objetivos era la fácil integración dentro de cualquier proyecto Android existente, dicha tarea resulta extremadamente sencilla si se hace uso de herramientas como Android Studio + Gradle.
Por último se ha evitado el uso de dependencias externas, de forma que MrRobotto 3D Engine es un proyecto totalmente autocontenido.

1.3.2. MrRobotto Studio

Capítulo 2

MrRobotto 3D Engine

Ahora que se conoce la finalidad de MrRobotto 3D Engine se procederá a explicar su funcionamiento, sus partes, y como estas se relacionan entre sí

2.1. Diseño de MrRobotto 3D Engine

A lo largo de esta sección se explicarán las distintas decisiones en el diseño de MrRobotto 3D Engine. Tal y como se aprecia en el esquema principal, ver 2.1, se buscan varios objetivos.
Por una parte se quiere poder portar desde un software de edición 3D como blender a un formato propio y este a su vez que sea fácil de editar proporcionando un editor capaz de ello.
Por otra parte se busca desacoplar la interfaz que usará el usuario del núcleo del motor proporcionando así seguridad de uso y una interfaz estable abierta a la extensión pero cerrada a la modificación.
Además, y tal y como se espera de un motor de gráficos 3D, está la gestión de la escena y de los recursos para una correcta visualización.
Y por último, pero no menos importante, la capacidad de interactuar con eventos procedentes del dispositivo.

2.1.1. Formato .MRR

La decisión de usar un formato propio frente a usar alguno ya ampliamente aceptado como podría ser *.fbx* o *.dae* viene fomentado por la necesidad de controlar en su totalidad los distintos datos y sobretodo, la forma en la que estos se almacenarán dentro de

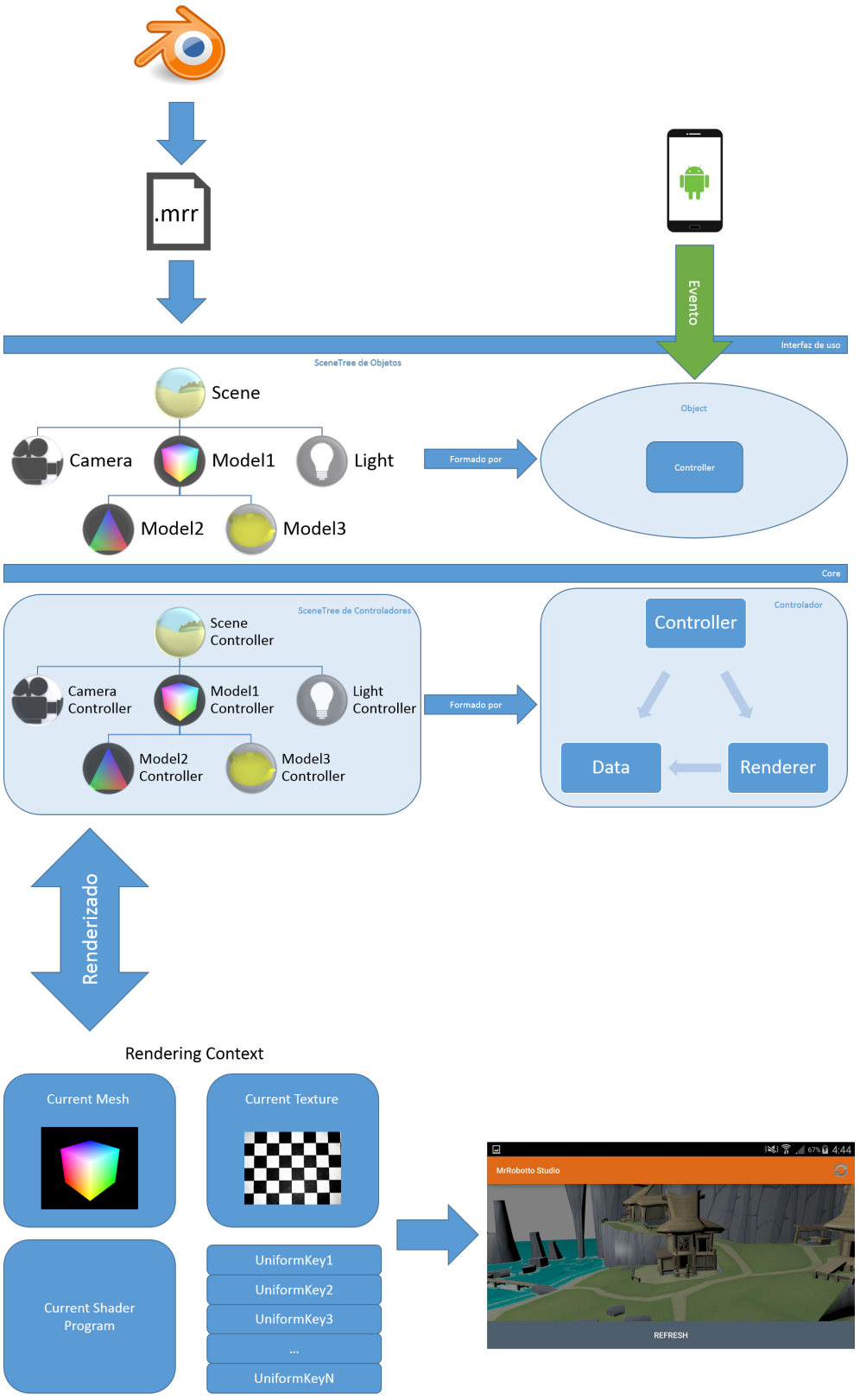


FIGURA 2.1: Esquema principal de MrRobotto 3D Engine

una aplicación.

Por ejemplo de cara al rendimiento se decidió almacenar los vértices de forma intercalada tal y como se recomienda en [1] a la vez que usar en todo momento *Index Buffer Objects* como se recomienda en [2].
También se tomó en cuenta la necesidad de modificar la configuración de cómo se mostrará la escena sin necesidad de código, si no la realización de esta mediante un fichero. Por ejemplo, gracias al formato MRR se puede decidir si una malla va a ser dibujada usando triángulos o bien líneas únicamente.

Aún así estas no son las únicas razones para proporcionar un formato propio, algunas de las citadas configuraciones podrían llevarse a cabo fácilmente haciendo uso de otros tipos de formatos más comunes. El mayor motivo detrás de la creación del formato MRR es la creación de un formato que fuera *Scene Object Centric*.
Con este término, *Scene Object Centric*, se hace alusión a un diseño centrado en los objetos de la escena, donde cada elemento es independiente del resto de elementos que pudiesen estar definidos en ella. De esta forma la tarea de comunicar un objeto con otro recae sobre el motor y no es arrastrada desde el formato. Únicamente hay un único punto de conexión entre los objetos y este se realiza mediante los denominados UniformKeys y UniformsGenerators

Toda la información acerca del formato y cómo está organizada la información almacenada en él puede verse en A

2.1.2. El SceneTree

El SceneTree es la estructura fundamental de almacenamiento de los objetos de la escena. Posee una estructura de árbol donde cada nodo puede tener un número variable de hijos, pero lo que lo diferencia de una estructura de árbol normal es que sus nodos se encuentran indexados tanto por el nombre del dato almacenado en el nodo como por su tipo, permitiendo así realizar búsquedas dentro de la estructura en $O(1)$
Su funcionamiento se explicará de forma detallada en 2.2.2.1

2.1.3. Elementos de una Escena

Para los elementos constituyentes de una escena 3D se decidió emular el diseño de componentes de otros motores 3D a la vez que la necesidad de que la API fuese lo más

sencilla y funcional que se pudiera.

2.1.3.1. Object

Los objetos genéricos tienen la función de ser la base de la jerarquía del resto de objetos de la escena, y en él ser verán reflejados todas las acciones compartidas. A saber:

- Gestión de las transformaciones geométricas:
  Se espera que un objeto de una escena pueda ser transformado por traslaciones, rotaciones y escalados dentro de la escena que los contiene.
- Inicialización:
  La inicialización de los objetos de la escena hace referencia a varias acciones que deben ser llevadas a cabo antes del uso del objeto, a saber estas pueden ser:
  - Establecimiento del comportamiento de los objetos frente a los eventos
  - Configuración de ciertos elementos en la GPU, como podría ser iniciar un Vertex Buffer Object.
  - Configuración previa dependiente del tamaño de la ventana donde va a realizarse el dibujado de la escena, como por ejemplo, la creación de la matriz de proyección.
- Actualización de los diferentes elementos que dependen de él en cada ciclo de renderizado, como podría ser la gestión de eventos o envío de datos a la GPU
- Métodos de comunicación con la escena y otros objetos

2.1.3.2. Model

Los modelos 3D son seguramente el elemento más representativos dentro del motor puesto que son los únicos elementos realmente visibles de la escena.
Los modelos no representan únicamente las mallas tridimensionales visibles en el mundo, si no que además engloba otras funcionalidades tales como

- Gestión de la visualización final:
  Un modelo será capaz de gestionar como será su aspecto final en la escena, es decir, un modelo será capaz de hacer uso de un sombreador (*Shader*) independiente al resto.

- Gestión de las animaciones:
  En el caso de que el modelo tuviera animaciones basadas en esqueleto, es el modelo quien tiene el control sobre este.
  En otros motores los esqueletos suelen manejarse de forma independiente al resto de elementos y como un objeto de la escena más, pero en el caso de MrRobotto 3D Engine se decidió que un esqueleto carece de sentido si no existe un modelo sobre el cuál se aplique.

- Gestión de materiales y texturas:
  Las texturas y materiales utilizados en cada modelo, aunque puedan ser compartidos entre multipes modelos, es tarea de cada uno el controlar la textura o material usado en cada momento.

2.1.3.3. Cámaras y Luces

La cámara es el objeto de escena encargado de proporcionar el punto de vista a través del cual será visualizado el mundo.
Además de ello también ofrece el tipo de proyección a utilizar, a escoger entre ortogonal y perspectiva.

Por otra parte las luces son los objetos de la escena encargados de proporcionar la iluminación

2.1.3.4. Scene

Posiblemente este es el objeto más conflictivo conceptualmente ya que no parece posible que la escena sea considerada como un objeto de la escena, sin embargo hay una justificación para esta decisión.
Aunque la escena no genere estrictamente hablando un cambio visible si se decide hacer una transformación geométrica sobre ella, o al reaccionar frente a un evento, la finalidad de que sea considerada como objeto de la escena es que de esa manera es posible configurar algunos aspectos visibles con datos procedentes desde el fichero MRR. Es decir, de esta forma algunos atributos como por ejemplo el color plano con el que se limpia la escena puede cambiarse en la configuración y no desde código.
Aún así existen más motivos que justifican esta decisión, y es que la escena, conceptualmente, es un buen lugar donde albergar la responsabilidad de la generación de uniforms

que sean dependientes de varios objetos. Este concepto se explica en la siguiente apartado.

2.1.4. Los Conceptos de UniformKey y UniformGenerator

En este apartado se tratará cómo se realiza el envío de datos desde CPU a GPU.
Con este fin se ha implementado en MrRobotto 3D Engine tres elementos muy relacionados entre sí.

- En primer lugar se tienen los **Uniforms**, que no son si no una representación de los *uniforms* usados en los *Shader Programs*.
  Dichos Uniforms poseen, entre otras cosas, dos campos fundamentales que hacen posible la comunicación

  - El `UniformType`, que determina el tipo uniform al que se hace referencia, como podría ser la matriz de proyección o la *ModelView Matrix*.
    Podría pensarse que este papel ya es desempeñado por el nombre del uniform dentro del *Shader Program*, sin embargo, definiendo el tipo del Uniform de esta forma permite que distintos *Shader Programs* puedan requerir un mismo Uniform sin necesidad de que su código presente las mismas variables obligatoriamente.
  - El `UniformId` representa el identificador dado por la GPU de la variable *uniform* en cuestión.

- Por otra parte se tienen los **UniformGenerators**, estos elementos tienen como función principal asignarle a cada objeto la funcionalidad necesaria para generar los valores de los Uniforms que este objeto sea el encargado de producir. Por ejemplo, la cámara será la encargada de producir la *View Matrix* pero no la *Model View Matrix* ya que esta depende de la cámara y también de la *Model Matrix*, la cuál es generada por un modelo.
  Todo UniformGenerator poseera un identificador único dentro del conjunto de generadores de un objeto individual.

- Por último está el elemento que actúa como puente entre los dos anteriormente detallados, los **UniformKeys**.
  Entre las funciones de los UniformKeys las más destacables son:

  - Almacenar el valor de los uniforms generados en los UniformGenerators.

- Proporcionar información acerca de los uniforms que un objeto es capaz de generar.
  De esta última afirmación se deduce que todo UniformKey ha de tener asociado obligatoriamente un UniformGenerator

Para conseguirlo el UniformKey cuenta con un campo que le permite enlazarse con el objeto Uniform, el ya comentado `UniformType` y por otra parte, otro campo que le permita comunicarse con su UniformGenerator asociado.

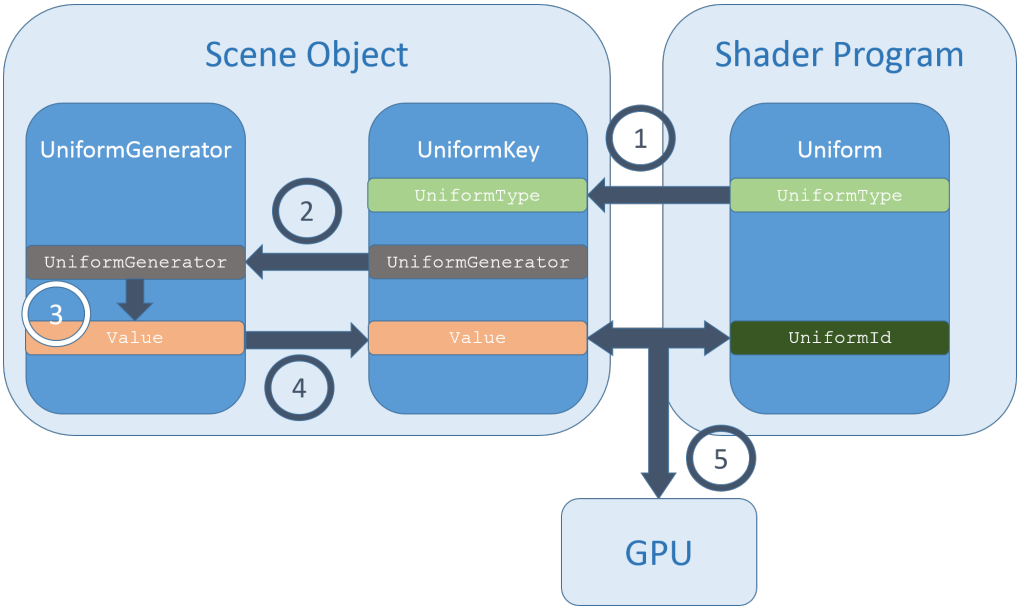


FIGURA 2.2: Funcionamiento de UniformKeys y UniformsGeneratos.
1. Búsqueda del objeto con el UniformType capaz de generar el Uniform dado.
2. Búsqueda dentro del objeto del UniformGenerator asociado al UniformKey.
3. Generación del valor del Uniform.
4. Almacenamiento del valor dentro del UniformKey.
5. Envío a la GPU el valor del Uniform generado.

De estos elementos, tanto Uniforms como UniformKeys son proporcionados por el fichero MRR como una parte más de un objeto, sin embargo, los UniformGenerators, son proporcionados vía código. Algunos de estos UniformGenerators, o al menos, los más usuales son provistos por el propio motor, sin embargo, la API ofrece la posibilidad de que el usuario pueda crear un nuevo UniformGenerator así como modificar el comportamiento de alguno ya existente tal y como se explica en.
TODO: Dar la referencia en el documento de como crear un uniform generator

TODO: Explicar los niveles

TODO: Explicar caso del MVP

2.1.5. La Necesidad del Rendering Context

El **Rendering Context** es el elemento encargado organizar y optimizar los datos utilizados al dibujar la escena.

Una de sus principales funcionalidades es la de controlar la lista de UniformKeys requeridos por cada objeto a la hora del renderizado.

Además también posee un papel vital de cara al rendimiento, pues es el encargado de minimizar los cambios de estados de la GPU. Esto se consigue por una parte realizando una ordenación previa de los objetos de la escena, haciendo que el coste de todos los cambios de estados a realizar durante la generación de un frame sea el menor posible y por otra parte, controlando qué elementos se encuentran presentes en la GPU, tarea desempeñada por el Rendering Context.

Por ejemplo, se ordenan todos los modelos que posean la misma textura, esta se envía una única vez a la GPU y es usada por todos estos modelos de, de esa forma se minimizará el número de cambio de texturas.

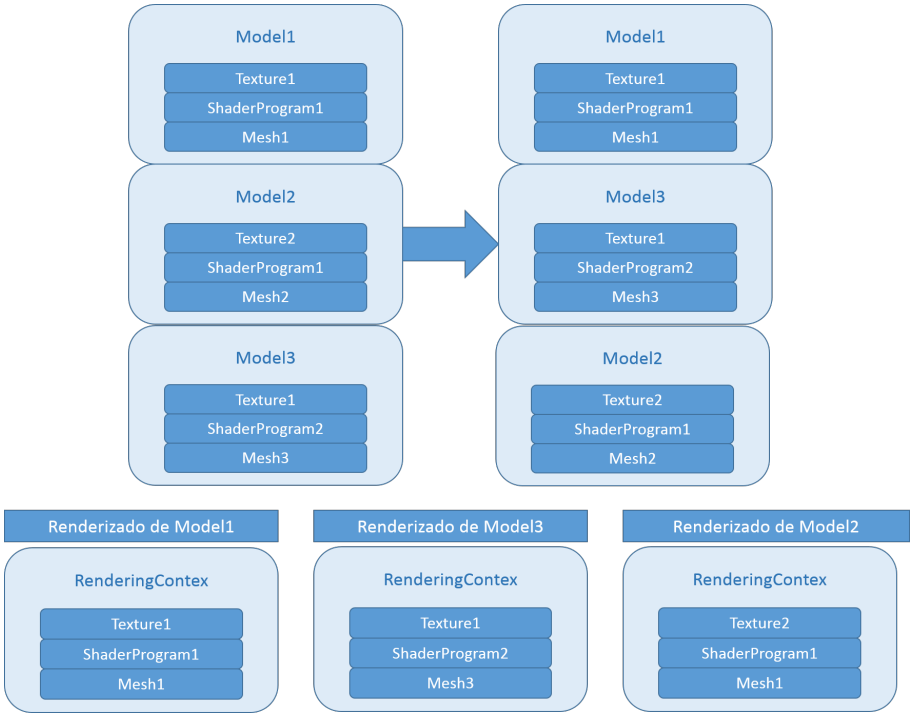


FIGURA 2.3: Ejemplo del funcionamiento del Rendering Context

2.1.6. Eventos

2.1.7. Algoritmo de Renderizado

Ahora que se han tratado todos los conceptos fundamentales que definen el funcionamiento del motor se puede establecer el comportamiento del ciclo del renderizado desde su inicialización hasta la generación de la imagen final.

En primer lugar se presenta el esquema de inicialización de los elementos gráficos.
Como se puede apreciar en primer lugar se inicializan todos los objetos uno a uno, se hacen llamadas a la GPU como podrían ser compilaciones de Shader Programs o configuración de texturas, tras ello se acude al RenderingSorter, encargado de ordenar los objetos de la escena en función del coste de los cambios de estado en la GPU. Finalmente se realizan las inicializaciones que requieran conocer el tamaño de la ventana

TODO: Explicar los diagramas

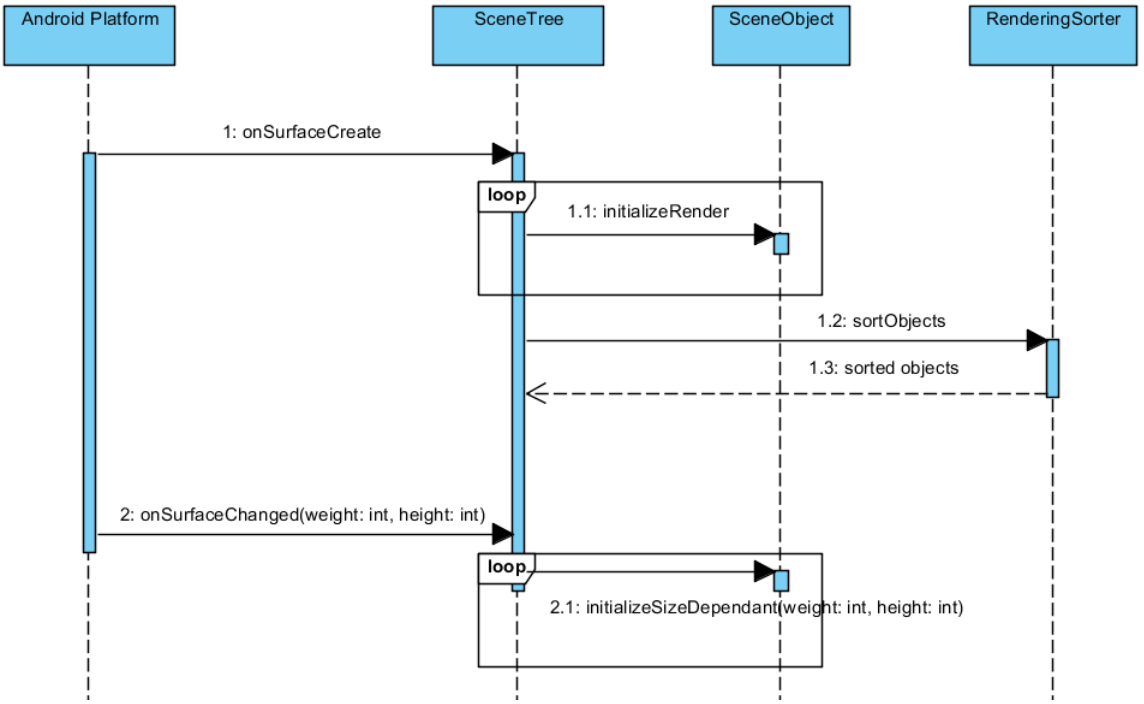


FIGURA 2.4: Inicialización de Recursos para el Renderizado

En el caso del renderizado de un frame lo primero que se realiza es actualizar el objeto según los eventos recibidos.
Tras ello tomamos el primer modelo de la lista de modelos ordenados generados en la inicialización y cargamos en la lista de UniformKeys todos los UniformKeys de este

modelo y del resto de objetos activos, a saber: la cámara actual, las luces y la escena. Una vez todos los UniformKeys han sido ordenados por nivel de dependencias se generan los valores de estos en los UniformGenerators y se envían a la GPU.

TODO: Editar y agregar la columna de model, recuerda que es para cada modelo

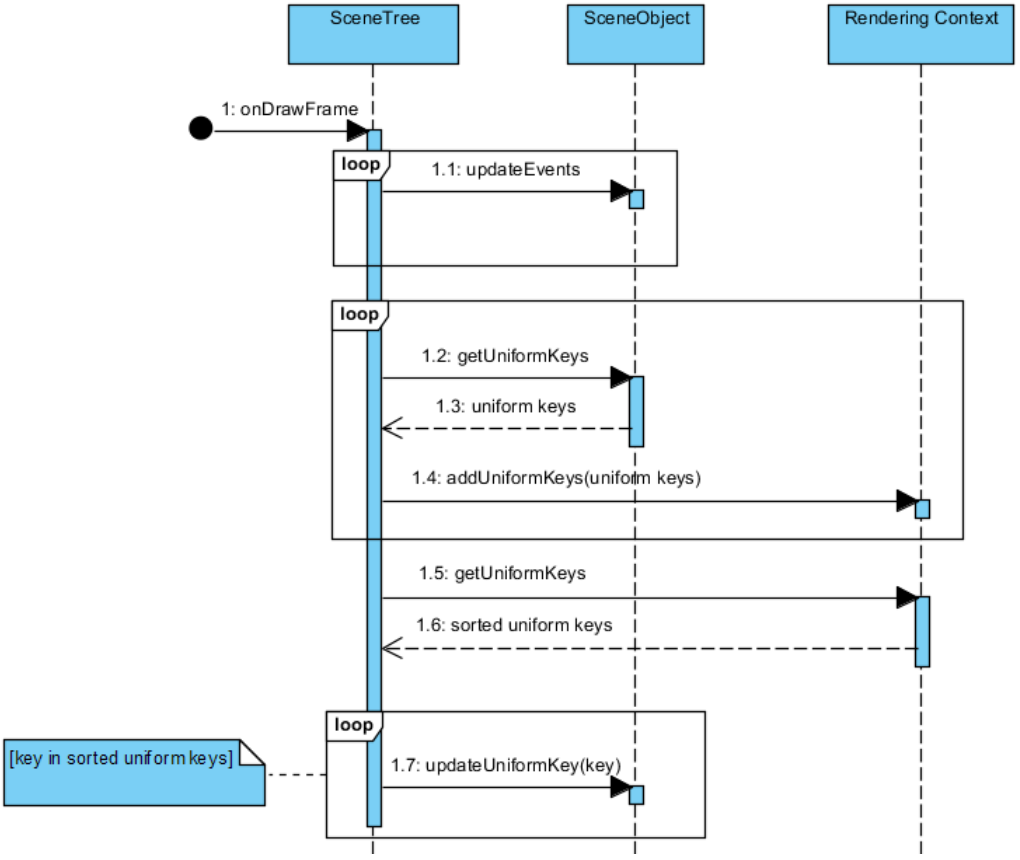


FIGURA 2.5: Renderizado de un Frame

2.2. Arquitectura de MrRobotto 3D Engine

2.2.1. Interfaz de Uso

La interfaz de uso está compuesta por los elementos pensados para ser usados por los usuarios de MrRobotto 3D Engine.
Dichos elementos proveen al usuario de acciones comunes al tratar con objetos en una escena 3D como por ejemplo trasladar un objeto, rotarlo, ejecutar una animación,... U otras funcionalidades más generales como la búsqueda de objetos dentro de una escena, control de la jerarquía o de gestión de eventos.
Además, dichas herramientas son gestionadas de forma muy similar a como se gestiona el

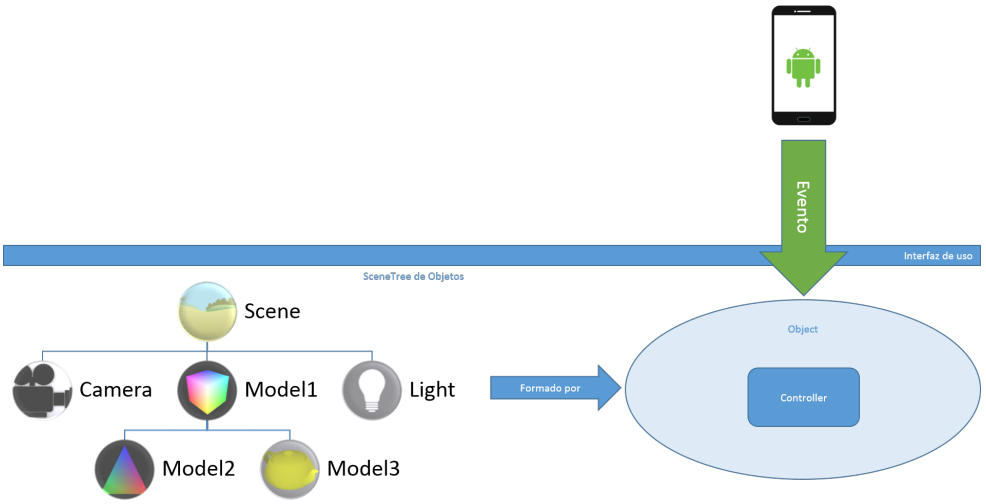


FIGURA 2.6: Interfaz del núcleo

ciclo de vida de las aplicaciones Android comúnes, permitiendo mantener un paradigma similar al de la plataforma sobre la que se trabaja, intentando así que el usuario se sienta lo más cómodo posible a la hora de integrar MrRobotto 3D Engine en su aplicación.

2.2.1.1. Integración en una Aplicación Android

A la hora de integrar MrRobotto 3D Engine dentro de una aplicación Android MrRobottoEngine es sin duda el primer objeto con el que se encuentra el usuario al usar MrRobotto 3D Engine. Es el encargado de la gestión del ciclo de vida de este, así como el encargado de gestionar los recursos usados y referencias a estos.

MrRobottoEngine
+MrRobottoEngine(androidContext : Context, surfaceView : MrSurfaceView)
+getResources() : MrResources
+setFps(fps : int) : void
+getSurfaceView() : MrSurfaceView
+setSurfaceView(surfaceView : MrSurfaceView) : void
+setAndroidContext(context : Context) : void
+getSceneTree() : MrSceneTree
+getObject(name : String) : MrObject
+getEventDispatcher() : MrEventDispatcher
+setEventDispatcher(eventDispatcher : MrEventDispatcher) : void
+loadSceneTree(inputStream : InputStream) : MrSceneTree
+loadSceneTreeAsync(inputStream : InputStream) : void
-freeResources() : void
+onInitialize() : void
+initialize() : void
+queueEvent(runnable : Runnable) : void

Para inicializar una instancia de esta clase el usuario requerirá por una parte de un *Context* de la plataforma Android, como podría ser una *Activity* y por otra una referencia

a una instancia de MrSurfaceView , esta clase hereda de la clase *View* de la plataforma Android y será donde se mostrará el contenido.

Una vez se ha obtenido una referencia a un objeto de la clase MrRobottoEngine el siguiente paso consiste en cargar una escena desde un *stream* de datos. Este *stream* de datos ha de contener datos en el formato MRR.
La carga de estos datos puede realizarse de dos formas distintas, de forma bloqueante o de forma no bloqueante, para ello se usarán los métodos loadSceneTree y loadSceneTreeAsync respectivamente.

Cuando la carga de datos haya finalizado se llamará automáticamente al método onInitialize, este método está pensado para ser sobrescrito en clases que hereden de MrRobottoEngine ya que proporciona un entorno seguro para acceder a la escena, sus objetos y proporcionar código de iniciación.

2.2.2. Objetos de la Escena

Una vez hemos cargado la escena que usaremos nos interesa conocer los distintos objetos que pueden utilizarse.

2.2.2.1. MrSceneTree

El objeto MrSceneTree es la interfaz que se le ofrece al usuario para acceder a la escena. Este es el punto de entrada a la hora de acceder a cualquier objeto.

2.2.2.2. MrObject

MrObject es la clase base de toda nuestra jerarquía y la que contiene los métodos más genéricos para el control de un objeto de la escena.
A pesar de ser una clase considerablemente importante de cara al uso, esta clase en realidad actúa como un envolvente de una clase de nivel inferior, *MrObjectController*, de la que se hablará más adelante.

Las posibles acciones que se pueden realizar con una instancia de MrObject podrían dividirse en cuatro grupos según su funcionalidad.

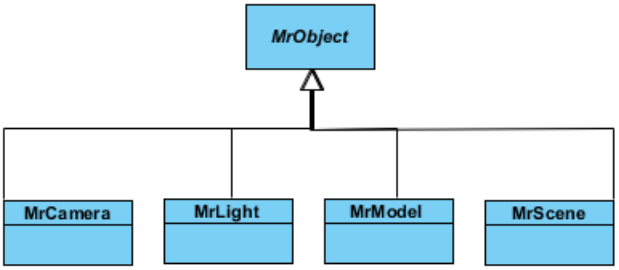
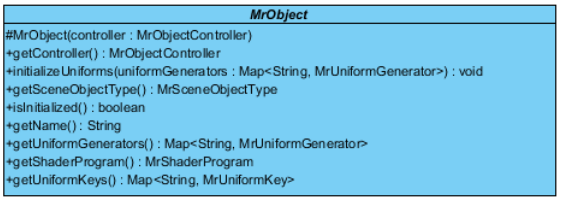


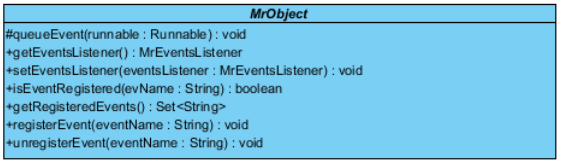
FIGURA 2.7: Jerarquía de MrObject

Estas son o bien métodos genéricos, como podrían ser el constructor u obtener el tipo de objeto, métodos orientados a la gestión de eventos, como agregar o eliminar un evento a procesar, de jerarquía, como acceder a los hijos o al padre de un objeto, o de transformación, como puede ser rotar, escalar o trasladar el objeto.

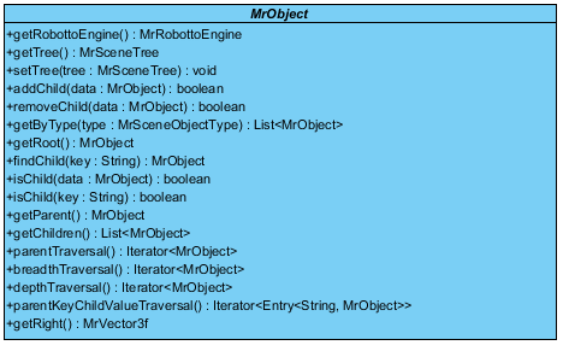
TODO: Aumentar el tamaño de la fuente



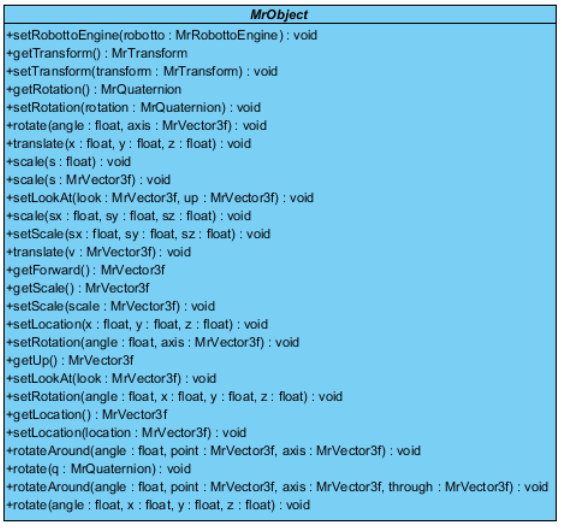
(A) Métods Genéricos



(B) Métodos de Gestión de Eventos



(C) Métodos de Gestión de JerarQuía



(D) Métodos de Transformaciones Geométricas

FIGURA 2.8: Métodos de MrObject agrupados por funcionalidad.

2.2.2.3. MrModel, MrCamera, MrLight y MrScene

2.2.3. Eventos

TODO: Gestión de eventos con mayor detalle, como la interfaz EventListener y EventDispatcher

2.2.4. Estructura de los Objetos

Como se ha comentado ya durante este documento, la clase MrObject y todas sus subclases, son interfaces de una estructura subyacente encargada de administrar los objetos de la escena, tanto desde la gestión de eventos, pasando por el renderizado, como la gestión de los datos del objeto de la escena.
Dicha estructura podría verse compuesta por tres partes diferenciadas.

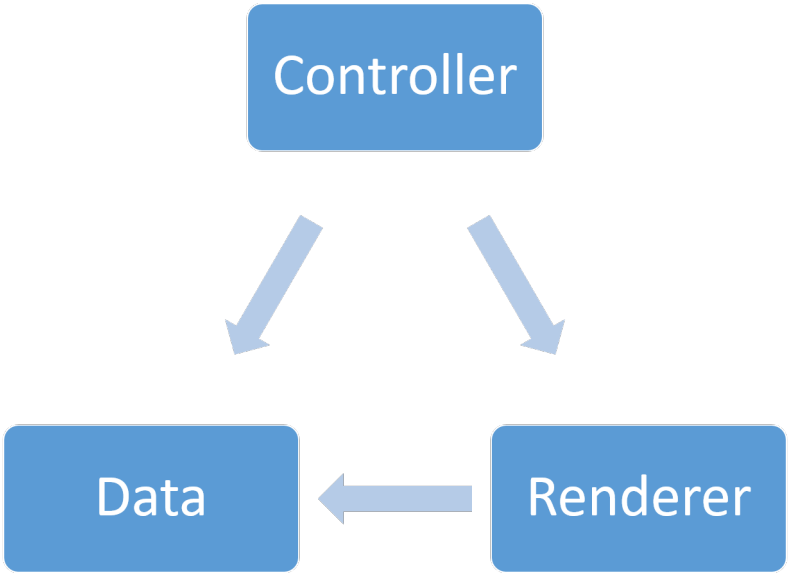


FIGURA 2.9: Estructura interna de los objetos.

Tal y como se puede apreciar en la figura 2.9 la estructura emula al patrón modelo-vista-controlador(MVC), y eso es precisamente lo que se buscaba, tres partes diferenciadas y cada una con unas responsabilidades definidas.

2.2.4.1. Contenedores de Datos

Los contenedores de datos serían la parte del modelo dentro del patrón MVC, su funcionalidad está restringida a almacenar datos necesarios del objeto que serán requeridos o bien por el controlador o bien por la parte del renderizador.

Todos los elementos contenedores de datos requieren que se herede de la clase MrObjectData
Todos los contenedores de datos se encuentran dentro del paquete mr.robotto.engine.core.data

2.2.4.2. Renderizadores

Los renderizadores hacen la función de la vista dentro del patrón MVC. Se encargan de tomar datos desde el modelo y presentarla como se requiera.
Los renderizadores están pensados para que todas las llamadas a OpenGL sean ejecutadas desde aquí, de esta forma todas las llamadas a la API se encontrarán concentradas en unas clases determinadas, lo cuál resulta muy beneficioso en cuanto a mantenibilidad de código se refiere.
Todos los objetos que se encarguen del renderizado deben implementar una cierta interfaz denominada MrObjectRender en la cuál es necesario implementar los siguientes métodos

- `void initializeRender(MrRenderingContext context, MrObjectData link)`
  En este método al renderer se le asignan el rendering context sobre el cuál trabajará y además el objeto al que estará asociado.
  Además en este método se inicia el objeto desde el punto de vista gráfico, es decir, se realizan tareas tales como configurar texturas, compilar Shader Programs y similares.
- `void initializeSizeDependant(int w, int h)`
  En este método se inician los recursos que dependan del tamaño de la ventana usada en ese momento.
- `boolean isInitialized()`
  Simplemente comprueba si el renderizador ha sido iniciado
- `void render()`
  Este método es el encargado de pasar datos a la tarjeta gráfica en cada frame.

Todos los renderizadores de objetos se encuentran dentro del paquete mr.robotto.engine.core.render

2.2.4.3. Controladores

La figura del controlador es la encargada de coordinar y dar sentido a todos los elementos que, en conjunto, conformen un objeto de la escena.

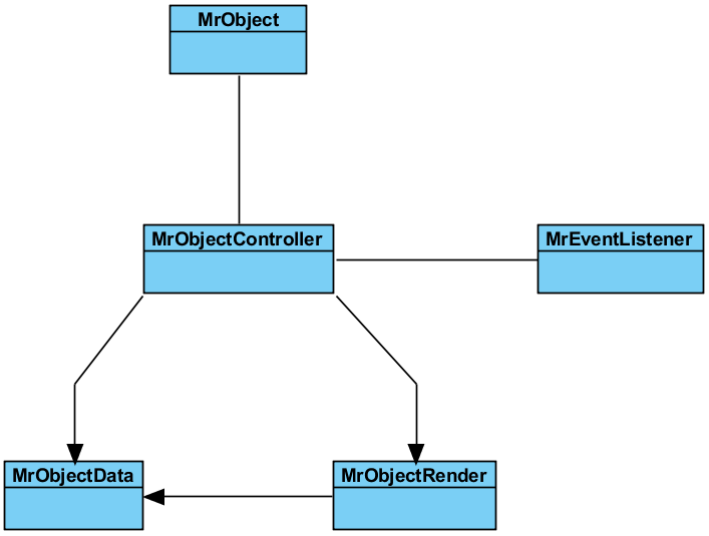


FIGURA 2.10: Elementos destacables de MrObjectController

2.2.5. MrObjectController y sus Componentes

2.2.6. MrModelController y sus Componentes

2.2.6.1. Mallas 3D

TODO: Comentar campos importantes como los VBO e IBO, y como relaciona cada campo de los buffers con los atributos del shader program (este .[en]lace.[en]tre datos viene especificado en el fichero MRR) TODO:Agregar como se pasan los attributes

2.2.6.2. Shaders

TODO: Shaders, Attributes, Uniforms y Generadores de Uniforms (estos últimos tienen especial interés ya que gracias a ellos en un cierto objeto dado podemos generar un uniform y relacionarlo con un uniform del shader, explicar ese enlace en profundidad en la sección de renderizadores)

2.2.6.3. Materiales y Texturas

TODO:

2.2.6.4. Animación

TODO:Explicar cosas como a partir de los KeyFrames importados desde el fichero se interpola la pose, así como la estructura de un esqueleto, los huesos,...

2.2.7. Paquete de Herramientas

Entre las utilidades más relevantes para el funcionamiento de MrRobotto 3D Engine cabe destacar el papel de los cargadores de datos, encargados de transformar los ficheros *".mrr"* a la estructura interna del motor, así como las distintas clases encargadas de representar estructuras de datos específicamente implementadas para acomodar los datos usados.

2.2.7.1. Cargadores de Datos

TODO: Explicar como funciona la carga de datos y cómo se realiza una carga paralela de por ejemplo los JSON y las texturas

2.2.8. Paquete de Estructuras de Datos

Para la implementación de partes esenciales de la arquitectura, como por ejemplo, la construcción de la propia escena, se requerían de ciertas estructuras de datos específicas para la aplicación.
Este es el caso de **MrTreeMap**, clase

Las siguientes estructuras de datos aquí presentadas han sido pensadas tanto como para representar los datos requeridos así como una gran rapidez a la hora de acceder a ellos y realizar búsquedas.
Con ese fin se ha desarrollado una herramienta auxiliar denominada **MrMapFunction**. Esta sencilla clase permite realizar un mapeo directo entre los objetos contenidos en las estructuras y las llaves asociadas a ellos.

$$\text{Object} \mapsto \text{Key}$$

TODO: Comentar las estructuras de datos implementadas, la más relevante, el árbol que luego usas como base clase para implementar el SceneTree

2.2.9. Paquete Matemático

2.2.9.1. MrLinearAlgebraObject

MrLinearAlgebraObject es la interfaz principal de todos los objetos que necesiten por una parte, almacenar datos numéricos, como podrían ser ls componentes de un vector, una matriz,... y por otra usarse como interfaz a la hora de transmitir datos desde la aplicación a la GPU en forma de *uniforms*. TODO: Clase base de los elementos matemáticos, y no matemáticos, pues será la clase fundamental para poder pasarlos como uniforms TODO: Comentar la alta optimización de las operaciones

2.2.9.2. Matrices

2.2.9.3. Cuaterniones

2.2.9.4. Vectores

2.2.9.5. MrTransforms

TODO: Clase base de las transformaciones geométricas, comentar cómo se combinan las clases anteriores para ofrecer operaciones tales como rotar alrededor de un punto

2.3. Conclusiones y Posibles Mejoras

TODO: Compartir modelos, mejorar el formato mrr, mala elección de org.json

Capítulo 3

MrRobotto Studio

3.1. La Aplicación MrRobotto Studio

3.2. Blender Scripting

TODO: Explicar como funciona el script de blender y como se exportan algunas secciones importantes como una Mesh, las animaciones,...

3.3. La Aplicación Django

TODO: Explicar la aplicación servidor, la pequeña base de datos SQLite usada, servicios implementados

3.4. Cliente Web

TODO: Comentar las distintas páginas y acciones a realizar en cada una

3.5. Cliente Android

TODO: Comentar como utilizar cada acción, como hace para actualizarse de forma automática, servicios a los que se conecta

Apéndice A

Especificación del formato MRR

El formato MRR es un formato de fichero para representación de escenas 3D usado por el MrRobotto 3D Engine. La extensión elegida para este tipo de ficheros es la de *'.mrr'*. Este formato utiliza internamente una mezcla entre formato JSON para los datos relacionados con la escena en sí y formato binario para los datos relacionados con las texturas.

A.1. Conceptos

A.2. Estructura del formato

El formato MRR posee de cinco secciones a distinguir tal y como se aprecia en la siguiente tabla:

Sección	**Cabecera**
Valor	`MRROBOTTOFILE`
`MRROBOTTOFILE`	Tag
Sección	**Sección de cabecera de JSON**
Valor	`JSON N`
`JSON`	Tag
`N`	Tamaño de la sección JSON representado como un número entero de 4 bytes en codificación big endian.
Sección	**Sección de datos JSON**
Valor	`json`
`json`	Cadena de carácteres en codificación ASCII de longitud N y con formato JSON
Sección	**Cabecera de la sección de texturas (Opcional)**
Valor	`TEXT M`
`TEXT`	Tag
`M`	Tanaño de la sección de texturas representado como un número entero de 4 bytes en codificación big endian.
Sección	**Sección de texturas (Opcional)**
Valor	`texturas`
`texturas`	Datos binarios de las texturas almacenadas y de longitud M.
Sección	**Sección de nombre de textura**
Valor	`NAME M name N`
`NAME`	Tag
`M`	Size of name represented as a big endian 4-bytes integer
`name`	String representing the name of texture file
`N`	Size of texture represented as a big endian 4-bytes integer
Sección	**Texture data Sección (Optional)**
Valor	`texture`
`texture`	Texture data of length M

A.2.1. Cabecera

En la cabecera del fichero se encuentra el *magic number* que se ha asignado al formato. Se trata de un tag almacenado como un cadena de *char* en codificación ASCII

`MRROBOTTOFILE`

A.2.2. Sección de cabecera JSON

En la cabecera de la sección se encuentra el tag que indica el inicio de la sección, `JSON` seguido por el número de bytes ocupado por la sección

A.2.3. Sección de datos JSON

A continuación se expondrán todos los campos usados en la sección JSON de forma jerárquica.

Sección	**Raíz del documento**
`Hierarchy`	Jerarquía de la escena
`SceneObjects`	Lista de objetos de la escena

Sección	`Hierarchy`
`Children`	Hijos del objeto raíz.
`Name`	Nombre del objeto raíz.

Sección	`Children`
`Children`	Hijos del objeto actual.
`Name`	Nombre del objeto raíz.

A.2.3.1. Ejemplo de sección JSON

```
{
    "Hierarchy":{
        "Children":[
            {
                "Children":[],
                "Name":"Camera"
            },
            {
                "Children":[],
                "Name":"Lamp"
            },
            {
                "Children":[],
                "Name":"Cube"
            }
        ],
        "Name":"Scene"
    },
```

```
"SceneObjects":[
    {
        "AmbientLightColor":[0.5,0.5,0.5,1.0],
        "ClearColor":[0.5,0.5,0.5,1.0],
        "Name":"Scene",
        "ShaderProgram":null,
        "Transform":{
            "Location":[0.0,0.0,0.0],
            "Rotation":[1.0,0.0,0.0,0.0],
            "Scale":[1.0,1.0,1.0]
        },
        "Type":"Scene",
        "UniformKeys":[
            {
                "Count":1,
                "Generator":"Generator_Model_View_Projection_Matrix",
                "Index":0,
                "Level":1,
                "Uniform":"Model_View_Projection_Matrix"
            },
            ...
        ]
    },
    {
        "Lens":{
            "AspectRatio":0.857556,
            "ClipEnd":100.0,
            "ClipStart":0.1,
            "FOV":35.0,
            "Type":"Perspective"
        },
        "Name":"Camera",
        "ShaderProgram":null,
        "Transform":{
            "Location":[0.0,-10.0,0.0],
            "Rotation":[1.0,0.0,0.0,0.0],
            "Scale":[1.0,1.0,1.0]
        },
        "Type":"Camera",
```

```
            "UniformKeys":[
                {
                    "Count":1,
                    "Generator":"Generator_View_Matrix",
                    "Index":0,
                    "Level":0,
                    "Uniform":"View_Matrix"
                },
                ...
            ]
        },
        {
            "Color":[1.0,1.0,1.0],
            "LightType":"Point",
            "LinearAttenuation":0.0,
            "Name":"Lamp",
            "QuadraticAttenuation":1.0,
            "ShaderProgram":null,
            "Transform":{
                "Location":[4.076245,1.005454,5.903862],
                "Rotation":[0.570948,0.169076,0.272171,0.75588],
                "Scale":[1.0,1.0,1.0]
            },
            "Type":"Light",
            "UniformKeys":[
                {
                    "Count":1,
                    "Generator":"Generator_Light_Position",
                    "Index":0,
                    "Level":0,
                    "Uniform":"Light_Position"
                },
                ...
            ]
        },
        {
            "Materials":[
                {
                    "Ambient":{
```

```
                "Color":[0.0,0.0,0.0,1.0],
                "Intensity":1.0
            },
            "Diffuse":{
                "Color":[0.8,0.0,0.004184,1.0],
                "Intensity":0.8
            },
            "Name":"Material",
            "Specular":{
                "Color":[1.0,1.0,1.0,1.0],
                "Intensity":0.5
            },
            "Texture":{
                "Index":1,
                "MagFilter":"Linear",
                "MinFilter":"Linear",
                "Name":"fire.png"
            }
        }
    ],
    "Mesh":{
        "AttributeKeys":[
            {
                "Attribute":"Vertices",
                "DataType":"float",
                "Pointer":0,
                "Size":3,
                "Stride":16
            },
            {
                "Attribute":"Normals",
                "DataType":"float",
                "Pointer":3,
                "Size":3,
                "Stride":16
            },
            ...
        ],
        "Count":888,
```

```
            "DrawType":"Triangles",
            "IndexData":[0,1,2,138,142,143,156,4,...],
            "Name":"Cube",
            "VertexData":[0.741746,0.741746,3.736173,0.57735,...]
        },
        "Name":"Cube",
        "ShaderProgram":{
            "Attributes":[
                {
                    "Attribute":"Normals",
                    "DataType":"vec3",
                    "Index":1,
                    "Name":"aNormal"
                },
                {
                    "Attribute":"Vertices",
                    "DataType":"vec3",
                    "Index":0,
                    "Name":"aVertex"
                },
                ...
            ],
            "FragmentShaderSource":"precision highp float;\nuniform mat4 fsmrMode
            "Name":"ShaderProgram_0",
            "Uniforms":[
                {
                    "Count":1,
                    "DataType":"mat4",
                    "Name":"mrMvpMatrix",
                    "Uniform":"Model_View_Projection_Matrix"
                },
                {
                    "Count":4,
                    "DataType":"mat4",
                    "Name":"mrBones",
                    "Uniform":"Bone_Matrix"
                },
                ...
            ],
```

```
            "VertexShaderSource":"uniform float mrDiffuseInt;\nuniform vec4 mrAmb
        },
        "Skeleton":{
            "Actions":[
                {
                    "FPS":24,
                    "KeyFrames":[
                        {
                            "Bones":[
                                {
                                    "Location":[1.0,0.0,1.0],
                                    "Name":"bottom",
                                    "Rotation":[0.757091,-0.0,0.65331,0.0],
                                    "Scale":[1.0,1.0,1.0]
                                },
                                {
                                    "Location":[0.989234,0.0,1.149418],
                                    "Name":"mid",
                                    "Rotation":[0.706029,0.0,0.708183,0.0],
                                    "Scale":[1.0,1.0,1.0]
                                },
                                ...
                            ],
                            "Number":1
                        },
                        {
                            "Bones":[
                                ...
                            ],
                            "Number":6
                        },
                      ...
                    ],
                    "Name":"ArmatureAction",
                    "Type":"Skeletal"
                }
            ],
            "BoneOrder":["bottom","mid","top","right"],
            "Pose":[
```

```
                    {
                        "Location":[1.0,0.0,1.0],
                        "Name":"bottom",
                        "Rotation":[0.757091,-0.0,0.65331,0.0],
                        "Scale":[1.0,1.0,1.0]
                    },
...
                ],
                "Root":{
                    "Children":[
                        {
                            "Children":[
                                {
                                    "Children":[],
                                    "Name":"top"
                                }
                            ],
                            "Name":"mid"
                        },
                        {
                            "Children":[],
                            "Name":"right"
                        }
                    ],
                    "Name":"bottom"
                }
            },
            "Transform":{
                "Location":[-1.311603,10.0,-1.501951],
                "Rotation":[1.0,0.0,0.0,0.0],
                "Scale":[0.717982,0.717982,0.717982]
            },
            "Type":"Model",
            "UniformKeys":[
                {
                    "Count":1,
                    "Generator":"Generator_Model_Matrix",
                    "Index":0,
                    "Level":0,
```

```
                    "Uniform":"Model_Matrix"
                },
               ...
            ]
        }
    ]
}
```

Bibliografía

[1] Apple. Best practices for working with vertex data, Julio 2014. URL https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/TechniquesforWorkingwithVertexData/TechniquesforWorkingwithVertexData.html.

[2] Kevin Brothaler. An introducttion to index buffer objecst (ibos), Mayo 2012. URL http://www.learnopengles.com/android-lesson-eight-an-introduction-to-index-buffer-objects-ibos/.