

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 documento 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

UNIVERSITY OF BARCELONA

# *Abstract*

Faculty of Mathematics

Double Degree in Computer Engineering and Mathematics

## **MrRobotto 3D Engine**

by Aarón NEGRÍN SANTAMARÍA

MrRobotto 3D Engine is a 3D game engine for Android platforms. It aims to be an usable, high lightweight and powerful tool.

The original idea of this project comes from the absence of a free software 3D engine easy to be integrated into Android projects

Moreover MrRobotto 3D Engine has been optimized from scratch, taking into account Android's hardware limitations since his beginning.

Additionally, complementary tools have been developed besides MrRobotto 3D Engine. The called MrRobotto Studio provides a functional environment designed to speed up and complete the software development experience.

In conclusion, both MrRobotto 3D Engine and MrRobotto Studio gives a truly functional tool in order to support the creation of 3D graphics applications in Android platforms.

Keywords: MrRobotto, Android, 3D graphics, videogame development

UNIVERSIDAD DE BARCELONA

# *Abstract*

Facultad de Matemáticas

Grado en Ingeniería Informática y Matemáticas

## **MrRobotto 3D Engine**

por Aarón NEGRÍN SANTAMARÍA

MrRobotto 3D Engine es un motor de juegos 3D para la plataforma Android. Su motivación principal es la de ser de uso sencillo, extremadamente ligero pero a su vez potente.

Este proyecto surge ante la carencia de un motor de juegos 3D de integración sencilla dentro de un proyecto de la plataforma Android y que cuente con una licencia libre.

Además MrRobotto 3D Engine ha sido creado teniendo presentes las múltiples limitaciones técnicas de los dispositivos sobre los que se ejecutará obteniendo así una plataforma altamente optimizada.

Por otra parte se incluyen herramientas adicionales, el denominado MrRobotto Studio, que hacen de MrRobotto 3D Engine una plataforma totalmente funcional al proporcionarse un entorno capaz de agilizar y completar la experiencia de desarrollo.

En conclusión, tanto MrRobotto 3D Engine como MrRobotto Studio conforman una verdadera herramienta funcional para la creación de aplicaciones que hagan uso de gráficos tridimensionales.

Palabras claves: MrRobotto, Android, gráficos 3D, desarrollo videojuegos

# Índice general

<b>Declaración de Autoría</b>	<b>I</b>
<b>Abstract(Inglés)</b>	<b>III</b>
<b>Abstract(Castellano)</b>	<b>IV</b>
<b>Contenidos</b>	<b>V</b>
<b>Lista de Figuras</b>	<b>VIII</b>
<b>Lista de Tablas</b>	<b>IX</b>
<b>1. Introducción</b>	<b>1</b>
1.1. Objetivos y Motivación . . . . .	1
1.2. Tecnologías utilizadas . . . . .	2
1.2.1. MrRobotto 3D Engine . . . . .	2
1.2.2. MrRobotto Studio . . . . .	2
<b>2. MrRobotto 3D Engine</b>	<b>4</b>
2.1. Diseño de MrRobotto 3D Engine . . . . .	4
2.1.1. Formato .MRR . . . . .	4
2.1.2. El SceneTree . . . . .	6
2.1.3. Elementos de una Escena . . . . .	6
2.1.3.1. Object . . . . .	7
2.1.3.2. Model . . . . .	7
2.1.3.3. Cámaras y Luces . . . . .	8
2.1.3.4. Scene . . . . .	8
2.1.4. Los Conceptos de UniformKey y UniformGenerator . . . . .	9
2.1.5. La Necesidad del Rendering Context . . . . .	11
2.1.6. Eventos . . . . .	12
2.1.7. Algoritmo de Renderizado . . . . .	12
2.2. Arquitectura de MrRobotto 3D Engine . . . . .	13
2.2.1. Interfaz de Uso . . . . .	13
2.2.1.1. Integración en una Aplicación Android . . . . .	15
2.2.2. Objetos de la Escena . . . . .	16

2.2.2.1.	MrSceneTree . . . . .	16
2.2.2.2.	MrObject . . . . .	16
2.2.3.	Eventos . . . . .	17
2.2.4.	Estructura de los Objetos . . . . .	18
2.2.4.1.	Contenedores de Datos . . . . .	19
2.2.4.2.	Renderizadores . . . . .	19
2.2.4.3.	Controladores . . . . .	20
2.2.5.	Componentes . . . . .	20
2.2.6.	MrModelController y sus Componentes . . . . .	21
2.2.6.1.	Mallas 3D . . . . .	21
2.2.6.2.	Materiales . . . . .	22
2.2.6.3.	Texturas . . . . .	23
2.2.6.4.	Animación . . . . .	23
2.2.6.5.	Shader Program . . . . .	24
2.2.7.	Paquete de Herramientas . . . . .	24
2.2.7.1.	Cargadores de Datos . . . . .	25
2.2.8.	Paquete de Estructuras de Datos . . . . .	25
2.2.9.	Paquete Matemático . . . . .	25
<b>3.</b>	<b>MrRobotto Studio</b>	<b>28</b>
3.1.	La Aplicación MrRobotto Studio . . . . .	28
3.2.	Blender Scripting . . . . .	28
3.2.1.	Datos . . . . .	29
3.2.2.	Exportadores . . . . .	29
3.3.	La Aplicación Django . . . . .	29
3.4.	Cliente Web . . . . .	30
3.5.	Cliente Android . . . . .	30
<b>4.</b>	<b>Resultados</b>	<b>31</b>
4.1.	Resultados de Rendimiento . . . . .	31
4.1.1.	Tiempos de carga . . . . .	31
4.1.2.	Renderizado de Múltiples Objetos Simples . . . . .	31
4.1.3.	Renderizado de Múltiples Objetos Animados . . . . .	31
4.1.4.	Renderizado de Múltiples Objetos y Múltiples Texturas . . . . .	32
<b>5.</b>	<b>Conclusiones y Posibles Mejoras</b>	<b>33</b>
5.1.	Conclusiones . . . . .	33
5.2.	Mejoras Futuras . . . . .	33
<b>A.</b>	<b>Especificación del formato MRR</b>	<b>34</b>
A.1.	Conceptos . . . . .	34
A.2.	Estructura del formato . . . . .	34
A.2.1.	Cabecera . . . . .	35
A.2.2.	Sección de cabecera JSON . . . . .	36
A.2.3.	Sección de datos JSON . . . . .	36
A.2.3.1.	Ejemplo de sección JSON . . . . .	37

**Bibliografía**

**45**



# Índice de figuras

2.1. Esquema principal de MrRobotto 3D Engine . . . . .	5
2.2. Funcionamiento de UniformKeys y UniformsGenerators . . . . .	10
2.3. Ejemplo del funcionamiento del Rendering Context . . . . .	12
2.4. Inicialización de Recursos para el Renderizado . . . . .	13
2.5. Renderizado de un Frame . . . . .	14
2.6. Interfaz del núcleo . . . . .	14
2.7. Jerarquía de MrObject . . . . .	16
2.8. Métodos de MrObject . . . . .	17
2.9. Estructura interna de los objetos . . . . .	18
2.10. Elementos destacables de MrObjectController . . . . .	20
2.11. Estructura de un VBO . . . . .	22
3.1. Datos almacenados en MrRobotto Studio . . . . .	29
4.1. Renderizado de objetos simples . . . . .	32

# Índice de cuadros

4.1. <a href="#">My caption</a> . . . . .	31
---	----

*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. Objetivos y Motivación

TODO: Insertar el abstract en cierto sentido

## 1.2. Tecnologías utilizadas

### 1.2.1. MrRobotto 3D Engine

Las distintas tecnologías utilizadas para el desarrollo de MrRobotto 3D Engine consti-  
tuyen 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 repositorio 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.

### 1.2.2. MrRobotto Studio

Para la parte de la aplicación de MrRobotto Studio se ha decidido usar las siguientes tecnologías:

- Django para la parte del servidor
- SQLite como base de datos
- Android SDK para la parte del cliente

- JQuery para las llamadas AJAX
- BootStrap para el estilo de la página

La decisión de usar Django como *backend* para la aplicación vino fundamentada por la necesidad de generar código de calidad, de rápido desarrollo y en un lenguaje del mayor alto nivel posible

Se consideró Node.js por su capacidad por la necesidad de ofrecer la tecnología de websockets, sin embargo, JavaScript llevaba a una velocidad de desarrollo inferior a la deseada.

Era importante conseguir que tanto el dispositivo Android como la interfaz web estuviesen notificados en cada momento de los cambios ocurridos en la otra parte, por ello, para la parte web era necesario usar tecnologías como AJAX.

Por último, pero no menos importante, está el tema del estilo, no era viable invertir tiempo modificando estilos de la interfaz web, es por eso que se optó por tecnologías ampliamente utilizadas en el sector web como BootStrap.

## 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

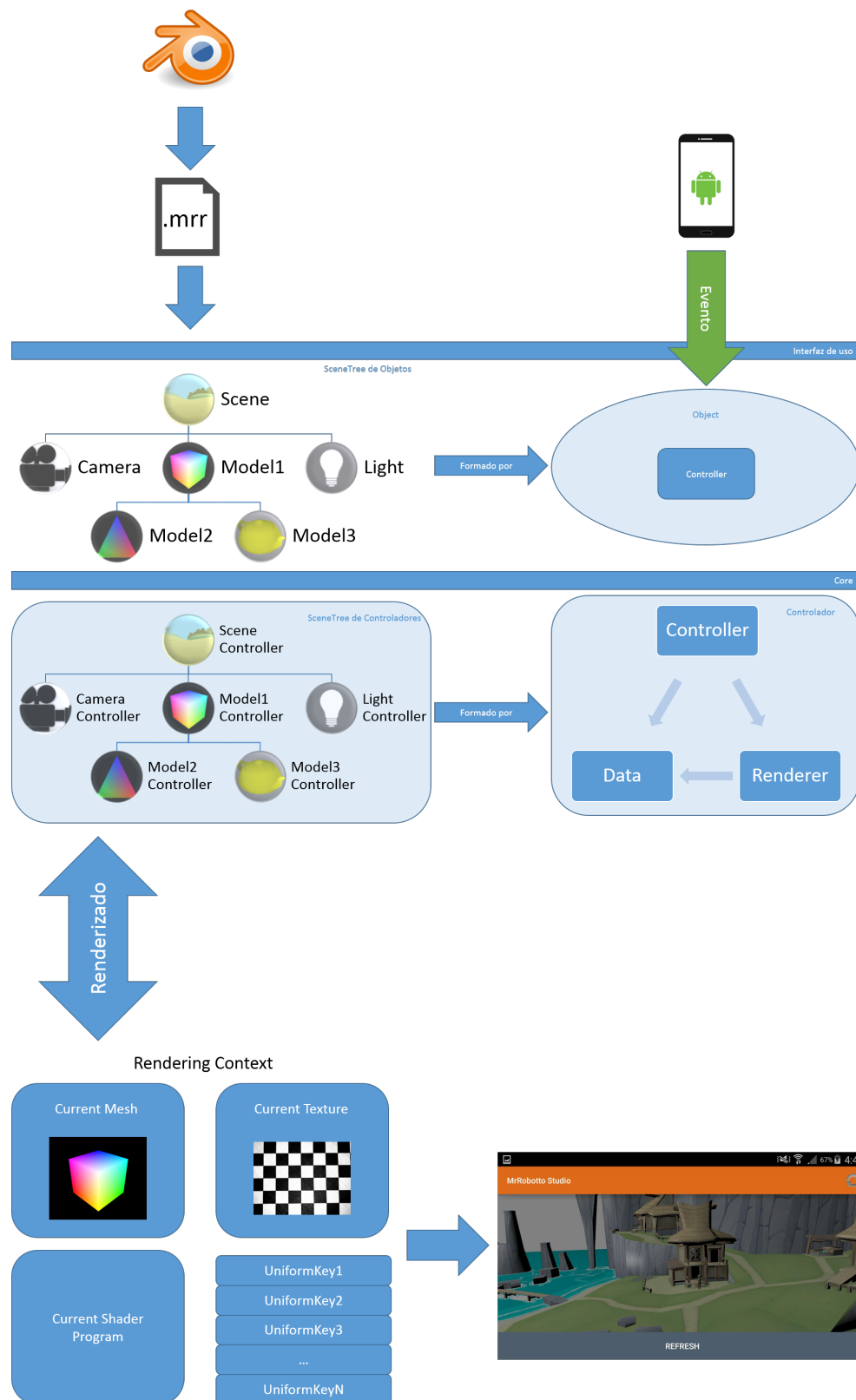


FIGURA 2.1: Esquema principal de MrRobotto 3D Engine



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 se 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 dibujo 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 los elementos 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 multiples 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. Las luces soportadas actualmente por MrRobotto 3D Engine se reducen únicamente a luces puntuales

### 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 la GPU.

Antes de comenzar se introducirán algunos términos referidos a la programación gráfica, por una parte se ha de conocer que es un Shader Program. Un Shader Program es un programa orientado a ser ejecutado sobre la GPU y que se encarga de realizar el sombreado sobre los distintos vértices de los objetos renderizados.

Dichos Shader Programs pueden recibir entradas de datos, estos se dividen en Attributes, que son datos dependientes de cada vértice y Uniforms, que son datos constantes dentro de cada ejecución del Shader Program, siendo así independientes de los vértices.

Ahora bien, una vez introducidos los conceptos, con el fin de enviar los Uniforms desde CPU a GPU de forma sencilla se han 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 a 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 poseerá 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.

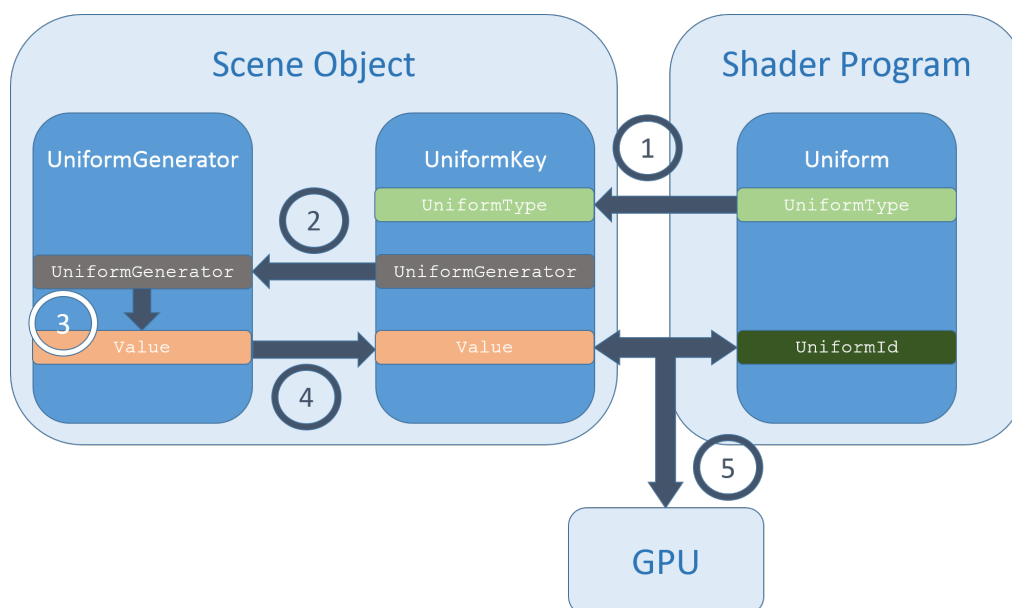


FIGURA 2.2: Funcionamiento de UniformKeys y UniformsGeneratos.

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.

Por último mencionar que los UniformGenerators pueden llegar a presentar un grave problema si no se controla. En el caso de que por ejemplo se necesitase calcular una cierta multiplicación de dos matrices  $A$  y  $B$  y que tanto  $A$  como  $B$  deben generarse también en un UniformGenerator. Ante esta situación podría pasar que  $AB$  fuese calculada antes de generar las matrices  $A$  y  $B$  para ese frame. Es por eso que se hace necesario añadir ciertas restricciones a los UniformGenerators, estos son los denominados niveles.

El nivel del UniformGenerator indica el orden en el que este debe ser calculado y va desde 0 en adelante.

En el ejemplo anterior  $A$  y  $B$  podrían tener niveles 0 ambas mientras que  $AB$  podría estar en el nivel 1, de esta forma se asegura que tanto  $A$  como  $B$  son generadas previamente a  $AB$

### 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.

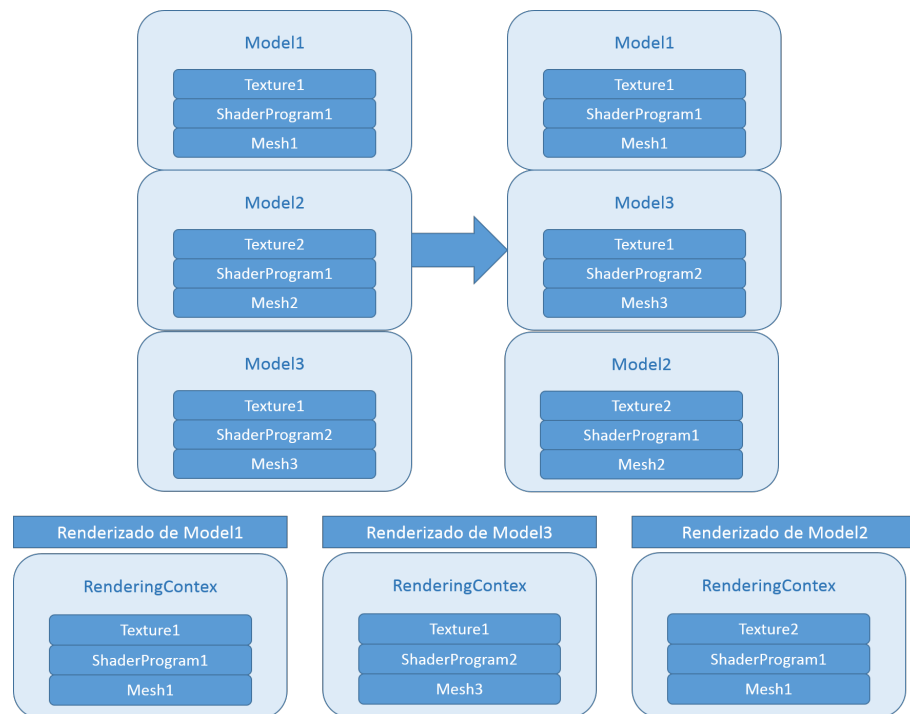


FIGURA 2.3: Ejemplo del funcionamiento del Rendering Context

### 2.1.6. Eventos

TODO:

### 2.1.7. Algoritmo de Renderizado

TODO: Modificar la explicación del renderingsorter 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

En el caso del renderizado de un frame lo primero que se realiza es actualizar el objeto según los eventos recibidos.

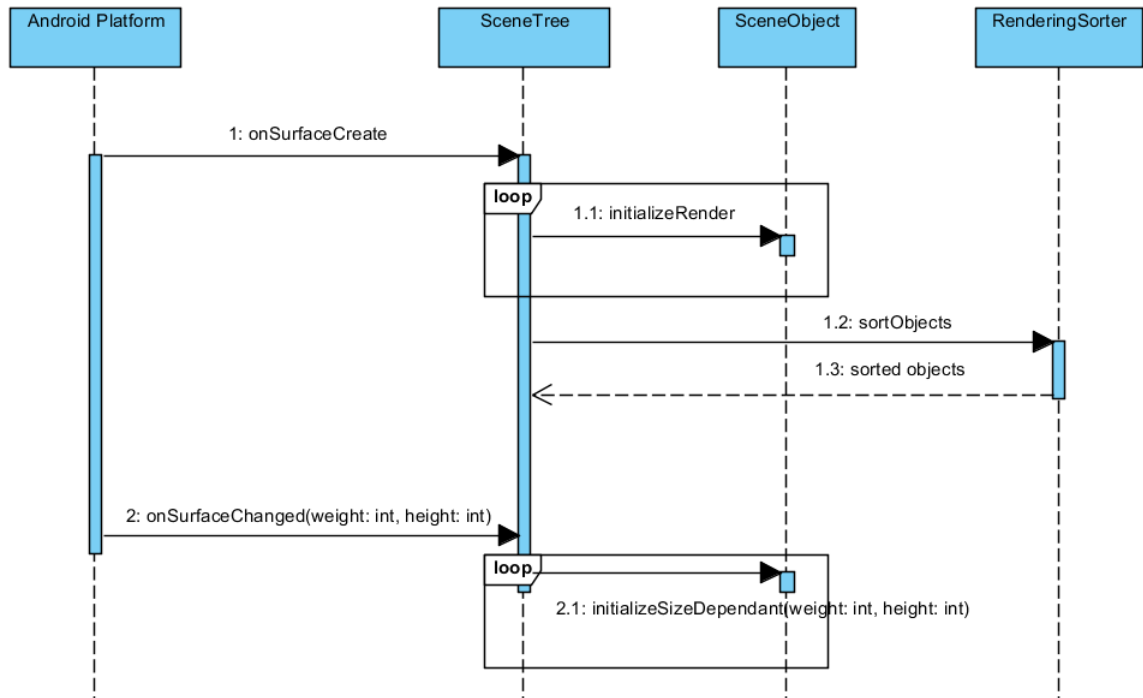


FIGURA 2.4: Inicialización de Recursos para el Renderizado

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

## 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 ciclo de vida de las aplicaciones Android comunes, permitiendo mantener un paradigma



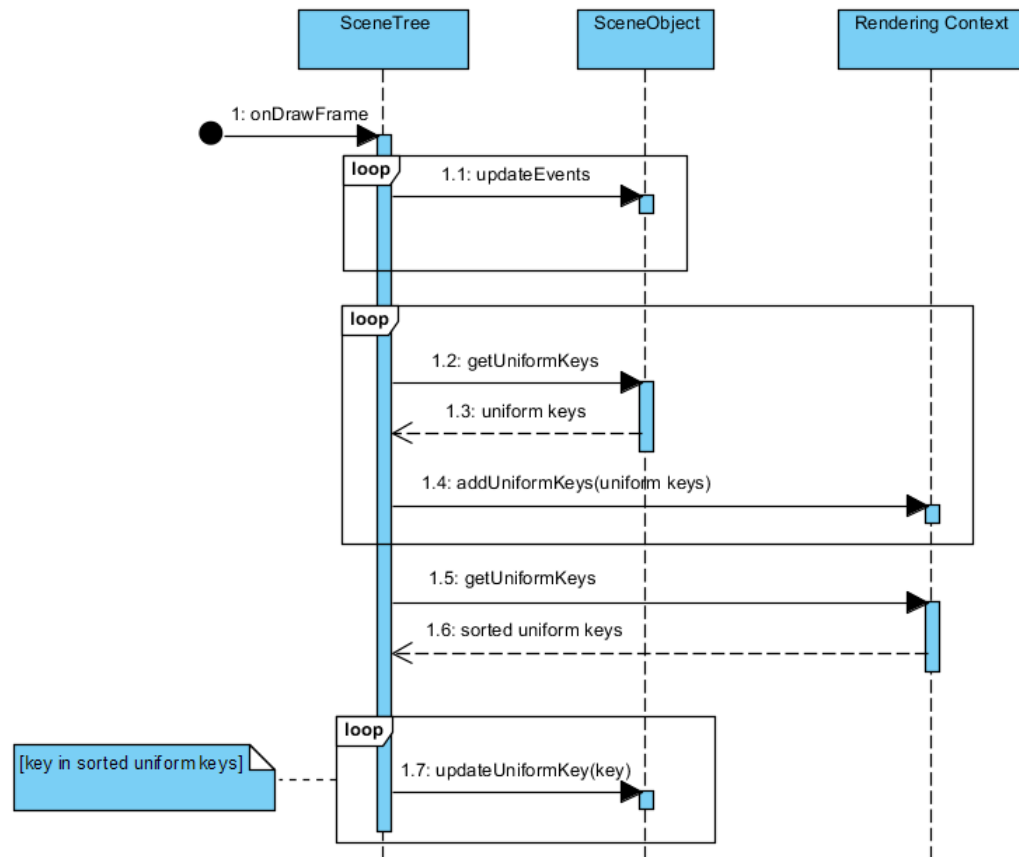


FIGURA 2.5: Renderizado de un Frame

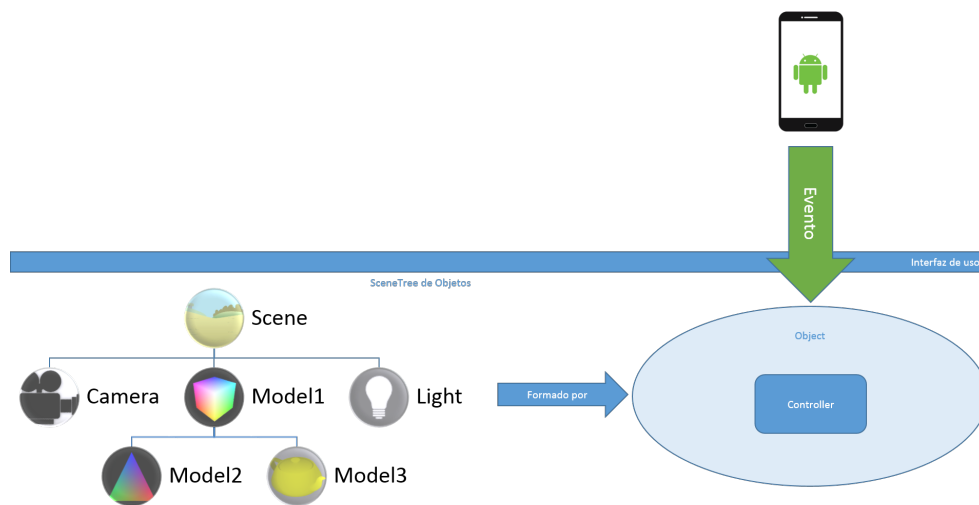


FIGURA 2.6: Interfaz del núcleo

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

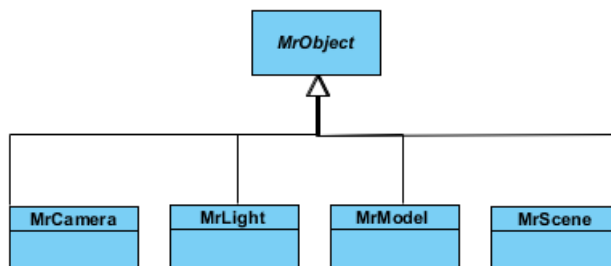


FIGURA 2.7: Jerarquía de MrObject

realidad actúa como un envoltorio 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.

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

```

MrObject
#MrObject(controller : MrObjectController)
+getController() : MrObjectController
+initializeUniforms(uniformGenerators : Map<String, MrUniformGenerator>) : void
+getSceneObjectType() : MrSceneObjectType
+isInitialized() : boolean
+getName() : String
+getUniformGenerators() : Map<String, MrUniformGenerator>
+getShaderProgram() : MrShaderProgram
+getUniformKeys() : Map<String, MrUniformKey>

```

(A) Métodos Genéricos

```

MrObject
#queueEvent(runnable : Runnable) : void
+getEventsListener() : MrEventsListener
+setEventsListener(eventsListener : MrEventsListener) : void
+isEventRegistered(evName : String) : boolean
+getRegisteredEvents() : Set<String>
+registerEvent(eventName : String) : void
+unregisterEvent(eventName : String) : void

```

(B) Métodos de Gestión de Eventos

```

MrObject
+getRobottoEngine() : MrRobottoEngine
+getTree() : MrSceneTree
+setTree(tree : MrSceneTree) : void
+addChild(data : MrObject) : boolean
+removeChild(data : MrObject) : boolean
+getByType(type : MrSceneObjectType) : List<MrObject>
+getRoot() : MrObject
+findChild(key : String) : MrObject
+isChild(data : MrObject) : boolean
+isChild(key : String) : boolean
+getParent() : MrObject
+getChildren() : List<MrObject>
+parentTraversal() : Iterator<MrObject>
+breadthTraversal() : Iterator<MrObject>
+depthTraversal() : Iterator<MrObject>
+parentKeyChildValueTraversal() : Iterator<Entry<String, MrObject>>
+getRight() : MrVector3f

```

(C) Métodos de Gestión de Jerarquía

```

MrObject
+setRobottoEngine(robotto : MrRobottoEngine) : void
+getTransform() : MrTransform
+setTransform(transform : MrTransform) : void
+getRotation() : MrQuaternion
+setRotation(rotation : MrQuaternion) : void
+rotate(angle : float, axis : MrVector3f) : void
+translate(x : float, y : float, z : float) : void
+scale(s : float) : void
+scale(s : MrVector3f) : void
+setLookAt(look : MrVector3f, up : MrVector3f) : void
+scale(sx : float, sy : float, sz : float) : void
+setScale(sx : float, sy : float, sz : float) : void
+translate(v : MrVector3f) : void
+getForward() : MrVector3f
+getScale() : MrVector3f
+setScale(scale : MrVector3f) : void
+setLocation(x : float, y : float, z : float) : void
+setRotation(angle : float, axis : MrVector3f) : void
+getUp() : MrVector3f
+setLookAt(look : MrVector3f) : void
+setRotation(angle : float, x : float, y : float, z : float) : void
+getLocation() : MrVector3f
+setLocation(location : MrVector3f) : void
+rotateAround(angle : float, point : MrVector3f, axis : MrVector3f) : void
+rotate(q : MrQuaternion) : void
+rotateAround(angle : float, point : MrVector3f, axis : MrVector3f, through : MrVector3f) : void
+rotate(angle : float, x : float, y : float, z : float) : void

```

(D) Métodos de Transformaciones Geométricas

FIGURA 2.8: Métodos de MrObject agrupados por funcionalidad.

El resto de tipos de objetos que pueden estar presentes en la escena, a saber: [MrModel](#), [MrCamera](#), [MrLight](#) y [MrScene](#), comparten gran parte de la interfaz de [MrObject](#), sin embargo, representan objetos de la escena los cuales tienen atributos únicos de su tipo.

### 2.2.3. Eventos

La gestión de los eventos a nivel interno se realiza en dos pasos.

El primero de ellos reside sobre la clase [MrEventDispatcher](#) pues es la encargada de recibir los eventos desde su fuente y tras ello enviarlo para su procesamiento dentro del bucle principal de renderización.

Dichos eventos deben establecer un identificador único para distinguir el tipo de evento enviado.

Además, a veces resulta necesario el envío de datos captados del evento se ha diseñado un elemento denominado [MrBundle](#), en este elemento un usuario que reimplementase el comportamiento del [MrEventDispatcher](#) asociado al motor en ese momento debería

hacer uso de esta herramienta para el envío de datos.

Una vez dichos datos llegan al hilo de ejecución del renderizado se procesan en orden de llegada para ser procesados dentro de los llamados [MrEventListener](#), estos elementos están asociados a cada uno de los objetos de la escena.

Ahora bien, no todos los objetos procesan todos los eventos recibidos, los [MrEventListener](#) registran una serie de eventos a procesar, de esa forma, si un [MrEventListener](#) tiene registrado el identificador de un cierto evento significa que será procesado por él, e ignorándolo en caso de que no esté registrado. En caso de que se procese el evento, el EventListener recibe como argumento la referencia al [MrBundle](#) asociado al evento recibido.

#### 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.

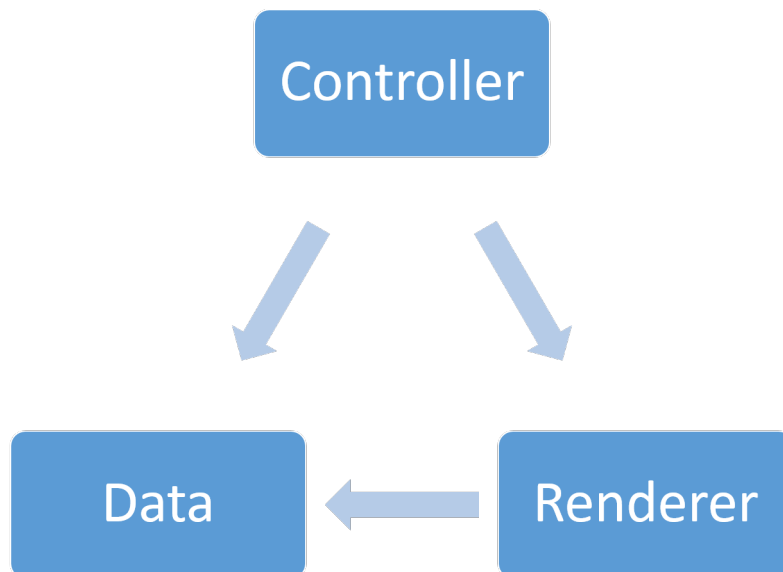


FIGURA 2.9: Estructura interna de los objetos.

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.

En los contenedores de datos también son almacenados tanto los UniformKeys de los objetos como los UniformGenerators que posean, y aunque su uso se realice mediante el controlador, este actúa como una interfaz.

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.robottto.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 dependen 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.

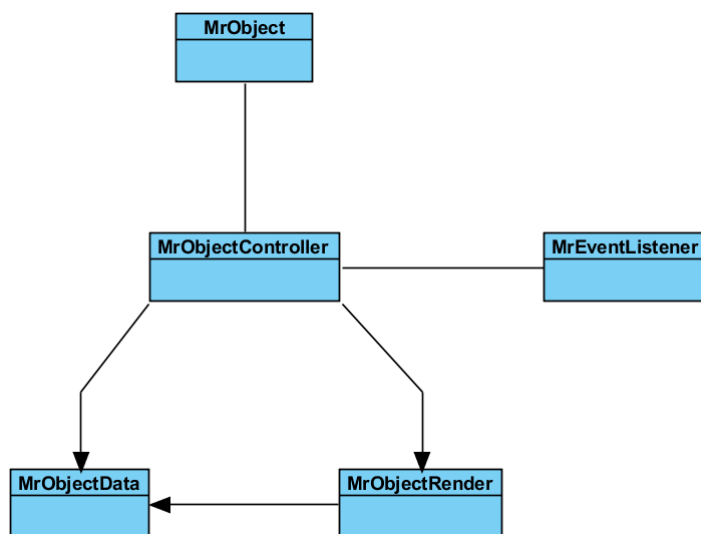


FIGURA 2.10: Elementos destacables de MrObjectController

Como se puede apreciar en el esquema se comunican tanto con la capa de datos como la del renderizado, además proporciona un contexto válido a los [MrObject](#) y a los [MrEventListener](#).

Todos los controladores deben heredar de la clase [MrObjectController](#) y se encuentran dentro del paquete [mr.robotto.engine.core.controller](#)

### 2.2.5. Componentes

Los componentes son elementos especiales dentro de MrRobotto 3D Engine encargados de almacenar datos y comunicarse con la GPU.

Aunque los componentes puedan resultar similares en funcionalidad a los objetos de la escena la diferencia radica en que los componentes tienen un uso casi exclusivo a nivel interno del motor.

Los componentes además disfrutan de una estructura interna que sigue el patrón MVC, aunque en su caso, y por motivos de simplificación, la capa de datos y renderizado se

han implementado como clases internas y siendo el controlador una interfaz a estas dos.

Así pues, la finalidad de estos elementos es la de conformar los atributos de la capa de datos de los objetos de la escena y simplificar el código del renderizado.

Todos los componentes deben heredar desde la clase base que se ha definido para este fin: [MrComponent](#)

### 2.2.6. MrModelController y sus Componentes

Por su relevancia e interés se ha decidido hacer una mención especial a los modelos, que como componentes visibles que son, poseen una cierta relevancia que debe ser tratada en detalle.

#### 2.2.6.1. Mallas 3D

Las mallas 3D, o como se han denominado en MrRobotto 3D Engine, [MrMesh](#), son componententes pertenecientes a los modelos.

Son los encargados de enviar los datos relativos a los vértices a la GPU.

Para representar una malla harán falta dos elementos fundamentales:

- Los Vertex Buffer Objects (VBO), que son listas de datos numéricos encargados de almacenar la información de cada uno de los vértices, como las coordenadas donde se encuentra, las normales, el índice del material que usan,...
- Los Index Buffer Objects (IBO), que son listas de índices que indican el orden en el que los vértices deben ser leídos.

Estos buffers en la implementación son los denominados [MrBuffer](#).

Ahora bien, los VBOs representan los datos de una forma no estructurada y se requiere poder diferenciar los distintos datos contenidos en ellos, con este fin existen los [MrBufferKey](#).

Los [MrBufferKey](#) son los encargados de determinar en qué índice del VBO empieza y acaba un dato específico de un vértice.

Dentro de sus atributos importantes están:

- **Pointer:** Índice del buffer de la primera aparición de un cierto dato.



- **Size:** Tamaño del dato
- **Stride:** Número de elementos el inicio de un cierto tipo de dato y su siguiente aparición.

Por ejemplo:

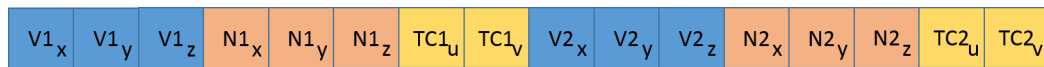


FIGURA 2.11: Estructura de un VBO.

V representa las coordenadas del vértice, N su normal y TC sus coordenadas de textura

En la figura 2.11 se aprecian tres tipos de datos almacenados en el buffer, luego sus valores de **Pointer**, **Size** y **Stride** son:

- Para las coordenadas:  
Su **Pointer** es 0 ya que en el 0 tenemos su primera aparición, su **Size** es de 3, mientras que el **Stride** es de 8 ya que hasta la octava posición no vuelve a haber un bloque de coordenadas
- Para las normales:  
Su **Pointer** es 3, su **Size** es de 3, y su **Stride** es de 8.
- Para las texturas:  
Su **Pointer** es 6, su **Size** es de 2, y su **Stride** es de 8.

Así pues, gracias a estos dos objetos, [MrBuffer](#) y [MrBufferKey](#), una malla tridimensional es capaz de enviar la información necesaria a los vértices a la GPU

#### 2.2.6.2. Materiales

Los materiales son los elementos encargados de discernir como afecta la luz a la superficie de la malla. Y se encuentran definidos por la clase [MrMaterial](#).

Cuando una luz actúa sobre una malla con un cierto material se usan los distintos atributos del material para determinar como este reaccionará ante la luz ambiental, difusa y especular. Por defecto en MrRobotto 3D Engine la reflexión de la luz se hace mediante *Blinn-Phong* sin embargo el código fuente de los shaders puede modificarse y conseguir así otro tipo de reflexión como toon-shading por ejemplo.

### 2.2.6.3. Texturas

Las texturas son un componente opcional de los modelos y se encuentran altamente relacionadas con los materiales pues definen el aspecto de las mallas al ser renderizadas. Las imágenes utilizadas como texturas proceden del fichero MRR y se almacenan en memoria. Una vez la textura asociada a una imagen es inicializada, y con ello, almacenada en memoria nativa, la imagen es liberada de los recursos del sistema.

Además, y ya que se trabaja sobre sistemas con pocos recursos, se recomienda que dichas imágenes sean siempre dentro de lo posible cuadradas, con lado de medida una potencia de dos y lo más pequeñas posibles. Las medidas usuales y recomendadas son de  $256 \times 256$  o  $512 \times 512$ .

Las texturas vienen definidas dentro de la clase [MrTexture](#)

### 2.2.6.4. Animación

La animación dentro de MrRobotto 3D Engine es la generada a partir de un esqueleto que afecte a la malla 3D.

Para conseguir esta funcionalidad se requieren múltiples elementos:

Por una parte hace falta un esqueleto, que consiste en un conjunto de huesos estructurados de forma jerárquica y donde cada uno actuará sobre un conjunto de vértices de la malla. Además el esqueleto es capaz de controlar las distintas animaciones que tiene asociadas, siendo capaz de proporcionar la localización y rotación de cada hueso a cada frame.

El esqueleto viene definido dentro de la clase [MrSkeleton](#) y como ya se ha dicho se encuentra compuesto por huesos definidos en la clase [MrBone](#).

Ahora bien, tal y como se ha comentado el esqueleto es capaz de controlar animaciones, pero para generar estas es necesario el uso de otras clases adicionales, estas son:

- Los [MrFrame](#) almacenan los frames de la animación. Cada uno tiene una lista de huesos asociados y es la clase encargada de interpolar los keyframes importados de la animación al motor.
- La [MrKeyFrameList](#) es la encargada de almacenar los keyframes importados de la animación. Y va generando los frames a medida que se solicitan. Si el keyframe

existe porque ha sido exportado se da este, en otro caso, se interpola a partir del keyframe previo a él y el siguiente.

- [MrSkeletalAction](#) refleja una animación producida desde un esqueleto. Almacena la lista de keyframes y es la clase que se encuentra por debajo

#### 2.2.6.5. Shader Program

Unos de los componentes más importantes dentro de los modelos son los Shader Programs, pues son los encargados de mantener el código que se usará en la GPU a la hora de renderizar.

Se encuentra formada por tres elementos fundamentales:

- **Attributes:** Son la entrada de datos por cada vértice del Shader Program, en ellos se encuentran los datos enviados desde una malla 3D, y en el caso de los modelos, los Attributes se corresponden biyectivamente con los [MrBufferKey](#) de la malla del modelo.  
Están representados por la clase [MrAttribute](#).
- **Uniforms:** Los Uniforms son elementos que permanecen constantes para los conjuntos de vértices enviados en bloque a la CPU.  
Almacenan un **UniformType** que es la constante que permite identificar de forma única al uniform dentro del shader.  
Están definidos dentro de la clase [MrUniform](#).
- **Shaders:** Los Shaders por último son la clase que almacena el código que se ejecutará en la GPU. Los dos tipos posibles de Shaders implementados dentro de MrRobotto 3D Engine son los Vertex Shaders, encargados del procesado por vértice de una malla y los Fragment Shaders, encargados del procesado por pixel de la imagen generada.

#### 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

La carga de datos se podría distinguir en tres partes.

En primer lugar y como punto de entrada la carga del MRR file del que se encarga la clase [MrMrrLoader](#) que se encarga de separar la sección JSON de la sección de las texturas.

Una vez separadas los bitmaps procedentes de las texturas se almacenan dentro de una instancia de la clase [MrResources](#), que se trata de una clase auxiliar usada durante la carga de datos a modo de almacenamiento de recursos temporales o compartidos.

Por otra parte, durante el procesamiento de la sección JSON se pasa la información a la clase [MrRobottoJsonLoader](#), en ella el trabajo se irá delegando de forma continua en clases especializadas, esto quiere decir que habrá un cargador de datos por cada sección de datos relevante, por ejemplo, podremos encontrar cargadores para los modelos, las luces o las cámaras ya que requieren un tratamiento distinto y generan un resultado distinto. Es por eso que todas estas clases en las que se delegan extienden la clase genérica [MrJsonBaseLoader](#), que proporciona una interfaz unificada y directa para la obtención de los datos.

### 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.

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

### 2.2.9. Paquete Matemático

El paquete matemático de MrRobotto 3D Engine es posiblemente uno de los paquetes con más peso dentro de la aplicación. Es por ello que se ha invertido una gran cantidad

de tiempo en su optimización y que sea sencillo de usar.

La interfaz base para todos los objetos matemáticos dentro de MrRobotto 3D Engine es la denominada [MrLinearAlgebraObject](#), su método más importante es posiblemente el método `float[] getValues` que es el encargado final de transmitir los datos almacenados en él hacia los Uniforms dentro de los Shader Programs.

Dependiendo de esta interfaz se encuentran clases usuales dentro de las aplicaciones 3D, a saber:

- Matrices 4x4: Encargadas de representar transformaciones lineales dentro del espacio tridimensional usando coordenadas homogéneas.  
Las matrices 4x4 están representada por la clase [MrMatrix4f](#)
- Cuaterniones: Orientados hacia las rotaciones  
Los cuaterniones está representada por la clase [MrQuaternion](#) dentro del motor.
- Vectores de 3 y 4 componentes: Usados para representar puntos en el espacio, direcciones e incluso colores almacenados como RGB o RGBA.  
Los vectores de 3 y 4 componentes vienen representados por las clases [MrVector3f](#) y [MrVector4f](#) respectivamente.

Sobre estas clases fundamentales que son capaces de representar las transformaciones comunes del espacio tridimensional se fundamenta la clase [MrTransform](#), clase con un gran peso dentro de MrRobotto 3D Engine pues es la encargada de gestionar todas las transformaciones de un objeto de la escena.

Lleva a cabo transformaciones simples como rotar, trasladar o escalar, pero a su vez también se encarga de operaciones más complejas como la de rotar alrededor de un eje arbitrario y pasando por un cierto punto, rotar alrededor de un punto,...

Además otra de sus funciones es la de mantener un sistema de coordenadas locales del objeto, permitiendo tener una referencia de conceptos como que es "ir hacia adelante o ir hacia arriba" para un objeto independientemente del mundo.

Por otra parte, y como ya se ha mencionado, esta sección se encuentra altamente optimizada. Los métodos de operaciones matemáticas requieren de forma obligatoria que se les inyecte una instancia de la clase resultante, de esta forma se consigue delegar en un nivel superior la gestión de la memoria.

El por qué de esto se debe a que resulta altamente costoso el hecho de, por ejemplo,

multiplicar dos matrices y tener que crear una nueva cada vez en la que se devuelva el resultado, y más aún si consideramos que gran parte de esas operaciones se producen en cada frame.

## Capítulo 3

# MrRobotto Studio

### 3.1. La Aplicación MrRobotto Studio

MrRobotto Studio consiste en una aplicación distribuida con un cliente web y un cliente Android respaldados ambos por un servidor construido sobre el framework web para Python, Django.

Esta aplicación se construye ante la necesidad de poder visualizar la escena actual durante el desarrollo. El proceso natural del desarrollo en Android consiste en inclusión de recursos, construcción del proyecto e instalación en el dispositivo. Realizar esta costosa operación para pequeños cambios como podría ser una pequeña traslación de un objeto, rotar la cámara,... llega a ser un proceso lento y tedioso, gracias a MrRobotto Studio se consigue visualizar el estado de la escena de forma instantánea.

### 3.2. Blender Scripting

Como se ha dicho, el motivo de existir de MrRobotto Studio se basa en la necesidad de simplificar el desarrollo, y entre esas tareas está el poder ofrecer un entorno completo en el que, hasta la exportación de las escenas almacenadas en blender se simplifique.

Para la exportación desde Blender se hace uso de la API para Python.  
Se explicará a continuación la estructura que sigue el script de exportación.

### 3.2.1. Datos

Los datos obtenidos de Blender, como podrían ser los vértices de las mallas, la localización de los huesos,... se almacenan en objetos planos.

Haciendo uso de la biblioteca de JSON para Python convertiremos estos objetos en instancias JSON una vez haya finalizado el proceso de exportación.

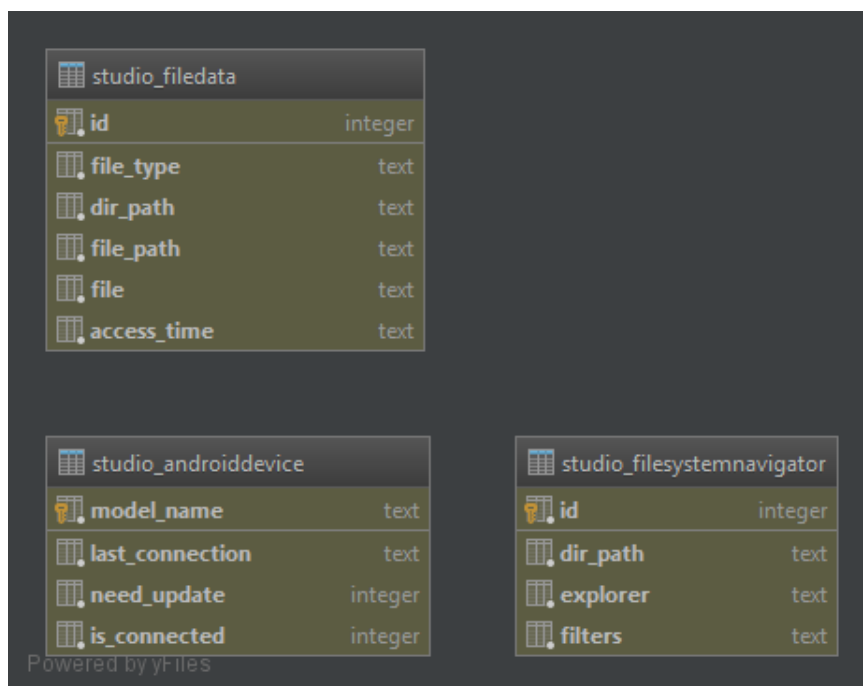
### 3.2.2. Exportadores

Los exportadores son los elementos fundamentales encargados de convertir objetos de Blender a los objetos planos que serán exportados.

Cabe destacar que mientras que algunos exportadores son relativamente sencillos como podrían ser los exportadores de los materiales otros

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



studio_filedata	
id	integer
file_type	text
dir_path	text
file_path	text
file	text
access_time	text

studio_androiddevice	
model_name	text
last_connection	text
need_update	integer
is_connected	integer

studio_filesystemnavigator	
id	integer
dir_path	text
explorer	text
filters	text

Powered by yFiles

FIGURA 3.1: Datos almacenados en MrRobotto Studio

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

El cliente Android es la parte de la aplicación que hace de mensajero entre el servidor Django y MrRobotto 3D Engine.

Está construída como una aplicación sencilla y ligera para evitar

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

## Capítulo 4

# Resultados

### 4.1. Resultados de Rendimiento

#### 4.1.1. Tiempos de carga

TODO: Cuadro con los tiempos de carga de una escena con un numero variable de modelos

#### 4.1.2. Renderizado de Múltiples Objetos Simples

Se ha decidido comprobar el rendimiento de MrRobotto 3D Engine al renderizar múltiples objetos simples, es decir, objetos carentes de texturas o animaciones.

Todos estos objetos son individuales, es decir, aunque se vean iguales cada uno posee una malla diferente asociada.

CUADRO 4.1: My caption

Nº de objetos	5	10	20	50	80
FPS	60	60	59	45	28

#### 4.1.3. Renderizado de Múltiples Objetos Animados

TODO: Cuadro con los FPS al renderizar una escena con por ejemplo 5, 20, 50 y 100 modelos realizando una animacion

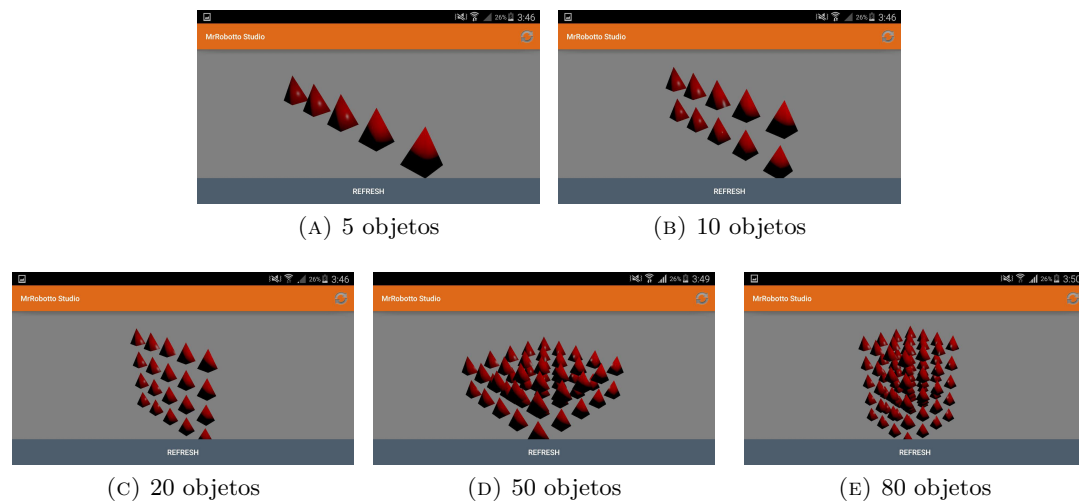


FIGURA 4.1: Renderizado de objetos simples

#### 4.1.4. Renderizado de Múltiples Objetos y Múltiples Texturas

TODO: Cuadro con los FPS al renderizar una escena con por ejemplo 5, 20, 50 y 100 modelos cada uno con textura distinta

## Capítulo 5

# Conclusiones y Posibles Mejoras

### 5.1. Conclusiones

### 5.2. Mejoras Futuras

TODO: Portar a C++ Mejorar la carga Mejorar el formato MRR Añadir otro tipo de luces Compartir modelos

## 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:

<b>Sección</b>	<b>Cabecera</b>	
<b>Valor</b>	MRROBOTTOFILE	
	MRROBOTTOFILE	Tag
<b>Sección</b>	<b>Sección de cabecera de JSON</b>	
<b>Valor</b>	JSON N	
	JSON	Tag
	N	Tamaño de la sección JSON representado como un número entero de 4 bytes en codificación big endian.
<b>Sección</b>	<b>Sección de datos JSON</b>	
<b>Valor</b>	json	
	json	Cadena de caracteres en codificación ASCII de longitud N y con formato JSON
<b>Sección</b>	<b>Cabecera de la sección de texturas (Opcional)</b>	
<b>Valor</b>	TEXT M	
	TEXT	Tag
	M	Tamaño de la sección de texturas representado como un número entero de 4 bytes en codificación big endian.
<b>Sección</b>	<b>Sección de texturas (Opcional)</b>	
<b>Valor</b>	texturas	
	texturas	Datos binarios de las texturas almacenadas y de longitud M.
<b>Sección</b>	<b>Sección de nombre de textura</b>	
<b>Valor</b>	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
<b>Sección</b>	<b>Texture data Sección (Optional)</b>	
<b>Valor</b>	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; \n uniform vec4 mrAmbi
    },
    "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/OpenGL\\_ES\\_Programming\\_Guide/TechniquesforWorkingwithVertexData/TechniquesforWorkingwithVertexData.html](https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGL_ES_Programming_Guide/TechniquesforWorkingwithVertexData/TechniquesforWorkingwithVertexData.html).
- [2] Kevin Brothaler. An introduction to index buffer objects (ibos), Mayo 2012. URL <http://www.learnopengles.com/android-lesson-eight-an-introduction-to-index-buffer-objects-ibos/>.