INSTITUTO POLITÉCNICO NACIONAL

CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN

Motor de juegos 3D para una plataforma móvil

Tesis

que para obtener el grado de

Maestría en Ciencias de la Computación

presenta:

Ing. Raymundo Pescador Piedra

Directores de tesis:

M. en C. Sergio Sandoval Reyes

y

Dr. José Giovanni Guzmán Lugo

Laboratorio de Comunicaciones y Redes de Computadoras

México D.F. Octubre de 2014

SIP-14 bis

INSTITUTO POLITÉCNICO NACIONAL
SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISIÓN DE TESIS

En la Ciudad de México, D.F. siendo las 11:00 horas del día 11 del mes de junio de 2014 se reunieron los miembros de la Comisión Revisora de la Tesis, designada por el Colegio de Profesores de Estudios de Posgrado e Investigación del:

Centro de Investigación en Computación

para examinar la tesis titulada:

"Motor de juegos 3D para una plataforma móvil"

Presentada por la alumno:

PESCADOR	**PIEDRA**	**RAYMUNDO**
Apellido paterno	Apellido materno	Nombre(s)

Con registro:

B	1	2	1	0	8	2

aspirante de: **MAESTRÍA EN CIENCIAS DE LA COMPUTACIÓN**

Después de intercambiar opiniones los miembros de la Comisión manifestaron ***APROBAR LA TESIS***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISIÓN REVISORA
Directores de Tesis

Dr. José Giovanni Guzmán Lugo

M. en C. Sergio Sandoval Reyes

Dr. Miguel Jesús Torres Ruiz

Dr. Rolando Menchaca Méndez

Dr. Eleazar Aguirre Anaya

Dr. Moisés Salinas Rosales

PRESIDENTE DEL COLEGIO DE PROFESORES

Dr. Luis Alfonso Villa Vargas

INSTITUTO POLITECNICO NACIONAL
CENTRO DE INVESTIGACION EN COMPUTACION
DIRECCION

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA DE CESIÓN DE DERECHOS

México D.F., el día **30** del mes de **junio** del año **2014**, el que suscribe, **Raymundo Pescador Piedra**, alumno del programa de Maestría en Ciencias de la Computación, con número de registro **B121082**, adscrito al **Centro de Investigación en Computación**, manifiesta que es autor intelectual del presente trabajo de Tesis, bajo la dirección de **M. en C. Sergio Sandoval Reyes** y **Dr. José Giovanni Guzmán Lugo**, y cede los derechos del trabajo titulado **Motor de juegos 3D para una plataforma móvil**, al Instituto Politécnico Nacional para su difusión, con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o directores del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección **hasternet@me.com**. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

Raymundo Pescador Piedra

Resumen

El desarrollo de dispositivos móviles con capacidad de procesamiento y memoria equiparables a computadoras personales, ha logrado abrir una brecha muy importante en el desarrollo de contenido multimedia para estos dispositivos. Una de las vertientes más importantes es el desarrollo de videojuegos para estas plataformas móviles.

En esta tesis, se propone la construcción de un motor de juegos que permita un diseño estructurado y la simplificación en el desarrollo de videojuegos tridimensionales para una plataforma móvil de gran presencia en el mercado actual: el sistema operativo *iOS*. Utilizando *OpenGLES* como el lenguaje de aceleración de gráficos, se crearon un compendio de clases, métodos y objetos implementados completamente en el lenguaje de alto nivel *Objective-C*, de manera que puedan ser utilizadas como una API de desarrollo altamente modular y extendible.

En base a este desarrollo, se realizarán pruebas de desempeño, con la intención de realizar una comparación con las tecnologías existentes. De igual manera, se pretende demostrar la validez y funcionalidad del motor, desarrollando una aplicación haciendo uso de esta *API*, de manera que las bondades y funcionalidades del mismo, se exhiban en un ambiente de desarrollo real.

Abstract

The development of mobile devices with processing capabilities and memory comparable to personal computers has managed to open a very important divide in the development of multimedia content. One of the most important is the game development for these mobile platforms.

In this thesis, we propose the construction of a three-dimensional game engine that allows a structured design and simplification in the development of three-dimensional video games for a mobile platform with large presence in the market: the *iOS* operating system. Using *OpenGLES* as graphics acceleration language, we pretend the creation of a collection of classes, methods and objects, implemented entirely in the high level language Objective-C, so that it can be used as a development API highly modular and extensible.

Based on this development, performance tests are conducted with the intention of making a comparison with existing technologies. Likewise, it aims to demonstrate the validity and functionality of the engine, we develope an application using this API, so that the benefits and features thereof, are displayed on a real development environment.

Agradecimientos

Al Instituto Politécnico Nacional que me ha brindado, a lo largo de casi 10 años, la oportunidad de prepararme como profesionista, persona y ciudadano.

Al Centro de Investigación en Computación, que me permitió expandir mis conocimientos a través de sus profesores e investigadores.

Por último, agradezco al Consejo Nacional de Ciencia y Tecnología (CONACYT), por el apoyo brindado durante mis estudios de maestría.

Índice General

Índice de Figuras

Glosario de Términos

Android	Sistema operativo basado en *Linux* diseñado principalmente para dispositivos móviles con pantalla táctil, como teléfonos inteligentes o tabletas. Su desarrollo lo inicia *Android Inc*, que *Google* respaldó económicamente y terminó comprando en 2005.
API	Del inglés *Application Programming Interface*. Conjunto de funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.
Box2D	Biblioteca de licencia libre que implementa un motor de física en dos dimensiones. Está programada en C++, desarrollado por Erin Catto, y distribuido bajo la licencia zlib.
Cocoa Touch	API para la creación de aplicaciones para el iPad, iPhone y iPod Touch de la compañía Apple. *Cocoa Touch* proporciona una capa de abstracción al sistema operativo *iOS*, de manera que expone las principales funciones de los dispositivos en una serie de bibliotecas disponibles para los desarrolladores.
Delegado	Del inglés *delegate*. Es un objeto que actúa en nombre de o en coordinación con otro objeto cuando el objeto se encuentra con un evento en un programa. El objeto de la delegación es, a menudo, una respuesta a mensajes, es decir, cuando el objeto delegado recibe un evento que se ajusta a las reglas de la delegación, éste recibe la respuesta y actúa de acuerdo a las reglas establecidas en el protocolo.
Framework	Estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de software concretos, que puede servir de base para la organización y desarrollo de software. Típicamente puede incluir soporte de programas, bibliotecas, y un lenguaje interpretado, entre otras herramientas, para así ayudar a desarrollar y unir los diferentes componentes de un proyecto.
FrameBuffer	Categoría de dispositivos gráficos, que representan cada uno de los píxeles de la pantalla como ubicaciones en la memoria de acceso aleatorio. También se le llama así, en el área de los sistemas operativos, a los dispositivos que usan o aparentan usar dicho método de acceso a gráficos.
GamePlay	La *jugabilidad* es un término empleado en el diseño y análisis de juegos que describe la temática del juego en términos de sus reglas de funcionamiento y de su diseño. Se refiere a todas las experiencias de un jugador durante la interacción con sistemas de juegos.
GLKit	Framework de *Cocoa Touch* introducido en la versión 5.0 del sistema operativo *iOS*. Este *framework* establece los mecanismos de inicialización, y configuración de *OpenGL*, reduciendo el trabajo del desarrollador en definir estas tareas.

GLSL	Acrónimo de *OpenGL Shading Language* (Lenguaje de Sombreado de *OpenGL*). Es una tecnología parte del API estándar *OpenGL*, que permite especificar segmentos de programas gráficos que serán ejecutados sobre el GPU. GLSL es un lenguaje de sombreado de alto nivel basado en el lenguaje de programación C.
GPU	Acrónimo de *Graphics Processing Unit*. Es un coprocesador dedicado al procesamiento de gráficos u operaciones de punto flotante, para aligerar la carga de trabajo del procesador central en aplicaciones como los videojuegos y o aplicaciones 3D interactivas.
GUI	Acrónimo de *Graphical User Interface*. Es un programa informático que actúa como interfaz de usuario, utilizando un conjunto de imágenes y objetos gráficos para representar la información y acciones disponibles en la interfaz.
HUD	Acrónimo de *Heads-Up Display*. Es la información que en todo momento se muestra en pantalla durante la partida, generalmente en forma de iconos y números. El *HUD* suele mostrar el número de vidas, puntos, nivel de salud y armadura, mini-mapa etcétera; todo ello dependiendo del juego.
IDE	Acrónimo de *Integrated Development Environment*. Término utilizado para referirse a un programa informático compuesto por un conjunto de herramientas de programación. Puede dedicarse en exclusiva a un solo lenguaje de programación o bien puede utilizarse para varios.
Instruments	Herramienta de análisis de desempeño de aplicaciones para sistemas operativos *OSX* y *iOS*. Permite el análisis dinámico de memoria y desempeño del procesador en tiempo de ejecución.
iOS	Sistema operativo móvil desarrollado por la la empresa *Apple Inc*. Originalmente desarrollado para el iPhone (*iPhone OS*), siendo después usado en dispositivos como el *iPod Touch*, *iPad* y el *Apple TV*.
Motor de Juegos	Término que hace referencia a una serie de rutinas de programación que permiten el diseño, la creación y la representación de un videojuego. Del mismo modo existen motores de juegos que operan tanto en consolas de videojuegos como en sistemas operativos. La funcionalidad básica de un motor es proveer al videojuego de un motor de renderizado para los gráficos 2D y 3D, motor físico o detector de colisiones, sonidos, scripting, animación, inteligencia artificial, redes, streaming, administración de memoria y un escenario gráfico.
MVC	Patrón de arquitectura de software que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario.

Objective-C	Lenguaje de programación orientado a objetos creado como un superconjunto del lenguaje C para que implementase un modelo de objetos parecido al de *Smalltalk*. Actualmente se usa como lenguaje principal de programación en *Mac OS X*, *iOS y GNUstep*.
OpenGL	Especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D
OpenGLES	Es una variante simplificada de la API gráfica *OpenGL* diseñada para dispositivos integrados tales como teléfonos móviles, PDAs y consolas de videojuegos.
Protocolo	(Dentro del contexto de *Objective-C*). Se denomina protocolo al conjunto de métodos y propiedades a las cuales se ajusta un objeto, en conformidad con su delegado. El objeto delegado, presenta la implementación de los métodos que se llaman, mientras que el objeto al que se delega, se encarga de realizar la llamada a los mismos.
Renderizado	Del inglés *render*. Término utilizado en informática para referirse al proceso de generar una imagen o vídeo mediante el cálculo de iluminación GI partiendo de un modelo.
SDK	Acrónimo de *Software Development Kit*. Se denomina así generalmente a un conjunto de herramientas de desarrollo de software que le permite al programador crear aplicaciones para un sistema concreto, por ejemplo ciertos paquetes de software, *frameworks*, plataformas de hardware, computadoras, videoconsolas, sistemas operativos, etc.
Shader	Módulo de software escrita en un lenguaje de sombreado que se puede compilar independientemente. Los shaders son utilizados para realizar transformaciones y crear efectos especiales, como por ejemplo iluminación, fuego o niebla. Para su programación los shaders utilizan lenguajes específicos de alto nivel que permitan la independencia del hardware.
XCode	Entorno de desarrollo integrado propiedad de Apple Inc. y distribuido gratuitamente junto con *Mac OSX*. *Xcode* trabaja conjuntamente con *Interface Builder*, una herencia de NeXT, una herramienta gráfica para la creación de interfaces de usuario.

Capítulo 1

INTRODUCCIÓN

El entretenimiento interactivo ha crecido de manera sorprendente a lo largo de las últimas dos décadas y esto no es obra de la casualidad. El avance en materia de computación permite el desarrollo de equipos de cómputo que fácilmente eclipsan a sus predecesores; en nuestros días es claro que esperamos una pequeña revolución cada año. El mundo virtual está cada vez mas presente en nuestro día a día.
Un pequeño eslabón dentro de este mundo son los videojuegos. Estos productos de software se han convertido, poco a poco, en elementos preponderantes de la comodidad del hombre moderno, creando un nicho de mercado de miles de millones de dólares. Actualmente ya no es necesario contar con complejos productos de hardware para poder disfrutar, en cualquier momento del día, un lapso de entretenimiento que permita distraernos de nuestras actividades. Basta con prender una consola de videojuegos en nuestro hogar, abrir alguna aplicación en nuestra computadora, o simplemente tomar nuestro teléfono celular y seleccionar alguna de las tantas aplicaciones presentes en el mercado, capaces de consumir desde 15 minutos de nuestro tiempo, hasta noches enteras, todo por el placer de conquistar el castillo, destruir los ejércitos enemigos o cosechar el maíz que necesitamos para nuestra granja. Todo un mundo de posibilidades en la palma de la mano. Por ello no es de extrañar que este importante nicho de mercado sea el centro de los reflectores. Pero ¿Qué lo hace realmente atractivo en estos días?
Pensemos por un segundo cómo funcionaba el negocio de los videojuegos en la década de los 80's, tomando como ejemplo la consola *SNES* (*Super Nintendo Entertainment System*), desarrollada por la empresa *Nintendo* en el año 1985. Esta consola era un dispositivo electrónico con un procesador de 8 bits, memoria RAM de 2 KB, alimentada a 230 V AC. Claramente podemos observar algo: era una computadora de propósito específico, el desarrollo del hardware es propiedad de la compañía que la desarrolló, y más importante aún, el desarrollo de software sobre esta plataforma se encontraba licenciado al 100%, esto es, si se quería desarrollar software para esta consola, se tenía que llegar a un acuerdo económico con *Nintendo*. *Nintendo*, en caso de aceptar, provee al equipo de desarrollo: hardware y software para programar. Esto ejemplifica que el desarrollo de software de juegos no podía ser alcanzado por desarrolladores que no fueran, en principio, empresas con el suficiente poder económico.

En el año 1993, surge uno de los juegos más populares que han existido, *Doom[1]*. Desarrollado por la empresa "*id Software*", *Doom* es un videojuego *FPS* (*First Person Shooter*), programado para el sistema operativo *DOS*. El argumento es simple, el usuario debe recorrer una serie de niveles donde debe disparar a diversos enemigos, cada uno de los cuales cuenta con las particularidades propias del nivel donde se encuentre y el escenario donde se desarrolle el mismo.
El proyecto *Doom* resultó ser un éxito comercial, pero más aún, a nivel de arquitectura de software, demuestra contar con un módulo dentro de su programación, un módulo de software que en el futuro se conocería como *motor de juegos*.

Un motor de juegos se define como el módulo de software que es capaz de abstraer los detalles de las tareas comunes en cualquier juego de video (tales como el manejo de memoria, el manejo de las estructuras de datos y recursos del juego, el *renderizado* de gráficos, las entradas de datos, etcétera), de manera que el usuario final (desarrolladores de juegos) puedan centrar su atención en los detalles específicos de su desarrollo*[2]*.

El proyecto *Doom* demostró que es posible generar este módulo totalmente separado de la lógica del juego, brindando la posibilidad de modificar simplemente los demás módulos del sistema, y obtener un juego completamente diferente; así, el motor de juegos se convierte en un núcleo capaz de generar una infinidad de juegos que posean estructuras definidas. En los siguientes años, miles de desarrolladores utilizaban el motor de *Doom*, o los propios para desarrollar sus propios videojuegos. Esta práctica pronto se extendió a otros proyectos, los desarrolladores de software aportaban más y más ideas a través de Internet, una nueva manera de desarrollar videojuegos había nacido.
Con el poder para desarrollar videojuegos, el mercado se abría a posibilidades ilimitadas, las computadoras, cada vez más presentes en todo el mundo, se presentaban como un mercado infinito, al cual se podía llegar con videojuegos novedosos, tan complejos y ambiciosos como el programador quisiera.
Tiempo después, un acontecimiento muy importante llegaría a potenciar aún más el desarrollo de videojuegos de manera independiente: el surgimiento de teléfonos inteligentes. Con la llegada de los teléfonos, pequeñas computadoras que se actualizan con el hardware más avanzado, y la disposición de las empresas que desarrollan dichos teléfonos de proveer las *API*s para desarrollar aplicaciones sobre ellos, el desarrollo de videojuegos solo necesitaba migrar el conocimiento del motor de juegos a esta nueva tecnología.

En las secciones siguientes se describirá el diseño de motores de juegos, sus características y los diferentes módulos que lo integran. Se discutirá acerca del sistema operativo *iOS*, plataforma que ha sido seleccionada para el desarrollo de este trabajo; asimismo, se describirá el funcionamiento de *OpenGL*, tecnología encargada del *renderizado* de los gráficos en estos dispositivos. La descripción de todos estos elementos, permiten brindar un preámbulo para el entendimiento del presente trabajo.

1.1 Motor de Juegos

El término motor de juegos nace alrededor de 1990, al surgir el proyecto *Doom*, creado por la empresa *id Software*. El proyecto *Doom* fue diseñado de tal manera que existía una separación bien definida entre los elementos de software centrales (el sistema de *renderizado* de gráficos, el sistema de detección de colisiones, o el sistema de audio) y los elementos de diseño y reglas del juego. Esto despertó el interés de la comunidad de desarrolladores, de manera que poco a poco, surgieron comunidades de programadores que tomaban como base los elementos centrales del núcleo del juego, y realizaban modificaciones sobre las capas que les sucedían. En este punto, es donde se da el nacimiento de las comunidades *mod*.
Las comunidades *mod*, se describen como grupos de desarrolladores o estudios independientes que se comunican, principalmente a través de Internet, con el fin de crear nuevos videojuegos modificando los existentes, utilizando *toolkits* gratuitos que ponen a disposición las compañías desarrolladoras de los juegos originales. De hecho, el término *mod* hace referencia a la palabra inglesa *modification*. Que describe el proceso de modificar el diseño de un juego, y adaptarlo a conveniencia.
A partir del final de la década de los 90, juegos como *Quake III Arena* y *Unreal Tournament*, fueron diseñados con la arquitectura de rehuso y "*modding*" como piedra angular. Estos motores eran totalmente modificables vía lenguajes de scripting, tales como *Quake C*, permitiendo a los desarrolladores contar con las herramientas para modificar, de manera fácil y totalmente estandarizada, dichos proyectos. Este cambio permitió a las empresas contar con un mayor control de las versiones y modificaciones que se realizaban sobre sus productos, permitiendo también a los desarrolladores utilizar el núcleo del juego para crear sus propios productos.

1.1.1 Arquitectura de un Motor de Juegos

Como se ha descrito en párrafos anteriores, un motor de juegos es el núcleo de un videojuego, capaz de poner en marcha la máquina de estados de un videojuego, y hacerlo correr sobre la plataforma para la cual está diseñado. Ahora bien, dentro de este núcleo, podemos encontrar varios módulos, bien definidos que cumplen, uno a uno, los propósitos para los cuales están diseñados. Un motor de juegos se compone, de manera sintetizada, por los elementos que se muestran en la Figura 1.1[*]

Figura 1.1 Arquitectura de un motor de juegos

La arquitectura muestra el diseño por capas del cual se compone el motor de juegos. El enlace existente entre cada una de las capas se encuentra delimitado por tecnologías que permiten, a las capa superiores, poder realizar, de manera transparente, operaciones que correspondan al nivel donde se encuentra.

[*] *[3]. Sección 1.6 Runtime Engine Architecture*

No es de extrañar entonces que la modularidad antes descrita de los motores de juegos actuales permita la reutilización de código y de funcionalidades que ya han sido implementadas anteriormente. A continuación se dará una breve descripción de cada una de las capas de la arquitectura, para mostrar las funcionalidades y alcances de cada una de ellas.

1.1.1.1 Capa de hardware.

La capa de hardware representa el sistema computacional o consola en la cual el juego correrá. El lenguaje utilizado para acceder a esta capa son primitivas en lenguaje ensamblador principalmente, encargadas de realizar llamadas de sistema directamente a componentes de hardware presentes en el sistema en el que se está desarrollando.

1.1.1.2 Capa de drivers.

La capa de drivers está compuesta de componentes de software de bajo nivel, los cuales son proveídos directamente por el desarrollador de hardware. Los drivers se encargan de manejar las llamadas a sistema y proveen un enlace entre los diferentes dispositivos presentes en el sistema de hardware.

1.1.1.3 Capa de sistema operativo.

El sistema operativo es la entidad que se encarga de la administración de todos los recursos de hardware presentes en un sistema computacional. El sistema operativo realiza llamadas directamente a los drivers, con lo cual pueda manejar y administrar todos los dispositivos presentes en el sistema.

1.1.1.4 Capa de *third-party SDKs* y *middleware*.

La capa de Middleware cuenta con diferentes librerías que permiten el desarrollo de funciones tales como manejo de estructuras de datos, dependiendo la plataforma que se seleccione. Entre las más importantes tenemos las siguientes:
- *DirectX*. Propiedad de *Microsoft[4]*. Consiste en una *API* para la programación de gráficos 3D. Actualmente se encuentra disponible para los sistemas operativos de 32 y 64 bits, así como para consolas Xbox y Xbox 360. *Microsoft DirectX* es un grupo de tecnologías diseñado para convertir los equipos basados en Windows en plataformas ideales para la ejecución y visualización de aplicaciones con abundantes elementos multimedia como gráficos a todo color, vídeo, animación 3D y extenso contenido de audio. *DirectX* incluye actualizaciones de rendimiento y seguridad, junto con muchas y nuevas características en todas las tecnologías, a todo lo cual podrán tener acceso las aplicaciones que utilicen las *API*s de *DirectX*. Se usa principalmente en aplicaciones donde el rendimiento es fundamental, como los videojuegos, aprovechando el hardware de aceleración gráfica disponible en la tarjeta gráfica.
El principal competidor de *DirectX* es *OpenGL*, desarrollado por *Silicon Graphics Inc*.

- *OpenGL*. *OpenGL*[†] es una interfaz de software para la aceleración de gráficos por hardware. Esta interface consiste en cerca de 150 comandos con los cuales se pueden especificar los objetos y operaciones para desarrollar aplicaciones con gráficos en dos o tres dimensiones.

OpenGL está diseñado como una interface independiente del hardware para ser implementado en diferentes plataformas de hardware. Para lograr estos alcances, no existen directivas para realizar interacciones con el usuario, o con capas de entrada de datos, debido a que se trata de evitar dependencia de hardware más allá de los procesadores de gráficos (GPU).

[†] *[5]. Capítulo 1: "Introduction to OpenGL"*

Para atacar esta limitante, existen capas que permiten interactuar, en base a la plataforma donde se encuentre, con las salidas del sistema.

Actualmente *OpenGL* se encuentra presente en varios productos del mercado, principalmente se encuentra en teléfonos inteligentes que cuentan con sistemas operativos *Android* e *iOS*.

En capítulos posteriores se hará énfasis en los componentes presentes en *OpenGL*. Como se ha comentado anteriormente, los dispositivos *iOS* cuentan con esta tecnología para aceleración de gráficos por hardware, por lo cual, su análisis resulta de particular interés para el presente trabajo.

1.1.1.5 Capa plataforma independiente del sistema.

En la realización de videojuegos es común el desarrollo de proyectos para diversos sistemas operativos. La capa independiente del sistema, se encarga de las adecuaciones necesarias para permitir al producto de software el ser capaz de adaptarse a la plataforma que lo contenga. Estas adecuaciones van desde la detección de la plataforma existente, la memoria residente, los archivos del sistema hasta los recursos de hardware de los cuales se puede hacer uso. En motores de juegos especializados, como es el caso del presente trabajo, esta capa es totalmente transparente o inexistente.

1.1.1.6 Capa núcleo del sistema.

Independientemente de la plataforma en la cual se desarrolle o ejecute el juego, existen directivas y primitivas que deben ser atendidas directamente por el sistema operativo. Procesamiento de cálculos matemáticos, asignación de memoria, localización de servicios, manejo y administración de flujos de entrada y salida, entre otros servicios mas son funciones que, aun cuando son requeridas y definidas por el motor de juegos, son realizadas por el sistema operativo, el cual tendrá la tarea de ejecutarlas en su totalidad.

1.1.1.7 Capa de recursos.

En proyectos de software de gran tamaño la generación de recursos se incrementa de manera importante. Desde complejos escenarios, hasta botones del menú de usuario, los recursos se convierten en un problema que crecen de manera exponencial en proporción a la complejidad del juego. En este caso, los desarrolladores generan manejadores de recursos, de manera que se engloban los diseños en archivos binarios, que pueden ser comprimidos. Esto permite la ubicación de recursos de manera más eficiente (se utilizan direcciones lógicas, en vez de físicas), reduce el tamaño de los proyectos, y usualmente se utiliza para 'blindar' los recursos, de manera que el usuario final no tiene acceso real a los diseños del juego.
En motores de juegos simples, o menos complejos, esta capa no se utiliza, debido a que la elaboración de este módulo no es práctica ni presenta beneficios reales.

1.1.1.8 Capa *renderizado* de bajo nivel.

El *renderizado* de bajo nivel, es el encargado de 'dibujar' todos los elementos que solicite el motor de juegos. En este nivel, el diseño se concentra en el *renderizado* de todas las colecciones de primitivas tan rápido y eficiente como sea posible. Las librerías de aceleración de gráficos presentadas en la sección 1.1.4, son las encargadas de realizar este *renderizado* de acuerdo a qué plataforma se refiera. Sin embargo, dichas librerías necesitan varios segmentos de código para inicializar, mantener y *renderizar* las diferentes superficies que se pretenden desplegar durante la ejecución de un videojuego. Para poder mantener la vida de la aplicación es necesario diseñar una interfaz gráfica, de manera que todos los elementos que requieren ser mostrados permanezcan 'con vida' mientras sean requeridos.

Como se observa, la capa de *renderizado* de nivel es uno de los retos más importantes en el diseño de un motor de juegos dado que es la entidad que se encarga de mostrar, animar, destruir y purgar los elementos dentro del ciclo de vida del videojuego.

1.1.1.9 Capa de administración de escenas.

Usualmente, el *renderizado* es el encargado de mostrar los gráficos en un momento exacto del ciclo de vida de la aplicación; sin embargo, en un nivel más específico del juego, existirán entidades que indicarán a la capa de *renderizado* los elementos que deben o no mostrar.
La capa de administración de escenas es uno de estos componentes. Al definir el flujo de escenas que seguirá el motor de juegos podemos establecer las reglas que seguirá el *renderizado*, permitiendo la optimización de recursos haciendo uso de modelos que permitan establecer la visibilidad o no de los objetos. Si se toma como ejemplo cualquier videojuego actual, podemos ver la delimitación de las escenas claramente definida. Desde el menú inicial del juego, hasta el último nivel del mismo, recorremos una sucesión de escenas las cuales requieren de reglas que permitan la transición de manera eficiente entre ellas.

1.1.1.10 Capa de efectos visuales.

Como su nombre bien lo describe, esta capa es la encargada de los efectos visuales que pueden ser mostrados por el sistema de *renderizado*. Sistemas de partículas, sombras dinámicas, iluminación de modelos en tiempo de ejecución, son algunos de estos efectos. Al ser módulos complejos, usualmente son considerados como productos del *renderizado* de bajo nivel, no pertenecientes a la misma capa.

1.1.1.11 Capa interfaz de usuario.

La capa de interfaz de usuario es la pantalla final que se mostrará al usuario final del sistema. El motor de juegos debe de proveer de herramientas necesarias para la construcción de interfaces de usuario gráficas (*GUI*), la capa *HUD* (*Heads-Up Display*), y herramientas de *ICG* (*In Game Cinematics*), las cuales no son mas que módulos que permiten al usuario final interactuar con el sistema.

1.1.1.12 Capa de depuración.

Como sistemas en tiempo real, los videojuegos requieren de herramientas que les permitan medir su desempeño, de manera que se puedan realizar mejoras y análisis de los mismos. Los motores de juegos a gran escala cuentan con diferentes herramientas para probar su desempeño de gráficos, consumo de memoria, tiempo de respuesta etc.

Los motores de juegos a pequeña escala hacen uso de herramientas propias de la plataforma en la cual serán ejecutados. Un ejemplo claro de este escenario es el uso de la aplicación *Instruments* como analizador de desempeño de aplicaciones para *iOS*. Esta aplicación es propiedad de Apple, pero es una medida más que conveniente para analizar el desempeño de cualquier videojuego que se quiera lanzar sobre un dispositivo con *iOS*.

1.1.1.13 Capa motor de física y colisiones.

El motor de Física y Colisiones es un elemento preponderante en cualquier motor de juegos. El motor de física y colisiones (*MFC*), es el encargado de realizar la simulación de un entorno donde las leyes de la física se encuentran presentes. Al crear un mundo que se rige bajo el *MFC*, todos los objetos que se encuentren dentro del mismo responderán de manera exacta a las leyes que el desarrollador haya fijado. Desde establecer gravedad, hasta impulsos sobre objetos con propiedades físicas, el *MFC* debe garantizar el comportamiento correcto de todos los actores dentro de este entorno.

Ahora bien, las funciones se extienden un poco mas allá; la detección de colisiones puede quedarse un paso atrás de una simulación real de física, pero permite al análisis de eventos donde dos cuerpos, delimitados por su extensión, se encuentran.

La detección de colisiones es un módulo muy importante dentro de un motor de juegos; si bien en algunos escenarios la simulación física parecerá excesiva, la detección de colisiones se encuentra en cualquier escenario donde se requiera el mínimo grado de semejanza al mundo real. Uno de los motores de física más utilizados en la actualidad es la librería *Box2D*.
Box2D[6] es un motor de código abierto escrito totalmente en *C++* para la simulación de cuerpos rígidos en 2D. *Box2D* es desarrollado por Erin Catto, bajo la licencia *zlib*. Por lo tanto, no requiere de permisos acerca de su utilización, solo aconseja el brindar crédito en caso de utilizar la librería. Una de las ventajas de *Box2D* es que, al ser escrito en *C++*, es fácilmente trasladado a diferentes plataformas. De igual manera, su integración con *OpenGL* es relativamente sencillo.

1.1.1.14 Capa de mecanismos de interfaz humana.

Todos los juegos requieren de datos de entrada provenientes del usuario, para este proceso se requieren dispositivos de entrada de datos tales como: teclado, ratón, palancas de juego, o controles especializados; todo esto dependiendo de la plataforma. En esta capa, el motor de juegos debe ser consciente de las posibilidades que se pueden presentar en todos estos dispositivos de entrada, de manera que se exploten todas las características de los mismos. Usualmente, los mecanismos de detección de entrada de datos, son tomados directamente por el sistema operativo, sin embargo, el interpretar estas entradas de datos debe ser tarea del motor de juegos.

1.1.1.15 Capa de audio.

La capa de audio es una de las más importantes dentro del desarrollo de videojuegos, pero a la cual no se le presta la atención que a otras capas. Esto es, principalmente, porque depende totalmente de cada plataforma. Los sistemas operativos brindan acceso a librerías de control de recursos de audio, de manera que los desarrolladores simplemente explotan las funcionalidades que se les pongan a disposición. Esto de una manera es entendible. La estandarización del audio se basa completamente en hardware, por lo que los desarrolladores deben atenerse a las posibilidades y limitaciones presentes en las plataformas sobre las cuales se encuentren desarrollando.

1.1.1.16 Capa de administración de multi-jugador y recursos en línea.

Esta capa, dentro de la arquitectura de un motor de juegos, puede ser de vital importancia para un videojuego y totalmente innecesaria para otro. Los videojuegos a gran escala prestan atención a los detalles de *multijugador* casi tanto como en las demás capas.
Imaginemos el videojuego FIFA 14 sin la posibilidad de jugar contra un oponente de manera local, o el juego Starcraft sin la posibilidad de campañas en línea. Ahora, si desarrollamos un videojuego de tipo rompecabezas en un teléfono inteligente, veríamos que la necesidad de centrar nuestros esfuerzos en una capa multijugador no presentaría ventaja alguna o utilidad.

1.1.1.17 Capa de reglas del *gameplay*.

EL término *gameplay* hace referencia a la acción que tiene lugar durante un videojuego, las reglas que gobiernan el mundo virtual en el cual el juego tiene lugar, las habilidades y características de los personajes, los objetos presentes en los niveles, así como los objetivos y misiones de los jugadores. Esta capa está escrita, la mayoría de las veces, en el lenguaje nativo en que fue escrito el motor de juegos, o en algún lenguaje de scripting generado a partir del mismo. En esta capa aparece la distinción existente entre un arquitecto de videojuegos, o un programador de los mismos.

El arquitecto está encargado de realizar los mecanismos que permitan exponer las funcionalidades necesarias para la creación de videojuegos, haciendo uso de los recursos de hardware y software presentes en una plataforma, de manera que la interacción con estos elementos se realiza de manera transparente. Un programador toma estas mecanismos, 'de alto nivel', y los utiliza para crear los objetos, clases, personajes y todos los elementos necesarios para integrar el videojuego como tal.
Un ejemplo claro de este escenario se presenta en el desarrollo de videojuegos para dispositivos móviles. Los arquitectos de videojuegos, que usualmente se encargan de desarrollar los motores de juegos, crean los mecanismos de las capas que hemos descrito anteriormente (usualmente este trabajo se realiza programando en C o C++). Como resultado de estas funciones se crean librerías escritas en lenguajes nativos dentro de las plataformas (por ejemplo, *Java* para dispositivos *Android*, y *Objective-C* para dispositivos *iOS*), las cuales son utilizadas por los programadores para crear el videojuego.

1.1.1.18 Capa de subsistemas específicos del juego.

En esta capa se realizan, en conjunto, las adecuaciones necesarias par crear los objetos del juego. Los diseñadores y programadores tienen como tareas: la creación de los personajes, escenarios y objetos del juego, para obtener los recursos dentro del videojuego.
La principal tarea de esta capa es la carga de recursos, las pruebas de tamaños y animaciones, el flujo de juego, la integración de escenarios, el movimiento de los elementos del juego, y las pruebas sobre los motores de colisiones con la integración de los gráficos finales, para el análisis de animaciones y mecánica de los personajes.

Se han descrito los elementos principales de un motor de juegos. Su arquitectura y capas comprenden la integración lógica de diferentes tecnologías y entidades presentes en una plataforma de desarrollo (por ejemplo, un dispositivo móvil). Esta integración debe hacerse utilizando una tecnología que pueda abarcar los diferentes módulos, y abstraerlos, de manera que el software y el hardware pueden integrar la arquitectura completa buscada. *OpenGL*, en base a sus directivas, es capaz de unir estas funcionalidades, de manera que se pueda crean el sistema completo, integrando capas de bajo nivel (capa de middleware), con capas de alto nivel (interfaz de usuario).

1.2 *OpenGL*[‡]

OpenGL (*Open Graphics Library*) es una interfaz de software para la generación de gráficos por hardware. Esta interfaz consiste en cerca de 150 distintas directivas que se utilizan para especificar los objetos y operaciones que se necesitan para producir objetos en tres o dos dimensiones. *OpenGL* ha sido diseñado como una especificación robusta y compacta, de manera que es independiente del hardware, lo que le permite estar presente en diferentes plataformas, por ello, no existen comandos para interacción de usuario dentro de sus directivas; en vez de eso, se deben utilizar funciones de más alto nivel las cuales están en función del hardware específico donde se esté trabajando. Por ejemplo, no sería lo mismo tratar de tomar las funciones de entrada de datos de un teléfono inteligente, con pantalla táctil, que recabar los datos de una computadora portátil. El mismo hardware es consciente de ello, por lo que tendría que utilizar las funciones y mecanismos que el software que gobierna a estos dispositivos provee.
De igual manera, *OpenGL* no contempla comandos de alto nivel para describir modelos de objetos 3D de manera general, debido a ello los modelos se diseñan como un conjunto de primitivas geométricas (puntos, líneas y polígonos), y mecanismos de enlace que permitan el 'ensamblado' de estas formas geométricas en el modelo final.

[‡] *[5]*, Capítulo 1: *"Introduction to OpenGL"*

En el desarrollo de aplicaciones sobre *OpenGL* se presentan dos vertientes, el enfoque del programador, y el enfoque del arquitecto.

1.2.1 Enfoque de la especificación de *OpenGL*[§]

1.2.1.1 Enfoque del programador

Para un programador, *OpenGL* es un conjunto de comandos que permiten la especificación de objetos geométricos en dos o tres dimensiones, junto a comandos que controlan cómo estos objetos son *renderizados* dentro de un *framebuffer*. En la mayor parte de su arquitectura, *OpenGL* provee una interfaz de *renderizado* inmediato, lo que permite que cualquier objeto que se especifique sea dibujado.
Al realizar un módulo de software sobre *OpenGL*, el desarrollador hace uso de estas directivas para generar los gráficos y lógica de la aplicación utilizando directamente las primitivas. El desarrollador, por la tanto, debe tener conocimiento completo del funcionamiento de los mecanismos de *renderizado*, de manera que todas las modificaciones que se desean realizar sobre los modelos generados (sea tridimensionales o bidimensionales) se realizan utilizando directamente las primitivas.
Este desarrollo tiene grandes ventajas debido a que se explota completamente el poder de la *API*, sin embargo, se limita el desarrollo a expertos en las primitivas de la misma. De igual manera, existen mecanismos de inicialización que resultan repetitivos y no tan accesibles a desarrolladores no tan especializados. La creación de interfaces de alto nivel que puedan disminuir la complejidad del desarrollo resulta de gran utilidad.

1.2.1.2 Enfoque del arquitecto de juegos

El enfoque del diseño del motor de juegos es la conceptualización de las primitivas de *OpenGL* hacia una API entendible para los programadores de videojuegos. Usualmente, el equipo de desarrollo de videojuegos se compone de programadores los cuales cuentan con experiencia en el lenguaje de programación nativo en la plataforma objetivo del desarrollo, así como experiencia en el desarrollo de videojuego.
El objetivo del arquitecto de juegos es desarrollar un conjunto de funciones que, haciendo uso de primitivas de *OpenGL*, mapeen estas funcionalidades a objetos del lenguaje en el que se está desarrollando el videojuego (generalmente un lenguaje de alto nivel), de esta manera, los programadores no requieren ser especialistas en *OpenGL*, sino que tan solo deben ser capaces de manejar el conjunto de funciones antes descritas.

1.2.2 Arquitectura de *OpenGL*[**]

OpenGL es una colección de primitivas que tienen como propósito el acceso a todas las funcionalidades que ofrece el hardware de gráficos. Internamente, actúa como una máquina de estados que instruye al procesador de gráficos (GPU) qué hacer, y qué modificaciones realizar, en una manera definida. Usando el *API*, se pueden modificar varios aspectos de los elementos que se despliegan, incluyendo el color, iluminación, texturas etc. Para entender el concepto de máquina de estados, imaginemos una impresora de serigrafía. Si establecemos la malla con un diseño en particular, con un color definido, todo lo que imprimamos, después de fijar el diseño y color será de las mismas características.
Ahora, si en algún momento cambiamos el color, en otro el diseño, o alternamos, observaremos modificaciones a las impresiones, pero solo ocurrirá al modificar la malla.

[§] *[7]*. Capítulo 1. *"Introduction"*
[**] *[8]*. Capítulo 1. *"The Exploration Begins . . . Again"*

De esta manera actúa la máquina de estados, toma la información presente en sus buffers en el momento del *renderizado*, y realiza el despliegue de los elementos.
En el núcleo de *OpenGL* está el *pipeline de renderizado*, el cual se muestra en la Figura 1.2.

Cada uno de los elementos de la figura mantiene operaciones de generación, y dibujado de imágenes sobre los dispositivos de salida.

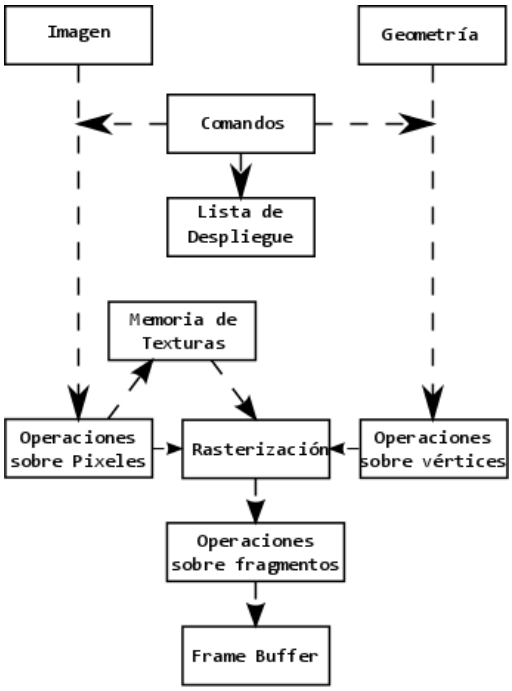


Figura 1.2 Pipeline de *renderizado* en *OpenGL*

Los elementos superiores (Imagen y Geometría) se refieren a estructuras de información definidas en base a las capacidades de los dispositivos, debido a que estos elementos se describen en función de datos externos (por ejemplo, imágenes o recursos que se utilicen como diseños), y las capacidades de los dispositivos para mostrar información (tamaño y funcionalidades de la pantalla de salida).
A estos datos se les aplican comandos (directivas de *OpenGL*) para modificar sus propiedades. En la siguiente capa aparece la rasterización, que básicamente es el despliegue de los modelos 2D o 3D en base a la perspectiva generada por las operaciones sobre vértices o propiedades de forma en base a las operaciones sobre pixeles (en este caso tenemos la generación de texturas, el manejo de iluminación, etcétera). Posteriormente aparecen las operaciones sobre fragmentos, las cuales son las adecuaciones finales, basadas en elementos de iluminación, manejo de cámaras y mapeo de los elementos como parte del ambiente gráfico generado.
A simple vista el manejo de *OpenGL* puede considerarse como complejo y demasiado abstracto, sin embargo, la utilización de una máquina de estados resulta ser la aproximación más eficiente debido a que el despliegue de gráficos se basa en una serie de pasos altamente definidos.

Traslademos la visualización de gráficos al mundo real. El ser humano enfoca la vista hacia un objeto (una equivalencia a la creación de un gráfico). Su percepción del objeto estriba totalmente en sus capacidades (grado de visión). Este objeto será afectado en base a las características del objeto (luminosidad del día, sombras), afectado por el ambiente en el que se encuentra inmerso. De igual manera, se tomará en cuenta la perspectiva en la cual es enfocado el objeto.

Con esta imagen en mente, falta simplemente trasladar este objeto al entorno donde se encuentra. Ahí entraría todo el campo de visión del observador, y el acoplamiento del objeto dentro del entorno global. Es fácil intuir que este proceso es altamente cambiante, los efectos del entorno pueden afectar al objeto en cualquier momento, o el observador puede cambiar la perspectiva del mismo.
Sin embargo, todas las operaciones de modificación de las características del objeto se remiten a las operaciones descritas anteriormente. El proceso es continuo, pero altamente repetitivo.

OpenGL es una tecnología ciertamente pesada, que trata de aprovechar al máximo las características del hardware, explotando al máximo recursos en cuestión de procesamiento y memoria. Aún así, el mercado de los dispositivos móviles (tema de atención del presente trabajo) se encuentra en un nivel diferente al de las computadoras personales, por lo cual fue necesaria la especificación de un conjunto reducido, pero igualmente poderoso de directivas de *OpenGL* para la aceleración de gráficos en dispositivos pequeños. Por ello surgió *OpenGL ES* (*OpenGL for Embedded Systems*).

1.2.3 La especificación de *OpenGL ES[9]*

OpenGL ES es una *API* ligera de bajo nivel para la generación e integración de gráficos avanzados utilizando un subconjunto bien definidos de *OpenGL*. *OpenGL ES* proporciona una interfaz de programación de aplicaciones de bajo nivel (*API*) entre aplicaciones de software y hardware de aceleración de gráficos.

Esta *API* ha sido optimizada, principalmente, para quitar redundancia existente en *OpenGL*, de manera que se garantice el funcionamiento de las directivas en dispositivos que presenten menos capacidades (memoria, procesamiento). Aún así, se pretende que exista un nivel de integración alto entre las aplicaciones que utilicen cualquiera de las dos librerías, de manera que el proceso de migración resulte tan fácil como sea posible. Las ventajas del uso de *OpenGL ES* se presentan a continuación:

Estándar de la industria, y libre de regalías. Cualquier persona puede descargar la especificación *OpenGL ES* y ejecutar o probar los productos basados en *OpenGL ES*.
Con amplio apoyo de la industria, *OpenGL ES* es el único estándar verdaderamente abierto de aceleración de gráficos por hardware para sistemas embebidos. A nivel de aceleración de gráficos, representa el nivel de abstracción más alto para los desarrolladores, independiente de la plataforma.
Tamaño reducido y bajo consumo de energía. Al ser diseñado para sistemas embebidos, se ha tratado de adaptar a ambientes reducidos. *OpenGL ES* está diseñado para poder trabajar con el mínimo espacio de almacenamiento, minimizando el tamaño de las instrucciones, y utilizando un tipo de dato flotante sumamente amigable. Esto resulta en binarios de menor tamaño, que ocupan menos espacio de almacenamiento en un dispositivo.
Transición transparente de aceleración de gráficos por software o hardware. Aunque *OpenGL ES* especifica un pipeline particular de procesamiento de gráficos, algunas llamadas individuales pueden ser ejecutadas en un hardware específico, correr en rutinas de software en el CPU, o utilizar una combinación de hardware-software. Esto significa que los desarrolladores pueden realizar implementaciones enfocadas a la aceleración por software, y dejar que la transición a hardware (por la aparición de nuevos dispositivos) sea de manera transparente.
API extensible y evolucionando constantemente. *OpenGL ES* permite que las innovaciones del hardware nuevo sean accesibles a través de esta API, utilizando el mecanismo de extensión presente en *OpenGL*; todo esto a través de actualizaciones (principalmente generadas por los sistemas operativos de los dispositivos).

Fácil de usar. Ya que *OpenGL ES* es un subconjunto de *OpenGL*, hereda la manera estructurada e intuitiva de diseño y manejo de comandos.
Bien Documentado. Dado que está basado en *OpenGL*, las numerosas fuentes de información existentes acerca de *OpenGL* sirven como referencias para el desarrollo de *OpenGL ES*.

La generación de gráficos y el mantenimiento de los mismos forma una parte importante en el desarrollo de un motor de juegos; sin embargo, las interacciones con las entradas de usuario, y la plataforma específica, no están incluidas dentro de las directivas de *OpenGL* (ver sección 1.2). Para lograr la interacción con el usuario a través de las entradas de datos del dispositivo, es necesario comunicarse con el sistema operativo presente en cada dispositivo, de manera que el desarrollo del motor incluya la lógica de bajo nivel, y la lógica a nivel sistema operativo.
El sistema operativo *iOS* (sistema operativo seleccionado para el desarrollo del motor de juegos), cuenta con una serie de herramientas de desarrollo de aplicaciones de manera nativa. Estas herramientas permiten hacer uso de diferentes *frameworks*, que permiten el acceso a diversas funcionalidades propias del dispositivo (por ejemplo, el uso de bluetooth, la cámara, etcétera). De esta manera, se facilita el desarrollo de aplicaciones que exploten las capacidades de los dispositivos.

1.3 El Sistema Operativo *iOS [10]*

La plataforma de *iOS* está basada en un modelo tipo propietario, teniendo como entorno de desarrollo una computadora Mac bajo el sistema Operativo Mac OSX (con la versión estable más actual, Mavericks), utilizando el *IDE XCode*. Integrado a *XCode*, proporciona simuladores basados en los dispositivos *iOS* más avanzados (iPhones y iPads). Todo este entorno tiene como lenguaje de programación *Objective-C*, lenguaje de alto nivel utilizado para desarrollar aplicaciones de manera nativa. Aún cuando se proporcionan simuladores que permiten la realización de pruebas, se permite ejecutar las aplicaciones en dispositivos reales enrolándose al sitio de desarrolladores de Apple; esto se consigue pagando una cuota anual de US $99. Dentro de este programa, se obtiene una entrada en el portal iTunes Store, que permite vender las aplicaciones hacia usuarios finales (cabe destacar que las universidades pueden aplicar para obtener una cuenta de manera gratuita). A diferencia de *Android*, las aplicaciones pasan por un riguroso proceso de revisión, en el cual se verifican las políticas de publicación que Apple ha establecido como estándares de calidad en sus aplicaciones.
El sistema operativo *iOS* (que cuenta con una arquitectura similar al OS X), corre sobre cuatro capas: Core OS/Kernel, Core Services, Media Support y la interfaz de *Cocoa Touch*. Este diseño por capas, permite al usuario adentrarse tanto como sea necesario para el desarrollo de aplicaciones, pero contando con interfaces intermedias que le permitan generar los módulos de la aplicación en base a la capa en la cual se encuentre. Por ejemplo, es posible obtener la lista de contactos de un dispositivo, a través de *Cocoa Touch*, como una tabla presente en una vista, sin embargo, haciendo uso de *Core Services*, se podría obtener una colección que permita la manipulación de cada uno de los campos presentes en cada entrada de la misma.

Una característica importante del sistema operativo *iOS* es la integración de software-hardware presente en estos dispositivos, de manera que se garantiza la funcionalidad de las aplicaciones en todos y cada uno de los dispositivos que soporten *iOS*. La completa integración de *iOS* con el hardware en el que se encuentra inmerso, permite asegurar que el software desarrollado sobre una versión específica del sistema operativo (*iOS* 7.x en su versión comercial estable) sea completamente funcional en cualquier dispositivo que cuente con dicha versión del sistema operativo. A diferencia de *Android*, *iOS* se enfrenta a un nivel de fragmentación (en base a hardware) de muy bajo impacto.

1.3.1 Requerimientos de hardware para desarrollar en *iOS[11]*

Para el desarrollo de aplicaciones sobre *iOS* es necesario el uso del *SDK* de *iOS* y el *IDE XCode*. *XCode* provee todo lo necesario para desarrollar aplicaciones sobre *iOS*; incluye un editor de código, un editor de interfaz gráfica, herramientas de compilación y depuración, etcétera; también contiene herramientas, compiladores y *frameworks* necesarios específicamente para *iOS* (de manera similar, existen herramientas símiles para desarrollar sobre OSX).
El lenguaje de desarrollo, como se mencionó en la sección anterior, es *Objective-C*. La estructura de programación sobre *iOS* se basa en el *framework Cocoa Touch*, el cual es un conjunto de *frameworks* que exponen, a manera de *APIS*, los diferentes componentes que el usuario puede utilizar para interactuar con el sistema operativo. *Cocoa Touch* puede ser visto como un enlace entre todas las funcionalidades que puede hacer uso el programador y el sistema operativo. Un ejemplo claro es la utilización de un *framework* de alto nivel para el uso del *bluetooth* del dispositivo, haciendo uso de este *framework* el usuario no necesita llamar funciones de bajo nivel para interactuar con él, sin tener que realizar módulos de bajo nivel para manejar los flujos de entrada salida de datos, sincronización, etcétera. Este esquema de programación ha sido criticado duramente por un una facción de desarrolladores, la cual confronta a *Apple* acerca de la poca flexibilidad al intentar utilizar funciones avanzadas del sistema operativo; sin embargo, existe una contraparte que se muestra cómoda dejando a un lado detalles de implementación de bajo nivel para centrarse en el desarrollo de la lógica específica de la aplicación.
La utilización de un lenguaje de programación como *Objective-C*, descrito anteriormente como un *superconjunto* de *C* y el lenguaje de desarrollo para aplicaciones *iOS*, permite una mayor adaptabilidad del manejo de directivas de *OpenGL* (escritas en *C++*). El *IDE XCode* es capaz de reconocer instrucciones de ambos lenguajes de programación en tiempo de compilación y depuración, dado que el compilador *GCC* es capaz de compilar ambos lenguajes. Esto reducirá la complejidad de integrar ambos entornos (*Cocoa Touch* y *OpenGL*).

1.3.2 *OpenGL ES* en *Cocoa Touch[12]*

Debido a que *OpenGL ES* es una *API* basada en C, es extremadamente portátil y ampliamente apoyada además de integrarse a la perfección con el lenguaje *Objective-C*, utilizado para desarrollar aplicaciones nativas para *iOS*.
Esto es de gran ayuda dado que, como se ha comentado en secciones anteriores, la especificación *OpenGL ES* no define una capa de ventanas, sino que el sistema operativo anfitrión debe proporcionar funciones para crear un contexto de *renderizado OpenGL ES* que acepte comandos y un *framebuffer*, donde los resultados de los comandos de generación de gráficos se escriben.
Trabajar con *OpenGL ES* en *iOS* requiere el uso de clases y *frameworks* especiales para crear y presentar una superficie de dibujo y el uso de la API de plataforma neutral para representar su contenido. Todo esto se realiza integrando dentro de la aplicación el *Framework OpenGL ES*, necesario para hacer uso de las directivas desde la capa *CocoaTouch* (Ver Figura 1.3).

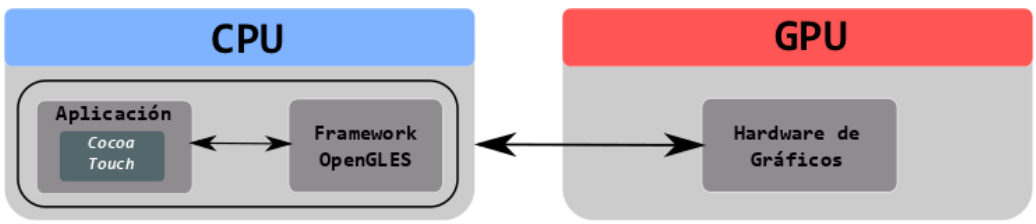


Figura 1.3 Interacción de la aplicación con *OpenGL ES*.

Teniendo en cuenta esta integración, el acceso a directivas de *OpenGL ES* se realizan enlazando el *Framework* correspondiente y teniendo en consideración que la superficie *OpenGL ES* se encuentra inmersa dentro de la vista principal de la aplicación. (Figura 1.4)

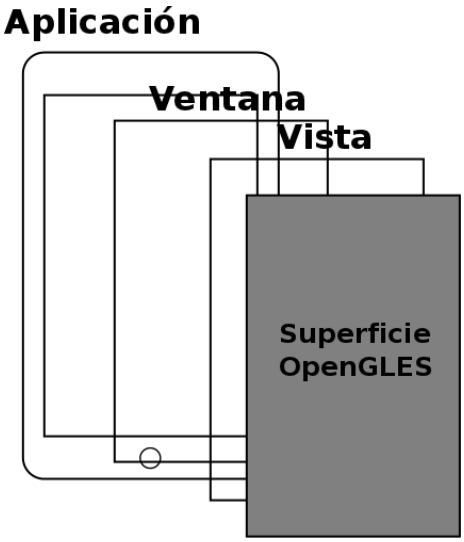


Figura 1.4 Jerarquía de vistas en una aplicación *iOS*.

Definido este enfoque, resulta clara la complejidad del desarrollo a este nivel de abstracción, dado que cualquier elemento que se despliegue en la pantalla del dispositivo, en una aplicación *OpenGL ES*, corre por cuenta del desarrollador.

1.4 Planteamiento del Problema

Como se ha descrito en párrafos anteriores, el desarrollo de un motor de juegos permite a los programadores concentrarse en la lógica del juego, dejando a un lado detalles de implementación de bajo nivel. El problema principal se centra en la elaboración de dicho módulo de software para el sistema operativo *iOS*.
Actualmente, existen varios productos destinados a esta tarea, la mayoría de los cuales son de tipo propietario, por lo cual es necesario la adquisición de derechos de las aplicaciones para poder realizar desarrollos sobre de ellos.
Por cuestión de practicidad (en la obtención y generación de los recursos), los motores existentes centran su atención en el desarrollo de videojuegos en 2D, imposibilitando el uso de modelos 3D dentro de los proyectos de software. Finalmente, los motores de juego actuales de mayor uso corren y trabajan sobre *frameworks* que se encuentran sobre la capa de lenguaje nativo de las plataformas sobre las que operan, restringiendo al desarrollador la depuración de los errores.
Aunado a estos problemas, el desarrollo de motores de juegos en tres dimensiones sobre plataformas móviles, es un campo poco explorado actualmente. Existen motores 3D complejos como *Unity[13]*, los cuales son costosos, y cuentan con *IDE*s independientes de la plataforma, lo que implica al desarrollador el uso de herramientas nuevas para el desarrollo. Existen también *frameworks* 3D como *Cocos3D[14]*, el cual es software de código abierto, pero no se encuentran bien documentado. El principal problema de desarrollar un videojuego 3D actualmente, es escoger un motor de juegos 3D que resulte conveniente para el desarrollo del mismo.

1.5 Objetivo

Diseñar e implementar un motor de juegos 3D, que permita el desarrollo de aplicaciones sobre el sistema operativo *iOS (versión 7. x)*.

1.5.1 Objetivos Particulares

1. Implementar la arquitectura de un motor de juegos 3D reducido y estable[††].
2. Diseñar e implementar un mecanismo de *renderizado* en base a modelos tridimensionales independiente del motor que pueda ser escalable y optimizado.
3. Definir, crear y documentar una serie de funciones y módulos de software que permitan el mantenimiento y puesta en marcha de la máquina de estados de *OpenGL*, definidos sobre el lenguaje *Objective-C*.
4. Crear una capa Middleware, que permita la exposición de funciones de *renderizado* de gráficos, basadas en primitivas de *OpenGL*, hacia el lenguaje *Objective-C*.
5. Crear una aplicación sencilla utilizando el motor de juegos desarrollado, con el fin de mostrar las funcionalidades del mismo.

1.6 Justificación

Actualmente el desarrollo de videojuegos ha entrado en un auge global. El fácil acceso a las computadoras personales, y la reducción de costos en las consola de videojuegos, ha fomentado que esta industria crezca de manera exponencial. Hace apenas un par de décadas, era poco común poseer una consola de última generación, hoy en día es un sistema de entretenimiento que pocas veces falta en cualquier hogar. A la par de este crecimiento, el surgimiento y rápida evolución de los teléfonos inteligentes acompañó a esta tendencia.
Los tiempos donde sólo altos ejecutivos tenían acceso a telefonía móvil ha quedado atrás. Hoy en día, es común que hasta niños pequeños cuenten con estos dispositivos, y sean capaces de utilizarlos de manera adecuada. Este tema ha resultado de interés comercial, pero también dentro de la comunidad científica. Por ejemplo, la Universidad de Denver*[15]* ha lanzado programas de licenciatura relacionados totalmente con videojuegos, particularmente los programas *Bachelor of Science in Animation and Game Development* y *Bachelor of Art in Animation and Game Development*. A continuación mostraremos un extracto de los principios de su programa de estudios:

"Why Study Game Development?...

The obvious reason to study Game Development: game programming and art/design are challenging, satisfying, and a lot of fun...

... The degree also has a broad applicability. Although the unifying is the creation of games, all the skills are directly applicable to the much broader fields as digital entertainment, computer science, and interactive training and simulation. Interactive simulation applications include scientific exploration, health sciences, general education, law enforcement, defense, justice, cable/entertainment, and security industries. Many industries and agencies are becoming more dependent on interactive computer training and simulation. We expect this trend to continue into the next few decades and hence the ability to apply this degree to many jobs and graduate programs will continue to grow."

[††] *En la sección 1.7, se describirán los módulos que esta arquitectura incluirá, así como la definición del término reducido, dentro de esta descripción.*

Poco a poco la comunidad científica cuenta el desarrollo de videojuegos como un campo de las ciencias de la computación altamente importante, dado el campo de proyección que tienen las habilidades que implica este conocimiento.

1.6.1 ¿Por qué desarrollar un nuevo Motor de Juegos 3D?

El desarrollo de un motor de juegos 3D descrito en este documento, pretende tomar su justificación en diferentes ámbitos. El primordial es el desarrollar un motor de juegos 3D totalmente libre y enfocado completamente a modelos tridimensionales, así como la posibilidad de modificar o interactuar con los *shaders*[‡‡] utilizados para el *renderizado* de los modelos en el formato standard *obj*[§§], una característica que no existe en los motores de juegos actuales. Los modelos bajo el estándar utilizado (obj) resultan ser los modelos más descriptivos y soportados por un amplio grupos de editores en 3 dimensiones libres.
Un punto adicional es el interés de desarrollar tecnología de punta. Al plantear este proyecto, se pretende desarrollar software de vanguardia, y vigente que pueda competir con los desarrollos de empresas u otros centros de investigación. El iniciar el desarrollo de un motor de juegos implica sentar las bases para el perfeccionamiento del mismo, de manera que, en un futuro, éste pudiera salir a la luz como un producto de software lo suficientemente robusto y estable, que permita a los desarrolladores tomarlo como base para el desarrollo de videojuegos.

Una de las contribuciones principales de esta tesis es la definición y exposición de la arquitectura de un motor de juegos 3D, así como de los mecanismos de renderizado de los modelos. Esta contribución toma fuerza dado que, aun cuando se encuentran soluciones en el mercado (tanto propietarias como de código abierto), ninguna expone su arquitectura funcional o la lógica utilizada por sus mecanismos de renderizado. Se propone el detallar al máximo las funcionalidades, interacciones y módulos que integran el sistema, de manera que este trabajo sirva como base para futuros desarrollos, no limitado a motores de juegos, sino también como visualizadores de gráficos en tres dimensiones.

1.6.2 ¿Por qué *iOS* y no *Android*?

La motivación para seleccionar el sistema operativo *iOS*, es el impacto comercial que tiene el desarrollo de videojuegos en dicha plataforma. La revista *Wired*, en su edición de febrero de 2013, presenta el artículo "*Why Game Creators Prefer iPhone to Android"[16]*. Si bien describe que *Android* ha abarcado parte del mercado de *iOS*, expresa que los desarrollos para *Android* pueden considerarse como adaptaciones de aplicaciones de *iOS*. Tomando como ejemplo el éxito de algunos videojuegos, el número de descargas de las aplicaciones en ambas plataformas es totalmente dispar. Otro elemento importante es la necesidad de manejar código en el lenguaje *C++*, para integrar *OpenGLES* a la aplicación móvil. El *IDE XCode* utiliza el compilador *gcc*, el cual tiene el poder de compilar código en *C-Objetivo* de la misma manera que el generado en el lenguaje *C*, de manera que la integración es transparente. Como último punto tenemos la fragmentación de los dispositivos *Android*, la cual presenta un nivel de abstracción mayor en un primer acercamiento al desarrollo de un motor de juegos. Por conveniencia y practicidad, este trabajo requiere acotar estos inconvenientes, pero sin restarle generalidad y poder. Por último, escoger el dispositivo con sistema operativo *iOS* más avanzado (actualmente el *iPhone 5*), resulta lo más conveniente desde el punto de vista comercial.

---

‡‡ *En la sección 3.5.5, relativa al diseño, se definirán las funciones y tareas de los shaders dentro de los mecanismos de renderizado del motor de juegos.*
§§ *El formato de archivo obj se definirá en secciones posteriores del Capítulo 4*

1.6.3 ¿Por qué *OpenGL*?

La utilización de *OpenGL ES* se basa principalmente en la presencia de esta tecnología en el hardware elegido. De igual manera, dado que el desarrollo sobre *Cocoa Touch* se realiza programando en el lenguaje *Objective C*, (super-extensión del lenguaje *C*), la utilización de las directivas de *OpenGL* se realiza de manera transparente. Aun así, es necesario hacer énfasis en que se utilizará *OpenGL ES*, la cual es simplemente una librería compacta y de menor consumo de potencia y procesamiento que *OpenGL*.

1.7 Alcances y Limitaciones del Proyecto

Como se ha descrito en las secciones anteriores, el objeto de este documento es el desarrollo de un motor de juegos 3D. En la sección 1.1.1, se describe la arquitectura de un motor de juegos. Estos elementos son los mínimos para considerar un motor de juegos completo y estable. Sin embargo, durante el desarrollo del presente trabajo, los elementos se reducirán, a fin de poder concretar, en tiempo y forma, el diseño e implementación de un motor de juegos funcional. Esta reducción de capas describe la característica de reducido, expuesta en los objetivos de este trabajo.
La Figura 1.5 muestra los módulos que el motor de juegos generado en el presente documento considerará, así como su función dentro del mismo.

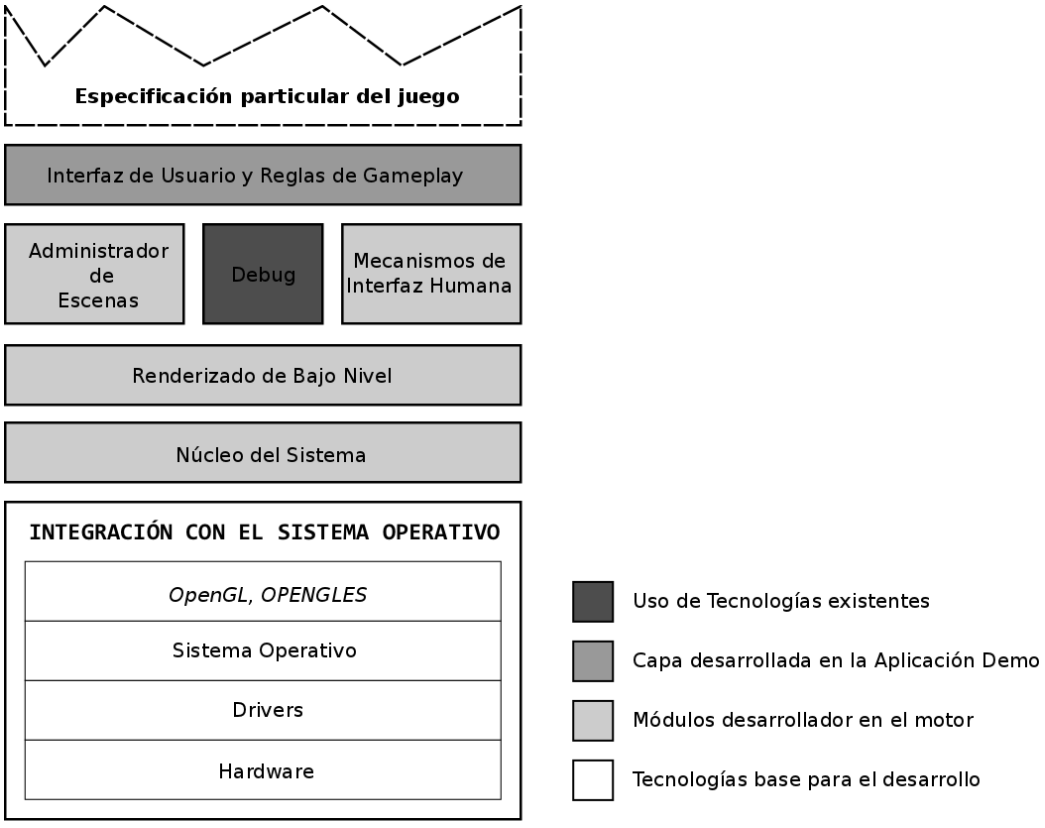


Figura 1.5 Elementos del motor de juegos desarrollado en este trabajo.

La capa más baja describe los elementos de la plataforma donde se desarrollará el sistema. Esta capa implica las funciones, librerías y herramientas de código que se utilizarán como bases para la construcción del motor de juegos.

Por ello, esta capa representa el entorno de desarrollo y las características de los dispositivos utilizados para el desarrollo del presente trabajo. En este nivel, simplemente se utilizarán las tecnologías presentes en el dispositivo de desarrollo.
La siguiente capa, de núcleo del sistema, se enfoca a la creación, puesta en marcha y mantenimiento de la máquina de estados de *OpenGL*.
La capa de *renderizado* de bajo nivel es el puente existente entre las funciones de bajo nivel (C++, directivas de *OpenGL*), y funciones existentes en el motor de juegos.
Utilizando funciones definidas en *Objective C*, el programador podrá tener acceso a funciones de bajo nivel; las cuales pueden ser: creación y modificación de gráficos, utilización de primitivas, o modificaciones sobre el estado de la máquina de estados de *OpenGL*.
El administrador de escenas, es, básicamente, una entidad capaz de manejar transiciones entre escenas de juego, esto con la finalidad de manejar la creación y destrucción de las mismas.
En el caso de los mecanismos de interfaz humana, nos referimos a la herramienta que permitirá la interacción con los dispositivos de entrada, de manera que la interacción con los usuarios esté garantizada a través de los mecanismos existentes (en el caso de los teléfonos inteligentes, la pantalla táctil, el acelerómetro, etc.).
La capa de depuración, aun cuando no será desarrollada en el presente trabajo de tesis, es incluida debido a la existencia de una herramienta que realiza estas funciones: la aplicación *Instruments[17]*. Esta herramienta es utilizada para la depuración de aplicaciones en *iOS*, con funciones de análisis de procesamiento, uso de memoria y rendimiento.
La capa del nivel más alto, es la exposición final de las funciones desarrolladas. Desde ella, el programador del videojuego hará uso de las funciones necesarias que utilizará. En ella también se exponen objetos necesarios para crear la interfaz de usuario, tales como botones, modelos, menús, etcétera.
Es importante acotar que el motor carecerá de un motor de física y colisiones. Los métodos y mecanismos, o la integración con alguna librería que proporcione estos servicios, quedan como trabajo a futuro.

1.8 Organización de la Tesis

1.8.1 Descripción de los Capítulos

Capítulo 1. En este capítulo se describen los aspectos introductorios al trabajo de tesis aquí presentado. Se describe un panorama de las tecnologías a utilizar, así como las herramientas de las cuales se hará uso. También se muestran las motivaciones, objetivos, limitantes y alcances de este trabajo.

Capítulo 2. En este capítulo se describe el escenario actual en cuanto a los sistemas existentes actualmente de los motores de juegos. Se realiza un análisis de las tecnologías vigentes, mostrando sus bondades y limitaciones a manera de preámbulo, para describir las ventajas y funcionalidades adicionales que incluirá el motor de juegos propuesto.

Capítulo 3. En este capítulo se discute el diseño del sistema. Mediante el uso de diagramas de clase y componentes, se describen las interacciones entre los diferentes módulos a desarrollar.

Capítulo 4. En este capítulo se describe el proceso de desarrollo del motor de juegos. Se muestran las entidades de software generadas, así como su integración dentro del sistema global, de manera que se expongan las funcionalidades principales de cada uno de estos módulos, así como el funcionamiento del proyecto de manera global.

Capítulo 5. En esta capítulo se exponen los resultados del desarrollo. Se realizan pruebas de desempeño, así como análisis de los productos generados. Se demuestra, mediante el análisis de la aplicación muestra creada en base al motor de juegos, el funcionamiento del sistema en un ambiente real de desarrollo.

Capítulo 6. En este capítulo se presentan las conclusiones acerca del sistema desarrollado, en base a los resultados obtenidos. También se indica los de trabajos a futuro, o mejoras al sistema, así como las aportaciones de la tesis.

Por último, se incluyen las referencias en el desarrollo del presente trabajo, así como la sección de anexos.

Anexos

A. Manual de operación del motor de juegos.
B. Coódigo fuente del motor de juegos.

Capítulo 2

ESTADO DEL ARTE

El desarrollo de videojuegos es una industria de miles de millones de dólares. Y los actores que ahí intervienen son: desde desarrolladores independientes, hasta grandes empresas con varias decenas de trabajadores. Aun cuando es tan heterogéneo este mercado, una de las principales preocupaciones de los desarrolladores (sea a gran o pequeña escala), es el de escoger el mejor motor de juegos, y todas las herramientas que lo acompañan *[18]*.
Actualmente existen varias soluciones. El primer filtro es seleccionar la tecnología a utilizar, así como la plataforma en la cual se realizará el desarrollo. En el caso particular de los dispositivos *iOS*, existen varios motores comerciales actualmente. Por un lado, tenemos los motores de juegos multiplataforma, los cuales utilizan como herramientas de desarrollo lenguajes de scripting, lo que permite el traslado de estas aplicaciones a diferentes equipos y sistemas operativos. Estos motores tienen la particularidad de utilizar lenguajes de programación diferentes a los lenguajes nativos de los sistemas operativos, *Lua* o *JavaScript* son ejemplo de ellos. Por otro lado existen los motores de juego particulares a una sola plataforma. En ellos, el desarrollo se realiza en el lenguaje de programación que utilizan los sistemas operativos como nativo, por lo que los desarrolladores se encuentran en un ambiente más cómodo, con la desventaja que el desarrollo solo estará enfocado en la plataforma seleccionada. Como se observa, existen ventajas y desventajas de las soluciones actuales, en este capítulo realizaremos un pequeño análisis de las tecnologías vigentes, con el fin de evaluar su funcionalidad[***].

2.1 Motores de Juegos de Plataforma Específica

Este tipo de motores de juegos, como su nombre lo indica, tiene la particularidad de exponer las funcionalidades gráficas de las plataformas donde corren, de manera que sean accesibles a los desarrolladores. En el caso del desarrollo sobre *iOS*, los motores de plataforma específica deben corresponder con el lenguaje de programación *Objective C*. Dentro de esta misma clasificación, entontraremos motores de licencia propietaria, y motores de licencia libre.

2.1.1 Motores Propietarios

Debido a que actualmente las empresas enfocan sus desarrollos a múltiples plataformas, existen pocos desarrollos de motores de juegos de plataforma específica (para *iOS,* particularmente) bajo esquema propietario. Esto se explica dado que las empresas dedicadas a desarrollar los motores de juegos y venderlos, ofrecen paquetes de integración a múltiples plataformas..

2.1.2 Motores de Uso Libre

El exponente más importante de los motores de juegos libres 3D es *Cocos3D[14]*. *Cocos3D* puede considerarse una extensión de el motor *Cocos2D*, estandarte del desarrollo de juegos de manera independiente en la plataforma *iOS*.

[***] *Debido al enfoque definido en este trabajo, el enfoque de análisis se realizará totalmente sobre las tecnologías de motores de juegos 3D, dejando a un lado tecnologías tales como 2 o 2.5 D.*

Cocos3D (*C3D*) utiliza la base de construcción de *Cocos2D* (*C2D*), de manera que la base inicial de ambos motores es la misma. Desde el levantamiento de la máquina de estados de *OpenGL* hasta el manejo de escenas y nodos, *C3D* comparte un gran número de similitudes con *C2D*, simplemente agrega la funcionalidad de los elementos *3D* a las escenas de la aplicación. (Ver Figura 2.1).

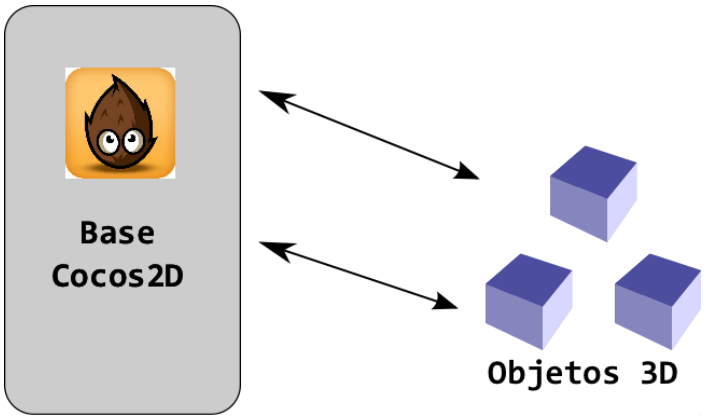


Figura 2.1 *Cocos3D* como extensión de *Cocos2D*.

Cocos3D agrega funciones de modelado *3D*, incluyendo modelos, perspectiva y proyección de cámaras, materiales e iluminación. Con esto, se pueden poblar escenarios *3D* exportados de editores como *Blender*, *3ds Max* o *Cheetah3D*, de manera que se pueda hacer uso de estos modelos como objetos dentro del mundo del juego.

La principal desventaja de *Cocos3D* es la falta de información acerca del funcionamiento del mismo. De igual manera, al ser una 'extensión' de *Cocos2D*, *Cocos3D* contiene adaptaciones de funcionalidad 2D hacia el mundo de tres dimensiones, de manera que estas adaptaciones no fueron desarrolladas hacia el modelado 3D.

2.2 Motores de Juego de Multiplataforma

2.2.1 Motores Propietarios

Los motores de juegos 3D multiplataforma actuales, debido a su complejidad y robustez, son desarrollados por empresas dedicadas al negocio de los videojuegos. En el ámbito de los motores de juegos propietarios, tenemos a *Unity[16]*.

La empresa *Unity Technologies*, propietaria de *Unity* lo describe como:
"... *Unity* es un entorno de desarrollo; un poderoso motor de *renderizado* integrado con un conjunto de herramientas y funciones que permiten la creación de contenido *3D*; publicación fácil en múltiples plataformas, posibilidad de comprar recursos en el *Asset Store* y una comunidad de soporte y mantenimiento enorme."*[13]*
Unity es un motor de juegos de gran aceptación, dado que su entorno permite la generación de aplicaciones para sistemas *iOS*, *Android*, *MacOSX*, *Windows* y *Linux*, así como aplicaciones para consolas de juegos tales como *PS3*, *Wii* y *Xbox*. Como bien se describe, el software *Unity* ofrece la posibilidad de crear juegos *3D* haciendo uso de *C#* o *JavaScript* para la programación de la lógica de juego. Esta solución resulta la más conveniente, dado que estos lenguajes permiten la estructuración de las instrucciones de control, y manejo de variables, de manera 'anónima', de forma que la reconstrucción de las mismas resulta mas fácil.

De igual manera, cuenta con un IDE de desarrollo que se puede descargar de su página web. (Ver Figura 2.2)

Figura 2.2 ***Entorno de desarrollo de Unity 3D.***

Debido a que el manejo de gráficos es realizado por el acelerador gráfico presente en el dispositivo (independientemente de cual sea éste), no es de extrañar que estas funcionalidades se reduzcan, como en motores convencionales, a instrucciones *OpenGL*. Simplemente se genera una capa de complejidad mayor, sobre la cual se realizan adaptaciones para cualquier plataforma. Desde un punto de vista más general, un motor de juegos multiplataforma, es una simple extensión de un motor de juegos convencional.
La programación en *Unity* se realiza a través de un lenguaje de *scripting* a través de *Mono*. Mono es una implementación de código abierto de *.NET Framework*. Los programadores pueden utilizar *UnityScript* (un lenguaje personalizado inspirado en la sintaxis *ECMAScript*), *C#* o *Boo* (que tiene una sintaxis inspirada en *Python*). A partir de la versión 3.0 añade una versión personalizada de *MonoDevelop* para la depuración de scripts.
Unity también incluye *Unity Asset Server* - una solución de control de versiones para todos los *Assets* de juego y scripts, utilizando *PostgreSQL* como *backend*; un sistema de audio construido con la biblioteca *FMOD*, con capacidad para reproducir audio comprimido *Ogg Vorbis*; reproducción de vídeo con códec *Theora*; determinación de cara oculta con *Umbra*, una función de iluminación *lightmapping* y global con *Beast*; redes multijugador *RakNet*, entre muchas más características.
La mayor debilidad de *Unity* es la escasa integración con los entornos de desarrollo específicos de cada uno de los dispositivos sobre los cuales se puede desarrollar. Esta característica delimita las capacidad del desarrollador para utilizar particularidades de los sistemas operativos. Su principal ventaja, es su robustez, así como la gran cantidad de información presente en sus foros y página web.

2.2.2 Motores de Uso Libre

Shiva3D[19] es un motor de juegos que permite el desarrollo de aplicaciones para *iOS*, *Android*, *Windows Phone*, BlackBerry, entre otros. Aún cuando cuenta con un entorno de desarrollo propio, a diferencia de *Unity*, permite la exportación de proyectos hacia los entornos de desarrollo específicos de cada plataforma (*Eclipse*, *XCode*).

En el lado de *iOS*, esto resulta una gran ventaja, dado que puede iniciarse el desarrollo en el editor de Shiva (escenas, lógica de juego principal), y después continuar el desarrollo del proyecto en *XCode*. (Ver Figura 2.3)

Figura 2.3 *Editor Shiva3D*.

Una de las fortalezas de utilizar *Shiva3D* es el hecho de que se debe pagar la licencia de desarrollo hasta el momento de publicar la aplicación hacia el *Store*. Antes de esto, está permitido el manejo y prueba de todas las capacidades del entorno de desarrollo y del *framework* en sí. Otro aspecto importante de *Shiva3D*, es su *Authoring Tool*. Esta herramienta posiciona a Shiva como la herramienta de desarrollo completamente multiplataforma. La *Authoring Tool* compila el proyecto generado por *Shiva3D* en un ejecutable para las siguientes plataformas: *Windows*, *Linux*, *Mac*, *iOS*, *iPad*, *Android*, *Palm* y *Wii*.
Como un común denominador de los motores multiplataforma, *Shiva3D* utiliza un lenguaje de scripting para el desarrollo de las aplicaciones; en su caso, utiliza *lua*. Después de construir el programa, los scripts se traducen a código *C++*, el cual es entendido por las herramientas de aceleración de gráficos por hardware.
Al igual que *Unity*, *Shiva3D* cuenta con una amplia variedad de foros de ayuda y una comunidad extensa de desarrolladores. También cuenta con recursos como tutoriales, videos y código de muestra para probar las capacidades del entorno.

El mayor inconveniente de *Shiva3D*, es que su editor solo existe para la el sistema operativo *Windows*, lo cual limita al desarrollo del proyecto (o la generación del esqueleto del mismo, como mínimo), a entornos con sistema operativo *Windows*.

2.3 Motor de Juegos propuesto

Como se mencionó anteriormente, el objetivo primordial de este trabajo es el desarrollo de un motor de juegos 3D para el sistema operativo *iOS*. La intención principal es subsanar algunas deficiencias que presentan los motores de juegos presentados en la sección anterior.
El motor de juegos que presenta este trabajo tiene como objetivo la integración total con el entorno de desarrollo *XCode*, de manera que el desarrollo de las aplicaciones se realice dentro de las normativas de desarrollo de aplicaciones establecidas por Apple*[20]*.
De esta manera se garantiza que no existirán conflictos en el proceso de publicación de las aplicaciones en el Apple Store; de igual manera, al estar integrada en el entorno de desarrollo *XCode*, las instrucciones desarrolladas en el motor de juegos propuesto serán expuestas en el lenguaje de desarrollo de aplicaciones nativas para *iOS*, el lenguaje *Objective-C*.

Debido a la presencia de *OpenGL ES* como librerías de aceleración de gráficos por software en los dispositivos *iOS*, la exposición de las funciones de *renderizado* se realizarán a través de estas directivas, de manera que la compatibilidad con todos los dispositivos Apple se garantiza. Es necesario acotar que se trabajará con la versión 2.0 y posteriores, de manera que dispositivos que trabajen con la versión 1.x de *OpenGL* quedan fuera del alcance de este sistema.

Se propone realizar una entidad de control, que permita la interacción entre los módulos que integran el motor. Esta entidad se plantea en base al patrón *Singleton[21]*, que permita la escalabilidad del desarrollo de los módulos en base a las funciones que se requieran en cada fase del desarrollo. El establecimiento de las directrices entre módulos y objetos necesarios para controlar el *renderizado* de gráficos y el mantenimiento de la máquina de estados de *OpenGL* se describirá en secciones posteriores, y se documentará completamente, con el fin de lograr el entendimiento de estas interacciones por parte del usuario final.

Como parte fundamental del motor de juegos, se realizará un componente capaz de generar y cargar modelos de manera fácil e intuitiva, de manera que se reduzca la complejidad en el manejo de modelos en tres dimensiones. Este desarrollo presentará una ventaja respecto a los motores de juegos anteriormente descritos, los cuales exigen del usuario un entendimiento intermedio-avanzado de la programación 3D.

Otro elemento importante es el enfoque de desarrollo de videojuegos al que este motor de juegos está destinado. Se pretende la realización de un motor de juegos simple, como primer acercamiento al desarrollo de juegos 3D, de manera que cualquier desarrollador pueda tomar este sistema como base hacia el desarrollo de videojuegos más complejos. Por lo tanto, el presente trabajo tiene como objetivo crear un motor de juegos que sea lo suficientemente estable para resultar funcional, pero que no exceda en complejidad el desarrollo de videojuegos a motores equivalentes en dos dimensiones.

Resumen

En este capítulo se han presentado las alternativas actuales al desarrollo de videojuegos. Se han descrito sus características, fortalezas y debilidades, brindando un panorama acerca de la actualidad de los motores de juegos. De igual manera se describe la propuesta que describe el presente trabajo, de manera que se pueda contextualizar, de una forma más objetiva, el desarrollo aquí presentado. El siguiente capítulo pretende mostrar las herramientas de desarrollo, arquitecturas y patrones de diseño, así como las propuestas de diseño del sistema.

Capítulo 3

DISEÑO DEL MOTOR DE JUEGOS

El desarrollo del motor de juegos incluye la definición, conceptualización y desarrollo de diferentes módulos que, de manera conjunta, puedan realizar todas las funciones que se esperan del motor. Desde los objetos inicialización de la superficie *OpenGL ES* hasta el manejador de eventos en tiempo de ejecución, cada uno de estos módulos necesitan ser entendidos y caracterizados para garantizar el funcionamiento del motor de juegos. De igual manera, es necesario describir una entidad de enlace, que sirva para comunicar estos módulos, y permita la compartición de recursos y objetos necesarios durante la ejecución de la aplicación. El presente capítulo tiene como objeto el describir los módulos antes mencionados, sus interacciones y características. De igual manera, se especifica el entorno en el cual se desarrolla el proyecto, así como las características de hardware y software requeridas para el desarrollo del mismo.

3.1 Requerimientos del Motor de Juegos

Como se ha comentado anteriormente, el presente trabajo tiene como objetivo el desarrollo de un motor de juegos sobre el sistema operativo *iOS*. El entorno necesario para el funcionamiento correcto del sistema, supone el acceso al entorno de desarrollo *XCode*, así como los *SDK* necesarios para completar el entorno de desarrollo completo. Las especificaciones de la plataforma de desarrollo necesarias se presentan a continuación:

- **Equipo** — Computadora Apple Mac.
- **Sistema Operativo de desarrollo** — OS X Mavericks (versión 10.9).
- **Entorno de Desarrollo** — *XCode* 5.

3.2 Inicialización de una aplicación *iOS*.

Durante el proceso de diseño, es necesario entender las fases por las cuales una aplicación *iOS* se inicializa. Esto debido a que es necesario establecer el punto de control a partir del cual el desarrollador puede determinar el comportamiento de la aplicación. De acuerdo con la especificación del ciclo de vida de una aplicación *iOS[22]*, se presenta la secuencia mostrada en la Figura 3.1.

A. El primer paso es la creación de un objeto aplicación(paso 3 en el flujo) . Por omisión, este proceso crea una instancia de **`UIApplication`**.
B. La función busca el nombre de la clase encargada de fungir como delegado de la aplicación. Crea una instancia de esta clase y la asigna como el *delegado*. Por omisión, el delegado de la aplicación es creado dentro del método **`main`** de la aplicación.

   Al crear un proyecto base en *iOS*, se genera una clase con el nombre: **`AplicacionAppDelegate`**. Esta clase es una instancia del objeto **`UIApplicationDelegate`** (Estado 6).

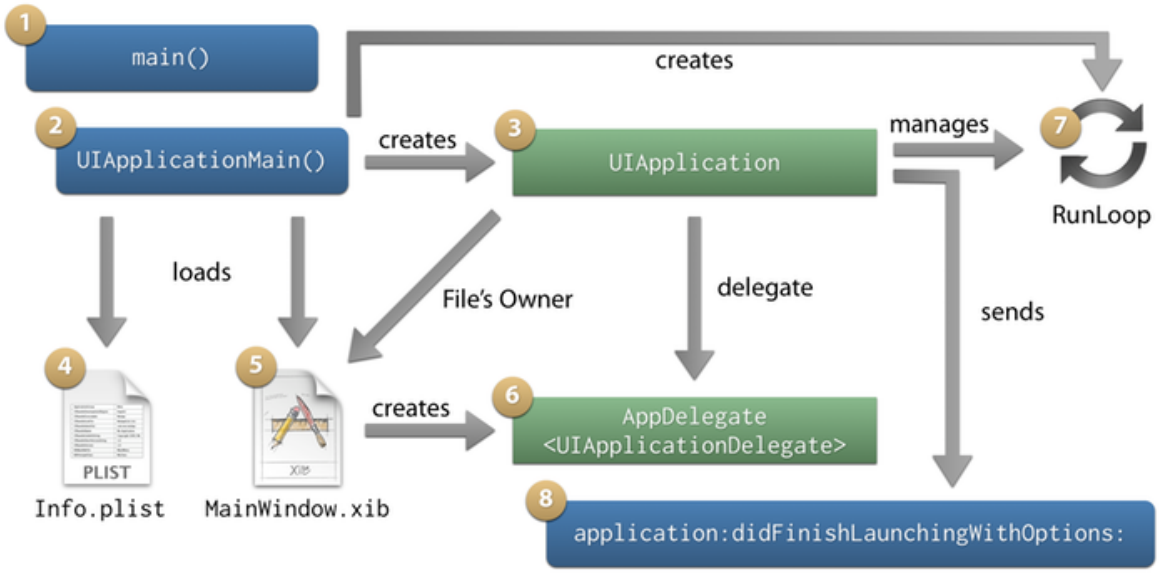


Figura 3.1 Ciclo de vida de una aplicación *iOS*.

C. El objeto **`UIApplicationMain`** creado carga el archivo de configuración **`Info.plist`**. Éste contiene información relacionada con íconos, recursos, cadenas que contienen el nombre de la aplicación así como su identificador.
   Un elemento importante es que aquí se encuentra el nombre del objeto interfaz que se utilizará para cargar la aplicación (comúnmente, llamado **`MainWindow.xib`**).
D. El objeto **`MainWindow`** contiene un objeto asociado al delegado de ***File's Owner***, así como un objeto **`UIWindow`**, el cual será utilizado como la ventana contenedora de la aplicación. Este objeto es una abstracción lógica de la superficie sobre la cual se inicializan las vistas, y está contenida dentro del objeto **`MainWindow.xib`**.
E. El método **`UIApplicationMain()`** crea un bucle de la aplicación (**RunLoop**) que es utilizado por la instancia **`UIApplication`** (estado 7), para procesar eventos como toques en pantalla o eventos de red. Este bucle creado es infinito, de manera que el método **`UIApplicationMain()`** 'nunca termina'.
F. Antes de procesar algún evento, la aplicación manda una señal hacia su *delegado* por medio del método **`application:didFinishLaunchingWithOptions:`**, llegando al estado 8. Este evento es uno de los más importantes y el más utilizado a nivel de desarrollo para la configuración de cualquier aplicación *iOS* que requiera el mínimo proceso de inicialización antes de presentar la aplicación al usuario. Justo en este método el control de la aplicación pasa a manos del desarrollador.

El *delegado* (el objeto **`AppDelegate`** descrito en el paso B), al recibir el mensaje de finalización de la configuración del sistema operativo y el alojamiento de la aplicación misma en memoria (con la reservación de recursos y la capacidad de manejar los eventos que se disparen en el dispositivo en ese espacio de vida) informa al desarrollador que la aplicación se ha iniciado.
En este punto se puede iniciar la lógica del motor, todo esto a través de una entidad que servirá de enlace para todos los módulos que integran el motor, de manera que exista una completa comunicación entre los mismos. A esta entidad, la denominaremos ***Gestor OpenGL***.

3.3 El Gestor *OpenGL*.

El presente documento describe el desarrollo de un motor de juegos basado en la arquitectura propuesta en la Figura 1.5. Esta arquitectura propone el desarrollo de 5 módulos principales[†††]:

1. Núcleo del Sistema
2. *Renderizado* de bajo nivel.
3. Administrador de escenas.
4. Mecanismos de interfaz humana.

Dentro de éstos módulos no se describe la entidad que servirá de enlace entre los mismos y los aspectos propios de la implementación (manejar adecuadamente los mecanismos de interfaz humana basado en las características del sistema, definir las características del *renderizado* a utilizar, etc). Esta entidad se denominará ***Gestor OpenGL***. La Figura 3.2 muestra la integración del ***Gestor OpenGL*** en la arquitectura antes propuesta.

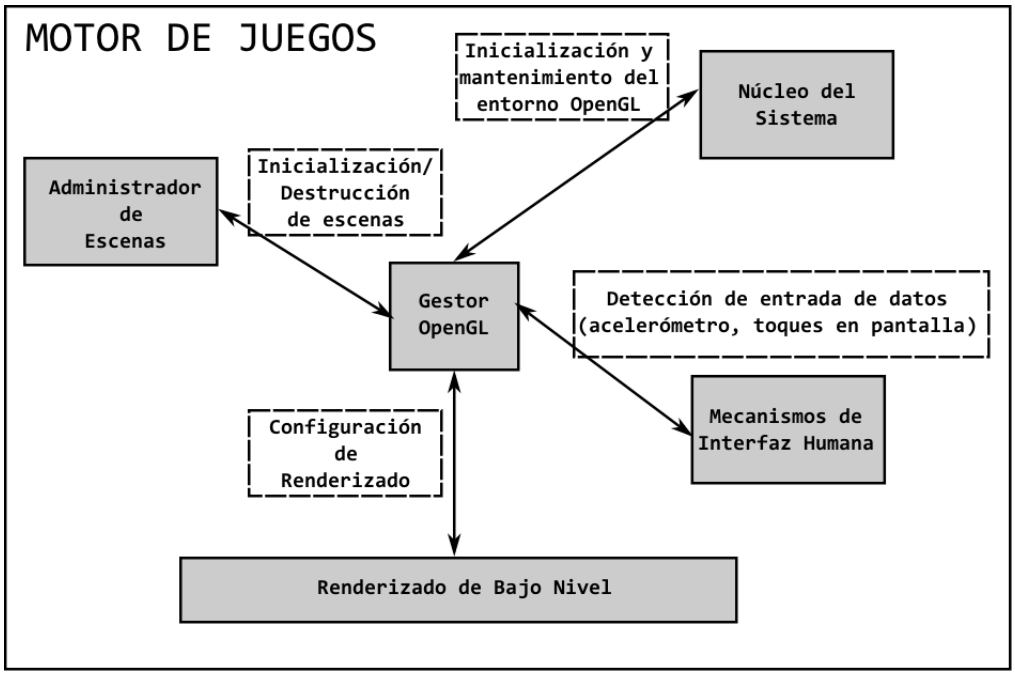


Figura 3.2 Integración del *Gestor OpenGL* a la arquitectura propuesta del motor de juegos.

La figura muestra las conexiones (a grandes rasgos) que existirán entre esta entidad y los módulos del motor. Cada una de estas conexiones se describirán a detalle en la sección de implementación para cada uno de estos módulos. Entretanto, en esta sección describiremos y definiremos las tareas propias del ***Gestor OpenGL***.

3.3.1 Arquitectura del *Gestor OpenGL*.

El punto de análisis del gestor será su definición. Definiremos al ***Gestor OpenGL*** como una entidad única de administración y enlace entre los diferentes elementos del motor de juegos.

[†††] *Como se he explicado en secciones anteriores, la capa de depuración estará presente en el motor de juegos, pero no forma parte de la implementación del mismo.*

El ***Gestor OpenGL*** será la entidad encargada de administrar las características globales de la aplicación tales como la configuración de *renderizado*, la entrada de datos, la actualización de pantalla en base al bucle de actualzación de la aplicación, etcétera; esto nos lleva a considerar que cualquier objeto, dentro del ciclo de vida de la aplicación creada por el motor, debe tener acceso a éstos mecanismos. Otra característica esencial en la naturaleza del mismo es que debe existir una sola instancia del objeto durante la vida de la aplicación. Para entender esta restricción, tomemos como ejemplo las funciones correspondientes a la configuración de *renderizado*. Una vez que hemos creado el ***Gestor OpenGL***, e inicialicemos la configuración que debe seguir el proceso de *renderizado* para el desplegado de los elementos del motor, proseguiremos ese camino con todos los nodos que se necesiten desplegar. Si en algún momento existiera inconsistencia en estos mecanismos, no se desplegarían los datos de manera correcta, provocando fallos. Un patrón de diseño que se adecúa totalmente a estas características es el patrón *singleton*[‡‡‡]. El diagrama de diseño del ***Gestor OpenGL*** se presenta en la Figura 3.3

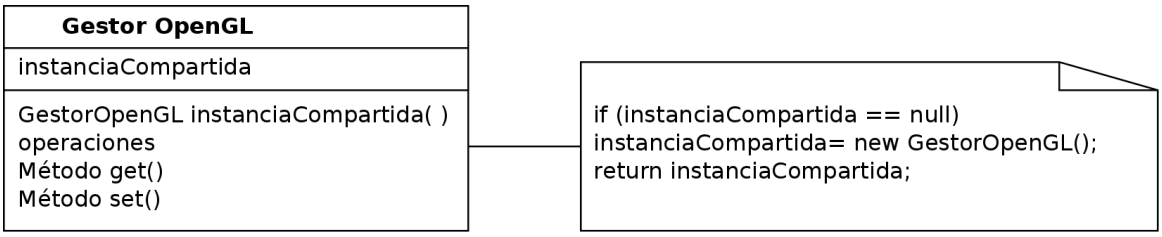


Figura 3.3 Diagrama de diseño del *Gestor OpenGL*.

Establecido el patrón de diseño que se considerará para el desarrollo del ***Gestor OpenGL***, se describirán las acciones y tareas que este llevará a cabo durante la aplicación.

3.3.2 Funciones del *Gestor OpenGL*.

Adicional a las funciones de conexión con los módulos del motor, el ***Gestor OpenGL*** realiza algunas tareas específicas, que sirven como apoyo del motor de juegos. La Figura 3.4 muestra el diagrama de secuencia de estas tareas.

1. Guardar la superficie *OpenGL*. Al guardar una referencia a la superficie garantizamos el acceso a la superficie *OpenGL* sobre la cual estamos trabajando, con el fin de tener mecanismos de acceso a los métodos y variables de la misma; de esta manera, el desarrollador podrá modificar características de la superficie, enfocadas directamente a las necesidades del juego, pero sin tener que interactuar con particularidades de *OpenGL* que dificulten la tarea. Dentro de estas funciones encontramos tres principalmente:

a) Iniciar los parámetros *OpenGL*. Debido a la existencia de varias versiones de *OpenGL* (1.x, 2.x, 3.x), es necesario establecer en las superficie que versión se utilizará para el motor. De igual manera, existen parámetros de profundidad, color, etc, que deben ser especificados antes de iniciar el desarrollo sobre esta plataforma.
b) Establecer el intervalo de animación. Por omisión, las superficies *OpenGL* se actualizan a una velocidad de 60 fotogramas por segundo, en caso de querer modificar este valor, es necesario establecerlo directamente sobre la clase **`GLKView`**.
c) Iniciar los parámetros de la cámara. Un mundo en tres dimensiones se caracteriza por el movimiento de la cámara (el punto de vista del observador), de esta manera la perspectiva de los objetos cambia, dando lugar a efecto de distancia y, en tres dimensiones, profundidad.

[‡‡‡] *Ver [21]*

Al iniciar la superficie debe situarse la cámara en un punto conveniente al desarrollador, de manera que sobre ese sistema de referencia inicia el desplegado de gráficos.

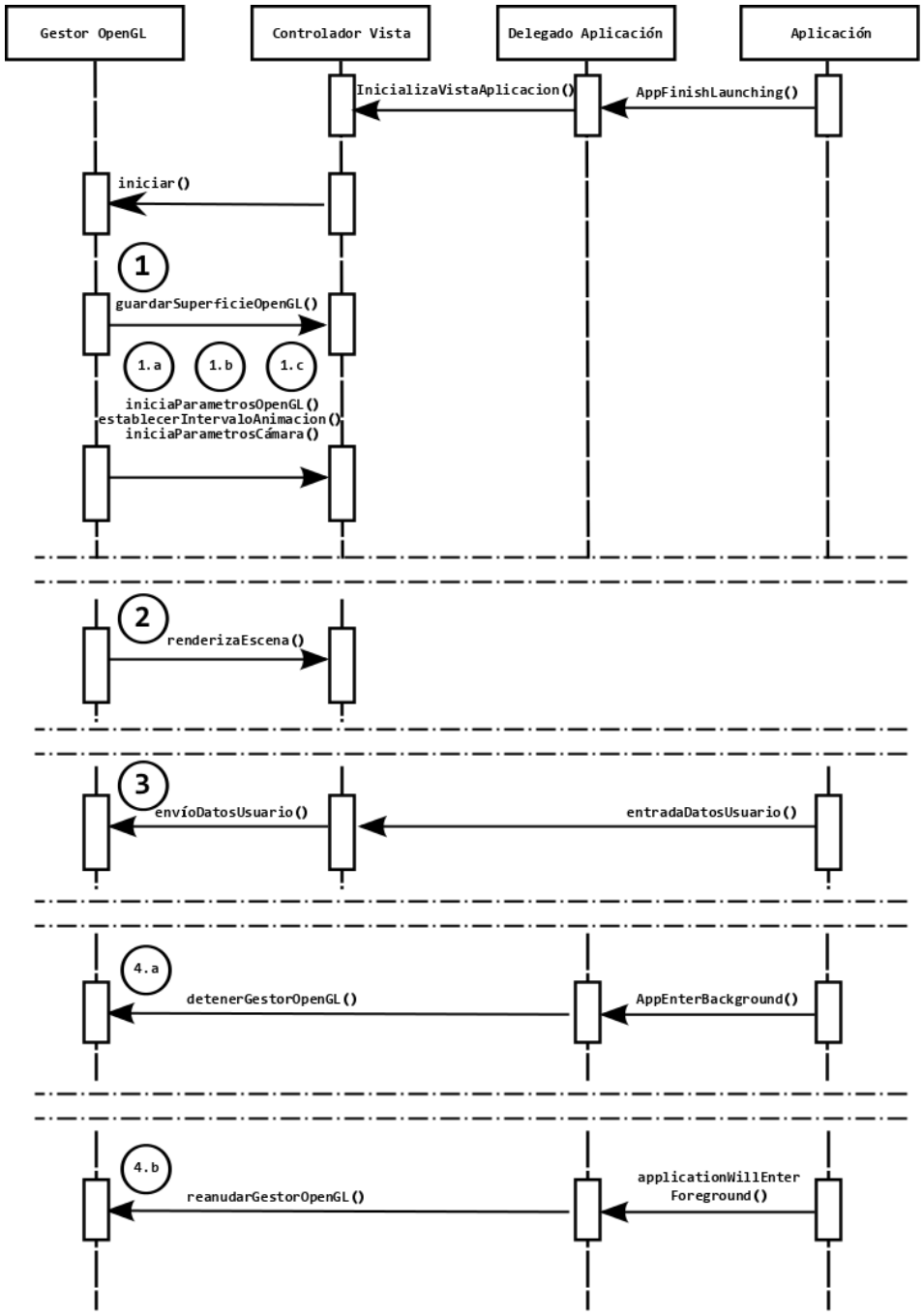


Figura 3.4 Tareas desarrolladas por el *Gestor OpenGL*.

2. Realizar navegación entre escenas. La vista **`GLKView`** es la superficie sobre la cual se generará la navegación entre escenas[§§§], que se presentan en la vida del juego. Esto se realiza enviando mensajes hacia la superficie por medio del ***Gestor OpenGL***.

[§§§] *El concepto de escena y su integración con el motor de juegos, se muestra en la sección 3.5.4, Definición y Composición Lógica de una escena*

La vista, al recibir estos mensajes, debe ser capaz de modificar la superficie, cargando los recursos y datos al iniciar la escena (**`initScene`**)
Estas funciones deben ser capaces de alcanzar la vista **`GLKView`**, de manera que los comandos de alto nivel (presentes en el ***Gestor OpenGL***), deben tener sus símiles en base a primitivas *OpenGL* que actuarán sobre la vista. Todo este proceso se realiza de manera interna, en el motor.

3. Entrada de datos de usuario. El usuario puede utilizar alguno de los mecanismos de entrada de datos (tocar la pantalla, usar el acelerómetro) durante la aplicación. El sistema operativo es el primero en tener conocimiento de la entrada de datos y enviar esta información a la aplicación que se encuentre en ejecución. La tarea del ***Gestor OpenGL*** es análoga a este proceso, su función es recabar la información directo de la vista, y enviarla hacia la escena que se encuentre en ejecución.

4. Pausa (a), reanudación (b) y finalización del contexto *OpenGL*. Una aplicación nativa de dispositivos móviles cuenta con diferentes estados delimitados por el sistema operativo.
Al inicio de la aplicación, se han indicado los pasos a seguir para inicializar el contexto; después de eso, la aplicación puede recibir mensajes del delegado de la misma (el objeto **`AppDelegate`**), indicando el cambio de estados de la aplicación. Estos dos estados se describen en la especificación de estados de una aplicación *iOS* de Apple *[23]*, y son los siguientes:

4.a Detener el ***Gestor OpenGL***. Esta fase se dispara cuando la aplicación entra al estado "background", esto pasa cuando el usuario sale de la aplicación haciendo click sobre el botón Home del dispositivo. Esto hace que la aplicación 'finalice'[****], y el usuario pueda hacer uso de otra aplicación. En este evento, el ***Gestor OpenGL*** debe pausar el contexto *OpenGL*, detener animaciones, pausar la música y detener métodos de actualización.

En esta fase, el delegado de la aplicación envía el mensaje por medio del método **`applicationDidEnterBackground().`** A nivel desarrollador, este método da la oportunidad de realizar configuración sobre la aplicación, permitiendo poner un menú de pausa, guardar variables de entorno, etcétera.

4.b Reanudar el ***Gestor OpenGL***. En este método se dispara cuando la aplicación regresa de un estado "background". En esta parte, es importante que el gestor reanude el contexto *OpenGL*, de manera que la aplicación vuelva a reproducir sus animaciones, métodos de actualización, etcétera. En esta fase, el delegado de la aplicación envía el mensaje por medio del método **`applicationWillEnterForeground()`**.

El proceso de finalización corre por parte del sistema operativo y se realiza al retirar de memoria la aplicación, de manera interna. Aun cuando el gestor debe especificar los mecanismos necesarios para finalizar adecuadamente las variables del motor, no existe interacción directa entre el sistema operativo y la aplicación.

Es necesario acotar que tanto los procesos de inicio de la aplicación como la finalización de la misma, corren por parte del sistema operativo, considerando esto, se hace uso de las llamadas a sistema presentes entre estos procesos para poner en marcha o liberar la memoria del ***Gestor OpenGL***.
Habiendo definido las características del ***Gestor OpenGL***, procederemos a describir los módulos que integran el motor de juegos en su arquitectura.

[****] *Debido a que los sistemas operativos actuales permiten que las aplicaciones se mantengan en background, el término finalizar no es del todo correcto, sin embargo, puede utilizarse para describir que la ejecución de la aplicación, en primer plano, ha llegado a su fin.*

3.4 El Núcleo del sistema.

Como se definió en la sección 1.1.1.6, existen procedimientos que quedan al margen del motor de juegos que se implemente, y son atendidos directamente por el sistema operativo. En el caso del motor de juegos presentado en este documento, el entorno *OpenGLES* y su inicialización presentan esas características.

3.4.1 Inicialización del contexto *OpenGL ES*

Al recibir el mensaje de inicialización por parte del sistema operativo, el delegado empieza el proceso de inicialización de la aplicación en el lado del desarrollador.
El primer paso es detectar la versión de *OpenGL ES* del dispositivo sobre el cual la aplicación se encuentra corriendo. Esto es importante para garantizar, o excluir las capacidades de funcionamiento del motor de juegos en el dispositivo. Dado que el presente proyecto, tiene como sistema operativo objetivo *iOS* 7.x se requiere, estrictamente, la presencia del *OpenGL ES 2.0* en el dispositivo.
El siguiente paso es definir una superficie destino del *renderizado* de gráficos. Según la especificación del manejo de *OpenGL* en *iOS[24]*, existen dos posibilidades: utilizar el *framework GLKit*, o utilizar la clase **`CAEAGLLayer`**, presente en el *framework QuartzCore.framework*, la cual es una superficie que provee de mecanismos para hacer llamadas a comandos en *OpenGL ES* como parte de *Core Animation*. En este punto, vale la pena establecer las ventajas de utilizar cualquiera de los dos acercamientos presentados anteriormente.
Utilizar la clase **`CAEAGLLayer`** como la superficie destino del *renderizado* de los gráficos, implica que el manejo de inicialización de los estados de *OpenGL* es mayor. Inicialmente, *Cocoa Touch* presentaba esta clase como el nivel de abstracción más bajo de *OpenGL ES*, así como el único acceso a estas capacidades. Al inicio del entorno (llegando hasta la versión 4.x de *iOS*), el desarrollo de aplicaciones *OpenGL ES* resultaba menos accesible a los desarrolladores. En *iOS* 5, se agrega *GLKit*, como un *framework* encaminado a facilitar el uso de *OpenGL ES*. Esto no debe tomarse como una reducción del nivel de abstracción del manejo de *OpenGL ES* en la plataforma, sino como mecanismos de automatización de tareas repetitivas y estandarizadas por parte del API (por ejemplo, la inicialización de las superficies, el establecimiento del bucle de *renderizado*, etc.)
Las ventajas antes presentadas, permiten al desarrollador enfocar su atención en las particularidades presentes después de la inicialización, dejando a un lado las tareas repetitivas de inicialización y mantenimiento. Por otra lado, el contar con una entidad bien definida, y totalmente aprobada por el entorno, representa una ventaja que debe ser utilizada como un buen preámbulo en el desarrollo, adicional a esto, motivación principal de un motor de juegos es atacar los problemas en tiempo de ejecución de los videojuegos, así como la interacción con el usuario final, no reinventar mecanismos de inicialización o mantenimiento de la máquina de estados *OpenGL*.

Como se comentó anteriormente, *GLKit* provee clases de vista y controlador que reducen el código para la inicialización y mantenimiento, que antes eran requeridos por parte de *OpenGL ES*. La clase **`GLKView`** maneja la infraestructura *OpenGL ES* necesaria para soportar las directivas de *renderizado* y dibujado de modelos, y el controlador **`GLKViewController`** provee un bucle de *renderizado* para el manejo de las animaciones presentes en la vista, haciendo uso de directivas *OpenGL ES*.

Estas clases heredan el patrón de diseño *MVC* (Modelo-Vista-Controlador), de manera que este modelo de programación se acerca al manejado principalmente para el desarrollo de aplicaciones sobre *Cocoa Touch*. El contenido que se despliega en la vista se carga 'sobre

demanda'. Al inicio de la vista, el método de dibujado es llamado, de manera que todos los comandos de inicialización son interpretados, para posteriormente desplegar el resultado de los mismos.
En materia de *renderizado* de gráficos, esta es la última etapa de la cual se hace cargo *GLKit*. (Ver Figura 3.5)

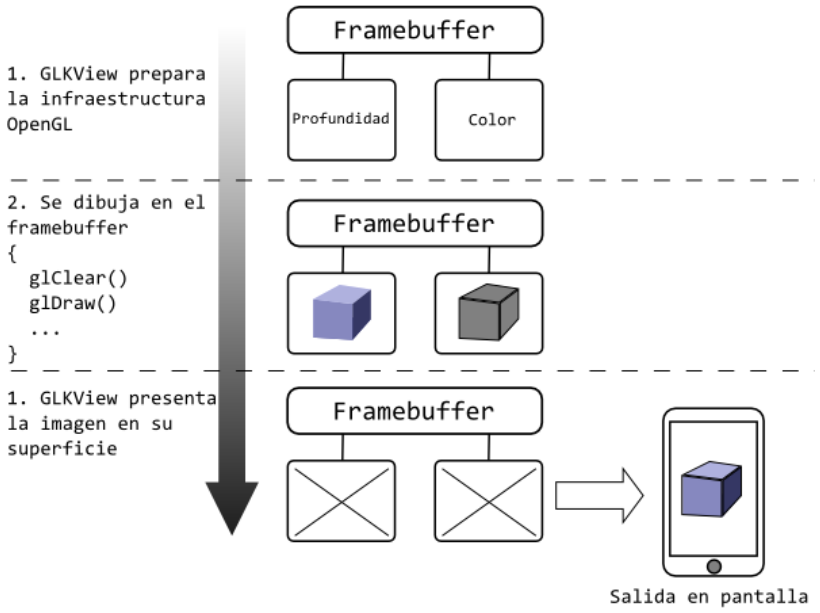


Figura 3.5 *Renderizado* de los gráficos en una vista *GLKView*.

Uno de los aspectos principales del manejo de *OpenGL ES* es el acceso al bucle de *renderizado* de gráficos. Este elemento es sumamente importante, dado que expone un mecanismo de carga de recursos, para su posterior presentación en el dispositivo de salida. *GLKit* cuenta con un mecanismo con estas características, presente en el controlador de la vista (**`GLKViewController`**). El bucle presente en el controlador de la vista sigue el patrón de diseño presente en la mayoría de las aplicaciones de juegos y simulación, el cual cuenta con dos fases: actualización y desplegado. Particularmente, en esta clase (**`GLKViewController`**), se llama el método **`glkViewControllerUpdate`.** La Figura 3.6 presenta este bucle de animación.

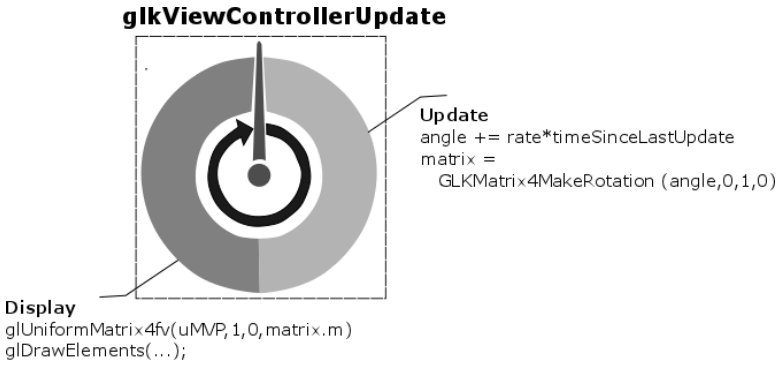


Figura 3.6 Ciclo del bucle de animación en GLKViewController.

Para la fase de actualización, el controlador llama a su propio método de actualización (o **`glkViewControllerUpdate`** de su delegado) .

En este método, se prepara para el siguiente *frame*. Por ejemplo, un juego puede utilizar este método para determinar las posiciones del jugador y sus enemigos basado en eventos de entrada recibidas desde el último *frame*, y realizar algún cálculo para obtener el siguiente estado de la animación. De igual manera, se puede hacer uso de mecanismos de temporización basados en el factor de actualización de este método, de manera que se pueda adecuar la actualización de las animaciones en base al desempeño de la aplicación.

Para la fase de desplegado de gráficos, el controlador llama el método de visualización, que a su vez llama a su método de *renderizado* de la vista . El método de *renderizado* es el encargado de enviar comandos de *OpenGL ES* hacia la *GPU* para *renderizar* su contenido.

Como opción, se puede utilizar la propiedad **`preferredFramesPerSecond`** para establecer un marco deseado de velocidad de actualización, y así optimizar el rendimiento del hardware; con esto, el controlador de la vista elige automáticamente una velocidad de fotogramas óptima cercana al valor establecido. Todos los mecanismos presentes en el bucle de animación deben ser llevados a una capa de abstracción mayor de manera que el desarrollador, que hasta este punto ha quedado al margen de la inicialización del entorno, pueda modificar la configuración de variables y ajustarlas a las necesidades de su aplicación.
Es necesario establecer que el gestor, al inicio de la aplicación, es el encargado de realizar toda esta configuración, a través del establecimiento de los parámetros necesarios en la inicialización de la superficie.

3.5 *Renderizado* de Bajo Nivel.

Uno de los retos del motor de juegos es establecer los procedimientos necesarios para la generación de modelos en tres dimensiones, y su manejo dentro de la estructura del motor como objetos de alto nivel (definidos en *Objective-C*). Esta sección se encarga de describir los procedimientos a seguir para lograr esta tarea.

El primer punto que debemos definir es el mecanismo que el motor utilizará para cargar los modelos tridimensionales dentro de la aplicación. Para ello, analizaremos la composición de los objetos en 3 dimensiones, los cuales se caracterizan por su modelo, textura y material[††††].

3.5.1 Modelo.

Los modelos con la representación geométrica de un objeto tridimensional. Los modelos pueden ser generados a través de código (en base a primitivas *OpenGL*, indicando los puntos que representan la figura geométrica), pero generalmente son leídos desde archivos, los cuales son generados utilizando software de diseño 3D. La geometría consiste en la información de los vértices, así como la descripción de la superficie que indica como se encuentran conectados los vértices para la creación de polígonos, triángulos, líneas o simples puntos. La Figura 13.7 muestra un ejemplo de un modelo.

Los vértices tienen diferentes tipos de atributos, unos de los más importantes es su posición dentro de la superficie. Otros atributos comunes es el vector normal, las coordenadas de la textura y el vector tangente. El vector normal define la superficie normal en la posición del vértice y es utilizado para calcular el factor de iluminación. Las coordenadas de textura son utilizadas para mapear la imagen de textura sobre la superficie del modelo.

[††††] *[25] Chapter 2.1 "Resource Handling"*.

El vector tangente, junto al vector normal son los encargados de definir la superficie que describen los vértices en el momento de recrear el modelo.

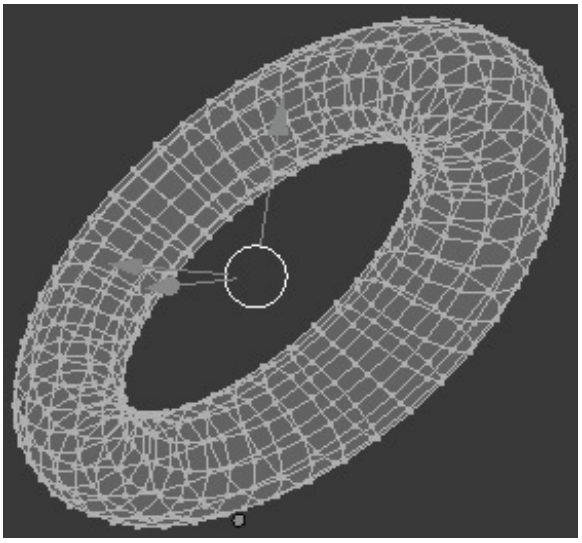


Figura 3.7 Modelo de un volumen torus sin textura, creado en Blender.

3.5.2 Textura.

Originalmente, las texturas eran imágenes bidimensionales mapeadas sobre superficies. Actualmente, son consideradas como estructuras de datos utilizados durante el *renderizado*. Las texturas pueden ser uni, bi o tridimensionales. La Figura 3.8 muestra el modelo del volumen torus enlazado con su textura.

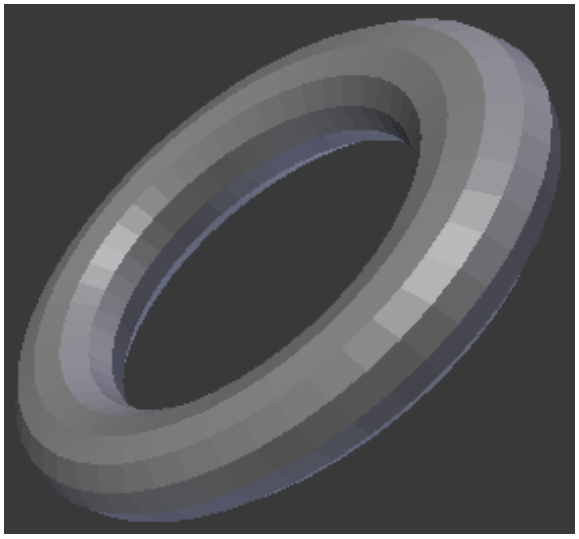


Figura 3.8 Modelo de un volumen torus con textura, creado en Blender.

La textura es mapeada, en base a la información del modelo, sobre la superficie del modelo. Una analogía podría ser el utilizar una funda sobre un monitor, la funda conoce la posición de cada uno de los puntos que la integran, de manera que embona sobre la superficie que cubre. Las texturas son utilizadas para manejar la luz que afecta el modelo, así como el color del mismo. En un modelo tridimensional, el afectar el color de la textura, afectará el color del modelo completo.

3.5.3 *Shaders*.

Los *shaders* son pequeñas porciones de código que se ejecutan sobre la unidad de procesamiento gráfico, enviando información de desplegado directamente desde la memoria o del CPU. Existen diferentes tipos de shaders, los cuales trabajan dentro de un pipeline; la Figura 3.8 muestra el proceso que se describe en base a los *shaders*.
Los *shaders* reciben la información de los atributos del modelo, y en base a ello se catalogan, dependiendo de las funciones que realizas sobre estos datos. Principalmente, tenemos tres tipos de *shaders*: vertex *shader*, *geometry shaders* y *fragment shaders*.

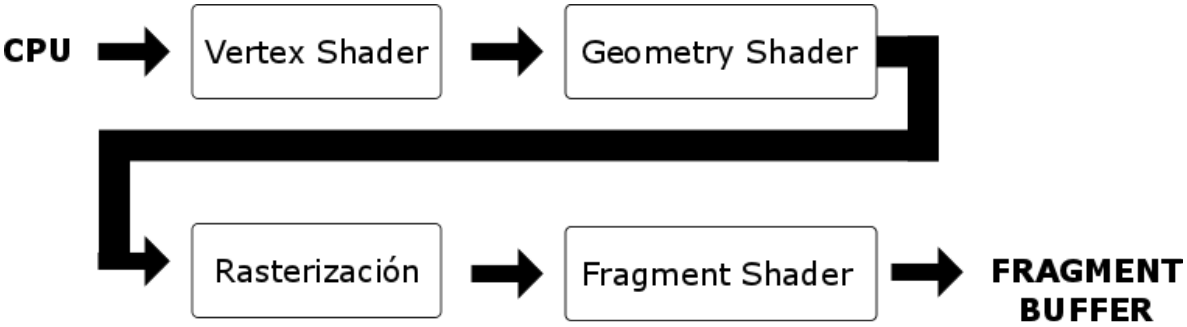


Figura 3.9 Pipeline de shaders.

Vertex shaders. Éstos *shaders* operan sobre vértices individuales, y reciben los datos desde los modelos. Estos *shaders* están encargados de modificar los atributos de los vértices, como posición, color o las coordenadas de las texturas.

Geometry shaders. Operan sobre primitivas individuales, como polígonos, puntos o líneas, y reciben la entrada proveniente de los *vertex shaders*. Estos *shaders* pueden emitir o no primitivas, por lo que su integración en el pipeline puede ser nula (de hecho, podrían omitirse en caso de no ser utilizados). Uno de los principales usos es la generación de volúmenes de extrusión y sombras por parte de un modelo.

Fragment shaders. Estos *shaders* operan sobre pixeles, de manera individual. La salida de este *shader* es el color de un pixel para escribirse directamente sobre el *frame buffer*. Este *shader* tiene la capacidad de definir la profundidad de los modelos y establecer el grado de iluminación y reflexión por parte de cada pixel. De igual manera, tiene la capacidad de 'esconder' u omitir pixeles que, debido al volumen del objeto, no sean visibles hacia la proyección de la cámara.

3.5.4 Material.

Un material es una colección de texturas, *shaders* y valores de uniformidad. Los materiales son creados en los procesadores de gráficos tridimensionales, para definir el comportamiento del modelo en su entorno. El definir un material 'liso', 'brillante' u opaco, afectará, en compañía del comportamiento del entorno, la manera en que el modelo interactúa en el entorno que lo rodea.

Definidos los elementos que componen un objeto tridimensional, es necesario diseñar una clase que permita la inicialización, manipulación y seguimiento de cada uno de estos atributos. En la siguiente sección definiremos el diseño y características de esta clase, sus principales métodos, propiedades, así como el patrón de diseño utilizado para definir la misma.

3.5.5 Definición de la clase *nodo*

En la definición de un motor de juegos, el renderizado de objetos juega un papel fundamental. La primer consideración a tomar en cuenta es la naturaleza de las tecnologías de aceleración de gráficos por hardware (OpenGL, Direct3D), las cuales se centran en generar gráficos de manera eficiente, sin embargo, no soportan arquitecturas de diseño complejas; por ello es necesario generar capas de mediación que permitan estructurar estas tecnologías hacia lenguajes de alti novel. Strauss and Carey *[26]* definen ..."*un mecanismo de renderizado realmente interactivo debe ser un mecanismo basado en objetos, que permita la transformación de los mismos de manera eficiente por parte de los usuarios*". Si se considera la naturaleza interactiva de los modelos presentes en un motor de juegos, esta definición encaja perfectamente.

Tomando como preámbulo las consideraciones anteriores, para la generación y manipulación de elementos propios del objeto tridimensional (base del motor de juegos 3D), es necesario definir una clase que permita almacenar sus atributos, de manera que puedan ser modificados en base a los métodos expuestos por el motor.
El primer paso, es definir el patrón de diseño de software que utilizaremos como base para los nodos: el patrón *prototipo*[‡‡‡‡].

3.5.5.1 El patrón de diseño prototipo.

Una arquitectura prototipo, se define bajo la siguiente premisa: "*Un objeto puede utilizarse como un generador de objetos que son similares entre sí*[§§§§] ". En esta etapa estableceremos que el motor atiende peticiones de creación de cualquier objeto a través de inicializadores prototipo debido a que debe administrar, durante la vida de la aplicación, el alojamiento de memoria de los mismos.

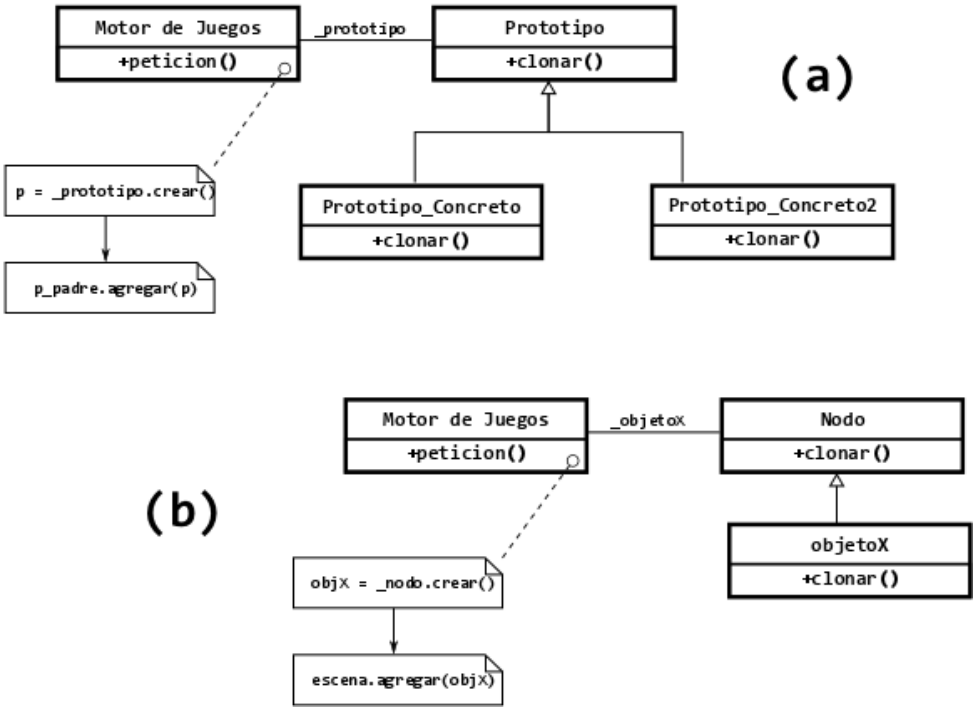


Figura 3.10 Creación de objetos del motor de juegos bajo la arquitectura prototipo.

[‡‡‡‡] *[27]. Chapter 1. "Game Design Patterns*
[§§§§] *Ídem.*

El mecanismo por el cual el motor almacenará, indistintamente de las particularidades de cada uno, los objetos que se generan dentro del motor, es a través de un objeto prototipo, que se caracterizará por métodos y variables base. Este objeto base se denominará nodo a lo largo de la siguiente sección. La Figura 3.10 muestra el proceso de inicialización de los objetos.
El motor, al recibir una llamada de creación de cualquier objeto, toma la llamada del inicializador prototipo, de manera que crea un objeto genérico (***nodo***), el cual es almacenado directamente en el motor, asignándole la membresía del objeto sobre el cual quiere agregarse (denominado objeto padre[*****]). Por ejemplo, se desea crear un objeto de tipo X.
El mensaje del inicializador prototipo llega al motor, el cual genera un objeto genérico **objetoX**, el motor lo asocia al objeto en el cual quiere ser agregado (en este caso, una escena **escenaJuego**), de manera que lo asigna como un hijo de este último. (Ver Figura 3.10b). Al observar esta figura, notamos que el prototipo de objeto pide una referencia al prototipo particular del objeto (denominado **objetoX**). Esto representación define que la creación de los objetos se realizará en cascada, atendiendo primero al prototipo base (el objeto que alojará memoria en el sistema operativo, a nivel más bajo), hasta el prototipo de jerarquía menor (en este caso una **objetoX**, pero podría llegar hacia objetos **modelo3d**, **label**, **menu**, etc). Por ello, en la figura 3.10a, observamos que existen objetos **Prototipo_Concreto**, los cuales definen cada uno de los objetos descritos anteriormente.

La elección del patrón de diseño prototipo como base de los nodos del motor de juegos, se realizó en base al análisis propuesto por Jamillah-Sufian*[28]*, "*... identificar las clase principales, las interacciones entre ellas y definiendo, al final del análisis, las dependencias entre las mismas, de manera que se reduzca la redundancia y obtengan, de manera simplificada, los métodos y variables comunes a todas las clases del sistema*".

3.5.5.2 Propiedades y métodos de la clase nodo.

En la sección anterior, se definió un ***nodo*** como el objeto base sobre el cual se generarán todos los elementos que el motor de juegos puede crear. Su fundamento práctico es el establecer propiedades generales que todos los elementos que dentro del motor de juegos presentarán, de manera que cualquier elemento herede de esta clase base, adquiriendo las propiedades y métodos comunes. La Figura 3.11 presenta un diagrama simplificado acerca de la definición del objeto nodo.

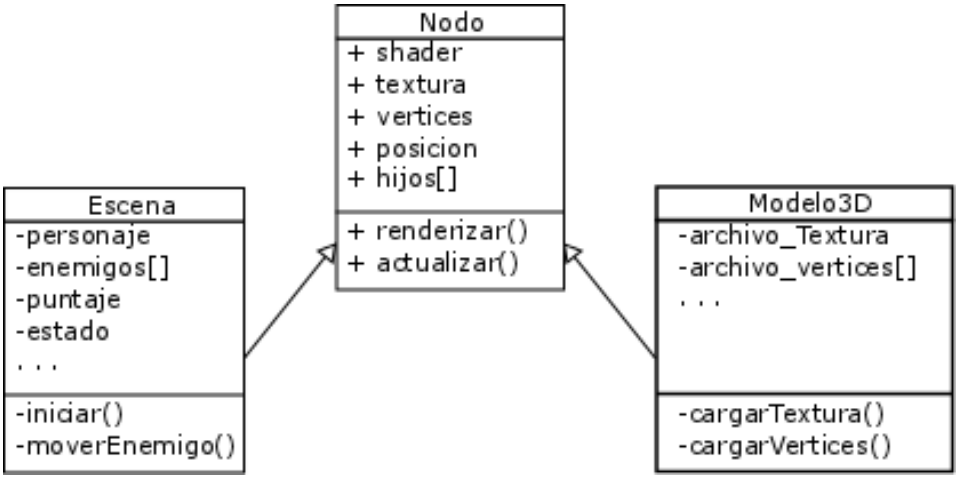


Figura 3.11 Diagrama simplificado del objeto *nodo*.

[*****] *En este punto no se ha hablado de la conexión de membresía nodo padre – nodo hijo. Este mecanismo se discutirá en la sección 3.5.5.1. Interacción del nodo tridimensional y la escena.*

El ***nodo*** se encarga de guardar los atributos, propiedades y métodos generales de cualquier objeto. De esta manera, podemos generalizar estas propiedades, y lograr la propagación, en entornos globales de los diferentes cambios durante el tiempo de ejecución. Tomando como ejemplo un ***nodo*** tipo automóvil, si agregáramos elementos a el como sus hijos (llantas, asientos, volante), bastaría con propagar el movimiento del vehículo hacia sus ***nodos*** hijos, de manera que, siguiendo el ejemplo, al moverse el carro, las llantas y demás elementos seguirán este movimiento. Esta definición del objeto nodo es retomada del trabajo presentado por Thomas y Kok-Why [29], para la definición de un objeto dentro de una arquitectura de escena grafo, para una arquitectura orientada a objetos dentro de un motor de juegos.

Es necesario comentar que en esta etapa de diseño, se establecen las propiedades más importantes que podemos observar en un ***nodo***[†††††], sin embargo, a lo largo del presente documento, se presentarán variables que caracterizan otras propiedades del mismo, y pueden ser consideradas como variables adicionales.

3.5.6 Mecanismos de *renderizado* de nodos.

El ***nodo***, como modelo base del motor, cuenta con un mecanismo que permite el despliegue en pantalla de su estructura a través de la superficie *OpenGL*; esto se logra definiendo las características de cada nodo, y enviándolas hacia el GPU para que este se encargue de desplegar su información. Este procedimiento se logra en base al mecanismo descrito en la sección 3.5.3 y utiliza como actores principales a los *shaders*.

Tomando en consideración que cualquier objeto dentro del motor es un nodo, todos estos objetos deben estar definidos por las propiedades definidas en la sección anterior. Por ello, al inicializar cualquier nodo, es necesario establecer los valores para estas variables. Los shaders definen un proceso de descomposición de cualquier modelo, hacia información que el GPU puede interpretar. El grandes rasgos, el proceso se visualiza en la Figura 3.12.

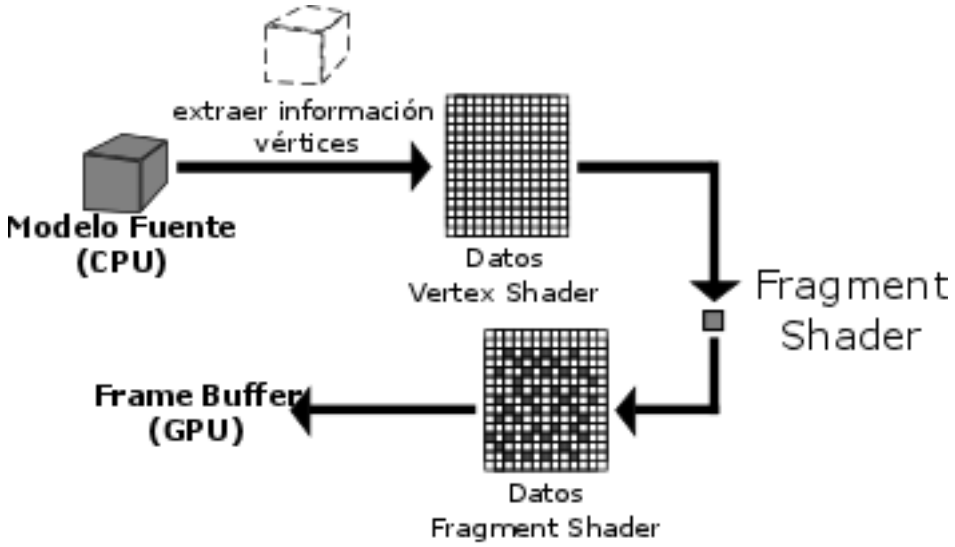


Figura 3.12 Proceso de *renderizado* del objeto *nodo*.

[†††††] *En las propiedades, encontramos una variable denominada shader. En secciones posteriores, se definirán sus características.*

Se observa que se provee un modelo fuente (ya sea un modelo predefinido por algún archivo fuente o información de manera manual) y se envía la información hacia los *shaders*, de manera que estos procesan su información de forma (*vertex shader*) y la información de composición y color (*fragment shader*). En base a este proceso, cualquier nodo puede ser procesado por el motor, y ser desplegado en la superficie *OpenGL*.

Dentro de los nodos, existen dos diferentes que agrupan los elementos principales que el presente trabajo describirá como los actores principales del motor, el nodo escena y el nodo modelo 3D.

3.5.6.1 Definición y composición lógica de una escena.

Una escena es un nodo, definido especialmente por el motor de juegos, que reserva un espacio de memoria dentro de la aplicación y permite desplegar, en base a las directivas que describa dentro de su composición, los gráficos que la componen. Temporal y físicamente (al decir físicamente, se refiere a un espacio de memoria), sólo puede existir una escena en un instante determinado del juego. En una escala jerárquica, la escena se muestra en la Figura 3.13

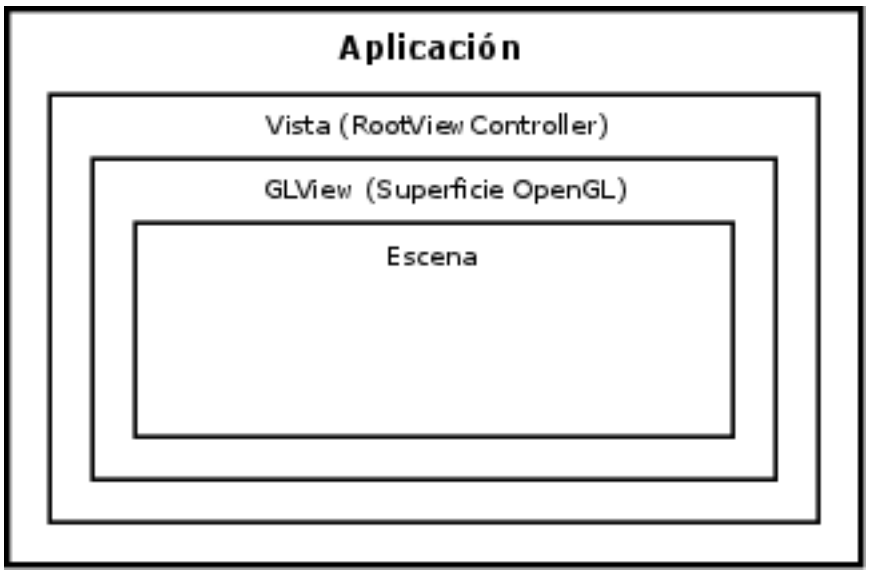


Figura 3.13 Jerarquía presente en el motor de juegos.

La escena se encuentra inmersa en la superficie *OpenGL*, por lo tanto, para la inicialización y mantenimiento de los elementos presentes en la misma, es necesario establecer un mecanismo que permita enlazar la escena en ejecución con la superficie *OpenGL*, de manera que los cambios que requiere la escena, en términos de despliegue de información y entrada de datos, se envíen hacia la superficie *OpenGL*. Esto se logra a través de los mecanismos de *renderizado* presentes en el ***nodo*** base. Dado que la escena hereda del nodo, para el *renderizado*, la escena simplemente debe llamar el método de *renderizado* presente en su superclase, y los mecanismos para la inicialización del objeto podrán ser utilizados.

3.5.6.1.1 Mecanismos de *renderizado* en las escenas.

Como se ha mencionado en la sección anterior, las escenas envían comandos de desplegado de gráficos directamente hacia la superficie *OpenGL*. Por lo tanto, es necesario especificar el funcionamiento de estos mecanismos en mayor detalle.
La escena acciona el inicializador prototipo, el cual se reporta con el motor de juegos, y su respectivo inicializador personalizado que corre por parte del desarrollador.

Es en este inicializador donde el programador incluye los métodos de dibujo de gráficos, especificación de variables, agendado de métodos e inicializadores de estados dentro de la aplicación. La Figura 3.14 muestra la arquitectura básica de esta inicialización.

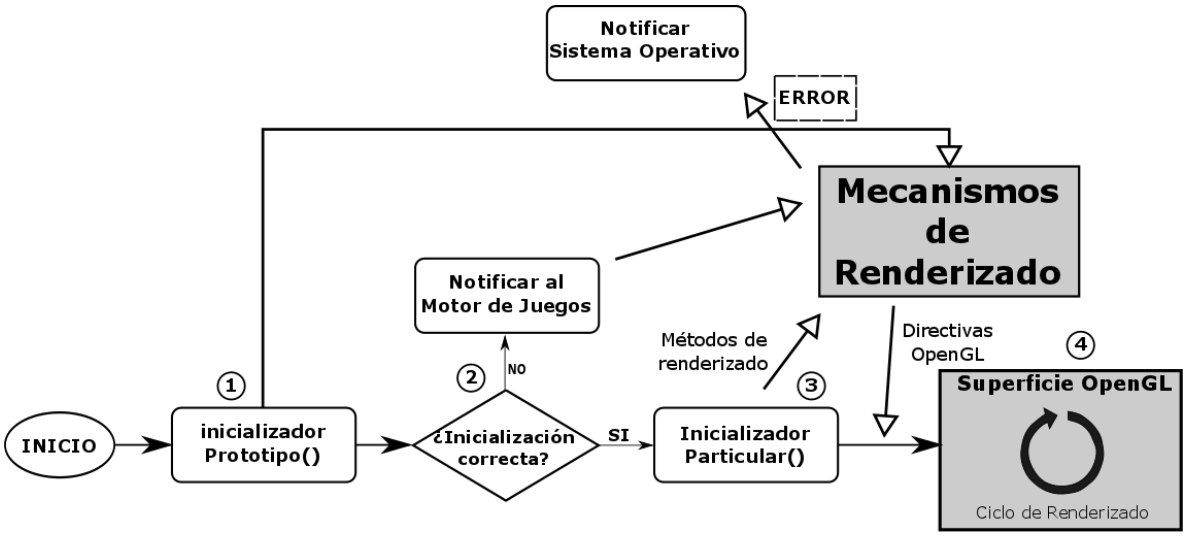


Figura 3.14 Arquitectura de la inicialización de la escena.

Inicalmente el inicializador prototipo reserva memoria en la aplicación, esto se logra comunicándose al motor de juegos, el cual gestiona el manejo de memoria de todos los objetos generados por el motor de juegos (Paso 1).
Posteriormente el inicializador de la escena recibe una respuesta por parte del motor de juegos, indicándole que la inicialización se realizó exitosamente, lo cual asegura, hasta ese momento, que el objeto se encuentra en memoria, listo para recibir mensajes por parte del usuario (Paso 2).
Los primeros mensajes que recibirá el objeto serán los métodos presentes en la inicialización personalizada, los cuales son métodos que se incluye en la construcción de la clase. Estos mensajes se mapearán con las directivas *OpenGL* que afectarán la superficie; es tarea del motor realizar este mapeo de manera transparente al usuario (Paso 3).
El último paso (Paso 4), se refiere al bucle de actualización de la superficie. En este punto, la escena es capaz de enviar primitivas de *renderizado* que permitan el despliegue de modelos.

Un punto importante es que, para establecer homogeneidad en la definición de la caracterización de los nodos, es necesario definir los elementos que el *shader* tomará para generar la escena. Como se comentó anteriormente, el envío hacia el *shader* puede contener información nula o inexistente, de manera que representará una entidad 'abstracta', que no contará con forma ni composición, pero que tendrá presencia dentro de la superficie *OpenGL*[‡‡‡‡‡].

Hemos definido la escena y los mecanismos que la integran, el siguiente paso es definir los nodos objeto tridimensional, los cuales poblarán la escena en base a modelos 3D predefinidos.

3.5.6.2 Definición del nodo tridimensional.

En la sección anterior se definió la escena como un nodo particular, con mecanismos de *renderizado* definidos, pero utilizados de manera especial (generar mecanismos de *renderizado*, pero sin el envío de información relativa a vértices).

[‡‡‡‡‡] *Los mecanismos específicos del renderizado de las escenas, se describirán en la sección 4.2.3.*

En esta sección definiremos el nodo modelo tridimensional, el cual explotará, en su totalidad, los mecanismos de *renderizado* ofrecidos por *OpenGL*.
Retomando la Figura 3.12, sabemos que para desplegar un modelo es necesario tomar la información de sus vértices, la información de los pixeles que la conforman, procesarla y enviar primitivas que utilicen esta información para *renderizar* el modelo. El caso de la escena, resulta un procedimiento simplificado, debido a que la información de vértices, y pixeles se establece de manera lógica, asignando los valores por medio de variables, las cuales se encuentran en la memoria del programa. Sin embargo, los modelos tridimensionales no funcionan de esa manera. Es imposible pensar que podemos dibujar, en base a vértices establecidos por programación, modelos tridimensionales complejos.
Es por ello que se harán uso de archivos generados por procesadores de gráficos tridimensionales como fuente de la información para los *shaders*. En la etapa de implementación ahondaremos mas en este tema, en esta sección nos limitaremos a establecer las propiedades que esperamos obtener de los archivos generados por estos procesadores.
Como un primer paso, para la generación de modelos tridimensionales, nos encontraremos con una propiedad antes descrita, pero solamente presente en modelos tridimensionales: la textura. Analicemos la Figura 3.15, la cual es una extensión de la Figura 3.12.

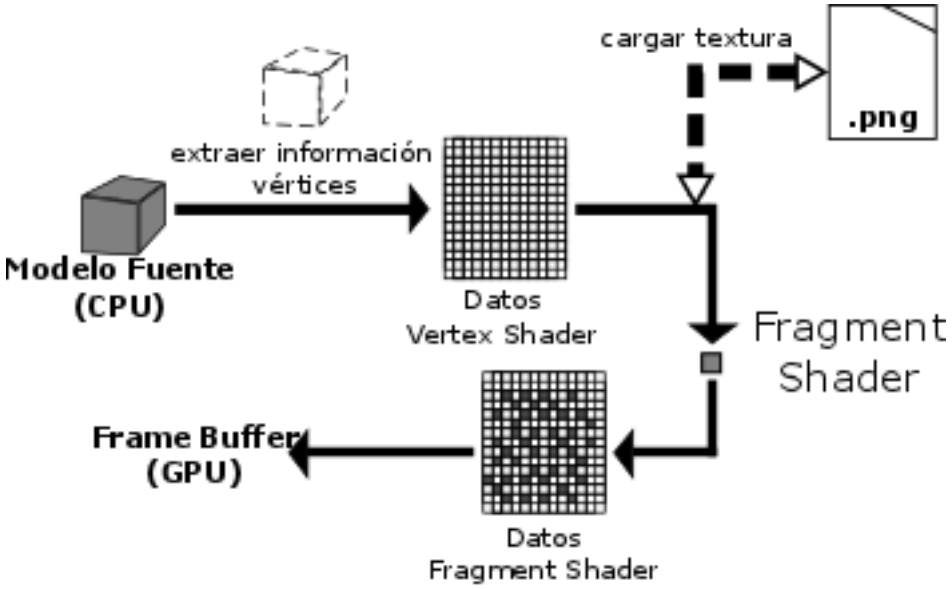


Figura 3.15 Proceso de *renderizado* del objeto *nodo* modelo tridimensional.

Se observa la inclusión de una propiedad mas llamada textura. Esta propiedad simplemente caracteriza una imagen (preferentemente .png), la cual envuelve el modelo. Esto es muy utilizado en la generación de colores, patrones o materiales dentro de un modelo, y permite añadir diseños y formas complejas a las superficies que describen el modelo.

En esta etapa, no se describen los mecanismos a utilizar para extraer la información que se muestra en la Figura 3.12, este procesamiento se describe, a detalle, en el capítulo 4, implementación.

3.5.6.3. Interacción del nodo tridimensional y la escena.

La escena de juego tiene como objeto el poblar la superficie por medio de objetos tridimensionales (dada la naturaleza del motor de juegos) inmersos en la misma.

Por ello, no es de extrañar que las modificaciones realizadas sobre la escena, deben afectar los modelos que se encuentran contenidos en ella[§§§§§]. Esto lleva a describir el mecanismo por el cual la escena, como padre de los modelos que la contienen, propaga los cambios realizados sobre de ella hacia sus hijos. De esta manera, las modificaciones se propagan en cascada. La Figura 3.16 muestra este mecanismo.

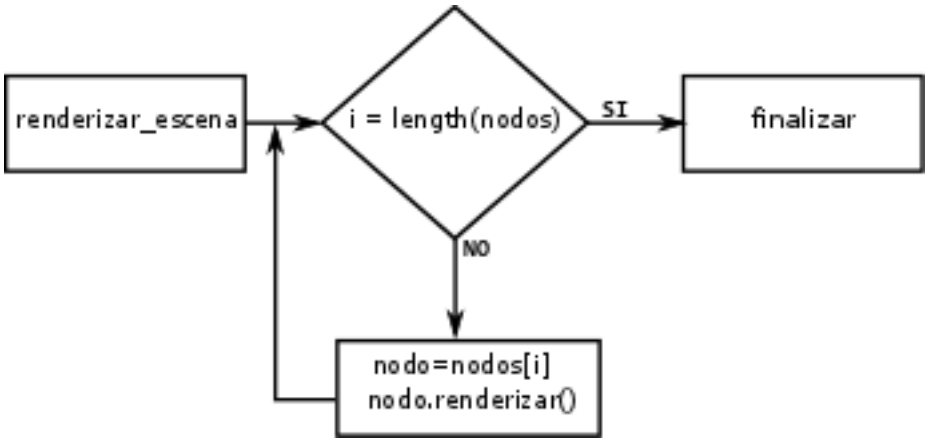


Figura 3.16 *Renderizado* de nodos dentro de su nodo padre.

La figura describe el *renderizado* dentro de un nodo padre, por lo cual, el funcionamiento descrito anteriormente relativo a las escenas y los modelos presentes en ella, se pueden generalizar para cualquier membresía entre nodos. El motor no limita la relación de membresía entre nodos a las escenas, por lo que los modelos pueden presentar estas interacciones.

La definición de este mecanismo de control referente a la escena y los objetos que la integran, se define en base al trabajo de Döllner and Hinrichs [30], los cuales definen el término de '*escena portátil*', asegurando que "*...la portabilidad de una escena, se consigue separando el renderizado del objeto y el mecanismo de renderizado del motor...*". Con esto se garantiza la propagación de los cambios presentes en una escena a los objetos que la integran.

En este punto se han definido las escenas, y los nodos que las componen (modelos tridimensionales). El siguiente paso es definir los mecanismos que gestionarán y administrarán las escenas dentro del ciclo de vida del motor.

3.6 *Administración de escenas.*

Durante el transcurso de un juego cualquiera, podemos observar la transición entre diferentes fases del juego, a las cuales accesamos de manera secuencial, ordenada y, más importante aún, de manera excluyente (sólo una fase a la vez). Empezando por el menú inicial, observamos diferentes etapas del juego; cada una de las cuales cuentan con características específicas, mecanismos de reacción a entrada de usuario, o recursos propios. A estas fases, las denominaremos escenas, por la analogía a las diferentes etapas de las obras de teatro o cinematográficas.
Cada escena debe ser inicializada, de manera que sus recursos y métodos puedan ser utilizados en el instante en que los necesitamos. Sin embargo, debe existir un mecanismo que permita la conmutación entre las escenas, de manera que las transiciones sea ordenado y claro.

[§§§§§] *Revisar la descripción de la propiedad hijos, descrita en la sección 3.5.5.2*

Este mecanismo se encuentra presente dentro del ***Gestor OpenGL*** y se describe en el diagrama de la Figura 3.17

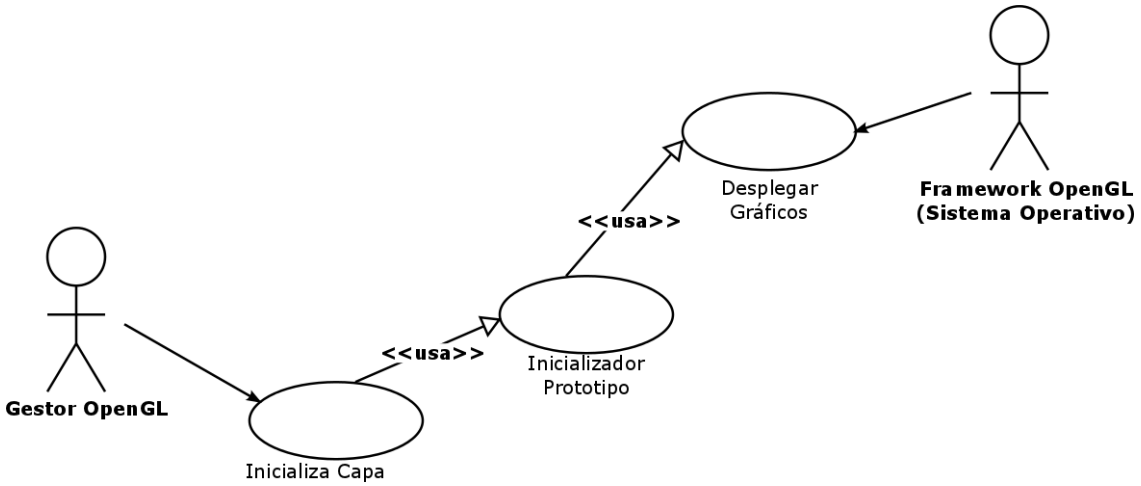


Figura 3.17 Diagrama de caso de uso de la gestión de escenas.

El ***Gestor OpenGL*** recibe la petición de inicializar escena e inspecciona el inicializador específico, el cual contará con los métodos y variables propias de la escena. Al entrar a este inicializador, debe interactuar con el inicializador prototipo y los comandos de reinicio de superficie, los cuales son la interacción con la superficie *OpenGL*; el objetivo del reinicio de la superficie es, a grandes rasgos, el de borrar el contenido presente, liberar memoria, y retirar métodos de actualización presentes en la superficie.
Al momento de retirar todas estas variables y métodos, la superficie queda disponible para atender los comandos presentes en el inicializador específico de la escena, de manera que los nuevos recursos y métodos intervengan en el momento en que son solicitados.
El proceso que describe la interacción entre los diferentes actores, al realizar la navegación entre escenas, se muestra en el diagrama de la Figura 3.18.

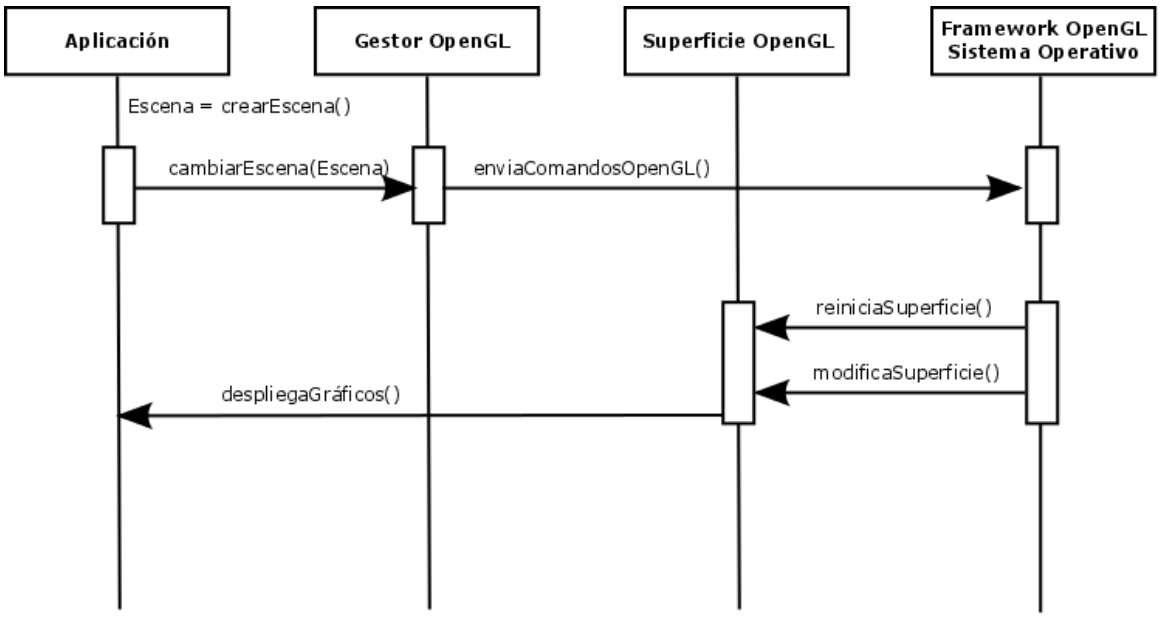


Figura 3.18 Diagrama de interacciones en la transición de escenas.

Es interesante observar que la escena se crea dentro de la aplicación, esta referencia se envía al ***Gestor OpenGL***, el cual mapea las instrucciones (las cuales son funciones de alto nivel, expuestas por el motor de juegos), hacia primitivas *OpenGL*, las cuales son enviadas al *Framework OpenGL* presente en el sistema operativo.
Éste es el encargado de solicitar el reinicio de la superficie para preparar la misma, y la modifica, en base a los comandos generados por el mapeo anterior. Por último, la superficie despliega los gráficos que son mostrados a nivel aplicación. Este proceso se puede observar de manera ampliada en la Figura 3.19.

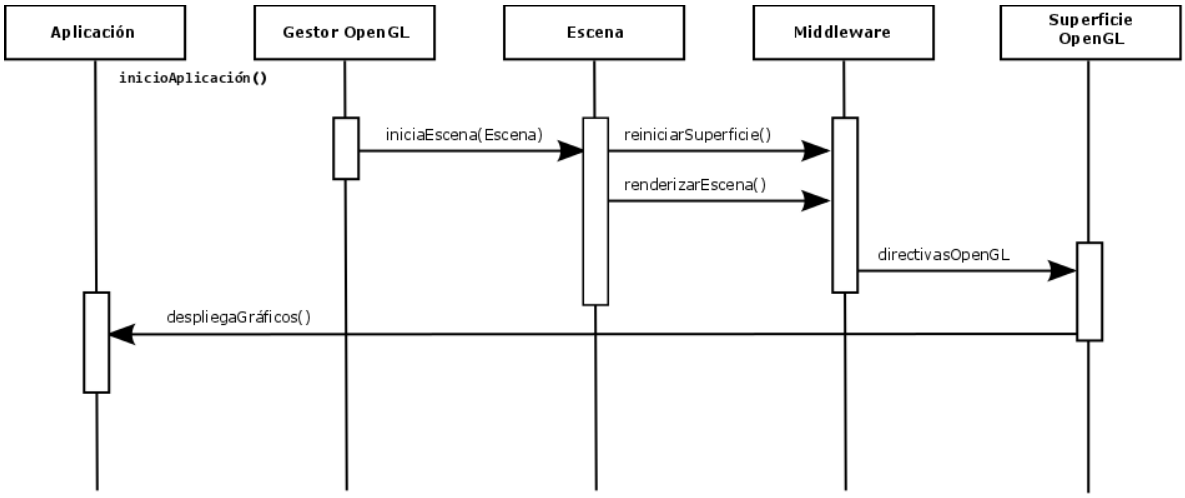


Figura 3.19 Envío de mensajes de *renderizado* a través de la escena.

Se observa la inclusión de un actor entre la superficie y la escena, denominado como middleware. Este código es uno de los elementos principales del motor, destinado a tomar los mensajes de alto nivel (especificados en la escena) y convertirlos a directivas *OpenGL*; hasta definir su implementación, este módulo se comportará como una caja negra, cuya función principal es tomar las instrucciones de alto nivel, y convertirles a directivas *OpenGL*, las cuales realicen el enlace con el GPU. La Figura 3.20 ejemplifica este proceso.

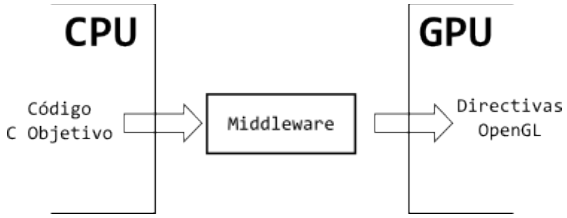


Figura 3.20 Transición a base del Middleware.

3.7 Mecanismos de Interfaz humana.

El objetivo primordial de un videojuego, es la interacción entre el jugador y el sistema de manera que es necesario establecer los mecanismos mediante los cuales el usuario interactuará con la lógica del juego en cada una de las escenas que este presenta.

En secciones anteriores, determinamos que la lógica del juego se encuentra inmersa en la escena y en cada escena que se despliega en pantalla, se establecen las reglas que deben seguirse para avanzar en el juego. Por ello, es necesario determinar como actuar en cada una de ellas. Es lógico pensar que la interacción con el menú inicial y una escena de juego es completamente diferente. Sin embargo, a través del documento hemos expresado tajantemente, que *OpenGL* no cuenta con mecanismos para interactuar con los elementos de entrada de datos.

Es por ello, que se propone la utilización de un enlace entre la escena, y la capa de la aplicación que pueda recabar la información de entrada de datos, presente en la entidad de control: el ***Gestor OpenGL***.

Regresando a la definición del gestor, observamos que contamos con los elementos suficientes para tomar este objeto como enlace de los módulos de entrada de datos y lógica de la escena. El Gestor puede ser accesado desde cualquier lugar (ya sea la vista principal de la aplicación, que es donde el sistema operativo recaba los datos de toques y acelerómetro, o la escena en ejecución en un momento dato). Esto permite realizar una conexión entre estos 2 objetos. La Figura 3.21 ejemplifica este proceso.

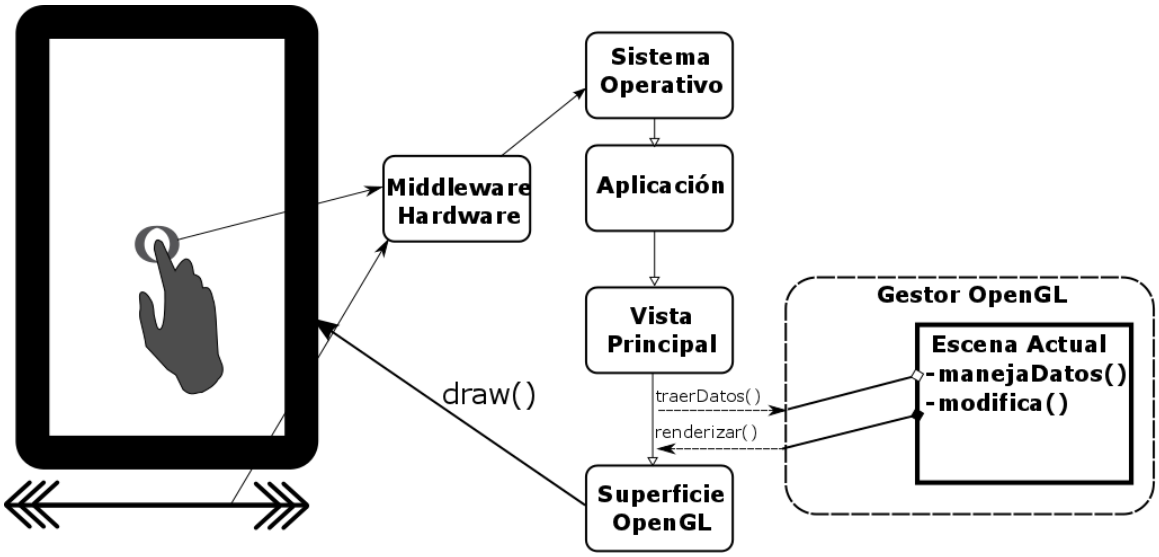


Figura 3.21 Mecanismos de entrada de datos por parte del usuario.

El primer mecanismo que intercepta la entrada del usuario es el hardware. Éste comunica al sistema operativo informándole que acaba de ocurrir un evento de entrada de datos (toque en pantalla, acelerómetro). El sistema operativo notifica a la aplicación en ejecución (de manera transparente al usuario), arrojando un mensaje directamente a la vista que se encuentra en pantalla. (vista principal). Debido a que el motor corre sobre la vista principal de la aplicación, es hasta este punto donde tenemos control sobre la entrada de datos. El siguiente paso es interceptar estos datos, por medio del gestor, y enlazarlos a la escena actual, la cual realizará el procesamiento de los datos y arrojará las modificaciones (si es que existen) directamente hacia la superficie *OpenGL*, que finalmente desplegará la información que le haya sido enviada.

Durante este capítulo se han explicado los procedimientos que se requieren implementar para la construcción del motor de juegos propuesto. De igual manera, definimos una entidad auxiliar, el ***Gestor OpenGL***, clase de vital importancia para la conexión entre los diferentes módulos de la arquitectura del motor. El siguiente capítulo, tendrá como objetivo la descripción de los procedimientos para la implementación de los módulos de la arquitectura, y sus relaciones con el gestor.

Capítulo 4

IMPLEMENTACIÓN DEL MOTOR DE JUEGOS

Establecidos los patrones que seguirá el motor de juegos propuesto, es necesario definir las clases principales, métodos de acceso e inicializadores necesarios para poner en marcha las diferentes funciones del mismo. En este capítulo, se describirán las clases principales, así como los métodos y propiedades que integrarán las mismas. De igual manera, se definen los elementos principales, archivos de configuración, *frameworks* y cabeceras necesarias para el correcto funcionamiento del motor de juegos todo esto; de igual manera se describirá la implementación de la entidad de control denominada ***Gestor OpenGL***.

4.1 Núcleo del sistema.

Siguiendo la estructura del capítulo 3, iniciaremos la implementación definiendo el núcleo del sistema. Como se definió en la sección 3.4, el núcleo del sistema puede ser interpretado como una etapa de configuración e inicialización, con el fin de preparar el entorno para el correcto funcionamiento del motor. En el motor propuesto, este proceso se utiliza para inicializar la entidad de administración del motor. El ***Gestor OpenGL***.

4.1.1 Inicialización del ***Gestor OpenGL*** (*OGLManager*).

El ***Gestor OpenGL*** (definido como el objeto *OGLManager* dentro de la arquitectura del motor), es la entidad encargada de realizar la inicialización del entorno *OpenGL* dentro de la aplicación. Para ello, necesita establecer los parámetros de la misma, inicializar los *shaders* y establecer la vista *OpenGL*. Es importante recordar que el gestor se implementa bajo un esquema *singleton*, de manera que sólo una instancia de este objeto puede existir durante la vida de la aplicación, lo que le permite ser accesada por cualquiera de las clases que integran la estructura del motor.

El primer paso es inicializar la instancia del manejador. Para respetar una arquitectura *singleton*, es necesario inicializar una instancia estática del objeto, y que esta inicialización sólo se realice una vez durante la vida de la aplicación.

```
+ (instancetype)sharedInstance {
  static dispatch_once_t pred;
  static OGLManager *_sharedInstance;
  dispatch_once(&pred, ^{ _sharedInstance = [[self alloc] init]; });
  return _sharedInstance;
}
```

Figura 4.1 Código Inicialización del *singleton*.

El código de la figura 4.1 inicializa la instancia estática del objeto, utilizando el procedimiento estándar de objetos en *CocoaTouch*. De esta manera se aloja memoria para este objeto (**`alloc`**) , y se inicializa con el inicializador prototipo (el método **`init`**). Dentro del inicializador prototipo, el primer procedimiento será inicializar la vista *OpenGL* con los parámetros del contexto, así como la inicialización de los *shaders*. El código de la Figura 4.2 muestra este procedimiento.

```
- (instancetype)init {
  if ((self = [super init])) {
      [self createShader];
  }
  return self;
}
```

Figura 4.2 Código del inicializador prototipo.

La función **createShader**[******] se compone con las instrucciones del código 4.4, destinado a compilar y enlazar los *shaders*.

Después de la inicialización, el gestor debe de configurar la vista *OpenGL* sobre la cual se trabajará. Para ello, toma la vista principal de la aplicación (contenida en el controlador principal), realiza la configuración y la almacena de manera que, a lo largo de la aplicación, se tenga acceso a esta vista de manera transparente. El código de la Figura 4.3 muestra la configuración de la vista dentro del ***Gestor OpenGL***.

```
- (void)configureOpenGLView:(GLKView*)aView{
    self.view           = aView;
    self.view.context   = [[EAGLContext alloc]
                                initWithAPI:kEAGLRenderingAPIOpenGLES2];
    self.view.drawableDepthFormat = GLKViewDrawableDepthFormat16;
    [EAGLContext setCurrentContext:self.view.context];
}
```

Figura 4.3 Código de configuración de la vista para el manejo de *OpenGL*.

El código muestra el proceso de almacenamiento de la vista dentro de una variable dentro del manejador, su configuración como vista *OpenGL* versión 2.0, la profundidad utilizada es la default, asignada con la configuración **GLKViewDrawableDepthFormat16**. Por último, configurada la vista, se utiliza el contexto de la misma como el contexto a utilizar durante la vida de la aplicación.

Como se observa, la inicialización del entorno resulta sencillo y se lleva a cabo con pocas líneas de código, esto debido a la utilización de la vista **GLKView** como superficie *OpenGL*.

4.2 *Renderizado* de Bajo Nivel.

El *renderizado* de modelos utilizando el motor de juegos, se compone de una serie de procedimientos específicos, que se presentan a continuación. Como primer paso, se analizará el desplegado de modelos tridimensionales, de manera que el concepto pueda ser utilizado como preámbulo del nodo escena.

4.2.1 *Renderizado* de Modelos 3D utilizando *OpenGL*.

Los modelos en 3 dimensiones se componen de una serie de puntos que, de manera organizada, son utilizados para crear rejillas, de manera que se generan superficies que representan objetos en 3 dimensiones. *OpenGL* cuenta con una serie de primitivas, destinadas exclusivamente a la estructuración de estos datos, de manera que la generación de los modelos se realice en orden.

[******] *La creación y definición de los shaders se presenta en la sección 4.2.*

OpenGL utiliza como convención, el manejo de triángulos como figura geométrica base para el *renderizado* de los modelos que maneja dentro de su funcionamiento.
En el ciclo de vida de una aplicación *OpenGL*, para crear uno de estos triángulos, simplemente es necesario el enviar primitivas con la información pertinente hacia el *GPU* (la posición de cada uno de los vértices del triángulo, y el color de relleno de la figura generada, por ejemplo). Sin embargo, dado que el motor de juegos expuesto en este documento se implementa sobre un entorno de mayor nivel (*iOS OS,* sobre *CPU*), es necesario contar con mecanismos de enlace que permitan la interacción entre éstos 2 entornos. Estos mecanismos se definieron en el capítulo anterior como *shaders*.

4.2.1.1 Desarrollo del *shader*.

El desarrollo del shader particular para el motor de juegos, se realiza sobre el lenguaje de sombreado *OpenGL* (*GLSL* por sus siglas en inglés). En el capítulo anterior se han definido los shaders, sus clasificaciones y funciones; en el desarrollo del presente trabajo, sólo se realizarán desarrollos sobre 2 *shaders* únicamente: el ***Vertex Shader*** y el ***Fragment Shader***, aún así, existen otros *shaders*, con características y propiedades diferentes que pueden ser utilizados para el *renderizado* de modelos[††††††].

4.2.1.1.1 Desarrollo del *Vertex Shader*.

Este *shader* es utilizado, primordialmente, para almacenar las características de forma de los modelos. Tiene como objetivo el almacenar la posición del objeto, su vector normal (utilizado para el escalamiento del modelo), las coordenadas sobre las cuales se deben mapear las texturas, de igual manera es utilizado para realizar el procesamiento del modelo con su entorno, de manera que es afectado en tiempo real por el estado de la aplicación. La estructura básica del *shader*, la podemos observar en el código de la Figura 4.4[‡‡‡‡‡‡]

La estructura de un *shader* se asemeja a un programa escrito en cualquier lenguaje de programación. Como un programa, tiene como función el tomar datos de entrada, realizar procesamiento, obtener resultados, y direccionar estos hacia el destino correcto; por lo cual se cuenta con un cuerpo principal delimitado por la función **`main`** (anotación 5).

```
uniform highp mat4 u_ModelViewMatrix;        // §§§§§§
uniform highp mat4 u_ProjectionMatrix;       //

attribute vec3     a_Position;               //1
attribute vec4     a_Color;                  //2
varying lowp vec4  frag_Color;               //3
varying lowp vec3  frag_Position;            //4

void main(void) {                            //5
      frag_Color      = a_Color;             //6
      gl_Position     = u_ProjectionMatrix*u_ModelViewMatrix*a_Position;   //7
      frag_Position   = (u_ModelViewMatrix*a_Position).xyz;                //8
      frag_TexCoord   = a_TexCoord;                                        //9
      frag_Normal     = (u_ModelViewMatrix * vec4(a_Normal, 0.0)).xyz;     //10
}
```

Figura 4.4 Código básico con la estructura del Vertex Shader.

[††††††] *Ver sección 3.5.3*
[‡‡‡‡‡‡] *El código se ha sintetizado para mostrar los elementos clave de un shader.*
[§§§§§§] *La utilidad y definición de estas matrices se definirá en la sección 4.2.2.3 Modificación de las propiedades del nodo*

En la anotación 1 y 2 podemos ver atributos que representarán variables de entrada, indicando la posición del vértice y el color al momento de crearse. Internamente, el *shader* recaba la información de los vértices haciendo uso de primitivas *OpenGL*, las cuales realizan un mapeo de los datos de entrada a estas variables[*******], por lo cual, al inicio del **`main`** del *shader*, estas variables contienen la información de los vértices del modelo fuente, esto se puede ver claramente en la anotación 8, la cual realiza una operación utilizando los valores almacenados en la variable **`a_position`** (la cual es un vector de longitud 3).
Las anotaciones 3 y 4 muestran un tipo de variable diferente, la cual permite el envío de información de un *shader* a otro. Particularmente en la anotación 3, esta variable simplemente toma el valor de la variable de entrada y la almacena, de manera que la información proveniente directamente del modelo pasará al siguiente *shader*[†††††††].

Es necesario comentar que los el pipeline de los *shaders*, no permite interactuar mas que en un solo sentido, por lo que el primer *shader* es el único que puede accesar al objeto fuente; de igual manera, el *vertex shader* solo puede enviar información al siguiente shader.

En el caso de la anotación 4, el *shader* realiza un procesamiento sobre la variable, antes de enviarla al siguiente *shader* (anotación 8).
En la anotación 7, podemos observar la variable **`gl_position`**, la cual no se encuentra declarada dentro del entorno. Esta variable es una propiedad directa del vértice, que expone hacia *OpenGL* la posición final del mismo de manera que, al aparecer dentro del entorno de *renderizado* de una superficie (en el caso del motor, la superficie *OpenGL*), será identificada como una primitiva directa, desplegándola y realizando el *renderizado*. Este proceso aplica a cualquier modificación realizada dentro de un *shader*, si en algún momento se asigna un valor a una propiedad directa que pueda ser interpretada por las primitivas, esta se expone hacia la máquina de estados de OpenGL.

Las anotaciones finales (9 y 10), describen las modificaciones que se realizan sobre los atributos restantes propios del *Vertex Shader* (textura y vector normal). En el caso de la textura, simplemente se asignan los valores coordenados que mapean la imagen que envuelve el modelo (recordemos el ejemplo de la sábana, descrito en la sección 3.5.2). Con ello, el vértice tendrá su asociación con la coordenada de la imagen. En el caso del vector normal, este si recibe modificaciones por parte de la matriz de proyección, debido a que la misma afecta las propiedades del vértice.

4.2.1.1.2 *Fragment Shader*.

Como se describió en secciones anteriores, este *shader* es el encargado de procesar cada pixel, de manera independiente, para agregarles las características necesarias y obtener un comportamiento tridimensional. En este *shader* se comprenden características tales como color, textura, iluminación, perspectiva, etcétera. El código de la Figura 4.5 muestra una versión sintetizada del *Fragment Shader*.

En la anotación 1 podemos observar la variable que el *shader* anterior dejó disponible para ser interceptada por los *shaders* siguientes, de esta manera el color, un elemento que solo puede ser manejado mediante este shader, puede ser afectado por los parámetros del entorno, tales como los diferentes tipos de luz y los efectos de profundidad.

[*******] *Este mecanismo se mostrará en la sección 4.2.1.3.1.*
[†††††††] *Para fines prácticos, solo se expresará que el vertex shader se encuentra antes del fragment shader en el orden del pipeline*

Adicional a lo anterior *shader*, se encarga de tomar los valores de entrada (los cuales provienen del entorno donde se encuentra inmerso el modelo) y realiza las operaciones sobre cada pixel (proveniente del *vertex shader*), de manera que al final el color real en el pixel del modelo se modifica de acuerdo a éstos parámetros.

```
varying lowp vec4 frag_Color;                        // 1
 ...
struct Light {
  lowp vec3 Color;
  lowp float AmbientIntensity;
  lowp float DiffuseIntensity;
  lowp vec3 Direction;
};
uniform Light u_Light;

void main(void) {

  // Ambient
  lowp vec3 AmbientColor    = u_Light.Color * u_Light.AmbientIntensity;

  // Diffuse
  lowp vec3 Normal          = normalize(frag_Normal);
  lowp float DiffuseFactor  = max(-dot(Normal, u_Light.Direction), 0.0);
  lowp vec3 DiffuseColor    = u_Light.Color * u_Light.DiffuseIntensity *
DiffuseFactor;

  // Specular
  lowp vec3 Eye             = normalize(frag_Position);
  lowp vec3 Reflection      = reflect(u_Light.Direction, Normal);
  lowp float SpecularFactor = pow(max(0.0, -dot(Reflection, Eye)), u_Shininess);
  lowp vec3 SpecularColor   = u_Light.Color * u_MatSpecularIntensity *
SpecularFactor;

  gl_FragColor              = u_MatColor * texture2D(u_Texture, frag_TexCoord) *
vec4((AmbientColor + DiffuseColor + SpecularColor), 1.0);       // 2
}
```

Figura 4.5 Código básico con la estructura del Fragment Shader.

Después de estas operaciones, observamos una variable de tipo **`gl_FragColor`**, la cual, al igual que **`gl_position`** en el *shader* anterior, puede ser interpretada por las primitivas de OpenGL directamente a la superficie.

Al analizar el funcionamiento de los shaders, podemos observar que cumplen las funciones que se describieron en el capítulo 3. El vertex shader modifica las propiedades del vértice y las expone hacia *OpenGL* por medio de **`gl_position`**, modificando la forma y disposición de los vértices, mientras que el *fragment shader* modifica la composición del pixel, tales como color, sombreado y profundidad a través de **`gl_FragColor`**.

En las secciones anteriores se ha descrito el funcionamiento de los shaders, sin embargo no se han definido los mecanismos que enlacen estas porciones de código hacia el motor. La siguiente sección se encarga de definir estos procedimientos.

4.2.1.2 Enlace, compilación y ejecución del *shader*.

Como se comentó en el inicio del capítulo, el *shader* se encuentra escrito completamente sobre un lenguaje basado en *OpenGL* (*GLSL*). Esto lo hace independiente del sistema operativo (adecuado al hardware compatible a *OpenGL*). El problema viene cuando las aplicaciones no se realizan simplemente sobre este lenguaje (como es el caso del motor de juegos). Para sortear este problema, es necesario establecer un método de enlace entre el lenguaje del *shader* y el lenguaje en el que se desarrolla la aplicación. Esta sección se encarga de describir este proceso.
El primer paso es verificar la compatibilidad de los 2 sistemas (en el caso del motor, este paso se prueba por definición al existir el *framework OpenGL* en *Cocoa Touch*), el segundo es crear un mecanismo de enlace entre el archivo que contiene las instrucciones del *shader*, y la aplicación. Esto se realiza utilizando *Objective-C*, llevando a la memoria de la aplicación el contenido del archivo y compilándolo, haciendo uso de *Cocoa-OpenGL*. (Ver código de la Figura 4.6)

```
- (GLuint)compileShader:(NSString*)shaderName withType:(GLenum)shaderType {  //0
  NSString* shaderPath      = [[NSBundle mainBundle] pathForResource:shaderName
ofType:nil];                    //1
  NSError* error;
  NSString* shaderString    = [NSString                          //2
     stringWithContentsOfFile:shaderPath encoding:NSUTF8StringEncoding
error:&error];
  if (!shaderString) {
    // Manage error!
    exit(1);
  }

  GLuint shaderHandle = glCreateShader(shaderType);
     //3

  const char * shaderStringUTF8 = [shaderString UTF8String];
  int shaderStringLength        = [shaderString length];
  glShaderSource(shaderHandle, 1, &shaderStringUTF8, &shaderStringLength);
      //4

  glCompileShader(shaderHandle);                                          //5

  GLint compileSuccess;
  glGetShaderiv(shaderHandle, GL_COMPILE_STATUS, &compileSuccess);
  if (compileSuccess == GL_FALSE) {
     // Manage error!
    exit(1);
  }
  return shaderHandle;
}
```

Figura 4.6 Código básico para la generación y compilación de un *shader*.

En la anotación 0, definimos el nombre del método. Se solicitan como parámetros el nombre del archivo que contiene el código del *shader* y el tipo del mismo de manera que el primer paso es suministrar el nombre del archivo que lo almacena (para poder llamar el archivo en base a su nombre), y el tipo de *shader*, para enviar al *framework* el tipo de compilación que necesita realizar (recordemos que existen varios tipo de *shader*). La anotación 1 y 2 muestran las instrucciones para accesar al archivo que contiene el *shader*. En caso de encontrarlo, se procede a la creación de un descriptor para guardar la referencia del *shader* que vamos a generar.

La anotación 3 muestra la creación de una variable que pueda almacenar el tipo de *shader* que queremos. La anotación 4 muestra el proceso para enlazar el archivo del *shader* con la aplicación, utilizando el contenido del archivo y la variable en la cual queremos almacenar el resultado, con esto obtenemos una referencia al *shader* que queremos compilar (anotación 5).

Este procedimiento se realizará por cada *shader* que generamos (en nuestro caso, dos). *OpenGL* permite la creación de una entidad de orden mayor, capaz de enlazar los *shaders* para conjuntarlos y que sea mas sencilla su administración; esta entidad se conoce como manejador de programa. Esto resulta de gran ayuda, debido a que facilita la escalabilidad en el número de *shader* con los que se está trabajando. El código de la Figura 4.7 muestra el proceso, de manera simplificada, para adjuntar varios *shaders* a un manejador de programa.

```
- (void)compileVertexShader:(NSString *)vertexShader
            fragmentShader:(NSString *)fragmentShader {                    //0
  GLuint vertexShaderName   = [self compileShader:vertexShader
                                       withType:GL_VERTEX_SHADER];         //1
  GLuint fragmentShaderName = [self compileShader:fragmentShader
                                         withType:GL_FRAGMENT_SHADER];     //2

  _programHandle = glCreateProgram();                                      //3
  glAttachShader(_programHandle, vertexShaderName);                        //4
  glAttachShader(_programHandle, fragmentShaderName);                      //5

  glBindAttribLocation(_programHandle,VertexAttribPosition,"a_Position");  //6
     . . .
  glLinkProgram(_programHandle);                                          //7
     . . .
  GLint linkSuccess;
  glGetProgramiv(_programHandle, GL_LINK_STATUS, &linkSuccess);
  if (linkSuccess == GL_FALSE) {
     // Manage error!
  }
}
```

Figura 4.7 Código para generar el manejador global de los 2 *shaders* generados.

La anotación 0 nos indica el nombre del método, al cual enviaremos como parámetros los nombres de los 2 archivos que contienen nuestros *shader*. La anotación 1 y 2 sirven para generar los descriptores para cada uno de los *shaders*, de manera que guardamos una referencia hacia ellos. La anotación 3 se encarga de la creación del manejador, mientras que las anotaciones 4 y 5 indican que los *shaders* se deben adjuntar al manejador antes creado. Por último, es necesario enlazar el manejador para que puede ser interpretado por la máquina de estados en tiempo de ejecución (el equivalente a crear un objeto ejecutable que puede ser leído o utilizado por la máquina de estados).
La anotación 7 resulta de sumo interés en el funcionamiento de los *shaders*. La instrucción señalada por esta anotación tiene como objeto el enlazar los datos provenientes del archivo fuente hacia el *shader*. Analizando los parámetros podemos ver que toma el manejador de los *shaders*, el cual servirá como la unidad para procesar la información. El siguiente parámetro indica un offset dentro de una estructura, la cual será utilizada como un mapeo hacia el archivo fuente, el cual se comportará como un arreglo de vértices; el último parámetro representa el nombre de la variable en la que se almacenarán los datos por cada uno de los atributos[‡‡‡‡‡‡‡].

[‡‡‡‡‡‡‡] *Ver Sección 4.2.1.1.1*

Al definir el manejo de las variables dentro del *shader* (sección 4.2.1.1) y su enlace desde la aplicación (descrito en los párrafos anteriores de esta sección), ahora es necesario detallar las estructuras de datos que intervienen en este proceso.

4.2.1.3 Composición de un modelo tridimensional.

En la sección anterior se describió el procedimiento (a grandes rasgos) para enlazar, generar y compilar un *shader*, asimismo definimos que el *shader* será el encargado de enviar la información del CPU hacia el GPU y realizar el *renderizado*. Sin embargo, hasta este momento no se ha definido como es que cualquier modelo tridimensional podrá ser usado como la fuente de información para un *shader*. Para ello, realizaremos un pequeño paréntesis, y revisaremos el tema correspondiente a los modelos tridimensionales.

Un modelo tridimensional es un objeto creado a partir de un conjunto de puntos en el espacio 3D, que están conectados por diversos datos geométricos tales como líneas, y superficies curvas. Actualmente, existen varios editores y procesadores de gráficos que permiten el diseño y elaboración de este tipo de modelos, todos ellos generados en base a métodos matemáticos ampliamente definidos, para fines de este documento, nos centraremos en el método conocido como modelo poligonal *[31]*.

El modelado poligonal es un método para crear un modelo 3D mediante la conexión de segmentos de línea a través de puntos en un espacio 3D. Estos puntos en el espacio también se conocen como vértices. Los modelos poligonales generados bajo este método son muy flexibles y pueden ser presentados por un equipo de cómputo con gran rapidez, sin embargo, dentro de sus limitaciones, tenemos la imposibilidad de crear una superficie curva exacta. Como contraparte a esta delimitación, tenemos la posibilidad de generar, en base a los archivos de datos planos, estructuras de datos ordenadas que muestren la disposición de estos vértices y polígonos, de manera que podemos leer la información de su forma y composición, inspeccionando esta estructura.

El modelado poligonal dio paso al surgimiento de un estándar en la industria que pudiera homologar la caracterización de modelos tridimensionales en un archivo bien definido: el archivo *.obj[32]*

Un archivo obj define la geometría y propiedades de los modelos tridimensionales, y los almacena en un archivo plano que puede representarse en un formato ASCII altamente estructurado. Según su especificación, el formato *obj* describe, en su contenido, una lista con 4 tipos de información de vértices. El código de la Figura 4.8 muestra una pequeña porción de código obtenida del un modelo tridimensional exportado en formato *.obj*.

```
mtllib monkey.mtl                          //1
o monkey_Suzanne                           //2
v 0.437500 -0.765625 0.164062              //3
 . . .
vt 0.631521 0.895237                       //4
. . .
vn -0.155869 -0.987170 -0.034638           //5
. . .
f 198/274/142 71/275/142 138/276/142       //6
. . .
```

Figura 4.8 Entradas de un archivo *obj*.

La anotación 1 simplemente nos indica el archivo que utiliza el modelo para la parte de material. Esta definición corresponde a un archivo que describiremos en párrafos subsecuentes. La anotación 2 nos indica el nombre del modelo mismo (archivo de salida). La anotación 3 inicia con la información relativa a la composición modelo.

La letra inicial **v**, indica que este renglón corresponde a la posición del vértice (con sus componentes x, y, z). La entrada **vt** indica el vector relacionado al mapeo de la textura con el mismo. La siguiente entrada (**vn**) representa el vector normal, el cual define un vector perpendicular desde el punto hacia la superficie. La funcionalidad primordial de un vector normal es la de establecer cuanta luz recibe este vértice. Por último, se listan las entradas correspondientes a las caras del modelo (**f**). Como su nombre lo indica, representan los vértices de las caras que formas los polígonos base (triángulos). Estas entradas son tercias de puntos, denotadas por el índice correspondiente a su orden en la entrada **v**. Así el primer índice de la cara del código de la Figura 4.8, se compone por la componente x del vértice 198, la componente y del vértice 274, y así respectivamente.

Un archivo *obj* se verá enlazado a un archivo que describa el material del modelo, complementando la información del mismo, de manera que podemos establecer el comportamiento del objeto con su entorno (efectos de luz, composición, reflectividad, etc), el archivo se define como un archivo tipo *mtl[33]*. La importancia de este archivo es que al definir un material, podemos establecer el comportamiento del objeto ante un efecto del entorno de vital importancia como lo es la luz. La Figura 4.9 muestra un ejemplo de las entradas presentes en una rchivo mtl.

```
newmtl monkey                          //1
Ns 96.078431                           //2
Ka 0.000000 0.000000 0.000000          //3
Kd 0.640000 0.640000 0.640000          //4
Ks 0.500000 0.500000 0.500000          //5
Ni 1.000000                            //6
map_Kd monkey.png                      //7
```

Figura 4.9 Entradas de un archivo *mtl*.

La anotación número 1 presenta el nombre del material (lo ideal es mostrar consistencia entre el nombre del archivo y el material). La anotación 2 presenta la entrada correspondiente al exponente especular para el material actual (denotado por la entrada **Ns)**. Esto define el foco de la iluminación especular hacia cada punto del modelo. La anotación 3 corresponde a la reflectividad del ambiente del material (**Ka**), la anotación 4 (**Kd**) corresponde a la reflectividad difusa del material, mientras que la entrada (**Ks**) representa la reflectividad especular.
La anotación 6 muestra la entrada correspondiente al índice de refracción del material (**Ni**). Por último, se define la imagen que es utilizada como material del objeto, bajo la entrada **map_Kd**[§§§§§§§]

La información que representan estos archivos es la necesaria para el correcto funcionamiento de los *shaders*, sin embargo, es necesario realizar procesamiento sobre estos archivos con la finalidad de automatizar el proceso de exportación de los datos hacia los *shaders*. La siguiente sección describe el proceso utilizado para realizar este procesamiento.

4.2.1.3.1 Procesamiento sobre el archivo fuente obj y mtl y la estructura Vertex.
Una vez que analizada la composición del archivo obj, es necesario procesar los datos para obtener estructuras que permitan regenerar los modelos en base a esta información. La Figura 4.1 muestra el procesamiento realizado sobre el archivo obj para obtener la información ordenada de manera conveniente al shader[********].

[§§§§§§§] *La definición y características de los parámetros anteriores no se discuten en el presente documento, debido a que no son el tema central del mismo.*

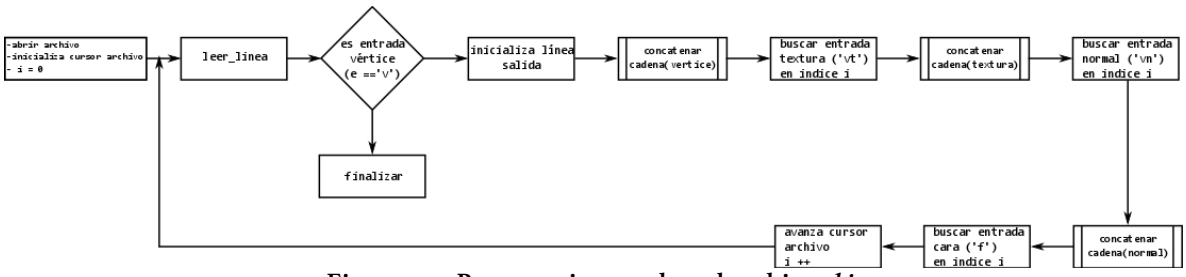


Figura 4.10 Procesamiento sobre el archivo *obj*.

Como se puede observar, el proceso consiste en reordenar la información del archivo, de manera que se integre, en una sola línea, toda la información correspondiente a cada vértice del modelo. El código de la Figura 4.10 muestra la salida que se obtiene de este procesamiento en un vértice en particular.

```
{{-0.500000,-0.500000,0.000000},{1,1,1,1},{0.308658,0.5000000},{-0.547559,-0.819480,-0.169205}}
```

Figura 4.11 Ejemplo de entrada de vértice en el archivo de salida del procesamiento.

Teniendo esta cadena, generada por cada uno de los vértices presentes en el modelo contenidos en un archivo, el siguiente paso es crear una estructura que pueda almacenar, respetando el orden, la información de cada vértice, y enviarla a través de *shader* hacia el *GPU*.
Para ello, se desarrolló una estructura complementaria, que permita almacenar la información. El código de la Figura 4.11 muestra la definición de esta estructura, a la cual nombramos *Vertex*.

```
typedef enum {                          //0
  VertexAttribPosition = 0,             //1
  VertexAttribColor,                    //2
  VertexAttribTexCoord,
  VertexAttribNormal                    //3
} VertexAttributes;

typedef struct {
  GLfloat Position[3];                  //4
  GLfloat Color[4];                     //5
  GLfloat TexCoord[2];                  //6
  GLfloat Normal[3];                    //7
} Vertex;
```

Figura 4.12 Código de la definición de la estructura de datos *Vertex*.

En la anotación 0, observamos la definición de un enumerador. Este enumerador es utilizado para desplazarnos a través de la estructura. Observamos la anotación 1. Este elemento del enumerador, señala el inicio de la estructura (la posición cero). Esto indica que el arreglo **Position[3]**, se encuentra en un *offset* 0 (cero) dentro de la estructura de datos. Los siguientes enumeradores (anotación 1 a 3), se refieren a los *offset* de las demás propiedades (color, textura y normal), nótese que no están definidos de manera literal, esto debido a que, en caso de necesidad de incrementar los atributos en esta estructura, no se necesite modificar estos valores (la asignación de los valores para estos denominadores se realizará en tiempo de compilación). La anotación 4 describe el vector de la posición (vector tamaño 3, para la posición x,y,z). La anotación 5 describe el color (con una nomenclatura *rgba*), la anotación 6 describe las coordenadas de textura, mientras que el vector normal (anotación 7) cuenta con 3 entradas para cada componente del eje ordenado tridimensional.

Regresando al código de la Figura 4.7, (sección 4.2.1.2), se podemos complementar el código posterior a la anotación 6. El código de la Figura 4.13 extiende el las entradas faltantes.

```
glAttachShader(_programHandle, vertexShaderName);                          //4
glAttachShader(_programHandle, fragmentShaderName);                        //5

glBindAttribLocation(_programHandle, VertexAttribPosition,"a_Position");  //6
glBindAttribLocation(_programHandle, VertexAttribColor,   "a_Color");     //7
glBindAttribLocation(_programHandle, VertexAttribTexCoord,"a_TexCoord");  //8
glBindAttribLocation(_programHandle, VertexAttribNormal,  "a_Normal");    //9
. . .
glLinkProgram(_programHandle);
```

Figura 4.13 Código correspondiente al enlace de la estructura con los atributos del *shader*.

La anotación 6, como anteriormente se describió, representa un mapeo entre el atributo del *shader* **`a_Position`**, y el atributo dentro de la estructura, con un offset representado por medio del enumerador **`VertexAttribPosition`**. Podemos observar que tanto el atributo **Position** dentro de la estructura **`Vertex`**, como el atributo **`a_position`** dentro del *vertex shader*, corresponden a un arreglo de capacidad 3; esta correspondencia, aun cuando sea en diferentes lenguajes (*C* y *GLSL*), permite que el mapeo entre variables se realice de manera transparente, en lo que a tamaño se refiere.

Por medio de este procedimiento, todas y cada una de las variables del modelo pueden ser enlazadas desde el entorno de la aplicación (*CPU*) hacia el *shader* (*GPU*), de manera que el *renderizado* se realice de manera correcta.

Dentro del *renderizado* de bajo nivel, el elemento primordial del motor es el nodo base, el cual hará uso de los mecanismos de *renderizado* descritos anteriormente para desplegarse en la superficie *OpenGL*.

4.2.2 Generación del objeto base (GENode).

El elemento principal dentro de la arquitectura del motor de juegos es el objeto nodo. Este objeto es un prototipo, del cual heredarán los objetos que se crean dentro del motor. Su principal función es definir un objeto base que el motor pueda manejar e interpretar, y que establezca métodos y propiedades comunes a todos los elementos dentro del mismo.

4.2.2.1 Inicialización del objeto GENode.

El definir propiedades y métodos comunes en una clase prototipo permite integrar, de manera sencilla, diferentes objetos que simplemente adecúen estas propiedades a sus necesidades y, en caso de ser necesario, agreguen más. De esta manera, los objetos pueden definir sus particularidades, sin perder la generalización que los caracteriza debido a su pertenencia al motor.
En base a las características presentes en los modelos tridimensionales y las necesidades del motor de juegos, se definen las propiedades principales que presentará el nodo prototipo. El código de la Figura 4.14 muestra las propiedades que presenta el objeto prototipo.

La anotación 0 nos señala el nombre del método. Analizando su estructura, observamos que es necesario proporcionar el *shader*.

Esto debido a que, sin importar que tipo de objeto sea, este prototipo representará las mismas reglas para el *renderizado* e interpretación de la información concerniente a sus vértices y pixeles.

Simplemente es referenciar el *shader* presente en el ***Gestor OpenGL***, de manera que se tenga uniformidad en el mecanismo de *renderizado*

```
-   (instancetype)initWithName:(char    *)name    shader:(BaseEffect    *)shader
vertices:(Vertex *)vertices vertexCount:(unsigned int)vertexCount {    //0
  if ((self = [super init])) {                                         //1
    _name                   = name;                                    //2
    _vertexCount            = vertexCount;                             //3
    _shader                 = shader;                                  //4
    _texture                = nil;                                     //5
    self.position           = GLKVector3Make(0, 0, 0);                 //6
    self.rotationX          = 0;                                       //7
    self.rotationY          = 0;                                       //8
    self.rotationZ          = 0;                                       //9
    self.scale              = 1.0;                                     //10
    self.matColor           = GLKVector4Make(1, 1, 1, 1);              //11
    self.children           = [NSMutableArray array];                  //12
       . . .
  }
  return self;
}
```

Figura 4.14 Código del método de Inicialización del objeto GENode.

La anotación 1 muestra un inicializador prototipo de *CocoaTouch*. Esto simplemente para alojar memoria y crear un objeto reconocible dentro del *framework*. La anotación 2 muestra una propiedad denominada **name**. Esta propiedad tiene como función la ayuda en la búsqueda de objetos cuando estos son agregados como hijos a un objeto. La siguiente anotación representa el conteo de vértices. Esta propiedad tiene como función, el conocer el número de vértices que integran el modelo, para poder *renderizar* el modelo de manera correcta. La anotación 5 representa la carga de la textura para el modelo en particular. Esta propiedad solo es utilizada por nodos caracterizados por modelos tridimensionales[††††††††].

Las anotaciones 6 a 11, contienen propiedades que servirán como modificadores en tiempo de ejecución para el nodo. Estas propiedades representarán las interfaces que el usuario del motor (el programador de videojuegos) deberá de modificar para obtener los resultados que desee sobre los nodos. Estas modificaciones se realizarán en cada actualización antes de se ejecute el método encargado de *renderizar* el modelo. La siguiente sección se encarga de describir este procedimiento.

4.2.2.2 *Renderizado* del objeto GENode.

El *renderizado* de los modelos se realiza utilizando como matriz de modificación, la correspondiente a la matriz del objeto que tienen asociado como padre, y asociándola con la matriz del nodo en cuestión. Como se ha expresado en las secciones anteriores, los modelos son objetos tridimensionales caracterizados por vértices, asociados entre si. La manera de llevar a cabo modificaciones en ellos, es realizando operaciones algebraicas simples (tales como multiplicaciones, sumas o inversiones) sobre los vértices, de manera que sus características se modifiquen, esto se logra utilizando funciones preestablecidas por *OpenGL*. El código de la Figura 4.15 muestra el método de *renderizado* para el modelo dentro de su estructura.

[††††††††] *El procedimiento de carga de texturas se discutirá en la sección 4.2.4; correspondiente a la creación de nodos de tipo modelo tridimensional*.

```
- (void)renderWithParentModelViewMatrix:(GLKMatrix4)parentModelViewMatrix { //0
  GLKMatrix4 modelViewMatrix = GLKMatrix4Multiply(parentModelViewMatrix,    //1
      [self modelMatrix]);
  for (GENode *child in self.children) {                                    //2

    [child renderWithParentModelViewMatrix:modelViewMatrix];                //3
  }
  _shader.modelViewMatrix   = modelViewMatrix;                              //4
  _shader.texture           = self.texture;                                 //5
  _shader.matColor          = self.matColor;                                //6
  [_shader prepareToDraw];                                                  //7
. . .
}
```

Figura 4.15 Código para el *renderizado* del modelo.

En la anotación 0 observamos la declaración del método. Como parámetro, se espera la matriz del nodo padre, que será la que realice las modificaciones al modelo en base al entorno. La notación 1 se encarga de modificar la matriz del modelo (accesada mediante la instrucción **`[self modelMatrix]`**), multiplicándola por la matriz del nodo padre. De ésta manera, los cambios del modelo padre, se verán reflejados hacia los hijos en cascada. La anotación 2 muestra, precisamente, como el nodo inspecciona su lista de hijos, propagando la matriz hacia un nivel menor (anotación 3). De esta manera, se sigue la cadena de modificaciones hasta el objeto más abajo en el árbol de hijos. Por último, las anotaciones 4 a 6, envían al *shader* la información necesaria para realizar el *renderizado*.

4.2.2.3 Modificación de las propiedades del nodo.

En la sección anterior, se describió la manera en que los nodos padre propagan los cambios que realizan a través de sus hijos. Sin embargo, no se abarcaron las modificaciones que se realizan, particularmente, a cada nodo. Esta característica resulta de vital importancia, dado que en ella se basa la caracterización de un motor de juegos: la posibilidad de modificar los nodos que se encuentran en su una escena utilizando las directivas propias del motor. En la sección 4.2.1, se enumeraron las propiedades que los nodos pueden utilizar para modificar su forma (posición, escala, rotación, color, etc.), sin embargo, no se ha descrito el mecanismo que realizan, en tiempo de ejecución, estas modificaciones. El código de la Figura 4.16 muestra el método utilizado para generar la matriz del modelo del nodo.

```
- (GLKMatrix4)modelMatrix {
  GLKMatrix4 modelMatrix    = GLKMatrix4Identity;                           //1
  modelMatrix   = GLKMatrix4Translate(modelMatrix,self.position.x,
self.position.y, self.position.z);                                          //2
  modelMatrix   = GLKMatrix4Rotate(modelMatrix, self.rotationX, 1, 0, 0);   //3
  modelMatrix   = GLKMatrix4Rotate(modelMatrix, self.rotationY, 0, 1, 0);   //4
  modelMatrix   = GLKMatrix4Rotate(modelMatrix, self.rotationZ, 0, 0, 1);   //5
  modelMatrix   = GLKMatrix4Scale(modelMatrix, self.scale, self.scale,
self.scale);                                                                //6
  return modelMatrix;
}
```

Figura 4.16 Código para la generación de la matriz del modelo.

La anotación 1 muestra la inicialización de la matriz de identidad. Esta inicialización es importante, debido a que si no existen modificaciones explícitas sobre el modelo del nodo, debe de generarse la matriz necesaria para resultar en una salida sin modificaciones.

Las anotaciones 2 a 6 muestran las directivas *OpenGL* que se utilizan para realizar las modificaciones sobre el modelo. De esta manera, los cambios que el usuario realiza sobre el modelo, pasarán directamente hacia la matriz del mismo, modificando su forma[‡‡‡‡‡‡‡‡].

4.2.2.4 Actualización del objeto GENode.

Una de las características principales de los videojuegos, es la capacidad de realizar animaciones o movimientos en los objetos, durante la ejecución del mismo. Por ello, es necesario la generación de mecanismos que permitan la actualización de las propiedades de los objetos (forma y posición) durante el tiempo de vida de la aplicación.
Por ello, el nodo requiere de un mecanismo que permita realizar este proceso. El código de la Figura 4.17 muestra el funcionamiento de este mecanismo.

```
- (void)updateWithDelta:(NSTimeInterval)dt {
  for (GENode *child in self.children) [child updateWithDelta:dt];
}
```

Figura 4.17 Código del método de actualización del modelo.

Analizando el modelo, se puede observar una analogía al método de *renderizado*. Sin embargo, se observa una diferencia respecto al parámetro que se espera. En este método observamos que el parámetro que se nos presenta es un intervalo de actualización, el cual es la pieza clave de la actualización de los objetos.
La piedra angular de la animación y el desarrollo de videojuegos es la creación de bucles infinitos de actualización donde se modifiquen los elementos en pantalla. El despliegue de recursos, la reproducción de multimedia y la interacción del usuario con el sistema basa su funcionamiento en la continua 'espera' de datos por parte del sistema. Esto se realiza utilizando mecanismos tan simples, como el agendado de bucles que realicen el sensado en intervalos de tiempo bajos. Entre más bajo es este tiempo, mayor será la capacidad de reacción por parte de la aplicación. En esta característica se basa el método de actualización de los nodos, utilizar el intervalo de actualización del sistema (específicamente, de la pantalla), para la actualización de las propiedades de los nodos[§§§§§§§§].
En este método, el usuario puede modificar las propiedades de los objetos, de manera que esos cambios se vean reflejados en tiempo de ejecución basados en el tiempo de la misma.

4.2.3 Generación del objeto escena (GEScene).

La escena es uno de los elementos primordiales en la construcción del motor de juegos. Dentro de su arquitectura, no podría existir ningún juego sin el concepto de la misma, por lo cual, el definir este elemento en el motor es de suma importancia. El nodo escena, en su caracterización más básica, implementa el inicializador descrito en el código de la Figura 4.18.

El nodo escena inicia con un arreglo de vértices nulo, definiendo el número de los mismos en cero. Esto toma lógica dado la naturaleza abstracta de la clase, debido a que no existe como un elemento, sino como un espacio de tiempo en el cual se atiende a ciertas características, con un grupo de modelos *renderizados* en particular, aún así, se encuentra a de definir el color de la pantalla en su espacio de ejecución (anotación 1), permitiendo modificar este valor en el momento de inicializarla.

[‡‡‡‡‡‡‡‡] *El funcionamiento y descripción de las operaciones matriciales realizadas para cada operación sobre el modelo, no se describen en el presente trabajo, debido a que quedan fuera del alcance del mismo. Para mayor referencia, puede consultar la entrada [34] de la bibliografía.*
[§§§§§§§§] *El concepto de actualización de pantalla, se discutirá en la sección 4.3.1*

```
- (instancetype)initSceneWithColor:(Color)color{
    if ((self = [super initWithName:"Scene" shader:[[OGLManager sharedInstance]
getShader] vertices:nil vertexCount:0])) {
        backgroundColor = color;                                          //1
    _gameArea           = CGSizeMake(9, 16);                              //2
    _sceneOffset        = _gameArea.height/2 /
                              tanf(GLKMathDegreesToRadians(90.0/2));      //3
    self.position       = GLKVector3Make(-_gameArea.width/2,
                             -_gameArea.height/2 + 10, -_sceneOffset);    //4
    }
    return self;}
```

Figura 4.18 Código del inicializador del nodo escena proporcionando color.

La propiedad de color se enviará directamente hacia la vista por medio del ***Gestor OpenGL***[*********]. En la anotación 2, observamos un atributo denominado área de juego (**_gameArea**). El área de juego es una superficie sobre la cual poblaremos nuestros nodos. Uno de los grandes retos dentro del desarrollo de videojuegos en dispositivos móviles se presenta en el establecimiento correcto de las superficies disponibles, en base a los dispositivos de salida *[35]*; por lo tanto, esta superficie debe establecerse en función de los requerimientos del juego a desarrollar utilizando como base cuantos elementos lo poblarán y las posiciones de los mismos. Para la configuración default de la escena se seguirá la siguiente estrategia: analizando el dispositivo sobre el cual se realizan las pruebas (iPhone 5), utilizaremos como base la relación de aspecto presente en su pantalla (9:16). Teniendo estos valores en consideración, podríamos generar una rejilla de 144 elementos, una cantidad aceptable para la realización de las pruebas pertinentes. Como configuración default, se establecerán estos valores. El siguiente paso es obtener el valor del eje z (anotación 3), que permita generar, en perspectiva, la superficie que acabamos de definir. Utilizando geometría básica, sabemos que es posible obtener, por medio de la función tangente, el valor del cateto adyacente en un triángulo rectángulo, utilizando el valor del cateto opuesto y el ángulo agudo del mismo. Con esto se lograría obtener el valor de profundidad necesario para que la superficie calculada anteriormente corresponda a la proyección que se obtenga de este offset. Es conveniente indicar que el valor del ángulo utilizado corresponde al valor de 45 grados, un valor utilizado ampliamente en el campo de los videojuegos como el campo de vista adecuado para despliegue de modelos en pantalla*[36]* . La anotación 4 establece la posición de la escena de juego. Es necesario fijar la posición de la escena justo al centro de la superficie de juego, debido a que, en el caso de las superficies *OpenGL*, la posición por omisión es el origen (0,0,0), y el punto de anclaje se encuentra situado en la esquina inferior izquierda.

Complementando el inicializador descrito anteriormente, existe la posibilidad de inicializar una escena especificando la superficie que se utilizará como área de juego. El código de la Figura 4.19 es el encargado de realizar este procedimiento.

```
- (instancetype)initSceneWithColor:(Color)color andGameArea:(CGSize)anArea{
    if ((self = [super initWithName:"Scene" shader:[[OGLManager sharedInstance]
getShader] vertices:nil vertexCount:0])) {
     backgroundColor = color;
     _gameArea       = anArea;
     _sceneOffset    = _gameArea.height/2 /
tanf(GLKMathDegreesToRadians(90.0/2));
     self.position   = GLKVector3Make(-_gameArea.width/2, -_gameArea.height/2 +
10, -_sceneOffset);
     }return self;}
```

Figura 4.19 Código del inicializador del nodo escena proporcionando color.

[*********] *Ídem*.

Podemos observar que simplemente tomamos el parámetro del área de juego proporcionado para calcular los valores correspondientes al área de juego.

4.2.3.1 Manejo de entrada de datos por medio del objeto GEScene.

4.2.3.1.1 Detección de toques en la escena.

La escena de juego representa un estado de la aplicación durante el cual, todos los elementos deben corresponder a las acciones definidas por la misma. Uno de estos elementos es la entrada de datos. Durante una escena se espera que el usuario realice acciones que modifiquen la aplicación, de manera que es necesario generar mecanismos que nos permitan interceptar estas interacciones y caracterizarlas de acuerdo a las necesidades específicas de la escena. Los métodos para la gestión de entrada de datos por medio de toques en la pantalla táctil se muestran en el código de la Figura 4.19.

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event { }
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event { }
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event { }
```

Figura 4.20 Código para el método de entrada de datos.

Estos métodos corresponden a los definidos por *CocoaTouch* sobre cualquier vista dentro del *framework*. Sin embargo, es necesario la exposición de estos métodos hacia la vista, de manera que, al recibir mensajes de entrada de datos, los notifique hacia la escena en ejecución[††††††††]. El código de la Figura 4.20 muestra este enlace.

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {          //0
  UITouch *touch          = [touches anyObject];                           //1
  CGPoint touchLocation   = [touch locationInView:[OGLManager
sharedInstance].view];                                                     //2
  touchLocation           = [self touchLocationToGameArea:touchLocation];  //3
}
```

Figura 4.21 Código para la detección de toques en pantalla por medio de la escena.

La anotación 0 nos indica que el código corresponde al método **touchesBegan** (el procedimiento es aplicable a cualquiera de los 3 métodos de entrada de datos).
El primer paso es tomar cualquier toque proveniente de la vista (todos ellos almacenados en el **NSSet** utilizado como parámetro). Posteriormente (anotación 3) obtenemos la posición del toque, haciendo uso de la vista *OpenGL* y el objeto **UITouch***[37]*. En este punto, es necesario realizar la adecuación de la posición obtenida en la vista hacia la nueva superficie de juego (generada en la sección anterior). Para ello, simplemente calculamos la proporción existente entre estas 2 posiciones. Este proceso lo muestra el código de la Figura 4.22

```
  float actualX     = touchLocation.x / ratioX;
  float actualY       = ([OGLManager sharedInstance].view.frame.size.height -
touchLocation.y) / ratioY;                                  //1
  CGPoint actual    = CGPointMake(actualX, actualY);
  return actual;
}
```

Figura 4.22 Conversión de posición global a posición de la superficie de juego.

[††††††††] *La interacción del motor de juego con la vista GLKView, se describirá en la sección 4.3.1*

Un detalle muy conveniente de observar, es la necesidad de invertir la dirección de incremento en el eje y, en la vista *OpenGL* a la superficie de juego; mientras que la vista *OpenGL* tiene el punto de anclaje en la esquina inferior izquierda, las vistas en *CocoaTouch* manejan el punto de anclaje en la vista superior izquierda, incrementando el valor del eje y en proporción decreciente a la pantalla.

4.2.3.1.2 Detección de aceleración en el dispositivo (acelerómetro).
Otro elemento de entrada de datos en el dispositivo móvil es el sensor de aceleración integrado al mismo. Al igual que en el caso de la detección de toques en pantalla, la obtención de la información de este componente se realizará a través de la vista de la aplicación. El código de la Figura 4.23 muestra la implementación del método que recupera la información del acelerómetro dentro de la escena.

```
- (void)getAcceleration:(CMAcceleration)acceleration{     }
```

Figura 4.23 Código Método para obtener la información de aceleración dentro de la escena de juego.

En este método, el objeto de tipo **`CMAcceleration`***[38]* cuenta con la información de aceleración del dispositivo en base a una estructura de datos para almacenar las3 direcciones de aceleración.
Al igual que en el caso de la entrada de datos a través de toques en pantalla, la información proveniente del acelerómetro será enviada de la vista *OpenGL* hacia la escena por medio del ***Gestor OpenGL***.

4.2.4 Generación del nodo modelo3D.

A diferencia del nodo escena, el nodo correspondiente a un modelo tridimensional necesita establecer, de manera particular, los mapas de vértices y texturas sobre los cuales se genera el mismo. Para ello, es necesario hacer uso del archivo generado en base al procedimiento que se presentó en la sección 4.2.1.3.1. El inicializador por default que presentaría un nodo de modelo tridimensional, se presenta en el código de la Figura 4.24.

```
- (instancetype)initWithShader:(BaseEffect *)shader {
  if ((self = [super initWithName:"Example"
                          shader:shader
                        vertices:(Vertex *)Example_Vertices
vertexCount:sizeof(Example_Vertices)/sizeof(Example_Vertices[0])])) {     //0
[self loadTexture:@"Example.png"];                                        //1
 . . .
}
  return self; }
```

Figura 4.24 Código del inicializador del nodo modelo 3D.

La anotación 0 muestra la referencia que se tiene hacia el archivo que contiene los vértices del modelo tridimensional. Asimismo, se utiliza la propiedad que define la textura del modelo, sólo presente en nodos correspondientes a modelos tridimensionales.

Es importante señalar que la carga de modelos y el archivo de salida esperado deben empatar completamente con las especificaciones del motor de juegos, de otra manera, no se garantiza el funcionamiento del mismo.

4.3 Mecanismos de interfaz humana.

4.3.1 Interacción con la vista *GLKView*.

Definidos los elementos dentro del motor de juegos, es necesario describir ahora la integración de los mismos con la vista de la aplicación, lo cual representaría el enlace final con *CocoaTouch*.
El primer paso consiste en definir la vista como una vista de tipo *GLKView*, de manera que podamos implementar los métodos correspondientes a su controlador (*GLKViewController*), y asignarla como la vista del ***Gestor OpenGL***. El código de la Figura 4.25 muestra este procedimiento.

```
- (void)viewDidLoad                                                    //0
{
    [OGLManager sharedInstance].view = self.view;                      //1
    Color col = {0, 0.0/255.0, 255.0/255.0};                           //2
    GEScene *scene = [[GameScene alloc] initSceneWithColor:col];       //3
    [OGLManager sharedInstance].scene = _scene;                        //4
    . . .
    [super viewDidLoad];
}
```

Figura 4.25 Código para la asignación de la vista e inicialización de la escena.

La anotación 0 señala que este procedimiento se realiza en el método que se dispara cuando la vista es cargada en memoria. El primer paso es inicializar el ***Gestor OpenGL*** (se inicializa inherentemente al llamar un método sobre su instancia estática). Posteriormente se genera una estructura de color para enviarla al inicializador de la escena. La escena es iniciada y asignada a la variable de la escena actual dentro del ***Gestor OpenGL***. Con este procedimiento se concluye la inicialización del motor de juegos dentro de la aplicación.
Las instrucciones de *renderizado* de cada unos de los elementos presentes en la escena en ejecución en el momento de la inicialización están listas para ser enviadas al *framework* para iniciar el proceso de despliegue; sin embargo, la superficie sobre la cual se estará dibujando, es la encargada de enviar los mensajes de *renderizado* hacia todos los elementos del motor. Este mecanismo se produce en cada llamada de despliegue que la vista recibe de su controlador. El código de la Figura 4.26 muestra este proceso.

```
- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect {                  //0
    Color col = [[[OGLManager sharedInstance] scene] getColor];            //1
    glClearColor(col.r, col.g, col.b,1.0);                                 //2
    GLKMatrix4 viewMatrix = GLKMatrix4Identity;                            //3
    [[OGLManager sharedInstance].scene
                              renderWithParentModelViewMatrix:viewMatrix]; //4
    . . .
}
```

Figura 4.26 Código del método de *renderizado* de pantalla sobre *GLKViewController*.

La anotación 0 indica el nombre del método encargado de la actualización de *renderizado*. La siguiente anotación (1) obtiene el color establecido en la escena actual del gestor para dibujar ese color sobre la superficie de la vista.
Por último, se crea una matriz identidad, la cual será propagada desde la escena actual hacia sus nodos hijos, de manera que el efecto cascada inicie en el preciso momento en que este método es llamado (anotaciones 3 y 4).

Al igual que el método de actualización de *renderizado*, existe un método de actualización, el cual envía mensajes desde el controlador hacia la vista. Este método forma parte del protocolo **`CLKViewDelegate`***[39]* y se muestra en el código de la Figura 4.27.

```
- (void)update {
   [[OGLManager sharedInstance].scene updateWithDelta:self.timeSinceLastUpdate];
}
```

Figura 4.27 Código del método de actualización dentro de la clase *GLKViewController*.

En este momento podemos observar la validez del método descrito en la sección 4.2.4. Si la vista envía mensajes de actualización en un determinado tiempo (tiempo fijado por el sistema en tiempo de ejecución), los nodos pueden atender a esta actualización, y modificar las propiedades de sus modelos, para lograr efectos de animación o movimiento.

4.3.2 Detección de toques en pantalla.

Por último, la vista es la superficie encargada de gestionar las entradas de datos (por acción del sistema operativo), para ello es necesario establecer un enlace entra la escena en ejecución (que define las respuestas a la entrada de datos), y la vista (la cual se encargará de detectar esta entrada de datos). El mecanismo que realiza este procedimiento se presenta en el código de la Figura 4.28.

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    [[OGLManager sharedInstance].scene touchesBegan:touches withEvent:event];
}
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    [[OGLManager sharedInstance].scene touchesMoved:touches withEvent:event];
}
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    [[OGLManager sharedInstance].scene touchesEnded:touches withEvent:event];
}
```

Figura 4.28 Métodos de entrada de datos por medio de toques en pantalla y enlace a escena.

4.3.3 Detección de toques en pantalla.

De manera análoga, la información proveniente del acelerómetro, recuperada por la vista, es enviada hacia la escena por medio del gestor.

```
-(void)outputAccelertionData:(CMAcceleration)acceleration{
    [[OGLManager sharedInstance].scene getAcceleration:acceleration];
}
```

Figura 4.29 Método de entrada de datos por medio del acelerómetro y enlace a escena.[‡‡‡‡‡‡‡‡‡]**.**

[‡‡‡‡‡‡‡‡‡] *Para una explicación mas detallada acerca del proceso de configuración para la entrada de datos a través del acelerómetro del dispositivo, refiérase al Anexo 1.*

Resumen

A lo largo de esta capítulo, se han presentado los mecanismos de implementación del motor de juegos. Se han descrito las interacciones entre los diferentes actores dentro del motor (***Gestor OpenGL***, nodos, escenas, etcétera), de manera que el lector pueda entender como todos estos elementos se integran a la arquitectura del motor y permiten el funcionamiento del mismo.

De igual manera, se abordaron temas de interés especial, referentes al *renderizado* de modelos tridimensionales haciendo uso de *OpenGL*. El siguiente capítulo, tiene como objeto presentar al lector los resultados obtenidos, así como un ejemplo de integración e implementación del motor de juegos hacia una aplicación real, de manera que los conceptos de diseño y desarrollo sobre el mismo, sean entendibles en mayor forma.

Capítulo 5

PRUEBAS Y RESULTADOS

La definición del motor de juegos se ha presentado en capítulos anteriores. En este punto, el lector tiene un conocimiento acerca de la arquitectura, diseño y composición del mismo. Sin embargo, es necesario establecer el procedimiento necesario para desarrollar aplicaciones utilizando el mismo. El presente capítulo tiene como objeto mostrar las funcionalidades presentes en el motor de juegos desarrollado, por lo cual se dividirá en dos secciones, el desarrollo de una aplicación demo, la cual servirá a manera de prueba de concepto y funcionalidad del sistema, y la comparación del motor contra un motor de juegos existente.

5.1 Descripción de la aplicación Demo.

La aplicación Demo consiste en un juego de video tipo arcade, el cual basa su lógica en esquivar los elementos que caen de la parte superior de la pantalla para evitar que choquen con el objeto principal, el cual es controlado por el usuario a través del acelerómetro del dispositivo. Los principales actores que se presentarán son los siguientes:

1. Objeto principal (usuario)
2. Elementos enemigos (obstáculos).

En las secciones siguientes, se describirá el procedimiento para generar este videojuego haciendo uso del motor de juego desarrollado en el presente documento.

El objetivo de la aplicación demo es la presentación de los módulos presentes en el motor de juegos, y que componen la lógica de un videojuego creado en base al mismo. Se presenta la funcionalidad del mecanismo de gestión de escenas, el renderizado de un modelo tridimensional, el manejo de entrada de datos hacia el dispositivo y la interacción con funcionalidades del sistema operativo.

5.1.1 Preparación del entorno.

El primer paso es preparar un entorno de desarrollo dentro del *IDE XCode* propicio para el desarrollo de una aplicación haciendo uso del motor de juegos. El anexo 1, sección 4 proporciona las instrucciones necesarias para la generación de este entorno.

5.1.2 Generación de la escena de juego.

El inicio de una aplicación desarrollada haciendo uso del motor requiere, como punto de partida, la generación de una escena de juego. Como se ha descrito anteriormente, la escena se define como el objeto lógico de despliegue de recursos, así como el encargado de manejar la lógica de la aplicación (desde mecanismos de entrada de datos, hasta particularidades de la funcionalidad del juego).

El primer paso es crear el archivo **GameScene**, definiéndolo como una clase que hereda del objeto **GEScene**§§§§§§§§§

Posteriormente, tenemos la rotación de la escena de juego completa, en 20 grados sobre el eje x.

El código de la Figura 5.1 muestra este procedimiento.

```
- (instancetype)initSceneWithColor:(Color)color andGameArea:(CGSize)anArea{
    if ((self = [super initSceneWithColor:color andGameArea:anArea])) {
        self.rotationX      = GLKMathDegreesToRadians(-20);
    }
    return self;
}
```

Figura 5.1 Código para la inicialización de la escena de juego y rotación de la misma.

Esta rotación puede ser considerada un detalle estético, dado que nos brindará una perspectiva en 3 dimensiones. Recordando como funcionan los modelos tridimensionales, si nos situamos en un ángulo recto (en cualquiera de los tres planos coordenados) respecto a cualquier modelo, simplemente tendremos la perspectiva de una de sus caras, dándonos una vista bidimensional. El rotar la escena completa, permite ver la escena 'desde arriba', lo cual generará un *renderizado* de todos los modelos agregados a esta escena, bajo esa premisa. En secciones posteriores analizaremos el comportamiento de la escena al modificar estos valores.

El código de la Figura 5.1 desplegará la escena, la cual se muestra en la Figura 5.2

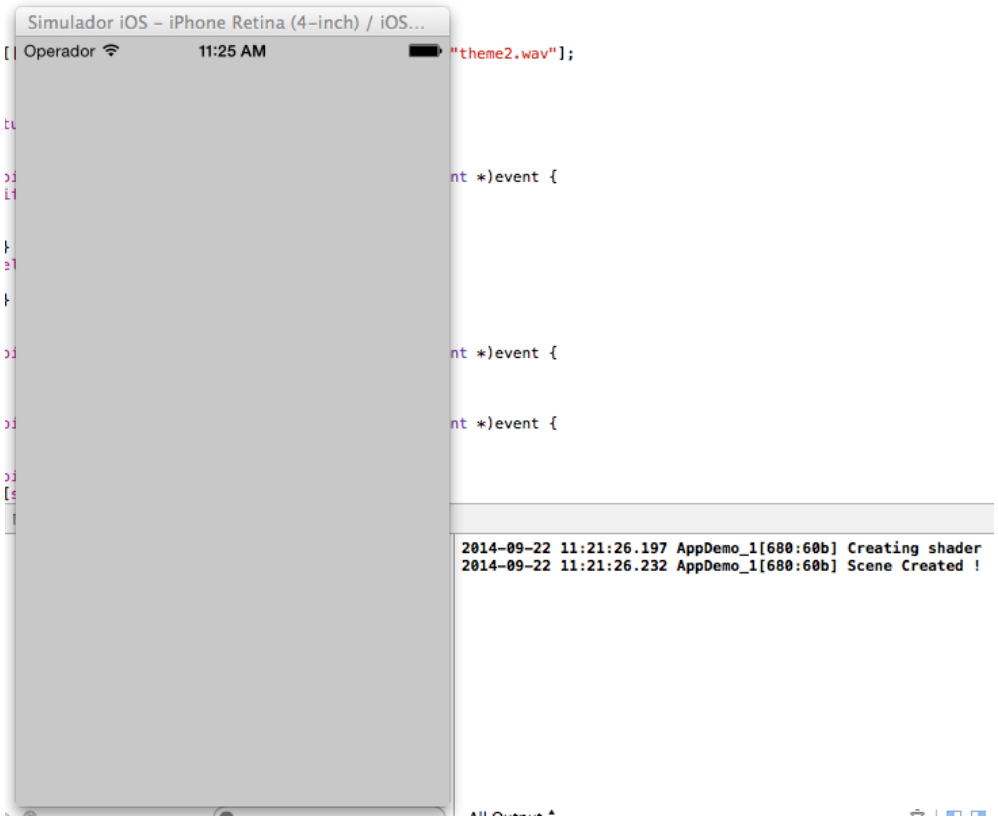


Figura 5.2 Visualización de la inicialización de la escena principal utilizando el simulador********.

§§§§§§§§§ *Ver Anexo 1, sección 4.*

********** *Para mayor referencia acerca del simulador, consulte la entrada [40] de la bibliografía.*

El mensaje que aparece en el log del entorno, muestra que la escena del motor ha sido creada satisfactoriamente.

5.1.2.1 Inicialización y agregado de un modelo tridimensional.

Dado que ya se ha inicializado la escena de juego, podemos agregar nodos a esta con la intención de *renderizarlos* a través de ella. El primer paso es generar un modelo tridimensional que representará el personaje del usuario; para ello, es necesario crear una clase de tipo modelo tridimensional, y asignarle un archivo fuente que lo caracterice[††††††††††]. En la aplicación demo, haremos uso de un modelo de carácter gratuito, que viene integrado dentro del entorno *Blender*, el modelo *monkey-suzanne*.

Para generar el modelo tridimensional es necesario descargar la última versión del editor de gráficos tridimensionales *Blender*[‡‡‡‡‡‡‡‡‡‡]. Una vez que se ha descargado el entorno, bastará con iniciarlo, y agregar un modelo de tipo monkey-suzzane en la escena.

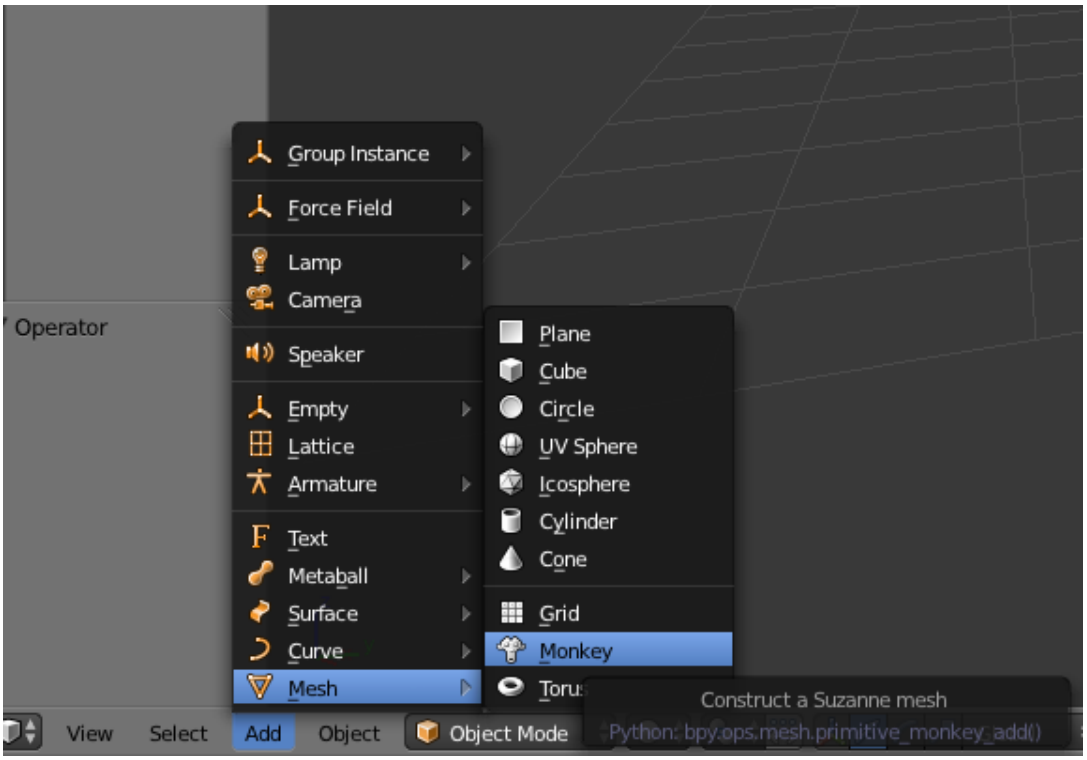


Figura 5.3 El modelo monkey-suzanne en *Blender*.

Al agregar el modelo podemos observar que es un modelo complejo (cuenta con varias aristas, vértices y formas) y no es necesario poseer conocimiento alguno de diseño para poder generarlo; el objeto, al ser agregado, es un modelo tridimensional completo en cuanto a forma (vértices y aristas)

Como se mencionó en secciones anteriores, para componer completamente un modelo trisimensional, es necesario establecer propiedades de material y textura, que acompañen a la forma geométrica del mismo. Con esto, podemos obtener los vectores que complementen la composición del modelo. Esto se puede consiguir utilizando el editor *Blender*, bajo los siguientes pasos:

[††††††††††] *Para más detalles acerca de este procedimiento, referirse al Anexo 1 sección 4.*

[‡‡‡‡‡‡‡‡‡‡] *(http://www.blender.org/download/)*

1. Agregar el material. Para agregar un material, utilizamos el menú de propiedades del modelo , y seleccionamos el ícono de material (Figura 5.4a) y agregamos un material (Figura 5.4b), el nombre que le asignaremos debe ser el mismo con el cual exportaremos el modelo al finalizar el proceso§§§§§§§§§§ (Figura 5.4c)

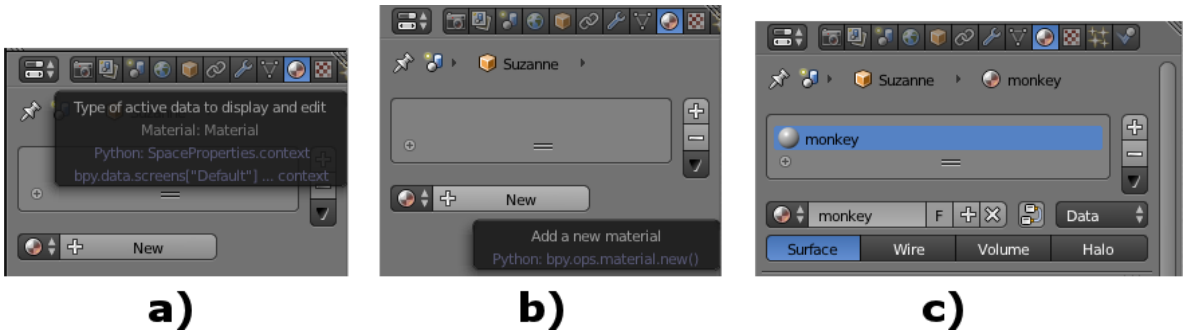


Figura 5.4 Agregado de material al modelo.

2. Generar la textura. Para agregar el archivo que sirva como textura del modelo tridimensional es necesario desenvolver la malla que envuelve el modelo tridimensional, de manera que esta malla pueda ser mapeada hacia una imagen bodimensional. El primer paso, es abrir otra ventana dentro de *Blender*, esto se realiza arrastrando la esquina superior derecha del editor hacia la izquierda, de manera que se descubra una nueva ventana (Figura 5.5).

Figura 5.5 Despliegue de una nueva ventana en *Blender*.

En la nueva ventana desplegada, seleccionamos la vista de tipo editor de imagen.

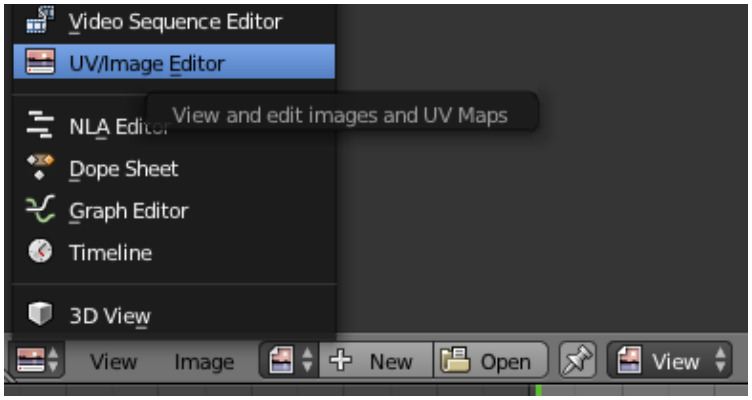


Figura 5.6 Cambio a vista editor de imagen.

Al contar con esta vista, podemos generar el archivo que será utilizado como textura en nuestro modelo. Dentro del Image Editor, seleccionamos Image->New Image.

§§§§§§§§§§ El asignar el mismo nombre al material y al modelo, es debido a que se recomienda presentar consistencia entre estos archivos que permita la manipulación correcta de los shaders sobre estos archivos

En la selección de color, seleccionamos los valores RGB para el color blanco (r=1, g=1, b=1), y como nombre seleccionamos el mismo nombre que utilizamos para el material (monkey). A continuación, guardamos la imagen (Image->Save Image) en una archivo bajo el nombre user_node.png. Es necesario guardar la imagen en la misma dirección donde exportaremos el modelo.

Por último, es necesario exportar el modelo en el formato requerido (obj). Para ello, seleccionamos el menú File-> Export-> Wavefront (.obj). Seleccionamos la ubicación donde almacenaremos todos los archivos y nombremos el archivo user_node.obj. En la configuración de exportación debemos de seleccionar *Include Normals* y *Triangulate Faces*. Esta información debe estar presente en el archivo de salida debido a que es necesaria para el correcto funcionamiento de los shaders. También es necesario seleccionar Change the *Forward* option to *Y Forward* and the *Up* option to *Z Up*, debido a que los marcos de referencias en el sistema coordenado de openGL son inversos alutilizado por *blender*.

Al final del proceso, debemos contar con 3 archivos, un archivo *mtl*, un archivo *obj* y un archivo *png*, todos ellos bajo el nombre user_node. Utilizando estos 3 archivos, se ejecuta el programa que descompone los modelos tridimensionales en estructuras de datos que contienen vértices y vectores mencionado en la sección 4.2.1.3.1

Despues del procesamiento sobre el modelo tridimensional, se tiene una estructura que puede ser leída por nuestros shaders, por lo cual simplemente es necesario agregarla en el código de la escena; para ello, se crea una variable dentro de la escena de juego que lo contenga (este objeto será modificado de manera recurrente mientras la escena se encuentre en ejecución). Posterior a ello, inicializaremos el objeto a través del inicializador prototipo y lo agregaremos como hijo a la escena de juego. (Figura 5.7).

```
@implementation GameScene {
  GEUserNode       *_userNode;
  . . .
- (instancetype)initSceneWithColor:(Color)color {
  if ((self = [super initSceneWithColor:color])) {
     _userNode          = [[GEUserNode alloc] initWithShader:
                          [[OGLManager sharedInstance] getShader]];
    _userNode.position    = GLKVector3Make(_gameArea.width/2,
                                           _gameArea.height * 0.05,
                                           0);
    _userNode.matColor    = GLKVector4Make(1.0, 0.5, 0.5, 1.0);
    [self.children addObject: _userNode];
```

Figura 5.7 Inicialización del objeto tridimensional e integración a la escena de juego.

La figura 5.8a muestra el resultado de agregar el nodo a la escena.

Como un ejercicio dentro del desarrollo de esta aplicación, la escala del modelo puede ser modificada alterando el valor del atributo **`scale`**, sobre el objeto usuario. La Figura 5.8b muestra el resultado de establecer este valor en 5 unidades (**`_userNode.scale = 5`**).
De la misma manera podemos afectar la rotación del objeto sobre un eje coordenado en este caso el eje Z (**`_userNode.rotationZ    = 2* M_PI_2`**). Modificando este valor, podemos ver como el nodo gira y muestra la parte posterior del modelo (Ver Figura 5.8c)

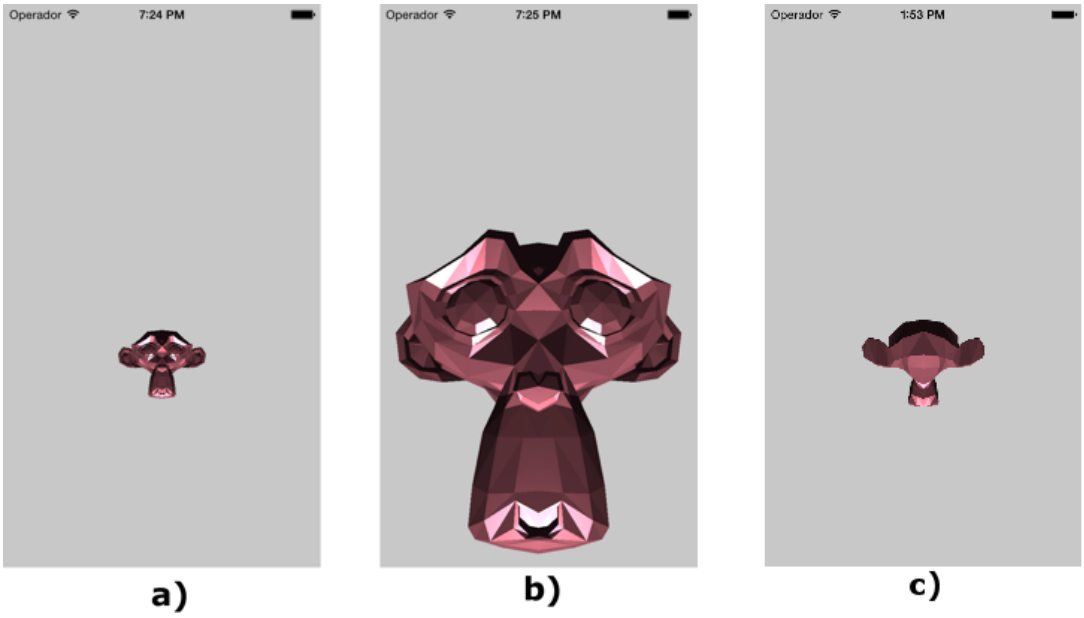


Figura 5.8 Inclusión de un modelo a la escena de juego.

En el caso de los objetos que interactuarán con el personaje, a manera de enemigos, es necesario generar objetos por cada uno de ellos, y almacenarlos dentro de un arreglo. Esto debido a que modificaremos su posición de manera continua, a través del bucle de actualización.

5.1.2.2 Modificación del bucle de actualización.

El bucle de actualización servirá como un mecanismo de temporizado, de manera que es posible, haciendo uso de variables auxiliares, establecer intervalos que disparen métodos y porciones de código específico. El código de la Figura 5.9 muestra este procedimiento.

```
- (void)updateWithDelta:(NSTimeInterval)dt {
    [super updateWithDelta:dt];                                          //0
    // Move the user node with the accelerometer
    float newX = _userNode.position.x + _accelerationInX;                //1
    if (newX <0)            {      newX = 0;           }
    if (newX > _xMargin)    {      newX = _xMargin;    }
    _userNode.position = GLKVector3Make(newX, _userNode.position.y,
_userNode.position.z);                                                   //2
    cicles ++;                                                           //3
    if (cicles == delayNextBall) {                                       //4
      . . .
      GEBall *enemy       = [[GEBall alloc] initWithShader:
                          [[OGLManager sharedInstance] getShader]];      //5
        enemy.position      = GLKVector3Make( _gameArea.width*xPos,      //6
                                              _gameArea.height*.65, 0);
        enemy.matColor      = GLKVector4Make(1.0, 1.0, 0, 1);            //7
        [self.children addObject:enemy];
        [_enemies addObject:enemy];
        cicles = 0;
    }
}
```

Figura 5.9 Código del método de actualización dentro de la escena de juego.

La anotación 0 muestra la llamada al método presente en la superclase. Las anotaciones 1 y 2 muestran el movimiento efectuado sobre el usuario, mediante una variable que recaba la información del acelerómetro[**********]. La siguiente anotación (3), indica una variable que sirve como un contador, cada que se entra al método de actualización, esta variable se incrementa, de manera que al encontrar la condición que establecimos como intervalo, podemos disparar instrucciones; en este caso, se agenda un enemigo en una posición aleatoria sobre el eje x, en la parte superior del área de juego (anotaciones 4 a 7).
Los enemigos deben moverse en dirección hacia el personaje, de manera que el usuario deba evitarlos. En el bucle de actualización de la capa de juego se agrega un enemigo en intervalos determinados, fijados por la lógica del juego. Para el movimiento de los enemigos, bastará con decrementar su posición sobre el eje y con el objeto de desplazarlo hacia el fondo de la pantalla, en cada entrada al método de actualización. Es necesario indicar que este código se encuentra en el bucle de actualización del objeto enemigo, no dentro de la escena.

El código de la Figura 5.10 muestra la modificación al bucle de actualización realizado sobre el objeto enemigo

```
- (void)updateWithDelta:(NSTimeInterval)aDelta {
    if (aDelta == 0) return;

    self.rotationY    += M_PI_4 * aDelta;                                    //1
    float newY      = self.position.y + _fallVelocity;                       //2
    self.position  = GLKVector3Make(self.position.x, newY, self.position.z); //3
    [super updateWithDelta:aDelta];                                          //4
}
```

Figura 5.10 Código del método de actualización dentro del objeto enemigo.

La anotación 0 muestra un efecto de rotación sobre el modelo de manera continua. La anotación 2 y 3 muestran la modificación de la posición del modelo en base a su método de actualización.
Esto muestra como el efecto en cascada del diseño del motor de juegos, permite establecer autonomía sobre los objetos, y afectar sus propiedades de manera específica, compartiendo un contexto mayor. La última anotación (4), simplemente llama el método de la superclase nodo

La Figura 5.11 muestra el efecto causado debido a estas modificaciones.

En la Figura 5.11a, se observa el objeto enemigo en la parte superior de la pantalla (justo al momento de aparecer). En cada ciclo de actualización su posición se verá alterada, de manera que se acercará hacia el fondo de la pantalla. La Figura 5.11b muestra la posición del enemigo después de un intervalo de tiempo, aproximándose gradualmente hacia la posición del usuario.

5.1.2.3 Detección de Colisiones.
Como se mencionó en capítulos anteriores, el motor de juegos no cuenta con un mecanismod e física y detección de colisiones. Sin embargo, haciendo uso de los atributos de los objetos, se pueden generar mecanismos, si bien limitados, suficientes para obtener detección de colisiones entre objetos tridimensionales.

[**********] *El obtención de la información del acelerómetro se discutirá en la siguiente sección.*

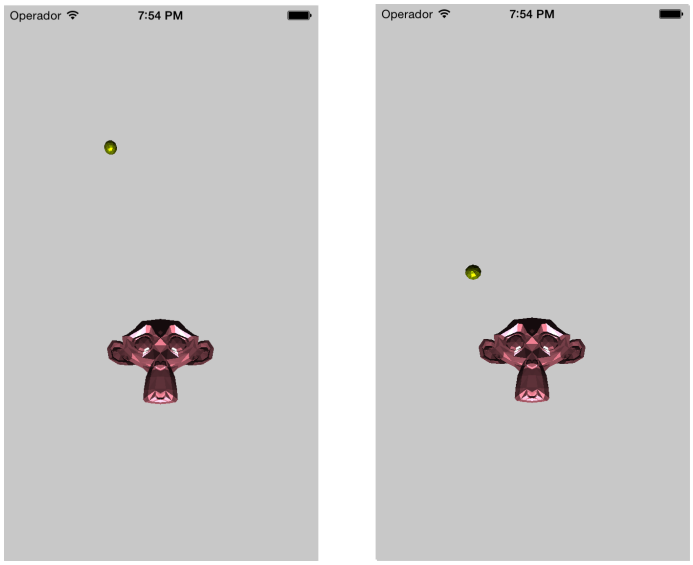


Figura 5.11 Movimiento de los enemigos en base a su bucle de actualización.

Para la detección de colisiones, se inspecciona la colección de enemigos (contenidos en un arreglo), y se determina la superficie que cada uno de estos describe en el plano coordenado x, y; posteriormente se determina la superficie descrita, sobre este mismo, por el objeto usuario; con esto, podemos determinar si las dos superficies se intersectan, indicando que los objetos chocan entre sí[††††††††††].
El código de la Figura 5.12 muestra el mecanismo descrito en el párrafo anterior.

```
CGRect userNodeRect = [_userNode
boundingBoxWithModelViewMatrix:GLKMatrix4Identity];                    //1
GEBall * ballToDestroy;
for (GEBall *enemy in _enemies) {                                      //2
     CGRect enemyRect = [enemy
                    boundingBoxWithModelViewMatrix:GLKMatrix4Identity];    //3
     if (CGRectIntersectsRect(enemyRect, userNodeRect)) {              //4
       ballToDestroy = enemy;
       NSLog(@"El enemigo golpea al personaje");                       //5
     }
     else{
       if (enemy.position.y <0) {                                      //6
          ballToDestroy = enemy;
         NSLog(@"Retirando enemigo");
          break;
       }
    }
    [_enemies removeObject:ballToDestroy];                             //7
    [self.children removeObject:ballToDestroy];                        //8
}
```

Figura 5.12 Código del método de actualización dentro del objeto enemigo.

[††††††††††] *Este procedimiento sólo tiene validez mientras los objetos se encuentren posicionados sobre el mismo plano (en este caso, el x,y), en un caso más general, esta detección de colisiones no generará los resultados esperados.*

La anotación 1 y 6 muestra un código auxiliar utilizado para obtener la superficie que describe el modelo sobre el plano x,y para el objeto usuario y el objeto enemigo (respectivamente). La anotación 2 muestra la inspección a la colección de enemigos presentes en la escena (en este caso, un arreglo). La anotación 4 muestra la condicional que determina si las 2 superficies se han intersectado; en caso de ocurrir la intersección, se envía un mensaje al Log (5)y se retira el objeto del arreglo de enemigos (7), así como de la escena (8). La anotación 6 muestra el mecanismo por el cual los enemigos que han alcanzado el fondo de la pantalla, sin afectar al usuario, son eliminados de la escena.

La Figura 5.13 muestra el resultado de la evaluación del mecanismo de detección de colisiones. En la Figura 5.13a se muestra un enemigo que no choca con el usuario, mientras que la Figura 5.9b muestra el resultado de los dos objetos chocando.

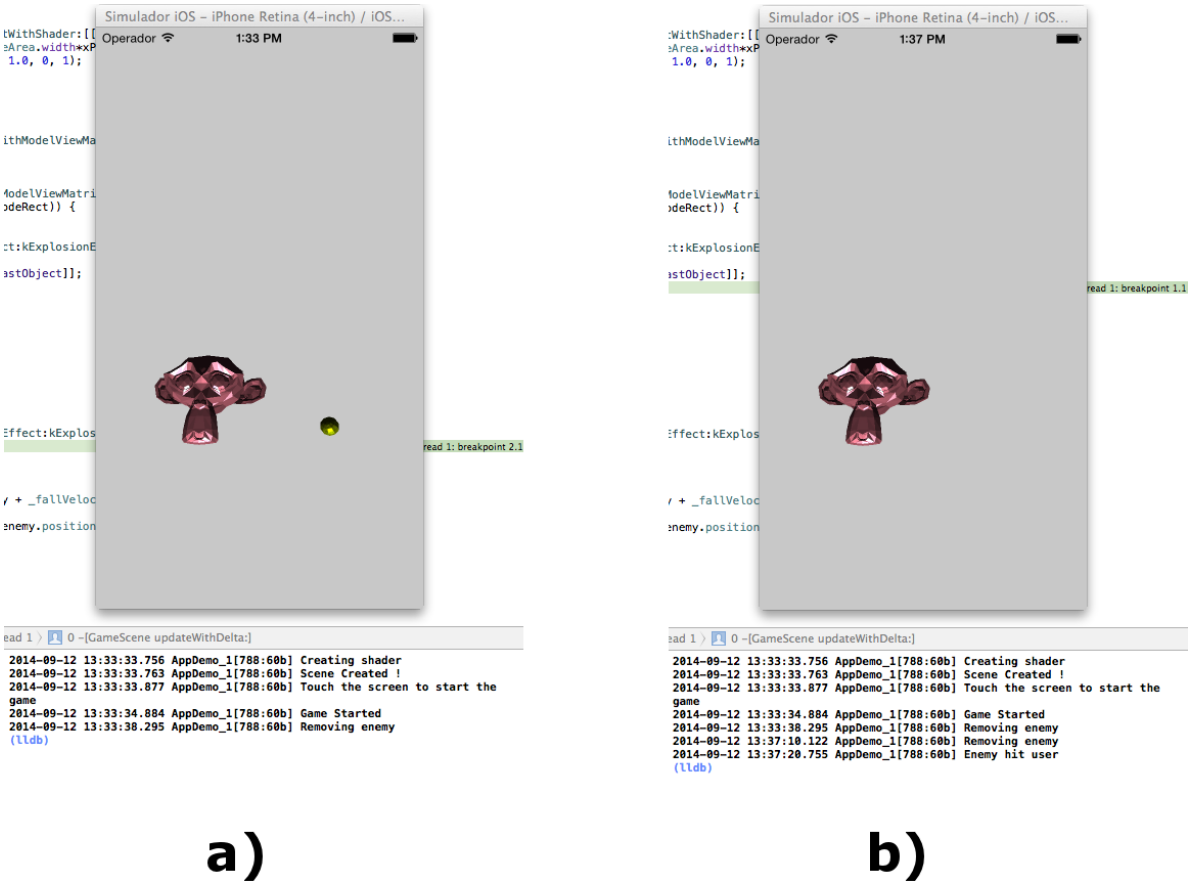


Figura 5.13 Resultado del mecanismo de detección de colisiones.

Hasta este momento solo se ha abordado la lógica de la aplicación, sin embargo es necesario establecer los mecanismos que permiten la interacción del usuario con la escena de juego.

5.1.3 Entrada de datos.

Definido el entorno de la aplicación, es necesario establecer la lógica de la aplicación referente al movimiento del usuario. Para cumplir este cometido, se harán uso de los mecanismos de entrada de datos presente en el dispositivo, lo cual involucra la detección de toques y el acelerómetro.

5.1.3.1 Detección de toques en pantalla.

Como se discutió en la implementación, la vista *OpenGL* es la encargada de recolectar la información de entrada de datos (toques en pantalla), para posteriormente enviar estos datos hacia el ***Gestor OpenGL***. El código de la Figura 5.14 muestra el proceso que describe la escena.

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    if (_gameStarted == NO) {
        _gameStarted = YES;
        NSLog(@"Juego iniciado");
    }
    else{
        return;
    }
}
```

Figura 5.14 Código de detección de toques en pantalla.

La detección utilizada en el código anterior, simplemente levanta una bandera para iniciar la escena de juego cuando toque la pantalla. El resultado se muestra en la Figura 5.15.

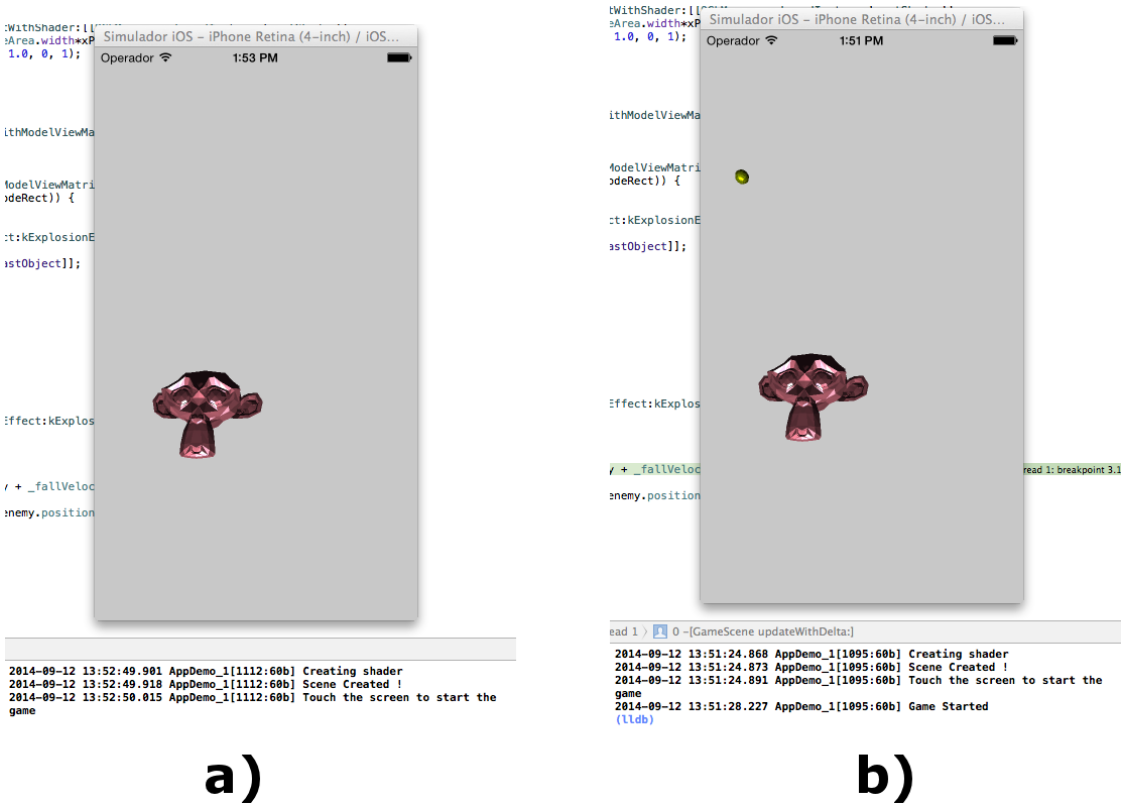


Figura 5.15 Resultado del mecanismo de detección de toques.

En la Figura 5.15a se observa el estado inicial de la aplicación, mientras que le Figura 5.11b muestra la inicialización del mecanismo que agrega enemigos a la escena, después que ha sido detectado el toque inicial.

5.1.3.2 Información proveniente del acelerómetro.
Para la recuperación de la información proveniente del acelerómetro, basta con implementar el método de la superclase escena, y almacenar la aceleración en una variable para su revisión y procesamiento posterior. El código de la Figura 5.16 demuestra este proceso.

```
- (void)getAcceleration:(CMAcceleration)acceleration{
    _accelerationInX = acceleration.x;
}
```

Figura 5.16 Código para de recolección de información de acelerómetro.

La Figura 5.17 muestra el efecto del movimiento del acelerómetro en el dispositivo.

En la Figura 5.17a observamos el personaje al inicio de la aplicación, la Figura 5.17b muestra el efecto del movimiento del acelerómetro inclinando sobre la izquierda del dispositivo. La Figura 5.17c muestra el efecto al girar el teléfono hacia el lado opuesto.

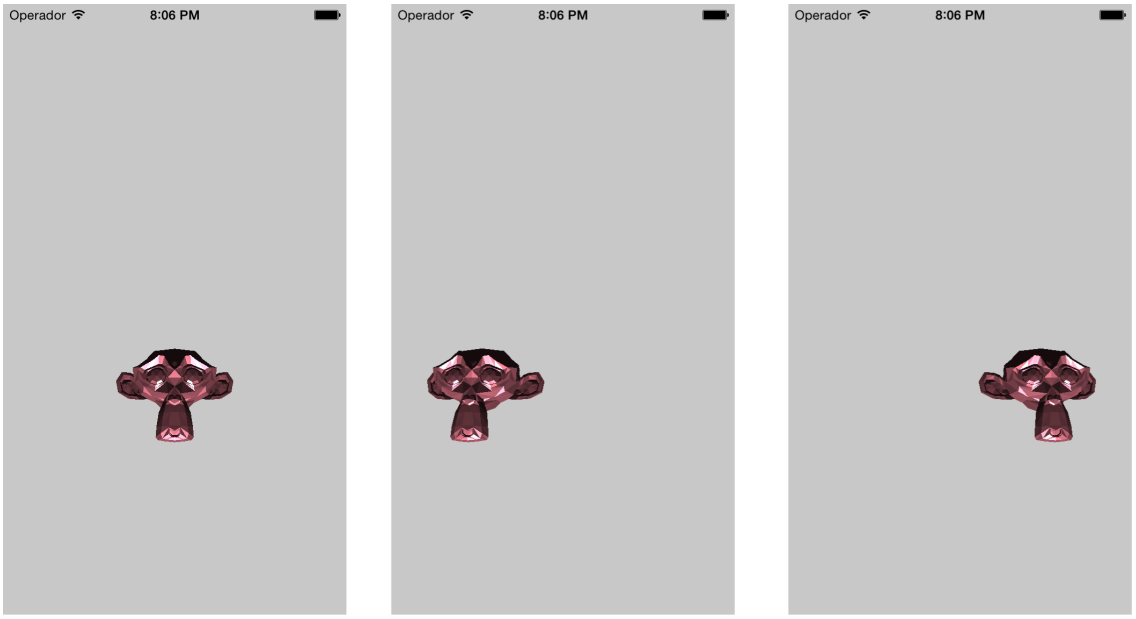


Figura 5.17 Movimiento del personaje por efecto del acelerómetro.

El mecanismo que envía los datos del acelerómetro hacia el bucle de renderizado se muestra en el código de la Figura 5.18

```
// Move the user node with the accelerometer
    float newX = _userNode.position.x + _accelerationInX;

    if (newX <0)            {      newX = 0;           }
    if (newX > _xMargin)    {      newX = _xMargin;    }

_userNode.position = GLKVector3Make(newX,
_userNode.position.y,_userNode.position.z);
```

Figura 5.18 Código de actualización de posición en base a información del acelerómetro.

5.1.4 Interacción con el sistema operativo.

Como una demostración de la integración de del motor de juegos hacia el sistema operativo, se propone la generación de un mecanismo de reproducción de sonidos que vincule la lógica del motor de juegos con las prestaciones de reproducción multimedia presentes en el sistema operativo iOS. El código de la Figura 5.91 muestra esta interacción.

```
- (AVAudioPlayer *)preloadSoundEffect:(NSString *)filename {
 NSURL *URL = [[NSBundle mainBundle] URLForResource:filename withExtension:nil];
 AVAudioPlayer *retval = [[AVAudioPlayer alloc] initWithContentsOfURL:URL
error:nil];
  [retval prepareToPlay];
  return retval;
}
...
```

Figura 5.19 Código para la integración de los recursos multimedia presentes en el dispositivo.

La integración con el motor de juegos se realiza en el detector de colisiones, donde reproduciremos un sonido cuando se detecte un impacto entre un enemigo y el personaje del usuario (Figura 5.20)

```
...
 [[OGLManager sharedInstance] playEffect:kExplosionEffect];
...
```

Figura 5.20 Código de la integración del reproductor de sonido con el detector de colisiones.

El juego finaliza cuando el usuario recibe 3 impactos de los enemigos, lo cual detiene el bucle de actualización para posteriormente terminar la escena (Ver Figura 5.21).

Figura 5.21 Finalización de la escena de juego.

5.2 Pruebas de desempeño.

Para el análisis de desempeño, se utiliza la herramienta de depuración y análisis de desempeño Instruments[‡‡‡‡‡‡‡‡‡‡‡].

5.2.1 Análisis de lógica *OpenGL*.

La primera prueba a la cual es sometida la aplicación generada por el motor, es el análisis del comportamiento de los comandos *OpenGL*. Se esta manera, podremos observar si existen fallos en la lógica del motor, que, en algún momento, afecten el funcionamiento del mismo (o que no se adecúen a las mejores prácticas de la programación).
Para ello, seleccionamos el análisis *OpenGL ES Analyzer*, el cual nos brindará la información necesaria. La Figura 5.22 muestra el resultado de la prueba.

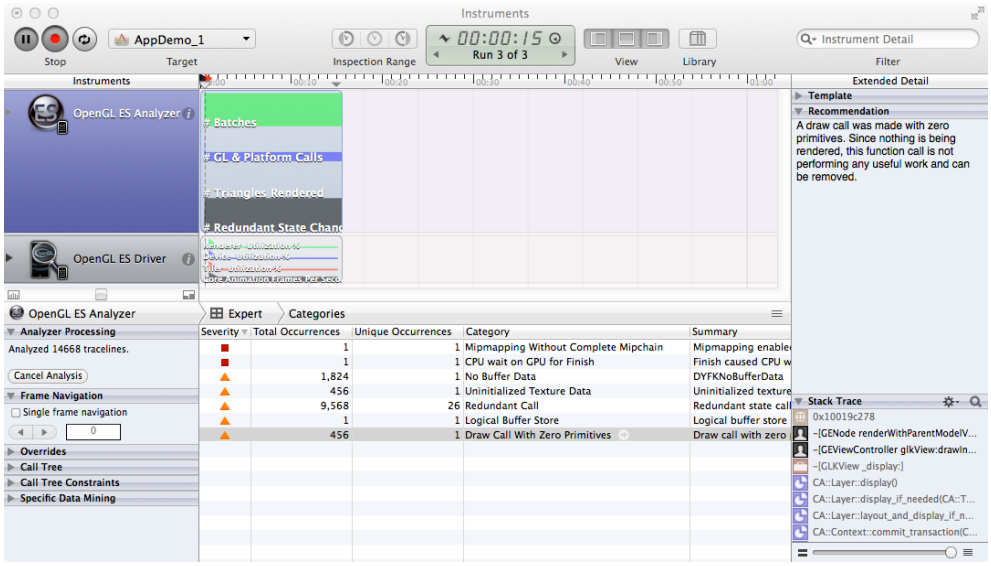


Figura 5.22 Resultado de la prueba *OpenGL ES Analyzer* (escena vacía).

Se observa que el software agrupa, en base a categorías, las diferentes incidencias ocurridas durante la ejecución de la aplicación. A continuación, se analizan las diferentes entradas arrojadas por el sistema:

MipMapping Without Complete Mipchain.
La descripción de esta incidencia, describe que se ha especificado, dentro del *shader*, un atributo de tipo textura el cual no es enviado nunca hacia el GPU. Con ello, encuentra un desbalanceo en las variables que se envían y las requeridas, lo cual podría resultar en un fallo. Este análisis es totalmente correcto, dado que la escena, al final es un objeto que hereda del objeto **`GENode`**, el cual presenta la textura como uno de sus atributos base, por lo cual, durante el *renderizado*, busca colectar y enviar esta información. Sin embargo, dada la naturaleza abstracta (en términos de existencia, no de su caracterización como clase), sabemos que no existe una textura que se agregue al objeto escena como tal.

[‡‡‡‡‡‡‡‡‡‡‡] *Ver [17]*

CPU wait on GPU to finish.
Esta entrada describe la implementación errónea de la destrucción del contexto *OpenGL* antes de la finalización de la aplicación. Al entrar en el estado *background*, la aplicación no libera los recursos de manera adecuada, de manera que el CPU debe esperar a que acaben las operaciones existentes en el GPU para poder finalizar o mandar e un estado de espera a la aplicación. Esto resulta en un retardo existente entre la entrada a *background*, y el 'pausado' de las operaciones *OpenGL*.
No Buffer Data, Uninitialized Texture data.
Al igual que la primera entrada, el analizador observa que las texturas son consideradas para ser enviadas hacia el *shader*, sin embargo, la información no existe.
Reduntant Call.
El analizador detecta el enlace y envío (de manera continua, por cada modelo tridimensionales) hacia el GPU como llamadas redundantes.
Aún cuando el *shader* envía la información para cada modelo en particular, el enlace la información hacia el GPU se realiza de manera recurrente durante cada ciclo de *renderizado*. Este código se podría agrupar en una entidad intermedia que permita la convergencia de estas instrucciones hacia un objeto particular que las despache.
Logical Buffer Store.
Aunada a la entrada anterior, la conmutación entre buffers de manera continua puede presentar inconsistencia si estas llamadas de enlace y envío de datos no terminan de manera correcta las operaciones. El analizador detecta que la conmutación entre los buffers se realiza sin retirar la información preexistente entre los mismos. Aún cuando el buffer toma la nueva información y trabaja con ella, sobrescribiendo el contenido anterior, es una buena práctica limpiar el buffer antes de enviar información.
Draw Call with Zero primitives.
La escena no presenta ningún vértice ni información útil para el *renderizado*. Aún así, el nodo base (GENode), envía mensajes para *renderizar* sobre la vértices de la estructura del modelo (información nula en el caso de al escena). Esta advertencia es detectada debido a que se envía *renderizar*, son información de vértices.

El siguiente análisis se realizará incluyendo el *renderizado* de un modelo tridimensional en la escena. El resultado se muestra en la figura 5.23. Como es de esperarse, las advertencias del escenario anterior, pero se presentan algunas referentes al modelo tridimensional.

MipMapping Usage.
Esta advertencia define que la textura utilizada (png), puede ser optimizada por medio de mipmapping, de manera que la carga de recursos sea de manera más rápida. Este proceso se puede realizar utilizando el procesador de gráficos, o por medio de un tratamiento en base a código.
Texture Formaty Compactness.
Al igual que el anterior, define que la textura utilizada puede ser optimizada. Esto debido a que la imagen utiliza 8-bits por canal, mientras que una imagen que utilice 16 bits por pixel, podría reducir significativamente el uso de recursos.
Recommended indexes Rendering.
La utilización de triángulos como figura base representa un ahorro de procesamiento y memoria. Sin embargo, los modelos que hemos cargado, no representan carga significativa al procesador, por lo que considera el uso de elementos en base a líneas, una mejora en definición. De igual manera, al considerarlos 'simples', reduciría el procesamiento de formas geométricas.

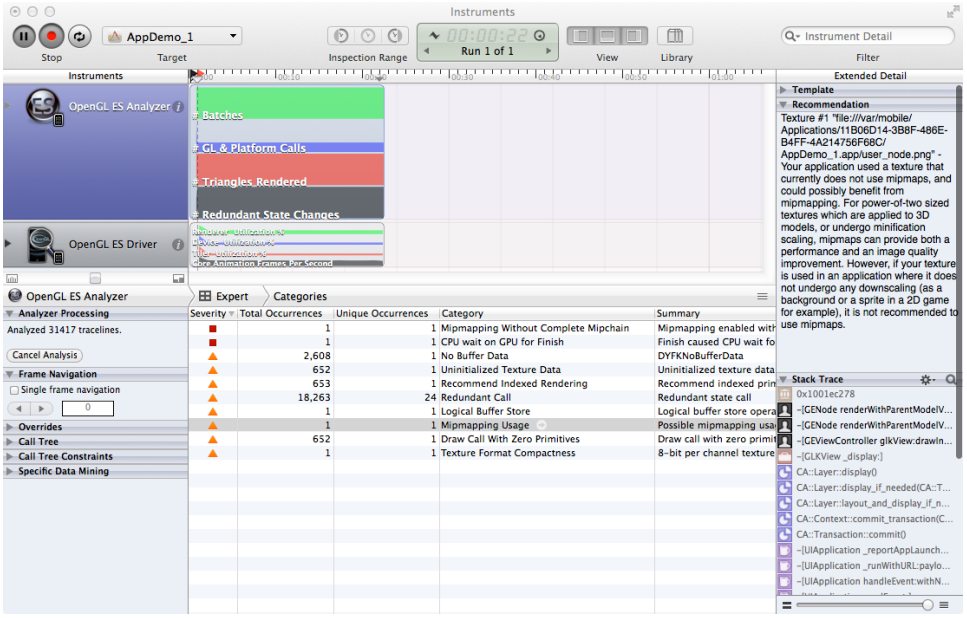


Figura 5.23 Resultado de la prueba *OpenGL ES Analyzer* (escena con un modelo tridimensional).

5.3 Comparativa con motores existentes.

En las secciones anteriores de este capítulo se describieron mecanismos que permiten validar la prueba de funcionalidad del motor de juego, sin embargo, es necesario realizar pruebas comparativas contra tecnologías existentes, que permitan posicionar el motor de juegos desarrollado dentro de los motores de juegos actuales.

En la secciones posteriores, se realizarán análisis sobre una aplicación común, esto es, una aplicación desarrollada bajo cada uno de los motores de juego que presente la misma lógica y elementos, de manera que la comparativa se realice en un ambiente estructurado. La aplicación que será utilizada para este experimento es la descrita en la sección 5.1
En el caso del motor de juegos contra el cual se realizará esta comparativa, se ha seleccionado Cocos3D, un motor ampliamente utilizado en el desarrollo de videojuegos sobre la plataforma iOS.

5.3.1 Análisis del procesamiento en el GPU.

El primer análisis al cual se someterán ambos motores es al análisis del funcionamiento del GPU durante la ejecución de una aplicación realizada sobre el motor. Este análisis sirve para establecer el procesamiento que se realiza sobre el procesador de gráficos (el encargado del renderizado de los modelos tridimensionales), y puede ser segmentado en los siguientes análisis:

1. Device Utilization. Unidad que describe el porcentaje de utilización del CPU por parte de la aplicación.
2. Renderer Utilization. Describe el porcentaje del renderizado que se encuentra trabajando en el pipeline de la máquina de estados. Esta unidad mide el procesamiento de información de geometría (los Vertex Shaders) del GPU

3. Tilder Utilization. Describe el porcentaje de memoria de texturas y composición de pixeles en la máquina de estados OpenGL. Esta evaluación se enfoca en analizar el comportamiento y desempeño de los shaders sobre pixeles (fragment shaders).
4. Core Animation Frames Per Second. Unidad encargada de mostrar la estabilidad del método de actualización de fotogramas en pantalla. Por definición el bucle corre a 60 fotogramas por segundo, sin embargo, el número de recursos en pantalla y la lógica de la aplicación puede afectar este comportamiento.

La Figura 5.24a muestra el resultado del test de desempeño OpenGL sobre la aplicación demo generada a partir del motor de juegos presentado en este trabajo, mientras que la figura 5.24b muestra el test evaluando la misma aplicación generada sobre Cocos3D.

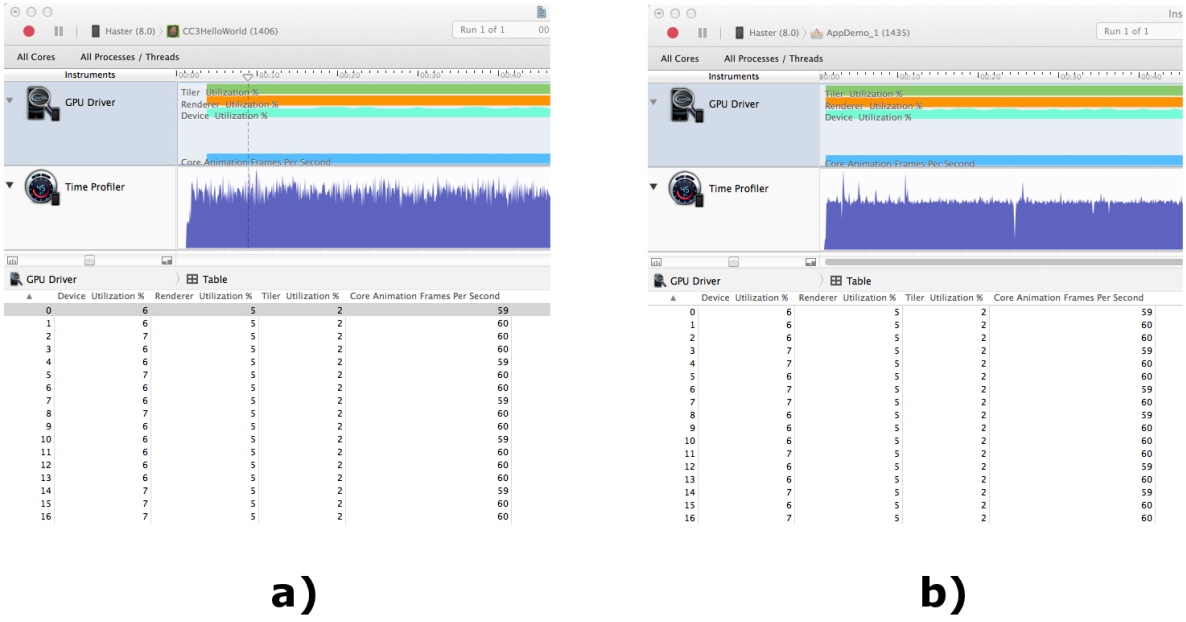


Figura 5.24 Análisis de OpenGLES sobre las 2 aplicaciones utilizando Instruments.

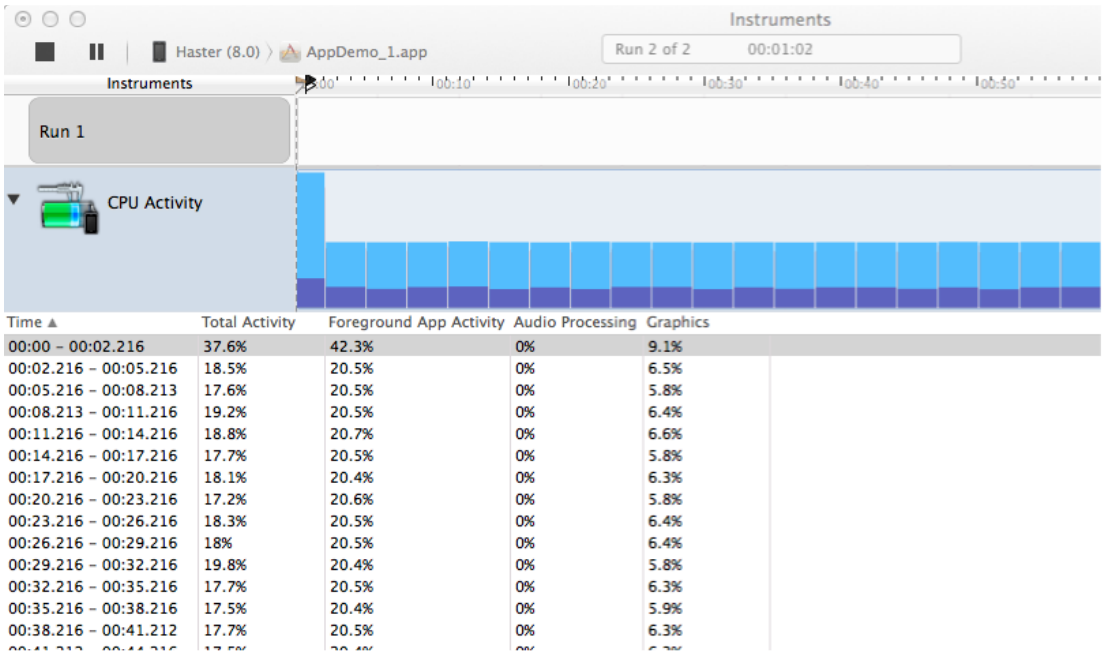
En la comparativa, podemos observar valores uniformes en ambos motores, lo que expresa que los mecanismos de renderizados utilizados en la aplicación desarrollada bajo el motor descrito en el presente documento, se comportan de manera correcta.

5.3.2 Análisis de desempeño en el CPU.

El análisis de energía y CPU es utilizado para evaluar, de manera más genérica, el desempeño de las aplicaciones que corren sobre iOS. Basado en la información proveniente del dispositivos en tiempo de ejecución, se obtienen los siguientes valores:

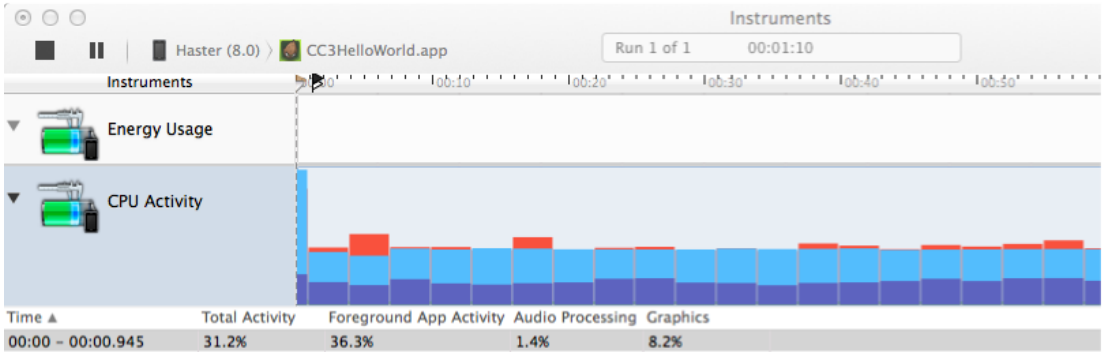
1. Total Activity. Expresa el porcentaje de procesamiento que el sistema operativo provee a la aplicación.
2. Foreground App Activity. Dentro del espacio de procesamiento que se provee a la aplicación, se mide el uso que la aplicación hace del mismo atendiento la lógica de su funcionamiento.
3. Audio Processing. Porcentaje de uso de las capacidades de audio del dispositivo.
4. Graphics. El procentaje del procesamiento de gráficos que se están utilizando.

La Figura 5.25 muestra el comparativo que se realiza entre la aplicación desarrollada usando el motor del presente trabajo (Figura 5.25a) y la aplicación desarrollada bajo el motor Cocos3D (Figura 5.25b)



Time ▲	Total Activity	Foreground App Activity	Audio Processing	Graphics
00:00 – 00:02.216	37.6%	42.3%	0%	9.1%
00:02.216 – 00:05.216	18.5%	20.5%	0%	6.5%
00:05.216 – 00:08.213	17.6%	20.5%	0%	5.8%
00:08.213 – 00:11.216	19.2%	20.5%	0%	6.4%
00:11.216 – 00:14.216	18.8%	20.7%	0%	6.6%
00:14.216 – 00:17.216	17.7%	20.5%	0%	5.8%
00:17.216 – 00:20.216	18.1%	20.4%	0%	6.3%
00:20.216 – 00:23.216	17.2%	20.6%	0%	5.8%
00:23.216 – 00:26.216	18.3%	20.5%	0%	6.4%
00:26.216 – 00:29.216	18%	20.5%	0%	6.4%
00:29.216 – 00:32.216	19.8%	20.4%	0%	5.8%
00:32.216 – 00:35.216	17.7%	20.5%	0%	6.3%
00:35.216 – 00:38.216	17.5%	20.4%	0%	5.9%
00:38.216 – 00:41.212	17.7%	20.5%	0%	6.3%

a)



Time ▲	Total Activity	Foreground App Activity	Audio Processing	Graphics
00:00 – 00:00.945	31.2%	36.3%	1.4%	8.2%
00:00.945 – 00:03.945	15.4%	14.2%	1.1%	6.1%
00:03.945 – 00:06.942	19%	13.2%	0.9%	5.3%
00:06.942 – 00:09.945	15.5%	15.1%	0.9%	6.9%
00:09.945 – 00:12.945	15.5%	14.9%	1%	5.7%
00:12.945 – 00:15.943	14.7%	15.2%	1%	5.4%
00:15.943 – 00:18.941	18.2%	15.1%	0.9%	5.8%
00:18.941 – 00:21.945	14.8%	14.9%	0.9%	6%
00:21.945 – 00:24.945	15.3%	14.9%	0.9%	7%
00:24.945 – 00:27.945	15.6%	14.9%	0.9%	7.1%
00:27.945 – 00:30.945	14.5%	14.9%	0.9%	6%
00:30.945 – 00:33.945	15.1%	15.1%	0.9%	5.8%
00:33.945 – 00:36.942	14.4%	15%	0.9%	5.2%

b)

Figura 5.25 Análisis de utilización de CPU sobre las 2 aplicaciones utilizando Instruments.

Se puede observar que el porcentaje de procesamiento que utilizan las 2 aplicaciones se mantiene a un nivel cercano (diferencia del 3-4 % menor en Cocos3D). Esto describe que el motor Cocos3D cuenta con un mayor grado de optimización en el uso del CPU.

El porcentaje de utilización de la aplicación durante su ejecución muestra que la aplicación desarrollada sobre Cocos3D requiere menos recursos para atender la lógica de la aplicación (una diferencia de 4-5 % en comparación a la aplicación desarrollada sobre el motor del presente trabajo). En el caso del procesamiento de gráficos, observamos que los mecanismos de renderizado utilizados en el motor son comparables en cuanto a procesamiento gráfico se requiere, mostrando una diferencia menor que los puntos anteriores (1.5 – 2.0 % a favor de Cocos3D).

El análisis por Instruments muestra que el desempeño de la aplicación desarrollador en el motor desarrollado es menor al de la aplicación realizada en Cocos3D, sin embargo, las métricas muestran que las diferencias son mínimas, lo que concluye que el desempeño puede considerarse como óptimo.

5.4 Análisis de resultados.

La prueba de concepto y las pruebas de rendimiento, permiten obtener conclusiones acerca del funcionamiento del motor. Dentro de éstas conclusiones, podemos definir 2 vertientes, las cuales describen las bondades del motor y las debilidades del mismo.

5.4.1 Características, funcionalidades y bondades del motor de juegos.

A continuación presentamos las funcionalidades desarrolladas en el motor de juegos:

1. Desarrollo de la arquitectura del motor. En base a la aplicación Demo, se puede observar que el motor contiene los módulos necesarios (en relación a la arquitectura propuesta), para funcionar como un motor de juegos. Con esto concluimos, que el desarrollo de un motor de juegos, per se, se consiguió.
2. Diseñar e implementar un mecanismo de renderizado de modelos tridimensionales para el archivo tipo obj. El desarrollo del *shader* que pueda renderizar un modelo obj permite la integración de estos modelos al motor de juegos, característica no existente en los motores libres actuales.
3. Documentación y definición de clases que componen el motor de juegos. En el Anexo 2, se presenta la documentación necesaria para entender e implementar las diferentes clases y funcionalidades del motor de juegos. Esto, como una referencia al desarrollador, de manera que exista un documento que permita el estudio de los métodos y propiedades del motor.
4. Establecimiento de los mecanismos de *renderizado* para los modelos tridimensionales. Para el funcionamiento del motor, se definieron mecanismos que permitan la integración de los modelos tridimensionales, de manera que puedan ser desplegados en pantalla. Con esto, se cumple la funcionalidad del motor de contar con características tridimensionales.
5. Se logra interacción con el sistema operativo para la entrada de datos. Se logra la creación de mecanismos que permitan el manejo de la información proveniente de los dispositivos de entrada de datos, de manera que la interacción con el usuario está garantizada (en base al acelerómetro y la pantalla táctil).
6. El análisis de desempeño no muestra fallas de lógica graves, lo que garantiza la estabilidad del motor de juegos, como base de una aplicación en *iOS*.

5.4.2 Fallas, limitaciones y debilidades del motor de juegos.

1. Falta de integración con las interfaces de usuario de *iOS*. El motor no presenta integración con las interfaces de usuario definidas por *Cocoa Touch* (vistas, botones, sliders, etc). Esto permitiría el despliegue y manejo de información de manera más eficiente y amigable al usuario.
2. Limitación a modelos desarrollador bajo tridimensionales bajo características estrictas. El uso de modelos se reduce a aquellos que sean susceptibles a la transformación de la información de vértices al formato necesario por el motor. El uso eficiente de otro tipo de modelos no queda garantizado.
3. Falta de definición de una pila de escenas en el gestor. Para mejorar la eficiencia y rapidez en la carga de escenas, se podrían implementar mecanismos de caché para las escenas. Esta integración permitiría la conmutación de escenas o la existencia de escenas a manera de pop over en la aplicación.
4. Falta de un mecanismo de reproducción de sonidos más eficiente. El usar el framework de reproducción de sonidos facilita los mecanismos de reproducción de multimedia. Sin embargo, no permite una integración transparente como parte del motor
5. Mejora y optimización del código *OpenGL*. El análisis de desempeño al cual se sometió la aplicación, revela que existen funciones y procedimientos que pueden optimizarse, mejorando el performance. Esta refinación de código permitiría establecer un código con mayor funcionalidad y de mayor calidad.
6. Establecimiento de una unidad de medida en la superficie de juego. Al desarrollar aplicaciones sobre el motor, se observa que no existe una estandarización de las unidades que se utilizarán durante el juego (todas las medidas están en función de las especificaciones del usuario, y deben ser calculadas en el momento de iniciar el desarrollo). El establecer un mecanismo que realice estas conversiones, permitiría estructurar el código y hacerlo mas uniforme y entendible.

Capítulo 6

CONCLUSIONES

El trabajo presenta como objetivo el motor de juegos, y, en base a los resultados obtenidos y la implementación descrita, se concluye que el desarrollo del mismo se logró de manera adecuada. En las secciones anteriores se han descrito las debilidades y fortalezas del trabajo desarrollado, sin embargo, no presentan fallas estructurales fuertes, que comprometan el correcto funcionamiento del motor como tal. Dentro del proceso de desarrollo se pueden presentar las siguientes conclusiones:

1. Los mecanismos de renderizado de un motor de juegos deben contar con un alto nivel de interacción con las capas superiores. Al analizar los elementos presentes en OpenGL, observamos que se provee al desarrollador con un número elevado de directivas y comandos para renderizar modelos tridimensionales, todo ello en base a una máquina de estados (contenida en el bucle de actualización-renderizado). En un motor de juegos, observamos la necesidad de crear una arquitectura a base de objetos, que permita traducir estas directivas a mecanismos más simples y de más fácil acceso a los desarrolladores de videojuegos.
2. Los motores de juegos son orientados a modelos tridimensionales específicos. El presente trabajo demostró su validez haciendo uso de un formato de objeto tridimensional no utilizado en motores de juegos actuales (el objeto obj). Con esto, se demuestra que los mecanismos de renderizado son específicos a cada formato de archivo en el cual se genera el modelo tridimensional.
3. El desarrollo sobre la tecnología *OpenGLES* se realiza de manera transparente en el sistema operativo *iOS*. Al iniciar este desarrollo, se plantea como sistema operativo objetivo el sistema *iOS 7.x*. Esta elección se basa en la posibilidad de compilar código c++ (sobre lo cual está basado *OpenGL*) utilizando el mismo compilador utilizado por el IDE de desarrollo *XCode* (gcc). El apéndice 1 muestra la simplicidad con que se logra inclusión de los archivos de configuración necesarios para desarrollar un juego utilizando el motor, así como la ejecución de un proyecto prueba. Como contraparte, desarrollar el presente trabajo en el sistema operativo Android, exige el conocimiento de las herramientas de compilación necesariar para generar los archivos binarios del motor, y la ejecución de los mismos en base al *NDK* (Native Development Kit) presente en la plataforma Android.
4. Las superficies *OpenGL* manejan un sistema de referencia diferente a las vistas presentes en *Cocoa Touch*. En el desarrollo del trabajo, se encontraron inconsistencias referentes a los sistemas coordenados presentes en las superficies *OpenGL*, en relación a los señalados en el framework *Cocoa Touch*.
   Estas consideraciones deben ser tomadas en cuenta para interactuar con mecanismos de nivel sistema operativo (por ejemplo, la salida del acelerómetro debe ser convertida a un sistema coordenado inverso en los 3 ejes coordenados, de manera que los incrementos presentes en CocoaTouch, son decrementos en la superficie *OpenGL*).

En las secciones posteriores, se describirán los logros alcanzados, las contribuciones del presente trabajo, así como los trabajos a futuro y mejoras al sistema.

6.1 Logros alcanzados.

El presente trabajo describe el proceso de desarrollo de un motor de juegos sobre un dispositivo móvil; en esta sección se enumeran los logros alcanzados en cada etapa del proceso.

1. Implementación de una arquitectura de motor de juegos. Al definir la arquitectura de un motor de juegos, de manera teórica se presentan los módulos a realizar. En este trabajo, se logran implementar los módulos necesarios para obtener un motor de juegos reducido[§§§§§§§§§§].
2. Implementación y diseño del mecanismos de renderizado. Haciendo uso del estándar *obj*, se definieron mecanismos para poder realizar el procesamiento del archivo, de manera que el mecanismo de renderizado presente en el motor, sea capaz de interpretar la información de vértices y formas para generar el modelo tridimensional.
3. Se definieron, crearon y documentaron[**********] las funciones necesarias para el manejo de la capa de estados OpenGL. Con esto, el motor presente ser amigable al usuario desarrollador y altamente modificable.
4. Se desarrolló la capa Middleware necesaria para la generación de los modelos tridimensionales, así como las funciones necesarias para modificar las propiedades de los modelos en tiempo de ejecución, módulo de interactividad altamente necesario para el correcto funcionamiento de un motor de juegos.

Como un logro global, el motor de juegos resulta ser altamente funcional para el desarrollo de aplicaciones, por lo que el objetivo general, de la construcción de un motor de juegos tridimensional para una plataforma móvil, se ha alcanzado completamente.

6.2 Contribuciones del presente trabajo.

Los procedimientos necesarios para el funcionamiento del motor, así como los módulos necesarios para la integración de las diferentes tecnologías se describieron a lo largo del este documento, de manera que se tiene una base firme sobre la cual analizar el desarrollo de mejorar y extensiones del mismo.
Con ello, se pretende tomar como punto de partida el motor de juegos realizado para desarrollos más complejos y que, en base a estas modificaciones, puedan resultar en un motor de juegos completo.

La principal contribución presentada en este trabajo, es el desarrollo del módulo de *renderizado* independiente del motor, lo que garantiza un desarrollo escalable y susceptible a mejoras y optimizaciones. La posibilidad de modificar el *shader*, presenta posibilidades de desarrollo para optimizar el funcionamiento del mecanismo de despliegue de modelos por parte de terceros, lo cual no es posible en los motores de juegos actuales.

Por último, podemos presentar como una contribución, el uso del motor como el inicio en el desarrollo de videojuegos, que permita al lector adentrarse en estas tecnologías, como un punto de entrada a un campo de desarrollo inmensamente grande. Esto permitiría una introducción simplificada al desarrollo de videojuegos en tres dimensiones.

---

§§§§§§§§§§ *El término reducido, se define en el capítulo 1.*
********** *Ver Anexo 'Documentación' en la distribución digital del presente trabajo*

6.3 Trabajos a futuro y mejoras al sistema.

En secciones anteriores, hemos descrito las funcionalidades del motor. Si embargo, el análisis de las fallas y limitantes del mismo presenta puertas abiertas a trabajos futuros sobre la misma dirección: la optimización y mejora en términos de rendimiento. Tomando el motor de juegos presentado, se pueden realizar mejoras de integración con el sistema operativo que permitan una mayor interacción con el mismo. Definir mejoras de optimización para el procesamiento y generación de gráficos tridimensionales.
De igual manera, el agregar mayores elementos al motor (botones, sliders, menues, etc), permitiría integrar mayores funcionalidades y brinda l desarrollador de elementos que permitan la generación de escenas más complejas y con mayores elementos.
De igual manera, existe la posibilidad de, en base al presente trabajo, desarrollar un motor de juegos sobre la nueva tecnología de aceleración de gráficos presentana por la compañía Apple: *Metal*. Tomando como base el presente motor, se puede realizar una implementación que permita una mejora en desempeño sobre el sistema operativo iOS.

Referencias

[1] ID Software. “The Doom Project”. [Fecha de Consulta: 10 Abril de 2013]. http://www.idsoftware.com/games/doom

[2] Game Career Guide. “What is a Game Engine?” [Fecha de Consulta 25 Septiembre 2013]. http://www.gamecareerguide.com/features/529/what_is_a_game_.php

[3] Jason Gregory. “Game Engine Architecture”. A.K Peters, 2009.

[4]MICROSOFT. “Instalador Web de tiempos de ejecución de usuario final de DirectX”. [Fecha de Consulta: 20 Abril de 2013]. http://www.microsoft.com/es-mx/download/details.aspx?id=35

[5] The Khronos OpenGL ARB Working Group. “OpenGL Programming Guide”. Addison-Wesley Professional, 2009.

[6] Box2D. “A 2D Physics Engine for Games”. [Fecha de Consulta: 15 Marzo de 2013]. http://box2d.org/

[7] Mark Segal, Kurt Akeley. “OpenGL Graphics System: A Specification”. Silicon Graphics Inc. 2004.

[8] Dave Astle, Kevin Hawkins. “Beginning OpenGL Game Programming”, Thomson, 2004.

[9] Khronos Group. “The Standard for Embedded Accelerated 3D Graphics”. [Fecha de Consulta: 28 Septiembre de 2013]. http://www.khronos.org/opengles/

[10] Wikipedia. "iOS (Operative System)". [Fecha de Consulta: 22 Mayo de 2014]. http://en.wikipedia.org/wiki/IOS

[11] Kim W. Tracy, IEEE. “Mobile application development experiences on Apple’s iOS and Android OS”. [Fecha de Consulta: 27 Septiembre de 2013]. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6248786

[12] iOS Developer Library. “OpenGL ES Programming Guide for iOS”. [Fecha de Consulta: 20 Septiembre de 2013]. https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/Introduction/Introduction.html

[13] Unity. “Unity 3D for iOS”. [Fecha de Consulta: 12 Mayo de 2013]. http://spanish.unity3d.com/

[14] Cocos3D. “The Brenwill Workshop”.[Fecha de Consulta: 1 Abril de 2013]. http://brenwill.com/cocos3d/

[15] Universidad de Denver. “Programa de Desarrollo de VideoJuegos”. [Fecha de Consulta: 17 Abril de 2013]. http://www.gamedev.cs.du.edu/

[16] WIRED. E-Magazine. “Why Game Creators Prefer iPhone to Android”. [Fecha de Consulta: 25 Abril de 2013] http://www.wired.com/gamelife/2013/02/android-vs-iphone-games/

[17] Apple Inc. “About Instruments”. http://developer.apple.com/library/mac/#documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/Introduction/Introduction.html

[18] F. Ted Tschang, School of Business, Singapore Management University. “Videogames As Interactive Experiential Products And Their Manner Of Development”. [Fecha de Consulta: 3 Abril de 2013]

[19] Shiva3D. “What is Shiva?” [Fecha de Consulta: 10 Septiembre de 2013] http://www.stonetrip.com/what-is-shiva-3d.html

[20] Apple Inc. “App Store Review Guidelines”. [Fecha de Consulta: 18 Septiembre de 2013]. https://developer.apple.com/appstore/resources/approval/guidelines.html

[21] msdn. “El Patrón Singleton”. [Fecha de Consulta: 27 Septiembre de 2013] http://msdn.microsoft.com/es-es/library/bb972272.aspx#EDAA

[22] Ole Begemann. “The App Launch Sequence on iOS”. [Fecha de Consulta: 30 Septiembre de 2013] http://oleb.net/blog/2011/06/app-launch-sequence-ios/

[23] iOS Developer Library. “App States and Multitasking”. [Fecha de Consulta: 6 Noviembre 2013] https://developer.apple.com/library/ios/documentation/iphone/conceptual/iphoneosprogrammingguide/ManagingYourApplicationsFlow/ManagingYourApplicationsFlow.html

[24] iOS Developer Library. “OpenGL ES Programming Guide for iOS”. [Fecha de Consulta: 17 Octubre 2013]https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/OpenGLESontheiPhone/OpenGLESontheiPhone.html#//apple_ref/doc/uid/TP40008793-CH101-SW1

[25] Erik Olsson, Royal Institute of Technology, School of Computer Science and Communication. "Mobile Phone 3D Games with OpenGL ES 2.0". Stockholm, Sweden, 2008.

[26] P. Strauss and R. Carey, ACM. “An object-oriented 3D Graphics Toolkit”, Silicon Computer Graphics Systems, 1992.

[27] Staffan Bjork, Jussi Holopainen. “Patterns in Game Design”. Cengage Learning, 2005.

[28] Jamilah Din, Sufian Idris. International Journal of Computer Science and Network Security. “Object-Oriented Design Process Model”, Universiti Kebangsaan Malaysia, 2009.

[29] Thomas C. S. Cheah, Kok-Why Ng, IEEE. "A Practical Implementation of a 3-D Game Engine", International Conference on Computer Graphics, Imaging and Visualization, 2005

[30] J. Döllner and K. Hinrichs. “A Generalized Scene Graph”, Vision, Modeling, Visualization 2000 (VMV 2000), Premier Press, 2000.

[31] wiseGeek. “What is 3D Modeling”. [Fecha de Consulta: 17 mayo de 2014]. http://www.wisegeek.com/what-is-3d-modeling.htm

[32] Code Reddy Inc. “The OBJC Specification”. [Fecha de Consulta: 2 de junio 2014]. http://www.martinreddy.net/gfx/3d/OBJ.spec

[33] Alias|Wavefront, Inc. “MTL material format (Lightwave, OBJ)”. [Fecha de Consulta:20 mayo 2014]. http://paulbourke.net/dataformats/mtl/

[34] Apple. “iOS Developer Library (GLKMatrix4 Reference)”.[Fecha de Consulta: 23 mayo 2014]. https://developer.apple.com/library/ios/documentation/GLkit/Reference/GLKMatrix4/Reference/reference.html

[35] Scirra. “Pseudo 3D Games”. [Fecha de Consulta: 2 de junio 2014]. https://www.scirra.com/tutorials/534/pseudo-3d-games

[36] Luca Chittaro, IEEE. “Visualizing Information on Mobile Devices”, Human-Computer Interaction Lab (HCI Lab) Dept. of Math and Computer Science, University of Udine, Italy, 2006.

[37] Apple. “iOS Developer Library (UITouch Class Reference)”.[Fecha de Consulta: 2 de junio 2014]. https://developer.apple.com/library/ios/documentation/uikit/reference/UITouch_Class/Reference/Reference.html

[38] Apple. “iOS Developer Library (CMAccelerometerData Class Reference)”.[Fecha de Consulta: 2 junio 2014].https://developer.apple.com/library/ios/documentation/coremotion/reference/CMAccelerometerData_Class/Reference/Reference.html

[38] Apple. “iOS Developer Library (GLKViewDelegate Protocol Reference)”.[Fecha de Consulta: 2 junio 2014].https://developer.apple.com/library/ios/documentation/GLkit/Reference/GLKViewDelegate_ProtocolRef/Reference/Reference.html

[40]Apple. “iOS Simulator Reference”. [Fecha de Consulta: 2 junio 2014].https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/iOS_Simulator_Guide/Introduction/Introduction.html

Anexo A.
Manual de Operación

CONTENIDO

1. Configuración de la aplicación base.

Para la utilización del motor de juegos, es necesario generar un entorno en el cual podemos hacer uso de las clases del mismo. Para ello, es necesario lanzar el IDE XCode (versión 5.0 o posterior). Se selecciona la opción de crear un nuevo proyecto en el menú inicial. En el menú de proyecto nuevo, se selecciona la opción **Single View Application**. (Ver figura 1.1)

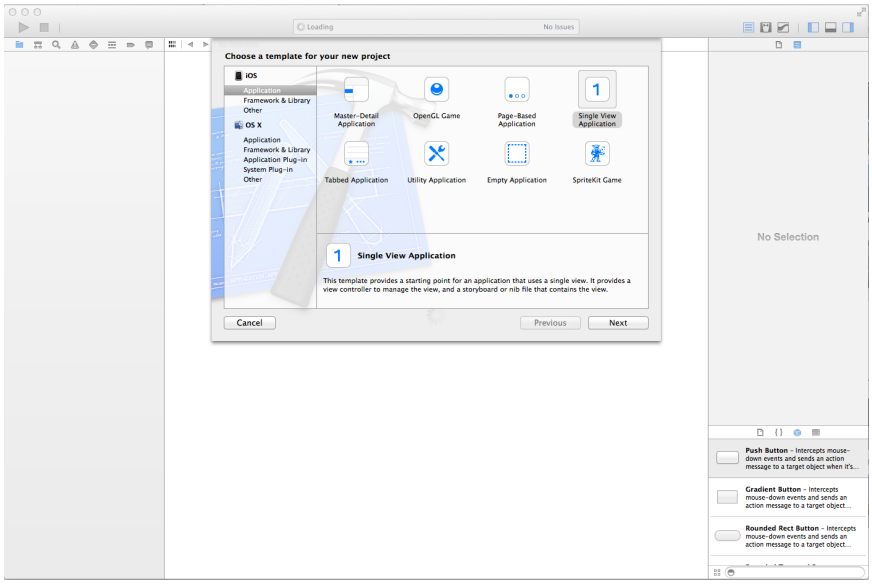


Figura 1.1 Creación de un proyecto nuevo para la utilización del motor de juegos.

En el menú siguiente, ingresamos los valores que correspondientes al nombre de la aplicación y el prefijo del entorno.

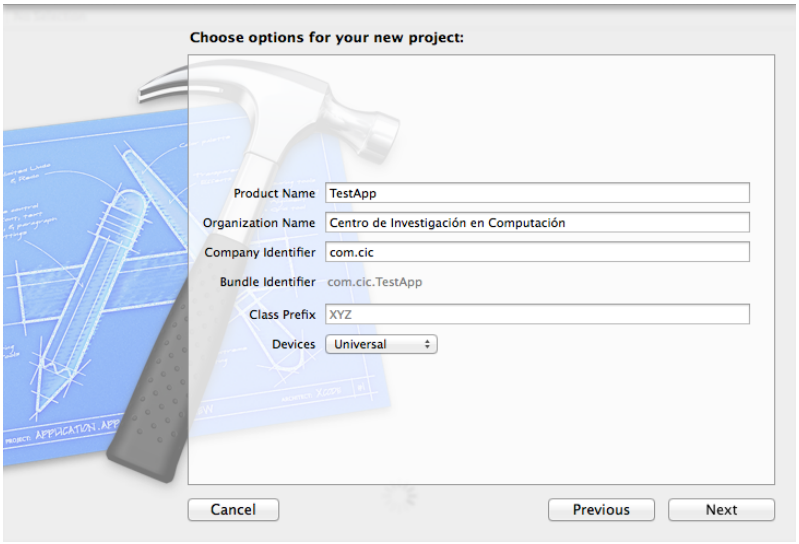


Figura 1.2 Datos necesarios para la creación del proyecto.

Por último, basta con especificar la ubicación donde guardaremos el proyecto.

2. Archivos del configuración del motor de juegos.

El motor de juegos se compone de una serie de clases, las cuales son necesarias para el funcionamiento del mismo. La figura 2.1 muestra el contenido del paquete que integra el motor de juegos.

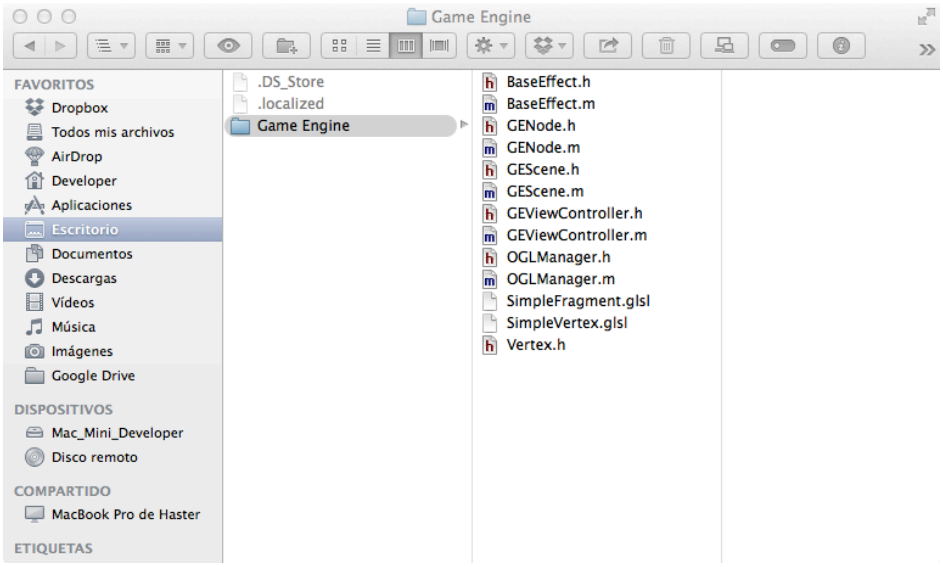


Figura 2.1 Archivos requeridos para el funcionamiento del motor de juegos.

El siguiente paso es agregar estos archivos en el proyecto. Esto se logra arrastrando los archivos, desde la ventana de Finder (el explorador de archivos), hacia el **Project Navigator.** Es imperativo el seleccionar la opción de "Copiar los archivos en caso de ser necesario" que aparece en la confirmación de copiado. (Ver Figura 2.2)

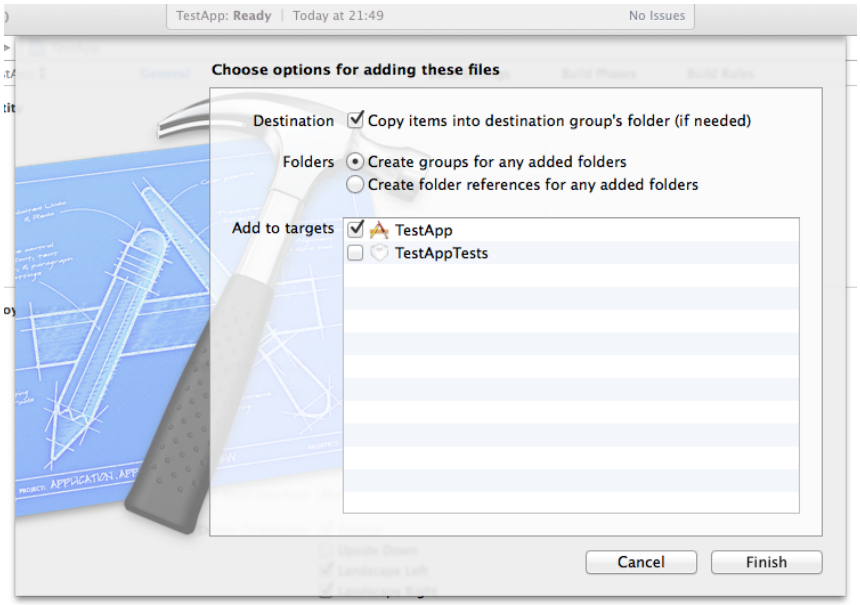


Figura 2.2 Copiado de archivos desde el Finder hacia el proyecto en XCode.

3. Configuración del entorno.

La figura 3.1 muestra los archivos que deben se incluidos dentro del proyecto para el funcionamiento del motor. Ahora es necesario realizar las adecuaciones necesarias para el correcto funcionamiento.
En el **Project Navigator**, seleccionamos el **Storyboard**[††††††††††††]. (lo encontraremos con el nombre **Main_iPhone.storyboard**) . Al seleccionarlo, se desplegará en pantalla el contenido del mismo. (Ver Figura 3.1).

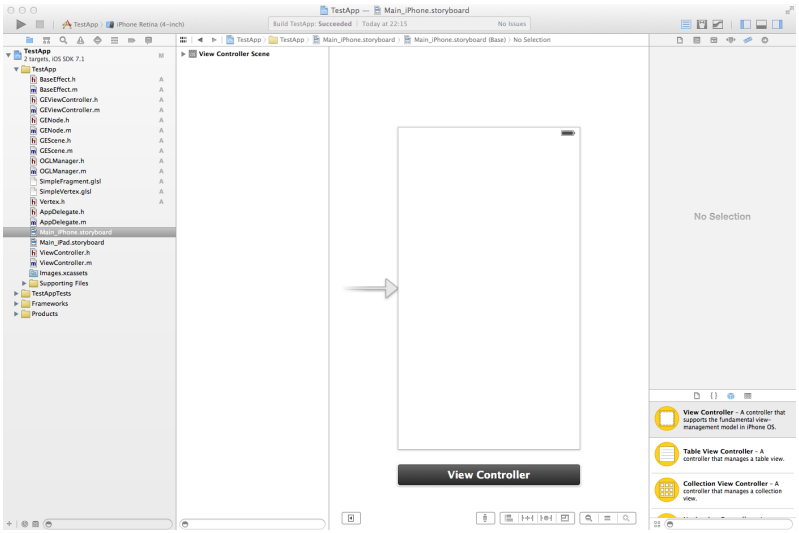


Figura 3.1 Vista del storyboard.

El siguiente paso es seleccionar la vista. Posterior a ello, se selecciona el **Identity Inspector** en el menú lateral derecho. El campo **Class** se modifica por **GLKView**.

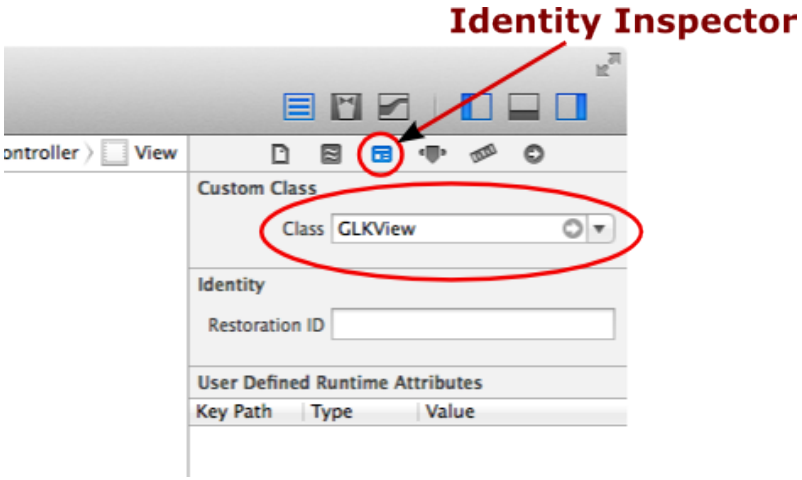


Figura 3.2 Modificación de la clase de la vista.

[††††††††††††] *En el presente documento, se realizará la configuración del entorno para las pruebas sobre el*
dispositivo iPhone. El procedimiento para in dispositivo iPad es análogo a lo aquí descrito.

A continuación, seleccionamos, en el **Storyboard**, el elemento denominado **View Controller** dentro del **Documento Outline.** Análogo al procedimiento anterior, cambiamos la clase que tiene asociado. En este caso, ingresamos el valor **GEViewController**.

Figura 3.2 Modificación de la clase del controlador.

Con esto se termina la configuración de las clases. Sin embargo, es necesario enlazar un **framework** para permitir la entrada de datos en base al acelerómetro. En el **Project Navigator**, seleccionamos la raíz del proyecto. Eso abrirá, en el editor, el menú correspondiente a la información del proyecto. Seleccionamos **Build Phases** en la barra superior, y la opción **Link Binary With Libraries**. Abrimos el abanico de esta opción, seleccionando el botón + para abrir el menú de **frameworks**. Buscamos el **framework CoreMotion**, y seleccionamos **Add**.

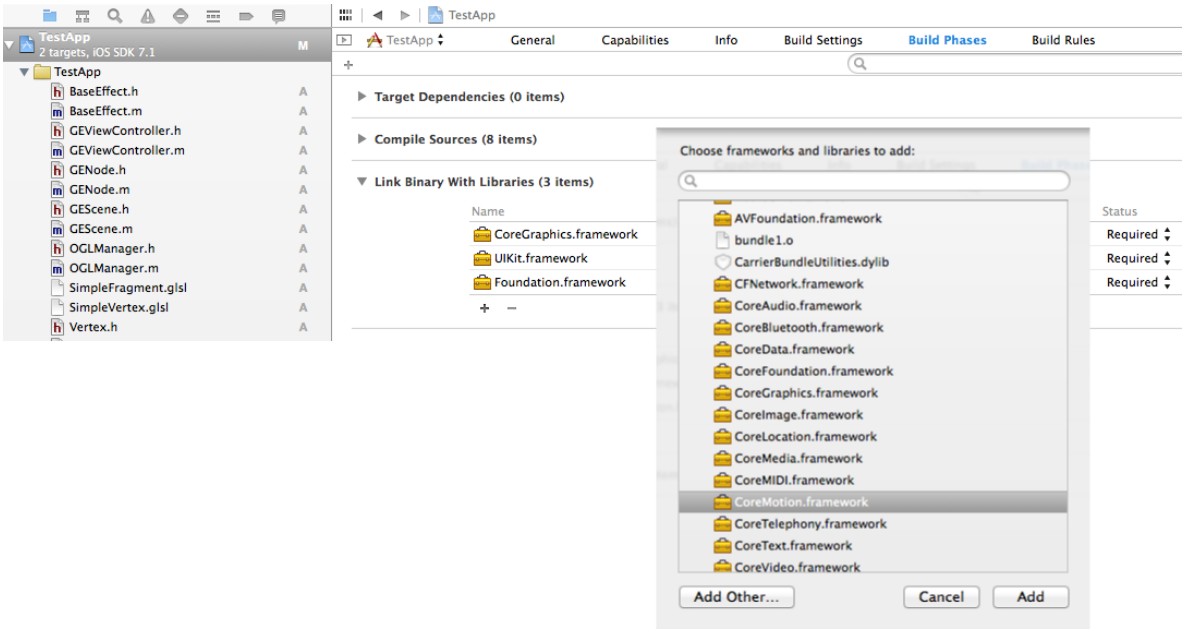


Figura 3.3 Inclusión del framework necesario para recibir información del acelerómetro.

Presionamos la combinación de teclas **cmd+B** para compilar el proyecto. Esto nos indicará que todo se encuentra configurado correctamente.

4. Creación de la primera escena.

Configurado el entorno, es necesario crear una escena, que nos permita iniciar nuestra aplicación haciendo uso del motor. Para ello, seleccionamos el menú **File > New** en la barra de herramientas superior. Seleccionamos la opción **File.** En el siguiente menú, bajo la categoría iOS, seleccionamos *Objective-C class*.

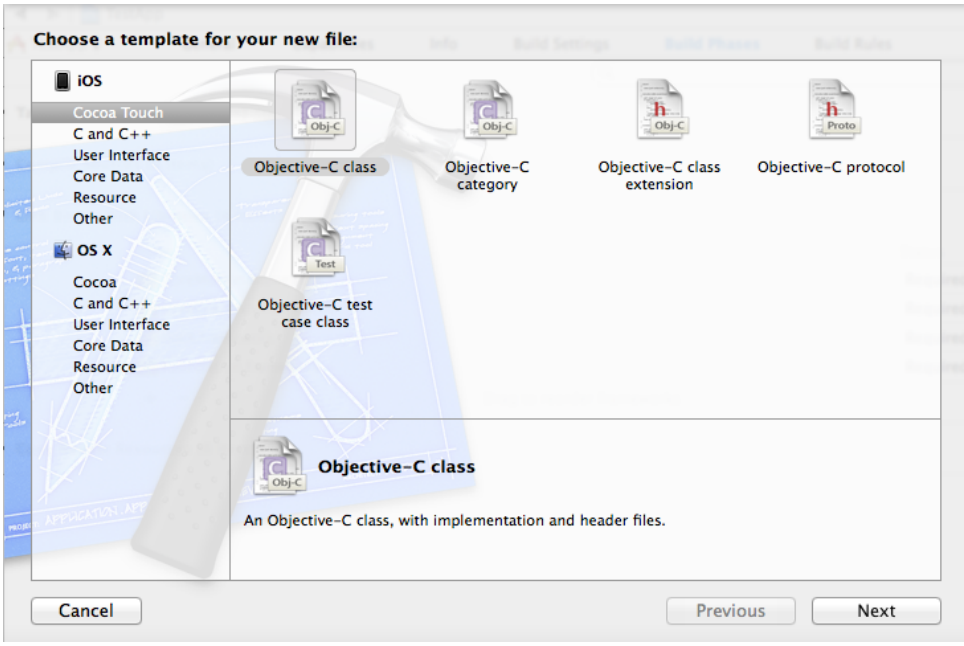


Figura 4.1 Inclusión de una nueva clase para una escena.

En el siguiente menú, asignamos un nombre a la escena, y la definimos como subclase de la clase **GEScene**.

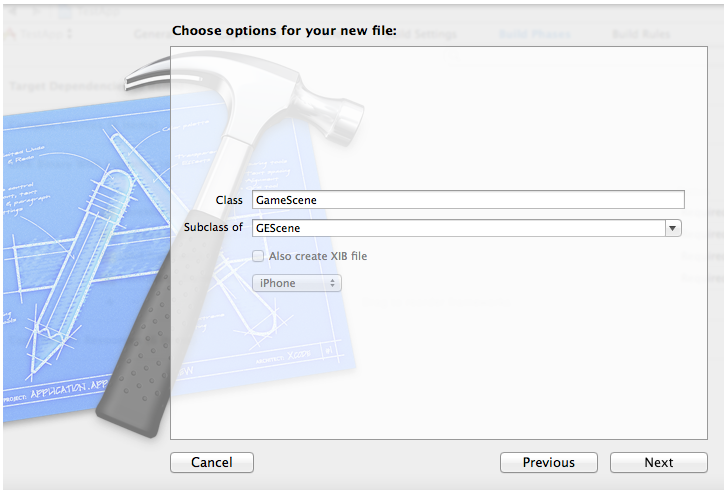


Figura 4.2 Definición de la nueva escena.

En el **Project Navigator**, seleccionamos el archivo **GEViewController.m** Se busca el método **`setupScene`**, justo debajo de la línea

```
[OGLManager sharedInstance].view = self.view;
```

Se ingresa el siguiente código:

```
Color col = {1.0, 0.0/255.0, 0/255.0};
GameScene  *scene = [[GameScene alloc] initSceneWithColor:col];

[OGLManager sharedInstance].scene = scene;
```

Con ello, se inicia una escena, con un color de fondo rojo. Con esto, el **IDE** arrojará errores acerca del objeto GameScene. Basta con importar la cabecera **GameScene.h** en el archivo **GEViewController.m**.

Si corremos la aplicación en el simulador (o en un dispositivo), la pantalla debe mostrar un color rojo sólido, y el log debe arrojar la leyenda **`Scene Started!`**. (Ver Figura 4.3)

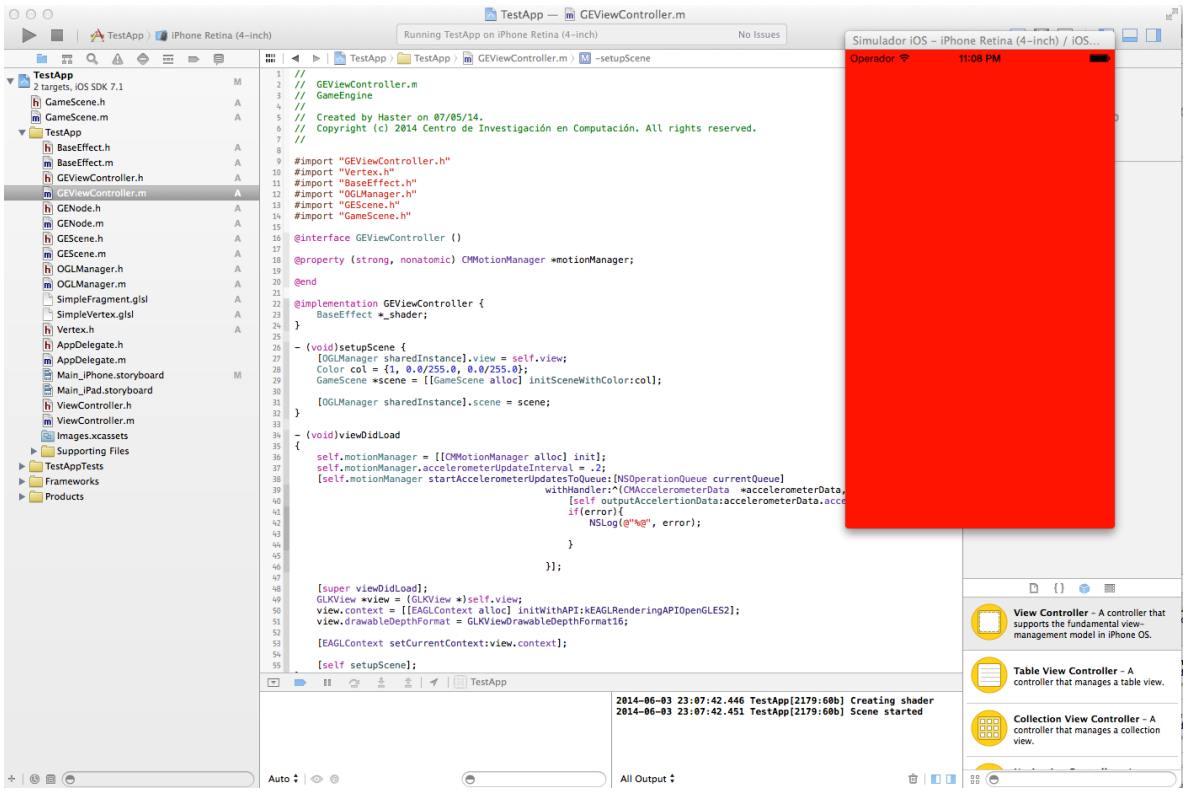


Figura 4.3 Configuración correcta de la escena en el motor de juegos.

Anexo B.
Código Fuente

Para el funcionamiento del motor de juegos, es necesario contar con las clases necesarias para la configuración de OpenGL y los mecanismo de renderizado. Los archivos necesarios se listan a continuación:

BaseEffect.h

BaseEffect.m

GENode.h

GENode.m

GEScene.h

GEScene.m

OGLManager.h

OGLManager.m

SimpleFragment.glsl

SimpleVertex.glsl

Vertex.h

Todas estas clases se encuentran en la distribución digital del presente trabajo.