UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Desarrollo de una aplicación gráfica multiplataforma con OpenGL y SDL

TESIS

Que para obtener el título de

Ingeniero en Computación

P R E S E N T A N

Albin Hernández González

Edgar Alberto Romero Popoca

DIRECTOR DE TESIS

Luis Sergio Valencia Castro

Ciudad Universitaria, Cd. Mx., 2016

Universidad Nacional Autónoma de México
UNAM
Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Agradecimientos:

A Jimena, a su padre, a Felipe, a mis gatos, a toda mi familia extendida, por su amor y apoyo, por estar presentes física y espiritualmente, por formarme como la persona que hoy soy, por enseñarme a amar, y por mantener la sonrisa que hay en mi cara todas las mañanas.

A la UNAM, por darme la oportunidad de conocer el mundo de la computación a través de sus ojos y la oportunidad de crecer como persona. A todos sus maestros y personas que dedican su vida a la formación de la comunidad universitaria.

A todos las personas que han tenido algo que ver con este trabajo, sin su apoyo y aportaciones mi educación no estaría completa. Sigamos creando.

- Albin Hernández González

A mi familia por apoyarme siempre y brindarme su confianza, especialmente a mis padres que me han apoyado durante toda mi vida y han estado a mi lado durante toda mi educación, no sólo durante mi permanencia en la Facultad de Ingeniería.

A la UNAM, a sus profesores, y a todas las personas que trabajan en ella, por darnos una de las mejores educaciones de toda latinoamérica y hacerlo sin costo alguno.

A mis amigos de la facultad y a todas las personas que me han apoyado a lograr mis metas y llegar a donde estoy en este momento.

- Edgar Alberto Romero Popoca

Índice

1. Resumen

El objetivo de la tesis es la creación de una aplicación gráfica que opere idénticamente en distintos sistemas operativos sin modificaciones al código fuente. El desarrollo se realizó con el lenguaje de programación C++ y las bibliotecas informáticas de OpenGL y SDL, además de otras herramientas de código abierto disponibles en Internet. Como requisitos, se consideraron los siguientes elementos básicos de una aplicación gráfica: creación y manejo de ventanas, carga y renderizado de modelos tridimensionales con materiales y texturas; transformaciones geométricas con matrices, carga y ejecución de programas para el GPU o *shaders*; control de dispositivos de entrada y salida, carga y reproducción de audio; registro e interpretación de eventos del sistema operativo y administración básica de procesos. La aplicación provee facilidades para la implementación de funcionalidad extendida: herramientas que facilitan la creación de niveles para un videojuego, implementación de funciones para un depurado rápido.

La aplicación se diferencia de otras herramientas en el mercado como Unity3D al compartir el código fuente, permitiendo al usuario ver la manera en que se crean y realizan las herramientas y operaciones anteriormente mencionadas. El usuario puede modificar cómo se realiza la carga de modelos a partir de sus vértices, índices y normales; analizar la creación de primitivas en tres dimensiones, las operaciones matemáticas involucradas, los shaders utilizados y su aplicación sobre cada uno de los modelos. El usuario puede inclusive modificar los shaders existentes para crear nuevos efectos o primitivas. En otros motores de aplicaciones gráficas, estas acciones son abstraídas para la conveniencia del usuario, es difícil saber cómo es que llevan a cabo todos los procesos para mostrar una imagen en pantalla, o qué se puede mejorar. La tesis sirve como apoyo para entender e implementarlos.

Tanto el motor como la aplicación son software libre, su código fuente se encuentra bajo la licencia pública general de GNU versión 3 (GNU General Public License ó GNU GPLv3) la cual permite a todos (alumnos de la facultad, organizaciones, etc) usar, estudiar, copiar, modificar o ampliar el software, siempre y cuando se incluya el encabezado de la licencia junto al código fuente y todos los trabajos derivados también se distribuyan bajo los términos de esta licencia.

2. Introducción

El desarrollo de aplicaciones multiplataforma requiere de conocimiento general sobre distintas áreas de software y hardware para construir aplicaciones que funcionen de la misma manera en distintos sistemas operativos. La palabra “plataforma” puede interpretarse como sistema operativo; ejemplos de plataforma pueden ser: Windows, Mac, Linux, y otros. Actualmente, un producto no puede ser llamado multiplataforma con sólo soportar diferentes versiones de un sistema operativo,[1] una aplicación puede considerarse multiplataforma si soporta por lo menos dos sistemas operativos distintos.

El software multiplataforma puede dividirse en dos tipos: uno que requiere de construcción individual o compilación para cada plataforma que soporta; y otro que puede ser ejecutado directamente en cualquier plataforma sin preparación especial, como por ejemplo: software escrito en un lenguaje de programación interpretado o su correspondiente bytecode precompilado, en estos lenguajes, tales como Java, un intérprete o compilador en tiempo de ejecución (JIT) es el componente responsable de ejecutarlos del mismo modo en todas las plataformas. Hoy en día estos componentes son estándares en todas las plataformas.[2]

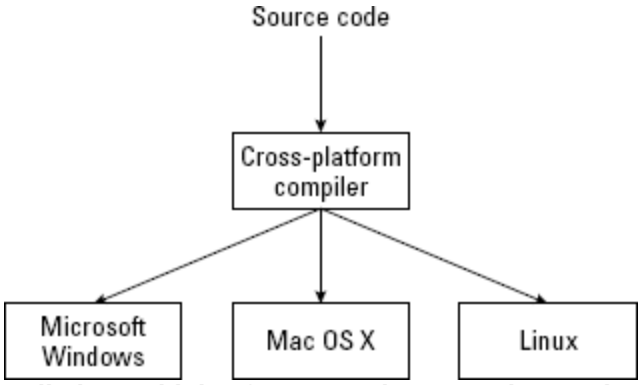


Figura 2.1 Un compilador multiplataforma puede convertir un solo código fuente en programas para múltiples sistemas operativos.[3]

Que una aplicación opere en múltiples plataformas quiere decir que el mismo código se ejecutará y comportará de la misma manera en todas las plataformas para las que fue diseñado. El compilador y otras herramientas serán específicos para cada máquina, y la Interfaz de Programación de Aplicaciones (*API*) puede variar, pero el código fuente es idéntico. La aplicación se desarrolla una sola vez, pero se puede utilizar varias veces en diferentes sistemas.[4]

El código fuente de una aplicación escrita en un lenguaje compilado requiere de diferentes ejecutables (llamados distribuciones) para cada plataforma. Algunos desarrolladores tienden a vender cada distribución de forma separada, lo cual significa que su producto debe ser comprado para cada plataforma en la que el usuario quiera utilizar el software, sin embargo, esto no es un requerimiento para el software

multiplataforma,[1] y existen distintas empresas distribuidoras de software como Steam, GOG, Humble Bundle y otras que han creado la tendencia de comprar el producto una sola vez y obtener todas las distribuciones para las plataformas en las que está disponible, ya sea Windows, Mac, Linux, Android, etc.[5][6]

Contexto del problema a resolver

Estado actual del mercado de las computadoras

El mercado actual de aplicaciones para computadoras es muy diverso, con aplicaciones para prácticamente todos los sectores de la industria: matemáticas, física, geografía, entretenimiento, astronomía, administración, salud, finanzas, música, enseñanza de idiomas, etc.

Debido al crecimiento de nuevas compañías, el consumidor en el mercado de la computación ha cambiado drásticamente, con la creciente y exitosa mercadotecnia de Apple, su sistema operativo y dispositivos móviles han logrado captar una parte importante de los consumidores. Como se puede observar en la figura 2.2, Apple logró un crecimiento mundial del 2.8% en el período del último cuarto del 2014 al último cuarto del 2015, mientras que el resto de las grandes compañías de PCs como HP, Asus, Dell, Acer y Lenovo registraron un crecimiento negativo (ventas de sus computadoras de escritorio, laptops y notebooks sin incluir tablets multimedia como el iPad).[7]

Preliminary Worldwide PC Vendor Unit Shipment Estimates for 4Q15 (Thousands of Units)

Company	4Q15 Shipments	4Q15 Market Share (%)	4Q14 Shipments	4Q14 Market Share (%)	4Q15-4Q14 Growth (%)
Lenovo	15,384	20.3	16,061	19.4	-4.2
HP	14,206	18.8	15,452	18.7	-8.1
Dell	10,236	13.5	10,783	13.1	-5.1
Asus	6,002	7.9	6,201	7.5	-3.2
Apple	5,675	7.5	5,519	6.7	2.8
Acer Group	5,277	7.0	5,939	7.2	-11.2
Others	18,940	25.0	22,635	27.4	-16.3
Total	**75,720**	**100.0**	**82,590**	**100.0**	**-8.3**

Source: Gartner (January 2016)

Figura 2.2 Ventas mundiales preliminares de PCs para el último cuarto del 2015.

Esta tendencia no sólo se ha mantenido en los últimos años, sino que ha aumentado desde que empezó a notarse a inicios del año 2006 en los Estados Unidos.[8]

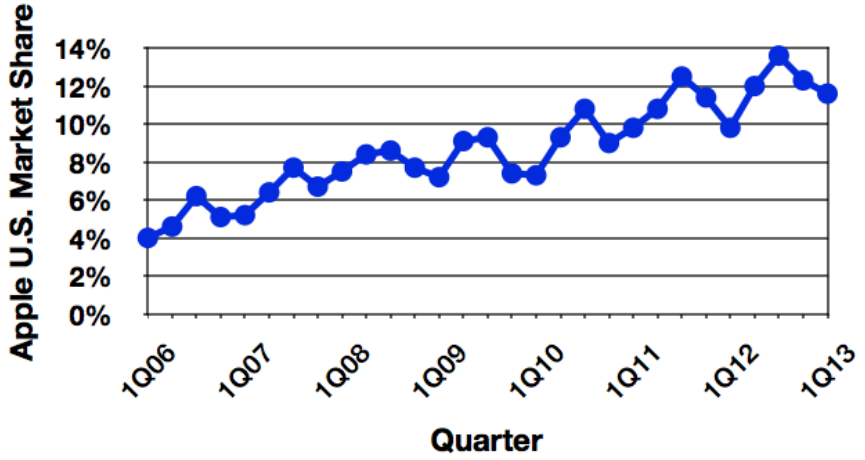
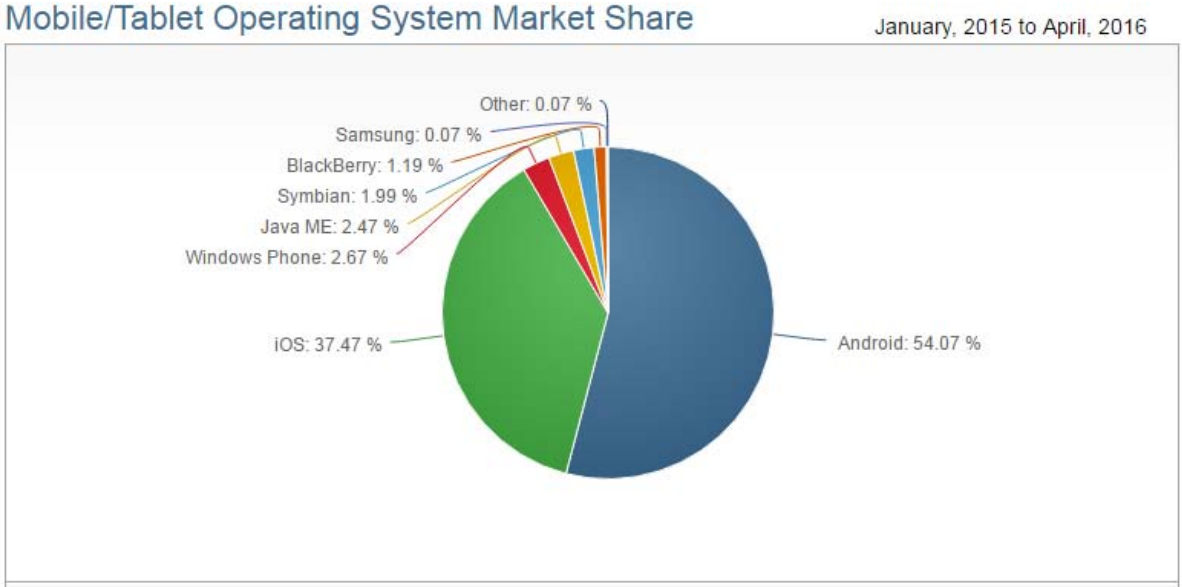


Figura 2.3 Tendencia de la cuota de mercado de Apple en Estados Unidos 1Q06-1Q13 (Gartner).

Este movimiento del mercado también se observa en los dispositivos móviles, el iPhone de Apple ha impulsado a la plataforma de escritorio Mac OS X, creando un ecosistema más viable para la venta de aplicaciones en estas plataformas.

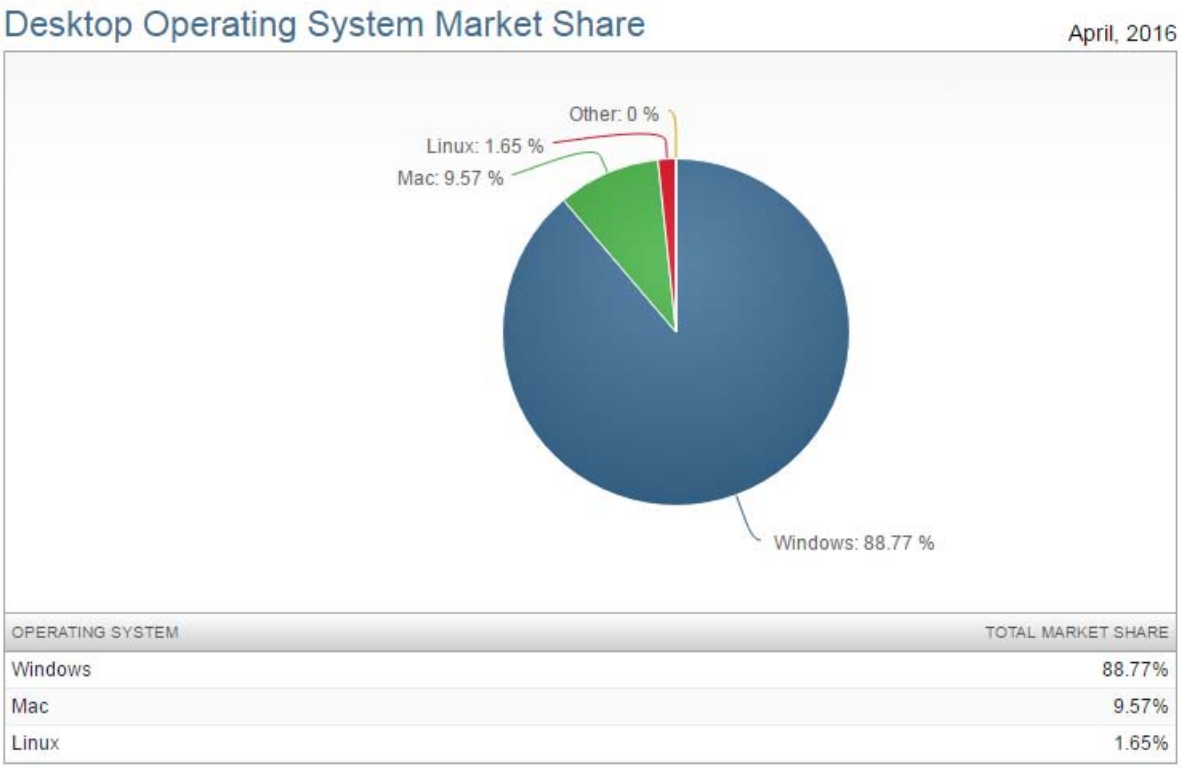


OPERATING SYSTEM	TOTAL MARKET SHARE
Android	54.07%
iOS	37.47%
Windows Phone	2.67%
Java ME	2.47%
Symbian	1.99%
BlackBerry	1.19%
Samsung	0.07%

Figura 2.4 Distribución del mercado actual de dispositivos móviles y tablets de acuerdo a su Sistema Operativo.

Al mismo tiempo, la comunidad de usuarios de Linux en todo el mundo ha crecido con lanzamientos como Ubuntu y PCLinuxOS. Linux además se vio impulsado en el mercado asiático por la falta de soporte de Windows XP.

Linux tiene una gran oportunidad de mercado, pues las aplicaciones gráficas interactivas nativas son pocas y no tan populares como en Windows.



OPERATING SYSTEM	TOTAL MARKET SHARE
Windows	88.77%
Mac	9.57%
Linux	1.65%

Figura 2.5 Distribución del mercado actual de computadoras de acuerdo al Sistema Operativo.[9]

China, uno de los líderes mundiales en el campo de la ciencia y la tecnología, ha tenido Microsoft Windows como el sistema operativo dominante por muchos años, sin embargo, debido a la falta de soporte para Windows XP y por razones de seguridad, el gobierno chino desarrolló una distribución de Linux llamada NeoKylin con la esperanza de dejar de depender de tecnología extranjera. Ubuntu Kylin viene pre instalado en el 40% de las computadoras Dell en China y es usado por muchas personas en el país. No es raro que gobiernos desarrollen su propio sistema operativo (usualmente basados en Linux) como alternativa a Windows, Corea del norte tiene uno, así como la ciudad de Munich y el gobierno Cubano.[10] Esto ha motivado a Microsoft a enfocarse en ofrecer una experiencia mejorada de Windows para facilitar la actualización, manejo y optimización de Windows 10 para las agencias gubernamentales de China y algunas empresas que son propiedad del gobierno.[11]

Con esta reciente diversificación en el mercado de las computadoras, la creación de software capaz de funcionar correctamente en distintas plataformas es importante. Aplicaciones limitadas a una sola plataforma no pueden aprovechar todo el mercado

disponible. El desarrollo de software está cambiando su enfoque, logrando que cada día más compañías desarrolladoras de software consideren brindar sus productos en más de una plataforma.

El desarrollo de aplicaciones multiplataforma ha incrementado drásticamente en los últimos años no sólo debido al crecimiento de sistemas operativos alternativos a Windows, sino también debido a la presencia de lenguajes de programación multiplataforma como Java, herramientas para el desarrollo como SDL y otros ambientes de desarrollo que han simplificado el proceso y su costo.[12] La popularidad de las tablets y la presencia de aplicaciones gratuitas han despertado también el interés por el desarrollo multiplataforma.

Una de las industrias ligadas a la computación que crece año con año es la industria de los videojuegos. Una serie de estudios realizados por la compañía de inteligencia DFC estima que el mercado global de los videojuegos crecerá de 67 billones de dólares en 2012 a 82 billones de dólares en 2017. Este estudio incluye las ganancias de consolas de hardware y software dedicado y portátiles, videojuegos de computadora y videojuegos para dispositivos móviles como teléfonos, tablets, reproductores de música y otros aparatos capaces de reproducir videojuegos como una de sus características. DFC explica que el crecimiento de la industria ha perdido impulso debido al declive de ventas en el segmento de las consolas.

De acuerdo con el analista de inteligencia de la DFC David Cole, las nuevas consolas de Nintendo, Microsoft y Sony ayudan a impulsar las ventas, sin embargo, el área más estable y segura de crecimiento es la de las computadoras, seguida de cerca por los dispositivos móviles.

Las ventas de videojuegos de computadora continúan incrementando a paso firme y se espera que sobrepase los 25 billones de dólares en 2017, de los 20 billones de dólares contemplados en 2012. Mientras que los videojuegos para móviles y smartphones están dominando el crecimiento en la categoría de portátiles. El índice de crecimiento más grande de todos, es el de la distribución en línea y modelos de negocio por el uso de servicios en línea, pues continúan quitando ganancias del modelo tradicional de venta de software en caja por medio de tiendas físicas. Se estiman ganancias por venta de videojuegos en línea de 35 billones de dólares para el 2017, de los 19 billones de dólares en 2011. Se prevé que para el 2017, el 39% del total de ventas de videojuegos para consolas será por medio de distribución en línea.[13]

El comportamiento registrado en los estudios de la DFC coincide con los estudios de otras grandes compañías de inteligencia respetadas como Gartner, e industrias ligadas a la industria de los videojuegos como Nvidia y Valve, se observa una tendencia creciente en el mercado de las computadoras y dispositivos móviles, con el mercado de los videojuegos en computadoras llegando incluso a superar a las consolas de videojuegos a partir del año 2014.

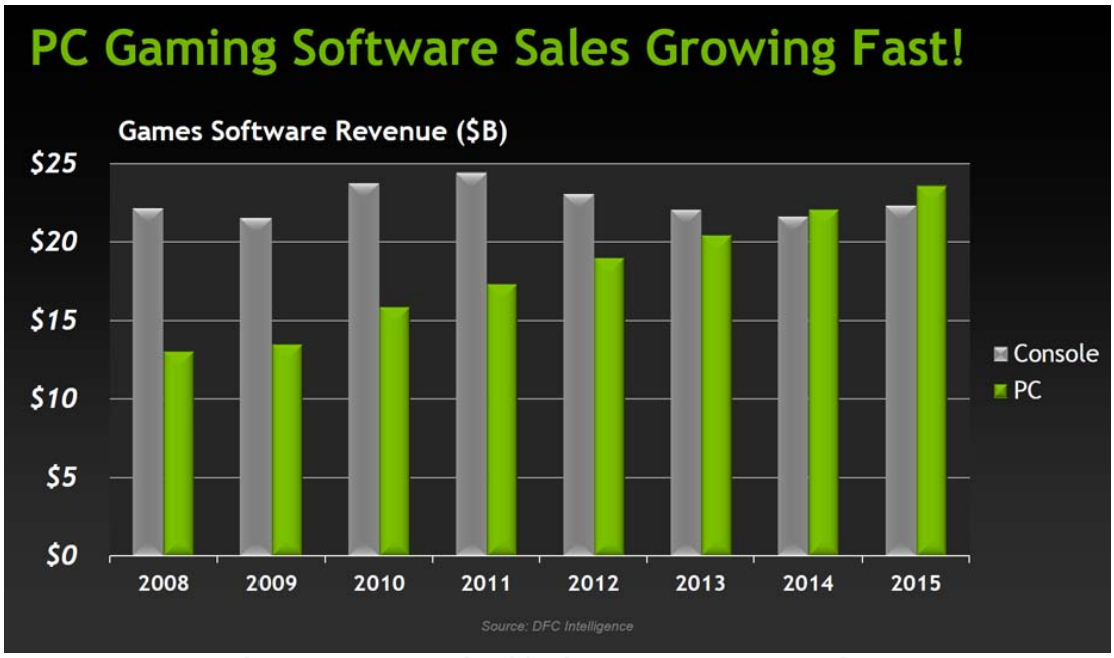


Figura 2.6 Ventas de videojuegos en PC y consolas.

Estos estudios coinciden también con lo reportado por la DFC, indicando que el medio de distribución óptimo para este tipo de aplicaciones es el digital, pues supera por mucho al medio de distribución físico en el campo de los videojuegos para computadora.

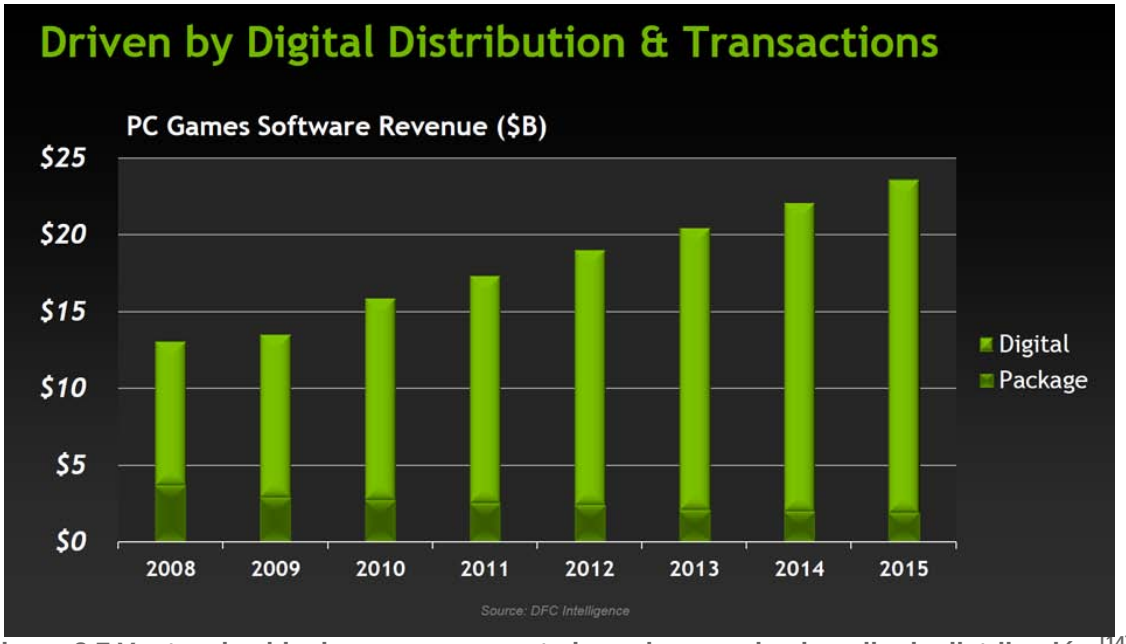


Figura 2.7 Ventas de videojuegos en computadora de acuerdo al medio de distribución.[14]

Al desarrollar aplicaciones para computadora, un factor importante a considerar es el hardware con el que cuentan los usuarios, la compañía distribuidora de videojuegos *Steam* considerada como una de las más importantes en la actualidad, realiza una encuesta mensual con el fin de recaudar este tipo de datos. En sus resultados se puede

observar la gran variedad de hardware y software utilizado, dominando el sistema operativo Windows, los procesadores Intel y las tarjetas de video NVIDIA.

La configuración de hardware y software más común en la plataforma digital de Steam es Windows 10 (64 bit), con memoria RAM de 8 GB, procesador de 2.3 Ghz a 2.69Ghz, 2 procesadores físicos, tarjeta de video NVIDIA GeForce GTX 970, y disco duro de 250 GB a 499 GB.

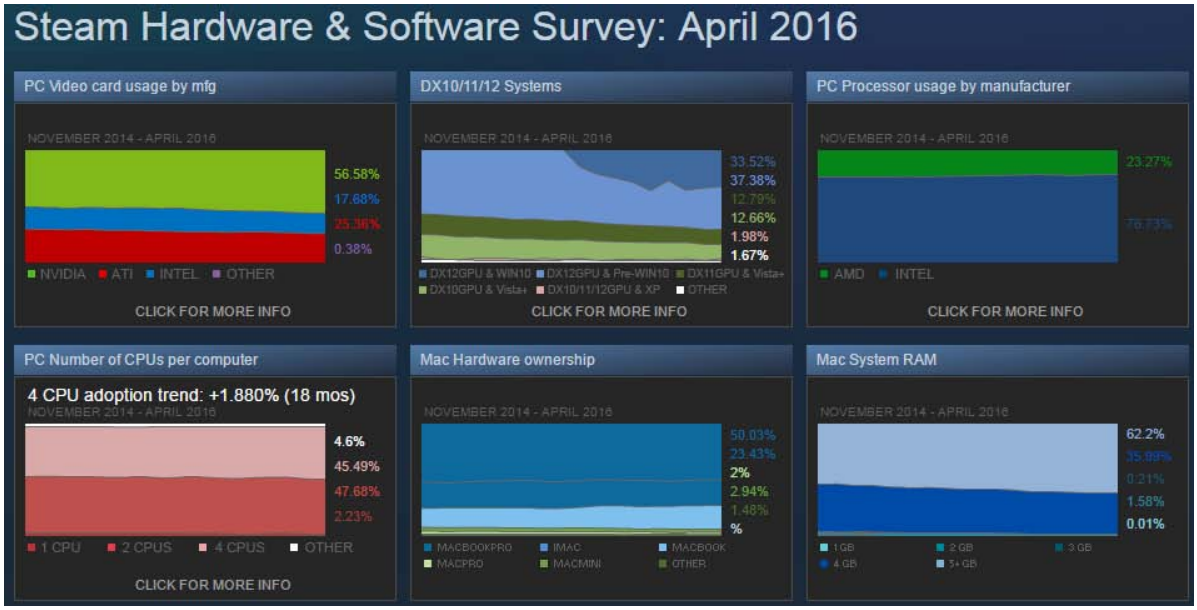


Figura 2.8 Estadísticas de Hardware y Software empleado por los usuarios de Steam.

Aún así, la presencia de Windows XP en el mercado aún es muy grande, Windows XP de 32 bits representa más del 2% del mercado de Steam, mientras que Mac tiene un 3.57% y Linux tiene 0.90%, como lo muestra la Figura 2.9. Por lo que toda aplicación multiplataforma debe considerar a Windows XP como un sistema relevante, pese a la gran proliferación de Windows 10.

OS Version			
Windows		95.43%	-0.33%
	Windows 10 64 bit	38.18%	+1.36%
	Windows 7 64 bit	32.53%	-0.81%
	Windows 8.1 64 bit	11.89%	-0.97%
	Windows 7	7.01%	+0.04%
	Windows XP 32 bit	2.03%	+0.04%
	Windows 8 64 bit	1.60%	-0.04%
	Windows 10	1.33%	+0.05%
	Windows 8.1	0.34%	0.00%
	Windows Vista 32 bit	0.22%	-0.01%
	Windows 8	0.15%	+0.01%
	Windows Vista 64 bit	0.11%	-0.01%
OSX		3.57%	+0.31%
	MacOS 10.11.4 64 bit	0.94%	+0.78%
	MacOS 10.10.5 64 bit	0.84%	+0.05%
	MacOS 10.11.3 64 bit	0.74%	-0.52%
	MacOS 10.9.5 64 bit	0.29%	+0.01%
	MacOS 10.11.2 64 bit	0.17%	-0.05%
	MacOS 10.11.1 64 bit	0.12%	-0.01%
	MacOS 10.11.0 64 bit	0.08%	0.00%
	MacOS 10.10.3 64 bit	0.08%	0.00%
	MacOS 10.7.5 64 bit	0.06%	+0.01%
	MacOS 10.8.5 64 bit	0.06%	+0.01%
	MacOS 10.10.4 64 bit	0.06%	+0.01%
	MacOS 10.10.2 64 bit	0.05%	0.00%
Linux		0.90%	+0.06%
	Ubuntu 14.04.4 LTS 64 bit	0.17%	0.00%
	Ubuntu 15.10 64 bit	0.17%	-0.01%
	Linux Mint 17.3 Rosa 64 bit	0.09%	0.00%
	Linux 64 bit	0.09%	+0.01%
	Ubuntu 16.04 LTS 64 bit	0.08%	+0.08%

Figura 2.9 Estadísticas de los sistemas operativos empleados por los usuarios de Steam.[15]

Fuera de la plataforma de venta de videojuegos Steam, Windows XP aún representa el 11% del mercado total de Windows en todo el mundo a la fecha de Octubre del 2015, como se observa en la figura 2.10.

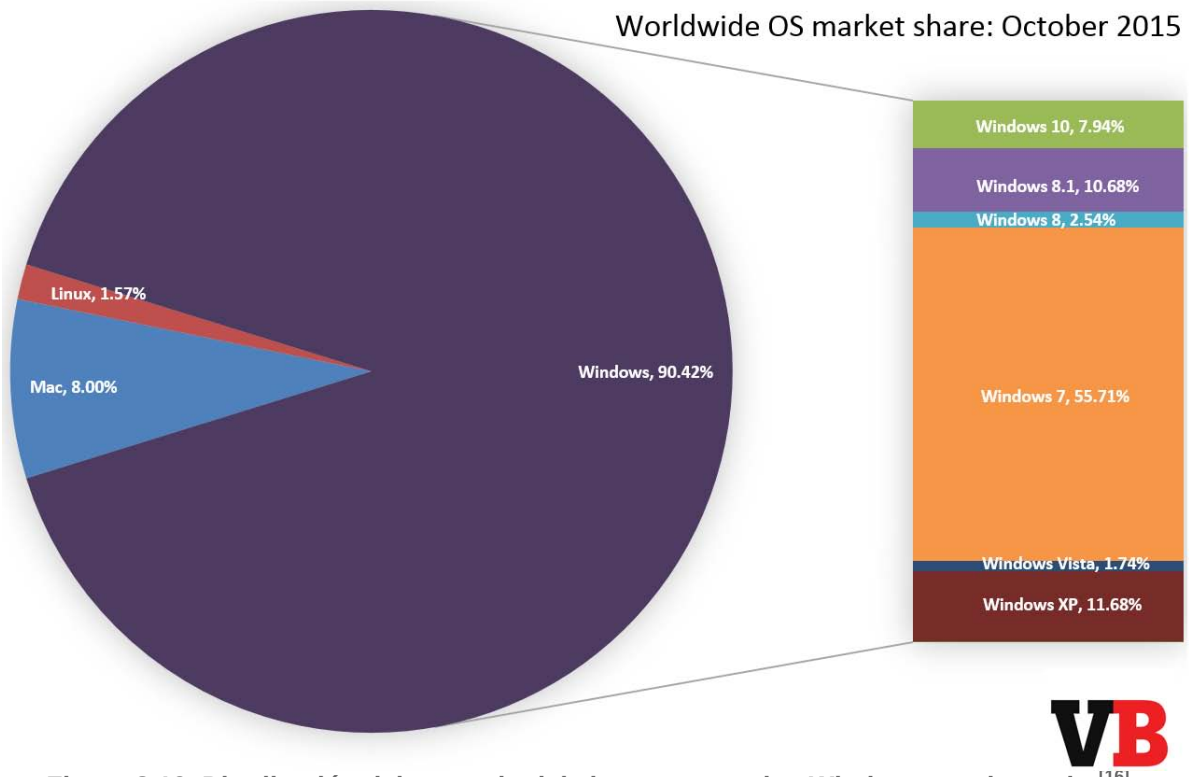


Figura 2.10 Distribución del mercado del sistema operativo Windows en el mundo.[16]

Ventajas del desarrollo de aplicaciones multiplataforma

Realizar una aplicación multiplataforma provee mayor accesibilidad. No hay necesidad de tener determinado sistema operativo o dispositivo. Se expone el producto a más clientes sin esfuerzo adicional.

El desarrollar aplicaciones multiplataforma da la posibilidad de conocer distintas estrategias de desarrollo. Se puede asignar todo el peso del desarrollo a una sola plataforma de acuerdo a las habilidades de los desarrolladores y posteriormente probarlo en las demás plataformas, o bien, una aplicación puede ser desarrollada y probada en múltiples plataformas en paralelo. A cada persona involucrada en el proyecto se le puede ofrecer la posibilidad de elegir la plataforma de desarrollo con la cual sea más experimentada.[10]

Una vez que se ha decido hacer una aplicación multiplataforma, la creación de un solo código base también tiene sus ventajas . En vez de crear múltiples ramas de código que se deben de mantener y actualizar por separado, los esfuerzos se concentran en mejorar la aplicación.

El desarrollo multiplataforma lleva a los desarrolladores a probar su código bajo múltiples condiciones: diferente hardware, gestión de memoria, funciones provistas en las librerías comunes. Incluso las diferencias existentes en las herramientas para la depuración llevan a la aplicación a tener mejor calidad, ya que ha sido probada extensamente. Por ejemplo, los compiladores particulares de cada plataforma tienen diferentes herramientas que ayudan a detectar distintos posibles errores a futuro.[17]

El exponer el código a varios compiladores, permite encontrar errores de manera temprana en las distintas fases de desarrollo. Incluso el seguimiento de bugs se facilita, no existen múltiples versiones de código con un mismo problema, arreglar el problema en una plataforma lo arregla en otra. No se necesita de tener un equipo de programadores distinto para mantener cada plataforma, los recursos pueden distribuirse de manera eficiente en un equipo de desarrollo.

Aunque la preparación y planeación de una aplicación multiplataforma puede llevar tiempo. El resultado final es que los desarrolladores pueden escribir código reusable que corra en múltiples plataformas con poco esfuerzo.

El desarrollo multiplataforma además promueve un mejor diseño de objetos programables, con pocas dependencias y útiles para el desarrollo de futuras aplicaciones.

Incluso hay ventajas para el departamento de marketing, una aplicación multiplataforma permite poder vender en distintas plataformas, con un sólo presupuesto destinado a la promoción y comercialización. Un mismo anuncio, con la leyenda “Disponible para PC, Mac y Linux” captura todos los mercados.

En el caso de las aplicaciones gráficas, en particular los videojuegos, los jugadores se encuentran más involucrados. Un juego multiplataforma que le da la opción a los jugadores de seguir su uso en dispositivos móviles, incrementa el tiempo que pasan dentro de la aplicación, pueden ir de la computadora a su iPhone o iPad para seguir jugando.[18]

Retos del desarrollo multiplataforma

Las aplicaciones gráficas en particular, deben enfocarse en la creación de una interfaz consistente y con elementos aislados de los provistos por alguna plataforma en particular, ya que no se puede contar con estos.

Es un reto para los desarrolladores, por las diferencias de hardware y software existentes, debe ponerse especial cuidado al momento del desarrollo en detalles como el set de caracteres utilizados para escribir, pues pueden generar problemas de compilación. Por ejemplo, la codificación del fin de línea no es la misma para Windows y Linux.

Se debe evitar utilizar algoritmos que consuman mucha memoria, porque los recursos de memoria no son los mismos para todas las plataformas.

Se requiere de una administración adicional al momento de construir una aplicación multiplataforma, deben establecerse diferentes reglas para diferentes compiladores. Para poder trabajar con los mismos datos sin problemas deben de haber recursos comunes en todas las plataformas.

Existen varios elementos que son comunes entre distintas plataformas, mientras que algunos son invariables. Algunos de estos elementos pueden explotarse para facilitar la tarea de desarrollo de una aplicación:

- Procesadores: distintas plataformas trabajan con procesadores con tecnologías similares, por ejemplo, la mayoría posee procesadores con varios núcleos, lo cual permite escribir aplicaciones con escalabilidad automática, el rendimiento mejora dependiendo del hardware y no depende del software.
- Lógica del juego: no cambia drásticamente dentro de las diferentes plataformas.
- Componentes comunes: Inteligencia Artificial, física, sonido e implementación de comunicación red pueden hacerse genéricos gracias a los estándares con los que trabajan.
- Lenguaje de programación: para facilitar el desarrollo debe de elegirse un lenguaje que sea soportado en todas las plataformas.

Las diferencias entre las distintas plataformas presentan un reto para la implementación, las más importantes son:

- Arquitectura del procesador: El procesador de cada plataforma puede variar y también así su arquitectura; las PCs emplean una arquitectura basada en un conjunto de instrucciones x86, las consolas de videojuegos emplean una arquitectura RISC, y los teléfonos móviles una arquitectura ARM.
- Librerías Gráficas: Las computadoras emplean las librerías de OpenGL y DirectX, mientras que las consolas de videojuegos utilizan librerías desarrolladas por sus creadores, además de, en algunos casos proveer las anteriores.
- Gestión de memoria: El proceso de gestión de memoria cambia dependiendo de la plataforma y compilador específicos, depende también del tipo de datos a utilizar.
- Endianness del procesador: Al realizar operaciones con datos en múltiples plataformas, los resultados serán distintos, creando fallas y bugs, esto se debe a que la endianness depende de cada procesador.

Conexión a otros problemas

Una aplicación gráfica, interactiva y multiplataforma requiere del diseño de una arquitectura capaz de:

- Crear el contexto bajo la cual correrá en varios sistemas operativos: la creación y administración de la ventana donde se ejecutará, el código necesario para poder recibir distintos dispositivos de entrada (teclado, mouse), y operaciones como la inicialización y terminación de la aplicación.
- Ejecutar la lógica de la aplicación interactiva: la creación de subsistemas que administren el estado de la aplicación, que puedan comunicar este estado a otros subsistemas, y recibir de éstos información (comandos de entrada), que refuercen las reglas de la aplicación para todos los objetos que se encuentran en él (sistema de física).
- Presentar los gráficos de la aplicación: ser responsable de dibujar el estado de la aplicación en pantalla, traducir los comandos de entradas en comandos reconocibles para la aplicación, enviar sonido a los dispositivos de salida.

Estos problemas inherentes en toda aplicación gráfica, se ven mitigados cuando la plataforma para la que se quiere desarrollar es una en específico. Sin embargo, cuando se quiere desarrollar una aplicación que pueda funcionar sin cambios en el código en otras plataformas, una serie estricta de reglas deben seguirse para poder conseguir este objetivo; no se puede contar con métodos específicos para una plataforma, ni con el hardware específico de una sola computadora, es por esto que la aplicación a desarrollar debe de:

- Simplificar y unificar las tareas de comunicación de la aplicación con el sistema operativo, en el caso de las tres plataformas objetivo, se debe evitar el contacto directo con X11, Quartz/Cocoa y DirectX, para GNU/Linux, Mac OSX y Microsoft Windows respectivamente.
- Tener un manejador de entrada y salida debe ser coherente entre los diferentes sistemas operativos, evitar el uso de recursos de entrada y salida que sean usados sólo en una plataforma específica.

Soluciones actuales en el mercado para el desarrollo de aplicaciones gráficas multiplataforma

Un objetivo de este trabajo es realizar una aplicación gráfica, por lo que es importante explorar las soluciones y herramientas que permitan entender las diferencias entre plataformas a bajo nivel y documentar las soluciones a los problemas más comunes.

Existen muchas herramientas y aplicaciones de código abierto disponibles para los desarrolladores con el propósito de crear aplicaciones multiplataforma, descargables y libres de costo para cualquiera con una conexión de internet.

A continuación se presenta un estudio de las distintas soluciones que fueron comparadas para la creación de la aplicación. Para considerar una herramienta, ésta debía:

- Ser gratuita
- Permitir el desarrollo multiplataforma
- Tener presencia en la industria, que se hayan desarrollado juegos con ella.

Aunque no se llegaron a analizar todas las herramientas, se conocían algunas de las elegidas debido a los estudios realizados en la universidad y a su popularidad actual.

OpenGL

OpenGL es una API multiplataforma para el renderizado de gráficos por computadora 2D y 3D, típicamente usada para interactuar con el GPU y lograr renderizado acelerado por el hardware.

La especificación de OpenGL describe una interfaz abstracta para dibujar 2D y 3D, aunque es posible implementar todas sus funciones vía software, está diseñada para ser implementada en su mayoría o completamente por hardware.

La API se encuentra definida por un número de funciones que pueden ser llamados por el programa cliente, junto con un número de constantes enumeradas, por ejemplo GL_TEXTURE_2D, que corresponde al número decimal 3553. Las funciones se encuentran definidas similares al lenguaje de programación C, pero son independientes del lenguaje. De este modo, OpenGL puede tener diversos vínculos con varios lenguajes, algunos de los más notables siendo JavaScript con WebGL, C con WGL, GLX y CGL, además de vínculos para iOS, y Android, Java con víclulos para Android.

Además de ser independiente de todos los lenguajes de programación, es independiente de todas las plataformas.

No obstante, la especificación de OpenGL no especifica nada acerca de cómo obtener y administrar un contexto para hacer uso de la interfaz, es una tarea que se deja al sistema de ventanas del sistema operativo. OpenGL se enfoca en renderizar, sin proveer APIs relacionadas con control de entrada y salida, audio o ventanas.

La interfaz de OpenGL está basada en procedimientos de bajo nivel que requieren al programador dictar paso a paso lo necesario para renderizar una escena. Esto contrasta con las APIs descriptivas, donde un programador sólo debe describir la escena y puede dejar que la biblioteca controle los detalles para representarla.

El funcionamiento básico de OpenGL consiste en aceptar primitivas tales como puntos, líneas y polígonos, y convertirlas en píxeles. Este proceso es realizado por una pipeline gráfica conocida como la máquina de estados de OpenGL. La mayor parte de los comandos de OpenGL emiten primitivas a la pipeline gráfica o bien configuran la

manera en que la pipeline procesa dichas primitivas. OpenGL es típicamente implementado como una biblioteca de puntos de entrada y hardware gráfico para apoyar a la librería, como se muestra en la figura 2.11.

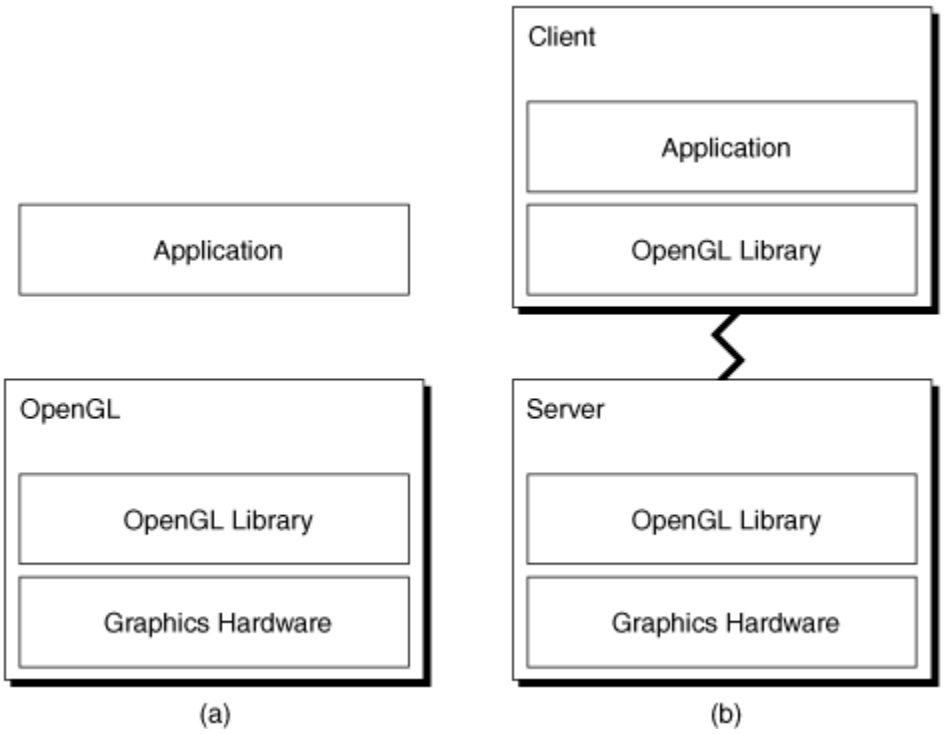


Figura 2.11 Dos Implementaciones típicas de OpenGL. En (a), las aplicaciones se vinculan con una biblioteca de puntos de entrada a OpenGL que pasan comandos al hardware. (b) Ilustra una implementación cliente-servidor. Las aplicaciones se vinculan con una biblioteca de puntos de entrada a OpenGL que pasan comandos a través de un protocolo de red. Un servidor (remoto o local) recibe los comandos y los renderizá usando el hardware del servidor.[19]

Para entender cómo funciona el proceso básico en OpenGL, la figura 2.12 representa una versión simplificada del pipeline de renderizado de OpenGL.

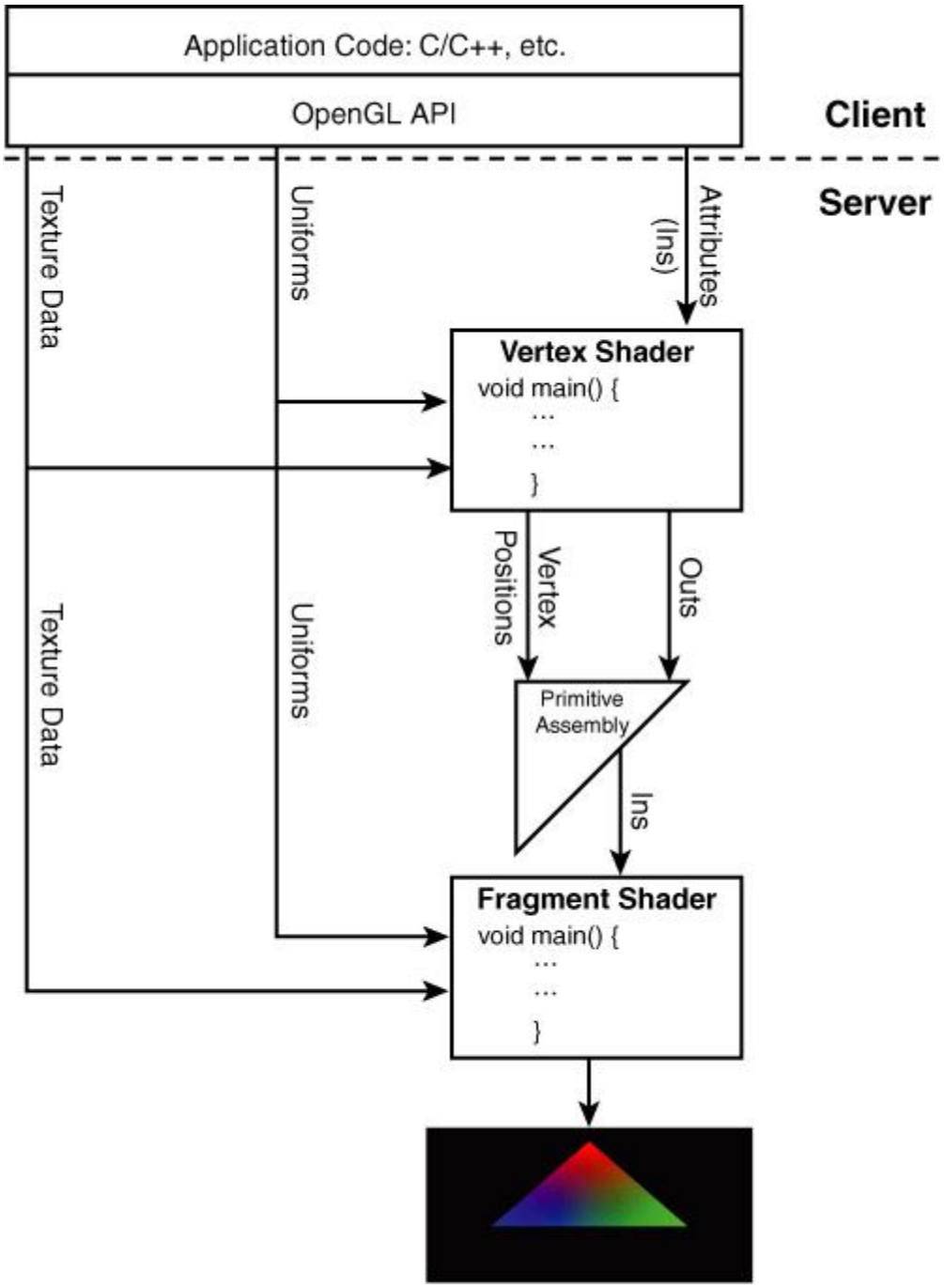


Figura 2.12 Pipeline simplificado de renderizado de OpenGL.

Notamos que la pipeline está dividida en dos. En la parte de arriba está el cliente, y en la parte de abajo se encuentra el servidor. El diseño básico de cliente-servidor es aplicado cuando la parte del cliente del pipeline está separada de la del servidor. En el caso de OpenGL, el cliente es código que reside en la memoria del CPU principal y se ejecuta en un programa, o en un controlador dentro de la memoria principal del sistema. El controlador ensambla comandos de renderizado y datos, y los manda al servidor para

su ejecución. En un computadora de escritorio típica, el servidor se encuentra en algún bus del sistema y es de hecho el hardware y memoria en la tarjeta de gráficos.

El cliente y servidor funcionan de manera asíncrona , es decir, son elementos independientes de hardware o software, o ambos. Para maximizar el rendimiento se desea tener ambos lados tan ocupados como sea posible. El cliente está continuamente ensamblando bloques de datos y comandos dentro de buffers que son mandados al servidor para su ejecución. El servidor ejecuta esos buffers, mientras al mismo tiempo el cliente se está preparando para mandar el próximo grupo de datos o información a renderizar. Si el servidor se queda sin trabajo alguna vez mientras espera al cliente, o si el cliente tiene que parar y esperar a que el servidor esté listo para recibir más comandos, a esto se le conoce como pipeline stall o bubble (pipeline atorada o detenida). Las pipeline stall son muy malas para el rendimiento, pues no se debe tener al CPU o al GPU inactivos mientras esperan por trabajo para hacer.

Los dos grandes cuadros de la figura 2.12 representan al shader de vertices y al shader de fragmentos. Los shaders son programas escritos en GLSL que se ejecutan en el GPU y se utilizan para hacer sombreado plano o shading.

GLSL (OpenGL Shading Language) es un lenguaje de alto nivel muy parecido a C que es compilado y vinculado por OpenGL y usualmente se ejecuta completamente en el hardware para gráficos.

Una gran ventaja de OpenGL es qué nueva funcionalidad es agregada a partir de extensiones, independientemente del sistema operativo. Es un API que no se encuentra atado a versiones específicas como DirectX, pueden realizarse un gran número de efectos nuevos sin la necesidad de obligar al usuario a cambiar de sistema operativo.

Java

Java es un lenguaje de programación de propósito general, concurrente, basado en clases y orientado a objetos, específicamente diseñado para tener tan pocas dependencias de implementación como sean posibles. Java permite a los desarrolladores escribir software en una plataforma y correrlo virtualmente en cualquier otra plataforma.[20]

El lenguaje Java emplea una máquina virtual (JVM) para correr todo el código que sea escrito por el lenguaje. Esto permite que el mismo binario ejecutable corra en todos los sistemas que soportan el lenguaje a través de la Java Virtual Machine.

Los ejecutables de Java no corren nativamente en el sistema operativo, a pesar de esto, la JVM es capaz de proveer servicios relacionados con el el sistema operativo, como entrada y salida de archivos y acceso de red, si se proveen los permisos necesarios. Las aplicaciones en Java son compiladas a bytecode Java que puede correr en cualquier máquina virtual Java (JVM) sin importar la arquitectura de la computadora. Para ello, se compila el código fuente escrito en lenguaje Java y se genera un código conocido como Java bytecode (instrucciones máquina simplificadas específicas de la plataforma Java). El

bytecode Java es un código intermedio entre el código fuente y el código máquina que entiende el dispositivo destino. El bytecode es ejecutado entonces en la máquina virtual (JVM), que interpreta y ejecuta el código. Además, se suministran bibliotecas adicionales para acceder a las características de cada dispositivo (como los gráficos, ejecución mediante hebras o threads y la interfaz de red) de forma unificada. Se debe tener presente que, aunque hay una etapa explícita de compilación, el bytecode generado es interpretado o convertido a instrucciones máquina del código nativo por el compilador JIT.

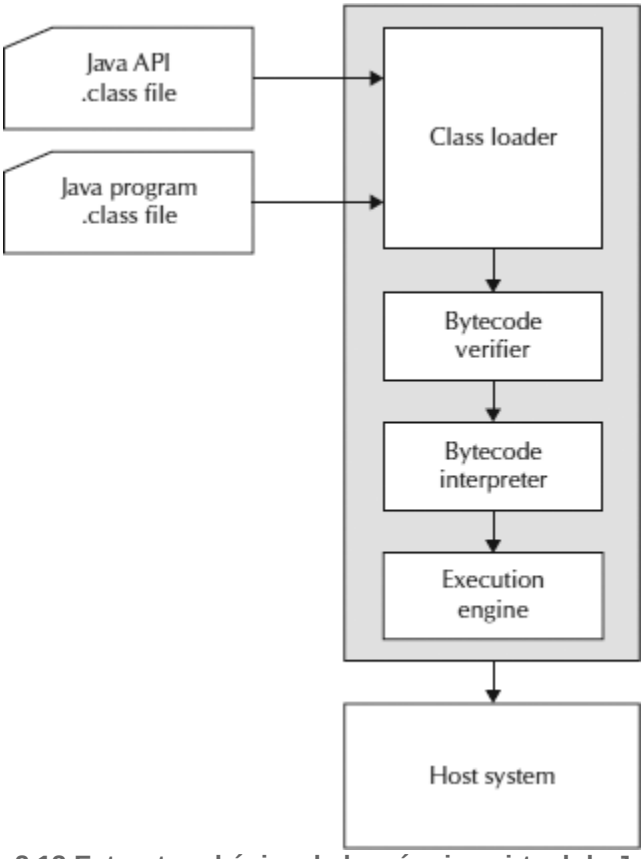


Figura 2.13 Estructura básica de la máquina virtual de Java.[21]

Los programas escritos en Java tienen la reputación de ser más lentos y requerir más memoria que los escritos en C++. Sin embargo, la velocidad de ejecución de los programas escritos en Java mejoró significativamente con la compilación justo a tiempo (JIT) para la versión de Java 1.1, la adición de funcionalidad del lenguaje dando soporte a un mejor análisis de código (como la clase StringBuffer, asserts opcionales, etc.), y optimizaciones en la máquina virtual de Java misma (como HotSpot que se convirtió en la JVM default de Sun en el año 2000). En diciembre de 2012 benchmarks mostraban que Java 7 era aproximadamente 44% más lento que C++.

En Java el problema de fugas de memoria se evita en gran medida gracias al recolector de basura automática (o *automatic garbage collector*). El programador determina cuándo se crean los objetos y el entorno en tiempo de ejecución de Java (Java runtime) es el responsable de gestionar el ciclo de vida de los objetos.[22]

Java provee varias opciones para vincularse con OpenGL, como Java OpenGL o LWJGL, las cuales proveen acceso a funciones de OpenGL con mente en la eficiencia.

C / C++

C es uno de los lenguajes de programación de propósito general más populares en todo el mundo. El lenguaje C no está atado a ningún sistema operativo en particular. Puede ser utilizado para desarrollar nuevos sistemas operativos. Véase la figura 2.14 en la cual el lenguaje C se muestra asociado con múltiples sistemas operativos. El lenguaje C está íntimamente asociado con el sistema operativo UNIX. El código fuente del sistema operativo UNIX está escrito en C. C corre bajo un número de sistemas operativos que incluye MS.DOS. Los programas en C son eficientes, rápidos y altamente portables, es decir, un programa escrito en C que se adhiera a los estándares puede ser fácilmente compilado en otra plataforma con mínimos cambios en el código fuente

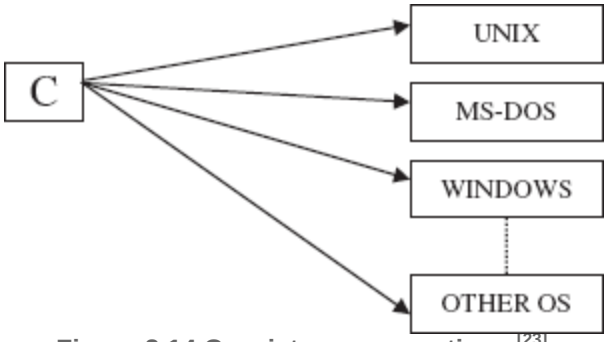


Figura 2.14 C y sistemas operativos.[23]

Aunque el lenguaje de programación C está enfocado al desarrollo de aplicaciones de bajo nivel, el lenguaje fue diseñado para promover el desarrollo multiplataforma. C es de más bajo nivel que la mayoría de los lenguajes; lo que significa que crea código que está más cerca a lo que las computadoras entienden (un flujo binario de 1's y 0's conocido como código máquina). C es usado cuando la velocidad, espacio y portabilidad son importantes. El lenguaje se encuentra disponible en una gran variedad de plataformas, desde microcontroladores hasta supercomputadoras. A diferencia de Java, C se corre nativamente en la plataforma objetivo.

Se debe de tener un cuidado especial al momento de utilizar bibliotecas dentro del lenguaje, ya que algunas no se encuentran en todas las plataformas o su comportamiento se encuentra definido de distinta manera. C es un lenguaje de programación imperativo y procedural, un programa en C consiste en bloques conocidos como funciones, cada función ejecuta una tarea específica independientemente.

C es la base de otros lenguajes de programación más avanzados, tales como C++, también está siendo usado actualmente para el desarrollo de aplicaciones móviles en la

forma de Objective C. Objective C es C estándar con una ligera adición de funcionalidad de programación orientada a objetos.[24]

El lenguaje de programación C++ provee a C con funcionalidad orientada a objetos, convirtiéndolo en un lenguaje híbrido o multiparadigma. C++ se considera como un lenguaje de programación de nivel intermedio. Es uno de los lenguajes más populares, y se encuentra implementado en una gran variedad de hardware y sistemas operativos. Debido a que es un compilador eficiente a código nativo, su ámbito de aplicación incluye software de sistemas, controladores de dispositivos, software embebido, software cliente-servidor de alto rendimiento y, software de entretenimiento como los videojuegos. C++ ha influenciado a muchos otros lenguajes de programación populares, más notablemente a C# y Java. Una particularidad de C++ es la posibilidad de redefinir los operadores (operator overloading), y de poder crear nuevos tipos de datos que se comporten como tipos fundamentales.[25]

Al desarrollar aplicaciones multiplataforma en C y C++ se requiere un poco más de diseño y planeación, ya que se debe asegurar utilizar estructuras y representaciones de datos que ayuden a hacer el programa más portable.[26] Sin embargo, existe gran soporte del lenguaje y la capacidad de ejecutar programas de forma nativa en cada una de las plataformas, lo que brinda el rendimiento y control necesarios para realizar todo tipo de aplicación.

SDL

Simple DirectMedia Layer es una biblioteca multimedia multiplataforma, diseñada para proveer acceso de bajo nivel al audio, teclado, mouse, joystick, hardware 3D vía OpenGL, y buffer de video 2D. SDL soporta las plataformas de Linux, Windows, Windows CE, BeOS, MacOS, Mac OS X, FreeBSD, NetBSD, OpenBSD, BSD/OS, Solaris, IRIX, y QNX. El código contiene soporte para AmigaOS, Dreamcast, Atari, AIX, OSF/Tru64, RISC OS, SymbianOS, y OS/2, pero estos no están oficialmente soportados.

SDL está escrito en C, pero trabaja con C++ nativamente, y tiene vínculos con otros varios lenguajes, incluyendo Ada, C#, D, Eiffel, Erlang, Euphoria, Go, Guile, Haskell, Java, Lisp, Lua, ML, Oberon, Objective C, Pascal, Perl, PHP, Pike, Pilant, Python, Ruby, Smalltalk, y Tcl.

SDL está distribuida bajo la licencia zlib, la cual permite usar SDL libremente y de manera gratuita para uso en programas comerciales.[27]

SDL presenta una interfaz sencilla para acceder a gráficos, audio y controles en varias plataformas. SDL tiene la palabra “layer” (capa) en su título porque en realidad es una colección de subrutinas y clases construidas alrededor de funciones específicas de cada sistema operativo. Su propósito principal es el de proveer un framework común para acceder a estas funciones. Para proveer mayor funcionalidad, muchas bibliotecas han sido creadas alrededor de estas funciones.

Unity3D

Unity es un ecosistema de desarrollo: un engine de rendereo poderoso totalmente integrado con un conjunto completo de herramientas intuitivas para crear contenido 3D; fácil publicación multiplataforma, mucha calidad, y muchos recursos como modelos, materiales y texturas disponibles en su tienda AssetStore.[28]

Unity tiene un IDE incluido, desarrollado por Unity Technologies. Unity es usado para el desarrollo de videojuegos para la web mediante plugins, para plataformas de escritorios y dispositivos móviles, incluso es capaz de publicar juegos en consolas. Es utilizado por más de un millón de desarrolladores. Aunque el motor de juegos está escrito en C/C++, soporta código escrito en C#, JavaScript o Boo. Actualmente soporta iOS, Android, Windows, Mac OS X, Linux/Steam OS, exploradores de internet, Flash, PS4, PSVita, Xbox One, Xbox 360, 3DS, WiiU, Samsung SMART TV y tvOS.

El motor gráfico hace uso de Direct3D para Windows, OpenGL para Mac, Windows y Linux, OpenGL ES para Android, iOS y APIs de terceros para las consolas como WiiU, en versiones recientes, incluso tiene soporte para la nueva librería de gráficos Metal de OS X. Unity tiene soporte para bump mapping, reflection mapping, parallax mapping, screen space ambient occlusion (SSAO), sombreado dinámico utilizando mapas de sombras y distintos efectos de post-processing en pantalla completa.

Unity soporta el uso de shaders via GLSL o Cg, además de contar con el lenguaje ShaderLab, el cual soporta programación básica de efectos. Unity puede detectar la mejor variación para la tarjeta de video en uso y, si nada es compatible, puede hacer uso de shaders más básicos para incrementar su rendimiento. El intérprete de scripts de Unity está escrito en Mono, la implementación open-source del Framework .NET. Los programadores también pueden hacer uso de UnityScript, C# o Boo.

Unity3D está disponible en dos versiones Unity Personal y Unity Professional. Unity Personal es la versión libre de costo que incluye una marca de agua y tiene funciones reducidas, mientras que Unity Professional es la versión completa con un costo de a partir de 75 dólares al mes y debe ser adquirido por al menos 12 meses, esta versión incluye funciones adicionales como occlusion culling e iluminación global, además de todas las nuevas funciones que surjan mientras estés suscrito.

OGRE

Object-Oriented Graphics Rendering Engine es un motor de renderizado orientado al renderizado de escenas en 3D, bastante flexible y escrito en C++, fue diseñado para hacer más fácil e intuitivo el desarrollo de aplicaciones usando gráficos 3D acelerados por hardware. La biblioteca abstrae todos los detalles de usar APIs como Direct3D y OpenGL. OGRE provee una interfaz basada en objetos y otras clases intuitivas. OGRE ha sido

utilizado en la creación de varios juegos comerciales y recientemente ha recibido soporte multiplataforma.

A pesar de que OGRE puede ser usada para hacer videojuegos, está diseñado para proveer una solución gráfica de gran calidad; para otras funciones como el sonido, uso de red, inteligencia artificial, colisiones, física, etc., se debe de integrar con otras bibliotecas.[29]

XNA

Microsoft XNA Game Studio permite a sus usuarios la creación de videojuegos mediante el lenguaje de programación C#. XNA es un conjunto de herramientas con un sistema en tiempo de ejecución proporcionado por Microsoft que facilita el desarrollo y gestión de videojuegos. XNA está basado en el framework .NET, con versiones para Windows, Windows Phone y Xbox (anteriormente también para Zune). En muchos aspectos XNA se puede considerar como un análogo al sistema de desarrollo de videojuegos de Microsoft mejor conocido, DirectX, pero está dirigido a desarrolladores interesados en escribir juegos ligeros capaces de ejecutarse en una variedad de plataformas de Microsoft. XNA es distribuido de manera gratuita, siempre y cuando no se utilice para crear videojuegos que se conecten a Xbox Live con la ausencia de un contrato previo con Microsoft.

XNA es la base de los videojuegos indie de Xbox. De acuerdo con un email mandado por Microsoft el 31 de enero de 2013, XNA ya no está siendo activamente desarrollado, y no tiene soporte para las nuevas capas de Windows 8 “Metro”, ni para la plataforma Windows RT. A pesar de esto XNA sigue siendo utilizado por algunos desarrolladores de videojuegos. Una versión multiplataforma de XNA 4 llamada MonoGame está siendo actualmente desarrollada, por el momento soporta las plataformas de Windows, OS X, Linux, iOS, Android, Play Station Mobile y OUYA.[30]

GIMP

El Programa de Manipulación de Imágenes de GNU (GNU Image Manipulation Program) es un programa de edición de imágenes de gráficos rasterizados, libre y gratuito. GIMP es la alternativa a Photoshop preferida por muchos debido a que es gratuita, a diferencia de Photoshop que tiene un precio muy elevado. GIMP se encuentra disponible en las plataformas GNU/Linux, Microsoft Windows, Mac OS X, Solaris, FreeBSD, OpenBSD y AmigaOS 4.
GIMP ha ido evolucionando a lo largo del tiempo, soportando nuevos formatos y con herramientas cada vez más potentes, GIMP además funciona con extensiones o plugins y scripts.

GIMP cuenta con aproximadamente 150 efectos estándares y filtros. GIMP permite el tratado de imágenes en capas, para poder modificar cada objeto de la imagen en forma totalmente independiente a las demás capas en la imagen, también pueden subirse o

bajarse de nivel las capas para facilitar el trabajo en la imagen, la imagen final puede guardarse en el formato xcf de GIMP que soporta capas, o en un formato plano sin capas, que puede ser png, bmp, gif, jpeg,tiff, además de otros.

Con GIMP es posible realizar un procesamiento de manera no interactiva, como procesamiento por lotes que cambien el color o convierta las imágenes.

GIMP lee y escribe la mayoría de los formatos de ficheros gráficos, entre ellos JPG, GIF, PNG, PCX, TIFF, y la mayoría de los PSD (de Photoshop), además de poseer su propio formato abierto de almacenamiento de ficheros, el XCF. Es capaz también de importar ficheros en PDF y postscript (PS). También puede importar imágenes vectoriales en formato SVG creadas, por ejemplo, con Inkscape.[31]

Blender

Blender es una herramienta de software multiplataforma, gratuita y de código abierto, dedicada especialmente al modelado, animación y creación de gráficos 3D.
Blender está disponible para Windows, Mac OS X, GNU/Linux, Solaris, FreeBSD e IRIX, en versiones de 32 y 64 bits, teniendo la característica de tener un tamaño realmente pequeño comparado con otros paquetes de 3D. Tiene una peculiar interfaz gráfica de usuario (GUI), que es criticada como poco intuitiva, pues no se basa en el sistema clásico de ventanas; pero tiene a su vez ventajas importantes sobre éstas, como la configuración personalizada de la distribución de los menús y vistas de cámara.

Blender posee la capacidad para crear una gran variedad de primitivas, incluyendo curvas, mallas poligonales, vacíos, NURBS y otras. Junto a las herramientas de animación se incluyen herramientas de cinemática inversa, deformaciones por armadura o cuadrícula, vértices de carga y partículas estáticas y dinámicas. Simulaciones dinámicas para cuerpos suaves, partículas y fluidos. Sistema de partículas estáticas para simular cabellos y pelajes, y shaders para lograr texturas realistas. Blender incluye herramientas para edición de audio y sincronización de video.

Blender tiene la capacidad de utilizar scripts en Python para la creación de herramientas personalizadas, importación y exportación de distintos formatos, automatización de tareas, etc. Blender es utilizado ampliamente en la industria para distintos proyectos y películas, destacando los modelos de la NASA que se pueden encontrar públicamente en su página de recursos, muchos de ellos en su formato nativo .blend.[32]

Git

En comparación con otros proyectos de desarrollo de software, para la creación de un videojuego se requieren de varias partes, aún para el más pequeño, se tendrán miles de líneas de código para el sonido, arte, distribución espacial de los objetos, scripts y más.

Para administrar todos los recursos y sus cambios a través de un esfuerzo conjunto o individual, se debe contar con un control de versiones. Las 3 características principales de Git son:

- Atomicidad: La atomicidad es la propiedad de una operación para ocurrir en un sólo instante desde que se invoca hasta que se genera una respuesta. La atomicidad evita que un proceso se quede incompleto
- Rendimiento: Aún manejando millones de archivos, realizar una operación en Git toma unos segundos. Otros sistemas de control de versiones guardan las diferencias entre cada una de las versiones de un archivo, pero Git almacena versiones completas de los archivos junto con sus deltas.
- Seguridad: Todo lo que Git emplea está protegido por un check-sum que utiliza la función de dispersión SHA-1 antes de ser guardado. Es imposible cambiar los contenidos de un archivo o directorio sin que Git lo sepa, ya que el checksum guardado es utilizado después para compararlo.

Git es gratuito y libre, debido a su modelo de fusiones (merging) único, es empleado por muchas compañías y proyectos como Google, Facebook, Microsoft, Twitter, Netflix y Linkedin entre otras.[33]

Justificación de las herramientas elegidas para la tesis

La adaptación de código de una plataforma a otra, representa un gasto innecesario de recursos en el proceso de desarrollo de la aplicación, además de ser un gasto de tiempo importante. Se pierde la oportunidad de tener una rápida introducción al mercado.

Debido a lo anterior es necesario contar con un esquema que permita la creación de aplicaciones multiplataforma sin que esto represente un gran esfuerzo para el equipo de desarrollo, y que minimice los tiempos de adaptación entre plataformas.

Las funciones para el renderizado de imágenes son más rápidas en las API específicas de cada sistema, y éstas no pueden generalizarse sin perder velocidad. Este es un punto interesante el cual se debe tener presente, se debe tratar de optimizar métodos de renderizado e interacción con el usuario para más de una plataforma.

Uno de los elementos comunes entre todas las plataformas es OpenGL®. Con pocos o nulos cambios en su implementación, resulta la herramienta ideal para el trabajo multiplataforma de gráficos por computadora.

El esquema de diseño y desarrollo que se propone aquí está basado en Frameworks comunes para todas las plataformas, los cuales son, OpenGL y SDL,

además del uso del lenguaje de programación C++. Entre los diferentes sistemas operativos y dispositivos que soportan el uso de dichos frameworks se encuentra GNU/Linux, Mac OSX, Microsoft Windows, Apple iOS y Google Android.

SDL es una librería de código abierto que se encarga de manejar todo lo referente al sistema operativo. Su fuerte está en el apoyo al desarrollo de aplicaciones multiplataformas. Es una herramienta invaluable que ha sido aplicada en numerosas ocasiones y su uso, puede llegar a ser vital en el mercado multiplataforma.

Se eligió usar SDL ya que ahorra trabajo simplificando la comunicación con los dispositivos de entrada específicos de cada sistema operativo. Además de su capacidad para crear ventanas vía WGL, GLX, AGL y Cocoa en cada sistema operativo, tiene controles para audio via DirectX, QuickTime, Core Audio y ya maneja los eventos en el teclado en Mac OS X.

Se tiene además la capacidad de crear una aplicación con más de un hilo, SDL provee la capacidad de crear distintos hilos para poder usar dentro del programa, aunque esta capabilidad será limitada por el hardware específico que corra la aplicación.

Cuando se utiliza SDL y se desarrolla una aplicación multiplataforma, se consigue:

- Evitar gastos innecesarios de adaptación al final del proceso de desarrollo
- Una introducción masiva al mercado, puntual y coordinada
- Un mayor enfoque en la calidad de la aplicación
- Evitar barreras relacionadas con la diferencia de plataformas

Con las facilidades que SDL brinda, facilita el poder enfocarse en los problemas importantes del desarrollo de una aplicación interactiva: su diseño e implementación.

Además se decidió usar la biblioteca GLTools para facilitar la creación de primitivas básicas. Para cargar los modelos 3D dentro de OpenGL se decidió usar la biblioteca ASSIMP debido a su portabilidad, facilidad de uso y personalización, además de estar disponible de forma gratuita y libre.

Para el desarrollo de esta tesis se empleó el sistema distribuido de control de versiones Git, esta herramienta de control de versiones y manejador de código fuente, es gratuita y se encuentra disponible en las plataformas Windows, POSIX, Linux y OS X.

Permite trabajar en el código de manera simultánea en distintas computadoras a través de ramas o versiones del mismo código locales que se pueden unir a la versión maestra del código de una manera fácil y verificando que no haya errores al momento de la unión. Además de brindar la posibilidad de compartir el código fuente a través de su repositorio.

Debido a que las herramientas comerciales para edición de gráficos y modelos 3D como Adobe Photoshop y 3D Studio Max tienen un elevado costo, se decidió utilizar las versiones alternativas que se encuentran disponibles de manera gratuita: Blender para el modelado 3D, y GIMP para las texturas y la edición de imágenes, ambas capaces de generar todo el contenido en formatos universales, listos para su exportación a distintos motores gráficos.

También se utilizó la herramienta Google Drive para la creación y redacción del documento escrito, de una manera activa por parte de ambos integrantes de la tesis, así como para la revisión por parte del director de tesis. De esta manera es más fácil revisar y detectar errores cometidos por algún integrante.

Para la depuración de la aplicación gráfica, además de los IDEs proporcionados por cada sistema operativo (Visual Studio, Code Blocks, XCode), se hizo uso de CodeXL de AMD, el cual puede adherirse a un proceso corriendo con OpenGL y revisar su estado actual, se pueden visualizar los distintos espacios en memoria que se tienen reservados así como las texturas que OpenGL tiene cargadas.

Pensado en que la aplicación pueda ser usada en el mayor número de computadoras, aunque no posean una tarjeta de video moderna, se decidió utilizar la versión 3.0 de OpenGL, pues esta es soportada a partir de la segunda generación de procesadores Intel Core inclusive sin necesidad de una tarjeta de video adicional, o por tarjetas de video bastante antiguas a partir de las series Radeon HD 2xxx de AMD y las series GeForce 8xxx de NVIDIA. OpenGL 3.0 presentó grandes cambios a su versión anterior (OpenGL 2.1), ya que introdujo un mecanismo para despreciar funcionalidad obsoleta y así poder simplificar la API en revisiones futuras, algunas funciones importantes marcadas como depreciadas son: el renderizado en modo directo usando glBegin y glEnd, listas de desplegado, GLSL 1.10 y 1.20.[34] OpenGL 3.0 da soporte a muchas de las características más modernas de OpenGL, siendo una de ellas el soporte para OpenGL Shading Language 1.30, importante en el proyecto pues se desea emplear shaders para obtener efectos modernos haciendo uso del GPU.

Objetivo

El objetivo general de este trabajo es el diseño y creación de una aplicación gráfica, interactiva que opere sobre un motor multiplataforma, sin cambios en el código fuente, bajo los siguientes sistemas operativos: Mac OS X®, Linux y Microsoft Windows®.

Este trabajo será de código abierto, podrá ayudar a facilitar el aprendizaje de algunos temas en el área de la computación gráfica, así como fomentar a otros alumnos a elaborar proyectos similares tomando este como base o apoyo, todo esto haciendo uso de herramientas gratuitas disponibles para cualquier persona. Las librerías generadas podrán ser extraídas por otras personas y usadas en sus proyectos personales, por medio del repositorio Git.

Se hará un juego de computadora estilo “Runner”, donde el jugador viajará a través de un escenario esquivando obstáculos al ritmo de la música y obteniendo puntos hasta llegar al final y pasar al siguiente nivel.

Alcance

La aplicación será desarrollada con una arquitectura orientada a múltiples plataformas y será computacionalmente intensiva: Usando OpenGL 3.0 y GLSL 1.30, renderizará video constantemente, mientras realiza distintas operaciones con punto flotante, y contendrá características elementales de las aplicaciones comerciales como: responderá a eventos del sistema operativo, tendrá un cargador de modelos, sistema de audio, hará uso de shaders. Sin embargo debido a su complejidad y a que el desarrollo de estas funciones de una manera completa llevaría mucho tiempo tanto en su diseño como en su desarrollo, no incluirá algunas características que otros productos comerciales ofrecen: contendrá inteligencia artificial y un sistema de colisiones muy básicos, no tendrá la integración con scripts en otros lenguajes, como es el caso de Unity3D con Python.

Se desean alcanzar los siguientes puntos importantes:

- Definición de un método eficiente para la creación de aplicaciones interactivas multiplataforma, imposición de estándares de programación guiados a la evasión de problemas durante el desarrollo multiplataforma.
- Diseño e implementación una aplicación gráfica interactiva que, pueda ejecutarse en distintas plataformas sin diferencias notables en diseño, interfaz y uso. La aplicación será compilada individualmente para cada plataforma pero, ningún cambio será hecho en el código fuente.
- Reducción significativa de los tiempos de desarrollo de una aplicación multiplataforma, riesgos de compatibilidad y tiempo de introducción al mercado.

La aplicación final tendrá una pantalla de inicio y un nivel finalizado, el usuario podrá presionar cualquier tecla para iniciar el juego y dentro de él podrá moverse con las teclas para esquivar los objetos que se le presenten.

Resultados Esperados

Se espera obtener una plataforma clara, fácil de usar y con la complejidad suficiente para el desarrollo de aplicaciones interactivas en una variedad de dispositivos. Sin cambios en el código fuente, se creará una aplicación gráfica, para comprobar el correcto funcionamiento de la herramienta a través de las distintas plataformas. Esta aplicación será compleja y tendrá un diseño competitivo con productos del mercado actual: tendrá un buen desempeño para aplicaciones en 3D, gran facilidad de uso para quienes manejen el lenguaje C++, y tendrá la capacidad de generar efectos modernos para utilizar en distintos tipos de juegos. Se espera así crear una posible alternativa a librerías como glut para el manejo de ventanas y contextos en OpenGL. Además se espera dejar el código fuente disponible en el repositorio Git de una manera accesible y con un pequeño tutorial con los requerimientos mínimos y como descargar y empezar a usar el código. De esta manera cumpliendo el objetivo de que el proyecto pueda ser utilizado por alumnos de la facultad como apoyo en el aprendizaje de algunos temas importantes en el área de la computación gráfica. Ya sea como base en proyectos similares o simplemente como un ejemplo de cómo se puede hacer una aplicación compleja sin necesidad de emplear librerías específicas de una sola plataforma.

3. Metodología

El objetivo de la tesis puede dividirse en dos: la creación de una plataforma para el desarrollo de aplicaciones gráficas, y la creación de dicha aplicación. Para resolver ambos problemas se debe recurrir a distintos paradigmas de programación, la principal metodología de desarrollo de software empleada será el Prototipado Evolutivo, haciendo uso de la Programación Orientada a Objetos, la cual es útil debido a las características de abstracción, encapsulamiento, modularidad, polimorfismo, y herencia.

Se debe comenzar con la planeación del juego o aplicación gráfica, se creará un documento de diseño del juego con los siguientes elementos:

1. Planeación del objetivo principal del juego, interacciones requeridas y requisitos mínimos que debe proveer el motor gráfico para proveer la experiencia.
2. Definición de los elementos artísticos: visuales y de sonido, que tendrá el juego. Así como efectos especiales y los requisitos que el motor gráfico debe cumplir para poder proveer esto.

Se creará también una arquitectura general de software con al menos 3 componentes:

1. *Aplicación*, este componente se encargará de encapsular las llamadas a métodos del sistema operativo, será el módulo encargado de inicializar y terminar la aplicación, es quién logrará la conexión multiplataforma con la aplicación principal.
2. *Lógica*, dentro de este módulo se definirán los comportamientos, eventos a buscar dentro de la interactividad, la lógica de la aplicación, manejo de entidades y de escena. Aquí se programaran algoritmos de detección de colisión, inteligencia artificial, y se tratará de que todos ellos mantengan una velocidad adecuada para una gran gama de computadoras.
3. *Vista*, este módulo se encargará exclusivamente de mostrar los gráficos por computadora, aquí se tendrán algoritmos de renderizado tradicional y modernos, utilizando *shaders* para conseguir efectos interesantes y portátiles.

Prototipado

Con este método se pueden observar resultados a corto plazo y hacer modificaciones de ser necesarias. El modelo de prototipado elegido fue el modelo de prototipado evolutivo, es decir, a partir de un prototipo inicial se desarrolla el sistema final.

El propósito principal de un prototipo es permitir al usuario final del software evaluar las propuestas del desarrollador por medio de pruebas actuales, en lugar de diseños basados en descripciones, como en otros modelos de desarrollo. Los prototipos pueden ayudar también a que el usuario final describa y muestre a los desarrolladores requerimientos que no habían sido contemplados, lo que vuelve interactiva la relación del desarrollador y sus clientes. Este método rompe con el ciclo tradicional de desarrollo en el que primero se desarrolla el programa por completo y después se solucionan las inconsistencias entre la fase de diseño y la de implementación, lo cual provoca mayores costos tanto de tiempo como de esfuerzo.

Fases del proceso de prototipado

El proceso de prototipado involucra los siguientes pasos:

Identificación de los requerimientos básicos

Determinar requerimientos básicos incluyendo la entrada y salida de datos, los detalles como la seguridad pueden ser ignorados.

Desarrollo del prototipo inicial

El prototipo inicial es desarrollado incluyendo una interfaz de usuario

Revisión

El usuario examina el prototipo y provee comentarios acerca de posibles cambios o adiciones

Mejora del prototipo

Usando los comentarios del usuario el prototipo puede ser mejorado. Se puede negociar el alcance del proyecto en el contrato de ser necesario. Si se hacen cambios importantes se debe desarrollar un nuevo prototipo.

Al prototipo de interfaz de usuario se le conoce comúnmente como prototipo horizontal, provee una vista general de un sistema o subsistema completo, enfocándose en la interacción con el usuario, más que en funcionalidades de bajo nivel, tales como acceso a bases de datos.

Un prototipo vertical es una versión más completa de un subsistema o función, es útil para obtener requerimientos específicos de alguna función.

El prototipado de software tiene muchas variantes, sin embargo, todos los métodos están de alguna manera basados en dos grandes tipos de prototipado: Prototipado Rápido o de eliminación, y Prototipado Evolutivo.

El prototipado por eliminación consiste en la creación de un modelo que eventualmente será descartado en lugar de formar parte del software final. El prototipado evolutivo en cambio es muy diferente, su objetivo principal es construir un prototipo robusto en una manera estructurada y refinarlo constantemente.

Para este proyecto se decidió usar un modelo de prototipado evolutivo, la razón fue que de esta manera construimos un programa muy sencillo que forma el núcleo de nuestro sistema y sobre de él se van agregando nuevas características y mejoras de una manera continúa.

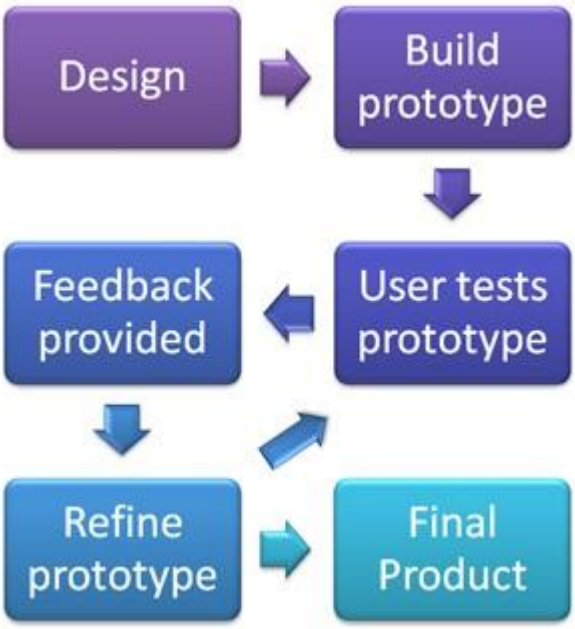


Figura 3.1 Prototipado evolutivo.[35]

Una gran ventaja que brinda este método para la realización de la tesis es que permite agregar características o hacer cambios que no se tomaron en cuenta durante la fase de requerimientos y de diseño.

Los prototipos evolutivos tienen la ventaja sobre los prototipos rápidos de ser sistemas funcionales. El diseño de un sistema funcional consiste en organizar el sistema en módulos o elementos que puedan elaborarse por separado, aprovechando las ventajas del desarrollo en equipo, en este caso por medio del repositorio GIT. De esta manera se puede dividir el trabajo en partes del sistema que cada uno entienda mejor o tenga mayor facilidad en desarrollar.

El prototipado evolutivo proporciona varias ventajas como la reducción de tiempos y costos, pues se detectan posibles cambios en fases tempranas del desarrollo del sistema, además se tiene un mayor envolvimiento con el usuario mejorando así la calidad del software y la satisfacción final del usuario. Por otro lado si no se tiene cuidado con el prototipado, se pueden omitir mejores soluciones por enfocarse en la solución que se tiene en el prototipo actual. Debe tenerse cuidado de no gastar mucho tiempo en soluciones muy complejas que sean temporales, pues debe tenerse en cuenta que no llegarán al producto final.[36]

Ya que se decidió usar el prototipado evolutivo, sólo se deben de establecer las fases generales del desarrollo de software, para llevar una mejor organización en el proyecto:

Análisis de requisitos

En la fase de análisis de requisitos, se analizan las necesidades que tiene el usuario final del software para determinar qué objetivos debe cubrir, en este caso se analizaron los requerimientos de la aplicación, sus características principales y se delimitó el alcance del proyecto.

Diseño del Sistema

Como se mencionó anteriormente, se diseñará un sistema funcional por las ventajas que brinda al desarrollo en equipo. En esta fase se hace el diseño arquitectónico que tiene como objetivo definir la estructura de la solución identificando grandes módulos (conjuntos de funciones que van a estar asociadas) y sus relaciones, definiendo así la arquitectura del sistema. Además se hace un diseño detallado que define la organización del código para comenzar la implementación.

Diseño del Programa

Es importante no confundir la fase de diseño del programa con la de diseño del sistema, en la fase de diseño del programa se realizan los algoritmos necesarios para el cumplimiento de los requerimientos del usuario así como también los análisis necesarios para saber qué herramientas usar en la etapa de codificación.

Codificación

La fase de codificación es en donde se implementa el código fuente, se puede hacer uso de prototipos así como de pruebas y ensayos para corregir errores. Como se utilizará el lenguaje de programación C++, se puede hacer uso de varias clases para una mejor organización, control y reusabilidad. También se puede hacer uso de bibliotecas y otros componentes reutilizables dentro del mismo proyecto para hacer que la programación sea un proceso mucho más rápido.

Pruebas

La fase de pruebas consiste en comprobar que los elementos del sistema funcionen correctamente y que cumplan con los requisitos, antes de entregar el software a los usuarios.

Como se eligió el prototipado y se tiene una relación activa con el usuario es decir, nosotros, las fases de implantación y mantenimiento pueden ser omitidas, pues estas fases son en donde se entrega el software y se ve si cumple con los requerimientos y expectativas del usuario.

Modelo Vista Controlador

Para visualizar los datos, la interfaz de usuario, los eventos y comunicaciones entre ellos, se utilizó del modelo vista controlador MVC (Model View Controller). 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. Las ideas principales detrás del modelo MVC son la reutilización de código y la separación de conceptos.

El modelo es la representación de la información con la cual el sistema opera, gestiona los accesos a dicha información, tanto consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones o lógica de negocio. Envía a la vista la parte de la información que debe ser mostrada constantemente. Las peticiones de acceso o manipulación de información llegan al modelo a través del controlador.

La vista presenta el modelo en un formato adecuado para interactuar (usualmente la interfaz de usuario) por tanto requiere de dicho modelo la información que debe representar como salida.

El controlador se encarga de la entrada de datos, responde a eventos como acciones del usuario e invoca peticiones al modelo cuando se hace alguna solicitud sobre la información. También puede enviar comandos a su vista asociada si se solicita un cambio en la forma en que se presenta el modelo, por ejemplo, desplazamiento o scroll por un documento o pantalla, por tanto se podría decir que el controlador hace de intermediario entre la vista y el modelo.

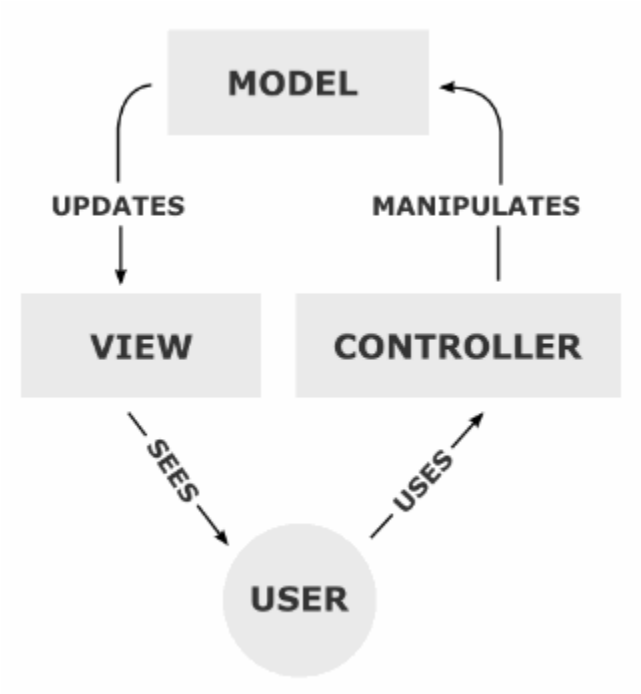


Figura 3.2 Interacción entre los elementos MVC

Además de dividir la aplicación en 3 componentes, MVC también define la interacción entre ellos y el flujo de control que se sigue generalmente es:

1. El usuario interactúa con la interfaz de usuario de alguna forma.
2. El controlador recibe la notificación de la acción solicitada por el usuario y gestiona el evento, frecuentemente a través de un gestor de eventos (handler) o callback.
3. El controlador accede al modelo, actualizándolo, posiblemente modificándolo dependiendo de la acción solicitada por el usuario.
4. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz de usuario donde se reflejan los cambios en el modelo.
5. La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente.[37]

4. Desarrollo

Análisis de requisitos

Para poder visualizar de manera más clara los requerimientos básicos en el desarrollo de la aplicación gráfica multiplataforma se comenzó por el diseño de la aplicación interactiva. Los requisitos de ella definirán los requisitos del motor gráfico multiplataforma a realizar.

Diseño de la aplicación - Documento de Diseño del Juego

El diseño de la aplicación se basó en la idea de crear un juego estilo runner capaz de funcionar en distintos sistemas operativos y en el mayor número de computadoras posibles, utilizando en ella efectos modernos. Los juegos estilo runner se caracterizan en que el personaje principal va recorriendo un nivel de manera horizontal, generalmente recogiendo objetos o esquivando obstáculos para conseguir la mejor puntuación posible.

Para determinar qué clase de aplicación se quería, se buscó inspiración en juegos estilo runner populares y divertidos:

- Pitfall.
- Canabalt.
- Jetpack JoyRide.
- Bit Trip Runner.
- Minion Rush.

Pitfall es uno de los primeros juegos de este estilo pues salió para la consola Atari y es el segundo juego más vendido de Atari después de Pac-Man, el personaje debía recorrer una jungla recogiendo tesoros y evitando distintos obstáculos.

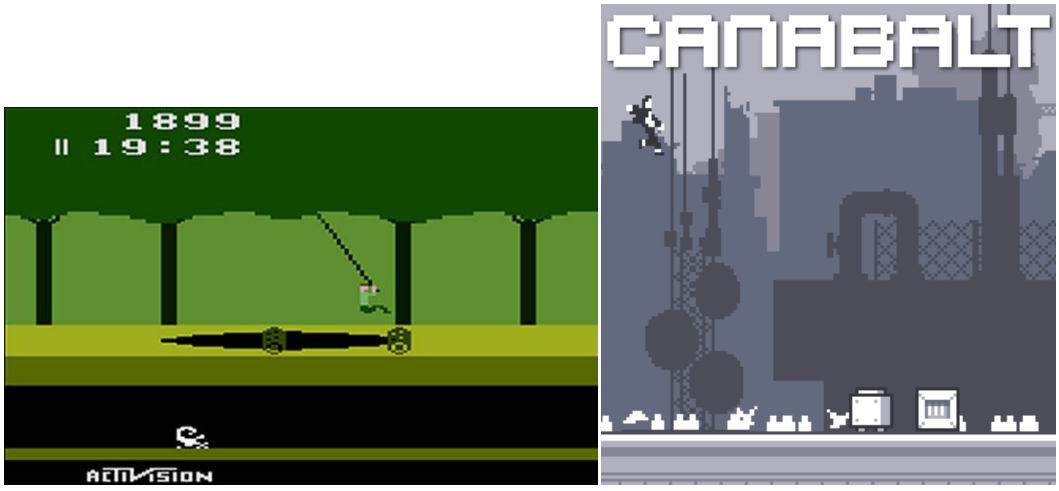


Figura 4.1 Juegos Pitfall (izquierda) y Canabalt (derecha).

El juego Canabalt destaca por tener un modo de juego muy simple en el que el jugador sólo oprime un botón para brincar, sin embargo, el juego posee un mundo muy complejo con distintos obstáculos que lo hacen muy entretenido a pesar de su simplicidad. Popularizó el género en el mercado móvil.
Jetpack JoyRide al igual que Canabalt hace uso de un control simple de un solo botón, sin embargo, incorpora la novedad de una mochila Jetpack en el personaje, al presionar el botón su mochila se enciende y el personaje se eleva, al soltarlo su mochila se apaga y el personaje cae.

Figura 4.2 Bit Trip Runner.

Mientras que Bit Trip Runner destaca por combinar gráficas coloridas y atractivas con música de 8-bits que mezcla géneros contemporáneos como la música techno y la electrónica, la cual va aumentando en velocidad y ritmo de acuerdo con la puntuación del jugador.
Por último Minion Rush es una aplicación runner en 3D, y es una de las más populares en el año 2013.

Para darle un toque creativo a la aplicación, después de analizar las anteriores aplicaciones se decidió crear una aplicación similar con los siguientes cambios:

- Eliminar la posibilidad de que el jugador pueda perder. La aplicación quiere enfocarse en darle una experiencia distinta al jugador, algo que pueda recordar y que no se convierta en una competencia entre sus amigos. El tiempo que los usuarios pasan con una aplicación puede ser muy corto y se desea que sea memorable.
- Sincronizar la pista de obstáculos y recompensas al ritmo de la música. Existen pocas aplicaciones de este género con esta variante, la más popular es Bit Trip Runner.
- Que la aplicación sea en 3D. La mayoría de las aplicaciones de este tipo son en 2D y estar en 3D puede dar una característica diferente.
- Se desea además, realizar un motor gráfico que sea capaz de correr la aplicación en dispositivos móviles en un futuro, con mínimos cambios.

De esta manera, se originó la idea de Super Happy Rocket:

Super Happy Rocket

Super Happy Rocket es un juego estilo runner donde el jugador controla un cohete a través de una ciudad colorida, esquivando obstáculos al ritmo de la música mientras obtiene estrellas para incrementar su puntaje. Una fusión entre el género runner y juegos de ritmo como Guitar Hero.

Definición de los elementos artísticos del juego

Super Happy Rocket es un juego sencillo, por lo que queremos colores simples y llamativos que evoquen la sensación de estar en una ciudad viva de noche. La figura 4.3 muestra imágenes que ayudan a visualizar la idea de mejor manera durante la fase de diseño, antes de iniciar la fase de desarrollo.

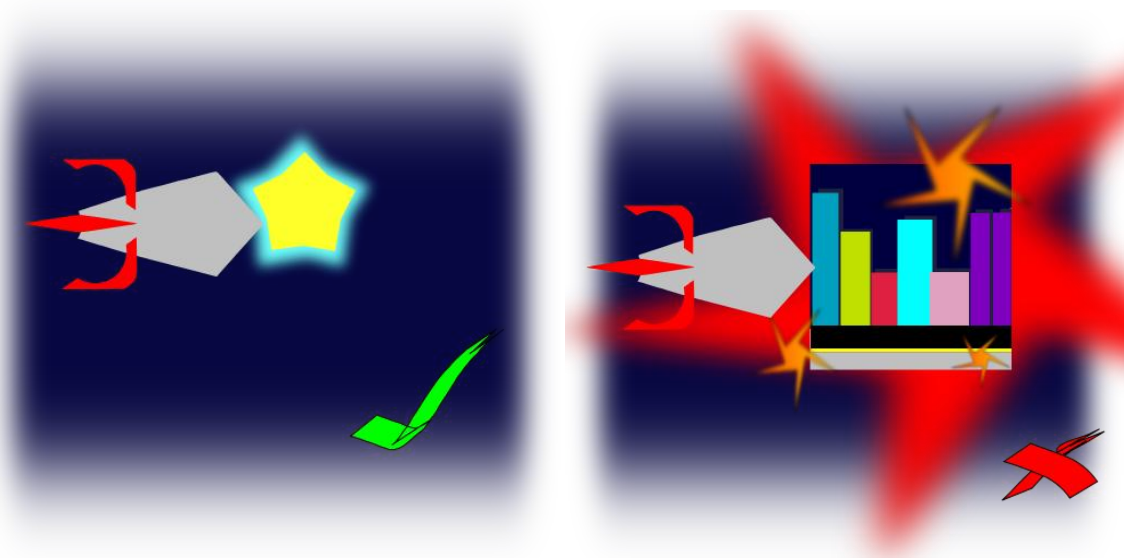


Figura 4.3 Diseño de las mecánicas de la aplicación demostrativa.

Para la música se tiene planeado utilizar música electrónica gratuita y con licencia para usar de manera comercial en cualquier lugar. Para ello se eligieron canciones de Professor Kliq, quien libera la mayoría de sus álbumes con una licencia CC flexible.

Un ejemplo del arte que se usará en el juego es la siguiente imagen utilizada como la ciudad de fondo:

Figura 4.4 Ejemplo del arte utilizado en Super Happy Rocket.

Diseño de la interactividad

El jugador sólo podrá moverse de izquierda a derecha, la cámara y el cohete guiarán todo el viaje a través de la ciudad. Se busca tener controles sencillos, fácilmente transmisibles a cualquier plataforma (de teclado a pantallas táctiles), que sean familiares para los jugadores.

Cada una de las acciones están vinculadas con un punto en el tiempo, se desea que la acción de esquivar u obtener algo vaya de acuerdo a la música.

El escenario del juego será una ciudad de noche, y se quiere que ésta siempre rodee al jugador. Por lo que para el fondo se planea realizar una Skybox, un cubo de 6 lados con una textura en cada lado unida en un cubeMap.

Para la realización de los modelos se utilizará Blender, solución open source, por lo que el motor gráfico a realizar debe soportar la importación de estos modelos a la aplicación con todos sus atributos, vértices, índices, materiales y texturas.

Como la música y su ritmo guiarán a la aplicación se debe de tener soporte de sonido multiplataforma. Soporte de más de 1 canal de audio para reproducir la música y los sonidos especiales. Además el motor debe de tener la capacidad de determinar cuánto tiempo ha pasado para poder determinar en qué momento actuar o disparar eventos. El motor debe de realizar un chequeo básico para asegurarse de reservar un canal de audio exclusivamente para la música del nivel, y otro para los sonidos especiales.

Dados estos requisitos básicos del juego. Se puede ahora determinar los requisitos técnicos del motor gráfico a desarrollar para soportar la aplicación.

Análisis de Requisitos del Motor

Como denominador común de todas las plataformas se tomará OpenGL 3.0 y GLSL 1.30, ya que es el estándar de OpenGL con mayor representación en la actualidad, lo soportan las series de procesadores Intel y AMD recientes, no hay ni siquiera necesidad de tener una tarjeta de video para utilizar a sus funciones más comunes.[38]

El motor debe de checar al momento de inicializar que todos los objetos, efectos y animaciones sean soportados por la computadora, por lo que se realizará una precarga de todos los objetos al momento de iniciar la animación. Aunque esto tomará un poco de tiempo, con esto se estará asegurando que todo esté funcionando correctamente y que tengamos caminos alternos de renderizado en caso de que exista algún efecto no soportado por el hardware.

Por cuestiones de tiempo, se limitará a realizar el renderizador en un solo hilo. Para realizar distintas operaciones en varios hilos usando OpenGL se requiere de un diseño complejo que permita compartir y modificar toda la información del estado de OpenGL de forma óptima y segura, ya que todas las llamadas al API de OpenGL se realizan sobre el estado, o contexto, que tiene un hilo.[39]

Por esta razón, se debe tener mucho cuidado al momento de programar objetos que realicen cambios en el estado de OpenGL. La plataforma debe asegurarse de que en todo momento exista un contexto válido y que no se modifiquen secciones críticas del contexto que se están utilizando. Por lo que se decidió encapsular tanto como fuese posible el renderizado de los objetos.

Las operaciones más costosas en OpenGL son las correspondientes a la consultas y cambios del estado actual, qué extensiones están soportadas, cuál es el modo actual, cuál es la textura siendo utilizada. Para evitar estos retrasos en el renderizado, la clase debe ser capaz de ordenar toda la geometría que existe en una escena, y acomodarla de tal modo que se realicen el menor número de consultas y cambios.

Cómo se realizará un juego de ritmo, se debe tener mucho cuidado en cómo van apareciendo los obstáculos. Realizando tantos cambios como sean necesarios para que el juego esté exactamente al compás de la música. De antemano, se puede notar que existirán muchas modificaciones en los niveles que se generarán, y se quiere realizar un prototipado rápido de éstos. Así que la mejor opción es separar tanto la generación de los niveles del motor principal como sea necesario para que no se tenga que recompilar partes del motor cuando haya que hacer cambios rápidos. De preferencia, que todo se tome a partir de un archivo, y que la capa lógica se encargue de determinar el comportamiento de cada uno de los objetos.

Se desea enviar tanta geometría como se pueda en un solo instante, reutilizando la mayor parte posible. Si un elemento comparte la misma textura que otro, entonces se debe tener cuidado en que no se genere más espacio en memoria del necesario para una misma textura. Se necesita una clase que se encargue de revisar las texturas que utilizan cada uno de los objetos y que pueda realizar los cambios apropiados.

Como el modelado se va a realizar en Blender, es necesario asegurarse que el motor entienda los archivos generados por el software, se quiere traducir los modelos de

Blender a un formato entendible por el motor, para que sea renderizado de forma óptima, reutilizando texturas si un modelo utiliza la misma en más de un lugar, cambiando texturas de manera dinámica si un objeto utiliza más de una textura. Que no introduzca vértices de más, a veces los archivos exportados tienen información redundante, así que se quiere que sólo se carguen los vértices que sean necesarios.

Otros lineamientos que se deben tener en cuenta para la programación del motor gráfico:

- Prohibido el uso del modo inmediato de OpenGL, se quiere que el motor gráfico eventualmente soporte dispositivos móviles, en ellos OpenGL ES ha desechado por completo los siguientes elementos del API:
  - glBegin/glEnd
  - Listas de desplegado.
  - GLSL 1.10/1.20
- Proveer métodos sencillos para la depuración de la aplicación, programar con OpenGL no regresa los errores usuales cuando se programa para la consola, por lo que se deben crear métodos para hallar errores y arreglarlos de forma sencilla.

.

Para proveer un motor sencillo, fácil de entender y mantener, se tomó inspiración de dos motores existentes: XNA y Unity3D.

Ambos encapsulan el motor gráfico, el soporte para sonido, de almacenamiento, las operaciones matemáticas y la obtención de los sistemas de entrada y salida de una computadora. Además, proveen clases y objetos pre-hechos que se asocian automáticamente a su pipeline, sin importar la plataforma, el usuario puede empezar realizando contenido e importarlo rápidamente desde una interfaz gráfica, como el caso de Unity 3D, o desde una plantilla de código provista por el SDK, como sucede en XNA.

El ámbito de desarrollo que proveen las dos herramientas es muy distinto, uno es una interfaz gráfica desde la cual se crean objetos a los cuales atributos, scripts, comportamientos y distintos efectos se les puede ser aplicados, y la otra es una biblioteca, conjunto de métodos y objetos pre-hechos que pueden ser extendidos por el usuario.

Habiendo en el pasado usado ambos, se derivó gran parte del diseño de XNA, mientras que se buscaron algunas características únicas de Unity3D.

Bases del Diseño

El framework de XNA es un conjunto de librerías de .NET sobre las que los desarrolladores se apoyan para construir sus juegos. Cuando se describe el framework de XNA y cómo está hecho, se puede pensar en él como una serie de layers o capas:

Figura 4.5 Arquitectura de XNA. Los recuadros verdes corresponden con la funcionalidad que ya viene incluída con el framework, los demás recuadros corresponden a elementos creados por los usuarios.[40]

Plataforma: La plataforma es la capa más baja de XNA y se compone de las APIs sobre las que está construida XNA, algunas de estas APIs son Direct3D, XACT, XINPUT y XCONTENT.

Núcleo del Framework: El núcleo del framework de XNA es la primera capa y proporciona la funcionalidad que otras capas heredan. Aquí se pueden encontrar diversas áreas agrupadas de acuerdo a su funcionalidad: Gráficos, Audio, Entradas, Matemáticas y Almacenaje. Para aquellos que todavía no saben escribir shaders, XNA incorpora una clase básica llamada BasicEffect que puede ser usada para pintar rápidamente sin tener que programar nada extra. XNA incluye los tipos de datos más usados para la programación de juegos como Vectores de dos, tres y cuatro componentes, matrices, planos, rayos, entre otros. También se incluyen boundingBox, boundingSphere y BoundingFrustrum, así como los métodos para hacer test de intersecciones y poder detectar colisiones.

Framework Extendido: La capa del framework extendido tiene como objetivo principal hacer el desarrollo de las aplicaciones algo fácil. Esta capa tiene dos componentes principales: el modelo de la aplicación y el content pipeline.

El modelo de aplicación tiene el propósito de abstraer al usuario de la plataforma donde se está ejecutando el juego y permitir centrarse en escribir el juego. No se tiene que preocupar sobre cómo crear una ventana, crear temporizadores o manejar mensajes. XNA proporciona todo eso. De la misma manera también ofrece el GraphicsComponent que hace increíblemente fácil la creación y manejo del GraphicsDevice (Dispositivo Gráfico), el cual es usado para renderizar.

Aquí también es donde se proporciona el modelo de componentes, que permite incorporar GameComponents escritos por otros desarrolladores de manera inmediata en el juego. De esta manera se pueden construir bibliotecas de componentes reutilizables.

Juegos: La capa de juegos es la capa más alta. Esta capa está formada por el código fuente del juego y su contenido, así como de componentes incorporados en el juego. El lenguaje de programación utilizado en XNA es C#.

Unity3D, por su parte, crea todos sus objetos dentro de un sólo archivo, y es necesario configurar a Unity para trabajar como un equipo, de tal modo que genere distintos archivos para los distintos objetos sobre los cuales un equipo puede estar trabajando. Su versión gratuita, no permite generar niveles como archivos de texto, los crea de forma binaria, lo cual dificulta el trabajo en equipo, ya que una herramienta de control de versiones encontraría diferencias en cada paso de su desarrollo.

Se puede deducir de los scripts que uno puede generar en Unity, que cuenta con estados similares a los de XNA. Con cada objeto pudiendo reconocer en qué momento ha sido creado, en qué momento es dibujado y en qué momento está siendo actualizado.

De este modo, se eligió basar el motor gráfico en una máquina de estados, los 4 estados mayores que se eligieron fueron: la carga inicial, el dibujo inicial, la actualización del dibujo, y la descarga final. Estos 4 elementos son comunes en todos los objetos de XNA, y se hace fácil la distribución de tareas siguiendo este esquema. Por lo que todos los objetos que se desarrollan poseen estos 4 métodos, volviendo su organización sencilla. Cuando la aplicación quiera dibujar algo, todos los objetos llamarán a su método de dibujo, así como la actualización, carga y descarga.

El uso de SDL facilita todo esto, ya que soporta y encapsula la comunicación con el sistema operativo en un gran número de plataformas, muchas más que en el caso de XNA y Unity3D.

Diseño del Sistema

Para la creación de la aplicación, se desea abstraer cada uno de los subsistemas, por lo que se crearán una clase de tipo singleton para cada uno de los subsistemas clave: gráficos, sonido, audio, entradas.

De tal modo que mediante llamadas genéricas, uno pueda agregar elementos para desplegar en la pantalla, y pueda crearse una aplicación gráfica agregando solamente contenido, instanciando algunas clases de objetos y niveles como sucede en Unity o XNA, permitiendo al usuario modificar las clases del mejor modo que encuentren sin tener que preocuparse sobre cómo se va a renderizar un objeto o en qué parte deben de activar o desactivar estados de OpenGL.

Arquitectura de alto nivel de la aplicación

Diseño del Programa y Codificación

Distintos motores gráficos basan su arquitectura en un sistema de renderizado específico, en el caso de XNA, existen varios estados de renderizados, los cuales pueden ser llamados de distintas maneras dependiendo de la eficiencia de una computadora, tratando de mantener siempre los mismos cuadros por segundo.

XNA no se hace ningún esfuerzo especial por optimizar la geometría que está siendo dibujada así que todo depende del programador. Se encarga más de encapsular todos los procesos del sistema operativo y trata de proveer herramientas sencillas para el desarrollo. Es una forma sencilla de comenzar para después mejorar e iterar, por lo que se decidió empezar con algo similar, tratando de obtener resultados primero y optimizando después.

Todos los juegos tienen una serie de operaciones que son llamadas constantemente para actualizar su estado. Aunque el jugador no interactúe con el juego, el juego debe de estar en constante funcionamiento, para la aplicación diseñada se planeó un ciclo sencillo de operación donde cada subsistema puede ser actualizado en cada cuadro renderizado.

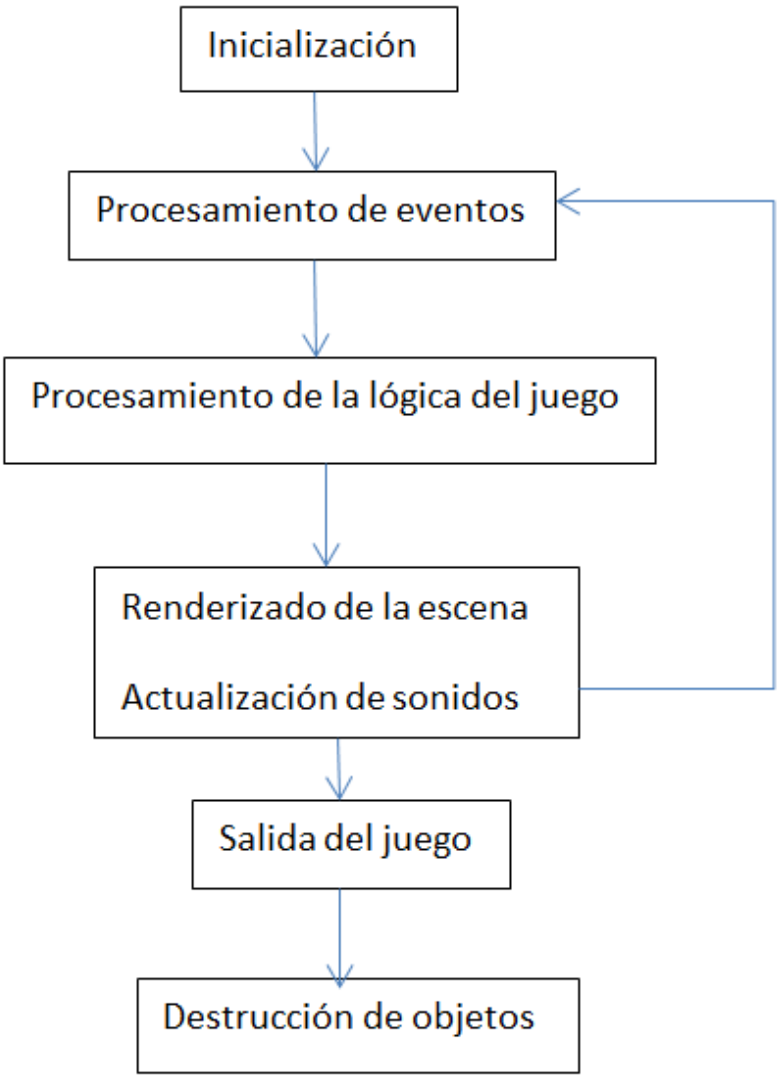


Figura 4.6 Ciclo de operación de la aplicación demostrativa.

Para el desarrollo de este ciclo y su abstracción en un motor gráfico de fácil depuración se tomó y se adaptó el modelo Modelo Vista Controlador (MVC) a la aplicación gráfica diseñada: los modelos son cada uno de los objetos que se tienen, la vista cada uno de los componentes responsables por presentar el estado de la aplicación y el controlador la lógica de la aplicación.

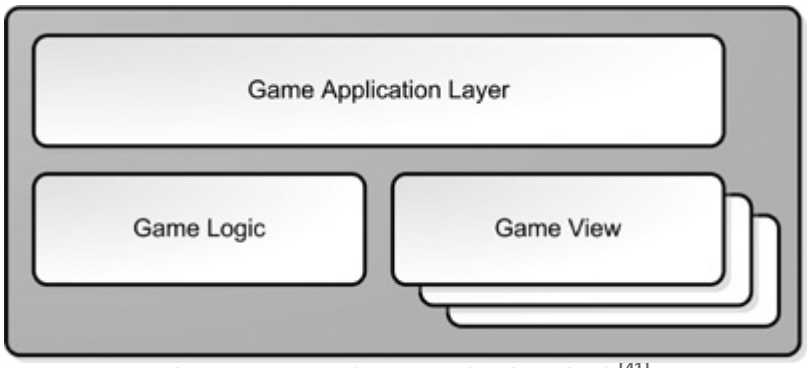


Figura 4.7 Arquitectura de alto nivel.[41]

La capa de aplicación sólo se encarga de preparar el contexto y las herramientas necesarias para correr dentro de los 3 sistemas operativos. Aquí es donde se programó una aplicación en SDL que provee un contexto de OpenGL y pregunta al sistema operativo sobre las extensiones soportadas por la arquitectura del sistema. Se inicializan además los sistemas de audio, métodos de entrada, y de carga de objetos.

Dentro de la capa lógica se encuentran todos los archivos intermediarios que regulan a cada una de las vistas, los controles para cada uno de los objetos en el juego, el orden de renderizado, subsistemas para salvar el estado del juego, control del renderizado de frames y control general de la aplicación.

En la capa de la vista, se programaron todos los objetos presentes en el mundo, las texturas que utilizan, los modelos en 3D de los cuales cargaremos, su posición inicial en el mundo, la música ambiental, el comportamiento que utilizarán y el comportamiento que se le puede dar al jugador.

Una vista detallada de lo anteriormente expuesto se puede observar a continuación:

La capa lógica

La capa lógica define las reglas de interacción dentro de la aplicación, define qué tanto puede interactuar el usuario con su entorno y los objetos que existen en él.

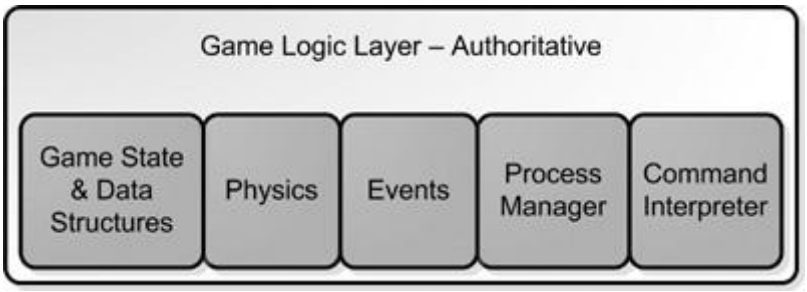


Figura 4.8 Capa lógica.

Dentro de esta capa se programaron cada una de las estructuras de datos, reglas, eventos y manejadores de recursos para la aplicación. La definición de los distintos subsistemas se hizo de la siguiente manera:

InputManager

InputManager
+keysHeld[SDL_NUM_SCANCODES] : bool +ctxt : Context * +event : SDL_Event
+InputManager() +InputManager(in context : Context*) +~InputManager() +isKeyPressed(in key : Uint32) : bool +handleKeys() : Uint32

Figura 4.9 Diagrama de la clase InputManager.

Para el control de la aplicación se desarrolló un simple manejador de eventos, el cual recibe datos de la capa de aplicación y responde de acuerdo a los eventos, esta clase, mantiene un arreglo de todas las teclas posibles que se pueden presionar en UNICODE, este arreglo puede consultarse para determinar si el usuario está presionando una tecla o no.

Scene Manager

SceneManager
+ctxt : Context * +currentLevel : int +levels : vector<BaseLevel*>
+SceneManager(in ctx : Context*) +~SceneManager() +Initialize() : const int +LoadContent(in level : const unsigned int) : const int +Draw() : const int +Update(in gameTime : Uint32) : const int +UnloadContent() : const int

Figura 4.10 Diagrama de la clase SceneManager.

Encargado de inscribir a todos los objetos de un nivel a los subsistemas indicados. SceneManager realiza la carga y descarga de todos los elementos en un nivel, música, sonidos, modelos y texturas de un nivel son analizadas y enviadas a un subsistema correspondiente para poder cargarlas e inicializar el nivel después. Es además el encargado de llamar los métodos que controlan a cada uno de los objetos via otra clase denominada el EntityManager.

En el motor gráfico, una escena o nivel contiene una serie de objetos o actores, los cuales dibujan un resultado final en la pantalla. Lo único que hace el administrador de escenas es manejar en qué momento debe de cargarse cierto nivel dependiendo del estado actual del Contexto.

Para hacer esto se creó una lista de todos los niveles existentes, la cual es definida al momento de compilar el proyecto. Con esta lista, al momento de cargar, el administrador de escena agrega todos los elementos de un nivel al subsistema indicado. Toma las texturas utilizadas en un nivel, por ejemplo, y las agrega al administrador de texturas. Para cada uno de los objetos se tiene un administrador, de tal modo que alguien siempre sepa la cantidad exacta de elementos que existen sobre cierto tipo, y cada uno puede realizar la liberación de su memoria de un modo particular.

Aunque la liberación de memoria es manejada por el destructor de cada objeto particular, es trabajo del administrador revisar la cantidad de referencias que existen, para asegurarse de que no haya pérdidas de memoria en ninguna parte del proceso.

Dependiendo de la escena además, distinta lógica es ejecutada, ya que algunas escenas pueden ser simples menús o pantallas de pausa. Por lo que el administrador de una escena consulta al administrador de entradas y salida, para determinar qué lógica se debe de ejecutar. La lógica ejecutada es una simple llamada a un evento particular, y el resultado final será determinado por esta otra capa de la aplicación.

EntityManager

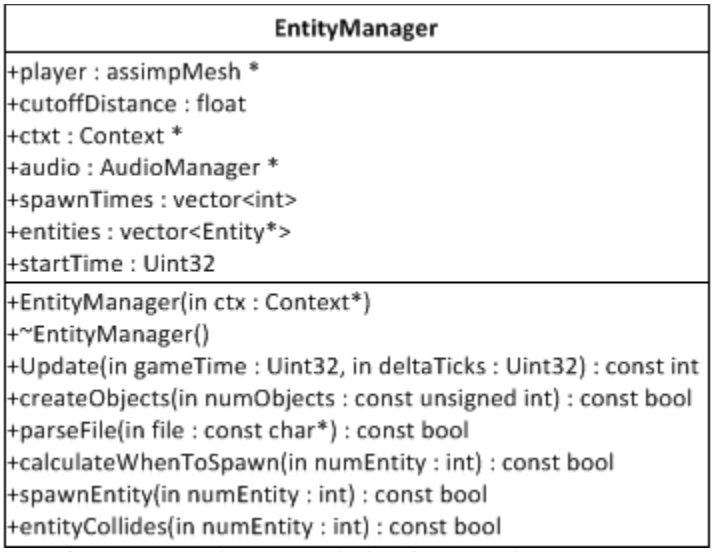


Figura 4.11 Diagrama de la clase EntityManager.

Es la lógica particular de un nivel, cada nivel puede tener un Manejador de Entidades distinto, aquí se lee un archivo de texto con la definición del nivel, los tiempos en los cuales cierta geometría debe aparecer en el escenario y dentro de la clase se determina su comportamiento. Es la lógica de cada uno de los objetos en el nivel y puede ser modificada para acomodar distintos niveles.

Renderer

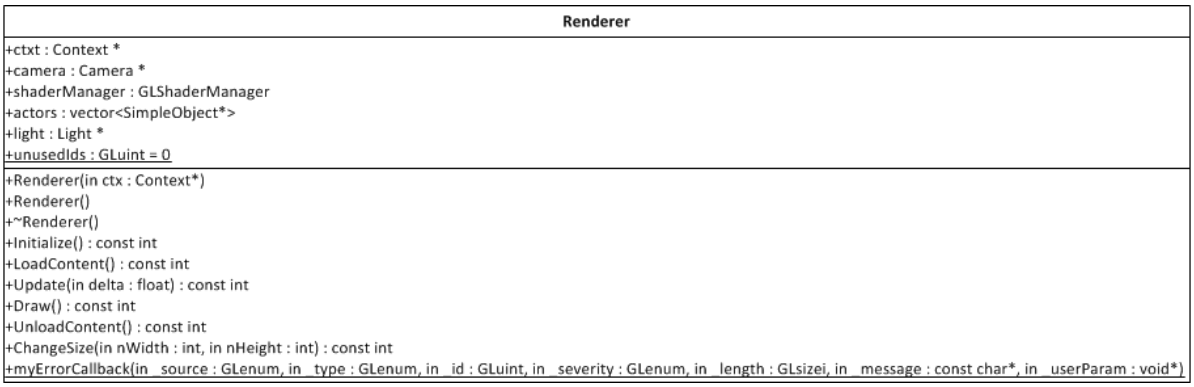


Figura 4.12 Diagrama de la clase Renderer.

Para el renderizado, se creó un simple grafo de escena, el cual recibe una sola clase de objetos para renderizar dependiendo del estado del juego, los cuales organiza dependiendo del efecto gráfico que están utilizando para no cambiar de estados en OpenGL y lograr un renderizado veloz.

La función del final, myErrorCallback, es crítica para el desarrollo de una aplicación gráfica. OpenGL provee un único método de consulta para errores: glGetError(), pero realizar una arquitectura con llamadas constantes a esta función puede ser tedioso,

por lo que algunos vendedores proveen la extensión ARB_debug_output, la cual nos permite declarar una función personal como un callback en caso de que haya algún error.

Esto se logró dentro de la clase Renderer del siguiente modo:

```
if( GLEW_ARB_debug_output ) {
glDebugMessageCallbackARB( &Renderer::myErrorCallback, NULL );
glEnable( GL_DEBUG_OUTPUT_SYNCHRONOUS );
glDebugMessageControlARB(GL_DONT_CARE,
                              GL_DONT_CARE,
                              GL_DONT_CARE,
                              0,
                              &Renderer::unusedIds,
                              true);
}
```

Así mismo se logró la implementación de la función de depurado de una manera sencilla:

```
void APIENTRY Renderer::myErrorCallback( GLenum _source,
      GLenum _type, GLuint _id, GLenum _severity,
      GLsizei _length, const char* _message,
      void* _userParam )
{
      printf( "%s\n", _message );
}
```

La utilidad de esta herramienta no está en su capacidad de imprimir errores, sino en detectarlos, poniendo un breakpoint en la función de callback, se puede obtener la pila del programa y determinar por qué se ha causado el error, además de tener el mensaje como guía.

El renderizador es la pieza más importante de la aplicación interactiva, alrededor de él se planearon el resto de los objetos, y fue diseñado de manera extensible. Al ser inicializado, se encarga de revisar la versión actual de OpenGL soportada por el equipo donde está siendo ejecutada la aplicación. Para utilizar todas las funciones de OpenGL 3.0, se hizo uso de GLEW, una librería encargada de revisar y habilitar las extensiones de OpenGL en varios sistemas operativos.

Windows sólo provee OpenGL 1.1 en Windows XP y OpenGL 1.4 en Windows Vista mediante el controlador opengl32.dll, mientras que en los otros sistemas operativos existen librerías nativas que soportan varias extensiones. Para hacer uso de las nuevas versiones de OpenGL en Windows, se necesita declarar apuntadores hacia las nuevas funciones de OpenGL, de modo que se puedan usar sin problema. La librería GLEW se encarga de realizar este proceso por el usuario en distintos sistemas operativos, ya que incluso en donde si hay soporte para nuevas versiones de OpenGL, algunas extensiones son específicas de ciertas marcas, y la librería se encarga de revisar su existencia en el sistema por nosotros.[42]

El renderizador inicializa una serie de shaders básicos para dibujar geometría sencilla, además de inicializar la cámara y algunos estados iniciales de OpenGL.

El renderizador genera un grafo de escena básico y extensible, el cual puede más tarde ser sustituido por una estructura de datos mucho más compleja, a él se le agregan elementos de un nivel o escena durante la fase de carga, los cuales posteriormente son puestos en orden, dependiendo de qué clase de textura o shader utilizan, de tal modo que en el menor número de cambios de estado todos los objetos sean dibujados. Esto es posible gracias a que el renderizador guarda apuntadores a todos los objetos en su estructura interna, por medio de los cuales puede consultar la textura a utilizar, el shader y las matrices de transformación a aplicar.

Al momento de ser destruido, el renderizador se encarga de borrar todos los shaders y objetos que se hayan quedado dentro del pipeline.

El renderizador entiende todos los objetos declarados por el desarrollador, pero para entender cómo renderizar otra clase de objetos, se hizo uso de la librería de código abierto Assimp.

Existe una sola limitante importante dentro de el renderizador, todos los objetos deben estar formados por triángulos, aunque el código podría extenderse para reconocer otro tipo de patrones, al final los objetos que se encuentran dentro del renderizador se registran dentro del GPU como VBOs(Vertex Buffer Objects), los cuales son espacios de memoria de vídeo reservados para dibujar todo tipo de polígonos.

Las distintas vistas de la aplicación

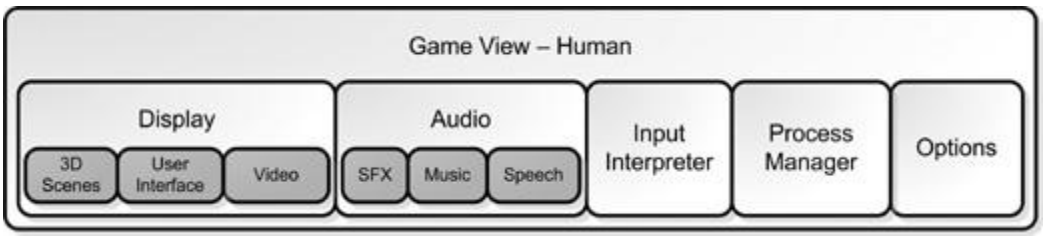


Figura 4.13 Vista de la aplicación desde el punto de vista del usuario.

Las distintas vistas de la aplicación son una colección de sistemas que se comunican con la lógica para presentar el juego de forma particular al usuario. Es aquí donde se finaliza la definición de objetos como el Renderer, donde se realizan las llamadas a OpenGL y se declaran los arreglos a utilizar, se compilan e intercambian los diferentes shaders a utilizar y las llamadas a sonido se hacen para reproducir el sonido dentro de la aplicación.

Todos estos atributos se definen de forma básica en una clase denominada *Level*, en la cual se declaran todos los objetos que contiene un mundo dentro de nuestra aplicación. Para modificar uno de los mundos que existen, sólo basta con modificar una de las instancias de esta clase y, para agregar más, sólo basta con instanciar más objetos de esta clase y declarar todo lo que habrá dentro de ellos, volviendo a esta capa en algo independiente de cómo se maneja la aplicación o los detalles sobre cómo deben de comportarse los objetos.

Las posiciones iniciales, las texturas asignadas, los sonidos, colores, vértices, shaders empleados por los objetos así como su comportamiento, son parte de otra vista.

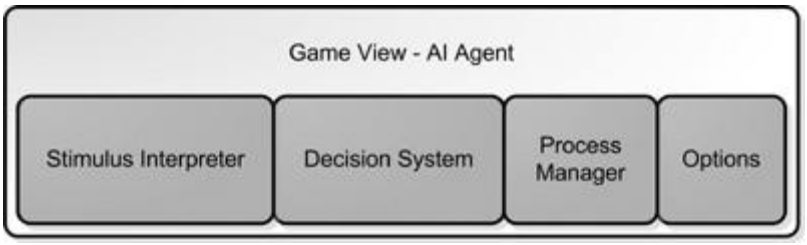


Figura 4.14 Agente de Inteligencia Artificial.

Los métodos de comunicación que existen entre los objetos del mundo y el estado del juego también crean la vista para el usuario, aunque su organización es distinta e independiente. Aquí se definen los comportamientos que tendrán los objetos en el mundo, otorgando atributos a cada una de las clases de objetos que hay, se puede decidir cómo se comportarán los objetos a través de la capa lógica.

Haciendo uso de la herencia y el polimorfismo, se crearon una serie de clases que forman la base para cada clase de objeto en el mundo. Se tomó como inspiración el proceso de renderizado para la definición de las clases que conforman los bloques básicos de la aplicación. Se reconocen las siguientes clases de objetos:

Simple Object

SimpleObject
+tBatch : bool +batch : GLBatch +triBatch : GLTriangleBatch +frame : GLFrame +modelViewMatrix : GLMatrixStack +vertex : vector<float> +index : vector<float> +normal : vector<float> +texture : vector<float> +color : vector<float> +textureFile : GLuint +tFile : string +shaderFile : GLuint +vShader : string +fShader : string +mScaleMatrix : M3DMatrix44f +scaleVector : M3DVector3f
+SimpleObject() +SimpleObject(in obj : const SimpleObject &) +~SimpleObject() +Initialize(in ctx : Context*) : const int +LoadContent() : const int +Draw() : const int +Draw(in camera : Camera*, in shaderManager : GLShaderManager*, in textureManager : TextureManager*, in light : Light*) : const int +Update() : const int +UnloadContent() : const int

Figura 4.15 Diagrama de la clase SimpleObject.

SimpleObject es la clase base para cada uno de los objetos renderizables en la escena, es la única clase que el renderizador reconoce, dependiendo del efecto que se desee al dibujar un objeto, cada hijo de esta clase debe de reimplementar los métodos de dibujado y de actualización. Un SimpleObject sólo puede ser usado dentro del contexto del Renderer, por lo que implementar aquí las reglas de su renderizado siempre resultará en un resultado correcto, el objeto siempre tendrá un contexto válido de OpenGL y uno está seguro que el orden de renderizado se preservará.

Assimp Mesh

assimpMesh
+ctxt : Context * +fileName : string +objects : vector<SimpleObject*> +verts : vector<vector<GLfloat>> +norms : vector<vector<GLfloat>> +texts : vector<vector<GLfloat>> +indexes : vector<vector<GLushort>> +maxIndex : vector<GLushort> +textures : vector<GLuint> +textureStruct : vector<Texture2D*> +numTextures : unsigned int +untexturedVerts : bool
+assimpMesh(in filename : const char*, in ctx : Context*) +assimpMesh(in mesh : const assimpMesh &) +~assimpMesh() +Initialize(in ctx : Context*) : const int +LoadContent() : const int +Draw(in camera : Camera*, in shaderManager : GLShaderManager*, in textureManager : TextureManager*, in light : Light*) : const int +Update() : const int +UnloadContent() : const int +recursiveProcess(in node : aiNode*, in scene : const aiScene*) +processMesh(in mesh : aiMesh*, in scene : const aiScene*) +createBatches()

Figura 4.16 Diagrama de la clase assimpMesh.

Para la carga de objetos realizados en herramientas como 3DS o Blender se implementó un cargador con la ayuda de la librería assimp, el cual mantiene un listado de todos los objetos que se han pedido dependiendo del estado del juego, para poder renderizarlos dentro del grafo de escena. Assimp también implementa un grafo de escena sencillo para modelos que contengan una jerarquía complicada.

Para transformar los nodos del grafo proporcionado por Assimp, se tomaron los datos y se reinterpretaron como un SimpleObject. Assimp tiene algunos problemas importando geometría, por lo que requiere de una fase de validación. Con frecuencia genera más geometría de la esperada, por lo que se tuvo que analizar cada uno de los elementos en el grafo para luego recrearlos como objetos que el renderizador pueda entender.

La capa de la aplicación

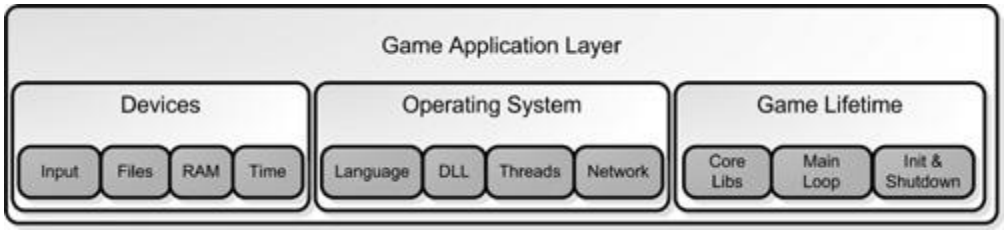


Figura 4.17 Capa de aplicación.

Para la inicialización de una aplicación gráfica, la capa de aplicación tiene los siguientes requisitos:

- Revisar los recursos del sistema: espacio en el disco duro, memoria, dispositivos de entrada y salida.
- Revisar la velocidad del CPU.
- Crear una ventana.
- Inicializar el sistema de audio.
- Cargar las opciones de un jugador y los archivos salvados.
- Crear una superficie para dibujar.
- Inicializar los varios subsistemas, Inteligencia Artificial, Física, etcétera.
- Carga de la escena principal.

Realizar esta tarea para cada una de las plataformas es una opción, uno puede explicarle al compilador vía macros qué código ejecutar en qué plataforma y comenzar así con el encapsulamiento. Sin embargo, esto es una tarea compleja, consume tiempo y requiere de muchos cambios si se quiere algo distinto de nuestra ventana. Cambiar tamaño, colores soportados, o si se desea que la aplicación se despliegue como una sola ventana o en pantalla completa. La elección de SDL facilita todo esto.

SDLApp

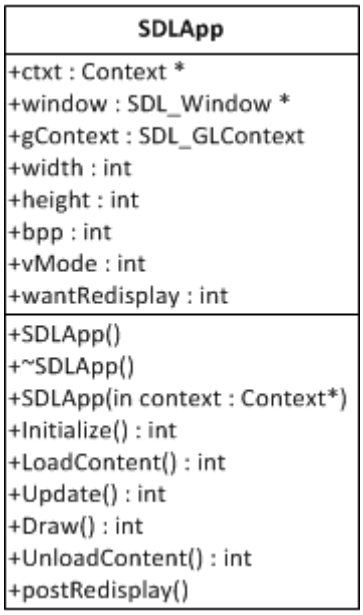


Figura 4.18 Diagrama de la clase SDLApp.

SDL provee la capacidad de comunicarse de forma abstracta con cada uno de los subsistemas de los distintos sistemas operativos. Para iniciar una aplicación en SDL, sólo basta con llamar:

```
if(SDL_Init(SDL_INIT_EVERYTHING) < 0) {
       return false;
   }
```

En todos los sistemas operativos, para inicializar una ventana con un contexto de OpenGL se requiere de instrucciones específicas para cada plataforma, en la parte de Información adicional del apéndice se incluye información sobre los requisitos de cada sistema operativo para lograr esto.

Para tener acceso a un contexto de OpenGL, una llamada ejemplo de SDL es:

```
if((Surf_Display = SDL_SetVideoMode(640, 480, 32, SDL_HWSURFACE | SDL_GL_DOUBLEBUFFER |
SDL_OPENGL)) == NULL) {
   return false;
}
```

Dependiendo de los parámetros con los que se llame, se le puede pedir a SDL que habilite sólo cierta funcionalidad para OpenGL. Antes de esta llamada, por ejemplo, se pueden definir atributos para el contexto de OpenGL como el tamaño de los búferes de profundidad:

```
SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE,      16);
SDL_GL_SetAttribute(SDL_GL_BUFFER_SIZE,            32);
```

Finalmente, para ponerle nombre a una ventana a través de todos los sistemas operativos, basta con:

```
/* Set the title bar in environments that support it */
SDL_WM_SetCaption("Tesis", NULL);
```

Con el sistema de sonido, SDL provee una alternativa similar:

```
//Initialize SDL_mixer
if( Mix_OpenAudio( 22050, MIX_DEFAULT_FORMAT, 2, 4096 ) == -1 )
{
  printf("Audio failed");
  return 1;
}
```

Para el caso de esta aplicación, se creó una clase denominada *Context*, dentro de la cual se guardan todos los valores necesarios para iniciar y hacer funcionar nuestra aplicación de acuerdo a la configuración acordada con el usuario.

Para evitar compilar nuevos valores de inicialización de OpenGL y SDL una y otra vez, se implementó la capacidad de definir todo mediante un archivo de texto, el tamaño de la ventana, número de colores, tamaño de búferes, incluso se puede determinar si se quiere hacer una aplicación con OpenGL o Software Rendering.

Esta flexibilidad, convierte a la capa de aplicación en algo independiente que puede usarse para cualquier clase de aplicación, no específicamente una aplicación gráfica, pero cualquier otra clase de contexto provisto por SDL puede ser inicializado por esta capa.

Administrador de Sonido

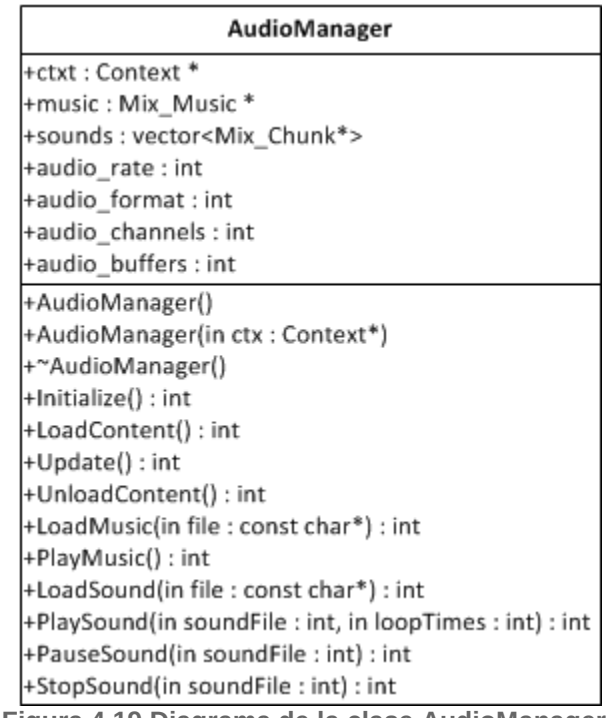


Figura 4.19 Diagrama de la clase AudioManager.

El único trabajo del administrador de sonido es cargar y tocar sonidos cuando se le indica, la carga se hace durante la fase de carga de un nivel. Para tocar un sonido basta con indicarle al contexto que debe de tocarse el sonido y el administrador lo hará. Existe lógica básica para no tocar más de un sonido en un mismo canal a menos que sea el efecto deseado.

Administrador de entradas y salidas

El administrador de entradas y salidas se encarga solamente de interpretar los distintos eventos obtenidos por el sistema operativo. En este caso interpreta los eventos enviados por el teclado y el mouse, de tal modo que se pueda presionar más de una tecla al mismo tiempo y la repetición de teclas que a veces envía el sistema operativo sea ignorada. Para esto se tiene un arreglo de todas las teclas posibles que se pueden presionar, y dependiendo del último evento, se le asigna un estado de encendido o apagado.

Interacción de las distintas partes de la arquitectura y comunicación entre procesos.

En una aplicación interactiva gráfica, hay una gran cantidad de procesos, cuyas salidas dependen de las salidas de otros, se requiere de una alta comunicación entre cada una de las partes de la aplicación para crear algo coherente con las reglas establecidos por los diseñadores. La solución para este problema, fue la creación de un espacio de memoria compartida: la clase Contexto.

La arquitectura diseñada completa puede verse mediante los siguientes diagramas de forma simplificada:

La clase Contexto

Todos los sistemas inicializados por la aplicación gráfica, el sonido, los cargadores de contenido y el renderizador, pueden consultarse fácilmente mediante el espacio de memoria compartida que es el Contexto.

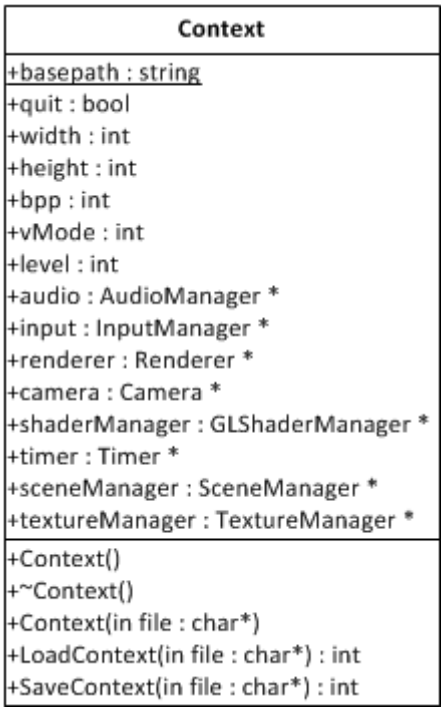


Figura 4.20 Diagrama de la clase Context.

La clase contexto contiene apuntadores hacia todos los elementos básicos de el motor gráfico, contiene el estado de cada uno de los subsistemas, para reconocer fácilmente cuándo debe de hacerse un cambio, cuál es el sonido que actualmente se está tocando o qué figura se está tratando de renderizar al momento; es un modo de sincronizar todos los elementos de la aplicación y, más aún, guardarlos en una estructura de datos coherente y poder implementar cosas como: métodos para salvar el juego, la configuración y el progreso del jugador, toda la memoria que debe de liberarse dependiendo de dónde se encuentra el jugador y cuáles elementos ya no se usarán a lo largo del juego.

La capa de aplicación es la encargada de inicializar a nuestro contexto, el cual se guarda en un archivo de texto. Si no existe un Contexto, es creado, y si existe, se lee para inicializar la aplicación de SDL de acuerdo a una serie de banderas y valores iniciales.

El contexto le indica a la aplicación que subsistemas inicializar, de acuerdo a la configuración que existe por defecto o la que se ha guardado en un archivo.

Una vez que se han inicializado todos los subsistemas, la aplicación procede a determinar la escena que debe de cargar. Para esto, se creó un administrador de escenas que, dependiendo del contexto, se encarga de cargar los modelos que se usarán.

Las relaciones entre los subsistemas pueden verse en la siguiente figura:

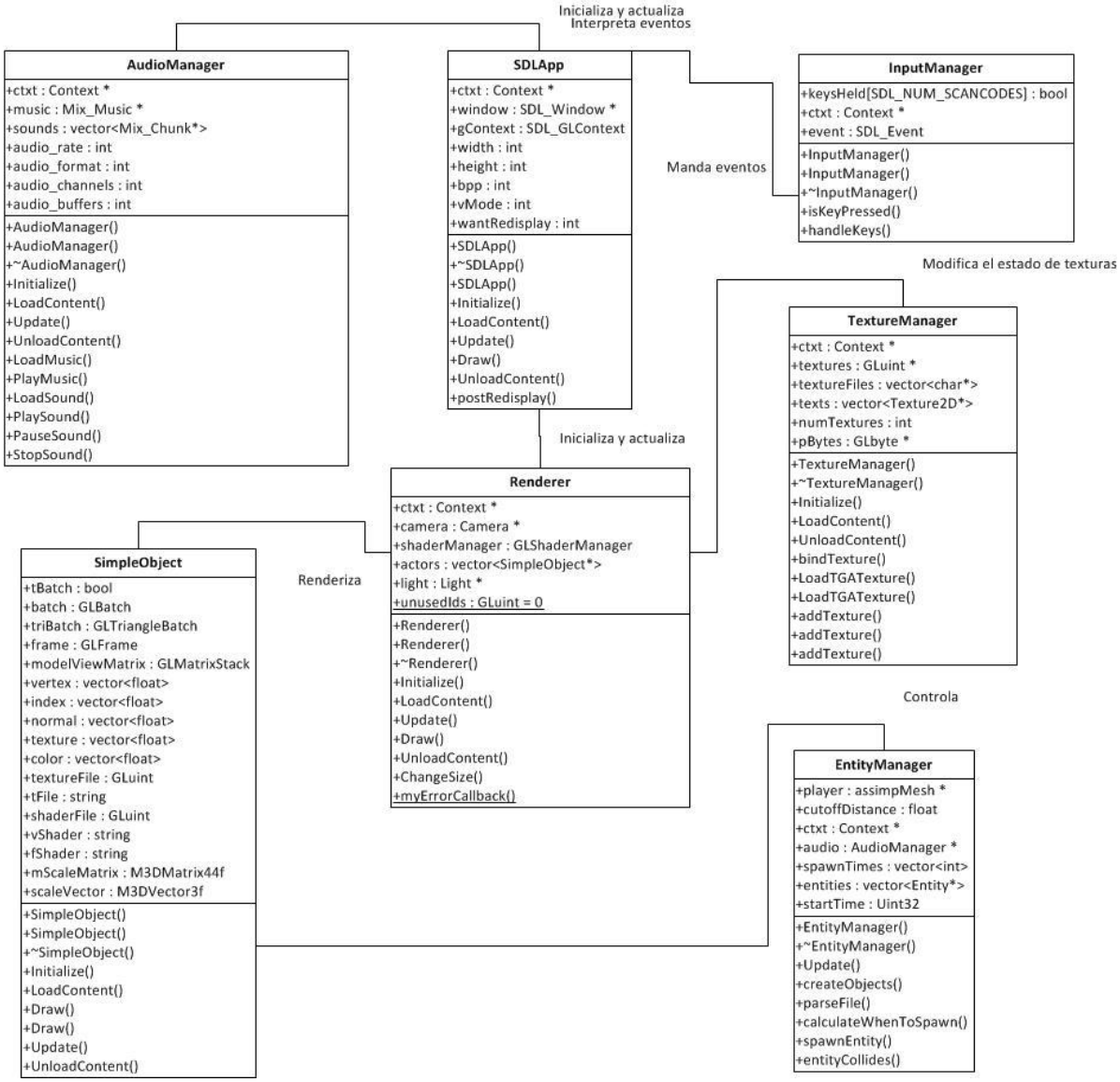
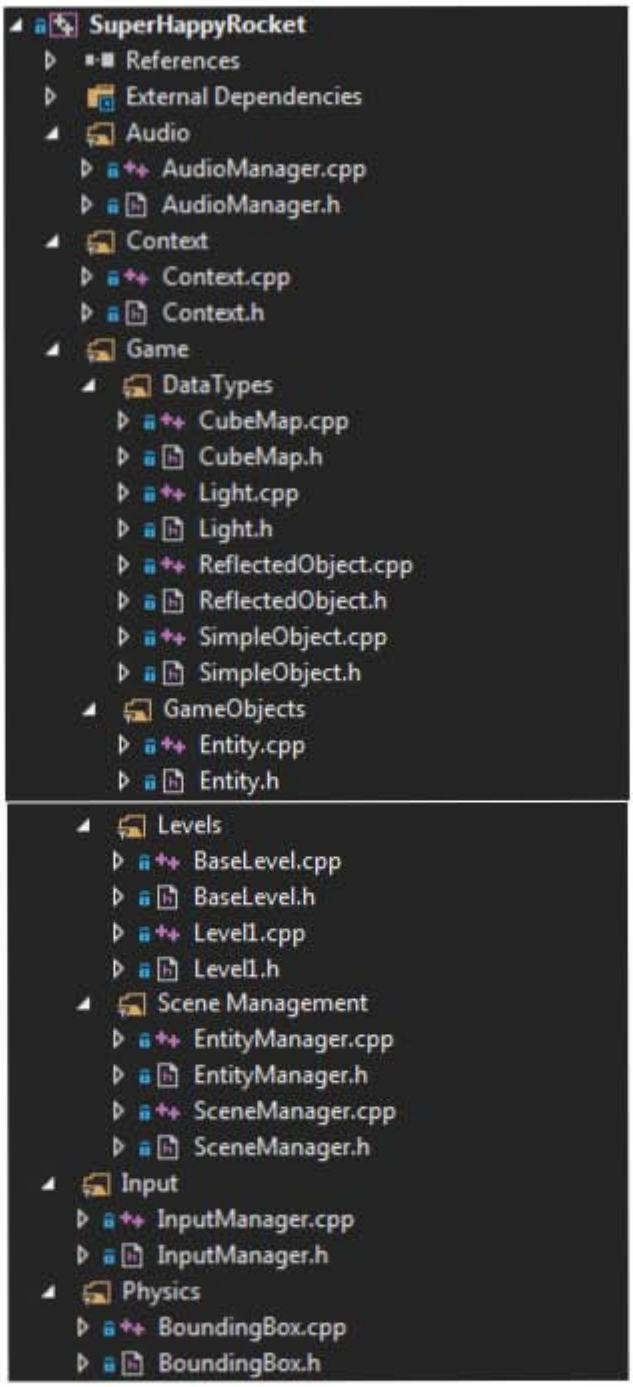


Figura 4.21 Relación entre los subsistemas de la aplicación.

Aquí se puede apreciar como la capa de aplicación, conformada por SDLApp, se encarga de inicializar y controlar las llamadas hacia los otros elementos del motor. SDLApp es el encargado de enviar el mensaje de inicialización a todos los subsistemas de la aplicación gráfica, InputManager se comunica con SDLApp para reconocer e interpretar las teclas presionadas por el usuario, mientras que AudioManager inicializa el sistema de Audio y toca los sonidos apropiados.

El rendererizador ordena SimpleObjects y llama a su método Draw() y Update() en cada frame, SimpleObject es la única clase de objeto que conoce y todos los objetos que dibujan algo en pantalla heredan de éste.

Una vez unidos todos los subsistemas dan lugar a la arquitectura final de la aplicación, como se muestra en la figura 4.22.


SuperHappyRocket
References
External Dependencies
Audio
AudioManager.cpp
AudioManager.h
Context
Context.cpp
Context.h
Game
DataTypes
CubeMap.cpp
CubeMap.h
Light.cpp
Light.h
ReflectedObject.cpp
ReflectedObject.h
SimpleObject.cpp
SimpleObject.h
GameObjects
Entity.cpp
Entity.h
Levels
BaseLevel.cpp
BaseLevel.h
Level1.cpp
Level1.h
Scene Management
EntityManager.cpp
EntityManager.h
SceneManager.cpp
SceneManager.h
Input
InputManager.cpp
InputManager.h
Physics
BoundingBox.cpp
BoundingBox.h

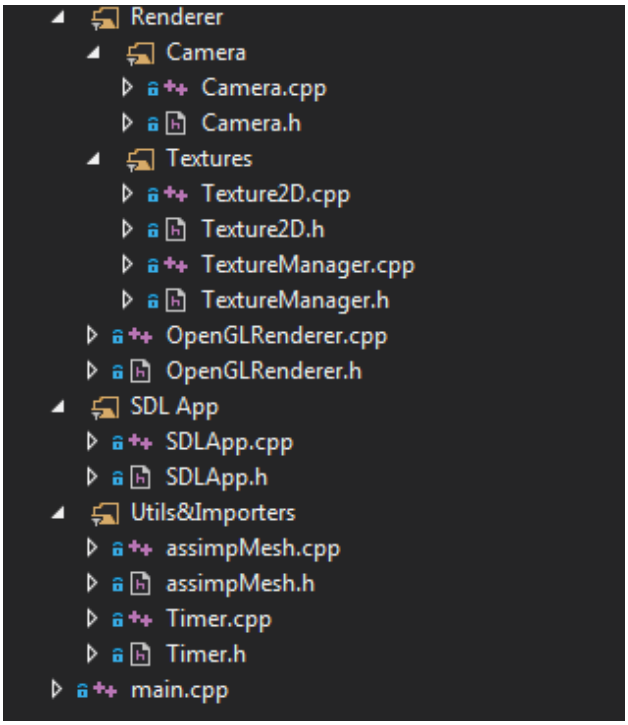


Figura 4.22 Arquitectura final de la aplicación.

Diseño de los personajes y modelos para la aplicación gráfica

Para poder realizar la aplicación demostrativa y cumplir con el objetivo planteado, se requirió hacer el diseño de un nivel incluyendo los personajes y la ambientación.

Para el diseño de algunos personajes, se utilizaron imágenes de algunos otros personajes conocidos como referencia, para el gato se utilizó una combinación entre el cuerpo del personaje Kirby, y para el diseño de la cara el gato de la caricatura japonesa Chi's Sweet Home, como se muestra en la figura 4.23.

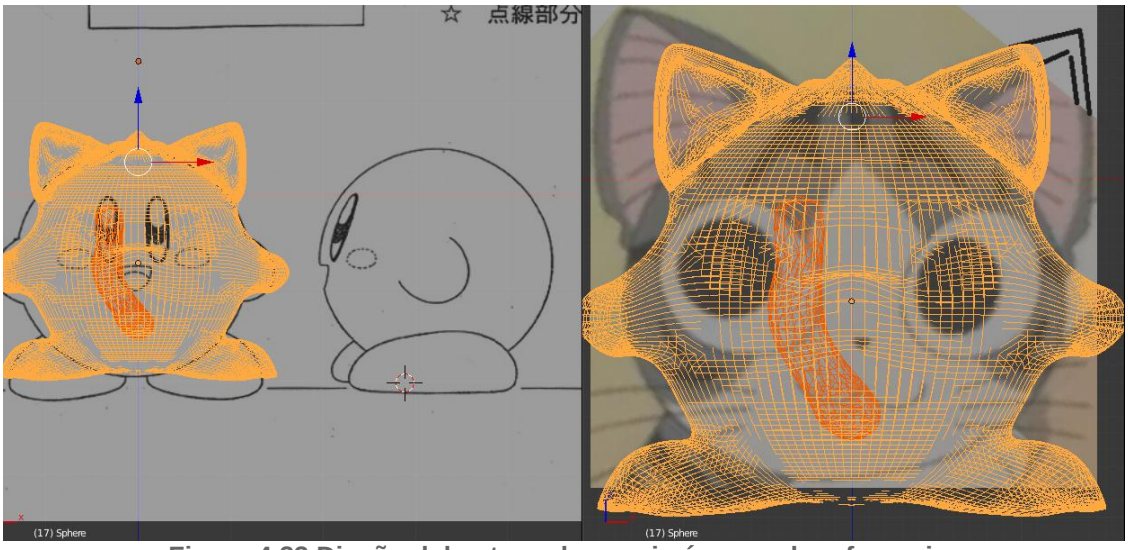


Figura 4.23 Diseño del gato en base a imágenes de referencia.

Para lograr un mejor modelo, que no parezca una simple esfera, se utilizó la técnica de esculpido en Blender, la cual permite crear mallas más realistas y con efectos como pelaje, relieves, etc. Sin embargo el resultado obtenido tiene un número muy elevado de polígonos tal y como se muestra en la figura 4.24.

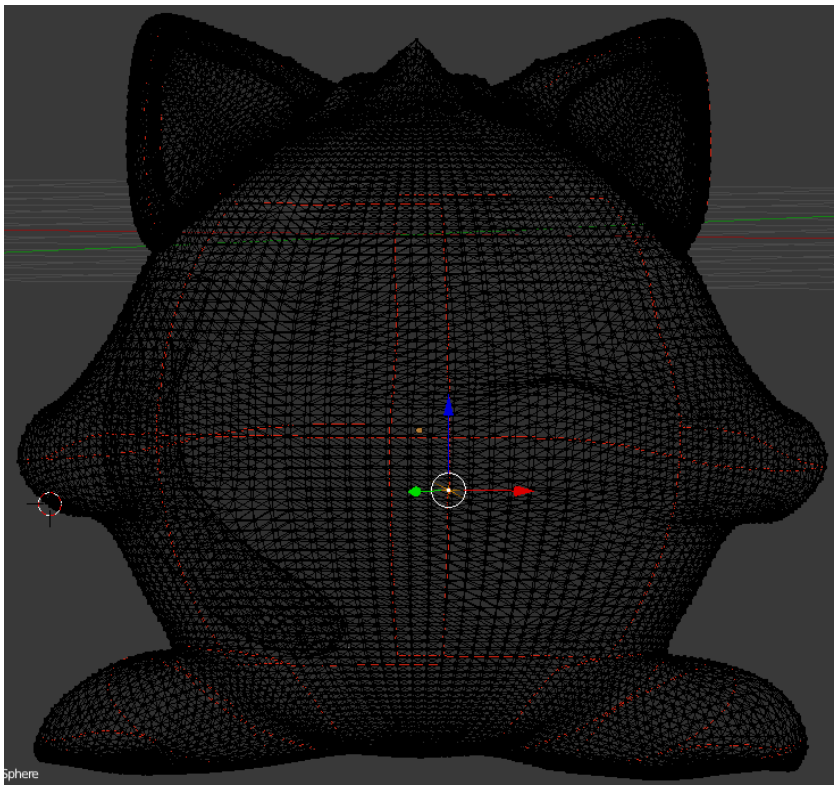


Figura 4.24 Modelo con un gran número de polígonos, obtenido después de utilizar la herramienta de esculpido de Blender.

Un modelo con tantos polígonos no es bueno, pues requiere de muchos recursos y tiempo para ser cargado, de igual forma siempre que se vaya a realizar una transformación con el modelo gastará muchos recursos en realizar las operaciones debido al gran número de vértices. Para solucionar este problema se optó por crear una versión con menos polígonos del modelo, el modificador de Blender “Decimate” permite reducir el número de vértices. Utilizando el modificador Decimate con un índice de 0.25, se logró obtener un modelo sin gran pérdida de definición, pero con mucho menos vértices, el modelo obtenido se muestra en la figura 4.25.

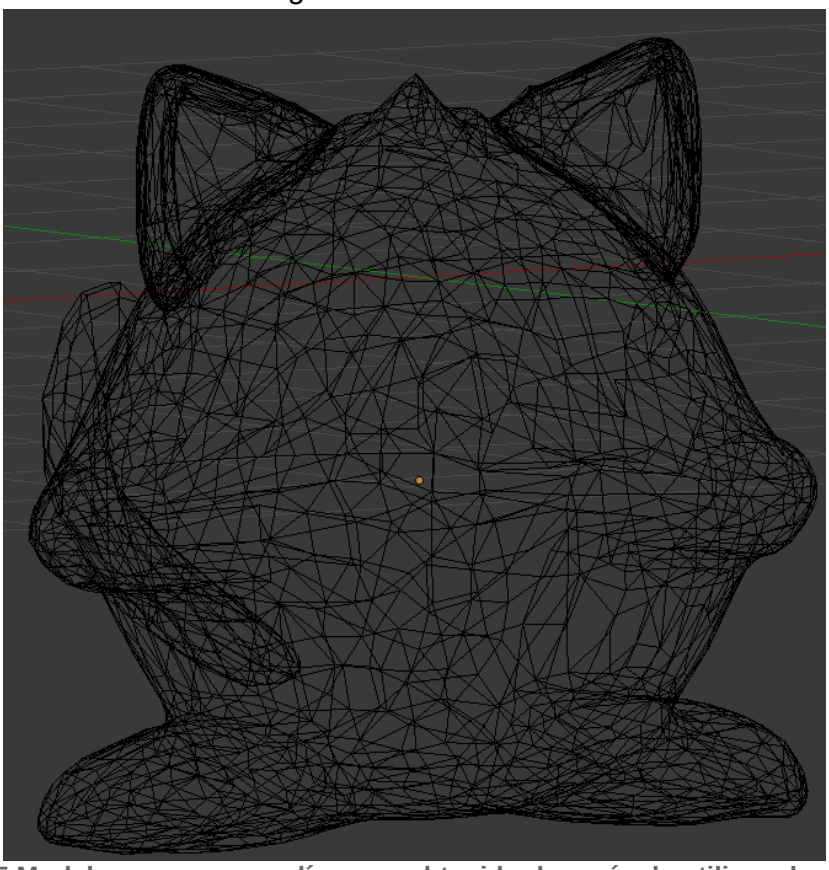


Figura 4.25 Modelo con menos polígonos, obtenido después de utilizar el modificador Decimate en Blender.

Los modelos 3D se hicieron con mallas poligonales basadas en triángulos, como se muestra en las figuras 4.26 y 4.27. El triángulo es el polígono más simple que existe en el espacio euclidiano, y esto facilita la forma de cargar los modelos en OpenGL. Las texturas utilizadas son potencias de 2 y cuadradas (256x256, 512x512, etc), la razón de usar texturas potencias de dos es que dichas texturas son las utilizadas en OpenGL para un cargado de texturas más óptimo y rápido, además algunas tarjetas gráficas antiguas no soportan texturas no-potencias de dos. El usar texturas no potencias de dos puede provocar un peor rendimiento e incluso que las texturas pierdan calidad y se deformen.

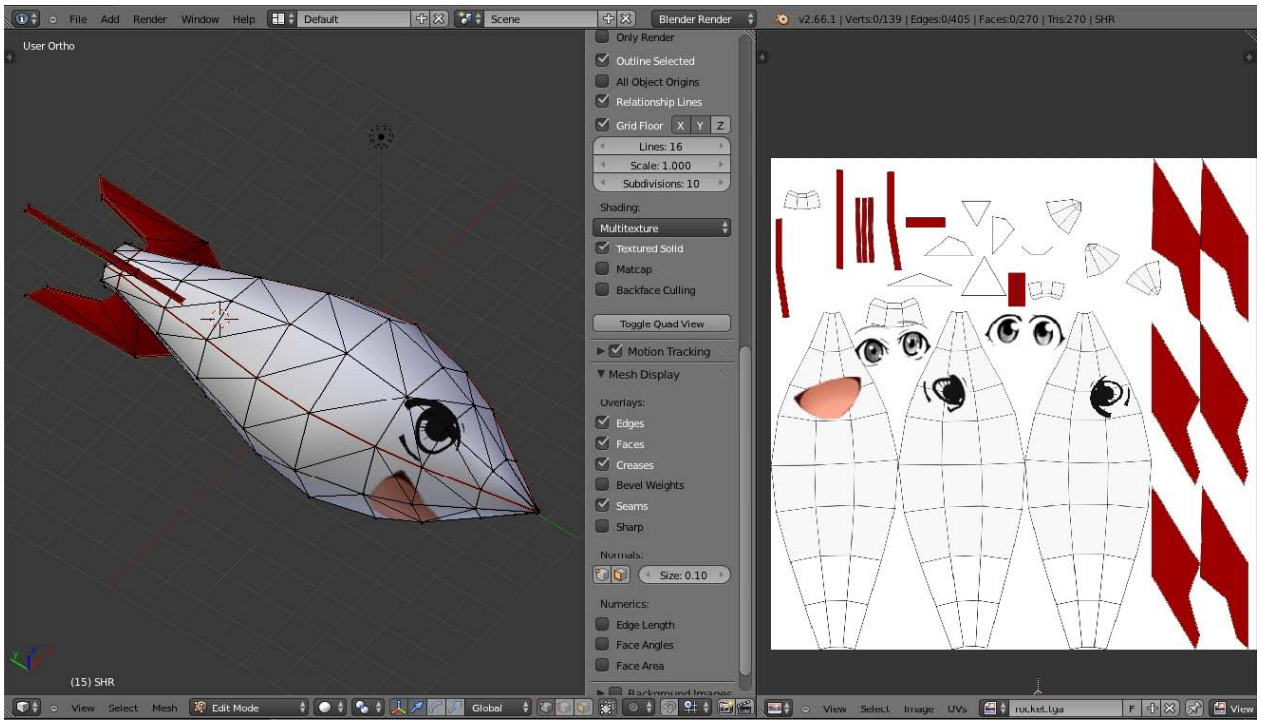


Figura 4.26 Modelado 3D y texturizado de uno de los personajes diseñados.

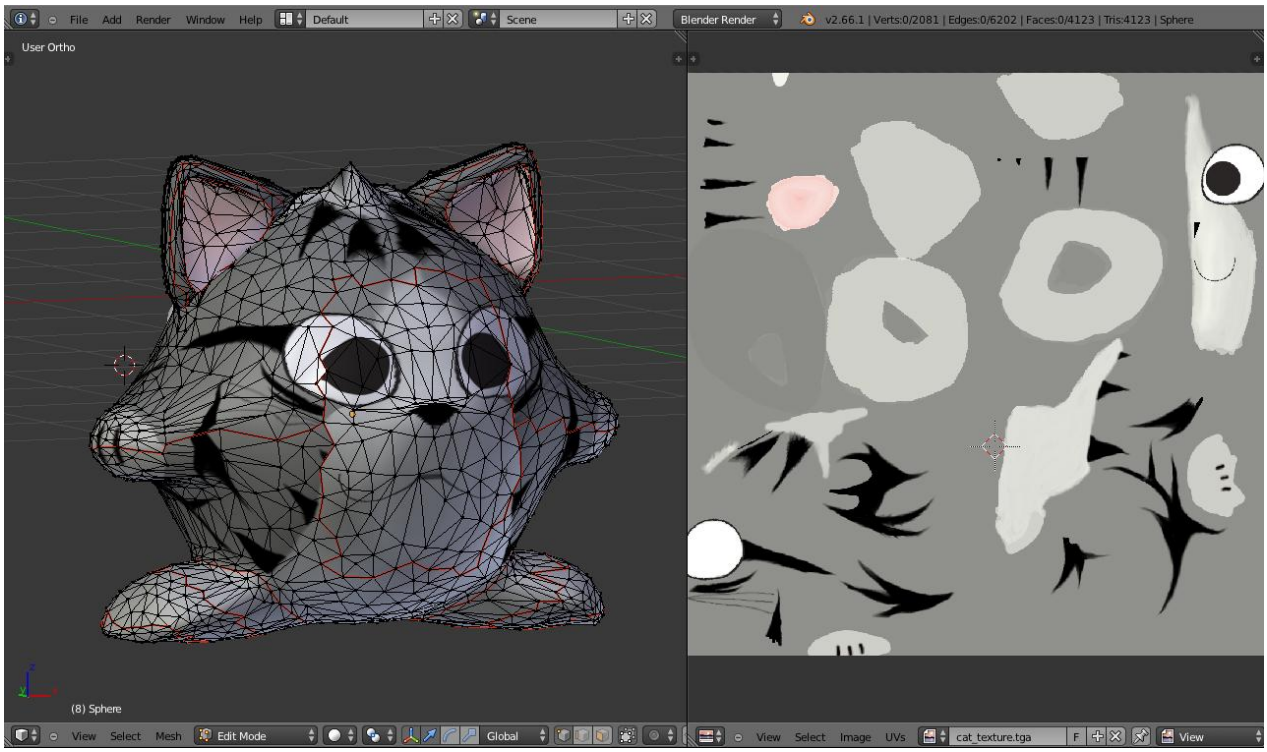


Figura 4.27 Modelado 3D y texturizado de uno de los personajes diseñados.

Para un correcto mapeo de las texturas se utilizó la herramienta de blender UV unwrap, la cual permite definir la forma en la cual la textura se envuelve alrededor de un modelo por medio de un mapa de vértices como se muestra en la figura 4.28.

El nivel diseñado para la aplicación demostrativa se lleva a cabo en una ciudad, por lo cual se diseñó un bloque conformado por varios edificios de pocos polígonos para utilizar en el fondo de la escena, y lograr que la ciudad se vea más realista que una simple textura de fondo. La figura 4.28 muestra el bloque de edificios empleado como ciudad de fondo, y el UV unwrap utilizado en uno de los edificios.

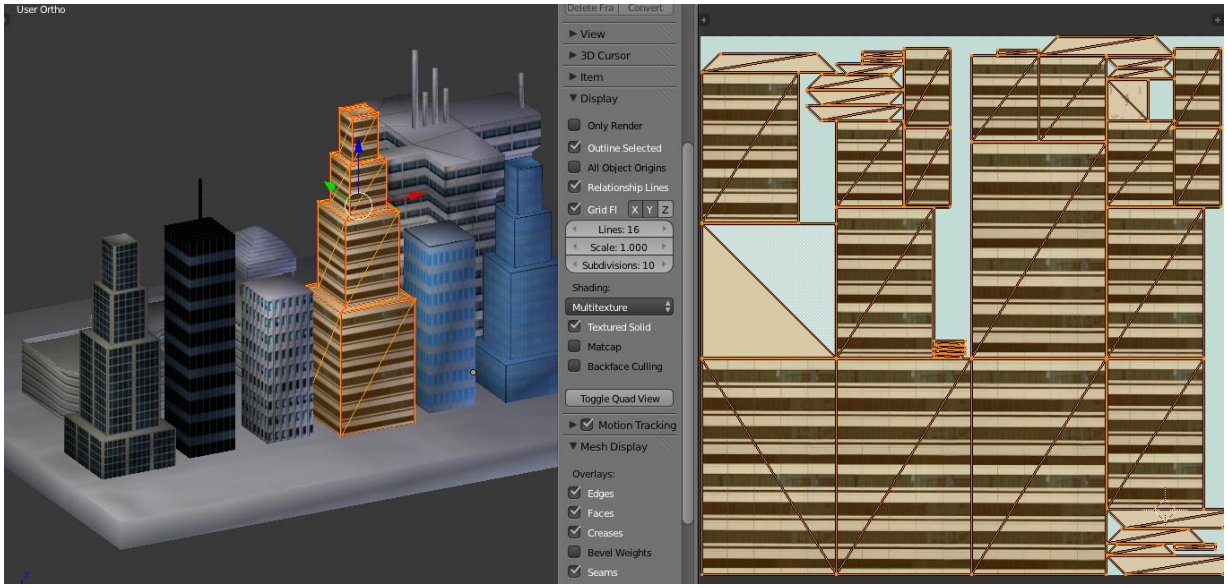


Figura 4.28 UV unwrap de un edificio de la ciudad diseñada.

Una vez renderizados los modelos dentro de la aplicación lucen así:

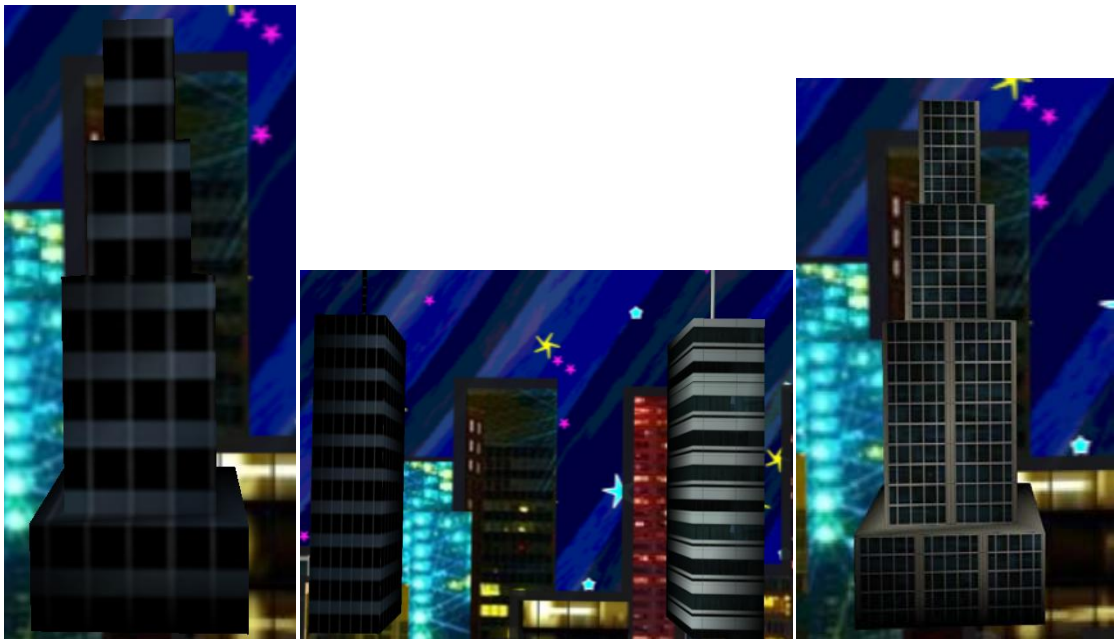


Figura 4.29 Modelos cargados dentro de la aplicación demostrativa.

Mientras que una escena en la aplicación demostrativa final se ve así:

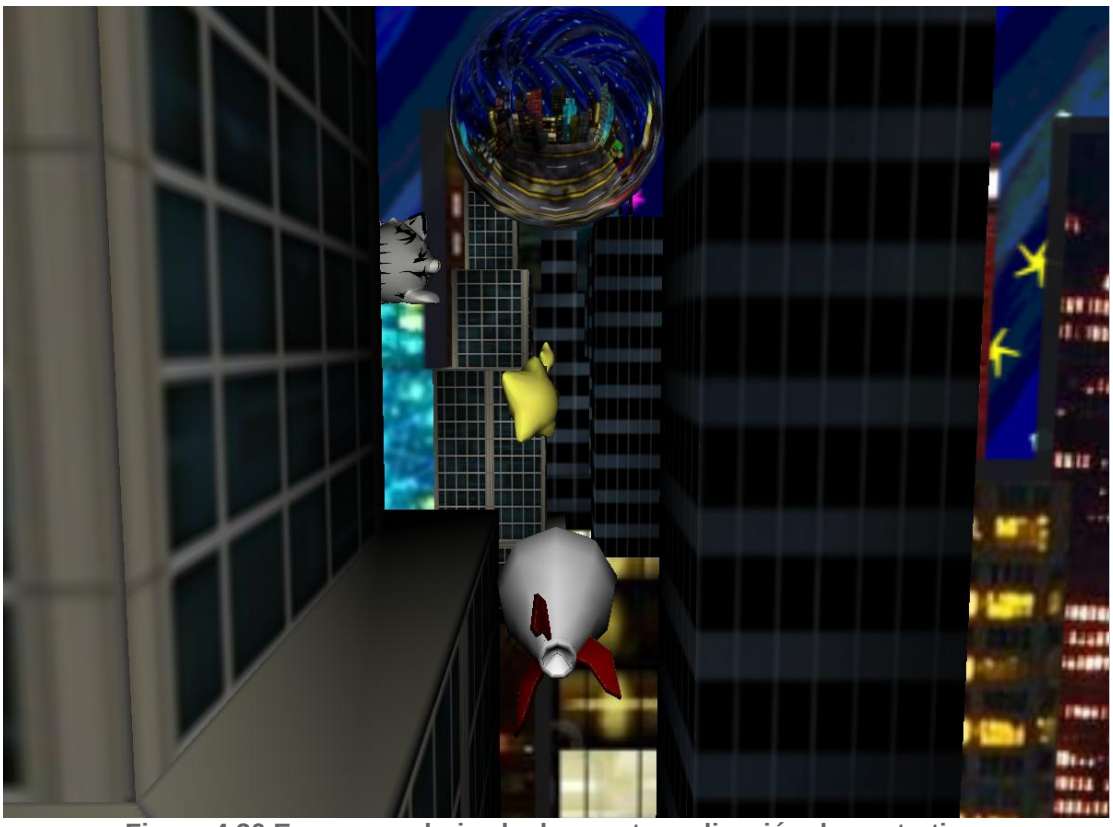


Figura 4.30 Escena renderizada de nuestra aplicación demostrativa.

Para efectos demostrativos, los mismos modelos con texturas fueron cargados en la herramienta de desarrollo UNITY, el resultado fue el siguiente:

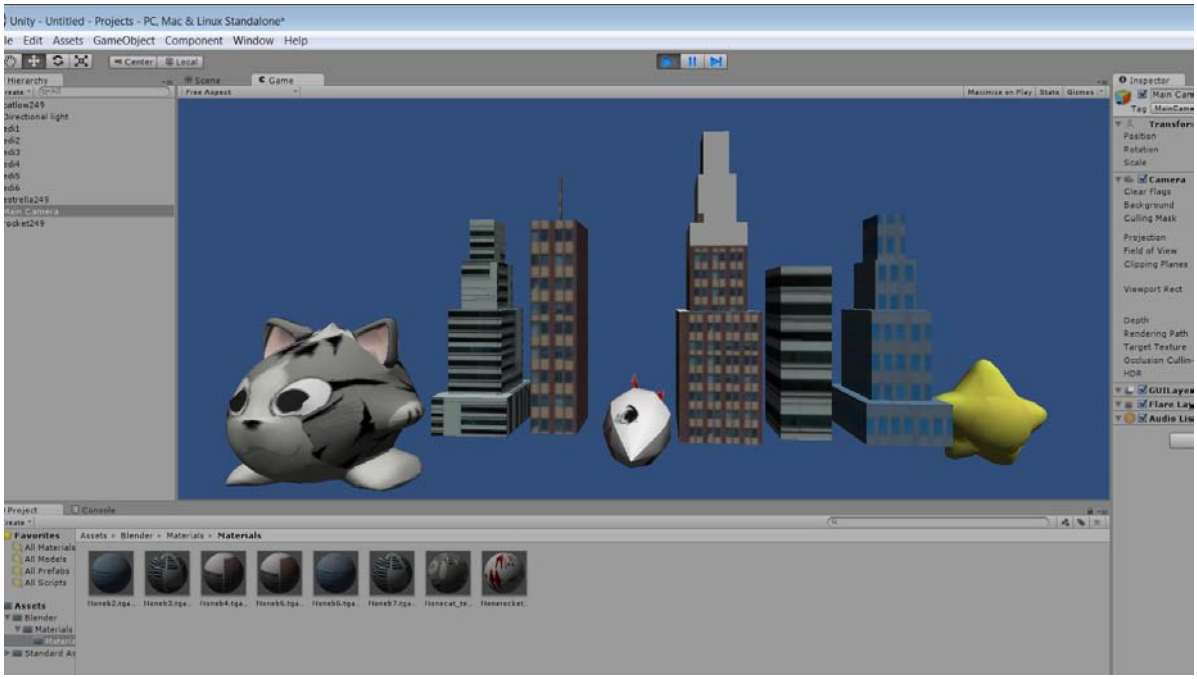


Figura 4.31 Modelos cargados en Unity.

Como puede apreciarse el mapeo de las texturas es un poco erróneo para algunos modelos, por lo que se consiguió un mejor mapeado que el provisto por UNITY, sin embargo, nuestro renderizador de modelos se limita a obtener texturas en formato .tga, mientras que para UNITY no es importante. Un punto a desarollar más adelante sería aceptar más formatos de texturas. El resultado final de los modelos renderizados por UNITY con sus opciones básicas es el siguiente:

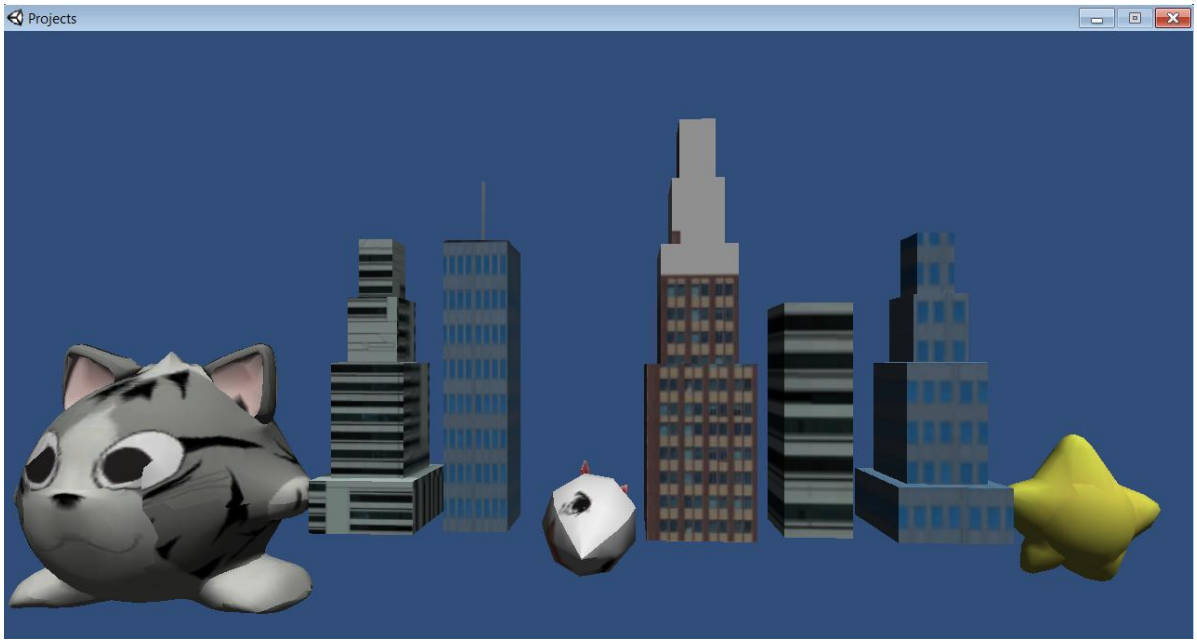


Figura 4.32 Renderizado de los modelos en Unity.

Pruebas

Para el desarrollo de las pruebas, se tomaron dos acercamientos. En cada uno de los archivos cpp creados se hicieron sencillas pruebas unitarias, para comprobar el resultado de las funciones más básicas. Algunos ejemplos de las pruebas que se realizaron son:

```
void VectorTest()
{

    cout << "Testing vectors and operations" << endl;

     Vector3<float> A;
     Vector3<float> B(4.0f,5.0f,6.0f); //direct input for Vector B
     Vector3<float> C;
     Vector3<float> D;
     Vector3<float> E;
     Vector3<float> F;

     A[0] = 1.0f;
     A[1] = 2.0f;
     A[2] = 3.0f;
     C = A + B;
     D = B - A;
     E = A * 5.0f;
     F = Vector3f::ZERO;
     cout << "A = " << A << endl;
     cout << "B = " << B << endl;
     cout << "C = A + B = " << C << endl;
     cout << "D = B - A = " << D << endl;
     cout << "E = A * 5 = " << E << endl;
     cout << "F = ZERO = " << F << endl << endl;


}
```

```
void MatrixTest()
{
    cout << "Testing Matrix and operations" << endl;

    Matrix3<float> matA(1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f);  // Direct input for the matrix
    Vector3<float> A;
    Vector3<float> B;
    Vector3<float> C;

    A[0] = 1.0f;
    A[1] = 2.0f;
    A[2] = 3.0f;
    B = A * 2;
    C = A + B;

    Matrix3<float> matB(A,B,C,1);  // Matrix created from 3 vectors
    Matrix3<float> matC(1.0f,2.0f,3.0f);  // Create a diagonal Matrix
    Matrix3<float> matD = matB;  // Assignment operator
    Matrix3<float> matE;
    Matrix3<float> matF;
    Matrix3<float> matG;

    matE.MakeZero();  //Make matE a Zero Matrix

    matF = matA * matE; //testing overloaded Operations
    matG = matA + matB; //testing overloaded Operations

    cout << matA; //matA (direct input)
    cout << matB; //matB (created from 3 vectors)
    cout << matC; //matC  (created as a diagonal  matrix)
    cout << matD; //matD  (testing assigment operator)
    cout << matE; //matE (zero matrix)
    cout << matF; //matF (testing overloaded operators)
    cout << matG; //matG (testing overloaded operators)

}
void AvectApoint()
{
    cout << "Testing AVectors and APoints Operations" << endl;

    Tuple<3,float> test;
    test[0] = 2.2f;
    test[1] = 3.3f;
    test[2] = 4.4f;

    Vector3f A(2.0f, 2.0f, 2.0f);

    APoint ap1(1.0f, 3.0f, 5.0f);       //initialized with 3 floats
    APoint ap2(1.0f, 1.0f, 1.0f);
    APoint ap3(ap1);                    //initialized from an APoint
    APoint ap4(test);                   //initialized from a Tuple
    APoint ap5(A);                      //initialized from a vector3f

    AVector av1(1.0f, 2.0f, 3.0f);      //initialized with 3 floats
    AVector av2(test);                  //initialized from a Tuple
    AVector av3(A);                     //initialized from a vector3f

    ap2 = ap1;                          //assignment operator
    ap3 = ap2 - av2;                    //poit - vector = point
    av1 = ap1 - ap2;                    //point - point = vector
    ap4 = ap2 * 3;                      //point by scalar
    ap5 += ap1;                         // += operator

    av1 = av3.Cross(av2);               //assignment and cross operator

    cout << av2.Length() << endl;       //length
    cout << "av1 = "<< av1.X() << " " << av1.Y() << " " << av1.Z() << endl<< endl;

}
```

Figura 4.33 Pruebas de operaciones entre vectores, matrices y puntos.

Estas pruebas básicas se corrían siempre que se realizaban cambios en el motor de gráficos, sus resultados debían ser siempre iguales y asegurando así la estabilidad del producto. Son una gran herramienta de trabajo si se buscan hacer cambios en la base de una aplicación, se pueden medir impactos, identificar regresiones, corregir problemas en pequeña escala antes de que se conviertan en errores difíciles de depurar. Para fines de este proyecto los resultados se comprobaron manualmente, pero podría automatizarse también este proceso programando una infraestructura que lo hiciera por nosotros.

Además, se realizó una pequeña suite de pruebas de integración. Aquí fue donde se desarrolló una pequeña aplicación de texto dentro de la cual se probaron las distintas operaciones entre matrices, transformadas y operaciones entre vectores. Estas pruebas no fueron gráficas, pero probaban casos de colisión con objetos, planos o puntos, de movimiento y de cambio de posición con respecto a ciertas entradas automatizadas, son una gran manera de introducir cambios en el código y saber que todo sigue funcionando de forma adecuada.

```
void HplaneTest()
{
    cout << "Testing homogeneus planes" << endl;
    HPlane plane1(1.0f, 1.0f, 1.0f, 3.0f);  //Specify N and c directly
    AVector av1(1.0f, 1.0f, 1.0f);      //initialized with 3 floats
    HPlane plane2(av1, 2.0f);
    APoint ap1(1.0f, 3.0f, 5.0f);       //initialized with 3 floats
    HPlane plane3(av1, ap1);
    HPlane plane4 = plane3;             //assignment operator
    cout << "Plane1 " << "( " << plane1[0] << ", " << plane1[1] <<
        ", " << plane1[2] << " ) " << plane1[3] <<endl;
    cout << "Plane2 " << "( " << plane2[0] << ", " << plane2[1] <<
        ", " << plane2[2] << " ) " << plane2[3] <<endl;
    cout << "Plane3 " << "( " << plane3[0] << ", " << plane3[1] <<
        ", " << plane3[2] << " ) " << plane3[3] <<endl;
    plane3.Normalize();                 //Normalize
    cout << "Normalized Plane3 " << "( " << plane3[0] << ", " << plane3[1] <<
        ", " << plane3[2] << " ) " << plane3[3] <<endl;
    cout << "Distance from plane 2 to AP1: " << plane2.DistanceTo(ap1) << endl << endl; //Distance to a point
}
```

```
void BoundTest()
{
    cout << "Testing Bounds" << endl;
    Bound bound1;
    APoint center(1.0f, 1.0f, 1.0f);
    bound1.SetCenter(center);
    bound1.SetRadius(2.0f);
    Bound bound2 = bound1;
    HPlane plane1(1.0f, 1.0f, 1.0f, 3.0f);
    cout << bound1.WhichSide(plane1) << endl;
    //test for intersection
    cout << bound1.TestIntersection(bound2) << endl;
    AVector av1(1.0f, 2.0f, 3.0f);
    cout << bound2.TestIntersection(center, av1, 1.0f, 2.0f) << endl ;
}
```

Figura 4.34 Pruebas de intersecciones y colisiones.

Finalmente se realizó una prueba gráfica para comprobar que todo funcionara correctamente, como la carga de modelos, shaders, archivos de audio, cubemaps, así como el correcto mapeado de texturas y sus normales. Se aplicaron también

transformaciones básicas para observar las interacciones entre algunos objetos. Se creó el nivel de depuración con el cual se hace la comparación con Unity3D.

Depuración de la aplicación

Para la depuración de la aplicación, se hizo uso de la herramienta gDebugger de AMD, ahora conocido como CodeXL, para utilizarla basta con abrir el ejecutable desde este programa y poner algunos breakpoints en los puntos de interés.

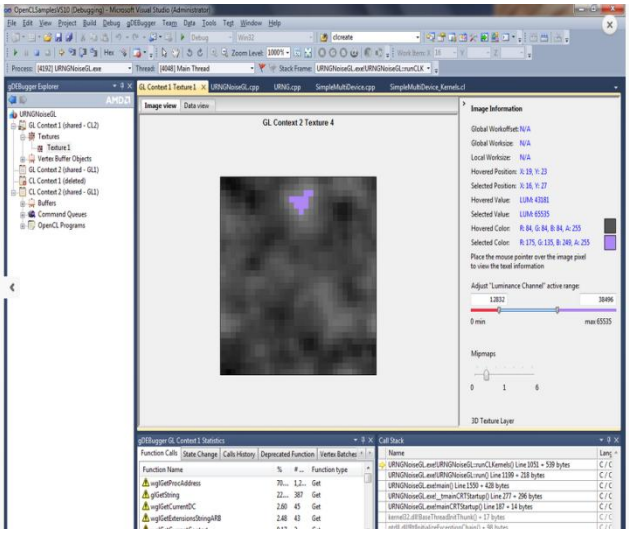


Figura 4.35 Herramienta CodeXL de AMD.

Control de versiones y disponibilidad de código fuente

Para el control de versiones se hizó uso de la herramienta Git y del sitio web github, para llevar el control de los cambios de forma remota y a distancia. Github permitió crear un repositorio local de todas las versiones de la aplicación, así como mantener este historial en línea, con el cual se pudo regresar a versiones anteriores o tomar sólo algunas líneas de código de éstas cuando se presentó algún conflicto con una nueva modificación.

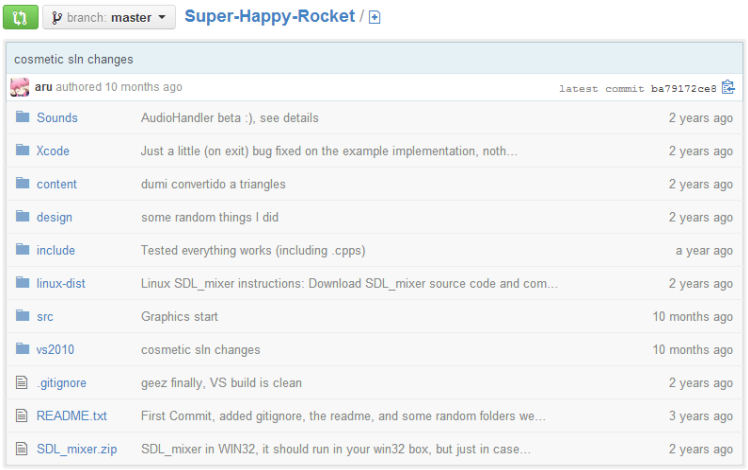


Figura 4.36 Herramienta de control de versiones Git.

Mediante una extensión en Visual Studio es posible obtener los cambios hechos por los demás y subir al servidor los cambios que uno mismo ha hecho. La diferencia con otros sistemas similares es que sólo se toma control de una línea a la vez en lugar de todo un archivo.

Nuestro código fuente puede ser obtenido de Github, en la siguiente URL: https://github.com/aru/superhappyrocket. En el apéndice de este trabajo se pueden obtener instrucciones para el desarrollo de una aplicación propia, se incluye un tutorial para la compilación y uso básico de nuestras librerías.

5. Conclusiones

La elaboración de este motor gráfico se inserta como una alternativa para el desarrollo multiplataforma dentro de la universidad, ya que parte de principios menos complejos que otras soluciones en el mercado, es una gran opción como una herramienta educativa para la elaboración de otras aplicaciones gráficas que no requieran de especificaciones muy elevadas. Es comparable a otras librerías de manejo de ventanas y contextos de OpenGL como glut.

Al haberse basado en experiencias previas con otros motores gráficos, se pudo derivar un producto de menor escala pero con mucho potencial. Se logró una deconstrucción de los elementos básicos que conllevan realizar un producto de tal magnitud.

La arquitectura de el motor gráfico, es sencilla y extendible, a diferencia de otras alternativas en el mercado, todo el código es visible y modificable, es fácil de entender por cualquier estudiante que haya cursado al menos una materia básica de Computación Gráfica y funge como base para mayores experimentos.

En comparación con otras herramientas en el mercado, como Unity3D, no es tan fácil crear aplicaciones, Unity3D tiene la interfaz gráfica directa que muestra lo que se verá en el producto final, pero esto podría agregársele en forma de otro nivel, donde puedan colocarse objetos que han de ser guardados en un grafo de escena para ser cargados después en otro modo de interacción.

Sin embargo, las herramientas con las que cuenta Unity3D o XNA, también se pueden encontrar de manera gratuita, desde depuradores gráficos hasta herramientas para controlar las versiones de cada archivo u objeto modificado. Quizás algo que se puede agregar son profilers, herramientas para determinar qué tan eficiente es una aplicación, que partes del código son las más ejecutadas, la duración de las llamadas a funciones, y la cantidad de memoria que está utilizando por el momento. Sin embargo, agregar esto no es tan difícil, bastaría con poner algunos timers en lugares donde se llaman cada uno de los 4 estados principales y agregar los tiempos para saber dónde se puede seguir optimizando la aplicación.

Es fácil aprender de la aplicación elaborada, no está compuesta de estructuras demasiado complejas y provee una buena idea sobre cómo desarrollar un producto más elaborado. El motor gráfico tiene un enfoque específico, el cual puede llegar a ser competitivo, se cuenta con un mejor exportador de Blender a un modelo intermedio interpretado por OpenGL que en Unity3D, aunque no posee la interfaz amigable de Unity3D los entusiastas de C++ pueden extraer y modificar el código para aprender de la tesis, si los requisitos de su aplicación son los adecuados. Puede utilizarse como herramienta alternativa de aprendizaje, en vez de usar glut u otras librerías intermedias para el desarrollo multiplataforma dentro del salón de clases.

Gracias a la realización de la tesis, se reforzaron y expandieron conocimientos adquiridos en la facultad, la creación de un motor que soporte una aplicación gráfica multiplataforma es una tarea extensa. Las áreas de oportunidad son muy variadas, y con un equipo pequeño es difícil sacar el máximo provecho a un sistema cuando el resultado final es mucho más importante. El producto realizado puede optimizarse en muchas áreas, pero la creación de un producto final es mucho más importante.

Aunque la separación de los objetos en la arquitectura diseñada no terminó siendo tan rigurosa, en una aplicación gráfica es necesario que varios objetos puedan conocer el estado de otros de manera rápida y eficiente, por lo que la creación de un segmento compartido de memoria es una gran lección aprendida durante la creación de esta aplicación.

El algoritmo de renderizado creado es sencillo, aunque hace uso de un arreglo ordenado de atributos para determinar cómo dibujar la geometría de un nivel, éste puede extenderse fácilmente, basta con modificar la estructura de datos con la que se guardan los objetos, para realizar una subdivisión por nodos de un árbol como en un octree,

después de la cual puede aplicarse el algoritmo aún sin modificación alguna, ordenando la geometría sólo una vez a menos que cambie de estado constantemente.

Para la realización de un motor gráfico se requiere de mucho trabajo reinventando formas de leer texturas, estructuras de datos generadas por herramientas de modelado o de creación de imágenes. Como ejercicio de aprendizaje, la creación de este motor llevó a intentar crear programas que realizarán estas tareas desde cero, sin embargo, al final se optó por usar librerías con el comportamiento que se deseaba emplear, ya que las opciones con las que uno debe de contar son muy variadas y tan sólo el diseño de una de estas aplicaciones tomaría mucho más tiempo y son grandes proyectos por su propia cuenta.

Se cumplió el objetivo de desarrollar una aplicación con las tecnologías de shaders disponibles para OpenGL 3.3, así como de manipulación básica de búferes, la geometría que se manda al renderizador es lo más óptima posible, se envían pocos vértices en un sólo arreglo y un arreglo de índices le indica de qué manera dibujar un polígono.

El motor gráfico es un buen ejercicio de diseño de software, ya que en vez de desarrollarlo todo dentro de un sólo archivo, se crea una estructura basada en modelos de la ingeniería de software para facilitar varias tareas y hacerlo en el mismo tiempo de compilación. Se reconoció un alcance razonable para la aplicación y se comprobó todo el conocimiento adquirido durante la carrera.

La aplicación puede correr en varias plataformas sin problemas y sin problemas de framerate, una de las lecciones aprendidas rápidamente en el desarrollo de esta aplicación fue el uso de timers, relojes independientes de la velocidad de procesamiento de una computadora. Es necesario que las animaciones y todas las demás llamadas al dibujo puedan ser realizadas en el mismo tiempo para proveer una experiencia consistente en la aplicación gráfica.

SDL ahorra mucho tiempo dentro del desarrollo de multiplataforma, en comparación a GLUT, por ejemplo, le permite a uno crear su propio sistema de manejo de eventos, en vez de depender del que se encuentra incluido, le permite al usuario entender más cómo funciona a bajo nivel el sistema de interrupciones del sistema operativo y aprender la mejor manera de interpretar sus entradas. Una aplicación gráfica modifica la interpretación de los comandos dependiendo del contexto, por lo que contar con un sistema de eventos customizable se vuelve muy importante para poder darle distintas acciones a distintas formas de presionar una misma tecla.

Retos y trabajo futuro

El desarrollo de la aplicación y motor gráfico deja las puertas abiertas para la creación y expansión de sus subsistemas. La creación de esta aplicación demostró que el diseño e implementación de un sistema complejo para aplicaciones gráficas no es

imposible con la planeación adecuada y el uso de las herramientas adecuadas. Tampoco debe de ser algo caro.

Como trabajo a futuro, está el sistema de física, el cual debe de atravesar el grafo de escena y realizar actualizaciones a los vértices de los objetos dependiendo de distintos cálculos como la gravedad, fricción o colisiones.

La separación en subsistemas realizada permite en un futuro paralelizar todos los procesos de carga, ya que ninguno hace uso exclusivo de recursos durante la fase de carga, todo puede ser agregado a memoria en lugares distintos y organizados.

Aún así, un reto existente es la interdependencia entre clases, ya que varias clases dependen de la existencia y conocimiento de otras para poder compilar, lo cual podría ocasionar problemas cuando estas estructuras sean modificadas. Para expandir aún más el motor gráfico, se podría considerar el eliminar algunas dependencias y en vez de eso hacer algunos objetos globales de los cuales todas las clases tengan conocimiento. Por ejemplo, ninguna clase debería heredar de la clase AudioManager, por lo que sería más sencillo exponer este patrón de diseño a los usuarios si fuese una estructura global única, cuyos métodos sean accesibles para todos, ya que varios objetos necesitan emitir sonido después de que un evento ha sucedido.

Aunque la aplicación fue realizada teniendo en mente la versión de OpenGL 3.0 y GLSL 1.3, el usuario puede fácilmente actualizar a una versión más alta de OpenGL si su GPU lo soporta, agregando así mejoras en memoria y rendimiento, además de soporte para varias nuevas funciones como GLSL 4.0 y los shaders de teselación.

6. Bibliografía

- Referencias Bibliográficas.

1. Alan Thorn, *Cross Platform Game Development*, 1a ed. USA: Jones & Bartlett Learning, 2008.

2. *Cross-platform* [en línea]. Disponible en Web: <https://en.wikipedia.org/wiki/Cross-platform> [Consulta: Abril 12, 2015].

3. Wallace Wang, *Beginning Programming All-in-One Desk Reference for Dummies*, 1a ed. USA: Willey Publishing Inc., 2008.

4. Steven Goodwin, *Cross-Platform Game Programming*, 1a ed. USA: Course Technology PTR, 2005.

5. *Steam Support* [en línea]. Disponible en Web: <https://support.steampowered.com/kb_article.php?ref=9439-QHKN-1308> [Consulta: Febrero 20, 2015].

6. *Humble Bundle* [en línea]. Disponible en Web: <https://en.wikipedia.org/wiki/Humble_Bundle> [Consulta: Febrero 20, 2015].

7. *Gartner says Worldwide PC Shipments Declined 8.3 Percent in Fourth Quarter of 2015* [en línea]. Disponible en Web: <http://www.gartner.com/newsroom/id/3185224> [Consulta: Mayo 3, 2016].

8. Erik Slivka. (Abril 10, 2013). *Apple´s US Shipments Fall 7.5% as Overall PC Market plunges 14% year over year* [en línea]. Disponible en Web: <http://www.macrumors.com/2013/04/10/apples-u-s-mac-shipments-fall-7-5-as-overall-pc-market-plunges-14-year-over-year/> [Consulta: Abril 19, 2015].

9. *Market Share Reports* [en línea]. Disponible en Web: <http://marketshare.hitslink.com/> [Consulta: Mayo 7, 2016].

10. *Cross Platform Development* [en línea]. Disponible en Web: <http://www.tortuga.com.au/products/ozibug/www/cross_platform_devel.html> [Consulta: Noviembre 17, 2014].

11. *A first look at the Chinese operating system the government wants to replace Windows* [en línea]. Disponible en Web: <http://qz.com/505383/a-first-look-at-the-chinese-operating-system-the-government-wants-to-replace-windows/> [Consulta: Mayo 7, 2016].

12. *Bringing Windows 10 to Public Sector Customers in China* [en línea]. Disponible en Web: <https://blogs.windows.com/windowsexperience/2015/12/16/bringing-windows-10-to-public-sector-customers-in-china/> [Consulta: Mayo 7, 2016].

13. John Gaudiosi. (Julio 18, 2012). *New reports forecast global video game industry will reach $82 Billion by 2017* [en línea]. Disponible en Web: <http://www.forbes.com/sites/johngaudiosi/2012/07/18/new-reports-forecasts-global-video-game-industry-will-reach-82-billion-by-2017/> [Consulta: Abril 20, 2015].

14. Rob Williams. (Septiembre 21, 2011). *NVIDIA talks PC gaming trends* [en línea]. Disponible en Web: <http://techgage.com/article/nvidia_talks_pc_gaming_trends/> [Consulta: Abril 20, 2015].

15. *Steam Hardware and Software Survey* [en línea]. Disponible en Web: <http://store.steampowered.com/hwsurvey> [Consulta: Mayo 4, 2016].

16. Emil Protalinski. (Abril 29, 2016). *Mozilla will retire Firefox support for OS X 10.6, 10.7 and 10.8 in August 2016* [en línea]. Disponible en Web: <http://venturebeat.com/2016/04/29/mozilla-will-retire-firefox-support-for-os-x-10-6-10-7-and-10-8-in-august-2016/> [Consulta: Mayo 7, 2016].

17. *Why Cross platform development* [en línea]. Disponible en Web: <http://cross-platformdevelopment.com/why-cross-platform/> [Consulta: Octubre 12, 2014].

18. Lakshmi Narasimhan. (Mayo 15, 2009). *Challenges of cross platform game development* [en línea]. Disponible en Web: <http://www.nasscom.org/sites/default/files/upload/Intel-NASSCOM-CrossPlatformChallenges_final.pdf> [Consulta: Abril 20, 2015].

19. Paul Martz, *OpenGL Distilled*, 1a ed. USA: Addison-Wesley Professional, 2006.

20. *Learn about Java Technology* [en línea]. Disponible en Web: <https://www.java.com/en/about/> [Consulta: Abril 21, 2015].

21. Poornachandra Sarang, *Java Programming*, 1a ed. USA: McGraw-Hill, 2012.

22. *Java (programming language)* [en línea]. Disponible en Web: <https://en.wikipedia.org/wiki/Java_(programming_language)> [Consulta: Abril 21, 2015].

23. Ashok Kamthane, *Programming in C*, 2a ed. India: Pearson Education, 2011.

24. Ivor Horton, *Beginning C*, 5a ed. USA: Apress, 2013.

25. *C++* [en línea]. Disponible en Web: <https://en.wikipedia.org/wiki/C%2B%2B> [Consulta: Abril 22, 2015].

26. Matt Bishop. (Agosto 20, 2011). *Portability in C -- A Case Study* [en línea]. Disponible en Web: <http://nob.cs.ucdavis.edu/bishop/papers/1986-cjourv1n4/> [Consulta: Abril 22, 2015].

27. *About SDL* [en línea]. Disponible en Web: <http://www.libsdl.org/> [Consulta: Mayo 4, 2016].

28. *Create the games you love with Unity* [en línea]. Disponible en Web: <http://unity3d.com/unity/> [Consulta: Mayo 4, 2016].

29. *OGRE - About* [en línea]. Disponible en Web: <http://www.ogre3d.org/about> [Consulta: Abril 23, 2015].

30. *Microsoft XNA* [en línea]. Disponible en Web: <https://en.wikipedia.org/wiki/Microsoft_XNA> [Consulta: Mayo 4, 2016].

31. *GIMP* [en línea]. Disponible en Web: <https://en.wikipedia.org/wiki/GIMP> [Consulta: Mayo 4, 2016].

32. *Blender (software)* [en línea]. Disponible en Web: <https://en.wikipedia.org/wiki/Blender_(software)> [Consulta: Mayo 4, 2016].

33. Git -- fast - version - control [en línea]. Disponible en Web: <https://git-scm.com/about> [Consulta: Mayo 4, 2016].

34. *Open Graphics Library (OpenGL)* [en línea]. Disponible en Web: <https://en.wikipedia.org/wiki/OpenGL> [Consulta: Mayo 8, 2016].

35. *Evolutionary Prototyping* [en línea]. Disponible en Web: <http://www.teach-ict.com/as_a2_ict_new/ocr/A2_G063/331_systems_cycle/prototyping_RAD/miniweb/pg3.htm> [Consulta: Abril 25, 2015].

36. *Software Prototyping* [en línea]. Disponible en Web: <http://en.wikipedia.org/wiki/Software_prototyping> [Consulta: Abril 25, 2015].

37. *Model-view-controller* [en línea]. Disponible en Web: <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller> [Consulta: Mayo 2, 2015].

38. *Supported Graphics APIs and Features* [en línea]. Disponible en Web: <http://www.intel.com/support/graphics/sb/CS-033757.htm> [Consulta: Mayo 7, 2015].

39. *OpenGL and multithreading* [en línea]. Disponible en Web: <https://www.opengl.org/wiki/OpenGL_and_multithreading> [Consulta: Febrero 2, 2015].

40. Cookiecups. (Agosto 25, 2006). *What is the XNA Framework* [en línea]. Disponible en Web: <http://blogs.msdn.com/b/xna/archive/2006/08/25/724607.aspx> [Consulta: Mayo 7, 2015].

41. Mike McShaffry, *Game Coding Complete*, 4a ed. USA: Course Technology Cengage Learning, 2012.

42. Nicholas Haemel, Graham Sellers, Richard S. Wright, *OpenGL SuperBible: Comprehensive Tutorial and Reference*, 6a ed. USA: Addison-Wesley Professional, 2013.

- Otra bibliografía consultada.

Aaftab Munshi, Dan Ginsburg, Dave Shreiner, *OpenGL ES 2.0 Programming Guide*, 1a ed. USA: Addison-Wesley Professional, 2008.

Benjamin Lipchak, Richard S. Wright, Nicholas Haemel, *OpenGL SuperBible: Comprehensive Tutorial and Reference*, 4a ed. USA: Addison-Wesley Professional, 2007.

Dave Shreiner, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*, 7a ed. USA: Addison-Wesley Professional, 2009.

David H. Eberly, *3D Game Engine Architecture*, 1a ed. USA: Focal Press, 2004.

IEEE Citation Reference [en línea]. Disponible en Web: <http://www.ieee.org/documents/ieeecitationref.pdf> [Consulta: Febrero 10, 2015].

IEEE Citation Style Guide [en línea]. Disponible en Web: <http://www.ijssst.info/info/IEEE-Citation-StyleGuide.pdf> [Consulta: Febrero 10, 2015].

List of Nvidia graphics processing units [en línea]. Disponible en Web: <https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units> [Consulta: Mayo 8, 2016].

List of AMD graphics processing units [en línea]. Disponible en Web: <https://en.wikipedia.org/wiki/List_of_AMD_graphics_processing_units> [Consulta: Mayo 8, 2016].

Michelle Menard, *Game Development with Unity*, 1a ed. USA: Course Technology PTR, 2011.

Randi J. Rost, Bill Licea-Kane, Dan Ginsburg, *OpenGL® Shading Language*, 3a ed. USA: Addison-Wesley Professional, 2009.

Robert P. Kuehne, J. D. Sullivan, *OpenGL® Programming on Mac OS® X: Architecture, Performance, and Integration*, 1a ed. USA: Addison-Wesley Professional, 2007.

TESIUNAM - Tesis del Sistema Bibliotecario de la UNAM [en línea]. Disponible en Web: <http://bcct.unam.mx/web/tesiunam.htm> [Consulta: Abril 20, 2014].

7. Apéndice

Glosario

AGL: Apple Graphics Library es la API de Apple Inc. para uso de gráficos 3D de OpenGL dentro de ventanas Carbon, se encuentra por encima del núcleo de OpenGL para Apple (CGL).

API: La Interfaz de Programación de Aplicaciones (del inglés Application Programming Interface) es un protocolo diseñado para ser usado como una interfaz por componentes de software para comunicarse entre sí. Una API es una biblioteca que puede contener especificaciones para rutinas, estructuras de datos, objetos clase, y variables.

La programación orientada a componentes (que también es llamada *basada en componentes*) es una rama de la ingeniería de software con énfasis en la descomposición de sistemas ya conformados en componentes funcionales o lógicos con interfaces bien definidas usadas para la comunicación entre componentes.

ARM: La arquitectura ARM describe una familia de microprocesadores basada en la arquitectura RISC diseñada por la compañía británica ARM Holdings. El nombre era un acrónimo de Acorn RISC Machine, pero fue cambiado por Advanced RISC Machine. Hasta el año del 2013, es la arquitectura de conjunto de instrucciones de 32 bits más usada en todo el mundo, por su uso en teléfonos móviles y televisores digitales.

Bytecode: también conocido como p-code (código portátil), es una forma de conjunto de instrucciones diseñado para la ejecución eficiente por un programa intérprete. A diferencia del código fuente legible por el humano, los bytecodes son códigos numéricos compactos, constantes y referencias (direcciones numéricas) que codifican el resultado del análisis sintáctico y semántico de elementos como el tipo, alcance y niveles de anidamiento de los objetos de un programa. Por ello permiten un mejor rendimiento que la interpretación directa del código fuente. El bytecode no es el código máquina de ninguna computadora en particular, y puede por tanto ser portable entre diferentes arquitecturas.

Carbon: Es una de las APIs basadas en C de Apple Inc. para el sistema operativo de Macintosh, provee compatibilidad con programas para los sistemas operativos ahora obsoletos MAC OS 8 y 9.

Endianness: La endianness se refiere a el formato en que los datos de más de un byte se almacena en memoria. El formato big-endian, consiste en representar el byte más significativo primero (de izquierda a derecha), así el valor hexadecimal 0x3A2B1C se codificará en memoria como {3A, 2B, 1C}. En el formato little-endian, el mismo valor se codificará como {1C, 2B, 3A}. Algunas arquitecturas de microprocesador pueden trabajar con ambos formatos, y son referidas como sistemas middle-endian.

Fin de línea: En informática, el marcador de fin de línea (EOL, end-of-line) o nueva línea, es un carácter especial o secuencia de caracteres que indican el fin de una línea de texto. La codificación del carácter de fin de línea no es la misma en todas las arquitecturas ni sistemas operativos, cosa que puede dar problemas cuando se intercambian datos entre ordenadores.

IDE: Un entorno de desarrollo integrado, llamado también IDE (Integrated Development Environment), es un programa compuesto por un conjunto de herramientas de programación. Puede dedicarse exclusivamente a un solo lenguaje de programación o bien puede utilizarse para varios. Un IDE consiste en un editor de código, un compilador, un depurador y un constructor de interfaz gráfica (GUI).

JIT: La compilación en tiempo de ejecución, también conocida por sus siglas en inglés JIT (just in time), o traducción dinámica es una técnica para mejorar el rendimiento de sistemas de programación que compilan a base de bytecode, consiste en traducir el bytecode a código máquina nativo en tiempo de ejecución.

La ventajas principal de la compilación JIT es que la mayoría del trabajo pesado de procesar el código fuente original y realizar optimizaciones básicas se realiza en el momento de compilar a bytecode, así, la compilación a código máquina del programa resulta mucho más rápida que partiendo del código fuente.

NURBS: Un B-spline racional no uniforme (Non Uniform Rational B-Spline) es un modelo matemático muy utilizado en la computación gráfica para generar y representar curvas y superficies. Las curvas y superficies NURBS son útiles por varias razones: Son invariantes bajo transformaciones afines, así como de perspectiva; proporcionan flexibilidad para diseñar una gran variedad de figuras, y reducen el consumo de memoria al almacenar figuras.

Ouya: Ouya es una videoconsola de código abierto que funciona con el sistema operativo Android, fue lanzada gracias al apoyo de donadores de Kickstarter en 2013. Todos los sistemas pueden ser empleados como kits de desarrollo, permitiendo que cualquier propietario de Ouya pueda ser también un desarrollador, sin necesidad de pagar por derechos de licencia. Debido a su éxito Ouya fue comprada por la compañía de productos para gamers Razer en el año 2015.

Pipeline: La segmentación pipeline es un método por el cual se consigue aumentar el rendimiento de algunos sistemas computacionales. Es aplicado, sobre todo, en microprocesadores y librerías gráficas.

RISC: Computadora con Conjunto de Instrucciones Reducidas (Reduced Instruction Set Computer) es un tipo de diseño de CPU generalmente utilizado en microprocesadores o microcontroladores con las siguientes características fundamentales: Instrucciones de tamaño fijo y presentadas en un reducido número de formatos, sólo las instrucciones de carga y almacenamiento acceden a la memoria de datos. Además estos procesadores suelen disponer de muchos registros de propósito general.

Singleton: El patrón singleton es un patrón de diseño, su intención es garantizar que una clase tenga sólo una instancia y proporcionar un punto de acceso global a ella. Un singleton se implementa creando en la clase un método que crea una instancia del objeto

sólo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente, se regula el alcance del constructor.

WGL: WGL o Wiggle es la interfaz del sistema de ventanas para la implementación de OpenGL en Microsoft Windows. WGL es análoga a GLX, que es la interfaz de OpenGL para los sistemas X Windows System o X11, así como CGL que es la interfaz de openGL para Mac OS X.

x86: El término x86 denota una familia de conjuntos de instrucciones basados en el microprocesador Intel 8086. x86-64 o x64 es una extensión del conjunto de instrucciones x86, utilizado en la microarquitectura de CPU. Contempla mejoras adicionales, como duplicar el número y el tamaño de los registros de uso general.

Tutorial

Requisitos mínimos

- Una tarjeta de video o procesador que soporte OpenGL 3.0 en el Sistema Operativo de elección (Segunda generación de procesadores Intel Core o tarjetas de vídeo superiores a NVIDIA GeForce 8xxx series, y AMD Radeon HD 2xxx series).
- Tener un IDE instalado, para el caso de Windows empleamos Visual Studio por lo menos en su versión 2010, una versión gratuita puede ser descargada en: https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx
-

Figura 7.1 Versión gratuita de Visual Studio.

Un requisito específico de la aplicación debido a su diseño, es que las texturas utilizadas deben de ser formato TGA, a pesar de que se soportan texturas no potencias de dos es muy recomendable utilizar texturas cuadradas (256x256, 512x512, etc) y potencias de dos para un cargado de texturas más óptimo y rápido. De igual manera, para los modelos se deben usar mallas poligonales basadas en triángulos pues esto nos simplifica la carga de modelos en OpenGL. El cargador de modelos tiene soporte para los formatos más comunes como son: Autodesk (.fbx), Blender (.blend), 3ds Max (.3ds), Wavefront Object (.obj), Collada (.dae) y LightWave (.lwo).

Donde obtener el código y cómo empezar a utilizarlo

En caso de tener duda de la versión de OpenGL que soporta su computadora, puede descargar la herramienta llamada OpenGL Extension Viewer, la cual le permite determinar la última versión de OpenGL soportada. La herramienta puede ser descargada de la página http://www.realtech-vr.com/glview/download.php sin costo alguno,

al descargarla y ejecutarla nos presentara una pantalla en la cual nos indica las especificaciones de nuestro sistema y la versión más nueva de OpenGL soportada.

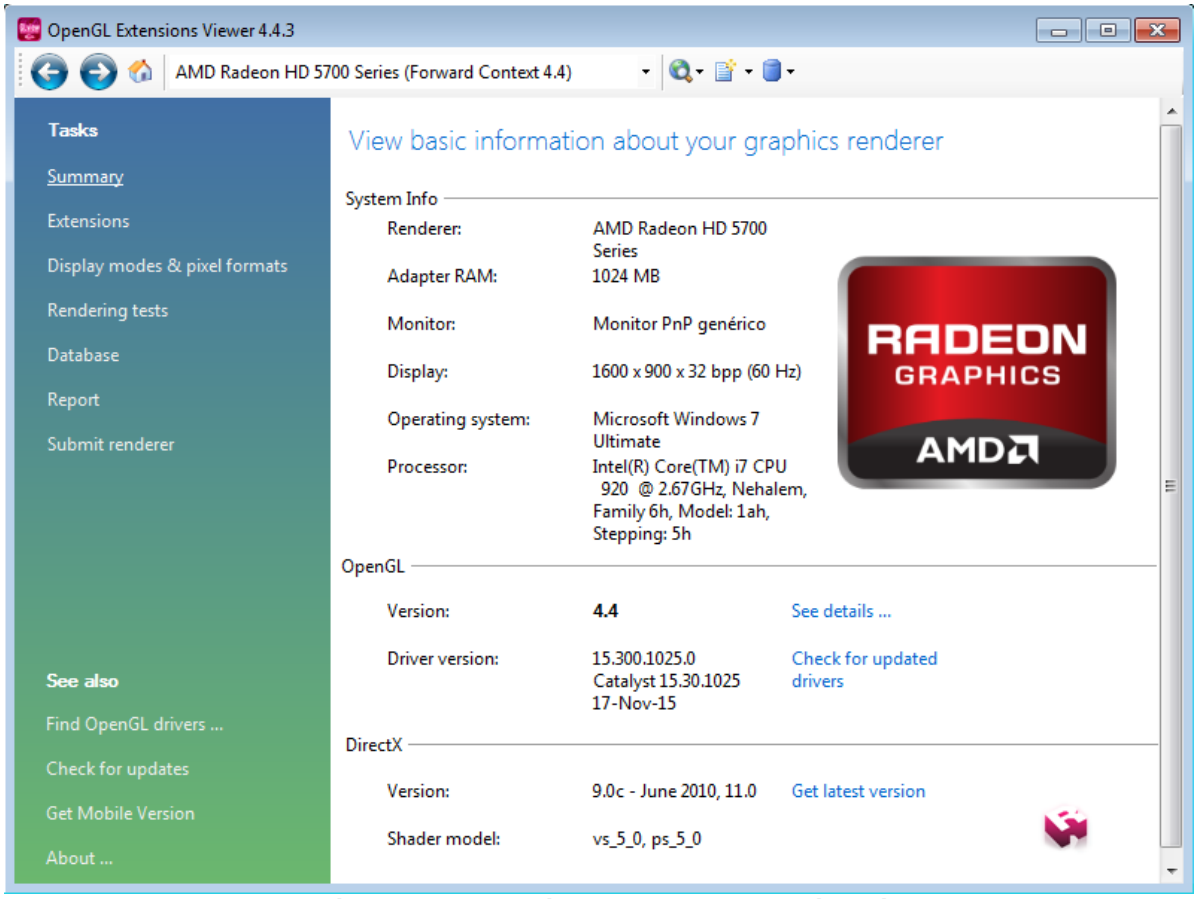


Figura 7.2 Herramienta OpenGL Extension Viewer.

OpenGL Extension Viewer incluso nos puede dar más detalles acerca de las funciones de OpenGL soportadas. En la figura 7.3 podemos observar el caso de una computadora de prueba en la cual se soporta el 100% de OpenGL hasta la versión 4.4, para la versión 4.5 nos indica que GLSL 4.5 no es soportado y por lo tanto no podemos hacer uso de OpenGL 4.5 en esta máquina.

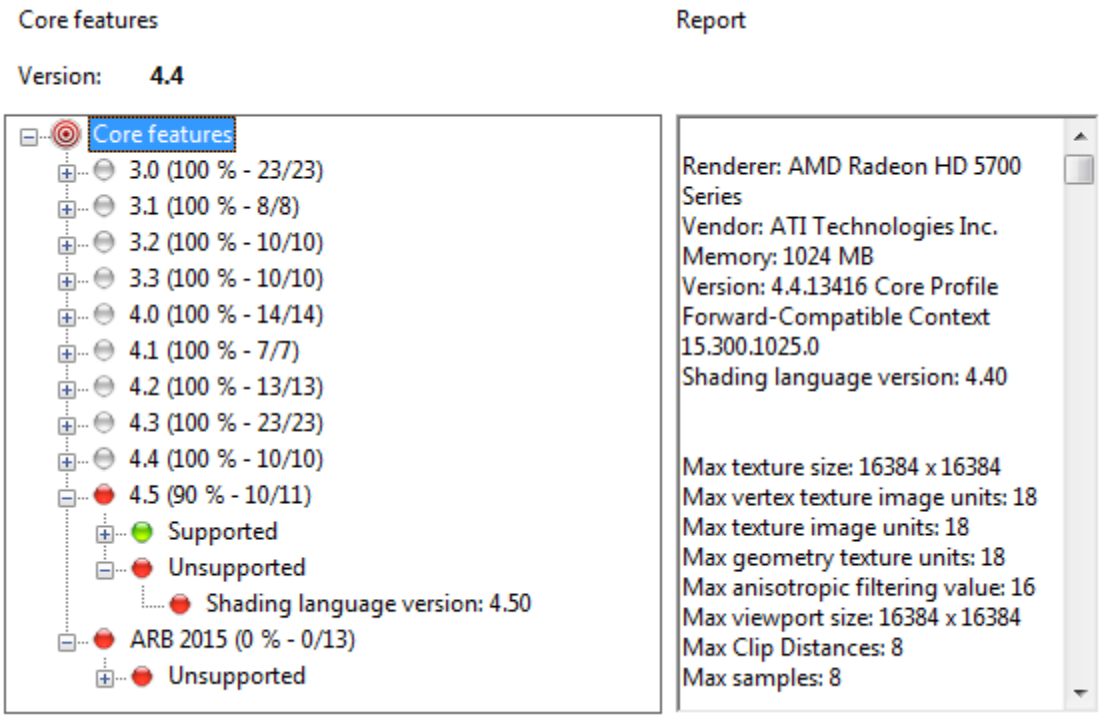


Figura 7.3 Vista detallada de OpenGL Extension Viewer.

Después de verificar que la computadora cumple los requisitos mínimos, el usuario está listo para utilizar el motor gráfico para desarrollar una aplicación rápida en la plataforma.

Para hacer uso de este trabajo en el sistema operativo Windows, pueden seguirse las siguientes instrucciones:

Primero, se comienza por crear un nuevo proyecto en Visual Studio, dentro del cual se construirá una aplicación gráfica usando como librería el proyecto que desarrollado en la tesis. Después de elegir la opción de crear un nuevo proyecto se debe elegir la opción de Win32 Project como el tipo de proyecto y le asignarle un nombre y ubicación al proyecto:

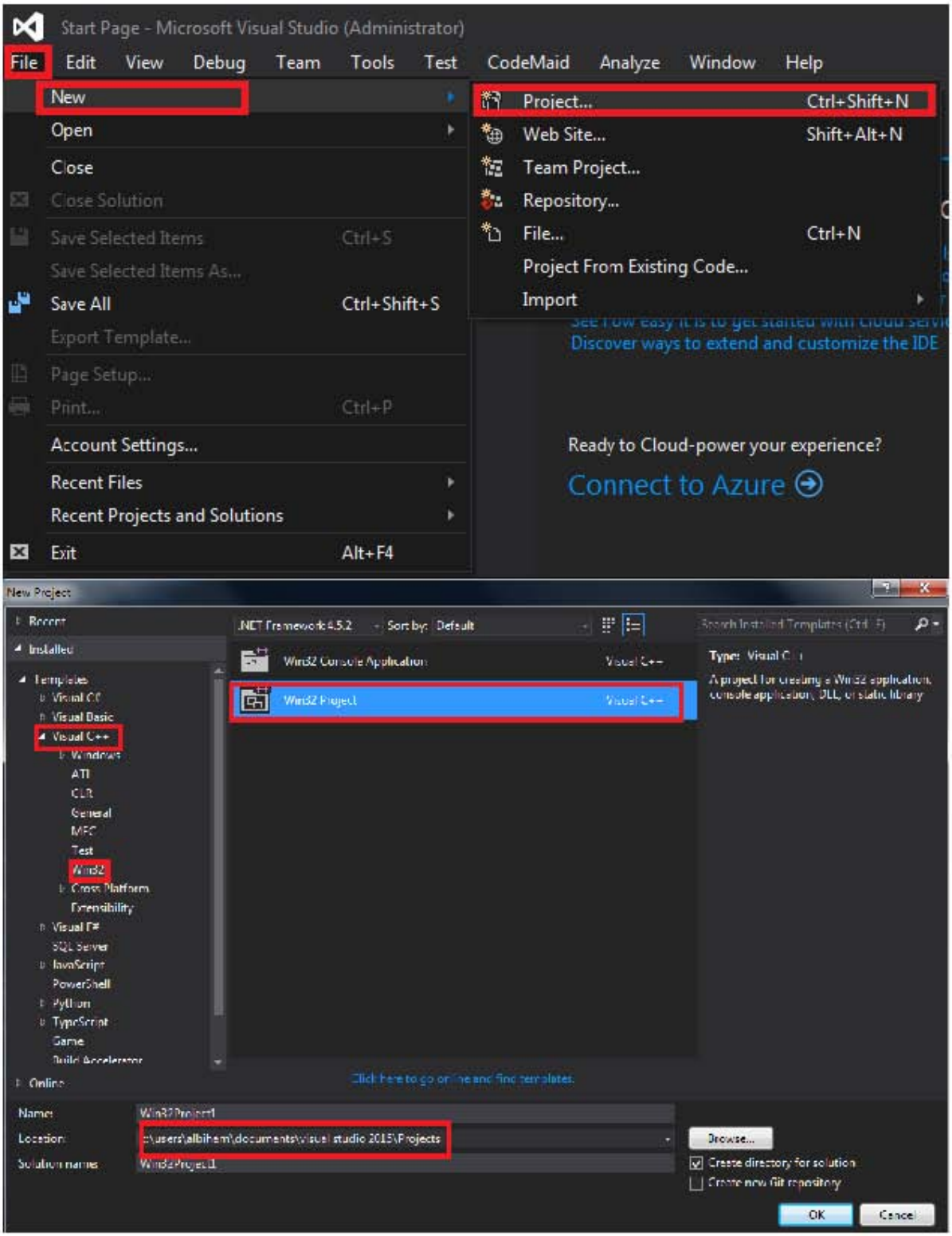


Figura 7.4 Creación de un nuevo proyecto en Visual Studio.

En la siguiente pantalla elegir la opción de siguiente y abrirá otra ventana en la que debemos elegir crear una aplicación de tipo *Console Application* y en opciones adicionales seleccionar *Empty project* :

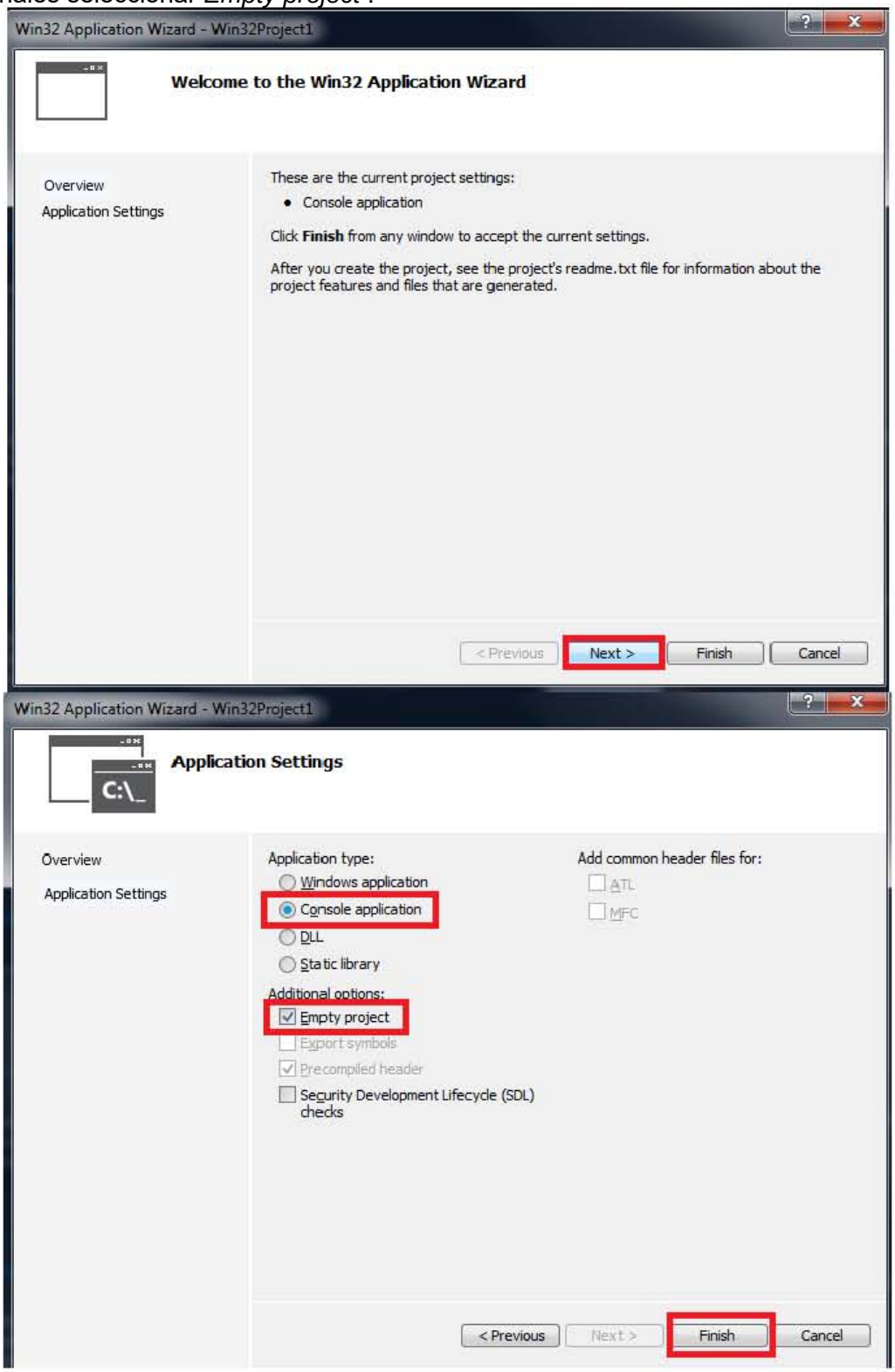


Figura 7.5 Configuraciones adicionales para el proyecto.

Después se debe agregar un nuevo item de tipo C++ (.cpp) dentro del fólder de código fuente:

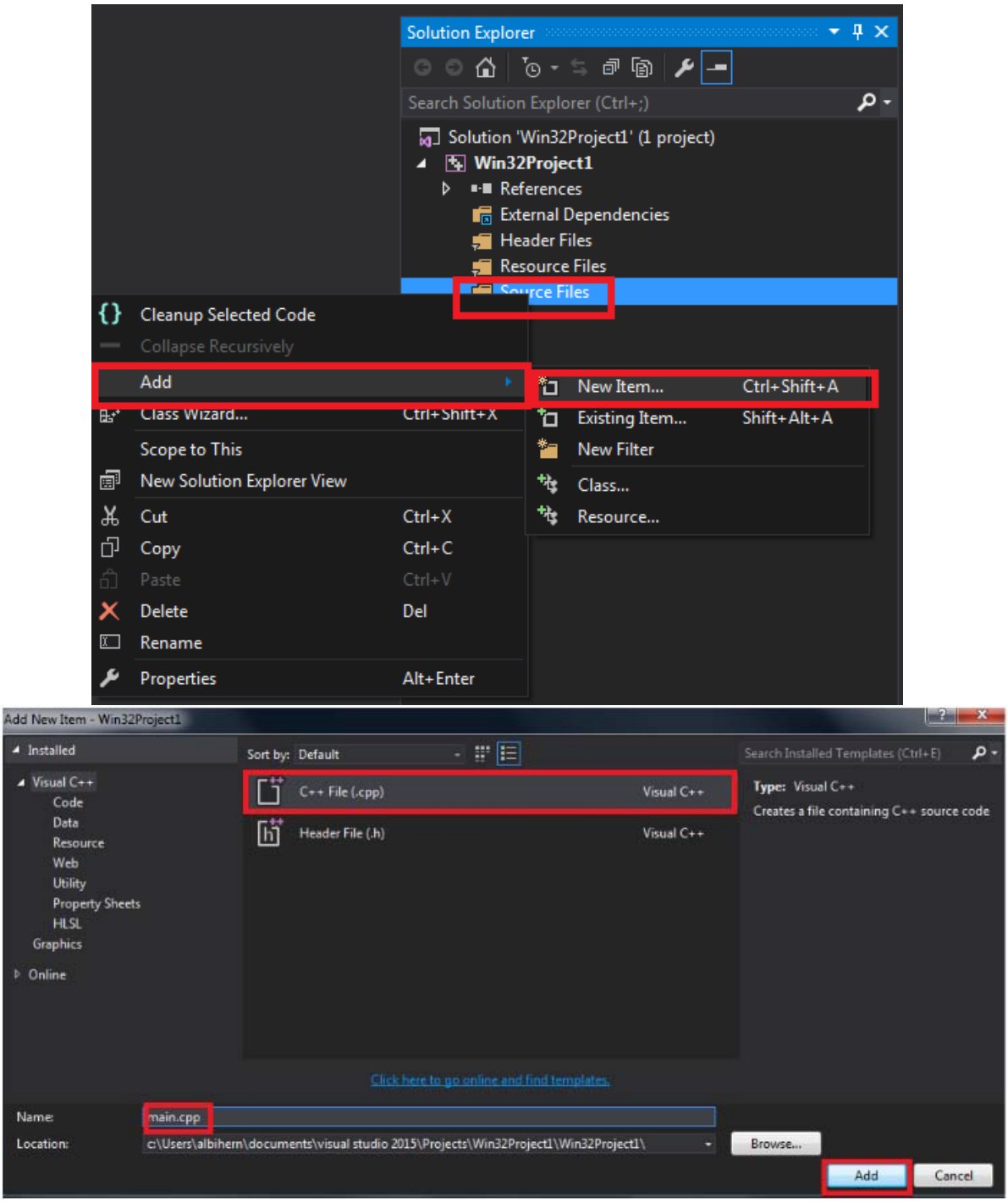


Figura 7.6 Adición de un nuevo archivo de tipo .cpp al proyecto.

Para tener acceso al código fuente debe ser descargado y descomprimido del repositorio en línea https://github.com/aru/superhappyrocket . Al abrir el vínculo aparecerá una ventana como la siguiente, en la que debemos elegir la opción *Download ZIP*:

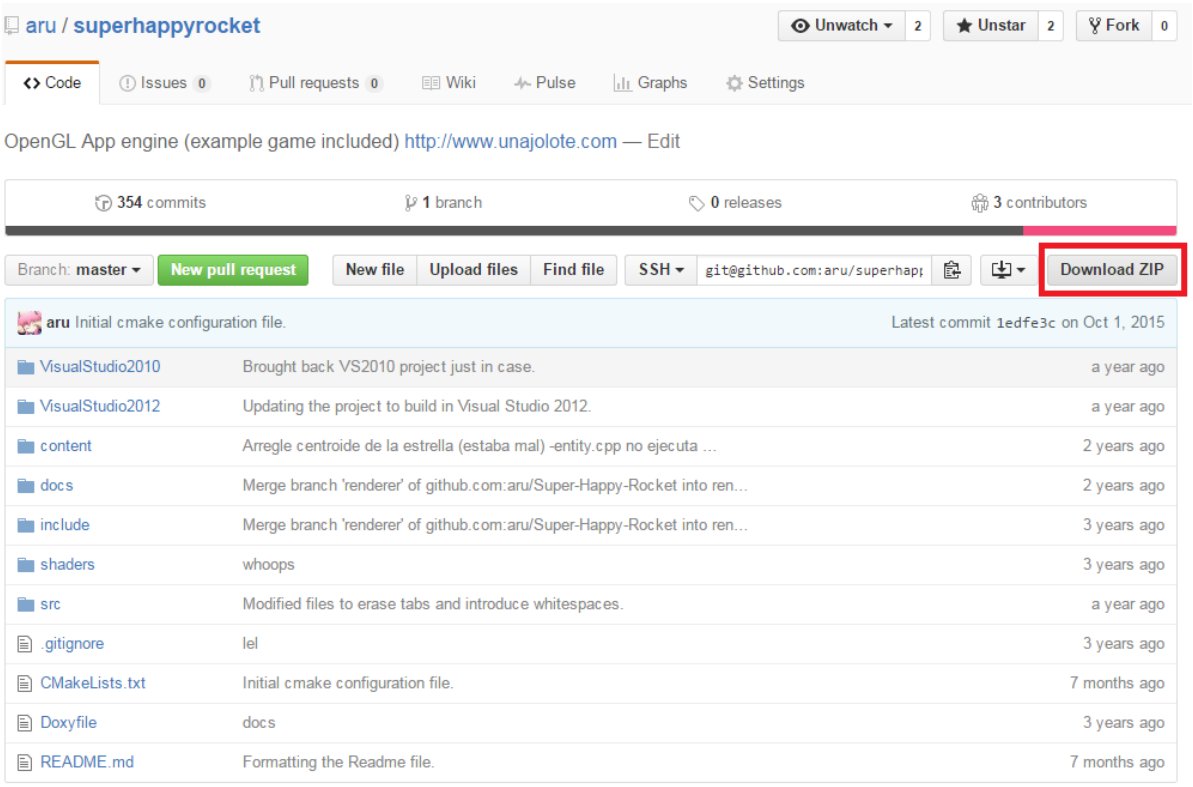


Figura 7.7 Descarga del código por medio de Git.

Una vez descargado, abrir el folder VisualStudio2012/SuperHappyRocket/ dentro del cual ya se tiene un proyecto listo para compilar, y abrir el archivo SuperHappyRocket.sln:

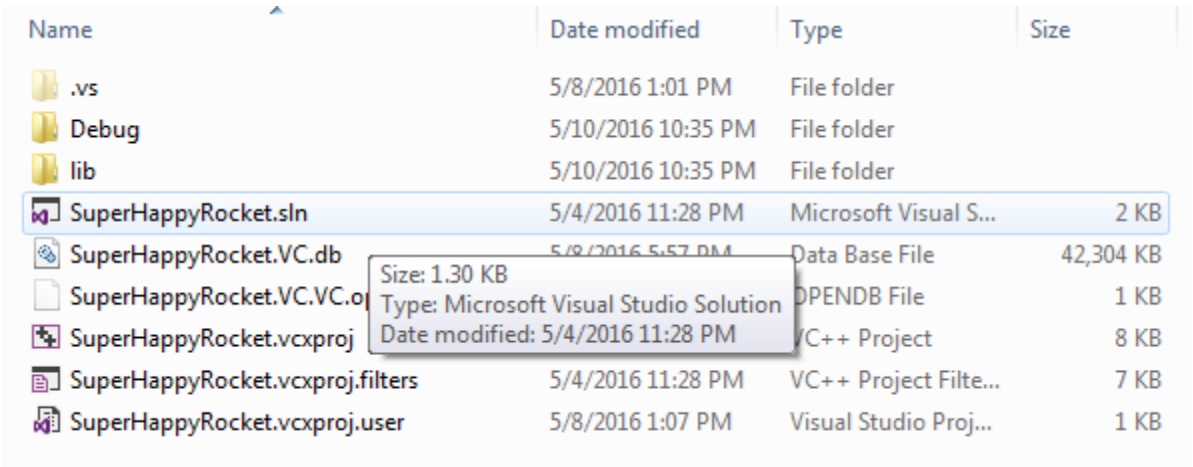


Figura 7.8 Ubicación del archivo SuperHappyRocket.sln.

Una vez dentro del proyecto, construirlo, no es necesario correr la aplicación ejemplo, la configuración para crear una librería está predeterminada.

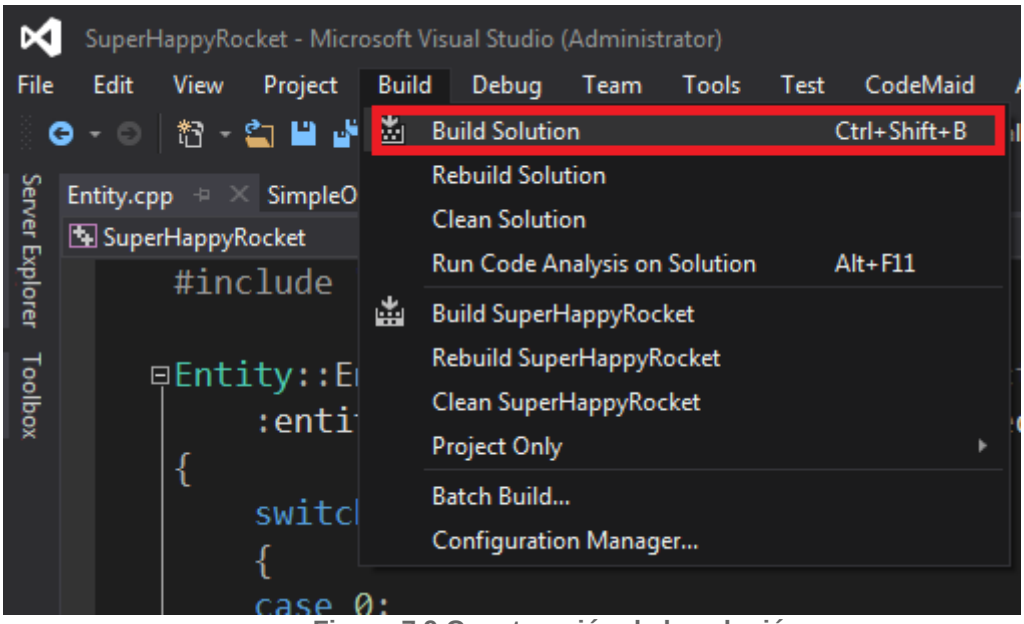


Figura 7.9 Construcción de la solución

Después se deben localizar los directorios donde se realizó la compilación de la solución (por defecto lib/) y copiar la carpeta lib al fólder en donde se creó el proyecto anterior:

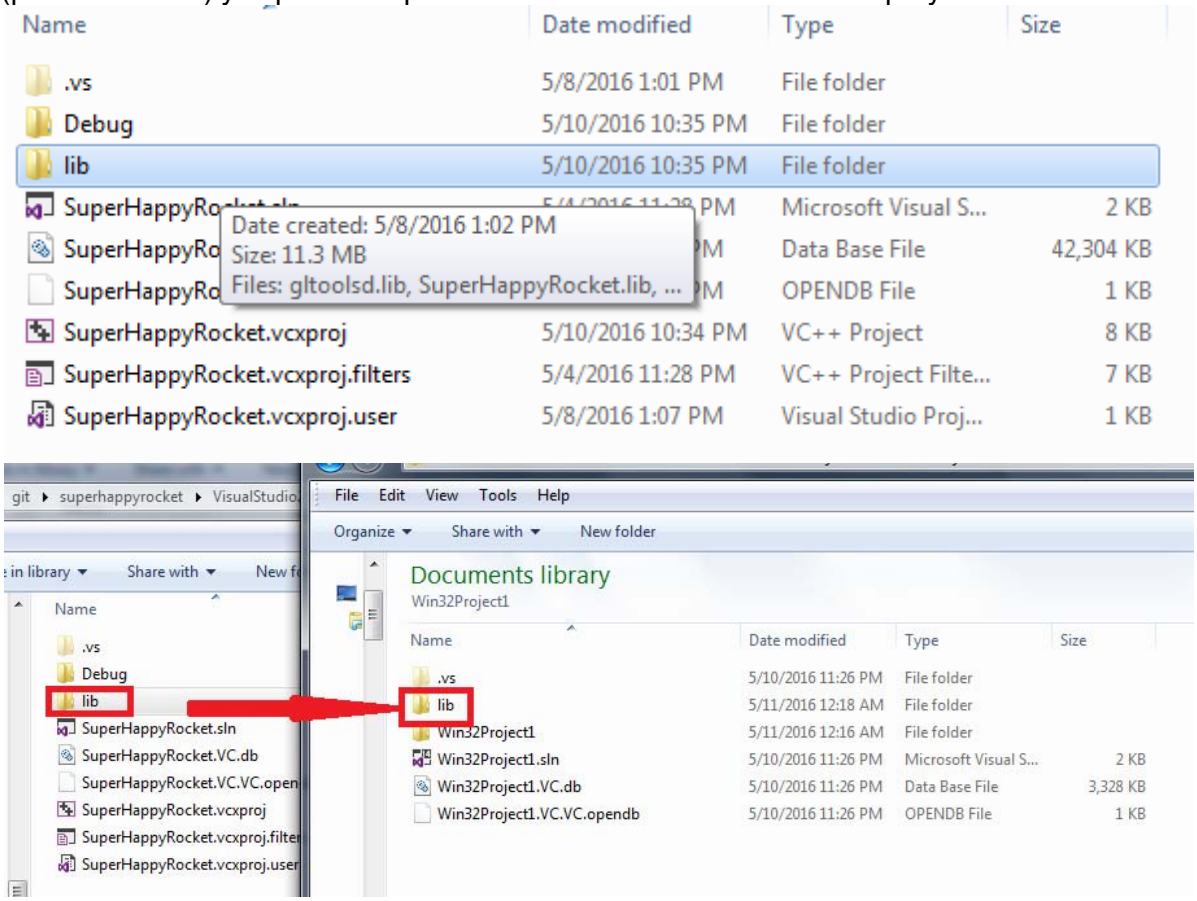


Figura 7.10 Ubicación y copiado de la carpeta lib a el nuevo proyecto.

En el proyecto de Visual Studio, se debe agregar estos directorios correspondientes a SDL, assimp, y GLTools, para su correcta compilación:

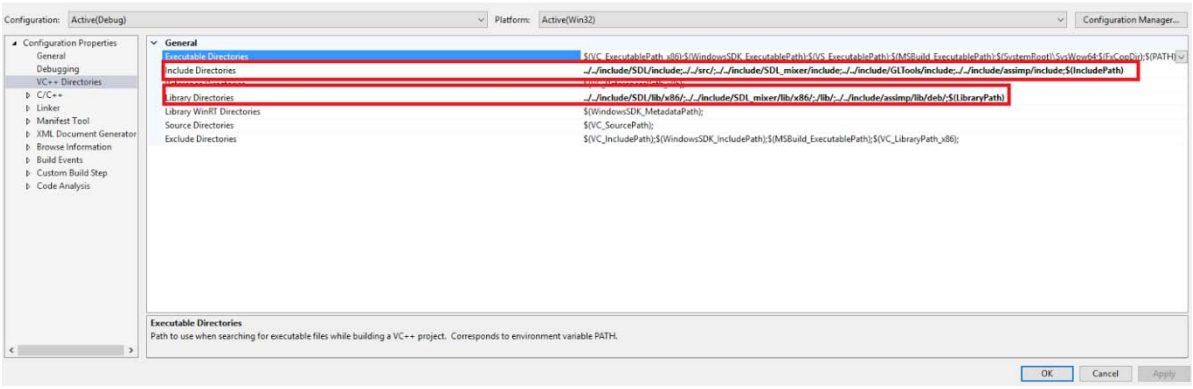


Figura 7.11 Directorios necesarios para el nuevo proyecto.

Sustituir las rutas relativas por la ruta absoluta donde se descargó y compiló la librería.

Agregar las librerías a utilizar en el proyecto de Visual Studio:

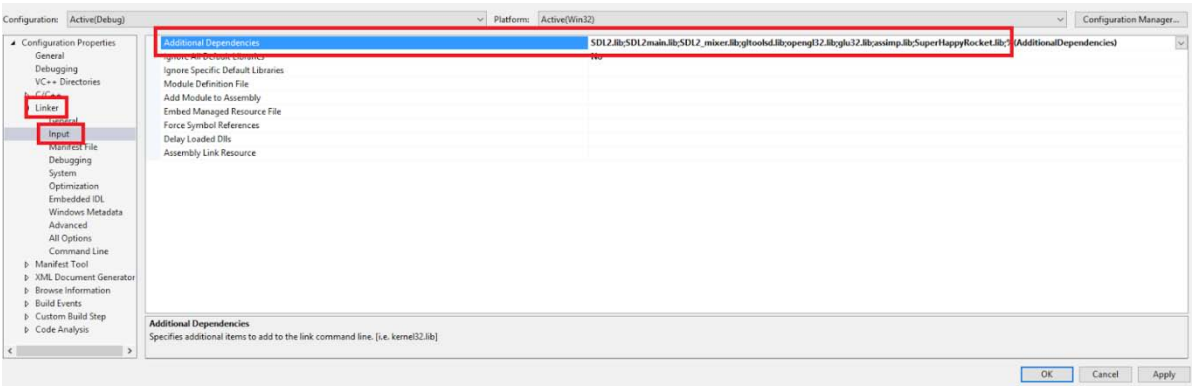


Figura 7.12 Dependencias adicionales para el proyecto.

Se debe copiar además el contenido de la carpeta include/dlls a este mismo directorio en una carpeta llamada Debug/, pues contiene archivos necesarios para que la aplicación funcione correctamente:

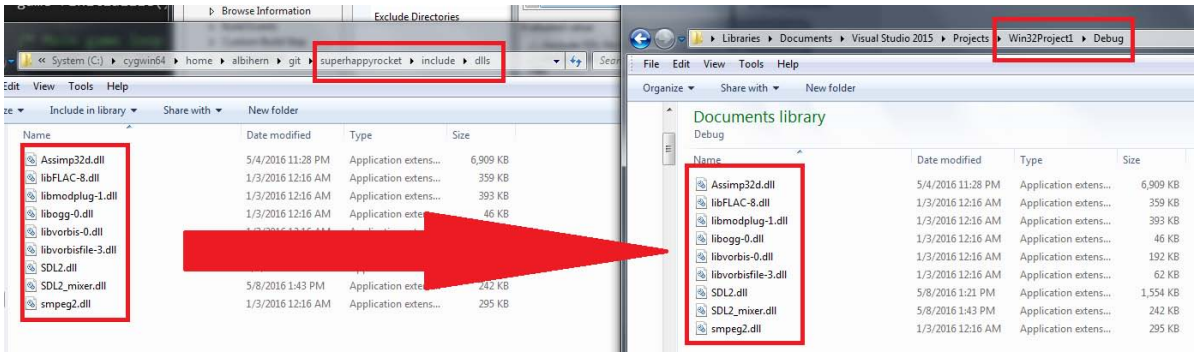


Figura 7.13 Copiado del contenido de la carpeta include/dlls a una carpeta llamada Debug/ en el nuevo proyecto.

Ahora se puede proseguir a crear un archivo de configuración para la aplicación, el cual se puede llamar context.txt, o el nombre que sea, pero tomar nota de él e introducir los siguientes valores dentro de él:

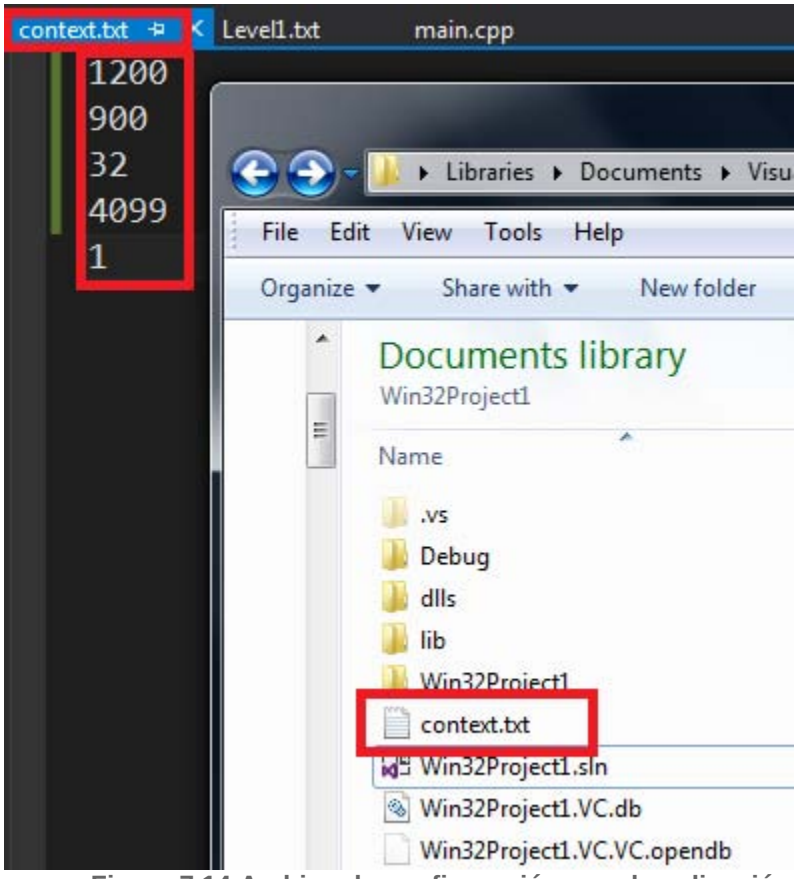


Figura 7.14 Archivo de configuración para la aplicación.

Estos valores corresponden respectivamente a:

- El largo de la ventana.
- El ancho de la ventana.
- Los bits de color por pixel.
- El modo de video (4099 es la bandera de OpenGL para SDL)

Después se debe de copiar el modelo de nuestro personaje principal *rocket249.3ds* del folder content/models/ al folder del nuevo proyecto que se acaba de crear:

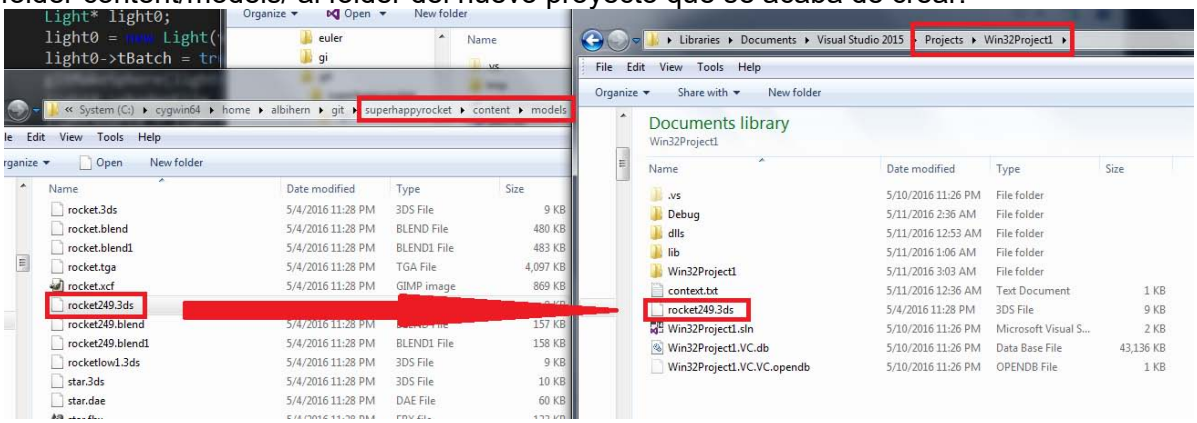


Figura 7.15 Copiado del modelo del cohete a el nuevo proyecto.

En el archivo creado del proyecto introducir el siguiente código base ejemplo:

```
#include "SDLApp.h"
#include "BaseLevel.h"

class UserLevel : public BaseLevel {
public:
  UserLevel(Context* ctx) :BaseLevel(ctx) {}
  ~UserLevel() {};
  const int Initialize() {
    ctxt->camera->cameraFrame.MoveForward(-15.0f);
    return SHR_SUCCESS;
  };
  const int LoadContent() {
    /* Add our Lights */
    static GLfloat vWhite[] = { 1.0f, 1.0f, 1.0f, 1.0f };
    static GLfloat vLightPos[] = { 0.0f, 1.0f, 0.0f, 1.0f };
    Light* light0;
    light0 = new Light(vLightPos, vWhite);
    light0->tBatch = true;
    gltMakeSphere(light0->triBatch, 1.0f, 15, 15);
    light0->shaderFile = GLT_SHADER_FLAT;
    lightSource = light0;
    /* Add an assimp model */
    rocket = new assimpMesh("./../rocket249.3ds", ctxt);
    rocket->frame.SetOrigin(0.0f, 0.0f, 12.0f);
    rocket->frame.RotateLocalZ(float(m3dDegToRad(90.0f)));
    rocket->frame.RotateLocalX(float(m3dDegToRad(-90.0f)));
    actors.push_back((SimpleObject*)rocket);
    return SHR_SUCCESS;
  }
  const int Update(Uint32 gameTime) { return SHR_SUCCESS; };
  void HandleInput(InputManager* input, Uint32 gameTime) { };
  /* Level content */
  assimpMesh* rocket;
};
```

```
int main(int argc, char* args[]) {
  /* Load the config file and create a context out of it */
  Context* context;
  context = new Context("./../context.txt");
  /* Create an SDL App out of this context */
  SDLApp* game;
  game = new SDLApp(context);
  /* Create a new level for our App */
  UserLevel level(context);
  /* Initialize this SDL App */
  game->Initialize(&level);
  /* Main game loop */
  while (context->quit == false) {
    /* Update the time */
    /* Handle Input */
    context->input->handleKeys();
    /* Draw the SDL App */
    game->Draw();
    /* Play Music */
    /* Do our postReDisplay */
  }
  /* Delete our 2 things */
  delete game;
  delete context;
  /* Return success */
  return 0;
}
```

Figura 7.16 Código base ejemplo para el nuevo proyecto.

Una vez copiado el código debe ser compilado para ver el resultado del nuevo proyecto recién creado a partir del repositorio disponible en Git:

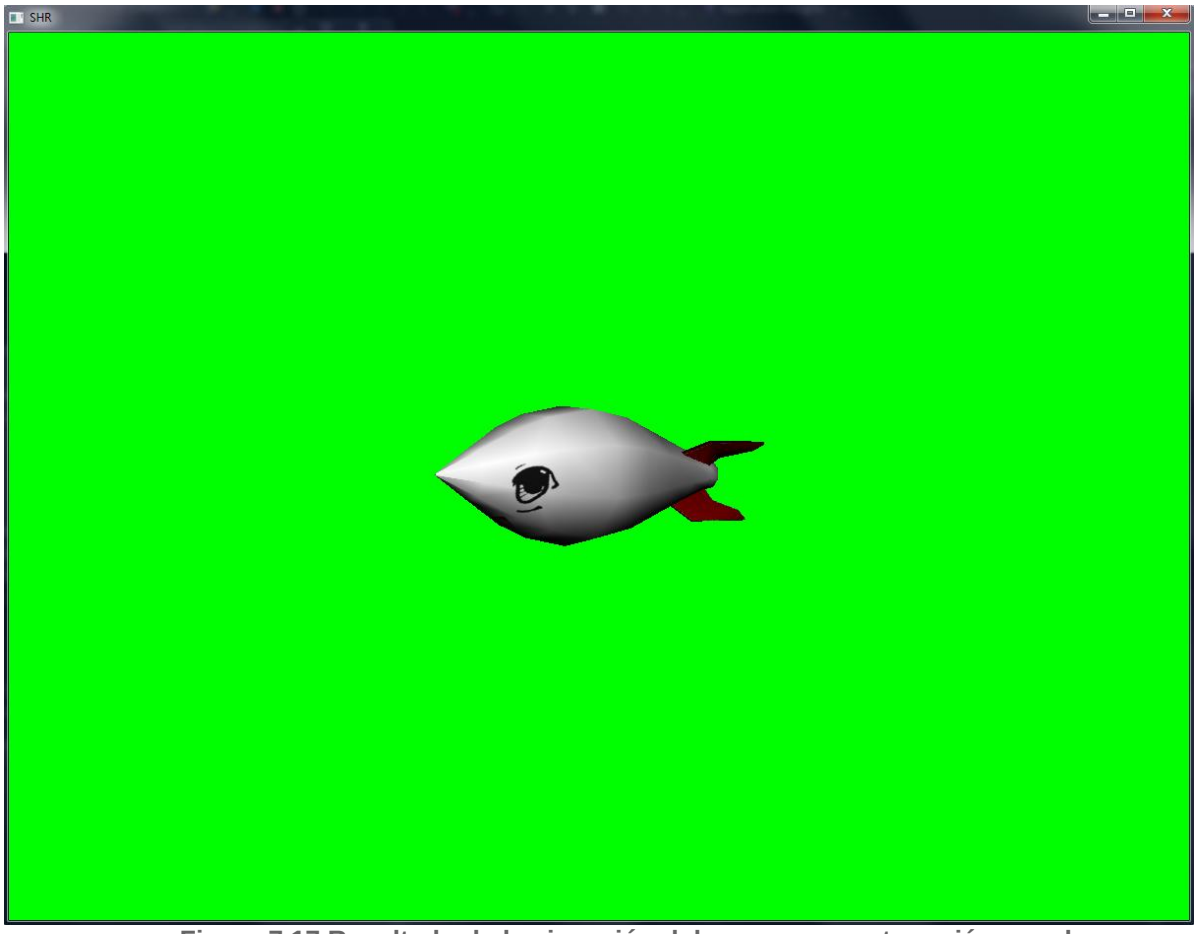


Figura 7.17 Resultado de la ejecución del nuevo proyecto recién creado.

Estas mismas instrucciones pueden seguirse para Linux o Mac mediante el uso del IDE Code:Blocks, dentro del proyecto de Git, también se incluye el archivo CMakeLists.txt:

```
Aru@Kanon[superhappyrocket]$ ls
CMakeLists.backup  GLTools       glad         shaders
CMakeLists.txt     build         linux-build  src
CMakeModules       dependencies  main.cpp     windows-build
```

Figura 7.18 Contenido del proyecto visto desde Linux.

Con el cual se descargan y configuran automáticamente las librerías que faltan, además se crea un archivo de proyecto para el IDE de elección en el sistema operativo que sea, para hacer uso de este solo basta instalar CMake:

Para instalar CMake en Mac OS X o Linux, es recomendable descargar el archivo .dmg o el .sh desde la página web https://cmake.org/download/ :

Binary distributions:

Platform	Files
Windows Installer: **Installer tool has changed. Uninstall CMake 3.4 or lower first!**	cmake-3.5.2-win32-x86.msi
Windows ZIP	cmake-3.5.2-win32-x86.zip
Mac OSX 10.6 or later	cmake-3.5.2-Darwin-x86_64.dmg
	cmake-3.5.2-Darwin-x86_64.tar.gz
	cmake-3.5.2-Darwin-x86_64.tar.Z
Linux x86_64	cmake-3.5.2-Linux-x86_64.sh
	cmake-3.5.2-Linux-x86_64.tar.gz
	cmake-3.5.2-Linux-x86_64.tar.Z
Linux i386	cmake-3.5.2-Linux-i386.sh
	cmake-3.5.2-Linux-i386.tar.gz
	cmake-3.5.2-Linux-i386.tar.Z

Figura 7.19 Página de descargas de CMake.

Mientras que para Ubuntu, por ejemplo, pueden ejecutarse los siguientes comandos:

```
sudo apt-get install software-properties-common
sudo add-apt-repository ppa:george-edison55/cmake-3.x
sudo apt-get update
sudo apt-get install cmake
```

Posteriormente, para generar un proyecto de Code:Blocks, basta con crear una carpeta en donde se realizará la compilación:

```
Aru@Kanon[superhappyrocket]$ mkdir linux_build
Aru@Kanon[superhappyrocket]$ ls
CMakeLists.backup  GLTools       glad         shaders
CMakeLists.txt     build         linux_build  src
CMakeModules       dependencies  main.cpp     windows-build
```

Figura 7.20 Creación de un subdirectorio para el proyecto.

Posteriormente sólo se debe cambiar al directorio que se acaba de crear y desde ahí ejecutar CMake con el IDE seleccionado :

```
Aru@Kanon[superhappyrocket]$ cd linux_build/
Aru@Kanon[linux_build]$ cmake -G "CodeBlocks - Unix Makefiles" ..
-- The C compiler identification is GNU 5.3.0
-- The CXX compiler identification is GNU 5.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++.exe
-- Check for working CXX compiler: /usr/bin/c++.exe -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Using /home/Aru/git/temp/superhappyrocket/dependencies to place downloaded packages
-- Looking for include file pthread.h
-- Looking for include file pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - found
-- Found Threads: TRUE
-- Found SDL2: /usr/lib/libSDL2main.a;/usr/lib/libSDL2.dll.a
-- SDL2 found in /home/Aru/git/temp/superhappyrocket/dependencies/src/sdl2/include and
-- Found SDL2_mixer: /usr/lib/libSDL2_mixer.dll.a (found version "..")
-- SDL2_Mixer found in /home/Aru/git/temp/superhappyrocket/dependencies/src/sdl2-mixer/
```

Figura 7.21 Creación del archivo en el IDE Code:Blocks.

Para darse una idea de los IDEs que soporta CMake se puede ejecutar *CMake --help | grep Generate*s :

```
Aru@Kanon[superhappyrocket]$ cmake --help | grep Generates
  Unix Makefiles               = Generates standard UNIX makefiles.
  Ninja                        = Generates build.ninja files.
  CodeBlocks - Ninja           = Generates CodeBlocks project files.
  CodeBlocks - Unix Makefiles  = Generates CodeBlocks project files.
  CodeLite - Ninja             = Generates CodeLite project files.
  CodeLite - Unix Makefiles    = Generates CodeLite project files.
  Eclipse CDT4 - Ninja         = Generates Eclipse CDT 4.0 project files.
  Eclipse CDT4 - Unix Makefiles= Generates Eclipse CDT 4.0 project files.
  KDevelop3                    = Generates KDevelop 3 project files.
  KDevelop3 - Unix Makefiles   = Generates KDevelop 3 project files.
  Kate - Ninja                 = Generates Kate project files.
  Kate - Unix Makefiles        = Generates Kate project files.
  Sublime Text 2 - Ninja       = Generates Sublime Text 2 project files.
                               = Generates Sublime Text 2 project files.
```

Figura 7.22 Algunos de los IDEs soportados por CMake.

El proyecto final de Code:Blocks, luce así:

Name	Date modified	Type	Size
CMakeFiles	5/23/2016 12:23 AM	File folder	
GLTools	5/23/2016 12:13 AM	File folder	
cmake_install.cmake	5/23/2016 12:13 AM	CMake File	2 KB
CMakeCache.txt	5/23/2016 12:13 AM	Text Document	38 KB
Makefile	5/23/2016 12:13 AM	File	24 KB
shr_engine.cbp	5/23/2016 12:13 AM	project file	23 KB
shr_engine.layout	5/23/2016 12:21 AM	LAYOUT File	1 KB

Figura 7.23 Proyecto resultado del uso de Code:Blocks.

El proyecto puede ser construido haciendo click derecho sobre el proyecto, y el resto de los pasos son iguales a los seguidos en Visual Studio. Con las librerías creadas hacer un nuevo proyecto similar al de Windows y construir el proyecto:

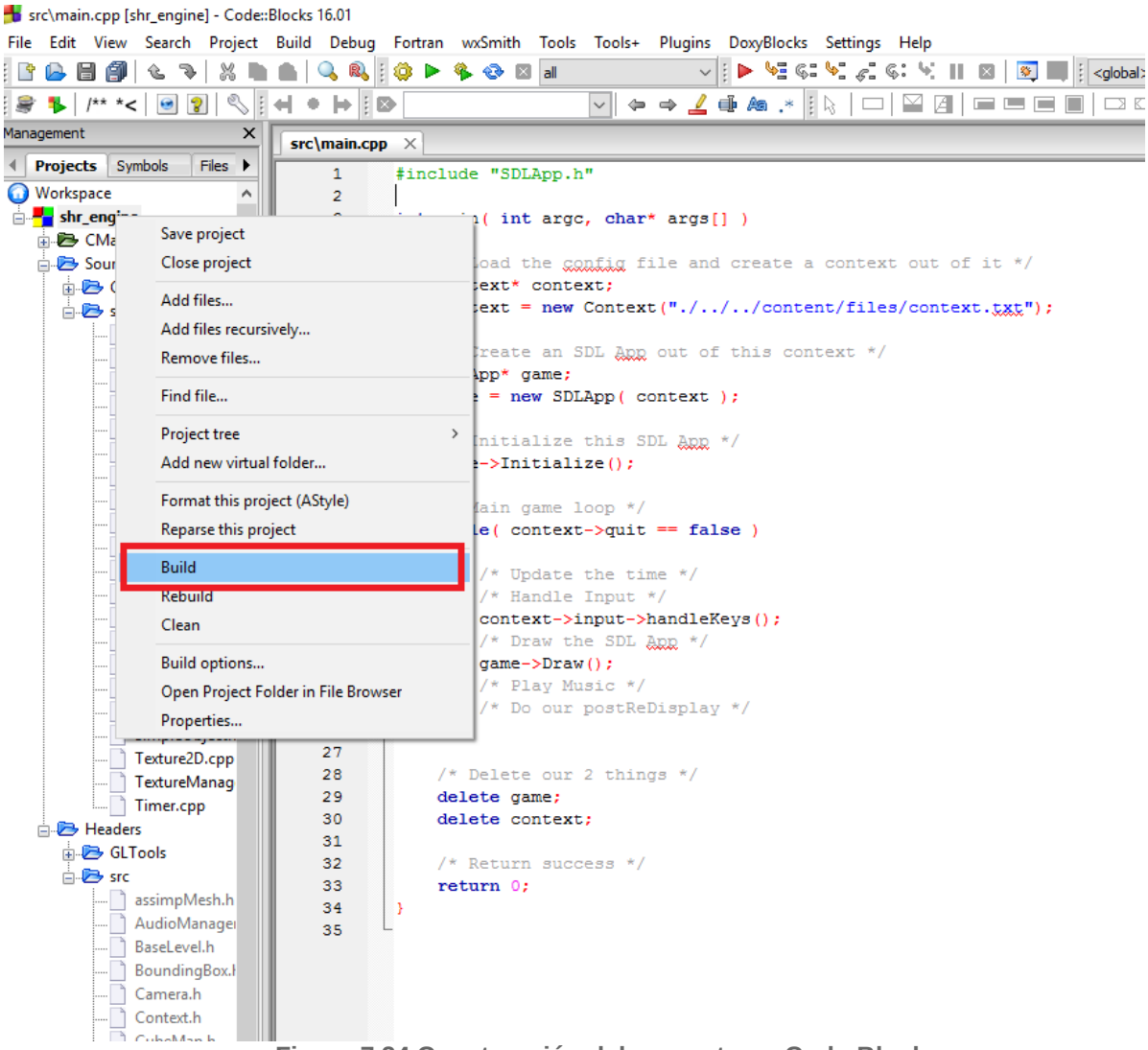


Figura 7.24 Construcción del proyecto en Code:Blocks.

Información Adicional

Inicialización de una ventana con un contexto OpenGL sin SDL

Por su naturaleza de bajo nivel, OpenGL dejá todo el control en manos de los programadores, el código base de OpenGL es portable entre varias plataformas y sistemas operativos. Como cada sistema operativo emplea diferentes métodos para el manejo de ventanas, cada sistema operativo tiene una diferente capa para ayudar a las aplicaciones a interactuar con OpenGL. Esto ayuda al driver a entender que tipo de buffers, formatos de color, y otras características deben ser usadas para cada caso especifico. El haber hecho uso de la herramienta de SDL, evitó la necesidad de usar herramientas específicas de cada plataforma, como lectura adicional, a continuación se presenta la forma de crear un contexto de OpenGL en las plataformas de interés.

OpenGL en Windows

En sistemas operativos Microsoft Windows, se utiliza un set de funciones ligadas a la API de Windows, llamado WGL (Windows-GL). Las funciones WGL tienen el prefijo *wgl* en frente del nombre de funciones y *WGL_* en frente del nombre de tokens, simbolizando que estás funciones son para interfaces entre Windows y OpenGL.
Microsoft brinda una implementación genérica de OpenGL como versión default en sus sistemas operativos. Si no se encuentra hardware 3D en el sistema o los drivers apropiados no están instalados, se utiliza la versión de OpenGL de Microsoft. La versión soportada por la mayoría de sistemas operativos de Microsoft es la 1.1. Está simplemente no es suficiente para ninguna aplicación 3D moderna.
Existen muchos métodos para dibujar una ventana en un sistema operativo de Windows. La más vieja y la que tiene más soporte es Windows GDI (Graphics Device Interface). GDI es estrictamente una interfaz de dibujo 2D y era acelerada por hardware hasta Windows Vista, aunque GDI aun esta disponible en las versiones de Windows Vista y posteriores, ya no es acelerada por hardware, la tecnología de dibujo preferida está basada en .NET y se llama Windows Presentation Foundation (WPF).
Cuando se emplea GDI, cada ventana posee un contexto del dispositivo (Device Context, DC) que recibe la salida de gráficos (output), y cada función GDI toma un DC como argumento para indicar cuál contexto se desea que afecte dicha función. Se puede tener varios contextos, pero solo uno por ventana. OpenGL tiene un identificador de contexto llamado rendering context (RC).
Antes de poder renderizar cualquier cosa, se necesita un contexto de OpenGL, antes de poder crear un contexto de OpenGL, se necesita un contexto GDI, y antes de poder obtener un contexto GDI, se necesita una ventana. Para crear una ventana se usa la función *CreateWindowEx* y se le indica el tipo de ventana que se quiere usar. Antes de poder crear una ventana, se necesita una clase para esa ventana. Para configurar la clase de la ventana, se hace una llamada a *RegisterClass*, cuyo prototipo es:

```
ATOM WINAPI RegisterClass(const WNDCLASS *lpWndClass);
```

La función *RegisterClass* recibe un puntero a una estructura *WNDCLASS*, la cual define a la clase. No todos los puntos de esta clase son empleados, pero algunos son importantes. La nueva clase de Windows se registra de la siguiente manera:

```
WNDCLASS cls;

::ZeroMemory(&cls, sizeof(cls));

cls.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
cls.lpfnWndProc = &WindowProc;
cls.hInstance = ::GetModuleHandle(NULL);
cls.lpszClassName = TEXT("OPENGL");

::RegisterClass(&cls);
```

Primero se inicializa el contenido de la estructura a cero usando *ZeroMemory* lo que significa que todos los miembros de la estructura que no fueron llenados serán ceros. Se indica el estilo de la estructura como `CS_HREDRAW | CS_VREDRAW | CS_OWNDC`, lo que le dice a Windows redibujar la ventana si su altura o ancho cambia, y dar su propio DC a cada ventana de esta clase. *WindowProc* es la función que maneja los mensajes , *hInstance* guarda la instancia de la aplicación, y *lpszClassName* es la dirección de un string con el cual se conoce a nuestra nueva clase, en el ejemplo "OPENGL". Después se puede crear la ventana llamando a *CreateWindowEx,* su prototipo es:

```
HWND WINAPI CreateWindowEx(DWORD dwExStyle,
                           LPCTSTR lpClassName,
                           LPCTSTR lpWindowName,
                           DWORD dwStyle,
                           int x,
                           int y,
                           int nWidth,
                           int nHeight,
                           HWND hWndParent,
                           HMENU hMenu,
                           HINSTANCE hInstance,
                           LPVOID lpParam);
```

dwExStyle y *dwStyle* son el estilo de la ventana, l*pClassName* es el nombre de la clase de la ventana que acaba de ser registrada. Los parametros *x, y, nWidth y nHeight* especifican la posición y tamaño de la ventana. *hWndParent* es el alias de la ventana padre. Se usa este valor si la ventana es creada dentro de otra, pero en este caso su valor será NULL. De igual manera la ventana creada no tendrá menu asi que *hMenu* también será NULL.

lpParam es un puntero que puede ser usado para lo que sea, cuando la ventana es creada, se puede obtener este parámetro.
El código para crear la ventana y obtener su device context es:

```
HWND hWnd = ::CreateWindowEx(WS_EX_APPWINDOW | WS_EX_WINDOWEDGE,
                             TEXT("OPENGL"),
                             TEXT("OpenGL Window"),
                             WS_CLIPSIBLINGS | WS_CLIPCHILDREN |
                             WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                             0, 0,
                             800, 600,
                             NULL,
                             NULL,
                             hInstance,
                             NULL);

HDC dc = ::GetDC(hWnd);
```

Una vez que creamos la ventana, se puede obtener su device context usando la función *GetDC.* Ahora se puede configurar el DC para el renderizado en OpenGL.
Antes de que OpenGL pueda renderizar en la ventana se debe configurar dependiendo de las necesidades de cada aplicación. Uno o dos buffers , buffer de profundidad, stencil, etc, una vez asignados estos valores a una ventana, no pueden ser cambiados, se debe destruir la ventana y después recrearla. OpenGL y Windows utilizan formatos de píxeles (pixel formats) para encapsular toda esta información en grupos. Se debe encontrar un pixel format que tenga las características que satisfagan las necesidades de la aplicación deseada. El pixel format es después usado para crear el contexto de renderizado de OpenGL.
Para encontrar un formato que cumpla con las necesidades de la aplicación se debe crear una instancia de esta estructura y llamar a la función *ChoosePixelFormat,* cuyo prototipo es:

```
int ChoosePixelFormat(HDC hdc,
                      const PIXELFORMATDESCRIPTOR *ppfd);
```

La función *ChoosePixelFormat* regresa el índice del formato que más se adecua de los soportados por el driver OpenGL instalado. Se puede entonces llamar *SetPixelFormat()* usando este índice para poner el formato del device context que la aplicación OpenGL renderizará, el código para hacer esto es:

```
PIXELFORMATDESCRIPTOR pfd;

::ZeroMemory(&pfd, sizeof(pfd));

pfd.nSize               = sizeof(pfd);
```

```
pfd.nVersion            = 1;
pfd.dwFlags             = PFD_DRAW_TO_WINDOW |
                          PFD_SUPPORT_OPENGL |
                          PFD_GENERIC_ACCELERATED |
                          PFD_DOUBLEBUFFER;
pfd.iPixelType          = PFD_TYPE_RGBA;
pfd.cColorBits          = 24;
pfd.cRedBits            = 8;
pfd.cGreenBits          = 8;
pfd.cBlueBits           = 8;
pfd.cDepthBits          = 32;

int iPixelFormat = ::ChoosePixelFormat(dc, &pfd);
::SetPixelFormat(dc, iPixelFormat, &pfd);
```

SetPixelFormat() sólo puede ser llamado una vez para un DC dado. Para cambiar el pixel format la ventana debe ser destruida y recreada.
Una aplicación típica de windows puede consistir de varias ventanas. Se puede elegir un pixel format para cada ventana, OpenGL utiliza un contexto de renderizado (rendering context) para recordar la configuración y estado actual. El tipo de dato del renderizado de OpenGL en un contexto de Windows es *HGLRC* y se puede crear llamando *wglCreateContext ()*. SI todo es exitoso regresa el identificador del nuevo contexto y se puede crear llamando a *wglMakeCurrent ()*, el código es el siguiente:

```
HGLRC rc = wglCreateContext(dc);
wglMakeCurrent(dc, rc);
```

Una vez que se tiene el contexto actual, se puede entrar al loop de la aplicación principal (main loop). Cada aplicación debería contener un loop o ciclo que revise si hay mensajes en la cola de la ventana y de ser así lidiar con ellos. El loop de mensajes para una aplicación simple es:

```
for (;;)
{
   if (::PeekMessage(&msg, hWnd, 0, 0, PM_REMOVE))
   {
       if (msg.message == WM_QUIT)
       {
           break;
       }
       ::TranslateMessage(&msg);
       ::DispatchMessage(&msg);
   }
   DrawScene();
   ::SwapBuffers(dc);
```

```
    }
```

El programa ejemplo solicita un pixel format de doble buffer con *PFD_DOUBLEBUFFER*. Un buffer actúa como el buffer frontal y el otro como el buffer trasero. Se puede dibujar en ellos llamando *glDrawBuffers ()* con *GL_FRONT* o *GLBACK.* Utilizar doble buffer permite a OpenGL dibujar la escena entera al buffer trasero sin mostrar resultados intermedios en pantalla. Cuando se acab de dibujar, se le indica a OpenGL que los buffers deben ser intercambiados llamando a *SwapBuffers ()* con el DC de la ventana.

```
// Do the buffer swap
SwapBuffers(dc);
```

La implementación de OpenGL de Windows tiene una función llamada *wglGetProcAddress ()* que permite obtener un puntero a una función de OpenGL soportada por el driver, su prototipo es:

```
PROC wglGetProcAddress(LPSTR lpszProc);
```

Esta función recibe el nombre de una función de OpenGL y regresa un puntero de función que puede ser usado para llamarla directamente. WGL da soporte a extensiones pero antes de usarlas se debe determinar cuáles extensiones son soportadas por los drivers de OpenGL, para ello se utiliza la función *wglGetExtensionsStringARB ()* que regresa una string que contiene todos los nombres de las extensiones WGL soportadas por el driver de OpenGL. Se usa de la siguiente manera:

```
PFNWGLGETEXTENSIONSSTRINGARBPROC wglGetExtensionsStringARB;

wglGetExtensionsStringARB = (PFNWGLGETEXTENSIONSSTRINGARBPROC)
                            wglGetProcAddress("wglGetExtensionsStr
ingARB");

const char * extension_string = wglGetExtensionsStringARB();
```

La variable e*xtension_string* es un puntero a una cadena separada por espacios que contiene las extensiones soportadas por el driver de OpenGL. Si esta cadena contiene *WGL_ARB_create_context* entonces significa que existen funciones avanzadas para la creación de contextos y se puede usar la extensión para crear una versión del contexto más avanzada que la default *wglCreateContext ()*.
Cuando se acaba de usar la aplicación, es necesario limpiar la ventana para cerrar OpenGL, primero se deben destruir todos los objetos creados (como texturas, buffers, y demás) y posteriormente borrar los objetos del sistema de ventanas. Primero se borra el contexto de OpenGL usando *wglDeleteContext ():*

```
BOOL wglDeleteContext(HGLRC hglrc);
```

Después se libera el DC usando *ReleaseDC:*

```
int ReleaseDC(HWND hWnd, HDC hDC);
```

Posteriormente se destruye la ventana usando *DestroyWindow:*

```
BOOL DestroyWindow(HWND hWnd);
```

Finalmente es necesario desregistrar la clase de la ventana usando *UnregisterClass:*

```
BOOL UnregisterClass(LPCTSTR lpClassName, HINSTANCE hInstance);
```

Llamando cada una de estas funciones en el orden contrario en el cual sus funciones correspondientes de configuración fueron llamadas se liberan los recursos al sistema operativo de una manera efectiva y se limpia todo lo utilizado.

OpenGL en MAC OS X

OpenGL es la API de renderizado nativa y preferida en la plataforma de MAC OS X. De hecho OpenGL es usado en los niveles más bajos del sistema operativo para el escritorio, GUI, y la API gráfica 2D de MAC OS X (Quartz). Existen 4 tecnologías distintas de programación en OpenGL disponibles en Mac, cual usar depende de como se prefiera crear aplicaciones y de las necesidades específicas de rendereo. Estás tecnologías se complementan y pueden ser usadas simultáneamente:

GLUT
NSOpenGL
GGL
GLKit

Estas tecnologías son usadas para hacer la configuración por OpenGL en una ventana o display de un dispositivo, después de eso se utiliza OpenGL de la forma usual.
Un programa basado en Cocoa puede ser creado usando un nuevo proyecto en XCode, después de agregar los frameworks de OpenGL y GLKit. Se selecciona el archivo MainMenu.xib dentro del folder de Resources lo cual abre el archivo XIB para su edición. En la biblioteca de objetos es necesario desplazarse hasta ver el objeto OpenGL View, se le da click y se arrastra a la ventana principal además de modificar su tamaño para coincidir con el de la ventana principal. Ahora se posee una ventana configurada para el uso de OpenGL casi lista para usarse. Esta vista requiere ser conectada a una clase de Cocoa derivada de *NSOpenGLView*.

Se empieza por crear una clase propia de Objective-C derivada de NSView(subclase). Esto crea dos archivos y los agrega al proyecto: GLCoreProfile.h y GLCoreProfile.m. La definición de la clase de vista es:

```
@interface GLCoreProfileView : NSOpenGLView
{
}
- (id)   initWithCoder:(NSCoder *)aDecoder;
- (void) drawRect:(NSRect) bounds;
- (void) prepareOpenGL;
- (void) reshape;
@end
```

Este es el esqueleto mínimo para una vista de OpenGL funcional. Ahora se ven los métodos modificados, siendo InitWithCoder el más importante ya que es el que inicializa la vista y contexto de OpenGL:

```
-(id)initWithCoder:(NSCoder *)aDecoder
{
NSOpenGLPixelFormatAttribute pixelFormatAttributes[] =
   {
   NSOpenGLPFAColorSize,       32,
   NSOpenGLPFADepthSize,       24,
   NSOpenGLPFAStencilSize,     8,
   NSOpenGLPFAAccelerated,
   NSOpenGLPFAOpenGLProfile, NSOpenGLProfileVersion3_2Core,
   0
   };

NSOpenGLPixelFormat   *pixelFormat   =   [[[NSOpenGLPixelFormat
alloc]
              initWithAttributes:pixelFormatAttributes]
autorelease];
NSOpenGLContext* openGLContext = [[[NSOpenGLContext alloc]
              initWithFormat:pixelFormat       shareContext:nil]
autorelease];

[super initWithCoder:aDecoder];
[self setOpenGLContext:openGLContext];
[openGLContext makeCurrentContext];

return self;
}
```

Después se configura el pixel format:

```
   NSOpenGLPixelFormatAttribute pixelFormatAttributes[] =
      {
      NSOpenGLPFAColorSize,       32,
      NSOpenGLPFADepthSize,       24,
      NSOpenGLPFAStencilSize,     8,
      NSOpenGLPFAAccelerated,
      NSOpenGLPFAOpenGLProfile, NSOpenGLProfileVersion3_2Core,
      0
      };
NSOpenGLPixelFormat *pixelFormat = [[[NSOpenGLPixelFormat alloc]
                     initWithAttributes:pixelFormatAttributes]
autorelease];
```

Las tareas de renderizado de OpenGL tipicas usualmente requieren de una configuración una sola vez, como precarga de texturas, shaders, geometría, etc. NSOpenGLView tiene un método que es llamado antes de que cualquier operación de renderizado ocurra, llamada *prepareOpenGL*, en el ejemplo muestra información a la consola acerca del contexto de renderizado seleccionado, y ajusta el buffer de color a un color gris oscuro:

```
- (void)prepareOpenGL
   {
   glClearColor(0.1f, 0.1f, 0.1f, 1.0f);

   printf("Version: %s\r\n", glGetString(GL_VERSION));
   printf("Renderer: %s\r\n", glGetString(GL_RENDERER));
   printf("Vendor: %s\r\n", glGetString(GL_VENDOR));
   printf("GLSL Version: %s\r\n",
                     glGetString(GL_SHADING_LANGUAGE_VERSION));
   }
```

Finalmente se llega a la parte en donde se puede poner código de renderizado, en el código ejemplo de renderizado siguiente se limpia el buffer de color:

```
- (void)drawRect:(NSRect)bounds
   {
   glClear(GL_COLOR_BUFFER_BIT);
   glFlush();
   }
```

OpenGL en Linux

OpenGL ha sido la API de renderizado 3D preferida en varias versiones de Linux. Linux ofrece varias maneras de acceder a OpenGL. Mesa3D es una implementación de

software que no depende del hardware y puede ser instalada en la mayoría de servidores X. El sistema X Window es una interfaz de usuario gráfica más intuitiva para los usuarios que la ventana de comandos, similar a Windows o Mac OS.
Para revisar si OpenGL es soportado en el sistema se debe ejecutar el comando *glxinfo* como se muestra a continuación:

```
glxinfo | grep rendering
```

Lo cual regresa una de dos opciones:

```
direct rendering: Yes
```

Ó

```
direct rendering: No
```

Si la respuesta es Yes, el sistema puede correr OpenGL, de lo contrario el sistema no soporta OpenGL o no se tienen los drivers instalados. En este caso se puede correr los siguientes comandos:

```
glxinfo | grep "OpenGL vendor"

glxinfo | grep "OpenGL version"
```

El sistema nos indicará la versión del driver de OpenGL instalada, si no se cuenta con OpenGL se puede instalar Mesa o instalar una tarjeta de vídeo que soporte renderizado 3D y tenga drivers para Linux.

Después de descargar e instalar Mesa3D, se debe descargar e instalar GLFW para interactuar con el sistema operativo y su sistema de ventanas. Además se hace uso de la librería GL3W para cargar e inicializar punteros a funciones de OpenGL. GL3W es un solo archivo fuente que es generado directamente de los archivos de cabecera oficiales de OpenGL usando un script de Python.
Para hacer uso de GLFW en las aplicaciones, se debe agregar la librería con el comando de vínculo:

```
-lglfw
```

GL3W es bastante sencillo pues es un solo archivo, y por lo mismo puede ser simplemente incluido en el proyecto. Después llamar *gl3wInit ()* antes de que la aplicación haga llamadas a OpenGL.
En X Windows, una interfaz común para la comunicación entre OpenGL y X Windows es GLX, esta interfaz es similar a WGL en Windows y a AGL en Mac.
Antes de poder crear una ventana se debe encontrar que display usará la aplicación:

```
Display *dpy = XOpenDisplay(getenv("DISPLAY"));
```

Esto nos regresa un puntero el cual puede ser usado después para indicarle al servidor X en donde se está trabajando.

Para crear una ventana se usa la función de X *XCreateWindow ().* El resultado es un handle para la nueva ventana. La función necesita una ventana padre pero se puede usar la ventana principal de X, además de otros parámetros como el tamaño y la posición. La declaración de la función completa luce así:

```
Window XCreateWindow(Display * dpy,
                     Window parent,
                     int x, int y,
                     unsigned int width,
                     unsigned int height,
                     unsigned int border_width,
                     int depth,
                     unsigned int class,
                     Visual *visual,
                     unsigned_long valuemask,
                     XSetWindowAttributes *attributes);
```

Después de crear la ventana se usa el handle obtenido para crear una ventana GLX correspondiente con el comando *glXCreateWindow ()*:

```
GLXWindow glXCreateWindow(Display * dpy,
                          GLXFBConfig config,
                          Window win,
                          const int *attrib_list);
```

Después de terminar de renderizar, se deben limpiar las ventanas creadas. Para destruir la ventana GLX se llama *glXDestroyWindow ()* con el handle obtenido al llamar *glXCreateWindow()*:

```
lXDestroyWindow(Display * dpy,
                GLXWindow window);
```

Finálmente, se destruye la ventana X creada originalmente. Se debe usar el comando *XDestroyWindow ()* con el handle de la ventana X:

```
XDestroyWindow(Display * dpy, Window win);
```

Documentación

Aquí se presentan las clases, estructuras, uniones, e interfaces más importantes con una pequeña descripción. La lista se encuentra ordenada alfabéticamente, pero no completamente, debido a la herencia de algunas clases:

- AudioManager
- BaseLevel
  - Level1
- BoundingBox
- Camera
- Context
- Entity
- EntityManager
- InputManager
- Renderer
- SceneManager
- SDLApp
- SimpleObject
  - assimpMesh
  - CubeMap
  - Light
  - ReflectedObject
- Texture2D

TextureManager

Timer

AudioManager Class Reference

`#include <AudioManager.h>`

Public Member Functions

	AudioManager (Context *ctx)
int	**Initialize** ()
int	**LoadContent** ()
int	**Update** ()
int	**UnloadContent** ()
int	**LoadMusic** (const char *file)
int	**PlayMusic** ()
int	**LoadSound** (const char *file)
int	**PlaySound** (int soundFile, int loopTimes)
int	**PauseSound** (int soundFile)
int	**StopSound** (int soundFile)

Public Attributes

Context *	**ctxt**
Mix_Music *	**music**
std::vector< Mix_Chunk * >	**sounds**
int	**audio_rate**
int	**audio_format**
int	**audio_channels**
int	**audio_buffers**

Detailed Description

AudioManager.h

Manages the audio on the current context responsible for loading, playing, pausing, stopping and shutting down the audio.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.

BaseLevel Class Reference

Public Member Functions

	BaseLevel (Context *ctx)
virtual const int	**Initialize** ()
virtual const int	**LoadContent** ()=0
virtual const int	**Draw** ()
virtual const int	**Update** (Uint32 gameTime)
virtual const int	**UnloadContent** ()

Public Attributes

Context *	**ctxt**
char *	**song**
vector< char * >	**sounds**
vector< SimpleObject * >	**actors**
vector< Texture2D * >	**textures**
Light *	**lightSource**
Timer *	**clock**

Uint32	**startTicks**
Uint32	**currentTicks**
Uint32	**deltaTicks**

Detailed Description

BaseLevel.h

Content is loaded depending of each level, after being loaded the content is drawn, updated, and unloaded after being used.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.

Level1 Class Reference

`#include <Level1.h>`

Public Member Functions

	Level1 (Context *ctx)
const int	**Initialize** ()
const int	LoadContent ()
const int	**Update** (Uint32 gameTime)
void	**HandleInput** (InputManager *input, Uint32 gameTime)
void	**pushEntityManagerObjects** ()

Public Member Functions inherited from BaseLevel

Public Attributes

assimpMesh *	**rocket**
M3DVector3f	**rocketForwardVector**

M3DVector3f	**rocketUpVector**
M3DVector3f	**rocketOrigin**
M3DVector3f	**cameraForwardVector**
M3DVector3f	**cameraUpVector**
M3DVector3f	**cameraOrigin**
M3DMatrix44f	**cameraMatrix**
assimpMesh *	**cat**
Entityanager *	**entityManager**
bool	**start**
float	**linear**
float	**angular**

Public Attributes inherited from BaseLevel

Detailed Description

Level1.h

This derived class comes from the base class BaseLevel, it is used to generate the first level used on the Super Happy Rocket project.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.

BoundingBox Class Reference

`#include <BoundingBox.h>`

Public Member Functions

	BoundingBox (float c0, float c1, float c2, float r1, float r2, float r3)

	BoundingBox (const BoundingBox &box)
const bool	**intersectsWith** (const BoundingBox &box)

Public Attributes

float	**c** [3]
float	**r** [3]

Detailed Description

BoundingBox.h

This class is used to create a bounding box that determines if an object intersects with any other object with a collision detection method.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.

Camera Class Reference

`#include <Camera.h>`

Public Member Functions

void	**MoveForward** (float delta)
void	**MoveLeft** (float delta)
void	**MoveUp** (float delta)
void	**RotateX** (float delta)
void	**RotateY** (float delta)
void	**RotateZ** (float delta)

Public Attributes

3DMatrix44f	**camera**
GLFrame	**cameraFrame**

GLMatrixStack	**modelViewMatrix**
GLMatrixStack	**projectionMatrix**
GLFrustum	**viewFrustum**
GLGeometryTransform	**transformPipeline**

Detailed Description

Camera.h

Manages the camera on the current context This class is in charge of getting the current camera frame, model view matrix, projection matrix and view frustum.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.

Context Class Reference

Public Member Functions

	Context (char *file)
int	**LoadContext** (char *file)
int	**SaveContext** (char *file)

Public Attributes

bool	**quit**
int	**width**
int	**height**
int	**bpp**
int	**vMode**
int	**level**
AudioManager *	**audio**

InputManager *	**input**
Renderer *	**renderer**
Camera *	**camera**
GLShaderManager *	**shaderManager**
Timer *	**timer**
SceneManager *	**sceneManager**
TextureManager *	**textureManager**

Static Public Attributes

static const string	**basepath**

Detailed Description

Context.h

The Game Context class, where we store things that are relevant to every object in the game, where the different managers are, how to access all of the stuff. Can be saved to a file and loaded for future support of many things, game saves, reloading, tab switching, etc. Window configuration and video mode is also stored in here

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.

Entity Class Reference

`#include <Entity.h>`

Public Member Functions

	Entity (const int type, Context *ctx)
void	**Update** (Uint32 gameTime)

Public Attributes

Context *	**ctxt**
int	**entityType**
assimpMesh *	**mesh**
BoundingBox *	**box**
bool	**move**
float	**speed**

Detailed Description

Entity.h

An entity is a spawnable object by our entity manager, it can be either a star or a building, and it will move accordingly it has a bounding box to calculate collisions with.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.

EntityManager Class Reference

Public Member Functions

	EntityManager (Context *ctx)
const int	**Update** (Uint32 gameTime, Uint32 deltaTicks)
const bool	**createObjects** (const unsigned int numObjects)
const bool	**parseFile** (const char *file)
const bool	**calculateWhenToSpawn** (int numEntity)
const bool	**spawnEntity** (int numEntity)
const bool	**entityCollides** (int numEntity)

Public Attributes

assimpMesh *	**player**

float	**cutoffDistance**
Context *	**ctxt**
AudioManager *	**audio**
vector< int >	**spawnTimes**
vector< Entity * >	**entities**
Uint32	**startTime**

Detailed Description

EntityManager.h

The EntityManager class is the responsible of creating and managing events related to our entity objects, such as collisions and spawning times.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.

InputManager Class Reference

`#include <InputManager.h>`

Public Member Functions

	InputManager (Context *context)
bool	**isKeyPressed** (Uint32 key)
Uint32	**handleKeys** ()

Public Attributes

bool	**keysHeld** [SDL_NUM_SCANCODES]
Context *	**ctxt**
SDL_Event	**event**

Detailed Description

InputManager.h

This class is responsible for processing the user input such as keys pressed or changes done to the current context.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.

Renderer Class Reference

Public Member Functions

	Renderer (Context *ctx)
const int	**Initialize** ()
const int	**LoadContent** ()
const int	**Update** (float delta)
const int	**Draw** ()
const int	**UnloadContent** ()
const int	**ChangeSize** (int nWidth, int nHeight)

Static Public Member Functions

static void APIENTRY	**myErrorCallback** (GLenum _source, GLenum _type, GLuint _id, GLenum _severity, GLsizei _length, const char *_message, void *_userParam)

Public Attributes

Context *	**ctxt**
Camera *	**camera**
GLShaderManager	**shaderManager**
vector< SimpleObject * >	**actors**
Light *	**light**

Static Public Attributes

static GLuint	**unusedIds** = 0

Detailed Description

OpenGLRenderer.h

This class is the core of our system, it manages all OpenGL rendering. Loads, draws, updates and shuts down our rendering context. Manages the ASSIMP library, textures, light and shader support.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.

SceneManager Class Reference

`#include <SceneManager.h>`

Public Member Functions

	SceneManager (Context *ctx)
const int	**Initialize** ()
const int	**LoadContent** (const unsigned int level)
const int	**Draw** ()
const int	**Update** (Uint32 gameTime)
const int	**UnloadContent** ()

Public Attributes

Context *	**ctxt**
int	**currentLevel**
vector< BaseLevel * >	**levels**

Detailed Description

SceneManager.h

The scene manager is responsible of loading, updating and unloading the content of the currently selected level.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.

SDLApp Class Reference

Public Member Functions

	SDLApp (Context *context)
int	**Initialize** ()
int	**LoadContent** ()
int	**Update** ()
int	**Draw** ()
int	**UnloadContent** ()
void	**postRedisplay** ()

Public Attributes

Context *	**ctxt**
SDL_Window *	**window**
SDL_GLContext	**gContext**
int	**width**
int	**height**
int	**bpp**
int	**vMode**
int	**wantRedisplay**

Detailed Description

SDLApp.h

This class is responsible of giving support to the SDL library, initializes, updates and shuts down the application window depending on the selected context. Manages events related to the window.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.

SimpleObject Class Reference

Public Member Functions

	SimpleObject (const SimpleObject &obj)
virtual const int	**Initialize** (Context *ctx)
virtual const int	**LoadContent** ()
virtual const int	**Draw** ()
virtual const int	**Draw** (Camera *camera, GLShaderManager *shaderManager, TextureManager *textureManager, Light *light)
virtual const int	**Update** ()
virtual const int	**UnloadContent** ()

Public Attributes

bool	**tBatch**
GLBatch	**batch**
GLTriangleBatch	**triBatch**
GLFrame	**frame**
GLMatrixStack	**modelViewMatrix**
vector< float >	**vertex**

vector< float >	**index**
vector< float >	**normal**
vector< float >	**texture**
vector< float >	**color**
GLuint	**textureFile**
string	**tFile**
GLuint	**shaderFile**
string	**vShader**
string	**fShader**
M3DMatrix44f	**mScaleMatrix**
M3DVector3f	**scaleVector**

Detailed Description

SimpleObject.h

A SimpleObject is a structure created to represent actors in our scene. A SimpleObject is drawn via the stockShaders.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.

assimpMesh Class Reference

Public Member Functions

	assimpMesh (const char *filename, **Context** *ctx)
	assimpMesh (const **assimpMesh** &mesh)
const int	**Initialize** (**Context** *ctx)
const int	**LoadContent** ()

const int	**Draw** (Camera *camera, GLShaderManager *shaderManager, TextureManager *textureManager, Light *light)
const int	**Update** ()
const int	**UnloadContent** ()
void	**recursiveProcess** (aiNode *node, const aiScene *scene)
void	**processMesh** (aiMesh *mesh, const aiScene *scene)
void	**createBatches** ()

Public Member Functions inherited from SimpleObject

Public Attributes

Context *	**ctxt**
string	**fileName**
vector< SimpleObject * >	**objects**
vector< vector< GLfloat > >	**verts**
vector< vector< GLfloat > >	**norms**
vector< vector< GLfloat > >	**texts**
vector< vector< GLushort > >	**indexes**
vector< GLushort >	**maxIndex**
vector< GLuint >	**textures**
vector< Texture2D * >	**textureStruct**
unsigned int	**numTextures**
bool	**untexturedVerts**

Public Attributes inherited from SimpleObject

Detailed Description

assimpMesh.h

Loads and processes the meshes and textures associated with them.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.

CubeMap Class Reference

`#include <CubeMap.h>`

Public Member Functions

	CubeMap (GLfloat size, const char *f1, const char *f2, const char *f3, const char *f4, const char *f5, const char *f6)
const int	**Initialize** (Context *ctx)
const int	**LoadContent** ()
const int	**Draw** (Camera *camera, GLShaderManager *shaderManager, TextureManager *textureManager, Light *light)
cost int	**UnloadContent** ()

Public Member Functions inherited from SimpleObject

Public Attributes

Context *	**ctxt**
GLfloat	**cubeMapSize**
const char *	**szCubeFaces** [6]
GLenum	**cube** [6]
GLbyte *	**pBytes**
GLint	**iWidth**
GLint	**iHeight**
GLint	**iComponents**
GLenum	**eFormat**

GLint	**locMVPSkyBox**
GLint	**locCubeMap**

Public Attributes inherited from SimpleObject

Detailed Description

CubeMap.h

The CubeMap class is used to create a cubemap or skybox to simulate the world's sky, it generates the cubemap from 6 different texture files each one representing the X, Y and Z positive and negative sides.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.

Light Class Reference

`#include <Light.h>`

Public Member Functions

Light (GLfloat *pos, GLfloat *col)

Public Member Functions inherited from SimpleObject

Public Attributes

M3DVector3f	**position**
M3DVector4f	**color**

Public Attributes inherited from SimpleObject

Detailed Description

Light.h

Responsible of creating light sources on the scene.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.

ReflectedObject Class Reference

`#include <ReflectedObject.h>`

Public Member Functions

	ReflectedObject (const char *upperTexture, GLint reflectedText)
const int	**Initialize** (Context *ctx)
const int	**LoadContent** ()
const int	**Draw** (Camera *camera, GLShaderManager *shaderManager, TextureManager *textureManager, Light *light)
const int	**UnloadContent** ()

Public Member Functions inherited from SimpleObject

Public Attributes

Context *	**ctxt**
GLbyte *	**pBytes**
GLint	**iWidth**
GLint	**iHeight**
GLint	**iComponents**
GLenum	**eFormat**
GLint	**locMVPReflect**
GLint	**locMVReflect**
GLint	**locNormalReflect**
GLint	**locInvertedCamera**
GLint	**locTarnishMap**

GLint	**locCubeMap**
GLuint	**reflectedTexture**

Public Attributes inherited from SimpleObject

Detailed Description

ReflectedObject.h

The ReflectedObject class is used to generate a reflected image of the object we load, in this case we create a reflection of the world to use in objects that have a reflection property.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.

Texture2D Class Reference

`#include <Texture2D.h>`

Public Member Functions

Texture2D (const Texture2D &txt)
Texture2D (const char *f, GLenum min, GLenum mag, GLenum wrap)
Texture2D (string *str, GLenum min, GLenum mag, GLenum wrap)
Texture2D (string str, GLenum min, GLenum mag, GLenum wrap)

Public Attributes

const char *	**file**
string	**textureString**
unsigned int	**textureFile**
GLenum	**minFilter**

GLenum	**magFilter**
GLenum	**wrapMode**

Detailed Description

Texture2D.h

Structure containing the data of a 2D texture.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.

TextureManager Class Reference

`#include <TextureManager.h>`

Public Member Functions

	TextureManager (Context *ctx)
const int	**Initialize** ()
const int	**LoadContent** ()
const int	**UnloadContent** ()
const int	**bindTexture** (int text)
const bool	**LoadTGATexture** (const char *szFileName, GLenum minFilter, GLenum magFilter, GLenum wrapMode)
const bool	**LoadTGATexture** (Texture2D *text)
const int	**addTexture** (string file)
const int	**addTexture** (char *file)
const int	**addTexture** (Texture2D *text)

Public Attributes

Context *	**ctxt**

GLuint *	**textures**
vector< char * >	**textureFiles**
vector< Texture2D * >	**texts**
int	**numTextures**
GLbyte *	**pBytes**

Detailed Description

TextureManager.h

Responsible for loading and initializing the 2D textures.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.

Timer Class Reference

`#include <Timer.h>`

Public Member Functions

void	**start** ()
void	**stop** ()
void	**pause** ()
void	**unpause** ()
Uint32	**get_ticks** ()
bool	**is_started** ()
bool	**is_paused** ()

Detailed Description

Timer.h

Timer that helps us with several clock related actions.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details http://www.gnu.org/licenses/.