UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

DESARROLLO DE FRAMEWORK Y PRUEBA DE CONCEPTO PARA CONSOLA NINTENDO DS

TESIS

QUE PARA OBTENER EL TÍTULO DE

INGENIERO EN COMPUTACIÓN

PRESENTAN:

HUGO MENDIETA PACHECO

ALEJANDRO VALENZUELA ROCA

DIRECTOR DE TESIS:

ING. RODRIGO GUILLERMO TINTOR PÉREZ

CIUDAD UNIVERSITARIA JUNIO DE 2008

A mi mamá y a mi papá por todo su cariño, por apoyarme siempre y en todo. Por los regaños, por los abrazos, los disgustos y las alegrías, los consejos, opiniones encontradas pero también libertades. Por poner una almohada cuando hay tropiezos. Por haberme brindado la oportunidad del viaje que está a punto de terminar, como el que comenzará. No lo habría logrado sin ustedes.

A mi familia, por estar siempre presentes a pesar de la distancia.

A mis profesores por darme herramientas, experiencia, conocimiento y reconocer en cada persona un individuo y no solo un número de cuenta.

A las Tías, por ayudarme en momentos difíciles. Por ser tan buenas personas conmigo.

A mi compañero de tesis, por haberse fletado la insana carrera contra el tiempo conmigo y ayudarme a hacer posible el siguiente sueño. Por haberme demostrado como es posible superar toda clase de adversidades.

A mis amigos por aportar toda su alegría, motivación, su tiempo, sus consejos y consuelos, y (de vez en cuando) sonsacarme. Odiaría olvidar agradecerle a alguno en particular por un descuido, por lo que no seré específico, pero quiero que sepan que todos ustedes me han aportado mucho, mucho más de lo que creen. Además serían culpables de que hacer engordar a la tesis dos kilos más, y ya no está tan flaca. A todo lo que hemos compartido, muchas, muchas gracias.

A todos los que creyeron en darme una segunda oportunidad.

A la Facultad, por haberme dado tantas posibilidades.

Al Laboratorio de Investigación y Desarrollo de Software Libre (LIDSOL), la comunidad del software y las artes libres, por mostrarme que no estoy loco, o que al menos no soy el único y es contagioso. Por permitirme, como dijo Isaac Newton, pararme en los hombros de gigantes, para poder ver más lejos.

A México, por ser tan bueno y aguantador, a pesar de tener políticos tan malos.

A ti, por leer este trabajo que consuma una gran cantidad de desvelos, inspiraciones y experimentos, ojalá te guste.

Alejandro Valenzuela Roca

A mi madre.
Por ser una gran mujer, una madre amorosa y por creer en mí y dejarme buscar mi propio camino, seguir mis sueños y vivir mi vida. En el amor hay confianza o no puede llamarse amor.

A Sonia.
Por ser una segunda madre, apoyarme y protegerme. Sin tí no habría sido posible que concluyera mis estudios. Con tu mano en mi hombro aprendí a sonreír.

A Sandra.
Por hacerme entender que sólo quien cae y se levanta sabe andar. No pares de andar, camina en este mundo y busca tu felicidad.

A mi padre.
Por demostrarme que el tiempo y el olvido seca la lluvia de dolores, a pesar de ser un hombre que habita donde nunca deja de llover.

A Alejandro.
Por ser una persona admirable, amable con sus semejantes, con ideales firmes y un corazón valiente. Aquel que traza su propio destino hace huella en la historia.

Para todos mis amigos.
Que a lo largo de las desveladas entendimos que mientras muchos duermen, algunos hacen sus sueños realidad.

Al lector.
Camina hacia adelante, siempre adelante, con pasos firmes y el corazón dispuesto y algún día te darás cuenta que tú eres el héroe que buscabas.

Hugo Mendieta Pacheco

Índice

Prólogo

A lo largo de la carrera de ingeniería se brinda una formación con bases teóricas muy sólidas y con aplicaciones muy específicas a esa teoría.

En cierta ocasión uno de nuestros profesores nos presentó una definición que nos pareció muy atinada de lo que es y lo que se espera de un ingeniero: “Un ingeniero es aquel que tiene ingenio para utilizar la teoría en aplicaciones que tengan un propósito para los demás. Las demás personas querrán que un ingeniero brinde soluciones o proponga mejores alternativas a las implementadas actualmente”. También sabemos que un ingeniero crea a partir de lo ya existente algo mejor, o incluso algo que no se haya hecho antes.

Las aplicaciones de la teoría van desde mejoramiento tecnológico y de procesos hasta brindar soluciones a problemáticas ya existentes o proponer nuevas cosas. Queda del ingeniero aplicar su “ingenio” en estas actividades y obtener retribución.

Es claro que el mercado del entretenimiento se ha vuelto uno de los más demandantes. Existe un gran campo de trabajo y de desarrollo profesional en la creación de tecnologías de realidad virtual, creación de videojuegos o en aplicaciones para películas, etc.

Además se sabe que el desarrollo de videojuegos es una de las disciplinas más exigentes en cuanto a conocimientos por parte de sus creadores. Son indispensables conocimientos de matemáticas, física, computación, e incluso de arte, historia y literatura para desarrollar un videojuego. Una gran variedad de disciplinas y por ende gente de muchos ámbitos trabajan en conjunto para crear un videojuego. Incluso hay quien afirma que cualquier problema de matemáticas, física o computación ya ha sido resuelto en algún videojuego.

Crear un juego no es fácil, y hacer que triunfe comercialmente es aún más difícil. Se sabe que una buena historia vende, pero sin buenos gráficos y efectos de sonido no se obtienen ventas satisfactorias; de la misma manera, una proeza tecnológica sin un buen argumento y sin factor de diversión es probable que sea un fracaso comercial. Uno de los principales problemas de desarrollo de videojuegos, aparte de los conocimientos de programación, es lo costoso que es crearlo. Para cada consola existen herramientas privativas de desarrollo, pero son inalcanzables para pequeños desarrolladores y nuevos emprendedores. Y aunque se crean aplicaciones de modo casero, son relativamente pocas las empresas que se dedican a hacer videojuegos para su venta al mercado global debido a su alto costo.

El *software* libre[1] desde hace algunos años ha demostrado ser una buena técnica de desarrollo de *software* y de mercadeo. Un producto de *software* libre puede ser financieramente costeable y remunerable además de ser competitivo con los productos provenientes de las empresas privadas. Además de esto, un producto de *software* libre brinda sus libertades inherentes a los desarrolladores lo

1 *Software* libre: *Software* cuya licencia permite su uso, redistribución, visualización y modificación de su código fuente, además de redistribución de dichas modificaciones, todo lo anterior sin costo y de manera legal.

cual serviría como una buena técnica para la diversificación y ampliación de títulos en el mercado de los videojuegos. Al reducir el costo de producción evitando pagar las licencias de las herramientas privativas[2], ayudará a que nuevos emprendedores creen empresas de desarrollo de videojuegos.

Se realizó este tema de tesis con *software* libre y para la consola[3] Nintendo DS, ya que esta es la consola portátil más vendida en el mundo, además de brindar muchas posibilidades innovadoras de aplicaciones de videojuegos. Nintendo dice que desarrolla consolas para divertir a la gente, y lo ha demostrado tanto con el DS como con el Wii[4]. Además porque una herramienta de *software* libre para el desarrollo de videojuegos en consolas portátiles crearía muchas posibilidades para otras personas, tanto en fuentes de trabajo, como para demostraciones artísticas, aplicaciones educativas, etc.

Se pretende que esta tesis ayude a realizar una diferencia en el desarrollo de videojuegos; ayudar a que más personas desarrollen sus propias aplicaciones para sus consolas sin tantas complicaciones y restricciones como hay en la actualidad y ampliar el campo de trabajo de creación de videojuegos a más sectores y a una mayor diversidad de personas.

Además en un futuro se desea que el **Framework** siga evolucionando, mejorando y de ser posible ampliarlo a otras consolas. Sabiendo las diferencias y restricciones, que sea posible crear un solo proyecto de videojuego y que éste funcione en diferentes consolas y plataformas.

Gente nueva trae consigo más ideas para crear nuevas vertientes y mejorar así a los videojuegos y abaratar sus costos.

2 *Software* privativo*: Software* cuya licencia restringe la redistribución, visualización, modificación o redistribución de modificaciones sin costo.
3 Aparato electrónico de esparcimiento, para reproducir videojuegos.
4 Consola de videojuegos de Nintendo de séptima generación.

1 - Análisis y descripción del problema

1.1 - Introducción

1.1.1 - ¿Qué es un **Framework**?

El concepto es difícil de definir por sí solo, su traducción del inglés sería “marco de trabajo”, pero aquellas personas que tengan experiencia programando lo utilizan adoptándolo como un concepto intuitivo. En otras áreas de trabajo, un **Framework** está compuesto por todas aquellas herramientas, tecnologías y procesos necesarios para resolver un problema.

El propósito de un **Framework**[5] es facilitar el trabajo del desarrollador, permitiéndole enfocarse en la aplicación deseada, y no en el funcionamiento del *hardware* o la funcionalidad de bajo nivel.

Se pueden crear aplicaciones sin usar un **Framework**, pero conforme se vaya incrementando el tamaño de los programas se necesitará crear un estándar de sintaxis, por lo que se llegará a crear un **Framework** propio.

Los **Frameworks** por lo general no están ligados a un lenguaje de programación específico y se pueden encontrar de muy diversos tipos: para el desarrollo web, aplicaciones de base de datos, etc.

Aunque su uso requiere aprender su sintaxis, sus bibliotecas y funcionalidades, a largo plazo reducirá el tiempo de programación y el mantenimiento.

Se puede decir entonces que un **Framework**[6] consiste de un conjunto específico de bibliotecas, programas y clases abstractas, diseñadas específicamente para trabajar juntas. Las bibliotecas de propósito general brindan funcionalidad como conexiones de red, de base de datos, la GUI[7], y más uso del *hardware*, además brinda compiladores, depuradores, etc. En conjunto forman el esqueleto de una aplicación adaptable a cualquier ámbito. También pueden incluir utilidades adicionales que trabajan en conjunto con las herramientas diseñadas.

5 Rada, Roy. “Framework-Based, Reengineering Software, p. 51.
6 Rogers, Gregory F. “Framework-Based Software Development in C++”, p. 3.
7 Interfaz gráfica de usuario (del inglés Graphical User Interface, 'GUI'), es un tipo de interfaz de usuario compuesta por imágenes y objetos gráficos para representar la información y acciones disponibles en la interfaz.

1.1.2 - Beneficios de usar un **Framework**

Un **Framework** está parcialmente completo para la mayoría de las aplicaciones. El trabajo del desarrollador que adopta el **Framework** es crear las partes propias de su problema específico.

Es cierto que existe una curva de aprendizaje de uso del **Framework** para poder usarlo plenamente, sin embargo, el **Framework** brinda la mayoría de las facilidades y resuelve los problemas de conexión de red, de reproducción de audio y demás características de un videojuego, por lo que el desarrollador ya no tiene que preocuparse por ese tipo de problemas.

Al evitarse la programación y la solución de esos bloques, se compensa el tiempo de la curva de aprendizaje. Y aún más, puede comprender más fácilmente otros desarrollos que usen el **Framework**.

1.1.3 - Breve historia de los videojuegos

El primer aparato considerado como un videojuego fue patentado en 1948: el "dispositivo de entretenimiento mediante rayos catódicos", un aparato que simulaba el disparo de un misil contra objetivos dibujados en relieve.

En el principio del desarrollo de los videojuegos, cercano a la década de 1960, los juegos eran mucho más simples, siendo desarrollados dentro de las universidades por las escasas personas que tenían acceso a las computadoras *mainframe*[8], en su tiempo libre.

Conforme la tecnología se fue abaratando y avanzando a la vez, las capacidades técnicas de las máquinas fueron aumentando y los juegos dejaron de ser desarrollados a escondidas para ser apreciados por un número cada vez mayor de personas.

De ser desarrollados para funcionar únicamente dentro de una computadora como un programa más, en la década de 1970 cobraron la suficiente importancia como para causar el desarrollo de computadoras expresamente diseñadas para videojuegos; dichas máquinas se instalaron en salones de juegos: las primeras arcadias. Para este entonces, el desarrollo de videojuegos había dejado de ser un pasatiempo sin seriedad para ser una industria con ganancias impresionantes y una competencia despiadada.

Posteriormente, en la década de los 80, las arcadias comenzaron a perder popularidad ante un nuevo fenómeno: las consolas de videojuegos y la producción de juegos para computadoras caseras.

8 Es una computadora central usada principalmente por una gran compañía para el procesamiento de una gran cantidad de datos.

Inicialmente las consolas caseras tenían solo unos pocos juegos disponibles, todos ellos alambrados dentro del aparato pero con el abaratamiento de las memorias ROM[9], comenzaron a aparecer consolas que aceptaban muchos juegos distintos, cada uno de éstos en un cartucho intercambiable.

En esta década también aparecen las primeras consolas portátiles. Conforme fueron evolucionando, la versatilidad de las pantallas de LCD[10], al tomar la forma de matriz de puntos en vez de estar limitadas a mostrar una serie de gráficos fijos, permitió la existencia de múltiples juegos para un mismo sistema, ya que al igual que las consolas caseras, en un principio tenían solo uno o dos juegos alambrados directamente. Estos juegos fueron también distribuidos en forma de cartuchos intercambiables.

Durante la década de 1990, la popularidad de los videojuegos se incrementó a un grado tal que provocó la aparición y extinción de varias generaciones de consolas caseras; las capacidades gráficas se incrementaron exponencialmente, al igual que las capacidades de sonido. Durante esta década también ocurrió una disminución considerable del interés del público por las arcadias, reduciéndolas prácticamente a géneros de videojuegos competitivos como las peleas, y máquinas con controles especiales que resultan demasiado caros para que una consola casera los incluya de manera rentable.

Se abarató y popularizó el *hardware* de generación de gráficos para computadoras caseras, también conocido como tarjetas de aceleración de gráficos.

En esta misma época se empezó a proporcionar a los usuarios la capacidad de generar su propio contenido; inicialmente lo principal fue la posibilidad de crear niveles propios pero conforme avanzaron los motores de videojuegos, la capacidad de crear personajes, modificar las reglas físicas e introducir series de sucesos también fueron incorporados.

Con la expansión del Internet y estandarización de las redes se popularizaron asimismo los juegos multijugador a distancia y surgió otra industria relacionada: la de juegos a través de Internet.

A partir del año 2000 han habido avances notables en el realismo de los gráficos en los videojuegos. Se habla de introducir una tarjeta de aceleración del cálculo de la física al mercado tal y como en la década pasada sucedió con las tarjetas aceleradoras de gráficos, las tarjetas de gráficos más recientes permiten correr programas que no generan gráficos pero para los cuales la arquitectura de cálculo paralelo que presentan es ideal. El Internet de banda ancha ha tenido una penetración cada vez mayor, lo cuál ha causado la aparición de nuevos géneros, tales como el de multijugador masivo en línea, y se ha buscado involucrar cada vez más al usuario en la creación de contenido propio; el kit de desarrollo XNA de Microsoft es prueba de ello; se exploran nuevas maneras de interactuar con los juegos y también se ha popularizado el desarrollo casero de videojuegos no oficiales para consolas comerciales; todo apunta a que involucrar más al usuario en el desarrollo de juegos podría definir qué casa productora resulta más competitiva e innovadora a largo plazo.

9 Memoria de solo lectura (del inglés *Read-Only Memory*), un dispositivo de almacenamiento que no permite la modificación de sus datos.

10 Pantalla de cristal líquido (del inglés *Liquid Crystal Display*) es una pantalla delgada y plana formada por un número de píxeles delante de una fuente de luz o reflectora.

1.1.4 - ¿Qué es un videojuego?

Hay muchos tipos de videojuegos: de carreras, de peleas, de estrategia, de plataformas, de disparos, con perspectiva de primera persona, etc., pero, ¿qué define un videojuego?

Un videojuego es en primer lugar, un programa de computadora; esta aplicación puede correr en una computadora normal o en una computadora especializada, en forma de consola de videojuegos o de arcadia.

Difiere de manera técnica de la mayoría de los programas para computadora, en que un videojuego simula continuamente sucesos, mientras que muchos de los programas solo actúan hasta que el usuario realiza alguna acción: mover el ratón, pulsar una tecla, etc.

Por ejemplo, un procesador de texto puede ser dejado desatendido durante horas sin que cambie su estado; si un videojuego se deja desatendido durante algunos minutos, es muy probable que al regresar el estado del juego no esté exactamente como fue dejado. Este estado interno que es representado como un mundo virtual puede ser tan complejo como un universo tridimensional o tan simple como un tablero con fichas.

Los videojuegos difieren asimismo de otras aplicaciones en el aspecto de tener reglas, que pueden ser explícitas o no, y a partir de las cuales el usuario debe realizar un esfuerzo para cumplir un objetivo.
Estas reglas pueden ser tanto lógicas como de simulación física, aunque una regla de simulación física es, en última instancia, una regla lógica también.

Una regla lógica podría ser, por ejemplo, que el jugador gana al conseguir un número específico de puntos. Una regla de simulación física podría ser que el jugador no puede traspasar ciertos objetos en el mundo virtual.

Una diferencia con respecto a los programas que realizan simulación física de manera científica, es que en un videojuego, la simulación física es inexacta e incompleta; lo suficientemente precisa para presentar resultados creíbles, pero lo suficientemente inexacta como para poder presentar dichos resultados muchas veces cada segundo en una computadora promedio. En contraste, una aplicación que realiza una simulación física científica puede demorarse incluso días en presentar resultados. En general, lo que le interesa al usuario de un videojuego son los efectos que los resultados de la simulación física producen; lo que le interesa al usuario de un programa de simulación física científica son los resultados en sí.

Otro aspecto importante es que, dentro de la mayoría de los videojuegos, existen entidades que toman decisiones que afectan el transcurso del juego y están fuera de control del jugador; dichas entidades usualmente aparecen en forma de enemigos y toman sus decisiones basados en procesos lógicos que colectivamente se conocen como Inteligencia Artificial. Los procesos de toma de decisiones que

emplean dichas entidades están presentes también en muchos otros tipos de aplicaciones: desde motores de búsqueda hasta los robots exploradores que se han enviado al planeta Marte.

Los videojuegos difieren también de las aplicaciones comunes en el hecho de tener una colección de imágenes, sonidos, música, personajes, escenarios y una historia; dicha colección es conocida como **contenido**. Mientras que en la mayor parte de las aplicaciones comunes es más costoso programar la lógica que ponerle "adornos", en los videojuegos el contenido tiene una relevancia a veces mayor que la lógica programada, al grado de que hoy en día las compañías de desarrollo de videojuegos llegan a tener un presupuesto dos veces más alto para contenido que para la programación de la lógica. Dicha "lógica programada" se conoce también bajo el nombre de **motor de videojuegos**.

Hoy en día es común encontrar varios juegos muy distintos entre sí, usando un mismo motor de videojuego con mínimas adecuaciones específicas a cada uno de ellos y como es de esperarse, el contenido es completamente distinto de juego a juego.

Se puede hacer entonces una división importante entre las piezas que conforman un videojuego; en su nivel más alto, se divide en:

- **Contenido**

- **Motor de videojuegos**

El contenido se compone a su vez de los siguientes aspectos:

- **Sonidos**
  Pequeñas muestras que se reproducen al haber colisiones, activar controles, etc. Por lo general en los videojuegos modernos son grabados y no generados artificialmente en el momento.

- **Música**
  Música de fondo que le agrega emoción y ambiente a la situación dentro del juego. Dependiendo de las capacidades de almacenamiento del medio en el cual se distribuye el videojuego, pueden ser sintetizados o pueden ser la reproducción de una grabación.

- **Imágenes**
  Las imágenes pueden ser entre otras cosas, texturas de objetos, gráficos usados en el juego y los menús.

- **Historia**
  La historia relata los sucesos que le ocurren a los personajes. Muchos de los juegos más memorables lo son por que tienen muy buena trama.

- **Personajes**
  Los personajes generalmente se dividen en dos: aquellos que pueden ser controlados por el jugador, y aquellos que no. Por lo general el jugador es el protagonista del juego y controla al personajes que es el protagonista de la historia, interactuando con los personajes que no son controlables por el jugador. Éstos pueden ser antagonistas (enemigos), ayudar al jugador o incluso decorativos, de muy variada complejidad.

- **Escenarios**
  Los escenarios son los lugares donde se lleva a cabo la acción. Por lo general, se busca representar un mundo virtual que es de cierta manera parecido al real, por lo que en los escenarios es común encontrar aquellos que se asemejan a montañas, playas, interiores de construcciones, etc.

Y como parte del motor de videojuegos, se distinguen:

- **Generación de gráficos**
  La funcionalidad de generación de gráficos consiste en un sistema que dibuje en la pantalla la geometría del mundo virtual que se desea, así como la geometría de los personajes involucrados, efectos visuales, etc.

- **Reproducción de sonidos**
  Dichos sistemas emiten la música y los efectos de sonido apropiados en el momento preciso; por lo general al reproducir dichos efectos éstos se emiten con una cierta diferencia de volumen entre los canales de audio disponibles para simular posicionamiento de la fuente de sonido con respecto al observador.

- **Simulación de física y detección de colisiones**
  Una gran cantidad de videojuegos actualmente simulan un mundo físico y requieren por lo tanto simular la interacción entre objetos dentro del mundo físico. Ello incluye detectar contactos (colisiones) entre los objetos que conforman ese mundo, al igual que simular el movimiento de dichos objetos.

- **Inteligencia artificial**
  La inteligencia artificial controla la toma de decisiones de las entidades no controladas por el jugador, tales como enemigos.

- **Comunicación a través de redes**
  Hoy en día la experiencia de competir contra otras personas de manera remota ha cobrado mucho auge, ya que es muy difícil igualar dicha experiencia con entidades controladas por la inteligencia artificial.

- **Interacción con el usuario**
  La interacción con el usuario generalmente viene dada por pulsaciones en botones y ángulos en palancas, aunque las nuevas consolas plantean controles innovadores que permiten, por ejemplo, que el personaje que el usuario controla copie los movimientos que éste realiza físicamente. En dichos casos es conveniente que el motor facilite la interpretación de dichos movimientos de manera más abstracta que, por decir, una serie de coordenadas.

Un motor de juegos es la pieza clave de *software* en un videojuego de computadora y otras aplicaciones interactivas con gráficos en tiempo real.

Un motor de juegos típicamente puede proveer las siguientes funcionalidades: generación de gráficos, uso de efectos de sonido y música, simulación de física o detección de colisiones, animación, inteligencia artificial, comunicación a través de redes y diversas optimizaciones.

En ocasiones también proveen la posibilidad de hacer funcionar con escasos o incluso ningún cambio el mismo juego en varias plataformas distintas.

Por lo general, las compañías desarrolladoras de videojuegos utilizan un mismo motor de juegos para varios juegos distintos, lo cual les permite ahorrar recursos.

Extendiendo la filosofía del reuso, hoy en día los motores se licencian completos o por piezas; algunos ejemplos de dichos motores son **Libsmog**, **Gamebryo**, **Havok** y **Unreal Engine**. Existen asimismo piezas más especializadas que pueden ser incorporadas a motores de juegos de distintos desarrolladores; ejemplos de dichas piezas son **Fork Particle**, **Bink Video** y **Wizaid**.

Operativamente hablando, un motor de juegos implementa el ciclo mostrado en la ilustración 1.

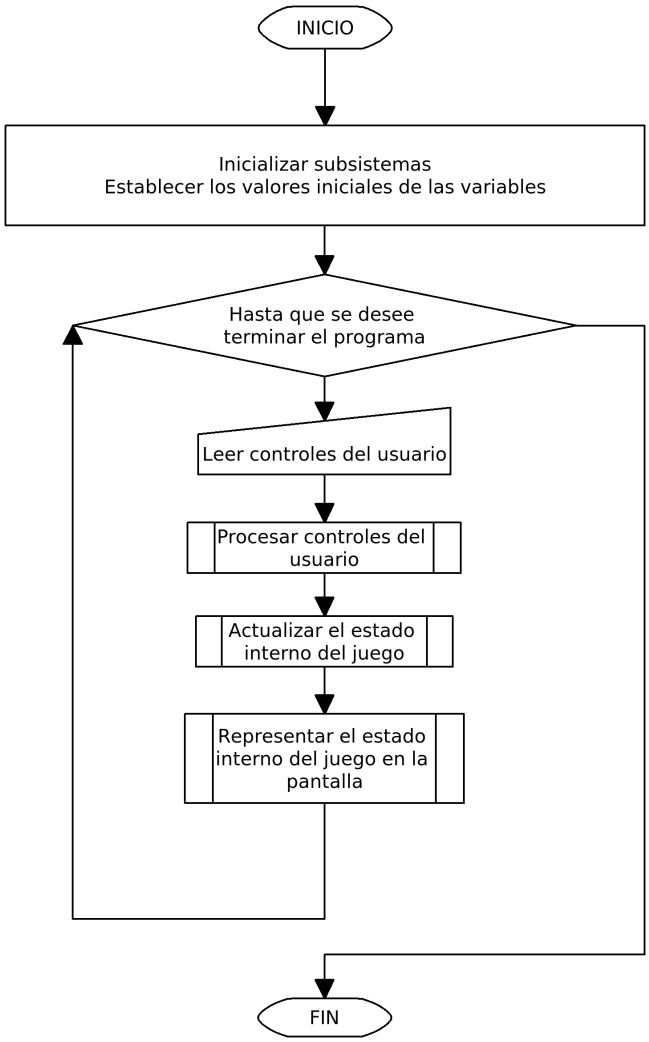

Ilustración 1: Motor de juegos, ciclo de simulación de un juego

Dicho ciclo es conocido como ciclo de simulación básico y evidencia la naturaleza iterativa del videojuego: se trata de un programa que continuamente se ejecuta y que va alterando variables físicas con cada iteración, calculando colisiones y desplazamientos, respondiendo de acuerdo a ciertas condiciones a los controles del usuario. Tras haber actualizado estas variables, el juego verifica reglas lógicas que de cumplirse pueden ser tomadas como una condición de fin de juego. Tales reglas lógicas pueden ser, por ejemplo, el número de puntos acumulados por el jugador o el haber ingresado a un lugar específico. En cada iteración los resultados son mostrados de manera gráfica en la pantalla.

El motor de videojuegos que se decidió emplear para realizar la tesis se llama MotorJ y es un proyecto realizado en el Laboratorio de Investigación y Desarrollo de Software Libre, LIDSOL.

2 - Planteamiento y descripción de la solución

Al plantear el presente trabajo de tesis se pretendió crear algo de utilidad, un desarrollo de *software* libre[11] que fuera útil para programadores y no programadores. Que cualquier persona pudiera participar, sin gran dificultad, en el desarrollo de un videojuego para la consola Nintendo DS. Al no ser posible predecir en ese momento la cantidad de trabajo y complejidad de dicha empresa, se buscó dentro de los modelos de desarrollo de *software*[12] alguno que brindara libertad de desarrollo. El modelo elegido para el desarrollo fue el de prototipos.

Como el modelo de prototipos es caracterizado por la idea de que una vez creado el prototipo es necesario volver a crear uno nuevo para mejorarlo, se buscaron las desventajas relevantes comúnmente mencionadas, las cuales son:

- ✗ El usuario tiende a crearse falsas expectativas al ver el prototipo de cara al sistema final.
- ✗ Por la velocidad de desarrollo, se desatienden puntos importantes como el mantenimiento a largo plazo.
- ✗ Por lo general el equipo de desarrollo se ve obligado a reconstruirlo una vez que el prototipo cumplió con su función.

En el desarrollo se planteó implementar algunas técnicas y costumbres para contrarrestar estas desventajas, las que pensamos establecer son:

- ✔ El **Framework** es un sistema de propósito general, y las 'pruebas de concepto'[13] únicamente verificarán la funcionalidad de las herramientas que se desarrollarán.
- ✔ La orientación del **Framework** será un desarrollo de *software* libre. Para que otros desarrolladores puedan usarlo sin complicaciones se definirán algunas técnicas y estándares durante la programación.
- ✔ La configuración general del **Framework** esta pensada para ser de propósito general y funcionar para cualquier videojuego, además si el futuro desarrollador desea modificarlo, al ser *software* libre podrá hacerlo.
- ✔ Se redactará una documentación donde se explique de forma clara la configuración y el uso de las clases, así como su ubicación dentro del **Framework**, todo esto con la finalidad de brindar la información suficiente para los programadores.
- ✔ Se usará un repositorio para un mejor trabajo en conjunto, una eficiente actualización y control de versiones del código del **Framework**.

11 Tramullas, Jesu; Garrido, Piedad. “Software Libre para servicios de información digital”, pp 1-6.
12 Rada, Roy. “Reengineering Software”, pp. 3-6.
13 Una validación de la herramienta desarrollada.

- ✔ Una vez concluida la tesis se busca liberar el **Framework** para su actualización por parte de nosotros los creadores y de la comunidad de desarrollo de videojuegos.

2.1 - Alcances

- **Obtener un motor de juegos suficientemente sencillo y completo para funcionar sobre la consola Nintendo DS**
  Se requiere un motor de juegos suficientemente sencillo y ligero pero completo, que contenga la mayor parte de las características usuales. Se decidió migrar el motor gráfico "MotorJ", un proyecto del Laboratorio de Investigación y Desarrollo de Software Libre de la Facultad (LIDSOL) a la consola, debido a su sencillez y la familiaridad que se tiene con él.

- **Uso de un 'paradigma de programación'[14] adecuado**
  El uso de un paradigma adecuado simplificará el desarrollo. En base a la investigación realizada, el más adecuado parece ser orientado a objetos, debido a que la analogía que pueden tener los distintos elementos que conforman a un videojuego con los objetos reales facilita la programación.

- **Reconstrucción del sistema de audio**
  Se pretende hacer funcionar el sistema de audio del MotorJ en el Nintendo DS. Se planteó la separación de la reproducción para la música y la de los efectos de sonido.

- **Conexiones de red inalámbrica**
  Se busca que el **Framework** disponga de conexiones red punto a punto y de punto a multipunto. Se planea que se facilite el uso de la red en las aplicaciones con el **Framework**. Se desean juegos multijugadores.

- **Uso de personajes**
  Se pretende crear una clase que sirva como ejemplo para definir personajes bípedos y que sirva también para el desarrollo de programadores en una ampliación para cuadrúpedos o aplicaciones específicas.

- **Aplicaciones de algoritmos de detección de colisiones**
  MotorJ ya incluye algunos algoritmos de detección de colisiones, sin embargo se busca que el **Framework** realice detección de colisiones más fácilmente aplicables. .

14 Un tipo de filosofía para el desarrollo de *software*.

- **Generador de terrenos**
  Para ayudar al desarrollo con el **Framework**, se piensa hacer una herramienta generadora de terrenos, que sirva como base para aplicaciones más específicas por parte de los desarrolladores.

- **Uso de la pantalla inferior táctil**
  Se busca usar la pantalla inferior y simplificar su uso.

- **Crear la prueba de concepto**
  Se planea realizar una prueba final que aplique todas las clases y herramientas desarrolladas a lo largo del proyecto; un videojuego multijugador 3D.

- **Optimización de código**
  Se sabe de antemano que la consola tiene menor capacidad que la PC, de modo que se tendrá que optimizar hasta donde sea posible los procesos.

2.2 - Limitaciones

- **La prueba de concepto final**
  Sólo ejemplificará el uso y buen funcionamiento de las herramientas desarrolladas, no tendrá historia ni concepto artístico.

- **Sólo es un Framework**
  El presente trabajo será muy general, queda a cada desarrollador realizar sus propias aplicaciones de acuerdo a las necesidades específicas de los diferentes videojuegos.

2.3 - Programación Orientada a Objetos

Para desarrollar el **Framework**, se emplearán los lenguajes C y C++ debido principalmente a las siguientes razones:

- El lenguaje de programación[15] C permite hacer aplicaciones muy eficientes en el uso de recursos, debido a que permite utilizar comandos de bajo nivel con comandos de alto nivel y la

15 Lenguaje que puede ser interpretado o compilado por una máquina y que le permite a ésta seguir órdenes específicas para realizar acciones útiles.

administración de la memoria queda bajo la responsabilidad del programador. En el caso de un videojuego, la velocidad de ejecución y la economía de recursos son un factor importante, incluso decisivo, si se trata de un sistema portátil[16].

- La biblioteca **Libnds** está desarrollada en C, probablemente por la razón anterior.

- La familiaridad que se tiene con el lenguaje C aprendido durante la carrera facilita asimismo la programación.

- El lenguaje C++ toma la mayor parte de la sintaxis, comandos y palabras reservadas del lenguaje C y lo extienden; de esta manera el lenguaje C++ fue diseñado para ser compatible con C.

- Las características de programación orientada a objetos del lenguaje C++ permiten una adecuada abstracción y simplifican la labor de la programación, ya que casi todos los objetos que se busca utilizar dentro de los videojuegos tienen un símil en la vida real.

2.3.1 - Características de la programación orientada a objetos

La programación orientada a objetos surgió a partir de la década de 1960 como una manera de intentar hacer al desarrollo de *software* más sencillo, de mayor calidad y obtener código más fácilmente reutilizable. Siendo una extensión del lenguaje de programación Algol, Simula fue el primer lenguaje que introdujo los conceptos de programación orientada a objetos: clases, subclases, métodos y encapsulamiento. El lenguaje de programación Smalltalk fue sin embargo el primero en ser llamado "orientado a objetos". Hoy en día la programación orientada a objetos tiene mucho auge y existen diversos lenguajes de programación orientada a objetos tales como Java y C#.

En la programación orientada a objetos, cada objeto puede ser considerado como una pequeña máquina independiente con una responsabilidad definida. Los comportamientos o acciones que dicha máquina puede realizar están relacionadas con el tipo de objeto que es. A diferencia de otros paradigmas de programación, en la programación orientada a objetos las estructuras de datos llevan consigo las operaciones necesarias para utilizar los datos, en vez de ser considerados por separado los datos y los comportamientos.

Los conceptos fundamentales de la programación orientada a objetos son:

- **Clase**
  Una clase define las características abstractas de un objeto, conocidas como propiedades, al igual que sus acciones o comportamientos, conocidos como métodos. A estas propiedades y

16 Glaeser, Georg. "Open Geometry", p 5.

métodos se les llama miembros de la clase. Las clases proveen modularidad y estructura en un programa, además de encapsulamiento.

- **Objeto**
  Cuando se realiza una instancia de una clase en tiempo de ejecución, se le conoce como objeto. Un objeto es, por lo tanto, una versión de una clase con características particulares.

- **Método**
  Las acciones que puede realizar una clase se llaman métodos. Por lo general están muy relacionadas con la naturaleza de la clase.

- **Herencia**
  La herencia ocurre cuando se crea una subclase que proviene de una clase, como en una analogía, de hijo a padre. Una subclase es una clase que adquiere todas las propiedades y métodos de la clase de la que proviene, es decir, hereda las propiedades y métodos presentes en el padre. Una subclase generalmente tiene propiedades y métodos adicionales a los de la clase padre.

- **Polimorfismo**
  Además de poder definir métodos adicionales, una subclase también puede cambiar la manera como realiza los métodos presentes en la clase padre.

- **Encapsulamiento**
  Una clase puede definir propiedades o métodos que el programador puede modificar o activar de manera externa, pero en muchos casos es necesario también definir propiedades y métodos que solo pueden ser modificados o activados por la misma clase. Cuando dichos miembros son visibles de manera general, se les conoce como miembros públicos; cuando no son visibles más que por la misma clase, se les llama miembros privados. En algunos lenguajes de programación existe una tercera clasificación llamada miembros protegidos, los cuales son visibles únicamente a la clase y a sus subclases. El poder tener miembros públicos y privados sirve para tener un mayor control sobre el comportamiento general del programa, ya que es más predecible, y también tiene como ventaja el que no sea necesario ver el código interno de la clase para poder ocuparla. Esto último facilita el trabajo al desarrollar programas en equipo, ya que se puede acordar entre programadores una serie de propiedades y métodos públicos que permanezcan constantes, mientras que el código interno de la clase pueda cambiar todo lo necesario para corregir errores y optimizar su funcionamiento.

2.3.2 - La programación orientada a objetos aplicada a los videojuegos

En un videojuego se tienen grandes beneficios al utilizar programación orientada a objetos, ya que al implementar un mundo virtual, se tienen símiles[17] de una gran parte de los entes que participan; por ejemplo, es posible considerar al terreno, los obstáculos físicos e incluso los enemigos como objetos y hacer que todos interactúen con el jugador de maneras distintas pero usando un método con el mismo nombre para todos, tomando en cuenta que la idea de la interacción puede ser la misma entre todos los objetos, aún cuando éstos no tengan un "ancestro" en común, es decir, no empleen herencia. La detección de colisiones podría ser un caso de dicho ejemplo.

Otro beneficio podría ser, esta vez ocupando herencia, tener una serie de enemigos que fueran muy similares entre sí, todos con un mismo ancestro que tuviera los métodos y propiedades indispensables a todos ellos, de manera que según se requiriera se fueran adicionando o modificando propiedades y métodos. Las ventajas son una simplificación del código, además de economía, ya que mientras no sean modificados los métodos de una clase padre a hijo, se seguirá usando el mismo código del padre.

Aún clases de objetos más abstractos que no necesariamente tengan una analogía física, como la toma de decisiones de un ente no controlado por el usuario, pueden ser considerados como objetos.

Por otro lado, hoy en día un videojuego está hecho por un gran número de personas que necesitan una adecuada coordinación. El emplear programación orientada a objetos permite delegar responsabilidades, acordando únicamente en aquellas propiedades y métodos que permanezcan públicos y permitiendo cierto grado de libertad a la persona que realiza la implementación. También es una manera de promover una cierta uniformidad en el diseño si se definen ancestros comunes para tipos de objetos similares.

Actualmente este paradigma de programación se aplica en diversos sistemas de creación de videojuegos. Algunos de estos sistemas son Unity3D, desarrollado en el lenguaje Mono C#, SimpleJ en el lenguaje Java, Vicious Engine en C++ y Microsoft XNA en Microsoft C#.

17 Eberly. David. "3D Game engine design", p 441.

3 - Diseño

El **Framework** es un prototipo; su desarrollo estará basado en un proyecto desarrollado en las materias relacionadas con computación gráfica de la carrera de Ingeniería en Computación, llamado MotorJ (se proporcionará más información sobre dicho proyecto más adelante), el cual es un motor gráfico para la PC. Se buscará hacer una implementación más general y crear herramientas específicas para el DS y construir así el **Framework**.

Existen dos corrientes de desarrollo para la consola. La primera emplea el SDK[18] privativo de Nintendo. Éste brinda todas las herramientas necesarias para crear juegos, pero al mismo tiempo es totalmente restrictivo en cuanto a derechos y utilización. No cualquier persona esta autorizada por Nintendo para utilizarlo, además de tener un precio realmente exorbitante sólo financiable por algunas empresas creadoras de videojuegos con varios años en el mercado. La segunda corriente son las bibliotecas libres **Libnds**, creadas por Michael Noland y Jason Rogers, quienes implementaron funciones muy similares a las de OpenGL[19] y en la que se han basado la gran mayoría de los desarrollo caseros.

Aún así es muy difícil realizar implementaciones a partir de **Libnds**, por lo que el presente **Framework** basado en **Libnds** funcionará para simplificar su implementación y poder realizar videojuegos de manera más sencilla.

Se desconoce la complejidad del proyecto, quizá en el trayecto se utilicen más herramientas libres para cumplir con los objetivos, sólo se integrarán al desarrollo, unificándolos bajo los estándares que se proponen en este trabajo y bajo las características que se necesiten. Todas las herramientas, bibliotecas o programas usados serán mencionados durante este capítulo y como es que fueron empleados y adaptados.

A partir del modelo de prototipo se buscará cumplir con los objetivos para realizar un **Framework** fácil de usar y aplicable al desarrollo de videojuegos en la consola. También se dedicará especial atención a la explicación de cómo es que fueron desarrolladas las pruebas y en el apéndice de las APIs[20], como es que se usan los módulos, funciones o clases.

Los pasos que se seguirán para aplicar al desarrollo, basados en el modelo de prototipos, son los siguientes:

18 Kit de Desarrollo de *software* (del inglés *software* Development Kit), se le considera a un conjunto de herramientas para que un programador cree aplicaciones con una orientación específica, por ejemplo Frameworks, *hardware* de simulación, sistemas operativos, etc.

19 Biblioteca de gráficos abierta (del inglés Open Graphics Library), es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D.

20 Interfaz de Programación de Aplicaciones (del inglés Application Programming Interface) es un conjunto de funciones, procedimientos o métodos que ofrece cierta biblioteca para ser utilizado por otro *software* como una capa de abstracción.

1. Definir los alcances del **Framework**.
2. Ver los requerimientos de cada alcance para trabajarlos individualmente.
3. Investigar cómo implementarlo.
4. Diseñar la clase o clases que satisfagan al requerimiento.
    - Al realizar una prueba general de las clases, pasar al punto 5.
    - Realizar pruebas a la clase y corregir errores.
    - Hacer una revisión de estándar a la nueva clase.
    - Actualizar el repositorio del **Framework**.
    - Regresar al punto 2 o pasar al punto 4.
5. Crear una prueba con las clases existentes.
    - Al realizar la prueba de concepto final, pasar al punto 6.
    - Probar el buen funcionamiento e interacción de las clases.
    - Hacer una revisión de posibles fallas y corregirlas.
    - Actualizar el repositorio del **Framework**.
    - Regresar al punto 2.
6. Realizar la prueba de concepto final.

La ilustración 2 muestra el diagrama modificado del modelos de prototipos que se empleará para el **Framework.**

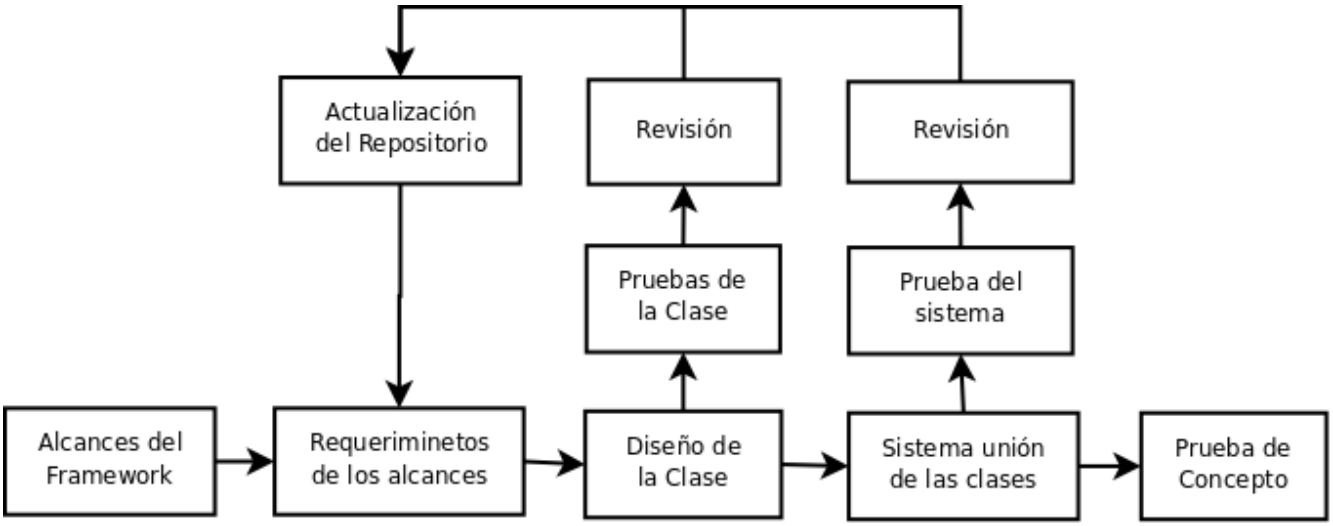

Ilustración 2: Modelo de Prototipos

Se busca aplicar este modelo durante el desarrollo y documentar el trabajo hecho durante las pruebas del **Framework** y sus módulos. Durante la fase de pruebas se observará como es que se realizó el trabajo.

Se pretende que la tesis sirva como base y documentación para los desarrollos y aplicaciones usando este **Framework**, intentando brindar la mayor cantidad de información bien documentada para los futuros desarrolladores.

Los siguientes subsistemas indicarán como es que se realizó el trabajo y cuales fueron las decisiones de implementación y una breve información de uso. La información completa de su uso estará en el API contenida en el Apéndice de la tesis.

3.1 - Desarrollo

3.1.1 - Uso de la programación orientada a objetos en el **Framework**

El **Framework** hará uso de la programación orientada a objetos principalmente como un recurso para simplificar sus aplicaciones, ya que es más fácil entender como funcionan las diferentes piezas que lo conforman, teniendo como analogía al mundo real, en el cual existen objetos que pueden realizar acciones y que tienen características, en vez de solo estructuras de datos como ocurre en la programación estructurada clásica.

Asimismo se intentará no abusar dicho paradigma, evitando usarlo en los casos en que simples funciones sean más fáciles de comprender y menos laboriosas de emplear por el programador.

Se tiene, por ejemplo, la clase que define el reproductor de música, **mjMusicPlayer**. Dicha clase podría ser considerada una versión "virtual" de un reproductor de cassettes o de discos compactos, ya que tiene esencialmente las mismas características y se pueden efectuar acciones parecidas sobre él.

De igual manera, existen varias clases más que forman parte del **Framework** como **mjNetworkCTL**, la cual realiza el control de la red inalámbrica del Nintendo DS, que no tienen un símil en el mundo real. Sin embargo el código resulta más fácil de entender como clase que como una serie de funciones, ya que existe un orden específico para su invocación al depender unas de otras.

Aplicando los conceptos de herencia, se tiene la clase **Universe**. Todos los juegos deben implementar al menos una subclase de dicha clase, ya que ésta tiene una serie de propiedades y métodos que resultan indispensables para el ciclo de simulación del motor de videojuegos usado, MotorJ. Cada subclase de universo define qué acciones realiza en los métodos señalados por **Universe**, además de poder definir propiedades exclusivas a dicha subclase a manera de variables.

Por otro lado, en la biblioteca **support**, la cuál tiene funciones matemáticas y de álgebra vectorial, no se encontró ningún beneficio en establecer un objeto que proveyera dichas acciones, por lo tanto se optó por no crear clase alguna.

3.1.2 - Motor de videojuegos MotorJ

Nuestro **Framework** tiene como pieza central un motor de videojuegos desarrollado a lo largo de los últimos semestres de la carrera por los autores, como un proyecto del Laboratorio de Investigación y Desarrollo de Software Libre, LIDSOL.

MotorJ es un motor de videojuegos programado en lenguaje C++ y es libre bajo la definición de la GNU[21], ya que su licencia contiene las 4 libertades que de acuerdo a esta organización, deben ser contempladas en un *software* para que éste sea considerado como "libre".

MotorJ se apoya a su vez en bibliotecas con licencias libres, permitiendo al programador y al usuario final realizar ajustes a fin de poder adaptar el *software* a sus necesidades particulares sin impedimento legal alguno, esto simplifica y promueve la reutilización y refinamiento del código por parte de todos los interesados.

MotorJ consta de varios módulos que cubren los principales sistemas que conforman un videojuego:

- Video
- Sonido
- Interacción con el usuario
- Red
- Simulación
- Inteligencia Artificial

El sistema de video está proporcionado por los archivos *main-universe.cpp*, y *app.cpp*; dichos módulos se encargan, una vez compilados, de inicializar el sistema de video y la máquina de estados usada por el API OpenGL.

El sistema de sonido está proporcionado por la clase mjMusicPlayer, contenida en el archivo *musicplayer-class.cpp,* y la clase mjWaveSample, contenida en el archivo *wavesample-class.cpp*. La primera clase está aunada a la variable global App que permite reproducir música de fondo, en formato Ogg Vorbis monoaural de calidades inferiores a "3" en la escala interna del mismo formato. Estas restricciones son debido a las limitantes de *hardware* que tiene el Nintendo DS en cuanto a capacidad de procesamiento y a memoria.

21 Acrónimo recursivo que significa GNU No es Unix (del inglés GNU is Not Unix), es un proyecto iniciado por Richard Stallman con el objetivo de crear un sistema operativo completamente libre.

Internamente, mjMusicPlayer utiliza la biblioteca libre *Tremor* de xiphophorus ligeramente modificada para funcionar en el Nintendo DS y realizar la decodificación del formato Ogg Vorbis.

La clase mjWaveSample permite abrir sonido en formato WAVE de 8 ó 16 bits, y con tasas de muestreo de 22,050 o 44,100 Hz, según las especificaciones de archivos WAVE de Microsoft. También está restringido a archivos monoaurales, sin embargo la clase proporciona un método para reproducir sonidos simulando el posicionamiento tridimensional, dentro de las capacidades del aparato, de la fuente de sonido acorde a lo que se ve en pantalla, por lo que utiliza las dos bocinas del Nintendo DS simultáneamente.

La interacción con el usuario está proporcionada a través de *main-universe.cpp* mediante las funciones *keysHeld* y *touchXY* de la biblioteca **Libnds**.

Los servicios de red están proporcionados a través de la clase mjNetworkCTL, contenida en el archivo *networkctl-class.cpp*. mjNetworkCTL se apoya a su vez en *dswifi*, una biblioteca libre que permite ocupar el sistema de red inalámbrica con los lenguajes C y C++ sin necesidad de recurrir a código de bajo nivel.

La simulación es realizada en cada ciclo de juego del motor. Para ello, se proporcionan varias funciones de detección de colisiones en el archivo *collisions.cpp*, y una clase, mjColTree, contenida en el archivo *coltree-class.cpp*.

La clase mjColTree se encarga de concatenar varios algoritmos de detección de colisiones, de manera que se puedan ahorrar pasos innecesarios cuando hay muchos objetos con los cuales se debe detectar colisión, anteponiendo uno o más algoritmos de detección de colisiones sencillo a uno complejo y más costoso de calcular, de manera que este último solo se realice cuando sea pertinente.

3.1.2.1 - El funcionamiento del MotorJ: el sistema de “universos”

MotorJ, implementa el ciclo de simulación de un videojuego a través del concepto de “universo”. En MotorJ, existe una clase Universe que se conforma casi exclusivamente de métodos virtuales, es decir, éstos métodos están declarados pero no están implementados, de manera que establece la forma de todas las subclases que se formen a partir de Universe.

La clase Universe tiene la siguiente estructura:

```
class Universe
{
      public:
            virtual void Init() {};
            virtual void ProcessEvent(s_event & event) {};
            virtual void Update(float t_trans) {};
            virtual void Redraw() {};
            virtual void Cleanup() {};
            Universe * GetNextUniverse();
            void SetNextUniverse(Universe * next);
            bool mustInit;
            bool mustCleanup;
      private:
            Universe * NextUniverse;
};
```

Cada juego entonces debe crear al menos una subclase de Universe y puede crear tantas subclases como requiera, mientras implemente todos los métodos que están marcados como virtuales en Universe.

Lo que se está haciendo en realidad, es implementar ciclos de simulación diferentes para cada universo, ya que el funcionamiento de MotorJ puede ser resumido mediante la ilustración 3.

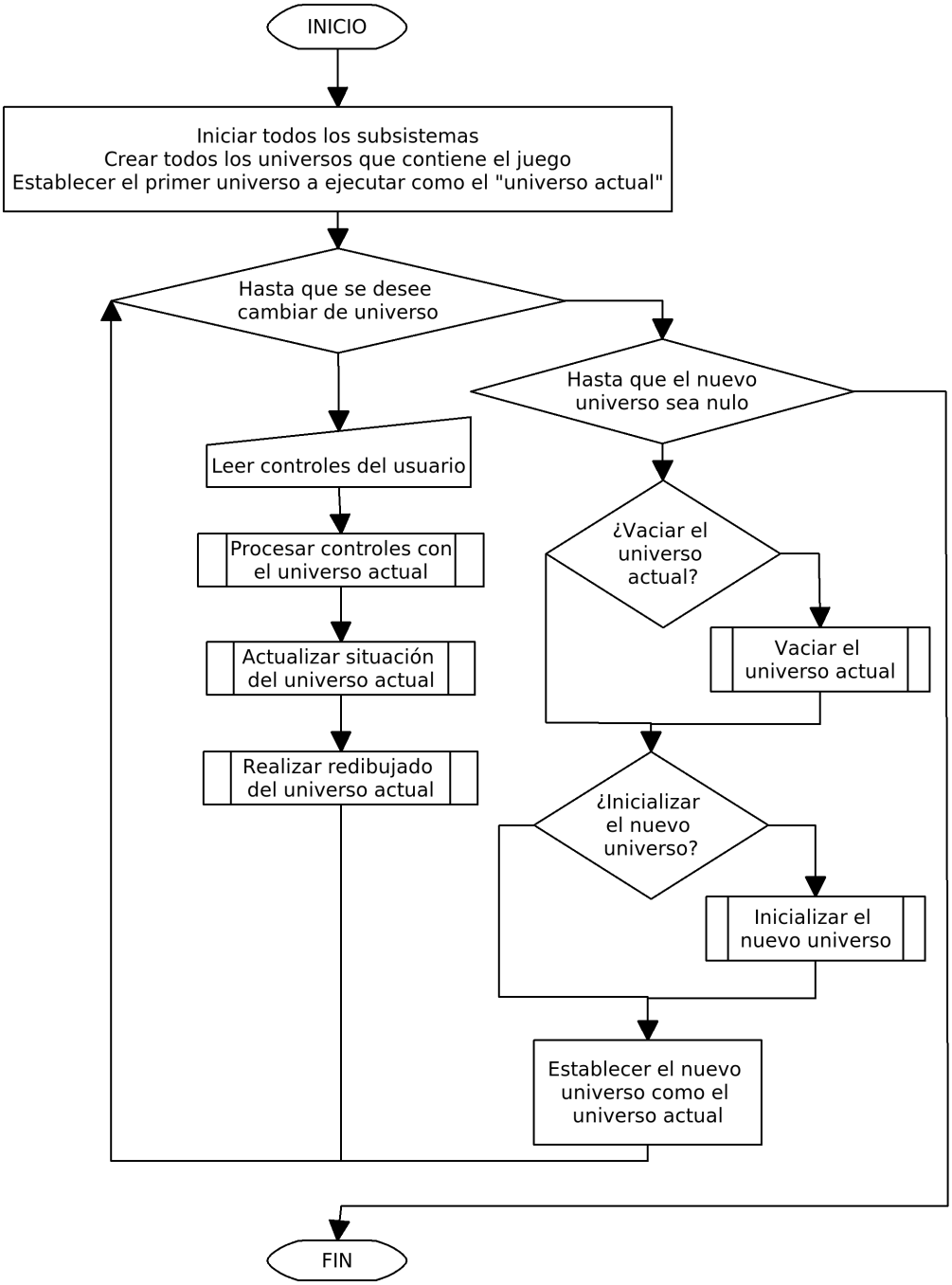

Ilustración 3: Ciclo de juego de MotorJ

La idea detrás de implementar esta subclase, es que se tenga un universo por cada "modo" de juego, es decir, que si se tienen varios escenarios con las mismas reglas físicas y en los cuales se responde a los controles de la misma manera, se implemente un solo universo que pueda emplear varios escenarios con una sola implementación de reglas físicas y respuesta a controles.

De esta manera, si se tuviera como ahora es común en muchos videojuegos, un mini-juego con reglas muy distintas a las del resto del juego, se implementaría su propio universo con sus propias reglas, ya que el programador es libre de cambiar de un universo a otro cuando lo requiera mediante el método **SetNextUniverse()**.

El programador puede asimismo decidir entre dejar las variables del universo sin modificar al cambiar de un universo a otro, de manera que al regresar al primer universo, éste se encuentre exactamente como se dejó.

En dicha subclase, el método **Init()** debe inicializar todas las variables pertinentes al universo actual, así como el sistema de video, en caso de ser aplicable. Este método será ejecutado por primera vez al crear el universo y también cada vez que se cambie de universo y se regrese a este, si la variable **mustInit** tiene el valor **true**.

El método **ProcessEvent()** recibe los datos de los controles. Con esta información, se deben modificar variables que posteriormente serán tomadas en cuenta en el método **Update()**.

En el método **Update()**, tomando en cuenta los cambios realizados en los controles y registrados en **ProcessEvent()**, se realiza la actualización de todas las variables físicas, tales como posiciones, velocidades y aceleraciones, así como la toma de decisiones por parte de las entidades de inteligencia artificial, detección de colisiones y la verificación de reglas lógicas. Cuando se trata de un juego multijugador, también es el lugar ideal para realizar la comunicación por red.

El método **Redraw()** debe ser el encargado de representar, mediante comandos de OpenGL, la situación del universo actual en la pantalla.

Estos últimos tres métodos serán ejecutados en ese orden continuamente, hasta que el usuario decida detener el juego o se cumpla la condición de salida. Cuando MotorJ se ejecuta en una computadora personal, el número de veces que se ejecutan estos tres pasos generalmente está limitado a 60 veces cada segundo. Cuando MotorJ se ejecuta en el Nintendo DS, está limitado a la tasa de actualización de las pantallas de dicha consola, lo cuál da como resultado un número muy similar a 60 veces por segundo. En ambos casos, la complejidad de los cálculos requeridos durante el método **Update()** o **Redraw()** puede ocasionar que se ejecute un número menor a 60 veces por segundo, siendo el número mínimo aceptable de cuadros por segundo, 30. Si la tasa de cuadros por segundo desciende de 30, es conveniente realizar optimizaciones en el código para liberar al procesador de cálculos innecesarios.

Finalmente, el método **Cleanup()** debe liberar toda la memoria reservada de manera dinámica. Este método será ejecutado al cambiar del universo actual a otro si la variable **mustCleanup** está establecida al valor **true**.

3.1.2.2 - Bibliotecas y clases de MotorJ auxiliares al desarrollo de videojuegos

MotorJ incluye diversas funciones y estructuras en sus distintas bibliotecas, relacionadas con el desarrollo de videojuegos; algunas son exclusivas a la PC o al Nintendo DS.

Al final del presente trabajo, se incluye un apéndice con los datos técnicos sobre el uso de dichas bibliotecas, al igual que las clases realizadas.

Las bibliotecas son:

- **cam_ctl.h**
  Provee una cámara seguidora automática de tercera persona, con varios parámetros ajustables.

- **collisions.h**
  Contiene funciones de detección de colisiones básicas, tales como traslape de esferas, rayo contra triángulo y cajas orientadas, dichas funciones se explican más adelante.

- **data_structs.h**
  Contiene una lista doblemente ligada genérica, con algunas optimizaciones para acelerar las búsquedas de un elemento particular.

- **events.h**
  Contiene varias definiciones y estructuras para ocupar más fácilmente las señales recibidas desde el teclado cuando se ejecuta en una PC, al igual que definiciones y estructuras que son útiles cuando se ejecuta en un Nintendo DS. Al estandarizar los controles, facilita el realizar proyectos que puedan funcionar en ambas plataformas con pocos cambios.

- **lanjobot.h**
  Contiene un humanoide articulado formado por cubos. Dicho humanoide sirve como objeto de prueba para representar personajes de manera genérica.

- **objects.h**
  Contiene varias figuras geométricas genéricas para ocuparlas en programas de prueba.

- **sound.h**
  Contiene funciones para cargar archivos de sonido en formatos Ogg (para música) y WAVE (para efectos de sonido). Estas funciones solo están disponibles en la versión para PC de MotorJ, sin embargo, su funcionalidad ya fue reemplazada en el Nintendo DS por las clases **mjMusicPlayer** y **mjWaveSample**, respectivamente.

- **stenciltricks.h**
  Contiene funciones que realizan distintos efectos visuales como cortinillas animadas. Esta biblioteca solo funciona en la versión de MotorJ para PC.

- **support.h**
  Es la biblioteca más importante de todo MotorJ, ya que contiene funciones de álgebra vectorial ampliamente usadas, entre otras también comunes.

- **textures.h**
  Contiene funciones de carga de texturas; está muy enfocada a ser usada junto con las funciones normales de texturas de OpenGL. Por el momento, solo están disponibles para PC.

Las clases son:

- **biped-class.h**
  Contiene la clase "bípedo animado", usada para animar de manera genérica personajes humanoides creados en Blender[22], de una manera muy parecida al **lanjobot**. De momento, solo está disponible para Nintendo DS.

- **clientsocket-class.h**
  Contiene la clase "*Socket*[23] de cliente", la cual sirve para facilitar la comunicación en *sockets* de tipo cliente para los protocolos TCP y UDP. Esta clase no está disponible de momento en la plataforma Windows ya que ocupa *sockets* POSIX.

- **coltree-class.h**
  Contiene la clase "Árbol de colisiones", la cual implementa una estructura para realizar detección de colisiones de manera jerárquica.

- **musicplayer-class.h**
  Contiene la clase "Reproductor de música". Dicha clase permite a los desarrolladores emplear archivos monoaurales de tipo Ogg Vorbis como música de fondo en los juegos. De momento, esta clase solo está disponible para el Nintendo DS.

- **networkctl-class.h**
  Contiene la clase "Control de red", la cual permite iniciar y detener la red inalámbrica del Nintendo DS de manera sencilla. Esta clase no está disponible para la PC debido a que no tiene uso alguno en dicha plataforma.

- **universe-class.h**
  Contiene la clase "Universo"; dicha clase implementa un ciclo de simulación particular, siguiendo el esquema definido por MotorJ.

22 Programa multiplataforma, dedicado especialmente al modelado y creación de gráficos tridimensionales.
23 La comunicación o flujo de datos entre dos dispositivos distintos, ya sean computadoras u dispositivos de comunicación.

3.1.2.3 - Más información sobre MotorJ

El proyecto MotorJ se encuentra hospedado en un repositorio ubicado en el **servidor de código de LIDSOL**[24], bajo el control de versiones Subversion[25]. Dicho repositorio es de acceso público para lectura; al ser un proyecto libre, se trata de estimular la participación voluntaria de todas las personas a las cuales dicho desarrollo les pueda resultar útil. Es por esta misma razón que se decidió desarrollarlo en inglés: para tratar de realizar un proyecto que tuviera un alcance y una colaboración que no estuviera limitado por las barreras del idioma, porque además hoy en día se compite en un entorno internacional, sobre todo al tratarse de videojuegos.

La documentación de dicho proyecto se está realizando asimismo en un sistema colaborativo tipo *Wiki*[26], también en inglés, en el **servidor de documentación de LIDSOL**[27].

En el apéndice se incluyen instrucciones para obtener y trabajar con el motor, así como los demás desarrollos que conforman el **Framework**.

3.1.2.4 - Inteligencia Artificial dentro de los videojuegos

En la actualidad cualquier decisión o acción que tome un jugador dentro de un entorno virtual, debió ser contemplada por el desarrollador, el videojuego fallaría si pasara algo que no se hubiera tomando en cuenta y programado previamente. Para contrarrestar las acciones de los jugadores y dar una sensación de interacción con los personajes y entes del videojuego se han utilizado técnicas como máquinas de estado. A pesar de que responde a las acciones o eventos ocurridos sobre dicho ente virtual, ésto no es Inteligencia Artificial (IA). Existe una diferencia entre lo que los desarrolladores de videojuegos piensan por IA y lo que dice la academia. En la mayoría de los juegos actuales la IA es prácticamente inexistente y sólo responden a una serie de decisiones simples.

Los personajes de los juegos no aprenden por sí solos, es el programador el que definió que deben hacer en tales circunstancias. Sin embargo ya existen algunos trabajos de aplicación de la IA dentro de los videojuegos o mundos virtuales. Aplicando técnicas de Redes Neuronales, Algoritmos Genéticos y sistemas de aprendizaje, se han logrado obtener algunos avances en el campo.

24 Repositorio de código de LIDSOL: http://svn.lidsol.net/

25 Sistema que permite preservar distintas versiones de un mismo desarrollo, así como avanzar y retroceder en dichas versiones, lo que facilita la detección y corrección de errores introducidos, así como la colaboración de varias personas de manera simultánea.

26 Sistema web cuyo fin es permitir a cualquier persona modificar el contenido de las páginas web que lo conforman, de manera que sea fácil colaborar en la generación de contenido útil.

27 Servidor de documentación de LIDSOL: http://wiki.lidsol.net/

Las aplicaciones son algo sencillas en la mayoría de los casos, la principal actividad que buscan es entablar una interacción con los humanos y dar soluciones en tiempo de ejecución a las decisiones tomadas por los jugadores. Como entablar conversaciones, planear estrategias de defensa o ataque, etc.

Para cada videojuego la IA es muy particular, incluso con herramientas tradicionales como las máquinas de estado, no es posible definir la misma aplicación que funcione para todos los eventos y circunstancias. Es responsabilidad de los desarrolladores definir sus requerimientos y como es que los entes responderán.

La IA es un potencial que pudiera revolucionar los videojuegos de cualquier género. Sin embargo aún es un campo de investigación y falta más desarrollo para hacer que dé buenos resultados para poder hacer aplicaciones para cada diferente tipo de videojuego.

3.1.3 - Subsistema de Video

El desarrollo no oficial (*Homebrew*) de la consola Nintendo DS está soportada por una biblioteca llamada **Libnds** creada por Michael Noland y Jason Rogers y se actualiza gracias a la labor de Dave Murphy. Esta biblioteca fue creada como una alternativa de código libre de su homónimo oficial el *SDK* de Nintendo. La biblioteca soporta casi todas las características de la consola como el *hardware* 2D y 3D, la pantalla táctil y el micrófono. Esta biblioteca incluye un pequeño conjunto de funciones muy similares a las de OpenGL.

OpenGL, del inglés *Open Graphics Library*, es una biblioteca de funciones multilenguaje y multiplataforma que permite producir gráficos 2d y 3D. Creada por Silicon Graphics en 1992. Por no estar apegado a ningún *hardware* ni Sistema Operativo además de poseer una gran estabilidad y rendimiento, es ampliamente usada en aplicaciones CAD, videojuegos, películas, simulaciones, métodos científicos y médicos.

La biblioteca de OpenGL describe cómo deben ser las funciones y que comportamiento deben tener, a partir de esta especificación lo fabricantes de *hardware* crean sus propias implementaciones, en otras palabras, sus propias bibliotecas que se vinculan con las de OpenGL, usando la aceleración del *hardware* en la medida de lo posible.

Los propósitos de OpenGL incluyen evitar las diferencias del *hardware* que requieran ciertas adecuaciones o incluso un completo cambio de las funciones, además de ocultar la complejidad del funcionamiento de las tarjetas gráficas, dando a los programadores una única biblioteca uniforme. OpenGL recibe una serie de procedimientos de bajo nivel, y primitivas tales como puntos, líneas o polígonos, además de los pasos como rotaciones, traslaciones o escalamientos que se necesitan para renderizar una escena, es decir, convertirlas en pixeles en la pantalla.

La ventaja de **Libnds** es que se asemeja mucho a las funciones de OpenGL, de modo que no se tuvieron demasiadas dificultades para hacer compilar el MotorJ para el Nintendo DS, exceptuando algunos subsistemas como ya se mencionó con anterioridad.

3.1.3.1 - Sistema de Video del Nintendo DS

Las diferencias con el OpenGL normal y con la versión de **Libnds** radican en las diferencias de *hardware*. El Nintendo DS tiene menos capacidad que una tarjeta gráfica moderna para computadoras de escritorio.

Libnds soporta funciones para la creación de geometrías y renderizado, se muestran algunos principales ejemplos en las siguientes listas:

Creación de polígonos relacionada con las función **glBegin**():

- **GL_TRIANGLES**: Cada tres vértices[28] se define un triángulo.
- **GL_TRIANGLE_STRIP**: Cada tres vértices se define un triángulo y los subsecuentes con sólo un vértice adicional..
- **GL_QUADS**: Cada cuatro vértices define un cuadrángulo.
- **GL_QUAD_STRIP**: Cada cuatro vértices define un cuadrángulo, y los siguientes cuadrángulos con la definición de dos vértices más.

Sólo existen estos dos tipos polígonos por que el Nintendo DS únicamente soporta triángulos o cuadriláteros.

Algunas funciones que afectan al uso de materiales y texturas, con la función **glMaterialf**():

- **GL_AMBIENT**: Define el color del ambiente para el material cuando la luz no recae sobre la cara.
- **GL_DIFFUSE**: Define el color difuso para el material cuando la normal de la cara es vista a la luz.

28 El elemento fundamental de la que está formado un gráfico. El punto común de los dos lados de un ángulo para una geometría.

- **GL_AMBIENT_AND_DIFFUSE**: El color del material en ambos casos.
- **GL_SPECULAR**: Define el color especular del material.
- **GL_SHININESS**: Define la brillantez del color del material.
- **GL_EMISSION**: El color que es independiente de las normales y la luz.

Estas funciones afectan al uso de materiales, si uno define materiales, debe definir las normales de las caras, así como la manera en que las recae la luz sobre ellas.

Funciones para texturizar, relacionadas a las funciones **glTexImage2d**() y **glTexParameter**()

- **TEXTURE_SIZE_#**: Donde '#' puede ser desde 8 hasta 1024 *texels*[29].
- **TEXGEN_OFF**: Se usan coordenadas de texturas inmodificables.
- **TEXGEN_NORMAL**: Se define una normal para la matriz de la textura., usada para texturizado esférico.
- **TEXGEN_POSITION**: Se igualan las coordenadas de la textura a los vértices.
- **GL_RGB32_A3**: Paleta de 32 bits y 3 colores de *alpha*[30] (transparencia).
- **GL_RGB4**: Paleta de 4 colores .
- **GL_RGB16**: Paleta de 16 colores .
- **GL_RGB256**: paleta de 256 colores .
- **GL_COMPRESSED**: Textura comprimida .
- **GL_RGB8_A5**: Paleta de 8 colores y 5 bits de *alpha*
- **GL_RGBA**: 15 bits de colores directos y 1 bit para el *alpha*.
- **GL_RGB**: 15 bits de colores directos y se define manualmente el *alpha*.

Libnds también define algunas funciones especiales para un tipo de estructura llamada *Display List*. En el **Framework** se realizó una implementación especial de dicha estructura, que se explicará más adelante.

29 Pixels pertenecientes a una textura.
30 Canal del RGB para definir niveles de transparencia.

En el **Framework** sólo se desarrolló el video 3D en la pantalla superior, se quedó para futuras actualizaciones el uso de la pantalla inferior para despliegue 2D, como lo hacen la mayoría de los juegos comerciales.

Para el **Framework** no se implementó la carga de texturas de manera dinámica, de modo que para texturizar y no usar materiales en los modelos geométricos es necesario que en el ejecutable, o mejor llamado ROM, contenga las imágenes para ser texturizadas. La limitante para el tamaño del ROM es de 4MB, puesto que es el tamaño de la RAM principal. El uso de texturas se deja para futuras actualizaciones del **Framework** y para su implementación por parte de los desarrolladores en sus aplicaciones.

Otra especificación del **Framework** es que los gráficos están siendo procesados por el ARM9, esto por que es un procesador más poderoso además de estar un poco más especializado. Otras cosas como la conexión de red inalámbrica se deja al procesador ARM7.

3.1.3.2 - Display List

Durante el desarrollo de las aplicaciones para la consola, se encontró con un inconveniente: debido a la gran cantidad de datos que se deben procesar, se veía disminuida la velocidad de la construcción de las geometrías usando OpenGL convencional. Con pocas geometrías no tiene problema, pero considerando la complejidad que puede llegar a tener un escenario, se optó por buscar otras alternativas.

Se decidió implementar primitivas usando geometrías precompiladas, conocidas como *Display List*. Los *Display List* tienen una aplicación muy particular para el Nintendo DS que difiere en su forma de los *Display List* de OpenGL[31].

Los *Display List* fueron creados para hacer que los programas se ejecuten más velozmente por el procesador. Por ejemplo, para crear un objeto como un personaje o un edificio, se tienen las primitivas que OpenGL dispone, como las cubos, esferas, lineas, etc., y de las transformaciones básicas desplazamiento, rotación y traslación. Esto significa que cada vez que el personaje deba ser mostrado en la pantalla es necesario crearlo usando las primitivas y las transformaciones, esto se traduce en trabajo para el procesador. El cálculo para el despliegue se hace para cada *frame*[32] de la animación. El *Display List* está compuesto por una serie de líneas de código, con indicaciones de cómo construir las geometrías, el color que debe tener la cara y parámetros de luz como brillantez, normales, etc.

31 Shreiner, Dave. “OpenGL, Reference Manual”. pp. 20, 21, 39.
32 Fotograma o cuadro, una imagen particular dentro de una sucesión de imágenes que componen una animación.

3.1.3.3 - Configuración de los Display List

Los *Display List* tienen una estructura y reglas de construcción muy particulares, las cuales se deben respetar al pié de la letra. Los *Display List* está compuestos por comandos FIFO, y cada uno tiene una función muy especial dentro de la geometría. Los comandos FIFO son los siguientes:

Comando	**Descripción**
`FIFO_COMMAND_PACK(c1,c2,c3,c4)`	Indica cuatro tipos de empaquetados de comandos de 32 bits
`FIFO_NOP`	Es un comando que indica que no se hará nada
`FIFO_COLOR`	Indica el color al vértice
`FIFO_VERTEX16`	Comando para vértices de 16 bits
`FIFO_TEX_COORD`	Para coordenadas de las texturas
`FIFO_TEX_FORMAT`	Formato de las texturas
`FIFO_PAL_FORMAT`	Comando para la paleta de texturas
`FIFO_CLEAR_COLOR`	Comando para limpiar el reverso del plano
`FIFO_CLEAR_DEPTH`	Define la profundidad del reverso del plano
`FIFO_LIGHT_VECTOR`	Define el origen de la dirección de la luz
`FIFO_LIGHT_COLOR`	Define el color de la luz
`FIFO_NORMAL`	Para la normal de los vértices subsecuentes
`FIFO_DIFFUSE_AMBIENT`	Define las propiedades del canal difuso del vértice
`FIFO_SPECULAR_EMISSION`	Establecer las propiedades especulares del material
`FIFO_SHININESS`	Define el resplandor de los vértices
`FIFO_POLY_FORMAT`	Define los atributos del polígono
`FIFO_BEGIN`	Comienza una lista de vértices para un polígono
`FIFO_END`	No produce gran efecto, no se recomienda su uso si se indica el tamaño de la lista
`FIFO_FLUSH`	Fuerza el procesamiento de los comandos de geometría anteriores
`FIFO_VIEWPORT`	Define el puerto de vista

Cabe hacer notar que para el **Framework** no utilizamos todos, esto por simplificar el proceso y no complicar más los cálculos y el desarrollo.

Para mostrar la estructura de un *Display List* se ejemplifica con un cubo unitario posicionado en el origen, en la ilustración 4, de la cual sólo se mostrará la configuración de la cara frontal:

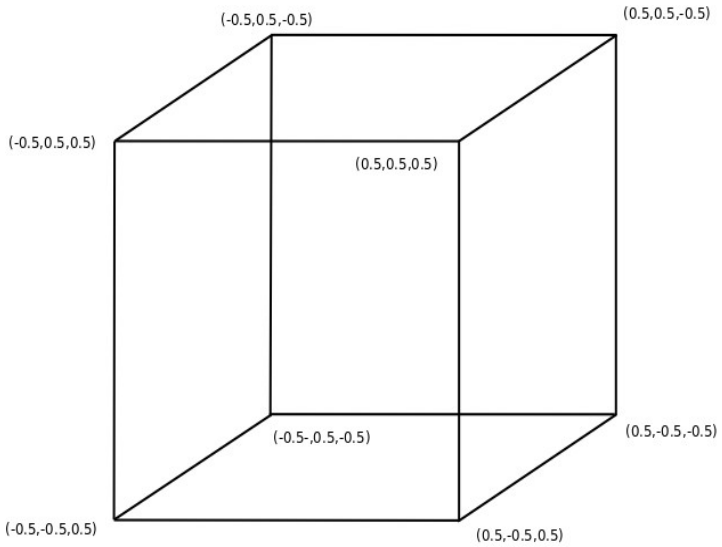

Ilustración 4: Cubo unitario posicionado en el origen

Además se incluyeron las coordenadas de los vértices para facilitar la comprensión de la construcción. El código sería el siguiente:

```
u32 front[] =
{
  16,
  FIFO_COMMAND_PACK(FIFO_BEGIN, FIFO_COLOR, FIFO_VERTEX16, FIFO_COLOR),
  GL_QUADS,
  RGB15(31,31,31),
  VERTEX_PACK(floattov16(-0.5),floattov16(-0.5)),
  VERTEX_PACK(floattov16(0.5),0),

  RGB15(31,31,31),
  FIFO_COMMAND_PACK(FIFO_VERTEX16, FIFO_COLOR, FIFO_VERTEX16, FIFO_COLOR),
  VERTEX_PACK(floattov16(0.5),floattov16(-0.5)),
  VERTEX_PACK(floattov16(0.5),0),

  RGB15(31,31,31),
  VERTEX_PACK(floattov16(0.5),floattov16(0.5)),
```

```
  VERTEX_PACK(floattov16(0.5),0),

  RGB15(31,31,31),
  FIFO_COMMAND_PACK(FIFO_VERTEX16, FIFO_END, FIFO_NOP, FIFO_NOP),
  VERTEX_PACK(floattov16(-0.5),floattov16(0.5)),
  VERTEX_PACK(floattov16(0.5),0),
}
```

Primeramente se tiene la línea que indica el tipo de estructura y el nombre de la geometría

```
u32 front[] =
{
      16,  //número de comandos
      //Especificación de la geometría
}
```

La primera línea después de iniciadas las llaves, es el número de comandos del que está compuesta la geometría. La división entre cada comando está indicada por una coma y no es necesario el salto de línea para que los diferencie.

`FIFO_COMMAND_PACK` indica los siguientes cuatro tipos de empaquetados de comandos de 32 bits, este comando contiene la información de los comandos que se esperan en las líneas subsecuentes. No es estrictamente necesario marcar cada comando que sigue, por ejemplo, existen algunos *Display List* que solo indican un `FIFO_VERTEX16` para varios `VERTEX_PACK()`, pero a lo largo del trabajo de desarrollo se pudo comprobar que es bueno indicar cada comando.

```
  FIFO_COMMAND_PACK(FIFO_BEGIN, FIFO_COLOR, FIFO_VERTEX16, FIFO_COLOR),
  GL_QUADS,
  RGB15(31,31,31),
  VERTEX_PACK(floattov16(-0.5),floattov16(-0.5)),
  VERTEX_PACK(floattov16(0.5),0),
  RGB15(31,31,31),
```

El `FIFO_COMMAND_PACK` indica los siguientes cuatro empaquetados, el primero es un `FIFO_BEGIN`, su correspondencia es el `GL_QUADS`, el cual dice que los siguientes vértices serán cuadriláteros. Posteriormente tenemos un `FIFO_COLOR` que marca al `RGB15(31,31,31)` definiendo así el color del vértice. Sigue un `FIFO_VERTEX16` el que agrupa a los `VERTEX_PACK()`, son dos porque solo reciben coordenadas bidimensionales, de modo que el primero es para el XY y el segundo es para el Z.

Formando así las especificaciones de forma, de color y de posición del primer vértice de una cara del cubo. El siguiente `FIFO_COLOR` especifica el color para el siguiente vértice.

Se puede observar que una coordenada 3D está formada por un par de `VERTEX_PACK(X,Y)`, también observamos que la construcción de la cara tiene sentido antihorario en la ilustración 5.

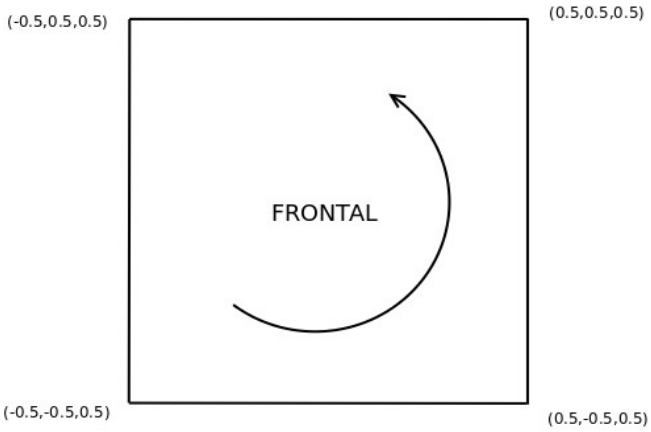

Ilustración 5: Cara frontal del cubo unitario

EL comando `RGB15(R,G,B)`, es usado para darle un color al vértice, en caso de colocar diferentes colores de un vértice a otro se creará un degradado entre los mismos. Los valores para RGB van del 0 con la menor cantidad de color al 31 con la mayor.

COLOR	
NEGRO	(0, 0, 0)
ROJO	(31, 0, 0)
VERDE	(0, 31, 0)
AZUL	(0, 0, 31)
BLANCO	(31, 31, 31)

Por último cabe resaltar que el comando `FIFO_COMMAND_PACK(FIFO_VERTEX16, FIFO_END, FIFO_NOP, FIFO_NOP),` indica que existen vértices después de él, luego termina la *Display List* y por último los siguientes comandos no operan, en otras palabras que se termina de definir la geometría.

Empleando las *Display List*, se crean las geometrías una sola vez, de manera que no importa cuantas veces debamos llamar al objeto, solo debemos escribir `glCallList(DisplayList).`

3.1.3.4 - Exportador de Blender

Usando los *Display List* se logra obtener más rendimiento, al renderizar más polígonos sin realizar muchos cálculos. Aunque se emplea mayor cantidad de espacio por incluir los archivos *Display List* en el archivo ejecutable, se considera que es una buena decisión. Sin embargo se nos presentó otro problema: buscando entre los desarrollos libres para el Nintendo DS todos llegaron a la misma conclusión, no es sencillo formar las *Display List*.

Para facilitar el trabajo de desarrollo en el **Framework**, se pensó en realizar un exportador para el formato *Display List*. De esta manera se puede simplificar el trabajo de crear las geometrías, edificios y personajes. La opción libre para la creación 3D fue Blender. El objetivo del exportador será obtener archivos con la estructura *Display List* que se requiere para el **Framework**

3.1.3.5 - Blender

Blender es un programa de animación y modelado 3D, de desarrollo libre y realmente pequeño a comparación con sus competidores no libres (Maya y 3DStudio principalmente). Además de ser multiplataforma tiene capacidad para una gran variedad de primitivas geométricas, incluyendo curvas, mallas poligonales, NURBS, *metaballs*, etc. Con las herramientas de animación, incluye cinemática inversa, vértices de carga y partículas estáticas y dinámicas, además de edición de audio y video, y algunas características como detección de colisiones, recreaciones dinámicas. También maneja el lenguaje interpretado Python para automatizar o controlar algunas tareas.

Sin embargo, el uso de Blender no es sólo el de crear complejas animaciones, también funciona como modelador 3D y de ese modo poder exportarlo a cualquier formato que se requiera. De modo que podemos definir las geometrías, modificarlas, asignarles materiales y obtener el *Display List*.

Python[33] es el lenguaje para escribir los exportadores, *plugins* y *scripts*[34] dentro de Blender. El editor 3D Blender tiene un editor de texto para escribir *scripts* en Python y los *scripts* deben ser colocados en el directorio ~/.blender/scripts y para acceder al editor de texto dentro de Blender, en el menú Window Type, Text Editor, o simplemente usar su editor de texto plano preferido para programar el *script* y guardar el archivo con la extensión .py.

Cabe resaltar algunas de las características que Python posee como lenguaje de programación:

- Es sensible a la justificación, esto es que para diferenciar el alcance de afectación de alguna estructura de control como un IF, las líneas que están justificadas después que su nivel son afectadas. Esto complica un poco el desarrollo dentro de Python.

- Es muy simple y versátil, porque el exportador fue hecho como lenguaje estructurado sin funciones, y respondió de manera óptima sin complicar o retrasar el proceso.

Para poder exportar es necesario saber que exportar, Blender tiene una serie de objetos en la escena como la cámara, la lámpara y la geometría. A su vez estos objetos tiene funciones y propiedades de los cuales podemos obtener la información que se requiere. Como se utiliza a Blender sólo para modelar las geometrías, no es necesario tener en escena la cámara y la lámpara, de modo que para cada proyecto, recomendamos eliminar dichos objetos.

33 Wartmann, Carsten. “The Blender Book”, pp. 198-211.
34 Es un conjunto de instrucciones orientadas a la automatización de tareas creando pequeñas utilidades.

Para poder exportar algún dato de la geometría de la escena es necesario incluir las siguientes líneas que obtienen todos los datos de la escena activa, posteriormente los objetos dentro de esos datos y por ultimo la geometría que es lo requerido.

```
sce = bpy.data.scenes.active
ob = sce.objects.active
mesh = ob.getData()
```

Con lo anterior se puede obtener líneas hacia el archivo como se muestra a continuación, en donde obtiene el nombre de la geometría (mesh).

```
out.write("u32 " + mesh.name + "[] =" + "\n" + "{" + "\n")
```

Esta línea daría como resultado en el archivo de salida algo como lo siguiente:

```
u32 AfrArmL[] =
{
```

La parte medular del exportador está contenida en tres líneas de código, éstas recorren para cada material los vértices de cada cara. Se buscó implementar de esta manera para intentar reducir líneas de salida innecesarias, en este caso la definición de colores cada vez que existe un cambio de material. Con esto sólo se hacen cambios de color equivalentes al número de materiales dentro de la geometría.

```
for mat in mesh.materials:
        for face in mesh.faces:
                if mesh.materials[face.materialIndex].name == mat.name:
```

Una variable que está presente dentro de la exportación es el número de funciones o *FIFOs* que serán ingresadas al archivo de salida. De modo que con cada función o *FIFO* se incrementa el número de comandos de salida.

En cada salida se debe verificar si el cuarto comando del *FIFO_COMMAND_PACK* ya fue ingresado, para crear un *FIFO_COMMAND_PACK* nuevo y comenzar el conteo de 4 y así un nuevo empaquetado.

Otra característica de Blender es que únicamente maneja dos formas básicas: el triángulo y el cuadrángulo, al igual que el Nintendo DS. De modo que simplifica mucho el trabajo, pues no es necesario calcular el número de vértices de cada cara y así definir la forma requerida o la subdivisión en partes más pequeñas.

Las ilustraciones 6, 7 y 8 muestran la estructura general del exportador:

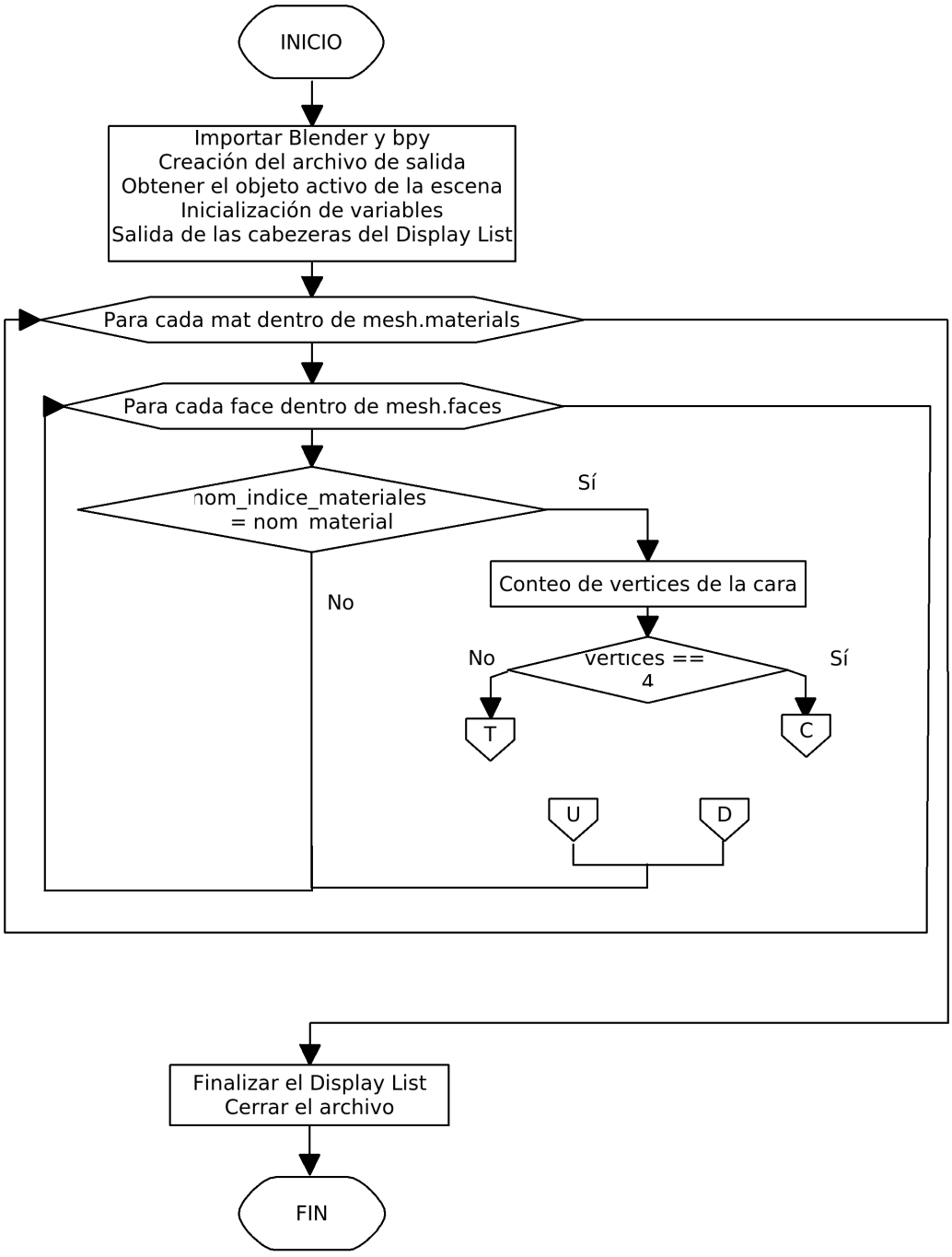

Ilustración 6: Diagrama de flujo del Exportador

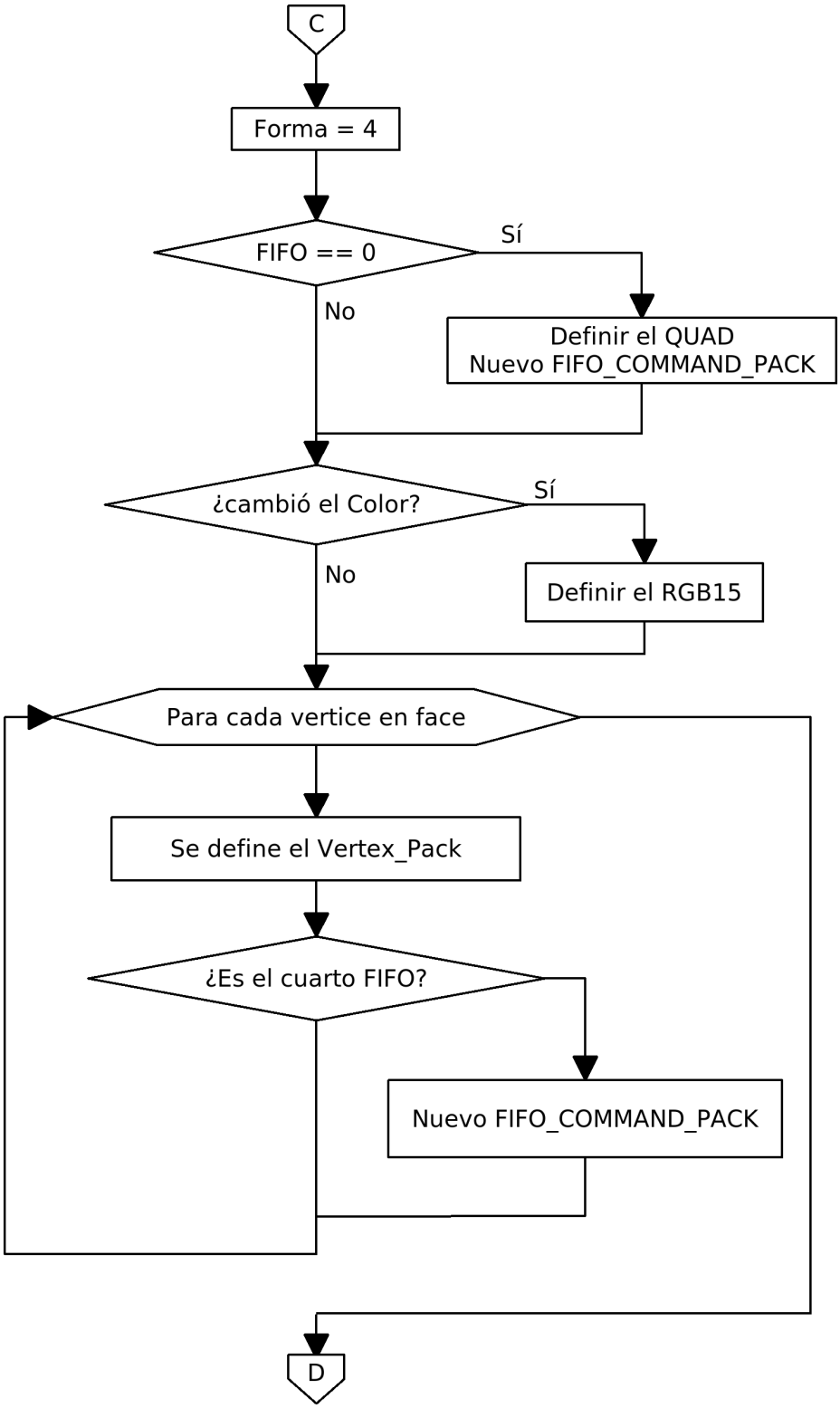

Ilustración 7: Sección de los cuadrángulos

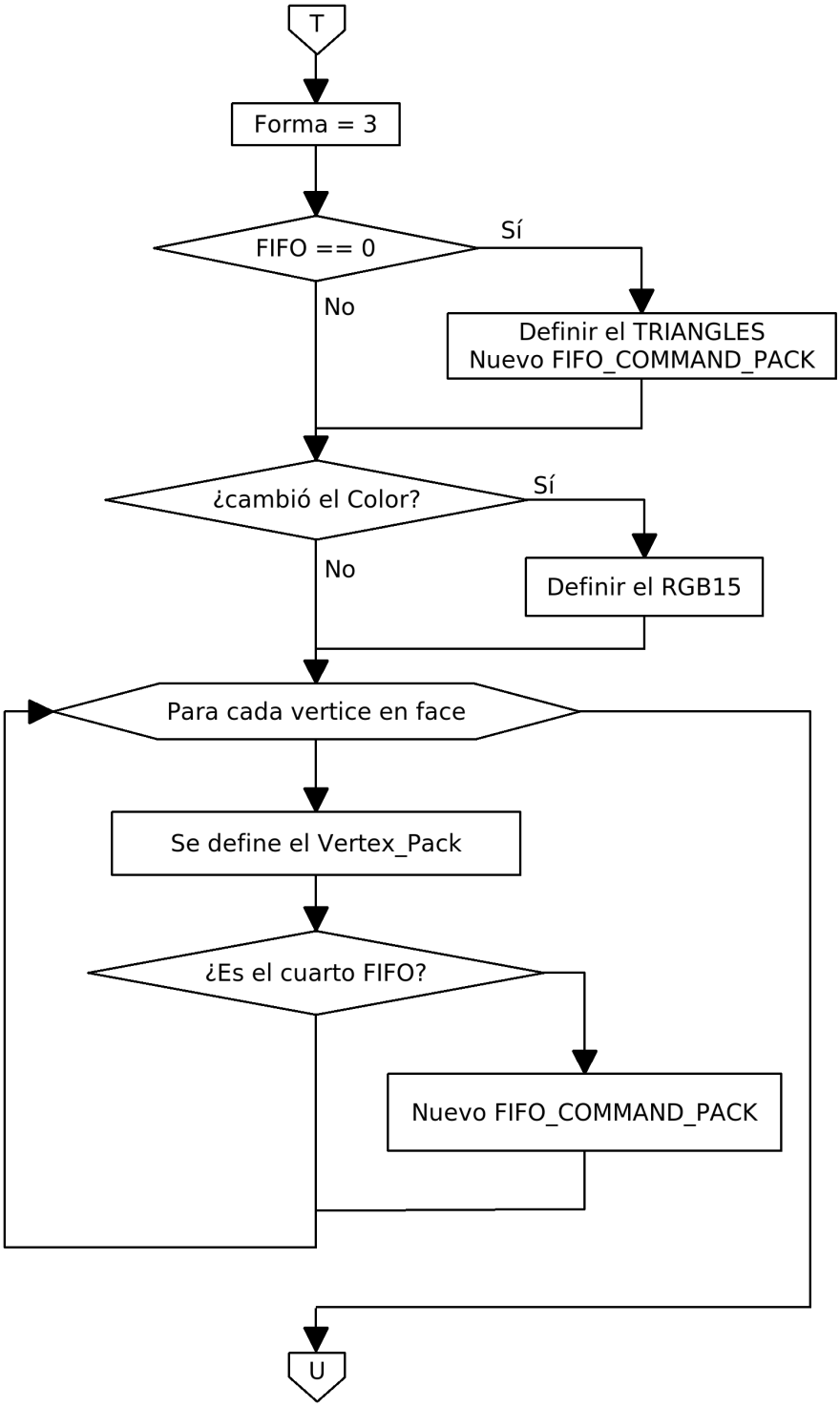

Ilustración 8: Sección para los triángulos

3.1.3.6 - Características del exportador

El exportador está incluido dentro del **Framework** como una herramienta de ayuda al desarrollo de personajes y otras geometrías para los videojuegos.

- Brinda una posibilidad de trabajo para el **Framework** para áreas que no estén muy relacionadas con la programación.
- Deja el diseño de personajes a cualquiera que quiera trabajar y ver su trabajo reflejado directamente en pantalla.
- No es necesario aprender a usar Blender en su totalidad, únicamente con manejo básico de vértices, caras, y asignación de materiales.
- Se deja el manejo de texturas para futuras ampliaciones del **Framework**, de modo que simplifica aún más el trabajo con el exportador.
- El exportador da como resultado un archivo con formato *Display List* completamente compatible para ser incluido y trabajado con el **Framework**.
- Al ser un *script* hecho en Python, el exportador puede ser incluido como un módulo parte de Blender y así exportar desde el mismo programa.
- El código puede ser adaptado para cualquier necesidad particular de cada desarrollo por parte de los programadores.

3.1.3.7 - Clase mjBiped()

Terminado el exportador de Blender para *Display List*, se realizó una clase genérica para incorporar las geometrías y poder usarlas con los personajes. Tomando como base el **lanjobot**, se implementaron bípedos humanoides y se realizó la clase **mjBiped**() para su aplicación.

El propósito de esta clase es la de crear personajes a partir de geometrías hechas con Blender y usarlos dentro de los juegos. La prueba de concepto incluirá algunos ejemplos de esto.

La clase **mjBiped**(), requiere de algunos parámetros para ser inicializada:

- Un archivo de cabecera con los *Display List*, y la ruta en la que se encuentran.
- La asignación de los *Display List* a la estructura del Bípedo.
- Referencias de posición para cada uno de los *Display List*.

La estructura general para el **mjBiped**() se ve en la ilustración 9.

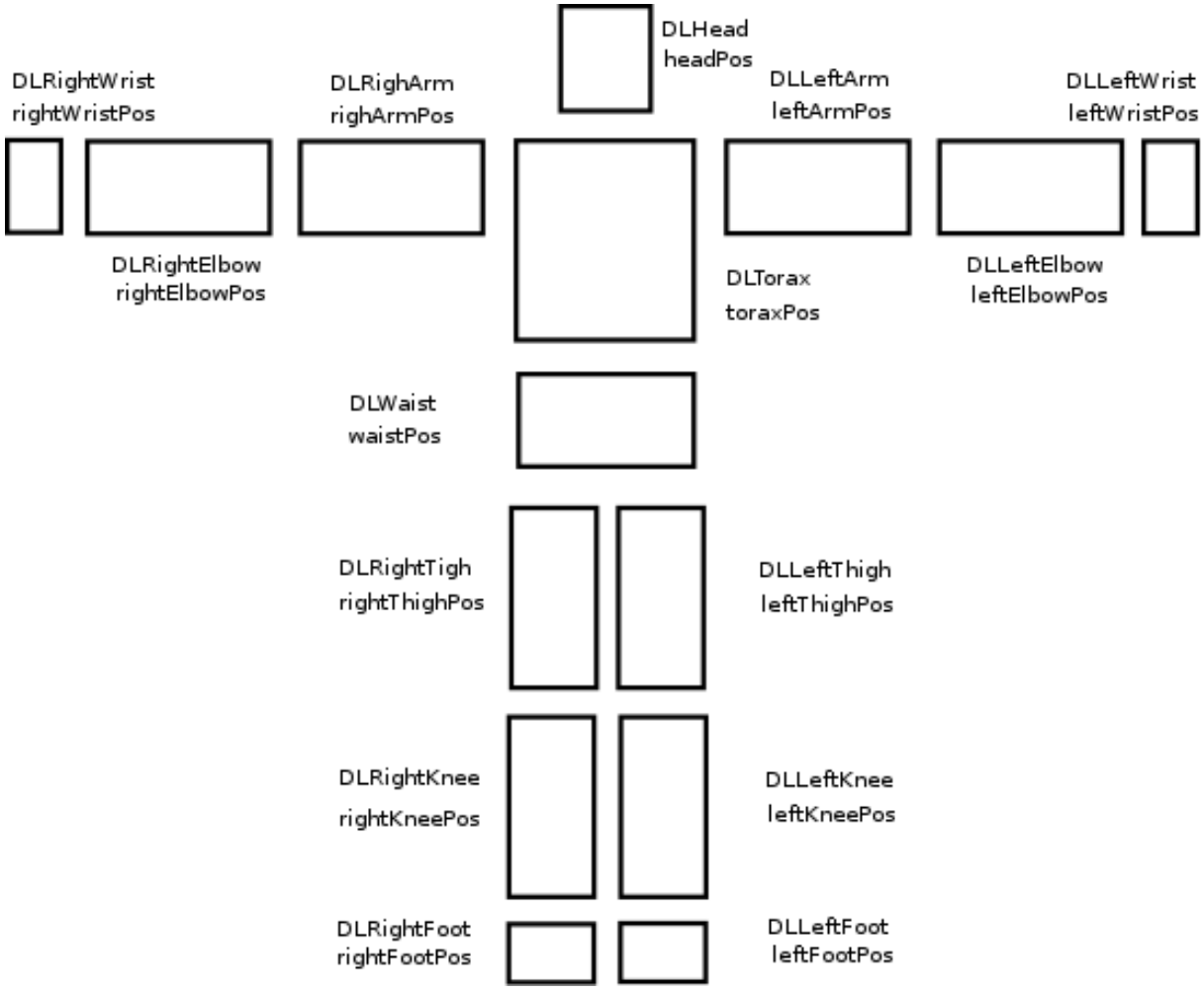

Ilustración 9: Estructura para un bípedo: mjBipedDef

Con esos componentes se puede definir los componentes del bípedo, así como la posición inicial de dichos componentes para ser construido y utilizado por la clase.

Esta jerarquía es general y aplica para definir a casi cualquier personaje, no importando las diferencias de torso o extremidades, incluso cuando el personaje no es simétrico en vestimenta.

Además el bípedo posee algunos movimiento básicos, tomados a partir del **lanjobot**, como son

- Inactivo.
- Caminar.
- Correr.
- Saltar y caer.
- Dar un golpe.

Cada uno de los movimientos anteriores están definidos tanto para la parte superior como la inferior del bípedo, esto con el fin de resolver un conflicto de correr y caminar cuando se da un golpe, porque la animación de la parte superior e inferior son diferentes en cada caso. Otros tipos de movimientos más especializados para cada juego específico deben ser programados por los desarrolladores.

Las medidas y proporciones del bípedo dependen del universo en el que estará. Tómese en cuenta que para el **Framework** se recomienda un límite en los modelos dentro de un cubo de (-4.0,-4.0,-4.0) a (4.0,4.0,4.0). Esta recomendación se debe a que durante las pruebas con modelos de unidades mayores a éstas el despliegue no responde de manera adecuada.

El bípedo es una ejemplificación sencilla de personajes con movimientos básicos, su función es mostrar este tipo de implementaciones para los programadores que quieran hacer su propio desarrollo usando el presente **Framework**. Ésta clase podría ser base para personajes cuadrúpedos, máquinas, personajes voladores, etc.

Un ejemplo de personajes creado a partir de *Display List* exportadas de Blender y construido con la clase **mjBiped**(), es el siguiente:

Tabla de ilustraciones de las vistas de un personaje

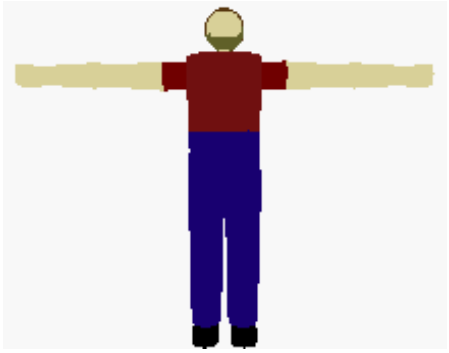

Ilustración 10: Vista Frontal

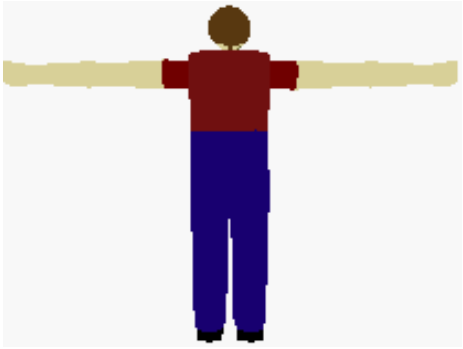

Ilustración 11: Vista posterior

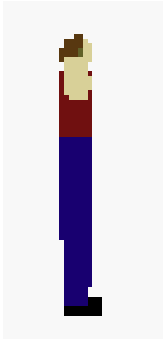

Ilustración 12: Vista lateral derecha

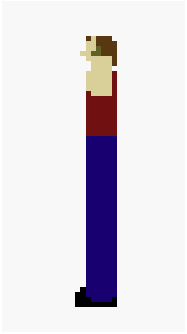

Ilustración 13: Vista lateral izquierda

3.1.4 - Subsistema de Sonido

3.1.4.1 - La importancia del sonido en los videojuegos

El audio en los videojuegos es un complemento atmosférico para los demás elementos del juego. Para la mayoría de los juegos de la actualidad con vista 3D en la pantalla, un entorno de sonido tridimensional es parte importante de esa sensación de hallarse realmente ahí.

En los años 70, cuando los videojuegos comenzaban a tomar fuerza y se volvían una forma de entretenimiento, los dispositivos de reproducción y almacenamiento de música eran analógicos, por lo que eran propensos a averías como ejemplo los cassetes. Aunado a que las consolas como la arcadia no disponían suficiente capacidad de almacenamiento ni de procesamiento para soportar la reproducción de música grabada por una banda sonora. Así que los medios digitales fueron la opción elegida para tener un fondo musical en los videojuegos, aunque monofónica, repetitiva y usada en ciertas partes del juego marcó el inicio del audio dentro de los videojuegos.

Con la mejora de las tecnologías se incorporó a las consolas chips de generadores de tonos de hasta 8 canales y algunos compositores trabajaron en la producción de música para algunos juegos. Con la evolución y para evitar que el procesador se dedicara solo a generar sonidos complejos, se buscaron otras alternativas, como la secuenciación MIDI, la síntesis o el muestreo digital. Con la mejora de la tecnología, el audio en los videojuegos mejoró notablemente llegando a reproducir música del CD sin ningún procesamiento, incluso a crear su propio formato de audio a reproducir.

También se le dedicó más trabajo a la producción del audio que a la reproducción, de modo que el músico o compositor puede crear su propia música sin la necesidad de saber programar o conocer las especificaciones o limitantes de la consola. Incluso llegó a tomarse música previamente existente en el mercado musical para los videojuegos.

En la actualidad la música en los videojuegos llegó a convertirse en un genero musical independiente. Se ha usado música clásica, rock, pop, electrónica, etc. Creando así un mercado para el audio en los videjuegos y dando a conocer compositores dedicados para este género.

El Nintendo DS es capaz de reproducir de manera nativa datos en formato WAVE de 8 ó 16 bits por muestra y hasta 44,100 Hz en cada uno de sus 16 canales de *hardware*. Los juegos usualmente tienen música sintetizada y efectos de sonido reales, es decir, grabaciones digitalizadas. En el **Framework** tratamos de simplificar la utilización de música y efectos de sonido reales a fin de realizar juegos con un audio más agradable.

3.1.4.2 - La música de fondo

La música de fondo en los videojuegos es una parte importante ya que al igual que en una producción cinematográfica, puede realzar de manera muy significativa los sucesos que se presencian en la pantalla. Es por esto que se favorece la utilización de música no sintetizada en el **Framework**

Para poder emplear música no sintetizada en los juegos, se investigó sobre diferentes formatos de sonido; a continuación se presentan las comparaciones entre varios formatos que fueron considerados:

Formato	Compresión	Pérdida de calidad	Licencia	Ventajas	Desventajas
MP3	Alta	Sí	Existen desarrollos con licencias libres	Es muy popular, existen muchas implementaciones para dispositivos embebidos	Una patente por parte de la organización Fraunhofer podría impedir su implementación sin pagar regalías aún tratándose de código libre. La distorsión resultante de la compresión son notorios al reproducir el sonido.
Ogg Vorbis	Alta	Sí	Libre	El formato fue desarrollado como *software* Libre desde un principio, existen implementaciones para dispositivos embebidos. La calidad de la descompresión es muy alta.	Su descompresión es costosa, no es tan popular como el formato MP3.
Monkey's Audio	Baja	No	Libre	Es un formato libre sin pérdidas de calidad.	Es el menos popular de los formatos considerados en la tabla, por lo tanto es menos factible que existan implementaciones optimizadas para dispositivos embebidos. Los archivos tienden a ocupar más espacio que otros formatos.
FLAC	Baja	No	Libre	Es un formato libre sin pérdidas de calidad.	Aunque existe una implementación libre, específica para el Nintendo DS, el código resultó muy difícil de leer para lograr su adopción. Los archivos tienden a ocupar más espacio que otros formatos.

Las prioridades al considerar formatos fueron: facilidad de adopción, la existencia de alguna implementación libre y el evitar patentes restrictivas, la relación de compresión con respecto al audio original y la calidad subjetiva de la misma, su popularidad, y la familiaridad de los desarrolladores con dichos formatos.

El formato que mejor cumplió con los objetivos pudo haber sido MP3, sin embargo hubo un factor determinante que finalmente impidió adoptarlo: Existen varias empresas y organizaciones con patentes en las diferentes tecnologías y procesos que conforman MP3, incluso entrando en conflictos legales entre ellas y la mayoría exigen el pago de regalías por comprimir y descomprimir audio en dicho formato, al menos para empresas. Prácticamente cualquiera de ellas podría volverse hostil al desarrollo del *software* Libre hasta que sus patentes expiren.

Ogg Vorbis es el segundo formato más popular de la tabla de formatos considerados, es muy similar en capacidades de compresión al MP3, sin embargo tiene ventajas muy importantes: la organización que es propietaria de la patente, Xiph.org Foundation, tiene como misión específicamente crear formatos multimedia cuyas patentes favorezcan sus implementaciones en *software* Libre y debido a esto existe una implementación oficial de dicha organización para dispositivos embebidos que es libre. Las únicas desventajas con respecto al MP3 son que requiere de un mayor procesamiento para su descompresión y que al ser menos popular que dicho formato, existen menos herramientas para procesar audio comprimido en formato Ogg Vorbis.

Esta última desventaja es en realidad una desventaja aparente, ya que se puede descomprimir el formato Ogg Vorbis a un formato Wave, el cual es soportado de manera universal, editarlo, procesarlo, y posteriormente comprimirlo de nuevo a Ogg Vorbis. Además, la comunidad del *software* Libre ha apoyado de manera muy significativa el formato Ogg Vorbis, especialmente mediante concienciación de persona a persona para reducir la brecha en popularidad entre uno y otro formato.

Existen otros formatos para sonido comprimido que no se consideraron: WMA, MP4 (comúnmente usada por Apple Computer), ATRAC3 (Sony). Las razones por las que no se consideraron es que tienen problemas de licenciamiento, es decir, no existen licencias libres, o tienen problemas de patentes similares al MP3 sin tener la popularidad de dicho formato.

3.1.4.3 - El formato o codec OggVorbis

Se trata de un formato de compresión de sonido, es decir, tomando el formato de CD como un formato sin pérdidas, es posible reducir el tamaño de un archivo de sonido hasta 10 veces y descomprimirlo posteriormente para reproducirlo.

Al compresor de audio Ogg Vorbis de referencia, *oggenc*, se le puede especificar como un parámetro numérico entre 0.0 y 10.0 la semejanza que se espera obtener al descomprimir el sonido con respecto al audio original. Cabe mencionar que esta semejanza se refiere a “que se oiga como el original”, mas no necesariamente a que los datos se asemejen bit a bit a los originales.

Ogg Vorbis es, por lo tanto, un formato con pérdida de calidad, es debido a esto que es factible comprimir sonido a tamaños comparativamente tan reducidos.

Sin embargo, para los humanos que no tienen un oído especialmente entrenado, a calidades medianas o bajas casi no se nota la diferencia, y si se realiza la compresión a calidad alta es prácticamente imposible. A este tipo de compresión con pérdidas se le conoce como Codificación Perceptible, ya que toma en cuenta las capacidades de percepción del usuario promedio y desecha los fragmentos de información que éste no percibe.

3.1.4.4 - Musica de fondo usando bibliotecas OggVorbis

Para poder utilizar archivos de sonido con formato Ogg Vorbis en la consola Nintendo DS, se recurrió a la implementación de referencia del *codec*[35] en dispositivos embebidos, la cual recibe el nombre de *Tremor*. La ventaja con respecto a la implementación común es que *Tremor* emplea únicamente operaciones con números enteros en vez de números flotantes, dado que la aritmética de números enteros generalmente requiere menos ciclos de reloj que la de números flotantes.

Tremor permite descomprimir un archivo en formato Ogg Vorbis a datos en formato Wave. Estos datos descomprimidos se encuentran muestreados a 44,100 Hz y cada una de las muestras ocupa 16 bits, y se puede configurar el código para que los bits más significativos aparezcan al principio o al final de la muestra, es decir, en la dirección más baja de memoria perteneciente a la muestra, conocido también como orden Big-Endian o en la dirección más alta de memoria perteneciente a la muestra, conocido como orden Little-Endian, respectivamente. En el caso del Nintendo DS, el sistema de sonido requiere que los datos se encuentren en orden Little-Endian.

Una vez que el código de *Tremor* fue configurado para funcionar en el Nintendo DS, se pueden usar las funciones de apertura, extracción de propiedades y descompresión de datos del *codec* Ogg Vorbis.

Para poder escuchar un archivo comprimido en formato Ogg Vorbis en el Nintendo DS, se requiere realizar las siguientes acciones:

1. El archivo debe estar presente en la tarjeta de memoria y su ruta debe ser especificada de manera absoluta.

2. Se debe abrir el archivo mediante la función estándar *fopen*. Posteriormente, el descriptor de archivo retornado por *fopen* debe suministrarse a la función *ov_open* de *Tremor*.

3. Una vez que *ov_open* ha determinado que se trata efectivamente de un archivo en formato Ogg Vorbis, se pueden extraer las propiedades que tendrá el sonido al ser descomprimido. Este paso se realiza con la función *ov_info*, y es necesario para determinar el número de canales que posee

35 Proveniente de Codificador-Decodificador, describe una especificación desarrollada en *software*, *hardware* o una combinación de ambos, con el objetivo de transformar un archivo, un flujo de datos o una señal.

el sonido. El Nintendo DS es estéreo, así que si se tratara de un archivo de más de dos canales tendría que mostrarse un mensaje de error y dicho archivo no podría ser reproducido.

4. Se comienzan a extraer las muestras de sonido mediante la función *ov_read*. Estas muestras de sonido deben ser almacenadas secuencialmente en un arreglo de carácteres por cada canal que tenga el archivo. En el caso de ser un archivo con varios canales de audio, inicialmente las muestras aparecen intercaladas por cada canal y deben ser separadas copiándolas al arreglo correspondiente al canal al que pertenecen.

5. Estos arreglos de carácteres deben ser suficientemente grandes como para almacenar una cantidad de muestras por cada canal que sea significativa para el sistema de sonido.

6. Los arreglos de carácteres se le suministran al sistema de sonido, indicándole las características del fragmento de audio: frecuencia de muestreo, tamaño del arreglo y tamaño de muestra.

7. Se debe leer el siguiente fragmento de audio en el archivo, mientras el sistema de sonido termina de reproducir el fragmento previamente leído.

8. Se debe repetir los pasos a partir del punto 4 hasta que *ov_read* informe que el archivo se ha terminado.

Tras experimentar con varios archivos de prueba, nos percatamos de que no era factible realizar la descompresión de un archivo con dos canales de audio sin afectar significativamente la fluidez de la animación, así que se optó por limitar las capacidades del **Framework** a reproducir archivos con datos de sonido para un solo canal.

Como lo sugieren los pasos anteriormente mencionados, se requirió plantear un esquema tipo productor-consumidor en el que el productor fuera el descompresor de Ogg Vorbis, y el consumidor, el sistema de sonido del Nintendo DS, debido a que al realizar la descompresión del archivo de sonido completo, éste rebasará el tamaño de la memoria de la consola si dicho archivo dura más de 50 segundos.

Tras probar con diversos tamaños para el arreglo que almacena temporalmente el fragmento de sonido descomprimido, se encontró que el tamaño ideal correspondía al equivalente a 1 segundo de sonido, es decir, 88,200 bytes, dado que son 44,100 muestras de 2 bytes para un segundo de sonido.

En el problema particular planteado, sin embargo, se desea que el arreglo permanezca intacto mientras está siendo reproducido, y que el sistema de sonido nunca tenga que esperar para comenzar a reproducir. Con estos requerimientos en mente, se decidió ocupar dos arreglos en vez de uno, de esta manera el productor puede rellenar el arreglo que no está siendo reproducido y esperar hasta que el consumidor termine de reproducir el otro arreglo, para luego intercambiar arreglos y que no existan pausas aparentes en la música de fondo, ya que cuando las hay son muy notorias.

Para lograr esto, debido a que no existe un sistema operativo que provea administración de procesos, se recurrió a ocupar uno de los cuatro *timers* o temporizadores de *hardware* que existen en el Nintendo

DS. Se configuró el cuarto temporizador para correr una rutina que intercambie los arreglos entre consumidor y productor, y envíe los datos de sonido al sistema de sonido de la consola.

Una vez resuelto el problema de mantener el consumidor pidiendo datos regularmente, se requiere asegurar que el productor funcionará sin afectar la animación.

Inicialmente, se estableció que la descompresión del fragmento de audio se realizara completamente, después de haber mandado los datos al reproductor de sonido, pero esto congelaba la animación totalmente durante algunas fracciones de segundo, lo cual es inaceptable.

Se llegó a la conclusión de que el descompresor debía funcionar durante cada frame, procesando una cantidad ligeramente mayor a la que correspondería a cada frame si la descompresión se quisiera realizar en "tiempo real". Se decidió descomprimir un poco más de lo necesario para cada frame, debido a que la naturaleza de la mayoría de los videojuegos hace que el trabajo necesario por parte del procesador para dibujar cada frame sea distinto (el número de polígonos puede variar según la cantidad de enemigos presentes en pantalla, la complejidad de la porción del escenario visible, etc.). De esta manera, si ocurriera un retraso en la animación debido a un exceso de polígonos momentáneo, no afectaría la reproducción del audio.

3.1.4.5 - La clase mjMusicPlayer()

Buscando simplificar al máximo la reproducción de sonido en formato Ogg Vorbis, se implementó la clase *mjMusicPlayer*.

Para reproducir un archivo de tipo Ogg Vorbis, únicamente es necesario suministrarle la ruta absoluta en el sistema de archivos de la tarjeta de memoria al método *LoadFile()* y posteriormente utilizar el método *Play()*. El ciclo principal del motor de videojuegos automáticamente se encargará de realizar la descompresión invocando al método *Update()*, y la rutina de intercambio de arreglos, *InterruptFunction()* será invocada automáticamente por el cuarto temporizador del Nintendo DS.

La clase *mjMusicPlayer* aparece como un objeto de la variable global *App*, y se acceden a sus métodos y propiedades mediante *App.MusicPlayer*.

Entre los métodos que tiene, se encuentran *Pause()*, el cual detiene la reproducción del audio, y *TimeSeek()*, para recorrer a algún momento específico del archivo de sonido.

Si se deseara cargar otro archivo Ogg Vorbis, simplemente se utilizaría el método *LoadFile()* sin mayor complicación; la clase automáticamente liberará la memoria utilizada por el algoritmo de descompresión.

3.1.4.6 - Efectos de sonido

Los efectos de sonido son muy importantes en un videojuego para aumentar la sensación de inmersión e indicarle al usuario lo que está sucediendo y que posiblemente éste no sea capaz de ver por tratarse de acción fuera de la pantalla.

En el caso de los efectos de sonido, por lo general se requieren reproducir varios a la vez, son de corta duración y por la misma naturaleza de los videojuegos, es imposible predecir el orden en el que serán necesarios y se requiere que cualquiera de ellos esté disponible instantáneamente. Tomando en cuenta todo lo anterior, además de que el Nintendo DS reproduce de manera nativa datos en formato Wave se decidió utilizar este mismo formato para almacenar los efectos de sonido y diseñar una clase que permitiera abrirlos de manera sencilla.

3.1.4.7 - La clase mjWaveSample()

La clase *mjWaveSample* sirve para emplear archivos Wave en los juegos de manera sencilla. Para utilizar archivos Wave con dicha clase, éstos archivos deben cumplir con los siguientes requisitos:

- Deben ser monoaurales
- No deben tener ningún tipo de compresión
- Deben ser cortos en duración para caber en el espacio libre en la memoria

Basta con crear tantas instancias de la clase como se requieran y cargar los sonidos mediante el método *LoadFile()*, indicando la ruta completa a partir de la raíz del sistema de archivos de la tarjeta. Una vez cargado el archivo, podrá ser usado mediante los métodos *Play()* y *Play3d()*.

El primer método, *Play()*, reproduce el sonido con el volumen y el balance que se le especifique, esto es, se le puede instruir al sistema de sonido por cuál de las dos bocinas emitir el sonido con mayor intensidad mediante un número que va del 0 al 128, donde el valor 0 indicará al sistema de sonido que debe reproducirlo exclusivamente por la bocina izquierda; un valor de 64 hará que se reproduzca en ambas bocinas por igual, un valor de 128 únicamente por la bocina derecha, y valores intermedios resultarán en el sonido siendo emitido con una menor intensidad por la bocina izquierda y una mayor intensidad por la derecha conforme el valor aumente desde 0 hasta 128. El volumen general también puede tomar valores desde el 0 hasta el 128, siendo el valor 0 silencio total y el 128, la mayor intensidad que tiene el sistema de sonido.

El método *Play3d()* permite simular que el sonido proviene de una fuente omnidireccional localizada en un punto específico del mundo tridimensional dentro del videojuego. Para ello, requiere que se le especifique la posición del observador, la dirección que el observador considera como "izquierda", la posición de la fuente de sonido, la atenuación debida a la distancia medida en unidades de OpenGL, y el volumen máximo.

De esta manera se puede dar información al usuario sobre la localización de los objetos aún cuando no aparezcan en pantalla, y una impresión más completa si son visibles.

Cuando ya no se requiera el sonido que tiene cargada la instancia, se puede ocupar otro mediante la instrucción *LoadFile()* o se puede destruir dicha instancia; en ambos casos la memoria ocupada por los datos del sonido será liberada.

3.1.5 - Subsistema de Red

3.1.5.1 - La importancia de la interacción a través de la red en los videojuegos

En los últimos años, la capacidad de los sistemas de videojuegos para conectarse a través de las redes e Internet ha cobrado una importancia muy grande dado que brinda la oportunidad a jugadores de lugares remotos, incluso de diferentes continentes, para competir entre sí aunque no se conozcan ni hayan intercambiado palabra alguna entre ellos con anterioridad.

Existen videojuegos donde la interacción a través de las redes son partidas breves, pero también existen algunos cuya interacción en línea es la simulación de un mundo virtual, con sucesos que abarcan una gran cantidad de días.

Como tal, el que un videojuego presente la posibilidad a varios jugadores de interactuar simultáneamente es un aspecto que lo hace mucho más atractivo; es debido a esto que se decidió facilitar el uso del equipo de red inalámbrica que el Nintendo DS posee.

3.1.5.2 - El equipo de red inalámbrica del Nintendo DS

El Nintendo DS tiene embebido un sistema de transmisión y recepción de señales de red inalámbrica compatibles con el estándar 802.11b de la IEEE en modo infraestructura. Mediante este estándar, si el *Access Point*[36] tiene conexión a Internet, dependiendo del juego, la consola permite interactuar con otras personas que tengan el mismo juego y que tengan también conexión a Internet, independientemente de su localización geográfica.

Los juegos oficiales de Nintendo también emplean otro protocolo de comunicación que es desconocido en su mayor parte a personas ajenas a Nintendo o sus licenciatarios para comunicarse de manera local, sin necesidad de un *Access Point*.

El equipo de red inalámbrica, al igual que el sistema de sonido del Nintendo DS, es accesible directamente solo a partir del procesador ARM7.

Para poder emplear la red inalámbrica del Nintendo DS, se utilizó la biblioteca *dswifi*, biblioteca libre y no oficial escrita por Stephen Stair. Esta biblioteca permite al Nintendo DS conectarse a un *Access Point* por medio del estándar 802.11b y enviar datos a través de los protocolos TCP/IP y UDP, para las versiones 4 y 6 de ambos protocolos utilizando instrucciones de alto nivel en lenguaje C.

3.1.5.3 - La biblioteca dswifi

La biblioteca *dswifi* fue escrita por Stephen Stair, para un concurso organizado por personas de la comunidad de desarrollo de videojuegos caseros entre 2005 y 2006.

Esta biblioteca fue creada para ayudar a todos los programadores de la comunidad de desarrollo de videojuegos caseros a emplear el equipo de red inalámbrica del Nintendo DS de manera que se pudieran realizar juegos con comunicación entre varias consolas, tanto local como a través de Internet. *dswifi* está disponible en el sitio de Stephen Stair como *software* Libre, bajo la licencia MIT.

La biblioteca, sin embargo, tiene una limitante: solo tiene implementado el modo infraestructura del estándar 802.11b, es decir, solo es posible realizar la comunicación entre varias consolas si existe un *Access Point* cercano, del cual se disponga la clave de acceso WEP, en caso de ser necesaria. La comunicación directa entre varias consolas sin utilizar un *Access Point* no está soportada en este

36 Un punto de acceso inalámbrico (del inglés WAP o AP: Wireless Access Point) en redes de computadoras es un dispositivo que interconecta dispositivos de comunicación inalámbrica para formar una red inalámbrica.

momento; un punto favorable es que al ser libre, existe la posibilidad de extenderla a futuro a fin de que soporte el modo *Ad-Hoc* de conexión, o incluso un modo completamente distinto, en funcionamiento, parecido al protocolo propietario de Nintendo, que tampoco requiere de un *Access Point* y se evitaría parte de la interferencia de las redes inalámbricas cercanas al lugar de juego. Otro punto favorable es que Stephen Stair previó estas posibilidades y su código fue diseñado tomándolas en cuenta; están fuera del alcance de esta tesis, pero es una posibilidad del proyecto a futuro.

La biblioteca *dswifi* permite además crear y utilizar *sockets* parecidos a los POSIX, TCP y UDP versiones 4 y 6, aunque no todas las funciones que se tienen en *sockets* POSIX existen en dswifi.

Para poder utilizarla correctamente, se requieren modificar varios aspectos del sistema de compilación normal de DevKitPro:

- Modificar el código fuente del ejecutable del ARM7 a fin de que procese las señales que aparecen a través del equipo de red inalámbrica.

- Establecer una rutina que maneje las interrupciones del equipo de red inalámbrica.

- Realizar un esquema de comunicación interprocesador, para que se pueda saber cuando existen datos nuevos en la red inalámbrica, desde el ejecutable del procesador ARM9.

- Establecer un temporizador o *TIMER* de *hardware* para realizar la actualización de las variables internas del Nintendo DS.

Los puntos anteriores son complicados de entender para un principiante y consideramos que sería una buena idea realizar una simplificación, ya que la mayor parte de los juegos requiere las mismas operaciones.

3.1.5.4 - La función de la red de un juego multijugador local en el Nintendo DS

En el caso del Nintendo DS, el tamaño físico de la consola no permite que varias personas la operen simultáneamente, es por ello que es necesario una consola para cada persona que desee participar en un juego multijugador y un medio de transporte entre las consolas para los datos relevantes del juego.

La ventaja, en el caso de los mecanismos provistos por *dswifi*, es que se trata de protocolos ya conocidos cuya efectividad ha sido comprobada, y de prácticamente aceptación universal.

Sin embargo, el tener la posibilidad de enviar datos entre las consolas de una manera sencilla es solo la solución a la mitad del problema. La otra mitad radica en la organización de dicha comunicación, ya que los juegos en red, en general requerirán lo siguiente:

- Eficiencia en la velocidad de la comunicación
- Alta tolerancia a los paquetes perdidos y fallas de la red
- Envío de paquetes a todos los jugadores simultáneamente, o casi simultáneamente
- Facilidad y rapidez para localizar a los jugadores que deseen participar
- Detección cuando ha habido una pérdida excesiva de paquetes y algún método de corrección

El **Framework** se enfocó en los juegos multijugador en red de área local, porque presentan menos complicaciones que a través de Internet. Por ejemplo, los juegos por Internet requieren o de un servicio centralizado que actúe como intermediario para coordinar el inicio de las partidas, o que los jugadores tengan conocimientos básicos sobre configuración de muros cortafuegos y su dirección IP pública, además de ser más propensos a las desconexiones y que el tiempo de retraso sea mayor.

Posteriormente, se planea continuar el desarrollo del **Framework** para brindar facilidades para realizar juegos multijugador por Internet.

Para los juegos multijugador a través de la red de área local el protocolo UDPv4 resulta muy atractivo, ya que tiene las siguientes ventajas:

- Se pueden enviar paquetes a todos los participantes simultáneamente utilizando la dirección *broadcast*[37] como destinatario. Esto también permite localizar a los participantes sin que tengan que realizar ninguna configuración por su parte más allá de haberle suministrado al Nintendo DS la información necesaria para conectarse al *access point*. Sería ideal enviar datos a través de grupos *multicast*[38], pero desafortunadamente no están disponibles algunas funciones relativas al los grupos susodichos y la documentación de *dswifi* es prácticamente inexistente
- Los paquetes perdidos no causan un retraso. Esto es simultáneamente una desventaja, ya que no hay manera de saber si un paquete llegó en buen estado o no

En la mayoría de los casos quizás no se requiera una arquitectura de red cliente-servidor, pero aún así es necesario algún tipo de coordinación.

Para este efecto, se podría considerar al primer jugador que quisiera iniciar un juego como el organizador; su consola estaría encargada de transmitir todos los mensajes importantes como inicio y fin de juego y hasta cierto punto podría moderar la interacción entre las demás consolas.

37 Una difusión, modo de transmitir información de un nodo emisor a una multitud de nodos receptores de manera simultánea.

38 Multidifusión, es el envío de la información en una red a múltiples destinos simultáneamente.

Un diagrama de flujo del inicio de un juego multijugador de red de área local con este esquema podría ser el de las ilustraciones 14 y 15.

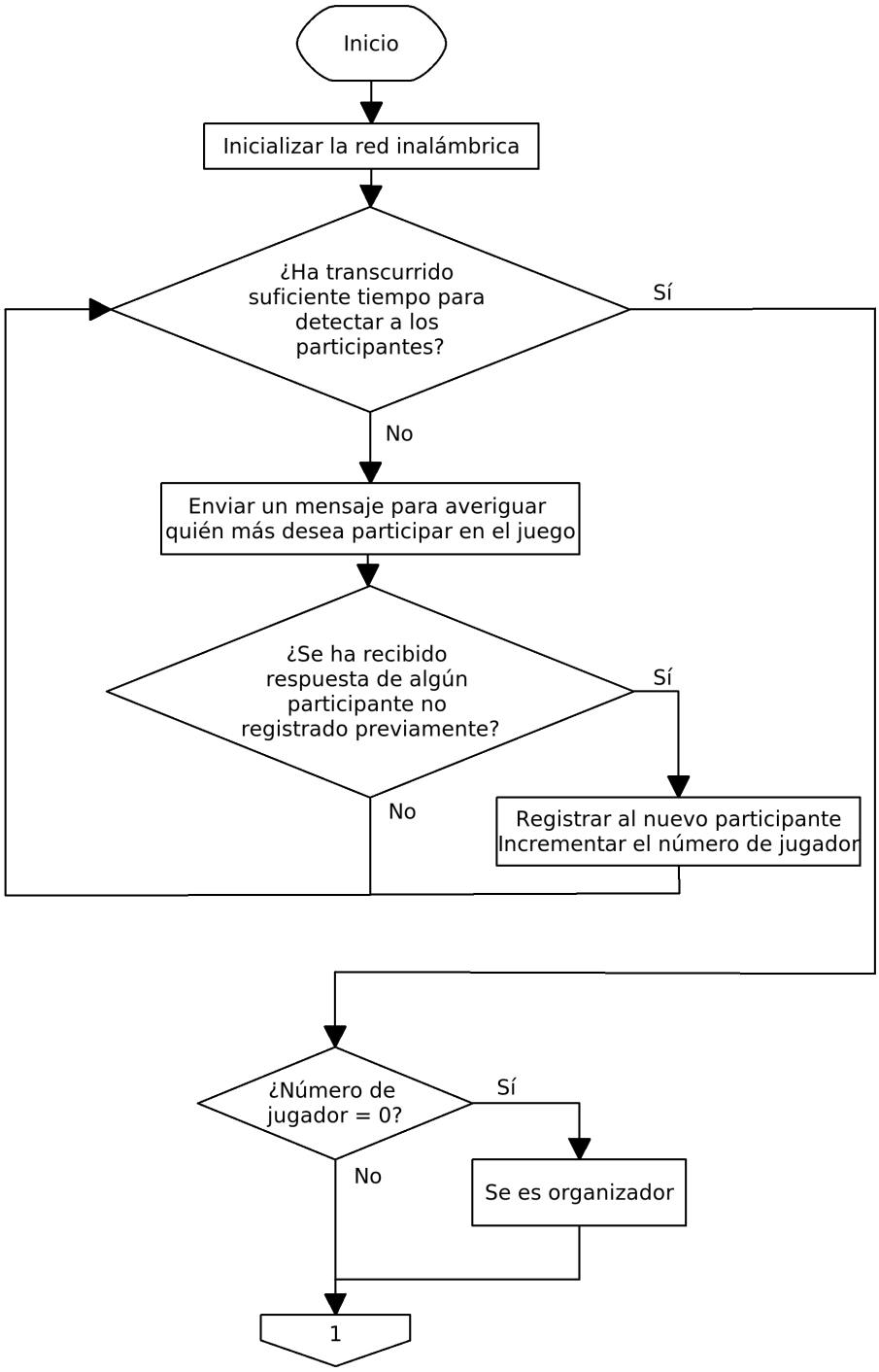

Ilustración 14: Diagrama de flujo del inicio de un juego multijugador de red LAN

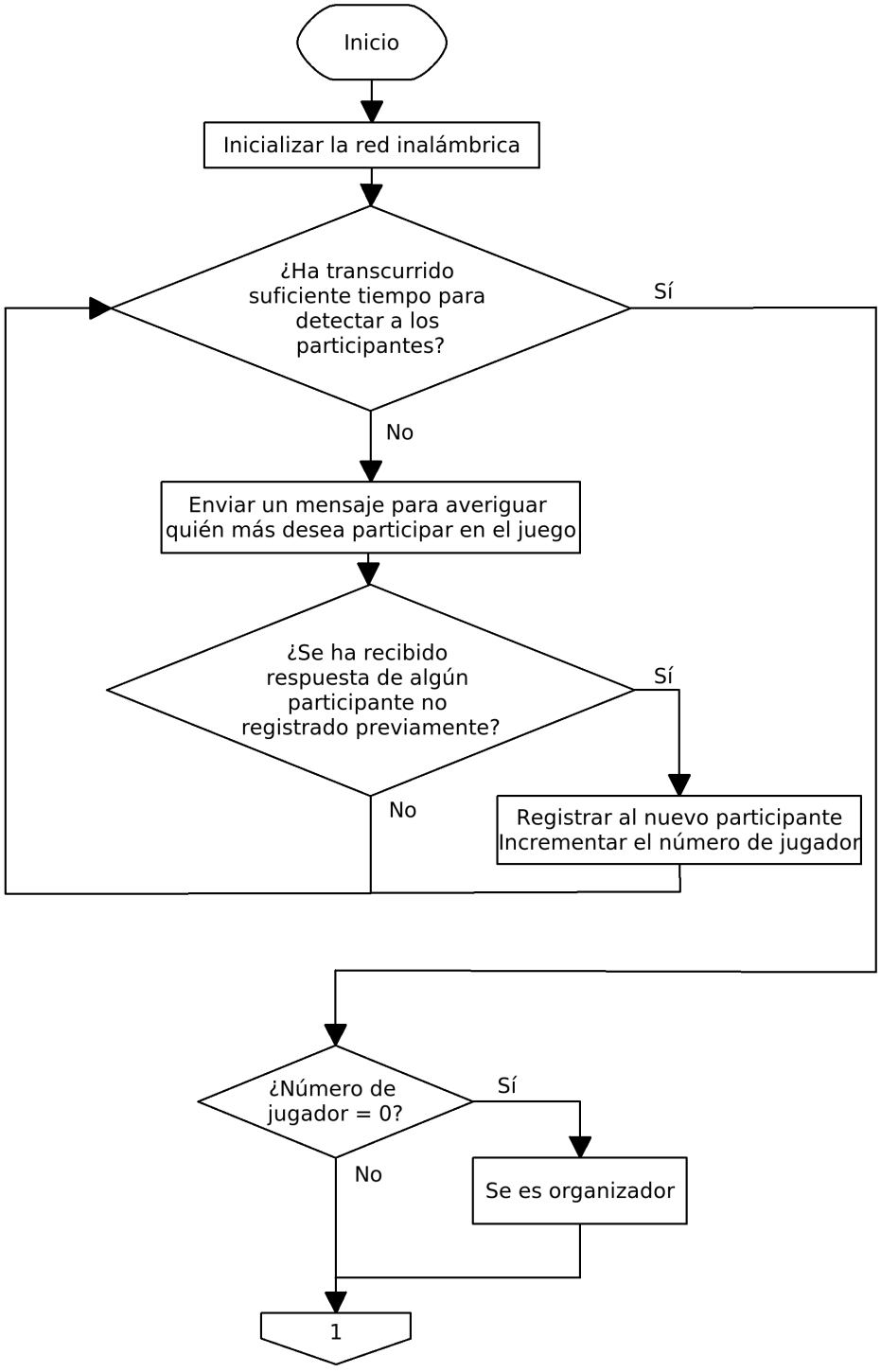

Ilustración 15: Diagrama de flujo del inicio de un juego multijugador de red LAN (continuación)

Los datos comunicados durante el transcurso del juego dependen mucho de la naturaleza del mismo, sin embargo, comúnmente serán:

- Ubicación en el mundo virtual
- Dirección
- Acción
- Sonido a emitir
- Velocidad
- Número de secuencia de animación
- Cuadro de la secuencia de animación en el que se encuentra

Estos datos generalmente serán enviados periódicamente y en el momento en el que ocurran cambios en alguna variable que no puedan ser previstos; por ejemplo, un cambio de acción provocado por el usuario, un cambio de dirección provocada por una colisión, etc., de manera que se pueda predecir lo más posible usando interpolación y se mantenga el uso de la red al mínimo, pero que esto no cause una discrepancia muy notoria de la situación del mundo virtual de una consola a otra.

Cada consola será entonces responsable de enviar los datos del jugador de dicha consola cuando sea pertinente, así como realizar detección de colisiones y actualizar las variables físicas únicamente del personaje controlado por el jugador. Con respecto a los otros jugadores, solo se realizará una interpolación de las posiciones en base a los últimos datos recibidos (velocidad, ubicación, etc.).

Una vez que se cumpla la regla de salida, la consola considerada como organizador enviará el mensaje de fin de juego y se declarará a alguien ganador.

3.1.5.5 - Implementación del manejo de red inalámbrica en MotorJ

En MotorJ, *dswifi* se encuentra embebido y listo para ocuparse; para este fin se provee un ejecutable para el procesador ARM7 que ya tiene las subrutinas necesarias para la actualización de la red inalámbrica; también se provee una estructura de comunicación interprocesador que tiene los mensajes requeridos para notificar a ambos procesadores de la presencia de datos para recibir o enviar a través de la red inalámbrica, y se manda llamar la rutina de actualización de variables internas desde un temporizador compartido con la actualización de la clase que procesa la música de fondo, ya que el Nintendo DS solo tiene 4 temporizadores de *hardware*.

Para emplear la red inalámbrica, basta con solicitarle al objeto *NetworkCTL*, que es una propiedad de la variable global *App*, que se conecte a un *Access Point* previamente configurado.

Esto se realiza en dos pasos: primero, mediante el método *dsStartAutoConnect()*. En cuanto el método *Status()* indique que el equipo de red está listo, se debe llamar al método *dsContAutoConnect()*.

Consultando nuevamente el método *Status()*, será posible saber si la consola fue capaz de conectarse y obtener una dirección IP válida para la red (ya sea almacenada en el firmware o por medio del protocolo DHCP; ésto se realiza automáticamente por la biblioteca *dswifi*), en cuyo caso se está listo para transmitir a través de la red inalámbrica, o si la consola no fue capaz de conectarse, para intentarlo otra vez.

Una vez que el Nintendo DS esté conectado, a través de la biblioteca *dswifi* es posible crear *sockets* de tipo TCP y UDP para las versiones 4 y 6 de ambos protocolos de manera muy similar a los *sockets* POSIX; sin embargo, los *sockets* POSIX no son sencillos de entender para un principiante, así que se creó una clase que facilita la creación y envío de datos a través de *sockets* TCP y UDP en modo cliente.

3.1.5.6 - La clase ClientSocket()

La clase *ClientSocket* sirve para facilitar la transmisión de datos a través de la red usando protocolos UDP o TCP v4 o v6.

Al crear una instancia de un *ClientSocket*, se debe especificar qué protocolo y versión usar, así como el puerto a abrir.

En vista de que en el Nintendo DS no existe un sistema operativo que provea funciones de administración de tareas, los juegos se verán forzados a realizar operaciones de recepción de datos como parte de su ciclo principal. Es por esto que se tomó la decisión de establecer automáticamente la propiedad de no-bloqueador en el *socket* creado por *ClientSocket*, a diferencia de los estándares POSIX. De esta manera no se detendrá la ejecución del programa si no se satisface el número de bytes esperados al intentar recibir datos; en vez de eso, se retornará “-1” como número de bytes recibidos y la ejecución continuará.

Una vez creada la instancia de un *ClientSocket*, se debe establecer su destinatario. Mediante el método DoConnect*()* se le puede especificar una dirección IP o un nombre de dominio; en caso de tratarse de un nombre de dominio, se realizará la resolución de dicho nombre a su dirección IP correspondiente y si se trata de un *socket* de tipo TCP, se intentará la conexión. Los *sockets* de tipo UDP no tienen como tal una conexión, únicamente se les configura el destinatario pero no es posible saber si efectivamente el destinatario deseado está recibiendo los datos. En el caso de haber ocurrido algún fallo, ClientSocket regresa un código de error que permite saber con precisión lo que no funcionó. Si la instancia es de tipo

UDP, como destinatario se puede establecer la dirección de *broadcast*, 255.255.255.255, de esta manera será posible enviar datos a todos los dispositivos de la red simultáneamente así como recibir de cualquiera de ellos.

Al haber logrado ejecutar el método DoConnect*()* de manera exitosa, la instancia estará lista para transmitir y recibir datos.

Utilizando el método SendData*()* se enviarán los datos especificados al destinatario; utilizando el método RecvData*()* se intentará recibir el número de bytes especificado. En el caso de UDP y de estar empleando la dirección *broadcast* como destinatario, es posible saber de qué IP provienen los datos empleando el método Who*()* inmediatamente después de haberlos recibido.

Una vez que ya no se requiere más, se puede realizar la desconexión de la instancia utilizando el método DoDisconnect*()*.

3.2 - Detección de colisiones

3.2.1 - Importancia de la simulación de la física en los videojuegos

La simulación de la física en los videojuegos determina en gran parte los sucesos dentro del videojuego y además se requiere que dicha simulación tenga un balance muy preciso entre el costo del cálculo y el realismo aparente, es decir, que se vea suficientemente real aún cuando la aplicación realice dicha simulación en tiempo real.

La mayor parte de las veces se requiere calcular colisiones contra todos los objetos del mundo virtual, para lo cual se implementaron diversos algoritmos que tienen diferente costo y pueden servir para diferentes propósitos.

3.2.2 - Algoritmos de detección de colisiones implementados

Algunos de los algoritmos de detección de colisiones implementados son los siguientes:

- Detección de colisiones por distancia y entre esferas

  Uno de los algoritmos de detección de colisiones menos costoso implementado en el *Framework* es la detección de colisiones por distancia.
  Se calcula la distancia entre dos puntos en el espacio y se determina si es menor a un valor previamente establecido; si la distancia entre los objetos es menor a dicho valor, se considera que hubo una colisión.

  Se trata de una variante del algoritmo de detección de colisiones entre dos esferas, en el cual se hace también un cálculo de la distancia, y el valor para comparar se establece con la suma de los radios de ambas esferas.

  La detección de colisiones por esferas, si no se ocupa para determinar colisiones entre dos objetos de forma esférica, se emplea generalmente como un primer paso para averiguar si dos objetos están suficientemente cerca como para emplear más cálculos en determinar si están colisionando con un mayor detalle. En algunos casos, a pesar de que los objetos no sean esféricos, es posible presentar una simulación de una colisión suficientemente realista con detección de colisiones entre esferas si el objeto tiene medidas uniformes.

- Detección de colisiones entre un punto y una caja alineada con los ejes

  Este algoritmo de detección de colisiones determina si un punto se encuentra dentro de una caja imaginaria cuyas caras son paralelas a los planos XY, XZ y ZY. Para esto, se suministran dos puntos correspondientes a las esquinas con las medidas más negativas y las medidas más positivas de la caja.

  A continuación se compara la posición del punto con las esquinas de la caja. Si todas las medidas del punto están entre los límites inferior y superior determinados por las esquinas de la caja que fueron suministradas, el punto se encuentra dentro de la caja.

  Este algoritmo se puede ocupar para determinar si un objeto se encuentra dentro de una cierta región, por ejemplo, un cuarto dentro de un edificio o una sección de un escenario.

- Detección de colisiones entre dos cajas alineadas con los ejes

  La detección de colisiones entre dos cajas alineadas implica determinar si dos cajas alineadas con los ejes se traslapan entre sí.

Tomando los dos puntos suministrados para representar una caja alineada con los ejes del algoritmo anterior, más otros dos puntos correspondientes a una segunda caja.
Para cada eje, el algoritmo retorna que no hay colisión (no hay traslape) si la componente mínima de la primer caja es mayor que la componente máxima de la segunda caja o si la componente máxima de la primer caja es menor que la componente mínima de la segunda caja.

Si nunca ocurre el caso anterior, se concluye que las dos cajas se traslapan.

3.2.3 - El árbol de detección de colisiones, *mjColTree()*

El árbol de colisiones es una clase que permite establecer varios algoritmos de detección de colisiones para que se ejecuten secuencialmente. Tiene como objetivo poder tener fácilmente un cálculo de colisiones con diferentes niveles de detalle, a fin de no ocupar el procesador innecesariamente calculando colisiones complejas entre dos objetos que están muy distanciados entre sí.
Por ejemplo, se podría concatenar primero un algoritmo de detección de colisiones de distancia, que es muy rápido de calcular, y si se detecta que los objetos están suficientemente cerca como para colisionar, se realiza a continuación un algoritmo de detección de colisiones más compleja, por ejemplo, un cálculo de colisiones entre cajas orientadas.

La forma de operación del árbol está pensada para ser sencilla. Primeramente se seleccionan los algoritmos a ejecutar y se van agregando al árbol, estableciendo los parámetros para realizar cada uno de los algoritmos seleccionados y qué resultado se espera para ir avanzando en los niveles. Si en algún nivel anterior al último no se cumple el resultado esperado, se deja de procesar el árbol y se regresa el número de nivel con el signo invertido. Al llegar al último nivel, se retorna el resultado que emita la función de detección de colisiones final. Todos los valores que emiten las funciones de detección de colisiones son mayores o iguales a cero, por lo que es posible determinar si se terminó de procesar el árbol o se quedó en algún paso intermedio.

Algunos algoritmos de detección de colisiones retornan también datos para realizar una corrección de efectos cuánticos; si es el caso, el árbol de colisiones modificará la variable marcada como “corrección”. El programador debe tener cuidado de saber cuando el algoritmo que está usando tiene la posibilidad de emitir tal corrección.

El funcionamiento del árbol de detección de colisiones se muestra en la ilustración 16:

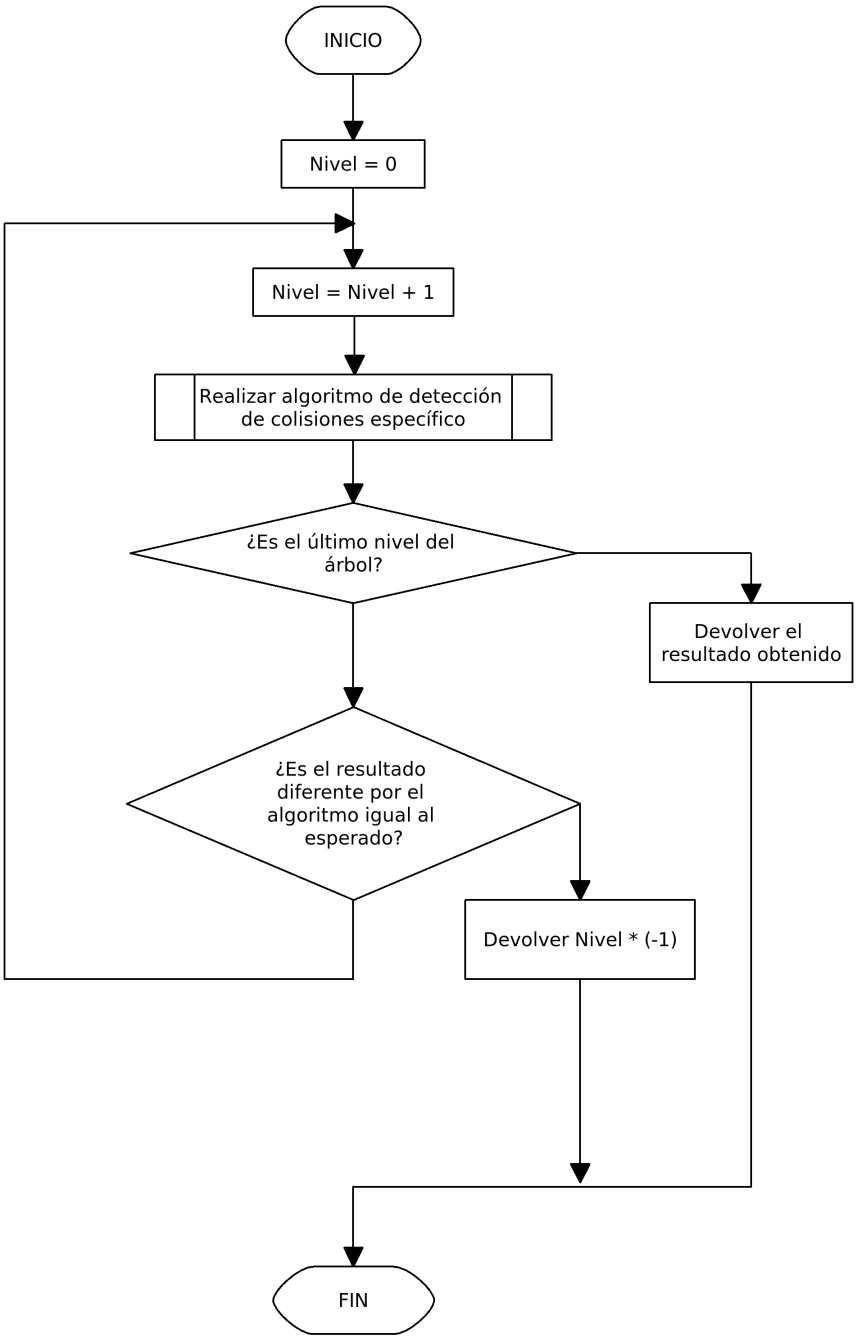

Ilustración 16: Funcionamiento del árbol de detección de colisiones, mjColTree

3.3 - Pruebas

Con base en el modelo de prototipos que se definió en el Diseño se realizó el desarrollo del **Framework**. Gran parte de la implementación y la manera en que se resolvieron los problemas, fué investigando y realizando una programación óptima para cada objetivo, todo a prueba y error, por que no se tenían ideas claras para realizar un diseño inicial.

En los siguientes subtemas se explicarán los problemas a los que se enfrentó y como fueron resueltos para lograr cumplir los objetivos de manera satisfactoria.

Para entender esta descripción se presenta la ilustración 17, que ejemplifica claramente la composición general de las clases, que clase depende de que módulo y como interactúan. Si se necesita mayor referencia en alguna clase favor de consultar el API correspondiente, contenidas en el Apéndice.

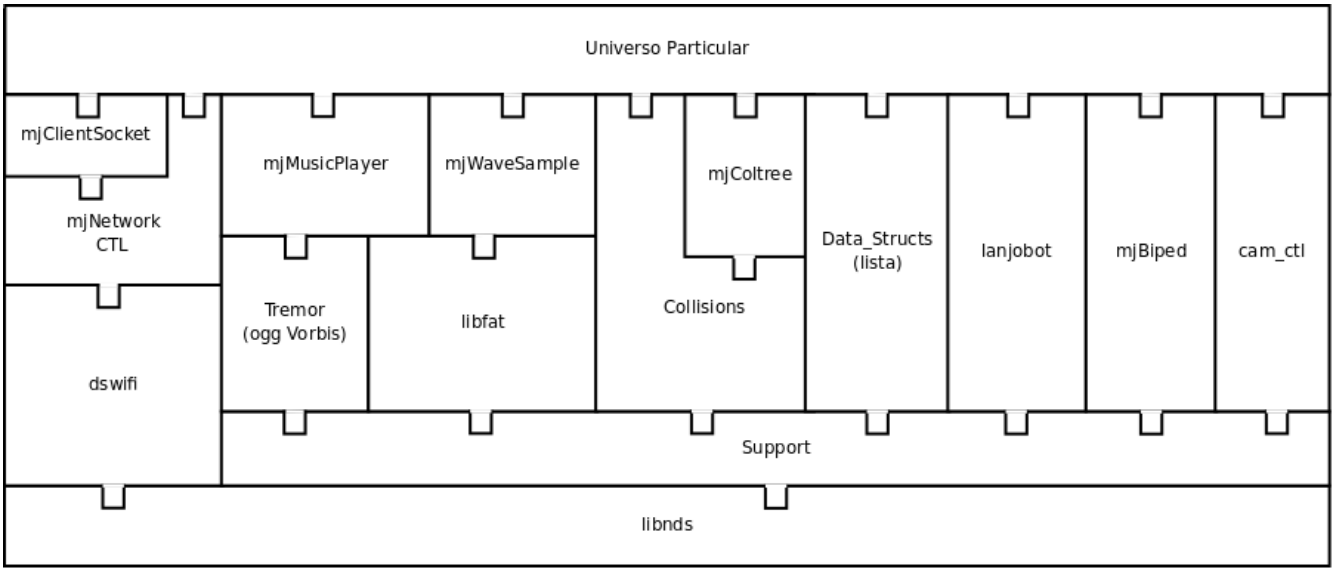

*Ilustración 17: Diagrama general de Clases del **Framework***

Las pruebas fueron realizadas a los siguientes módulos y clases:

- Humanoide “Lanjobot”.
- Cámara.
- MusicPlayer.
- Cubo *Display List*.
- Movimiento de caminata y correr.
- Modelo poligonal de Blender.
- Wifi.
- Wifi, varios jugadores en un mismo espacio virtual.
- Efectos de sonido Wav.
- Generador de terrenos y detección de colisiones.
- Versión final del MusicPlayer.
- Estructuras de detección de colisiones.
- Biped.
- Prueba de Concepto.

Muchas de las pruebas se realizaron usando el emulador libre del Nintendo DS, Desmume, un programa que simula el *hardware* de la consola, para realizar una prueba rápida sin tener que cargar el ejecutable en la tarjeta y probarlo en la consola. Cuando ya se contaban con versiones funcionales de las pruebas, éstas eran ejecutadas en el *hardware* real, al igual que las versiones preliminares que requerían dispositivos que no era posible emular, como las conexiones inalámbricas.

3.3.1 - Humanoide “Lanjobot”

Cuando se logró adaptar MotorJ al finalizar la compilación correctamente con el sistema de compilación del Nintendo DS, se probó primero con un triángulo generado con comandos de OpenGL; posteriormente se intentó mostrar un humanoide animado hecho por cubos. Fueron necesarios varios ensayos para detectar un comportamiento extraño causado por la inclusión de las instrucciones de OpenGL **glNormal3f**. Una vez que éstas fueron retiradas, el humanoide animado funcionaba de la manera esperada.

3.3.2 - Cámara

A la prueba anterior se le estableció una cámara virtual, la cual inicialmente no funcionó como se esperaba ya que la escala empleada en el Nintendo DS difiere significativamente de la empleada en la PC. Tras varios intentos se logró que el desplazamiento de la cámara fuera lo suficientemente suave aún con las coordenadas empleadas por el Nintendo DS.

3.3.3 - MusicPlayer

Para conseguir que funcionara el reproductor de música, hubo que adaptar el código fuente de Tremor, la implementación de referencia para dispositivos embebidos del decodificador del formato Ogg Vorbis. Una vez que dicha biblioteca no presentó errores de compilación, las primeras pruebas hicieron evidente que el Nintendo DS no era capaz de reproducir archivos Ogg Vorbis en estéreo sin que la fluidez de la animación bajara considerablemente. Se optó por probar con archivos Ogg Vorbis monoaurales; el uso de dichos archivos redujo considerablemente la actividad del procesador, lo suficiente para poder funcionar simultáneamente a la animación. Se estructuró la clase mjMusicPlayer para ser similar a un reproductor de discos compactos en cuanto a funcionalidad y se definió un temporizador de *hardware* por omisión para realizar la descompresión periódica de los datos de música. El componente se agregó a la prueba anterior para su verificación.

3.3.4 - Cubo DisplayList

Durante el despliegue del lanjobot se notó que no estaba siendo desplegado rápidamente, de modo que se optó por buscar alguna alternativa a las funciones de OpenGL para crear las geometrías y tratar de mejorar la cantidad de despliegues en pantalla en cada segundo.

Esta aplicación se tuvo que investigar e implementar para tener una mayor capacidad de despliegue de polígonos reemplazando una clase del MotorJ, el mjSolidCube(), en su versión para el Nintendo DS. Se pretendía que su primera aplicación fuera la de construir al lanjobot con *Display List*, que anteriormente estaba formado por funciones de OpenGL. En pruebas posteriores, se encontraría que la razón de la lentitud en primer lugar era en realidad el paso de la animación especificado era demasiado pequeño, pero aún así la alternativa investigada presenta una ventaja muy significativa al ser usada con una cantidad muy grande de polígonos.

La primera problemática a la que se enfrentó fue la falta de información. Pese a que existe este tipo de estructuras en OpenGL, la consola Nintendo DS requiere de una forma muy particular que no es el estándar. La única fuente realmente confiable era los comentarios del archivo videoGL.h de la biblioteca **libnds**. En los desarrollos *Homebrew* era un misterio como funcionaban, de modo que se resolvió eso descubriendo su estructura y funcionamiento mediante pruebas.

Por simplicidad se decidió no emplear todas las funcionalidades del *Display List* en el **Framework** ya que complicaría demasiado todos los demás módulos, únicamente despliegue de vértices para formar caras de las geometrías.

3.3.5 - Movimiento de caminata y correr

Utilizando el humanoide animado formado por bloques (lanjobot), se asignó el movimiento en direcciones relativas a la cámara a los controles de cruz direccional. El botón B se asignó a cambiar la animación de caminata por la animación de correr y a que se desplazara más rápidamente. El lanjobot caminó entonces en la dirección especificada sobre un plano infinito. Algunas personas ajenas a la tesis que utilizaron la prueba se percataron de algunos movimientos extraños en ciertas direcciones, lo cual sirvió para encontrar un error en las operaciones que realizábamos para procesar los controles. El error fue entonces reparado.

3.3.6 - Modelo poligonal de Blender

Pese a que se realizó un cubo para el lanjobot, y se podía animar de manera óptima, se regresó al problema de creación de *Display List* de manera efectiva, y no estar definiendo vértice por vértice en la estructura.

Con el uso de un **Framework** se pretende facilitar el desarrollo de aplicaciones y en vista que se busca que este **Framework** sea usado por otras personas y buscando facilitar su trabajo, se pensó en la forma de crear *Display List* de alguna otra manera más efectiva.

Las opciones eran: usar desarrollar algún modelador 3D o emplear alguno ya existente. La elección fue emplear a Blender por ser un desarrollo libre. Para esto se investigó la manera de crear un exportador en ese programa, la manera en que se obtienen los datos que se necesitan de la escena activa y cómo usar Blender como modelador.

Existieron algunos problemas usando el lenguaje interpretado Python por su sintaxis y en la creación del *script* de exportación al tratar de obtener la salida deseada para el *Display List*.

A mucha prueba y error se encontró la manera de definir materiales, hacer un correcto recorrido de las caras, saber si son triángulos o cuadriláteros, etc.

La propia estructura del *Display List* complicó demasiado la exportación. Ya que este archivo indica lo que seguirá a continuación, por ejemplo, la primera línea es el número de comandos que habrá en el *Display List*, los `FIFO_COMMAND_PACK`, dicen en paquetes de 4, que tipo de comandos seguirán a continuación, etc.

Con este exportador se encontró que el despliegue en coordenadas mayores a un cubo de (-4.7,-4.7,-4.7) a (4.7,4.7,4.7) aproximadamente, tiene un comportamiento muy extraño y diferente al deseado, puesto que no se despliegan las geometrías correctamente. Es por eso que en el **Framework** se recomienda que las dimensiones en las geometrías no excedan del cubo (-4.0,-4.0,-4.0) a (4.0,4.0,4.0), por tener un comportamiento ideal en esas regiones.

Con el exportador de Display List se logró desplegar en pantalla más de 120,000 polígonos en una geometría UVSphere, sin complicaciones para la consola.

A Blender se le ha criticado su dificultad de uso y por tener una interfaz poco intuitiva para inexpertos. De modo que se pensó también en hacer un mini manual de uso de Blender tratando de resolver la mayoría de las problemáticas del modelado para uso del exportador en el **Framework**. Este manual se encuentra en el Apéndice de la tesis.

3.3.7 - Wifi

Después de lograr entender cómo establecer los subsistemas de la red inalámbrica (comunicación interprocesador, temporizador de *hardware*, etc.), se logró realizar la asociación y disociación de la consola a un *Access Point* ordinario. Una vez lograda la asociación a un *Access Point*, el objetivo de la prueba cambió a ser crear un *socket* no bloqueador UDP que pudiera transmitir y recibir datos hacia y desde la dirección *broadcast*, 255.255.255.255.

Se logró la comunicación entre dos consolas, sin embargo el *socket* estaba siempre en modo bloqueador. Al probar infructuosamente numerosas veces establecer las propiedades del *socket* como no bloqueador usando el API POSIX, se optó por probar una función aparentemente relacionada, casi sin documentación, encontrada en las cabeceras de *dswifi*. Dicha función resultó útil y se decidió construir una clase que facilitara el uso de *sockets* a personas que no tienen experiencia en el manejo de *sockets* POSIX. A dicha clase le llamó mjClientSocket. Se comprobó entonces la comunicación con la clase mjClientSocket.

3.3.8 - Wifi, varios jugadores en un mismo espacio virtual

Tras haber completado la comunicación con la clase mjClientSocket, se agregó a la prueba pasada del movimiento de caminata y correr. A través de dicha clase, se estableció comunicación entre dos consolas Nintendo DS y se enviaron paquetes con la posición y el estado del lanjobot controlado por cada jugador, de manera que se mostraran en pantalla los dos humanoides. Se hicieron varias pruebas enviando actualizaciones a distintos intervalos e interpolando los estados y posiciones del jugador remoto en cada consola entre cada actualización recibida. Se concluyó que el comportamiento ideal resultaba de enviar 15 actualizaciones cada segundo, además de cuando variaba el estado de los controles (y por lo tanto la dirección o rapidez del humanoide controlado por el jugador).

3.3.9 - Efectos de sonido Wav

Se investigó la estructura interna del formato Wave más común (debido a que no es un estándar, existen numerosas versiones) y se implementó una clase que pudiera abrir dicho formato mientras fuera monoaural. Inicialmente se implementó solo un método para reproducirlo con atenuación y balance específicos, sin embargo se investigó una manera calcular la atenuación y el balance para simular la posición física de la fuente de sonido con respecto al observador y se creó otro método para implementarla.

3.3.10 - Generador de terrenos y detección de colisiones

Como Blender resultaba un entorno muy adecuado para generar modelos tridimensionales para personajes y objetos pero muy complicado de usar y extender para generar escenarios, se decidió crear una herramienta que auxiliara en la generación de terrenos formados por triángulos, usando MotorJ en su versión para PC.

Utilizando el generador de escenarios se elaboró un terreno hecho con triángulos y se ajustó el código de la prueba de caminata a fin de que respondiera a detección de colisiones contra el terreno. Al probar varias veces, se detectó una anomalía en el cálculo la posición resultante que emite dicho algoritmo y se arregló el error. Al probar con distintos terrenos se descubrió otro error que ocurría con el algoritmo de detección de colisiones cuando se traslapaban verticalmente dos partes del mismo terreno. Para solucionar esto, se propuso dividir en “regiones” el terreno, lo que promovió a su vez el desarrollo de la clase mjColTree para realizar detección de colisiones de manera más automatizada. Dicha clase sería útil también para posteriores pruebas.

3.3.11 - Versión final del MusicPlayer

Al concluir la prueba del terreno y la detección de colisiones, se incluyó el mjMusicPlayer. Se le agregó al mjMusicPlayer la funcionalidad de cambio de pista sonora y se verificó que funcionara; asimismo se modificó la cantidad de datos de sonido descomprimidos cada segundo a fin de hacerlo más tolerante a una actividad gráfica demandante.

3.3.12 - Clase mjBiped

La conjunción de las aplicaciones de *Display List* en la tesis, es la implementación de los modelos obtenidos del exportador de Blender en una clase para ser desplegada en pantalla como personajes de los videojuegos.

Tratando de ser congruente con nuestras aplicaciones, nos basamos en el lanjobot para construir bípedos humanoides.

Se creo la clase mjBiped(), a la cual se le pasarían una lista de *Display List* y la posición que llevarían para ser construidos. Las complicaciones fueron más de diseño en la construcción de los bípedos y la posición, que en su aplicación tratando de sustituir los lanjobot en la prueba de concepto.

Se definieron los mismo movimientos que tenía el lanjobot, pero ahora en una forma más general para cualquier humanoide.

3.3.13 - Prueba de Concepto

La prueba de concepto, desarrollada para demostrar todo el **Framework**, como tal, utiliza todas las clases creadas ya que es un videojuego pequeño multijugador con personajes poligonales distintos, música, efectos de sonido, simulación de física básica (detección de colisiones, gravedad), escenario y reglas de ganar/perder.

Los jugadores pueden explorar un escenario creado a base de triángulos de distintos tamaños e inclinaciones haciendo caminar a su personaje mediante las teclas direccionales de la consola. Al presionar el botón **B**, el personaje correrá en vez de caminar, mientras que al presionar el botón **A**, dará un salto. La simulación de física incluye simulación de la aceleración debido a la gravedad: el personaje controlado por el jugador puede caer al vacío si no es cuidadoso y se sale del terreno; en este caso, el personaje aparecerá nuevamente en un lugar al azar.

El juego está diseñado para permitir interacción entre varias personas, cada una con su consola, a través de un *Access Point*. Al presionar el botón **START**, las consolas intentarán enlazarse y se coordinarán de manera que la primer consola en lograr la conexión determinará el momento del inicio del juego, además de la posición inicial de otros objetos dentro del mismo, es decir, dicha consola estará en modo organizador. Los demás jugadores que se unan al juego enviarán una notificación al organizador para que éste presione nuevamente **START** para comenzar la partida.

Al comenzar la partida multijugador, los participantes y un cubo giratorio que cambia de color aparecerán en posiciones aleatorias sobre el terreno. El objetivo es llegar hasta el cubo antes que los demás participantes, ya que al pasar sobre él, el personaje lo obtiene. Una vez que se obtiene el cubo, el objetivo es regresar al lugar a partir del cual el jugador comenzó, el cuál estará marcado con algún símbolo propio de su personaje. Los demás participantes deberán entonces tratar de impedir que quien tiene el cubo regrese a su lugar de origen, ya que si lo consigue se le sumará un punto a su marcador.
Para impedir que quien obtuvo el cubo anote un punto, los demás participantes pueden chocar contra él, en cuyo caso ambos serán desviados, darle un golpe presionando el botón **Y**, en cuyo caso quien recibe el golpe será repelido, así como saltar sobre él para robarle el cubo. Si alguien cae al vacío mientras tiene el cubo, ambos serán puestos nuevamente de manera aleatoria en algún lugar del terreno. Al caer al vacío con o sin el cubo se pierde un punto.
Cuando alguien consigue anotar un punto, el cubo es colocado aleatoriamente en algún lugar del terreno para que nuevamente compitan por él.

El primer jugador en juntar 5 puntos será declarado ganador y el juego comenzará de nuevo.

La música empleada durante la partida puede ser cambiada presionando el botón **SELECT**.
Las pistas sonoras son *empty* por *Grace Valhalla, Casa*, por *Vate*, y *Artificial Creation* por *Subversive Activity*.

Todas las pistas sonoras empleadas fueron liberadas bajo términos *Creative Commons*[39] que permiten su uso por otras personas libremente, basta con mencionar el nombre del autor; algunas de ellas incluso permiten ser modificadas. Estos términos favorecen la cooperación entre personas, de maneras completamente opuestas a los términos tradicionales bajo los que se rigen la mayoría de las pistas sonoras y representan un modelo viable de producción muy parecido al del *software* libre.

Para lograr realizar la prueba de concepto final se perfeccionaron la mayor parte de las piezas del *Framework*, ya que en varios casos la implementación planeada no tenía la utilidad esperada durante la ejecución, por lo que tuvo que ser cambiada a una más práctica.

Algunas de las modificaciones que se realizaron fueron:

- Los datos del árbol de detección de colisiones requirieron ser almacenados de manera pública para permitir usar un mismo árbol con diferentes objetos, por ejemplo, un mismo árbol para calcular colisiones con diferentes jugadores.
- El reproductor de música requirió cambios para tolerar un número de cuadros por segundo más variable que en las otras pruebas.
- Se redujo el número de operaciones necesarias en los *sockets* de cliente al eliminar la funcionalidad de filtrar sus propias emisiones por el protocolo UDP, que resultó no solo no tener utilidad real, sino presentar resultados erróneos, pero que en concepto parecía conveniente.
- Se decidió dividir la animación del bípedo en varios sectores a fin de poder presentar movimientos compuestos como los de soltar un golpe y correr simultáneamente.

39 Licencias de uso libres. Mayor información: http://creativecommons.org/

4 - Conclusiones

El desarrollo del **Framework** ha sido un trabajo arduo. Pese a que fue basado en MotorJ, se tuvieron que reconstruir muchos módulos y a veces partir desde cero para lograr las aplicaciones. Fue mucho trabajo de investigación en secciones desconocidas y se encontró con la falta de información específica.

En el **Framework** se pretendió definir un estándar a seguir, para que no fuera complicado para el usuario entender el manejo de las funciones y clases, al igual que facilitar la búsqueda en los directorios.

Se logró desarrollar la mayoría de los objetivos que definimos en los alcances. Aunque cabe decir que faltaron un par de aspectos tales como facilitar el uso de *sockets* tipo servidor y texturizar los modelos geométricos, se deja para futuras implementaciones dichos aspectos; además de que tal y como se planteó en uno de los objetivos, el trabajo queda abierto a contribuciones y adaptaciones para cualquier aplicación específica por parte de los usuarios.

En la prueba de concepto se pretende dar a conocer las posibilidades del **Framework** y aplicar las utilidades, las cuales son:

- Facilitar el desarrollo de videojuegos para la consola Nintendo DS.
- Uso de casi todo el *hardware* del Nintendo DS, exceptuando el micrófono, memoria auxiliar en puerto GBA y las capacidades gráficas en la pantalla secundaria.
- Comunicación inalámbrica entre consolas.
- Detección de colisiones.
- Reproducción de música de fondo.
- Efectos de sonido para las acciones.
- Uso de la pantalla táctil en el control de la cámara.
- Demostrar un ejemplo del generador de terrenos.
- Uso de personajes diseñados en Blender.
- Demostración del desarrollo de un proyecto usando únicamente *software* y arte libre.

El presente proyecto no habría sido posible sin el *software* libre, ya que los kits de desarrollo para Nintendo DS privativos, además de ser de venta y distribución restringida, son demasiado costosos.

Nuestro proyecto fue desarrollado en el **Laboratorio de Investigación y Desarrollo de Software Libre de la Facultad de Ingeniería** (LIDSOL) y está basado y sustentado en las bibliotecas y herramientas libres **Libnds**, **dswifi**, **tremor**, **DevkitARM**, **DevkitPRO**, **Desmume, Blender** y **SubVersion**, compilando en **Debian** y **Ubuntu GNU/Linux**. De la misma manera, con este proyecto, colaboraremos a la comunidad de desarrollo libre de videojuegos liberándolo bajo una licencia libre.

Durante el desarrollo de los objetivos de la tesis, la investigación y recopilación de información fue constante, así como la prueba y error de ideas propias. Dicho proceso hace apreciar las dificultades de programar en una plataforma con un *hardware* limitado y por consiguiente concientiza sobre la importancia de desarrollar código eficiente.

Por otro lado, se aprecia a los desarrollos libres en los que fue basado el **Framework** por su claridad en el código y la importancia de una buena documentación. Por eso se decidió realizar una exhaustiva documentación de las funciones, clases y herramientas que conforman el **Framework** en el apéndice, esperando que esto facilite la labor de futuros desarrolladores.

Con la liberación del **Framework** al público en general, se espera que despierte entusiasmo por el desarrollo de videojuegos y aplicaciones gráficas afines para la consola, lo cuál se desea que a su vez favorezca la colaboración en el crecimiento del **Framework**. También se espera que pueda ser visto como un incentivo para que los alumnos de las carreras relacionadas tengan más interés en las materias teóricas al ver una aplicación específica de muchos de los temas vistos.

Se planea continuar trabajando en la actualización y expansión del **Framework** a otras consolas, buscar su mejora y fomentar el desarrollo libre por parte de quienes lo usan. Pretendiendo con esto hacer del **Framework** un gran proyecto de renombre y volverlo una opción para el desarrollo de juegos comerciales.

3.3.6 - Modelo poligonal de Blender

Pese a que se realizó un cubo para el lanjobot, y se podía animar de manera óptima, se regresó al problema de creación de *Display List* de manera efectiva, y no estar definiendo vértice por vértice en la estructura.

Con el uso de un **Framework** se pretende facilitar el desarrollo de aplicaciones y en vista que se busca que este **Framework** sea usado por otras personas y buscando facilitar su trabajo, se pensó en la forma de crear *Display List* de alguna otra manera más efectiva.

Las opciones eran: usar desarrollar algún modelador 3D o emplear alguno ya existente. La elección fue emplear a Blender por ser un desarrollo libre. Para esto se investigó la manera de crear un exportador en ese programa, la manera en que se obtienen los datos que se necesitan de la escena activa y cómo usar Blender como modelador.

Existieron algunos problemas usando el lenguaje interpretado Python por su sintaxis y en la creación del *script* de exportación al tratar de obtener la salida deseada para el *Display List*.

A mucha prueba y error se encontró la manera de definir materiales, hacer un correcto recorrido de las caras, saber si son triángulos o cuadriláteros, etc.

La propia estructura del *Display List* complicó demasiado la exportación. Ya que este archivo indica lo que seguirá a continuación, por ejemplo, la primera línea es el número de comandos que habrá en el *Display List*, los `FIFO_COMMAND_PACK`, dicen en paquetes de 4, que tipo de comandos seguirán a continuación, etc.

Con este exportador se encontró que el despliegue en coordenadas mayores a un cubo de (-4.7,-4.7,-4.7) a (4.7,4.7,4.7) aproximadamente, tiene un comportamiento muy extraño y diferente al deseado, puesto que no se despliegan las geometrías correctamente. Es por eso que en el **Framework** se recomienda que las dimensiones en las geometrías no excedan del cubo (-4.0,-4.0,-4.0) a (4.0,4.0,4.0), por tener un comportamiento ideal en esas regiones.

Con el exportador de Display List se logró desplegar en pantalla más de 120,000 polígonos en una geometría UVSphere, sin complicaciones para la consola.

A Blender se le ha criticado su dificultad de uso y por tener una interfaz poco intuitiva para inexpertos. De modo que se pensó también en hacer un mini manual de uso de Blender tratando de resolver la mayoría de las problemáticas del modelado para uso del exportador en el **Framework**. Este manual se encuentra en el Apéndice de la tesis.

3.3.7 - Wifi

Después de lograr entender cómo establecer los subsistemas de la red inalámbrica (comunicación interprocesador, temporizador de *hardware*, etc.), se logró realizar la asociación y disociación de la consola a un *Access Point* ordinario. Una vez lograda la asociación a un *Access Point*, el objetivo de la prueba cambió a ser crear un *socket* no bloqueador UDP que pudiera transmitir y recibir datos hacia y desde la dirección *broadcast*, 255.255.255.255.

Se logró la comunicación entre dos consolas, sin embargo el *socket* estaba siempre en modo bloqueador. Al probar infructuosamente numerosas veces establecer las propiedades del *socket* como no bloqueador usando el API POSIX, se optó por probar una función aparentemente relacionada, casi sin documentación, encontrada en las cabeceras de *dswifi*. Dicha función resultó útil y se decidió construir una clase que facilitara el uso de *sockets* a personas que no tienen experiencia en el manejo de *sockets* POSIX. A dicha clase le llamó mjClientSocket. Se comprobó entonces la comunicación con la clase mjClientSocket.

3.3.8 - Wifi, varios jugadores en un mismo espacio virtual

Tras haber completado la comunicación con la clase mjClientSocket, se agregó a la prueba pasada del movimiento de caminata y correr. A través de dicha clase, se estableció comunicación entre dos consolas Nintendo DS y se enviaron paquetes con la posición y el estado del lanjobot controlado por cada jugador, de manera que se mostraran en pantalla los dos humanoides. Se hicieron varias pruebas enviando actualizaciones a distintos intervalos e interpolando los estados y posiciones del jugador remoto en cada consola entre cada actualización recibida. Se concluyó que el comportamiento ideal resultaba de enviar 15 actualizaciones cada segundo, además de cuando variaba el estado de los controles (y por lo tanto la dirección o rapidez del humanoide controlado por el jugador).

3.3.9 - Efectos de sonido Wav

Se investigó la estructura interna del formato Wave más común (debido a que no es un estándar, existen numerosas versiones) y se implementó una clase que pudiera abrir dicho formato mientras fuera monoaural. Inicialmente se implementó solo un método para reproducirlo con atenuación y balance específicos, sin embargo se investigó una manera calcular la atenuación y el balance para simular la posición física de la fuente de sonido con respecto al observador y se creó otro método para implementarla.

3.3.10 - Generador de terrenos y detección de colisiones

Como Blender resultaba un entorno muy adecuado para generar modelos tridimensionales para personajes y objetos pero muy complicado de usar y extender para generar escenarios, se decidió crear una herramienta que auxiliara en la generación de terrenos formados por triángulos, usando MotorJ en su versión para PC.

Utilizando el generador de escenarios se elaboró un terreno hecho con triángulos y se ajustó el código de la prueba de caminata a fin de que respondiera a detección de colisiones contra el terreno. Al probar varias veces, se detectó una anomalía en el cálculo la posición resultante que emite dicho algoritmo y se arregló el error. Al probar con distintos terrenos se descubrió otro error que ocurría con el algoritmo de detección de colisiones cuando se traslapaban verticalmente dos partes del mismo terreno. Para solucionar esto, se propuso dividir en “regiones” el terreno, lo que promovió a su vez el desarrollo de la clase mjColTree para realizar detección de colisiones de manera más automatizada. Dicha clase sería útil también para posteriores pruebas.

3.3.11 - Versión final del MusicPlayer

Al concluir la prueba del terreno y la detección de colisiones, se incluyó el mjMusicPlayer. Se le agregó al mjMusicPlayer la funcionalidad de cambio de pista sonora y se verificó que funcionara; asimismo se modificó la cantidad de datos de sonido descomprimidos cada segundo a fin de hacerlo más tolerante a una actividad gráfica demandante.

3.3.12 - Clase mjBiped

La conjunción de las aplicaciones de *Display List* en la tesis, es la implementación de los modelos obtenidos del exportador de Blender en una clase para ser desplegada en pantalla como personajes de los videojuegos.

Tratando de ser congruente con nuestras aplicaciones, nos basamos en el lanjobot para construir bípedos humanoides.

Se creo la clase mjBiped(), a la cual se le pasarían una lista de *Display List* y la posición que llevarían para ser construidos. Las complicaciones fueron más de diseño en la construcción de los bípedos y la posición, que en su aplicación tratando de sustituir los lanjobot en la prueba de concepto.

Se definieron los mismo movimientos que tenía el lanjobot, pero ahora en una forma más general para cualquier humanoide.

3.3.13 - Prueba de Concepto

La prueba de concepto, desarrollada para demostrar todo el **Framework**, como tal, utiliza todas las clases creadas ya que es un videojuego pequeño multijugador con personajes poligonales distintos, música, efectos de sonido, simulación de física básica (detección de colisiones, gravedad), escenario y reglas de ganar/perder.

Los jugadores pueden explorar un escenario creado a base de triángulos de distintos tamaños e inclinaciones haciendo caminar a su personaje mediante las teclas direccionales de la consola. Al presionar el botón **B**, el personaje correrá en vez de caminar, mientras que al presionar el botón **A**, dará un salto. La simulación de física incluye simulación de la aceleración debido a la gravedad: el personaje controlado por el jugador puede caer al vacío si no es cuidadoso y se sale del terreno; en este caso, el personaje aparecerá nuevamente en un lugar al azar.

El juego está diseñado para permitir interacción entre varias personas, cada una con su consola, a través de un *Access Point*. Al presionar el botón **START**, las consolas intentarán enlazarse y se coordinarán de manera que la primer consola en lograr la conexión determinará el momento del inicio del juego, además de la posición inicial de otros objetos dentro del mismo, es decir, dicha consola estará en modo organizador. Los demás jugadores que se unan al juego enviarán una notificación al organizador para que éste presione nuevamente **START** para comenzar la partida.

Al comenzar la partida multijugador, los participantes y un cubo giratorio que cambia de color aparecerán en posiciones aleatorias sobre el terreno. El objetivo es llegar hasta el cubo antes que los demás participantes, ya que al pasar sobre él, el personaje lo obtiene. Una vez que se obtiene el cubo, el objetivo es regresar al lugar a partir del cual el jugador comenzó, el cuál estará marcado con algún símbolo propio de su personaje. Los demás participantes deberán entonces tratar de impedir que quien tiene el cubo regrese a su lugar de origen, ya que si lo consigue se le sumará un punto a su marcador.
Para impedir que quien obtuvo el cubo anote un punto, los demás participantes pueden chocar contra él, en cuyo caso ambos serán desviados, darle un golpe presionando el botón **Y**, en cuyo caso quien recibe el golpe será repelido, así como saltar sobre él para robarle el cubo. Si alguien cae al vacío mientras tiene el cubo, ambos serán puestos nuevamente de manera aleatoria en algún lugar del terreno. Al caer al vacío con o sin el cubo se pierde un punto.
Cuando alguien consigue anotar un punto, el cubo es colocado aleatoriamente en algún lugar del terreno para que nuevamente compitan por él.

El primer jugador en juntar 5 puntos será declarado ganador y el juego comenzará de nuevo.

La música empleada durante la partida puede ser cambiada presionando el botón **SELECT**.
Las pistas sonoras son *empty* por *Grace Valhalla, Casa*, por *Vate*, y *Artificial Creation* por *Subversive Activity*.

Todas las pistas sonoras empleadas fueron liberadas bajo términos *Creative Commons*[39] que permiten su uso por otras personas libremente, basta con mencionar el nombre del autor; algunas de ellas incluso permiten ser modificadas. Estos términos favorecen la cooperación entre personas, de maneras completamente opuestas a los términos tradicionales bajo los que se rigen la mayoría de las pistas sonoras y representan un modelo viable de producción muy parecido al del *software* libre.

Para lograr realizar la prueba de concepto final se perfeccionaron la mayor parte de las piezas del *Framework*, ya que en varios casos la implementación planeada no tenía la utilidad esperada durante la ejecución, por lo que tuvo que ser cambiada a una más práctica.

Algunas de las modificaciones que se realizaron fueron:

- Los datos del árbol de detección de colisiones requirieron ser almacenados de manera pública para permitir usar un mismo árbol con diferentes objetos, por ejemplo, un mismo árbol para calcular colisiones con diferentes jugadores.
- El reproductor de música requirió cambios para tolerar un número de cuadros por segundo más variable que en las otras pruebas.
- Se redujo el número de operaciones necesarias en los *sockets* de cliente al eliminar la funcionalidad de filtrar sus propias emisiones por el protocolo UDP, que resultó no solo no tener utilidad real, sino presentar resultados erróneos, pero que en concepto parecía conveniente.
- Se decidió dividir la animación del bípedo en varios sectores a fin de poder presentar movimientos compuestos como los de soltar un golpe y correr simultáneamente.

39 Licencias de uso libres. Mayor información: http://creativecommons.org/

4 - Conclusiones

El desarrollo del **Framework** ha sido un trabajo arduo. Pese a que fue basado en MotorJ, se tuvieron que reconstruir muchos módulos y a veces partir desde cero para lograr las aplicaciones. Fue mucho trabajo de investigación en secciones desconocidas y se encontró con la falta de información específica.

En el **Framework** se pretendió definir un estándar a seguir, para que no fuera complicado para el usuario entender el manejo de las funciones y clases, al igual que facilitar la búsqueda en los directorios.

Se logró desarrollar la mayoría de los objetivos que definimos en los alcances. Aunque cabe decir que faltaron un par de aspectos tales como facilitar el uso de *sockets* tipo servidor y texturizar los modelos geométricos, se deja para futuras implementaciones dichos aspectos; además de que tal y como se planteó en uno de los objetivos, el trabajo queda abierto a contribuciones y adaptaciones para cualquier aplicación específica por parte de los usuarios.

En la prueba de concepto se pretende dar a conocer las posibilidades del **Framework** y aplicar las utilidades, las cuales son:

- Facilitar el desarrollo de videojuegos para la consola Nintendo DS.
- Uso de casi todo el *hardware* del Nintendo DS, exceptuando el micrófono, memoria auxiliar en puerto GBA y las capacidades gráficas en la pantalla secundaria.
- Comunicación inalámbrica entre consolas.
- Detección de colisiones.
- Reproducción de música de fondo.
- Efectos de sonido para las acciones.
- Uso de la pantalla táctil en el control de la cámara.
- Demostrar un ejemplo del generador de terrenos.
- Uso de personajes diseñados en Blender.
- Demostración del desarrollo de un proyecto usando únicamente *software* y arte libre.

El presente proyecto no habría sido posible sin el *software* libre, ya que los kits de desarrollo para Nintendo DS privativos, además de ser de venta y distribución restringida, son demasiado costosos.

Nuestro proyecto fue desarrollado en el **Laboratorio de Investigación y Desarrollo de Software Libre de la Facultad de Ingeniería** (LIDSOL) y está basado y sustentado en las bibliotecas y herramientas libres **Libnds**, **dswifi**, **tremor**, **DevkitARM**, **DevkitPRO**, **Desmume, Blender** y **SubVersion**, compilando en **Debian** y **Ubuntu GNU/Linux**. De la misma manera, con este proyecto, colaboraremos a la comunidad de desarrollo libre de videojuegos liberándolo bajo una licencia libre.

Durante el desarrollo de los objetivos de la tesis, la investigación y recopilación de información fue constante, así como la prueba y error de ideas propias. Dicho proceso hace apreciar las dificultades de programar en una plataforma con un *hardware* limitado y por consiguiente concientiza sobre la importancia de desarrollar código eficiente.

Por otro lado, se aprecia a los desarrollos libres en los que fue basado el **Framework** por su claridad en el código y la importancia de una buena documentación. Por eso se decidió realizar una exhaustiva documentación de las funciones, clases y herramientas que conforman el **Framework** en el apéndice, esperando que esto facilite la labor de futuros desarrolladores.

Con la liberación del **Framework** al público en general, se espera que despierte entusiasmo por el desarrollo de videojuegos y aplicaciones gráficas afines para la consola, lo cuál se desea que a su vez favorezca la colaboración en el crecimiento del **Framework**. También se espera que pueda ser visto como un incentivo para que los alumnos de las carreras relacionadas tengan más interés en las materias teóricas al ver una aplicación específica de muchos de los temas vistos.

Se planea continuar trabajando en la actualización y expansión del **Framework** a otras consolas, buscar su mejora y fomentar el desarrollo libre por parte de quienes lo usan. Pretendiendo con esto hacer del **Framework** un gran proyecto de renombre y volverlo una opción para el desarrollo de juegos comerciales.

Apéndices

Índice de ilustraciones

API

La API, al igual que la documentación del *Framework* está disponible en la dirección http://motorj.mexinetica.com/ en su versión más actualizada.

Especificaciones Técnicas de la consola Nintendo DS

Figura 1: Nintendo DS

Figura 2: Nintendo DS Lite

Nintendo DS, del inglés *Dual Screen*, posee nuevas características a las otras consolas portátiles existentes. Las dos pantallas crean un nuevo concepto de juego, al tener dos vistas de la misma escena, o mejor y más rápido acceso a los menús o una nuevo tipo de interacción con los juegos, un micrófono que permite interacciones en el juego tales como inflar un globo o hablar con algún jugador al otro lado del mundo, además de una conexión inalámbrica con el protocolo 802.11 o el protocolo de Nintendo, que responde de 10 a 30 metros según algunas condiciones físicas.

La pantalla táctil de abajo responde a una pluma especial que viene con la consola o a la presión del dedo de los jugadores, lo cual permite a los jugadores interactuar de una nueva manera. Además posee los tradicionales botones, cuatro de dirección, A; B; X, Y, L y R, *start* y *select*.

El hardware esta diseñado para renderizar en 3D solo una pantalla a la vez y 2D en la otra, o simultáneamente 2D en ambas pantallas. Debido la limitada memoria para las texturas en pantalla, solo puede desplegar como máximo una textura de 1024x1024 pixeles.

La siguiente tabla muestra más detalladamente las especificaciones técnicas:

Generales		
	Peso 275 gr.	
	Dimensiones de 148.7 x 84.7 x 28.9 mm.	
Procesadores		
	ARM946E-S 32bit RISC CPU, 66MHz (NDS9 video)	
	ARM7TDMI 32bit RISC CPU, 33MHz (NDS7 sonido)	
Memoria Interna		
	RAM principal (8192KB en la versión *debug*)	4096KB
	WRAM (64K mapeado para NDS7, plus 32K mapeable para NDS7 o NDS9)	96KB
	TCM/Cache (TCM: 16K Data, 32K Code) (Cache: 4K Data, 8K Code)	60KB
	VRAM (direccionable como BG/OBJ/2D/3D/Palette/Texture/WRAM memory)	656KB
	OAM/PAL (2K OBJ, 2K *Standard Palette* RAM)	4KB
	Memoria interna 3D (104K Polígonos RAM, 144K Vértices RAM)	248KB
	Wifi RAM	8KB
	Firmware FLASH	256KB
	BIOS ROM (4K NDS9, 16K NDS7, 16K GBA)	36KB
Video		
	Pantallas LCD (de 256x192 pixeles, 3 pulgadas, 18bit de color y luz trasera, 62 x 46 mm y 77 mm de diagonal)	2
	2D *video engines*	2
	3D *video engine* (puede ser asignada para la pantalla superior o inferior)	1
	Tamaño del pixel	0.24 mm
	Distancia entre las pantallas	21 mm
Gráficas		
	120,000 triángulos por segundo con todos los efectos de hardware a 60 *frames* por segundo.	
	262,144 colores	

	4 millones de vértices por segundo	
	Sistema de 656 KB de memoria de video,	
Sonido		
	16 canales de sonido (16x PCM8/PCM16/IMA-ADPCM, 6x PSG-Wave, 2x PSG-Noise)	
	2 unidades de captura de sonido (para efectos de eco, etc)	
	Salida: 2 Bocinas estéreo y un *socket* de audífonos	
	Entrada: 1 micrófono y un *socket* para micrófono	
Controles		
	Botones de dirección	4
	Botones	8
	Touchscreen	Pantalla inferior
Puertos de comunicación		
	Wifi IEEE802.11b	
Especiales		
	Dispositivo de administración de la Batería	
Memoria externa		
	1 puerto NDS (para juegos NDS) (bus de datos cifrado de 8bit, y un bus de 1 bit serial)	
	Puerto GBA (para expansiones del NDS, o para juegos GBA)	
Puede iniciar desde		
	NDS Cartridge (NDS mode)	
	Firmware FLASH (NDS mode) (eg. by patching firmware via ds-xboo cable)	
	Wifi (NDS mode)	
Batería		
	Batería de Litio recargable 3.7V 1000mAh (DS-Lite)	
	Tiempo de carga completa	4 hrs
	Tiempo de duración	10 hrs aprox.
Idiomas		
	Inglés Francés, Alemán, Italiano, Japonés y Español	

Instalación del Framework

Para realizar la instalación del *Framework* es necesario contar con una computadora que tenga instalado alguna distribución del sistema operativo GNU/Linux y conexión a Internet.

En el presente manual se proporcionarán las instrucciones para la distribución Debian GNU/Linux utilizando el entorno de escritorio GNOME; dichas instrucciones pueden ser empleadas sin modificación para las distribuciones basadas en Debian tales como Ubuntu con el mismo entorno de escritorio, y reemplazando *Synaptic* por el gestor de paquetes propio de cada distribución para cualquier otra, así como en otros entornos de escritorio localizando las mismas aplicaciones o equivalentes. Los comandos de texto escritos en la terminal no requieren cambios aún en estos casos.

1. Se inicia el gestor de paquetes *Synaptic*, ubicado en el menú Sistema, opción Administración, como muestra la figura :

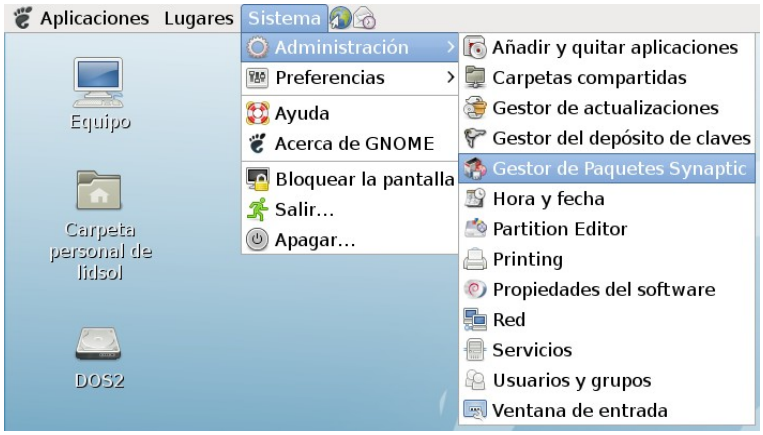

Figura 3: Ubicación de Synaptic

Nota: Dicha acción requerirá ingresar como superusuario o adquirir privilegios de superusuario, necesarios para instalar *software*.

2. Se presiona el botón “buscar”, ubicado en la esquina superior derecha de *Synaptic* (figura 4).

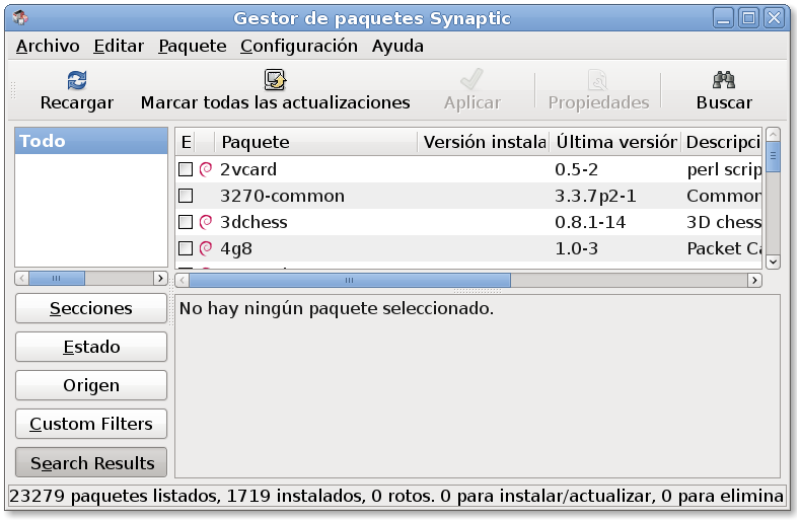

Figura 4: Interfaz principal de Synaptic

Como parámetro se le suministra *Subversion*, que es el nombre del sistema de control de versiones usado por el proyecto (figura 5).

Figura 5: Diálogo de búsqueda

3. Una vez que el gestor de paquetes localice *Subversion*, se pulsa el botón derecho del ratón sobre dicha opción y se selecciona el comando “marcar para instalar”, como se muestra en la figura 6:

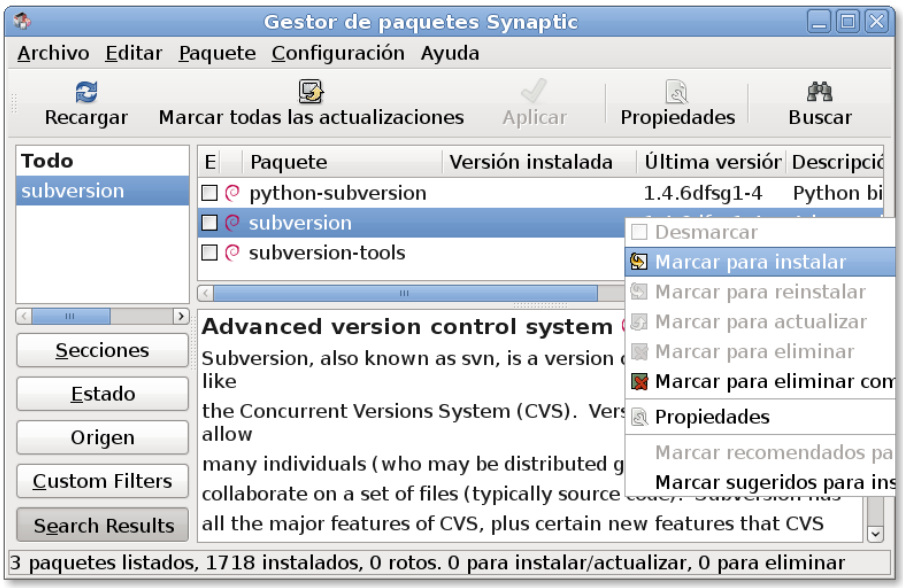

Figura 6: Comando "Marcar para instalar"

4. Se repiten los pasos 2 y 3, pero esta vez proporcionando y marcando para instalar como nombre de paquete *Wget*, que es un programa de descarga de archivos empleado por un *script* de configuración posterior.

5. Se repiten los pasos 2 y 3 para el paquete *Desmume*, el cual es un emulador que resulta útil para pruebas rápidas, aunque no siempre presente resultados correctos.

6. Se pulsa el botón Aplicar, localizado en la parte superior central del gestor de paquetes. Synaptic mostrará las dependencias de la instalación de ambos paquetes y pedirá confirmación. Una vez confirmado, el gestor de paquetes procederá a descargar los datos e instalarlos. Una vez completado, mostrará el siguiente diálogo (figura 7):

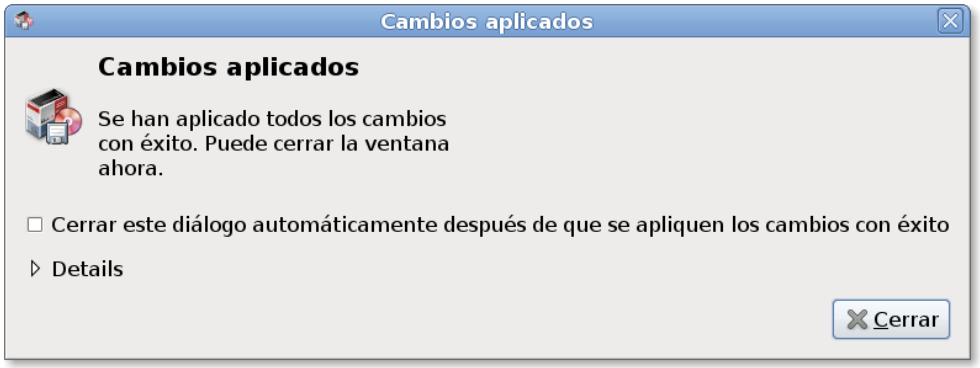

Figura 7: Confirmación de la instalación

La instalación de los paquetes dependientes está completada. Se cierra el gestor de paquetes *Synaptic* y se procede entonces a descargar los datos del servidor de código del laboratorio.

7. Se abre una terminal de texto. Dicho programa se encuentra en el menú Aplicaciones, opción Accesorios (ver figura 8):

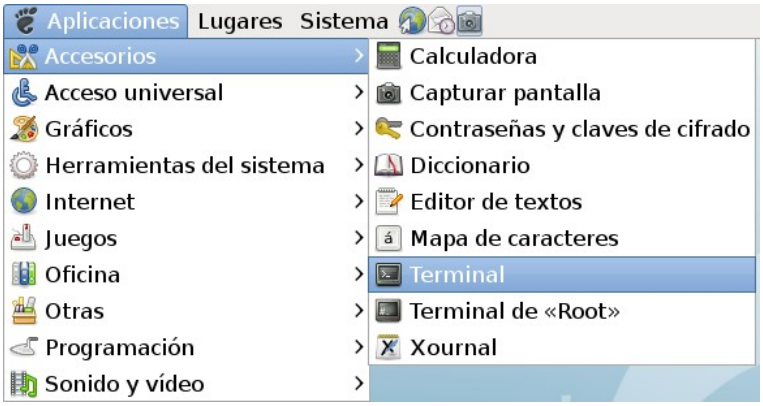

Figura 8: Ubicación de Terminal

8. Se escribe el comando "`mkdir ~/framework`" y se pulsa Enter; dicho comando creará un directorio llamado "framework" en el directorio principal del usuario actual.

9. Se escribe el comando "`cd ~/framework`" y se pulsa Enter; dicho comando cambiará el directorio actual al que fue creado en el comando anterior (figura 9).

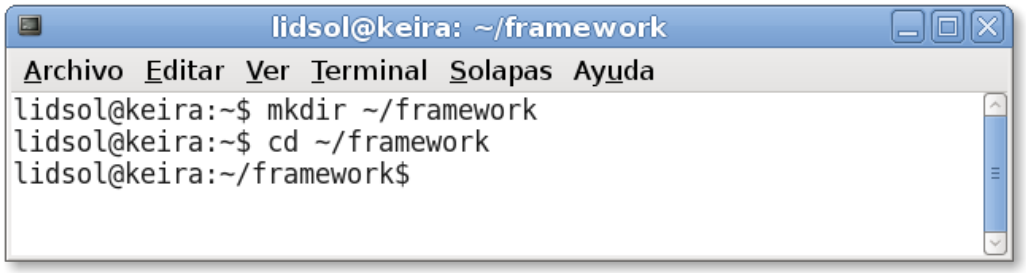

Figura 9: Creación del directorio que alojará al Framework

10. Se escribe el comando "`svn checkout http://svn.lidsol.net/motorj`" y se presiona Enter para obtener el código fuente del proyecto mediante el sistema Subversion como se muestra en la figura 10:

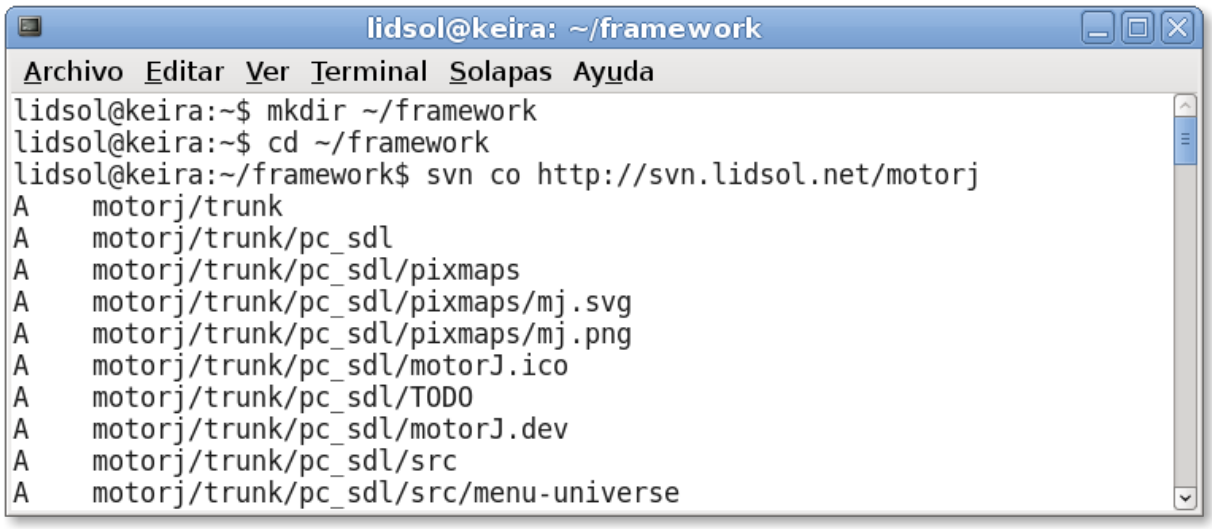

Figura 10: Comando "svn checkout" para obtener el código fuente

11. Se escribe entonces el comando:
    "`~/framework/motorj/trunk/ds/devkitpro_environment_install_gnulinux.sh`" y se presiona Enter para ejecutar el script de instalación del entorno de compilación. Dicho script descargará y ubicará los datos automáticamente.

    Al completar la configuración, se mostrará una pantalla como la de la figura 3:

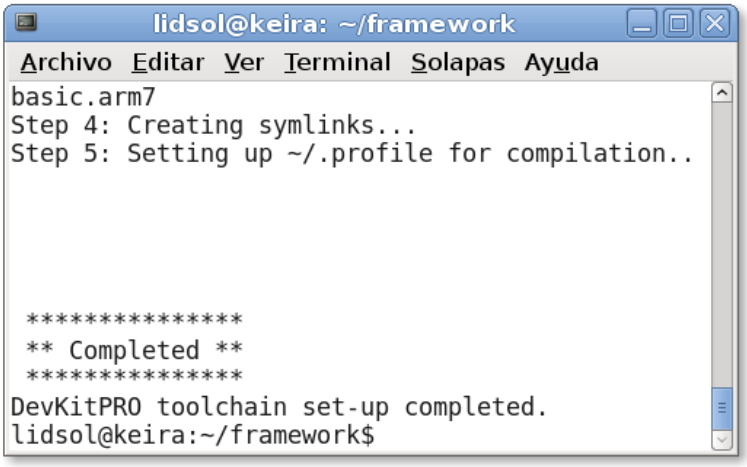

Figura 11: Fin del script de configuración e instalación del entorno

12. Una vez completado el script anterior, es necesario únicamente cerrar la sesión gráfica y volver a iniciarla; a partir de ese momento, se está listo para compilar proyectos realizados con el Framework.

Compilación del proyecto de prueba del Framework

El proyecto de prueba del Framework es una animación del humanoide "lanjobot", visible desde distintos ángulos al tener una cámara interactiva.

1. Se abre una terminal de texto. Dicho programa se encuentra en el menú Aplicaciones, opción Accesorios.

2. Dentro de la terminal, se escribe el comando "`cd ~/framework/motorj/trunk/ds`" y se presiona *ENTER*. Esto cambiará el directorio actual al que tiene los archivos necesarios para crear un nuevo proyecto para Nintendo DS.

3. Se escribe el comando "`./create_new_ds_project.sh`" y se presiona *ENTER*, lo cual ejecutará el *script* de generación de nuevo proyecto.

4. Al ser ejecutado, dicho *script* solicita la ruta completa al directorio que contendrá el nuevo proyecto. Se escribe "`/home/`*`usuario`*`/ejemplo`", cuidando de reemplazar *usuario* por el nombre de usuario con el que se haya iniciado sesión, como se muestra en la figura 12:

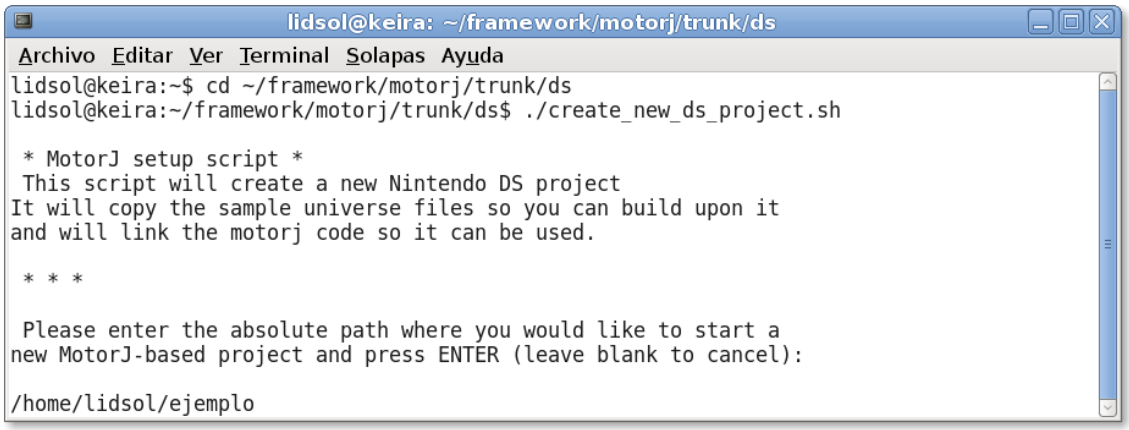

Figura 12: Script de generación de nuevo proyecto

   Al presionar *ENTER*, se habrá creado la estructura de archivos necesaria para el nuevo proyecto.

5. Se escribe "`cd ~/ejemplo`" y se presiona *ENTER* para cambiar de ubicación al directorio del nuevo proyecto.

6. Finalmente, al escribir "`make test`" y presionar *ENTER*, se realizará la compilación del código fuente y al ser exitosa, se lanzará el emulador con el proyecto de prueba (figuras 13 y 14).

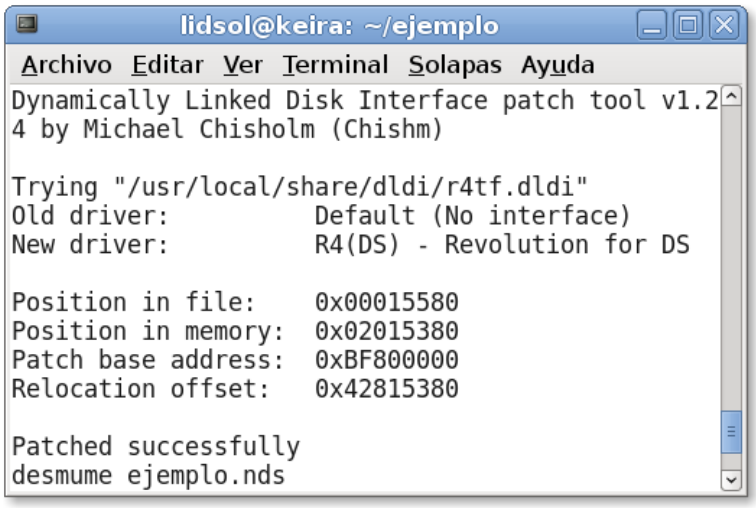

Figura 13: Resultado de la compilación

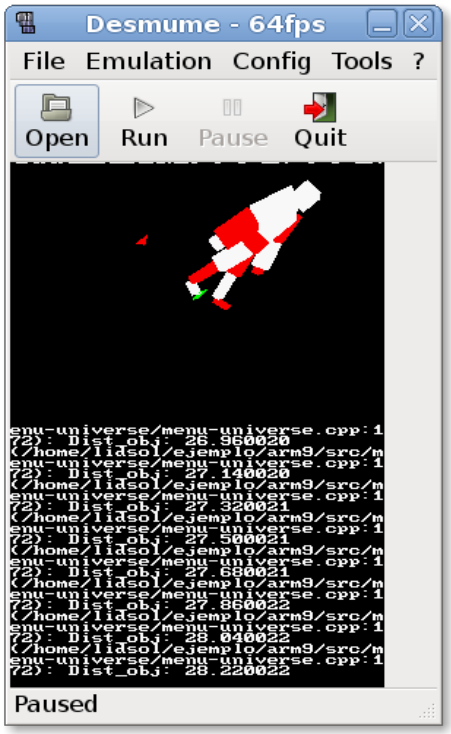

Figura 14: Emulador Desmume ejecutando el programa de prueba

7. Para mayor información sobre la estructura de directorios, es conveniente referirse a la dirección http://motorj.mexinetica.com/

Exportador de Blender para *Display List*

1. Instalación

Una vez instalado el *Framework*, el exportador se encuentra como parte del MotorJ. Para realizar la instalación en Blender es necesario localizar el archivo DisplayListExp.py ubicado en la ruta `~/framework/motorj/trunk/ds/blender-scripts/` y copiarlo al path de Blender para el usuario la ruta de blender. En sistemas basados en UNIX se encuentra en `~/.blender/scripts/`.

a) El comando para realizar la copia es:

```
$cp ~/framework/motorj/trunk/ds/blender-scripts/DisplayListExp.py ~/.blender/scripts/.
```

b) Ahora se procede a abrir Blender, la pantalla inicial es similar a la siguiente:

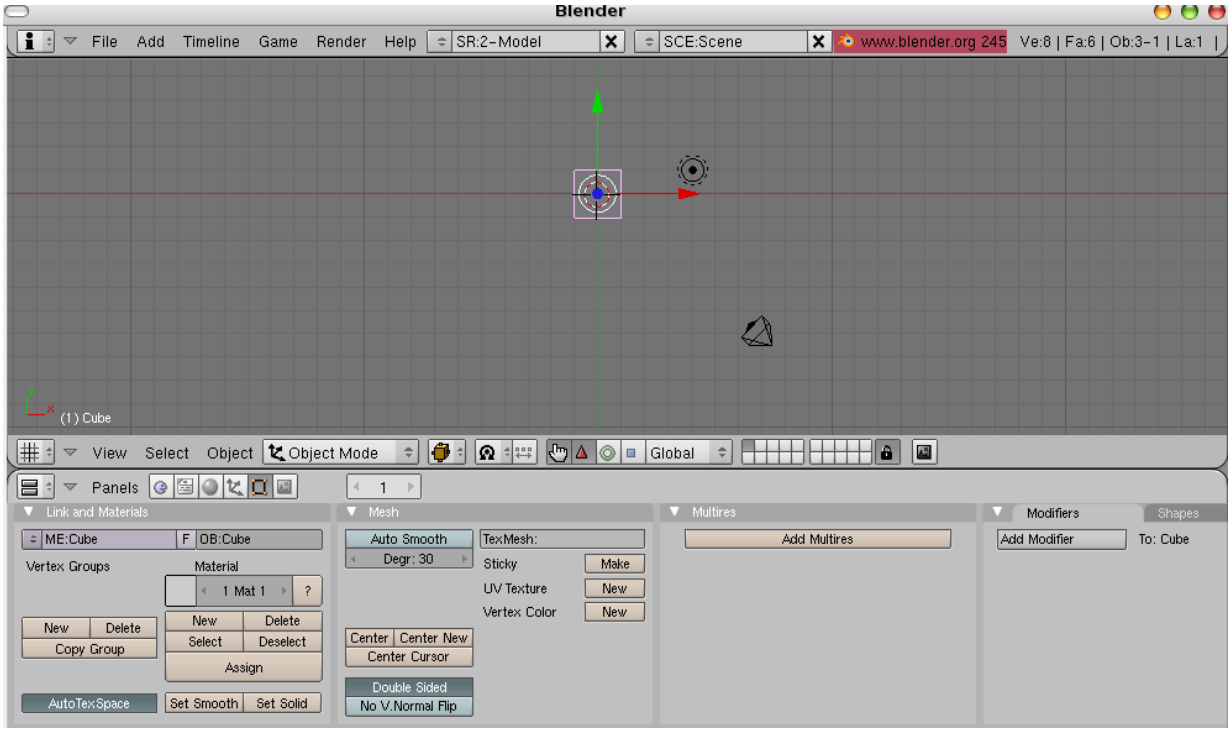

Figura 15: Pantalla inicial de Blender

1. En el menú “Window Type” se selecciona “Scripts Window”

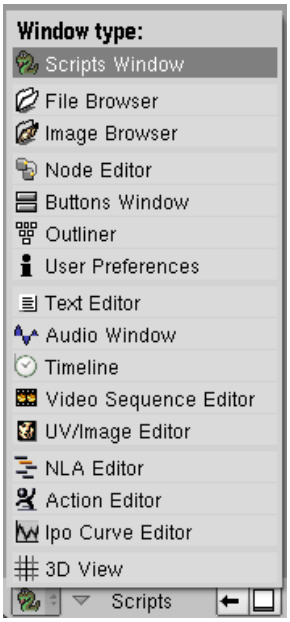

Figura 16: Menú "Window Type"

2. Se selecciona la opción “Update Menus” del menú “Scripts”

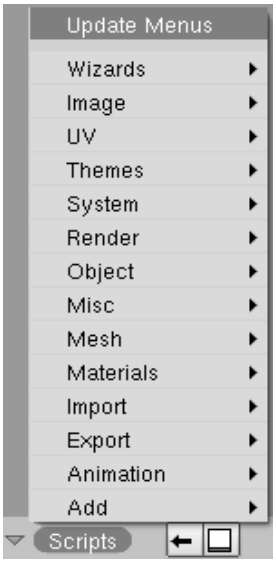

Figura 17: Menú "Scripts"

Concluido este proceso ya está incluido el exportador dentro del menú “File”; dentro de la opción “Export” se encontrará el script “Display List NDS”.

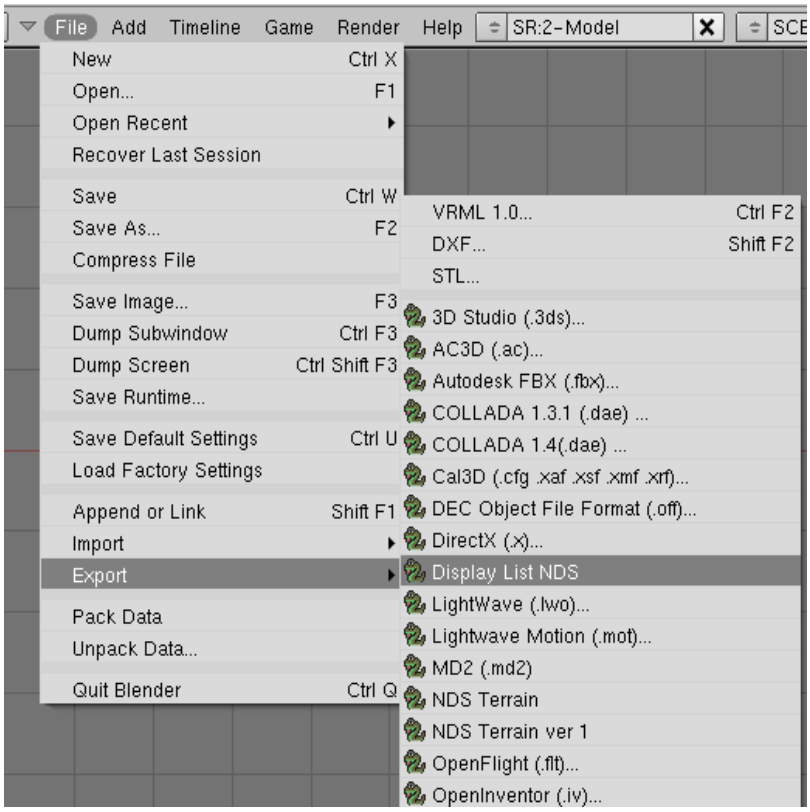

Figura 18: Exportador incorporado al submenú "Export" del menú "File"

2. Ejemplo de creación y exportación

Esta sección tratará sobre como realizar un ejemplo completo de creación de una cabeza, así como la exportación para obtener un *Display List*.

1) Dentro de Blender[40] se seleccionan los objetos no geométricos de la escena presionando el botón **Shift** y el botón derecho del *mouse* para borrarlos presionando la tecla **Supr**.

2) Una vez teniendo la escena limpia, se puede observar el cursor en el origen de los ejes coordenados. Se procede a verificar la vista superior de la escena, para ello se selecciona el menú **View**->**Top**.

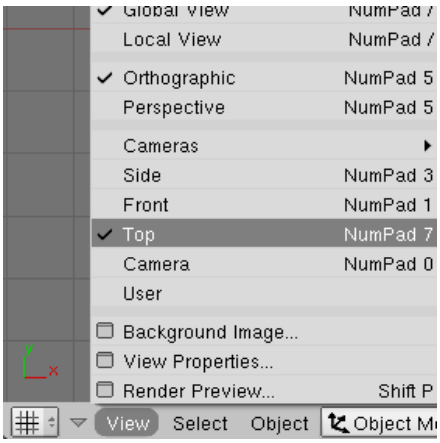

Figura 19: Opción "Top" del menú "View"; Vista superior

3) Se procede a insertar un objeto en la escena presionando el **espacio** (barra espaciadora) para obtener el menú de edición de objetos y se elige la opción **Add**->**Mesh**->**UVsphere**. Se creará con las siguientes especificaciones:

Segments	16
Rings	12
Radius	0.002

4) Esto crea una esfera, pero es demasiado pequeña para verse en escena, de modo que se deberá acercar la vista con el *scroll* o presionando la tecla **Ctrl** + el botón central del *mouse* hasta obtener una vista cómoda para trabajar con la figura. En caso de que la esfera se desvíe del centro de la vista, es posible desplazarla presionando la tecla **Shift** + el botón central del *mouse*:

40 Wartmann, Carsten. “TheBlender Book”, pp. 40-52, 63-79.

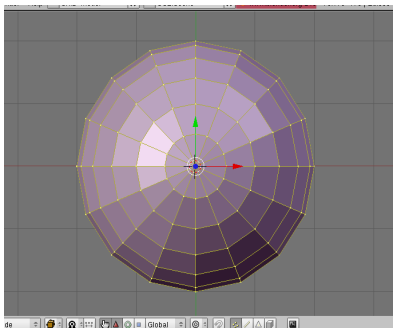

Figura 20: UVsphere seleccionada en vista superior

5) Se creará la cabeza a partir de la esfera que fue incorporada a la escena. Se cambia la vista a frontal **View**->**Front**, y se centra la esfera con **Shift** + el botón central del mouse.

6) Se verifica que el modo de edición de vértices se encuentre activado, para ello se presiona el **Tab** repetidas veces y se observa como cambia el objeto de su color base (usualmente gris) a uno morado con los vértices seleccionados en amarillo, para lo subsecuente es necesario permanecer en modo de edición de los vértices.

7) Presionando la tecla **A** perdemos la selección activa y deja a la esfera en color azul con los vértices en rosa pálido. Si volvemos a presionarla se observa que se seleccionan todos los vértices del objeto.

8) La tecla **B** tiene la función de establecer el modo de selección, si se presiona una vez se marca un cursor con dos líneas sobre los ejes, este sirve para hacer selecciones rectangulares. En caso de volver a presionar la tecla **B**, cambiará a un circulo que seleccionará todos los vértices que se encuentren en esa área marcada al presionar el botón izquierdo del *mouse*. Para salir del modo de selección presione la tecla **Esc**. Usando alguna de los dos modos de selección hagamos una selección como sigue:

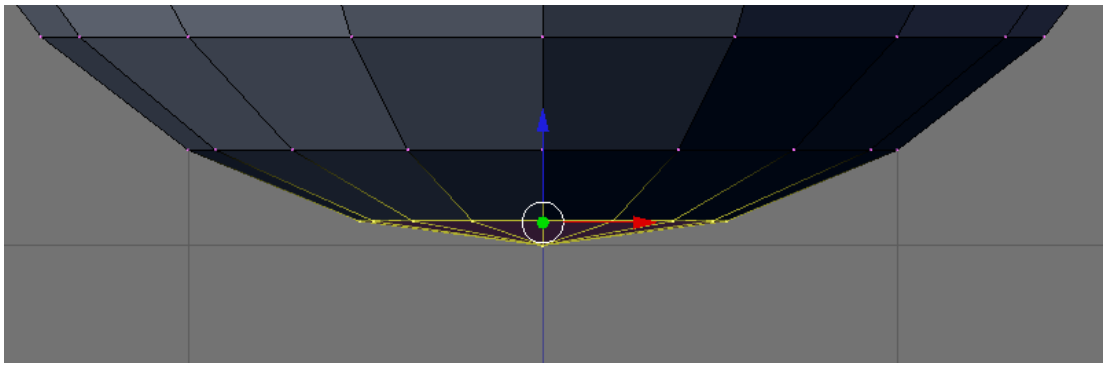

Figura 21: Selección para el cuello, vista frontal

9) Lo siguiente es extruir las caras. La extrusión implicará crear una copia del borde de los vértices del área seleccionada y mover la selección hacia abajo, con el fin de crear el cuello. Para ello se presiona la tecla **E**, se selecciona por **Region** y se escribe 0.005, o se presiona la barra espaciadora **Edit->Extrude**; **Region** y se escribe 0.005, dando como resultado:

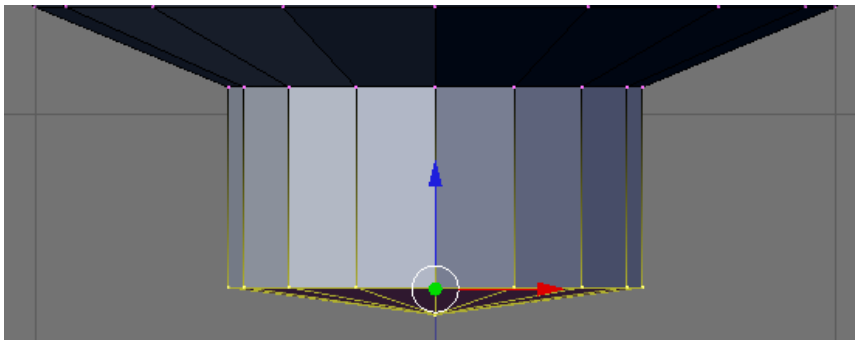

Figura 22: Extrusión del cuello

10) Se deselecciona presionando la tecla **A** y se cambia nuevamente a la vista View->Side, y se procede con una selección de vértices. Es posible observar que al seleccionar los vértices que conforman una cara se ilumina la cara de color rosa. Es necesario tomar en cuenta que cuando se hace una selección todos los vértices serán en esa área serán seleccionados, eso implica que los vértices de la cara posterior también se seleccionarán, tener cuidado con esto en futuros trabajos.

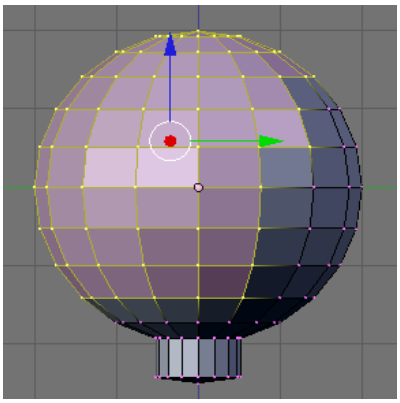

Figura 23: Selección de caras para definir el cabello

11) Para formar el cabello se extruye el área seleccionada con la secuencia **Edit->Extrude**; **Region** y se presiona **Enter**, posteriormente se presiona el **Espacio**, **Transform->Scale** y se observa que se puede definir el tamaño de lo que será el cabello. El tamaño queda a su consideración.

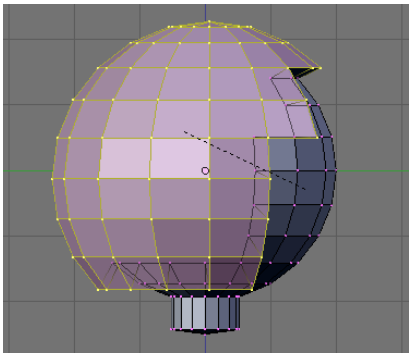

Figura 24: Extrusión del cabello

12) Lo siguiente es diferenciar al cabello de la piel, para esto se asignarán materiales y colores. Se comienza con una selección un poco más detallada uniendo el contorno del lo que será el cabello. Recomendamos el uso del modo de selección circular, por adaptación de su tamaño. Para deseleccionar algún vértice en particular se presiona la tecla **Shift** + el botón derecho del mouse. Y para mover la vista a perspectiva se presiona el botón central del mouse hacia la dirección que deseada. La selección queda como sigue:

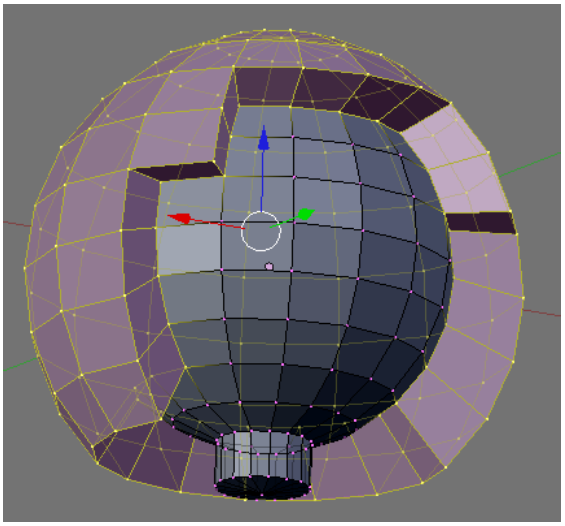

Figura 25: Vista en perspectiva de la cabeza

13) La asignación de los materiales se realiza en el panel inferior, en la sección **Link and Material.** En **Material** se presiona el botón **New** y luego el boton **Assign**. Del lado izquierdo de "1 Mat 1", se puede seleccionar un color sólido para dicho material. Se pretende elegir un color parecido al del cabello.

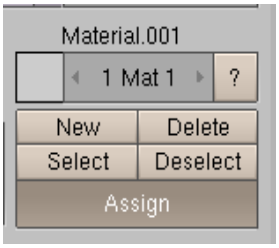

Figura 26: Sección "Link and Material"; asignación de materiales

14) Se prosigue seleccionando los vértices pertenecientes a la piel. Se le asigna un nuevo material y color de color de piel.

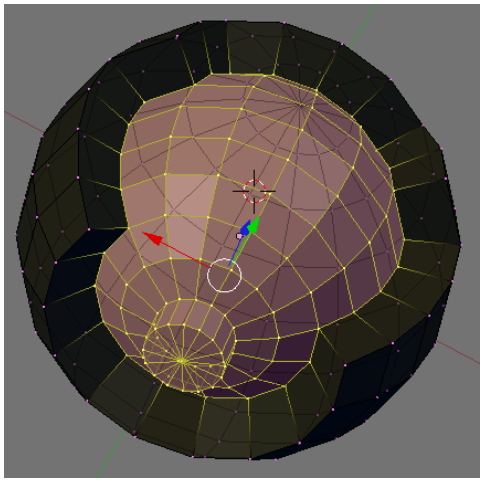

Figura 27: Cabeza con materiales

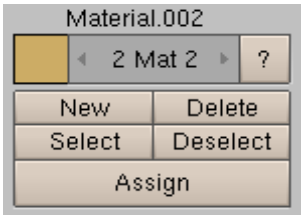

Figura 28: Asignación de los materiales

15) Al probar los otros menús de **Material**, se puede indicar que material usar y seleccionarlo con **Select** o deseleccionarlo con **Deselect**.

16) Ahora resta indicar el nombre a la geometría, en otras palabras asignarle nombre al objeto. En la misma zona de **Link and Material** se ingresa el nombre al objeto, sustituyendo los nombres genéricos como *Sphere*. En este caso se le asigna "Face":

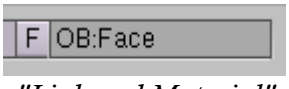

Figura 29: Sección "Link and Material"; nombre del objeto

17) El movimiento de vértices para proporcionar mejores rasgos a los modelos queda como trabajo para el usurario. Cualquier transformación será por medio de los vértices de cada modelo. La siguiente tabla muestra como hacer las transformaciones básicas de Traslación, Rotación y Escalamiento:

Transformación	Localización	Descripción
Traslación		
	Espacio->Transform->Grab/Move	Realiza una translación de todos los elementos seleccionados en cualquier dirección indicada por el mouse.
	Espacio->Transform->Grab/Move on Axis->X,Y,Z Local/Global	Se limita el desplazamiento a solo un eje coordenado, realizando el desplazamiento local o global.
Rotación		
	Espacio->Transform->Rotate	Se realiza la rotación con respecto a la posición del cursor.
	Espacio->Transform->Rotate on Axis->X,Y,Z Local/Global	La rotación sobre algún eje coordenado.
Escalamiento		
	Espacio->Transform->Scale	Escalamiento de los vértices siguiendo a los movimientos del mouse.
	Espacio->Transform->Scale on Axis->X,Y,Z Local/Global	Escalamiento limitado a sólo uno de los ejes coordenado.

18) Para poder exportar es indispensable guardar el modelo antes. Menú **File->Save as**; y se le asigna un nombre.

19) Una vez guardado el modelo el proceso de exportación es: **Menú File->Export->Display List NDS,** resta escribir la extensión '.h' al archivo para ser incluido y usado en el **Framework**.

Glosario

Access Point. Un punto de acceso inalámbrico (del inglés WAP o AP: *Wireless Access Point*) en redes de computadoras es un dispositivo que interconecta dispositivos de comunicación inalámbrica para formar una red inalámbrica.

Alpha. Canal del RGB para definir niveles de transparencia.

API. Interfaz de Programación de Aplicaciones (del inglés *Application Programming Interface*) es un conjunto de funciones, procedimientos o métodos que ofrece cierta biblioteca para ser utilizado por otro *software* como una capa de abstracción.

Blender. Programa multiplataforma, dedicado especialmente al modelado y creación de gráficos tridimensionales.

Broadcast. Una difusión, modo de transmitir información de un nodo emisor a una multitud de nodos receptores de manera simultánea.

Codec. Proveniente de Codificador-Decodificador, describe una especificación desarrollada en *software*, *hardware* o una combinación de ambos, con el objetivo de transformar un archivo, un flujo de datos o una señal.

Consola. Aparato electrónico ce esparcimiento, para reproducir videojuegos.

Frame. Fotograma o cuadro, una imagen particular dentro de una sucesión de imágenes que componen una animación.

GNU. Acrónimo recursivo que significa GNU No es Unix (del inglés *GNU is Not Unix*), es un proyecto iniciado por Richard Stallman con el objetivo de crear un sistema operativo completamente libre.

GUI. Interfaz gráfica de usuario (del inglés *Graphical User Interface, 'GUI'*), es un tipo de interfaz de usuario compuesta por imágenes y objetos gráficos para representar la información y acciones disponibles en la interfaz.

Homebrew. Desarrollo casero no oficial.

LAN. Red de Area Local (del inglés *Local Area Network*).

LCD. Pantalla de cristal líquido (del inglés *Liquid Crystal Display*) es una pantalla delgada y plana formada por un número de píxeles delante de una fuente de luz o reflectora.

lenguaje de programación. Lenguaje que puede ser interpretado o compilado por una máquina y que le permite a ésta seguir órdenes específicas para realizar acciones útiles.

Mainframe. Es una computadora central usada principalmente por una gran compañía para el procesamiento de una gran cantidad de datos.

Multicast. Multidifusión, es el envío de la información en una red a múltiples destinos simultáneamente.

OpenGL. Biblioteca libre de gráficos (del inglés Open Graphics Library), es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D.

Paradigma de programación. Un tipo de filosofía para el desarrollo de *software.*

Prueba de concepto. Una validación de la herramienta desarrollada.

ROM. Memoria de solo lectura (del inglés *Read-Only Memory*), un dispositivo de almacenamiento que no permite la modificación de sus datos.

Script. Es un conjunto de instrucciones orientadas a la automatización de tareas creando pequeñas utilidades.

SDK. Kit de Desarrollo de Software (del inglés *Software Development Kit*), se le considera a un conjunto de herramientas para que un programador cree aplicaciones con una orientación específica, por ejemplo *Frameworks*, *hardware*, sistemas operativos, etc.

Sistema de control de versiones. Sistema que permite preservar distintas versiones de un mismo desarrollo, así como avanzar y retroceder en dichas versiones, lo que facilita la detección y corrección de errores introducidos, así como la colaboración de varias personas de manera simultánea.

Socket. La comunicación o flujo de datos entre dos dispositivos distintos, ya sean computadoras u dispositivos de comunicación.

Software libre. *Software* cuya licencia permite su uso, redistribución, visualización y modificación de su código fuente, además de redistribución de dichas modificaciones, todo lo anterior sin costo y de manera legal.

Software privativo. *Software* cuya licencia restringe la redistribución, visualización, modificación o redistribución de modificaciones sin costo.

Subversion. Sistema que permite preservar distintas versiones de un mismo desarrollo, así como avanzar y retroceder en dichas versiones, lo que facilita la detección y corrección de errores introducidos, así como la colaboración de varias personas de manera simultánea.

Texel. *Pixels* pertenecientes a una textura.

Vértice. El elemento fundamental de la que está formado un gráfico. El punto común de los dos lados de un ángulo para una geometría.

Wii. Consola de videojuegos de Nintendo de séptima generación.

Wiki. Sistema web cuyo fin es permitir a cualquier persona modificar el contenido de las páginas web que lo conforman, de manera que sea fácil colaborar en la generación de contenido útil.

Bibliografía

[1] Rogers Gregory F.
Framework-Based Software Development in C++
Prentice Hall
United States of America
1997

[2] Akenine-Möller Tomas
Haines Eric
Real-Time Rendering, Second Edition
A K Peters
India
2002

[3] Gay Warren W.
Linux Socket Programming by example
Que
United States of America
2000

[4] Castañeda de Isla Puga Erik
Geometría Analítica en el Espacio
Facultad de Ingeniería
México
2002

[5] Bruegge Bernd
Dutoit Allen
Ingeniería de Software Orientado a Objetos
Prentice Hall
México
2002

[6] Rada Roy
Reengineering Software
AMACOM
USA
1999

[7] Tramullas Jesús
Garrido Piedad
Software Libre para la Información Digital
Prentice Hall
México
2002

[8] Glaeser Georg
Stachel Hellmuth
Open Geometry
Springer
USA
1999

[9] Shreiner Dave
OpenGL, Reference Manual
Addison-Wesley
USA
2000

[10] Wartmann Carsten
The Blender Book
Linux Journal press
USA
2002

[11] Eberly David
3D Game engine design
Morgan Kaufmann
USA
2005

Direcciones de internet consultadas

[a] http://www.devkitpro.org/

[b] http://nocash.emubase.de/gbatek.htm

[c] http://www.desmume.com/

[d] http://chishm.drunkencoders.com/libfat/

[e] http://www.double.co.nz/nintendo_ds/

[f] http://osdl.sourceforge.net/main/documentation/misc/nintendo-DS/homebrew-guide/HomebrewForDS.html

[g] http://www.sonicspot.com/guide/wavefiles.html#wavefileheader

[h] http://www.csc.villanova.edu/~mdamian/Sockets/TcpSockets.htm

Pistas sonoras empleadas en la prueba de concepto

[1] Grace Valhalla
Empty
PEAK ~
Independiente, 2006, Francia
http://gracevalhalla.hautetfort.com/

[2] Vate
Casa
VOLK
Independiente, 2006, México
http://vate.com.mx/

[3] Subversive Activity
Artificial Creation
Psydom Revolutions
Psydom Records, 2007, Brasil
http://www.jamendo.com/en/artist/psydom/