UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

Desarrollo de la aplicación "calakmul virtual" usando el motor de juegos nebula.

T E S I S

QUE PARA OBTENER EL TITULO DE:

INGENIERO EN COMPUTACION

P R E S E N T A :

JOSE LARIOS DELGADO

DIRECTOR: DR. JESUS SAVAGE CARMONA

FACULTAD DE INGENIERIA

U N A M

2004

Dedicatoria

A mi familia:

A mi Mamá
A mi Papá
A mi Hermano
A mi Hermana

Sors immanis
et inanis,
rota tu volubilis,
status malus,
vana salus
semper dissolubilis,
obumbrata
et velata
michi quoque niteris;
nunc per ludum
dorsum nudum
fero tui sceleris.

Agradecimientos

Un sincero agradecimiento al Dr. Jesús Savage Carmona, mi asesor de tesis, por su apoyo y por la oportunidad que me dio para hacer este trabajo.

A mis padres porque sin ellos yo no sería lo que soy.

A mi hermano cuyo ejemplo siempre me motiva a seguir adelante.

A mi hermana por toda la ayuda que me ha proporcionado.

Agradezco a mis profesores por todas sus enseñanzas y conocimientos.

A todos mis amigos y compañeros por brindarme su amistad a lo largo de la carrera.

Índice General

Índice de Figuras

Índice de Tablas

Introducción

Historia de la graficación por computadora.

En la década de los 50's un equipo conformado por las empresas International Business Machines (IBM) y la General Motors (GM), resultó la creación de un programa llamado Design Aumented by Computers (DAC-1). El DAC-1 tenía como principal objetivo aceptar coordenadas y crear la representación. Una vez que se tenían todos los datos se podía ver el modelo creado desde diferentes direcciones y ángulos. Aunque el programa DAC-1 fue creado en 1959 se presentó hasta la conferencia de Detroit en 1964.

En 1962 Iván Sutherland de la universidad del Massachusetts Institute of Technology (MIT) escribió un programa que controlaba directamente un tubo de rayos catódicos (CTR por sus siglas en inglés) para mostrar líneas y luz, que daban como resultado una figura geométrica. El programa denominado "sketchpad" permitía guardar los dibujos que eran creados. Los dibujos creados con el programa de Sutherland eran vectores en lugar de bitmaps, los resultados que obtenía eran bastante precisos, ya que el usuario determinaba la figura geométrica que deseaba, y ciertas indicaciones si se trataba de un rectángulo por ejemplo, sólo definía dos esquinas, la computadora calculaba automáticamente el tamaño y posición de las líneas.

Sketchpad propició el nacimiento de la ciencia de gráficas controladas por computadora. Dos años más tarde, Sutherland colaboraría con el doctor David Evans para iniciar la exploración de mezclas entre arte y ciencia (computacional). Fue la universidad de Utah la primera que tuvo un laboratorio académico específico para desarrollar gráficas por ordenador. De la investigación realizada en la universidad de Utah, hoy toman sus bases los paquetes gráficos, de los de diseño hasta los de realidad virtual.

No pasó mucho tiempo sin que las compañías se empezaran a interesar por las gráficas en computadora, IBM, por ejemplo lanzó al mercado la IBM 2250, la primera computadora comercial con un sistema gráfico. La compañía Magnavox, a su vez obtuvo la licencia para distribuir un sistema de videojuegos creado por Ralph Baer, el producto fue denominado Odyssey. El Odyssey fue el primer producto orientado al consumidor con gráficas generadas por computador.

Dave Evans fue contratado por la universidad de Utah para crear el laboratorio de ciencias de la computación. Evans tomo como interés principal el desarrollar gráficas por computadora. Evans contrató a Sutherland, y es en Utah donde Sutherland perfecciona una interfaz de HMD (head mounted display), que había desarrollado algunos años antes. En ese periodo, Evans y Sutherland eran frecuentemente asesores de compañías, no obstante, constantemente se encontraban frustrados por la falta de tecnología, razón que más adelante los llevó a fundar su propia empresa.

En los años 70's un estudiante de la clase de Sutherland en la universidad de Utah, Edwin Catmull vislumbró a la animación por computadora como una evolución natural de la animación tradicional. Todavía en Utah creó una animación, se trataba de su mano abriéndose y cerrándose. Uno de sus objetivos se convirtió en realizar una película completamente generada por ordenador. De la universidad de Utah surgió un gran avance tecnológico en el campo, John Warnock fue uno de los pioneros digitales, y fundó una de las empresas más importantes que cambió el curso de la historia en cuanto a diseño digital se refiere, fundó Adobe. Otro egresado de la universidad de Utah no es menos notorio, Jim Clark, fundador de Silicon Graphics Inc. (SGI)

1970 también marcó una revolución en el mercado televisivo. Cadenas como la CBS empezaron a usar productos desarrollados para animar en la computadora. La empresa Computer Image Corporation (CIC) desarrolló combinaciones de Hardware y Software para acelerar procesos de animación tradicional, por medios digitales. CIC ofrecía ANIMAC, SCANIMATE y CAESAR, con estos programas se podían escanear los dibujos, crear trayectorias, aplicar principios de animación tradicional tales como estiramiento y encogimiento.

En el campo de la animación 3D, se creo un nuevo tipo de representación digital, el algoritmo de Henri Gouraud. Este permite que los contornos de los polígonos no se vean tan lineales, ya que esto destruía la sensación de una superficie suave. El algoritmo crea la interpolación de color entre polígonos y de esta forma logra una mejor representación de superficies curvas. La ventaja sobre el método tradicional (la representación plana) es que

la superficie en efecto parece perder dureza en la representación, con sólo una pequeña penalización en el tiempo que toma hacer la representación.

En 1971 surge el microprocesador, utilizando tecnología de circuitos integrados, los componentes electrónicos fueron miniaturizados. La compañía Atari fue creada y en 1972 crea el primer videojuego de "máquina" (arcade), Pong. Evans y Sutherland (E&S) se encontraban ya fabricando hardware propio para evitar algunas de las limitantes tecnológicas que algunos años antes habían experimentado. Uno de los sistemas más impresionantes credo precisamente por E&S era "Picture System", incluía una tableta gráfica y un buffer en color. Triple I, en 1974 desarrolló equipo para poder filmar las imágenes realizadas en computadora. Otro de sus inventos fue la creación de aceleradores gráficos. Los desarrollos de Triple I fueron un gran avance que permitía que las gráficas sintéticas pudieran ser utilizadas en cine.

Ed Catmull realizó su tesis de doctorado sobre una nueva manera de representar las superficies. Esta nueva técnica llamada z-buffer ayuda en el proceso de esconder las partes de las superficies que no serán vistas por el usuario en la representación final. Además del z-buffer, Catmull incluyó un nuevo concepto, el de mapeo de texturas. La historia cuenta que en una discusión con otro de sus compañeros, a Catmull se le ocurrió que si a un objeto en la vida real se le podían aplicar imágenes para representar a otra cosa, en un mundo virtual no había razón para no hacerlo.

El matemático francés Dr. Benoit Mandelbrot publicó un ensayo que permitió añadir realismo a las escenas generadas por computadora. El documento "A Theory of Fractal Sets", explica que una línea es un objeto unidimensional, el plano es un espacio bidimensional; no obstante, si la línea describe una curva de manera que cubra la superficie del plano deja de ser unidimensional, aunque tampoco es bidimensional. El Dr. Mandelbrot se refirió a este espacio como una dimensión fraccionaria. Las aplicaciones principales que se le dieron a las teorías de Mandelbrot fueron las de creación de terrenos aleatorios, así como la creación de texturas en las cuales existen subdivisiones dentro de un mismo patrón.

Después de su graduación, Catmull fue contratado por la empresa Applicon, donde no duró mucho tiempo, ya que recibió una oferta de trabajo para fundar el laboratorio de animación por computadora del Instituto Tecnológico de Nueva York (NYIT). Algunos de los trabajadores de la Universidad de Utah también fueron invitados y aceptaron el trabajo en el NYIT. Los primeros programas de animación desarrollados dentro del NYIT fueron para apoyar la animación tradicional. La primera aplicación que Catmull desarrollo fue "tween", que permitía realizar la interpolación entre cuadros. También se desarrollo un sistema de escaneo y pintura que posteriormente se convirtió en el sistema de producción de Disney, el CAPS (Computer Animation Production System).

El NYIT creó un departamento dedicado a la investigación de gráficas 3D, y por dos años su principal proyecto fue el de crear una película, "the works", nunca fue concluida, de hecho, pruebas preliminares fueron bastante desalentadoras. Ante el fracaso del corto "Tubby the tuba", varios empleados salieron del NYIT. Al parecer el director del instituto

nunca acepto que se contrataran directores de cine para crear la película, razón por la cual el resultado no era el mejor que se podía haber obtenido.

James Blinn, desarrolló un algoritmo similar al de texturado, pero en vez de representar color representaba profundidad. Los colores mapeados provocan que la superficie tenga un relieve o una depresión. Las partes blancas de la imagen son representados como protuberancias, mientras las partes oscuras representan las depresiones. Dotando de texturas y relieves se pueden crear modelos bastante realistas. El algoritmo fue nombrado "bump map". Otro algoritmo presentado por Blinn es el de reflectividad, con el cual se simula un reflejo del ambiente en el que se encuentra el objeto.
De la universidad de Cornell, Rob Cook planteó un nuevo algoritmo que erradicaba algunas de las limitantes de las representaciones anteriores. Cook aprecio que las representaciones de la época eran de apariencia plástica. Usando la variable de energía luminosa que emite la luz virtual logro crear un material que se parece al de un metal pulido. Los métodos anteriores consideraban el brillo de la luz sintética.

La graficación por computadora constituyó un pequeño y especializado campo hasta principios de la década de 1980, sobre todo debido al elevado costo del hardware y por consiguiente los escasos programas de aplicación. Aparecen entonces las computadoras personales con pantallas gráficas de barrido, como Apple Machintosh, IBM y su familia que popularizan los gráficos de mapas de bits para la interacción con el usuario.

Las computadoras se han convertido en una herramienta poderosa para producir imágenes en forma rápida y económica. De hecho, no existe ninguna área en que no se puedan aplicar las gráficas por computadora con algún beneficio y como consecuencia, no es sorprendente encontrar que se haya generalizado tanto el uso de las gráficas por computadora. A pesar de que las primeras aplicaciones en la ingeniería y la ciencia debían depender de equipo costoso y complicado, los avances en la tecnología de la computación han hecho que las gráficas interactivas por computadora sean una herramienta práctica.

Hoy en día vemos que la graficación por computadora se utiliza de manera rutinaria en diversas áreas, como en la ciencia, ingeniería, empresas, industria, gobierno, arte, entretenimiento, publicidad, educación, capacitación y presentaciones.

Programación Orientada a Objetos

Actualmente una de las áreas más candentes en la industria y en el ámbito académico es la orientación a objetos. La orientación a objetos promete mejoras de amplio alcance en la forma de diseño, desarrollo y mantenimiento del software ofreciendo una solución a largo plazo a los problemas y preocupaciones que han existido desde el comienzo en el desarrollo de software: la falta de portabilidad del código y reusabilidad, código que es difícil de modificar, ciclos de desarrollo largos y técnicas de codificación no intuitivas.

Un lenguaje orientado a objetos ataca estos problemas. Tiene tres características básicas: debe estar basado en objetos, basado en clases y capaz de tener herencia de clases.

El elemento fundamental de la OOP es, como su nombre lo indica, el objeto. Podemos definir un objeto como un conjunto complejo de datos y programas que poseen estructura y forman parte de una organización.

Esta definición especifica varias propiedades importantes de los objetos. En primer lugar, un objeto no es un dato simple, sino que contiene en su interior cierto número de componentes bien estructurados. En segundo lugar, cada objeto no es un ente aislado, sino que forma parte de una organización jerárquica o de otro tipo.

Estructura de un objeto.

Un objeto puede considerarse como una especie de cápsula dividida en tres partes:

-Relaciones
-Propiedades
-Métodos

Las relaciones permiten que el objeto se inserte en la organización y están formadas esencialmente por punteros a otros objetos.

Las propiedades distinguen un objeto determinado de los restantes que forman parte de la misma organización y tiene valores que dependen de la propiedad de que se trate. Las propiedades de un objeto pueden ser heredadas a sus descendientes en la organización.

Los métodos son las operaciones que pueden realizarse sobre el objeto, que normalmente estarán incorporados en forma de programas (código) que el objeto es capaz de ejecutar y que también pone a disposición de sus descendientes a través de la herencia.

Encapsulamiento y ocultación

Cada objeto es una estructura compleja en cuyo interior hay datos y programas, todos ellos relacionados entre sí, como si estuvieran encerrados conjuntamente en una cápsula. Esta propiedad (encapsulamiento), es una de las características fundamentales en la OOP.

La capacidad de presentación de información dentro de un objeto se divide en dos partes bien diferenciadas:

Interna: La información que necesita el objeto para operar y que es innecesaria para los demás objetos de la aplicación. Estos atributos son denominados privados y tienen como marco de aplicación únicamente a las operaciones asociadas al objeto.

Externa: La que necesitan el resto de los objetos para interactuar con el objeto que definimos. Estas propiedades se denominan públicas y corresponde a la información que necesitan conocer los restantes objetos de la aplicación respecto del objeto definido para poder operar.

Podemos imaginarla encapsulación como introducir el objeto dentro de una caja negra donde existen dos ranuras denominadas entrada y salida. Si introducimos datos por la entrada automáticamente obtendrá un resultado en la salida. No necesita conocer ningún detalle del funcionamiento interno de la caja.

El término encapsulación indica l capacidad que tienen los objetos de construir una cápsula a su alrededor, ocultando la información que contienen (aquélla que es necesaria para su funcionamiento interno, pero innecesaria para los demás objetos) a las otras clases que componen la aplicación.

Aunque a primera vista la encapsulación puede parecer superflua, tengamos en cuenta que existen muchas variables utilizadas de forma temporal: contadores y variables que contienen resultados intermedios, etc. D no ser por la encapsulación estas variables ocuparían memoria y podrían interferir en el funcionamiento del resto de los objetos.
La encapsulación no es exclusiva de los lenguajes de programación orientados a objetos. Aparece en los lenguajes basados en procedimientos (PASCAL, C, COBOL, ETC.) como una forma de proteger los datos que se manipulan dentro de las funciones.

Los lenguajes OOP incorporan la posibilidad de encapsular también las estructuras de datos que sirven como base a las funciones. Aportan por tanto un nivel superior en cuanto a protección de información.

La encapsulación nos permite el uso de librerías de objetos para el desarrollo de nuestros programas. Recordemos que las librerías son definiciones de objetos de propósito general que se incorporan a los programas. Al ser el objeto parcialmente independiente en su funcionamiento del programa en donde está definido, ya que contiene y define todo lo que necesita para poder funcionar, es fácil utilizarlo en los más variados tipos de aplicaciones. Si aseguramos, depurando las propiedades y las operaciones dentro de la clase que el objeto función bien dentro de una aplicación, con una correcta encapsulación el objeto podrá funcionar en cualquier otra.

Otra de las ventajas de la encapsulación es que, al definir el objeto como una caja negra con entradas y salida asociadas, en cualquier momento podemos cambiar el contenido de las operaciones del objeto, de manera que no afecte al funcionamiento general del programa.

La encapsulación está en el núcleo de dos grandes pilares de la construcción de sistemas; mantenibilidad y reusabilidad.

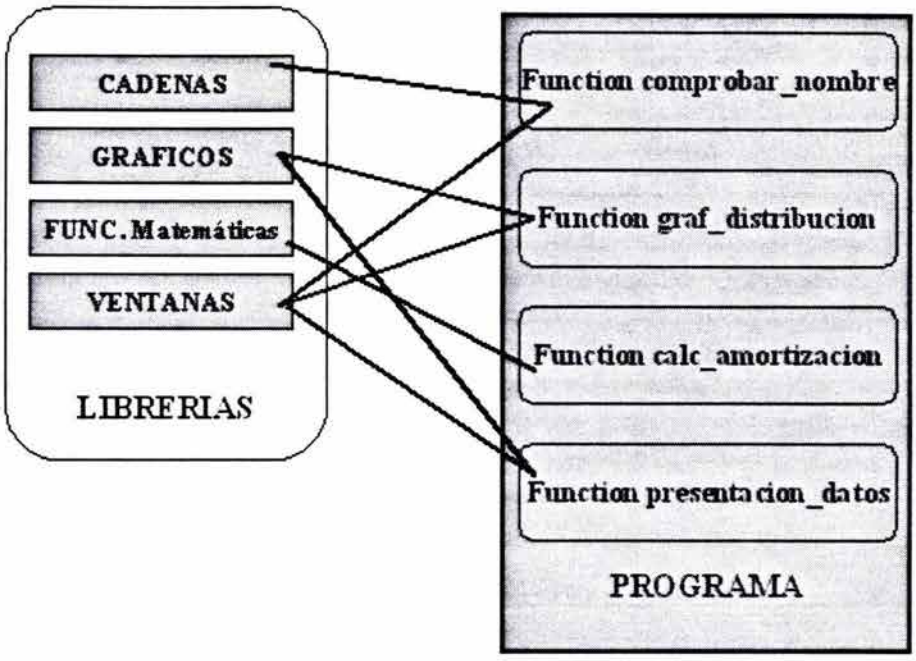


Figura I.1. Programación orientada a objetos, encapsulamiento.

El hecho de que cada objeto sea una cápsula facilita enormemente que un objeto determinado pueda ser transportado a otro punto de la organización, o incluso a otra organización totalmente diferente que precise de él. Si el objeto ha sido bien construido, sus métodos seguirán funcionando en el nuevo entorno sin problemas. Esta cualidad hace que la OOP sea muy apta para la reutilización de programas.

Mantenibilidad:

Cualidad que indica que un programa o sistema debe ser fácilmente modificable. Es decir que los cambios en las condiciones externas (como la definición de una nueva variable) implicarán modificaciones pequeñas en el programa / sistema. El concepto de mantenibilidad implica que un programa, al igual que un ser vivo debe ser capaz de adaptarse a un medio ambiente siempre cambiante.

Reusabilidad:

Cualidad que nos indica qué partes del programa (en este caso objetos) pueden ser reutilizados en la confección de otros programas. Ello implica que los objetos definidos en un programa pueden ser extraídos del mismo e implantados en otro sin tener que realizar modificaciones importantes en el código del objeto. El objeto final es que el programador construya una librería de objetos que le permita realizar programas basándose en la técnica de cortar y pegar. Ésta extrae (corta) código de otras aplicaciones ya realizadas y las implementa (pega) en la aplicación a realizar donde, tras algunos retoques, la nueva aplicación estará lista para funcionar. Como podrá observar el concepto de reusabilidad, permite reducir el tiempo de realización, ganando en claridad, mantenibilidad y productividad.

La encapsulación de datos se muestra como una herramienta poderosa que nos permite ganar en tiempo de desarrollo y claridad, con el único coste adicional de definir con precisión las entradas y salida de nuestras operaciones.

Polimorfismo:

El polimorfismo es una nueva característica aportada por la OOP. Esta propiedad indica la posibilidad de definir varias operaciones con el mismo nombre, diferenciándolas únicamente en los parámetros de entrada. Dependiendo del objeto que se introduzca como parámetro de entrada, se elegirá automáticamente cual de las operaciones se va a realizar.
Ya está habituado al operador <<suma>> que está presente en todos los lenguajes de programación. Sin embargo, los operadores <<suma de fracciones>> y <<suma de números complejos>> no existen en casi ningún lenguaje de programación.

Los lenguajes OOP permiten definir un operador <<suma>> tal que reconozca que tipo de objeto se le está aplicando, a través de operaciones de objetos. Previamente deberá definir la fracción y el número complejo como una clase y la operación suma como una operación de una clase.

Definiendo adecuadamente las operaciones suma de fracciones y suma de números imaginarios, el operador suma devolverá, en el caso que los operandos sean fracciones, una fracción y, en el caso de los números imaginarios, otro número imaginario.

Es posible extender el concepto e incluso definir operaciones como suma de bases de datos.

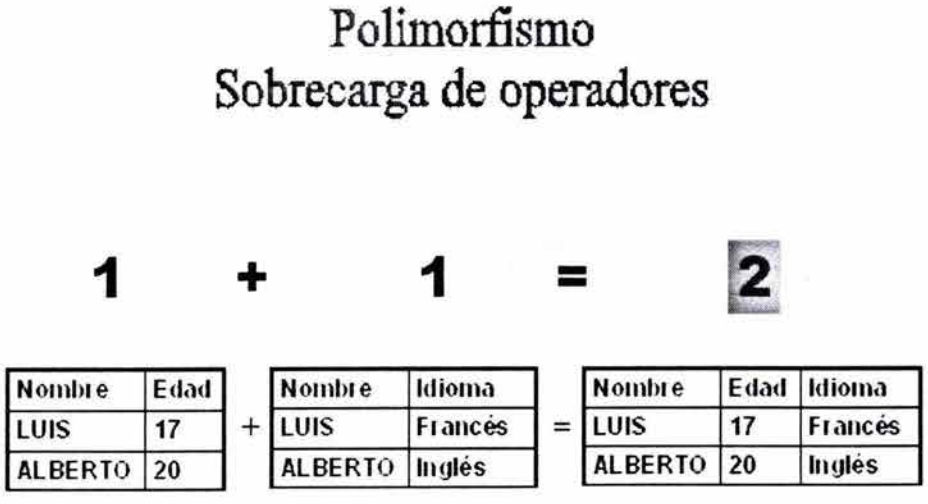


Figura I.2. Programación orientada a objetos, polimorfismo.

Una de las ventajas más importantes, sin entrar en la redefinición de operadores es permitir la realización de las clases que definen un programa de forma totalmente independiente al programa donde se utilizan. Gracias a la encapsulación y el polimorfismo, aunque se utilicen los mismos nombre con las operaciones en dos clases distintas, el programa reconoce a que clase se aplica durante la ejecución.

Como se podrá observar el polimorfismo y la encapsulación de datos están íntimamente ligados y nos permiten un mayor grado de mantenibilidad y reusabilidad que los lenguajes tradicionales. Esta es precisamente una de las causas de la revolución que ha supuesto la introducción de los lenguajes orientados a objetos dentro de la programación.

Herencia:

La herencia es la última de las propiedades relativas a la OOP, Consiste en la propagación de los atributos y las operaciones a través de distintas sub-clases definidas a partir de una clase común.

Introduce, por tanto, una posibilidad de refinamiento sucesivo del concepto de clase. Nos permite definir una clase principal y, a través de sucesivas aproximaciones, cualquier característica de los objetos. A partir de ahora definiremos como sub-clases todas aquellas clases obtenidas mediante refinamiento de una (o varias) clases principales.

La herencia nos permite crear estructuras jerárquicas de clases donde es posible la creación de sub-clases que incluyan nuevas propiedades y atributos. Estas sub-clases admiten la definición de nuevos atributos, así como crear, modificar o inhabilitar propiedades.

Además, es posible que una sub-clase herede atributos y propiedades de más de una clase. Este proceso se denomina herencia múltiple.

La herencia es, sin duda alguna, una de las propiedades más importantes de la OOP, ya que permite, a través de la definición de una clase básica, ir añadiendo propiedades a medida que sean necesarias, además, en el sub-conjunto de objetos que sea preciso.

La herencia permite que los objetos pueden compartir datos y comportamientos a través de las diferentes sub-clases, sin incurrir en redundancia. Más importante que el ahorro de código, es la claridad que aporta al identificar que las distintas operaciones sobre los objetos son en realidad una misma cosa.

Relaciones:

Las relaciones entre objetos son, precisamente, los enlaces que permiten a un objeto relacionarse con aquellos que forman parte de la misma organización.

Las hay de dos tipos fundamentales:

Relaciones jerárquicas. Son esenciales para la existencia misma de la aplicación porque la construyen. Son bidireccionales, es decir, un objeto es padre de otro cuando el primer objeto se encuentra situado inmediatamente encima del segundo en la organización

en la que ambos forman parte; asimismo, si un objeto es padre de otro, el segundo es hijo del primero. Una organización jerárquica simple puede definirse como aquella en la que un objeto puede tener un solo padre, mientras que en una organización jerárquica compleja un hijo puede tener varios padres.

Relaciones semánticas. Se refieren a las relaciones que no tienen nada que ver con la organización de la que forman parte los objetos que las establecen. Sus propiedades y consecuencias sólo dependen de los objetos en sí mismos (de su significado) y no de su posición en la organización.

Propiedades.

Todo objeto puede tener cierto número de propiedades, cada una de las cuales tendrá, a su vez, uno o varios valores. En OOP, las propiedades corresponden a las clásicas "variables" de la programación estructurada. Son, por lo tanto, datos encapsulados dentro del objeto, junto con los métodos (programas) y las relaciones (punteros a otros objetos). Las propiedades de un objeto pueden tener un valor único o pueden contener un conjunto de valores más o menos estructurados (matrices, vectores, listas, etc.). Además, los valores pueden ser de cualquier tipo (numérico, alfabético, etc.) si el sistema de programación lo permite.

Pero existe una diferencia con las "variables", y es que las propiedades se pueden heredar de unos objetos a otros. En consecuencia, un objeto puede tener una propiedad de maneras diferentes:

Propiedades propias. Están formadas dentro de la cápsula del objeto.

Propiedades heredadas. Están definidas en un objeto diferente, antepasado de éste (padre,"abuelo", etc.). A veces estas propiedades se llaman propiedades miembro porque el objeto las posee por el mero hecho de ser miembro de una clase.

Métodos.

Una operación que realiza acceso a los datos. Podemos definir método como un programa estructurado escrito en cualquier lenguaje, que está asociado a un objeto determinado y cuya ejecución sólo puede desencadenarse a través de un mensaje recibido por éste o por sus descendientes.

Son sinónimos de 'método' todos aquellos términos que se han aplicado tradicionalmente a los programas, como procedimiento, función, rutina, etc. Sin embargo, es conveniente utilizar el término 'método' para que se distingan claramente las propiedades especiales que adquiere un programa en el entorno OOP, que afectan fundamentalmente a la forma de invocarlo (únicamente a través de un mensaje) y a su campo de acción, limitado a un objeto y a sus descendientes, aunque posiblemente no a todos.

Si los métodos son programas, se deduce que podrían tener argumentos, o parámetros. Puesto que los métodos pueden heredarse de unos objetos a otros, un objeto puede disponer de un método de dos maneras diferentes:

Métodos propios. Están incluidos dentro de la cápsula del objeto.

Métodos heredados. Están definidos en un objeto diferente, antepasado de éste (padre,"abuelo", etc.). A veces estos métodos se llaman métodos miembro porque el objeto los posee por el mero hecho de ser miembro de una clase.

Beneficios que se obtienen del desarrollo con OOP

Día a día los costos del Hardware decrecen. Así surgen nuevas áreas de aplicación cotidianamente: procesamiento de imágenes y sonido, bases de datos, automatización de oficinas, ambientes de ingeniería de software, etc. Aún en las aplicaciones tradicionales encontramos que definir interfases hombre-máquina "a-la-Windows" suele ser bastante conveniente.

Por otra parte los costos de producción de software siguen aumentando; el mantenimiento y la modificación de sistemas complejos suelen ser una tarea trabajosa; cada aplicación, (aunque tenga aspectos similares a otra) suele encararse como un proyecto nuevo, etc.

Todos estos problemas aún no han sido solucionados en forma completa. Pero como los objetos son transladables (teóricamente) mientras que la herencia permite la reusabilidad del código orientado a objetos, es más sencillo modificar código existente porque los objetos no interaccionan excepto a través de mensajes; en consecuencia un cambio en la codificación de un objeto no afectará la operación con otro objeto siempre que los métodos respectivos permanezcan intactos. La introducción de tecnología de objetos como una herramienta conceptual para analizar, diseñar e implementar aplicaciones permite obtener aplicaciones más modificables, fácilmente extendibles y a partir de componentes reusables. Esta reusabilidad del código disminuye el tiempo que se utiliza en el desarrollo y hace que el desarrollo del software sea más intuitivo porque la gente piensa naturalmente en términos de objetos más que en términos de algoritmos de software.

Problemas derivados de la utilización de OOP en la actualidad

Curvas de aprendizaje largas. Un sistema orientado a objetos ve al mundo en una forma única. Involucra la conceptualización de todos los elementos de un programa, desde subsistemas a los datos, en la forma de objetos. Toda la comunicación entre los objetos debe realizarse en la forma de mensajes. Esta no es la forma en que están escritos los programas orientados a objetos actualmente; al hacer la transición a un sistema orientado a objetos la mayoría de los programadores deben capacitarse nuevamente antes de poder usarlo.

Dependencia del lenguaje. A pesar de la portabilidad conceptual de los objetos en un sistema orientado a objetos, en la práctica existen muchas dependencias. Muchos lenguajes orientados a objetos están compitiendo actualmente para dominar el mercado. Cambiar el lenguaje de implementación de un sistema orientado a objetos no es una tarea sencilla; Por ejemplo C++ soporta el concepto de herencia múltiple mientras que SmallTalk no lo soporta; En consecuencia la elección de un lenguaje tiene ramificaciones de diseño muy importantes.

Determinación de las clases. Una clase es un molde que se utiliza para crear nuevos objetos. En consecuencia es importante crear el conjunto de clases adecuado para un proyecto. Desdichadamente la definición de las clases es más un arte que una ciencia. Si bien hay muchas jerarquías de clase predefinidas usualmente se deben crear clases específicas para la aplicación que se esté desarrollando. Luego, en 6 meses ó 1 año se da cuenta que las clases que se establecieron no son posibles; En ese caso será necesario reestructurar la jerarquía de clases devastando totalmente la planificación original.

Desempeño. En un sistema donde todo es un objeto y toda interacción es a través de mensajes, el tráfico de mensajes afecta el rendimiento. A medida que la tecnología avanza y la velocidad de procesamiento, potencia y tamaño de la memoria aumentan, la situación mejorará.

Idealmente, habría una forma de atacar estos problemas eficientemente al mismo tiempo que se obtienen los beneficios del desarrollo de una estrategia orientada a objetos. Debería existir una metodología fácil de aprender e independiente del lenguaje, y fácil de reestructurar que no drene el rendimiento del sistema.

Objetivos

Calakmul representa un sitio arqueológico de gran importancia en la historia de la cultura maya y sin embargo hoy en día resulta ser uno de los más desconocidos a nivel nacional, estando por encima de éste lugares de mucho más renombre como Palenque, monte Alban, Teotihuacan, etc.

Esto se debe, en parte, a su recientemente descubierto 1980 por lo que no se le ha hecho mucha difusión, Calakmul, declarado Patrimonio de la Humanidad por la UNESCO el 27 de Junio del 2002. esta situado en medio de una Reserva Nacional protegida, Por esto Calakmul está destinado a permanecer parcialmente cubierto por la selva. La lejanía de este sitio arqueológico próximo a la frontera con Guatemala, no ha permitido que sean exhibidas algunas de las invaluables máscaras y piezas arqueológicas encontradas en 17 tumbas descubiertas hasta el momento.

De ahí que Calakmul sea el sitio arqueológico idóneo para desarrollar un proyecto en el cual se propone el uso de las gráficas por computadora como medio de exhibición y difusión de nuestro patrimonio cultural. La creación de un entrono virtual representa una manera efectiva de dar conocer este sitio puesto que en la realidad la espesa vegetación de la selva así como la falta de caminos y brechas por las cuales transitar hace de una visita a Calakmul un recorrido muy demandante.

Otra característica por la que Calakmul puede beneficiarse de reproducciones virtuales es la interactividad que se puede lograr con el usuario ya que en la actualidad no es posible exhibir en el sitio las invaluables piezas arqueológicas ahí encontradas. Mediante una simulación del sitio arqueológico el usuario podría no sólo observar las piezas sino podría obtener información e interactuar con otros personajes.

El problema principal que se establece al hacer un proyecto de este tipo es el largo tiempo y del alto costo de desarrollo, esto se debe a que la aplicación involucra áreas de estudio no muy desarrolladas en México, tal es el caso de la graficación por computadoras, además existe una falta de estándares.

De acuerdo con los requisitos que se pueden presentar al desarrollar el proyecto, en donde básicamente se solicitan altos niveles de interactividad con el usuario, representaciones gráficas lo más realistas posibles para sistemas en tiempo real y un bajo costo tanto en producción como en equipo de computo, una de las posibles soluciones es la utilización de un motor de juegos, estos le ofrecen a los programadores no sólo las herramientas necesarias para realizar aplicaciones con un despliegue gráfico de objetos tridimensionales sino también ofrecen un conjunto extra de herramientas con las cuales se hace mas fácil la tarea de programar aplicaciones como la que se pretende desarrollar.

También se pretende hacer uso de un API para la creación de historias interactivas, API que es capaz de proporcionar funciones básicas para el desarrollo de la aplicación, algunas de las funciones que el API proporciona son: animación, detección de colisiones, búsqueda de caminos, entidades basadas en máquinas de estados, conversión de texto a voz, etc.

Se espera que este proyecto proporcione una herramienta de estudio útil a los arqueólogos e investigadores además favorecer el turismo a una ciudad maya poco visitada hasta el momento.

Técnicamente se espera que la aplicación ofrezca un despliegue tridimensional del sitio deseado, en donde el usuario pueda moverse libremente y se le permita interactuar con objetos y personas que se encuentren en el lugar, todo esto enfocado a que el usuario aprenda de la experiencia de usar el software. De manera general se espera que la implementación en software funcione para varios tipos de recorridos virtuales y no esté limitada a un recorrido fijo en un solo lugar, que el costo económico y el tiempo necesario para cambiar el sitio o zona arqueológica del recorrido no sean elevados ya que en un museo los cambios que se le puedan hacer a las exhibiciones son muy importantes.

Entre algunas otras cosas se pretende que este entorno también tenga vida artificial, así como un agente inteligente, personificado por un arqueólogo, que ayude al visitante a recorrer el sitio. Todo esto con la finalidad de que el recorrido sea más interesante.

Capítulo 1
El motor de juegos Nebula

1.1 Nebula

Nebula es un motor de código abierto para juegos y visualización 3D en tiempo real, el cual está escrito en C++, reconoce scripts de tcl/tk[1], phyton y lua, su sistema de dibujo soporta tanto DirectX 8.1[2] como OpenGL[3] además corre bajo los sistemas operativos Windows y Linux.

Nebula sólo ofrece el marco de trabajo, esto quiere decir que para crear una aplicación con Nebula es necesario escribir algo de código en C++. El programador tiene el control. Nebula no es una aplicación para el desarrollo rápido de juegos en tres dimensiones, provee las herramientas para que los programadores puedan crear juegos 3D de una manera sencilla y eficiente.

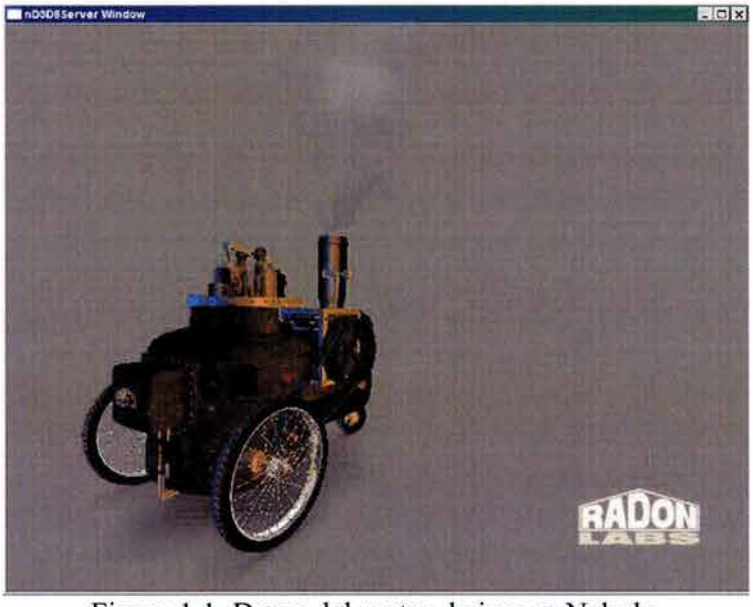


Figura 1.1. Demo del motor de juegos Nebula.

[1] Abreviatura de Tool Comand Language, el cual es un poderoso lenguaje de programación interpretado desarrollado por John Ousterhout. Una de las principales características de Tcl es que puede ser fácilmente implementado en otras aplicaciones debido al uso de librerías personalizadas de Tcl.

[2] DirectX es una colección de APIs desarrolladas por Microsoft, las cuales permiten a los programadores hacer programas que hagan uso de características del hardware de una computadora sin tener que saber exactamente que hardware estará instalado en la máquina donde el programa será finalmente ejecutado. DirectX logra esto mediante la creación de una capa intermedia que traduce comandos genéricos de hardware en comandos específicos para cada pieza específica de hardware. DirectX principalmente permite que las aplicaciones multimedia tomen ventaja de la aceleración por medio de hardware proporcionado por los aceleradores gráficos.

[3] OpenGL es un lenguaje para gráficas 3D desarrollado por Silicon Graphics. Existen dos implementaciones principales de OpenGL. Microsoft OpenGL el cual está construido sobre el sistema operativo Windows y está diseñado para mejorar el rendimiento en hardware que soporte el estándar de OpenGL. Como OpenGL por otra parte es una implementación de software específicamente diseñado para máquinas que no poseen un acelerador gráfico.

1.1.1. Conocimientos necesarios para el uso de Nebula.

Nebula no es una herramienta de trabajo para un usuario final ya que para poder trabajar con Nebula es necesario que los programadores que deseen hacer una aplicación posean conocimientos de:

-La tecnología empleada en los sistemas de dibujo en tiempo real, como matrices y transformaciones.
-Conceptos de animación 3D.
-El flujo principal de la aplicación que se desea realizar.

Para usar Nebula de una manera efectiva es recomendable haber programado una aplicación 3D en tiempo real, como por ejemplo un cubo girando. Un conocimiento profundo sobre OpenGL y DirectX también son de gran ayuda debido a que el API de Nebula en su interfaz de dibujo es muy similar a OpenGL. Haber programado un juego 3D o una aplicación 3D interactiva, la cual requiere de entradas del usuario para interactuar con el mundo 3D, ayudan a entender en detalle el tipo de problemas (de programación y de arquitectura) que Nebula pretende resolver. Actualmente Nebula carece de documentación introductoria, esta carencia hace que el motor sea difícil de usar y entender si es que no se está acostumbrado a programar aplicaciones de este tipo.

Con base en estos conocimientos, le será posible al programador escribir el ciclo principal, ejecutar la lógica apropiada y usar los servicios que Nebula proporciona.

1.2. Compilar Nebula

En este apartado se busca dar una pequeña guía para compilar el paquete de Nebula, ya que compilar el motor de juegos es el primer paso para poder hacer una aplicación, antes de indicar los pasos para compilar Nebula es importante mencionar los requerimientos necesarios para hacer esto.

-Tener instalado un compilador de visual C++, MS Visual C 6/7(.NET).
-Instalar el SDK[4] (Software Development Kit) de DirectX 8/8.1.
-Instalar la versión 4.5 o superior de STLport[5] el cual se encuentra en la página http://www.stlport.org/.
-La distribución de tcl8.4 de la página http://www.scriptics.com.
-Instalar las librerías DevIL[6] (http://openil.sourceforge.net). Para compilar Nebula sólo son necesarios los binarios. El código fuente de estas librerías solamente es necesario si se desea agregar o quitar formatos de imágenes.

---

[4] Abreviación de kit de desarrollo de software, es un paquete de programación el cual permite al programador desarrollar aplicaciones para una plataforma específica. Generalmente un SDK incluye una o más APIs, herramientas de programación y documentación.

[5] Para más información acerca de STLPort y para que sirve visitar la dirección de Internet http://www.stlport.org/.

[6] Para más información acerca de DevIL y para que sirve visitar la dirección de Internet http://openil.sourceforge.net.

Si se descargó el paquete de Nebula desde el CVS, es necesario copiar las librerías dinámicas de TCL y TK dentro del directorio nebula\bin\win32. Además de descargar una versión precompilada de DevIL y colocar las librerías dinámicas en la misma carpeta.

Una vez que se han cumplido con estos requerimientos es posible compilar Nebula, el paquete de Nebula para el sistema operativo Windows contiene el código fuente del motor, una vez que se ha abierto el paquete hay que correr el script[7] updsrc.tcl, en general hay que correr este script cada vez que se quiera actualizar el proyecto en el que se está trabajando. Si se está usando visual studio 7 o .net es necesario crear una variable de ambiente en el sistema como se muestra acontinuación.

```
set N_VSNET=yes
```

Esto hará que los archivos del proyecto sean compatibles con visual studio 7. En una línea de comandos de DOS hay que ejecutar la siguiente instrucción.

```
cd nebula\code\src
```

La ruta indicada en esta instrucción debe de ser la ruta en la que se instalo el paquete de Nebula. Posteriormente hay que ejecutar el script updsrc.tcl.

```
tclsh84 updsrc.tcl
```

Es posible compilar y correr Nebula sin los siguientes cambios, pero es recomendable que se tenga la ruta nebula\bin\win32 dentro del path de la computadora, para hacer esto sólo hay que hacer las siguientes modificaciones.

```
rem --- Nebula config vars ---
rem --- Cambiar para que concuerde con el directorio de instalación de Nebula ---
SET NOMADS_HOME=c:/dev/nebula

rem --- Directorio a los ejecutables de Nebula y librerías dinámicas ---
PATH %PATH%;%NOMADS_HOME%\bin\win32

rem ---Línea de comando para ambiente de Visual C ---
CALL D:\tools\vc6\vc98\bin\vcvars32.bat
```

Una vez que se ejecuto el script updsrc.tcl, es posible abrir el proyecto desde visual studio, dicho script se encarga de crear el archivo con extensión .dws este archivo se encuentra en el directorio nebula/code/vstudio/. Una vez abierto el proyecto de visual

---

[7] El termino script en general es usado como un sinónimo del termino macro o de los archivos de procesamiento por lotes, un script es una lista de comandos que pueden ser ejecutados sin la interacción del usuario. Un lenguaje de scripts es un lenguaje de programación sencillo en el cual se pueden escribir scripts.

studio, en la lengüeta "Tools/Options.../Directories/Executable files" hay que agregar el directorio donde se encuentran los archivos binarios de Tcl. Si se está usando Stlport su path tiene que estar arriba de todos los demás, esto en la lengüeta "Tools/Options.../Directories/Include". Lo único que resta es seleccionar el proyecto activo y hacer que visual studio lo compile. Todos los archivos ejecutables, y las librerías dinámicas que se crean son guardados dentro de la carpeta "nebula/bin/win32".

1.3. Programación en Nebula

Como se menciono [1.1] para crear una aplicación con Nebula hay que crear un ciclo principal, fuera de éste es necesario inicializar los servidores que se requieren, pongamos como ejemplo el caso del servidor gráfico, el cual se encarga del despliegue en pantalla así como de la creación de las ventanas de la aplicación, o del servidor de sonidos el cual permite la reproducción del audio. Una vez que todos los servicios requeridos están activos en el ciclo principal se programa todo aquello que el motor tiene que tomar en cuenta en un ciclo de ejecución así como de la lógica necesaria para que la aplicación funcione como se desea.

Por ejemplo la animación por medio de interpolación es uno de los servicios que ofrece Nebula, en pocas palabras usando Nebula es relativamente sencillo el crear una animación en la que algún objeto gire o se mueva, una toma de decisiones más compleja como establecer que algunos scripts sean ejecutados al cumplirse ciertas condiciones dentro del ciclo principal de la aplicación no pueden ser realizados de la misma manera que la animación por interpolación. Para asociar un comportamiento más complejo a un objeto es necesario hacer una subclase de dicho objeto en Nebula.

Es así que para dicha subclase se programan los nuevos métodos que corresponden a los comportamientos deseados, los cuales tienen que ser llamados y actualizados en el ciclo principal.

1.3.1. Creación de una aplicación usando Nebula.

Debido a que Nebula sólo se encarga de ofrecer las herramientas de programación, existen varias formas de crearla mediante el uso del motor.

Se puede hacer una aplicación independiente mediante el uso de un sistema constructor propio que ligue las librerías de Nebula, este no es el método más recomendable ya que no utiliza el sistema constructor de Nebula y si el programador no posee la experiencia suficiente es posible que la aplicación no esté bien integrada con Nebula.

Lo más recomendable es usar el sistema constructor de Nebula para la creación de las librerías y del archivo ejecutable. He aquí el método preferido. Para hacer esto.

Crear las clases y subclases necesarias para la aplicación supongamos como ejemplo las clases nWorld y nAplication, crear un archivo .pak el cual contiene los parámetros de construcción de nWorld y de nAplication, nWorld tendrá que ser construido como un paquete, mientras que nAplication tendrá que ser construido como un archivo ejecutable, agregar la función principal a el archivo naplication_main.cc, este será el punto de entrada al programa, correr el script updsrc.tcl, entonces la aplicación y librerías serán construidas y Nebula será capaz de cargar todas las librerías que se hayan creado para el proyecto.

1.3.2. Ejemplo de un ciclo principal sencillo.

En el apéndice B se presenta un archivo de C++ el cual ejemplifica la filosofía de programación que persigue Nebula, en términos generales en este ejemplo se habilitan varios servicios de Nebula, se cargan los scripts y se le indica al motor de juegos que los ejecute, entonces el ciclo principal maneja las entradas y le envía al servidor gráfico todo aquello que tiene que dibujar. Es importante señalar que la arquitectura de este programa no es representativa de la arquitectura de todos los programas de Nebula.

No es propósito de este ejemplo explicar en su totalidad como es que las alpicaciones que usan el motor de juegos Nebula funcionan, la intención es dar una idea general de cómo se crean los servidores y como dentro del ciclo principal del programa se actualizan los datos de dichos servidores para el despliegue, el tiempo, las entradas, etc.

1.4. Características de Nebula

Nebula proporciona una serie de servicios y estructuras de datos los cuales buscan ahorrarle al usuario esfuerzo al programar una aplicación relacionada con el campo de las gráficas por computadora.

Una organización jerárquica de todos los objetos, la cual resulta muy útil en aplicaciones como los videojuegos. En esta jerarquía es donde un objeto conocido como el servidor del núcleo almacena todas las instancias de todas las clases derivadas de nRoot, debido a esto nRoot es la superclase, estas clases forman los nodos y las relaciones padre / hijo que existen entre todos los nodos en un árbol (ver figura 1.2).

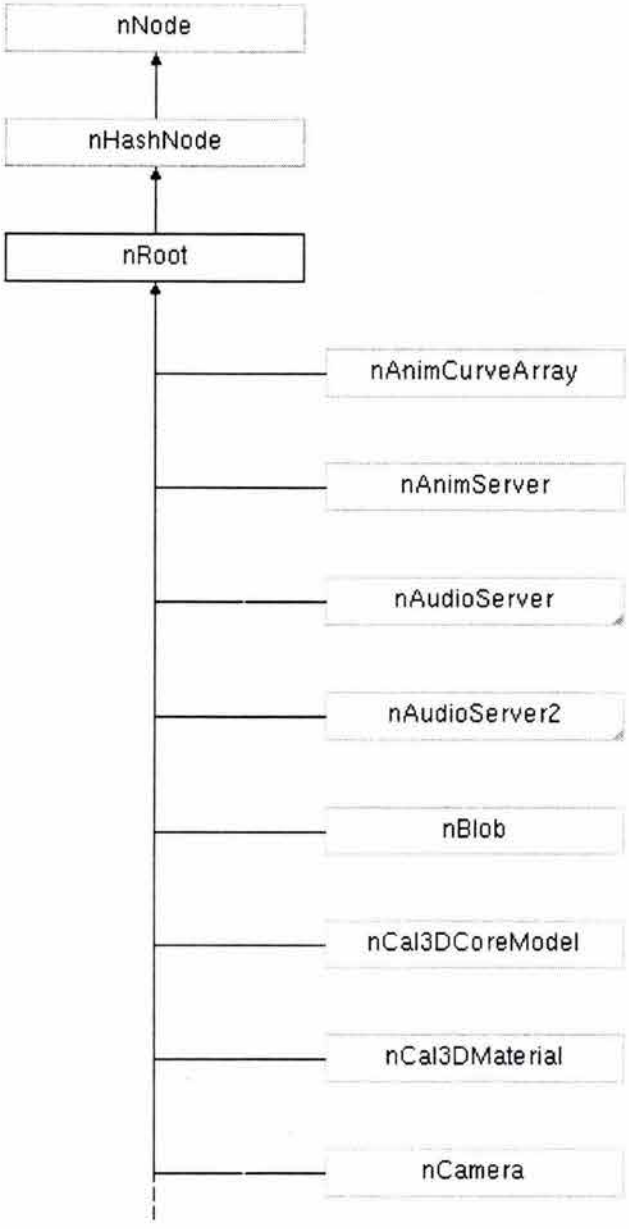


Figura 1.2. Relaciones padre / hijo entre clases de Nebula.

Así como la jerarquía de clases un grafo de escena es una manera de organizar y estructurar información a ser dibujada en una aplicación 3D, en muchos casos las librerías de los grafos de escena son la principal forma de estructurar y organizar información. En Nebula estas jerarquías proporcionan una manera lógica de organizar y estructurar los programas, permitiendo representar la información 3D independientemente de la estructura de la escena.

El nKernelServer maneja la jerarquía para la aplicación y la presenta como si fuera un sistema de archivos, así pues la aplicación siempre tiene un directorio o nodo en el que se encuentra trabajando en donde siempre se tienen algunas de las siguientes operaciones disponibles.

-Cambiar del directorio o nodo de trabajo.
-Enlistar los hijos del nodo actual.
-Obtener la ruta del nodo o directorio actual.
-Cambiar del nodo de trabajo a cualquier otro arbitrariamente mediante el uso de rutas absolutas o relativas.

En C++ algunas de las funciones para navegar a través de la estructura jerárquica pueden ser usadas mediante cadenas de caracteres o apuntadores a objetos nRoot, a continuación se muestran algunos sencillos ejemplos.

La siguiente línea de código crea un nuevo nodo.

```
nRoot* node = kernelServer->New("n3dnode", "/usr/scene/bob");
```

Borrar un nodo de la jerarquía.

```
if (node) node->Release();
```

Cambiar el nodo de trabajo para que sea cualquier ruta indicada dentro de la jerarquía usando direcciones absolutas o relativas.

```
kernelServer->SetCwd("../sibling");
kernelServer->SetCwd("/usr/scene");
```

También.

```
kernelServer->SetCwdObject(curNode->GetParent());
kernelServer->SetCwdObject(otherNode);
```

Enlistar los hijos del nodo actual.

```
nRoot* curChild;
for (curChild=(nRoot*)curNode->GetHead(); curChild; curChild=(nRoot*)curChild->GetSucc())
{
  ...
}
```

Obtener la ruta completa del nodo actual.

```
char buf[N_MAXPATH];
curNode->GetFullName(buf, N_MAXPATH);
```

Buscar un nodo dada su ruta.

```
node = kernelServer->LookUp("/usr/scene");
```

Meter el nodo actual dentro de una pila y hacer a otro nodo el nodo de trabajo.

```
kernelServer->PushCwd(otherNode);
```

Hacer el nodo de trabajo al último nodo de la pila.

```
kernelServer->PopCwd();
```

Nebula proporciona soporte de objetos gráficos y no gráficos, los objetos gráficos son las mallas que se dibujan, los objetos no gráficos son los objetos de la aplicación los cuales almacenan el estado de ésta. Otros objetos no gráficos incluyen los propios servidores de Nebula. Nebula proporciona un marco de trabajo unificado para almacenar ambas clases de objetos en una misma estructura de datos.

Nebula ofrece la libertad al programador de estructurar la jerarquía de objetos, aunque existe una estructura general que se recomienda seguir, una de las reglas principales es que se recomienda separar los objetos gráficos de los objetos de la aplicación, a continuación se muestra la estructura general que se recomienda seguir al crear la jerarquía de objetos.

/sys – Sistema de Nebula
 /lib – librerías de la aplicación.
  /shaders – librería de pixel shaders.
  /objects – librería de los objetos de la aplicación.
  /visuals – librería de objetos gráficos.

 /app – clases y servicios relacionados con la aplicación.
  /interface – proporciona el manejo de la interfaz.
  /camera – cámara controlada por el usuario.
  /world – mundo, ambiente y comportamiento de la simulación
   /octree – estructura especial.
   /map – datos del terreno o escena.
   /objects – objetos de la aplicación.
    /drone1 – objeto ejemplo.
    /drone2 ...etc

La idea detrás de esta estructura es que todo lo que se encuentra en el directorio /app sea especifico al contenido de una aplicación.

De la forma en la que Nebula fue concebido es posible que el programador use sólo lo que necesite del marco de trabajo. Todos los objetos pueden ser objetos de Nebula (derivados de nRoot) en donde toda la aplicación está uniformemente almacenada dentro de la jerarquía de clases teniendo siempre un fácil acceso al marco de trabajo de Nebula (clonar, scripts de Tcl, apuntadores inteligentes, etc).

1.4.1. Scripts de TCL.

Nebula interpreta scripts de Tcl como una manera sencilla de probar la funcionalidad de un objeto, un script puede ser modificado rápidamente e instantáneamente ser ejecutado para tratar de acceder a algún nuevo método, también puede llamar scripts de Tcl en cualquier momento que se desee (por ejemplo cada cuadro o cada vez que se detecte una colisión) proporcionando mayor flexibilidad a la aplicación en tiempo de ejecución.

1.4.2. La consola de Nebula.

Con Nebula se pueden inspeccionar y alterar los objetos en tiempo real mediante el uso de la consola, la función de la consola principalmente es la de un depurador ya que se puede preguntar por el estado de un objeto y correr los métodos de dicho objeto de Nebula, mientras la aplicación se está ejecutando en tiempo real, e inmediatamente ver los resultados, para lograr esta funcionalidad los objetos de la aplicación deben ser objetos de Nebula (descendientes de nRoot) esta característica se vuelve muy importante para depurar los nuevos objetos que se agregaron a la aplicación.

1.4.3. Características generales.

Algunas otras características del motor de Juegos Nebula son:

-Persistencia de objetos, cada objeto puede ser representado como una cadena de comandos de Tcl.
-Efectos especiales tales como sistemas de partículas.
-Librerías externas que pueden ser agregadas al motor para obtener mayor funcionalidad (ODE, Cal3D, cargador de niveles de quake3, etc.).
-La reproducción del sonido ya es soportada por el motor de juegos mediante el uso del servidor de sonidos.
-Soporte multiplataforma (Windows / Linux, DirectX / OpenGL).
-Nebula proporciona algunos servicios que permiten animar el mundo automáticamente cada cuadro.
-El motor trata de proporcionar un manejo transparente en el dibujo de las escenas, protegiendo al programador de preocuparse del cambio en los estados del dibujo de la escena.

1.4.4. Características no proporcionadas por Nebula.

Algunas de las características que el motor de Juegos Nebula no proporciona son:

-El motor no tiene acciones predeterminadas para llevar a cabo en cada cuadro, la aplicación es la que debe definir que hacer en cada cuadro.
-No proporciona un ciclo principal, es responsabilidad del programador decidir como es actualizado el mundo dentro del ciclo principal, teniendo en cuenta decisiones como cada cuanto tiempo el mundo necesita ser actualizado, cada cuanto tiempo la física, scripts e inteligencia artificial son llamados, cuales objetos son dibujados en un determinado cuadro, que hacer cuando hay una colisión, cuando se cierra la aplicación.

1.5. Arquitectura general del motor de juegos Nebula

Al hacer una aplicación con Nebula es importante tener en cuenta algunos puntos sobre la arquitectura del motor para así tener un mejor desempeño de éste. Es importante crear una clase personalizada del mundo, la cual defina que es lo que sucede en el ciclo principal.

En el ciclo principal es importante permitir que Nebula haga su trabajo, es decir permitir que Nebula actualice los servidores, para que esto suceda la función Trigger() de los servidores debe ser llamada dentro del ciclo principal de la aplicación. Algunos de los servidores más importantes que deben ser actualizados son: nInputServer, nConServer, nTimeServer y nGfxServer.

1.5.1. Los objetos servidor.

Los objetos servidores poseen un solo nombre y se encargan de proporcionar servicios específicos a otros objetos de Nebula, los cuales son llamados objetos cliente. Algunos de los objetos servidor incluyen:

Gfx. El servidor gráfico está envuelto alrededor de un API[8] 3d (OpenGL o D3D) y se encarga del dibujo en pantalla. Ofrece un pequeño API general el cual está optimizado para dibujar vertex buffers, no simples primitivas. Trabajo de bajo nivel como lo es el manejo de texturas, iluminación, transformaciones, etc, es asignado al API 3d para dar lugar a la aceleración por hardware

Time. El servidor de tiempo proporciona una fuente de tiempo real para simular la velocidad de los cuadros a dibujar. También ofrece un contador de tiempo el cual puede ser usado por algún objeto cliente.

Script. El servidor de scripts conecta el mecanismo interno de scripts con un lenguaje de scripts específico. Traduce comandos emitidos por los objetos en declaraciones del script y viceversa. El servidor de scripts también implementa un puerto IPC[9] que permite a otras aplicaciones, corriendo en otras máquinas, comunicarse con los objetos locales.

File. El servidor de archivos conecta el mecanismo de persistencia con el sistema de archivos. Posee el manejador de archivos el cual los objetos usan para cargar y salvar.

Input. El servidor de entradas es el punto donde se recolectan todos los eventos de entrada. Relaciona eventos de entrada de bajo nivel con eventos de la aplicación también puede ligar la ejecución de scripts con eventos de entrada.

Console. El servidor de la consola está construido alrededor del servidor gráfico, el servidor de entradas y el servidor de scripts, el servidor implementa una consola dentro de la aplicación. Lee datos del teclado por medio del servidor de entradas, manda líneas de comandos al servidor de scripts para su evaluación y dibuja texto a través del servidor gráfico.

La consola es una de las características más importantes de Nebula, es como trabajar con una consola convencional, pero en lugar de inspeccionar y manipular archivos uno puede al invocar comandos de script inspeccionar y manipular directamente objetos de C++. Los objetos pueden ser creados, borrados, copiados, etc. Finalmente la consola se

[8] Abreviación de Aplication Program Interface (Interfaz para la programación de aplicaciones). Un API consiste en un conjunto de rutinas, protocolos y herramientas para la construcción de aplicaciones de software. Una API hace más fácil desarrollar un programa al proporcionar todos los bloques de construcción, un programador sólo necesita juntar dichos bloques para crear la aplicación. A pesar de que las APIs están diseñadas fundamentalmente para el uso de los programadores las APIs resultan buenas para los usuarios ya que garantizan que todos los programas que usan un API en común tendrán una interfaz similar, esto hace más sencillo el aprender a usar nuevos programas.

[9] Comunicación entre procesos. Una capacidad soportada por algunos sistemas operativos la cual permite a un proceso comunicarse con otro proceso. Los cuales pueden estar ejecutándose en la misma computadora o en diferentes computadoras conectadas a través de una red.

puede comunicar con otras aplicaciones vía telnet[10] esto es que las aplicaciones en Nebula pueden ser controladas remotamente a través de la consola.

Sgraph. El servidor de grafos de escena es un dispositivo de dibujo de alto nivel, el cual recolecta los objetos a dibujar, haciendo algunas optimizaciones antes de que sean dibujados por el servidor gráfico.

Math. El servidor matemático implementa un software especializado en operaciones con arreglos de vértices. Cualquier objeto que haga operaciones simples con grandes arreglos de vértices debe usar el servidor matemático.

A continuación se ejemplifica como es que se actualizan algunos de estos servidores con código en C++.

El siguiente códigoactualiza los canales del tiempo.

```
timeServer->Trigger();
```

Se recolecta cualquier entrada de sistema operativo, revisando los eventos asociados con dichas entradas, si el servidor de entradas no se actualiza no se podrá mover el ratón, no se registrarán eventos de entrada al presionar alguna tecla, etc.

A diferencia del método Trigger() del servidor de tiempo, este toma un parámetro el cual le proporciona el tiempo actual.

```
double t = timeServer->GetFrameTime();
inputServer->Trigger(t);
```

Con el siguiente código se recolecta cualquier tecla presionada, y corre comandos etc, si no se realiza esta llamada dentro de la aplicación no se podrá hacer uso de la consola, el servidor de la consola es automáticamente actualizado cuando el servidor de entradas es actualizado.

```
consoleServer->Trigger();
```

El método de actualización del servidor gráfico maneja la ventana de despliegue, si no se actualiza el servidor la ventana no mostrará cambio alguno en la escena.

```
gfxServer->Trigger();
```

Como se menciono anteriormente estas funciones deben ser llamadas dentro del ciclo principal, generalmente cada cuadro, esto no quiere decir que algunas cosas no puedan ser actualizadas a diferentes tasas, como actualizar el servidor de entradas a 30hz y todos los demás servidores cada cuadro. Existen otros servidores, además de los anteriormente expuestos, los cuales tienen sus propios métodos Trigger() y que normalmente son menos importantes para el correcto funcionamiento de Nebula, algunos de estos servidores son: nParticleServer y nScriptServer.

---

[10] Programa que emula una terminal para redes TCP/IP. Telnet conecta una computadora con un servidor en la red. Se pueden dar comandos al programa y estos se ejecutarán como si se escribieran directamente en la consola del servidor. Esto permite controlar al servidor y comunicarse con otros servidores en la red.

1.5.2. La filosofía de Nebula.

Debido a la dificultad de poner en términos simples cual es la mejor filosofía a tomar cuando se crea una aplicación usando Nebula, se ha decidido hacer una lista de cosas que son consideradas buenas cuando se está haciendo uso del motor de juegos.

Usar el sistema constructor de Nebula. El sistema es muy fácil de usar y ahorra el tener que descifrar que configuración de proyecto usar. También el sistema constructor generará automáticamente los archivos del proyecto para Visual Studio6, Visual Studio 7, de esta manera el código puede ser reutilizado por más gente.

Tomar ventaja de las funciones de scripts, de esta forma las nuevas clases creadas pueden ser modificadas a través de la consola, obtener su estado, ser clonadas, etc.

Programar de una manera modular, esto significa diseñar y programar con la filosofía de que cada clase realiza una función bien definida tratando de reducir al mínimo el efecto de la interacción entre ellas, el contenido de cada función es cohesivo mientras que el nivel de acoplamiento entre las clases es bajo. Se debe tratar de no usar variables de control y banderas dentro de los parámetros de las funciones. La programación modular alienta a dividir las funciones en dos tipos, aquellas que controlan el flujo del programa y aquellas que manejan detalles de bajo nivel como lo es mover datos entre estructuras.

1.6. Uso del sistema constructor de Nebula

Como ya se menciono el sistema constructor de Nebula proporciona una forma sencilla de agregar nuevos módulos a la aplicación que se desea desarrollar, así pues cuando se desea hacer un paquete o crear un proyecto Nebula proporciona la herramienta llamada classbuilder.tcl. Dicha herramienta genera automáticamente un archivo .pak, un archivo fuente y un archivo de cabecera, aunque en ocasiones será necesario modificar el archivo .pak manualmente, para explicar de una manera más clara que son los archivos .pak y como se usan supongamos que la aplicación consiste de dos módulos nGame y nWorld.

Primero se crea el archivo nworld.pak.

```
# define module
  beginmodule nworld
    setdir game
    setfiles { nworld_main nworld_cmds }
    setheaders { nworld }
  endmodule
  # define target
  begintarget world
    settype package
    setmods { nworld }
    setdepends { nkernel.lib nnebula.lib }
    setlibs_win32 { nkernel.lib nnebula.lib }
    setlibs_unix  { nkernel nnebula }
  endtarget
```

Se tiene un modulo, nworld y un objetivo, se indica que el objetivo es del tipo paquete, esto significa que se generará un archivo world.dll[11], y que para generar dicho archivo se necesita el módulo nworld.

Posteriormente se crea el archivo ngame.pak.

```
beginmodule ngame
    setdir game
    setfiles { ngame_main ngame_cmds }
    setheaders { ngame }
  endmodule
  begintarget mygame
    settype exe
    setmods { ngame }
    setdepends { nkernel.lib nnebula.lib }
    # add .lib file if it is necessary
    setlibs_win32 { nkernel.lib nnebula.lib world.lib }
    setlibs_unix  { nkernel nnebula }
  endtarget
```

Hay que resaltar que el objetivo mygame es del tipo ejecutable, esto significa que ngame.pak generará el archivo mygame.exe, además se indica que mygame necesita de la librería llamada world.lib.

Una vez que se han creado estos archivos con classbuilder.tcl se puede ejecutar el script upsrc.tcl, upsrc.tcl se encargará de actualizar el espacio de trabajo nnebula.dsw, al abrir éste se observará que tanto ngame como nworld han sido agregados al proyecto.

Para hacer un espacio de trabajo propio sólo es necesario ingresar la siguiente línea de código al inicio del archivo ngame.pak.

```
workspace mygame
```

Este comando generará mygame.dsw, al abrir este marco de trabajo se observará que el proyecto sólo contiene el paquete ngame, para que los demás paquetes sean agregados será necesario hacer las siguientes modificaciones.

```
beginmodule ngame
    setdir game
    setfiles { ngame_main ngame_cmds }
    setheaders { ngame }
  endmodule
  begintarget mygame
    settype exe
    setmods { ngame }
    setdepends { nworld }
    setlibs_win32 { nkernel.lib nnebula.lib world.lib }
    setlibs_unix  { nkernel nnebula }
```

[11] Archivo de librería de ligado dinámico. Librería de datos o funciones que pueden ser usados por una aplicación de Windows. Generalmente un dll proporciona una o varias funciones a las cuales un programa tiene acceso creando una liga estática o dinámica al archivo dll.

```
endtarget
```

También es posible tener todos los archivos .pak de un nuevo proyecto en un solo archivo .pak, para el ejemplo anterior el archivo .pak sería.

```
workspace mygame
beginmodule nworld
   setdir game
   setfiles { nworld_main nworld_cmds }
   setheaders { nworld }
endmodule
beginmodule ngame
   setdir game
   setfiles { ngame_main ngame_cmds }
   setheaders { ngame }
endmodule
begintarget mygame
   settype exe
   setmods { nworld ncamera ngame }
   setdepends { nworld ncamera }
   setlibs_win32 { nkernel.lib nnebula.lib world.lib }
   setlibs_unix  { nkernel nnebula }
endtarget
```

1.6.1. Aclaración sobre el comando settype.

Settype dll no genera un TOC y sólo puede incluir una clase cuyo nombre debe ser el mismo que el del dll, si el nombre de la clase difiere del nombre del dll el script marcará un error.

Settype package, por otra parte permite para múltiples clases la generación de un TOC, permitiéndole al usuario nombrar las clases como desee, Nebula usará el TOC para determinar que dll cargar.

1.7. Agregar un nuevo módulo a Nebula

Agregar un nuevo módulo usando el marco de trabajo de Nebula es muy simple, a continuación se presenta una de las muchas maneras posibles de hacer esto, en esta sección se asume que la clase que se desea agregar trabajará con la interfaz de scripts de Nebula.

1.7.1. Creando el esqueleto del módulo.

La forma más sencilla de crear el esqueleto de una nueva clase, es tomar el código que se encuentra dentro del directorio /code/templates y usarlo como base, una forma más automatizada es usar el script que se encuentra dentro del directorio /code/templates/classbuilder.tcl ahora se dará una breve explicación de como editar este código para que sea funcional.

Se comenzará por explicar que es lo que se requiere dentro del archivo de cabecera.

```
#ifndef N_CLASSNAME_H
#define N_CLASSNAME_H

/**
@class nombre de la clase.

@Descripción breve de la clase

Descripción detallada de la clase
*/
#ifndef N_SUPERCLASSNAME_H
#include "subdir/nsuperclassname.h"
#endif

#undef N_DEFINES
#define N_DEFINES nClassName
#include "kernel/ndefdllclass.h"

//------------------------------------------------------------------------
class nClassName : public nSuperClassName
{
  public:
  /// constructor
  nClassName();
  /// destructor
  virtual ~nClassName();
  /// persistency
  virtual bool SaveCmds(nFileServer* fileServer);

  /// apuntador a nKernelServer
  static nKernelServer* kernelServer;
};
//------------------------------------------------------------------------
#endif
```

En este ejemplo sólo es necesario reemplazar todas las referencias a nClassName con el nombre de la nueva clase, también es necesario reemplazar las referencias a nSuperClassName con la clase de la que se hereda.

Dentro del archivo de código fuente, el archivo .cc, se programa la funcionalidad de la nueva clase, la parte más importante de esta clase se encuentra en la primera línea de código y en la llamada a nNebulaScriptClass, sin este código la nueva clase no funcionará a través de la interfaz de scripts de TCL, si la clase no va a implementar la interfaz de scripts se puede usar la macro nNebulaclass.

```
#define N_IMPLEMENTS nClassName
#include "subdir/nclassname.h"
nNebulaScriptClass(nClassName, "nsuperclassname");
//------------------------------------------------------------------------
/**
*/
nClassName::nClassName()
{
```

```
}
//------------------------------------------------------------------------------
/**
*/
nClassName::~nClassName()
{
}
```

De nuevo es necesario reemplazar todas las referencias a nClassName con el nombre de la nueva clase y reemplazar nsuperclassname con el nombre de la clase padre.

El último archivo que debe ser programado es el archivo de comandos, la utilidad de este archivo es que los métodos de la nueva clase pueden ser usados por medio de la consola y scripts de tcl. Ahora se presenta de una manera detallada el esqueleto de este archivo.

```
#define N_IMPLEMENTS nClassName
#include "subdir/nclassname.h"
#include "kernel/nfileserver.h"
```

Después de incluir las cabeceras necesarias, se debe definir cada una de las funciones que van a poder ser usadas mediante el uso de scripts, aquí se define una función llamada xxx.

```
static void n_xxx(void* slf, nCmd* cmd);
```

A continuación se agregan las llamadas a función, el primer parámetro se desglosa de la siguiente manera, el primer carácter corresponde al tipo de dato que regresa la función, éste es seguido por un guión bajo y el nombre de la función, esta cadena termina con otro guión bajo y el tipo de dato que necesita la función como parámetro. El segundo parámetro es un identificador para la llamada de la función, es importante que este identificador no sea usado en alguna de las clases padres. El último parámetro es la función que se desea invocar cuando esta función es llamada, necesita ser una de las funciones definidas anteriormente ya que no es posible llamar a funciones miembro directamente.

Tipo	Carácter
Void	v
String (char*)	s
Float	f
Bolean	b
Integer	i
Object	o
List	l

Tabla 1.1. Letras para cada tipo de parámetro.

```
void
n_initcmds(nClass* cl)
{
  cl->BeginCmds();
  cl->AddCmd("v_xxx_v", 'XXXX', n_xxx);
  cl->EndCmds();
}
```

La siguiente función de ejemplo llama a la función miembro XXX d ela clase nClassName.

```
static
void
n_xxx(void* slf, nCmd* cmd)
{
  nClassName* self = (nClassName*) slf;
  self->XXX();
}
```

La última parte del archivo de comandos es usada por el servidor de persistencia (nFileServer).

```
bool
nClassName::SaveCmds(nFileServer* fs)
{
  if (nSuperClassName::SaveCmds(fs))
  {
    nCmd* cmd = fs->GetCmd(this, 'XXXX');
    fs->PutCmd(cmd);
    return true;
  }
  return false;
}
```

Cualquier dato que el objeto desee tener de una forma persistente debe ser guardado vía objeto nCmd durante el método SaveCmds. Se debe asegurar que exista una llamada al método SaveCmds en la clase nSuperClassName.

1.8. El ciclo de dibujo de Nebula

Es útil conocer como es que Nebula dibuja una escena antes de tratar de escribir nuevas clases visuales. Hay tres clases con las que se necesita lidiar para poder dibujar , nGfxServer el API de dibujo de bajo nivel, nSceneGraph encargada de optimizar el dibujo de la jerarquía visual, nVisNode es la clase raíz de todos los objetos que pueden ser agrupados en jerarquías visuales los cuales saben como dibujarse ellos mismos, ejemplos de subclases de nVisNode son nMeshNode (define la geometría a ser dibujada), nTexArrayNode (define texturas), nShaderNode (describe materiales para superficies) o n3dNode (agrupa objetos en jerarquías, define la posición y orientación en tres dimensiones). Esta jerarquía se puede extender al hacer subclases de nVisNode o alguna de sus subclases.

La jerarquía visual clásica de Nebula consiste de un objeto n3dNode como padre y un objeto nMeshNode, nTexArrayNode y nShaderNode como hijos, esta jerarquía genera un objeto poligonal con textura el cual puede ser posicionado, orientado y escalado.

El ciclo de dibujo de Nebula es algo como lo que a continuación se presenta:

-BeginScene() es invocado por el objeto grafo de escena.
-Attach() es invocado por cada objeto raíz de la jerarquía visual que desea ser dibujado.
-EndScene() es llamado por el objeto grafo de escena.

El segundo paso es llamado el paso de agregado, por que durante este paso los objetos visuales se agregan ellos mismos a la escena. La finalidad del paso de agregado es dar al grafo de la escena una descripción altamente detallada del cuadro actual a ser dibujado. Un nodo grafo de escena contiene apuntadores a objetos que pueden proporcionar posición, geometría, textura, material e información sobre las fuentes de luz.

Al final del paso de agregado el objeto grafo de escena tiene la jerarquía completa de nodos de la escena con los apuntadores a todos los objetos que necesitan ser llamados a dibujar, antes de que suceda esto el objeto grafo de escena realiza los siguientes pasos para dibujar la escena.

Paso de transformación. A cada objeto nodo 3d se le pide que actualice su estado al invocar el método compute(). Posteriormente las correspondientes matrices 3D son transformadas de acuerdo a las coordenadas del observador.

Paso de Iluminación. Todos los nodos grafos de escena que contienen información sobre las fuentes de luz son organizados, el nodo de luz es actualizado al invocar el método compute(). Una vez que todas las luces fueron actualizadas son dibujadas a través de nGfxServer::SetLight().

Paso de organización. Los nodos restantes son organizados de acuerdo a su prioridad de dibujo, textura y opacidad. Esto optimiza el cambio de estado de las texturas y asegura que las superficies transparentes sean dibujadas correctamente.

Paso de dibujo. A cada nodo de material y de malla se le pide que actualice su estado interno al invocar el método compute(). Después los materiales y las texturas son dibujadas a través del servidor gráfico, finalmente se le indica al nodo malla que se dibuje a si mismo con lo cual finaliza el ciclo de dibujo.

Capítulo 2
Visita virtual a Calakmul

2.1. Historia de Calakmul

Al mencionar las ruinas arqueológicas de Calakmul a muchas personas les parecerán desconocidas, por lo que incluir un breviario histórico de esta ciudad maya recientemente encontradas ayudará a entender la importancia y la visión que se tiene al desarrollar una aplicación cuya principal finalidad es proporcionar un recorrido virtual por estas ruinas arqueológicas.

Figura 2.1. Calakmul, sitio arqueológico.

2.1.1. Preclásico.

La historia de los mayas de las Tierras Bajas centrales tuvo sus orígenes en el transcurso del Preclásico Medio, entre 900 y 300 a. C. A finales de este periodo, se construyeron en la región los templos más antiguos de la civilización maya y se difundieron los logros más sobresalientes de esa cultura por todas las Tierras Bajas. Aún en años recientes, la información arqueológica disponible describía a los mayas del Preclásico Medio como una sociedad agrícola básicamente igualitaria, con cacicazgos simples, que vivía en aldeas dispersas a lo ancho y largo de las Tierras Bajas. Esta visión se ha modificado por las investigaciones realizadas en Nakbé, Guatemala, en donde se reporta la construcción de monumentales estructuras cívico - ceremoniales durante ese periodo.

En el Preclásico, la región de Calakmul y el norte del Fetén guatemalteco integraron una esfera regional en la que Calakmul compartió con El Mirador, Nakbé y Uaxactún la historia geopolítica del área maya. En las Tierras Bajas centrales parecen definirse desde su más temprana ocupación dos polos que marcarán las relaciones y conflictos que se establecieron durante todo el periodo Clásico. Hasta antes del hallazgo de una estructura de más de 12 m de altura fechada entre 400 y 200 a. C., gracias a las recientes excavaciones en el interior de la llamada Estructura 11 de Calakmul, uno de los basamentos más grandes del Preclásico Tardío, se postulaba a Nakbé como el sitio predominante de la segunda mitad del Preclásico Medio (600-400 a. C.). De ese lugar provenía la información más temprana

del área sobre una arquitectura monumental de carácter cívico-ceremonial, lo cual da cuenta de la existencia de un sistema de control social desde este periodo. En la arquitectura del Preclásico de Nakbé se han encontrado estructuras que alcanzan los 13 m de altura. Esto indica que, durante el Preclásico Medio, en el área maya central, al igual que en otras regiones, ya se elaboraba una arquitectura de mampostería.

A finales del Preclásico Medio y principios del Tardío (400200 a. C.), en la región en la que se localizan los asentamientos de Nakbé y Calakmul surgió un sistema de organización sociopolítica que permitió la consolidación de estructuras de poder, las que permitirán emprender obras urbanas de gran envergadura. En el siguiente periodo, la complejidad social que ya se manifestaba en el Preclásico Medio y el crecimiento de la población propiciarían el surgimiento de ciudades de considerable magnitud, entre las que destaca Kaminaljuyú, en el alti-plano maya de Guatemala.

Para ese mismo momento, en las Tierras Bajas centrales del interior de la selva tropical, en el núcleo geográfico del mundo maya, se erigieron en El Mirador imponentes complejos triádicos y se construyeron en Calakmul las estructuras más voluminosas de toda su historia. Asimismo, se inició en Uaxactún y Tikal una larga tradición arquitectónica. En el transcurso del Preclásico Tardío, Calakmul, El Mirador y Uaxactún se consolidaron como una entidad regional con un sistema de organización política que mantuvo diferencias con sus vecinos del sur, especialmente con Tikal, con la que establecería una permanente hostilidad durante los dos siguientes periodos.

Después del incremento poblacional generalizado de las Tierras Bajas mayas, y de la aparición de las primeras sociedades complejas, hacia el 600 a. c., el eje de donde surgen los rasgos que van a caracterizar la esencia de la cultura de los mayas incluye Nakbé, El Mirador, Uaxactún, Calakmul y Tikal. Los dos últimos no sólo son los asentamientos más tempranos donde se han encontrado, hasta la fecha, evidencia de una secuencia de arquitectura pública ininterrumpida de más de 14 siglos, que va desde 400 a. C. hasta 900d. C.; también destacan por la gran cantidad de estelas[1] y monumentos fechados, en los que se plasmó una historia que abarca todo el periodo Clásico, desde el siglo III hasta principios del X.

La arquitectura de mampostería[2] con decoración modelada en estuco más temprana que se había registrado hasta antes de los recientes hallazgos de Calakmul, pertenece al llamado Edificio 34 del complejo El Tigre de El Mirador y a la famosa Pirámide EVII de Uaxactún, con sus mascarones que flanquean las cuatro escalinatas del basamento. Tikal también se integraría a esta tradición del Preclásico, en la que predominan los mascarones estucados, como los encontrados en la Acrópolis Norte o en estructuras del Mundo Perdido. Calakmul también participó activamente de esta tradición. La Estructura II, conserva todos sus atributos formales y estructurales, es a la fecha la construcción más temprana conocida en las Tierras Bajas centrales. La complejidad iconográfica de esta estructura muestra que, desde sus más tempranas manifestaciones, la arquitectura pública maya tenía una carga

[1] Monumento conmemorativo que se erige sobre el suelo en forma de pedestal o lápida.

[2] Obra hecha con piedras desiguales ajustadas y unidas con argamasa sin un orden establecido.

ideológica y que en ella se plasmaban conceptos mágico-religiosos y se recreaban los mitos de origen, asociados a las estructuras de poder. El gran basamento representa la "montaña de la creación".

La compleja iconografía del friso del edificio y los cuerpos del basamento de esa montaña, ornamentados con grandes mascarones modelados en estuco, representaban imágenes cósmicas que relacionaban a los gobernantes con el mundo de los dioses. Otro rasgo que distingue a la estructura es que en ella se encuentra el ejemplo más antiguo de un edificio abovedado. El recinto cubierto por una bóveda de cañón corrido, única en su género en el área maya, tiene una superficie de 22.68 m2 (8.10 x 2.80 m).

Hacia finales del Preclásico, en la vecina ciudad de Tikal se inició la construcción de bóvedas en saledizo, rasgo distintivo de la arquitectura maya. Estas primitivas bóvedas fueron elaboradas para techar las tumbas de destacados personajes. Hacia la primera mitad del siglo I a. C. se establecieron las bases y los elementos tanto formales como estructurales que van a definir la tradición arquitectónica del área maya, en especial el uso de la bóveda en saledizo. Los ejemplos mejor conocidos hasta la fecha son los palacios abovedados del Grupo H de Uaxactún y los de la Acrópolis Norte de Tikal.

2.1.2. Clásico.

A principios del periodo Clásico tuvo lugar el perfeccionamiento de la escritura y las fechas calendáricas grabadas en estelas, altares y dinteles, y se plasmaron textos en los muros de edificios, en cerámica o sobre otros artefactos. Con ello se inicia el verdadero periodo histórico de la civilización maya. Fue en las Tierras Bajas centrales en donde se perfeccionó el sistema de la cuenta larga y la escritura jeroglífica, que permitieron relatar la historia de estos pueblos, los cuales alcanzaron su esplendor entre 500 y 800 d. C.

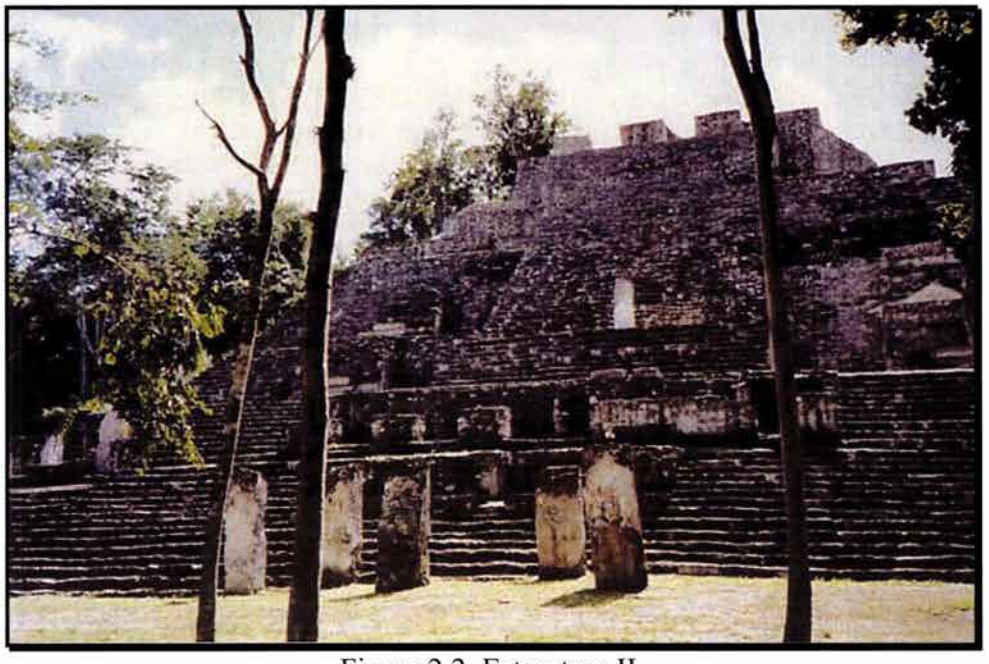


Figura 2.2. Estructura II.

Fechada en 292d.C., la Estela 29 de Tikal es, hasta ahora, el monumento más antiguo que se conoce y el que marca el inicio del Clásico en las Tierras Bajas. Con ella se inaugura el culto individualizado de los gobernantes, para lo cual se combinaban imágenes con fechas de sucesos históricos. El surgimiento de los grandes estados del periodo Clásico se dio durante los primeros siglos de nuestra era. Al inicio de ese periodo, Tikal fue el centro en torno al cual giró la vida política del retén central. Diversos reinos rendían tributo a sus gobernantes, quienes establecieron y consolidaron una de las dinastías más influyentes de su época. En el siglo siguiente, los intereses de la casa real de Tikal entraron en conflicto con los de otros estados, en especial con los del que sería su principal rival por más de dos siglos, Calakmul, capital del *cuchcabal* de la Cabeza de Serpiente.

2.1.3. La cabeza de serpiente.

En sus inicios, las inscripciones de Tikal, como la mencionada Estela 29, sólo hacían referencia a aspectos de carácter cronológico y genealógico. Décadas más tarde, las relaciones de vasallaje de sitios como Bejucal y Motul de San José, ambos en Guatemala, nos indican que fueron conquistados reinos cercanos a esa urbe. En el año 378 d. C., Uaxactún que durante el Preclásico estuvo afiliado a la esfera del norte, interactuando con Calakmul, Nakbé y El Mirador perdió su independencia. En esta fecha, Rana Humeante, hermano de Gran Garra de Jaguar, ahaw de Tikal, lo ocupó y celebró un rito de victoria, acto que por su importancia fue registrado en la Estela 32 de Tikal. Existen claras evidencias de que en su política de expansión Tikal extendió su influencia a reinos tan lejanos como Caracol, en Belice. En este sitio se registró la ascensión del gobernante Y-ahawte en 553 d.C., bajo el patrocinio del gobernante doble pájaro, de Tikal.

Mientras en el sur, Tikal con el ascenso al trono de K´awil Chean en 411, alcanzaba su cúspide, en el norte, el cuchcabal de la cabeza de serpiente se configuró como una gran unidad política, en la cual Calakmul parece representar el centro donde se sancionan las alianzas y se dirimen los conflictos.

De las últimas interpretaciones de los textos epigráficos se desprende que los reyes del periodo clásico maya pusieron en práctica esquemas de organización política que les permitieron crear y mantener una amplia re de alianzas. Estas "unidades políticas" se sustentaron en una diplomacia que puso en práctica tanto las alianzas matrimoniales como el vasallaje. Los dos grandes centros que durante el clásico se disputaron la hegemonía de las tierras bajas centrales fueron Tikal y Calakmul.

En el siglo V Tikal, bajo el gobierno del decimoprimer señor, llamado cielo tempestuoso, cristalizó y consolidó la política de expansión iniciada por Gran Garra de jaguar y su hermano rana humeante, con lo cual dicha ciudad alcanzó su prestigio como centro de poder. Para estas fechas el gobernante de Calakmul emprendió la tercera remodelación de la estructura II, un gran basamento del preclásico con enormes mascarones que enmarcan una monumental escalinata, al pie de la cual se dedicó la Estela 114, en el año 435 d. C.

Estos cambios dieron inicio en Calakmul a grandes obras de remodelación que, sin modificar la traza urbana de la ciudad establecida desde el Preclásico, le proporcionaron una nueva fisonomía.

En el transcurso de este siglo, el *cuchcabal* de la Cabeza de Serpiente empezó a consolidarse y, al igual que Tikal, intensificó su política exterior. Los primeros registros de probables relaciones entre Calakmul y otros centros de poder están documentados en Dzibanché, Quintana Roo, en el año 495 d. C. Pero fue a partir del gobierno de Tun K'ab Hix cuando las alianzas se intensificaron para terminar con la hegemonía de Tikal. Es probable que para estas fechas el sistema de organización política del *cuchcabal* tomara su forma definitiva y haya sido aplicado por Calakmul en su política de expansión.

Este modelo de organización sociopolítica imperó en las Tierras Bajas mayas del norte, y aún era practicado por los grupos que poblaban la península a la llegada de los españoles. Estaba formado por un conjunto de gobernantes de pueblos subordinados que mantenían una interrelación compleja de índole político-religiosa y estaban enlazados por el poder que residía en un "pueblo cabecera". Un *cuchcabal* no tenía una frontera establecida y su trazo no era lineal; asimismo, su jurisdicción abarcaba varios pueblos, los cuales tampoco tenían un límite definido ni una delimitación territorial. Así, en el sistema de organización del *cuchcabal*, un gobernante podía aceptar en su jurisdicción al dirigente de un pueblo lejano sin preocuparse por la continuidad territorial.

Las características del "territorio" del *cuchcabal* dependían del principio de vínculo entre personas, que estaba determinado por la relación gobernante gobernados y en el que no se concebía la propiedad privada de la tierra. Si el gobernante de un Estado estaba subordinado a un *ahaw*, la tierra administrada por el primero también era del segundo, y ésta era la manera en la que el gobernante de un *cuchcabal* determinaba su territorio. El territorio de un pueblo era un conjunto de tierras cultivadas u ocupadas por los que aceptaban el mando de un *ahaw*. En el concepto maya respecto al territorio, el vínculo político, económico, social y religioso entre gobernante y gobernados tenía un valor trascendental. En este contexto, la estructura jurídica del cuchcabal estaba regida por el asentamiento principal o centro rector donde residía el ahaw.

2.2. Calakmul en números

-Extensión del sitio arqueológico de Calakmul: 25 Kilómetros cuadrados.
-Elementos arqueológicos registrados: 6000.
-Número aproximado de habitantes de la ciudad en el clásico tardío: 50000.
-Cantidad aproximada de acrópolis de gran tamaño construidas en ese entonces: 10.
-Tumbas excavadas o localizadas en la estructura II: 10.
-Máscaras funerarias fabricadas con mosaicos de jade que se han encontrado en Clakmul: 9.
-Cantidad de estelas todas correspondientes al Clásico encontradas hasta ahora en Calakmul: 120.
-Fecha de la estela 114, la más antigua registrada en Calakmul: 435 d.C.
-Superficie que abarca la reserva de la biosfera de Clakmul: 723185ha.
-Especies de marsupiales que habitan en la reserva de la biosfera de Calakmul: 6.
-Especies de felinos que habitan en la reserva de la biosfera de Calakmul: 5.
-Especies de aves que habitan en Calakmul: 338.

-Especies de anfibios que habitan en la reserva de la biosfera de Calakmul: 18.
-Especies de reptiles que habitan en la reserva de la biosfera de Calakmul: 75.

2.3. Calakmul reserva de la biosfera

Conocer la reserva de la biosfera de Calakmul es sin duda una experiencia fascinante, ya que se trata de un área de singular belleza, con una gran riqueza natural y cultural, que además tiene una magia muy especial. Asimismo, se puede apreciar su magnificencia desde las alturas, tanto en un sobrevuelo como desde la plataforma superior de una pirámide de la gran cultura maya.

2.3.1. Un área natural protegida.

La reserva de la biosfera de Calakmul, Campeche, en el sureste mexicano, es la segunda área natural protegida terrestre más grande de México, con una extensión de 723 185 ha. Las áreas naturales protegidas por un decreto federal que norma su uso tienen como finalidad la conservación de sus recursos y ambientes naturales, y se crean para asegurar el aprovechamiento sustentable de los ecosistemas y sus elementos. Además, son un campo propicio para la investigación y el estudio de los ecosistemas y su equilibrio. En estas áreas se busca generar, rescatar y divulgar el conocimiento de las prácticas y tecnologías tradicionales o nuevas que permitan su preservación y el aprovechamiento de la biodiversidad. En algunos casos, como en el de Calakmul, las áreas naturales protegidas contribuyen a resguardar los entorno s naturales de zonas, monumentos y vestigios arqueológicos, históricos y artísticos importantes para la recreación, la cultura e identidad nacional y de los pueblos indígenas.

Las reservas de la biosfera se crean en áreas biogeográficas relevantes a nivel nacional, que son representativas de uno o más ecosistemas no alterados significativamente y que requieren ser preservados y restaurados. En ellos habitan especies representativas de la biodiversidad nacional, en especial las endémicas, es decir, que sólo se pueden encontrar en ese sitio, ya sea amenazadas o en peligro de extinción.

La reserva de la biosfera de Calakmul, se encuentra en el recién creado municipio de Calakmul, en la parte sureste de Campeche, y colinda, al este, con el estado de Quintana Roo, al sur, con el Petén guatemalteco, al oeste, con los municipios de Candelaria y Champotón, y al norte, con el municipio de Hopechén. Cabe mencionar que Calakmul está considerado como el primer "municipio ecológico" del país, ya que la reserva ocupa 47% del municipio.

En 1986, la entonces Secretaría de Desarrollo Urbano y Ecología recibió una propuesta del Dr. William Fojan para crear un área natural protegida en la región de Calakmul. En el verano de ese mismo año la secretaría visito el área por primera vez.

La reserva de la biosfera de Calakmul fue creada por un decreto en 1989. Tiene dos zonas "núcleo", una en el norte, de 147915 ha, y otra en el sur, con 100345 ha. En las zonas núcleo, consideradas como las mejor conservadas de la reserva, se fomenta la conservación,

la investigación y la educación ecológica, y se limitan o prohíben los usos que alteran los ecosistemas. La zona de amortiguamiento es de 474924 ha; este tipo de zonas, como se indica en la Ley General del Equilibrio Ecológico y Protección al Ambiente, se crean para proteger las zonas núcleo del impacto exterior y en ellas se realizan actividades productivas emprendidas por las comunidades que ahí habitan, las cuales son compatibles con los objetivos, criterios y programas de aprovechamiento sustentable en los términos establecidos tanto en su derecho como en su programa de manejo.

Calakmul es una de las regiones del territorio nacional con gran riqueza de especies, tanto de plantas como de animales. Por su parte, la reserva de la biosfera de Calakmul es uno de los tesoros naturales y culturales más grandes de México, y se distingue por ser en Norteamérica la que tiene la población más importante de felinos, algunos de los cuales son especies amenazadas.

Figura 2.3. El jaguar es una de las especies que habita Calakmul.

La vegetación es también abundante y variada. Las lluvias temporales crean un clima propicio para algunas especies de plantas arbóreas, entre las que se encuentran epífitas como las bromelias y las orquídeas. El agua se acumula en extensiones de terrenos bajos y anegadizos, característicos de la región, denominados en maya akalchés.

2.4. Proyecto visita virtual a Calakmul

Como se puede observar [2.1] Calakmul representa un sitio arqueológico de gran importancia en la historia de la cultura maya y si embargo aun hoy en día resulta ser uno de los más desconocidos en el ámbito nacional, estando por encima de este lugares de mucho más renombre como Palenque, monte Alban, Teotihuacan, etc.

Esto se debe en parte a que fue recientemente descubierto 1980 y apenas se está estudiando por lo que no se le ha hecho mucha difusión, Calakmul, declarado Patrimonio de la Humanidad por UNESCO en Junio 27 del 2002. está situado en medio de una Reserva Nacional protegida [2.3], Calakmul está destinado a permanecer parcialmente cubierto por

la selva. Sus inmensas estructuras de 50 metros de altura, se alzan por encima del nivel de los árboles tropicales y sólo subiendo hasta sus cimas pueden verse las demás pirámides. La lejanía de este sitio arqueológico próximo a la frontera con Guatemala, no ha permitido que sean exhibidas ninguna de sus invaluables máscaras y piezas arqueológicas encontradas en las 17 tumbas descubiertas hasta el momento.

De ahí que Calakmul sea el sitio arqueológico idóneo para desarrollar un proyecto en el cual se propone el uso de la Realidad Virtual como medio de exhibición y difusión de nuestro patrimonio cultural tanto para su uso en la comunidad científica como para ser mostrado al público general.

El resultado de este proyecto permitirá la mejor comprensión de las características de la antigua civilización maya, se busca proporcionar una herramienta de estudio útil a los arqueólogos e investigadores y favorecer el turismo a un destino maya poco visitado y publicitado a la fecha.

La creación de un entrono virtual representa una manera efectiva de conocer este sitio arqueológico puesto que en la realidad únicamente pueden verse el resto de las pirámides de más de 50 metros de altura, observar el entorno por encima de las copas de los árboles. Entre algunas otras cosas se pretende que este entorno tenga vida artificial, así como un agente inteligente que, personificado por un arqueólogo, ayude al visitante a recorrer el sitio. Todo esto con la finalidad de que el recorrido sea más interactivo e interesante.

Otra característica por la que Calakmul puede beneficiarse de una reproducción virtual, es la imposibilidad que se tiene para exhibir en el sitio las invaluables piezas arqueológicas ahí encontradas (máscaras, utensilios, cerámica). Mediante una simulación del sitio arqueológico el usuario podría no sólo observar las piezas sino también obtener información y datos arqueológicos de cada una de las piezas ahí encontradas.

2.4.1. Análisis de requisitos.

El análisis de requisitos es una tarea de ingeniería de software que cubre el hueco entre la definición del software a nivel sistema y el diseño de software.

El enfoque que se le ha dado a esta aplicación es buscar que el resultado final pueda ser fácilmente acondicionado para su uso principalmente en museos donde la aplicación estará en algún tipo de exhibidor, por lo que el usuario principal del software será el público en general, teniendo en consideración que este tipo de usuario puede estar o no familiarizado con el uso de equipo de computo, que las edades de los usuarios van desde niños hasta adultos mayores y que el trato y manejo que se le puede dar a la aplicación pude no ser el más adecuado. Por otra parte la implementación de la solución queda bastante abierta permitiendo libertad de incluir funcionalidades extras siempre y cuando estas cumplan con los requisitos básicos del software.

Una buena salida o resultado de la aplicación es que ésta ofrezca un despliegue tridimensional del lugar o sitio deseado, en el cual se le permita al usuario moverse a través de éste ofreciéndole la experiencia de estar en el sitio de interés, además que se le permita

interactuar con objetos y personas que se encuentren en el lugar, todo esto con la finalidad de que el usuario aprenda de la experiencia de usar el software. De manera general se debe buscar que la implementación en software funcione para varios tipos de recorridos virtuales y no esté limitado a un recorrido fijo en un solo lugar, que tanto el costo económico como el tiempo para cambiar el sitio o zona arqueológica del recorrido no sean muy elevados ya que para una aplicación orientada a ser usada en un museo la rotación y cambios que se le puedan hacer a la exhibición son muy importantes.

El entorno en el que principalmente será usada la aplicación como ya se menciono es el encontrado en un museo, donde además de considerar el tipo de usuario, el cual resulta ser muy variado, hay que tomar en cuenta cuestiones como el espacio ya que en un museo no se puede tener todo el espacio que se desea puesto que existen muchas otras exhibiciones en el mismo lugar, a demás hay que tener en consideración la relación del número de personas que pueden hacer uso de la aplicación (exhibición) con el tiempo que a estas les tome hacer uso de la aplicación, a estas restricciones también hay que añadir restricciones del tipo económico, que el equipo físico (hardware) para correr la aplicación sea lo más barato posible, el costo de desarrollar la aplicación así como el tiempo entre algunos otros.

2.4.2. Especificaciones de la aplicación.

-El equipo de computo necesario para correr la aplicación debe ser lo más estándar posible, es decir de preferencia debe correr en una computadora personal.

-La interfaz de manejo para el usuario debe ser sencilla, un control para moverse en el ambiente virtual y un botón para interactuar con el entorno.

-La aplicación debe permitir que el cambio del contenido sea sencillo y rápido.

-El usuario debe tener la libertad de poder recorrer el mundo como a él le parezca sin que esté restringido a seguir caminos o rutas preestablecidas.

-La aplicación deberá de proporcionar información sobre los objetos que se encuentren en el recorrido.

-La aplicación debe no sólo dar un recorrido por un mapa, el mapa debe contener personajes los cuales deben tener animaciones y diálogos para interactuar un el usuario.

-La comunicación, los personajes dentro del recorrido virtual deben de ser capaces de comunicarse por medio de audio con el usuario, tratando de evitar en lo posible la lectura del texto.

-El recorrido virtual debe de tener la presencia de un llamado guía de turistas cuyas funciones principales son las de dar información y guiar al usuario a través del mundo virtual.

-El costo por desarrollar la aplicación debe ser reducido.

-La aplicación no sólo debe ser capaz de representar las estructuras del sitio arqueológico, en este caso Calakmul, sino que también debe ser capaz de poder representar cosas como la vegetación del lugar.

-La representación gráfica debe ser de una buena calidad incluyendo efectos como bump-maping[3], lensflare[4], etc.

---

[3] Técnica usada para añadir más detalle a una imagen sin incrementar el número de polígonos, esta técnica se basa principalmente en el cálculo de los reflejos de la luz pare crear un relieve en la superficie de un objeto, con el objetivo de que la textura de este se vea más realista.

-El software resultante debe de poder ser modificable por los programadores en caso de ser necesario cambiar o incluir alguna funcionalidad.

2.4.3. Dominio de la información.

Todas las aplicaciones de software pueden denominarse colectivamente como un procesamiento de datos, el software se construye para procesar datos, para transformar datos de una forma a otra, para aceptar la entrada de información, manipularla de alguna manera y producir una salida de información.

El primer principio operativo de análisis requiere el examen del dominio de la información. Este dominio contiene tres visiones diferentes de los datos y el control a medida que se procesa cada uno en un programa de computadora.

El contenido de la información representa los objetos individuales de datos y de control que componen alguna colección mayor de información a la que transforma el software, a continuación se hace el análisis de los objetos de datos y de control que la aplicación Calakmul virtual usará.

Objetos de datos.

Objeto personaje, información que maneja: La entrada del teclado, del ratón o del periférico con el que se pretenda controlar al personaje, la malla y texturas para dibujar al personaje en pantalla, información sobre su posición en el mundo virtual, el estado en el que se encuentra el personaje (animación que está realizando, colisiones, etc), sonidos que el personaje debe emitir (diálogos, pisadas, etc) y datos para interactuar con el entorno (malla para colisión, velocidad al moverse, aceleración, etc).

Objeto personaje controlado por el sistema, información que maneja: En algunos casos la entrada del periférico con el que se pretenda controlar al personaje ya que por este medio el usuario puede tener interacción con los personajes controlados por el sistema, la malla y texturas para dibujar en pantalla, información sobre su posición en el mundo virtual, el estado en el que se encuentra (animación que está realizando, colisiones, etc), datos para interactuar con el entorno (malla para colisión, velocidad al moverse, aceleración, etc), sonidos que el personaje debe emitir (diálogos, pisadas, etc) y la información sobre los caminos por los cuales el personaje controlado por el sistema puede transitar.

Objeto del mundo, información que maneja: la malla y texturas para dibujar al objeto en pantalla, en algunos casos la entrada del periférico con el que se pretenda controlar al personaje ya que por este medio el usuario puede tener interacción con los objetos en el mundo virtual, información sobre su posición en el mundo virtual, la malla de colisiones, el estado en el que se encuentra (principalmente colisiones) y sonidos que el objeto debe emitir (principalmente sonidos ambientales).

---

[4] Termino en ingles que se refiere al reflejo que se produce en una cámara cuando la cámara esta apuntando directamente al sol.

Objeto estructura, información que maneja: la malla y texturas para dibujar la estructura en pantalla, información sobre su posición en el mundo virtual, la malla de colisiones, el estado en el que se encuentra (principalmente colisiones),

Objeto cámara, información que maneja: Información de configuración de la cámara (distancia de dibujo, FOV, resolución, etc), su posición en el mundo virtual, la malla de colisiones e información a cerca de que objeto se encuentra observando.

Objeto zona de colisión, información que maneja: La malla de colisiones y si dicho objeto está haciendo colisión con algún otro objeto del mundo virtual.

Objetos de control.
Control del despliegue en pantalla, función principal: Encargarse de dibujar en pantalla todos los objetos que así lo requieran, esto implica dibujar las mallas, texturas y sombreados de la escena.

Control de la reproducción de sonidos, función principal: Encargarse de controlar y reproducir los sonidos de todos los objetos que así lo requieran así como realizar los cálculos de distancias a las cuales los sonidos son percibidos.

Control de colisiones, función principal: Informar cuando es que a ocurrido una colisión y entre que objetos ocurrió esto, para así poder programar los comportamientos deseados entre los objetos del mundo virtual.

Control de entradas al sistema, función principal: Informar al sistema sobre las entradas que ocurran de los dispositivos periféricos para así poder asociar eventos o comportamientos a estas entradas.

El flujo de la información representa como cambian los datos y el control a medida que se mueven dentro de un sistema, los objetos de entrada se transforman para intercambiar información, hasta que se transforman en información de salida.

Ahora se presenta un diagrama en el que se pretende explicar el flujo de la información en el sistema Calakmul virtual Con el diagrama del flujo de la información sólo se intenta hacer más fácil entender las relaciones entre los objetos de datos y los objetos de control así como las transformaciones que sufren los datos, el diagrama no tiene nada que ver con el verdadero flujo de programa para la aplicación final.

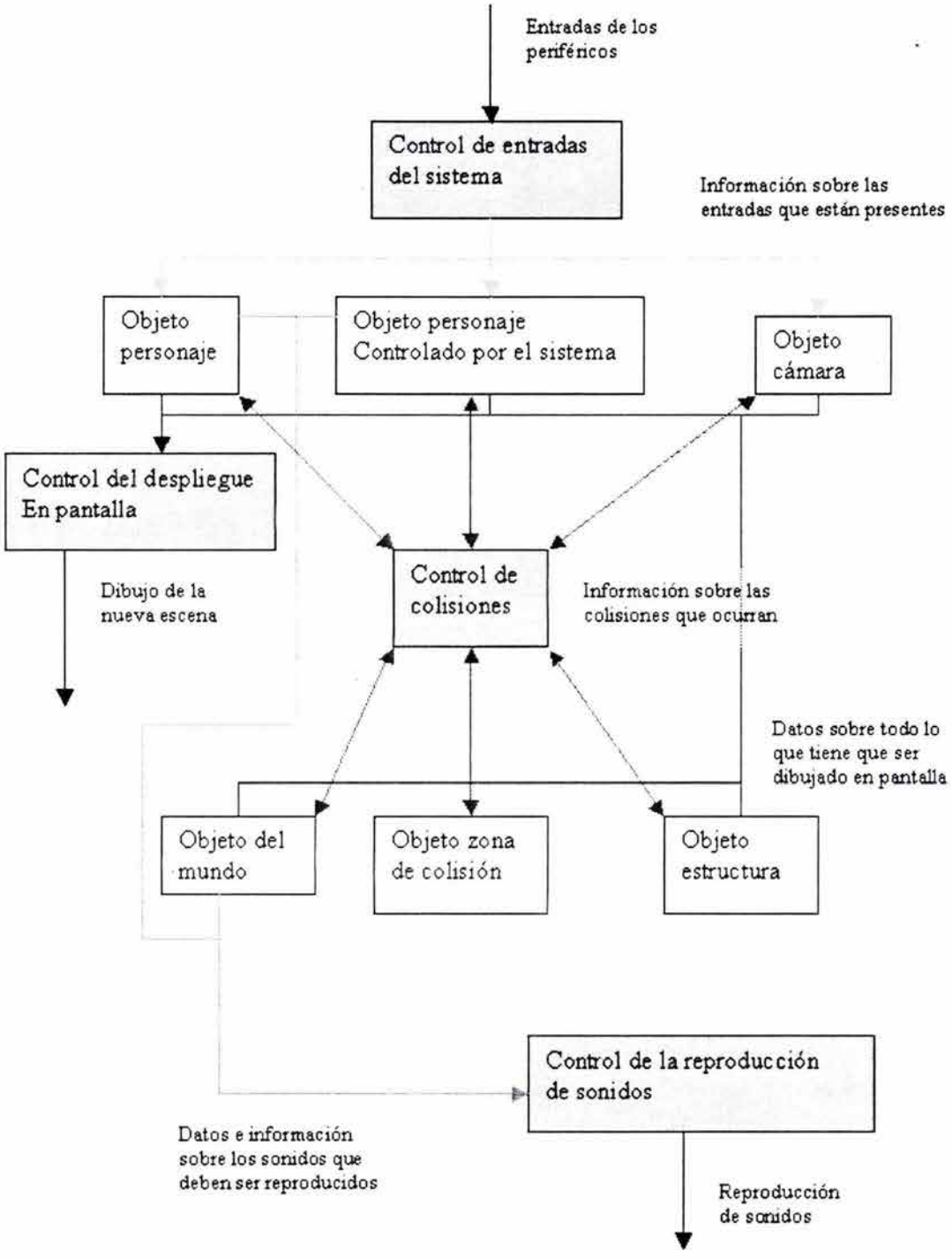


Figura 2.4. Flujo de la información.

Se observa, por ejemplo, que todos aquellos objetos de datos que necesitan ser presentados en pantalla le dan la información necesaria al control de despliegue en pantalla para ser dibujados, lo mismo pasa con todos aquellos objetos que emiten algún sonido, un caso interesante es el del control de colisiones ya que todos los objetos de datos que necesitan tener conocimiento de las colisiones que se realizan con otros objetos le dan la

información al control de colisiones, principalmente su posición y su malla de colisiones, para que éste les informe si hay colisión y con que objeto se produjo esta colisión.

2.4.4. Modelado.

La mayoría del software responde a los acontecimientos del mundo exterior. Esta característica estímulo-respuesta forma la base del modelo del comportamiento. Un programa de computadora siempre está en un estado, un modo de comportamiento observable exteriormente, que cambia sólo cuando ocurre algún acontecimiento.

Para el proyecto de Calakmul virtual que se está analizando, el comportamiento principalmente recae en los objetos que interactúan en el mundo virtual y por medio de estos es que el estado del programa cambia.

En la siguiente tabla se presentan los acontecimientos que afectan a un personaje y el comportamiento que éste debe realizar ante dicho acontecimiento.

Acontecimiento	Comportamiento
Entrada avanzar.	El personaje se mueve hacia delante, con respecto a su posición actual y hacia donde esté observado.
Entrada retroceder.	El personaje se mueve hacia atrás, con respecto a su posición actual y en sentido contrario de donde esté observado.
Entrada avance lateral.	El personaje se mueve lateralmente, izquierda o derecha, con respecto a su posición actual y en un ángulo de noventa grados de donde esté observando.
Entrada mover vista.	El personaje gira cuando se indica un giro sobre el eje Y, cuando se indica sobre el eje X éste es ignorado.
Colisión con un personaje controlado por el sistema.	El personaje realiza una acción específica dependiendo del personaje con el que hubo colisión o corre un script donde se le indican las acciones a realizar, se le impide al personaje continuar con su movimiento.
Colisión con un objeto del mundo virtual.	El personaje realiza una acción específica dependiendo del objeto con el que hubo colisión o corre un script donde se le indican las acciones a realizar, se le impide al personaje continuar con su movimiento.

Colisión con una estructura del mundo virtual.	El personaje realiza una acción específica dependiendo de la estructura con la que hubo colisión o corre un script donde se le indican las acciones a realizar, se le impide al personaje continuar con su movimiento.
Colisión con una zona de colisión.	El personaje realiza una acción específica dependiendo de la zona con la que hubo colisión o corre un script donde se le indican las acciones a realizar.

Tabla 2.1. Comportamiento del personaje controlado por el usuario.

En la siguiente tabla se presentan los acontecimientos que afectan a la cámara y el comportamiento que ésta debe realizar ante dicho acontecimiento.

Acontecimiento	Comportamiento
Entrada avanzar.	La cámara debe de avanzar junto con el personaje.
Entrada retroceder.	La cámara debe de retroceder junto con el personaje.
Entrada avance lateral.	La cámara se debe mover lateralmente junto con el personaje.
Entrada mover vista.	La cámara gira cuando se indica sobre el eje Y o el eje X.
Entrada de interacción.	La cámara debe de revisar a que objeto está apuntando (objeto, estructura, personaje controlado por el sistema, etc), y en base a esto realizar una acción específica o correr un script en donde se indican las acciones a realizar.
Colisión con un personaje controlado por el sistema.	La cámara realiza una acción específica dependiendo del personaje con el que hubo colisión o se corre un script donde se indican las acciones a realizar.
Colisión con un objeto del mundo virtual.	La cámara realiza una acción específica dependiendo del objeto con el que hubo

	colisión o se corre un script donde se indican las acciones a realizar.
Colisión con una estructura del mundo virtual.	La cámara realiza una acción específica dependiendo de la estructura con la que hubo colisión o se corre un script donde se indican las acciones a realizar.

Tabla 2.2. Comportamiento de la cámara.

En la siguiente tabla se presentan los acontecimientos que afectan a los personajes controlados por el sistema y el comportamiento que estos deben tener ante dicho acontecimiento.

Acontecimiento	Comportamiento
Entrada de interacción.	El personaje controlado por el sistema debe de revisar si está siendo apuntado por la cámara y en base a esto realizar una acción específica o correr un script en donde se indican las acciones a realizar.
Colisión con la cámara.	El personaje controlado por el sistema realiza una acción específica o se corre un script donde se indican las acciones a realizar.
Colisión con el personaje.	El personaje controlado por el sistema realiza una acción específica o se corre un script donde se indican las acciones a realizar.
Colisión con una zona de colisión.	El personaje controlado por l sistema realiza una acción específica dependiendo de la zona con la que hubo colisión o corre un script donde se le indican las acciones a realizar.

Tabla 2.3. Comportamiento de los personajes controlados por el sistema.

En la siguiente tabla se presentan los acontecimientos que afectan a los objetos del mundo virtual y el comportamiento que estos deben tener ante dicho acontecimiento.

Acontecimiento	Comportamiento
Entrada de interacción.	El objeto del mundo virtual debe de revisar si está siendo apuntado por la cámara y en base a esto realizar una acción específica o correr un script en donde se indican las acciones a realizar.
Colisión con la cámara.	El objeto del mundo virtual realiza una acción específica o se corre un script donde se indican las acciones a realizar.
Colisión con el personaje.	El objeto del mundo virtual realiza una acción específica o se corre un script donde se indican las acciones a realizar.

Tabla 2.4. Comportamiento de los objetos.

Esta tabla también es aplicable para las estructuras y para las zonas de colisión, en el caso de las zonas de colisión no existe el acontecimiento de entrada de interacción.

2.5. Algunas soluciones que ofrece el mercado

2.5.1. Recorridos virtuales con aplicaciones basadas en Internet.

Existe una gran cantidad de aplicaciones cuyo objetivo es dar un recorrido virtual por zonas arqueológicas, museos, monumentos, etc. mediante el uso de un navegador de Internet, como el Internet Explorer, a través de la red, la idea principal es que cualquier persona con acceso a una conexión a Internet se pueda conectar y visitar las páginas que ofrecen los recorridos por diferentes lugares de interés.

Este tipo de proyectos son muy variados en su presentación e implementación, por ejemplo muchos sólo tienen un enfoque del diseño gráfico ya que para su realización toman aplicaciones comerciales que permiten cargar el ambiente diseñado, dichas aplicaciones funcionan en su mayoría como plug-ins[5] del navegador y a través del lenguaje conocido como VRML[6] este lenguaje permite la creación de ambientes tridimensionales y

[5] Módulo de hardware o software que agrega una característica o servicio a un sistema más grande.

[6] VRML es un lenguaje textual que describe escenas 3D. Difiere de lenguajes de programación como C, en que en vez de describir como debe actuar el ordenador a través de series de comandos específicos del lenguaje, describe como una escena 3D debería aparecer. Como en HTML, estos puntos son requerimientos absolutos para un lenguaje estándar de la red. Muy pronto los diseñadores decidieron que VRML no fuera una extensión de HTML, el cual esta diseñado para texto, no para gráficos. También VRML requiere incluso más refinamiento que HTML. Se espera que una escena típica de VRML estará compuesta por más objetos y será

ofrece al usuario una experiencia de inmersión en tiempo real en la cual puede navegar libremente por el mundo tridimensional, esto ayuda a que el usuario tenga una idea espacial del lugar que está recorriendo virtualmente, aunque en general este tipo de recorridos no están exentos de algunos problemas muy bien ubicados. Algunos de estos problemas que comparten este tipo de aplicaciones son la limitada interactividad que ofrecen para con el usuario, ya que en su mayoría el único tipo de interactividad que ofrecen es el poder moverse libremente en el entorno otro problema fundamental es la representación de gráficos, es decir estos tienen que ser muy sencillos teniendo en cuanta las representaciones que hoy en día es posible lograr, esto principalmente se debe a que por su naturaleza el usuario tiene que recibir toda la información por medio de la red, aun así teniendo esto en consideración hay que resaltar que estas páginas son lentas en descargar, dicha limitación se refleja en una reducción en la inmersión que puede experimentar el usuario, este problema aunado al anteriormente mencionado provocan que la experiencia en su conjunto deje mucho que desear.

Figura 2.5. Escena hecha con VRML

Algunos proyectos similares a los expuestos en este punto son los implementados por medio de fotografías panorámicas[7], estas al ser fotografías del lugar permiten un mayor detalle y un mayor realismo aparentemente, el problema fundamental es que estas aplicaciones sólo permiten la libertad de la vista de 360 grados pero todos los demás movimientos son restringidos, por lo que el usuario no puede avanzar, retroceder o moverse en cualquier otra dirección, además la interacción de cualquier otro tipo con el entorno es prácticamente nula.

---

presentado por muchos más servidores que un típico documento HTML. HTML es un estándar aceptado, con implementaciones existentes que dependen de él.
Para impedir que el proceso de diseño de HTML y el de VRML se mezclen ambos lenguajes deberían hacerse por separado, de forma que como lenguaje de red, VRML tendrá éxito o fallara independientemente de HTML.

[7] Las llamadas fotografías panorámicas consisten en tomar una serie de fotos en un mismo punto variando el ángulo de estas hasta completar los 360 grados, para posteriormente ser integradas en una sola imagen.

2.5.2. Recorridos virtuales predefinidos.

Otro enfoque que se le ha dado a los recorridos virtuales, principalmente en los museos, es el de hacer recorridos ya predefinidos y presentarlos en un teatro a un auditorio, con un narrador que explica y da datos históricos sobre lo que el público está observando, como estos recorridos al momento de exhibirse no son creados en tiempo real permiten que el nivel de detalle sea muy grande debido a que los cuadros fueron dibujados con herramientas profesionales para el modelado de escenas en tres dimensiones permitiéndole a los diseñadores no sólo representar lugares existentes sino recrear los sitios en su forma original con vegetación y población, así pues estos recorridos ofrecen lo más avanzado en cuanto a gráficas por computadora se refiere, gráficas que hoy en día son imposibles de presentar en tiempo real.

Figura 2.6. Escena 3D creada para un video.

Si bien este tipo de implementación permite una mayor inmersión debido a que las gráficas son muy realistas no está exenta de algunos problemas, un problema insalvable que presentan es la interactividad, como las gráficas que se pretenden presentar son de muy alta calidad la única manera de presentarlas es en un formato de video, teniendo que haber sido previamente procesadas y dibujadas cuadro por cuadro en sistemas de computo especializados, así pues en un formato de video la interacción es nula, uno sólo puede ver y conocer lo que se le indica y explica, si se quería tener otra vista de algún objeto o ver más detenidamente cierto monumento es imposible puesto que el recorrido ya se diseño y no puede ser cambiado sin prácticamente tener que hacer todo de nuevo, lo que implica nuevos costos y tiempo, otro problema que se presenta es que todas estas implementaciones son pensadas para un grupo de personas muy heterogéneo, al ser presentadas en un formato tipo teatro no pueden dar demasiados detalles sobre ciertos aspectos, de lo contrario podría parecer aburrido a fastidioso para algunas personas, de tener un formato más flexible y personal el usuario realizaría una visita de acuerdo con sus gustos personales.

2.5.3. Recorridos virtuales experimentales.

En este punto se busca englobar un grupo muy variado de aplicaciones y proyectos que como su nombre lo indica como único punto en común es que fueron o están siendo desarrolladas con el objetivo de innovar e incorporar nuevas tecnologías en el campo de las visitas a lugares de manera virtual.

Estos proyectos buscan incorporar tecnologías como la realidad aumentada[8], reconocimiento de patrones[9], inteligencia artificial, trajes de realidad virtual, reconocimiento de voz , cuevas[10], etc. Comentar acerca de todos estos proyectos es muy complicado ya que son muy variados y con objetivos muy diferentes, así pues para algunos de estos proyectos lo principal es la inmersión, como el caso del uso de cuevas, y crear ambientes donde las persones se sientan verdaderamente dentro de estos, para otros el enfoque es la interacción de las personas con el ambiente, por medio de agentes inteligentes que respondan a acciones de las personas.

El único problema de estos proyectos que se puede vislumbrar de manera genérica sin hacer un análisis a fondo de cada uno de ellos, análisis que no pretende ser cubierto en este punto, es que la tecnología con la que se está trabajando en mucho de los casos puede resultar cara, por ejemplo adquirir un equipo de lentes de realidad virtual resulta en una inversión aproximada de 200 dólares, costo que debe de agregarse al del equipo de computo que se necesita para que los lentes funcionen ya que los lentes solamente son un dispositivo de despliegue.

Costos que de no tomarse en cuenta pueden resultar en una implementación poco viable. De esto se deduce que para poder implementar algunos de estos proyectos sea necesaria una considerable inversión.

2.6. Elección del motor de Juegos Nebula

Una vez que se ha definido el objetivo fundamental de la aplicación Calakmul virtual, así como sus requerimientos, funciones y comportamientos básicos se puede hacer un análisis de acerca de cual es la mejor manera de implementar el software en cuestión.

---

[8] Realidad aumentada Intenta mejorar la percepción del mundo real agregando información gráfica y textual. Es la combinación de gráficos en 3D y texto sobre puesto a imágenes y video reales, en tiempo real. La RA puede utilizar los mismos dispositivos utilizados en ambientes virtuales.

[9] Campo de estudio de la computación, de como un patrón debe ser percibido por los órganos sensoriales. Además, el mismo patrón o alguno similar (de la misma clase) debe haberse percibido y recordado previamente. Finalmente, debe establecerse alguna correspondencia entre la percepción actual y lo recordado.

[10] Cuevas: en este caso termino que se refiere a un dispositivo de despliegue el cual consiste en un espacio tridimensional que envuelve a la persona con el objetivo de darle la sensación de encontrarse en el lugar que se proyecta en la cueva.

2.6.1. ¿Por qué un motor de juegos?

Al hacer el estudio de algunas de las soluciones que hay en el mercado [2.5] similares a la que se piensa construir y hacer una comparación con los requisitos que se tienen para la aplicación es fácil observar que en casos como los recorridos con aplicaciones basadas en Internet, si bien permiten al usuario navegar con cierta libertad por el mundo virtual estas están muy lejos de ofrecer la interacción que se pretende alcanzar con el ambiente, los personajes y los objetos que se encuentren en éste.

En el caso de los recorridos predefinidos la solución que ofrecen está todavía más alejada de la que se pretende obtener, ya que no sólo no ofrecen el nivel de interacción deseado sino que además no permiten al usuario navegar libremente por el mundo virtual, lo que ofrecen estas aplicaciones en su mayoría es un alto nivel de realismo.

De acuerdo con los requisitos en los que básicamente se solicitan altos niveles de interactividad con el usuario, representaciones gráficas lo más realistas posibles para sistemas en tiempo real y un bajo costo tanto en su producción como en el equipo necesario para su ejecución la solución más adecuada es la utilización de un motor de juegos, ya que estos le ofrecen a los programadores no sólo las herramientas necesarias para realizar aplicaciones con un despliegue gráfico de objetos tridimensionales sino que estos ofrecen un conjunto extra de herramientas con las cuales se hace más fácil la tarea de programar aplicaciones que hacen uso de entradas de los periféricos, reproducción de archivos de audio, el calculo de colisiones, la reproducción de animaciones y hasta en algunos casos la ejecución de scripts.

2.6.2 ¿Por qué el motor de juegos Nebula?

Una vez que se a determinado que un motor de juegos ofrece quizá la solución en software más adecuada al problema es necesario hacer otra elección. Programar un motor de juegos para posteriormente hacer la aplicación de Calakmul virtual en base a este motor presenta algunas ventajas ya que al ser un motor construido no es necesario pagar cuestiones como las licencias o la compra de un motor es decir uno puede hacer prácticamente lo que quiera con él, por otra parte el hacer un motor de juegos es bastante complicado y requiere de mucho tiempo, hacer un motor de juegos tan sólo ameritaría la presentación de un trabajo como este así que por cuestiones de tiempo y de cantidad de trabajo esta solución no es vista como factible en este momento.

Hoy en día existen una gran cantidad de motores de juego en el mercado, ¿Cuál escoger?, ¿Cuál es aquel que se adecua mejor a nuestras necesidades?, para poder contestar satisfactoriamente a estas preguntas y hacer una elección correcta es necesario hacer un análisis de algunas de las opciones que se tienen en cuanto a motores de juegos.

Comercialmente la variedad de motores de juegos es muy grande y variada, en este ámbito encontrar algún motor de juegos que responda a las necesidades que se tienen no sería difícil, el problema fundamental con estos es que en su gran mayoría para poder hacer uso de estos es necesario pagar una licencia, y estas representan un costo muy elevado, algunos motores se pueden usar sin fines de lucro pero el problema que surge es que de esta manera no se tiene acceso al código fuente del motor y entonces se está limitado a las funciones ya programadas y compiladas que originalmente tiene el motor, si el motor no

cumple con alguna de las especificaciones o requisitos que se piden es imposible agrégasela.

Por otra parte existen las llamadas aplicaciones de código abierto[11] (open sourcce), las cuales no tienen costo alguno y además se tiene el código fuente con el que se le pueden hacer los cambios necesarios al motor para que éste se comporte como lo deseamos. En este punto se analizaron y observaron varios motores de juego de código abierto, de este análisis se descartaron varios motores por motivos como:

-Falta de mantenimiento, por lo que se habían quedado en un retraso con respecto a los avances que ha habido en el campo de la graficación por computadora.
-Estar programados de manera estructurada, lo cual no es un problema en si mismo pero en el caso de los motores de juegos es mucho más recomendable que estos sean orientados a objetos debido a que son más fáciles de entender y modificar
-La falta de varios de los requisitos y especificaciones deseadas, es posible programar algunas de estas deficiencias del motor ya que se tiene el código fuente, pero si estas son tan grandes como la falta de un módulo por el cual se controlen las entradas de los periféricos o la falta un módulo con el que se puedan reproducir archivos de audio, implica una gran cantidad de código que tiene que ser programado.
-La falta del soporte de un lenguaje de scripts, debido a que en las aplicaciones en las que se pretende facilitar la tarea de modificar contenidos y comportamientos sin tener que modificar el código fuente necesitan de un lenguaje de scripts por medio del cual se pueda modificar todo esto.

El análisis fue hecho a cinco motores de código abierto, explicar las deficiencias que mostró cada uno de estos motores llevaría mucho espacio por que se ha decidido sólo comentarlo de manera general. Una vez hecho este análisis se eligió el motor de juegos más apropiado para Calakmul virtual, el motor de juegos Nebula cumple de la mejor manera con los requisitos y especificaciones surgidas del análisis de requerimientos.

[11] Las aplicaciones de código abierto (open source) son aquellas en las que su creador además de ofrecer la aplicación sin ningún costo para el usuario también ofrece el código fuente de su aplicación para que este sea mejorado o adecuada a las necesidades particulares del usuario.

Capítulo 3
Análisis orientado a objetos

3.1. Análisis orientado a objeto

El objetivo del análisis orientado a objetos es desarrollar una serie de modelos que describan el sistema al trabajar para satisfacer el conjunto de requisitos previamente definidos. El modelo de análisis ilustra información, funcionamiento y comportamiento.

3.2. Análisis del dominio

El análisis en sistemas orientados a objetos puede ocurrir a muchos niveles diferentes de abstracción. Al nivel de las aplicaciones el modelo de objetos se centra en los requisitos específicos pues estos afectan la aplicación que se va a construir.

3.2.1. El proceso de análisis del dominio.

En este punto ya se analizó parte del dominio, puesto que se definió el tipo de sistema y el área de trabajo en el que la aplicación va a estar basada [2.4], con una funcionalidad determinada por las especificaciones y requisitos planteados.

También se revisaron y recolectaron una muestra representativa de aplicaciones del dominio, de esta revisión se determino que lo más conveniente para el desarrollo de la aplicación Calakmul virtual es el uso del motor de juegos de código abierto Nebula, a continuación se presenta el análisis en detalle que se hizo de dicho motor.

3.2.2. Identificación de objetos candidatos reusables.

Se analiza cuales de los objetos del motor de juegos Nebula van a poder ser usados dentro de la aplicación Calakmul virtual, así como las razones del porque de la selección de dichos objetos, para el análisis del motor de juegos Nebula no sólo se tomaron en consideración las clases que trae el propio motor sino que se consideraron todas las clases que tiene Nebula al agregársele el módulo ncal3d y el API para la creación de historias interactivas.

Clases ncollide. Las clases pertenecientes a ncollide son clases que se encargan de dar el servicio de detección de colisiones a otros objetos de Nebula, así pues estas clases generan reportes de las colisiones de un objeto así como el tipo de colisión y el objeto con el que se dio la colisión, también almacenan la información acerca de las mallas de colisión que posee un objeto. Se necesita usar la detección de colisiones para el desarrollo de la aplicación Calakmul virtual, ya que dentro de los comportamientos descritos [2.4.4] muchos son activados mediante la detección de colisiones con objetos de un tipo en particular. Clases ncollide: nCNode, nCollClassNode, nCollideObject, nCollideReport, nCollideReportHandler, nCollideServer, nCollideShape.

Clases ndinput8. Las clases de ndinput8 tienen por objeto el recibir y reconocer entradas de periféricos conectados a la computadora, estas clases realizan su función mediante el uso del API de Windows conocida como Direct Input, al hacer uso de este API, Nebula tiene la posibilidad de acceder a cualquier periférico de entrada que el sistema operativo encuentre conectado al sistema, entiéndase ratón, teclado, joystick, etc. Para la aplicación en desarrollo es muy importante tener una forma por medio de la cual el usuario

de entrada al sistema para modificar entradas y estados de éste, de la igual forma es importante que esta entrada no esté limitada exclusivamente al uso del teclado o del ratón, sino que sea posible hacer uso de periféricos más amigables para el usuario como los controles de video juegos o joysticks. Clases ndinput8: nDI8Server.

Clases ndirect3d8 y nopengl. Estas clases tienen por objeto el manejo de la capa intermedia entre el hardware encargado del despliegue en pantalla y el motor de juegos, principalmente ofrecen a los objetos que así lo requieran el servicio de dibujarse en pantalla. Nebula tiene dos métodos para mandar a dibujar en pantalla, mediante el uso de OpenGL o mediante Direct3D, en general la mayoría del Hardware acelerador de video diseñado para Windows soporta ambos, si se quisiera usar el motor de juegos en otro sistema operativo como linux se tendrían que usar las clases pertenecientes a nopengl. El desplegar en pantalla el mundo virtual, los personajes, los objetos, etc. es una de las funciones básicas con las que debe cumplir la aplicación por lo que el uso de las clases ndirect3d8 y nopengl es muy recomendable. Clases ndirect3d8 y nopengl: nD3D8IndexBuffer, nD3D8PixelShader, nD3d8Server, nD3D8Texture, nGlPixelShader, nGlServer, nGlTexture, nGlVertexPool.

Clases ndsound. Las clases de ndsound tienen la finalidad de reproducir los sonidos que los objetos tengan asociados, además las clases de nsdound se encargan de calcular la distancia y ubicación de los sonidos con respecto del observador. Las clases ndsound realizan su función mediante el uso del API de Windows conocida como Direct Sound, al hacer uso de este API, Nebula tiene la posibilidad de reproducir archivos de audio independientemente del hardware que tenga la computadora para este propósito, basta con que el sistema operativo lo haya detectado y tenga los controladores apropiados para que Nebula mediante el API de Direct Sound le pida al sistema operativo que se reproduzca un sonido. Para la aplicación en desarrollo poder dar salidas al usuario mediante el uso de sonidos es un requisito fundamental ya que una de las ideas principales es que los personajes controlados por el sistema se comuniquen con el personaje controlado por el usuario mediante mensajes hablados. Clases ndsound: nDSoundChannel y nDSoundServer.

Clases nkernel. Estas clases son fundamentales para el motor de juegos Nebula, ya que estas clases representan el núcleo del motor y están encargadas de una gran variedad de funciones, algunas de estas funciones son por ejemplo el recibir y leer los comandos de TCL que se le den al motor ya sea por medio de la consola o por medio de archivos, manejar los apuntadores del motor de juegos para que todos los objetos y servidores que son creados en la aplicación puedan ser manejados de una forma muy similar a la de un sistema de archivos, las clases nkernel también se encargan del manejo del directorio sobre el cual está corriendo la aplicación, es decir indican cual es el directorio actual sobre el que está trabajando Nebula, para abrir, guardar archivos o cambiar de directorio, se encargan de dar el soporte para que una aplicación basada en Nebula pueda recibir comandos a través de Internet mediante el uso de sockets, interpretan los comandos de TCL, permiten la creación de un reloj propio de la aplicación, el cual es muy importante para controlar cosas como el número de cuadros por segundo o saber intervalos de tiempos entre eventos, por último una de las funciones más importantes que llevan a cabo las clases nkernel es la de llevar el control del hilo principal de ejecución del sistema, hilo en el que se actualiza la pantalla, servidores, objetos, etc. Para que una aplicación pueda correr haciendo uso del motor de

juegos Nebula es necesario hacer que la aplicación use las clases de nkernel por lo tanto para el desarrollo de la Calakmul virtual no sólo son reusables estas clases sino que son necesarias. Debido a que la lista de clases que conforman las clases nkernel es muy larga aquí se mencionan las clases más importantes: nCmd, nFileServer, nIpcServer, nIpcClient, nThread, nTimeServer, etc.

Clases nnebula. Este conjunto de clases está enfocado a la creación de objetos dentro del modelo jerárquico de objetos que usa Nebula, así pues este conjunto de clases definen los objetos que Nebula maneja como nodos de un árbol donde estos nodos pueden ser objetos que poseen métodos con los que se cargan a memoria los archivos de texturas, sonidos, mallas 3D, etc, también dentro de este conjunto de clases se encuentran las clases para la creación de los objetos servidores dentro de la jerarquía, los cuales no son dibujados en pantalla, pero ofrecen una gran variedad de servicios a los objetos del motor, dentro de esta estructura en forma de árbol los nodos de dicho árbol corresponden a nodos de objetos 3D, nodos de malla, nodos de audio, nodos de textura, nodos de sombreado, nodos de animación, nodos para los servidores, etc. Las clases de nnebula junto con las clases de nkernel representan las clases básicas del motor de juegos, ya que es posible crear una aplicación usando sólo estas clases. Debido a que la aplicación va a estar basada en este motor de juegos es forzoso usar estas clases para que la aplicación pueda ser compilada. Debido a que la lista de clases que conforman las clases nnebula es muy larga aquí se mencionan las clases más importantes: n3DNode, nAudioServer, nChannelServer, nConServer, nGfxServer, nInputServer, nLenseFlare, nMeshNode, nTexArrayNode, nShaderNode, nVisNode, etc.

Clases nopcode11. Las clases de nopcode11 son clases en las cuales se han programado operaciones que se realizan frecuentemente en aplicaciones de graficación en tiempo real, por ejemplo estas clases ofrecen una gran variedad de operaciones para vectores de tamaño n, operaciones optimizadas para matrices de 4x4, rotaciones, translaciones, escalamientos, etc. Estas operaciones se emplean de modo muy recurrente en el manejo de mallas, modelos tridimensionales y cámaras. nopcode11 También posee operaciones que son usadas por el sistema de detección de colisiones de Nebula. De acuerdo con los requisitos es necesario incluir todas estas cosas en la aplicación por que un conjunto de clases para el manejo matemático resulta muy conveniente. Debido a que la lista de clases que conforman las clases nopcode11 es muy larga aquí se mencionan las clases más importantes: Matrix3x3, Matrix4x4, etc.

Clases nist. Las clases de nist fueron desarrolladas en el laboratorio de interfaces inteligentes de la facultad de ingeniería de la UNAM por el ingeniero Enrique Larios Delgado como parte de un API para facilitar la creación de historias interactivas usando el motor de juegos Nebula, este conjunto de clases ofrecen al programador la facilidad de crear una serie de objetos como lo son personajes, los cuales ya incluyen métodos para cargar la malla, textura, animaciones, métodos para controlar los estados y animaciones del personaje, moverlo, girarlo, detectar las colisiones con otros objetos de las clases nist, etc.

Ofrece objetos conocidos como NPC's [1] que tienen métodos muy similares a los personajes con la diferencia de que estos no son controlados por entradas del usuario, sino que son capaces de seguir caminos predeterminados, los NPC's son personajes que pueden moverse por el mundo virtual sin la necesidad de que un usuario los controle. Nist también proporciona clases para objetos y estructuras, en las cuales ya están programados los comportamientos para que los personajes no puedan atravesarlos por último las clases nist permiten crear un objeto cámara que ya tiene muchas funcionalidades así como varios tipos y estilos. Es necesario usar estas clases en el desarrollo de la aplicación Calakmul virtual debido a que son clases muy cercanas a lo que se establece en los requerimientos y comportamientos de la aplicación, sólo será necesario hacer algunas pequeñas modificaciones a estas clases para que el sistema funcione como se requiere. Clases nist: nIstCamera, nIstCursor, nIstEntity, nIstNPC, nIstPlayer, nIstObject, nIstStruct, nIstWorld.

Clase npath. Esta clase permite crear curvas conocidas como NURBS[2], sólo hay que definir una serie de puntos, con sus coordenadas en sistema cartesiano, y esta clase se encarga de crear una curva que pase por todos los puntos indicados en el orden en el que se indicaron. El uso de esta clase es necesario debido a que la clase nIstNPC, la cual pertenece a las clases nist, hace uso de estas curvas para que los personajes controlados por el sistema se muevan a través del mundo virtual, no se pueden usar las clases de nist en la aplicación sin usar la clase npath.

Clases ncal3d. Las clases de ncal3d en su conjunto permiten el cargar mallas que posean animaciones generadas mediante el software de diseño 3d Studio max, al usar un plug–in de este software conocido como Biped, el cual permite hacer animaciones mediante el uso de esqueletos, las clases de ncal3d no sólo permiten que estas mallas y esqueletos sean portados a Nebula, sino que además proporciona un sistema de mezcla entre las animaciones que se hayan portado. Estas clases se pueden usar en Calakmul virtual debido a que es necesario tener un sistema de animaciones mediante el cual los personajes al cambiar de estado cambien la animación en la que se encuentran, por ejemplo al cambiar del estado inmóvil a avanzar, la animación en la que el personaje está parado debe cambiar a una en la que mueve los pies, varias de las clases que pertenecen a nist usan ncla3d para el manejo de las animaciones de los personajes. Clases ncal3d: nCal3dCoreModel, nCal3dMaterial y nCal3dModel.

3.2.3. Adaptaciones a los objetos reusables.

El objetivo de este apartado es: Una vez que se han encontrado las clases que pueden ser usadas en la aplicación es necesario identificar aquellas que necesitan alguna adaptación en particular para que los objetos se comporten de acuerdo a las especificaciones y requerimientos.

---

[1] Abreviación (Non Player Characters) surgida del ámbito de los videojuegos, la cual se refiere a todos aquellos personajes que son controlados por el sistema y cuya finalidad principal es la de interactuar con el personaje controlado por el usuario.

[2] Abreviación (Non Uniform Rational b-spline) representaciones de objetos en tres dimensiones.

A continuación se mencionan las clases que necesitan ser modificadas así como las modificaciones en particular para cada una de estas clases:

Clase nJoyMouseDevice. Clase que pertenece a las clases de nnebula, es necesario modificar esta clase para que las entradas al mover el ratón sobre los ejes X y Y sean como lo requiere la creación de una vista de primera persona, ya que la entrada original del ratón entrega números que decrecen cuando el ratón deja de moverse, valores que pueden provocar vibraciones en la vista.

Clase nIstCamera. Clase que pertenece a las clases de nist. Aunque la cámara posee originalmente un tipo de cámara conocido como "vista de primera persona"[3], ésta no se comporta como se espera, ya que si bien esta cámara gira y avanza junto con el personaje como se pide en los requerimientos, este tipo de cámara no tiene la posibilidad de que el usuario pueda girar la vista a través del eje X es decir que el usuario no sólo pueda ver hacia los lados sino que también pueda ver hacia arriba o abajo, además la forma en la que está implementada esta cámara no es la que se necesita ya que originalmente el giro del personaje es el que hace girar la cámara, cuando la cámara tiene que ser la que haga girar al personaje ya que así se puede evitar que al hacer girar la cámara en el eje X el personaje en el mundo virtual se incline, También es necesario modificar la cámara para que ésta pueda informar que clase de objeto se está observando y los objetos puedan tener conocimiento de que están siendo observados a través de la cámara, esta modificación debe ir acompañada junto con un mapeo a una entrada del usuario para que de esta manera sea posible desencadenar comportamientos que son requeridos por el sistema.

Clase nIstPlayer. Clase que pertenece a las clases nist. Una de las modificaciones que se tiene que hacer a esta clase como ya se menciono es que al crear una cámara con vista de primera persona los giros del personaje en el eje Y sean controlados por la cámara a través del ratón, pero cuando la cámara presente un giro en el eje X (el ratón se mueva verticalmente) el personaje no sufra ningún giro. Otra modificación que es necesaria hacer a la clase nIstPlayer es la de incrementar el número y tipo de colisiones que desencadenan un evento o comportamiento ya que la clase personaje originalmente sólo presenta comportamientos al chocar con otros objetos, comportamientos como no atravesar a otros personajes, cosas o estructuras, el personaje sólo recibe fuerzas de repulsión cuando se detecta una colisión con otro objeto que pertenece a alguna de las clases de nist.

Clase nIstNPC. Clase que pertenece a las clases de nist. Los cambios que se le tienen que hacer a esta clase son en la parte de la detección de colisiones con otros objetos que pertenecen a la clase nist. En este momento la clase nIstNPC solamente reporta comportamientos cuando hay una colisión entre su malla de colisiones y la de un objeto nIstPlayer pero es necesario que los objetos de la clase nIstNPC tengan conocimiento de que la cámara está apuntando hacia ellos, con estas modificaciones se puede lograr que los

[3] En el campo de los videojuegos esta cámara también es conocida como "First Person Shooter" consiste en dar el punto de vista al usuario de lo que esta observando el personaje en el mundo virtual, este tipo de cámara busca darle una mayor inmersión al usuario al tratar de darle la sensación de que él es quien esta dentro del mundo virtual.

personajes controlados por el sistema tengan reacciones o ejecuten archivos de scripts cuando los usuarios estén mirando directamente al NPC.

Clase nIstObject. Clase que pertenece a las clases de nist. Los cambios que se le tienen que hacer a la clase nIstObject también son en la detección de colisiones con otros objetos que pertenecen a la clase nist. En este momento esta clase solamente ejecuta un script de TCL cuando hay una colisión entre su malla de colisiones y la de un objeto nIstPlayer pero es necesario hacer las modificaciones para que los objetos nIsObject tengan conocimiento de que la cámara está apuntando hacia ellos, lo que se busca con estas modificaciones es que cuando el usuario esté observando un objeto en particular también se pueda ejecutar un script de TCL en el que por ejemplo se le de información al usuario sobre ese objeto en particular.

Clase nIstStruct. Clase que pertenece a las clases de nist. Los cambios que se le tienen que hacer a la clase nIstStruct son en la detección de colisiones con otros objetos que pertenecen a la clase nist. En este momento esta clase sólo se encuentra dentro del reporte de colisiones para que los objetos nIstPlayer no puedan atravesar los modelos que corresponden al suelo o los edificios, es necesario hacer las modificaciones para que los objetos nIstStruct tengan conocimiento tanto de las colisiones que ha habido con los personajes como saber si la cámara está apuntando hacia ellos, lo que se busca con estas modificaciones es que cuando el usuario esté observando o haga contacto con una estructura en particular también se pueda ejecutar un script de TCL en el que por ejemplo se le de información al usuario sobre ese lugar en particular.

Una vez que se han definido los objetos es importante estimar qué porcentaje de una aplicación típica podría construirse usando los objetos reusables. Para una aplicación como la que se piensa construir es difícil definir con gran precisión el porcentaje que estas clases representen del producto final, ya que si bien se tiene algo así como un 80% de las clases necesarias para construir la aplicación no implica que se tenga ya el 80% de la aplicación ya que falta considerar cosas como los contenidos los cuales en muchos casos son los que más tiempo se llevan en crear y agregar a la aplicación.

3.3. Proceso del análisis orientado a objetos

El proceso de AOO no comienza con una preocupación por los objetos. Más bien comienza con una comprensión de la manera en la que se usará el sistema: por las personas, si el sistema es de interacción con el hombre; por otras máquinas, si el sistema está envuelto en un control de procesos; o por otros programas; si el sistema coordina y controla otras aplicaciones. Una vez que se ha definido el escenario, comienza el modelado del software.

Las secciones que siguen definen una serie de técnicas que pueden usarse para recopilar requisitos básicos del usuario y después definen un modelo de análisis para un sistema orientado a objetos.

3.3.1. Casos de uso o utilización.

Para crear un caso de uso, el analista debe primero identificar los diferentes tipos de personas (o dispositivos) que usan el sistema o producto. Estos actores actualmente representan papeles ejecutados por personas (o dispositivos) cuando el sistema está en operación. Definido de una manera más formal un actor es cualquier cosa que se comunique con el sistema o producto y que sea externo a él.

Los actores que se comunican con la aplicación Calakmul virtual son los usuarios que por medio del software pueden observar y conocer la zona arqueológica de Calakmul sin la necesidad de hacer un viaje al lugar físicamente el cual se encuentra en una zona de difícil acceso, Los usuarios principalmente reciben información didáctica sobre las pirámides, tumbas, ornamentos, vasijas, etc. Que se han encontrado en Calakmul al interactuar con el sistema por medio de un personaje el cual controla y maneja en una representación virtual dela zona arqueológica.

Una vez que los actores han sido identificados, pueden desarrollarse casos de uso. El caso de uso describe la forma en la cual un actor interactúa con el sistema.

-El actor puede controlar un personaje dentro del mundo virtual, el control del personaje implica que puede indicarle a éste que avance, que retroceda, que camine hacia la izquierda, derecha, o que gire, el control del personaje también considera el control de la vista del personaje es decir el actor puede indicarle al personaje que voltee a los lados hacia arriba o abajo. El actor por medio del control que tiene del personaje puede interactuar con el mundo virtual de varias formas, al observar a cierta distancia otros objetos del mundo virtual, al hacer contacto el personaje que controla con otros objetos, al mover al personaje por el mundo y pasar por ciertas zonas de importancia y por último al combinar la acción de observar otros objetos y recibir una entrada del actor denominada "acción".
-El actor recibe información del sistema, esta información es presentada al actor de manera visual y sonora, esto se puede resumir en que el actor recibe información general sobre el sitio arqueológico de Calakmul, datos históricos, explicaciones, referencias, etc. El actor por medio de las acciones que lleva acabo en el mundo virtual cambia el estado de los objetos y la información que ve y escucha, por ejemplo ya se mencionaron las formas en las que un actor mediante el personaje puede interactuar con el mundo virtual, entonces si el actor observa a un personaje controlado por el sistema éste puede comenzar una animación en la que saluda y reproducir un archivo de audio en el cual dice su nombre y se presenta ante el actor.
-El sistema no necesita que el actor le informe acerca de cambios en el entorno exterior al sistema ni necesita de periféricos externos para recibir entradas que no sean exclusivamente aquellos periféricos necesarios para controlar al personaje.
-El actor no debe necesitar mucha información para poder hacer uso del sistema, como básicamente usar el sistema es poder controlar al personaje, el control debe ser sencillo e intuitivo para el actor.

3.3.2. Modelado de clases – responsabilidades – colaboraciones.

Una vez que se han desarrollado los escenarios de uso básicos para el sistema, es tiempo de identificar las clases candidatas, e indicar sus responsabilidades y colaboraciones. El modelado de clases-responsabilidades-colaboraciones (CRC) aporta un medio sencillo de identificar y organizar las clases que resulten relevantes al sistema o requisitos del producto.

Las responsabilidades son los atributos y operaciones relevantes para la clase. Puesto de forma simple, una responsabilidad es cualquier cosa que conoce o hace la clase. Los colaboradores son aquellas clases necesarias para proveer a una clase con la información necesaria para completar una responsabilidad. En general, una colaboración implica una solicitud de información o una solicitud de alguna acción.

Los atributos representan características estables de una clase, esto es, información sobre la clase que debe retenerse para llevar a cabo los objetivos del software especificados. Los atributos pueden a menudo extraerse del planteamiento de alcance o discernirse a partir de la comprensión de la naturaleza de la clase. Las operaciones pueden extraerse desarrollando un análisis gramatical sobre la narrativa de procesamiento del sistema. Los verbos se transforman en candidatos a operaciones. Cada operación elegida para una clase exhibe un comportamiento de la clase.

Las clases cumplen con sus responsabilidades en una de dos maneras: una clase puede usar sus propias operaciones para manipular sus propios atributos, cumpliendo por lo tanto con una responsabilidad particular, o una clase puede colaborar con otras clases. Wirfs-Brock y sus colegas definen las colaboraciones de la siguiente forma:

"Las colaboraciones representan solicitudes de un cliente a un servidor en el cumplimiento de una responsabilidad del cliente. Una colaboración es la realización de un contrato entre el cliente y el servidor... Decimos que un objeto colabora con otro, si para ejecutar una responsabilidad, necesita enviar cualquier mensaje al otro objeto. Una colaboración simple fluye en una dirección, representando una solicitud del cliente al servidor. Desde el punto de vista del cliente, cada una de sus colaboraciones está asociada con una responsabilidad particular implementada por el servidor".

Las colaboraciones identifican relaciones entre clases. Cuando todo un conjunto de clases colabora para satisfacer algún requisito, es posible organizarlas en un subsistema (un elemento del diseño). Las colaboraciones se identifican determinando si una clase puede satisfacer cada responsabilidad. Si no puede, entonces necesita interactuar con otra clase. Por consiguiente, una colaboración.

El modelado clases – responsabilidades – colaboraciones que se hizo para este trabajo sólo se presentan aquellas clases en las que se hicieron cambios para que su comportamiento fuera el adecuado para la aplicación que se está desarrollando y también para aquellas clases que necesitan ser creadas ya que de todas las clases que se han identificado como reusables no existe ninguna clase que cumpla con las tareas para las que estas nuevas clases van a ser creadas.

Nombre: nIstPlayer	
Tipo de Clase: Rol	
Características: Intangible, agregada, secuencial, temporal.	
Responsabilidades:	**Contribuyentes:**
Actuar conforme a las entradas que le dé el usuario	NInputServer, nIstCamera
Interactuar	nIstNPC, nIstStruct, nIstObject, nIstCollZone, nIstStairs
Controlar animaciones	nCal3DModel
Controlar la física	nIstWolrd
Conocer sus cualidades físicas	
Actualizar el estado interno	
Manejo de colisiones	nCollideServer

Tabla 3.1. Modelado responsabilidades – colaboraciones, clase nIstPlayer.

nIstPlayer. Clase originalmente incluida dentro de las clases nist y cuyas responsabilidades principales de acuerdo con el modelo CRC[4] son controlar al personaje que representa al usuario dentro de la historia. Manejar las entradas del usuario que tengan la intención de controlar al personaje y llamar a las animaciones adecuadas. Dentro del modelado CRC que se hizo de la clase nIstPlayer se indican las modificaciones que hay que hacerle a la clase, las entradas del usuario que afectan al personaje no son enviadas a la clase exclusivamente por el servidor de entradas, sino que la cámara también le debe dar información a la clase sobre las entradas que da el usuario, además la clase nIstPlayer debe interactuar con otras dos nuevas clases, nIstCollZone y nIstStairs.

Nombre: nIstNPC	
Tipo de Clase: Cosa	
Características: Agregada, intangible, secuencial, transitoria.	
Responsabilidades:	**Contribuyentes:**
Actuar	
Controlar animaciones	nCal3DModel
Responder a Interacción	nIstPlayer, nIstCamera
Seguir scripts activados por eventos	nScriptlet
Controlar la física	nIstWolrd
Conocer cualidades físicas	
Actualizar el estado interno	
Manejo de colisiones	nCollideServer
Seguir caminos	nPath
Comunicarse con el usuario	nVoiceServer

Tabla 3.2. Modelado responsabilidades – colaboraciones, clase nIstNPC.

[4] Abreviación que hace referencia al modelo Clases Responsabilidades Colaboraciones (CRC).

nIstNPC. Clase originalmente incluida dentro de nist y cuyas responsabilidades principales de acuerdo con el modelo CRC son la de controlar a todos los personajes de la historia que no son controlados por el usuario. Debido a que los comportamientos que los personajes controlados por el sistema deben realizar son muy variados para pensar en poder programarlos todos, es muy importante que estos personajes puedan ejecutar scripts que les darán un comportamiento único. Dentro del modelado CRC que se hizo de la clase nIstPlayer ya se indican las modificaciones que hay que hacerle a la clase, nIstNPC no sólo debe responder a la interacción con el personaje controlado por el usuario sino que la cámara también debe contribuir a este propósito.

Nombre: nIstObject	
Tipo de Clase: Cosa	
Características: Agregada, Intangible, secuencial, transitoria	
Responsabilidades:	**Contribuyentes:**
Responder a Interacción	nIstPlayer, nIstCamera
Seguir scripts activados por eventos	nScriptlet
Controlar la física	nIstWolrd
Conocer cualidades físicas	
Actualizar el estado interno	
Manejo de colisiones	nCollideServer

Tabla 3.3. Modelado responsabilidades – colaboraciones, clase nIstObject.

nIstObject. Clase originalmente incluida dentro de nist y cuya responsabilidad principal de acuerdo con el modelo CRC es controlar a cualquier objeto y las interacciones que éste puede tener con el usuario, del modelado CRC que se hizo de la clase nIstObject se indica que la clase no sólo debe tener interacciones con nIstPlayer sino que también deben de existir con la clase nIstCamera.

Nombre: nIstStruct	
Tipo de Clase: Cosa	
Características: Intangible, agregada, secuencial, transitoria	
Responsabilidades:	**Contribuyentes:**
Seguir scripts activados por eventos	nScriptlet
Controlar la física	nIstWolrd
Actualizar el estado interno	
Manejo de colisiones	nCollideServer

Tabla 3.4. Modelado responsabilidades – colaboraciones, clase nIstStruct.

nIstStruct. Clase originalmente incluida dentro de las clases nist y cuya responsabilidad principal de acuerdo con el modelo CRC es la de controlar a todos los objetos que forman parte del escenario, no pueden cambiar su posición. Esto incluye al

piso, las paredes y el techo de cualquier estructura presente, originalmente nIstStruct sólo era responsable de informar de las colisiones ocurridas con objetos nIstPlayer mediante el servidor de colisiones para que esta clase pudiera llevar acabo el comportamiento adecuado, de acuerdo al modelado CRC que se hizo de la clase nIstStruct también debe de interactuar con la clase nIstCamera.

Nombre: nIstCamera	
Tipo de Clase: Propiedad	
Características: Abstracta, atómica, secuencial, transitoria.	
Responsabilidades:	**Contribuyentes:**
Seguir scripts activados por eventos	nScriptlet
Actualizar el estado interno	
Conocer del estilo de cámara	
Presentar la posición y la orientación desde donde es vista	nSceneGraph2
Seguir caminos	nPath
Responder a las entradas en el modo de cámara libre y primera persona	nInputEvent

Tabla 3.5. Modelado responsabilidades – colaboraciones, clase nIstCamera.

nIstCamera. Clase originalmente incluida dentro de las clases nist y cuya responsabilidad principal de acuerdo con el modelo CRC es mantener la posición y la orientación desde la cual se presenta la escena. Además, esta clase cuenta con diferentes estilos de cámara que le permiten cambiar la orientación y la posición siguiendo criterios artísticos, del modelado CRC que se hizo de la clase nIstCamera, ésta no sólo debe de responder a las entradas en el modo de cámara libre sino que también responde a las entras del usuario cuando está en el modo de vista de primera persona, contribuyendo con nIstPlayer para el control del personaje, además nIstCamera debe de manejar colisiones, para que los demás objetos puedan tener reacciones cuando la cámara los esté apuntando.

Nombre: nIstCollZone	
Tipo de Clase: Cosa	
Características: Agregada, Intangible, secuencial, transitoria	
Responsabilidades:	**Contribuyentes:**
Responder a Interacción	nIstPlayer
Seguir scripts activados por eventos	nScriptlet
Conocer cualidades físicas	
Actualizar el estado interno	
Manejo de colisiones	nCollideServer

Tabla 3.6. Modelado responsabilidades – colaboraciones, clase nIstCollZone.

nIstCollZone. Clase cuya responsabilidad principal de acuerdo con el modelo CRC es controlar las llamadas zonas de colisiones y las interacciones que estas pueden tener con el usuario, del modelado CRC que se hizo de la clase nIstCollZone se indica que la clase a diferencia de nIstObject sólo debe tener interacciones con nIstPlayer ya que los objetos nIstCollZone no son visibles por la cámara, no tiene sentido que la cámara interactúe con las zonas de colisión.

Nombre: nIstStairs	
Tipo de Clase: Cosa	
Características: Intangible, agregada, secuencial, transitoria	
Responsabilidades:	**Contribuyentes:**
Controlar la física	nIstWolrd
Actualizar el estado interno	
Manejo de colisiones	nCollideServer

Tabla 3.7. Modelado responsabilidades – colaboraciones, clase nIstStairs.

nIstStairs. Clase cuya responsabilidad principal de acuerdo con el modelo CRC es la de controlar a todos los objetos conocidos como escaleras, a diferencia de nIstStruct estos objetos no son visibles por el usuario ya que lo único que importante para los objetos de esta clase es su malla de colisión, cuando se le reporta a un objeto de la clase nIstPlayer que existe colisión con un objeto nIstStairs, éste debe tener programado un comportamiento diferente al que se presenta cuando existe una colisión con un objeto nIstStruct, un objeto nIstStruct impide que el personaje atraviese su malla de colisión asociada , cuando existe una colisión con un objeto nIstStairs el objeto de la clase nIstPlayer debe tener programado el comportamiento que se espera del personaje al encontrase en unas escaleras, así como la clase nIstCollZone, nIstStairs no interactúa con la cámara.

Nombre: nIstWorld	
Tipo de Clase:	
Características:	
Responsabilidades:	**Contribuyentes:**
Actualizar todo los elementos que contiene	nIstNPC, nIstPlayer, nIstObject, nIstCollZone
Conocer el valor de la gravedad	
Cargar las estructuras de la escena	nIstStruct, nIststairs

Tabla 3.8. Modelado responsabilidades – colaboraciones, clase nIstworld.

nIstWorld. Clase originalmente incluida dentro de las clases nist y cuya responsabilidad principal de acuerdo con el modelo CRC es contener y actualizar a todos los elementos del mundo virtual, clases identificadas por el prefijo IST (Interactive StoryTelling). La clase nIstWorld carga las estructuras de la escena, originalmente nIstworld busca archivos que se encuentren dentro de una carpeta con un nombre ya

preestablecido y que además dichos archivos tengan un nombre genérico, cargando todo aquello que cumpla con estas condiciones, a nistwolrd se le necesitan hacer modificaciones para cambiar la forma en que las estructuras son cargadas, además de que debe proporcionar un método y comando con el cual se puedan cargar los objetos nIstCollZone.

Nombre: nIstEntity	
Tipo de Clase: Contenedor	
Características: Intangible, agregada	
Responsabilidades:	**Contribuyentes:**
Contener valores y datos que se aplican a todas las clases que hereda	nIstWorld
Contener métodos comunes a las clases que van a heredar	NCollideObject, nCollideShape
Ser la clase padre necesaria para agregar a todas las demás clases dentro de la jerarquía de Nebula	nRoot

Tabla 3.9. Modelado responsabilidades – colaboraciones, clase nIstEntity.

nIstEntity. Clase originalmente incluida dentro de las clases nist de acuerdo con el modelo CRC esta clase es responsable de contener los valores de constantes físicas que se aplican a todos los entes que contiene, así como de proporcionar métodos comunes a todas estas clases (nIstPlayer, nIstNPC, nIstObject, nIstStruct, nIstStairs, nIstCollZone, nIstCursor).

Nombre: nSceneGraph2	
Tipo de Clase: Cosa	
Características: Abstracta, agregada, secuencial, temporal	
Responsabilidades:	**Contribuyentes:**
Conocer todos los elementos de la escena gráfica	nVisNode
Definir el punto y la orientación desde el cual es dibujada la escena	nIstCamera
Dibujar todos los elementos visibles desde cierta posición y orientación	nGfxServer

Tabla 3.10. Modelado responsabilidades – colaboraciones, clase nSceneGraph2.

nSceneGraph2. Clase perteneciente al motor de juegos Nebula, no se desarrolló en este proyecto, se incluye dentro del presente trabajo debido a que ésta es la clase responsable de contener a todos los objetos que aparecen en el mundo virtual, es la clase

responsable de conocer la posición desde la cual se dibuja la escena y de indicar a los objetos de la escena cuales y cuando se dibujan.

3.3.3. Definición de estructuras y jerarquías.

Una vez que se han identificado las clases y objetos usando el modelo CRC, nos podemos centrar en la estructura del modelo de clases y las jerarquías resultantes que surgen al emerger clases y subclases.

A continuación se muestra la jerarquía de todas las clases que pueden estar en el mundo virtual, todas estas heredan de la clase nIstEntity, como se indica en el modelo CRC nIstEntity es la clase en la que se programan métodos comunes a todas estas clases, por ejemplo todas estas clases deben tener métodos con los cuales crear y cargar su malla de colisión en este diagrama [ver figura 3.1.] se incluyen las nuevas clases que deben ser creadas en el desarrollo de la aplicación Calakmul virtual (nIstCollZone, nIstStairs).

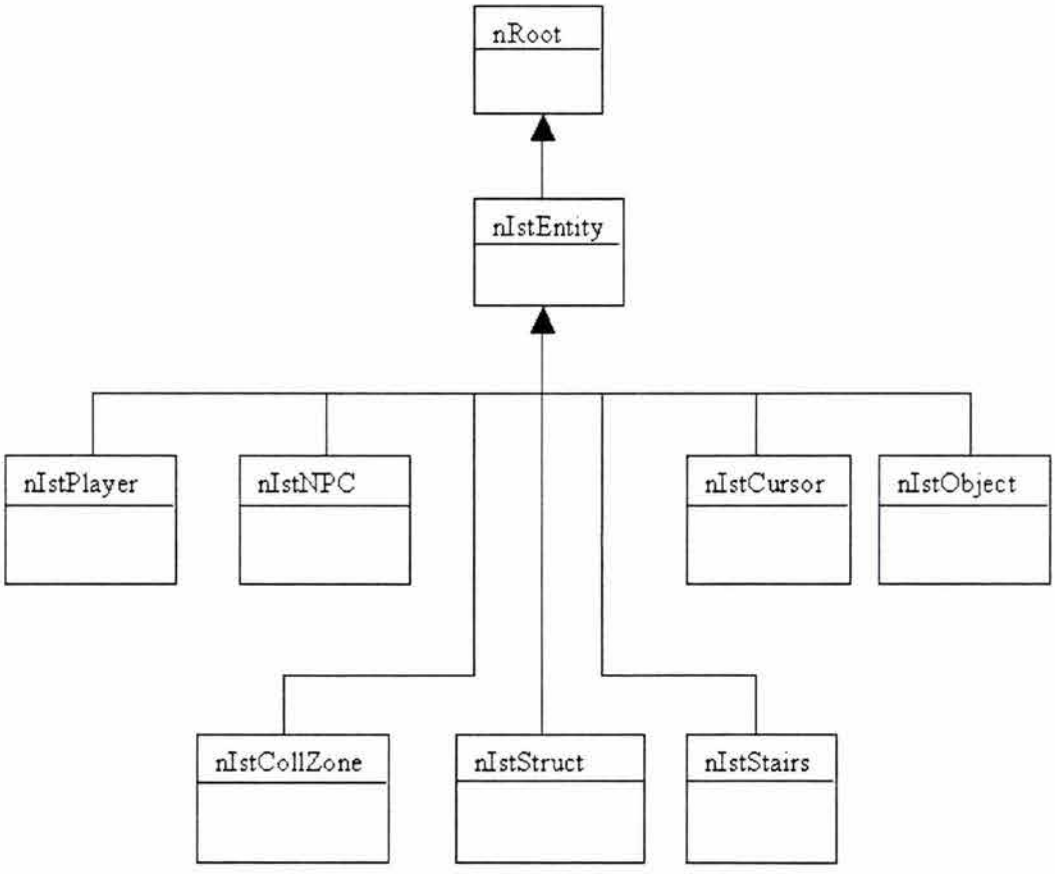


Figura 3.1. Jerarquía de clases que heredan de nIstEntity.

nRoot es la superclase de rodas las clases de alto nivel en Nebula, la cual define los siguientes mecanismos.

-Ligar al espacio jerárquico de nombres.

-Crear un mecanismo de referencia, el cual implementa apuntadores seguros a otros objetos de C++.
-Información de los objetos en tiempo real, a los objetos se les puede preguntar a que clase pertenecen o de que clase heredan.
-Persistencia de objetos, los objetos pueden guardar su estado actual dentro de una cadena de comandos.
-Proveer de una interfaz de scripts, se reciben comandos que son traducidos en llamadas a métodos nativos de C++.

Básicamente todas las clases que se agregan a Nebula deben de heredar de nRoot ya sea de una manera directa o indirecta, es decir heredar de una clase que a su vez herede de nRoot, así pues nIstEntity se agrega dentro de la jerarquía de clases de Nebula al heredar de nRoot, herencia que también necesitan nIstPlayer, nIstNPC, nIstObject, nIstStruct, nIstStairs, nIstCollZone, nIstCursor. La clase nIstEntity posee métodos comunes a las clases que van a heredar, de esta manera se hace uso de una ventaja de la programación OO ya que en lugar de programar los mismos métodos en cada una de las clases, estos simplemente se heredan, Como se muestra [figura 3.2.] nIstCollZone y nIstStairs también van a heredar de nIstEntity debido a que comparten varias de las necesidades con las otras clases.

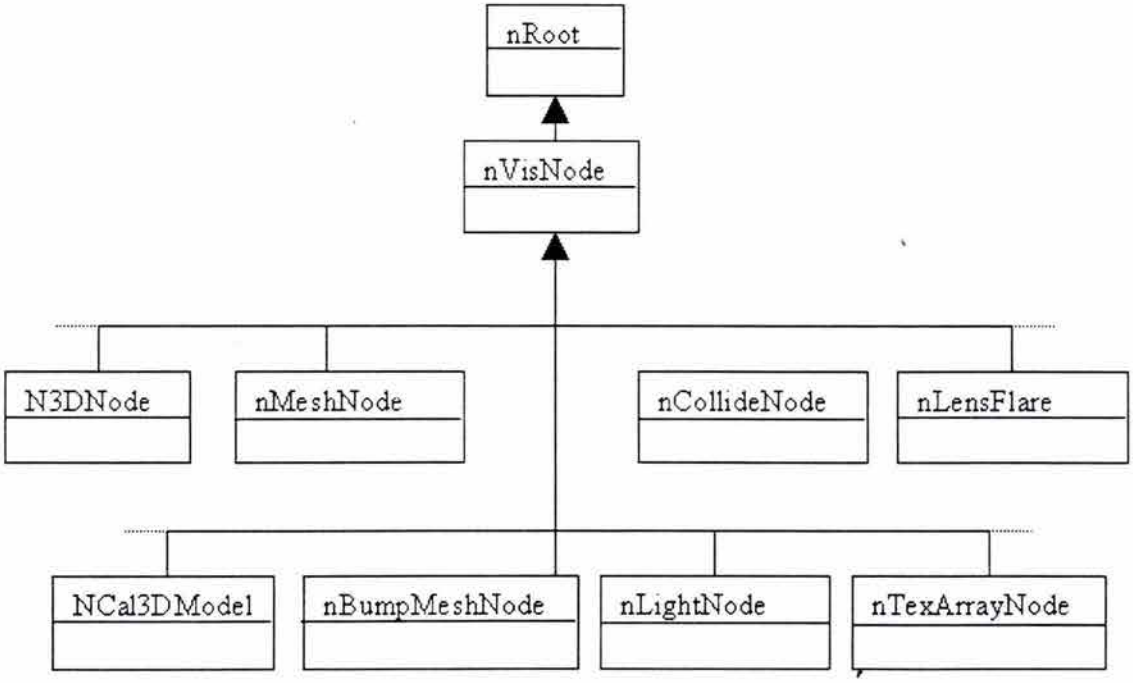


Figura 3.2. Jerarquía de clases nVisNode.

En la figura anterior, se presenta la jerarquía de los objetos que pueden aparecer en escena, todos las clases que de una u otra forma tienen que ver con objetos que se encuentren dentro del mundo virtual deben de heredar de la clase nVisNode, ya que todos los objetos que representan gráficamente objetos tridimensionales existen como subclases

de nVisNode. Como subclases de nVisnode se encuentran clases encargadas de agrupar y transformar objetos en el mundo virtual, clases que definen mallas estáticas dentro de la jerarquía, clases que envuelven objetos de colisión para que la malla de colisión sea visible con propósitos de depuración, clases que encapsulan los atributos de iluminación etc .Un detalle a remarcar de esta jerarquía es la aparición de la clase nCal3DModel como subclase de nVisNode.

En el diagrama no se muestran todas las clases que heredan de nVisNode por cuestiones de espacio, en la figura se busca mostrar cual es la idea de la jerarquía que maneja Nebula para las clases que se manejan dentro del mundo virtual.

Ahora se mostrará la jerarquía del motor de juegos Nebula de una manera más global, esta jerarquía está presentada en una forma simplificada ya que son cientos las clases que forman en su totalidad a Nebula.

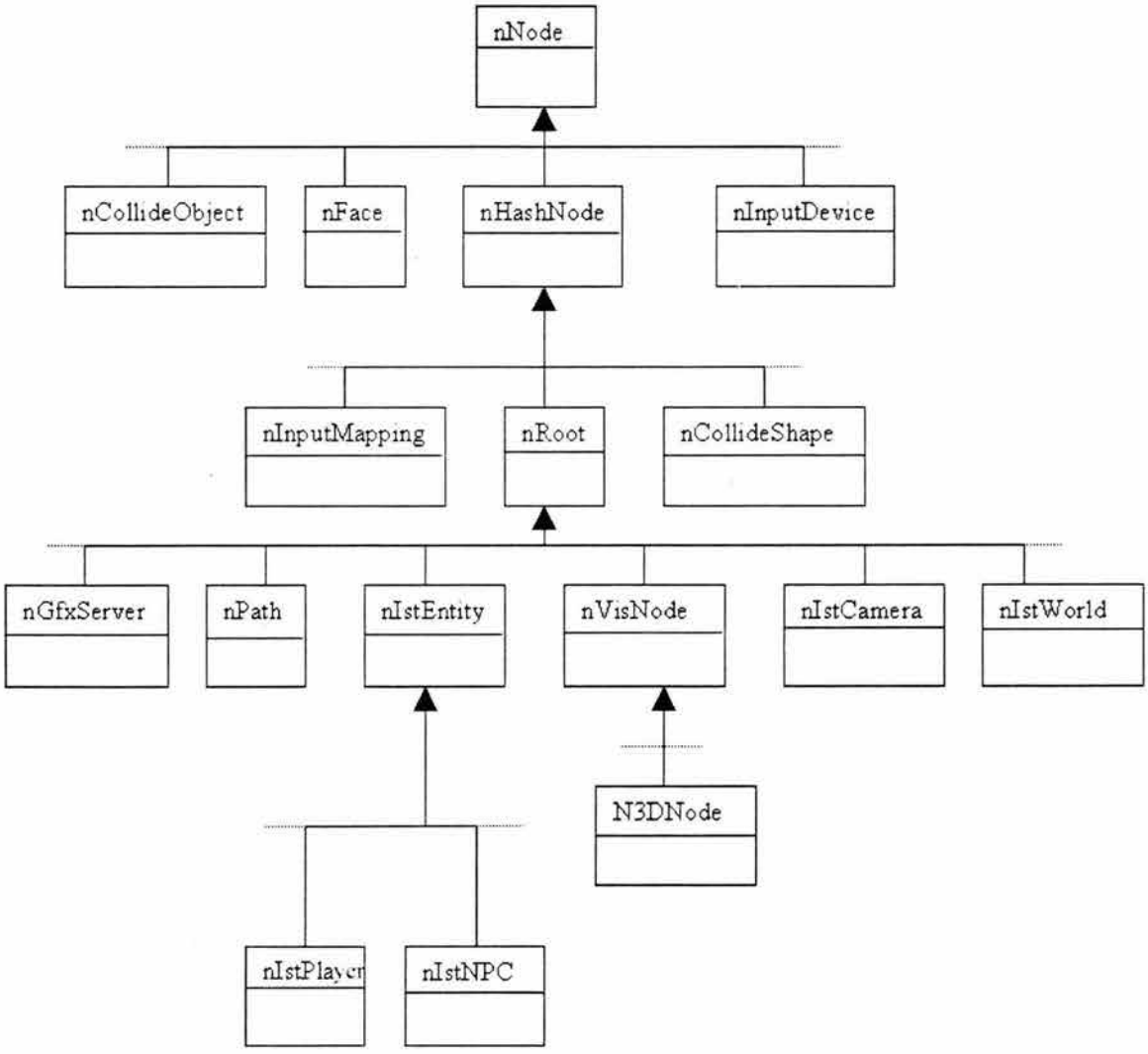


Figura 3.3. Jerarquía de clases de Nebula simplificada.

De la figura anterior [figura 3.3.] caben resaltar los siguientes puntos. La clase nNode que aparece como la raíz del árbol es una clase que lo único que hereda a todas las demás es la implementación de nodos como una lista doblemente ligada, el siguiente nodo importante dentro de la jerarquía es nHashnode, esta clase proporciona a su herencia la posibilidad de poder ser manejados como elementos dentro de una hashlist cuya característica principal es la de poseer una tabla con la cual se pueden encontrar elementos de la lista más rápidamente, la siguiente clase importante en la jerarquía es nRoot.

3.3.4. Modelo objeto – relación.

Una vez que se han comprendido las responsabilidades de cada clase. El siguiente paso es definir aquellas clases colaboradoras que ayudan en la realización de cada responsabilidad. Esto establece la «conexión» entre las clases.

Una relación existe entre dos clases cualesquiera que estén conectadas. Debido a esto los colaboradores siempre están relacionados de alguna manera. El tipo de relación más común es la binaria (existe una relación entre dos clases). Cuando se analiza dentro del contexto de un sistema OO, una relación binaria posee una dirección específica que se define a partir de que clase desempeña el papel del cliente y cuál actúa como servidor.

Modelo objeto – relación de la clase nIstWorld. La clase nIstWorld actualiza el estado de por lo menos un objeto (nIstPlayer, nIstNPC, nIstObject, nIstCollZone) dentro del mundo virtual por lo que la relación que nIstWorld tiene con dichos objetos es de 1 a 1, dentro del mundo virtual es posible que muchos o nigún objeto de las clases mencionadas exista, relación 0 a muchos. De manera similar sucede con los objetos de las clases nIstStruct y nIstStairs, nIstWorld carga las mallas de colisión de por lo menos uno de estos objetos, relación 1 a 1, pero en el mundo virtual puede que varios objetos de las clases indicadas necesiten que nIstWorld cargue sus mallas de colisión. La figura 3.4 muestra de manera gráfica el modelo objeto – relación para la clase nIstWorld.

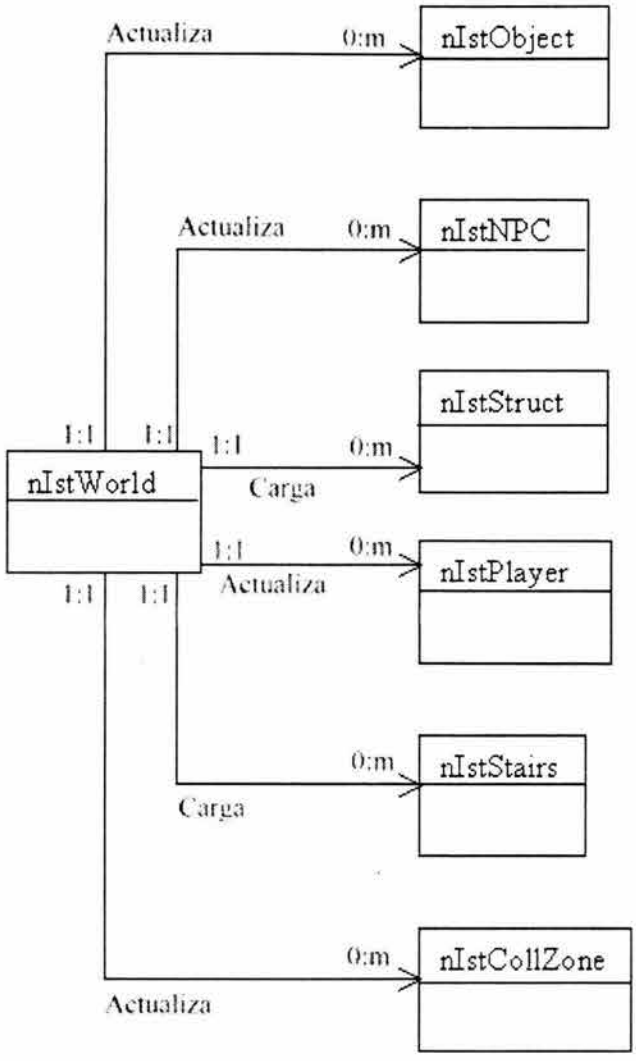


Figura 3.4. Modelo objeto – relación para la clase nIstWorld.

Modelo objeto – relación de la clase nIstCamera. La clase nIstCamera responde a varias o ninguna entrada, para nInputEvent no es necesario que la clase nIstCamera responda a alguno de los eventos de entrada producidos por el usuario, lo mismo sucede con la clase nScriptlet con la diferencia de que la finalidad de esta clase es la de activar scripts. nIstPath puede definir 0 ó muchos caminos independientemente de si la cámara los sigua o no, la cámara no necesita seguir un camino necesariamente. Objetos de la clase nIstPlayer sufren de rotaciones indicadas por la cámara, toda rotación de la cámara no necesariamente indica una rotación del objeto. La figura 3.5 muestra gráficamente el modelo objeto – relación para la clase nIsCamera.

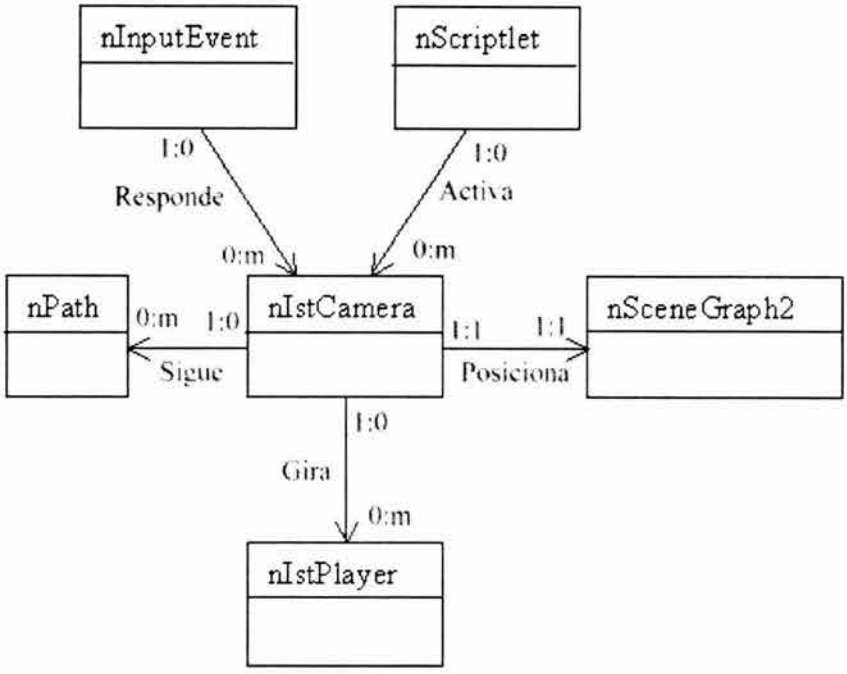


Figura 3.5. Modelo objeto – relación para la clase nIstCamera.

Modelo objeto – relación de la clase nIstPlayer. Para este modelo las relaciones de la clase nIstPlayer con las clases nIstWorld y nIstCamera ya fueron comentadas dentro de los modelos objeto – relación de estas clases. La clase nIstPlayer necesita que el servidor de colisiones le informe de las colisiones que han ocurrido con objetos de otras clases, no puede existir más que un servidor de colisiones, por su parte el servidor de colisiones no necesariamente debe dar información a un objeto de la clase nIstPlayer, ya que puede no existir algún objeto de esta clase. Cada objeto de la clase nIstPlayer tiene un modelo de malla, materiales, animaciones y un esqueleto del modelo, la clase nCal3DModel es responsable de la presentación gráfica de los personajes, dibuja la malla y animaciones, una instancia de la clase nCal3DModel sólo puede dibujar a un personaje y un mismo personaje no puede ser enviado a dibujar por varios objetos nCal3DModel. La figura 3.6 muestra de manera gráfica el modelo objeto – relación para la clase nIstPlayer.

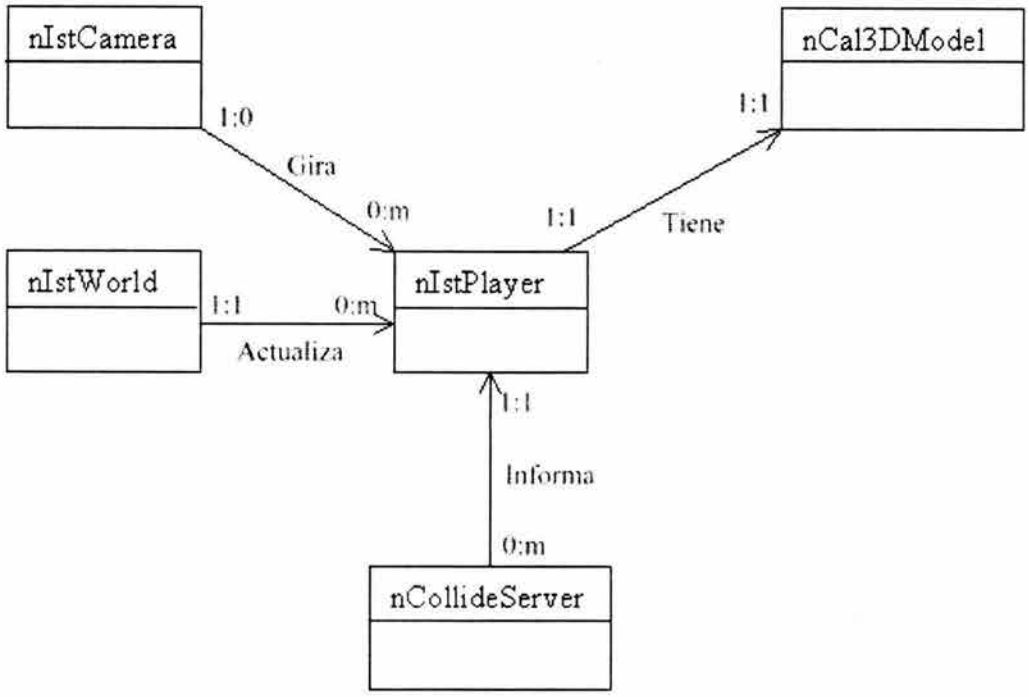


Figura 3.6. Modelo objeto – relación para la clase nIstPlayer.

Modelo objeto – relación de la clase nIstCollZone. Las relaciones que presenta la clase nIstCollZone ya fueron comentadas en el modelo de la clase nIstPlayer, cabe la pena resaltar que los objetos de la clase nIstCollZone no necesitan de alguna clase cuya finalidad sea la de dibujar en pantalla, así como nCal3DModel dibuja la malla 3d para los objetos de la clase nIstPlayer, estos objetos no necesitan de representación gráfica, lo importante es su malla de colisión. La figura 3.7 muestra gráficamente el modelo objeto – relación para la clase nIstCollZone.

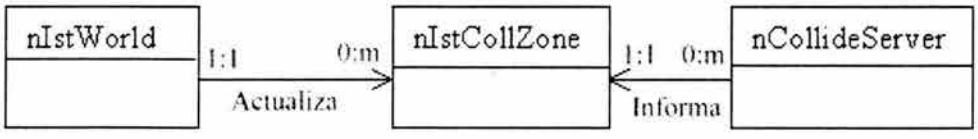


Figura 3.7. Modelo objeto – relación para la clase nIstCollZone.

Ahora se muestra el modelo objeto relación de una manera más general para la parte del sistema que fue creada o modificada. En la figura se muestran varios objetos cuyos modelos ya fueron analizados.

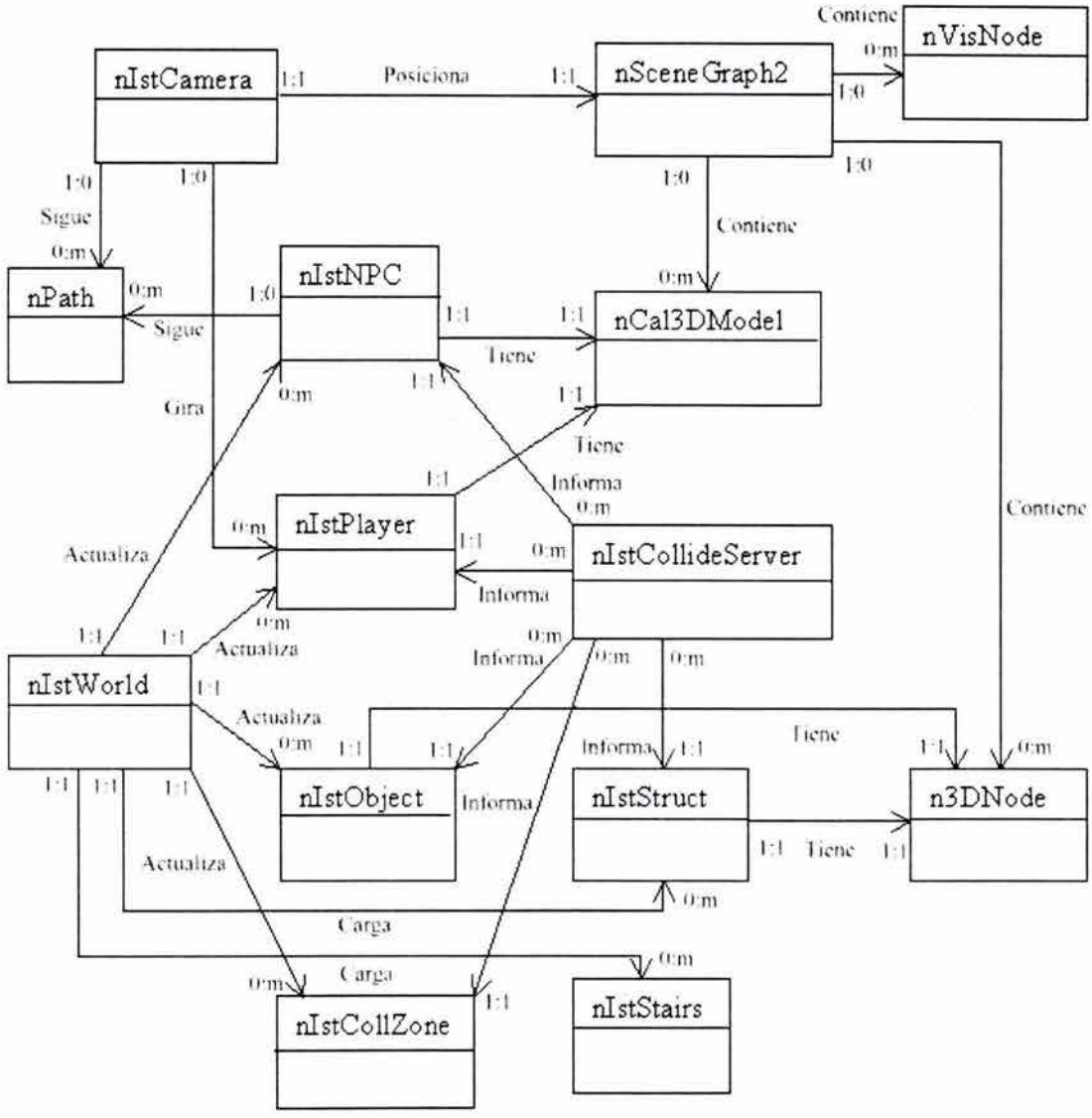


Figura 3.8. Modelo objeto – relación general.

Se considera conveniente hacer las siguientes observaciones, los objetos de las clases nIstCollZone y nIstStairs no tienen ninguna relación con alguna clase que se encargue del despliegue en pantalla, a diferencia de nIstStruct y nIstObject relacionados con la clase n3Dnode. Para la clase nIstStairs no es necesario tener alguna relación con el servidor de colisiones, la clase nIstPlayer recibe información de cuando se da una colisión con un objeto de la clase nIstStairs para realizar el comportamiento adecuado.

3.3.5. Modelo objeto – comportamiento.

El modelo CRC y el de objeto-relación representan elementos estáticos del modelo de análisis OO. Ahora se representa el comportamiento dinámico del sistema o producto OO. Para hacer esto se representa el comportamiento del sistema como una función de eventos específicos y tiempo.

El modelo objeto-comportamiento indica cómo responderá un sistema OO a eventos o estímulos externos. La forma como se modelaron estos comportamientos fue mediante diagramas en los que se muestran los estados activos (aquellos en los que se realiza una transformación continua o proceso) de cada objeto y los eventos (a veces conocidos como disparadores o triggers) que causan la transición entre estos estados activos. A continuación se presentan los diagramas de transiciones para varias de las clases.

Modelo objeto – comportamiento de la clase nIstPlayer. En la figura 3.9 se muestra un diagrama de transiciones donde sólo se presentan estados activos relacionados con el movimiento. El comportamiento de esta clase debe de modificarse, además de los estados que ya tiene la clase (avanzar, detenido, retroceder, girar a la derecha e izquierda) se deben de agregar dos estados, el avance lateral tanto a la izquierda como a la derecha, es decir estos nuevos comportamientos deben permitir que el personaje que controla el usuario pueda moverse lateralmente, con un ángulo de noventa grados con respecto a la dirección en la que está observando.

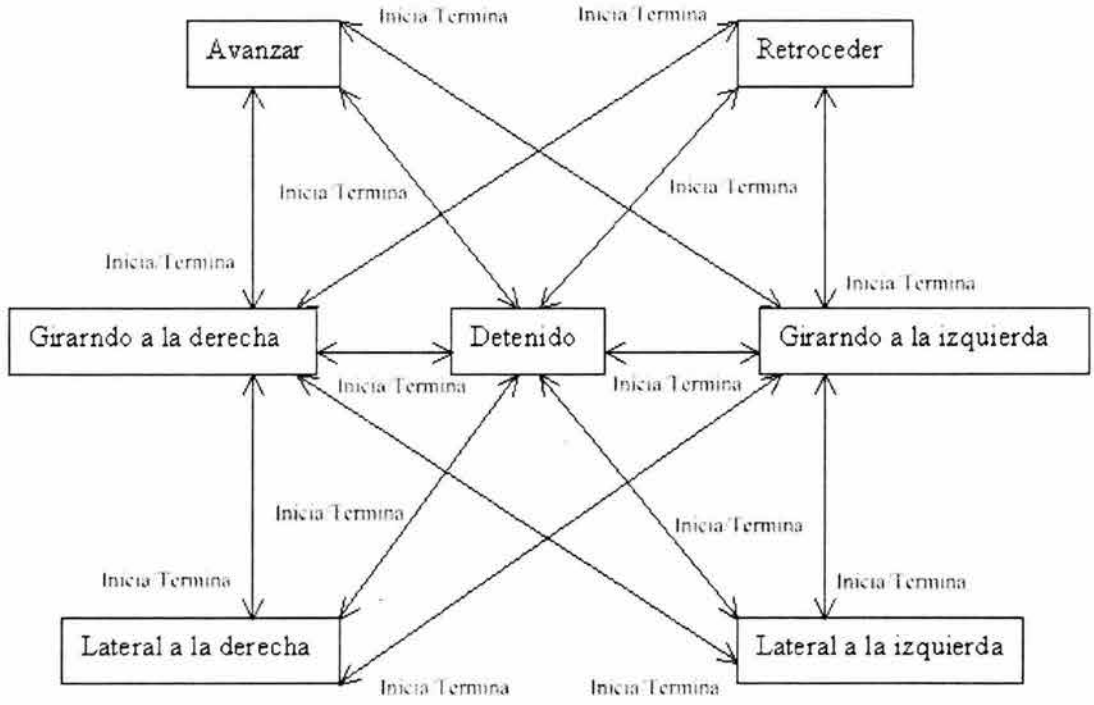


Figura 3.9. Modelo objeto –comportamiento de la clase nIstPlayer.

Los comportamientos de los personajes no se reducen a los movimientos que estos pueden realizar, también existen comportamientos asociados con las colisiones que ocurren dentro del mundo virtual con las demás clases. En la figura 3.10 se ilustra esta serie de comportamientos. En el modelo hace falta especificar que tipo de colisión es la que produce el comportamiento indicado, por ejemplo el personaje pasa de un estado de caída libre a uno en donde se encuentra en equilibro cuando se da una colisión con un objeto de la clase nIstStruct o que el personaje pasa del movimiento horizontal al equilibrio cuando se da una colisión con un objeto de la clase nIststruct, nIstObject o nIstNPC.

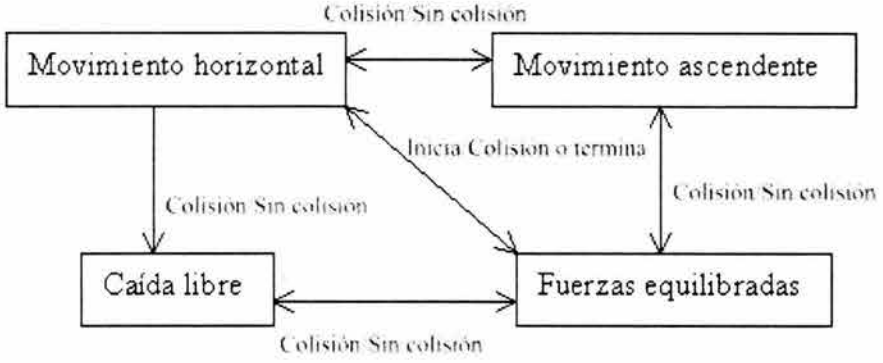


Figura 3.10. Modelo objeto –comportamiento de la clase nIstPlayer.

Se podrían presentar otros modelos de comportamientos asociados con la clase nIstPlayer pero en su mayoría estos estados de la clase están relacionados con la animación que realiza el objeto y la acción que está llevando a cabo.

Modelo objeto – comportamiento de la clase nIstCamera. El modelo que se presenta en la figura 3.11 muestra el comportamiento que debe tener la clase cuando ésta cambia entre los diferentes tipos de cámara. Como ya se menciono [3.2.3] es necesario hacer ciertas adaptaciones a la clase nIstCamera para que pueda ser reusada dentro del sistema en desarrollo, en el modelo objeto – comportamiento que se muestra no son visibles las modificaciones en el comportamiento de la clase, debido a que estos cambios están relacionados con el comportamiento de la cámara cuando ésta ya se encuentra en el modo de vista de primera persona.

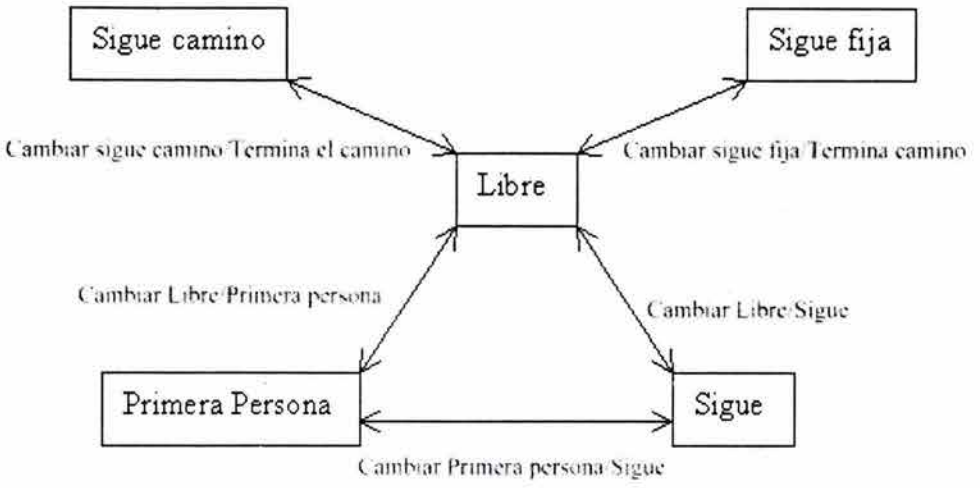


Figura 3.11. Modelo objeto –comportamiento de la clase nIstCamera.

Además de los comportamientos que presenta la clase nIstCamera al cambiar entre los diferentes tipos de cámara esta clase tiene otros modelos de comportamiento los cuales se observan cuando la cámara está en un estilo particular de cámara, ese es el caso de la cámara libre cuyo modelo objeto – comportamiento se muestra en la figura 3.12. Para que esta serie de comportamientos se lleven a cabo es necesario que la cámara esté en el modo de cámara libre.

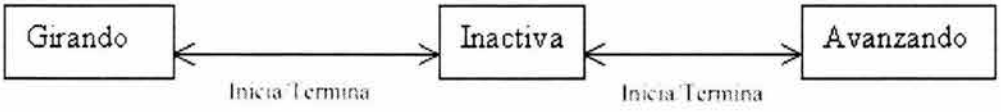


Figura 3.12. Modelo objeto –comportamiento de la clase nIstCamera.

Al igual que el modelo de comportamientos mostrado en la figura 3.12. existe un diagrama de comportamientos cuando la cámara está en vista de primera persona (FPS) y cuando está en el modo de seguir a un objeto del mundo virtual (chase[5]).

[5] Palabra en ingles cuya traducción es seguir, aplicado al ámbito del manejo de cámaras indica que la cámara sigue a un objeto o personaje a cierta distancia y con un ángulo de inclinación dado por lo que la cámara debe de mantener la misma velocidad del objeto que sigue, se debe sobre entender que para observar el funcionamiento de una cámara que sigue el objeto sobre el cual se enfoca debe estar en movimiento.

Modelo objeto – comportamiento de la clase nIstNPC. Los comportamientos que presentan los personajes controlados por el sistema pueden ser muy variados y complejos, no sólo eso también dichos comportamientos en general deben ser diferentes entre todos los personajes, pensar en programar todos estos comportamientos no es factible debido a que esto haría que la clase nIstNPC fuera muy pesada limitando el número de personajes controlados por el sistema así como el número de comportamientos de cada objeto, de ahí la importancia de que los NPC's y otros objetos puedan llamar a ejecutar scripts en los cuales se indiquen estos comportamientos especializados para cada objeto en particular. [ver figura 3.13.]

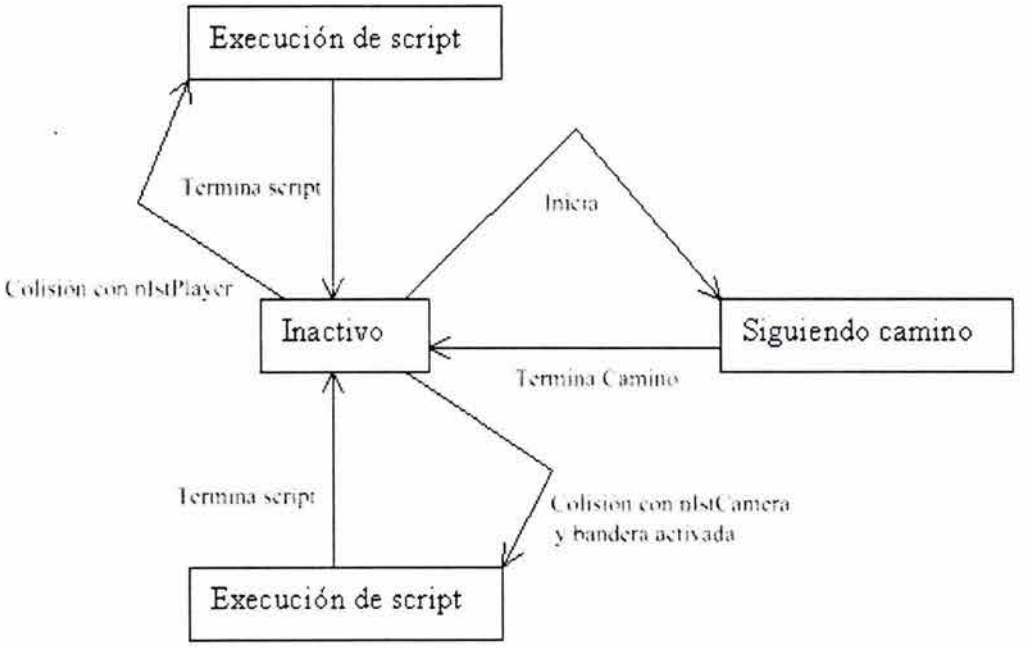


Figura 3.13. Modelo objeto –comportamiento de la clase nIstNPC.

En el modelo original de la clase nIstNPC la llamada a ejecución de un script sólo se llevaba a cabo cuando ocurría una colisión con un objeto de la clase nIstPlayer, en este nuevo modelo la ejecución de scripts también se puede llevar acabo al existir una colisión con la cámara y revisar que una bandera esté activada, este comportamiento corresponde físicamente al de observar al personaje controlado por el sistema y presionar la tecla o botón indicado para interactuar.

Modelo objeto – comportamiento de las clases nIstObject y nIstStruct. Originalmente los modelos objeto – comportamiento de estas clases era muy diferente, se ha decidido cambiar dichos modelo por el que se muestra en la figura 3.14. Este nuevo modelo ofrece una mayor interactividad con el usuario, también se hace uso del recurso de la ejecución de scripts debido a la gran flexibilidad que estos ofrecen en cuanto a la programación de comportamientos, por ejemplo al hacer uso de los scripts el comportamiento que resulte de la interacción del usuario con el objeto puede afectar no sólo a los objetos involucrados sino que podría alterarse todo lo que hay en el mundo virtual. Al igual que la clase nIstNPC la ejecución de scripts no sólo se da cuando existe una colisión con un personaje sino que también es posible revisar una bandera cuando ocurre una colisión con la cámara.

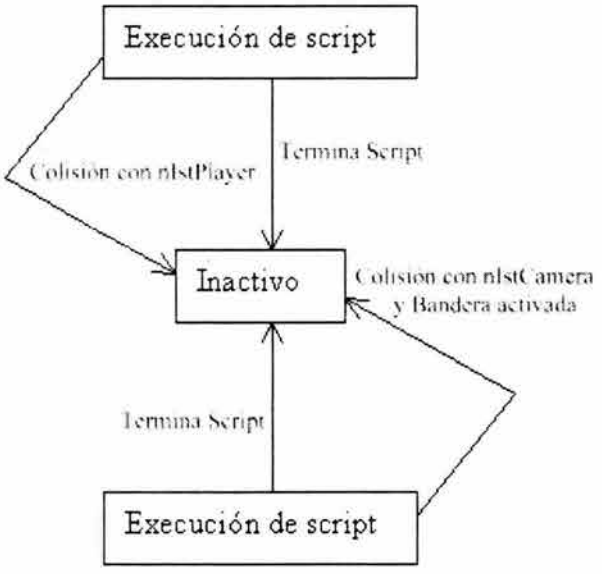


Figura 3.14. Modelo general para las clases nIstObject y nIstStruct.

Capítulo 4
Diseño orientado a objetos

4.1. Introducción

El diseño orientado a objetos (DOO) busca transformar el modelo de análisis creado usando el análisis orientado a objetos en un modelo de diseño el cual sirve como un anteproyecto para la construcción del software, el diseño orientado a objetos constituye un tipo de diseño que logra un cierto número de diferentes niveles de modularidad. Al programar en un lenguaje OO los componentes principales están organizados en módulos denominados subsistemas. En suma el diseño orientado a objetos debe describir la organización de datos específicos de atributos y los detalles procedimentales de las operaciones individuales.

La naturaleza única del diseño orientado a objetos descansa en su capacidad de apoyarse en cuatro conceptos importantes en el diseño de software: abstracción, ocultación de la información, independencia funcional y modularidad.

4.2. Proceso de diseño del sistema

En las siguientes secciones se analizan en detalle las actividades relacionadas con el diseño del sistema.

4.2.1. Partición del modelo de análisis.

En el diseño de sistemas OO se divide el modelo de análisis para definir colecciones cohesivas de clases, relaciones y comportamientos. Estos elementos del diseño se empaquetan como un subsistema.

En general, todos los elementos de un subsistema comparten alguna propiedad en común. Pueden estar envueltos en la realización de la misma función; pueden residir dentro del mismo producto de hardware; o pueden manejar la misma clase de recursos. Los subsistemas se caracterizan por sus responsabilidades; esto es, un subsistema puede identificarse por los servicios que realiza. Al usarse dentro del contexto del diseño de un sistema OO, un *servicio* es una colección de operaciones que realizan una función específica.

Una vez que se ha definido lo que es un subsistema y como es posible identificarlo se presentan a continuación los principales subsistemas que se encuentran en el desarrollo de Calakmul virtual, se considera conveniente recordar que muchos de los subsistemas que necesita la aplicación ya son proporcionados íntegramente por el motor de juegos Nebula.

Subsistema gráfico. La responsabilidad del subsistema gráfico es la de mandar a dibujar en pantalla la escena que se ha cargado, esto implica que el subsistema debe proporcionar el soporte de cómo cargar todos los elementos de la escena, en que posición se encuentra cada uno de estos elementos, como tienen que ser dibujados, etc. Este subsistema en particular debe de mantener comunicación muy variada con otros subsistemas, por ejemplo del subsistema de scripts debe recibir comandos en tiempo de ejecución, con el subsistema de archivos debe de tener la comunicación necesaria para que, por ejemplo, pueda cargar una textura.

Subsistema de scripts. Este subsistema tiene la responsabilidad de leer comandos de un lenguaje de scripts y traducirlos en llamadas a métodos en tiempo de ejecución, así como de la interpretación del lenguaje en el que se encuentran los script necesaria para obtener la información que contienen los scripts. Una parte muy importante de este subsistema consiste en la capa de comunicación que éste debe de tener con las clases que permitan el uso de comandos.

Subsistema reloj. La responsabilidad conjunta de este subsistema es como su nombre lo indica, ofrecer la funcionalidad de un reloj, llevar una cuenta del tiempo desde que se inicio el programa, la finalidad de este sistema no es sólo sincronizar o disparar eventos, en sistemas de graficación en tiempo real estos subsistemas son necesarios para conocer intervalos de tiempo entre sucesos, llamadas de funciones o métodos. La comunicación principal de este subsistema con otros es simple y sencillamente la de dar a conocer el tiempo actual, además son necesarias algunas otras peticiones como la de reiniciar el reloj.

Subsistema de administración de archivos. La función principal de este subsistema es la de ofrecer un manejo de archivos por medio del cual otros subsistemas puedan navegar a través del árbol de directorios del sistema operativo, con la finalidad de que estos puedan acceder a los archivos que ahí se encuentren. Otros subsistemas pueden pedirle al subsistema administrador de archivos cambiar del directorio de trabajo o que se les informe sobre cual es el directorio de trabajo actual.

Subsistema de administración de entradas al sistema. Este subsistema está encargado de informar a los subsistemas que así lo requieran sobre las entradas que se le han dado al sistema a través de dispositivos periféricos como el teclado o ratón. Proporciona un mapeo de entradas con posibles eventos.

Subsistema de cálculos matemáticos. La responsabilidad conjunta de este subsistema es la de implementar una librería de operaciones matemáticas que son usadas frecuentemente dentro del sistema, donde otros subsistemas le piden que se realice cierta operación y el subsistema de cálculos matemáticos da como respuesta el resultado de la operación requerida.

Subsistema de audio. De manera similar al subsistema gráfico el subsistema de audio tiene la responsabilidad de reproducir todos los sonidos que se encuentran en la escena, así como de realizar todos los cálculos necesarios para que la reproducción del audio sea la adecuada, por ejemplo que el sonido que emite un ave a lo lejos se escuche con un volumen más bajo que el producido por un ave enfrente del observador. La comunicación que establece con otros subsistemas en general es igual a la que establece el subsistema gráfico la diferencia radica en la finalidad de esta comunicación.

De manera general estos son los principales subsistemas presentes en el proyecto Calakmul virtual, debido a que el sistema tiene grandes semejanzas con sistemas conocidos como videojuegos, de ahí que se haya optado por un motor de juegos para su desarrollo, es que casi todos estos subsistemas ya están implementados por Nebula. Tanto sus funciones

como relaciones con otros subsistemas han sido creadas con la finalidad de hacer más fácil la programación de este tipo de sistemas. El modelo que maneja Nebula para hacer uso de estos subsistemas es por medio de servidores [ver 1.5.1], en donde objetos conocidos como servidores se encargan de proporcionar el uso de los subsistemas a los objetos de Nebula que los requieran.

4.2.2. El componente para la gestión de datos.

La gestión de datos abarca dos áreas distintas: la gestión de datos críticos para la propia aplicación, y la creación de una infraestructura para el almacenamiento y recuperación de objetos. En general, la gestión de datos se diseña por capas. La idea es aislar los requisitos de bajo nivel para la gestión de datos de los de alto nivel para la gestión de los atributos del sistema.

Dentro del contexto del sistema, un sistema de gestión de datos se usa a menudo como un almacén común de datos para todos los subsistemas. Los objetos requeridos para manipular esta base de datos son miembros de las clases reusables identificadas usando el análisis de dominio.

Otra ventaja de usar el motor de juegos Nebula es que el componente para la gestión de datos ya ha sido desarrollado y abarca las dos áreas, la gestión de datos críticos está a cargo del subsistema de administración de archivos, como se menciono [4.2.1] este subsistema permite a Nebula navegar por el sistema de archivos, abrir y guardar archivos, estos archivos representan en su mayoría el dominio de los datos para las clases de la aplicación, por ejemplo la clase nMeshNode usa este subsistema para abrir los archivos en los que se encuentran los modelos 3D, cada clase interpreta de manera diferenta la información contenida en los archivos.

El almacenamiento y recuperación de objetos está cubierto por nNode, nHashNode y nRoot, dentro de estas clases Nebula proporciona los métodos para que los objetos puedan ser creados y accesados dentro de una estructura de datos la cual asemeja a un árbol, donde cada instancia de una clase, es decir cada objeto, está representado por un nodo dentro del árbol. Dada esta estructura de datos estas clases también ofrecen las funciones para recorrer dicho árbol, así pues todos los objetos que heredan de estas clases tienen la posibilidad de formar parte de la infraestructura que tiene Nebula para almacenar y recuperar objetos.

4.2.3. El componente de interfaz hombre-máquina.

Aunque el componente interfaz hombre-máquina (IHM) se implementa dentro del contexto del dominio del problema, la interfaz por sí misma representa un subsistema de importancia crítica para la mayoría de las aplicaciones modernas. El modelo de análisis OO contiene escenarios de uso (llamados casos de uso) y una descripción del papel que tienen los usuarios (llamados actores) al interactuar con el sistema. Esto sirve como dato de entrada al proceso de diseño de la IHM.

De manera semejante al componente para la gestión de datos, Nebula proporciona al programador un subsistema de interfaz hombre – máquina basado en un lenguaje para la

programación de interfaces gráficas conocido como TK[1], este lenguaje no fue desarrollado por Nebula y no es parte del motor en un estricto sentido, ya que la distribución de Nebula no contiene ningún código de TK para compilar, es necesario instalar TK para usarlo con Nebula, lo que Nebula hace es usar un dll por medio del cual puede acceder a TK y crear una interfaz. Por medio de TK se puede hacer una interfaz con casi todas las características de una interfaz de Windows, es decir con botones, ratiobuttons, checklists, etiquetas, cuadros de texto, etc. Esto da una gran flexibilidad en la programación de la interfaz hombre – máquina. La interfaz de TK puede comunicarse con la aplicación desarrollada sobre Nebula por medio del uso de comandos de TCL, los cuales son interpretados por el servidor de scripts.

4.3. Proceso de diseño de objetos

El objetivo de este punto es especificar el propósito de cada clase así como los mecanismos que la conectan con otras clases, se proporcionan los detalles necesarios para construir cada clase.

En el contexto del diseño orientado a objetos, el diseño de objetos se centra en las «clases». Se debe de desarrollar un diseño detallado de los atributos y operaciones que incluye cada clase, y una especificación completa de los mensajes que conectan la clase con sus colaboradores.

4.3.1. Descripciones de los objetos.

La descripción de un objeto puede tomar dos formas posibles.

Una descripción del protocolo en la que se establece la interfaz de un objeto definiendo cada mensaje que el objeto puede recibir y la correspondiente operación que el mismo ejecuta al recibir el mensaje y una descripción de la implementación que muestra detalles de ella para cada operación implicada por un mensaje que se pasa al objeto. Los detalles de implementación incluyen información acerca de la parte privada del objeto, esto es, detalles internos acerca de las estructuras de datos que describen los atributos del objeto y detalles procedimentales que describen las operaciones.

A continuación se hace una descripción del protocolo para las clases que se han identificado dentro del análisis así como de las clases que necesitan una modificación de su protocolo.

nIstPlayer

MENSAJE (personaje) - StartFW: MODIFICA banderas isFW, isSTPFW, isSTPBW, CAMBIA la animación del personaje;
MENSAJE (personaje) - StartBW: MODIFICA banderas isBW, isSTPFW, isSTPBW, CAMBIA la animación del personaje;
MENSAJE (personaje) - StartRL: MODIFICA la bandera isRL, CAMBIA la animación

[1] TK es una extención de TCL la cual le proporciona al programador un sistema de ventanas para la creación de interfaces hombre – máquina.

del personaje;
MENSAJE (personaje) - StartRR: MODIFICA la bandera isRR, CAMBIA la animación del personaje;
MENSAJE (personaje) - StartSL: MODIFICA banderas isSL, isSTPSL, isSTPSR, CAMBIA la animación del personaje;
MENSAJE (personaje) - StartSR: MODIFICA banderas isSR, isSTPSL, isSTPSR, CAMBIA la animación del personaje;
MENSAJE (personaje) - StartMRL: MODIFICA la bandera isMRL;
MENSAJE (personaje) - StartMRR: MODIFICA la bandera isMRR;
MENSAJE (personaje) – StopFW: MODIFICA banderas isFW, isSTPFW, CAMBIA la animación del personaje;
MENSAJE (personaje) – StopBW: MODIFICA banderas isBW, isSTPBW, CAMBIA la animación del personaje;
MENSAJE (personaje) – StopRL: MODIFICA la bandera isRL, CAMBIA la animación del personaje;
MENSAJE (personaje) – StopRR: MODIFICA la bandera isRR, CAMBIA la animación del personaje;
MENSAJE (personaje) – StopSL: MODIFICA banderas isSL, isSTPSL, CAMBIA la animación del personaje;
MENSAJE (personaje) – StopSR: MODIFICA banderas isSR, isSTPSR, CAMBIA la animación del personaje;
MENSAJE (personaje) - StopMRL: MODIFICA la bandera isMRL;
MENSAJE (personaje) - StopMRR: MODIFICA la bandera isMRR;
MENSAJE (personaje) - ActShoot: CAMBIA la animación del personaje;
MENSAJE (personaje) – SetRotSpeed: COFIGURA la velocidad de rotación del personaje;
MENSAJE (personaje) – SetMaxRunSeed: CONFIGURA la velocidad máxima a la que puede moverese el personaje;
MENSAJE (personaje) – SetRunAcel: CONFIGURA la aceleración del personaje;
MENSAJE (personaje) – GetRotSpeed: REGRESA la velocidad de rotación del personaje;
MENSAJE (personaje) – GetMaxRunSeed: REGRESA la velocidad máxima a la que puede moverese el personaje;
MENSAJE (personaje) – GetRunAcel: REGRESA la aceleración del personaje;
MENSAJE (personaje) – Setn3DNodeTar: ESTABLECE cual es el nodo para el personaje;
MENSAJE (personaje) – SetnCalModTar: ESTABLECE el modelo que representa a el personaje;
MENSAJE (personaje) – Getn3DNodeTar: RETRORNA cual es el nodo para el personaje;
MENSAJE (personaje) – GetnCalModTar: REGRESA el modelo que representa a el personaje;
MENSAJE (perosnaje) – Trigger: ACTUALIZA el estado interno del personaje;
MENSAJE (personaje) – Collide: MANEJA el comportamiento del personaje frente a las colisiones que ocurran;
MENSAJE (personaje) – UpdatePosition: MANEJA atributos del objeto relacionados con el movimiento del personaje, ACTUALIZA la posición del personaje;

nIstNPC

MENSAJE (NPC) - Trigger: ACTUALIZA el estado interno del NPC ;
MENSAJE (NPC) – Collide: MANEJA los eventos del NPC frente a las colisiones que ocurran;
MENSAJE (NPC) – AnimWalk: CAMBIA la animación del NPC;
MENSAJE (NPC) – AnimStrut: CAMBIA la animación del NPC;
MENSAJE (NPC) – AnimIdle: CAMBIA la animación del NPC;
MENSAJE (NPC) – SetRotSpeed: COFIGURA la velocidad de rotación del NPC;
MENSAJE (NPC) – SetRunSeed: CONFIGURA la velocidad a la que se mueve el NPC;
MENSAJE (NPC) – GetRotSpeed: REGRESA la velocidad de rotación del NPC;
MENSAJE (NPC) – GetRunSeed: REGRESA la velocidad a la que se mueve el NPC;
MENSAJE (NPC) – Setn3DNodeTar: ESTABLECE cual es el nodo para el NPC;
MENSAJE (NPC) – SetnCalModTar: ESTABLECE el modelo que representa a el NPC;
MENSAJE (NPC) – Getn3DNodeTar: RETRORNA cual es el nodo para el NPC;
MENSAJE (NPC) – GetnCalModTar: REGRESA el modelo que representa a el NPC;
MENSAJE (NPC) – EnableEventColl: MODIFICA una bandera que habilita la execución de scripts cuando sucede una colisión con el NPC;
MENSAJE (NPC) – DisableEventColl: MODIFICA una bandera que deshailita la execución de scripts cuando sucede una colisión con el NPC;
MENSAJE (NPC) – GetLastCollEntity: REGRESA el último objeto con el que se dio una colisión;
MENSAJE (NPC) – SetStatePathSeeking: COLOCA al NPC en modo para seguir un camino en el tiempo determinado;
MENSAJE (NPC) – IsPathSeeking: REGRESA el valor de la bandera StatePtSk;
MENSAJE (NPC) – Initialize: Agrega los scripts que van a ser llamados por el NPC;

nIstObject

MENSAJE (objeto) – Initialize: Agrega los scripts que van a ser llamados por el objeto;
MENSAJE (objeto) - Trigger: ACTUALIZA el estado interno del objeto ;
MENSAJE (objeto) – Collide: MANEJA los eventos del objeto frente a las colisiones que ocurran;
MENSAJE (objeto) – CreateModel: ESTABLECE cual es el nodo para el objeto, ESTABLECE el modelo que representa a el objeto, ESTABLECE el sombreado del objeto, Establece la textura del objeto;
MENSAJE (objeto) – EnableEventColl: MODIFICA una bandera que habilita la execución de scripts cuando sucede una colisión con el objeto;
MENSAJE (objeto) – DisableEventColl: MODIFICA una bandera que deshailita la execución de scripts cuando sucede una colisión con el objeto;
MENSAJE (NPC) – GetLastCollEntity: REGRESA el último objeto con el que se dio una colisión;

nIstStrct

MENSAJE (estructura) - Trigger: ACTUALIZA el estado interno de la estructura;
MENSAJE (estructura) – Collide: MANEJA los eventos de la estructura frente a las colisiones que ocurran;

nIstCamera

MENSAJE (cámara) - Trigger: ACTUALIZA el estado interno de la cámara ;
MENSAJE (cámara) – Collide: MANEJA los eventos de la cámara frente a las colisiones que ocurran;

MENSAJE (cámara) – Collide: MODIFICA la posición de la cámara de acuerdo a las coordenas X,Y, Z;
MENSAJE (cámara) – SetStyleChase: HACE que la cámara siga a un objetivo, ESTABLECE la distancia a la cual sigue al objetivo, ESTABLECE la altura, ESTABLECE el ángulo de inclinación;
MENSAJE (cámara) – SetStyleLockedChase: HACE que la cámara siga a un objetivo, ESTABLECE la distancia a al objetivo, ESTABLECE la orientacion de la cámara sobre X, Y, Z;
MENSAJE (cámara) – SetStyleAimPath: INDICA cual es el objetivo que debe de observar la cámara, ESTABLECE cual es el camino que debe recorrer la cámara, MODIFICA la posición de la cámara de acuerdo al camino que recorre, MODIFICA la orientacion de la cámara para que ésta siempre observe el objetivo, ESTABLECE el tiempo en el que se debe de recorrer el camino;
MENSAJE (cámara) – SetStyleLookPath: ESTABLECE la orientacion de la cámara, ESTABLECE cual es el camino que debe recorrer la cámara, MODIFICA la posición de la cámara de acuerdo al camino que recorre, ESTABLECE el tiempo en el que se debe de recorrer el camino;
MENSAJE (cámara) – SetStyleFPS: INDICA cual es el objetivo desde el cual se observa, ESTABLECE la altura de la cámara;
MENSAJE (cámara) – SetStyleFree: ESTABLECE el modo de cámara libre;
MENSAJE (cámara) – EnableSelection: MODIFICA el valor de la bandera EnSelection;
MENSAJE (cámara) – handleInput: MANEJA las entradas que da el usuario para controlar la cámara;
MENSAJE (cámara) – getQ: REGRESA el valor del quaternion[2] de la cámara;

---

[2] Un Quaternion representa una rotación esférica en tres dimensiones como un vector de cuatro componentes y de lungitud unitaria.

$$q = [n_x*\sin(\theta/2)\ n_y*\sin(\theta/2)\ n_z*\sin(\theta/2)\ \cos(\theta/2)],$$

MENSAJE (cámara) – getT: REGRESA la posición de la cámara;
MENSAJE (cámara) – Qxyzw: ESTABLECE el quaternion de la cámara;
MENSAJE (cámara) – EnableEventColl: MODIFICA una bandera que habilita la ejecución de scripts cuando sucede una colisión con la cámara;
MENSAJE (cámara) – DisableEventColl: MODIFICA una bandera que deshailita la ejecución de scripts cuando sucede una colisión con la cámara;
MENSAJE (cámara) – GetLastCollEntity: REGRESA el último objeto con el que se dio una colisión;
MENSAJE (cámara) – handleViewer: CONTROLA el comportamiento de la cámara libre;
MENSAJE (cámara) – handleChaseViewer: CONTROLA el comportamiento de la cámara en modo de seguir;
MENSAJE (cámara) – handleFPSViewer: CONTROLA el comportamiento de la cámara en el modo "vista de primera persona";

nIstWorld

MENSAJE (mundo) - Trigger: ACTUALIZA el estado interno del mundo así como de todos los objetos que contiene ;
MENSAJE (mundo) - GetGravity: REGRESA el valor de la gravedad en el mundo virtual;
MENSAJE (mundo) - SetGravity: FIJA el valor de la gravedad en el mundo virtual;
MENSAJE (mundo) - AddCollideBSPNode: CARGA al servidor de colisiones la malla de colisión que se encuentre en el archivo collmesh.n3d el cual contiene una representación de todas las estructuras con las que puede darse una colisión;
MENSAJE (mundo) - AddCollideStairNode: CARGA al servidor de colisiones la malla de colisión que se encuentre en el archivo collstairs.n3d el cual contiene una representación de todas las escaleras con las que puede darse una colisión;
MENSAJE (mundo) - NewCollideObject: CREA un nuevo objeto con el que se puede dar una colisión;
MENSAJE (mundo) - NewCollideShape: CARGA una nueva malla de colisión desde el archivo indicado;

nIstStairs

MENSAJE (escalera) - Trigger: ACTUALIZA el estado interno de la escalera;
MENSAJE (escalera) – Collide: MANEJA los eventos de la escalera frente a las colisiones que ocurran;

Con q*q=1, el vector n= [n_x n_y n_z] es un vector de tres componentes de longitud unitaria: n*n=1. El vector unitario n especifica el eje de rotación. El ángulo de rotación sobre ese eje es θ y sigue la regla de la mano derecha. La repesentación ángulo – eje de la rotación es [nθ].

nIstCollZone

MENSAJE (zona de colisión) – Initialize: Agrega los scripts que van a ser llamados por la zona de colisión;
MENSAJE (zona de colisión) - Trigger: ACTUALIZA el estado interno de la zona de colisión ;
MENSAJE (zona de colisión) – Collide: MANEJA los eventos que presenta la zona de colisión frente a las colisiones que ocurran;
MENSAJE (zona de colisión) – CreateZone: ESTABLECE cual es el nodo para la zona de colisión, ESTABLECE el modelo que representa a la zona de colisión,
MENSAJE (zona de colisión) – EnableEventColl: MODIFICA una bandera que habilita la execución de scripts cuando sucede una colisión con la zona;
MENSAJE (zona de colisión) – DisableEventColl: MODIFICA una bandera que deshailita la execución de scripts cuando sucede una colisión con la zona;
MENSAJE (zona de colisión) – GetLastCollEntity: REGRESA el último objeto con el que se dio una colisión;

A continuación se hace una descriprción de la implementación para las clases que se han identificado dentro del análisis así como de las clases que necesitan modificar su implementación. En la descripción de la implementación sólo se detallan las operaciones principales de cada clase.

Clase:

nIstPlayer

Método:

nada Trigger (delta de tiempo)

Como se menciono en la descripción del protocolo la operación que realiza este método es la de actualizar el estado interno del personaje, esta operación se describe en una línea, pero para explicar su función se requiere de varios renglones. Para empezar este método debe de ser llamado dentro de cada ciclo de programa y el parámetro delta de tiempo que requiere su llamada debe ser el intervalo de tiempo transcurrido desde la última vez que se llamo al método.

Además se actualiza la posición de la malla de colisión del personaje, se llama al método de actulización de la clase nIstEntity el cual tiene la tarea de manejar fuerzas como la gravedad. Al encargarse del estado del personaje este método sabe por medio de banderas y una sencilla maquina de estados, una serie de comparaciones, cual es la entrada que ha proporcionado el usuario, si una bandera como isRR es activada, en este caso esta bandera estará activada cuando el usuario de la entrada de girar a la derecha, el método Trigger hace las operaciones necesarias para que en pantalla el personaje que controla el usuario gire hacia la derecha, en general primero se hace una revisión de otras banderas,

por ejemplo el personaje no puede comenzar a girar si se está deteniendo, se obtiene la posición o ángulo del nodo que contiene la malla del personaje y se hace la translación o rotación de éste. En algunos casos como los proporcionados por isSTPFW o isSTPBW se cambia la animación del personaje debido a que dentro de estos estados es posible pasar de la animación de correr a la de caminar, recordando que en los demás estados la animación se cambia en los métdos que modifican las banderas.

Método:

nada Collide (nada)

El método Collide implementa el comportamiento que debe de tener el personaje cuando ocurre una colisión, al igual que el método Trigger éste debe de ser llamado dentro de cada ciclo de programa, ahora se detalla que es lo que ocurre cada vez que se llama a este método, para empezar se obtiene el número de colisiones que hay con la malla de colisiones del personaje.

```
nCollideReport** report;
int num_colls = collideObject->GetCollissions(report);
```

De cada colisión reportada se obtienen las normales de los planos así como el nombre de la clase, con el nombre de la clase se hacen comparaciones y cuando se identifica la clase se ejecuta el código que controla el comportamiento del personaje, dentro de cada una de estas comparaciones se generan y modifican las fuerzas surgidas de las colisiones que afectan al personaje controlado por el usuario, aquí se programa que al chocar con un objeto o un NPC el personaje no pueda atravesarlos o que al chocar con una escalera el personaje reciba una fuerza ascendente que le permita subir a las estructuras, para poder programar todo esto son necesarios los vectores normales.

Clase:

nIstNPC

Método:

nada Trigger (delta de tiempo)

La misma idea que se uso para el método Trigger en la clase nIstPlayer funciona para la clase nIstNPC, el método debe ser llamado cada ciclo de programa, la diferencia radica en que los estados en los que se puede encontrar un personaje no son los mismos estados en los que se puede encontrar un NPC, de acuerdo al modelo objeto – comportamiento [3.3.5] son básicamente tres los estados en los que se puede encontrar un NPC, inactivo, siguiendo un camino o ejecuntando un script. Dentro de este método sólo es necesario saber si el NPC está siguiendo un camino, esto se hace mediante la comparación de una bandera, si esta condición se cumple se obtiene la posición actual del nodo 3D ligado con la malla del NPC, del camino se obtiene cual es la posición que debe tener el NPC , se rota y se translada al NPC, por último se llama al método Collide de la clase nIstNPC.

Método:

nada Collide (nada)

Collide implementa todos los comportamientos que puede realizar el NPC cuando ocurre una colisión, el método Collide para los NPC's no es tan detallado como lo es para el personaje controlado por el usuario, ya que para un NPC en caso de ocurrir una colisión el único comportamiento que se espera de éste es detenerse y no atravezar al objeto con el cual se reporta contacto, siempre que ocurre una colisión con un NPC se corre el script señalado por el método Initialize dejando la libertad de que por medio de este script se identifique la clase del objeto con el que hubo colisión así como de la programación de todos los comportamientos que se deben realizar.

Método:

nada Initialize (nada)

El método initialize agrega los scripts de TCL que deben ser ejecutados al cumplirse las condiciones requeridas por el programador, este método busca los archivos de script dentro de un directorio ya preestablecido, la ruta sobre la cual se buscan los scripts es "/sys/share/scriptlets/" a partir del directorio base sobre el cual está se ejecuta la aplicación, dentro del direcotrio scriptlets debe existir una carpeta cuyo nombre debe corresponder con el nombre del objeto, si el objeto se llama npc01 la ruta completa en la que se debe crear el archivo de escript sería "/sys/share/scriptlets/npc01/", por el momento el nombre del archivo de script está limitado al nombre de collscriptlet, este método no es exclusivo de la clase nIstNPC y prácticamente realiza la misma función para las demás clases que lo tienen, nIstObject, nIstCollZone, nIstStruct, nIstCamera, por lo que se considera redundante incluir estos métodos en la descripción de la implementación de estas clases.

Clase:

nIstWorld

Método:

nada Trigger (tiempo)

El método de actualización de la clase nIstWorld es muy importante, ya que dentro de éste se realiza la actualización de todas las entidades que existen dentro del mundo virtual, esto para evitar que dentro del ciclo principal de la aplicación se tenga que hacer la actualización de forma directa, este método tiene la responsabilidad de informar a los demás métodos de actualización el intervalo de tiempo transcurrido desde la última vez que se les llamo. Para obtener esta información dentro del ciclo principal se debe tener algo similar al código que a continuación se presenta.

```
ks->ts->Trigger();
double t = ks->ts->GetFrameTime();
        :
        :
istworld->Trigger((float)t);          //Dentro de istworld se actualizan todas las entidades
```

Posteriormente el método realiza una resta entre el nuevo valor indicado por el servidor de tiempo y el dato anterior que se guardo, el resultado es la delta de tiempo que necesitan los demás métodos de actualización, por último dentro del método Trigger se llama al método UpdateEntities este método es quien verdaderamente se encarga de llamar a los métodos de actualización de todos los objetos que se encuentren dentro del mundo virtual.

Método:

nada UpdateEntities (nada)

El método UpdateEntities busca todos los hijos de nIstEntity y los actualiza al invocar sus métodos Trigger y Collide, para obtener todos los objetos que heredan de nIstEntity el método contiene un ciclo en el que un apuntador obtiene la referencia de los objetos hijos de nIstEntity que se han creado en el mundo virtual, una vez que se tiene este apuntador sólo es cuestión de llamar a los métodos Trigger y Collide.

```
entity->Trigger(dt);
entity->Collide();
```

Método:

nada AddCollideBSPNode (ubicación de la malla de colisión)

El método AddBSPNode se encarga de cargar la malla de colisión para las estructuras del mundo virtual, así que básicamente este método le indica al servidor de colisiones cual es y donde se encuentra la malla que representa a las estrcturas, también le indica al servidor de colisiones la clase a la cual pertenece esta malla, en el caso del método AddBSPNode la clase siempre es nIstStruct. Se crea un nuevo objeto en el contexto de colisiones, se crea un objeto de la clase nIstStruct dentro del directorio de Nebula llamdo "/world/iststruct", hay que señalar que este método carga la malla de colisiones que se le indica en la ubicación que requiere como parámetro, pero por el momento el nombre del archivo debe de ser collmesh.n3d. Este método está diseñado para ser llamado al momento de cargar la escena, lo que se busca es que sólo sea una malla de colisiones para todas las estructuras de la escena, ya que encargarse de una sola malla resulta mucho más efeciente que cargar una malla de colision para cada estructura.

Método:

nada AddCollideStairNode (ubicación de la malla de colisión)

El método AddStairNode tiene la misma función que AddBSPNode, tiene la función de cargar la malla de colisión para las escaleras del mundo virtual, la implementación del método es la misma que AddBSPNode ya que las operaciones que se realizan son las mismas, la diferencia que existe entre ambos métodos radica en la clase para la que se agrega la malla de colisión, AddBSPNode agrega un objeto de la clase nIstStruct, AddCollideStairNode agrega un objeto de la clase nIstStairs, el método funciona

igual que AddCollideBSPNode, carga la malla de colisiones que se le indica en la ubicación que requiere como parámetro, pero por el momento el nombre del archivo debe de ser collstairs.n3d.

Clase:

nIstObject

Método:

nada Trigger (delta de tiempo)

Actualizar el estado interno de los objetos resulta más sencillo que actualizar el estado interno del personaje o los NPC's, debido a que los objetos sólo pueden ejecutar un script [3.3.5], por este motivo el método de actualización de la clase nIstObject sólo debe de actualizar la posición de la malla de colisión del objeto previniendo el caso en el que debido a la execución de un script éste halla modificado su posición.

Método:

nada CreateModel (malla, textura, iluminación, posición en el mundo virtual)

Este método busca implementar una forma sencilla de agregar objetos al mundo virtual, la idea es que poder llamar este método através de la consola o scripts de TCL para cargar los objetos que sean necesarios, los parámetros malla y textura contienen el nombre de los archivos desde los que se carga la malla y la textura del objeto, el parámetro iluminación es una bandera que indica si la luz del mundo virtual afecta al objeto, la posición está dada por la serie de coordenadas (X,Y,Z).

CreateModel crea un nodo 3D (n3DNode) dentro del directorio de Nebula "/usr/scene", el nombre que se le da a este nodo corresponde con el nombre del objeto por eso no es necesario pasar un parámetro que defina el nombre del nodo 3D, dentro de éste se crean los nodos necesarios para que un objeto sea visible en del ciclo de dibujo de Nebula, es decir un nodo de malla (nMeshNode) en el cual se carga el archivo que contiene el modelo 3D del objeto, un nodo de textura (nTexArrayNode) en el cual se carga el archivo que contiene la textura que se le debe de aplicar al objeto y un nodo de sombreado (nShaderNode) en el que se programan los llamados shaders[3], para todos los objetos que se creen estos tres nodos serán llamdos de la misma manera: mesh, tex y shader. Por último el método también carga y da de alta la malla de colisiones del objeto.

Método:

nada Collide (nada)

Collide implementa todos los comportamientos que puede realizar un objeto cuando ocurre una colisión, el método Collide para los objetos no es tan complicado como los

[3] Un shader consiste en una serie de algoritmos que describen matemáticamente como son dibujados los materiales individualmente sobre un objeto y como las fuentes de luz interectuan para modificar su apariencia en general.

métodos Collide para personajes o NPC's, en caso de darse una colisión el único comportamiento que se espera de un objeto es la ejecución de un script, siempre que ocurre una colisión con un objeto se corre el script indicado por el método Initialize dejando la libertad de que a través de este script se identifique la clase del objeto con el que hubo colisión así como la programación de todos los comportamientos que se deben realizar, recordando que el nombre del archivo de script está limitado al nombre de collscriptlet y éste se debe de encontrar en el directorio del sistema "/sys/share/scriptlets/" dentro de una carpeta que se llame igual que el objeto.

Clase:

nIstCamera

Método:

nada Collide (nada)

El método collide implantado para la cámara es prácticamente igual al de la clase nIstObject , para una cámara el único comportamiento relacionado con la presencia de una colisión es la ejecución de un script al igual que el método collide de la clase nIstObject siempre que ocurre una colisión con una cámara se corre el script señalado por el método Initialize, este método también está sometido a las mismas restricciones impuestas por el método initialize .

Método:

nada SetPosition (posición en el mundo virtual)

SetPosition modifica la posición de la cámara dentro del mundo virtual, los parámetros que requiere son las nuevas coordenadas X, Y, Z en las que se quiere colocar a la cámara, este método no modifica la orientación de la cámara, la cámara conserva la orientación que tenía antes de que cambiara su posición.

Método:

nada Trigger (delta de tiempo)

Actualizar el estado interno de la cámara implica, determinar el tipo de cámara que se tiene en ese momento y controlar los eventos y estados en los que se puede encontrar cada tipo de cámara. Para determinar el tipo de cámara en el método Trigger se revisa una bandera que indica cual es el tipo de cámara que se tiene, estas banderas son modificadas por los métodos SetStyleChase, SetStyleFree, etc. Una vez identificado el tipo de cámara se llevan acabo los comportamientos específicos de cada cámara, por ejemplo que la cámara sigua a su objetivo o que la cámara libre avance, también se realiza la lectura de entradas para los tipos de cámara que así lo requieran, si el comportamiento de la cámara es muy complejo o extenso Trigger invoca a otros métodos de nIstCamera los cuales tienen la función de controlar un tipo particular de cámara (handlers) , por último como varios de los métodos de actualización anteriormente descritos se actualiza la posición de la malla de colisión para que ésta concuerde con la posición de la cámara.

Método:

nada handleInput (evento de entrada)

El método handleInput como su nombre lo señala se encarga de manejar las entradas al sistema através del teclado y ratón para el manejo de la cámara, el parámetro evento de entrada se obtiene por medio del servidor de entradas (input server) e indica una entrada especifica al sistema, por ejemplo la tecla Ctrl está siendo presionada. Para obtener el parámetro evento de entrada es necesario que dentro del ciclo principal de la aplicación el servidor de entradas proporcione todas las entradas que han ocurrido en un ciclo, así pues en el ciclo principal de la aplicación debe de haber algo similar al siguiente código.

```
nInputEvent *ie;
ie = inp->FirstEvent()
```

Donde inp es un apuntador al servidor de entradas (nInputServer). Dado que este método no es llamado de manera indirecta por el método de actualización de otra clase, éste debe ser llamado directamente dentro del ciclo principal de la aplicación.

```
cam->handleInput(ie);
```

En el método handleInput se compara el tipo de evento así como la entrada que produjo el evento, sólo se comparan los tipos y llaves de interés para el control de la cámara de otra forma sería muy ineficiente, cuando se reconoce el tipo y la entrada se hace un cambio en el valor de una bandera, estas banderas le indican a los métodos manejadores cual es el estado y comportamiento de la cámara.

Método:

nada handleViewer (delta de tiempo)

Este método se en carga de manejar y contorlar los comportamientos de la cámara libre,como se menciono el método handleInput modifica una serie de banderas las cuales están relacionadas directamente con las entradas del usuario. Si la bandera llamada lbm está activada quiere decir que el botón derechó del ratón está siendo presionado, dependiendo de los valores de estas banderas es que se realizan las rotaciónes y traslaciones requeridas para el adecuado comportamiento de la cámara.

Cuando la cámara se encuentra en el modo de cámara libre presenta los siguientes comportamientos, al mantener presionado el bóton izquierdo del ratón y moverlo hacia abajo la cámara debe de avanzar en la dirección en la que está observando, al mantener presionado el bóton izquierdo del ratón y moverlo hacia arriba la cámara debe de moverse en dirección opuesta a la que está observando, al mantener presionado el bóton derecho del ratón y moverlo la cámara debe de girar en la dirección en la que está moviendo el ratón, si el raton se mueve a la derecha la cámara debe girar a la derecha, si se mueve hacia arriba la cámara gira hacia arriba y así sucesivamente, si el ratón se mueve sin presionar alguno de sus botones la cámara debe de permanecer inactiva. El parámetro delta de tiempo que requiere el método es usado para asegurar que la cámara se mueva o gire a una velocidad

constante, ya que no es posible asegurar que todos los ciclos de ejecución tarden el mismo intervalo de tiempo.

Método:

nada handleFPSViewer (delta de tiempo)

Este método se en carga de manejar y contorlar los comportamientos de la cámara en el modo de vista de primer persona, como se menciono el método handleInput modifica una serie de banderas las cuales están relacionadas directamente con las entradas del usuario. De acuerdo con los valores de estas banderas es que se realizan las rotaciónes y traslaciones requeridas para el adecuado comportamiento de la cámara.

Cuando la cámara se encuentra en el modo de vista de primera persona presenta los siguientes comportamientos, al mover el ratón la cámara debe de girar en la misma dirección, si el ratón se mueve a la derecha la cámara debe girar a la derecha, si se mueve hacia arriba la cámara gira hacia arriba y así sucesivamente, la rotación sobre el eje X, movimiento vertical del ratón, está limitada debido a que resulta confuso el permitir que la cámara llege a estar de cabeza, también en este método se tiene parte del control del personaje, cuando el ratón se mueve horizontalmete se produce un giro sobre el eje Y, cada vez que la cámara en vista de primera persona gira sobre el eje Y el personaje gira exactamente el mismo ángulo que la cámara, esto permite que el control del personaje pueda ser una mezcla entre el uso del teclado y el ratón. El parámetro delta de tiempo que requiere el método es usado para asegurar que la cámara se mueva o gire a una velocidad constante, ya que no es posible asegurar que todos los ciclos de ejecución tarden el mismo intervalo de tiempo.

Método:

nada handleChaseViewer (delta de tiempo)

Este método se en carga de manejar y contorlar los comportamientos de la cámara cuando sigue a un objeto , como se menciono el método handleInput modifica una serie de banderas las cuales están relacionadas directamente con las entradas del usuario. De acuerdo con los valores de estas banderas es que se realizan las rotaciónes y traslaciones requeridas para el adecuado comportamiento de la cámara.

Cuando la cámara sigue a un objeto presenta los siguientes comportamientos, al presionar el botón derecho del ratón y moverlo de forma vertical la cámara debe girar sobre el eje X, si el ratón se mueve hacia arriba la cámara gira hacia arriba, si el ratón se mueve hacia abajo la cámara gira hacia abajo, al igual que en handleFPSViewer el ángulo de giro de la cámara está limitado, la cámara no puede ser colocada debajo del objeto que sigue, por último la posición de la cámara con respecto al objetivo es actualizada en este método haciendo que la cámara siempre conserve la misma distancia con respecto al objeto que sigue. El parámetro delta de tiempo que requiere el método es usado para asegurar que la cámara se mueva o gire a una velocidad constante, ya que no es posible asegurar que todos los ciclos de ejecución tarden el mismo intervalo de tiempo.

Método:

nada SetStyleChase (objetivo, altura, distancia)

Este método tiene la función de colocar el estilo de cámara en el que ésta sigue a un objeto el cual es llamado objetivo, en este método se modifica un atributo de la clase nIstCamera en el que se indica cual es el tipo de cámara activa, el parámetro objetivo es un apuntador a un nodo 3D el cual es la referencia a seguir para la cámara, de esta manera el objetivo puede ser un personaje un NPC, un objeto, etc. El parámetro altura simplemente determina la altura con respecto de del eje Y a la que se coloca la cámara, el parámetro distancia es usado por el método para colocar la cámara con esa separación del objetivo, la distancia al objetivo se establece sobre el eje Z.

Método:

nada SetStyleAimPath (objetivo, camino, tiempo)

Este método tiene la función de colocar el estilo de cámara en el que ésta sigue un camino preestablecido mientras se apunta a un objeto el cual es llamado objetivo, al igual que SetStyleChase se modifica un atributo de la clase nIstCamera en el que se indica cual es el tipo de cámara activa, el parámetro objetivo es un apuntador a un nodo 3D el cual es la referencia a observar para la cámara, aunque ésta cambie de posición, de esta manera el objetivo puede ser un personaje un NPC, un objeto, etc. El camino está determinado por dos caracteres, uno que indica cual es el inicio y otro el final, antes de poder hacer que una cámara siga un camino es necesario haber creado el camino antes el cual consta de una serie de puntos unidos por una curva, esta curva es la que sigue la cámara, el tiempo es el intervalo en el que se desea que la cámara recorra el camino señalado, visto de otra manera este parámetro establece la velocidad de la cámara ya que la longitud del camino no varia, si el tiempo que se establece para recorrer el camino es muy pequeño la velocidad con la que se desplace la cámara será mucho mayor en comparación a un tiempo muy grande para recorrer el mismo camino, la velocidad de la cámara es constante.

Método:

nada SetStyleLookPath (camino, orientación)

Este método es muy similar a SetStyleAimPath, pero su implementación es más sencilla ya que en este estilo la cámara siempre conserva la misma orientación cuando sigue el camino mientras que SetStyleAimPath modifica su orientación para no perder de vista al objetivo, este método también modifica un atributo de la clase nIstCamera en el que se indica cual es el tipo de cámara activa, el camino también se indica por dos caracteres, uno que indica cual es el inicio y otro el final.

Método:

nada SetStyleFPS (objetivo, personaje, altura)

Set StyleFPS tiene la función de colocar el estilo de cámara conocido como vista de primera persona, modifica un atributo de la clase nIstCamera donde se indica que esta es la cámara activa, el parámetro objetivo es un apuntador a un nodo 3D el cual es la referencia

al personaje el cual inidica la posición y orientación que debe de tener la cámara, objetivo no puede apuntar a un nodo 3D que no esté asociado con un personaje. El parámetro personaje es un apuntador a un objeto nIstPlayer el cual tiene control sobre el estado del personaje, cuya malla y animaciones se encuntran en el nodo 3D señalado por objetivo, el parametro altura le indica la posición que debe mantener la cámara con repecto al eje Y del nodo3D.

En general estos son los métodos más importantes y en cierto grado los más complicados en el diseño de la aplicación y como se menciono por ese motivo son los que se incluyen en la descripción de la implementación, la clase nIstPlayer tiene varios métodos aparte de los que se incluyeron en la descripción, la implimentación de estos métodos no incluidos resulta muy sencilla. El método StartFW de la clase nIstPlayer sólo debe de modificar el valor de algunas banderas.

```
isFW=animWalk=true;
isSTPFW=isSTPBW=false;
```

Posterirmente cambiar el ciclo de animación que se esté realizando en ese momento.

```
model->NewCycle(4,0.5F,0.2F);
```

Y eso es todo, de igual manera hay muchos otros métodos de varias clases cuya sencilles a impulsado a no ser incluidos en la descripción de la implementación. Por otra parte no se incluyo la descripción de la implementación de clases como nIstCollZone y nIstStairs debido a que casi todos los metodos de estas clases (ver descripción del protocolo) son identicos a los de otras clases, Los métodos de la clase nIstCollZone son casi iguales a los de nIstObject, la diferencia en la implementación de ambas clases radica en que los objetos de la clase nIstCollZone no son visibles para el usuario, sólo son la malla de colisión por lo que no es necesario que estos tengan un nodo 3D que los represente gráficamente además las colisiones con la cámara no modifican ningún estado o comportamiento. Los métodos de la clase nIstStairs son casi iguales a los de nIstStruct, en este caso no existe una diferencia en la implementación de la clase nIstStairs, a la clase a la que le sirve esta aparente redundancia es a nIstPlayer ya que los comportamientos programados en su método Collide son diferentes cuando se registra una collisión con un objeto nIstStairs.

Capítulo 5
Implementación del sistema

5.1. Implementación de la filosofía de Nebula

Como se mencionó en el capítulo uno al hacer una aplicación con Nebula es necesario tener en cuenta varios aspectos de diseño que Nebula maneja para el desarrollo de aplicaciones. Este capítulo busca explicar de forma general como es que la filosofía de programación de Nebula se usa en la programación de la aplicación Calakmul virtual.

5.1.1. Construcción del espacio de trabajo.

Para modificar o crear el espacio de trabajo (work space) Nebula tiene un sistema constructor con el que se pueden agregar nuevos módulos al motor de juegos, de acuerdo con el análisis y diseño de la aplicación es necesario incluir dos nuevos módulos dentro del paquete de historias interactivas (archivo nist.pak), estos dos módulos están compuestos por los archivos "cc" y "h" de las clases nIstCollZone y nIstStairs, al agregar estos módulos al paquete de historias interactivas se puede crear un nuevo proyecto en donde las clases nIstCollZone y nIstStairs ya han sido incluidas.

Dentro del capítulo uno [1.6] se dio una breve explicación del funcionamiento del sistema constructor de Nebula, ahora se presenta como es que se uso este sistema para el desarrollo de la aplicación Calakmul virtual, para empezar no fue necesario el uso de classbuilder debido a que el archivo nist.pak ya había sido creado, dentro de nist se encuentran módulos como nistplayer, nistnpc, nistobject, niststruct, etc. Incluir los nuevos módulos dentro de este paquete responde al hecho de que estas clases están íntimamente relacionadas con las clases del paquete para la creación de historias interactivas, por ejemplo ambas clases heredan de nIstEntity. Así pues para incluir los módulos sólo se modifico el archivo nist.pak agregando el siguiente código.

```
beginmodule nistcollzone
   setdir ist
   setfiles { nistcollzone_main nistcollzone_cmds }
   setheaders { nistcollzone }
endmodule
#--------------------------------------------------------------------

beginmodule niststairs
   setdir ist
   setfiles { niststairs_main niststairs_cmds }
   setheaders { niststairs }
endmodule
```

El comando beginmodule define el nombre del módulo, con setdir se indica el directorio en el que se encuentran los archivos del módulo, los siguientes dos comandos definen cuales son dichos archivos, el módulo nistcollzone está compuesto de tres archivos, dos de código fuente(nistcollzone_main.cc y nistcollzone_cmds.cc) y un archivo de cabecera (nistcollzone.h), estos archivos se encuentran dentro de una carpeta llamada ist, esto no implica que todos los archivos se encuentren dentro de la misma carpeta, pues la ruta en la que se debe de encontrar la carpeta ist para los archivos de código fuente es:

\code\src\ist

Mientras que la ruta en la que debe de estar el archivo de cabecera es:

\code\inc\ist

También hay que agregar estos módulos dentro de la lista de módulos del paquete nist.

```
begintarget nist
  settype package
  setmods {
    nistplayer
        nistnpc
    nistworld
    nistentity
        nistobject
        niststruct
        nistcursor
        nistcamera
        nistcollzone
        niststairs
  }
```

Con estas modificaciones se crea un nuevo espacio de trabajo para la aplicación, un espacio de trabajo en el que están incluidos los módulos que se deben de desarrollar. Recordando que para actualizar el espacio de trabajo sólo se debe de ejecutar el archivo upsrc.tcl.

El espacio de trabajo de Calakmul Virtual incluye dos nuevas clases dentro de nist, los archivos con terminación main.cc son archivos en los que se incluye el código fuente de la clase, es decir en este archivo se crea la clase, se definen sus atributos públicos y privados, métodos públicos y privados, herencia, librerías, etc. Los archivos ".h" son los archivos de cabecera en donde se agregan las definiciones de las clases, de igual forma para que una aplicación pueda usar estas clases es necesario que los archivos de cabecera sean incluidos, por último los archivos cmds.cc son archivos en los que se programan los comandos por medio de los cuales los objetos de una clase pueden ser accesados y modificados en tiempo real a través de scripts de Tcl o de la consola de Nebula.

5.1.2. El cuerpo principal de la aplicación.

Además de todas las nuevas clases y módulos que tienen que ser agregados para hacer que la aplicación en desarrollo trabaje bajo el esquema de programación de Nebula se debe de tener un archivo principal en el que de forma general se programa el ciclo principal de la aplicación, se inicializan los servidores y se incluyen las librerías correspondientes. Dentro del ciclo de principal de ejecución se llaman a todos los métodos de actualización de los objetos descritos en el análisis y diseño, cada ciclo de ejecución está comprendido por el intervalo de tiempo que tarda en repetirse es decir el tiempo que tarda en ser llamado nuevamente el método o el tiempo que transcurre antes de que ocurra la ejecución de una misma operación.

Debido a que este archivo principal es de gran importancia para la implementación del sistema se considera conveniente analizar con más detalle su funcionamiento y construcción, a continuación se presenta el código fragmentado contenido en el archivo nistserv.cc con comentarios y aclaraciones sobre la función de cada parte, ya que fue bajo este esquema de funcionamiento que las nuevas clases de calakmul virtual fueron programadas.

Como en la mayoría de los programas primero se incluyen las librerías que van a ser usadas por la aplicación, por ejemplo si la aplicación va a reproducir algún tipo de audio es necesario crear un servidor de sonidos, para poder crear un servidor de sonidos se tiene que incluir el archivo ndsoundserver2.h en la definición de cabeceras, este ejemplo es extensivo a todos los servicios y clases usados dentro de la aplicación.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "kernel/nkernelserver.h"
#include "kernel/ntimeserver.h"
#include "gfx/ngfxserver.h"
#include "kernel/nfileserver2.h"
#include "input/ninputserver.h"
#include "script/ntclserver.h"
#include "misc/nmathserver.h"
#include "gfx/nscenegraph2.h"
#include "gfx/nchannelserver.h"
#include "misc/nconserver.h"
#include "misc/nparticleserver.h"
#include "node/n3dnode.h"
#include "misc/nspecialfxserver.h"
#include "audio/ndsoundserver2.h"
#include "collide/ncollideserver.h"
#include "shadow/nshadowserver.h"
#include "voice/nvoiceserver.h"
#include "kernel/nremoteserver.h"

#include "ist/nistworld.h"
#include "ist/nistentity.h"
#include "ist/nistplayer.h"
#include "ist/nistcamera.h"
```

Del anterior código se puede observar que primero se agregaron librerías propias de C++, la inclusión de estas responde a las necesidades del programa, como se van a manejar cadenas de caracteres es conveniente incluir string.h. Posteriormente se encuentran declaradas librerías de Nebula, se observa que la mayoría de los nombres de la librerías contiene la palabra "server", estas librerías permiten la creación de los servidores del sistema, si se van a correr scripts en tiempo de ejecución se incluye el archivo ntclserver.h, si se quieren recibir entradas al sistema a través del API de direct input se incluye el archivo ninputserver.h, etc. Por último se incluyen las nuevas clases que han sido programadas para la aplicación, estas como ya se vio permiten la creación de personajes, cámaras, objetos, estructuras, un espacio para agregar a los objetos, actualizarlos, etc.

Una vez que se han agregado todas las librerías necesarias, se programa la función principal (main), cuando se corre la aplicación es a partir de la función main que inicia la ejecución del código programado, para Nebula la función main se declara como en cualquier otro programa.

```
int main(int argc, char *argv[])
{
char *port_name = "istserv";
   char *server_class = "nglserver";
   char *mode = "w(800)-h(600)";
   char *startup = NULL;
   char *script_server = "ntclserver";
   bool grid = true;
   bool nosleep = false;
   bool argerr;
   long i;
   char arg_cmd[1024];
         arg_cmd[0] = 0;

         argerr=false;
         for (i=1; i<argc && !argerr; i++) {
                  char *arg = argv[i];
                  if (strcmp(arg,"-port")==0) {
                           if (++i < argc) port_name = argv[i];
                           else argerr=true;
      } else if (strcmp(arg,"-startup")==0) {
         if (++i < argc) startup = argv[i];
         else argerr=true;
      } else if (strcmp(arg,"-script")==0) {
         if (++i < argc) script_server = argv[i];
         else argerr=true;
      } else if (strcmp(arg,"-nogrid")==0) grid=false;
      else if (strcmp(arg,"-nosleep")==0) nosleep=true;
      else if (strcmp(arg,"-server")==0) {
         if (++i < argc) server_class = argv[i];
         else argerr=true;
      } else if (strcmp(arg,"-mode")==0) {
         if (++i < argc) mode = argv[i];
         else argerr=true;
      } else if (strcmp(arg, "-args")==0) {
         // los argumentos son guardados para que puedan ser usados por scripts, etc.
         for (++i; i<argc; i++) {
            strcat(arg_cmd,argv[i]);
            strcat(arg_cmd," ");
         }
      } else printf("unknown arg: %s\n",argv[i]);
         }
         if (argerr) {
                  printf("arg error, exiting.\n");
                  return 5;
         }
```

Después de la función main se inicializan varias banderas y cadenas de caracteres con la finalidad de configurar la aplicación, la configuración ocurre al momento de ejecutar la aplicación a través de una línea de comandos, donde todas las opciones y configuraciones que se indican después de llamar al archivo ejecutable pueden ser accesadas a través de los parámetros argc y argv de la función main. El código posterior tiene la función de determinar que es lo que el usuario introdujo en los argumentos, el argumento "-port" indica el puerto al cual se va a conectar la aplicación, "-startup" define un script que se va a correr al iniciar la aplicación, "-mode" es un argumento con el que se configura al servidor gráfico, resolución, modo ventana o pantalla completo, etc.

```
nEvent sleeper;
   nKernelServer* ks      = new nKernelServer;
   nFileServer2* fs2      = (nFileServer2*)      ks->New("nfileserver2",       "/sys/servers/file2");
   nGfxServer* gfx        = (nGfxServer *)       ks->New(server_class,         "/sys/servers/gfx");
   nInputServer* inp      = (nInputServer *)     ks->New("ndi8server",         "/sys/servers/input");
   nTclServer* script     = (nTclServer *)       ks->New(script_server,        "/sys/servers/script");
   nSceneGraph2* graph    = (nSceneGraph2 *)     ks->New("nscenegraph2",       "/sys/servers/sgraph2");
   nShadowServer* shadow  = (nShadowServer *)    ks->New("nsbufshadowserver", "/sys/servers/shadow");
   nChannelServer* chn    = (nChannelServer*)    ks->New("nchannelserver",     "/sys/servers/channel");
   nConServer *con        = (nConServer *)       ks->New("nconserver",         "/sys/servers/console");
   nMathServer *math      = (nMathServer *)      ks->New("nmathserver",        "/sys/servers/math");
   nParticleServer *part  = (nParticleServer *)  ks->New("nparticleserver",    "/sys/servers/particle");
   nSpecialFxServer *fx   = (nSpecialFxServer *) ks->New("nspecialfxserver",   "/sys/servers/specialfx");
   n3DNode *root          = (n3DNode *)          ks->New("n3dnode",            "/usr/scene");

        ks->New("nroot",          "/sys/share/paths");
        ks->New("nroot",          "/sys/share/scriptlets");

        // Crea el servidor de audio
nDSoundServer2 *as2  = (nDSoundServer2 *)  ks->New("ndsoundserver2",     "/sys/servers/audio");
nCollideServer *cs= (nCollideServer *)     ks->New("nopcodeserver",      "/sys/servers/collide");
nVoiceServer *vs= (nVoiceServer *)         ks->New("nsapivoiceserver",   "/sys/servers/voice");
nIstWorld *istworld= (nIstWorld *)         ks->New("nistworld",     "/world");
nIstCamera *cam = (nIstCamera *)           ks->New("nistcamera",    "/world/istcamera");
```

Se crean los servidores que requiere la aplicación, estas declaraciones son prácticamente iguales a las que se usan al crear un objeto cualquiera, primero se crea un apuntador del tipo del objeto, con este apuntador se puede acceder al objeto que se crea por medio de del método new, El método New es proporcionado por el servidor del núcleo, este método además de crear un objeto de la clase indicada lo agrega dentro una estructura similar a un árbol de directorios, la cual sirve para que los métodos de los objetos que se crean puedan ser llamados en tiempo de ejecución a través de comandos de tcl, de esta manera se crea un servidor de archivos, gráfico, de entradas al sistema, de scripts, de sombras, consola, matemático, de audio, etc. Para una descripción más a fondo de la función de algunos de estos servidores véase capítulo 1.

```
//Se indica la posición de inicio de la cámara
cam->SetPosition(2500.0f,120.0f,-8700.0f);
        cs->BeginCollClasses();
                cs->AddCollClass("nistcollplayer");
                cs->AddCollClass("nistcollnpc");
                cs->AddCollClass("nistcollobject");
```

```
            cs->AddCollClass("nistcollstruct");
            cs->AddCollClass("nistcollcamera");
        cs->EndCollClasses();

        cs->BeginCollTypes();
            cs->AddCollType("nistcollplayer","nistcollplayer",COLLTYPE_EXACT);
            cs->AddCollType("nistcollplayer","nistcollobject",COLLTYPE_EXACT);
            cs->AddCollType("nistcollplayer","nistcollstruct",COLLTYPE_EXACT);
            cs->AddCollType("nistcollnpc","nistcollstruct",COLLTYPE_EXACT);
            cs->AddCollType("nistcollplayer","nistcollnpc",COLLTYPE_EXACT);
            cs->AddCollType("nistcollnpc","nistcollobject",COLLTYPE_EXACT);

            cs->AddCollType("nistcollcamera","nistcollnpc",COLLTYPE_EXACT);
            cs->AddCollType("nistcollcamera","nistcollobject",COLLTYPE_EXACT);
            cs->AddCollType("nistcollcamera","nistcollstruct",COLLTYPE_EXACT);

            cs->AddCollType("nistcollobject","nistcollobject",COLLTYPE_IGNORE);
            cs->AddCollType("nistcollstruct","nistcollstruct",COLLTYPE_IGNORE);
            cs->AddCollType("nistcollstruct","nistcollobject",COLLTYPE_IGNORE);

            cs->AddCollType("nistcollcamera","nistcollplayer",COLLTYPE_IGNORE);
        cs->EndCollTypes();

        n_printf("Setting coll config completed\n");
```

A continuación se configura el sistema de colisiones, "cs" es un apuntador a un objeto de la clase nCollideServer y por medio de este apuntador se pueden llamar a los métodos del objeto, con BeginCollClasses, AddCollClass y EndCollClasses se le indica al servidor cuales son aquellas clases entre cuyos objetos existen colisiones, con los métodos BeginCollTypes, AddCollType y EndColltypes se le dice al servidor el tipo de colisión que existe entre las diferentes clases definidas, así pues mientras la colisión que ocurra entre dos personajes debe de ser exacta, la colisión entre dos estructuras debe de ser ignorada. Aclarando que los cálculos que hace el servidor de colisiones para determinar si estas existen entre los objetos del mundo virtual son en base a la malla de colisión asociada con el objeto y no con la malla que se dibuja del objeto.

```
const char* res;
        if (arg_cmd[0]) {
            nEnv *args = (nEnv *) ks->New("nenv","/sys/share/args");
            args->SetS(arg_cmd);
        }
  ks->ts->EnableFrameTime();
  script->Run(arg_cmd, res);
if (ks->GetRemoteServer()->Open("nebula"))
  {
    gfx->SetDisplayMode(mode);
    if (gfx->OpenDisplay())
    {
      if (startup)
      {
        n_printf("running startup script '%s'\n",startup);
        script->RunScript(startup, res);
        if (res) n_printf("%s\n",res);
      }
```

Es importante señalar que "ts" es jun apuntador al servidor de tiempo y "gfx" es un apuntador al servidor gráfico. Con el uso de EnableFrameTime se logra que una llamada al método getFrameTime regrese un valor idéntico entre dos llamadas al método de actualización (Trigger) del servidor de tiempo, esto se hace con la finalidad de que la relación de cuadros por segundo de la aplicación siempre sea la misma. SetDysplayMode configura la ventana de visualización, como se observa en el código anterior la cadena de caracteres "mode" se asigna por medio de los parámetro de entrada de la función main, todos aquellos caracteres después de indicar la opción mode son guardados dentro de la cadena "mode", en esta cadena se puede indicar la resolución, el número de pixeles por pulgada, si se la aplicación va acorren en modo de ventana o en modo de pantalla completa, etc. OpenDisplay crea la ventana de visualización de la aplicación.

La cadena de caracteres "startup" también es asignada a través de los parámetros de entrada de la función main, todos aquellos caracteres después de indicar la opción startup son guardados dentro de la cadena "startup", en esta cadena se indica el nombre de un archivo de tcl en el que se crean los personajes, objetos escenario, etc. Con RunScript se ejecuta este script.

```
if (as2) as2->OpenAudio();

n_printf("Start up  script completed\n");

nRState rs(N_RS_LIGHTING,N_TRUE);
gfx->SetState(rs);
n_printf("Setting gfx state completed\n");

ks->ts->ResetTime();

root->RenderContextCreated(0);
```

Se inicia el servidor de audio, al llamar al método OpenAudio el servidor inicia la reproducción de sonidos asociados con nodos 3D, se establece el estado del servidor gráfico, ResetTime reinicia el reloj que maneja el servidor de tiempo. Por otra parte root es un apuntador a un nodo 3D el cual será utilizado como la raíz de toda la estructura de nodos que se van a crear dentro de la aplicación, los objetos de la clase nvisnode que necesitan crear objetos para cada elemento que va a ser dibujado en pantalla usan el método RenderContextCreated para crear nuevas instancias de dichos objetos.

```
bool running = true;
    while (gfx->Trigger() && running)
    {

      if (!script->Trigger()) running = false;

      ks->Trigger();
      ks->ts->Trigger();
      double t = ks->ts->GetFrameTime();
      inp->Trigger(t);

      ks->GetRemoteServer()->Trigger()
```

Se crea una bandera (running) que tiene la función de indicarle a la aplicación cuando salir del ciclo principal de ejecución, al iniciar el ciclo principal de ejecución (ciclo while), una de las condiciones para salir del ciclo es que la bandera running cambie su valor también dentro de las condiciones del ciclo principal se actualiza el servidor gráfico, al llamar al método Trigger de la clase GfxServer básicamente se dibuja un nuevo cuadro en la ventana de visualización.

Lo primero que se hace dentro del ciclo principal es actualizar los servidores que maneja la aplicación, se actualiza el servidor de scripts de Tcl, si ocurre algún error en la actualización del servidor la bandera running cambia de valor, se actualiza el servidor, se actualiza el servidor del kernel, de forma general al actualizar el servidor del kernel lo que se hace es actualizar el estado del manejador de memoria y actualizar el estado de las variables en memoria, se actualiza el servidor del tiempo, el método GetFrameTime obtiene un intervalo de tiempo idéntico entre dos llamadas al método de actualización del servidor también se hace una actualización del servidor de entradas.

```
nInputEvent *ie;
        if ((ie = inp->FirstEvent()))
        {

          if(ie->GetType() == N_INPUT_KEY_DOWN )
          if( ie->GetKey() == N_KEY_ESCAPE )
          con->Toggle();

          do cam->handleInput(ie);
          while ((ie = inp->NextEvent(ie)));  //manejo de entradas de la cámara

          inp->FlushEvents();
        }
       vs->Trigger();
       istworld->Trigger((float)t);                    //Actualiza personajes, objetos, etc
```

Se procede al manejo de entradas al sistema, para entender esta parte de código es necesaria una explicación más detallada, el servidor de entradas guarda los eventos de entrada que ocurran en una especie de lista, estos eventos son almacenados de acuerdo al orden en el que se produjeron los primeros eventos que se recibieron son los primeros eventos de la lista , de estos eventos de entrada se puede conocer que periférico los recibió, su tipo y valor, así pues se declara una variable capaz de recibir este tipo de dato y se le pide al servidor de entradas el primer evento de la lista, con los métodos GetType y GetKey se sabe el tipo y la tecla que produjo el evento, si se presiona la tecla Escape se activa la consola, con->Toggle() invoca a un método del servidor de la consola, si la consola no está activada la activa pero si ésta ya está activada entonces la desactiva.

La siguiente sección de código se encarga del manejo de entradas de la cámara, mientras haya eventos de entrada en la lista del servidor, se invoca al método manejador de eventos de la cámara (handleInput), este método sabe que acciones debe realizar la cámara de acuerdo a las entradas que halla dado el usuario, luego se eliminan todos los eventos guardados por el servidor de entradas, para dar paso a los eventos que ocurran en el siguiente ciclo de ejecución.

```
    if (as2)
     {

        as2->BeginScene(t);
     }

     if (gfx->BeginScene())
     {
        nRState rs;
        matrix44 vwr;
        vwr = cam->GetTransform();

        // Actualiza el servidor de partículas
        part->Trigger();

        // Se configura la matriz del observador
        gfx->SetMatrix(N_MXM_VIEWER,vwr);

        // Canales de tiempo
        chn->SetChannel1f(chn->GenChannel("time"), (float) t);
        chn->SetChannel1f(chn->GenChannel("gtime"), (float) t);

        vwr.invert_simple();
        if (graph->BeginScene(vwr))
        {
           fx->Begin();
           graph->Attach(root, 0);
           //Desactiva los nodos de efectos especiales.
           fx->End(graph);
           graph->EndScene(true);
        }
         //Se dibuja la consola si es que ésta está activada,
         //como se definio en este caso si se presiono la tecla esc.
        con->Render();
        gfx->EndScene();
        if (!nosleep) n_sleep(0.02);
     }
     if (as2)
     {
        as2->EndScene();
     }
  }
  //termina el ciclo principal de ejecución.
```

La variable "vwr" es una matriz cuadrada de 4x4, GetTransform obtiene la matriz de transformación de la cámara, dicha matriz es almacenada por "vwr", SetMatrix le indica al servidor gráfico que el cuadro a dibujar es el que tiene un observador en la ubicación señalada por la matriz de transformación, en este caso el observador está en la ubicación y orientación de la cámara.

El método BeginScene de nSceneGraph inicia un nuevo cuadro para la escena que se está dibujando, se toma la matriz inversa de la cámara (vwr) como la raíz de todas las transformaciones, Attach agrega toda la jerarquía asociada a un nVisNode al grafo de la

escena, así es que Attach se invoca con el nVisNode más alto dentro de la jerarquía, EndScene finaliza el grafo de escena, es importante anotar que los nodos agregados al grafo de la escena son clasificados para asegurar los cambios mínimos de estados de dibujo y después ser dibujados.

Con esto termina el ciclo principal de la aplicación, todo el código anterior se repite cada ciclo y es el código más importante de la aplicación, el código que sigue a continuación sólo es ejecutado una vez que se a salido del ciclo principal de ejecución, y es decir cuando se va a cerrar la aplicación

```
root->RenderContextDestroyed(0);
        if (as2)
         {
           as2->CloseAudio();
           as2->Release();
         }
        gfx->CloseDisplay();
    }
        ks->GetRemoteServer()->Close();
  }

  ks->ts->DisableFrameTime();

  if (root)   root->Release();
  if (part)   part->Release();
  if (math)   math->Release();
  if (con)    con->Release();
  if (chn)    chn->Release();
  if (shadow) shadow->Release();
  if (graph)  graph->Release();
  if (script) script->Release();
  if (gfx)    gfx->Release();
  if (inp)    inp->Release();
  if (fs2)    fs2->Release();
  delete ks;
  return 0;                          //salida correcta.
}
```

Para cerrar la aplicación básicamente lo que se hace es terminar y eliminar todos los servidores que se crearon por medio de la llamada a Release, en el caso del nodo 3D "root" lo que se hace es eliminar toda la jerarquía de nodos que creo la aplicación, liberando el espacio que estos ocupaban en memoria, en el caso del servidor gráfico y el servidor de audio antes de llamar al método Release es necesario invocar los métodos CloseDisplay y CloseAudio respectivamente, CloseDisplay cierra la ventana de despliegue gráfico y CloseAudio termina la reproducción del audio, una vez hecho esto se pueden eliminar los servidores.

5.1.3. Los scripts de la aplicación.

Además de la programación del archivo ejecutable de la aplicación es necesario crear una serie de scripts encargados principalmente de cargar los contenidos que se

necesitan dentro del mundo virtual, es decir scripts que carguen estructuras, objetos, personajes, sonidos, etc. La ventaja fundamental de cargar los contenidos de la aplicación a través de scripts es que se logra una mayor flexibilidad al momento de cambiar o alterar los contenidos de la aplicación, si esto se hiciera por medio de código en C al momento de querer cambiar por ejemplo un objeto o una textura sería necesario modificar el código y volver a compilar la aplicación.

Incluir el código y la explicación de cada uno de los archivos de script resultaría de poco interés y llevaría mucho espacio, el código completo de los scripts de Tcl se ha incluido en el apéndice B , por lo que a continuación se da una breve explicación de las funciones principales que realiza cada script.

Archivo calakmultk.tcl

Este archivo es un script de Tk, interfaz de usuario, por medio de la cual se puede configurar algunas características del despliegue gráfico de la aplicación, cosas como la resolución, modo de despliegue, API de dibujo, etc. La idea es que el usuario de una forma sencilla, por medio de botones y listas pueda configurar el modo en el que va a correr la aplicación. Es importante señalar que este archivo no contiene ningún comando de Nebula, está escrito en su totalidad como script de Tk, simplemente asigna valores y opciones que después son agregadas al momento de correr la aplicación.

Archivos calakmulgui.tcl y calakmulguid3d.tcl

Estos scripts de manera general realizan la misma función, calakmulgui es ejecutado cuando en la configuración de la aplicación se eligió el API de OpenGl mientras que calakmulguid3d se ejecuta cuando se eligió el API de Direct 3D, así pues estos son el primer script que carga la aplicación, antes que cualquier otra cosa estos scripts definen y cargan los procedimientos necesarios para el funcionamiento de la aplicación, se define un procedimiento que permite eliminar la escena que se está dibujando, también se define un procedimiento en el que se crea la interfaz gráfica del usuario a través de Tk, además se cargan una serie de procedimientos almacenados en otros archivos de Tcl, los cuales se explican más adelante, se carga el archivo de rutas para los personajes controlados por el sistema, se crean variables de ambiente en la que se indican cosas como la ubicación de los mapas y las texturas, el nombre de la cámara, etc. Se configura el servidor gráfico de acuerdo a las opciones elegidas, también se le indica al servidor de colisiones cuales son las mallas de colisión para la cámara, las estructuras y las escaleras, se crea el personaje controlado por el usuario, los personaje controlados por el sistema, los objetos con los que puede interactuar el usuario y las zonas de colisión, por último se llama un procedimiento que carga la escena que se va a dibujar en pantalla.

Archivo istgenlib.tcl

El archivo istgenlib.tcl contiene un procedimiento en el cual se definen el sombreado y las texturas que va a usar el personaje controlado por el usuario al momento de dibujarse en pantalla.

Archivo istncalcore.tcl

El script istncalcore es un procedimiento con el cual se crea un nuevo modelo de cal3d dentro de la aplicación. Para lograr esto primero se crea un nodo del tipo ncal3dcoremodel, se carga el esqueleto con el cual se va a animar la malla del personaje, se cargan las diferentes animaciones que el personaje es capaz de realizar, posteriormente se carga la malla asociada al esqueleto, al final se cargan los materiales para esto se toman los nodos de sombreado y de texturas definidos en el script "istgenlib.tcl". la ventaja de usar un sistema de animación como el que ofrece cal3d es que al ser un sistema de animación esqueletal, los archivos en los que se guardan las animaciones son mucho más pequeños en tamaño al compararlos con archivos de animación por keyframe, esto es muy fácil de explicar, los archivos de animación por keyframe indican como se debe de mover cada vértice de la malla , mientras más detallada sea la malla se necesitan más vértices y mientras más vértices tenga la malla el tamaño del archivo de la animación se incrementa.

Archivo istplayer.tcl

El script contenido en istplayer carga una serie de procedimientos, el procedimiento createistplayer indica cuales es el nodo 3D y cual es el nodo del modelo de Cal ligados con la animación y el movimiento, el procedimiento configistplayer configura varios parámetros que afectan al personaje, como la velocidad máxima que puede alcanzar el personaje, su aceleración, su velocidad al girar y la malla de colisión del personaje. Además de los procedimientos anteriormente descritos instplayer.tcl contiene otros procedimientos relacionados con la creación y configuración de un cursor, este cursor está relacionado con las funciones de diseño, ya que con este cursor se pueden crear caminos por los cuales pueden transitar la cámara o los personajes controlados por el sistema.

De manera muy similar al archivo istplayer.tcl el archivo istnpc.tcl contiene una serie de procedimientos cuya finalidad es la de crear y configurar los personajes controlados por el sistema, así pues el procedimiento createistnpc es equivalente a createistplayer y el procedimiento configistnpc es equivalente a configistplayer.

Archivo istobject.tcl

Con este script se carga un procedimiento con el cual se pueden crear objetos en el mundo virtual, el procedimiento createistobject necesita el nombre del objeto, el archivo donde se encuentra la malla 3D, la textura que se va a mapear a dicho objeto y su posición, de esta manera se busca simplificar el proceso de creación de contenido, en el caso de los objetos, la malla de colisión corresponde a la misma malla 3D que se dibuja en el mundo virtual.

Archivo istkeymapping.tcl

El archivo istkeymapping.tcl contiene un procedimiento cuya función es la de asignar las entradas que recibe la aplicación, le indica al inputserver (servidor de entradas) cuales son las posibles entradas que puede proporcionar el usuario así como las acciones que estas producen, en este procedimiento se manejan las entradas para el personaje controlado por el usuario, algunas acciones específicas para los personajes controlados por el sistema y el control del cursor.

Archivo istwindow.tcl

Básicamente este es una archivo de Tk y contiene toda una serie de procedimientos necesarios para la creación de la interfaz gráfica de usuario, por medio de la cual se pueden controlar varios aspectos de la aplicación, a través de la interfaz de usuario se pueden cargar nuevos personajes y objetos dentro del mundo virtual, sin necesidad de hacer uso de ninguna clase de scripts, también permite crear y guardar nuevas rutas para la cámara y los personajes controlados por el sistema, además la interfaz de usuario ofrece la posibilidad de introducir comandos de Tcl a la aplicación sin la necesidad de tener abierta la consola, se puede elegir el estilo de la cámara (libre, primera persona, siguiendo a un personaje o una ruta, etc) , por último se pueden cargar diferentes escenas previamente definidas.

Archivo istcollzone.tcl

Istcollzone.tcl carga un procedimiento con el cual se pueden crear zonas de colisión en el mundo virtual, al procedimiento createistzone se le indica el nombre de la nueva zona de colisión, el archivo donde se encuentra la malla de colisión y la posición que va a tener en el mundo virtual, de esta manera la creación de contenido se simplifica.

Archivo loadl1.tcl

Este archivo contiene un solo procedimiento, startl1, este procedimiento tiene la finalidad de cargar una nueva locación o escena, antes de llamar al procedimiento startl1 es necesario borrar la escena que está presentando la aplicación, el procedimiento reconfigura al servidor gráfico, crea todos los nuevos objetos necesarios para dicha locación, carga el nuevo mapa mediante la ejecución de un script llamado calakmull1.tcl, caraga la vegetación presente en la escena mediante la ejecución de un script llamado foliage.tcl, de la misma manera los archivos loadl2.tcl y loadl3.tcl poseen sus respectivos procedimientos mediante los cuales se pueden cargar diferentes escenas.

Archivo calakmull1.tcl

Como se menciono anteriormente este archivo es ejecutado por el procedimiento startl1 que se encuentra dentro del archivo loadl1.tcl y contiene una serie de comandos de Tcl y Nebula mediante los cuales se crea gráficamente una nueva escena, así mismo los archivos calakmull2.tcl y calacmul3.tcl crean diferentes escenas al ser invocados por loadl2.tcl y loadl3.tcl respectivamente.

Archivo foliage.tcl

El archivo foliage.tcl se encarga de la creación de toda la vegetación que aparece en el mundo virtual, dentro de /usr/lib/ se crea un modelo de cada una de las plantas que se va a dibujar en escena, posteriormente mediante el uso de varios procedimientos definidos dentro de foliage.tcl y el uso de nodos de ligado (nlinknode) se repiten estos modelos tantas veces como lo necesite la escena.

Una vez explicados la mayoría de los scripts es necesario aclarar que para varios de los archivos de scripts sus nombres poseen la partícula "d3d", como se menciono al inicio al momento de correr la aplicación es posible hacerlo a través de Opengl o Direct3D, el funcionamiento de algunas características como el bumpmapping o las sombras en tiempo real se manejan diferente dependiendo del API que se esté usando, así pues los archivos con esta partícula son archivos de scripts que se usan cuando la aplicación está corriendo

bajo Direct3d pero que tienen exactamente la misma función que su contraparte para Opengl.

5.2. Descripción del funcionamiento general de la aplicación

Como se estableció en [5.1] la aplicación al momento de ejecución llama a varios scripts los cuales cargan el contenido de la aplicación, crean la interfaz gráfica de usuario, cargan una serie de procedimientos por medio de los cuales se pueden llevar a cabo procesos más complejos, procesos que están compuestos por un gran número de comandos de Nebula, etc.

El objetivo de esta sección es tratar de describir el funcionamiento de la aplicación Calakmul virtual de una manera general, en la que sea fácil observar como se relaciona la aplicación con los scripts y los estados por los que atraviesa el sistema.

5.2.1. Diagrama de flujo del sistema.

El diagrama que se presenta a continuación trata de mostrar los eventos que se van sucediendo desde el momento en el que se ejecuta la aplicación

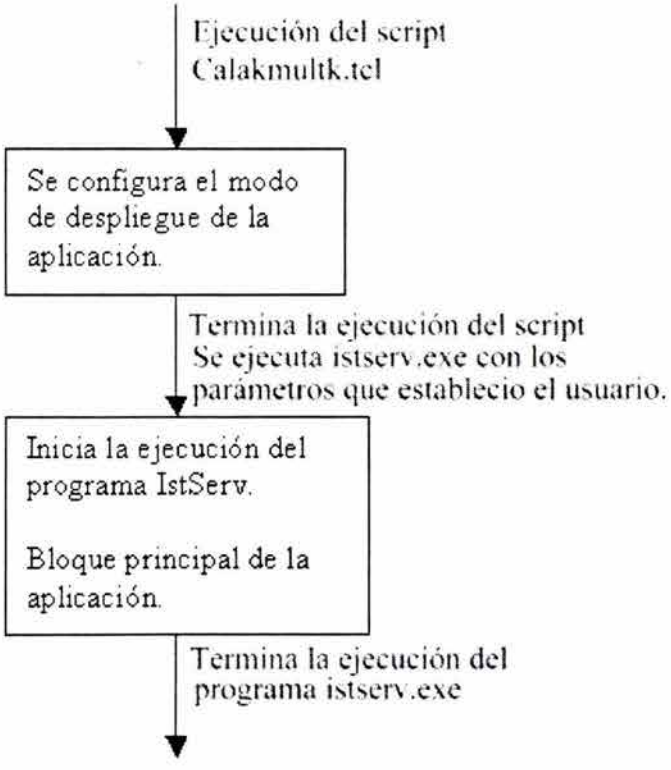


Figura 5.1. Diagrama de flujo general.

El diagrama 5.1. Indica como es que funciona la capa más externa de la aplicación, en realidad lo que primero se llama a ejecución es un script de Tk llamado Calakmultk.tcl, éste simplemente despliega una interfaz en la que el usuario puede configurar el modo en el

que desea que corra la aplicación, al momento en el que el usuario pulsa el botón con la etiqueta OK, se ejecuta la aplicación istserv con las opciones que el usuario eligió, por ultimo se termina la ejecución del script. Debido al espacio disponible para cada diagrama resulta conveniente dividir el diagrama de flujo general en bloques más pequeños. A continuación se presenta un diagrama más detallado del contenido del bloque principal de la aplicación mostrado en la figura 5.1.

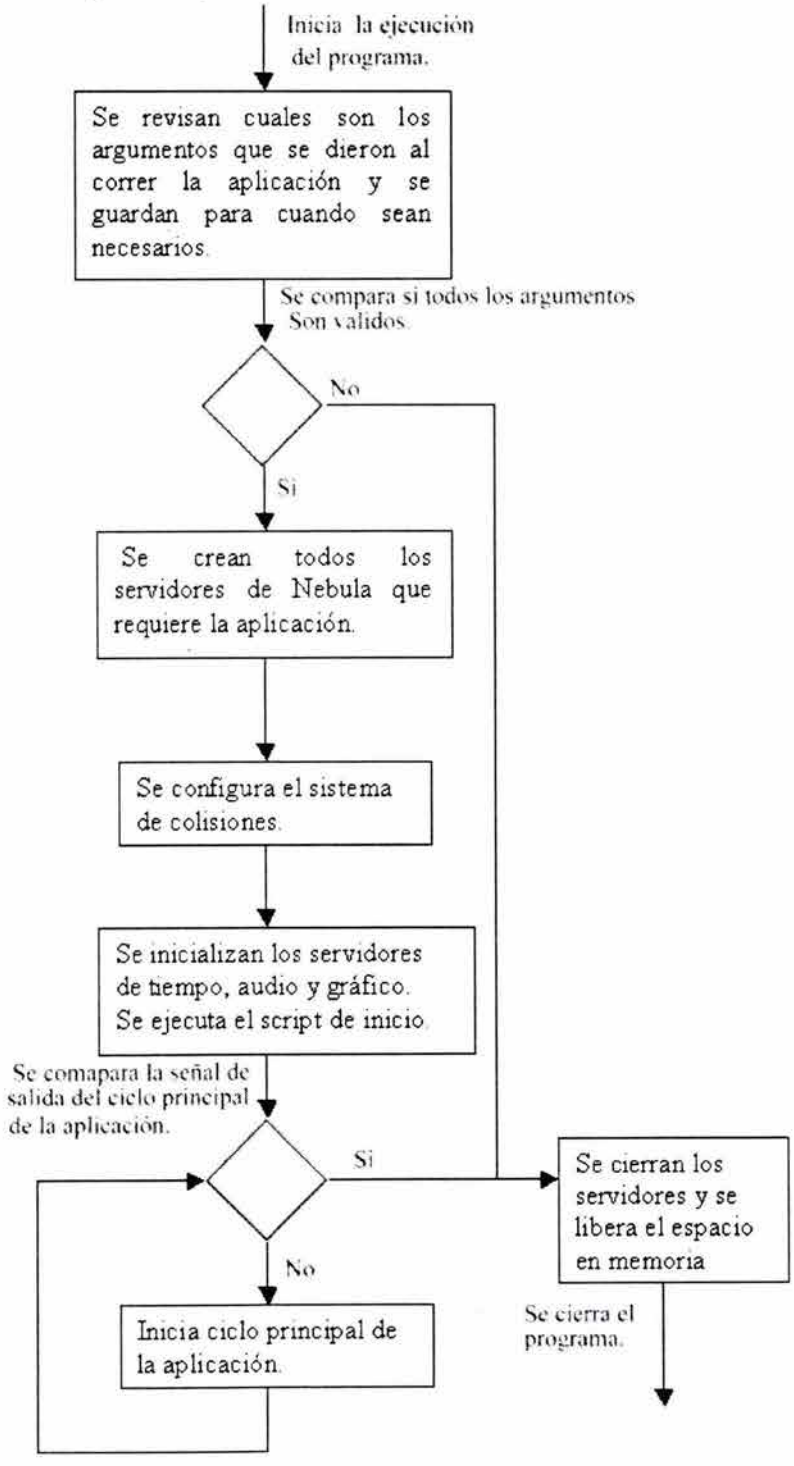
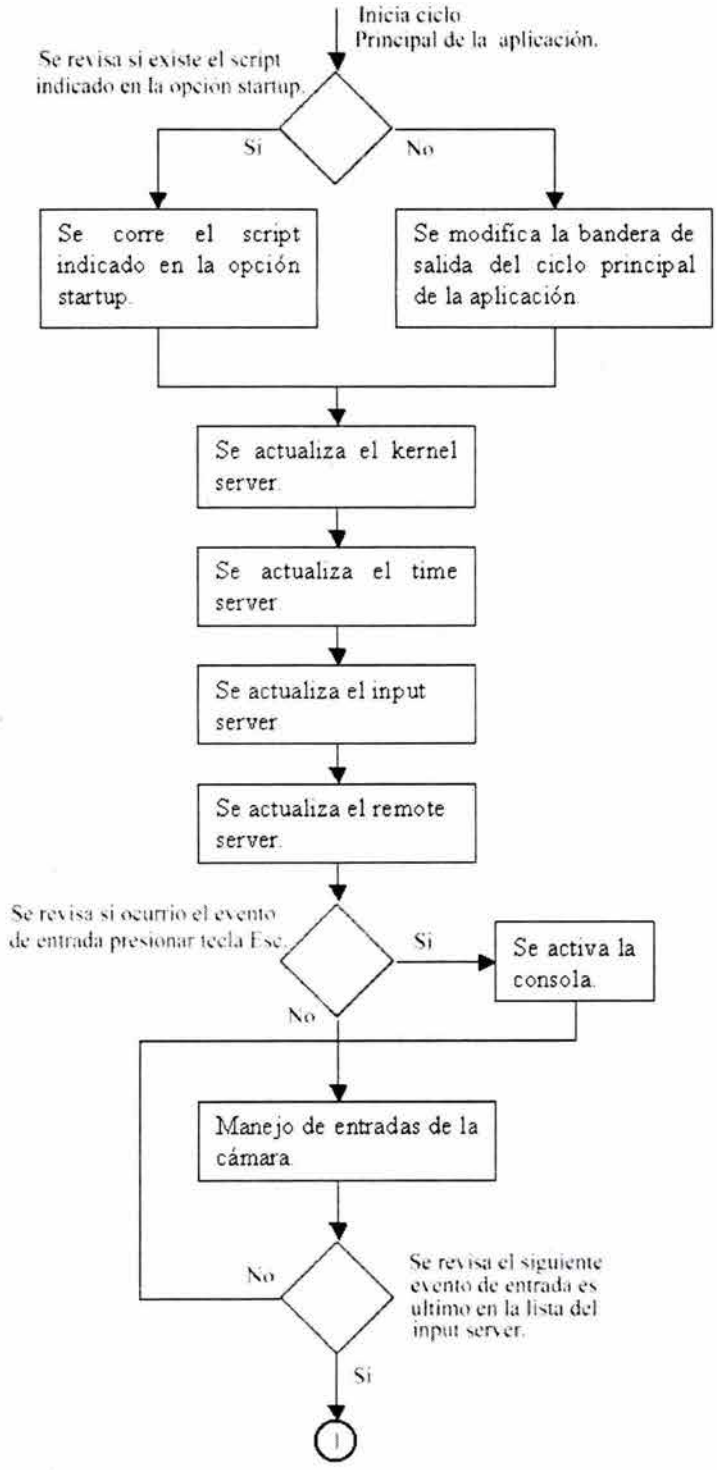


Figura 5.2. Diagrama de flujo del bloque principal de la aplicación.

En el diagrama de la figura 5.2 presenta a detalle el funcionamiento del bloque principal de la aplicación. Como se indicó en [5.1.1] primero se crean todos los servidores de la aplicación, se pasa a una etapa de configuración de los servidores y se ejecuta un script cuya función principal es la de cargar contenidos, hecho todo lo anterior se pasa al ciclo principal, el ciclo principal de la aplicación aparece como un solo bloque en el diagrama pero en realidad este ciclo es más complejo en la figura 5.3. se muestra el contenido de este ciclo.



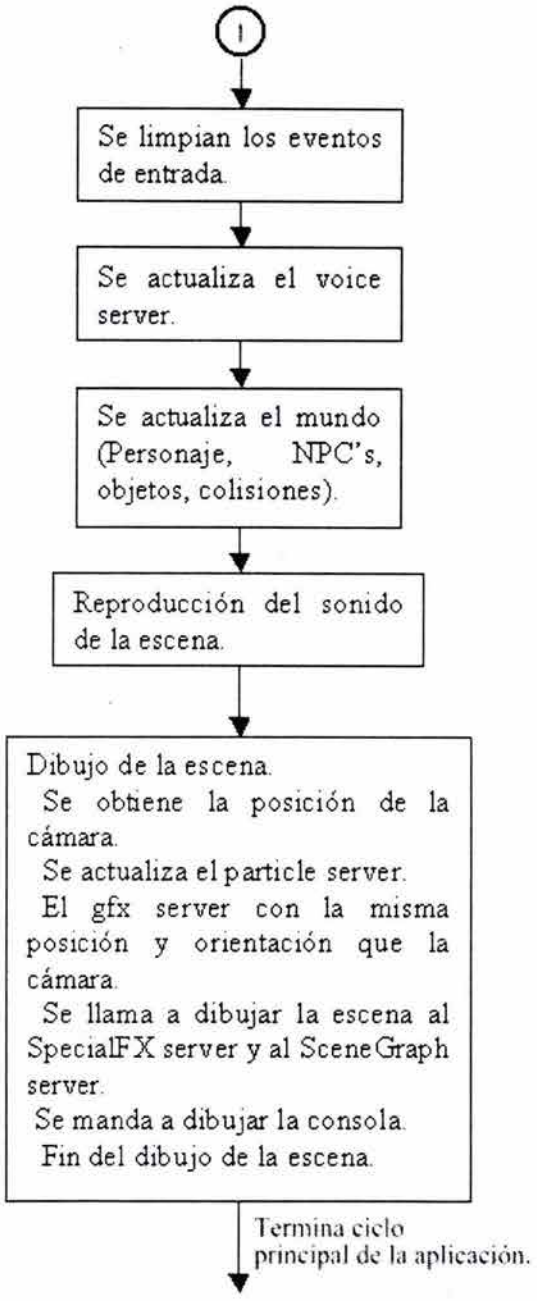


Figura 5.3. Diagrama de flujo del ciclo principal de la aplicación.

Después de hacer el análisis del funcionamiento de la aplicación a través de los diagramas de flujo se podría pensar que la aplicación trabaja por completo de una manera secuencial y estructurada, pero en realidad esto no es así ya que debido a lo complejo que resulta hacer un diagrama en el que se muestre el funcionamiento de varios hilos de ejecución de un programa se opto por la forma más sencilla de presentar la información. Así en el diagrama 5.3 sólo se muestra el hilo principal de la aplicación, existen otros hilos

de ejecución además del ciclo principal, en realidad cada servidor tiene su propio hilo, por ejemplo, el input server sigue atendiendo las entradas del usuario mientras que en otro hilo de ejecución el servidor de tiempo continua realizando la medición del tiempo que ocurre entre las llamadas a su método de actualización. Al estar programados de esta manera los servidores de Nebula pueden atender las peticiones que les hagan los objetos clientes sin importar en que parte del ciclo principal de ejecución se encuentre el programa.

Capítulo 6
Manual de usuario

6.1. Requerimientos del sistema

6.1.1. Requerimientos mínimos.
-Procesador compatible con Intel a 700 Mhz
-128 MB de memoria RAM
-30 MB de disco duro
-Tarjeta de video con 64 MB de memoria
-DirectX 7.0
-Instalar la versión más reciente de Tcl/Tk (versión 8.4 o superior)

6.1.2. Requerimientos recomendados.
-Procesador compatible con Intel a 1 Ghz
-256MB de memoria RAM
-30 MB de disco duro
-Tarjeta de video con 128 MB de memoria
-DirectX 9.0
-Instalar la versión más reciente de Tcl/Tk (versión 8.4 o superior)

6.2. Instalación

El proceso de instalación de la aplicación Calakmul virtual es muy sencillo, simplemente hay que seguir los siguientes pasos.

-Ejecutar el programa Install.exe
-Indicarle al programa el directorio en el que se instalará la aplicación
-Seguir las instrucciones que presente en pantalla el programa instalador

La instalación coloca todos los archivos en sus directorios correspondientes de tal forma que la aplicación está lista para ser ejecutada por el usuario una vez concluida la instalación.

6.3. Como correr la aplicación

Existen varias formas de correr la aplicación Calakmul virtual.

Las formas más comunes y simples son las de dar doble click sobre el icono de acceso directo llamado Calakmul, el cual es creado por el instalador y se encuentra en el escritorio de la computadora, también es posible correr la aplicación por medio del menú Inicio, sólo es necesario abrir el menú Inicio, Programas, Calakmul y dar un click sobre el programa llamado Calakmul. Cuando se ejecuta la aplicación por uno de estos métodos siempre se abrirá el menú de configuración de la antes de que se ejecute lo que sería propiamente la aplicación.

También se puede correr la aplicación a través de una línea de comandos de DOS, Para hacer esto primero es necesario abrir una línea de comandos de DOS y cambiar de directorio al directorio en el que se instalo la aplicación, una vez en dicho directorio hay que moverse a \Calakmul\data\ist.

Un ejemplo de una ruta completa a la que hay que acceder con la línea de comandos sería:

C:\...ruta de instalación...\ Calakmul\data\ist>

En esta dirección no existe ningún archivo ejecutable, este directorio es el directorio raíz de la cual la aplicación obtiene los datos que carga en tiempo de ejecución, por este motivo es que la aplicación se tiene que correr desde dicho directorio. El archivo ejecutable que corre la aplicación se llama istserv.exe y se encuentra dentro del siguiente directorio:

C:\...ruta de instalación...\ Calakmul\bin\win32>

Así lo que se tiene que escribir en una línea de comandos de DOS para correr la aplicación sería:

C:\...ruta de instalación...\ Calakmul\data\ist> C:\...ruta de instalación...\ Calakmul\bin\win32\istserv.exe

La línea anterior no basta para que la aplicación corra, si se trata de ejecutar istsserv se enviara un mensaje de error y la aplicación se cerrara, es necesario agregar algunas opciones a la línea de comandos para configurar la forma en la que se correrá la aplicación. A continuación se muestran todas las opciones que se pueden agregar a la línea de comandos.

-server. La opción server se usa para indicar el tipo de servidor gráfico con el que se va a correr la aplicación, esta opción sólo soporta los argumentos "nd3d8server" y "nglserver", con el primer argumento el proceso de graficación se realiza con el API de Direct 3D, con el segundo se usa el API de OpenGL, si se omite esta opción el servidor gráfico que se usará será "nglserver".

-mode. La opción mode configura al servidor gráfico, el argumento de esta opción es una cadena de caracteres en la que se establece la resolución, el número de bits por píxel, el dispositivo de despliegue y el modo en que se mostrará la aplicación, si se omite esta opción el servidor gráfico correrá a una resolución de 400x300 y con un color de 32 bits por píxel en pantalla.

-startup. Esta opción carga un script de tcl, dicho script se encarga de crear el contenido de la aplicación así como de ejecutar otros scripts relacionados con esta función, en el caso de Calakmul virtual los archivos de script que realizan estas funciones son calakmulgui.tcl si la aplicación va a correr bajo OpenGL y calakmulguid3d.tcl si se va a usar Direct 3D.

De las opciones anteriormente mostradas "startup" es la única obligatoria a especificar para que la aplicación funcione.

6.4. Menú de configuración de la aplicación

Se tendrá acceso al menú de configuración de la aplicación si se corre la aplicación a través del menú de inicio, por el icono de acceso directo que el instalador

crea en el escritorio o si simplemente se ejecuto el script Calakmultk.tcl. En la siguiente imagen se presenta el menú de configuración.

Figura 6.1. Menú de configuración de la aplicación.

El menú de configuración fue pensado como una herramienta sencilla para que el usuario a través de una interfaz gráfica tipo Windows pueda establecer el modo en el que quiere que corra la aplicación. Como se observa en la Figura el menú de configuración presenta las siguientes opciones.

-3D Device. Permite escoger el API con el que el servidor gráfico trabajará.
-Resolution. Permite escoger la resolución en la que se despliega la aplicación.
-BPP. Permite indicar el número de bits de color para cada píxel.
-Run. Con este botón se ejecuta la aplicación Calakmul virtual conservando las opciones que se escogieron en el menú de configuración.
-Close. Cierra el menú de configuración sin ejecutar la aplicación.

6.5. Menú de la aplicación

El menú de la aplicación es una ventana de TK la cual se crea al momento de correr la aplicación y tiene la función de proporcionarle al usuario una interfaz por medio de la cual sea más rápido y sencillo moverse dentro del mundo virtual. La siguiente figura muestra el menú de la aplicación.

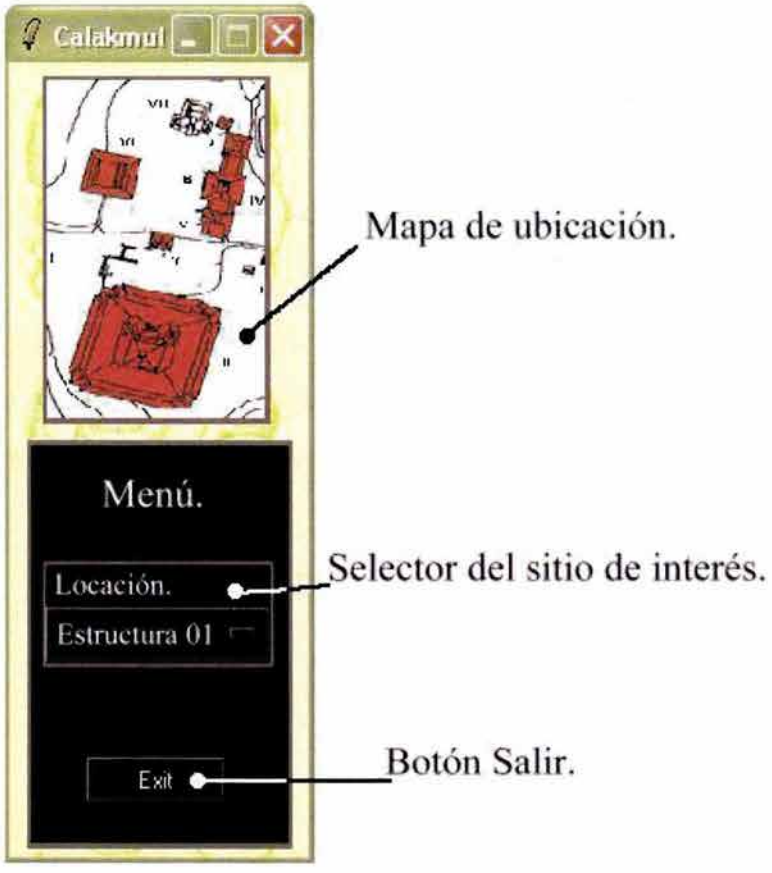


Figura 6.2. Menú de la aplicación.

Como se puede observar el menú de configuración está compuesto por los siguientes elementos.

-Mapa de ubicación. Como su nombre lo indica este mapa le informa al usuario la ubicación del personaje que controla dentro del mundo virtual, el mapa es una representación del sitio arqueológico visto desde arriba, la zona iluminada representa el área o pirámide en la que se encuentra el personaje.

-Selector del sitio de interés. La función de este control es trasladar al personaje controlado por el usuario de una locación a otra, de forma que el usuario sólo tiene que seleccionar de una lista el lugar que desea visitar para que la aplicación lo lleve ahí, evitando que el usuario tenga que realizar viajes en los que puede perder el interés.
Botón cerrar. Este botón cierra la aplicación.

-Botón exit. Cierra la aplicación.

6.6. Controles del personaje

Existen varias acciones que el usuario puede realizar dentro del mundo virtual, el usuario puede desplazarse libremente por el sitio, obtener información de varios de los objetos y estructuras e interactuar con los personajes controlados por el sistema. Para realizar todas estas acciones el usuario debe de controlar a un personaje, a continuación se indican los controles del personaje y la función de cada uno.

-Avanzar y retroceder. La acción de avanzar está controlada por la tecla "W", la acción de retroceder es controlada por la tecla "S", mientras más tiempo se deje presionado alguno de estos botones el personaje se moverá más rápido en esa dirección.

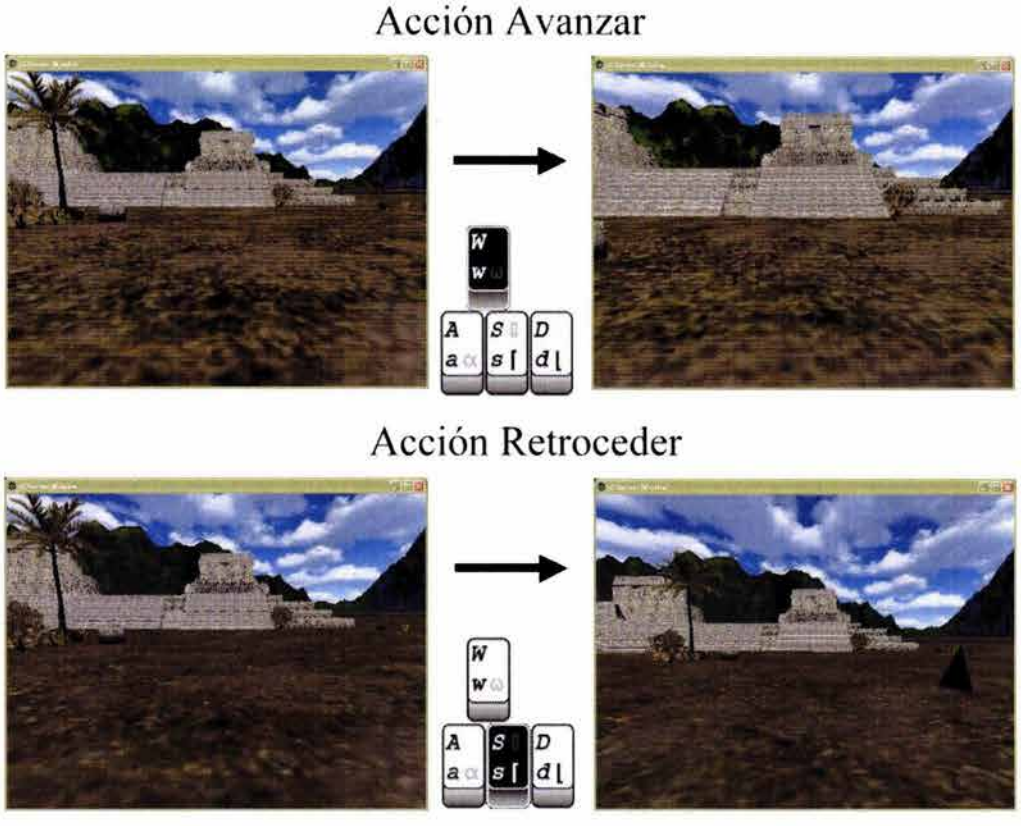


Figura 6.3. Control avanzar y retroceder.

-Movimiento lateral Izquierda y Derecha. La acción de moverse lateralmente a la izquierda está controlada por la tecla "A", la acción de moverse lateralmente a la derecha está controlada por la tecla "D", a diferencia de la acción de avanzar o retorcer la velocidad del personaje al moverse lateralmente siempre es constante.

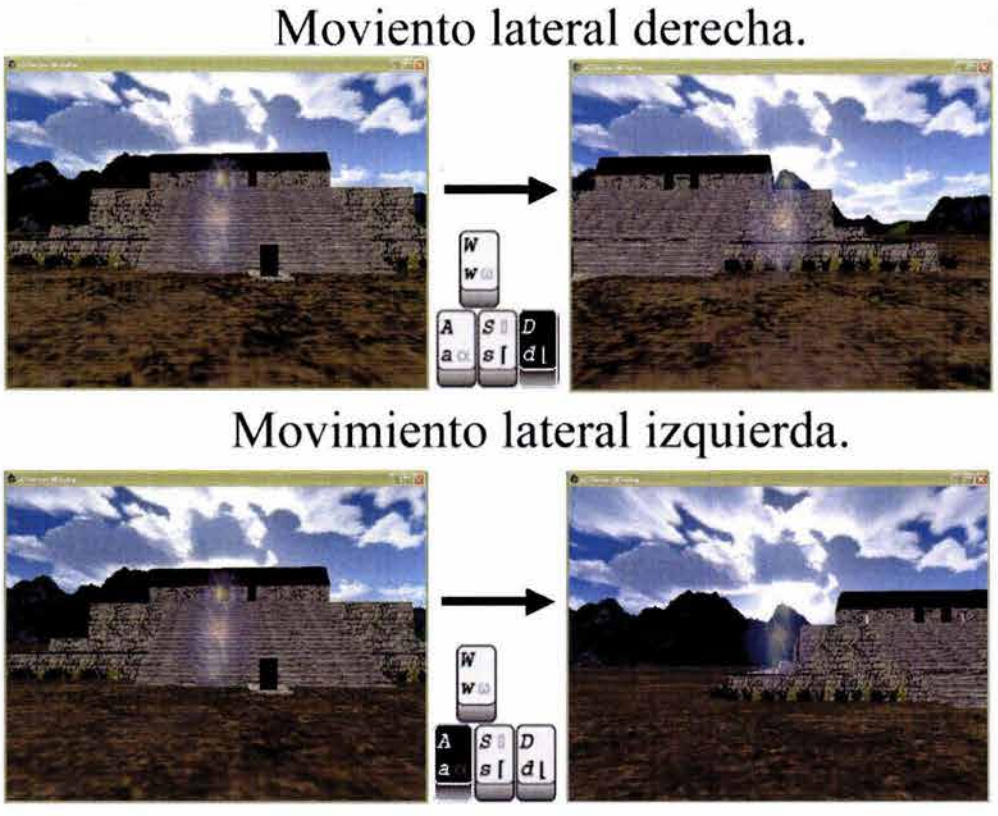


Figura 6.4. Control movimiento lateral.

-Giro horizontal y vertical. La acción de giro horizontal está controlada por el movimiento horizontal del ratón, cuando el ratón se mueve a la derecha el personaje gira en sentido horario, cuando el ratón se mueve a la izquierda el personaje gira en sentido antihorario. La acción de giro vertical está controlada por el movimiento vertical del ratón, cuando el ratón se mueve hacia arriba el ratón la vista gira hacia el cielo, cuando el ratón se mueve hacia abajo la vista gira hacia el suelo.

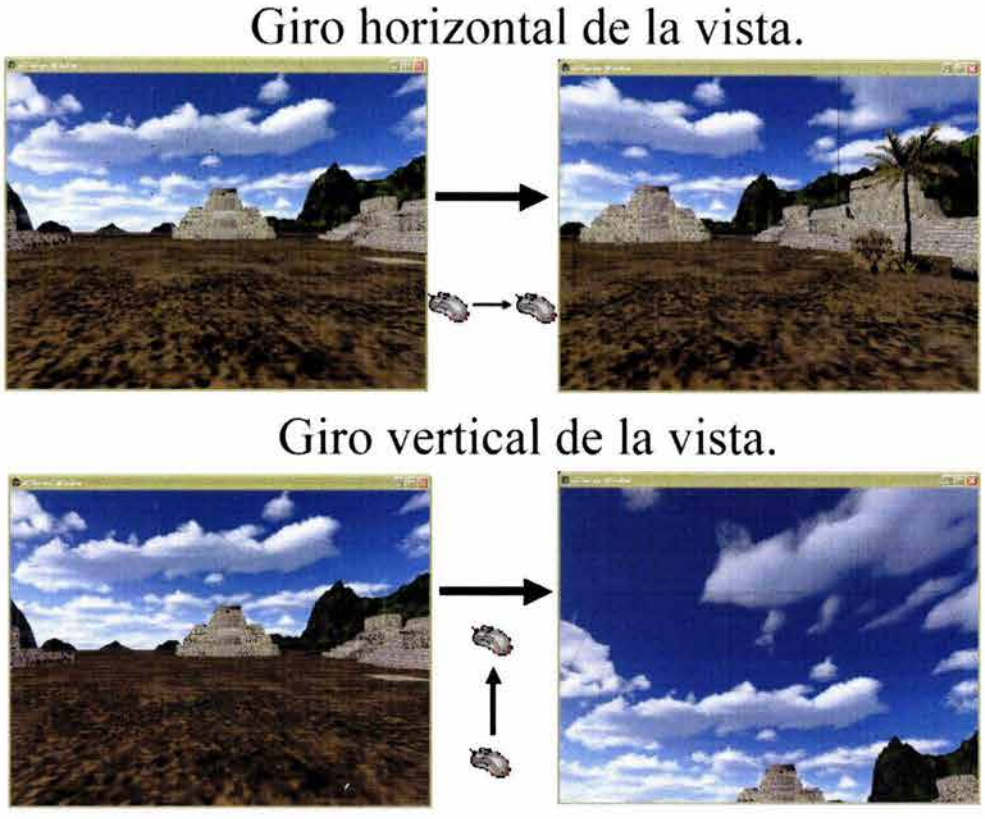


Figura 6.5. Control del giro horizontal y vertical

-Botón de interacción. La acción de interactuar está controlada por la barra espaciadora del teclado, a través de la acción de interactuar el usuario puede obtener información de los objetos y una gran cantidad de comportamientos de los personajes controlados por el sistema, para realizar la acción de interactuar sólo es necesario acercarse al objeto o personaje, centrarlo en la vista y presionar la barra espaciadora.

Figura 6.6. Control Interacción.

6.7. Como cerrar la aplicación

Para terminar la ejecución de la aplicación "Calakmul Virtual" sólo es necesario presionar el botón "Exit" que se encuentra en la parte inferior del menú de la aplicación [Ver 6.5.] o presionar el botón de cerrar, el botón representado por una cruz, de la ventana principal de la aplicación. Es altamente recomendado que se cierre la aplicación por alguno de los métodos anteriormente expuestos ya que de esta forma se asegura que tanto la memoria como los recursos usados por la aplicación queden libres.

Figura 6.7. Ubicación del botón cerrar.

Conclusiones.

Posterior a la realización del proceso de ingeniería de *software* orientado a objetos (OO),en el que se definió el problema y vislumbró la posible solución, se obtuvieron las especificaciones y requerimientos de la aplicación, se modeló el sistema que cumpliera con los requerimientos y se diseñaron e implementaron las clases obtenidas del análisis, es posible afirmar que el objetivo de este trabajo se ha cumplido. Se desarrolló una aplicación que transporta al usuario a las ruinas arqueológicas de Calakmul y que permite al usuario interactuar con la historia que el desarrollador concibió, o al menos definió, si se trata de un sistema automatizado encargado de realizar la visita virtual.

El producto final muestra que las clases diseñadas cumplen con sus responsabilidades de manera adecuada. La creación de la aplicación muestra que es factible utilizar esas clases y sus relaciones en un mundo virtual en el cual el usuario puede interactuar . En realidad, la última prueba que este proyecto tendría que pasar para decir que cumplió satisfactoriamente con su objetivo, es la de su utilización en el ambiente de trabajo para el cual fue diseñado, es decir que sea usado en un museo con todas las problemáticas y dificultades que esto implica.

El uso de un proceso de ingeniería de *software* orientado a objetos, además, ayudó a desarrollar la aplicación de manera más formal, en especial tratándose de un campo tan nuevo y con metodologías de diseño poco probadas. El rigor del análisis OO, permitió reconocer, rediseñar y reusar clases que de otra forma posiblemente no se habría hecho, en especial, si se pondera la poca experiencia que se tenía en el campo.

Con este proyecto se permite una mejor comprensión de una de las más antiguas civilizaciones de México, además proporciona una herramienta de estudio útil a los arqueólogos e investigadores así como favorece el turismo a un destino maya poco visitado y de difícil acceso por ahora.

Técnicamente la aplicación ofrece un despliegue tridimensional del lugar o sitio deseado, en el cual se le permite al usuario moverse a través de éste ofreciéndole la experiencia de estar en el sitio de interés, además puede interactuar con objetos y personas que se encuentren en el lugar, todo esto enfocado a que el usuario aprenda de la experiencia de usar el software. De manera general la implementación en software funciona para varios tipos de recorridos virtuales y no está limitado a un recorrido fijo en un solo lugar, por ejemplo en este caso es fácil cambiar el recorrido de Calakmul al de Palenque, que tanto el costo económico como el derivado del tiempo necesario para cambiar el sitio o zona arqueológica del recorrido no son muy elevados ya que la aplicación está orientada a ser usada en un museo donde la rotación y cambios que se le puedan hacer a las exhibiciones son muy importantes.

Entre algunas otras cosas el entorno tiene vida artificial, así como un agente inteligente que, personificado por un arqueólogo, ayuda al visitante a recorrer la ciudad. Todo esto con la finalidad de que el recorrido sea más interesante.

Volviendo con lo relativo al proceso de desarrollo, también se pudieron constatar las ventajas de utilizar librerías de *software* para la satisfacción de ciertos requisitos que ya han sido resueltos por alguien mas. Sin embargo, la mala o carente documentación del código puede complicar este proceso. Al final, considerando los puntos a favor y los puntos en contra, la reutilización de *software* siempre es recomendable, en especial cuando se trata de proyectos muy grandes. Después de todo, ¿quién quiere reinventar la rueda?.

En cuanto al alcance y relevancia de este trabajo. Se puede argumentar que por su enfoque no propiciará ningún cambio de ciento ochenta grados en la manera como se hacen los proyectos relacionados con las gráficas por computadoras ya que en realidad, ese nunca fue el propósito de éste. El enfoque que este trabajo muestra que es posible desarrollar en México aplicaciones y soluciones de software dentro de esta área y no sólo en áreas como las bases de datos, que aunque el campo de las gráficas por computadoras apenas se está dando a conocer en el país no es imposible trabajar en proyectos relacionados con el área.

En esta parte final del trabajo, es necesario admitir que la posibilidad de que esta aplicación sea utilizada en otros proyectos diferentes a los que se tenía previsto es reducida. De cualquier manera, con la realizaciónde éste se hizo la presentación y el análisis de diversos conceptos relacionados con las gráficas por computadora. Además, lo poco conocido del campo, implicó el uso de un proceso de ingeniería de *software* para el desarrollo de un sistema de cierta complejidad, labor pocas veces realizada durante el recorrido de un ingeniero en computación a lo largo de la carrera. Además, la creación de este tipo de aplicaciones a manos de la propia industria del entretenimiento digital es algo que ya se ve hoy en día, sin lugar a dudas. Compañías especializadas desarrollan motores de juegos con lo último en tecnología para sus propios video juegos, además estas compañías licencian sus motores para recuperar los costos de investigación y diseño, mientras que benefician a compañías más pequeñas a las que les sería imposible desarrollar algo tan complicado y que al licenciar el motor obtienen tecnología de punta.

La contribución de este trabajo es proponer una base de desarrollo para otros sistemas similares, proporcionándoles ciertas funcionalidades básicas así como una arquitectura recomendada. Además, este trabajo permite analizar la aplicación de la ingeniería de *software* orientada a objetos para el desarrollo de un grupo de clases que presenta una complejidad de implementación intermedia.

Apéndice A

Documentación de la aplicación Calakmul virtual

A.1. Índice de la documentación de clases

A.2. nistcamera Documentación de clases

Referencia de la Clase nlstCamera

Clase encargada de controlar el punto de vista desde el cual el usuario observa la escena.

`#include <nistcamera.h>`

Tipos públicos

enum **Style** { **STATIONARY**, **LOCKED_CHASE**, **CHASE**, **AIM_PATH**, **LOOK_PATH**, **FPS**, **FREE**, **PLANE** }

Métodos públicos

nlstCamera ()
Método constructor de la clase.

virtual void **Initialize** ()
Inicializa varios valores de la clase.

virtual **~nlstCamera** ()
Método destructor de la clase.

virtual bool **SaveCmds** (nPersistServer *fileServer)
Método encargado de la persistencia, salva un archivo de comandos.

const matrix44 & **GetTransform** () const
Obtiene la matriz de transformación de la cámara.

void **Trigger** (float dt)
Actualiza el estado del objeto.

void **Collide** (void)
Maneja el comportamiento del objeto cuando éste colisiona.

void **AddImpulse** (vector3 addv)
void **SetPosition** (float x, float y, float z)
Cambia la posición de la cámara.

void **SetStyleStationary** (vector3 targetPos, float angleX, float angleY, float angleZ, float dist)
Activa el modo estacionario de la cámara.

void **SetStyleStationary** (n3DNode *target, float angleX, float angleY, float angleZ, float dist)
Activa el modo estacionario de la cámara.

void **SetStyleStationary** (vector3 targetPos, vector3 cameraPos)
Activa el modo estacionario de la cámara.

void **SetStyleStationary** (n3DNode *target, n3DNode *camer)
Activa el modo estacionario de la cámara.

void **SetStyleChase** (n3DNode *target, float height, float prefDist)
Activa el modo de persecución de la cámara.

void **SetStyleLockedChase** (n3DNode *target, float angleX, float angleY, float angleZ, float dist)
Activa el modo de persecución fija de la cámara.

void **SetStyleAimPath** (n3DNode *target, const char *orig, const char *dest, float len)
Activa el modo en el que la cámara sigue un camino.

void **SetStyleLookPath** (const char *origPos, const char *destPos, const char *origAng, const char *destAng, float len)
Activa el modo de cámara en el que sigue un camino y su orientación es fija.

void **SetStyleFPS** (n3DNode *target, nIstPlayer *player, float height)
Activa el modo de primera persona de la cámara.

void **SetStyleFree** ()
Activa el modo libre de la cámara.

void **SetStylePlane** (float dist, vector3 targetPos)
void **EnableSelector** (bool s)
void **handleInput** (nInputEvent *ie)
Maneja las entradas del usuario de acuerdo al modo en que se encuentre la cámara.

quaternion **getQ** (void)
Obtiene la matriz de orientación de la cámara mediante un cuaternión.

vector3 **getT** (void)
Obtiene la posición de la cámara.

void **Qxyzw** (float x, float y, float z, float w)
Fija la matriz de orientación de la cámara mediante un cuaternión.

void **EnableEventColl** (const char *collScriptFile)
Habilita la ejecución de scripts por medio de eventos.

void **DisableEventColl** ()
Deshabilita la ejecución de scripts por medio de eventos.

nIstEntity * **GetLastCollEntity** (void)
Regresa la última entidad con la que colisiono el objeto.

Atributos públicos estáticos

nKernelServer * **kernelServer**
pointer to nKernelServer

nClass * **loacl_cl**

Métodos privados

void **handleViewer** (float t)
void **handleChaseViewer** (void)
void **handleFPSViewer** (float t)
vector3 * **SeekPath** (float dt)
quaternion * **SeekAngPath** (float dt)

Atributos privados

nAutoRef< nInputServer > **ref_is**
nAutoRef< nGfxServer > **ref_gs**
n3DNode * **cameraNode**
matrix44 **cameraMat**
matrix44 **targetMat**
n3DNode * **myTarget**
nIstPlayer * **myPlayer**
nPath * **actualPath**
bool **seekingBack**
float **pathLength**
float **pathTime**
nPath * **actualAngPath**
bool **seekingAngBack**
float **fpsHeight**
float **yLast**
float **chaseCamHeight**
float **chaseDistance**
float **chaseAngleRad**
float **planeDist**
nIstCamera::Style **cameraStyle**
nIstCamera::Style **lastCamStyle**
bool **cur_up**
bool **cur_down**
bool **cur_left**
bool **cur_right**
bool **ctrl**
bool **shift**
bool **lmb**
bool **mmb**
bool **rmb**
bool **normalize**
long **old_x**
long **old_y**
long **act_x**
long **act_y**
int **scrWidth**

int **scrHeight**
bool **xpbtn**
bool **xnbtn**
bool **ypbtn**
bool **ynbtn**
float **x_abs_ang**
float **xpang**
float **xnang**
float **ypang**
float **ynang**
float **distance**
matrix44 **Rx**
matrix44 **Ry**
bool **EnSelection**
nTclScriptlet * **collScriptlet**
bool **EnEvColl**
nlstEntity * **lastCollEntity**

Descripción detallada

Clase encargada de controlar el punto de vista desde el cual el usuario observa la escena.

Clase encargada de controlar el punto de vista desde el cual el usuario observa la escena. Esta clase contiene información acerca del punto de vista desde el cual se dibuja la escena. esta información es almacenada en una matriz de 4X4, también se encarga de manejar las entradas del usuario.

Documentación del constructor y destructor

nlstCamera::nlstCamera ()

Método constructor de la clase.

Este método construye nuevas instancias de la clase.

virtual nlstCamera::~nlstCamera () `[virtual]`

Método destructor de la clase.

Este método destruye instancias creadas de la clase.

Documentación de las funciones miembro

void nlstCamera::Collide (void)

Maneja el comportamiento del objeto cuando éste colisiona.

Se identifica si ha ocurrido una colisión y se simula el comportamiento de la colisión.

void nlstCamera::DisableEventColl () [inline]

Deshabilita la ejecución de scripts por medio de eventos.

void nlstCamera::EnableEventColl (const char * *collScriptFile*) [inline]

Habilita la ejecución de scripts por medio de eventos.

Parámetros:

collscriptfile nombre del archivo que contiene el script a ejecutar.

nlstEntity * nlstCamera::GetLastCollEntity (void) [inline]

Regresa la última entidad con la que colisiono el objeto.

Devuelve:

El nombre del objeto con el que colisiono.

quaternion nlstCamera::getQ (void) [inline]

Obtiene la matriz de orientación de la cámara mediante un cuaternión.

Devuelve:

Un cuaternión, el cual conriene la orientación actual de la cámara.

vector3 nlstCamera::getT (void) [inline]

Obtiene la posición de la cámara.

Devuelve:

Un vector cuyas tres componentes coresponden a un punto de cordenadas XYZ en el espacio.

const matrix44 & nlstCamera::GetTransform () const [inline]

Obtiene la matriz de transformación de la cámara.

Con esta matriz se obtiene la posición y orientación desde la cual se observa la escena.

Devuelve:

Matriz de posición y orientación de la cámara.

void nlstCamera::handleInput (nInputEvent * *ie*)

Maneja las entradas del usuario de acuerdo al modo en que se encuentre la cámara.

Si el modo de la cámara no requiere de este método simplemente se deja desatendido, el estilo libre, de persecución y de primera persona requieren entras del usuario.

Parámetros:

ie evento de entrada que debe ser generado por el servidor de entradas.

virtual void nIstCamera::Initialize () `[virtual]`

Inicializa varios valores de la clase.

Con la llamada a este método se crean y agregan los scripts para cada acción.

void nIstCamera::Qxyzw (float *x*, float *y*, float *z*, float *w*) `[inline]`

Fija la matriz de orientación de la cámara mediante un cuaternión.

virtual bool nIstCamera::SaveCmds (nPersistServer * *fileServer*) `[virtual]`

Método encargado de la persistencia, salva un archivo de comandos.

void nIstCamera::SetPosition (float *x*, float *y*, float *z*)

Cambia la posición de la cámara.

Ubica a la cámara en la posición indicada por xyz, esto sólo funciona cuando la cámara se encuentra en un modo estacionario.

Parámetros:

x posición eje X, y posición eje Y, z posición eje Z.

void nIstCamera::SetStyleAimPath (n3DNode * *target*, const char * *orig*, const char * *dest*, float *len*)

Activa el modo en el que la cámara sigue un camino.

En este modo la cámara sigue un camino indicado por una curva sin dejar de apuntar a un objeto.

Parámetros:

target objeto de escena al que apunta la cámara, orig definición del punto de origen del camino, dest definición del punto de destino, len tiempo en el que se desea recorrer el camino.

void nIstCamera::SetStyleChase (n3DNode * *target*, float *height*, float *prefDist*)

Activa el modo de persecución de la cámara.

En este modo la cámara sigue al objeto indicado a una cierta altura y distancia definidas.

Parámetros:

target objeto de escena al que persigue la cámara, height altura al que se coloca la cámara, prefDist distancia a la cual se mantiene la cámara del objeto que persigue.

void nlstCamera::SetStyleFPS (n3DNode * *target*, nlstPlayer * *player*, float *height*)

Activa el modo de primera persona de la cámara.

En este modo la cámara le ofrece al usuario el punto de vista que tiene un personaje dentro del mundo virtual.

Parámetros:

target personaje de la escena del cual se toma la vista de primera persona, player objeto personaje asociado con al nodo 3d target, height altura al que se coloca la cámara.

void nlstCamera::SetStyleFree () `[inline]`

Activa el modo libre de la cámara.

En este modo el usuario puede controlar libremente la cámara.

void nlstCamera::SetStyleLockedChase (n3DNode * *target*, float *angleX*, float *angleY*, float *angleZ*, float *dist*)

Activa el modo de persecución fija de la cámara.

En este modo la cámara sigue al objeto indicado sin cambiar de posición.

Parámetros:

target objeto de escena al que sigue la cámara, angleX orientación en el eje x de la cámara con respecto al objetivo, angley orientación en el eje y de la cámara con respecto al objetivo, anglez orientación en el eje z de la cámara con respecto al objetivo, dist distancia de la cámara con respecto al objetivo.

void nlstCamera::SetStyleLookPath (const char * *origPos*, const char * *destPos*, const char * *origAng*, const char * *destAng*, float *len*)

Activa el modo de cámara en el que sigue un camino y su orientación es fija.

En este modo la cámara sigue un camino indicado por una curva apuntando siempre en la misma dirección.

Parámetros:

origPos definición del punto de origen del camino, destPos definición del punto de destino, origAng definición del punto de origen de la ruta de la cámara, destAng definición del punto de destino de la ruta de la cámara, len tiempo en el que se desea recorrer el camino.

void nlstCamera::SetStyleStationary (n3DNode * *target*, n3DNode * *camer*)

Activa el modo estacionario de la cámara.

Parámetros:

target objeto de escena al que apunta la cámara, camer objeto de escena con la posición de la cámara.

void nlstCamera::SetStyleStationary (vector3 *targetPos*, vector3 *cameraPos*)

Activa el modo estacionario de la cámara.

Parámetros:

targetPos vector con la posición del objeto al que apunta la cámara, cameraPos vector con la posición de la cámara.

void nlstCamera::SetStyleStationary (n3DNode * *target*, float *angleX*, float *angleY*, float *angleZ*, float *dist*)

Activa el modo estacionario de la cámara.

Parámetros:

target objeto al que apunta la cámara, angleX orientación en el eje x de la cámara con respecto al objetivo, angley orientación en el eje y de la cámara con respecto al objetivo, anglez orientación en el eje z de la cámara con respecto al objetivo, dist distancia de la cámara con respecto al objetivo.

void nlstCamera::SetStyleStationary (vector3 *targetPos*, float *angleX*, float *angleY*, float *angleZ*, float *dist*)

Activa el modo estacionario de la cámara.

Parámetros:

targetPos vector con la posición del objeto al que apunta la cámara, angleX orientación en el eje x de la cámara con respecto al objetivo, angley orientación en el eje y de la cámara con respecto al objetivo, anglez orientación en el eje z de la cámara con respecto al objetivo, dist distancia de la cámara con respecto al objetivo.

void nlstCamera::Trigger (float *dt*)

Actualiza el estado del objeto.

Se identifica el modo de operación de la cámara y se llama al método manejador para dicho modo.

Parámetros:

dt intervalo de tiempo transcurrido desde la última vez que se llamo al método Trigger.

A.3. nistcollzone Documentación de clases

Referencia de la Clase nIstCollZone

Clase encargada de crear y controlar las zonas de colisión.

`#include <nIstCollZone.h>`

Métodos públicos

nIstCollZone ()
constructor

virtual **~nIstCollZone** ()
destructor

virtual void **Initialize** ()
Inicializa varios valores de la clase.

virtual bool **SaveCmds** (nPersistServer *fileServer)
persistency

void **Collide** ()
Maneja el comportamiento o eventos del objeto cuando éste colisiona.

void **Trigger** (float dt)
Actualiza el estado del objeto.

void **CreateZone** (const char *meshfile, float x, float y, float z)
Método que crea una nueva zona de colisión.

void **EnableEventColl** (const char *collScriptFile)
Habilita la ejecución de scripts por medio de eventos.

void **DisableEventColl** ()
Deshabilita la ejecución de scripts por medio de eventos.

nIstEntity * **GetLastCollEntity** (void)
Regresa la última entidad con la que colisiono el objeto.

Atributos públicos estáticos

nKernelServer * **kernelServer**
pointer to nKernelServer

Atributos privados

nAutoRef< nGfxServer > **ref_gs**
nAutoRef< nInputServer > **ref_is**
nAutoRef< nConServer > **ref_con**
nAutoRef< nScriptServer > **ref_ss**
nAutoRef< nSceneGraph2 > **ref_sg**
nAutoRef< nCollideServer > **ref_collide**
nAutoRef< n3DNode > **ref_objectnode**
n3DNode * **objectnode**
nMeshNode * **mesh**
nString **meshF**
bool **EnEvColl**
nString **collScriptFile**
nIstEntity * **lastCollEntity**

Descripción detallada

Clase encargada de crear y controlar las zonas de colisión.

Esta clase es un contenedor que mantiene una referencia a una zona con la cual pueden darse colisiones diferenciandose de los objetos al no tener una representación gráfica y no afectar la física del personaje controlado por el usuario, esta clase presenta métodos para crear las zonas de colisión así como métodos para manejar los eventos producto de una colisión.

Documentación del constructor y destructor

nIstCollZone::nIstCollZone ()

constructor

Este método construye nuevas instancias de la clase.

virtual nIstCollZone::~nIstCollZone () `[virtual]`

destructor

Este método destruye instancias creadas de la clase.

Documentación de las funciones miembro

void nIstCollZone::Collide ()

Maneja el comportamiento o eventos del objeto cuando éste colisiona.

void nlstCollZone::CreateZone (const char * *meshfile*, float *x*, float *y*, float *z*)

Método que crea una nueva zona de colisión.

La finalidad de este método es ofrecer una forma sencilla al programador de crear y agregar nuevas zonas de colisión.

Parámetros:

meshfile nombre del archivo que contiene la geometría con la que se colisiona, x posición en el eje X, y posición en el eje Y, z posición en el eje Z.

void nlstCollZone::DisableEventColl () `[inline]`

Deshabilita la ejecución de scripts por medio de eventos.

void nlstCollZone::EnableEventColl (const char * *collScriptFile*) `[inline]`

Habilita la ejecución de scripts por medio de eventos.

Parámetros:

collscriptfile nombre del archivo que contiene el script a ejecutar.

nlstEntity * nlstCollZone::GetLastCollEntity (void) `[inline]`

Regresa la última entidad con la que colisiono el objeto.

Devuelve:

El nombre del objeto con el que colisiono.

virtual void nlstCollZone::Initialize () `[virtual]`

Inicializa varios valores de la clase.

Con la llamada a este método se crean y agregan los scripts para cada acción.

virtual bool nlstCollZone::SaveCmds (nPersistServer * *fileServer*) `[virtual]`

persistency

void nlstCollZone::Trigger (float *dt*)

Actualiza el estado del objeto.

Parámetros:

dt intervalo de tiempo transcurrido desde la última vez que se llamo al método Trigger.

A.4. nistobject Documentación de clases

Referencia de la Clase nIstObject

Clase encargada de representar a los objetos que existen dentro del mundo virtual.

`#include <nIstObject.h>`

Métodos públicos

nIstObject ()
Método constructor de la clase.

virtual **~nIstObject** ()
Método destructor de la clase.

virtual void **Initialize** ()
Inicializa varios valores de la clase.

virtual bool **SaveCmds** (nPersistServer *fileServer)
Método encargado de la persistencia, salva un archivo de comandos.

void **Collide** ()
Maneja el comportamiento o eventos del objeto cuando éste colisiona.

void **Trigger** (float dt)
Actualiza el estado del objeto.

void **AddImpulse** (vector3 addv)
Usado en las colisiones para repeler el objeto en movimiento.

void **CreateModel** (const char *meshfile, const char *texfile, bool lightEn, float x, float y, float z)
Método optimizado para crear la representación gráfica de un objeto.

void **SetImage** (const char *i)
void **EnableEventColl** (const char *collScriptFile)
Habilita la ejecución de scripts por medio de eventos.

void **DisableEventColl** ()
Deshabilita la ejecución de scripts por medio de eventos.

nIstEntity * **GetLastCollEntity** (void)
Regresa la última entidad con la que colisiono el objeto.

Atributos públicos estáticos

nKernelServer * **kernelServer**
pointer to nKernelServer

Métodos privados

void **CreateShader** (void)

Atributos privados

nAutoRef< nGfxServer > **ref_gs**
nAutoRef< nInputServer > **ref_is**
nAutoRef< nConServer > **ref_con**
nAutoRef< nScriptServer > **ref_ss**
nAutoRef< nSceneGraph2 > **ref_sg**
nAutoRef< nCollideServer > **ref_collide**
nAutoRef< n3DNode > **ref_objectnode**
nTclScriptlet * **collScriptlet**
n3DNode * **objectnode**
nMeshNode * **mesh**
nShaderNode * **shader**
nTexArrayNode * **tex**
nString **meshF**
nString **texF**
bool **lightE**
bool **EnEvColl**
nIstEntity * **lastCollEntity**
const char * **imgDir**

Descripción detallada

Clase encargada de representar a los objetos que existen dentro del mundo virtual.

Esta clase se puede ver como un contenedor que mantiene una referencia a la representación gráfica de un objeto. Además presenta un método con la función de cargar de manera sencilla la representación gráfica del objeto.

Documentación del constructor y destructor

nIstObject::nIstObject ()

Método constructor de la clase.

Este método construye nuevas instancias de la clase.

virtual nIstObject::~nIstObject () `[virtual]`

Método destructor de la clase.

Este método destruye instancias creadas de la clase.

Documentación de las funciones miembro

void nlstObject::AddImpulse (vector3 *addv*)

Usado en las colisiones para repeler el objeto en movimiento.

void nlstObject::Collide ()

Maneja el comportamiento o eventos del objeto cuando éste colisiona.

void nlstObject::CreateModel (const char * *meshfile*, const char * *texfile*, bool *lightEn*, float *x*, float *y*, float *z*)

Método optimizado para crear la representación gráfica de un objeto.

Parámetros:

meshfile nombre del archivo que contiene el modelo 3d del objeto, texfile Nombre del archivo que contiene la textura del objeto, lightEn Habilita la iluminación del objeto, x posición en el eje X, y posición en el eje Y, z posición en el eje Z.

void nlstObject::DisableEventColl () `[inline]`

Deshabilita la ejecución de scripts por medio de eventos.

void nlstObject::EnableEventColl (const char * *collScriptFile*) `[inline]`

Habilita la ejecución de scripts por medio de eventos.

Parámetros:

collscriptfile nombre del archivo que contiene el script a ejecutar.

nlstEntity * nlstObject::GetLastCollEntity (void) `[inline]`

Regresa la última entidad con la que colisiono el objeto.

Devuelve:

El nombre del objeto con el que colisiono.

virtual void nlstObject::Initialize () `[virtual]`

Inicializa varios valores de la clase.

Con la llamada a este método se crean y agregan los scripts para cada evento.

virtual bool nIstObject::SaveCmds (nPersistServer * *fileServer*) `[virtual]`

Método encargado de la persistencia, salva un archivo de comandos.

void nIstObject::Trigger (float *dt*)

Actualiza el estado del objeto.

Parámetros:

dt intervalo de tiempo transcurrido desde la última vez que se llamo al método Trigger.

A.5. niststruct Documentación de clases

Referencia de la Clase nIstStruct

Clase encargada de representar a las estructuras que existen dentro del mundo virtual.

`#include <nIstStruct.h>`

Métodos públicos

nIstStruct ()

Método constructor de la clase.

virtual **~nIstStruct** ()

Método destructor de la clase.

virtual bool **SaveCmds** (nPersistServer *fileServer)

Método encargado de la persistencia, salva un archivo de comandos.

void **Collide** ()

Maneja el comportamiento o eventos del objeto cuando éste colisiona.

void **Trigger** (float dt)

Actualiza el estado del objeto.

void **AddImpulse** (vector3 addv)

Usado en las colisiones para repeler el objeto en movimiento.

void **EnableEventColl** (const char *collScriptFile)

Habilita la ejecución de scripts por medio de eventos.

void **DisableEventColl** (void)

Deshabilita la ejecución de scripts por medio de eventos.

bool **GetSelected** (void)

Atributos públicos estáticos

nKernelServer * **kernelServer**
pointer to nKernelServer

Atributos privados

nAutoRef< nGfxServer > **ref_gs**
nAutoRef< nScriptServer > **ref_ss**
nAutoRef< nSceneGraph2 > **ref_sg**
nAutoRef< nCollideServer > **ref_collide**
nAutoRef< nIstWorld > **ref_world**
bool **EnEvColl**
nString **collScriptFile**
bool **Selected**

Descripción detallada

Clase encargada de representar a las estructuras que existen dentro del mundo virtual.

Esta clase es un contenedor dentro de nIstWorld y contiene una liga a la representación gráfica de la estructura.

Documentación del constructor y destructor

nIstStruct::nIstStruct ()

Método constructor de la clase.
Este método construye nuevas instancias de la clase.

virtual nIstStruct::~nIstStruct () `[virtual]`

Método destructor de la clase.
Este método destruye instancias creadas de la clase.

Documentación de las funciones miembro

void nIstStruct::AddImpulse (vector3 *addv*)

Usado en las colisiones para repeler el objeto en movimiento.

Parámetros:

addv Vector que indica la dirección del impulso.

void nIstStruct::Collide ()

Maneja el comportamiento o eventos del objeto cuando éste colisiona.

void nIstStruct::DisableEventColl (void) `[inline]`

Deshabilita la ejecución de scripts por medio de eventos.

void nIstStruct::EnableEventColl (const char * *collScriptFile*) `[inline]`

Habilita la ejecución de scripts por medio de eventos.

Parámetros:

collscriptfile nombre del archivo que contiene el script a ejecutar.

virtual bool nIstStruct::SaveCmds (nPersistServer * *fileServer*) `[virtual]`

Método encargado de la persistencia, salva un archivo de comandos.

void nIstStruct::Trigger (float *dt*)

Actualiza el estado del objeto.

Parámetros:

dt intervalo de tiempo transcurrido desde la última vez que se llamo al método Trigger.

A.6. niststairs Documentación de clases

Referencia de la Clase nIstStairs

`#include <nIstStairs.h>`

Métodos públicos

nIstStairs ()

Método constructor de la clase.

virtual **~nIstStairs** ()

Método destructor de la clase.

virtual bool **SaveCmds** (nPersistServer *fileServer)

Método encargado de la persistencia, salva un archivo de comandos.

void **Collide** ()

Maneja el comportamiento o eventos del objeto cuando éste colisiona.

void **Trigger** (float dt)
Actualiza el estado del objeto.

void **AddImpulse** (vector3 addv)
Usado en las colisiones para repeler el objeto en movimiento.

Atributos públicos estáticos

nKernelServer * **kernelServer**
pointer to nKernelServer

Atributos privados

nAutoRef< nGfxServer > **ref_gs**
nAutoRef< nScriptServer > **ref_ss**
nAutoRef< nSceneGraph2 > **ref_sg**
nAutoRef< nCollideServer > **ref_collide**

Descripción detallada

encargada de crear y controlar las estructuras en las que el personaje controlado por el usuario puede subir.

La función de esta clase en escencia es la misma a la de la clase nIstStruct, la diferencia entre estas dos clases radica en el comportamiento que estas producen en el personaje controlado por el usuario al ocurrir una colisión.

Documentación del constructor y destructor

nIstStairs::nIstStairs ()

Método constructor de la clase.

Este método construye nuevas instancias de la clase.

virtual nIstStairs::~nIstStairs () `[virtual]`

Método destructor de la clase.

Este método destruye instancias creadas de la clase.

Documentación de las funciones miembro

void nIstStairs::AddImpulse (vector3 *addv*)

Usado en las colisiones para repeler el objeto en movimiento.

Parámetros:

addv Vector que indica la dirección del impulso.

void nIstStairs::Collide ()

Maneja el comportamiento o eventos del objeto cuando éste colisiona.

virtual bool nIstStairs::SaveCmds (nPersistServer * *fileServer*) `[virtual]`

Método encargado de la persistencia, salva un archivo de comandos.

void nIstStairs::Trigger (float *dt*)

Actualiza el estado del objeto.

Parámetros:

dt intervalo de tiempo transcurrido desde la última vez que se llamo al método Trigger.

A.7. nistplayer Documentación de clases

Referencia de la Clase nIstPlayer

Clase encargada de recibir las entradas del usuario y de controlar al personaje.

`#include <nIstPlayer.h>`

Métodos públicos

nIstPlayer ()

Método constructor de la clase.

virtual **~nIstPlayer** ()

Método destructor de la clase.

virtual bool **SaveCmds** (nPersistServer *fileServer)

Método encargado de la persistencia, salva un archivo de comandos.

void **Trigger** (float dt)

Actualiza el estado del objeto.

void **Collide** (void)

Maneja el comportamiento del objeto cuando éste colisiona.

void **UpdatePosition** (float dt)

Actualiza la posición del personaje.

void **AddImpulse** (vector3 addv)

Usado en las colisiones para repeler el objeto en movimiento.

void **StartFW** ()
Coloca al personaje controlado por el usuario en el estado de caminar hacia delante.

void **StartBW** ()
Coloca al personaje controlado por el usuario en el estado de caminar hacia atras.

void **StartRL** ()
Coloca al personaje controlado por el usuario en el estado de girar a la izquierda.

void **StartRR** ()
Coloca al personaje controlado por el usuario en el estado de girar a la derecha.

void **StartSL** ()
Coloca al personaje controlado por el usuario en el estado de moverse lateralmente a la izquierda.

void **StartSR** ()
Coloca al personaje controlado por el usuario en el estado de moverse lateralmente a la derecha.

void **StartMRR** ()
Coloca al personaje controlado por el usuario en el estado de girar a la derecha por acción del ratón.

void **StartMRL** ()
Coloca al personaje controlado por el usuario en el estado de girar a la izquierda por acción del ratón.

void **StopFW** ()
Deteniene la acción de caminar hacia delenate.

void **StopBW** ()
Deteniene la acción de caminar hacia atras.

void **StopRL** ()
Deteniene la acción de girar a la izquierda.

void **StopRR** ()
Deteniene la acción de girar a la derecha.

void **StopSL** ()
Deteniene la acción de moverse lateralmente a la izuierda.

void **StopSR** ()
Deteniene la acción de moverse lateralmente a la derecha.

void **StopMRR** ()
Deteniene la acción del giro a la derecha controlado por el ratón.

void **StopMRL** ()
Deteniene la acción del giro a la izquierda controlado por el ratón.

void **ActShoot** ()
Cambia la animación del personaje.

void **SetRotSpeed** (float rs)
Fija la velocidad de giro del personaje.

void **SetMaxRunSpeed** (float ms)
fija la velocidad máxima que puede alcanzar el personaje controlado por el usuario al avanzar.

void **SetRunAccel** (float ac)
Fija la aceleración constante que tiene el personaje al avanzar.

float **GetRotSpeed** ()
Obtiene la velocidad de giro del personaje.

float **GetMaxRunSpeed** ()
Obtiene la velocidad máxima que puede alcanzar el personaje controlado por el usuario al avanzar.

float **GetRunAccel** ()
Obtiene la aceleración constante que tiene el personaje al avanzar.

void **SetN3DNodeTar** (const char *path)
Establece el nodo 3d que contiene el modelo del personaje.

const char * **GetN3DNodeTar** () const
Indica el nodo 3d que contiene el modelo del personaje.

void **SetNCalModTar** (const char *path)
Fija el modelo que representa al personaje.

const char * **GetNCalModTar** () const
Indica el modelo que representa al personaje.

void **SetCollServ** (const char *path)
Fija el servidor de colisiones.

const char * **GetCollServ** () const
Indica el servidor de colisiones.

Atributos públicos estáticos

nKernelServer * **kernelServer**

Atributos privados

nAutoRef< nGfxServer > **ref_gs**
nAutoRef< nInputServer > **ref_is**
nAutoRef< nConServer > **ref_con**
nAutoRef< nScriptServer > **ref_ss**
nAutoRef< nSceneGraph2 > **ref_sg**
nAutoRef< nCollideServer > **ref_collide**
n3DNode * **playernode**
nCal3DModel * **model**
float **run_accel**
float **max_speed**
float **rot_speed**
bool **isFW**
bool **isBW**
bool **isRL**
bool **isRR**
bool **isSTPFW**
bool **isSTPBW**
bool **isSL**
bool **isSR**
bool **isSTPSL**
bool **isSTPSR**
bool **isMRL**
bool **isMRR**
nString **splayernode**
nString **smodel**
bool **animWalk**

Descripción detallada

Clase encargada de recibir las entradas del usuario y de controlar al personaje.

Mediante las entradasdel usuario el servidor de entradas llama a métodos de esta clase, estos controlan y actualizan la animación y posición del modelo que representa al usuario.

Documentación del constructor y destructor

nlstPlayer::nlstPlayer ()

Método constructor de la clase.

Este método construye nuevas instancias de la clase.

virtual nlstPlayer::~nlstPlayer () `[virtual]`

Método destructor de la clase.

Este método destruye instancias creadas de la clase.

Documentación de las funciones miembro

void nlstPlayer::ActShoot () [inline]

Cambia la animación del personaje.

void nlstPlayer::AddImpulse (vector3 *addv*) [inline]

Usado en las colisiones para repeler el objeto en movimiento.

Parámetros:

addv Vector que indica la dirección del impulso.

void nlstPlayer::Collide (void)

Maneja el comportamiento del objeto cuando éste colisiona.

Se identifica si ha ocurrido una colisión, se identifica la clase del objeto y se simula el comportamiento de la colisión con dicho objeto.

const char * nlstPlayer::GetCollServ () const [inline]

Indica el servidor de colisiones.

Devuelve:

Ruta dentro de la jerarquía de Nebula donde se encuentra el servidor de colisiones.

float nlstPlayer::GetMaxRunSpeed () [inline]

Obtiene la velocidad máxima que puede alcanzar el personaje controlado por el usuario al avanzar.

Devuelve:

Máxima velocidad que se puede alcanzar en m/s.

const char * nlstPlayer::GetN3DNodeTar () const [inline]

Indica el nodo 3d que contiene el modelo del personaje.

Devuelve:

Ruta de la jeraquía de Nebula en la que se encuentra el nodo que contiene al modelo.

const char * nlstPlayer::GetNCalModTar () const [inline]

Indica el modelo que representa al personaje.

Devuelve:

Ruta en la que se encuentra el archivo que contiene al modelo.

float nlstPlayer::GetRotSpeed () `[inline]`

Obtiene la velocidad de giro del personaje.

Devuelve:

Velocidad de giro en grados/segundos.

float nlstPlayer::GetRunAccel () `[inline]`

Obtiene la aceleración constante que tiene el personaje al avanzar.

Devuelve:

Aceleración del personaje.

virtual bool nlstPlayer::SaveCmds (nPersistServer * *fileServer*) `[virtual]`

Método encargado de la persistencia, salva un archivo de comandos.

void nlstPlayer::SetCollServ (const char * *path*) `[inline]`

Fija el servidor de colisiones.

Parámetros:

path ruta dentro de la jerarquía de Nebula donde se encuentra el servidor de colisiones.

void nlstPlayer::SetMaxRunSpeed (float *ms*) `[inline]`

fija la velocidad máxima que puede alcanzar el personaje controlado por el usuario al avanzar.

Parámetros:

ms Máxima velocidad que se puede alcanzar en m/s.

void nlstPlayer::SetN3DNodeTar (const char * *path*) `[inline]`

Establece el nodo 3d que contiene el modelo del personaje.

Parámetros:

path ruta de la jeraquía de Nebula en la que se encuentra el nodo que contiene al modelo.

void nlstPlayer::SetNCalModTar (const char * *path*) `[inline]`

Fija el modelo que representa al personaje.

Parámetros:

path ruta en la que se encuentra el archivo que contiene al modelo.

void nlstPlayer::SetRotSpeed (float *rs*) `[inline]`

Fija la velocidad de giro del personaje.

Parámetros:

rs Velocidad de giro en grados/segundos.

void nlstPlayer::SetRunAccel (float *ac*) `[inline]`

Fija la aceleración constante que tiene el personaje al avanzar.

Parámetros:

ac aceleración del personaje.

void nlstPlayer::StartBW () `[inline]`

Coloca al personaje controlado por el usuario en el estado de caminar hacia atras.

void nlstPlayer::StartFW () `[inline]`

Coloca al personaje controlado por el usuario en el estado de caminar hacia delante.

void nlstPlayer::StartMRL () `[inline]`

Coloca al personaje controlado por el usuario en el estado de girar a la izquierda por acción del ratón.

void nlstPlayer::StartMRR () `[inline]`

Coloca al personaje controlado por el usuario en el estado de girar a la derecha por acción del ratón.

void nlstPlayer::StartRL () `[inline]`

Coloca al personaje controlado por el usuario en el estado de girar a la izquierda.

void nlstPlayer::StartRR () `[inline]`

Coloca al personaje controlado por el usuario en el estado de girar a la derecha.

void nlstPlayer::StartSL () `[inline]`

Coloca al personaje controlado por el usuario en el estado de moverse lateralmente a la izquierda.

void nlstPlayer::StartSR () `[inline]`

Coloca al personaje controlado por el usuario en el estado de moverse lateralmente a la derecha.

void nlstPlayer::StopBW () `[inline]`

Deteniene la acción de caminar hacia atras.

void nlstPlayer::StopFW () [inline]

Deteniene la acción de caminar hacia delenate.

void nlstPlayer::StopMRL () [inline]

Deteniene la acción del giro a la izquierda controlado por el ratón.

void nlstPlayer::StopMRR () [inline]

Deteniene la acción del giro a la derecha controlado por el ratón.

void nlstPlayer::StopRL () [inline]

Deteniene la acción de girar a la izquierda.

void nlstPlayer::StopRR () [inline]

Deteniene la acción de girar a la derecha.

void nlstPlayer::StopSL () [inline]

Deteniene la acción de moverse lateralmente a la izuierda.

void nlstPlayer::StopSR () [inline]

Deteniene la acción de moverse lateralmente a la derecha.

void nlstPlayer::Trigger (float *dt*)

Actualiza el estado del objeto.

En este método se actualizan las animaciones y movimientos del personaje de acuardo con la entrada que de el usuaio.

Parámetros:

dt intervalo de tiempo transcurrido desde la última vez que se llamo al método Trigger.

void nlstPlayer::UpdatePosition (float *dt*)

Actualiza la posición del personaje.

La nueva posición se obtiene de integrar el vector de aceleración acon la dirección que lleva el personaje y luego el de velocidad.

Parámetros:

dt intervalo de tiempo transcurrido desde la última vez que se llamo al método UpdatePosition.

A.8. nistnpc Documentación de clases

Referencia de la Clase nlstNPC

Clase encargada de controlar los personajes del sistema de acuerdo con las acciones del usuario.

```
#include <nIstNPC.h>
```

Métodos públicos

nIstNPC ()
Método constructor de la clase.

virtual **~nIstNPC** ()
Método destructor de la clase.

virtual void **Initialize** ()
Inicializa varios valores de la clase.

virtual bool **SaveCmds** (nPersistServer *fileServer)
Método encargado de la persistencia, salva un archivo de comandos.

void **Trigger** (float dt)
Actualiza el estado del objeto.

void **Collide** (void)
Maneja el comportamiento del objeto cuando éste colisiona.

void **UpdatePosition** (float dt)
Actualiza la posición del personaje.

void **AddImpulse** (vector3 addv)
Usado en las colisiones para repeler el objeto en movimiento.

void **AnimWalk** ()
LLama a la animación "caminar" del NPC.

brief LLama a la animaci n girar del NPC *void **AnimStrut ()
brief LLama a la animaci n parado del NPC *void **AnimIdle ()
void **SetRotSpeed** (float rs)
Fija la velocidad de giro del NPC.

void **SetRunSpeed** (float rs)
fija la velocidad del NPC al avanzar.

float **GetRotSpeed** ()
Indica la velocidad de giro del NPC.

float **GetRunSpeed** ()
Indica la velocidad del NPC al avanzar.

void **SetN3DNodeTar** (const char *path)
Establece el nodo 3d que contiene el modelo del NPC.

const char * **GetN3DNodeTar** () const
Indica el nodo 3d que contiene el modelo del NPC.

void **SetNCalModTar** (const char *path)
Fija el modelo que representa al NPC.

const char * **GetNCalModTar** () const
Indica el modelo que representa al NPC.

void **SetCollServ** (const char *path)
Fija el servidor de colisiones.

const char * **GetCollServ** () const
Indica el servidor de colisiones.

void **EnableEventColl** (const char *collScriptFile)
Habilita la ejecución de scripts por medio de eventos.

void **DisableEventColl** ()
Deshabilita la ejecución de scripts por medio de eventos.

nIstEntity * **GetLastCollEntity** (void)
Regresa la última entidad con la que colisiono el objeto.

bool **SetStatePathSeeking** (const char *orig, const char *dest, double duration)
Coloca al NPC en el modo de seguir un camino pre establecido.

bool **IsPathSeeking** (void)
Pregunta si el NPC se encuentra siguiendo un camino.

Atributos públicos estáticos

nKernelServer * **kernelServer**

Métodos protegidos

bool **BeginPathSeeking** (const char *orig, const char *dest, double duration)
vector3 * **SeekPath** (float dt)

Atributos privados

nAutoRef< nGfxServer > **ref_gs**
nAutoRef< nInputServer > **ref_is**
nAutoRef< nConServer > **ref_con**
nAutoRef< nScriptServer > **ref_ss**
nAutoRef< nSceneGraph2 > **ref_sg**
nAutoRef< nCollideServer > **ref_collide**
nAutoRef< n3DNode > **ref_npcnode**
nAutoRef< nCal3DModel > **ref_npcmodel**
nTclScriptlet * **collScriptlet**
n3DNode * **npcnode**

nCal3DModel * **npcmodel**
float **run_speed**
float **rot_speed**
double **pathDur**
bool **seekingBack**
double **pathTime**
nPath * **actualPath**
bool **StatePtSk**
bool **EnEvColl**
nString **collScriptFile**
nIstEntity * **lastCollEntity**
nString **snpcnode**
nString **snpcmodel**

Descripción detallada

Clase encargada de controlar los personajes del sistema de acuerdo con las acciones del usuario.

Mediante la interacción que realiza el usuario con los personajes controlados pr el sistema esta clase es capaz de responder mediante scripts, también controla y actualiza la animación y posición del modelo que representa al NPC, además esta clase puede seguir caminos pre establecidos.

Documentación del constructor y destructor

nIstNPC::nIstNPC ()

Método constructor de la clase.

Este método construye nuevas instancias de la clase.

virtual nIstNPC::~nIstNPC () `[virtual]`

Método destructor de la clase.

Este método destruye instancias creadas de la clase.

Documentación de las funciones miembro

void nIstNPC::AddImpulse (vector3 *addv*) `[inline]`

Usado en las colisiones para repeler el objeto en movimiento.

Parámetros:

addv Vector que indica la dirección del impulso.

void nIstNPC::AnimWalk () `[inline]`

LLama a la animación "caminar" del NPC.

void nlstNPC::Collide (void)

Maneja el comportamiento del objeto cuando éste colisiona.

Para esta clase en particular este método llama a los scripts que contienen los comportamientos del NPC ante eventos que ocurran en el mundo virtual.

void nlstNPC::DisableEventColl () `[inline]`

Deshabilita la ejecución de scripts por medio de eventos.

void nlstNPC::EnableEventColl (const char * *collScriptFile*) `[inline]`

Habilita la ejecución de scripts por medio de eventos.

Parámetros:

collscriptfile nombre del archivo que contiene el script a ejecutar.

const char * nlstNPC::GetCollServ () const `[inline]`

Indica el servidor de colisiones.

Devuelve:

Ruta dentro de la jerarquía de Nebula donde se encuentra el servidor de colisiones.

nlstEntity * nlstNPC::GetLastCollEntity (void) `[inline]`

Regresa la última entidad con la que colisiono el objeto.

Devuelve:

El nombre del objeto con el que hubo colisión.

const char * nlstNPC::GetN3DNodeTar () const `[inline]`

Indica el nodo 3d que contiene el modelo del NPC.

Devuelve:

Ruta de la jeraquía de Nebula en la que se encuentra el nodo que contiene al modelo.

const char * nlstNPC::GetNCalModTar () const `[inline]`

Indica el modelo que representa al NPC.

Devuelve:

Ruta en la que se encuentra el archivo que contiene al modelo.

float nlstNPC::GetRotSpeed () `[inline]`

Indica la velocidad de giro del NPC.

Devuelve:

Velocidad de giro en grados/segundos.

float nlstNPC::GetRunSpeed () `[inline]`

Indica la velocidad del NPC al avanzar.

Devuelve:

Velocidad en m/s.

virtual void nlstNPC::Initialize () `[virtual]`

Inicializa varios valores de la clase.

Con la llamada a este método se crean y agregan los scripts para cada evento.

bool nlstNPC::IsPathSeeking (void)

Pregunta si el NPC se encuentra siguiendo un camino.

Devuelve:

1 si el NPC en ese momento esta siguiendo un camino preestablecido.

virtual bool nlstNPC::SaveCmds (nPersistServer * *fileServer*) `[virtual]`

Método encargado de la persistencia, salva un archivo de comandos.

void nlstNPC::SetCollServ (const char * *path*) `[inline]`

Fija el servidor de colisiones.

Parámetros:

path ruta dentro de la jerarquía de Nebula donde se encuentra el servidor de colisiones.

void nlstNPC::SetN3DNodeTar (const char * *path*) `[inline]`

Establece el nodo 3d que contiene el modelo del NPC.

Parámetros:

path ruta de la jeraquía de Nebula en la que se encuentra el nodo que contiene al modelo.

void nlstNPC::SetNCalModTar (const char * *path*) `[inline]`

Fija el modelo que representa al NPC.

Parámetros:

path ruta en la que se encuentra el archivo que contiene al modelo.

void nlstNPC::SetRotSpeed (float *rs*) `[inline]`

Fija la velocidad de giro del NPC.

Parámetros:

rs Velocidad de giro en grados/segundos.

void nlstNPC::SetRunSpeed (float *rs*) `[inline]`

fija la velocidad del NPC al avanzar.

Parámetros:

rs velocidad en m/s.

bool nlstNPC::SetStatePathSeeking (const char * *orig*, const char * *dest*, double *duration*)

Coloca al NPC en el modo de seguir un camino pre establecido.

Parámetros:

orig definición del punto de origen del camino, dest definición del punto de destino del camino, duration teimpo que debe tardar en NPC en recorrer el camino.

Devuelve:

Si fue posible encontrar un camino del origen al destino señalados.

void nlstNPC::Trigger (float *dt*)

Actualiza el estado del objeto.

En este método se actualizan las animaciones y movimientos del personaje. por ejemplo cunado el NPC sigue un camino.

Parámetros:

dt intervalo de tiempo transcurrido desde la última vez que se llamo al método Trigger.

void nlstNPC::UpdatePosition (float *dt*)

Actualiza la posición del personaje.

Parámetros:

dt intervalo de tiempo transcurrido desde la última vez que se llamo al método UpdatePosition.

Apéndice B

Código fuente de la aplicación Calakmul Virtual

Debido a la gran cantidad de espacio que requeriría mostrar todo el codigo se ha optado por solo presentar el código de las clases más importantes, nIst Camera, nIstCollZone, nIstStairs, nIstPlayer y nIstNPC. Recordando que las clases nIstObject y nIstStruct son muy similares a nIstCollZone e nIstStairs respectivamente.

B.1. nIstCamera.h

```
#ifndef N_ISTCAMERA_H
#define N_ISTCAMERA_H
//------------------------------------------------------------------------------

#include "ist/nistentity.h"

#ifndef N_ROOT_H
#include "kernel/nroot.h"
#endif

#ifndef N_AUTOREF_H
#include "kernel/nautoref.h"
#endif

#ifndef N_MATRIX_H
#include "mathlib/matrix.h"
#endif

#undef N_DEFINES
#define N_DEFINES nIstCamera
#include "kernel/ndefdllclass.h"

#include "ist/nistplayer.h"
#include "script/ntclscriptlet.h"

//------------------------------------------------------------------------------

class nInputEvent;
class nPath;
class nInputServer;


class N_PUBLIC nIstCamera : public nIstEntity
{
public:
  enum Style
  {
    STATIONARY,
    LOCKED_CHASE,
                CHASE,
                AIM_PATH,
                LOOK_PATH,
                FPS,
                FREE,
                PLANE
  };
```

```
public:
  // Constructor
  nIstCamera();

        virtual void Initialize();

  // Destructor
  virtual ~nIstCamera();
  // Persistency
  virtual bool SaveCmds(nPersistServer* fileServer);

  // Get camera transform
  const matrix44& GetTransform() const;

        //nIstEntity metods
        void Trigger(float dt);
        void Collide(void);

        //Used in collisions to repell the moving object
        void AddImpulse(vector3 addv);

  // Set camera position (only when stationary)
  void SetPosition(float x, float y, float z);

        // Set camera style
        void SetStyleStationary(vector3 targetPos,float angleX,float angleY,float angleZ,float dist);
        void SetStyleStationary(n3DNode *target,float angleX,float angleY,float angleZ,float dist);
        void SetStyleStationary(vector3 targetPos,vector3 cameraPos);
        void SetStyleStationary(n3DNode *target,n3DNode *camer);

        void SetStyleChase(n3DNode *target,float height, float prefDist);

        void SetStyleLockedChase(n3DNode *target,float angleX,float angleY,float angleZ,float dist);

        void SetStyleAimPath(n3DNode *target,const char *orig,const char *dest,float len);

        void SetStyleLookPath(const char *origPos,const char *destPos,const char *origAng,const char *destAng,float
len);

        void SetStyleFPS(n3DNode *target, nIstPlayer *player,float height);

        void SetStyleFree();

        void SetStylePlane(float dist, vector3 targetPos);

        void EnableSelector(bool s);

        void handleInput(nInputEvent *ie );

        quaternion getQ(void);
        vector3 getT(void);
        void Qxyzw(float x,float y,float z,float w);


        void EnableEventColl(const char *collScriptFile);
        void DisableEventColl();
        nIstEntity *GetLastCollEntity(void);

  /// pointer to nKernelServer
  static nKernelServer* kernelServer;
        static nClass *loacl_cl;

private:
        nAutoRef<nInputServer>    ref_is;
        nAutoRef<nGfxServer>              ref_gs;

        n3DNode *cameraNode;              //only used with CHASE

  matrix44 cameraMat;
        matrix44 targetMat;
```

```
        n3DNode *myTarget;

        nIstPlayer *myPlayer;

        nPath *actualPath;                          //used in AIM_PATH
        bool seekingBack;                   //used in path seek

        float pathLength;           //used in both paths in seconds
        float pathTime;

        nPath *actualAngPath;       //used in LOOK_PATH
        bool seekingAngBack;

        float fpsHeight;            //used in FPS
        float yLast;

        float chaseCamHeight;               //used in CHASE
        float chaseDistance;
        float chaseAngleRad;

        float planeDist;                            //Used in PLANE

    nIstCamera::Style cameraStyle;
        nIstCamera::Style lastCamStyle;

        bool cur_up;
        bool cur_down;
        bool cur_left;
        bool cur_right;
        bool ctrl;
        bool shift;
        bool lmb;
        bool mmb;
        bool rmb;
        bool normalize;
        long old_x;
        long old_y;
        long act_x;
        long act_y;
        int scrWidth;
        int scrHeight;
        bool xpbtn;
        bool xnbtn;
        bool ypbtn;
        bool ynbtn;
        float x_abs_ang;
        float xpang;
        float xnang;
        float ypang;
        float ynang;

        float distance; //used in LOCKED_CHASE

        matrix44 Rx,Ry;   // auxiliar rotation matrixes

        void handleViewer(float t);         //used in FREE
        void handleChaseViewer(void);               //used in CHASE
        void handleFPSViewer(float t);

        vector3 *SeekPath(float dt);                        //used in AIM_PATH and LOOK_PATH
        quaternion *SeekAngPath(float dt); //used in LOOK_PATH

        //Enable selection
        bool EnSelection;

        nTclScriptlet *collScriptlet;
        bool EnEvColl;
        nIstEntity *lastCollEntity;

};
```

```
//---------------------------------------------------------------

inline
void nlstCamera::EnableEventColl(const char *collScriptFile)
{

        this->EnEvColl=collScriptlet->ParseFile(collScriptFile);

}
inline
nlstEntity *nlstCamera::GetLastCollEntity(void)
{
        return this->lastCollEntity;
}

inline
void nlstCamera::DisableEventColl()
{
        this->EnEvColl=false;
}
//---------------------------------------------------------------
inline
void nlstCamera::EnableSelector(bool s)
{
        EnSelection=s;
}

inline
const matrix44& nlstCamera::GetTransform() const
{
   return cameraMat;
}

inline
void nlstCamera::SetStyleFree()   //done
{
        this->lastCamStyle=this->cameraStyle;
        if(lastCamStyle == nlstCamera::FPS) myTarget->SetActive(true);
        this->cameraStyle=nlstCamera::FREE;
}

inline
quaternion nlstCamera::getQ(void)
{
        return cameraMat.get_quaternion();
}

inline
vector3 nlstCamera::getT(void)
{
        return cameraMat.pos_component();
}

inline
void nlstCamera::Qxyzw(float x,float y,float z,float w)
{
        quaternion qa= quaternion(x,y,z,w);

        vector3 pos=cameraMat.pos_component();

        cameraMat.ident();
        cameraMat.mult_simple(matrix44(qa));
        cameraMat.translate(pos);

}
//---------------------------------------------------------------------------

#endif
```

B.2. nIstCamera_main.cc

```
#define N_IMPLEMENTS nIstCamera
//------------------------------------------------------------------------------
//  (C) 2002        ling
//------------------------------------------------------------------------------
#include "node/n3dnode.h"
#include "input/ninputserver.h"
#include "ist/nistcamera.h"
#include "path/npath.h"
#include "collide/ncollideobject.h"


#include <string>

nNebulaScriptClass(nIstCamera, "nroot");


//hack
vector3 upVector(0,2,0);



//------------------------------------------------------------------------------
/**
*/
nIstCamera::nIstCamera() :
    cameraMat(),
        targetMat(),
        myTarget(),
        cameraStyle(nIstCamera::FREE),
        lastCamStyle(nIstCamera::FREE),
        ref_is(kernelServer, this),
        ref_gs(kernelServer,this),
        seekingBack(false),
        seekingAngBack(false),
        actualPath(NULL),
        actualAngPath(NULL),
        fpsHeight(50),
        EnSelection(true),
        EnEvColl(false)
{

    cameraMat.ident();
        targetMat.ident();

        //Used in Free Style for input handling
        cur_up    = false;
        cur_down  = false;
        cur_left  = false;
        cur_right = false;
        ctrl      = false;
        shift     = false;
        lmb = false;
        mmb = false;
        rmb = false;
        normalize = false;

        xpbtn = false;
        xnbtn = false;
        ypbtn = false;
        ynbtn = false;

        x_abs_ang=0;
```

```
        xpang=0.0000f;
        xnang=0.0000f;
        ypang=0.0000f;
        ynang=0.0000f;

        pathLength=0;
        pathTime=0;

        this->ref_gs    = "/sys/servers/gfx";
    this->ref_is    = "/sys/servers/input";

        act_x = 0;
        act_y = 0;

}

//------------------------------------------------------------------------------
/**
*/
nIstCamera::~nIstCamera()
{
        collScriptlet->~nTclScriptlet();
}

//------------------------------------------------------------------------------

//Note: here is where we create and add the scriptlets for EVERY action
void nIstCamera::Initialize()
{
        nIstEntity::Initialize();

        std::string scriptPath="/sys/share/scriptlets/";

        scriptPath+=GetName();
        scriptPath+="/";
        this->collScriptlet= (nTclScriptlet *) kernelServer->New("ntclscriptlet",( scriptPath+"collscriptlet").c_str() );

}
//------------------------------------------------------------------------------

void nIstCamera::Collide(void)
{
        if(this->EnSelection)
        {
                if (collideObject == NULL)
                        return;

                nCollideReport **report;
                int num_colls = collideObject->GetCollissions(report);
                if (num_colls == 0 ) //If there arent' any collitions then exit
                        return;

                for (int i = 0; i < num_colls; ++i)
                {
                        nIstEntity* other = (nIstEntity*)report[i]->co2->GetClientData();


                        if (this == other)
                                other = (nIstEntity*)report[i]->co1->GetClientData();

                        if (other == NULL)
                                continue;

                        other->Select();

//------------------------------------------------------------------------------
                        char buffy[30];
                        lastCollEntity=other;
                        strcpy(buffy,other->GetClass()->GetName() );
```

```
                        if( !strcmp( buffy,"nistplayer") || !strcmp( buffy,"nistobject") || !strcmp( buffy,"niststruct") ||
!strcmp( buffy,"nistnpc"))
                        {

                                if(EnEvColl)
                                {

                                        collScriptlet->Run();

                                }

                        }
//-------------------------------------------------------------------------

                } //end for collisions
        } //end if Enabled Selection
}


void nlstCamera::AddImpulse(vector3 addv)
{
        //it doesn't do nothing right now
}


void nlstCamera::SetPosition(float x, float y, float z)
{
        if(cameraStyle == nlstCamera::STATIONARY || cameraStyle == nlstCamera::FREE )
                cameraMat.translate(x,y,z);
}

/**
  Update the camera, move it around and stuff.
*/
void nlstCamera::Trigger(float dt)
{
        matrix44 m=cameraMat;


        if(cameraStyle==nlstCamera::PLANE)
        {
                int b1,b2;
                ref_gs->GetDisplayDesc(b1,b2,scrWidth,scrHeight);

                vector3 v( 0.0001f*planeDist*(float)((scrHeight/2)-act_y ), 0.003f*planeDist*(float) ( (scrHeight/2)-act_y ),
0.002f*planeDist*float(act_x-scrWidth/2)  );
                m.translate(v);
        }
        collideObject->Transform(0.0F,m);  //selection Mesh Position

  switch (cameraStyle)
  {

    case nlstCamera::STATIONARY:
      // Do nothing
     break;

    // Locked camera at a position relative to entity direction
    case nlstCamera::LOCKED_CHASE:
     {

                                myTarget->GetT(cameraMat.M41,cameraMat.M42,cameraMat.M43);

                                cameraMat.M41 += cameraMat.M31 * distance;
                                cameraMat.M42 += cameraMat.M32 * distance;
                                cameraMat.M43 += cameraMat.M33 * distance;


     }
```

```
	break;

	case nlstCamera::CHASE:
	{
				if(myTarget)
				{
					handleChaseViewer();
					cameraMat=myTarget->GetM();

					vector3 pos =cameraMat.pos_component();

					vector3 camPos =
vector3(cameraMat.M31*distance,cameraMat.M32*distance,chaseCamHeight);

					cameraMat.M41=0.0F; cameraMat.M42=0.0F; cameraMat.M43=0.0F;
					cameraMat.rotate_x(n_deg2rad(90));
					cameraMat.rotate_z(n_deg2rad(180));

					quaternion quat=cameraMat.get_quaternion();
					quaternion qSwp=quaternion(quat.x,-quat.z,quat.y,quat.w);

					cameraMat.ident();
					cameraMat.translate(camPos);
					cameraMat.mult_simple(qSwp);

					pos.y+=fpsHeight;
					cameraMat.translate(pos);

					cameraMat.lookat(pos,vector3(0,fpsHeight,0));
				}


	}
	break;

			case nlstCamera::AIM_PATH:
	{
				if(pathTime<=pathLength )
				{

					cameraMat.ident();
					vector3 *newPos=SeekPath(dt);

					cameraMat.translate(*newPos);

					vector3 tarPos;
					myTarget->GetT(tarPos.x,tarPos.y,tarPos.z);
					cameraMat.lookat(tarPos,upVector);

				}
				else
				{

					this->lastCamStyle=this->cameraStyle;
					this->cameraStyle=nlstCamera::FREE;
				}

	}
	break;

			case nlstCamera::LOOK_PATH:
	{
				if(pathTime<=pathLength )
				{

					vector3 *pos=SeekPath(dt);
					quaternion *qt=SeekAngPath(dt);
```

```
                                this->cameraMat.ident();
                                this->cameraMat.mult_simple(matrix44(*qt));
                                this->cameraMat.translate(*pos);

                        }
                        else
                        {
                                this->lastCamStyle=this->cameraStyle;
                                this->cameraStyle=nIstCamera::FREE;
                        }

        }
        break;

                case nIstCamera::FPS:
        {
                        if(myTarget)
                        {
                                this->handleFPSViewer(dt);

                                vector3 pos =cameraMat.pos_component();

                                cameraMat.rotate_y(n_deg2rad(180));

                                quaternion quat=cameraMat.get_quaternion();
                                quaternion qSwp=quaternion(quat.x,quat.y,quat.z,quat.w);

                                cameraMat.ident();
                                cameraMat.mult_simple(qSwp);

                                cameraMat.translate(pos);

                        }

                }
        break;

                case nIstCamera::FREE:
        {
                        this->handleViewer(dt);
        }
        break;


    }

}

//------------------------------------------------------------------
void nIstCamera::handleInput(nInputEvent *ie )
{
        if(this->cameraStyle != nIstCamera::FREE  && this->cameraStyle != nIstCamera::CHASE && this->cameraStyle !=
nIstCamera::FPS
                                        && this->cameraStyle != nIstCamera::PLANE)
                return;

    if (ie->IsDisabled()) return;

    switch (ie->GetType()) {
        case N_INPUT_KEY_DOWN:
            {
                switch (ie->GetKey()) {
                    case N_KEY_CONTROL: ctrl=true; break;
                    case N_KEY_SHIFT:   shift=true; break;
                    case N_KEY_SPACE:   normalize=true; break;
                    default: break;
                }
            }
```

```
    break;

case N_INPUT_KEY_UP:
    {
        switch (ie->GetKey()) {
            case N_KEY_CONTROL: ctrl=false; break;
            case N_KEY_SHIFT:   shift=false; break;
            default: break;
        }
    }
    break;

case N_INPUT_BUTTON_DOWN:
    {
        switch (ie->GetButton()) {
            case 0:
                                                {
                                                        if(ie->GetDeviceId()==N_IDEV_RELMOUSE)
                                                                lmb=true;
                                                }
                                                break;
            case 1:
                                                {
                                                        if(ie->GetDeviceId()==N_IDEV_RELMOUSE)
                                                                rmb=true;
                                                }
                                                break;
            case 2:
                                                {
                                                        if(ie->GetDeviceId()==N_IDEV_RELMOUSE)
                                                                mmb=true;
                                                }
                                                break;
                                                case 3:
                                                {
                                                        xpbtn=true;
                                                        if(cameraStyle==FPS)
                                                                myPlayer->StartMRR();
                                                }
                                                break;
                                                case 4:
                                                {
                                                        if(ie->GetDeviceId()==N_IDEV_RELMOUSE)
                                                        {
                                                                xnbtn=true;
                                                                if(cameraStyle==FPS)
                                                                        myPlayer->StartMRL();
                                                        }
                                                }
                                                break;
                                                case 5:
                                                {
                                                        if(ie->GetDeviceId()==N_IDEV_RELMOUSE)
                                                                ypbtn=true;
                                                }
                                                break;
                                                case 6:
                                                {
                                                        if(ie->GetDeviceId()==N_IDEV_RELMOUSE)
                                                                ynbtn=true;
                                                }
                                                break;

            default: break;
        }
    }
    break;
case N_INPUT_BUTTON_UP:
    {
```

```
        switch (ie->GetButton()) {
          case 0:
                                        {
                                                if(ie->GetDeviceId()==N_IDEV_RELMOUSE)
                                                        lmb=false;
                                        }
                                        break;
          case 1:
                                        {
                                                if(ie->GetDeviceId()==N_IDEV_RELMOUSE)
                                                        rmb=false;
                                        }
                                        break;
          case 2:  mmb=false; break;
                                        case 3:
                                        {
                                                if(ie->GetDeviceId()==N_IDEV_RELMOUSE)
                                                {
                                                        xpbtn=false;
                                                        if(cameraStyle==FPS)
                                                                myPlayer->StopMRR();
                                                }
                                        }
                                        break;
                                        case 4:
                                        {
                                                if(ie->GetDeviceId()==N_IDEV_RELMOUSE)
                                                {
                                                        xnbtn=false;
                                                        if(cameraStyle==FPS)
                                                                myPlayer->StopMRL();
                                                }
                                        }
                                        break;
                                        case 5:
                                        {
                                                if(ie->GetDeviceId()==N_IDEV_RELMOUSE)
                                                        ypbtn=false;
                                        }
                                        break;
                                        case 6:
                                        {
                                                if(ie->GetDeviceId()==N_IDEV_RELMOUSE)
                                                        ynbtn=false;
                                        }
                                        break;
          default: break;
        }
      }
      break;
                case N_INPUT_MOUSE_MOVE:
                        {
                                if(ie->GetDeviceId()==N_IDEV_MOUSE)
                                {
                                        old_x=act_x;
                                        old_y=act_y;

                                        act_x = ie->GetAbsXPos();
                                        act_y = ie->GetAbsYPos();

                                }
                        }
                        break;
    default: break;
  }

}
```

```
//-------------------------------------------------------------------
void nIstCamera::handleViewer(float t)
{
   matrix44 tm;
   matrix44 tmp;
   float tx,ty,tz;
   float rx,ry;
   tx=0.0f; ty=0.0f; tz=0.0f;
   rx=0.0f; ry=0.0f;


   if (shift) {
      if (cur_up)    rx -= 0.5f;
      if (cur_down)  rx += 0.5f;
      if (cur_left)  ry -= 0.5f;
      if (cur_right) ry += 0.5f;
   } else {
      if (ctrl) {
         if (cur_up)    ty -= 6.0f;
         if (cur_down)  ty += 6.0f;
      } else {
         if (cur_up)    tz -= 6.0f;
         if (cur_down)  tz += 6.0f;
         if (cur_left)  tx -= 6.0f;
         if (cur_right) tx += 6.0f;
      }
   }
   if (rmb) {
 //     ry += ((float)(old_x-act_x)) * 0.1f;
 //     rx += ((float)(old_y-act_y)) * 0.1f;

                        if(xpbtn)
                                xpang=1.0f;
                        else
                                xpang=0.000f;

                        if(xnbtn)
                                xnang=-1.0f;
                        else
                                xnang=0.000f;

                        if(ypbtn)
                                ypang=1.0f;
                        else
                                ypang=0.000f;

                        if(ynbtn)
                                ynang=-1.0f;
                        else
                                ynang=0.000f;

                                ry = (xpang + xnang) * 2.5*t;
                                rx = (ypang + ynang) * 2.5*t;

   }//end Secundary mouse button FREE STYLE

   if (lmb) {
                        if(ypbtn)
                                ypang=180.00f;
                        else
                                ypang=0.000f;

                        if(ynbtn)
                                ynang=-180.00f;
                        else
                                ynang=0.000f;

                                tz = (ypang + ynang) * 6*t;

   } //end Main mouse button FREE STYLE
```

```
    if (normalize) {
       Rx.ident();
       Ry.ident();
       cameraMat.ident();
       vector3 t(0.0f,2.5f,0.0f);
       cameraMat.translate(t);
       normalize = false;
    }

    Rx.rotate_x(rx);
    Ry.rotate_y(ry);

    // Translation auf existierende Matrix
    tm.ident();
    tm.translate(tx,ty,tz);
    tm.mult_simple(cameraMat);


    vector3 vTrans(tm.M41,tm.M42,tm.M43);

    cameraMat.ident();
    cameraMat.translate(vTrans);

    tmp = Ry;
    tmp.mult_simple(cameraMat);
    cameraMat = tmp;
    tmp = Rx;
    tmp.mult_simple(cameraMat);
    cameraMat = tmp;

//  old_x = act_x;
//  old_y = act_y;
}

//-------------------------------------------------------------------------------------------

void nIstCamera::handleFPSViewer(float t)
{
    matrix44 tmp, playerMat;
        float rx,ry;

//      ry = ((float)(old_x-act_x)) * 0.35f;
//      rx = ((float)(old_y-act_y)) * 0.10f;

        if(xpbtn)
            xpang=48.000f;
        else
            xpang=0.000f;

        if(xnbtn)
            xnang=-48.0000f;
        else
            xnang=0.000f;


        if(ypbtn)
            ypang=0.84f;
        else
            ypang=0.000f;

        if(ynbtn)
            ynang=-0.84f;
        else
            ynang=0.000f;

        ry = (xpang + xnang)*2*t;
        rx = (ypang + ynang)*2*t;
```

```
            x_abs_ang+=rx;
            if(x_abs_ang >= 1.500f)
            {
                        rx = 0.000f;
                        x_abs_ang=1.5000f;

            }
            if(x_abs_ang <= -1.000f)
            {
                        rx = 0.000F;
                        x_abs_ang=-1.000f;
            }

            vector3 r;
            myTarget->GetR(r.x,r.y,r.z);

            r.y += ry;

            myTarget->Rxyz(r.x,r.y,r.z);

            vector3 posp =myTarget->GetM().pos_component();
            cameraMat.ident();
            posp.y+=fpsHeight;
            cameraMat.translate(posp.x,posp.y,posp.z);

            Ry.rotate_y(n_deg2rad(r.y-yLast));

            tmp = Ry;
      tmp.mult_simple(cameraMat);
      cameraMat = tmp;

            Rx.rotate_x(rx);
            tmp = Rx;
      tmp.mult_simple(cameraMat);
      cameraMat = tmp;

//  old_x = act_x;
//   old_y = act_y;
            yLast=r.y;

}

//-------------------------------------------------------------------------------------------
void nIstCamera::handleChaseViewer(void)
{
            if (rmb) {
  //    chaseAngleRad += ((float)(old_y-act_y)) * 0.05f;

                        if(ypbtn)
                                    ypang=0.100f;
                        else
                                    ypang=0.000f;

                        if(ynbtn)
                                    ynang=-0.100f;
                        else
                                    ynang=0.000f;

                        chaseAngleRad += (ypang * 0.1f) + (ynang * 0.1f);

                        if(chaseAngleRad > 100*PI/180) chaseAngleRad = 100*PI/180;
                        if(chaseAngleRad < 0) chaseAngleRad = 0.0001F;

                        distance=n_cos(chaseAngleRad)*chaseDistance;
                        chaseCamHeight=n_sin(chaseAngleRad)*chaseDistance;
      }

}
```

```
// Set Camera Style
//Plane
void nIstCamera::SetStylePlane(float dist, vector3 targetPos)  //done
{
        this->lastCamStyle=this->cameraStyle;
        if(lastCamStyle == nIstCamera::FPS) myTarget->SetActive(true);
        this->cameraStyle=nIstCamera::PLANE;

        cameraMat.ident();
        cameraMat.rotate_z(0.0);
        cameraMat.rotate_x(n_deg2rad(-45.0));
        cameraMat.rotate_y(n_deg2rad(270));


        cameraMat.translate(targetPos);


        cameraMat.M41 += cameraMat.M31 * dist;
  cameraMat.M42 += cameraMat.M32 * dist;
  cameraMat.M43 += cameraMat.M33 * dist;

        planeDist=dist;
}


//stationary
void nIstCamera::SetStyleStationary(vector3 targetPos,float angleX,float angleY,float angleZ,float dist) //done
{
        this->lastCamStyle=this->cameraStyle;
        if(lastCamStyle == nIstCamera::FPS) myTarget->SetActive(true);
        this->cameraStyle=nIstCamera::STATIONARY;

        cameraMat.ident();
        cameraMat.rotate_z(n_deg2rad(angleZ));
        cameraMat.rotate_x(n_deg2rad(-angleX));
        cameraMat.rotate_y(n_deg2rad(angleY));


        cameraMat.translate(targetPos);


        cameraMat.M41 += cameraMat.M31 * dist;
  cameraMat.M42 += cameraMat.M32 * dist;
  cameraMat.M43 += cameraMat.M33 * dist;
}

void nIstCamera::SetStyleStationary(n3DNode *target,float angleX,float angleY,float angleZ,float dist)        //done
{
        vector3 pos;
        target->GetT(pos.x,pos.y,pos.z);
        this->SetStyleStationary(pos,angleX,angleY,angleZ,dist);
}

void nIstCamera::SetStyleStationary(vector3 targetPos,vector3 cameraPos)              //done, I think so
{
        this->lastCamStyle=this->cameraStyle;
        if(lastCamStyle == nIstCamera::FPS) myTarget->SetActive(true);
        this->cameraStyle=nIstCamera::STATIONARY;
        cameraMat.ident();
        cameraMat.translate(cameraPos);
        cameraMat.lookat(targetPos,upVector);
}

void nIstCamera::SetStyleStationary(n3DNode *target,n3DNode *camer)               //done, I think so
{
        this->lastCamStyle=this->cameraStyle;
        if(lastCamStyle == nIstCamera::FPS) myTarget->SetActive(true);
        this->cameraStyle=nIstCamera::STATIONARY;
        cameraMat.ident();
```

```
        vector3 camPos, tarPos;

        camer->GetT(camPos.x,camPos.y,camPos.z);
        cameraMat.translate(camPos);

        target->GetT(tarPos.x,tarPos.y,tarPos.z);
        cameraMat.lookat(tarPos,upVector);
}

//chase

void nlstCamera::SetStyleChase(n3DNode *target,float height,float prefDist)
{
        this->lastCamStyle=this->cameraStyle;
        if(lastCamStyle == nlstCamera::FPS) myTarget->SetActive(true);
        this->cameraStyle=nlstCamera::CHASE;

        myTarget=target;

        chaseAngleRad=PI/4;
        distance=n_cos(chaseAngleRad)*prefDist;
        chaseCamHeight=n_sin(chaseAngleRad)*prefDist;
        fpsHeight=height;

        chaseDistance=prefDist;

}

//locked chase
void nlstCamera::SetStyleLockedChase(n3DNode *target,float angleX,float angleY,float angleZ,float dist)  //done
{
        this->lastCamStyle=this->cameraStyle;
        if(lastCamStyle == nlstCamera::FPS) myTarget->SetActive(true);
        this->cameraStyle=nlstCamera::LOCKED_CHASE;
        myTarget=target;

        vector3 pos;
        myTarget->GetT(pos.x,pos.y,pos.z);

        cameraMat.ident();
        cameraMat.rotate_z(n_deg2rad(angleZ));
        cameraMat.rotate_x(n_deg2rad(-angleX));
        cameraMat.rotate_y(n_deg2rad(angleY));

        cameraMat.translate(pos);

        cameraMat.M41 += cameraMat.M31 * dist;
  cameraMat.M42 += cameraMat.M32 * dist;
  cameraMat.M43 += cameraMat.M33 * dist;
        distance=dist;

}

//Aim Path
void nlstCamera::SetStyleAimPath(n3DNode *target,const char *orig,const char *dest,float len)      //todo
{
        this->lastCamStyle=this->cameraStyle;
        if(lastCamStyle == nlstCamera::FPS) myTarget->SetActive(true);

        std::string pth="/sys/share/paths/";
        std::string or=orig;
        std::string de=dest;

        actualPath= (nPath *) kernelServer->Lookup( (pth+or+"_"+de).c_str() );

        if(!actualPath)
        {
                actualPath= (nPath *) kernelServer->Lookup((pth+de+"_"+or).c_str());
                if(!actualPath)
```

```
                return;

        seekingBack=true;
    }
    else
        seekingBack=false;

    if(target)
        myTarget=target;
    else
        return;


    this->cameraStyle=nIstCamera::AIM_PATH;
    pathLength=len;
    pathTime=0.0F;

    cameraMat.ident();

    vector3 *cameraPos;
    if(!seekingBack)
        cameraPos=actualPath->PointAt(0.0F);
    else
        cameraPos=actualPath->PointAt(1.0F);

    cameraMat.translate(*cameraPos);

    vector3 tarPos;
    myTarget->GetT(tarPos.x,tarPos.y,tarPos.z);
    cameraMat.lookat(tarPos,upVector);
}

//Look path
void nIstCamera::SetStyleLookPath(const char *origPos,const char *destPos,const char *origAng,const char *destAng,float
len)
{
    std::string pth="/sys/share/paths/";
    std::string or=origPos;
    std::string de=destPos;


    actualPath= (nPath *) kernelServer->Lookup( (pth+or+"_"+de).c_str() );

    if(!actualPath)
    {
        actualPath= (nPath *) kernelServer->Lookup((pth+de+"_"+or).c_str());
        if(!actualPath)
            return;

        seekingBack=true;
    }
    else
        seekingBack=false;

    or=origAng;
    de=destAng;

    actualAngPath= (nPath *) kernelServer->Lookup( (pth+or+"_"+de).c_str() );

    if(!actualAngPath)
    {
        actualAngPath= (nPath *) kernelServer->Lookup((pth+de+"_"+or).c_str());     //this path needs the
information in the w coordintate
        if(!actualAngPath)
            return;

        seekingAngBack=true;
    }
    else
        seekingAngBack=false;
```

```
		//Setting the new camera style and saving the last
		this->lastCamStyle=this->cameraStyle;
		if(lastCamStyle == nlstCamera::FPS) myTarget->SetActive(true);
		this->cameraStyle=nlstCamera::LOOK_PATH;
		pathLength=len;
		pathTime=0.0F;
		//Getting the initial t and q from the paths
		vector3 *cameraPos;
		if(!seekingBack)
			cameraPos=actualPath->PointAt(0.0F);
		else
			cameraPos=actualPath->PointAt(1.0F);

		quaternion *cameraQ;
		if(!seekingAngBack)
			cameraQ=actualAngPath->HPointAt(0.0F);
		else
			cameraQ=actualAngPath->HPointAt(1.0F);
		//setting the new cameraMat
		this->cameraMat.ident();
		this->cameraMat.mult_simple(matrix44(*cameraQ));
		this->cameraMat.translate(*cameraPos);
}

void nlstCamera::SetStyleFPS(n3DNode *target,nlstPlayer *player,float height)
{
		if(target)
		{
			this->lastCamStyle=this->cameraStyle;
			if(lastCamStyle == nlstCamera::FPS) myTarget->SetActive(true);
			this->cameraStyle=nlstCamera::FPS;
			myTarget=target;
			myTarget->SetActive(false);
			float xTrash, zTrash;
			myTarget->GetR(xTrash,yLast,zTrash);
			fpsHeight=height;
			myPlayer=player;

			x_abs_ang=0;
			Ry.ident();
			Ry.rotate_y(n_deg2rad(yLast));
			Rx.ident();
		}
}

vector3 *nlstCamera::SeekPath(float dt)
{
		if(cameraStyle==nlstCamera::AIM_PATH || cameraStyle==nlstCamera::LOOK_PATH)
		{
			float u;
			pathTime+=dt;

			if(!seekingBack)
			{
				if( (u= float( pathTime/pathLength) ) <= 1.0F )
					return actualPath->PointAt( u );
				else
					return actualPath->PointAt( 1.0F );
			}
			else
			{
				if( (u= float( (pathLength-pathTime)/pathLength) ) >= 0.0F )
					return actualPath->PointAt( u );
				else
					return actualPath->PointAt( 0.0F );
			}
		}
		else
			return NULL;
}
```

```
quaternion *nIstCamera::SeekAngPath(float dt)
{
	if( cameraStyle==nIstCamera::LOOK_PATH)
	{
		float u;

		if(!seekingAngBack)
		{
			if(  (u= float( pathTime/pathLength) ) <= 1.0F )
				return actualAngPath->HPointAt( u );
			else
				return actualAngPath->HPointAt( 1.0F );
		}
		else
		{
			if(  (u= float( (pathLength-pathTime)/pathLength) ) >= 0.0F )
				return actualAngPath->HPointAt( u );
			else
				return actualAngPath->HPointAt( 0.0F );

		}

	}
	else
		return NULL;
}
```

B.3. nIstCollZone.h

```
#ifndef N_ISTCOLLZONE_H
#define N_ISTCOLLZONE_H
//------------------------------------------------------------------------------
/**
   @class nIstCollZone

   @brief a brief description of the class

   a detailed description of the class

*/
#ifndef N_ROOT_H
#include "kernel/nroot.h"
#endif

#ifndef N_AUTOREF_H
#include "kernel/nautoref.h"
#endif

#ifndef N_MATRIX_H
#include "mathlib/matrix.h"
#endif

#ifndef N_STRING_H
#include "util/nstring.h"
#endif


#ifndef N_ISTENTITY_H
#include "ist/nistentity.h"
#endif

#include "kernel/nroot.h"
#include "kernel/nkernelserver.h"
#include "kernel/ntimeserver.h"
#include "node/nmeshnode.h"
```

```
#include "node/ntexarraynode.h"
#include "node/nshadernode.h"
#include "gfx/ngfxserver.h"
#include "gfx/nscenegraph2.h"
#include "input/ninputserver.h"
#include "misc/nconserver.h"
#include "misc/nparticleserver.h"
#include "kernel/nscriptserver.h"
#include "misc/nspecialfxserver.h"
#include "collide/ncollideserver.h"
#include "script/ntclscriptlet.h"

#include "util/nstring.h"

#undef N_DEFINES
#define N_DEFINES nIstObject
#include "kernel/ndefdllclass.h"

//------------------------------------------------------------------------------

class N_PUBLIC nIstCollZone : public nIstEntity
{

public:
   /// constructor
   nIstCollZone();
   /// destructor
   virtual ~nIstCollZone();

         virtual void Initialize();

   /// persistency
   virtual bool SaveCmds(nPersistServer* fileServer);

   // Called in world loop
   void Collide();
   void Trigger(float dt);
         void CreateZone(const char *meshfile,float x,float y,float z);
         void EnableEventColl(const char *collScriptFile);
         void DisableEventColl();
         nIstEntity *GetLastCollEntity(void);

   /// pointer to nKernelServer
   static nKernelServer* kernelServer;

private:
         nAutoRef<nGfxServer>                  ref_gs;
         nAutoRef<nInputServer>     ref_is;
         nAutoRef<nConServer>       ref_con;
         nAutoRef<nScriptServer>    ref_ss;
         nAutoRef<nSceneGraph2>       ref_sg;
         nAutoRef<nCollideServer>    ref_collide;
         nAutoRef<n3DNode>                                 ref_objectnode;

         n3DNode *objectnode;
         nMeshNode *mesh;

         nString meshF;

         bool EnEvColl;
         nString collScriptFile;
         nIstEntity *lastCollEntity;

};
```

```
//------------------------------------------------------------
inline
void nIstCollZone::EnableEventColl(const char *collScriptFile)
{

//          this->EnEvColl=collScriptlet->ParseFile(collScriptFile);
            this->EnEvColl=true;
            this->collScriptFile=collScriptFile;
}

inline
nIstEntity *nIstCollZone::GetLastCollEntity(void)
{
            return this->lastCollEntity;
}

inline
void nIstCollZone::DisableEventColl()
{
            this->EnEvColl=false;
}

//------------------------------------------------------------

#endif
```

B.4. nIstCollZone_main.cc

```
#define N_IMPLEMENTS nIstCollZone
//------------------------------------------------------------------------------
//  Jose Larios Delgado
//------------------------------------------------------------------------------

//  includes

#include "ist/nIstCollZone.h"
#include "ist/nIstPlayer.h"
#include "ist/nIstEntity.h"
#include "ist/nistworld.h"
#include "kernel/nenv.h"
#include "collide/ncollideserver.h"
#include "collide/ncollideobject.h"
#include "kernel/ntimeserver.h"
#include "node/n3dnode.h"
#include "node/nshadernode.h"
#include <string.h>

class n3DNode;
class nChannelServer;
class nCollideObject;
class nCollideShape;
class nIstWorld;

nNebulaScriptClass(nIstCollZone, "nroot");

//------------------------------------------------------------------------------

nIstCollZone::nIstCollZone():
            ref_gs(kernelServer,this),
//          ref_gui(kernelServer,this),
        ref_is(kernelServer,this),
        ref_ss(kernelServer,this),
        ref_sg(kernelServer,this),
        ref_con(kernelServer,this),
        ref_collide(kernelServer,this),
            EnEvColl(false),
            meshF()
{
```

```
    this->ref_gs    = "/sys/servers/gfx";
    this->ref_is    = "/sys/servers/input";
    this->ref_ss    = "/sys/servers/script";
    this->ref_sg    = "/sys/servers/sgraph2";
    this->ref_con   = "/sys/servers/console";
    this->ref_collide = "/sys/servers/collide";


}
//------------------------------------------------------------------------------
/**
*/
nIstCollZone::~nIstCollZone()
{

}

//------------------------------------------------------------------------------

//Note: here is where we create and add the scriptlets for EVERY action
void nIstCollZone::Initialize()
{
        nIstEntity::Initialize();

//      std::string scriptPath="/sys/share/scriptlets/";

//      scriptPath+=GetName();
//      scriptPath+="/";
//      this->collScriptlet= (nTclScriptlet *) kernelServer->New("ntclscriptlet",( scriptPath+"collscriptlet").c_str() );

}

//------------------------------------------------------------------------------

void nIstCollZone::CreateZone(const char *meshfile,float x,float y,float z)
{
        char auxpath[30];
        int aux;

        n_assert("/usr/scene");

        strcpy(auxpath,"/usr/scene/");

        strcat(auxpath,nRoot::GetName() );
        aux=strlen(auxpath);

        this->objectnode    = (n3DNode *)               kernelServer->New("n3dnode",auxpath );

        objectnode->Txyz(x,y,z);
        objectnode->Rx(-90);
        objectnode->SetActive(false);

        strcat(auxpath,"/mesh");
        this->mesh                  = (nMeshNode *)         kernelServer->New("nmeshnode",auxpath);

        mesh->SetFilename(meshfile);
        meshF=meshfile;

        nIstEntity::SetCollideClass("nistcollobject");
        nIstEntity::SetCollideShape(nRoot::GetName(),meshfile);
}

void nIstCollZone::Trigger(float dt)
{
        collideObject->Transform(0.0F,objectnode->GetM() );


}
```

```
void nIstCollZone::Collide()
{
        if (collideObject == NULL)
    return;

  nCollideReport **report;
  int num_colls = collideObject->GetCollissions(report);
  if (num_colls == 0  )
    return;

  for (int i = 0; i < num_colls; ++i)
  {
    nIstEntity* other = (nIstEntity*)report[i]->co2->GetClientData();

    if (this == other)
    {
      other = (nIstEntity*)report[i]->co1->GetClientData();
    }

    if (other == NULL)
      continue;
            char buffy[30];
            lastCollEntity=other;
            strcpy(buffy,other->GetClass()->GetName() );

            if( !strcmp( buffy,"nistplayer") || !strcmp( buffy,"nistobject") || !strcmp( buffy,"niststruct") || !strcmp(
buffy,"nistnpc"))
            {

                  if(EnEvColl)
                  {

                  //      collScriptlet->Run();
                        char *resul;
                        ref_ss->RunScript(collScriptFile.Get(),resul);
                  }
            }
      }
}
```

B.5. nIstStairs.h

```
#ifndef N_ISTSTAIRS_H
#define N_ISTSTAIRS_H
//------------------------------------------------------------------------------
/**
  @class nIstStairs

  @brief a brief description of the class

  a detailed description of the class

*/
#ifndef N_ROOT_H
#include "kernel/nroot.h"
#endif

#ifndef N_AUTOREF_H
#include "kernel/nautoref.h"
#endif

#ifndef N_MATRIX_H
#include "mathlib/matrix.h"
#endif

#ifndef N_STRING_H
```

```
#include "util/nstring.h"
#endif

#ifndef N_ISTENTITY_H
#include "ist/nistentity.h"
#endif

#include "kernel/nroot.h"
#include "kernel/nkernelserver.h"
#include "kernel/ntimeserver.h"
#include "gfx/ngfxserver.h"
#include "gfx/nscenegraph2.h"
#include "kernel/nscriptserver.h"
#include "misc/nspecialfxserver.h"
#include "collide/ncollideserver.h"

#undef N_DEFINES
#define N_DEFINES nIstStairs
#include "kernel/ndefdllclass.h"
//------------------------------------------------------------------------------

class N_PUBLIC nIstStairs : public nIstEntity
{

public:
   /// constructor
   nIstStairs();
   /// destructor
   virtual ~nIstStairs();

   /// persistency
   virtual bool SaveCmds(nPersistServer* fileServer);

   // Called in world loop
   void Collide();
   void Trigger(float dt);

         //Used in collisions to repell the moving object
         void AddImpulse(vector3 addv);

   /// pointer to nKernelServer
   static nKernelServer* kernelServer;

private:
         nAutoRef<nGfxServer>                ref_gs;
   nAutoRef<nScriptServer>    ref_ss;
   nAutoRef<nSceneGraph2>     ref_sg;
         nAutoRef<nCollideServer>    ref_collide;

};
//------------------------------------------------------------------------------
#endif
```

B.6. nIstStairs_main.cc

```
#define N_IMPLEMENTS nIstStairs
//------------------------------------------------------------------------------
// Jose Larios Delgado  Class that implements the behavior to the stairs
//------------------------------------------------------------------------------

// includes

#include "ist/nIstStairs.h"
#include "ist/nIstEntity.h"
#include "ist/nistworld.h"
#include "kernel/nenv.h"
#include "collide/ncollideserver.h"
#include "collide/ncollideobject.h"
```

```
#include "kernel/ntimeserver.h"
#include "node/n3dnode.h"

class n3DNode;
class nChannelServer;
class nCollideObject;
class nCollideShape;
class nIstWorld;

nNebulaScriptClass(nIstStairs, "nroot");

//------------------------------------------------------------------------------
nIstStairs::nIstStairs():
                                    ref_gs(kernelServer,this),
        ref_ss(kernelServer,this),
        ref_sg(kernelServer,this),
        ref_collide(kernelServer,this)
{
            this->ref_gs    = "/sys/servers/gfx";
   this->ref_ss    = "/sys/servers/script";
   this->ref_sg    = "/sys/servers/sgraph2";
   this->ref_collide = "/sys/servers/collide";

}

//------------------------------------------------------------------------------
nIstStairs::~nIstStairs()
{
}

void nIstStairs::Trigger(float dt)
{
}

void nIstStairs::Collide()
{
}

void nIstStairs::AddImpulse(vector3 addv)
{
}
```

B.7. nIstPlayer.h

```
//============================================================================
// ist/nIstPlayer.h
// 2003 Enrique Larios
//         Jose Larios
//------------------------------------------------------------------------------

#ifndef N_ISTPLAYER_H
#define N_ISTPLAYER_H

// includes

#include "kernel/nroot.h"
#include "kernel/nkernelserver.h"
#include "kernel/ntimeserver.h"
#include "gfx/ngfxserver.h"
#include "gfx/nscenegraph2.h"
#include "input/ninputserver.h"
#include "misc/nconserver.h"
#include "misc/nparticleserver.h"
#include "kernel/nscriptserver.h"
#include "misc/nspecialfxserver.h"
#include "collide/ncollideserver.h"
```

```
#include "cal3d/ncal3dmodel.h"
#include "ist/nistworld.h"
#include "ist/nistentity.h"

#undef N_DEFINES
#define N_DEFINES nIstPlayer
#include "kernel/ndefdllclass.h"

#define SPACECOORD_SCALE 45.0F

//------------------------------------------------------------------------------
class N_PUBLIC nIstPlayer : public nIstEntity
{
private:

        nAutoRef<nGfxServer>                    ref_gs;
   nAutoRef<nInputServer>     ref_is;
   nAutoRef<nConServer>       ref_con;
   nAutoRef<nScriptServer>    ref_ss;
   nAutoRef<nSceneGraph2>     ref_sg;
        nAutoRef<nCollideServer>    ref_collide;

        n3DNode *playernode;
        nCal3DModel *model;

        float run_accel;                // in m/s^2
        float max_speed;

        float rot_speed;      // in degree/s

        bool isFW;
        bool isBW;
        bool isRL;
        bool isRR;
        bool isSTPFW;
        bool isSTPBW;
        bool isSL;
        bool isSR;
        bool isSTPSL;
        bool isSTPSR;
        bool isMRL;
        bool isMRR;

        nString splayernode;
        nString smodel;

        //helpers
        bool animWalk;

public:
   nIstPlayer();
   virtual ~nIstPlayer();

   static nKernelServer* kernelServer;

        virtual bool SaveCmds(nPersistServer* fileServer);

        //nIstEntity metods
        void Trigger(float dt);
        void Collide(void);
        void UpdatePosition(float dt);

        //Used in collisions to repell the moving object
        void AddImpulse(vector3 addv);

        //input communication
        void StartFW();
        void StartBW();
        void StartRL();
        void StartRR();
```

```
        void StartSL();
        void StartSR();
        void StartMRR();
        void StartMRL();
        void StopFW();
        void StopBW();
        void StopRL();
        void StopRR();
        void StopSL();
        void StopSR();
        void StopMRR();
        void StopMRL();

        void ActShoot();

        //Config
        void SetRotSpeed(float rs);
        void SetMaxRunSpeed(float ms);
        void SetRunAccel(float ac);
        float GetRotSpeed();
        float GetMaxRunSpeed();
        float GetRunAccel();

        void SetN3DNodeTar(const char *path);
        const char * GetN3DNodeTar() const;
        void SetNCalModTar(const char *path);
        const char * GetNCalModTar() const;

        void SetCollServ(const char *path);
        const char * GetCollServ() const;
};

inline
const char * nIstPlayer::GetN3DNodeTar() const
{
        return splayernode.Get();
}

inline
const char * nIstPlayer::GetNCalModTar() const
{
        return smodel.Get();
}

inline
void nIstPlayer::SetRotSpeed(float rs) {
        this->rot_speed=rs;
}

inline
void nIstPlayer::SetMaxRunSpeed(float ms) {
        this->max_speed=ms * SPACECOORD_SCALE;
}

inline
void nIstPlayer::SetRunAccel(float ac) {
        this->run_accel=ac * SPACECOORD_SCALE ;
}

inline
float nIstPlayer:: GetRotSpeed() {
        return rot_speed;
}

inline
float nIstPlayer::GetMaxRunSpeed() {
        return max_speed;
}
```

```
inline
float nIstPlayer::GetRunAccel() {
        return run_accel;
}

inline
void nIstPlayer::SetN3DNodeTar(const char *path) {
        n_assert(path);
        splayernode=path;
        this->playernode =(n3DNode *) kernelServer->Lookup(path);
}

inline
void nIstPlayer::SetNCalModTar(const char *path) {
        n_assert(path);
        smodel=path;

        this->model = (nCal3DModel *) kernelServer->Lookup(path);
}

inline
void nIstPlayer::SetCollServ(const char *path)
{
        n_assert(path);
        this->ref_collide=path;
}

inline
const char * nIstPlayer::GetCollServ() const
{
        return ref_collide.getname();
}

//Start
inline
void nIstPlayer::StartFW() {
        isFW=animWalk=true;
        isSTPFW=isSTPBW=false;

        if( model )
                model->NewCycle(4,0.5F,0.2F);
}

inline
void nIstPlayer::StartBW() {
        isBW=animWalk=true;
        isSTPFW=isSTPBW=false;

        if( model )
                model->NewCycle(4,0.5F,0.2F);
}

//------------------------------------------------------------------
inline
void nIstPlayer::StartSL() {
        isSL=animWalk=true;
        isSTPSL=isSTPSR=false;
        if( model )
                model->NewCycle(4,0.5F,0.2F);
}

inline
void nIstPlayer::StartSR() {
        isSR=animWalk=true;
        isSTPSL=isSTPSR=false;
        if( model )
                model->NewCycle(4,0.5F,0.2F);
}
```

```
//-----------------------------------------------------------------------
inline
void nIstPlayer::StartRR() {
        isRR=true;
        if( model )
                model->NewCycle(3,0.5F,0.2F);
}

inline
void nIstPlayer::StartRL() {
        isRL=true;
        if( model )
                model->NewCycle(3,0.5F,0.2F);
}

//-----------------------------------------------------------------------
inline
void nIstPlayer::StartMRR() {
        isMRR=true;

}

inline
void nIstPlayer::StartMRL() {
        isMRL=true;
}

//-----------------------------------------------------------------------
//Stop
inline
void nIstPlayer::StopFW() {
        isFW=false;
        isSTPFW=true;

        if( model && !isBW && !isRR && !isRL )
                model->NewCycle(4,0.5F,0.2F);                   //Later you can add here the stopping
animation
}

inline
void nIstPlayer::StopBW() {
        isBW=false;
        isSTPBW=true;

        if( model && !isFW && !isRR && !isRL)
                model->NewCycle(4,0.5F,0.2F);           //Later you can add here the stopping animation
}

//-----------------------------------------------------------------
inline
void nIstPlayer::StopSL() {
        isSL=false;
        isSTPSL=true;
        if( model && !isRR && !isRL && !isSR )
                model->NewCycle(4,0.5F,0.2F);
}

inline
void nIstPlayer::StopSR() {
        isSR=false;
        isSTPSR=true;
        if( model && !isRR && !isRL && !isSL )
                model->NewCycle(4,0.5F,0.2F);
}

//-----------------------------------------------------------------
```

```
inline
void nIstPlayer::StopRR() {
        isRR=false;

        if( isFW || isBW)
                model->NewCycle(4,0.5F,0.2F);
        else if(model && !isRR )
                model->NewCycle(0,0.5F,0.2F);
}

inline
void nIstPlayer::StopRL() {
        isRL=false;

        if( isFW || isBW)
                model->NewCycle(4,0.5F,0.2F);
        else if(model && !isRR )
                model->NewCycle(0,0.5F,0.2F);
}

//------------------------------------------------------------
inline
void nIstPlayer::StopMRR() {
        isMRR=false;
}

inline
void nIstPlayer::StopMRL() {
        isMRL=false;
}

//------------------------------------------------------------
inline
void nIstPlayer::ActShoot() {
        if( model )
                model->Action(2,0.1F,0.2F);
}

inline
void nIstPlayer::AddImpulse(vector3 addv)
{
        velVector+=addv;
}
//------------------------------------------------------------------------------
#endif
```

B.8. nIstPlayer_main.cc

```
#define N_IMPLEMENTS nIstPlayer
//==============================================================================
// ist/nistplayer_main
// 2003 Enrique Larios
//      Jose Larios
//------------------------------------------------------------------------------

// includes
#include "ist/nIstPlayer.h"
#include "ist/nIstEntity.h"
#include "ist/nistworld.h"
#include "kernel/nenv.h"
#include "collide/ncollideserver.h"
#include "collide/ncollideobject.h"
#include "kernel/ntimeserver.h"
#include "node/n3dnode.h"
#include <string.h>
```

```
class n3DNode;
class nChannelServer;
class nCollideObject;
class nCollideShape;
class nIstWorld;

nNebulaScriptClass(nIstPlayer, "nroot");
//============================================================================
//
//----------------------------------------------------------------------------

nIstPlayer::nIstPlayer()
      : ref_gs(kernelServer,this),
        ref_is(kernelServer,this),
        ref_ss(kernelServer,this),
        ref_sg(kernelServer,this),
        ref_con(kernelServer,this),
        ref_collide(kernelServer,this),
                              splayernode(),
                              smodel()
{
  // servers
        this->ref_gs    = "/sys/servers/gfx";
  this->ref_is    = "/sys/servers/input";
  this->ref_ss    = "/sys/servers/script";
  this->ref_sg    = "/sys/servers/sgraph2";
  this->ref_con   = "/sys/servers/console";
  this->ref_collide = "/sys/servers/collide";

        //state
        isFW=false;
        isBW=false;
        isRL=false;
        isRR=false;
        isSTPFW=false;
        isSTPBW=false;
        isSL=false;
        isSR=false;
        isMRL=false;
        isMRR=false;

        max_speed = 2.5F * SPACECOORD_SCALE;                    //in m/s
        run_accel = 2.0F * SPACECOORD_SCALE;                    //in m/s^2
        rot_speed = 60.0F;                              //in deggrees/s

}

nIstPlayer::~nIstPlayer()
{
}

void
nIstPlayer::Trigger(float dt)
{
        //Updating collideObject's position
        collideObject->Transform(0.0F, playernode->GetM() );
        nIstEntity::Trigger(dt);    //to handle forces and gravity
        //Takes care of the player state
        if(isRR)
        {
                if( !isSTPFW || !isSTPBW)                       //you can't rotate while stopping
                {
                        vector3 r;
                        playernode->GetR(r.x,r.y,r.z);
                        r.y -= rot_speed*dt;
                        playernode->Rxyz(r.x,r.y,r.z);

                        velVector.set(0.99F*velVector.x,velVector.y,0.99F*velVector.z); //brake a bit while turning
                }
        }
```

```
if(isRL)
{
	if( !isSTPFW || !isSTPBW)			//you can't rotate while stopping
	{
		vector3 r;
		playernode->GetR(r.x,r.y,r.z);
		r.y += rot_speed*dt;
		playernode->Rxyz(r.x,r.y,r.z);
		velVector.set(0.99F*velVector.x,velVector.y,0.99F*velVector.z);  //brake a bit while turning
	}
}

if(isMRR)
{
	if( !isSTPFW || !isSTPBW)
		velVector.set(0.99F*velVector.x,velVector.y,0.99F*velVector.z);
}

if(isMRL)
{
	if( !isSTPFW || !isSTPBW)
		velVector.set(0.99F*velVector.x,velVector.y,0.99F*velVector.z);
}

if (isFW)
{
	if (!isSTPBW && !isSTPSL && !isSTPSR)
	{
		vector3 r;
		matrix44 m;

		m=playernode->GetM();
		r = m.y_component();
		r.norm();
		r *=-run_accel;
		accForVector += r*mass;

		if(velVector.len() > this->max_speed/2 && animWalk)
//start running
		{
			model->BlendCycle(1,0.5F,0.2F);
			animWalk=false;
		}
		else if(velVector.len() <= this->max_speed/2  && !animWalk)		//start walking
		{
			model->BlendCycle(4,0.5F,0.2F);
			animWalk=true;
		}
	}
}

if(isBW)
{
	if(!isSTPFW && !isSTPSL && !isSTPSR)
	{
		vector3 r;
		matrix44 m;

		m= playernode->GetM();
		r = m.y_component();
		r.norm();

		//integrating
		r *= run_accel;
		accForVector += r*mass;

	}

}
```

```
if (isSL)
{
        if (!isSTPBW && !isSTPFW && !isSTPSR)
        {
                vector3 r;
                matrix44 m;
                m=playernode->GetM();
                r = m.x_component();
                r.norm();
                //integrating
                r *=run_accel;
                accForVector += r*mass;
        }
}

if(isSR)
{
        if (!isSTPBW && !isSTPSL && !isSTPFW)
        {
                vector3 r;
                matrix44 m;
                m= playernode->GetM();
                r = m.x_component();
                r.norm();
                //integrating
                r *=-run_accel;
                accForVector += r*mass;
        }
}

if(isSTPFW)
{
        vector3 r(velVector.x,0.0,velVector.z);

        if(r.len()>4)                           //value that might change depending on the scale enrique
        {
                velVector.set(0.97F*velVector.x,velVector.y,0.97F*velVector.z);
        }
        else
        {
                velVector.set(0.0F,velVector.y,0.0F);
                model->ClearCycle(4,0.2F);  //clear walking
                model->ClearCycle(1,0.2F);   //clear running

                if(isRR || isRL)
                        model->NewCycle(3,0.5F,0.2F);

                if(isBW || isSL || isSR)
                        model->BlendCycle(4,0.5F,0.2F);
                else
                        model->NewCycle(0,0.5F,0.2F);

                isSTPFW=false;
        }
}

if(isSTPBW)
{
        vector3 r(velVector.x,0.0,velVector.z);

        if(r.len()>4)                   //value that might change depending on the scale enrique
        {
                velVector.set(0.97F*velVector.x,velVector.y,0.97F*velVector.z);
        }


        else
        {
                velVector.set(0.0F,velVector.y,0.0F);
```

```
                    model->ClearCycle(4,0.2F);  //clear walking
                    model->ClearCycle(1,0.2F);   //clear running

                    if(isRR || isRL)
                            model->NewCycle(3,0.5F,0.2F);

                    if(isFW || isSL || isSR)
                            model->BlendCycle(4,0.5F,0.2F);

                    else
                            model->NewCycle(0,0.5F,0.2F);

                    isSTPBW=false;
            }
    }

    if(isSTPSL)
    {
            vector3 r(velVector.x,0.0,velVector.z);

            if(r.len()>4)                           //value that might change depending on the scale enrique
            {
                    velVector.set(0.97F*velVector.x,velVector.y,0.97F*velVector.z);
            }
            else
            {
                    velVector.set(0.0F,velVector.y,0.0F);
                    model->ClearCycle(4,0.2F);  //clear walking
                    model->ClearCycle(1,0.2F);   //clear running

                    if(isRR || isRL)
                            model->NewCycle(3,0.5F,0.2F);

                    if(isSR || isFW || isBW)
                            model->BlendCycle(4,0.5F,0.2F);

                    else
                            model->NewCycle(0,0.5F,0.2F);

                    isSTPSL=false;
            }
    }

    if(isSTPSR)
    {
            vector3 r(velVector.x,0.0,velVector.z);

            if(r.len()>4)                           //value that might change depending on the scale enrique
            {
                    velVector.set(0.97F*velVector.x,velVector.y,0.97F*velVector.z);
            }
            else
            {
                    velVector.set(0.0F,velVector.y,0.0F);
                    model->ClearCycle(4,0.2F);  //clear walking
                    model->ClearCycle(1,0.2F);   //clear running

                    if(isRR || isRL)
                            model->NewCycle(3,0.5F,0.2F);

                    if(isSL || isFW || isBW)
                            model->BlendCycle(4,0.5F,0.2F);

                    else
                            model->NewCycle(0,0.5F,0.2F);

                    isSTPSR=false;
            }
    }
}
```

```
void nIstPlayer::Collide(void)
{
        if (collideObject == NULL)
    return;

  nCollideReport** report;
  int num_colls = collideObject->GetCollissions(report);

  if (num_colls == 0)
    return;

  for (int i = 0; i < num_colls; ++i)
  {
    nIstEntity* other = (nIstEntity*)report[i]->co2->GetClientData();
    vector3 contact_normal = report[i]->co1_normal;
                vector3  other_contact_normal = report[i]->co2_normal;

    if (this == other)
    {
      other = (nIstEntity*)report[i]->co1->GetClientData();
      contact_normal = report[i]->co2_normal;
                        other_contact_normal = report[i]->co1_normal;
    }

                contact_normal.norm();
                other_contact_normal.norm();

    if (other  == NULL)
      continue;
                char buffy[30];
                //copy the name
                strcpy(buffy,other->GetClass()->GetName() );


                if( !strcmp( buffy,"nistplayer") || !strcmp( buffy,"nistobject")  )
                {
                        //just a hack, only recive force if moving
                        if(isFW || isBW || isSL || isSR )
                        {
                                float comp;

                                if(    ( comp=other_contact_normal%(forVector+accForVector) ) <= 0     )
                                {
                                        vector3 normalForce=other_contact_normal*( -1.0F*comp   );
                                        this->accForVector+=normalForce;
                                }

                                if(( comp=other_contact_normal%( velVector - other->GetVelVector() )) <= 0 )
                                {
                                        vector3 impulse=other_contact_normal*( -1.05F*comp );   // 1.05F some
elasticity to the collition
                                        this->velVector+=impulse;
                                }
                        }
                }
                else if( !strcmp( buffy,"niststruct") )
                {
                        float comp;

                        if(other_contact_normal.y<0)
                        {
                                n_printf("wrong pointing normal!  Correct the map normals\n");   //this only when
debugging maps
                                other_contact_normal.y*=-1.0;

                        }
```

```
                        if(     ( comp=other_contact_normal%(forVector+accForVector) ) <= 0 )
                        {
                                vector3 normalForce=other_contact_normal*( -1.0F*comp );
                                this->accForVector+=normalForce;
                        }

                        if((comp=other_contact_normal%velVector) <=0 )   //check this because maybe it's the map
enirque, obiously last conditon makes the if redundant
                        {
                                vector3 impulse=other_contact_normal*( -1.0F*comp );
                                this->velVector+=impulse;
                        }
                } //end else if niststruct

                else if( !strcmp( buffy,"niststairs") )
                {
                        float comp;

                        if(other_contact_normal.y<0)
                        {
                                n_printf("wrong pointing normal!  Correct the map normals\n"),   //this only when
debugging maps
                                other_contact_normal.y*=-1.0;
                        }

                        other_contact_normal.y=1.0;

                        if(     ( comp=other_contact_normal%(forVector+accForVector) ) <= 0 )
                        {
                                printf("%f\n",other_contact_normal.y);
                                vector3 normalForce=other_contact_normal*( -1.5*comp );
                                normalForce.x=0;
                                normalForce.z=0;
                                this->accForVector+=normalForce;
                        }

                        if((comp=other_contact_normal%velVector) <=0 )   //check this because maybe it's the map
enirque, obiously last conditon makes the if redundant
                        {
                                vector3 impulse=other_contact_normal*( -1.5*comp );
                                impulse.x=0;
                                impulse.z=0;
                                this->velVector+=impulse;
                        }
                } //end else if niststairs
        } //end for
}       //end nistplayer collide
void nIstPlayer::UpdatePosition(float dt)
{
        this->forVector+=accForVector;
        this->accForVector.set(0.0F,0.0F,0.0F);                 //cleaning
        this->aclVector=this->forVector*invMass;                //Getting the acceleration
        vector3 velXZComp(velVector.x,0.0,velVector.z);
        if(velXZComp.len() > this->max_speed )                  //handling the horizontal max speed
        {
                velXZComp.norm();
                velXZComp*=max_speed;
                this->velVector.set(velXZComp.x,velVector.y,velXZComp.z);
        }
        //integrating
        this->velVector+=aclVector*dt;  //Getting dV and adding it to vel  v1=v0+dv

        vector3 p;
        playernode->GetT(p.x,p.y,p.z);
        //integrating
        p+=this->velVector*dt;                  //Getting dX and adding it to pos X1=X0+dX
        //setting the new position
        playernode->Txyz(p.x,p.y,p.z);

}
```

B.9. nIstNPC.h

```
//================================================================================
// ist/nIstNPC.h
// 2002 Enrique Larios
//        Jose Larios
//--------------------------------------------------------------------

#ifndef N_ISTNPC_H
#define N_ISTNPC_H

// includes

#include <string>
#include "kernel/nroot.h"
#include "kernel/nkernelserver.h"
#include "kernel/ntimeserver.h"
#include "gfx/ngfxserver.h"
#include "gfx/nscenegraph2.h"
#include "input/ninputserver.h"
#include "misc/nconserver.h"
#include "misc/nparticleserver.h"
#include "kernel/nscriptserver.h"
#include "misc/nspecialfxserver.h"
#include "collide/ncollideserver.h"
#include "path/npath.h"
#include "script/ntclscriptlet.h"
#include "cal3d/ncal3dmodel.h"
#include "ist/nistworld.h"
#include "ist/nistentity.h"

#undef N_DEFINES
#define N_DEFINES nIstNPC
#include "kernel/ndefdllclass.h"

//--------------------------------------------------------------------------
class N_PUBLIC nIstNPC : public nIstEntity
{
private:
        nAutoRef<nGfxServer>                ref_gs;
   nAutoRef<nInputServer>     ref_is;
   nAutoRef<nConServer>       ref_con;
   nAutoRef<nScriptServer>    ref_ss;
   nAutoRef<nSceneGraph2>     ref_sg;
        nAutoRef<nCollideServer>    ref_collide;

        nAutoRef<n3DNode>                           ref_npcnode;
        nAutoRef<nCal3DModel>               ref_npcmodel;

        nTclScriptlet *collScriptlet;

        n3DNode *npcnode;
        nCal3DModel *npcmodel;

        float run_speed;                // in m/s
        float rot_speed;     // in degree/s

        //path seek State specific attibutes
        double pathDur;
        bool seekingBack;
        double pathTime;
        nPath *actualPath;

        //Special states
        bool StatePtSk;
```

```
        //Enable Actions on diferent Events
        bool EnEvColl;
        nString collScriptFile;

        nIstEntity *lastCollEntity;

        nString snpcnode;
        nString snpcmodel;

protected:
        bool BeginPathSeeking(const char *orig,const char *dest,double duration);
        vector3 *SeekPath(float dt);

public:
  nIstNPC();
  virtual ~nIstNPC();
        virtual void Initialize();

  static nKernelServer* kernelServer;

        virtual bool SaveCmds(nPersistServer* fileServer);

        //nIstEntity metods
        void Trigger(float dt);
        void Collide(void);
        void UpdatePosition(float dt);

        //Used in collisions to repell the moving object
        void AddImpulse(vector3 addv);

        //Anim. Note: might change with when more animations are added
        void AnimWalk();
        void AnimStrut();
        void AnimIdle();

        //Config
        void SetRotSpeed(float rs);
        void SetRunSpeed(float rs);
        float GetRotSpeed();
        float GetRunSpeed();

        void SetN3DNodeTar(const char *path);
        const char * GetN3DNodeTar() const;
        void SetNCalModTar(const char *path);
        const char * GetNCalModTar() const;

        void SetCollServ(const char *path);
        const char * GetCollServ() const;

        void EnableEventColl(const char *collScriptFile);
        void DisableEventColl();
        nIstEntity *GetLastCollEntity(void);
        bool SetStatePathSeeking(const char *orig,const char *dest,double duration);
        bool IsPathSeeking(void);

};

inline
const char * nIstNPC::GetN3DNodeTar() const
{
        return ref_npcnode.getname();
}

inline
const char * nIstNPC::GetNCalModTar() const
{
        return ref_npcmodel.getname();
}
```

```
inline
void nlstNPC::SetRotSpeed(float rs) {
        this->rot_speed=rs;
}

inline
void nlstNPC::SetRunSpeed(float rs) {
        this->run_speed=rs;
}

inline
float nlstNPC:: GetRotSpeed() {
        return rot_speed;
}

inline
float nlstNPC:: GetRunSpeed() {
        return run_speed;
}

inline
void nlstNPC::SetN3DNodeTar(const char *path) {
        n_assert(path);
        snpcnode=path;
        this->ref_npcnode=path;
        this->npcnode = ref_npcnode.get();  //modifiable hack
}

inline
void nlstNPC::SetNCalModTar(const char *path) {
        n_assert(path);
        snpcmodel=path;
        this->ref_npcmodel=path;
        this->npcmodel = ref_npcmodel.get();    //modifiable hack
}

inline
void nlstNPC::SetCollServ(const char *path)
{
        n_assert(path);
        this->ref_collide=path;
}

inline
const char * nlstNPC::GetCollServ() const
{
        return ref_collide.getname();
}

inline
void nlstNPC::AnimWalk() {
        if( npcmodel )
                npcmodel->NewCycle(4,0.5F,0.2F),
}

inline
void nlstNPC::AnimStrut() {
        if( npcmodel )
                npcmodel->NewCycle(3,0.5F,0.2F);
}

inline
void nlstNPC::AnimIdle() {
        if( npcmodel)
                npcmodel->NewCycle(0,0.5F,0.2F);
}
```

```
inline
void nIstNPC::AddImpulse(vector3 addv)
{
        velVector+=addv;
}

inline
nIstEntity *nIstNPC::GetLastCollEntity(void)
{
        return this->lastCollEntity;
}

inline
void nIstNPC::EnableEventColl(const char *collScriptFile)
{

        this->EnEvColl= TRUE;
        this->collScriptFile=collScriptFile;
}

inline
void nIstNPC::DisableEventColl()
{
        this->EnEvColl=false;
}
//------------------------------------------------------------------------------
#endif
```

B.10. nIstNPC_main.cc

```
#define N_IMPLEMENTS nIstNPC
//==============================================================================
// ist/nistnpc_main
// 2003 Enrique Larios
//      Jose Larios
//------------------------------------------------------------------------------

// includes
#include "ist/nIstNPC.h"
#include "ist/nIstEntity.h"
#include "ist/nistworld.h"
#include "kernel/nenv.h"
#include "collide/ncollideserver.h"
#include "collide/ncollideobject.h"
#include "kernel/ntimeserver.h"
#include "node/n3dnode.h"
#include <string.h>

class n3DNode;
class nChannelServer;
class nCollideObject;
class nCollideShape;
class nIstWorld;

nNebulaScriptClass(nIstNPC, "nroot");
//==============================================================================
//
//------------------------------------------------------------------------------

nIstNPC::nIstNPC()
    : ref_gs(kernelServer,this),
      ref_is(kernelServer,this),
      ref_ss(kernelServer,this),
      ref_sg(kernelServer,this),
      ref_con(kernelServer,this),
      ref_collide(kernelServer,this),
```

```
                        ref_npcnode(kernelServer,this),
                        ref_npcmodel(kernelServer,this),
                        snpcnode(),
                        snpcmodel(),
                        run_speed(160),
                        rot_speed(65),
                        actualPath(NULL),
                        StatePtSk(false),
                        seekingBack(false),
                        EnEvColl(false),
                        lastCollEntity(NULL),
                        pathTime(0),
                        pathDur(0)
{
  // servers
        this->ref_gs   = "/sys/servers/gfx";
  this->ref_is   = "/sys/servers/input";
  this->ref_ss   = "/sys/servers/script";
  this->ref_sg   = "/sys/servers/sgraph2";
  this->ref_con  = "/sys/servers/console";
  this->ref_collide = "/sys/servers/collide";
}

nIstNPC::~nIstNPC()
{
        collScriptlet->~nTclScriptlet();
}

//Note: here is where we create and add the scriptlets for EVERY action
void nIstNPC::Initialize()
{
        nIstEntity::Initialize();
        std::string scriptPath="/sys/share/scriptlets/";
        scriptPath+=GetName();
        scriptPath+="/";
        this->collScriptlet= (nTclScriptlet *) kernelServer->New("ntclscriptlet",( scriptPath+"collscriptlet").c_str() );
}

void
nIstNPC::Trigger(float dt)
{
        this->collideObject->Transform(0.0F, npcnode->GetM() );
        nIstEntity::Trigger(dt);        //to handle forces and gravity
        if(StatePtSk)
        {
                if(pathTime<pathDur )
                {
                        //Getting Actual position from n3dnode
                        vector3   actPos;
                        npcnode->GetT(actPos.x,actPos.y,actPos.z);
                        //Getting the new position from the path
                        vector3 *newPos=SeekPath(dt);
                        //creating the new direction vector
                        vector3 * newDir=new vector3(newPos->x-actPos.x,0.0F,newPos->z-actPos.z);
                        newDir->norm();
                        newDir->y=0;
                        //just in case the loop was too fast causing no difference between act and new position
                        if(newDir->len() >0)
                        {
                                vector3 w;
                                npcnode->GetR(w.x,w.y,w.z);             //getting the old rotation

                                if(newDir->x>=0)
                                        w.y = n_rad2deg( float( acos( newDir->z ) ) );
                                else
                                        w.y = -n_rad2deg( float( acos( newDir->z ) ) );

                                //rotating
                                npcnode->Rxyz(w.x, w.y, w.z);
                                //setting the new position
```

```
                                        npcnode->Txyz(newPos->x,newPos->y,newPos->z);
                                }
                        }
                        else
                        {
                                this->StatePtSk=false;
                                this->AnimIdle();
                        }
                }//end StatePtSk
                //Updating collideObject's position
                collideObject->Transform(0.0F, npcnode->GetM() );
}

void
nIstNPC::Collide(void)
{
        if (NULL == collideObject)
    return;

  nCollideReport** report;
  int num_colls = collideObject->GetCollissions(report);
  if (0 == num_colls)
    return;

  for (int i = 0; i < num_colls; ++i)
  {
    nIstEntity* other = (nIstEntity*)report[i]->co2->GetClientData();
    vector3* contact_normal = &report[i]->co1_normal;

    if (this == other)
    {
      other = (nIstEntity*)report[i]->co1->GetClientData();
      contact_normal = &report[i]->co2_normal;
    }

    if (other  == NULL)
      continue;
                        char buffy[30];
                        lastCollEntity=other;
                        strcpy(buffy,other->GetClass()->GetName() );

                        if( !strcmp( buffy,"nistplayer") || !strcmp( buffy,"nistobject") || !strcmp( buffy,"niststruct") || !strcmp(
buffy,"nistnpc"))
                        {
                                (*contact_normal).norm();
                                (*contact_normal).y=0;

                                //just a hack, only recive force if moving
                                if(StatePtSk )
                                {
                                        (*contact_normal).norm();
                                        (*contact_normal).y=0;
                                        velVector += (*contact_normal)*(-run_speed)*1.5;   //TODO just that right now in
Trigger when in PtSk State npc doesn't pay attention to velVevtor

                                }
                                else
                                {
                                        other->AddImpulse(-(*contact_normal)*(-run_speed)*1.5);
                                }

                                if(EnEvColl)
                                {
                                        const char *res;
                                        ref_ss->RunScript(collScriptFile.Get(),res);
                                }
                        }
        }
}
```

```
void nIstNPC::UpdatePosition(float dt)
{
}

bool nIstNPC::SetStatePathSeeking(const char *orig,const char *dest,double duration)
{
		if(this->BeginPathSeeking(orig,dest,duration) )
		{
			this->StatePtSk=true;
			this->AnimWalk();
			return true;
		}
	return false;
}

bool nIstNPC::IsPathSeeking(void)
{
	return this->StatePtSk;
}

bool nIstNPC::BeginPathSeeking(const char *orig,const char *dest,double duration)
{
	std::string or, de, sub;
	std::string pth="/sys/share/paths/";
	or=orig; de=dest; sub="_";
	actualPath= (nPath *) kernelServer->Lookup( (pth+or+sub+de).c_str() );
	if(!actualPath)
	{
		actualPath= (nPath *) kernelServer->Lookup((pth+de+sub+or).c_str());
		if(!actualPath)
			return false;
		seekingBack=true;
	}
	else
	{
		seekingBack=false;
	}

	pathDur=duration;
	pathTime=0;
	return true;
}

vector3 *nIstNPC::SeekPath(float dt)
{
	if(StatePtSk)
	{
		float u;
		pathTime+=dt;

		if(!seekingBack)
		{
			if(  (u= float( pathTime/pathDur) ) <= 1.0F )
				return actualPath->PointAt( u );
			else
				return actualPath->PointAt( 1.0F );
		}

		else
		{
			if(  (u= float( (pathDur-pathTime)/pathDur) ) >= 0.0F )
				return actualPath->PointAt( u );
			else
				return actualPath->PointAt( 0.0F );
		}
	}
	else
		return NULL;
}
```

B.11. Ejemplo de un ciclo principal sencillo

```
//--------------------------------------------------------------------
// DESCRIPCION:
// Una esfera  gira mientras se mueve de izquierda a derecha y una caja gira mientras
// hace la figura de un ocho alrededor de la esfera.
//--------------------------------------------------------------------

// Todas las cabeceras, clases, etc son colocadas aquí.

#include "load_scripts.h"

int main(int argc, char *argv[])
{
  char *port_name = "gfxserv";
  char *server_class = "nglserver";
  char *mode = "w(512)-h(512)";
  char *script_server = "ntclserver";
   bool grid = true;
   bool nosleep = false;
   const char* result;
   V.ident();
   vector3 t(0.0f,2.5f,0.0f);
   V.translate(t);

   // Se inicia un nuevo servidor del núcleo.
  nKernelServer* ks      = new nKernelServer;

 // Se inician los servidores básicos.
nFileServer2* fs2      = (nFileServer2*)      ks->New("nfileserver2",        "/sys/servers/file2");
nGfxServer* gfx        = (nGfxServer *)       ks->New(server_class,          "/sys/servers/gfx");
nInputServer* inp      = (nInputServer *)     ks->New("ninputserver",        "/sys/servers/input");
nTclServer* script     = (nTclServer *)       ks->New(script_server,         "/sys/servers/script");
nSceneGraph2* graph    = (nSceneGraph2 *)     ks->New("nscenegraph2",        "/sys/servers/sgraph2");
nShadowServer* shadow  = (nShadowServer *)    ks->New("nsbufshadowserver",   "/sys/servers/shadow");
nChannelServer* chn    = (nChannelServer*)    ks->New("nchannelserver",      "/sys/servers/channel");
nConServer *con        = (nConServer *)       ks->New("nconserver",          "/sys/servers/console");
nMathServer *math      = (nMathServer *)      ks->New("nmathserver",         "/sys/servers/math");
nParticleServer *part  = (nParticleServer *)  ks->New("nparticleserver",     "/sys/servers/particle");
nSpecialFxServer *fx   = (nSpecialFxServer *) ks->New("nspecialfxserver",    "/sys/servers/specialfx");
n3DNode *root          = (n3DNode *)          ks->New("n3dnode",             "/usr/scene");

//Carga la caja
//Runscript = Evalúa un archivo de script.
//Lookup = Busca por los directorios indicados y regresa  el objeto nRoot correspondiente.
script->RunScript("..\\load_scripts\\box\\box.n",result);
box = (n3DNode*) ks->Lookup("/load_scripts/box");

//Carga la esfera.
// RunScript = Evalúa un archivo de script.
// Lookup = Busca por los directorios indicados y regresa el objeto nRoot correspondiente.
script->RunScript("..\\load_scripts\\ball\\ball.n",result);
ball = (n3DNode*) ks->Lookup("/load_scripts/ball");

//Define un Nuevo modo de presentación. El cual no será mostrado
//hasta que el dispositivo de despliegue sea reabierto por medio
//de  CloseDisplay()/OpenDisplay().
gfx->SetDisplayMode(mode);

//Carga el dispositivo de despliegue definido por setDisplayMode().
//Esto restaurará la ventana de la aplicación y creará un nuevo dispositivo de direct3d.
gfx->OpenDisplay();

//Si la variable running  tiene el valor de false, la ventana del nGlServer
//se abrirá y entonces se cerrara automáticamente.
bool running = true;

//Inicia el ciclo principal.
```

```
while (gfx->Trigger() && running)
{
  if (!script->Trigger()) running = false;

  //Se actualiza el servidor del kernel.
  ks->Trigger();
  ks->ts->Trigger();

  double t = ks->ts->GetFrameTime();
  inp->Trigger(t);

  //Comienza la entrada del ratón y del teclado.
  nInputEvent *ie;

  // Primer evento de entrada.
  if ((ie = inp->FirstEvent()))
  {
     //¿Ha sido presionada una tecla del ratón o del teclado?
     // handleInput() es una función del usuario.
     do handleInput(con,ie); while ((ie = inp->NextEvent(ie)));
     inp->FlushEvents();
  }

  //Se establece la velocidad a la cual la pantalla se mueve cuando una tecla
  // del ratón o del teclado han sido  presionadas handleViewer() es una función
  //creada por el usuario.
  handleViewer();

  // Dibuja la escena.
  if (gfx->BeginScene())
  {
     nRState rs;
     matrix44 vwr;
     vwr = V;

     //Invierte una matriz de 4x4, la cual consiste  de una matriz de
     // rotación de 3x3 y una traslación (todo aquello que tiene [0,0,0,1]
     // en la columna de más a la derecha) esto es  mucho más sencillo que
     //invertir una matriz  real de 4x4.
     vwr.invert_simple();

     // Se actualiza el servidor de partículas.
     part->Trigger();

     //Se  inicia la matriz del observador (camara).
     gfx->SetMatrix(N_MXM_VIEWER,V);
     chn->SetChannel1f(chn->GenChannel("time"), (float) t);
     chn->SetChannel1f(chn->GenChannel("gtime"), (float) t);

     // Comienza la escena.
     if (graph->BeginScene(vwr))
     {
        fx->Begin();

        // Une una jerarquía del visnode a la escena.
        graph->Attach(root, 0);
        fx->End(graph);

      // Fin de la escena.
        graph->EndScene(true);
     }

     /*Se dibuja la malla en la pantalla con un fondo azul*/
      if (grid)
      {
         rs.Set(N_RS_TEXTUREHANDLE,0);  gfx->SetState(rs);
         rs.Set(N_RS_LIGHTING,N_FALSE); gfx->SetState(rs);

          // Se fija una de las matrices de proyección.
          gfx->SetMatrix(N_MXM_MODELVIEW,vwr);
```

```
            gfx->Begin(N_PTYPE_LINE_LIST);

            // Se fija el color de fondo.
              gfx->Rgba(0.1f,0.5f,2.0f,1.0f);

            // Se fija el tamaño de la malla.
              float p;
              for (p=-25.0f; p<=+25.0f; p+=1.0f)
              {
                  gfx->Coord(p, 0.0f, -25.0f);
                 gfx->Coord(p, 0.0f, +25.0f);
                 gfx->Coord(-25.0f, 0.0f, p);
                 gfx->Coord(+25.0f, 0.0f, p);
              }

              // Termina Gfxserver
              gfx->End();
             rs.Set(N_RS_LIGHTING,N_TRUE); gfx->SetState(rs);
         }
          /* Finaliza el dibujo de la malla*/

          //Se Finaliza la escena.
          gfx->EndScene();
      }
    }

    //Fin del ciclo principal.
   //Se cierra el dispositivo de despliegue abierto con la función OpenDisplay().
   //Esto minimiza la ventana de la aplicación y destruye el dispositivo de Direct3d
    gfx->CloseDisplay();

    if (root)   root->Release();
    if (part)   part->Release();
    if (math)   math->Release();
    if (con)    con->Release();
    if (chn)    chn->Release();
    if (shadow) shadow->Release();
    if (graph)  graph->Release();
    if (script) script->Release();
    if (gfx)    gfx->Release();
    if (inp)    inp->Release();
    if (fs2)    fs2->Release();
    if (load_scripts)  load_scripts->Release();
    if (box)                                    box->Release();
    if (ball)                                ball->Release();
   delete ks;
   return 0;
}
```

Bibliografía

[GAR88] De Asís Garrote, María Dolores. Formas de Comunicación en la Narrativa. Ed. Fundamentos, Madrid España, 1988, pp. 18-86.

[DEI99] Deitel, Harvey. ¿Cómo Programar en C++?, Pearson Education, 2da ed, México, 1999.

[DUA02] Duarte Pérez, Ricardo. Un Modelo para el Control del Movimiento Humano Basado en Cinemática Inversa, Tesis de maestría, UNAM, Posgrado en Ciencia e Ingeniería de la Computación, 2002.

[ERA03] Erasmatron. http://www.erasmatazz.com

[ESS95] Esslin, Martin. An Anatomy of Drama, Ed. Hill & Wang, Nueva York EE.UU., 1995, pp. 8-94.

[IZA03] Información Tocable. http://www.zgdv.de/zgdv/departments/z5/Z5Presse/Presse_2002_02/

[GEI03] Geist. http://www.zgdv.de/zgdv/departments/z5/Z5Projects/Geist_1/index_html_en

[MIM03]. Motor Mimesis. http://mimesis.csc.ncsu.edu/

[NEB03] Motor de juegos nebula. http://nebuladevice.sourceforge.net

[NIC80] Nicoll, Allardyce. The Theory of Drama, Arno Press, Nueva York EE.UU., 1980, pp. 25-39.

[PER96] Perlin K, Goldberg A. Improv: A system for scripting interactive actors in virtual worlds. Computer Graphics, 1996;29(3).

[PRE97] Pressman, Roger. Ingeniería del software: Un enfoque práctico, McGraw Hill, 4ta ed., México, 1997, pp. 367-424.

[RTJ03] HTN. http://www.wheelie.tees.ac.uk/users/f.charles/publications/ conferences/2001/ca2001.pdf

[RUS96] Russell, Stuart. Inteligencia Artificial: Un enfoque moderno, Prentice Hall, México, 1996, pp. 1-23

[WOO96] Woo, Mason. OpenGL programming guide: the official guide to learning OpenGL, Addison Wesley, 2da ed., EE.UU., 1996, pp. 2-26.

[SZI99] Szilas, Nicolas. Interactive drama on computer: beyond linear narrative. In: AAAI Fall Symposium. Menlo Park, CA: AAAI Press 1999. pp 150-156.

[LAW01] Lawrence, Deborah. Social Dynamincs of storytelling: Implications for story base design. 2001.

[WEH97] Wehling, Jasón. Aproveche las noches con Java, Prentice Hall, México, 1997, pp. 2-18.

[SKO02] Skov, Mikael. Designing interactive narrative systems: is object-orientation useful?, Computer Graphics, 2002; (26)1.