01132
54

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

Desarrollo de un API para la creación de Historias Interactivas

Tesis

que para obtener el título de

Ingeniero en Computación

Presenta

Enrique Larios Delgado

Director de Tesis:
Dr. Jesús Savage Carmona

México, D. F. 2003

Universidad Nacional
Autónoma de México

Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

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

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

PAGINACION

DISCONTINUA

Dedicatoria

A mi familia:

A mi Madre
A mi Padre
A mi Hermano
A mi Hermana
A Todos

Que sin su apoyo este logro no sería posible.

Agradecimientos

Quiero expresar un sincero agradecimiento a mi asesor de tesis **Jesús Savage Carmona** por el apoyo que me ha dado para terminar este trabajo.

A mi **hermano** cuya ayuda contribuyo a la realización de este trabajo.

Un agradecimiento para mis profesores (as) por sus enseñanzas y su experiencia.

Al equipo del Laboratorio de Interfaces Inteligentes por el apoyo recibido.

A todos los compañeros que contribuyeron con el desarrollo de esta tesis.

Índice General

Índice General ***I***

Índice de Figuras ***V***

Índice de Cuadros ***VII***

Resumen ***1***

Introducción ***3***

1. Historias Interactivas por Computadora (HIC) ***7***

1.1. Historias Interactivas 9
1.1.1. La historia 9
1.1.2. La narración 10
1.1.3. El drama 11
1.1.4. La interactividad. 15
1.1.5. Definición de Historia Interactiva 15
1.1.6. Drama o Narrativa Interactiva 16
1.2. Mecanismos de Interactividad en el Drama Interactivo por Computadora 17
1.2.1. Mecanismo de ramificación 17
1.2.2. Mecanismo de Interactividad Superpuesta 17
1.2.3. Simulaciones y Agentes Inteligentes 17
1.2.4. Mecanismo de Historias Emergentes 17
1.2.5. Drama Interactivo a través de Funciones Narrativas 18
1.3. Antecedentes y Contexto de los sistemas de HIC 18
1.4. Características del API para la Creación de HIC 20

2. *Teoría del Diseño de APIs con un Enfoque Orientado a Objetos (OO)* **23**
2.1. Definición de API 25
2.2. Pautas en el Diseño de APIs 26
2.2.1. El Usar el Nivel Adecuado de Abstracción 26
2.2.2. Usar el Menor Punto de Contacto Posible 26
2.2.3. Capacidad de Entregar un API libre de Implementación 26
2.2.4. Modelo Claro para el usuario 26
2.2.5. Soporte y Recuperación de Fallos 26
2.2.6. Condiciones de Fallo Claramente Delineadas y Reportadas 27
2.2.7. Presentar el API con un Modo Debug 27
2.3. Ventajas de la Utilización de APIs 27
2.4. El Paradigma Orientado a Objetos 28
2.4.1. Clases 28
2.4.2. Objetos 28
2.4.3. Atributos 28
2.4.4. Métodos 28
2.4.5. Conceptos de Programación Orientada a Objetos 29
2.4.5.1. Encapsulamiento 29
2.4.5.2. Herencia 29
2.4.5.3. Polimorfismo 29
2.5. Objetos como Gestores de Complejidad 29
2.6. Objetos como Gestores de Cambio 31
2.7. Tipos de Objeto y su Aplicación 32
2.7.1. Objetos de Servicios 32
2.7.2. Objetos Mensajeros 32
2.7.3. Objetos "Ligeros" 33
2.8. Uso de un Enfoque Orientado a Objetos para el Diseño de un API 34
2.9. Técnicas de Análisis para un Sistema Orientado a Objetos 35
2.9.1. Análisis de Casos de Utilización 35
2.9.2. Modelado Clase-Responsabilidad-Contribución 36
2.9.3. Identificación de Estructuras y Jerarquías 36
2.9.4. Definición de Subsistemas 37
2.9.5. 2.9.5. El Modelo Objeto-Relación 37
2.9.6. El Modelo Objeto-Comportamiento 38

3. *Necesidades de un API para Historias Interactivas por Computadora* **41**
3.1. Análisis de los Problemas del Desarrollo de Sistemas de HIC 45
3.1.1. Carencia de una Metodología de Diseño Común 45
3.1.2. Carencia de Base Común de Desarrollo 49
3.1.3. Carencia de un Nivel Adecuado de Interactividad 50
3.1.4. No Aplicación de los Principios del Drama 51
3.2. Características de un API para HIC 51
3.2.1. Diferenciación entre la Narrativa y el Mundo de la Historia 52
3.2.2. Posibilitar que las Acciones del Usuario Modifiquen los Eventos de la Historia 53
3.2.3. Capacidad del Autor de Controlar la Presentación de los Actos 54

3.3. Características de un API para Calakmul Virtual 55
3.3.1. Clase Mundo para Contener a Todas las Entidades de la Historia. 55
3.3.2. Clase Escena para Contener las Representaciones Gráficas 56
3.3.3. Presentar en un Ambiente 3D en Tiempo Real el Mundo de la Historia 56
3.3.4. Control Intuitivo y en Tiempo Real del Agente del Usuario 57
3.3.5. Soporte de un Lenguaje de Script que Permita el Control de los Objetos en el Mundo 57
3.3.6. Clase Cámara Controlable Desde el Lenguaje de Script 58
3.3.7. Clase Entidad que Permita el Uso del Polimorfismo con Todas las Clases de los Objetos del Mundo 59
3.3.8. Funcionalidad de Alto Nivel para el Control de los Personajes 60
3.3.9. Soporte de una Simulación Básica de la Física del Mundo 61
3.3.10. Soporte de Funcionalidades Básicas de Interfaz de Usuario 62
3.3.11. Funcionalidad Básica de Búsqueda y Seguimiento de Caminos 62
3.3.12. Soporte de Funcionalidad Básica de conversión de Texto a Voz 63
3.4. Necesidad de un Medio de Presentación Gráfica en la Narrativa Dramática en los Sistemas HIC 63

4. Análisis OO del API de Historias Interactivas. *65*
4.1. Introducción 67
4.2. Análisis del Dominio 67
4.2.1. Introducción 67
4.2.2. Definición del Dominio 68
4.2.3. Recolección de Aplicaciones Representativas del Dominio 68
4.2.4. Análisis de las Aplicaciones de la Muestra 70
4.2.5. Desarrollo de un Modelo de Análisis 71
4.3. Modelado de Clase-Responsabilidad-Contribución 71
4.4. Modelado Objeto-Relación 82
4.5. Modelado Objeto-Comportamiento 83

5. Diseño OO del API de Historias Interactivas. *89*
5.1. Introducción 91
5.2. Descripción de Objetos 92
5.3. Optimización y Detalle de los Métodos 95
5.4. Experiencia de Implementación 99

6. Aplicación y Trabajo Futuro *101*
6.1. Calakmul Virtual 103
6.1.1. Antecedentes 103
6.1.2. Funciones del Sistema Calakmul virtual y el API de HIC 104
6.1.3. Experiencia en el Desarrollo de Aplicación al Usar el API de HIC 104
6.2. Trabajo Futuro 105
6.2.1. Animación basado en Cinemática Inversa 105
6.2.2. Expresión Facial 106
6.2.3. El Servidor de Historias 106
6.3. Pruebas y Resultados 106

Conclusiones		*123*
Anexo A	*Documentación del API para HIC*	*127*
Anexo B	*Código Fuente del API para HIC*	*193*
Bibliografía		*229*

Índice de Figuras

Figura 4.1. Jerarquía de Clases de nIstEntity. 77
Figura 4.2. Jerarquía de clases de nVisNode. 78
Figura 4.3. Jerarquía de clases de nVoiceServer. 78
Figura 4.4. Jerarquía de clases adicionales. 79
Figura 4.5. Estructura del mundo de la historia. 79
Figura 4.6. Estructura de un modelo nCal3D. 80
Figura 4.7. Estructura de una escena. 80
Figura 4.8. Estructura de un material. 81
Figura 4.9. Estructura de un ente controlado por el usuario. 81
Figura 4.10. Modelado objeto-relación del sistema. 82
Figura 4.11. Diagrama de estados de la clase nIstCamera. 83
Figura 4.12. Diagrama de estados de la clase nIstCursor. 84
Figura 4.13. Diagrama de estados de la clase nIstObject. 84
Figura 4.14. Diagrama de estados de la clase nIstNPC. 85
Figura 4.15. Diagrama de estados de la clase nIstPlayer. 86
Figura 4.16. Diagrama de estados de la clase nIstWorld. 86
Figura 4.17. Diagrama de estados de la clase nVoiceServer. 87
Figura 4.18. Diagrama de estados de la clase nPath. 87
Figura 6.1. Secuencia de imágenes. Cámara estilo estacionario. 108
Figura 6.2. Secuencia de imágenes. Cámara estilo persecución. 109
Figura 6.3. Secuencia de imágenes. Cámara estilo persecución fija. 110
Figura 6.4. Secuencia de imágenes. Cámara estilo apuntando y siguiendo una ruta. 111

Figura 6.5. Secuencia de imágenes. Cámara estilo orientación predefinida y siguiendo ruta de la clase nPath. 112

Figura 6.6. Secuencia de imágenes. Cámara estilo primera persona. 113
Figura 6.7. Secuencia de imágenes. Cámara estilo libre. 114
Figura 6.8. Secuencia de imágenes. Animación cíclica correr 114
Figura 6.9. Secuencia de imágenes. Animación sencilla de disparo con arco 115
Figura 6.10. Secuencia de imágenes. Mezcla de animaciones: Disparar y caminar. 116
Figura 6.11. Secuencia de imágenes. Simulación de la gravedad. 118
Figura 6.12. Secuencia de imágenes. Simulación de la inercia. 119
Figura 6.13. Secuencia de imágenes. Simulación de las fuerzas de reacción en estructuras de la clase nIstStruct. 120
Figura 6.14. Secuencia de imágenes. Ejecución de un script activado por colisión. 121

Índice de Cuadros

Cuadro 4.1. Tarjeta de Índice CRC de nIstPlayer. 71
Cuadro 4.2. Tarjeta de Índice CRC de nIstNPC. 72
Cuadro 4.3. Tarjeta de Índice CRC de nIstObject. 72
Cuadro 4.4. Tarjeta de Índice CRC de nIstStruct. 73
Cuadro 4.5. Tarjeta de Índice CRC de nIstCamera. 73
Cuadro 4.6. Tarjeta de Índice CRC de nIstWorld. 74
Cuadro 4.7. Tarjeta de Índice CRC de nIstCursor. 74
Cuadro 4.8. Tarjeta de Índice CRC de nPath. 74
Cuadro 4.9. Tarjeta de Índice CRC de nVoiceServer. 75
Cuadro 4.10. Tarjeta de Índice CRC de nCal3DCoreModel. 75
Cuadro 4.11. Tarjeta de Índice CRC de nCal3DMaterial. 76
Cuadro 4.12. Tarjeta de Índice CRC de nCal3DModel. 76
Cuadro 4.13. Tarjeta de Índice CRC de nSceneGraph2 76
Cuadro 6.1. Error en las coordenadas de la clase nPath. 117

Resumen.

Capítulo 1. Historias Interactivas por Computadora (HIC).

El campo de las historias interactivas está surgiendo como un área de gran interés en la industria del entretenimiento digital y de la educación interactiva. Pero, lo reciente de este campo provoca que existan pocos ejemplos que realmente se puedan considerar como historias interactivas. El primer paso para entender lo que es, o mejor dicho, lo que debería ser una historia interactiva es presentar un marco teórico dentro del cual se va a trabajar. Parte fundamental de ese marco consiste en la definición de términos comunes y en la construcción de los conceptos básicos. Una historia es una sucesión de acciones que existen por sí mismas. De manera artística, generalmente, hay dos formas mediante las que se transmiten historias: La narrativa y el drama. El género de la narrativa literaria mas utilizado para contar historias es la novela, mientras que en el drama, casi todos sus géneros transmiten algún tipo de historia. Los tres géneros en los que se divide el drama son: la comedia, la tragedia y la farsa. En cuanto al significado de interactividad, se puede decir que es la capacidad del sistema para hacer que las acciones del usuario modifiquen a las acciones de la historia.

Capítulo 2. Teoría del Diseño de APIs con un Enfoque Orientado a Objetos (OO).

Se puede definir a un API (Interfaz de Programación de Aplicaciones, por sus siglas en inglés) como la capa mediante la cual el desarrollador de una aplicación puede interactuar, utilizando los medios proveídos por algún lenguaje de programación, con una

serie de rutinas, protocolos o herramientas para acceder a la funcionalidad de éstas. La ventaja de trabajar con un API es que el *software* tiende a ser mas robusto, ya que está pensado para ser reutilizado. Por otra parte, el paradigma orientado a objetos se basa en una noción muy fácil de asumir para la mayoría: Si todo en el mundo esta hecho de objetos, ¿por qué no aplicar esto al desarrollo de software? Es algo en lo que todos están familiarizados.

Capítulo 3. Necesidades de un API para Historias Interactivas por Computadora

Para desarrollar cualquier sistema de software, eso incluye a un API, es necesario saber que problemas tiene que resolver. Y en un área tan multidisciplinaria como las historias interactivas, es necesario establecer de manera muy clara que se requiere de cada clase. Uno de los problemas mas graves que aquejan a las historias interactivas es la falta de una base de software que provea de la funcionalidad requerida en la mayoría de los proyectos. La solución de este problema es el principal objetivo para el desarrollo del API de historias interactivas.

Capítulo 4. Análisis OO del API de Historias Interactivas.

El proceso de análisis orientado a objetos permite al desarrollador obtener la primera representación técnica del sistema. Con ella se desarrollan una serie de modelos que describen el software de computadora al trabajar para satisfacer una serie de requisitos. Y que en el caso del API, surgen de las necesidades encontradas en varias aplicaciones de historias interactivas, y en particular de los requisitos del proyecto Calakmul[1], realizado en el Laboratorio de Interfaces Inteligentes.

Capítulo 5. Diseño OO del API de Historias Interactivas.

El proceso de diseño OO tiene la intención de definir y optimizar las clases que se reconocieron en la fase de análisis, para dejar en claro lo que se desea del sistema, antes de empezar a codificar. En el caso del API, el diseño OO hizo posible detectar varios métodos candidatos a ser divididos debido a su complejidad.

Capítulo 6. Aplicación y Trabajo Futuro.

Los requisitos de la aplicación Calakmul virtual[1] jugaron un rol muy importante en la manera como se diseño este API, esto se debe a que esta aplicación ya estaba prevista de antemano para ser la primera en la que se utilice el API para su desarrollo. Es por esta razón que se presenta la experiencia que se ha tenido desarrollando esta aplicación utilizando el API desarrollado en este trabajo.

---

[1] Proyecto que está en desarrollo en este momento, y que se planea utilice el API que se desarrolló en este trabajo. Mas acerca de este proyecto en la sección 6.1.

Introducción

En la actualidad, el entretenimiento digital ha tomado gran relevancia en la sociedad, llegando incluso a competir con las tradicionales y poderosas industrias del entretenimiento, como el cine y la televisión. Por dar un ejemplo, el tamaño del mercado de los videojuegos en Estados Unidos, ya es mas grande que el de toda la industria cinematográfica de ese mismo país.

Dentro de las diferentes formas de entretenimiento y educación que existen, sin lugar a dudas la de escuchar historias o la de verlas representadas siempre han tenido un lugar preponderante. Ya que sin importar la forma en la que la historia nos es presentada, ya sea en un libro, una novela, una obra de teatro, una serie de televisión o una película, las historias tienen una significativa importancia social, educativa y cultural. Ya que en ellas se transmiten muchos valores y conocimientos a los individuos de cada sociedad.

Dentro de los aspectos altamente sociales que tiene el contar una historia, también se encuentran una serie de privilegios que se le dan al narrador. Por ejemplo, al narrador se le concede el derecho de tomar la palabra por un tiempo mucho mayor al tiempo de los que lo escuchan en ese momento. Otro privilegio del narrador es el contrato de verosimilitud con su audiencia. Y en donde no se le pide que cuente eventos veraces, sino verosímiles. Esto quiere decir que se le pide que cuente eventos que sean creíbles dentro del marco de la historia. Por esto motivos se dice que la narración de una historia es un evento eminentemente social, donde él que cuenta la historia es normalmente una persona que se siente con un alto estatus entre el grupo de personas, ya que será necesario que logre

mantener la atención y el interés de la audiencia por un periodo de tiempo prolongado. Para consultar una análisis mas profundo acerca de la importancia social del narrador de historias, consultar [LAW01]. Además, hay otro aspecto social y cultural muy importante al contar historias, que gran parte de la identidad cultural y de los valores de cualquier sociedad son transmitidos a los individuos de ésta mediante historias contadas ya sea de manera oral, escrita o actuada en obras. Estas historias relacionan al individuo con los mitos fundacionales y con los conceptos morales comunes a cada cultura. Finalmente, las historias son usadas como factor educativo, debido a que a través de ellas se pueden transmitir a los individuos de una sociedad los valores, conceptos morales y reflexiones de una manera sencilla, práctica y a su alcance; sin tener que usar acercamientos demasiado teóricos.

El gran incremento del poder de procesamiento de las computadoras, así como de su capacidad gráfica, ha propiciado la aplicación de éstas en una gran variedad de campos. En particular, el entretenimiento y la educación son áreas que han recibido una gran atención de aquellos que buscan nuevas aplicaciones a las computadoras. Y al considerar el importante papel que tienen las historias en cualquier sociedad, tanto al entretener como al educar, es que se ha considerado utilizar las capacidades de la computadora para la creación de sistemas que presenten *historias interactivas*. Esto quiere decir, historias cuyos eventos pueden ser modificados por las acciones de él o las personas que la presencian.

En estos momentos, el campo de las Historias Interactivas en Computadora (HIC como se presentará de aquí en adelante) es un campo incipiente, sin lugar a dudas con muchos de los problemas que comúnmente enfrentan este tipo de áreas. Algunos de estos problemas son una base teórica que apenas se esta desarrollando, una falta de metodologías de diseño establecidas y sobre todo la falta de una base de aplicaciones ya construidas, que permitan el aumento de la experiencia general en este campo. Esto trae como consecuencia que la mayoría de las veces se utilicen modelos que apenas se están desarrollando, sin que ninguno satisfaga del todo las cualidades dramáticas e interactivas que se desearían en una historia de calidad.

Una rápida revisión de los mecanismos hasta ahora usados para contar historias interactivas muestra tres mecanismos básicos para implementar la interactividad, de los cuales los sistemas de HIC actuales aplican uno o varios. El primero es la ramificación de la historia (conocido en inglés como *branching*) donde el usuario se enfrenta a diferentes opciones de acción en un momento dado, y la narrativa cambia de acuerdo con esas elecciones. El segundo es la superposición de la interactividad a una historia secuencial. A lo largo de un desarrollo secuencial de una historia, una escena muy interactiva aparece, ésta puede ser una pelea, una acertijo, etc. Pero que finalmente, aunque la escena sea muy interactiva, ésta se constituye como una capa superpuesta a la narrativa en sí misma, ya que ésta no tiene ningún efecto en el desarrollo de la historia. Por último, esta el modelo de simulación, que es altamente interactivo, en el sentido de que las acciones tienen impacto real en el curso de los eventos, pero que tiene el problema de que la sucesión de eventos que emerge de la interacción no es realmente una historia.

La posibilidad de que surjan nuevos modelos de HIC que corrijan las deficiencias de los modelos arriba mencionados es complicada. Ya que el desarrollo de estos sistemas toma tiempo, además de que no es fácil para el investigador especializado en la narrativa con pocos o nulos conocimientos de programación hacer su aportación en el desarrollo de las HIC. De igual manera, para el investigador especializado en las ciencias de la computación o en la ingeniería, el trabajo que se requiere para crear un sistema interactivo que aproveche de manera digna los recursos gráficos que las computadoras ofrecen, no es una tarea menor. Además de que muchas veces carece de los conocimientos necesarios para crear historias o sistemas que generen historias con las características dramáticas suficientes para interesar al usuario y mantener su atención.

Por estas razones, el objetivo de este trabajo de tesis es el desarrollo de una Interfaz de Programación de Aplicaciones (API por sus siglas en inglés) que permita la creación sencilla de HIC, de manera que en lo posible, se eliminen los problemas de programación y de desarrollo que inhiben la investigación y creación de HIC con mayor interactividad, así como con una mejor calidad dramática en la presentación. Poniendo énfasis en su desarrollo mediante la aplicación de la ingeniería de software. Esto con el propósito de crear un API robusto y capaz de satisfacer los requisitos básicos de un desarrollador de historias interactivas.

De la misma manera que el API para gráficas por computadora OpenGL provocó el aceleramiento del desarrollo de toda la industria de las tarjetas gráficas aceleradoras. Con este sencillo API se espera, aunque sea de manera particular y local, que propicie el desarrollo del campo de las HIC. Además, de manera particular se espera que permita desarrollar de manera mas rápida los proyectos del Laboratorio de Interfaces Inteligentes. En especial el proyecto Calakmul virtual[1].

Y como para crear HIC se requiere de cocimientos en diversos campos de las ciencias de la computación, como por ejemplo: el de la inteligencia artificial, el de la graficación por computadora o el de la animación y el modelado del cuerpo humano; así como de conocimientos en la narrativa y en el drama; se intenta que el API contenga las herramientas mas básicas para que los interesados en desarrollar HIC puedan probar sus teorías y algoritmos en un periodo de desarrollo mas corto. Por ejemplo, quienes trabajan en el área de la inteligencia artificial, se espera que puedan probar de manera sencilla e inmediata sus algoritmos, en un ambiente con calidad gráfica cercana a la de los sistemas de punta. De la misma manera se espera que los desarrolladores con antecedentes en el área de la literatura, que estén interesados en desarrollar sistemas generadores de historias dramáticas, tengan un conjunto mínimo de herramientas de inteligencia artificial que les permita probar sus ideas, sin que tengan volver a inventar la rueda desarrollando todo desde cero.

[1] Proyecto que está en desarrollo en este momento, y que se planea utilice el API que se desarrolló en este trabajo. Mas acerca de este proyecto en la sección 6.1.

Organización.

En el capítulo 1, se realiza un análisis mas profundo al aquí presentado, de los diversos modelos de interactividad y la forma como éstos se implementan. Además, se presentan los conceptos básicos relacionados con las HIC que se usarán a lo largo de toda la tesis. En especial en este capítulo, se hace una definición de los términos relacionados con la narrativa y el drama que aparecen a lo largo de este trabajo.

En el capítulo 2, se realiza un análisis de las diferentes opciones de implementación de APIs que se tienen, además de una ponderación acerca de las ventajas y desventajas de las mismas. Se hace una presentación enfatizando a los APIs desarrollados mediante el enfoque orientado a objetos, los conceptos básicos de éstos, así como los diferentes patrones de diseño que se podrían aplicar.

En el capítulo 3, se analizan las necesidades particulares de un API para HIC. Se analizan las características básicas que un API para HIC debe tener para ser útil, así como en que características debe ser flexible para facilitar la inclusión de nuevas funciones. Se revisan los problemas mas comunes a los que se enfrenta un desarrollador de HICs y se proponen las características del API que las resuelvan. También contiene una breve discurso acerca de la necesidad de representaciones visuales para una narrativa dramática.

En el capítulo 4, se realiza el análisis orientado a objetos (OO) del API para la creación de HIC, se explica que clases constituyen al API y las relaciones entre ellas. Se muestra como desde el análisis se tiene en mente la solución de los problemas que aquejan desarrollo de HICs, así como que mecanismo de interactividad será el mas favorecido por el API.

En el capítulo 5, se presenta el diseño OO de las entidades con estado interno, así como de las respuesta mediante "scripts", que son partes fundamentales de las herramientas contenidas en el API. Se realiza una descripción detallada de su diseño, de sus alcances y limitaciones, así como de su fundamento teórico.

En el capítulo 6, se comentan las experiencias que se tuvieron en la implementación del API, como se hizo para satisfacer todas las necesidades que estaban marcadas en el diseño. También se da un resumen acerca de la aplicación del API en el proyecto Calakmul virtual. Además en este capítulo se hace una breve mención de los posibles trabajos futuros a realizar al API.

Finalmente, se espera que el API que se desarrolló sea factor en la investigación y desarrollo de las HIC, de manera que se propicie la creación de éstas con mayor calidad dramática e interactiva.

Capítulo 1

Historias Interactivas por Computadora

1.1. Historias Interactivas.

Para hablar de historias interactivas, primero se tienen que hacer las definiciones de los términos que se emplearán de manera recurrente. Además, se tiene que delimitar las áreas de la ciencia bajo las cuales se estudiará este campo y bajo que enfoque se hará. La primera definición es el concepto de historia, listando sus características y la manera en que ésta puede ser presentada al espectador.

1.1.1. La historia.

La historia es la sucesión de acciones o eventos que existen por sí mismos, o sea, son eventos que no tienen que ser narrados o actuados para existir. Son los acontecimientos que acometen a los personajes de la historia [GAR88]. Además estos acontecimientos se suceden de manera verosímil a lo largo de toda la historia. Es decir, que son acontecimientos que parecen reales y que son creíbles en contexto con los otros acontecimientos, aunque no sean veraces. Esto queda explicado es una cita que Aristóteles, en Poética, le atribuye a Homero: "decir cosas falsas como es debido" y concluye que: "se debe de preferir lo imposible verosímil a lo posible creíble".

Cuando se habla de que los eventos de la historia existen por sí mismos, quiere decir que la historia incluye todas las acciones, aun las que no son explícitamente contadas. También implica que la historia existe aún cuando no es contada, dando lugar a nociones como las del "mundo" de la historia. Es en este supuesto "mundo" es donde se suceden las acciones de la historia y donde existen los personajes. Esta característica particular de las historias es la que comúnmente se expresa mediante la frase "la historia vive por sí misma". Por eso es de gran importancia el entender la diferencia entre la historia en sí, y la forma en que ésta es comunicada al publico o lector. Además, la historia es orgánica, o sea, que se trata de una estructura entera y completa.

Después de definir el término historia, que se empleara a lo largo de todo este trabajo, es necesario explicar las formas como ésta es transmitida. Las obras artísticas mediante las cuales se puede transmitir una historia principalmente pertenecen a la literatura y al drama. Dentro de la literatura, el texto narrativo literario es, particularmente, capaz de transmitir una historia. Mientras que en el Drama, generalmente, toda obra de una u otra manera tiene una historia, entendida ésta como acciones que se suceden. Pues en griego antiguo, drama literalmente significa acción. Como siguiente paso, debido a lo presentado sobre las formas de transmisión de las historias, es necesario definir al drama y a la narración, además profundizar en la función de sus componentes y en sus tipologías.

1.1.2. La narración.

La narración es una expresión de tipo artístico y social que relata los eventos que suceden en una historia de un modo interesante y no completo [GAR88]. Los eventos se cuentan de manera no completa porque, al presentarse, su disposición secuencial no siempre corresponde a la disposición con la que éstos se dieron en el mundo de la historia. Además, no todos los eventos que ocurren en la historia son siempre presentados. Esto con la intención de darle mayor suspenso e interés a la narración. La narración implica el concepto de narrador, que es quien transmite la historia. Y que en el caso de la comunicación oral se trata de la persona que realmente nos cuenta los eventos de la historia, mientras que en el caso de un texto narrativo, se trata de la instancia intermedia entre el autor del texto y la materia novelística.

Como parte del análisis de la narración es importante reflexionar sobre la función del narrador y sobre la función del personaje. La función del narrador es la de "representación", combinada con la de "control" o dirección: ya que controla la estructura textual, en el sentido de que es capaz de citar el discurso de los personajes en el interior de su propio discurso. Además de estas dos funciones obligatorias, el narrador es libre de ejercer una función opcional de "interpretación", es decir, de manifestar su opinión, ideología o parecer con respecto a lo que sucede en la historia. Las funciones primarias y obligatorias en el caso de los personaje son el actuar y el interpretar. Aunque, estas funciones pueden unirse en el caso de que uno o varios personajes de la historia sean el narrador.

Las tipologías narrativas, clasificaciones utilizadas en el análisis literario, se definen según en que aspecto se basan para hacer la categorización de las distintas obras. En el caso de la tipología que se va a definir a continuación se concentra el la relación entre narrador y destinatario, en sus relaciones con el relato y con la historia. La dicotomía del narrador y del actor permite, en principio, establecer dos narrativas fundamentales: la *narración heterodiegética* y la *narración homodiegética*.

La narración es heterodiegética si el narrador no figura en la historia (diegesis) como personaje. Dentro de este tipo de narrativa, hay una importante tipología, que toma como punto de estudio la profundidad de la perspectiva narrativa en relación con lo percibido. Esta tipología distingue tres tipos narrativos: el autorial, el actorial y el neutro. En el tipo *autorial* el narrador tiene una percepción externa e interna ilimitada. En el tipo *actorial* la percepción externa del narrador adopta la perspectiva de un actor, y esta limitada por la extrospección de ese actor mismo. De la misma manera para las percepciones internas, el narrador está limitado con la introspección de ese actor. En el *neutro* la percepción externa también es limitada, pero del mismo modo que un registro de cámara. Solo se registra el mundo perceptible, por lo que se hace imposible una percepción interna.

La narración es homodiegética (o en mucho casos conocida como en primera persona) si las funciones del narrador y del actor están desempeñadas por el mismo personaje. Cabe remarcar que la importancia del papel que desempeña en la historia el personaje que es el narrador puede ser el protagonista o un puede ser un simple papel que le permite ser testigo de los acontecimientos. En este tipo de narración, también, según si el personaje narrador o

si el personaje actor es el centro de orientación del lector, se distingue en la narración homodiegética el tipo narrativo autorial y el tipo narrativo actorial.

Con esta pequeña presentación de la narrativa y de sus tipologías se espera que el lector esté al tanto acerca de las formas textuales mediante las que se puede trasmitir una historia. Sobre todo, en consideración con el tema de esta tesis, al ponderar el utilizar este arte como método para presentar historias interactivas. Para profundizar mas acerca de la narrativa, revisar el texto [GAR88].

1.1.3. El drama.

El drama es un arte en el que los eventos de la historia son presentados de manera directa al espectador. De esta primer definición se puede extraer que el drama necesita de la presencia de una audiencia. También se entiende que el drama no cuenta una historia a una audiencia, la historia es interpretada ante una audiencia por un grupo de actores. Por lo tanto el drama nunca puede ser tomado como un mero trabajo de literatura escrita o impresa. Y aunque estas características diferencian al arte dramático, es necesario apuntar que éstas simplemente indican la forma externa y la circunstancia necesarias para la presentación dramática. Para realizar una definición mas completa es necesario presentar también algunas características internas que puedan relacionar a todo lo "dramático". Aunque es muy prudente mencionar que, una definición de drama que se considere completa y que sea universalmente aceptada no existe. Por lo tanto es necesario el buscar una definición mediante aproximaciones sucesivas, en las que se presenten las diferentes teorías que existen sobre el drama.

Teoría de la imitación. En esta antigua teoría que se puede rastrear hasta tiempos de Cicerón, el drama es una copia de la vida, un espejo en el que se refleja y concentra la verdad. Esta definición, si es que así puede llamarse, ha sido utilizada y citada muchas veces a lo largo de la historia, ha sido la base de innumerables disquisiciones, particularmente en el renacimiento. Aun en tiempos relativamente modernos, esta teoría ha encontrado quien la sustente, los realistas del siglo diecinueve. El problema es que si se toma esta concepción en su mas estricta y cruda interpretación, entonces el drama sería un simple extracto de la vida. Es decir, que la meta de cualquier dramaturgo sería proveer en el escenario de una replica lo mas cercana a lo que en realidad ocurrió o hacerlo en tales términos que parezca real. El realismo total es imposible en una obra, simplemente sabemos que el público no cree que este viendo algo real. Además, en primer lugar, hay obras donde las situaciones están totalmente alejadas de la realidad. Concluyendo, se puede utilizar el sentido mas amplio de imitación, el drama imita a la vida sólo en el sentido de proveer un tipo de semblanza de la existencia humana en el escenario.

Teoría del conflicto. La noción de conflicto es muchas veces descrita como el principal mecanismo del drama (Jenn 1991). En especial esta noción de conflicto ocurre cuando un persona o personas de la obra son puestas, consiente o inconscientemente, en contra de algún personaje antagonista, de alguna circunstancia o del destino. Un caso particular común, es cuando alguna posible acción para un personaje no es compatible con sus valores. En esta teoría, el drama esta centrado sobre este conflicto principal. Y que según su resultado, será la transformación desde el estado inicial al final de la historia. Un primer

punto es que la característica de conflicto no es privativa del genero dramático únicamente, por lo que no sirve de característica peculiar que nos permita definir al drama. Aún así, es necesario aclarar, que la noción de conflicto puede indicar lo que mas nos atrae de la mayoría de los grandes dramas, pero no sirve para delimitar el genero dramático.

Teoría moderna del Drama. Después de presentar las dos teorías anteriores, donde se indican los aciertos así como los limites de su alcance y sus errores, es obvio que no se puede llegar una definición de drama completa y en la que no se hallen excepciones. Ya que en estas áreas del conocimiento, los limites normalmente son muy difusos. Los enfoques modernos para una teoría del drama [ESS95] se aproximan a este problema al buscar las características que hacen diferente al arte dramático de otras artes.

En este caso, un primer punto de coincidencia con la teoría de la imitación en su sentido amplio, es el hecho de que el drama expresa una semblanza con la existencia del ser humano. De igual manera, esta aproximación acepta que es necesario la existencia de un público y de un grupo de actores para que exista el drama. Pero al ser una teoría actual, influenciada por la existencia y preponderancia de los medio electrónicos, no solo limita al drama a las obras presentadas en un teatro donde un grupo de actores presenta la obra sobre un escenario ante un público que se tomo la molestia de ir al teatro donde la obra se presentaba para verla. En esta teoría también se incluye a el drama presentado en medios masivos de comunicación, la radio y la televisión, al igual que al cine y para los propósitos de este trabajo, también se incluirán los espectáculos multimedia y los sistemas interactivos en computadora.

Entonces, para definir al drama es necesario encontrar las características que lo diferencian de otras artes. Tomando esta regla para buscar la definición del drama se puede ver que el conflicto no se puede utilizar para definirlo, ya que el conflicto está presente en la mayoría del drama, pero también en otras formas artísticas, como la ficción narrativa. En este caso, un punto que resalta de inmediato, es la manera en la que el drama puede expresar un pensamiento de manera económica, al consumir menos tiempo y al hacerlo de la forma mas elegante. Y como dice el principio de la navaja de Occam, algo que exprese un pensamiento de esta forma será mas cercano a la realidad. Como ejemplo, imagínese todas las relaciones e interacciones que hay en el dialogo de una obra entre los personajes, el tono de sus voces, sus gestos, el modo que reaccionan él uno con el otro. Todos estos son aspectos que abarcan mucho mas allá del simple dialogo; y que para transmitir mediante el lenguaje escrito hubiera requerido de descripción profunda del carácter de los personajes, de los tonos de su voz, además, de que se requeriría de especial maestría para transmitir todas las sutilezas que el cuerpo humano puede transmitir. Y aún con esa maestría, simplemente por las limitaciones del medio escrito, no se podría lograr el efecto que puede tener una escena. Simplemente porque todos los elementos visuales, el ambiente donde toma lugar la acción, son inmediatamente comunicados por el escenario, (esto aplica igual para los dramas de televisión y para el cine). Mientras que en una novela, el autor debe de describir secuencialmente cada aspecto del lugar, nunca pudiendo lograr una presentación inmediata que abarque todo el escenario.

En cuanto a la *estructura del drama*, la creación de interés y suspenso yace bajo toda construcción dramática. Con esto, podemos ver que lo propuesto en la teoría de conflicto,

en parte, tiene de razón. La noción de conflicto puede estar presente en el drama, en cuanto a su carácter creador de suspenso, ya que siempre esperáremos con interés el resultado de ese conflicto de carácter bipolar. Pero puede verse que no es lo único capaz de mantener en suspenso al público: la sorpresa, el sobresalto y la extrañeza son otras formas de crear suspenso dentro de una obra. En palabras sencillas, se puede decir que el aspecto mas primitivo y mundano de la estructura del drama es el despertar y el mantener la atención del público a través de la expectativa, el interés y el suspenso. Dos aspectos mas a considerar dentro de la estructura que todo drama debe de seguir son: el evitar la monotonía y la clara indicación del progreso de la historia. Como fácilmente se puede ver, estos aspectos también buscan el mantener la atención y el aumentar el interés del público.

Es importante remarcar, que en concordancia con puntos anteriores donde se mostraron las características que permiten diferenciar al drama de otras artes, la estructura del drama también esta marcada por una característica que lo diferencia: la necesidad de mantener la atención del público. Simplemente considérese el caso de la novela, en ella el lector puede dejar de leer en el momento en que se sienta cansado y continuar su lectura cuando lo desee, el autor no tiene la necesidad de mantener la atención del lector a lo largo de toda la obra. En cambio, en una obra de teatro o en una película el público tomó la decisión de asistir a un lugar y hora determinada y está dispuesto a gastar unas horas de su tiempo para presenciar el evento. En el caso de la televisión y el radio, el público tiene que sintonizar un canal o estación a una cierta hora para poder presenciar la obra, además de que si esta no le interesa lo suficiente, simplemente puede cambiar de estación. Por lo tanto, en la estructura misma del drama debe de estar la intención de mantener la atención del público a través de los medios ya mencionados. Y en el caso del teatro, la estructura no sólo toma en cuenta el tiempo que es posible mantener la atención del público, sino que tiene que considerar las capacidades físicas de los actores para poder presentar esa obra noche a noche. En esta reflexión acerca de la estructura del drama, en donde se ha analizado considerando el enfoque de cada medio; así pues se puede aplicar un enfoque especial al drama interactivo en computadora. Ya que en este medio, aunque también es necesario interesar y mantener la atención del usuario, hay una característica de sistemas que puede cambiar en mucho la forma como los autores tradicionales de drama estructuran su obra. Los sistemas mediante los que esta historias se presentan, normalmente, permiten salvar en el estado de ésta en el disco duro, esto con el fin de permitir que la sesión se continué en otro momento. Esta característica sin lugar a dudas, dificulta el uso de la estructura tradicional del drama, ya que obliga al autor a concebir nuevas formas en las que se presenten los momentos de suspenso. Además esto le presenta al desarrollador un reto dentro del campo puramente técnico, debe evitar que esos momentos de salvar el estado del sistema puedan romper con la estructura de la historia y hacer que se pierda el suspenso.

La tragedia y la comedia. Los términos más frecuentemente usados en el vocabulario del drama son aquellos que denotan los diferentes géneros del drama. Sobre todo los dos géneros básicos: la tragedia y la comedia. Una inmensa cantidad de debate y de especulación existe en torno a este tema, además de que estos conceptos teóricos han ejercido una profunda influencia el la practica actual de escribir obras, actuarlas y producirlas. Y aún así, de manera sorprendente, no hay un consenso sobre todo esto, no hay una definición aceptable y que sea generalmente aceptada.

Aunque, hay que ser muy claros en donde radica la importancia de definir de manera clara los géneros del drama. La importancia es de carácter práctico mas que teórico. Un director que se encuentra en la etapa de preparación tiene que tomar una decisión acerca del genero al que pertenece la obra. No por algún principio abstracto, sino simplemente desde el punto de vista de cómo será actuada. Esta necesidad práctica, sin lugar a dudas también existe para los desarrolladores de dramas interactivos. Además esta necesidad recae en un punto sensible de los sistemas interactivos actuales, y es que el desarrollo de los actores virtuales todavía se encuentra en etapas iniciales, no se puede decir otra cosa más que su expresividad todavía está muy limitada.

Una división de los modos literarios, propuesta por Northorp Frye, puede arrojar un poco mas de luz acerca de la definición de tragedia y comedia, así como de sus similitudes y diferencias. Esta división toma como característica fundamental la relación entre los lectores y los personajes. En un mundo mítico y heroico los lectores admiran a los personajes como dioses o grandes hombres; en un modo realista se ven a sí mismos en el nivel de los personajes; en un modo irónico se sienten superiores a ellos y los miran con mofa o desprecio. Trasladando esas divisiones al drama podemos ver que los personajes de la *tragedia* son dioses, héroes o grandes hombres, y son admirados por el público. Los personajes del drama realista y de la comedia son vistos al nivel del público. Mientras que en la farsa, otro género del drama, los personajes son definitivamente vistos hacía abajo. En especial, tiene sentido plantear que el género de un drama se define mediante la relación del público con los personajes, cuando hay obras que se pueden catalogar como tragedia o comedia dependiendo de la presentación y la manera en la que los personajes actúan. Como ejemplo de esta tipo de obra está *Los Huertos de Cerezos* de Chekhov. Al realizar este análisis acerca de los géneros del drama, se encuentra otro aspecto donde el drama interactivo en computadora tiene que desarrollarse. En la capacidad de sus actores virtuales de comprometerse con el público mediante su presencia, sus gestos, su pose, su tono de voz, sus expresiones, etc. En especial a la luz de que será mediante estos aspectos que el público sabrá si la obra se trata de una tragedia o de una comedia. Debido a las actuales limitaciones en el campo de la animación, de la generación de expresiones faciales en modelos 3D, muchas veces los actores virtuales solventan sus deficiencias de expresividad con gestos, poses y movimientos exagerados que hace que su expresión se balancee en una línea muy delgada: entre realizar su cometido de hacer que el público se comprometa con el personaje y de volver al actor virtual en un ente que el público solo mira hacia abajo, volviéndolo apto únicamente para la farsa de pastelazo.

En la tarea de encontrar si la diferencia principal entre tragedia y comedia está en la manera como el público mira a los personajes de la historia, Freud la apoya al proponer que la risa es causada por la liberación de ansiedad que surge de darnos cuenta que el infortunio que vemos venir no nos afecta directamente. Reforzando la idea de que cuando vemos obras donde pasan situaciones trágicas, dependerá de cómo veamos a los personajes. Puesto que, si por la manera como actúan y se presentan nos hacen verlos como inferiores; al ver que alguno de ellos le pasa una desgracia esto nos causará risa. Ya que el público sentirá que eso nunca les podrá pasar a ellos.

En cambio, la tragedia busca provocar la catarsis en su público. En general, después de ver una gran tragedia uno se siente exaltado porque ha visto seres humanos superiores encarar la adversidad y el infortunio de manera noble, con valentía y dignidad.

Otro aspecto que puede agregarse al análisis de la tragedia y la comedia es la cantidad de conocimiento que el público tiene sobre lo que va a pasar comparado con lo que saben los personajes. En una comedia el público normalmente se le hace saber mas de lo que pasa que a los personajes, reforzando el sentimiento de superioridad sobre ellos. Así cuando algún personaje trata de tomar alguna decisión de la que el público ya conoce sus consecuencias, la reacción esperada es que éste se ría, por funesta que la consecuencia sea, ya que el público se da cuenta que eso no le podía pasar al él porque, de antemano, conocía las consecuencias. En el drama en cambio, el público sabe menos que los personajes acerca de lo que va a pasar. Con esto el suspenso, la tensión y la expectación sobre el futuro aumentan. En especial, si los personajes lograron comprometerse con el público. Como ejemplo de esto, está la obra *Tartuffe* de Moliere. Donde el público se ríe de la manera en la que Tartuffe engaña a Orgon, aunque esto implique la ruina de un buen hombre y de su familia. Simplemente porque el público se siente superior a Orgon, ya que el autor hace que Orgon sea ciego a la hipocresía de Tartuffe , mientras que el público claramente conoce de sus malvadas intenciones. Para profundizar mas acerca de la naturaleza del drama y de su teoría consultar [ESS95] y [NIC80].

1.1.4. La interactividad.

En cuanto a la noción de interactividad, el otro concepto importante n este trabajo, ésta hace referencia a la capacidad que se le da al espectador de cambiar el curso de las acciones de la historia, de manera que éste tiene un rol activo, como un agente libre que puede actuar de manera trascendente y concordante con el mundo de la historia.

El grado y la manera en la que el publico puede interactuar es variado, y se analiza mas adelante en este trabajo. Además de que la forma de interactuar será muy dependiente de que método se utilice para transmitir la mencionada historia.

1.1.5. Definición de Historia Interactiva.

Después de plantear estos conceptos base, se definirá un sistema de Historias Interactivas por Computadora; Se trata de un sistema de *software*, que normalmente se ejecuta en una computadora personal, que contiene el mundo virtual donde están todos los objetos y personajes que toman parte en la historia. El sistema de *software* de Historias Interactivas simula la evolución de los eventos que forman parte de la historia, siguiendo diversos mecanismos que se muestran en [1.2.], modificando la sucesión de éstos de acuerdo con las entradas que el usuario proporciona y la forma en la que el desarrollador del sistema programó que las cambiaran. El usuario del sistema puede participar como un solo personaje en la historia y por ende controlándolo, o como un ente externo a la historia que puede ver los eventos y tiene la capacidad de controlar o influenciar a varios o a todos los personajes de la historia. Una parte muy importante de un sistema de Historias

Interactivas por Computadora (HIC), es aquélla que presenta al usuario los eventos que se dan en el mundo de la historia que son trascendentes para el desarrollo de la misma y para mantener la atención del usuario. Por lo tanto se puede decir que esta parte del sistema se trata de un narrador, un escenario o una pantalla de cine virtual; dependiendo de la forma en que se planee transmitir la historia interactiva. La complejidad de esta parte del sistema puede ser muy variada, al igual que la manera en la que presenta los eventos.

Según la manera en que se decida presentar la historia, esta parte del sistema puede presentar simples salidas de texto, que describen los eventos así como los personajes y los lugares donde éstos toman lugar; como un texto narrativo interactivo. Puede narrarnos la historia mediante *software* de conversión de texto a voz y un sistema de generación de lenguaje natural, dando lugar a un narrador virtual. O puede mostrarnos los eventos en tiempo real, mediante actores virtuales que son generados a través de algún motor gráfico, dando lugar a un drama interactivo.

1.1.6. Drama o Narrativa Interactiva.

Después de haber definido los términos de drama y narrativa, además de haber profundizado en cada una de estas artes, se puede pasar a hacer un pequeño análisis que liste las características del drama y la narrativa interactiva, así como de algunos de sus pros y contras.

Por el lado de la narrativa interactiva hay dos áreas principales: la que utilizaría la generación de voz para implementar un "cuenta cuentos virtual" y la que mediante salidas de texto crearía una "novela interactiva". Del lado del drama interactivo, la implementación principal sería un sistema que mediante gráficas 3D presentara actores virtuales, mediante modelos humanoides animados, llevando acabo su representación en un escenario. Los diálogos de estos actores puede ser transmitido mediante salidas de texto, sistemas de síntesis de voz o diálogos pregrabados. Por el lado de la expresividad de estos actores, las poses, los gestos, los movimientos y las expresiones que éstos realizan pueden ir desde el rango abstracto y poco realista, hasta niveles de gran realismo y detalle. Claro que con su costo en poder de procesamiento y en complejidad del sistema.

El área de aplicación de cada tipo de sistema de historias interactivas, al igual que su implementación. es diferente. En el caso del "cuenta cuentos virtual" se puede aplicar en el área educativa, en especial con los niños pequeños. Esto porque la posibilidad de interacción con esos sistemas es reducida. Los sistemas de "novela interactiva" también pueden utilizarse para el área educativa, pero con aplicación para todas las edades. Además se puede utilizar en la investigación en el área de generación de historias de manera procedual. También pueden tener un campo de aplicación en ciertos nichos del campo del entretenimiento. Los sistemas de "drama interactivo" tienen un área de aplicación mas grande. Pues pueden usarse en la educación, el entretenimiento, la realidad virtual y la recreación de lugares y hecho históricos para museos y escuelas.

1.2. Mecanismos de Interactividad en el Drama Interactivo por Computadora.

Como se menciono en [1.1.], los sistemas de HIC pueden modificar los eventos de la historia de diversas formas, dependiendo de que mecanismo de interactividad utilicen, cuando reciben la entrada del usuario. A continuación se presentan los mecanismos mas usados para dar la interactividad a las HIC, incluyendo los comúnmente usados en la industria del entretenimiento, así como los que apenas se están desarrollando en el ámbito académico.

1.2.1. Mecanismo de Ramificación.

En este tipo de mecanismo de interactividad, el usuario se le presentan puntos a lo largo del desarrollo de la historia en los que debe de elegir que acción tomar, la narración es diferente de acuerdo con las elecciones del usuario. Para explicarlo de manera sencilla, este tipo de sistemas son como los libros infantiles, en los que dependiendo de que acción quiere tomar el usuario, se salta a diferentes páginas donde las historias continúan de manera separada. El problema que surge con este tipo de mecanismo, es que todo camino que tome la historia debe ser diseñado cuidadosamente por el autor, por lo tanto, cancelando cualquier intento de ampliar las opciones de elección para lograr la apariencia de que el usuario es un agente libre en el mundo de la historia, ya que cualquier intento de cubrir todas las posibles ramificaciones que generan las acciones del usuario implicaría un enorme trabajo para el autor.

1.2.2. Mecanismo de Interactividad Superpuesta.

En este mecanismo la historia se desarrolla a través de una sucesión lineal de eventos, en los que solo se interponen escenas interactivas en las que las acciones tienen poco o ningún efecto sobre el desarrollo de la historia; por ejemplo unas escena de batalla en la que la historia no continua hasta que el usuario es capaz de ganar. En este tipo de sistemas el autor de la historia tiene la ventaja de que puede profundizar en una narrativa única, dando la posibilidad al desarrollo de historias con una gran calidad dramática, pero en realidad el carácter interactivo del sistema solo es una capa superpuesta a la narrativa misma.

1.2.3. Simulaciones y Agentes Inteligentes.

Este tipo de sistemas están formados por agentes controlados mediante mecanismos de inteligencia artificial, que modelan a los personajes, objetos y el ambiente que conforman el mundo en el que se desarrolla la historia, el usuario puede interactuar con los agentes que conforman a los personajes de la historia, y éstos responden de manera reactiva a las acciones del usuario, a veces hasta tomando algunas acciones de manera activa. El problema con este tipo de sistemas es que la historia, si es que existe, no contiene los elementos fundamentales del drama, como son la transformación y el conflicto, ya que se trata de sistemas donde los agentes están programados para responder sin tomar en cuanta ningún tipo de consideración dramática de la historia.

1.2.4. Mecanismo de Historias Emergentes.

Este es un mecanismo que se puede considerar parecido al anterior, ya que también esta basado en agentes inteligentes y no hay una planeación de la historia por el autor. En lo que se diferencian es que aquí se busca agregar a los agentes ciertas tendencias que produzcan la "emergencia" de la historia. a partir de las acciones del usuario y de las respuestas de los

agentes. Cabe mencionar que este es un campo en pleno desarrollo, por lo que un análisis del estado de desarrollo este tipo de mecanismo es todavía prematuro.

1.2.5. Drama Interactivo a través de Funciones Narrativas.

Este tipo de sistema de HIC esta también en pleno desarrollo, y esta basado en los estudios narrativos de Vladimir Propp [PRO58], donde a los personajes de una historia se les asignan diferentes funciones que tienen que cumplir, y en donde los agentes que son los personajes de la historia toman sus acciones no debido a algoritmos de inteligencia artificial, sino por la causalidad de la narrativa como un todo. Esto quiere decir que el mayor peso de la inteligencia artificial no se pone en los agentes que controlan a los personajes de la historia, sino en la parte del sistema que sirve de narrador. Uno de los principales impulsores de los sistemas basados en funciones narrativas es Nicolas Szilas. Para consultar mas acerca de sus propuestas véase [SZI99].

Este listado de mecanismos de interactividad para HIC, no es completo, ya que hay sistemas que utilizan variaciones a los modelos presentados o sistemas que combinan dos o mas estos modelos. Aún así, se puede considerar que esta lista cubre las áreas mas representativas del desarrollo del mecanismo de interactividad en las HIC.

1.3. Antecedentes y Contexto de los Sistemas de HIC.

El contar historias siempre ha tenido un papel muy importante en la sociedad, pues esta ha sido la manera privilegiada cómo se ha transmitido el conocimiento, la educación, la identidad cultural, los valores morales y la experiencia de generación en generación. De la misma manera el escuchar una historia a sido de las actividades lúdicas y de entretenimiento practicadas desde que se invento el lenguaje. Aún el la actualidad, el estar presente en la narración de una historia dramática es una actividad común, aunque ahora el medio de transmisión no sea el oral, sino que las acciones son representadas directamente al espectador como en una obra de teatro o en una película.

De la misma manera es interesante analizar el contexto social en que se da el contar una historia: durante que actividades comunes, ¿quién es el que cuenta las historias normalmente?, ¿qué debe tener una historia para ser interesante?, ¿y qué debe de tener para ser creíble?, ¿qué dinámica se da entre el narrador y espectador cuando se cuenta una historia?.

En la actualidad, el gran aumento que han tenido las capacidades gráficas y de procesamiento de las computadoras ha permitido que el entretenimiento digital se desarrolle de manera acelerada, explorando nuevas áreas y conceptos. Por ejemplo en los Estados Unidos, una rama del entretenimiento digital, la industria del videojuego, tiene ganancias anuales mayores a la de la industria del cine, que es una de las industrias del entretenimiento mas tradicionales.

La importancia ya mencionada de la narración de historias en la sociedad, así como el gran desarrollo que ha tenido la industria del entretenimiento digital y las capacidades de las computadoras han causado una explosión en la investigación de las historias interactivas en computadora.

Como toda área del conocimiento nueva, la de las historias interactivas por computadora tiene sus problemas y traspiés. Como el párrafo [1.2] menciona: los tres primeros mecanismos, que son los mas utilizados comercialmente, tienen todavía muchos problemas en presentar una historia realmente interactiva que sea atractiva para el espectador. En especial, la falta de interés del espectador en la trama de la historia que presenta un sistema de HIC se presenta por el poco análisis que han llevado a cabo los desarrolladores de estos sistemas acerca de que hace a una historia dramáticamente interesante. Ya que la mayoría de los que iniciaron el desarrollado de las HIC provienen del área de las Ciencias o de las Ingeniería, se puede entender la falta de desarrollo de la teoría narrativa y dramática aplicada a los sistemas de HIC.

Otro problema que aqueja a los proyectos de desarrollo de sistemas de HIC, debido a lo novel del campo, es el del largo tiempo y gran costo de desarrollo. Esto se debe a la poca experiencia que tienen muchos de los desarrolladores del área, así como a la gran multitud de disciplinas que toman parte en el desarrollo de uno de estos proyectos, causando la dispersión del esfuerzo del equipo de trabajo en diversas áreas, todas ellas en sí mismas muy complicadas. Algunas de estas áreas son la inteligencia artificial, las graficación por computadora, la animación, la conversión de texto a voz, el reconocimiento de voz, la generación de lenguaje natural, etc. Esto sin contar con las disciplinas relacionadas con la narrativa y el dramatismo.

Los problemas antes mencionados causan que gran parte del tiempo de los equipos de desarrollo e investigación en el área de las HIC sea utilizado para la creación y desarrollo de funciones básicas comunes a todos los otros proyectos de HIC, en lugar de la investigación de nuevos mecanismos mas interactivos y con historias de mayor valor dramático. Por ejemplo, algunas de las funciones básicas comunes a todos los proyectos que cuentan con representación gráfica de la historia son: Creación o adaptación del motor gráfico, animación de los personajes participantes en la historia, detección de colisiones, soporte de "scripting", entre muchas otras.

Por lo tanto, para que exista la posibilidad de investigación y el eventual desarrollo de nuevos modelos y teorías en el campo de las HIC, los tiempos que se aplican al desarrollo de las funciones básicas del sistema deben ser reducidos. De igual manera, la participación de investigadores dentro del área de la narrativa debe de ser fomentada.

Es por eso que se busca construir una Interfaz de Programación de Aplicaciones (API por sus siglas en inglés) para el desarrollo sencillo y eficaz de sistemas de HIC, de manera que en lo posible, se eliminen los problemas de programación y desarrollo que inhiben la investigación y creación de HIC con mayor interactividad, así como con una mejor calidad dramática en la narrativa. Esto al facilitar que investigadores del área de la narrativa puedan realizar sus investigaciones con una menor ayuda de los programadores e ingenieros (tal vez hasta ninguna, sí son lo suficientemente hábiles programando). Del mismo modo que OpenGL, un API para gráficas por computadora, provocó la aceleración en el desarrollo de toda la industria de la graficación por computadora. Con este sencillo API se piensa lograr, aunque sea en una pequeña cantidad de proyectos del Laboratorio de Interfaces

Inteligentes, que los sistemas de HIC se desarrollen de manera mas rápida, propiciando el desarrollo de éste.

1.4. Características del API para la Creación de HIC.

Se propone el desarrollar un API para la creación de HIC como parte de la solución de los problemas que aquejan a este campo con tan novel desarrollo. Si bien, no como solución a la base de los problemas que agobian a las HIC, sí como medio para propiciar la investigación que llevará a la solución de esos problemas.

Es por eso que este proyecto tiene como objetivo el desarrollar un API que permita reducir los tiempos de programación, aumentar la eficiencia y la eficacia de los recursos que se tienen para los proyectos, permitiendo que el enfoque del desarrollo se de en la investigación y la creación de sistemas interactivos de mayor calidad; no en la "reinvención de la rueda" del desarrollo de funciones ya antes implementadas por otros desarrolladores. Este API debe de cumplir con las normas básicas de facilidad de uso y de manejo y nomenclatura intuitiva que todo API debe de tener. Así como presentar las características y librerías de funciones necesitadas comúnmente por todos los desarrolladores de HIC. Cabe desatacar que las características y funciones mencionadas, pertenecen a diversos campos de las ciencias de la computación, por lo que su integración al API debe realizarse mediante un análisis concienzudo de las necesidades básicas de los desarrolladores.

Después de plantear el objetivo que se busca alcanzar al implementar un API para la creación de historias interactivas, es prudente delimitar los alcances del mismo. Este API presentará herramientas para el desarrollo de sistemas de HIC en los que los eventos de la historia se le presentan al usuario mediante un motor gráfico 3D que es capaz de presentar a los personajes mediante modelos formados por superficies poligonales animadas. El API estará conformado principalmente por clases que presentan la funcionalidad que los desarrolladores normalmente requieren. Y es mediante instancias de estas clases, que se permitirá que el usuario cree las entidades que existen en el mundo de la historia. No esta por demás remarcar que la propuesta de este API se hace unilateralmente y con miras a que sea usado por cualquiera que lo considere útil, es por eso que aunque planeado con un enfoque genérico, que lo hace útil a cualquier desarrollador del área de las HIC, el API esta planeado para que sea utilizado por otros proyectos que están siendo desarrollados en Laboratorio de Interfaces Inteligentes.

En la delimitación de las características y de la librería de funciones presentada por el API, se espera que una parte de ésta sea desarrollada como parte del proyecto, mientras que las otras serán agregadas de tecnologías ya existentes, pero mediante una capa "envolvente" de *software* (Conocido en inglés como Wrapper) se hará que estas funciones concuerden con la nomenclatura y estructura del API.

Algunas de las características del API que se van a implementar como parte del proyecto son: las de simulación de la física bajo la que actúan todas las entidades, la repuesta mediante "scripts" a determinados a eventos, el diseño de la jerarquía de clases coherente con las relaciones existentes en un mundo de la historia.

El otro ámbito del proyecto, que debe ser delimitado, es para que tipo de desarrollador va a estar enfocado el API. Para el tipo de desarrollador se debe de considerar las habilidades del que este programando, el tiempo que se le va a dedicar al proyecto, la rama del conocimiento en la que el desarrollador esta más familiarizado, la cantidad de integrantes del proyecto de desarrollo, la capacidad para el desarrollo de modelos y texturas etc. Como todo proyecto de abstracción, la construcción de un API generaliza las necesidades de una HIC, omitiendo características que a lo mejor son de gran importancia para otros proyectos. Esto sin lugar a dudas es inevitable, ya que nunca se podrá complacer las necesidades de todos. Es por eso que, en general, las funciones del API están pensadas de manera que satisfacen las necesidades de proyectos futuros, realizados en el Laboratorio de Interfaces Inteligentes

En general, las formas como se puede utilizar el API se pueden agrupar en dos grandes grupos que son: Los que agregan el comportamiento específico que desean mediante "*scripts*", usando objetos que en su totalidad pertenecen a las clases ya existentes en el API y los que agregan nueva funcionalidad creando clases que heredan de las clases ya existentes, actividad alcanzable únicamente programando en C++, además de agregar el comportamiento específico también mediante "*scripts*".

Capítulo 2.

Teoría del Diseño de APIs con un Enfoque Orientado a Objetos.

En construcción de una aplicación siempre se ha propuesto como una buena práctica el usar un diseño que prevea explícitamente la reutilización de código, que está listo para futuras expansiones y que reduce el problema a un grupo de problemas mas pequeños con el fin de manejar mejor la complejidad de los mismos. Hay varios modelos y arquitecturas para construir aplicaciones, todos ellos implícita o explícitamente impulsan el uso de las buenas prácticas del diseño de *software*.

Una de estas arquitecturas es la de sistema abierto (Open System Architecture). En esta arquitectura hay al menos dos interfaces: La *interfaz de programación de aplicación* mejor conocida por sus siglas en inglés API (Application Programming Interface). Cuya función es la de proveer de toda característica particular o funcionalidad requerida de la aplicación y *la interfaz de usuario* mejor conocida por sus siglas en inglés UI (User Interface). Cuya propósito es permitir al usuario acceder a la funcionalidad que el API proporciona.

La visión de la arquitectura de sistema abierto está delineada de una manera muy general. Las aplicaciones complejas que se desarrollan hoy en día tienen varios APIs que funcionan como capas de abstracción, proporcionando cada una de ellas una funcionalidad de mas alto nivel. De la misma manera, una aplicación puede tener varias interfaces de usuario que se especializan en la manera como el usuario va a interactuar con la aplicación. Por ejemplo, una aplicación puede permitir mediante una UI especializada que el usuario ingrese y vea todos los datos mediante una simple interfaz de texto, mientras que otra UI gráfica (normalmente conocida como GUI) permite al usuario acceder al sistema por medio del ratón, el puntero y los menús.

2.1. Definición de API.

Un API es la frontera a través de la cual las aplicaciones de *software* invocan los servicios y la funcionalidad proporcionados por un conjunto de rutinas, protocolos, aplicaciones o herramientas; utilizando los medios proveídos por los lenguajes de programación. Estos medios pueden incluir procedimientos, funciones, operaciones, objetos de datos compartidos, etc. Un API puede proporcionar un rango muy amplio de servicios y funcionalidad dependiendo del tipo de aplicaciones para las cuales el API está enfocado. Un API provee un método para desarrollar aplicaciones de *software* de manera mas barata y rápida. Las metas que debe tener cualquier API que se este desarrollando son las siguientes:

?? El API debe de sobrepasar la vida de su implementación inicial.
?? Debe de ser claro para los usuarios
?? Debe ser fácil de usar
?? Debe ser fácil de librar de errores.
?? Debe ayudar al perfeccionamiento de su implementación.

2.2. Pautas en el Diseño de APIs.

El uso de las pautas que a continuación se presentan puede llevar a facilitar las metas, arriba mencionadas, que el diseño de un API debe de cumplir:

2.2.1. El Usar el Nivel Adecuado de Abstracción.

Cuando se habla de usar el nivel adecuado de abstracción muchas veces se olvida que lo que esta en juego es la habilidad del API para sobrevivir al enfrentar a una actualización planeada. La mayoría de las veces, los que diseñan un API olvidan el anticipar esto. La interfaz que se proporciona debe ser lo suficientemente abstracta para permitir una implementación significativamente diferente, o sino cualquier código que use el API se volverá completamente inusable.

2.2.2. Usar el Menor Punto de Contacto Posible.

Esta pauta tiene la intención de proveer que el API sea los mas sencillo posible, con el menor número de llamadas posibles, pero tampoco menos d las necesarias. Cuando hay demasiados métodos que llamar para lograr que el API haga lo que deseamos, éste se vuelve pesado y engorroso.

2.2.3. Capacidad de Entregar un API libre de Implementación.

Con esta pauta obviamente no queremos decir que la implementación no se necesite entregar. Quiere decir que debe de haber un componente de API puro, tal como un archivo de cabecera en el caso de C y C++. Ya que es un error muy común el que en éstos archivos se incluya información acerca del tamaño, naturaleza y comportamiento de los objetos. No ayuda que las variables de los objetos sean protegidas (protected) o privadas (private), ya que de todas maneras se requiere que el código que usa el API sea recompilado cuando los detalles de estás implementaciones cambian. Los APIs por lo general solo deben de contener una pieza de datos identificable, un apuntador a una clase de implementación, que este totalmente oculta del archivo de cabecera.

2.2.4. Modelo Claro para el usuario.

Un diseño demasiado complejo o una abstracción demasiado cerebral pueden tener un impacto negativo. Hay que tomar en cuenta las necesidades del usuario, la abstracción que se haga, debe de seguir teniendo relación con el problema para el que originalmente fue planeado el API, aun cuando después se piense usar el API para otros problemas relacionados. El nombre de las llamadas y parámetros deben de sugerir su función.

2.2.5. Soporte y Recuperación de Fallos.

El API debe ser capaz de recuperarse de todas las condiciones reales de falla, tales como archivos faltantes, falta de espacio en disco o permisos de escritura erróneos. Esto no quiere decir que el API debe de sobrevivir a todo uso incorrecto del API. La naturaleza en la que el sistema se recupera debe estar planeada para cada tipo de fallo que pudiese ocurrir.

2.2.6. Condiciones de Fallo Claramente Delineadas y Reportadas.

El API además de reportar los fallos, si se encuentra en modo debug, deberá de reportar los errores de manera precisa y categórica. Los reportes deben de presentar tanto número de error como una cadena de reporte, que presente mensajes claros para los usuarios. Presentar un listado de los errores en la documentación, de modo que se pueda consultar de manera precisa que produjo el error.

2.2.7. Presentar el API con un Modo Debug.

Las fallas son producidas por condiciones en el mundo real, y aplican aún cuando el API es llamado correctamente. Una versión debug del API debe ser también presentada, en ella también se detectan las condiciones de error, que ocurren cuando el API es usado de manera incorrecta o con datos erróneos. La carga de esta detección normalmente es alta y no del todo compatible con una versión de producción del API, de ahí la necesidad de dos modos de uso diferentes.

2.3. Ventajas de la Utilización de APIs.

Ya de la simple definición de API, se puede ver una de las ventajas que tiene usarlos para el desarrollo de una aplicación de *software*. Esta ventaja es que el API presenta una frontera fuera de la cual nuestra aplicación es desarrollada; que puede permanecer constante mientras que las rutinas, protocolos y aplicaciones que implementan los servicios que presta el API pueden cambiar, ya sea para corregir un bug, para agregar funcionalidad o para portarlo a otro sistema, mientras que nuestra aplicación permanece sin modificarse.

Cuando las aplicaciones se desarrollan mediante un API, la reutilización y la aplicabilidad de la misma se ven incrementadas. Al construir un sistema de *software*, el problema que se busca resolver muchas veces es muy grande y complejo, por lo que se recomienda que este sea dividido en problemas mas pequeños que faciliten su solución. Al usar un API esto se vuelve explícito en el diseño, ya que dividimos al sistema en partes; en donde cada una nos proporciona cierta parte de la funcionalidad requerida. El uso un API propicia la reutilización al permitir separar la funcionalidad de una aplicación de su interfaz de usuario. Por ejemplo, al separar cierto subsistema de un API que construimos para una cierta aplicación, gracias a que esta diseñado para funcionar completamente separado de todos los otros componentes de ésta, al implementar todos los subsistemas de esta forma abrimos la posibilidad de utilizar este subsistema en otra aplicación futura. Un API propicia la aplicabilidad al permitir usar UI especializadas para diferentes tipos de situaciones.

Otra ventaja de diseñar aplicaciones mediante una API es que el sistema es más robusto, permitiendo mayor facilidad para encontrar los errores, ya que el código que implementa la funcionalidad principal está separado del que implementa la interfaz de usuario.

2.4. El Paradigma Orientado a Objetos.

Todo lo que existe en este mundo se puede clasificar como un objeto. Existen en las cosas hechas por el hombre, en la naturaleza, en los negocios, en la tecnología. Estos objetos se pueden crear, clasificar, modificar, combinar, destruir, etc. La idea de crear una abstracción que permite el desarrollo de *software* y que modela esta visión del mundo tiene como propósito que el desarrollador pueda entender y manejar de una mejor manera el sistema de *software* que esta desarrollando, pues está usando conceptos con los que él está familiarizado.

A continuación se muestran los conceptos en los que se basa el enfoque orientado a objetos. Para mayor profundidad acerca de la programación orientada a objetos, y en especial de los lenguajes orientados a objetos mas populares consultar [DEI99] y [WEH97].

2.4.1. Clases.

Una clase es un concepto que engloba las abstracciones de datos (conocido en OO como atributos) que definen a la clase y que solo pueden se accedidos, manipulados y modificados de manera correcta por una cápsula de abstracciones procedimentales (conocidos en el OO como métodos). Por ende, una clase encapsula a los atributos y los métodos con los que estos datos se manipulan. En una analogía con la arquitectura, las clases son como un plano o anteproyecto de edificio, no son el edificio mismo.

2.4.2. Objetos.

Los objetos son instancias de una clase, y todo objeto que pertenece a cierta clase tiene los atributos pertenecientes a esa clase y métodos con los que se manipulan. Los objetos tienen tres características principales: estado, comportamiento e identidad. El estado de un objeto está dado por sus variables de instancia. El comportamiento del objeto de está definido por sus métodos: lo que es capaz de hacer. Estos métodos afectan a las variables de instancia , dándole al objetos un nuevo estado. Su identidad, es la dirección de la imagen del objeto, que normalmente es accesible a los programadores a través de una referencia. En teoría todos los objetos tienen identidad, estado y comportamiento, en la práctica hay objetos que solo interesan por sus datos o por sus métodos. Por ejemplo, los objetos conocidos como mensajeros, sus atributos son lo importante, mientras que carecen de métodos relevantes. Mas adelante se analizarán los diferentes tipos de objetos según la importancia que le dan a su comportamiento y a su estado.

2.4.3. Atributos.

Un atributo puede verse como una relación binaria entre una clase y cierto dominio, o sea, que un atributo puede tomar un valor definido por un dominio enumerado. Por ejemplo, considérese la clase zapato que tiene un atributo talla. Por ejemplo, para zapatos de hombre, el dominio de valores de talla serían los valores de enteros entre 22 y 29.

2.4.4. Métodos.

Un objeto encapsula datos (en sus atributos) y los algoritmos que manipulan a esos datos. Estos algoritmos son llamados métodos y pueden ser vistos como módulos de manera convencional. Los métodos determinan el comportamiento del objeto.

2.4.5. Conceptos de Programación Orientada a Objetos.

A continuación, se presentan varios de los conceptos fundamentales en la programación orientada a objetos.

2.4.5.1. Encapsulamiento. Es una forma de ocultar y proteger información. Las clases no permiten que otras clases tengan acceso y modifiquen sus atributos si no es a través del método adecuado.

2.4.5.2. Herencia. Este es un concepto muy importante que promueve la reutilización de código. La idea básica consiste en que las clases nuevas se construyen a partir de clases ya existentes. La clase original se conoce como *superclase*, mientras que la clase que se construyo en base a la otra clase se conoce como *subclase*. La herencia da lugar a lo que se conoce como *jerarquía de clases*, que es la estructura arborescente que marca las relaciones superclase-subclase entre todas las clases diseñadas. La herencia también indica especialización. La clase mas reciente, la subclase es más especializada que su superclase. Una subclase hereda todos los atributos y métodos asociadas a su superclase. Esto significa que todas las estructuras de datos y algoritmos originalmente diseñados e implementados para la superclase están disponibles a todas las subclases que derivan de la misma superclase. Debido a esto, la jerarquía de clases se convierte en un mecanismo en el cual los cambios, en los altos niveles, se propagan inmediatamente a todas las subclases.

2.4.5.3. Polimorfismo. Esta característica permite que métodos con el mismo nombre, pero de diferentes clases puedan ser utilizados sin necesidad de saber la clase especifica a la que un objeto pertenece. Por ejemplo, las clase figura tiene el método dibujar, que se los hereda a sus subclases cuadro y circulo. Es obvio que el método dibujar de circulo no es el mismo que el de cuadrado, cada clase los implementa de manera diferente. Sin embargo, cuando recibimos un objeto que hereda de figura, no necesitamos saber a que clase pertenece el objeto, sino que simplemente llamamos al método dibujar y el método asociado a la clase del objeto será llamado de manera automática. Esto obviamente reduce el acoplamiento de los objetos, haciéndolos a cada uno más independiente.

Como el énfasis del paradigma orientado a objetos esta en las personas, no en las computadoras. La intención de desarrollar mediante objetos no es para optimizar y ayudar a la manera en las que las computadoras ejecutan el *software*, sino para que los desarrolladores sean mas productivos, en especial al ayudar a los programadores a manejar la complejidad y el cambio en su *software*.

2.5. Objetos como Gestores de Complejidad.

Los programadores, al tener desarrollar sistemas en computadoras con una capacidad computacional siempre creciente, se encuentran en una situación de dos filos. Pues siempre tendrán capacidad de sobra en máquinas con las que trabajan, pero enfrentarán el desarrollo de *software* mas complejo y grande, con la intención de aprovechar ese poder computacional extra. Una de las maneras en las que los objetos asisten a los programadores es al ayudar en el manejo de la creciente complejidad del *software*.

Un objeto que fue bien diseñado debe de ser entendible. Los grandes sistemas de *software* son difíciles de entender. En cambio, si un sistema esta compuesto de objetos individuales, de cualquier manera, cada objeto puede contener una cantidad de complejidad que puede ser totalmente comprendida. Los programadores entonces pueden comprender el sistema como un todo en términos del comportamiento de los objetos que lo componen y las interacciones entre ellos. Por lo tanto todo objeto que se diseñe debe de contener una cantidad de complejidad tal que pueda ser rápidamente entendida por un programador. Durante el diseño OO, los requerimientos del sistema se dividen en áreas de responsabilidad. Esa área de responsabilidad se le asigna a una clase. Para que esa clase cumpla con sus requerimientos se planea un grupo de métodos para que las instancias de esa clase puedan cumplir con sus responsabilidades. Si a esa clase se le asigno un área razonable de responsabilidad, la complejidad de los objetos instancia de esa clase contienen un nivel razonable de complejidad.

Además, si las clases están llamadas debido a objetos relevantes o a conceptos del dominio del problema la capacidad para entender es sistema es claramente ayudada. Además del hecho ya mencionado de que los sistemas orientados a objetos están basados en la manera como se organización las tareas humanas. Cuando se tiene una meta, se contrata a personas que nos ayudan a alcanzar esa meta. Cada persona accede a tomar algún rol, realizar algún trabajo, mientras uno los organiza y dirige los esfuerzos para alcanzar esa meta. De la misma manera, para lograr la meta en un sistema, se deben de reclutar objetos en los que se espera que cada uno cumpla con sus obligaciones delineadas por su contrato. Al organizar y dirigir los servicios provistos por los objetos, se puede alcanzar la meta propuesta para el sistema.

Todo objeto a demás de ser entendible, debe ser discutible, esto es que cada clase debe de tener su nombre. Para los programadores trabajando en el sistema, el nombre de la clase de algún objeto, representa el área de responsabilidad y el conjunto de servicios a través de los cuales el objeto cumple su responsabilidad. De esta manera mientras los miembros de un equipo de diseño descubren, nombran y crean objetos durante el diseño orientado a objetos, también se desarrolla un vocabulario . Un vocabulario que esta compuesto de los nombres de las clases y sus significados, facilita la comunicación entre los programadores trabajando en el sistema. Y como mientras mas efectivamente sean capaces los programadores de comunicarse, mas efectivos serán en el control de la complejidad.

Para finalizar, los objetos son propiciadores de la detección de errores. Cuando se crea un sistema, y los requerimientos de éste son divididos entre diferentes objetos, se tiende a encapsular el código encargado de manipular cierto tipo de datos en lo métodos del objeto que contiene esos datos. Esto mantiene al código que maneja ciertos datos de ser replicado y esparcido a lo largo del sistema. En el dado caso que se llegará a descubrir un error que causa que ciertos datos sean corrompidos, los desarrolladores únicamente tienen que buscar en un lugar para encontrar el error: El código que define los métodos de ese objeto. Una vez que ese error esta reparado en el código del objeto, se puede estar completamente seguro que ha sido reparado en cualquier parte, ya que el código del objeto, es único se encarga de manipular los datos. Este proceso de encapsular código con métodos en los objetos, combinado con una fuete separación de la interfaz y la implementación ,

ayuda a los programadores a construir objetos que pueden ser librados de errores concienzudamente, y por lo tanto confiables y robustos. Ya que para construir un sistema robusto se requiere que sus partes sean robustas Los objetos de manera inherente propician la producción de sistemas que, a pesar de su complejidad, son robustos por que todas sus partes son robustas.

2.6. Objetos como Gestores de Cambio.

Además de la complejidad, la otra realidad fundamental del desarrollo de *software* que los desarrolladores deben enfrentar es el cambio. Si un proyecto de *software* logra pasar sus primeras etapas, es muy probable que la base de código tenga una larga vida. Con cada nuevo lanzamiento vienen nuevos requerimientos. El código existente es retocado y mejorado para reparar bugs y para agregar funcionalidad. Los objetos también presentan una gran ayuda en este otro reto que los programadores enfrentan.

Uno de los principales ideales detrás del enfoque orientado a objetos es una fuerte separación entre la interfaz de los objetos y la implementación. El principal enemigo del cambio en un sistema de *software* es el acoplamiento, las interdependencias entre varias partes de un sistema. El objetivo de mantener la implementación y la interfaz separadas es para ayudar a los programadores a minimizar el acoplamiento en sus sistemas, tratando de evitar de que pequeños cambios en una parte del sistema causen un desastre en alguna otra parte no relacionada del mismo. En un sistema orientado a objetos, las interfaces de objeto son el punto de acoplamiento entre las diferentes partes del sistema, ya que las partes del sistema son objetos. Como las interfaces de los objetos son el único punto de acoplamiento entre las partes, muchos tipos de cambos pueden hacerse a la implementación de los objetos sin romper lo que el código de los otros objetos espera

Otra manera en la que los objetos ayudan a los programadores a lidiar con el cambio es mediante el polimorfismo. Los programadores pueden tratar con el cambio usando a los objetos como módulos intercambiables, permiten cambiar una implementación de una interfaz de objeto y usar otra diferente. Por ejemplo, se vuelve muy fácil el diseñar e implementar una nueva subclase la cual tiene objetos que se pasados a código que solo conocía de la superclase funcione sin ningún problema.

Finalmente, los objetos también ayudan a manejar el cambio porque los contratos con los objetos pueden ser muy abstractos. Un contrato con un objeto quiere decir lo se promete que el objeto va a hacer cuando se invocan a sus métodos de instancia, usualmente en términos de comportamiento. Los datos de instancia normalmente son mantenidos de manera privada, y como resultado los datos no forman parte del contrato de un objeto. Los contratos expresados en términos de comportamiento son mas abstractos que los basados en datos. Lo mas abstracto que sea el contrato, la mayor cantidad de manera en que se podrán cumplir los requisitos indicados por el contracto. El alto nivel de abstracción al que pueden plantease los contratos con los objetos da mas opciones a los programadores que necesitan cambiar o agregar una nueva implementación. Expresado en palabras llanas, mientras mas alto el nivel de abstracción, mayor la libertad de los programadores para hacer cambios de implementación que sean compatibles con las expectativas de comportamiento que los demás tienen de ese objeto.

2.7. Tipos de Objeto y su Aplicación.

Como ya se mencionó, los objetos tienen tres características fundamentales: estado, comportamiento e identidad. El estado está dado por el valor de sus atributos, mientas que el comportamiento está dado por los métodos. El paradigma de la programación orientada a objetos estimula el uso de objetos que prestan servicios, y que mantienen sus datos encapsulado, permitiendo que sean modificados únicamente mediante los métodos de la clase. Además se alienta a que los datos, quienes mantienen el estado del objeto, sean los que controlen la manera como se comporta el objeto, esto es, los objetos prestan un grupo de servicios que cambian dependiendo del estado del mismo.

Y aunque esta es la norma general, en la practica, los objetos no siempre pueden ser vistos como prestadores de servicios, a veces se necesita que un objeto simplemente almacene la información, mientras que otras veces se necesitan objetos que solo presten servicios. Es de esta forma que los objetos se clasifican según su uso.

2.7.1. Objetos de Servicios.

Este tipo de objeto es el que mas comúnmente se encuentra cuando se diseña un sistema de manera correcta, ya que este tipo de objetos presenta muchos beneficios. Esto debido a uno de los mas importantes conceptos en la programación orientada a objetos el encapsulamiento, que nos dice que los objetos contienen datos y comportamiento. Por lo tanto los objetos se pueden ver como paquetes de datos, de comportamiento o de ambos. Pero, para obtener el mayor beneficio del encapsulamiento, es conveniente pensar en los objetos como paquetes de comportamiento, o sea no pensar que los objetos portan información sino que nos prestan servicios. Esto tiene como propósito el que los datos no estén expuestos. Ya que si los datos están expuestos, el código que manipula esos datos se esparce a largo de todo el programa. En cambio, si son los servicios de alto nivel son lo que está expuesto, el código que manipula esos datos está concentrado en un lugar: la clase.

2.7.2. Objetos Mensajeros.

En ocasiones, el diseño lleva a pensar en objetos que simplemente son paquetes de información. Cuado esto esta diseñado de manera correcta, a estos objetos se les conoce como mensajeros. Estos objetos permiten el empacar los datos. Por lo común, estos objetos tienen una vida corta. Los datos que el mensajero transporta le son pasados al método constructor del objeto, entonces el mensajero es enviado a los que deben recibir el mensaje. Los receptores del mensaje obtienen la información mediante métodos get, una vez que el receptor extrajo la información del mensajero, éste usualmente destruye al mensajero.

La mayoría de los diseños de objetos deberían de ser de tipo orientado a servicios. En ocasiones, se encuentran datos que no se sabe que hacer con ellos. No se puede mover algún tipo de código a esos datos, porque no se sabe que cosa debe hacer el código. Es en esos casos en donde los datos se encapsulan en un mensajero y se envían a algún receptor que si conozca el comportamiento implicado por esos datos. El receptor extraerá y tomará la acción apropiada.

Un ejemplo muy común de objetos mensajeros son las excepciones. Las excepciones son objetos compuestos de una pequeña cantidad de datos, que son pasados al constructor y algunos métodos de acceso que permiten a las cláusulas, que son los receptores, obtener la información.

2.7.3. Objetos "Ligeros".

Este tipo de objetos contiene poca o ninguna información. Pero más que por la cantidad de datos que contienen, este tipo de objetos se reconoce porque su comportamiento no se ve afectado por su estado. Esto quiere decir, que los métodos que tiene este objeto no toman en cuenta los datos contenidos en sus atributos.

Este tipo de objeto se utiliza sobre todo como un componente de comportamiento capaz de ser desconectado a voluntad. El uso de este objeto esta restringido a modelar comportamientos para un estado dado, es por eso que sus métodos no prestan atención a sus variables de instancia, ya que solo se comportan considerando un estado.

Por ejemplo, un objeto elevador con sus métodos llamadaSubir() y llamadaBajar(), que son usados por los botones que están afuera del elevador, actúan de diferente manera según la dirección que lleva el elevador. Es ese caso en el código del método llamadaSubir() debe tener un condicional que tome en cuenta si el elevador sube y baja. Si el lugar de esa solución utilizamos objetos ligeros para dar el comportamiento del elevador, tendríamos simplemente dos objetos uno para el estado cuando el levador está subiendo y otro para cuando el elevador esta bajando. Esta, a primera vista, extraña decisión de diseño tiene un propósito: El permitir cambiar el comportamiento del objeto elevador de manera sencilla en caso de alguna nueva necesidad, porque como ya se mencionó, una de las ayudas que los objetos dan a los programadores es la de facilitar el manejo del cambio. Supóngase que ahora se decide agregar al elevador dos nuevos estado, el de subir rápido y el de bajar rápido, el comportamiento del elevador no puede ser el mismo a una llamada de llamadaSubir() cuando se encuentra en el estado de subiendo que cuando se encuentra en le estado de subiendo rápido. Si se utilizó el diseño de manejar el comportamiento de cada estado mediante un objeto ligero, lo único que se tendría que hacer es crear dos nuevos tipos de objetos ligeros, el de subiendo rápido y el de bajando rápido, mientras que si se hubiera usado un condicional, se tendrían que haber agregado dos condiciones mas, siendo esta una situación mucho mas proclive al error.

Se puede tomar como regla para decidir el usar objetos ligeros como los responsables del comportamiento de un objeto de servicio para un estado dado lo siguiente: Si la cantidad de estados que el objeto de servicios va a tener es grande, por lo tanto implica que los métodos contendrán gigantescas estructuras de if o case, y la posibilidad de que los comportamientos asociados a cada uno de éstos cambie es grande. Entonces es conveniente adoptar el modelo diseño de objeto de servicio con dependiente del estado comportamiento agregable mediante objetos ligeros.

2.8. Uso de un Enfoque Orientado a Objetos para el Diseño de un API.

Después de presentar la teoría, conceptos y metas del uso de APIs, así como del enfoque orientado a objetos en el desarrollo de aplicaciones de *software*, no es difícil de notar la relación entre ambos. En los párrafos anteriores de este capítulos se han presentado las metas, que con el objetivo de incrementar la productividad así como la calidad de las aplicaciones de *software*, ambos paradigmas proponen.

La primer meta que llama la atención es el de la reutilización de código. Ya que ambos enfoques lo predican. De ahí que la intención de crear un API mediante el enfoque orientado a objetos no sea nada disparatada. De la misma manera, la separación entre interfaz e implementación que el diseño de APIs propone como una buena práctica del desarrollo de aplicaciones de *software*; el enfoque orientado a objetos lo hace obligatorio, o al menos mucho mas explicito. Haciendo de éste otro punto mas a favor dela aplicación enfoque orientado a objetos para el diseño de APIs.

Cuando en las pautas para el diseño de un API, se pide que el modelo de abstracción sea inteligible para el usuario. Nos viene a la memoria la premisa fundamental por la que se planteó el enfoque orientado a objetos: Todo lo que nos rodea en el mundo se puede considerar que está compuesto por objetos que se pueden clasificar y jerarquizar. Esto quiere decir que se puede utilizar la forma en la que el enfoque orientado a objetos modela un sistema de *software*. O sea poniéndolo en los términos en los que comprendemos el mundo real, para poder mantener el API inteligible para el usuario.

En el diseño de un API, también se recomienda que el nivel de abstracción sea lo suficientemente elevado para permitir que la implementación de las rutinas y protocolos que prestaran los servicios del API puedan variar significativamente. Con la aplicación del enfoque orientado a objetos, no solo vemos que se facilita la selección del nivel de abstracción , sino que nos da una herramienta extra para lidiar con las actualizaciones que implica el agregar cierta funcionalidad. Esta herramienta es la herencia, ya que con ella se pueden sobrecargar métodos de modo que realicen las funcione extras que se necesitan. Además también, mediante el polimorfismo se puede asegurar que el código cliente del API, ni siquiera necesite saber de la existencia de la nueva clase.

Es por eso que mediante estas comparaciones, y otras que se podrían hacer, que se puede afirmar que al realizar un diseño de APIs, que cumple desea cumplir con sus pautas y compromisos; es complementado, incluso propiciado, por el uso de un enfoque orientado a objetos. Es mas, por la cantidad de coincidencias tan abrumadora, parecería incluso que solo se pueden diseñar APIs mediante el enfoque orientado a objetos. Aunque esto de ninguna manera es cierto, ya también se pueden usar lenguajes proceduales para el desarrollo y uso de APIs, por ejemplo, el famosísimo API de gráficas por computadora *OpenGL*, que simplemente utiliza C, y que es uno de los APIs mas utilizados. Mas información de *OpenGL* en [WOO96].

2.9. Técnicas de Análisis para un Sistema Orientado a Objetos.

Después de que se han mostrado algunas de las ventajas que tiene el uso del enfoque orientado a objetos en el diseño de APIs. Se presentaran las técnicas que se deben de aplicar para el análisis de un sistema. Cabe destacar que lo mas importante de este análisis no es definir los objetos de manera inmediata, sino que promueven la comprensión de la manera en la que el sistema será usado. Si el sistema será usado por personas o por otras máquinas, que tipo de usuarios serán los que interactúen con el sistema, si el sistema esta envuelto en el control de procesos; o por otros programas o si el sistema coordina y controla otras aplicaciones. Una vez que se ha definido el escenario, se puede comenzar el modelado se *software*.

Las técnicas, que se definen a continuación, se pueden usar para recopilar los requisitos básicos del usuario y después definen presenta un modelo de análisis para un sistema orientado a objetos. Para conocer mas a profundidad acerca de estas técnicas véase [PRE97].

2.9.1. Análisis de Casos de Utilización.

Cuando se realiza un análisis con la intención de desarrollar un sistema, la recopilación de requisitos es siempre el primer paso. Esto se puede llevar a cabo de varias formas, por ejemplo, mediante una entrevista en la que el analista o el ingeniero de *software* definen los requisitos básicos del sistema y del *software*.

Basándose en esos requisitos, se crean un conjunto de escenarios de manera que cada uno identifique una parte del uso que se le dará al sistema a construir. Los escenarios, a menudo son llamados casos de uso, aportan una descripción acerca de cómo el sistema será usado.

Es en esta parte del análisis donde se identifican los diferentes tipos de actores, categoría que incluye a personas o dispositivos, que van a utilizar el sistema cuando éste este operando. Un actor es cualquier cosa externa al sistema que se comunique con éste. En esta parte del análisis se debe de diferenciar entre actor y usuario, ya que los actores corresponden a personas o dispositivos que desempeñan diferentes papeles.

Los casos de utilización se definen, ya una vez que los actores han sido identificados, en ellos se describe la manera en la que un actor interactúa con el propio sistema. Al plantear los casos de utilización, se indican las principales tareas o funciones a realizar por el actor, así como la presentación de un análisis del flujo de información del sistema. Además del flujo de información principal, se tienen que indicar otros casos de uso como los que pueden surgir cuando el actor informa al sistema de cambios en el entorno exterior. Otro caso de utilización en el que se puede agregar flujos de información son en los que el actor es informado sobre algún cambio inesperado. Es de notar que cada caso debe de aportar un escenario no ambiguo de interacción entre el actor y el *software*.

2.9.2. Modelado Clase-Responsabilidad-Contribución.

Es en este paso donde, después de conocer lo que se quiere que haga el sistema, además de la manera en la que se quiere, que se procede a identificar las clases candidatas, indicando sus responsabilidades y contribuciones. Este modelado tienen como propósito el identificar y organizar las clases importantes para el sistema o a lo requisitos del producto. Cuando se plantean las responsabilidades de una clase, simplemente se está hablando de cualquier cosa que conoce o hace la clase. Para considerar que clases son colaboradoras de otras clases, se busca la relación de dependencia, en la que para cumplir con una responsabilidad una clase necesita de la información que le puede proporcionar su clase contribuidora.

Para la identificación de objetos existen diversas propuestas y pautas. Una de ellas es la que indica las formas mas comunes en las que se puede manifestar un objeto.

Estructuras. Definen una clase de objetos. Por ejemplo, computadoras, vehículos, aviones o sensores.
Eventos. Sucesos que ocurren dentro del contexto de operación del sistema. Por ejemplo, la terminación de una serie de transacciones financieras o el aviso de un robot.
Cosas. Objetos que son parte del dominio de información del problema. Entre ellas tenemos a las señales, las presentaciones o los informes.
Roles. Desempeñados por personas que interactúan con el sistema, como los ingenieros, los diseñadores o los contadores.
Lugares. Que son el marco en los que se da el problema, por ejemplo la planta de energía o el muelle de carga.
Grupos Organizacionales. Que tienen relevancia en el funcionamiento de la aplicación, como las divisiones, los comités o los grupos.

Claro que hay que hacer notar que no todo objeto potencial puede ser incluido en el modelo de clase-responsabilidad-contribución (CRC). Es necesario que el objeto presente ciertos atributos. En el caso de las responsabilidades, que pueden ser implementadas por uno o varios métodos deberán de identificarse mediante un análisis gramatical a la narración del procesamiento del sistema. En este caso los verbos se transforman en candidatos a métodos.

En el caso de las contribuciones, estás se identifican a través de las relaciones entre las clases. Por ejemplo, si una clase necesita mandarle un mensaje de cualquier tipo a otra clase para poder cumplir con una responsabilidad, ésta se trata de una clase contribuidora.

Para finalizar, el modelo CRC es la representación inaugural del modelo de análisis para un sistema OO, entre las ventajas que presenta, es que se pueden realizar las primeras pruebas, al llevar a cabo una revisión dirigida por casos.

2.9.3. Identificación de Estructuras y Jerarquías

Después de la identificación de métodos, atributos y objetos, l foco del análisis se centra ahora en la identificación de las jerarquías resultantes de las clases y subclases y de la estructura del modelo de clases. Es en está parte del análisis que se puede realizar una generalización-especialización para las clases identificadas. Como por ejemplo la clase

perro, se refina en un conjunto de especializaciones, labrador, pastor alemán, bull terrier, etc. Esto crea una jerarquía de clases simple. Otra posibilidad es que el objeto identificado en el modelo CRC se compone de un número de partes que pueden definirse a su vez como objetos.

La ventaja de este tipo de representaciones estructurales es que permiten obtener los medios gráficos que dividan al modelo CRC en dominios mas pequeños y manejables. Con esto se permite tener un detalle mayor para la revisión y el diseño.

2.9.4. Definición de Subsistemas.

Cuando se trata de un sistema muy complejo, se pueden llegar a tener cientos de clases y varias decenas de estructuras. Es por eso que surge la necesidad de presentar una representación concisa de los modelos CRC.

Los subconjuntos de clases que contribuyen entre sí para llevar a cabo un conjunto de responsabilidades unidas, se conoce normalmente como un subsistema. Un subsistema es una abstracción que aporta una referencia a los detalles en el modelo de análisis. Si cuando desde el exterior de el subsistema se le puede tratar como una caja negra que contiene un conjunto de responsabilidades y que posee sus propios contribuidores externos. De esta manera un subsistema implementa uno o varios contratos con sus colaboradores externos. Un contrato es la lista específica de solicitudes que los contribuidores pueden hacer a un subsistema, y que se espera que éste sea capaz de cumplirlos

2.9.5. El Modelo Objeto-Relación.

Con el modelo CRC mostrado en 2.9.2. se han establecido ya los primeros elementos de las relaciones de clases y objetos. Se dice que hay una relación entre dos clases cuando éstas están conectadas. Es de esta manera que las clases contribuidoras siempre están relacionadas. El tipo mas común de relación es la conocida como relación binaria, es una relación que tiene una dirección y que se debe definir a partir de la clase que funciona como servidor y de cual desempeña el papel de cliente.

Para definir las relaciones se puede realizar un análisis de los verbos que aparecen en la formulación de casos del sistema. Se pueden crear los siguientes tipos de relaciones, según los verbos que aparezcan:

?? Que indican locación física o emplazamiento, Es indicada por verbos como es parte de, contenido en.
?? De comunicación, indicada por verbos como transmite a u obtenido de.
?? De propiedad, cuando aparecen verbos como incorporado por, se compone de.
?? Que indican el cumplimiento de una condición, por ejemplo dirige, coordina, controla.

Cuando las relaciones se han establecido y nombrado, se propone conveniente el establecer la cardinalidad en cada extremo de las mismas. Existen cuatro opciones: 0 a 1, 1 a 1, 0 a muchos y 1 a muchos. Este tipo de análisis añade otra dimensión al modelo de análisis general. Porque no solo se identifican las relaciones entre los objetos, sino que se pueden definir todas las vías importantes de mensajes.

2.9.6. El Modelo Objeto-Comportamiento.

Después de realizar los modelos estáticos del análisis OO, es necesario pasar al comportamiento dinámico del sistema. En este paso se representa el comportamiento del sistema como una función del tiempo y de la aparición de eventos específicos. Aquí se intenta mostrar como responderá el sistema a eventos externos. Los siguientes pasos proporcionan una guía acerca de lo que el analista debe de tomar en cuenta para crear el modelo:

1. Evaluar todos los casos de utilización para comprender por completo la forma en la que se interactuara con el sistema.
2. Analizar los eventos y su preponderancia para dirigir la secuencia de interacción y comprender la relación entre objetos específicos con estos eventos.
3. Crear una rastro de eventos para cada caso de utilización.
4. Construir un diagrama de transición de estados para el sistema.
5. Revisar el modelo Objeto-Comportamiento para verificar la consistencia y la precisión con los casos de utilización.

Para identificar a los eventos mediante los casos de utilización, debemos recordar que los casos de uso representan una secuencia de actividades que incluyen al sistema y a los actores, y en general se puede decir que un evento ocurre cada vez que un actor intercambia información con el sistema OO.

El siguiente paso para realizar el modelo de comportamiento es el de seleccionar los estados, tanto de los objetos como del propio sistema. Diferenciar estas dos caracterizaciones de estados es importantes, ya que, el estado del sistema OO siempre será dado por la conjunción de los estados de todos los objetos que lo conforman.

Un estado de un objeto adquiere características pasivas y activas. Las características pasivos son simplemente el estado actual de todos los atributos de un objeto, contenidos en las variables de instancia. Las características activas de un estado están dadas por ciertas variables de instancia que hacen que el objeto entre en un periodo, de duración definido o no, de actividad. Por ejemplo en un objeto que representa la lectora de tarjetas de crédito, el evento "la tarjeta ha sido insertada" llevara al objeto lectora del estado en descanso, al estado comparando, que es un estado en el que el objeto permanecerá activo procesando la información, hasta que confirme si el dato es correcto o no. En ese momento pasará a otro estado dependiendo del resultado de la confirmación. Este tipo de eventos también se conoce como disparadores (Triggers en inglés).

Una parte muy importante de este modelo es la representación gráfica simple de los estados activos de un sistema y los eventos disparadores que llevan transición de uno a otro. Esta representación se puede llevar a cabo mediante un grafo, como se hace con los autómatas, donde cada nodo representa estado un estado activo y cada vértice representa un evento. Se pueden colocar nombre al lado de los nodos y de los vértices para indicar el nombre de los estado y de los eventos. Los vértices tienen dirección, ya que la flecha indica de que estado a que estado se pasa cuando un evento específico sucede. En un transición a veces también es necesario especificar un guardián. Un guardián es una condición booleana que debe de satisfacerse para que se pueda darse la transición. Como por ejemplo, al darle

la clave al objeto lector, esta debe de ser de 8 dígitos para entonces pasar al estado comparación. Cabe destacar que en la teoría de los autómatas tipo de condición no existe para la transición de estados, en realidad con los autómatas el grafo se extendería, agregando estados para indicar la recepción desde uno hasta ocho dígitos antes de pasar al estado confirmando. Claro que para un modelo de Objeto-Comportamiento esto no es practico, ya que haría los grafos demasiado grandes. Es por eso que se permite la idea del guardián, que es una condición booleana que se debe de cumplir antes que se de la transición.

Un segundo tipo de representación de comportamiento del análisis Orientado a Objetos lo forma una representación de los estados para el sistema. Esta representación abarca un modelo simple de rastro de eventos que presenta como los eventos causan las transiciones de objeto a objeto y un diagrama de transición de estados que ilustra el comportamiento de cada objeto durante el procesamiento.

Cuando los objetos han sido identificados, el analista crea una representación acerca de cómo los eventos provocan el flujo desde un objeto hasta otro. Esta representación se conoce como el rastro de eventos, y es una versión resumida del caso de utilización, en ella se presentan los eventos que causan el comportamiento de pasar de un objeto a otro.

Cuando el rastro de eventos se haya desarrollado por completo, se puede construir un diagrama de flujo de eventos a partir del diagrama de rastro de eventos. Los eventos que apunten hacia un objeto y que hagan que el objeto cambie de estado servirán para poder asociar a la transición de estados con las responsabilidades de la clase.

Los diagramas y modelos que hasta este momento se han creado, gracias al análisis OO, forman una muy buena base para la creación de la *especificación de requisitos del software*.

Capítulo 3.

Necesidades de un API para Historias Interactivas por Computadora.

Como en todo proceso de desarrollo de *software*, al desarrollarse un API, lo primero que se debe hacer es delimitar y definir los requisitos que debe de satisfacer el sistema de *software*. En el caso del desarrollo de un API orientado a objetos, se deben de considerar las necesidades funcionales mas importantes y que la mayoría de los posibles usuarios de este API pueden necesitar.

Por obvias razones de delimitación, ya que sería muy difícil realizar una entrevista o al menos, una consulta, con diferentes grupos que actualmente desarrollan sistemas de HIC; se ha decidido que el API se base principalmente en las necesidades de un proyecto que se desarrolla actualmente en el Laboratorio de Interfaces Inteligentes. Este proyecto, con el nombre provisional de *Calakmul virtual*, tiene como intención el permitir a los usuarios viajar y conocer las ruinas de Calakmul a través de un modelo 3D de las mismas. Los usuarios viajan a través del modelo de Calakmul mediante un avatar que los representa en el mundo virtual. El personaje principal con él que los usuarios interactuaran a través de su avatar es un guía virtual, que los llevará a conocer el lugar. Además el guía les contará acerca de la historia y la relevancia de cada lugar.

Por lo que, aunque siempre se tendrá en mente al proyecto de Calakmul virtual cuando se diseña el API de HIC; también se tomarán en cuenta las necesidades mas generales de otros proyectos, mediante las publicaciones que éstos hacen. Ya que en estas publicaciones es donde se pueden identificar ciertas necesidades comunes. Y aunque a primera vista la aplicabilidad del API de HIC a una visita turística no es aparente, ya que Calakmul virtual es una visita interactiva a un sitio arqueológico, la manera como el guía llevará a los usuarios a los diversos lugares de las ruinas mientras les explica que sucedió en cada punto si se puede ver como una historia interactiva.

El agente que controlará al guía de turistas puede ser dotado de cierta Inteligencia Artificial, que le permitirá interactuar con los usuarios para mostrarles todos los lugares del sitio. Además el agente deberá de decirles a los usuarios lo que pasó en cada uno de ellos. El problema que surge al sólo aplicar los conceptos de inteligencia artificial es que la visita puede volverse aburrida o simplemente muy predecidle. Esto por que los alcances de la Inteligencia Artificial son todavía muy pequeños para poder dotar al guía de turistas con la suficiente capacidad para suplantar a un guía de turistas humano. Es importante destacar que la necesidad de un guía virtual controlado por la máquina es indiscutible, ya que de no usarlos, implicaría utilizar un operador humano para controlar al guía de turistas por cada sistema de Calakmul virtual.

Es en la problemática del guía virtual donde esta una de las principales aplicaciones del API de Historias Interactivas para desarrollar el sistema de visita virtual a Calakmul. Ya que mediante el API de HIC se puede proporcionar al agente que controlará al guía de turistas de las funciones que requiere de una manera mucho mas sencilla pero mas interesante para el usuario; resultado que difícilmente se obtendría con la simple aplicación de la inteligencia artificial. En lugar de intentar dotar de un agente con inteligencia artificial, implementada por una gran cantidad de algoritmos complejos con resultados no muy satisfactorios; para controlar al guía virtual, se puede recurrir a guiones (Mejor conocidos por su nombre en inglés como *scripts*) para dotar al guía con un comportamiento mucho mas especifico y definido del que se puede lograr con la inteligencia artificial de

hoy día. Estos *scripts* serían utilizados por los desarrolladores para indicar a todos los agentes que secuencia de acciones ejecutar en dado caso que se cumpla una condición, que reciban algún evento u orden.

El guía que mostrará el sitio de Calakmul tiene un rol preponderante en el sistema de visita virtual, ya que es a través de éste que se podrán conocer todos los lugares del sitio arqueológico. El guía virtual también es la fuente principal de interactividad del sistema, ya que a través de éste es como el sistema responderá a la interacción del usuario. Hay una gran variedad de acciones que el guía puede realizar, pero entre las mas importantes están: la de informar al usuario para que eran usadas en la antigüedad cada una de las secciones del sitio arqueológico, que es presentado al usuario mediante un modelo 3D, para contar la historia del pueblo que habitó en el sitio, además de la historia de sus pueblos vecinos. El guía también tendrá la muy importante función de contar las leyendas y mitos que están relacionados con el lugar. Gracias a los guiones, está planeado que el guía no solo sea un agente que esté hablando todo el tiempo, sino que se pueda modificar su comportamiento si se le pregunta algo en específico, si se le pide ir a ciertos lugares en un orden particular o si se le entrega algún objeto especial.

Al delimitar el API y enfocarlo principalmente a las necesidades de Calakmul virtual, obviamente se desechan varias de las opciones de HIC mencionadas en el primer capítulo, ya que es obvio que si se necesita una representación gráfica 3D del lugar donde sucede la historia. Los tipos de sistemas interactivos que tienen como salida la voz, esto es como un convertidor de texto a voz, así como los que tienen la salida de texto simple quedan excluidos. Aún así, con la salida a través de video que muestra el ambiente 3D en tiempo real, la salida de voz ha sido siempre un gran método de comunicación para transmitir las historias, por lo que se decidió que de todas maneras se usar el convertidor de texto a voz para que los diferentes agentes, en especial el guía de turistas se pueda comunicar con los usuarios.

De la necesidad de presentar modelos 3D, tanto del sitio arqueológico como de los personajes de los usuarios, es necesario que el API incluya la funcionalidad de un motor de juegos 3D. En donde los usuarios son representados a través que modelos que existen e interactúan con el modelo que representa al sitio arqueológico de Calakmul. Estas representaciones de los usuarios les deben de permitir interactuar con todos los otros agentes que existen en el mundo, en especial con el guía. El API debe de soportar el agregar y el borrar de objetos en tiempo de ejecución, también es recomendable que promueva la reutilización de ciertos objetos, como también es importante que aproveche el polimorfismo. El API también debe de soportar un nivel básico de simulación de la física y de la detección de colisiones. Para finalizar, otra característica importante que deben de tener los agentes que participan en el sistema del HIC es la de seguir caminos previamente definidos.

Como ya se mencionó en la parte de teoría de diseño de APIs del capítulo anterior, cuando se diseña un API es benéfico que éste se diseñe pensando que sobrepasará el uso particular que se le está dando. Dicho esto, aun cuando ya se mencionó que las necesidades básicas que se van a satisfacer son las del proyecto Calakmul Virtual. Es por eso que este

proyecto de API tomará mucho en cuenta las necesidades de Calakmul. Esto sin caer en diseños de sistema monolíticos, totalmente enfocados y optimizados para una aplicación, pero imposible de aprovechar en otras. En conclusión, las características y necesidades funcionales que el proyecto Calakmul tenga, serán totalmente satisfechas por el API, pero el API contará también con una variedad de características y funcionalidades que lo harán atractivo para el desarrollo de otros proyectos de historias interactivas. La modularidad es otro factor que se va a ponderar altamente, considerando necesario separar de manera intuitiva los diferentes componentes de funcionalidad. El facilitar la reutilización del código será una gran prioridad, ya que el sentido de crear un API sería nulo si no se permite que otros puedan utilizar el código generado.

De ahí la decisión de realizar este trabajo mediante el enfoque orientado objetos, dándole especial atención al diseño, de manera que éste pueda ser aprovechado por otros proyectos. Y es mediante la programación orientada a objetos, que provoca el diseño y la codificación intuitiva y explicita, que también se planea dejar que el API sea extensible para que otros desarrolladores puedan agregar las características y capacidades funcionales que su propio proyecto requiera; y que debido a las delimitaciones del API desarrollado, no se suministraron. Parte del resultado de este trabajo será mostrar los requisitos mencionados se pueden satisfacer mediante la aplicación de la programación orientada a objetos.

3.1. Análisis de los Problemas del Desarrollo de Sistemas de HIC.

Al plantearse la necesidad de construir un API para el desarrollo de sistemas de HIC, es totalmente claro que el primer paso en su diseño, debe de ser un análisis de la problemática la que los desarrolladores de éstos se enfrentan. A continuación se presenta un listado de los problemas que se consideran mas trascendentes y a los que más comúnmente se enfrenta el desarrollador de historias interactivas. Este listado de conformo mediante la experiencia propia del desarrollo de sistemas, así como a través de la lectura de artículos sobre diversos proyectos de HIC.

3.1.1. Carencia de una Metodología de Diseño Común.

Al utilizar un API, el desarrollador cuenta con una herramienta con la que puede realizar de manera mas sencilla su trabajo. El problema en el campo de los sistemas de historias interactivas es que la manera como éstos se desarrollan todavía no está completamente definida, es más, en algunos casos las propuestas de diseño llegan a ser contradictorias. Es como si se tratará de hacer una herramienta para una facilitar una actividad que no sabemos como se realiza.

Los ejemplos de diseño que se pueden ver aplicados en los sistemas actuales son variados, y muestran lo joven que es este campo. En este inciso se mostraran algunos de los diseños mas comunes, se ponderaran sus ventajas y desventajas de manera breve, así como también se indicará cual será el patrón de diseño que se usara para este proyecto. Es importante remarcar la diferencia entre los mecanismos de interactividad que se mostraron en [1.2] y los modelos de diseño que a continuación se presentan, ya que los primeros se

refieren al tipo de experiencia que tiene el usuario con la interactividad del sistema, mientras que los segundos se refieren fundamentalmente a conceptos de diseño de la arquitectura de la aplicación a la luz de ingeniería de *software*. Es por eso que en este capítulo, enfocado a las necesidades que debe cubrir un API, el tema de la falta de una metodología común de diseño se hace presente. Ya que, él tener que desarrollar un API para un tipo de aplicación que no tiene un patrón de diseño estándar es sin lugar a dudas un problema. Además, la forma como se denomina cada a uno de ellos tiene la intención de describir las características particulares de cada diseño. Ya que por lo nuevo del campo todavía no existe un consenso en los diferentes modelos de diseño de *software* aplicables a un sistema de historia interactiva, ni en su nomenclatura. Para encontrar una discusión mas profunda y una propuesta de solución a esta problemática consúltese [ULR02]. Algunos de las metodologías de diseño mas comunes se presentan a continuación.

De Capa de Historia Lineal y de Capa de Control. El diseño de estos sistemas es muy común en el campo de los juegos de video, en este patrón de diseño el sistema se divide fundamentalmente en dos capas: la primera, en la cual se lleva el control del avance secuencial de los eventos de la historia, mientras que en la otra se controla el comportamiento individual de los objetos y personajes de la historia.

Con Generador de Historia, Ejecutor y Mediador. Este tipo de arquitectura es particularmente interesante, y se encuentra en la vanguardia de las historias interactivas. En esta arquitectura el sistema se divide en tres partes: Un planeador de alto nivel que es conocido como el Generador de Historia (GH), que crea una historia automáticamente, a través de algoritmos de búsqueda que se desarrollan en el espacio de las acciones que todo objeto o personaje existente en la historia puede realizar. Un Ejecutor, que es la parte del sistema que convierte el plan de historia creado por el GH y lo convierte en acciones de bajo nivel, que los diferentes personajes y objetos deben de realizar para que el plan pueda llevarse a cabo. Y un mediador, que es la parte del sistema que esta al tanto de las acciones de bajo nivel del usuario, que podrían interferir con la coherencia y causalidad de las acciones indicadas en el plan de la historia. Las técnicas con las que el mediador responde a las acciones de un usuario capaces alterar la historia son dos: La desactivación de la acción y la modificación del plan. La primera técnica implica que cuando el Mediador detecta alguna acción capaz de alterar el plan de la historia, éste deberá de responder con alguna acción veraz dentro del mundo de la historia que desactive el efecto que la acción del usuario tendría. Por ejemplo, si el usuario encontrase una llave que le permitiera entrar antes de tiempo a un cuarto donde se esconde un secreto trascendental para la historia. El sistema mediador tendría que responder causando que la llave se atascara en el cerrojo de la puerta. Evitando que el usuario entre en el cuarto antes de tiempo, donde se encuentra un secreto que se supone debe estar oculto al usuario hasta mas entrada la historia. Además la posibilidad de que la llave se atasque en el cerrojo es lo suficientemente alta como para mantener la veracidad de la historia. La segunda técnica, como nombre lo implica, causará que el mediador le pida al GH que cree otro plan de historia cuyo punto inicial será la acción del usuario que cambio el plan anterior. Los factores que harán que el mediador aplique una u otra técnica son varios: El costo computacional y de memoria que tomará el crear un nuevo plan de historia desde la acción del usuario que modificó el plan anterior. La posible perdida de veracidad que implica el desactivar varia veces una acción dañina para el plan de la historia. Por ejemplo, si el personaje principal de la historia, que es un detective,

le dispara con intención de matar a un personaje que mas adelante en la historia jugara un rol fundamental, puede causar que el Mediador desactive esa acción al hacer fallar el tiro del detective. El problema viene cuando el usuario que maneja al detective continua disparando contra ese personaje, ya que la veracidad de la historia quedará seriamente dañada si el detective continuará fallando por causa de la técnica de desactivación que esta realizando el mediador. Por último, el mediador puede decidir que técnica aplicar al realizar un análisis del valor dramático, si es que el sistema mediador tiene la capacidad para esto, del plan de historia actual contra el plan de historia generado desde la acción disruptiva del usuario.

De Narrativa Emergente. En este tipo de diseño el énfasis se da al desarrollo de agentes autónomos creíbles. La suposición subyacente es que una historia coherente emergería del comportamiento autónomo de los agentes, con un poco de ayuda de las no muy frecuentes intervenciones del controlador de drama. En este tipo de diseño, la historia debería de emerger del fondo hacia arriba, basada en los comportamientos procedimentales que están descritos de una manera paramétrica por el autor.

En años recientes se notado a dificultad que los autores tienen para describir los parámetros de una narrativa emergente de manera ad hoc. En realidad, muchas veces es mas fácil razonar por medio ejemplos en lugar de por lógica, y es mas fácil describir un fenómeno que a través de todos los parámetros. Además, se ha encontrado el problema del bajo valor dramático de las historias que este tipo de diseño produce. Este problema se puede entender de manera muy simple: Imaginemos que la inteligencia artificial que controla el comportamiento autónomo de loas agentes alcanzara un grado tan elevado que causará que éstos se comportaran como cualquier ser humano y que los parámetros que controlan el comportamiento fueran capaces de producir cualquier comportamiento humano, lo que se tendría sería una simulación perfecta del mundo real, de la interacción humana y sus efectos en los otros. Pero pocas veces puede decir que la vida real tiene los valores dramáticos de una historia hecha por algún actor. Aún con la inclusión del controlador de drama el énfasis de esta metodología de diseño está colocada erróneamente. Ya que las acciones de la historia no deben de estar gobernadas por la causalidad de la historia, que la acción realizada por un personaje no está gobernada por que otros eventos ocurrieron, sino porque están gobernadas por la causalidad de la narrativa completa, o sea que un personaje realiza una acción porque es necesario para la trama.

Con Capa de Historia, de Escena y de Personaje. En este tipo de diseño, al sistema se le divide en tres, esto con la intención de crear niveles de abstracción que faciliten la construcción y prueba de su historia interactiva. Puesto que toda actividad artística permite al desarrollador admirar el resultado parcial de su labor. Es una parte fundamental de esta propuesta que los desarrolladores de HIC sean capaces de ver y probar su trabajo sin la necesidad de que toda la aplicación esté terminada. Cuestión que debido a limitaciones tecnológicas y de diseño ha estado muy limitada a los desarrolladores de historias interactivas. La necesidad de probar la historia antes de que este terminada tiene especial importancia si se considera que este tipo de obra requiere del análisis de su desarrollo con respecto a la interacción con el usuario. Y es basándose en la necesidad de un "escenario de ensayo" para cualquier historia que este en desarrollo que este modelo de diseño propone tres capas de sistema, donde la modularidad y la autonomía de escalabilidad benefician el

trabajo de equipos interdisciplinarios además de facilitar la revisión del trabajo ya realizado.

Este diseño jerárquico del sistema tiene como finalidad, según sus proponentes, la de resolver los problemas y peculiaridades que surgen, en diferentes niveles, en este tipo de sistemas. En especial, este tipo de diseño jerárquico presupone que no es conveniente, a veces hasta posible, que el autor de la historia interactiva pueda manejar toda la programación y desarrollo de contenido para crear una historia interactiva. De ahí la propuesta de dividir al sistema en partes que permitan la división de responsabilidades de manera intuitiva, tal como sucede en la producción de películas para cine o en el desarrollo de videojuegos comerciales. Pues en este tipo de proyectos el desarrollo se lleva a cabo mediante equipos heterogéneos donde la interdisciplinaridad implica la conjunción de personal afín a las artes y al área de la técnica o de la ingeniería. En áreas, como la cinematografía, donde este tipo de creación artística grupal esta mas desarrollada, cada miembro tiene un cierto grado de libertad acerca de la toma de decisiones de un área específica, mientras que un solo autor o director están a cargo de la experiencia del usuario y de la visión en general del proyecto.

La arquitectura que se propone mediante este diseño consiste en tres capas de motores. Estos son componentes que en tiempo de ejecución se encargan de la presentación de la historia a diferentes niveles de abstracción, estos son: *La capa de historia*, que se encarga de controlar el flujo de la historia en el nivel estructural mas alto. En esta capa, como en las otras, el nivel de automatización se puede escalar. Y puede ir desde un escenario lineal, totalmente controlado por el autor, pasando por un sistema de ramificación de historia hasta llegar a un sistema de generación de historia procedual que se base en modelos abstractos de narrativa, como las funciones de Propp. *La capa de escena,* esta capa está encargada de seleccionar de una base de datos de escena o de construir una escena, dependiendo del grado de automatización que se le dé, que llene la función narrativa que el motor de historia ha seleccionado. Con este tipo de motor se espera que se desarrollen estructuras narrativas con una complejidad suficiente, cosa que la narrativa emergente no logra, sino que además se pretende la eliminación de muchos detalles que no son relevantes para la historia. En este tipo de diseño se supone que el motor de escena filtra y guía el drama interactivo en un nivel de acción de acuerdo con la función dramática de cada una. *La capa de personaje.* Esta tiene como principal función la de dar servicios de salida para la animación, el sonido y el dialogo. Cada motor de un actor autónomo debe de tomar en cuenta la personalidad del personaje. Además debe de seguir los diálogos y direcciones de escena que el motor de escena le indique. El grado de automatización en este motor implica la habilidad del motor para adaptar la presentación al contexto de la historia. Por ejemplo, en un caso muy automatizado, el avatar del personaje es capaz de seguir direcciones de escena abstractas, dependiendo de alguna representación del espacio, y sus movimientos pueden estar parametrizados de acuerdo al estado de animo en que se encuentre el personaje. En cambio, en un motor no automatizado, el motor de personaje simplemente mostraría animaciones ya almacenadas.

Como se puede notar por los diferentes, y hasta contrarios, conceptos usados en el diseño de aplicaciones de HIC. La estandarización y aceptación de métodos comunes aplicables al diseño de estas aplicaciones todavía es lejano, por lo que para desarrollar este

API se debe de elegir un metodología en particular, ya que es imposible pensar en un API que sea pueda usar en proyectos con diferentes metodologías de diseño. Sería como querer construir una herramienta que facilita cierta labor, cuando todavía no se sabe como realizar dicha labor. Es por esto, que es necesario elegir a que tipo de diseño de aplicación va a estar enfocado el API, y por lo tanto es necesario proponer una metodología de diseño medianamente estandarizada. Aunque es necesario remarcar, que el API no se impone al desarrollador limitando la manera en la que diseña su aplicación, sino que mediante una clara delimitación de las responsabilidades de cada uno, permite que cierta funcionalidad común y necesaria en varios proyectos solo tenga que ser desarrollada una vez. Como es el caso de OpenGL, una librería gráfica que permite al desarrollador tener una interfaz con el *hardware* gráfico. Esta interfaz esta diseñada de manera independiente del *hardware*, lo que provee de operaciones de alto nivel, aunque esto puede costar un poco en términos del rendimiento del sistema.

En el caso de este proyecto, como ya se menciono anteriormente, es necesario que el API se desarrolle considerando una metodología de diseño. La elección de la metodología que este API requeriría se realizó tomando en cuenta la complejidad que cada una de éstas conlleva al implementarlas, los resultados que otros proyectos han reportado la aplicar alguna de estas metodologías, el grado de control que esta metodología permite tener tanto al implementar como al utilizar alguna aplicación con ella y el aporte general que proporciona al proceso de desarrollo de aplicaciones desde el punto de vista de la ingeniería de *software*.

Considerando todos estos aspectos, la metodología bajo la que se propone utilizar el API es la de tres capas: la de historia, la de escena y la de personaje. Esta elección se da considerando la manera en la que está metodología cumple con los requisitos antes mencionados. Por ejemplo, el implementar un API que trabaje bajo esa metodología de diseño no es demasiado complejo, comparado con otras, si se toma en cuenta que el grado de producción automatizada puede ser variado de acuerdo con pruebas empíricas que comparen la linealidad de la historia con el valor dramático de la misma. Ya que la generación automática de historia, de escena y de comportamiento de personaje se consideran como características que vuelven a la historia menos lineal, siempre es bueno poder probar hasta que grado de libertad de estas características comienzan a interferir con la funciones dramáticas que el autor deseaba poner en su historia. Y como la capacidad de prueba antes de terminar la aplicación es una de los requisitos de la metodología de tres capas, se puede suponer que un API con esta metodología propiciará la investigación del punto de equilibrio entre ambas necesidades, que a primera vista parece no existir. Otro punto que favoreció la elección de esta metodología fue que desde un principio sus proponentes tuvieron la intención de aplicar el enfoque de ingeniería de *software* al desarrollo de este tipo de aplicaciones.

3.1.2. Carencia de Base Común de Desarrollo.

Este es un problema que en mayor o menor medida afecta a toda la industria de desarrollo de *software*. Y se resume simple y coloquialmente en la frase de "reinventar la rueda". En la actualidad la gran mayoría de los paradigmas y metodologías de diseño promueven la reutilización de código con un éxito relativo, esto debido a una mal diseño de interfaces o a una implementación que resulta dependiente de las características particulares de cierto proyecto. Y es debido a la novedad, ya mencionada, de este campo que los

problemas que inhiben la reutilización de código se presentan con gran facilidad. Y especialmente debido al problema mencionado en [3.1.1] que éste problema se agrava, ya que es difícil detectar alguna parte de código que se pueda considerar como reutilizable, ya que implementa necesidades funcionales comunes a vario proyectos, sin una metodología de diseño común con la que todos hayan sido diseñados.

Es obvio que el proyecto de desarrollar un API para la creación de historias interactivas surge con la intención de resolver este problema. Ya que se considera que un campo del desarrollo de *software* en el que se tiene que crear herramientas ad hoc cada vez que se inicia un proyecto es un campo en el que no se ha alcanzado una maduración plena. Claro que puede haber proyectos que por su necesidades específicas tengan que desarrollar sus propias herramientas, pero para un campo, como el de las historias interactivas, es mejor que éstos sean las excepciones y no la regla.

Además, la falta de reutilización de código y la inexistencia de herramientas comunes de diseño, produce otros problemas en un campo tan joven como lo es el de las historias interactivas por computadora. Esto debido a que el tiempo de investigación y desarrollo de cualquier proyecto se ve reducido al tener que implementar funciones que son necesarias en todos los proyectos de HIC y que un diseño totalmente incompatible con el que nuestra aplicación se está usando impide la reutilización de código utilizado en proyectos anteriores. De la misma manera sucede con las herramientas que se tienen que desarrollar y que usan tiempo que se podría aplicar a solucionar problemas como los mencionados en los incisos siguientes. Se puede ver entonces que al desarrollarse un API que solucione este problema no solo se facilita la labor del desarrollador de HIC sino que permite que éste realice la investigación necesaria para solucionar otros problemas que afectan a este campo.

3.1.3. Carencia de un Nivel Adecuado de Interactividad.

El hecho de que desarrollo de las historias interactivas esté influenciado por otros campos del entretenimiento como lo son los videojuegos, la cinematografía y la narrativa escrita ha propiciado que el grado de interactividad, capacidad que tienen las acciones del usuario para afectar a los eventos de la historia, que se le presenta a un usuario sea muy difícil de definir. Esta dificultad se debe a la noción de que este grado de interactividad se haya en un punto intermedio entre la interactividad que presenta una novela o una película, que es ninguna, y la interactividad de un videojuego, donde el jugador es el responsable de la generación de la historia, o al menos la manera en la que se dan los eventos de la misma. A primera vista se podría pensar que el problema no presenta dificultad alguna, hacer algo que sea mas interactivo que una película pero menos que un juego. En realidad, el problema surge de la autoría y calidad de la historia producida por un sistema con un grado de interactividad "entre una película y un videojuego".

Obviamente el problema no es tan simple, ya que al considerar que en un sistema interactivo las acciones del usuario modifican la manera en como se dan los eventos de la historia, la acreditación de la autoría de una historia se vuelve un concepto difuso. En un medio narrativo tradicional como un libro es obvio que la autoría se le puede acreditar al escritor. En el caso de un sistema interactivo puede que todas las historias presentadas al usuario, y que surgieron según las diferentes acciones de éste, en realidad todas habían sido

creadas previamente por el autor. Esto gracias a que la cantidad de acciones relevantes a la historia que se permite que el usuario tome en realidad es pequeño, permitiendo que el autor desarrolle cada una de ellas. Este tipo de implementación no produce un sistema donde el usuario pueda decir que sus acciones tengan una verdadera influencia en los eventos de la historia, dando lugar a experiencia interactivas muy reducidas. Para evitar esto, se puede pensar en permitir que parte de la autoría caiga en el sistema o en el usuario, la problema de hacer esto es que se puede afectar la calidad de la historia que se presenta al usuario. Se dice que la calidad se puede ver afectada, porque él responsable de producirla, el autor, es alguien debe saber de las características de una historia dramática y con intriga. Si parte de esta responsabilidad de pone en el usuario o en el sistema, como lo implica la interactividad, la capacidad de producir una historia que cumpla con estas características se vuelve mas complicada. Lo que hace que una historia se vuelva dramática implica que un sistema que intente generarlas deba comprender, o al menos manejar, conceptos muy abstractos. En el caso de que la responsabilidad de generar la historia se coloque en el usuario, puede que éste tampoco pueda generar una historia con mucho valor dramático, no todos somos escritores, además de que muy probablemente causará que la historia carezca de todo atisbo de intriga.

3.1.4. No Aplicación de los Principios del Drama.

En conjunto con el problema anterior, la falta de aplicación y conocimiento de los principios del drama por parte de los desarrolladores de HIC ha causado que la calidad dramática de las historias interactivas sea generalmente mala. Esto a llegado al extremo de volverse la calidad esperada de los sistemas interactivos. El problema, como es de esperarse, no es de fácil solución. Pero si se desea crear sistemas realmente interactivos, donde las acciones del usuario realmente tengan relevancia en la trama de la historia, mientras ésta mantiene su valor dramático y su coherencia. Es necesario el desarrollo de sistemas que generen la historia en respuesta a las acciones del usuario y donde los conceptos y la teoría acerca de la narrativa y de lo que hace a una historia algo verdaderamente dramático este presente.

Pero aún al aplicar estos conceptos, la solución del problema se mantiene distante. El final de cuentas, en palabras llanas, se requiere de un sistema capaz de hacer lo que un escritor hace, además de poder cambiar ese trabajo en cuestión de instantes debido a alguna acción del usuario. De cualquier manera, la creación de un marco teórico y la sistematización del conocimiento en las HIC son acciones necesarias para solucionar problemas como éstos. De la misma manera que la Inteligencia Artificial tuvo que abandonar el optimismo y la heterodoxia que la caracterizaron en sus inicios, el campo de las historias interactivas tendrá que pasar por un periodo de formalización. En el cual también se puede descubrir que muchas de las cosas que se pensaban fáciles de realizar, en realidad, implican más de lo que se consideraba. De cualquier manera, la falta de formalidad en el enfoque de narrativa utilizado hasta ahora es un problema que se resolverá cuando los equipos de desarrollo de HIC se vuelvan mas interdisciplinarios e incluyan a personal del campo de la narrativa y la lingüística. Para un análisis mas profundo acerca de los problemas que aparecen en equipos interdisciplinarios que desarrollan historias interactivas consúltese [CRA02].

3.2. Características de un API para HIC.

Para realizar un acercamiento deductivo a las características que el proyecto de Calakmul virtual debe tener, en lugar de simplemente listarlas, se partirá de las necesidades mas generales, que todo sistema de historias interactivas por computadora requiere tener. Este tipo de acercamiento, además tiene la ventaja propiciar que las necesidades de otros proyectos de historias interactivas también se vean incluidos, aunque el énfasis se de en satisfacer los requerimientos del proyecto Calakmul virtual.

Las características y necesidades funcionales que a continuación se expresaran, provienen de un análisis conceptos expresados en el capítulo uno. De la diferencia entre la historia y la narrativa es que surgen un buen número de las características, que en general, un API de HIC debe de aportar para poder llamarse como tal. Otras de las características de un API de HIC aparecen del hecho que es mas importante mantener el carácter dramático de la obra a tener la mejor inteligencia artificial controlando a los actores de la historia. La otra característica trascendental de un sistema de este tipo viene en el nombre mismo, es la interactividad que el API permite, que tan bueno es el API para cambiar los eventos que se suceden debido a las acciones del usuario, que es la audiencia, mientras mantiene una trama coherente y de alta calidad dramática.

Y es que las cualidades mencionadas son de vital importancia para cualquier sistema de HIC, y para Calakmul virtual en particular, ya que sin una calidad dramática adecuada, por muy importantes que sean los hechos históricos que el guía virtual narre; el usuario puede perder rápidamente la atención en lo que el guía dice . Si el guía simplemente dice lo que sabe, se convertirá en una enciclopedia con interfaz 3D, donde los datos, las cifras y las citas simplemente serán lanzadas al usuario. Produciendo un efecto negativo en lo que la audiencia pueda aprender al usar Calakmul, siendo que el ámbito educativo es principal campo de aplicación de éste. De la misma manera, la coherencia de la trama es un punto muy difícil de lograr con la inteligencia artificial con la que se pueden dotar los agentes hoy en día, por lo que el API debe de contar con otros medios que proporcionen esta funcionalidad sin necesidad de recurrir a una inteligencia artificial muy complicada.

A continuación se listan las características que se considera que todo API de HIC debe de alguna manera u otra presentar para poder llamarse así. Esto debido a que estas características se definieron tomando en cuenta las necesidades que surgen de las definiciones del capítulo uno.

3.2.1. Diferenciación entre la Narrativa y el Mundo de la Historia.

Esta característica es la de mayor trascendencia, y una de las que todo sistema que quiera llamarse de historias interactivas debe de cumplir. Pues ésta permite que la historia se desarrolle de la misma forma como sucede cuando es narrada de manera tradicional. En la narrativa tradicional lo que sucede en el mundo de la historia nunca es totalmente contado, o la menos no todo es contado en un orden secuencial. Ya que si esto sucediera, en lugar de narrativa, que debe de contar con cierto valor dramático para poder atraer la

atención del público, se trataría de un simple listado de eventos de lo que pasa en el mundo de la historia. Los eventos que la narrativa presenta muestran la intención del autor y mantienen al público en un estado activo, ya que el público debe de estar analizando las acciones que le son presentadas y suponiendo que paso en aquéllas que no.

Traduciendo esto a características de un sistema de *software* significa que el sistema debe de ser capaz de presentar solo ciertos eventos, la manera en la que el sistema los presenta se abordará mas adelante, mientras que mantiene el estado de todas las entidades que existen en el mundo de la historia. Obviamente, el estado de las entidades del mundo de las historia deben de cambiar su estado, tanto con los eventos que el narrador "virtual" presenta al usuario, como con los eventos que suceden en el mundo de la historia de los que el usuario nunca se entera.

Esta característica que todo sistema de HIC debe de tener y que por ende cualquier API de HIC debe de presentar tiene muy diversas formas de implementarse, en 3.2. se presentara la forma como esta característica se necesita implementar específicamente para el proyecto Calakmul virtual. Además, hay una gran cantidad de medios para narrar una historia, y que también se pueden aplicar para los sistemas de historias interactivas. Las características particulares que el narrador virtual, la parte del sistema de HIC que presentara los eventos al espectador, debe tener para Calakmul virtual también se presentan en [3.2.].

3.2.2. Posibilitar que las Acciones del Usuario Modifiquen los Eventos de la Historia.

La historias interactivas se deben, en un primer momento, al desarrollo de la industria de los videojuegos. Una gran familia de juegos altamente interactivos, pero con una historia de poco valor dramático, pobló las primeras generaciones de videojuegos. Conforme el mundo de los videojuegos se profundizaba, también lo hacia la necesidad de historias mas profundas. Debido al éxito del mercado de juegos, en especial de los videojuegos de computadora, surgió la intención de aprovechar las crecientes capacidades gráficas de la computadora con propósitos educativos. Es de estas dos vertientes que surgió la idea de aprovechar las capacidades de las computadoras para probar sistemas en los que el interés se centrara en la historia y en la posibilidad del usuario de cambiarla.

La manera como estas características se deben implementar en un sistema de *software* debe permitir que las acciones que se suceden en la historia puedan ser definidas de manera sencilla por el desarrollador. El sistema además de llevar cuenta de los eventos que ya se han presentado y de los que todavía faltan, debe de poder manejar las entradas que el usuario hace al sistema con la intención de modificar esos eventos.

Es claro que la manera como las acciones de los usuarios cambian la sucesión de eventos de la historia dependerá del modelo de interactividad que se use. Los modelos de interactividad mostrados en el 1.2. tienen obvias diferencias en la cantidad de eventos que el usuario puede modificar y en la frecuencia en la que puede hacer esto. Aun así, esto se puede considerar como una característica general que todo sistema de historias interactivas

necesita, ya que en todos existe la necesidad de describir cada evento y de indicar la manera en la que se sucederán éstos.

Es apropiado aclarar que esta característica puede ser implementada, en algunos casos, en la aplicación misma en lugar de en el API. El problema con este tipo de diseño es que impide cualquier intento por reutilizar el sistema. Ya que reutilizarlo para otra historia implica el cambiar el código del mismo. Echando por tierra cualquier posibilidad de reutilización. Por ejemplo, en los sistemas de historias interactiva que se utiliza el modelo de ramificación los eventos de la historia se pueden programar en el código del sistema mismo. Mediante el uso de *ifs o cases* para cambiar el curso de las acciones de la historia, que también están codificadas en el sistema mismo, se logra que el código del sistema contenga a la historia. Por obvias razones este tipo de diseño no se recomienda para ninguna solución.

La implementación recomendable para esta característica implica el soporte de algún formato de archivo de datos en el que se puedan describir y almacenar los eventos, las condiciones en las que éstos se darán, así como la manera en la que el usuario los puede modificar. Generalmente, para describir la sucesión y los tipos de eventos es preferible usar algún tipo de lenguaje. Esto debido a que un lenguaje de programación provee de mayor flexibilidad comparado con los archivos de datos. De cualquier manera, la decisión de utilizar alguna de estas dos opción es indiscutiblemente superior a la de poner la historia en el código del propio sistema.

Otro detalle a remarcar, es que esta característica de un API para HIC implica el soporte eficientemente de un método de entrada de datos, en él que los usuarios puedan indicarle al sistema que cambios quieren hacerle al desarrollo de la historia. Este soporte de entradas de usuario también es dependiente del tipo de interactividad y de la forma en que el sistema presenta la narrativa al usuario, por ejemplo, un sistema de narrador virtual en el que la interactividad se da mediante ramificaciones, requiere de un sistema muy básico para la captura de las entradas del usuario. En cambio un sistema en el que la historia es presentada en un ambiente 3D con un modelo de interactividad reciente, como el de las funciones narrativas, requiere de un sistema avanzado para la captura de las entradas del usuario.

3.2.3. Capacidad del Autor de Controlar la Presentación de los Actos.

De lo mencionado en el capítulo 1, se puede intuir que lo importante al presentar una historia es el contar las acciones que se dan en el mundo de la historia de manera interesante para el espectador, ya sea al presentar los evento en un orden diferente al que se dieron o al omitir algunos de ellos. De esto se puede ver que la capacidad que tenga el autor de la historia interactiva para controlar al narrador será un punto decisivo para el desarrollo de HIC de calidad. Ya que es mediante el narrador como el espectador se entera de lo que sucede en el mundo de la historia que es mantenido por el sistema mismo. Los medios mediante los cuales el narrador se comunica con su espectador son muchos: lenguaje natural, video, símbolos, imágenes en 2D o en 3D, etc. De manera que la flexibilidad y la eficacia para controlar al narrador dependerán del medio por el cual se comunique el

narrador con el público, ya que son diferentes las formas como se van a aprovechar y controlar las características de cada forma de comunicación.

De la misma manera que en el punto anterior, la forma mas eficaz y sencilla para controlar al narrador virtual es mediante algún tipo de lenguaje de programación, es mas, se puede utilizar el mismo lenguaje con el que se controlan los eventos de la historia. Lo recomendable sería que mediante este hipotético lenguaje de control del narrador se le pueda indicar al sistema que eventos mostrar y cuales no, el orden en que éstos se dan y la manera como deben de ser presentados. Es en este último punto donde toma importancia el medio a través del cual el sistema nos presenta la historia. Por ejemplo, si el sistema presenta la historia en un ambiente 3D, sería recomendable que lenguaje de control de presentación permitiera un control de cámara similar al que se tiene en el cine. Por otro lado, si la manera como se presentan las historias es a través de un narrador virtual que utiliza un sistema de generación de texto y de conversión de texto a voz, sería recomendable que el lenguaje permitiera controlar la entonación, el ritmo y la intención con la que el narrador virtual presenta los eventos de la historia.

En el proyecto de Calakmul, en particular, se debe de considerar que el narrador virtual, o mejor dicho el sistema que presenta los eventos, es a través de gráficas en 3D que son generadas mediante un motor de juegos. Las particularidades del sistema de presentación del proyecto Calakmul virtual se listan a detalle en el próximo capítulo.

3.3. Características de un API para Calakmul Virtual.

Después de revisar las características que un API de HIC debe de satisfacer en lo general para poder considerase como tal. Se analizaran las características particulares del proyecto Calakmul virtual. En algunos casos estas características son casos particulares de las características ya mencionadas. Además se presentan algunas otras características que se requieren del API y que surgen debido a necesidades particulares del proyecto. El proceso deductivo que se utiliza para obtener las características particulares del proyecto tiene como propósito lograr que el diseño del mismo facilite su utilización en otros proyectos, mientras que satisface las necesidades particulares de Calakmul virtual.

A continuación se muestran las características que el API de HIC que se va a usar para implementar Calakmul virtual debe de cumplir.

3.3.1. Clase Mundo para Contener a Todas las Entidades de la Historia.

Esta característica del API de HIC que será usada en el proyecto Calakmul virtual es una particularización de la característica general mostrada en [3.1.1.]. Se requiere el uso de una clase Mundo porque el proyecto Calakmul virtual esta desarrollado en un motor de juegos hecho en C++, y para aprovechar las ventajas del enfoque orientado a objetos es necesario contar con una clase que contenga las referencias a todo lo que esta en el mundo de la historia. Además desde el punto de vista técnico, toda entidad que sea agregada a la clase Mundo entrara en el ciclo de actualización del sistema. Este patrón de diseño además

asegura la separación entre los objetos que están activos y los que existen en el mundo de la historia.

Es decir, el aspecto principal de esta característica es que las clases que la clase Mundo puede contener son objetos que pertenecen a las clases que mantienen el estado de las entidades de la historia, no de sus representaciones gráficas. En especial, se puede aprovechar que esta clase tenga referencia a todas las entidades de la historia para asignarle el comportamiento de actualizar el estado de todas ellas.

3.3.2. Clase Escena para Contener las Representaciones Gráficas.

Esta otra característica que también surge como forma particular de [3.1.1.]. De la misma manera que se requiere de la clase Mundo para que contenga los objetos y a los estados de los mismos, también se requiere de una clase Escena que contenga las clases con la representación gráfica de los objetos que se presentan al usuario. Cabe remarcar que la clase Escena no necesariamente contendrá la misma cantidad de entidades que la clase mundo, esto se debe al hecho que lo que se le presenta al usuario mediante las escenas de la narrativa, no contiene a todos los objetos de la historia al mismo tiempo. En cambio, los objetos que son contenidos por la clase Mundo, son todos los objetos relevantes que existen en el mundo de la historia, haciendo necesario que se mantenga su estado a lo largo del funcionamiento del sistema. En especial, los objetos que contenga la clase Escena deberán de permitir el posible encendido y apagado de la visibilidad de los mismos. Esta particularidad del funcionamiento es necesaria para facilitar que solo los objetos y entidades requeridos por la narración sean los que aparezcan en las diferentes escenas de la narrativa.

De esta manera la clase Escena junto con una clase Cámara son los objetos que permitirán controlar la narrativa. Pues mediante un objeto de la clase Cámara es que se establece la manera y la posición desde la cual se presenta la escena al usuario, mientras que la clase Escena controla lo que aparece al usuario, mostrando solo las entidades que estén contenidas en ellas y que tengan el atributo de visible activado.

3.3.3. Presentar en un Ambiente 3D en Tiempo Real el Mundo de la Historia.

El proyecto Calakmul Virtual tiene la intención de mostrar el sitio arqueológico de Calakmul a los usuarios, simulando una visita a guiada al lugar. Como parte de esa experiencia se requiere que el API soporte el mostrar modelos 3D detallados de los diferentes edificios que se encuentren en el lugar. Además se requiere que el API permita al usuario navegar alrededor de todos ellos.

De estos requerimientos es que surge la necesidad de integrar un motor de juegos al API, porque al hacerlo se pueden mostrar ambientes 3D e interactuar con ellos de una manera sencilla. Esta característica además presenta el reto de integrar el API del motor de juego con el API de HIC de una manera intuitiva, que no interfiera con la facilidad de uso y con la uniformidad de la interfaz del API.

Un punto particular de este requerimiento es la necesidad de presentar un guía virtual que nos lleve a través del sitio arqueológico y que nos vaya explicando la historia y

los usos de cada lugar. Esta necesidad requiere que el motor de juegos que se escoja tenga la capacidad de mostrar modelos de figuras humana con animaciones básicas. Además de mostrar los modelos de los edificios y de los personajes que están en la escena, el motor debe de ser capaz de informar al desarrollador de la interacción entre ellos. Esto implica que el API también debe de presentar un sistema de detección de colisiones entre los modelos de todo lo que aparece en pantalla.

3.3.4. Control Intuitivo y en Tiempo Real del Agente del Usuario.

Está característica que el API debe de presentar y que es un caso particular de [3.1.2] implica que el usuario mediante entradas con el teclado y el ratón pueda controlar acciones sencillas del agente que lo representa en el ambiente virtual tales como el moverse a través de él, tomar objetos e interactuar con otros personajes que estén en el mundo de la historia y que son controlados por el sistema. En un primer acercamiento esta característica del API puede se satisfecha mediante un control muy similar al de los actuales juegos de computadora, donde el control para navegar a través del ambiente se da mediante las flechas del teclado mientras que con el ratón se controla hacia donde apunta la vista del personaje o con que objeto se quiere interactuar. Futuras versiones pueden incluir mecanismos de control de mas alto nivel, en los que la inteligencia artificial y el reconocimiento de voz son requerido, como por ejemplo un mecanismo en el que simplemente se le diga al agente lo que tiene que hacer y éste cumpla la orden de acuerdo con lo que sus algoritmos de planeación indiquen. Pero cabe aclarar que aun dado el caso que se puedan implementar mecanismos de control avanzados, como el mencionado de reconocimiento de voz, la implementación mediante ratón y teclado ha probado ser propiciadora del factor de inmersión, ya que el usuario siente que esta ahí gracias a la respuesta inmediata que el agente tiene a sus entradas, mientras que con los mecanismos mas avanzados, donde el usuario le da comandos de alto nivel al agente para que éste planee como realizarlos y luego los ejecute tienen la desventaja de reducir la inmersión del usuario. Causando que sienta que no esta ahí, sino que está guiando el viaje de alguien mas.

Para precisar esta característica de control, se tiene que tomar en cuenta que la necesidad de respuesta en tiempo real implica que el usuario sienta que el agente que lo representa, actúa justo en el momento en el que se le indica. Por ende esto requiere que la carga que el sistema de historias interactivas haga a la computadora no sea mayor a la que ésta puede soportar. Ya que esto causaría un retraso entre la entrada del usuario y la respuesta del agente. El problema con esta característica es que a menudo no se puede saber cual va a ser la respuesta del sistema sino hasta que ya se esta corriendo es situaciones de carga de trabajo reales. De ahí que es recomendable que el API, como parte del sistema de historia interactivas, permita al desarrollador el afinar su rendimiento mediante la ponderación de algunos parámetros, como lo son la resolución de la pantalla, el nivel de detalle, etc.

3.3.5. Soporte de un Lenguaje de Script que Permita el Control de los Objetos en el Mundo.

Como parte de la necesidad de interactividad, que las acciones del usuario cambien el curso en el que se dan los sucesos de la historia, se requiere de una manera sencilla para indicarle al sistema la manera como se darán esos eventos. Además, esta característica de

ninguna forma, debe de atar al sistema para que sea reutilizado con otras historias. Es decir, que no implique la manera en la que se dan los eventos este codificada en el propio sistema. Estos requerimientos implican que se le pueda decir al sistema, de una manera sencilla, como se deben de dar los eventos en la historia, eventos que implican a los objetos de la historia, sin tener que cambiar el código del sistema en si mismo. Es ahí donde entra la necesidad del lenguaje de Script con acceso a los objetos del sistema. Ya que un lenguaje de script permite la misma flexibilidad y control que los lenguajes compilados con los que se construye el sistema, pero sin la necesidad de la recompilación, son el medio ideal para controlar la manera en como se dan los sucesos en la historia.

Analizando que implica esta característica para el API, se puede ver la necesidad de un interprete que lea los scripts, para que después, mediante una interfaz llamen a los métodos de los objetos de la clase Mundo. Además, implica que el sistema debe de tener características multihilos, ya que debe ser capaz de mantener el sistema actualizando sus objetos, mientras que el interprete analiza los archivos de script que controlan la historia.

Otra ventaja de esta implementación del sistema que controla los eventos que suceden en la historia es que mediante políticas de acceso, que indican a que objeto puede controlar un script, se pueden idear sistemas en los que se usen unos scripts para controlar el flujo general en el que se dan los eventos, mientras que se pueden implementar otros scripts que controlan el comportamiento individual de un objeto en un evento en particular. Por ejemplo, se pueden crear scripts que controlen lo que un objeto en particular debe de hacer cuando choca contra el avatar del usuario, mientras que otro script, mas general, controla la historia de la visita. Este esquema de scripts además tiene la ventaja de permitir reutilizar ambos tipos de script según sea necesario. Por ejemplo, si se requiere cambiar la historia, el script mas general puede ser cambiado, de la misma manera el script que controla el comportamiento particular de un objeto al chocar con el avatar del usuario puede ser reutilizado con la nueva historia, o pude ser cambiado según las necesidades del desarrollador. Eso sí, sin haber recurrido nunca a modificar al código del sistema. Mas acerca del uso de scripts para el control de historias interactivas en [PER96].

3.3.6. Clase Cámara Controlable Desde el Lenguaje de Script.

En API debe de proveer al usuario con objetos y sus respectivos comportamientos tales que cubran las necesidades funcionales comunes a las de la mayoría de los usuarios de éste. En el caso de la clase Cámara, ésta debe de implementar los comportamientos básicos que todo desarrollador desearía para presentar ambientes 3D. Con esto se quiere decir que la clase Cámara deberá de poder realizar todos los tipos de tomas que una cámara realiza en la realidad y que los creadores cinematográficos y televisivos emplean para narrar sus historias de una manera dramática. Y es que estos diferentes tipos de tomas permiten al autor enfatizar lo que cierto personaje esta haciendo, sintiendo o diciendo. Permiten que realice tomas donde solo muestre vistas parciales de la escena, con la intención de que sea el espectador el que tenga que suponer que pasaba en la pare de la escena que no podía ver. Los tipos mas comunes de tomas, y que son comportamientos que se espera que la clase Cámara implemente, son: la persecución, la de primera persona, la que sigue un camino, la que apunta, etc.

Y es que es importante remarcar que si en una historia transmitida oralmente el narrador es el que se encarga de transmitir los eventos de una manera que sean interesantes para el usuario, con la posibilidad de dar énfasis a ciertos eventos. En las historias donde las acciones son presentadas a los usuarios de manera directa, como en el cine, es necesario que la cámara pueda ser controlada para transmitir los eventos con la misma flexibilidad y expresividad con la que un narrador lo hace oralmente. Además, un objeto que pertenezca a la clase Cámara también tendrá que cumplir con las características de [3.2.5] haciendo factible el controlar las acciones que se darán en la historia así como la manera como éstas serán presentadas con el mismo lenguaje de script.

A continuación, se profundiza un poco mas en los diferentes tipos de toma que una cámara realiza, así como los requisitos que se tienen para los comportamientos que representaran esos tipos de tomas en la clase Cámara. Una de las tomas mas comunes es la conocida como de tercera persona, o de persecución, que es donde la cámara apunta a algún objeto particular que se encuentra en la escena, manteniéndose alejada de éste siempre a una distancia constante. El comportamiento de persecución de la clase Cámara se espera que sea posible aplicarse a cualquier objeto de la clase Escena, además de que también es necesario que las distancia a la que la cámara persigue al objeto sea definida por el autor. Otro tipo de toma común es la de primera persona, en donde la cámara muestra lo que algún personaje esta viendo, permitiendo que el espectador sienta que se trata de lo que él está viendo. En este caso se espera que la clase Cámara también pueda realizar este tipo de toma desde cualquier objeto que se encuentre en la escena. Además en la toma de primera persona al igual que en la de tercera, se requiere que la cámara actualice su posición y su orientación de acuerdo con la posición y la orientación del objeto que sea su objetivo, todo esto sin que el usuario tenga que manejar las tomas por él mismo. Para ponerlo en palabras mas claras, el comportamiento de la clase Cámara debe d e satisfacer lo que en la vida real haría la cámara junto con el camarógrafo. Para concluir, otro tipo de toma que se debe de incluir es en el que la cámara sigue un camino definido por el autor, mientras que la cámara apunta a algún objeto en particular que se encuentre en la escena, aun cuando el objeto que se encuentre en la escena también cambie de posición.

3.3.7. Clase Entidad que Permita el Uso del Polimorfismo con Todos las Clases de los Objetos del Mundo.

Ya que el mundo en el que la historia se va a desarrollar contendrá objetos de diversas clases, que además se crearan en tiempo de ejecución de manera imprevisible por el desarrollador, es necesario que todos éstos tengan algún comportamiento e interfaz común. De esta premisa es que surge la necesidad de que el API contenga una clase de la que todas las clases de los objetos que van a existir en el mundo de la historia deben heredar, y un número básico de funciones que deben de implementar.

El propósito de la mencionada clase Entidad será el obligar a que todo objeto que se agregue al mundo de la historia cumpla con un mínimo de funcionalidad necesaria, además de facilitar la interacción entre el mundo y todos los objetos que contiene así como entre ellos mismos. La funcionalidad mínima de la que se habla son tres métodos que la clase Entidad, una clase base, contiene. Estos métodos representaran el comportamiento para actualizar el estado interno del objeto, para detectar su interacción al colisionar con otros objetos, y para actualizar su posición en la escena. Del tipo de clase a la que pertenezca el

objeto dependerá el comportamiento esperado. Por ende, cada clase implementará de manera diferente cada uno de estos métodos.

El obligar a que todo objeto que exista en el mundo de la historia pertenezca a la clase Entidad o a alguna clase que hereda de ésta presenta una gran ventaja en la interacción entre objetos. Esto debido a que cualquier objeto que tenga que interactuar con otro objeto puede estar seguro de que éste, al menos, puede ser convertido a la clase Entidad. Y debido a que se está utilizando un diseño orientado a objetos, el comportamiento que este objeto realizará, no será el de la clase Entidad, sino el de la clase a la que pertenece, esto gracias al polimorfismo.

Estos métodos además serán el medio a través del cual todos los objetos serán actualizados por el objeto de la clase Mundo que los contiene. Es a través de estos métodos que serán llamados en un ciclo de actualización, que todo objeto modificará su estado, calculará la manera en que su interacción con otros objetos los afectará, para que finalmente actualicen su posición.

Una última ventaja que esta característica de diseño presenta, es que facilitara el almacenar y agregar objetos a la clase mundo, ya que la estructura de datos que internamente permita al objeto mundo contener a todos los objetos que se construyan y luego se le agreguen, puede ser algún tipo de lista ligada sencilla, que contenga las referencias a objetos de la clase Entidad. Esto, en lugar de preocuparse por crear una lista ligada que tenga que mantener las referencia a todo tipo de clase a la que un objeto pueda pertenecer. Además representa una ventaja al agregar una nueva clase, ya que simplemente necesita heredar de la clase Entidad para poder ser agregada al mundo, en lugar de tener que modificar el código de la lista ligada para soportar una nueva clase cada vez que una de éstas es creada.

3.3.8. Funcionalidad de Alto Nivel para el Control de los Personajes.

Como se mencionó en el punto anterior, el API debe de presentar una clase que contenga los comportamientos mas básicos requeridos por todo objeto que exista en el mundo de la historia. Pero, además es necesario que el API contenga una serie de clases, cuyas instancias son los objetos que pueden existir en el mundo de la historia, y que por lo tanto deben de heredar de la clase Entidad; agregándole comportamientos especializados según los requisitos de cada una. Al hablarse de una serie de clases se hace especial referencia a las clases de los actores, controlados por la computadora o por el usuario, del los objetos y de los modelos de la escena; que son totalmente necesarias para presentar una historia interactiva.

Pero el requisito de que el API tenga esta clases, no está cumplido por su simple existencia, sino que además se le debe de dar énfasis al alto nivel de las instrucciones con las que éstas serán controladas. El comportamiento requerido para cada una de estas clases es muy diferente, pero bastante claro según el nombre de cada clase, lo importante es que este comportamiento se pueda usar de manera sencilla, con llamadas a métodos que claramente impliquen un comportamiento asociado con cada clase. Por ejemplo, para la clase de actor controlado por la computadora debe de ser posible mediante una simple llamada a un método, el indicarle que se dirija hacia el lugar que se le a indicado, siendo

responsabilidad de la clase el seguir el camino que debe de seguir, mientras que el modelo mostrado realiza una animación de caminar.

Y es que el requisito de que las instrucciones sean de alto nivel, tiene como propósito el permitir que los objetos pertenecientes a estas clases puedan ser controlados por desarrolladores con poca experiencia en la programación mediante scripts. Con esto se busca propiciar que los equipos de desarrollo de historias interactivas puedan estar compuestos de una manera mas heterogénea. Esto quiere decir, que se espera que al permitir controlar a los objetos mediante métodos de alto nivel accesibles a través de scripts se facilite el acceso de miembros artistas, creadores de contenido y con mayor experiencia en las necesidades de una historia, a los equipos de desarrollo de historias interactivas. Además, él que estas clases se diseñen de manera correcta y con métodos que indican de manera explicita el comportamiento de una clase, hasta facilitara el trabajo de los miembros programadores del equipo al permitir una integración llana de todos los elementos.

Para finalizar, cabe aclarar que al hablar de estas clases, no se hace referencia a las representaciones gráficas que ve el usuario. Estas son las clases, que por estar contenidas en el objeto mundo, mantienen el estado en el que se encuentra algún objeto, además proporcionan el comportamiento esperado de esa clase dado el estado en el que se encuentre el objeto. En estas clases se tendrá simplemente una referencia a otro objeto, que será su representación gráfica, contenida en el objeto escena. Volviendo al ejemplo del método caminar para la clase actor, ésta simplemente llamara al método de animación contenido en el objeto gráfico, mientras que la clase Actor será la que actualiza su posición al ir moviéndose.

3.3.9. Soporte de una Simulación Básica de la Física del Mundo.

Además de los requisitos relacionados con las características narrativas del sistema, es necesario que el API presente ciertas características funcionales que sirvan para mostrar un comportamiento concordante con las leyes físicas mas básicas. El porque de este requisito del API, es bastante claro, la necesidad de que toda acción que los personajes y los objetos existentes en el mundo de la historia deben de subordinarse y realizarse siguiendo las leyes físicas, esto con miras a mantener el realismo de éstos. Además, esta característica debe estar implementada de una manera totalmente invisible para el desarrollador, es decir que el comportamiento de todo objeto de cualquier clase del API debe regirse por las leyes físicas básicas sin necesidad de que el desarrollador tenga que realizar alguna programación explicita.

La leyes físicas básicas que se espera que el API implemente son: la gravedad, el de la imposibilidad de que dos cuerpos no ocupen un mismo lugar, la inercia, la acción y la reacción de fuerzas estáticas, y algunas características básicas del comportamiento de la luz. A continuación se explica de manera un poco mas detallada lo que se implica con cada una de las leyes físicas mencionadas. La necesidad de la existencia de la gravedad implica que los cuerpos deben de ser atraídos hacia el suelo en un movimiento acelerado en cualquier momento y en cualquier lugar de la escena, sin necesidad de que el desarrollador tenga que programar algo. Tan solo con colocar un objeto en una posición elevada, éste debe de caer al estar bajo la fuerza de gravedad. En algunos otros sistemas de HIC se puede desear cambiar el valor de la gravedad o incluso su desactivación, a este respecto el API si

permite la modificación del valor o la desactivación de la fuerza de la gravedad, pero la opción por defecto será la de un valor de la fuerza de gravedad completamente normal, ya que no se tienen ninguna intención de alterar esta ley física en el proyecto Calakmul virtual. La implementación de la imposibilidad de que dos cuerpos ocupen un mismo lugar implica la existencia de un sistema de detección de colisiones capaz de activar, cuando las superficies limites de dos objetos se tocan, fuerzas reactivas que eviten que éstos lleguen a ocupar el mismo lugar. De esta forma cuando un personaje trate de atravesar una pared, simplemente se espera que éste no sea capaz de hacerlo, sin ninguna otra intervención del desarrollador que poner la pared en primer lugar. Que el API implemente la inercia, ley a la que está sometido todo objeto real, implica que un cuerpo en movimiento continúe en ese estado hasta que alguna fuerza lo detenga, de la misma manera con un objeto estático. Cabe recordar que también se espera que la inercia se aplique a la fuerza de gravedad. El requisito de fuerzas reactivas de carácter estático y dinámico es indispensable con la existencia de gravedad, ya que sin estas, el piso sería incapaz de sostener a cualquier objeto bajo el efecto de la gravedad. Para explicar esto, simplemente se debe de recordar que cualquier objeto que se encuentra en reposo cuando está bajo la influencia de la gravedad se debe a que el piso aplica una fuerza reactiva a ésta en sentido inverso. Como último punto, cuando se habla del un comportamiento básico de la luz, implica sobre todo características tales como el sombreado y la proyección de sombras, sin lugar a dudas como un comportamiento esperado y que da mayor realismo a lo presentado por el sistema. Por ejemplo, cualquier representación gráfica de un objeto o personaje que se agregue a la escena en cualquier instante, debe de proyectar sombra e iluminarse de acuerdo con la configuración de las luces de la escena, sin ningún otro requisito al desarrollador mas que el haber agregado el objeto a la escena.

3.3.10. Soporte de Funcionalidades Básicas de Interfaz de Usuario.

Las características que necesita cumplir el API no solo se limitan al soporte de la narrativa y en al de la simulación, sino que también incluyen aquéllas que permitirán que el usuario pueda comunicarle al sistema de manera sencilla sus datos de entrada. En particular, es necesario que el API posibilite el utilizar una interfaz gráfica 2D, aún cuando se esté en un ambiente 3D, como medio de entrada. Esta interfaz gráfica de usuario (GUI) es necesaria principalmente para transmitir ordenes complejas y altamente flexibles a los personajes controlados por el sistema. En específico, la GUI permitirá que el usuario pueda indicarle al guía de turistas de Calakmul virtual a donde desea ir de una manera mucho mas cómoda y sencilla que mediante algún tipo de comandos y de una manera mas confiable que utilizando algún tipo de reconocimiento de voz, ya que esta tecnología todavía se encuentra en desarrollo.

Una parte de este requisito en el que debe de ponerse atención, es en la necesidad de que las ventanas y cuadros de dialogo en 2D puedan aparecer en alguna parte de la pantalla, mientras que en la otra se sigue mostrando la escena 3D de Calakmul. Este requisito tiene la intención de mostrar las ventanas de la GUI como los "diálogos" o los "globos" que aparecerían en el arte secuencial. En especial es necesario que el GUI que se construya con el API soporte las ya que mediante ellas se espera que el usuario pueda indicar a los personajes cual es la opción que desea dentro de una cantidad limitada de opciones.

3.3.11. Funcionalidad Básica de Búsqueda y Seguimiento de Caminos.

Este requisito que esta enfocado a los personajes controlados por el sistema, tiene una aplicación de particular importancia en el guía de turistas virtuales. Como se ha mencionado en las característica [3.3.8] para facilitar el trabajo del desarrollador, es necesario que la funcionalidad que los personajes presenten sea de alto nivel. Al considerar las necesidades particulares de este proyecto, es indudable, que una funcionalidad de alto nivel que el API debe de presentar a través de los objetos personaje es la de seguimiento y búsqueda de caminos. Esto debido a la gran cantidad de veces que un personaje controlado por el sistema de Calakmul nos llevará de un lugar a otro, a través de un camino ya predefinido o de uno que no conoce y que debe de descubrir.

Para el cumplimiento de este requisito, como parte la necesidad de instrucciones de alto nivel, solo debe de ser necesario que el desarrollador indique que punto desea que el personaje parta y a que punto desea que éste llegue. En concordancia con el alto nivel de las instrucciones, el punto de partida y el de llegada solo deben de ser indicados mediante algún tipo de nombre que sirva como identificador, dejando libre al desarrollador de cualquier requisito de coordenada espacial para definirlos.

3.3.12. Soporte de Funcionalidad Básica de conversión de Texto a Voz.

De la misma manera que es necesario que el API permita el uso de una GUI para que el sistema le comunique al usuario de las opciones que posee y para el usuario le comunique a éste cual escoge. Es necesario que el API presente la funcionalidad de conversión de texto a voz accesible a todos los personajes de la historia. Esto con la intención de que los personajes controlados por el sistema puedan dirigir pequeñas frases, escritas por el desarrollador, al usuario. Haciendo de la sesión del usuario una experiencia mas profunda, al permitir la emisión de mensajes voz, que redundará en la mayor inmersión del usuario en el sitio arqueológico de Calakmul. En particular, se puede aplicar la tecnología de conversión de texto a voz en los saludos y despedidas que los personajes que se hallen en el sitio hagan al usuario cuando éste comience o termine de hablar con ellos.

Y es que aunque la tecnología de reconocimiento de voz se desecho como medio de entrada de los usuarios, debido al estado de relativa inmadurez del área. La tecnología de conversión de texto a voz ha tenido grandes avances, los suficientes para considerar esta tecnología como un medio para transmitir al usuario información no trascendental que tiene la función del sumergirlo mas en su experiencia.

Un punto importante de este requisito es que los desarrolladores puedan escoger de entre un rango de voces con diferentes timbres y tonos, de tal modo, que dependiendo del personajes que supuestamente esté hablando; el desarrollador pueda indicarle al API las características de la voz con las que desea que sea enviado el mensaje.

3.4. Necesidad de un Medio de Presentación Gráfica para la Narrativa Dramática en los Sistemas HIC.

En el capítulo 1 se presentaron los conceptos básicos de la narrativa, entre ellos se habló del drama, que es un tipo especial de narrativa en la que las acciones son directamente representadas ante el espectador. De esta definición podemos decir que una obra de teatro, una película y un videojuego son dramas. Es necesario aclarar que no todos los dramas son visuales, las radionovelas son un tipo de drama.

De la definición y de las necesidades del drama y del conocido efecto que tiene en el espectador la presentación de imágenes, surge la necesidad de que las historias interactivas cuenten con un medio para presentar gráficamente a los espectadores los eventos que suceden en la historia. En especial, es atractivo considerar la utilización de gráficas por computadora, si se considera el gran avance del *hardware* acelerador de gráficas 3D así como su bajo precio; esto debido a la demanda de los videojuegos. Con esto la elección de un medio gráfico 3D para presentar los eventos historia al usuario queda completamente justificada. Actualmente la mayoría de los sistemas de computo personal (PC) son capaces de presentar gráficas 3D en tiempo real de gran calidad. El uso de estas capacidades para la presentación de las historias interactivas obviamente trae consigo una espectacularidad muy apreciada por el usuario..

Los puntos en contra de esta elección también son varios, aunque hay varias soluciones que los atenúan o los eliminan. La complejidad para utilizar el *hardware* acelerador de gráficos es alta, pero con el uso de librerías gráficas como OpenGL o Direct3d que sirven de interfaz con este *hardware*, su uso se facilita de manera notable. Pero no sólo es necesario poder utilizar el *hardware* de aceleración, sino el aprovecharlo para presentar modelos 3D complejos, para solucionar esto se puede utilizar lo que se conoce como un motor de juegos, donde la carga, el manejo y la animación de modelos 3D ya está implementada. Y que normalmente utilizan las librerías gráficas arriba mencionadas para lograrlo. Por último, el presentar los eventos en un ambiente 3D, requiere que el modelo de los personajes permita que el usuario reciba sus emociones y diferencie sus acciones. Esto, mas que ser un desventaja en sí, es un requisito que implica que el desarrollador deberá de aplicar su ingenio y creatividad para transmitir al usuario lo que el personaje está haciendo sin necesidad de siempre tener un modelo altamente detallado del cuerpo humano. Ya que para esto si se puede aprovechar la imaginación y la capacidad de abstracción del usuario.

Para finalizar, la intención de esta parte del capítulo es dar una razón clara de porque se requiere utilizar de gráficas 3D como medio de presentación de una historia interactiva. Al final, se considera que las ventajas de hacerlas sobrepasa a las desventajas, en especial cuando se considera que ya hay soluciones aprovechables eliminarlas o reducirlas.

Capítulo 4.

Análisis Orientado a Objetos(OO) del API

4.1. Introducción.

Después de plantear la problemática que se presenta en la construcción de sistemas de historias interactivas y de enumerar las características requeridas de un API que busque agilizar el desarrollo de éste tipo de aplicaciones. Es necesario realizar un análisis orientado a objetos para obtener un modelo que sirva como la primera representación técnica del sistema. El modelo de análisis que surja de este proceso debe de tomar como base los requisitos que se plantearon en el capítulo anterior para identificar las clases relevantes al sistema, así como sus responsabilidades y las jerarquías a las que pertenecen. Para un análisis completo acerca de las ventajas de aplicar el paradigma orientado a objetos en sistemas interactivos véase [SKO02].

El proceso de análisis que se realiza en este capítulo, permitirá especificar la función del API de una manera mas técnica. A grandes rasgos, la estructura de este proceso es la siguiente: Se recopilan los requisitos, se identifican las clases, se especifica una jerarquía de clases, se representan las relaciones de objeto a objeto y se modela el comportamiento del objeto. El objetivo del proceso de análisis es desarrollar una serie de modelos que describan el *software* de computadora, en este caso del API, al trabajar para satisfacer una serie de requisitos, en este caso de los problemas del desarrollo de sistemas HIC. Los pasos concretos para lograr este objetivo varían según el autor, aunque cabe remarcar que todos tienen semblantes semejantes. En especial cuando se comparan con el proceso mencionado en el párrafo anterior. En cuanto a los pasos utilizados específicamente en este análisis son los propuestos por Pressman en [PRE97].

Es válido cuestionar si los pasos del análisis OO que convencionalmente se utilizan en las aplicaciones se pueden utilizar para desarrollar un API. Un API que fundamentalmente debe ser reutilizable y portador de la implementación de una serie de requisitos funcionales comunes a varias aplicaciones. La respuesta corta es que no. Aunque si se revisa con mas detalle, los métodos de análisis para modelar librerías de *software* reutilizable, simplemente agregan unos cuantos pasos a lo que sería el modelado común. Estos pasos extra tienen la función de obtener los requisitos comunes de una serie de aplicaciones que pertenecen a un dominio previamente definido. A continuación se presenta la técnica de análisis aplicada a los proyectos de desarrollo de librerías de *software* reutilizable. Esta técnica es conocida como el análisis de dominio.

4.2. Análisis del Dominio.

4.2.1. Introducción.

Cuando se quiere construir una biblioteca de clases reutilizables ampliamente aplicables a una categoría completa de aplicaciones, como es el caso con el API de HIC, se tiene que realizar un análisis OO de nivel medio de abstracción conocido como análisis de dominio. El análisis de dominio tiene el objetivo de reconocer, analizar y especificar requisitos comunes de un dominio de aplicación específico, con el propósito de la reutilización en múltiples proyectos dentro de ese dominio. Mediante la identificación de clases, objetos, responsabilidades y jerarquías el análisis de dominio reconoce capacidades reutilizables que son comunes a varias aplicaciones.

Es interesante apuntar como el análisis de dominio es una técnica completamente aplicable a al construcción de APIs, aunque el dominio de éstos sea mas limitado que el de las libreras de *software*.

4.2.2. Definición del Dominio.

El API desarrollado en este proyecto, está planeado para ser utilizado en el desarrollo de aplicaciones del campo de la educación o del entretenimiento, y cuya función es la de transmitir una historia al usuario. Estas aplicaciones pueden ser desarrolladas tanto con fines de investigación como de lucro. Las aplicaciones deben de correr en computadoras personales con sistema operativo Windows. Además, la manera como éstas deben presentar las acciones que suceden en la historia a los usuarios tiene ciertas restricciones. Las acciones deben de ser presentadas de manera gráfica y directa, lo que implica la necesidad de actores virtuales que las realicen. Para que una aplicación sea considerada en este domino, debe permitir que las acciones del usuario tengan un verdadero efecto en la manera como se dan los eventos de la historia, esto quiere decir que las aplicaciones deben de ser interactivas.

4.2.3. Recolección de Aplicaciones Representativas del Dominio.

En está sección se listan diversas aplicaciones que caen dentro del dominio especificado en la sección anterior. Y aunque la arquitectura y el enfoque que cada una tiene son diferentes, todas comparten algunos requisitos comunes. Requisitos que pueden ser cubiertos por el API que se desarrolla en este trabajo. Es importante recordar, que como requisito del dominio, estas aplicaciones deben de mostrar la historia de manera gráfica. A continuación, se listan las aplicaciones que son representativas de este dominio, se incluye el nombre del proyecto o aplicación, una pequeña descripción de lo que hace y donde conseguir mas información sobre éstos.

GEIST. Es un sistema que presenta una historia interactiva a lo largo de un viaje real a través de una ciudad. Esto se logra mediante dispositivos de realidad aumentada. Su principal propósito es mostrar de manera interesante y entretenida la información acerca de los lugares históricos que los usuarios van visitando a lo largo de su recorrido. Los usuarios de este sistema reciben dispositivos de localización, que permiten saber cuando se acercan a ciertos lugares donde supuestamente se encuentra un espíritu. Este espíritu (De ahí el nombre, ya que espíritu en alemán es Geist) se muestra a través de los lentes de realidad aumentada, con la intención de que le cuente su historia al usuario. La interactividad en este sistema proviene de cuales lugares visite el usuario y en cuanto tiempo lo haga. De ahí que se requiera de adaptabilidad en la historia presentada. Por lo tanto una parte de este sistema se encarga de mantener las funciones narrativas de la historia. Esta parte del sistema esta basado en las funciones narrativas propuestas por Vladimir Propp. Este es un proyecto del Departamento de Historias Digitales del ZGDV en Alemania. Mas información acerca de este sistema en [GEI03].

IzA. Este es otro sistema desarrollado por el Departamento de Historias Digitales del ZGDV, pero con un mayor enfoque a la conversación y a los actores digitales. Su nombre proviene de las siglas de la frase en alemán significa "información tocable". La función de este sistema es estar en un punto de información a la entrada del ZGDV. El sistema transmite información comercial mediante la comunicación entre el usuario y los

personajes virtuales. Y es debido a su enfoque, que el mayor énfasis se le pone a las animaciones y a los gestos de los personajes virtuales. Mas información acerca de este sistema y la base teórica que lo sustenta en [IZA03].

Historias Interactivas con RTJ. Este proyecto es desarrollado en la Universidad de Teesside, en Inglaterra. Su principal interés está en la interacción de los espectadores con los personajes virtuales de una historia interactiva. Las situaciones y los personajes de la historia están basados en los de la famosa serie de televisión norteamericana Friends. Existen dos formas para que los espectadores puedan interactuar con los personajes. La primera es mediante "consejos" que los espectadores pueden "dar" a los personajes. La otra es a través de la capacidad de modificar el escenario al poner y quitar objetos relevantes en el desarrollo de la historia. Los personajes de la historia deciden que hacer a través de un método conocido como Red de Tareas Jerárquicas (RTJ), un método basado en la descomposición de tareas y en la planeación. Cabe destacar que para presentar los modelos y las gráficas este sistema está basado en el motor de juegos Unreal Tournament. Para mayor información de este proyecto ver [RTJ03].

Erasmatron. Es un sistema cuya principal preocupación es la de facilitar el desarrollo de historias interactivas. Su desarrollador, Chris Crawford, es un impulsor del acercamiento multidisciplinario en el desarrollo de historias interactivas. Por esta razón, parte de su sistema es un ambiente de desarrollo de scripts amigable para los novatos de la programación, como es el caso de los artistas. En este sistema también se hace patente la búsqueda de un método de diseño propio de los sistemas de historias interactivas. Para información de este proyecto en [ERA03].

Mimesis Engine. Proyecto desarrollado por el grupo Liquid Narrative de la Universidad del Norte de Carolina. El propósito del sistema es presentar historias interactivas en las que el usuario pueda interactuar mientras que se mantiene la coherencia de la historia. En el desarrollo de este proyecto también utilizan un motor de juegos: El Unreal Tourment Engine. Su sistema se divide en dos partes principales, el controlador y el servidor mimesis. El controlador mimesis es la parte de sistema que genera la historia y le envía al servidor mediante instrucciones de alto nivel para que éstas sean realizadas. La función del servidor mimesis es la de convertir las acciones de alto nivel generadas por el controlador en varias tareas de bajo nivel, que son entendibles para el motor de Unreal. La manera en la que el sistema intenta mantener la coherencia de la historia es la siguiente: Cuando el servidor detecta una acción del usuario que es contraria a la historia, éste le envía un mensaje al controlador para que decida que hacer. La decisión del controlador tiene dos resultados: La generación de una nueva historia que sea coherente con la acción del usuario, aunque esto puede ser, en términos de poder computacional, muy costoso o la decisión del controlador de evitar está acción mediante algún tipo de protección. Por ejemplo, el usuario puede decidir matar a un personaje importante en el desarrollo de la historia, esta decisión es transmitida al controlador mimesis. El controlador puede entonces decidir por generar una nueva historia que concuerde con la muerte de este personaje, si no es muy costoso, o puede decidir que la pistola del usuario debe fallar con tal de que la historia continué. Para mayor información de este proyecto visitar a [MIM03].

4.2.4. Análisis de las Aplicaciones de la Muestra.

El análisis a los sistemas mostrados en la sección anterior, tiene como propósito el encontrar las clases, objetos y requisitos comunes en éstos. Y que por lo tanto deben de ser incluidos en el API que se está desarrollando. Estos requisitos comunes serán la base para que en la siguiente sección se realice un modelo de análisis como él que se haría con una aplicación cualquiera, con la excepción que se trata de los requisitos comunes a varias aplicaciones y no los requisitos de un usuario. Y es por esta razón que no se incluyo el análisis de los *casos de uso*, ya que dependerá en mucho de la forma como el desarrollador de sistemas interactivos utilice este API, la forma en como el usuario utilizará el sistema.

Una de las primeras características que notamos de estos sistemas es que los desarrolladores no implementan la parte que presenta las gráficas 3D. Esto, por obvias razones, los libera de tener que implementar algo tan complejo como un motor gráfico, o mas aún, como un motor de juegos. Esto *implica* algún motor de juegos 3D debe de ser una parte fundamental del API. Ya que sería ilógico pensar en presentar un API sin esta funcionalidad, y que esté diseñado para trabajar de manera sencilla con cualquier motor que el desarrollador escoja. La elección del motor que será parte del API se hará basándose en ciertas características que éste debe de cumplir, y que se presentarán mas tarde.

Una clase común en todas las aplicaciones revisadas es la del actor virtual. Claro que en cada aplicación ésta tiene características diferentes. Pero al hacer un análisis más profundo se pueden extraer ciertas responsabilidades base que les son comunes a todas. La primera de ellas es la diferencia entre la clase actor, que es la responsable de guardar el estado de animo y el humor del personaje, y la clase encargada de la representación gráfica y animación de éste. Para clarificar este punto, se puede tomar el proyecto mimesis. En el la representación gráfica de cualquier actor es responsabilidad del motor de Unreal y en especial de las clases que dentro de éste son responsables, mientras que el estado interno del personaje se maneja desde *el controlador* mimesis. De la misma manera se puede ver con el proyecto de Friends, donde las gráficas también son manejadas por el motor de Unreal, mientras que agentes basados en RTJ son los que deciden que hará ese personaje.

La siguiente característica común a varios proyectos es el soporte de lenguajes de script. Con los que se controlan a los objetos y personajes de la historia. En especial se puede notar en caso de los proyectos donde se utiliza el motor Unreal, ya que éste cuenta con el soporte para un lenguaje de script conocido como Unreal script. En el caso del Erasmatron, el soporte de un lenguaje de script, es la base del mismo. Por lo mismo, el interprete del script fue desarrollado por ellos mismos. Debido a esto, se decidió que el motor de juegos que el API incluya también debería de contar con el soporte de algún lenguaje de script. Esta fue una de las principales razones por las que se tomó la decisión de incluir el motor de juegos Nebula, además del hecho de que el código de este motor esta abierto. Para encontrar todo lo relacionado con este motor de juegos consultar es [NEB03].

Las causas por las que se incluyeron en el API ciertas clases y responsabilidades son varias, pero en general, se puede decir que siempre se tuvo en mente la forma como habría sido mas fácil el trabajo de los desarrolladores. Por ejemplo, en varias aplicaciones se nota la intención del desarrollador de construir su aplicación con una arquitectura lógica y entendible, así como de la aplicación de metodologías que se basan en la teoría del drama

y de la narración. Impulsando está misma meta, el API promoverá que el desarrollador cree aplicaciones mediante métodos de diseño estándar y con arquitecturas adecuadas.

4.2.5. Desarrollo del Modelo de Análisis.

Este es el último paso del análisis de dominio, su función es la de servir de base para el diseño y construcción de los objetos. De aquí en adelante el proceso de análisis es casi el mismo que se usaría para cualquier aplicación. Como primer punto en la creación de este modelo es necesario identificar a las clases relevantes a todos los sistemas, basándose en las necesidades comunes a todas las aplicaciones mostradas en [4.2.3.].

Se muestra a continuación el proceso de análisis que se llevo a cabo. Los requisitos y la base teórica se muestran en [2.9.2].

4.3. Modelado de Clase-Responsabilidad-Contribución.

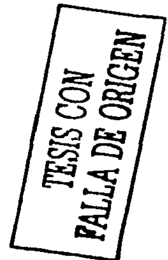
Después de analizar las aplicaciones que pertenecen al dominio definido, con este modelado se identifican y organizan las clases candidatas, que cumplen con las pautas mostradas en [2.9.2] y que se encontraron de manera recurrente en las diversas aplicaciones analizadas. Las clases se presentan como tablas divididas en tres secciones. Donde se presenta el nombre de la clase, las responsabilidades de ésta y sus clases contribuyente. Con respecto al prefijo Ist que precede al nombre de todas las clases, se colocó para indicar que estas pertenecen al API de historias interactivas (Interactive StoryTelling por sus siglas en inglés).

Antes de pasar al modelado CRC, es importante aclarar que algunas de las clases que se presentan no fueron desarrolladas como parte de este proyecto, sino que se incluyeron en el API para cumplir con los requisitos funcionales, y con las clases que son requeridas por los sistemas de historias interactivas. Tal es el caso de las clases nCal3DModel, nCal3DcoreModel y nCal3Dmaterial. De la misma manera, en este modelado también se presentan algunas clases que pertenecen al motor de juegos Nebula y no al API. La razón para presentarlos es la misma, para indicar la clase que cumple con los requisitos funcionales detectados el capítulo anterior. En esta situación se encuentra la clase nSceneGraph2. Hecha esta aclaración, a continuación se presenta el modelado CRC.

nIstPlayer. Esta clase es la que se encargará de controlar el personaje que representa al usuario dentro de la historia. Por lo tanto, esta clase será la responsable de manejar las entradas del usuario que tengan la intención de controlar a su personaje y llamar a las animaciones adecuadas.

Nombre: nIstPlayer	
Tipo de Clase: Rol	
Características: Agregada, secuencial, temporal.	
Responsabilidades:	**Contribuyentes:**
Responder a entradas de usuario	nInputServer
Interactuar	nIstNPC
Controlar animaciones	nCal3DModel
Controlar la física	nIstWolrd
Conocer sus cualidades físicas	
Actualizar el estado interno	
Manejo de colisiones	nIstNPC, nIstStruct, nIstObject

Cuadro 4.1. Tarjeta de Índice CRC de nIstPlayer.



nIstNPC. Clase del API responsable de controlar a todos los personajes de la historia que no son controlados por el usuario. Como se puede prever, los comportamientos y acciones que estos actores deben realizar son muy variados para siquiera pensar en programarlos todos, de ahí la importancia que puedan seguir scripts que les darán un comportamiento único. También es importante mencionar la relación que esta clase tiene con la clase nPath y con la clase nVoiceServer como forma de adquirir mayor funcionalidad.

TESIS CON
FALLA DE ORIGEN

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

Cuadro 4.2. Tarjeta de Índice CRC de nIstNPC.

nIstObject. Está clase es responsable de controlar a cualquier objeto, de ahí la importancia de separar la clase encargada de la representación gráfica de la encargada de controlar, que se encuentre en el escenario y que puede ser tomado por el personaje que representa al usuario, o por cualquier otro personaje controlado por la computadora.

Nombre: nIstObject	
Tipo de Clase: Cosa	
Características: Agregada, tangible, secuencial, transitoria	
Responsabilidades	**Contribuyentes**
Responder a Interacción	nIstPlayer
Seguir scripts activados por eventos.	nScriptlet
Controlar la física	nIstWolrd
Conocer cualidades físicas	
Actualizar el estado interno	
Manejo de colisiones	nIstNPC, nIstStruct, nIstPlayer
Ser tomados y entregados	nIstPlayer

Cuadro 4.3. Tarjeta de Índice CRC de nIstObject.

nIstStruct. Esta clase tiene solo una responsabilidad, la de controlar a todos los objetos que forman parte del escenario y que no pueden ser movidos. Esto obviamente incluye al piso, las paredes y el techo de cualquier estructura presente.

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

Cuadro 4.4. Tarjeta de Índice CRC de nIstStruct.

TESIS CON
FALLA DE ORIGEN

nIstCamera. Clase responsable de 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.

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	nInputEvent

Cuadro 4.5. Tarjeta de Índice CRC de nIstCamera.

nIstWorld. Clase responsable de contener y actualizar a todos elementos de la historia interactiva, clases identificadas por el prefijo IST (Interactive StoryTelling). Además, esta clase es responsable de contener los valores de constantes físicas que se aplican a todos los entes que contiene.

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

Cuadro 4.6. Tarjeta de Índice CRC de nIstWorld.

nIstCursor. Clase sencilla, que se incluyó en el API para tener un apuntador que pueda indicar la localización de algún punto.

Nombre: nIstCursor	
Tipo de Clase: Cosa	
Características: Tangible, agregada, secuencial, temporal	
Responsabilidades:	Contribuyentes:
Actuar de acuerdo a las entradas del usuario	nInputServer
Conocer sus cualidades físicas	
Actualizar su estado	

Cuadro 4.7. Tarjeta de Índice CRC de nIstCursor.

TESIS CON FALLA DE ORIGEN

nPath. Clase responsable de almacenar los puntos del recorrido de la clase nIstNPC, o de la clase nIstCamera. Cabe mencionar que esta clase si es desarrollada en este proyecto, aunque parte de su funcionalidad es implementada por la clase NurbsCurvef que fue tomada del paquete Nurbs++. El hecho que esta clase no cuente con el prefijo Ist en su nombre indica que no es un elemento que se pueda considerar como de la historia.

Nombre: nPath	
Tipo de Clase: Cosa	
Características: Abstracto, agregada, secuencial, permanente,	
Responsabilidades:	Contribuyentes:
Conocer los puntos que definen su recorrido	
Interpolar el recorrido entre dos puntos	NurbsCurvef
Conocer su origen y su destino	
Necesita asegurar su persistencia.	

Cuadro 4.8. Tarjeta de Índice CRC de nPath.

nVoiceServer. Clase responsable de atender las peticiones de conversión de voz a texto hechas por la clase nIstNPC. Al igual que la clase nPath, esta clase es desarrollada en el proyecto, aunque parte de su funcionalidad es implementada por varias clases del Speech SDK 5.1 de Microsoft. La razón por la que esta clase no cuenta con el prefijo Ist es la misma que de la clase nPath.

Nombre: nVoiceServer	
Tipo de Clase: Entidad externa	
Características: Abstracta, agregada, concurrente, temporal	
Responsabilidades	Contribuyentes
Conocer su estado cuando está hablando y cuando está libre.	SPVOICESTATUS
Convertir texto a voz cuando se le requiera	ISpVoice
Actualizar su estado	
Conocer el número de voces	
Cambiar la voz en uso	

Cuadro 4.9. Tarjeta de Índice CRC de nVoiceServer.

TESIS CON FALLA DE ORIGEN

nCal3DCoreModel. Clase externa al motor Nebula, perteneciente al paquete nCal (Paquete que permite cargar modelos y sus animaciones basadas en esqueleto). Esta clase está encargada de almacenar las animaciones, los materiales, las mallas y el esqueleto de un modelo. Y aunque esta clase no fue desarrollada en este proyecto, se presenta en este modelado para indicar la clase responsable de cubrir estas funciones.

Nombre: nCal3DCoreModel	
Tipo de Clase: Cosa	
Características: Abstracta, agregada, secuencial, temporal	
Responsabilidades	Contribuyentes
Conocer los materiales de un modelo	nCal3DMaterial
Conocer las animaciones de un modelo	CalCoreModel
Conocer las mallas de un modelo	CalCoreModel
Conocer el esqueleto de un modelo	CalCoreModel

Cuadro 4.10. Tarjeta de Índice CRC de nCal3DCoreModel.

nCal3DMaterial. También se trata de una clase perteneciente al paquete nCal, no desarrollada en este proyecto. Esta clase contiene la información del sombreado y la textura de alguna superficie.

Nombre: nCal3DMaterial	
Tipo de Clase: Cosa	
Características: Tangible, agregada, secuencial, temporal	
Responsabilidades:	**Contribuyentes:**
Conocer la textura de una malla	nTexArrayNode
Conocer el sombreado de una superficie	nShaderNode
Conocer el nivel de detalle de una malla (LOD)	

Cuadro 4.11. Tarjeta de Índice CRC de nCal3DMaterial.

nCal3DModel. Clase perteneciente al paquete nCal, y presentada para mostrar a la responsable de la representación gráfica de los personajes, tanto de nIstPlayer como de nIstNPC. Tiene la responsabilidad de dibujar al personaje con la malla adecuada y realizando la animación que se indique.

Nombre: nCal3DModel	
Tipo de Clase: Cosa	
Características: Tangible, agregada, secuencial, temporal	
Responsabilidades:	**Contribuyentes:**
Conocer a su modelo base	nCal3DCoreModel
Realizar ciclos de su animación	CalModel
Mezclar animaciones	CalMixer
Realizar animaciones una vez	CalModel
Dibujarse en pantalla	nSceneGraph2

Cuadro 4.12. Tarjeta de Índice CRC de nCal3DModel.

TESIS CON
FALLA DE ORIGEN

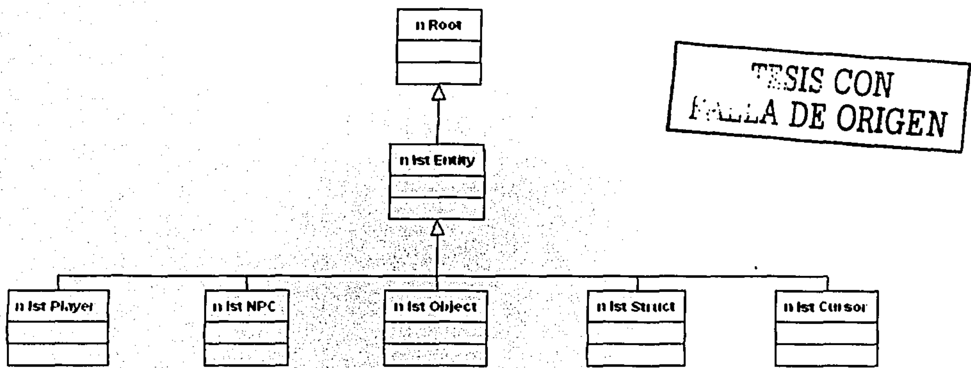
nSceneGraph2. Clase perteneciente al motor de juegos Nebula, por lo que no se desarrolló en este proyecto. Esta clase se presenta porque es la que cumple con el requerimiento de una clase responsable de contener todos los objetos que aparecen en escena. Además, es responsable de conocer la posición desde la cual se dibuja la escena y de indicarles a todos los objetos de la escena cuando dibujarse.

Nombre: nSceneGraph2	
Tipo de Clase: Cosa	
Características: Abstracta, agregada, secuencial, temporal	
Responsabilidades:	**Contribuyentes:**
Tener los elementos de escena	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

Cuadro 4.13. Tarjeta de Índice CRC de nSceneGraph2

Jerarquías de clases.

Después de presentar el modelado CRC de las clases del API, como parte del análisis, es necesario presentar las distintas jerarquías de clases. Como primer paso, se presenta la jerarquía de todos los objetos que pueden estar en el mundo de la historia. Es importante notar la aparición de la clase nIstEntity, como superclase de todas ellas, así como a la clase nRoot como superclase de ésta. Se decidió por la creación de la clase nIstEntity para aprovechar el polimorfismo y un grupo de funciones que todas las clases el API comparten, y que es mejor que reciban a través de la herencia. En cuanto a la clase nRoot, ésta es la clase base de los objetos que se crean para el motor Nebula, por lo tanto se decidió que nIstEntity debía de heredar de ella.



TESIS CON FALLA DE ORIGEN

Figura 4.1. Jerarquía de Clases de nIstEntity.

A continuación, se presenta la jerarquía de los objetos que pueden aparecer en escena, y que por lo tanto heredan de la clase nVisNode, que es la clase base y contiene la funcionalidad básica para poder ser agregado a la clase encargada de la escena, la clase nSceneGraph2. Un detalle a remarcar de esta jerarquía es la aparición de la clase nCal3DModel como subclase de nVisNode, cosa que puede parecer extraña. Pero si se analiza con mas detalle se le encontrara el sentido a su aparición. Porque la clase necesita estar en la escena para poder dibujar la imagen del modelo que representan.

TESIS CON
FALLA DE ORIGEN

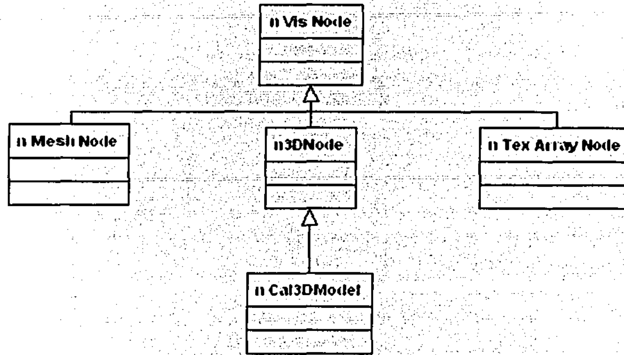


Figura 4.2. Jerarquía de clases de nVisNode.

La siguiente figura, contiene una jerarquía que se consideró importante presentar como muestra de un patrón de diseño preocupado por la portabilidad. Como se mencionó anteriormente, parte de la funcionalidad del servidor de voz, está dada por una librería de Microsoft, decisión que hace imposible portarlo a otra plataforma que no sea Windows. La decisión de hacer a la clase nVoiceServer una clase abstracta, permite que mediante el polimorfismo, todas el código que se construya, simplemente utilice una instancia de la clase nSAPIServer, aunque tengan un apuntador a nVoiceServer.

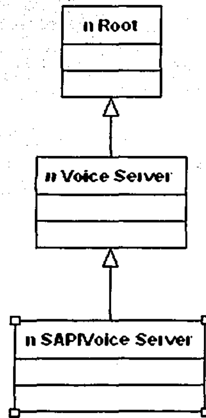


Figura 4.3. Jerarquía de clases de nVoiceServer.

En la siguiente figura simplemente se muestra la jerarquía de las demás clases del API, y que heredan de la clase base del motor Nebula, nRoot. En especial, las clases que requieren de la persistencia, como la clase nPath, se benefician de está herencia ya que parte de la funcionalidad de la clase nRoot es la de la persistencia del estado de los objetos al salvarlo en un script.

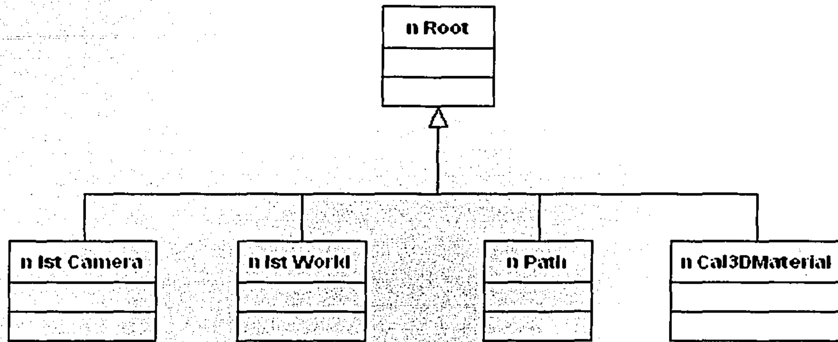
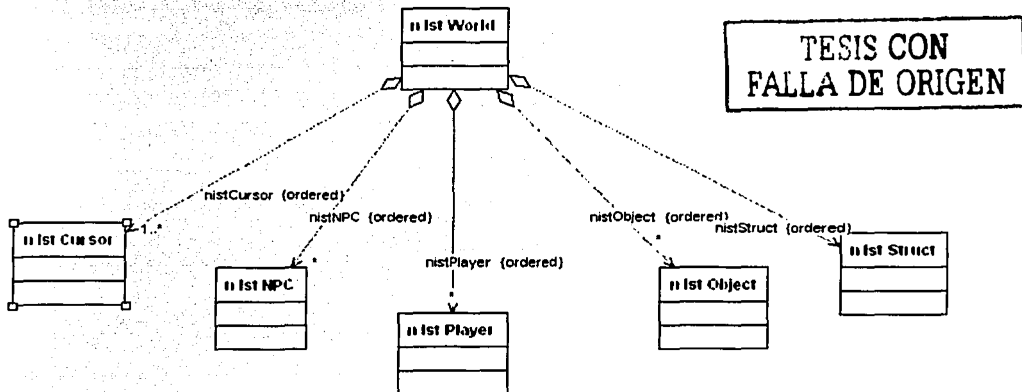


Figura 4.4. Jerarquía de clases adicionales.

Estructuras.

En las siguientes figuras se presentan las diferentes estructuras con las que trabajaría un sistema diseñado con este API. Por ejemplo, la siguiente figura muestra las clases que la clase mundo, nIstWorld, puede contener y actualizar.



TESIS CON
FALLA DE ORIGEN

Figura 4.5. Estructura del mundo de la historia.

ESTA TESIS NO SALE
DE LA BIBLIOTECA

La siguiente estructura muestra la forma como se conforma un elemento gráfico para representar al personaje de un usuario o de un agente. Como ya se mencionó anteriormente, estas clases están incluidas en un paquete que no fue desarrollado en este proyecto, pero que sin lugar a dudas por el importante papel que cumplen en API es necesario analizar su estructura. Es importante indicar que en esta estructura la relación de la clase nCal3DModel con la clase nCal3DCoreModel es de 1 a 1 como indica la figura, pero en cambio, un nCal3DCoreModel puede tener una relación de 0 a muchos con la clase nCal3DModel. Esto se debe a que la clase nCal3DCoreModel funciona como un almacén de animaciones que varias instancias del modelo pueden utilizar.

TESIS CON
FALLA DE ORIGEN

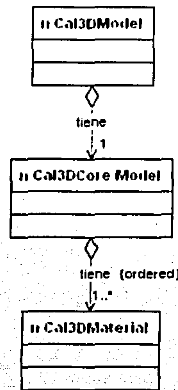


Figura 4.6. Estructura de un modelo nCal3D.

La siguiente figura muestra la estructura con la que se construye una escena, estructura similar a un árbol invertido, formado por cualquier objeto que pertenezca a una clase herede de nVisNode.

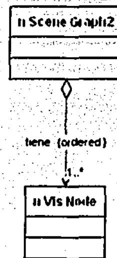
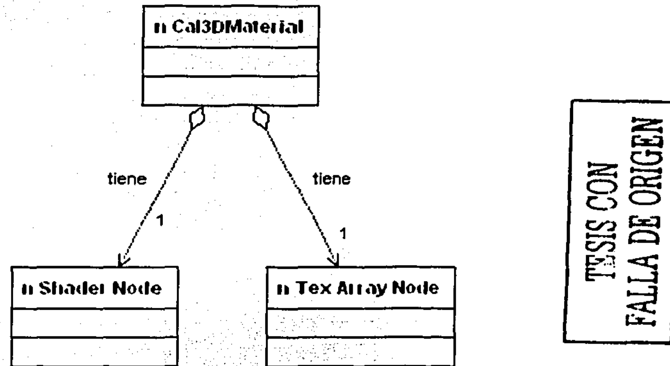


Figura 4.7. Estructura de una escena.

La siguiente figura muestra la sencilla estructura que conforma los materiales del paquete Cal3d. La responsabilidad de esta estructura, formada por tres clases, es indicar el sombreado y la textura que se aplicará a cada una de las mallas que conforman a los modelos del paquete Cal3d.



TESIS CON
FALLA DE ORIGEN

Figura 4.8. Estructura de un material.

Para finalizar, se presenta la manera como se relaciona la clase que mantiene el estado interno del personaje y la clase encargada de dibujar en la pantalla el modelo animado que representa al personaje. Es importante aclarar que esta relación también se da entre la clase nIstNPC y la clase nCal3DModel.

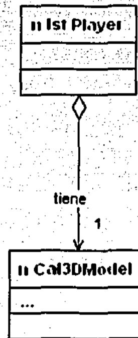


Figura 4.9. Estructura de un ente controlado por el usuario.

4.4. Modelado Objeto-Relación.

Después de que con el modelado CRC se obtuvieron los primeros atisbos de relaciones entre las diferentes clases. Para comprender de manera mas profunda las relaciones entre ellas, es necesario que primero se establezca claramente la responsabilidad de cada una de ellas. Para después encontrar las clases que colaboran con ella y con las cuales, sin lugar a dudas, existe una relación.

En la figura siguiente se presenta las relaciones como mas comúnmente se darían en un sistema que utilice el API. Cabe hacer la aclaración, que con la intención de dar una idea mas general y completa de cómo se utilizarían las clases desarrolladas, también se presentan las relaciones que algunas de las clases desarrollados normalmente tendrían con las clases contenidas en el motor Nebula. Por ejemplo, la relación entre nIstStruct y la clase nMeshNode.

TESIS CON
FALLA DE ORIGEN

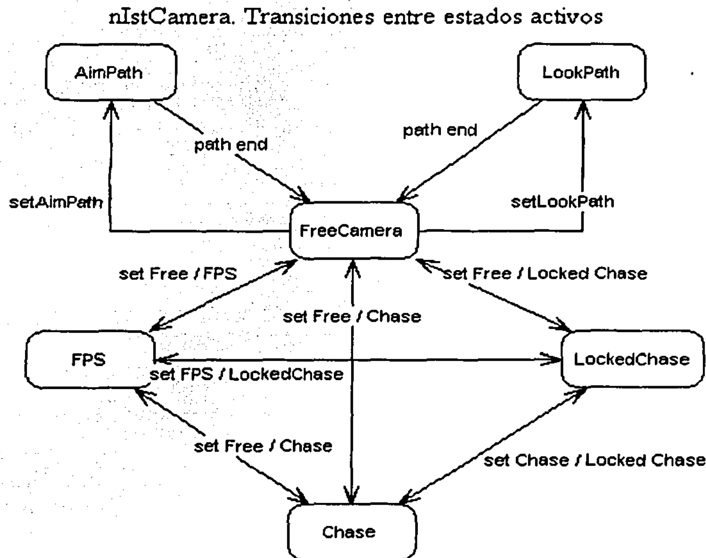
Figura 4.10. Modelado objeto-relación del sistema.

4.5. Modelado Objeto-Comportamiento.

Para terminar la parte de análisis en el desarrollo del API para HIC es necesario realizar una transición al comportamiento dinámico que tendrán los diferentes objetos de las clases ya identificadas. Para realizar esto se debe representar el comportamiento de los objetos como una función de eventos específicos y tiempo.

La forma como este comportamiento se modela es mediante diagramas que 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 identificadas:

nIstCamera. El diagrama de transiciones de esta clase muestra la manera en la que cambia entre los diferentes estilos de toma. De estos estilos, se pude considerar que el estilo libre, FreeCamera, controlado por el usuario se trata del estado activo base de esta clase. Un detalle importante a destacar de los eventos pathend, fin de camino, es que se trata de los únicos que no son controlados por el usuario o la propia historia, sino que son lanzados cuando el recorrido que la cámara esta realizando se termina.



TESIS CON FALLA DE ORIGEN

Figura 4.11. Diagrama de estados de la clase nIstCamera.

nIstCursor. Clase cuyo único propósito es indicar de manera gráfica una posición. Es debido a esta sencillez, que los estados con que la clase cuenta en realidad son los cinco estados activos que el diagrama presenta. Como se puede ver, todos los estados de esta clase tienen que ver con el movimiento. De manera obvia el estado base de esta clase es el estado detenido (Stop).

nIstCursor. Transiciones entre estados activos.

TESIS CON
FALLA DE ORIGEN

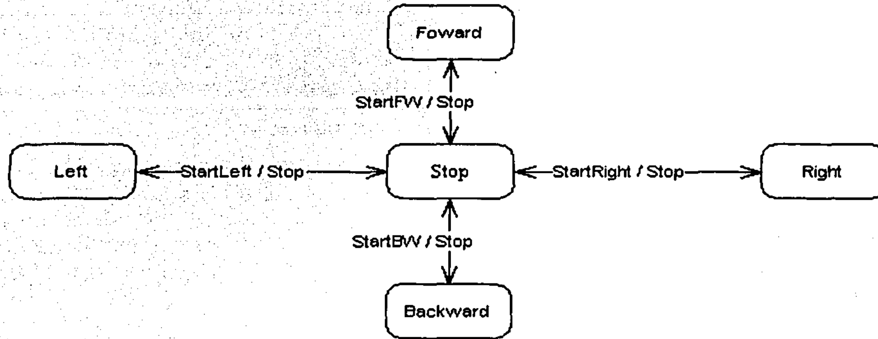


Figura 4.12. Diagrama de estados de la clase nIstCursor.

nIstObject. Como se indico en el modelado CRC, la única responsabilidad de esta clase es la de mantener el estado de un objeto que puede o no ser tomado por algún personaje de la historia. Esto se refleja muy claramente en su diagrama de transiciones.

nIstObjet. Transiciones entre estados activos.

Figura 4.13. Diagrama de estados de la clase nIstObject.

nIstNPC. Clase de la que sin lugar a dudas se espera un comportamiento muy complicado, ya que es la responsable del control de los actores virtuales. A primera vista el diagrama de estados puede aparentar no mostrar esa complejidad. En realidad lo que sucede es que varios de los estados de esta clase implican la interpretación de scripts que permiten proveer a los objetos de esta clase con un comportamiento mucho mas complejo y diferenciado. Como se puede ver el diagrama hace énfasis a los eventos que pueden activar la interpretación de un script. Eventos tales como chocar contra algo o alguien, "ver" a algún personaje o que alguno de sus atributos alcance un valor dentro de cierto rango.

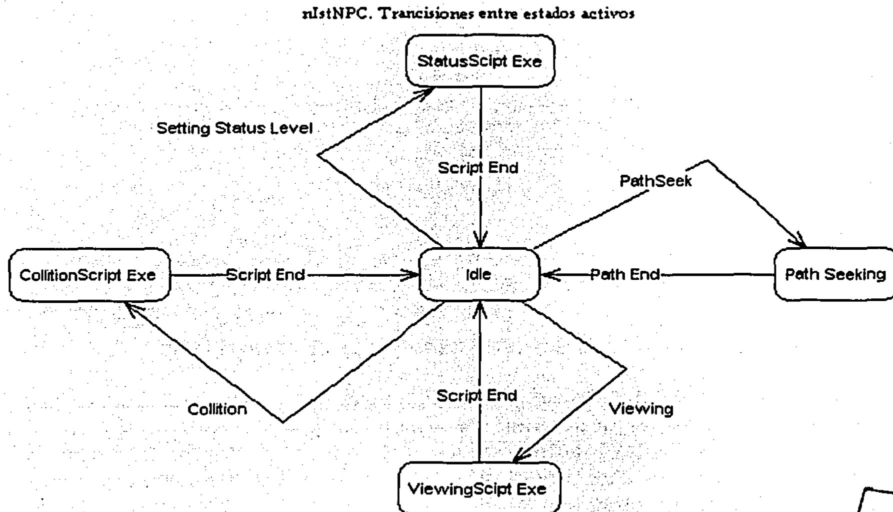
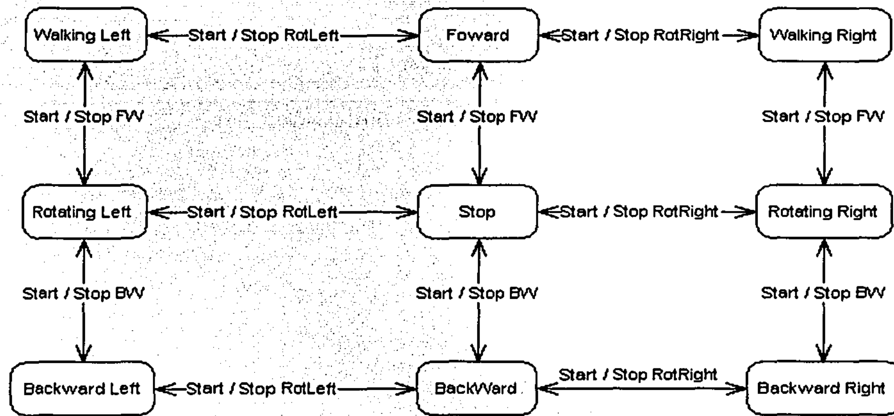


Figura 4.14. Diagrama de estados de la clase nIstNPC.

TESIS CON
FALLA DE ORIGEN

nIstPlayer. El diagrama de transiciones de esta clase en realidad solo presenta los estados activos relacionados con el movimiento. Esto se debe a que los otros estados relevantes de esta clase son los relacionados con la animación que ésta realiza. Además, esta responsabilidad no recae totalmente en esta clase y depende del número y tipo de animaciones que el usuario haya utilizado.



TESIS CON FALLA DE ORIGEN

Figura 4.15. Diagrama de estados de la clase nIstPlayer.

nIstWorld. La cantidad de estados activos esta clase es muy reducida, debido a que su principal responsabilidad es la de contener a todas las entidades de la historia. Además se debe resaltar que la naturaleza de los estados activos de esta clase es diferente al de los demás clases Ist, ya es a través de esta clase y de su estado actualizar el medio por el cual se actualizan todas las demás clases.

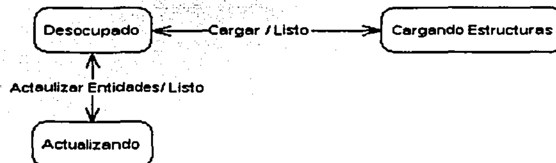
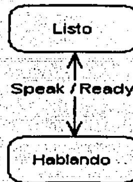


Figura 4.16. Diagrama de estados de la clase nIstWorld.

nVoiceServer. El diagrama de transiciones de esta clase en muy sencillo y claro. El único punto necesario aclarar es que la procedencia del evento Ready (Listo). Este evento es lanzado por la misma clase cuando ha terminado la petición de convertir texto a voz.

nVoiceServer. Transiciones entre estados activos.



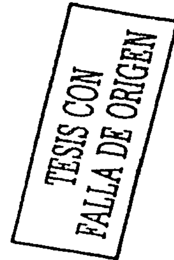

TESIS CON
FALLA DE ORIGEN

Figura 4.17. Diagrama de estados de la clase nVoiceServer.

nPath. El diagrama de esta clase merece un análisis mas detallado. Pues como se podrá advertir, es el único diagrama en el que se han incluido un estado inicial y un estado final. En realidad, después de su proceso de inicialización, esta clase no cuenta con estados activos. Esto debido a que su principal función es la de almacenar los puntos que forman un camino e interpolar los puntos que existen entre estos puntos. Este diagrama muestra el proceso mediante el cual un objeto de esta clase es inicializado correctamente después de que se creo.

nPath. Transiciones entre estados activos.

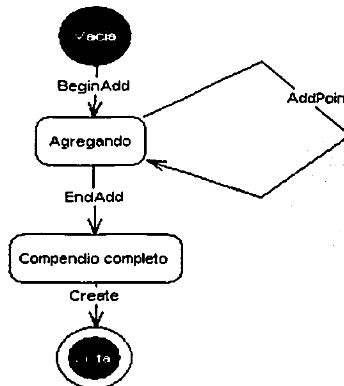


Figura 4.18. Diagrama de estados de la clase nPath.

Capítulo 5.

Diseño Orientado a Objetos (OO) del API.

5.1. Introducción.

La intención del proceso de diseño orientado a objetos, que es presentado en este capítulo, es transformar el modelado obtenido en el análisis en un anteproyecto para el desarrollo de *software*. Además, con este tipo de diseño se logra un cierto número de diferentes niveles de modularidad que dividen a los diferentes componentes del sistema en módulos conocidos como subsistemas, las operaciones y los datos que hacen funcionar a esos subsistemas, también está encapsuladas en objetos. Esta por supuesto que es una de las ventajas que se obtiene de aplicar el diseño orientado a objetos. De cualquier manera, debido a que en este proyecto se está desarrollando un API y no una aplicación la ventaja que tiene el diseño OO para crear subsistemas se ve ampliamente disminuida. Esto debido a que la creación de subsistemas, en general, es parte del trabajo de los desarrolladores de aplicaciones, no del desarrollador del API.

En el caso del diseño de una aplicación, normalmente, se consideran dos etapas principales: El diseño del sistema y el diseño de los objetos. En el proceso de diseño de sistema se realiza una partición al modelo de análisis con el propósito de obtener los subsistemas que conforman el sistema, se hacen consideraciones acerca de la concurrencia en la ejecución de cada una de estos, en los que se encargaran de la gestión de datos, de tareas, de recursos y en la comunicación entre cada uno de ellos. En este caso, debido a que el proyecto está enfocado al desarrollo de un API, la construcción de subsistemas se deja en manos del desarrollador de la aplicación. Y aunque en el proceso de análisis que se levo a cabo en el capítulo anterior, se presentaron estructuras con el tamaño suficiente para considerarlas como subsistemas. Todas ellas en realidad son estructuras que varían de tamaño según lo que el usuario agregue. Tal es el caso de la estructura que se puede formar con la clase del mundo (nlstWorld) o con la clase de la escena (nSceneGraph2). Por lo que la manera como el usuario defina estas estructuras, que fácilmente se podrían convertir en subsistemas, se deja a su completa elección. Debido a lo expuesto anteriormente, en el proceso de diseño que se realiza en este capítulo de deja fuera cualquier intento de diseñar el sistema, poniendo todo el énfasis en el diseño de los objetos que componen al API.

El proceso de diseño de objetos tiene como propósito especificar cada método y atributo que tiene cada clase. Además busca especificar la manera en la que una clase se comunica con sus colaboradores. Es importante remarcar, que al tratarse de un API, el diseño se aplicará a la especificación de clases y no a la de objetos. El proceso de diseño OO se divide en tres pasos: Descripción de clases, la optimización y detalle de los métodos y la definición de componentes de programa e interfaces. A continuación, se presenta el proceso de diseño de las clases del API.

5.2. Descripción de Clases.

El propósito de este paso en el proceso de diseño es establecer la interfaz de las clases, definiendo de manera breve la operación que éste realiza con cada uno de todos los posibles mensajes que puede recibir. El formato utilizado en esta descripción es el siguiente:

Valor de retorno ***Mensaje***(valor1, valor2 , ..., valorN) //Comentario de lo que realiza

Que como se podrá notar es muy similar a la manera como se definen las funciones y los métodos en el lenguaje C y C++. La elección de este formato es obvia cuando se considera que el API será implementado con C++. Aun así, es necesario indicar que esta primer descripción de clase todavía no se puede utilizar como la base para crear el archivo de cabecera de C++ (también conocido como archivo h), ya que todavía es necesario que pase por un proceso de optimización. Donde es muy posible que este mensaje de subdivida en una serie de mensajes mas sencillos que cumplan con todo lo que realizaba el mensaje original. De esta manera es que la descripción de las clases es la siguiente:

nIstEntity.

nada **Trigger**(DeltaTiempo)	//Causa que la clase se actualice, considerando que ha pasado DeltaTiempo desde la anterior actualización.
nada **SetMass**(masa)	//Fija el valor de la masa del objeto representado.
masa **GetMass**(nada)	//Regresa el valor de la masa del objeto representado.

nIstPlayer.

nada **StartFW**(nada)	//Indica al personaje que camine hacia delante.
nada **StartBW**(nada)	//Indica al personaje que camine hacia atrás.
nada **StartRR**(nada)	//Indica al personaje que gire a la derecha.
nada **StartRL**(nada)	//Indica al personaje que gire a la izquierda.
nada **StopFW**(nada)	//Detiene al personaje de caminar hacia delante.
nada **StopBW**(nada)	//Detiene al personaje de caminar hacia atrás.
nada **StopRR**(nada)	//Detiene al personaje de girar a la derecha.
nada **StopRL**(nada)	//Detiene al personaje de girar a la izquierda.
nada **SetRunAccel**(aceleración)	//Fija el valor de la aceleración al correr.
aceleración **GetRunAccel** (nada)	//Regresa el valor de la aceleración al correr.
nada **SetRotSpeed**(velAngular)	//Fija el valor de la velocidad al girar.
velAngular **GetRotSpeed** (nada)	//Regresa el valor de la velocidad al girar.
nada **SetNCalModTar**(modelo)	//Fija el modelo que representa al personaje.
modelo **GetNCalModTar** (nada)	//Regresa el modelo que representa al personaje.

nIstNPC.

nada **EnableEventColl**(script)	//Habilita la ejecución de un script cuando sucede una colisión.
nada **DisableEventColl** (nada)	//Deshabilita la ejecución de scripts cuando ocurren las colisiones.
nada **SetNCalModTar**(modelo)	//Fija el modelo que representa al personaje.
modelo **GetNCalModTar** (nada)	//Regresa el modelo que representa al personaje.
nada **SetStatePathSeeking** (camino,tiempo)	//Coloca al personaje en modo se seguir al nPath *camino* en determinado *tiempo*.
nada **EnableEventView**(script,entidad)	//Habilita la ejecución de un *script* cuando el personaje ve a *entidad*.
nada **DisableEventView** (nada)	//Deshabilita la ejecución de scripts cuando ve a alguna entidad.

nIstObject.

nada **CreateModel**(malla,tex,posición)	//Fija la representación gráfica del objeto.
nada **Pick**(nada)	//Indica al objeto que ha sido levantado, para que haga invisible a su representación gráfica.
nada **ThrowAt**(posición)	//Indica al objeto que ha sido arrojado en el lugar *posición*, se debe volver visible.
nada **EnablePick**(nada)	//Habilita que el objeto sea tomado.
nada **DisablePick**(nada)	//Deshabilita que el objeto sea tomado.
nada **EnableEventColl**(script)	//Habilita la ejecución de un script cuando sucede una colisión.
nada **DisableEventColl** (nada)	//Deshabilita la ejecución de scripts cuando ocurren las colisiones.

nIstStruct.

nada **EnableEventColl**(script)	//Habilita la ejecución de un script cuando sucede una colisión.
nada **DisableEventColl** (nada)	//Deshabilita la ejecución de scripts cuando ocurren las colisiones.

nIstCamera.

nada **handleInput**(eventoEntrada)	//Se encarga de recibir las entradas del usuario con respecto a la cámara.
nada **SetStyleAimPath** (objetivo,camino,tiempo)	//Coloca el estilo de la cámara que apunte a *objetivo*, mientras que sigue a *camino* en determinado *tiempo*.
nada **SetStyleChase** (objetivo,altura,distancia)	//Coloca el estilo de la cámara que siga a *objetivo* de tal *altura* y a una cierta *distancia*.
nada **SetStyleFPS** (objetivo,altura)	//Coloca el estilo de la cámara como vista de primera persona de *objetivo* con determinada *altura*.

nada **SetStyleFree**(nada)	//Coloca al estilo de la cámara en libre, lo que permite que la controle el usuario.
nada **SetStyleLockedChase** (objetivo,orientación,distancia)	////Coloca el estilo de la cámara que siga a *objetivo* con cierta *orientación* y a cierta *distancia*.
nada **SetStyleLookPath** (camino,orientación)	//Coloca el estilo de la cámara que tenga cierta *orientación*, mientras que sigue a *camino*.
nada **SetStyleStationary** (objetivo,posición)	//Coloca a la cámara en el estilo de apuntar a *objetivo*, mientras se mantiene fija en *posición*.
Matrix44 **GetTransform**(nada)	//Regresa la posición y la orientación de la cámara mediante una matiz homogénea de 4 por 4.

nIstWorld.

nada **Trigger**(DeltaTiempo)	//Causa que la clase se actualice, considerando que ha pasado DeltaTiempo desde la anterior vez que se actualizo.
nada **AddCollideBSPNode** (nodoStruct,directorio)	//Carga de manera automática al servidor de colisiones todas las estructuras que se hallen en *nodoStruct*, buscando en el *directorio* definido.
nada **SetGravity**(gravedad)	//Fija el valor de la gravedad en el mundo.
gravedad **GetGravity** (nada)	//Regresa el valor actual de la gravedad en el mundo.

nIstCursor.

nada **StartFW**(nada)	//Indica al cursor que avance hacia delante.
nada **StartBW**(nada)	//Indica al cursor que avance hacia atrás.
nada **StartR**(nada)	//Indica al cursor que avance hacia la derecha.
nada **StartR**(nada)	//Indica al cursor que avance hacia izquierda.
nada **StopFW**(nada)	//Detiene al cursor de avanzar hacia delante.
nada **StopBW**(nada)	//Detiene al cursor de avanzar hacia atrás.
nada **StopR**(nada)	//Detiene al cursor de avanzar hacia la derecha.
nada **StopL**(nada)	//Detiene al cursor de avanzar hacia la izquierda.
nada **SetRunSpeed**(aceleración)	//Fija el valor de la velocidad al moverse.
aceleración **GetRunSpeed** (nada)	//Regresa el valor de la velocidad al moverse.
nada **SetN3DNodeTar**(modelo)	//Fija el modelo que representa al cursor.
modelo **GetN3DNodeTar** (nada)	//Regresa el modelo que representa al cursor.

nPath.

nada **BeginAddHPoint** (origen)	//Coloca a la clase en el modo de agregar puntos de coordenadas homogéneas al camino, recibe el nombre del punto *origen* del camino.
nada **AddHPoint** (puntoH)	//Agrega un punto de coordenada homogéneo *puntoH* por donde debe de pasar el camino.
nada **EndAddHPoint** (destino)	//Saca a la clase del modo agregar. Le indica además el nombre del punto *destino*.
punto **PointAt** (u)	//Regresa el *punto* perteneciente al camino que corresponde al arco de cuerda homogéneo u .

nVoiceServer.

número **GetNumVoices** (nada)	//Regresa el *número* de diferentes voces que soporta el servidor.
lógico **Ready**(nada)	//Regresa un valor *lógico* que indica sí el servidor esta listo para volver a producir voz.
resultado **SetVoice**(voz)	//Fija la *voz* con la que va a hablar el servidor. Regresa el *resultado* de sí fue o no posible realizar la acción.
resultado **Speak** (texto)	//Convierte el *texto* de entrada en voz. Regresa el *resultado* de sí fue o no posible realizar la acción.
nada **Trigger**(nada)	//Causa que la clase se actualice.

Para finalizar, es necesario indicar que varias de las clases que se analizaron en el capítulo anterior, las que pertenecen al motor Nebula o al paquete Cal3d. No se presentan en esta descripción de clases porque no hay necesidad de diseñarlas, esas clases ya están implementadas. Él agregarlas al análisis tiene como propósito mostrar la relación de las clases que si se van a diseñar con las ya existentes. En especial, si las clases ya existentes tienen la función de contribuir en alguna de las responsabilidades de las clases analizadas.

5.3. Optimización y Detalle de los Métodos.

Cuando se define algún método que dará cierto comportamiento a una clase, se crea un algoritmo que lo implemente. En la mayoría de los casos, el algoritmo es una simple secuencia computacional o procedimental que puede implementarse en un solo módulo de *software*, donde están contenidas todas las instrucciones necesarias para realizar dicha labor. Sin embargo, si el comportamiento es complejo, puede que sea necesario la modularización del método.

El motivo de está sección, por lo tanto, será el revisar los métodos planteados en la sección anterior. Poniendo especial atención al comentario que indica lo que se espera que realice el método. Esto con la intención de encontrar aquellos métodos que sean susceptibles de ser divididos en módulos, y por ende mejor detallados. También es propósito de este proceso de diseño la optimización de los métodos. En especial del aseguramiento de una utilización eficiente de los recursos y de facilitar la implementación.

A continuación se presentan algunos de los métodos más importantes detectados en el proceso de detallado. En esta sección se lista el método, así como a la clase a la que pertenece, que le fue aplicado el proceso de detallado y optimización. Además, se da una breve explicación de los nuevos módulos que se crearon y de su función.

Método:
nada **Trigger**(DeltaTiempo)
Clase:
nIstEntity.

Para entender la función de este método, hay que recordar que esta clase surgió del proceso de creación de jerarquías. Su propósito es aprovechar el polimorfismo para permitir que la clase nIstWorld pueda interactuar con los todos objetos que contiene sin ningún problema de a que clase pertenezcan. El propósito de este método es indicarle al objeto cuanto tiempo ha pasado desde la actualización anterior, además de aprovechar para que en ese momento el objeto se actualice. El algoritmo que realiza la secuencia de actualización en realidad esta implementado en todas las subclases que heredan de nIstEntity, tal es el caso de nIstPlayer, nIstNPC o nIstObject. Por lo tanto, la implementación del método Trigger en la clase nIstEntity no hace nada en realidad. Sin embargo, es posible utilizar a esta clase como medio para forzar un diseño mas modular en todas sus subclases. La actualización de una entidad, independientemente de a que clase pertenezca, puede ser dividida en al menos tres la actualización de tres aspectos: La actualización de su estado, de su interacción física con otros objetos y de su posición en el espacio.

A cada una de las tres partes de la actualización se les puede asignar un método. La implementación de estos métodos, aun así quedará como responsabilidad de las subclases de nIstEntity, pero así se logra que cada implementación tenga un propósito mas claro. En lugar del método Trigger, se propone que la clase nIstEntity tenga los siguientes tres métodos responsables de su actualización.

nada **Trigger**(DeltaTiempo)	//Causa que la clase se actualice su estado solamente, considerando que ha pasado DeltaTiempo desde la anterior actualización.
nada **Collide**(nada)	//Causa que la clase se actualice en relación a la colisión con otros objetos.
nada **UpdatePosition**(DeltaTiempo)	//Actualiza la posición del objeto, considerando los cambios sufridos por su estado interno y por las colisiones con otros objetos.

Con esta modularización se indica de manera mas precisa la responsabilidad de cada método. Sin embargo, también tiene como consecuencia que algunos de estos métodos heredados a todas las subclases de nIstEntity se implementen para no realizar nada. Tal es el caso del método UpdatePosition para la clase nIstStruct, ya que esta clase está diseñada para representar a todas las estructuras que nunca se van a mover.

Método:

nada **handleInput**(eventoEntrada)

Clase:

nIstCamera.

Este método fue diseñado para que la cámara pueda recibir las entradas del usuario y ser controlada mediante ellas. Pero, a fin de optimizarlo, se debe de analizar cuales son las situaciones en las que el objeto cámara realmente debe recibir entradas del usuario. Al revisar el diagrama de transiciones de la clase nIstCamera en [4.5.]. Se puede notar que solamente hay dos estados en los que realmente se reciben entradas del usuario, ya que éstos son los que implican estilos de cámara controlados por el usuario. El estilo de cámara libre y el de persecución son los estilos que mediante entradas del ratón permiten al usuario modificar la posición u orientación de la cámara. Una posible forma de separar en varios módulos a este método es separando la funcionalidad necesaria para manejar las entradas de usuario cuando el objeto se encuentra en el estilo libre y cuando se encuentra en el estilo de persecución. De esta manera, la propuesta de optimización incluye la creación de tres métodos: Un método homónimo que se encargue del manejo general de las entradas de usuario, responsabilizándose de modificar los atributos necesarios del objeto. Un método que modifique la posición y orientación de la cámara cuando ésta se encuentra en estilo libre n base a eso atributos y otro método que se encargue de la misma labor sí el objeto se encuentra en estilo de persecución. A continuación se muestra la descripción de los métodos:

nada **handleInput**(eventoEntrada)	//Se encarga de recibir las entradas del usuario relativas a la cámara. Modifica los atributos de ésta dependiendo de *eventoEntrada*.
nada **handleViewer**(nada)	//En base a ciertos atributos de la clase, este método maneja las entradas del usuario cuando la cámara se encuentra en estilo libre.
nada **handleChaseViewer**(nada)	//En base a ciertos atributos de la clase, este método maneja las entradas del usuario cuando la cámara se encuentra en estilo de persecución.

Un detalle a remarcar, es el hecho que los dos últimos métodos no reciben ningún parámetro. A primera vista esto puede parecer extraño, ya que estos métodos tienen la supuesta responsabilidad de manejar las entradas de los usuarios. El hecho es que los prototipos de esos métodos no aparentan permitir la recepción de alguna información. Lo que en realidad sucede, es que la primera fase de manejo de la información de entrada del usuario ya fue realizada por el método handleInput, y su resultado almacenado en atributos de la clase directamente, que son accesibles a los dos métodos especializados.

Método:

nada **Trigger**(DeltaTiempo)

Clase:

nIstWorld.

La decisión de optimizar este método surge debido a la responsabilidad de la clase nIstWorld de contener a todos los personajes y objetos que existen en el mundo de la historia. Por lo tanto, como parte de su proceso de actualización, la clase debe de actualizar a todos los objetos que contiene. En cuanto al aspecto semántico de esta declaración un hay ningún problema, en cambio, en el aspecto de diseño de *software* si lo hay. El tener un solo método que se encargue de la actualización del estado interno de la clase nIstWorld y de la actualización de los objetos que esta contiene presenta un serio problema. El problema es que al querer modificar el código relacionado con la actualización de los atributos de la clase mundo nos tendremos que enfrentar también al código de actualización de entidades, también sucede lo mismo viceversa.

La solución para optimizar este método es la creación de un módulo encargado de la actualización de la clase mundo, y de otro modulo encargado de indicarle a todos los objetos contenidos por la clase que se deben de actualizar. De cualquier manera, el método encargado de actualizar los objetos seguramente será llamado dentro del método encargado de actualizar a la clase. Aun así, esta separación de ámbitos redundará en una codificación mas óptima y robusta. La descripción de los dos métodos es la siguiente:

nada **Trigger**(DeltaTiempo)	//Causa que la clase se actualice su estado interno solamente, considerando que ha pasado DeltaTiempo desde la anterior actualización.
nada **UpdateEntities** (nada)	//Método privado que causa que se actualicen todos los objetos contenidos en el objeto mundo.

De la misma forma que en la optimización anterior, se puede ver que aquí también el nuevo método UpdateEntities no recibe ningún parámetro. La razón de esto es la misma que el caso anterior, al hacerse cargo de la actualización de los atributos de la clase, el método Trigger aumenta el atributo del avance del tiempo en la clase nIstWorld. Permitiendo que a través de este atributo interno el método UpdateEntities sepa cuanto tiempo a pasado, sin la necesidad de que se le envíe un parámetro indicándole el lapso de tiempo entre cada actualización.

Método:

nada **EndAddHPoint** (destino)

Clase:

nPath.

Al revisar la forma como se diseñaron los pasos para construir la curva que forma el camino de la clase nPath se encontró un problema que mengua la robustez con la que esta clase puede ser operada. Ya que posibilitaba la destrucción o modificación irreversible de la

curva por una simple llamada errónea al método BeginAddHPoint o al método AddHPoint. Esto debido a que al invocarlos modifican o inicializan los puntos que definen a la curva. Para evitar ese problema, es necesaria la creación de variables de almacenamiento temporales que guarden todos los puntos que se quieran agregar y que se inicialicen al comenzar este proceso. Por lo tanto también es necesaria la creación de un nuevo método que escriba el contenido de las variables temporales en las variables utilizadas por la curva verdadera. La ventaja de esta disposición es que solo hay un método capaz de modificar la curva que utiliza la clase como camino. De esta forma se reduce de sobre manera la posibilidad que una llamada errónea a un método pueda destruir la curva que representa al camino.

nada **EndAddHPoint** (destino) //Saca a la clase del modo agregar puntos a las variables temporales. Le indica además el nombre del punto *destino*.

nada **CreateNurbCurve** (nada) //Copia el contenido de las variables temporales que contienen los puntos a las variables que son usadas por la curva.

Como último punto de esta sección es importante indicar que son mas las optimizaciones realizadas a los métodos descritos en [5.2.]. En la implementación final del API que las que aquí se muestran, pero es debido a motivos compendio que éstas no se presentan.

5.4. Experiencia de Implementación.

La intención de los dos últimos capítulos ha sido presentar el proceso de análisis y diseño orientado a objetos que precede a la implementación de todo proyecto desarrollado bajo este paradigma. La intención de esta última sección es transmitir la experiencia que se obtuvo al implementar el API después de la realización de estos procesos. Con especial énfasis en los beneficios que trajo al proyecto el uso de los mismos.

Además, esta sección también tiene la intención de transmitir las complicaciones que surgieron al implementar algunos de los métodos descritos en el modelado de diseño. Así como de las complicaciones que surgieron debido a las librerías de *software* y al motor gráfico que finalmente se escogieron. En particular, se presentan las complicaciones que surgieron debido a la naturaleza de *software* libre que tenían la mayoría de estos. Con excepción de la implementación de la clase nVoiceServer, donde se utilizó un conjunto de herramientas de desarrollo de aplicaciones desarrolladas por Microsoft, y que permiten el reconocimiento y la conversión de texto a voz.

Sin lugar a dudas, la aplicación de la ingeniería de *software* al proceso de creación del API de historias interactivas causó que el tiempo de desarrollo se redujera. En especial, al considerar que se trata de un área sumamente nueva y en la cual el desarrollador no tenía experiencia previa. Se le puede atribuir al análisis orientado a objetos (OO), y en especial a la técnica de análisis de dominio, que un desarrollador con una nula experiencia haya podido obtener las clases y las relaciones necesarias entre ellas para poder implementar el API en tan poco tiempo. Ya que al contar con las clases necesarias, el desarrollo del API rápidamente se pudo enfocar al diseño e implementación de las mismas. Además, el análisis de dominio facilitó el satisfacer la necesidad de producir *software* reutilizable. Ya

que atrajo la atención a esta cuestión desde las etapas iniciales del desarrollo. Permitiendo obtener los probables requisitos de un amplio grupo de aplicaciones en el campo de las historias interactivas, mientras que se cumplía con los requisitos específicos de la aplicación Calakmul virtual. El proceso de Diseño OO también trajo sus beneficios al desarrollo del proyecto. En especial, permitió la detección de varios métodos que eran susceptibles de ser divididos en métodos mas pequeños, esto con la intención de evitar la existencia de métodos excesivamente complejos.

Sin lugar a dudas, que uno de los requisitos funcionales mas difíciles de satisfacer fue el de las clases responsables de los modelos de los personajes y de sus animaciones. Con esto obviamente se refiere a las ya antes mencionadas clases nCal3DModel, nCal3DCoreModel y nCal3DMaterial. Y aunque estas clases no fueron implementadas, sino que, debido a su naturaleza de proyectos de *software* de código abierto, estas fueron agregadas al motor de juegos Nebula. La falta de soporte y documentación lo volvió en un proceso complicado. Y aunque la existencia de este tipo de problemática es conocida en el ámbito de los proyectos de código libre. La falta de documentación y de soporte en este paquete en particular fue muy notoria. En un grado un poco menor, la implementación de las curvas que definen el camino contenido en la clase nPath también presentaron esta problemática. Lo que la atenuó fue el hecho de que la librería seleccionada para implementar esas funciones era relativamente mas conocida, se trata de la librería de nurbs (Non Uniform Rational B-splines) conocida como NURBS++. Esta librería contiene clases que representan curvas y superficies nurbs de varias dimensiones. El problema con esta librería es el excesivo uso de templates de C++, que la vuelven muy complicada de entender.

Para finalizar, la experiencia en la implementación de la clase nIstVoiceserver, la clase encargada de la conversión de texto a voz, fue una experiencia agridulce. Ya que, en cuanto a la calidad y cantidad de documentación disponible, ésta era mucho mayor que la de los proyectos de código abierto. En cuanto al acceso al código fuente, éste era nulo. Y en cuanto a la complejidad de la interfaz para que otros objetos se comunicaran con ella ésta era excesiva. Esto se debió al uso de tecnologías muy complicadas por parte de Microsoft como por ejemplo COM, o el uso de cadenas de caracteres Unicode[1].

Aun después de está larga lista de quejas, que surge de la reutilización de código implementado por otras personas. Sigue siendo necesario remarcar, que la experiencia de implementación, indica que fue mejor reutilizar el *software* creado por otras personas que tener que implementar todas esas funciones desde cero.

[1] Unicode es un esquema de codificación de caracteres que usa 2 bytes por cada carácter. La Organización Internacional de Estándares (ISO) define un número en el rango de 0 a 65535 (2^{16}-1) para todo carácter y símbolo en cada mensaje. El esquema Unicode es utilizado en todos las cadenas de caracteres de las nuevas librerías de Microsoft.

Capítulo 6.

Aplicación y Trabajo Futuro.

6.1. Calakmul Virtual.

6.1.1. Antecedentes.

La intención de este capítulo es mostrar la manera como el API se está utilizando en la vida real. Desde un principio, el API estuvo proyectado para utilizarse en el desarrollo del proyecto Calakmul virtual, que se realiza en el Laboratorio de Interfaces Inteligentes. Por lo tanto, como primera parte de este capítulo, se presentará la experiencia que hasta ahora se ha tenido de su uso en este proyecto. De antemano se puede advertir, al ver la forma como el API satisface con los requisitos del proyecto, que varios de los conceptos y funcionalidades incluidos en éste están fuertemente influenciados por los requisitos del proyecto Calakmul virtual. Cabe recordar, como se mencionó en el capítulo 3, que el diseño del API siempre tuvo como principal objetivo su uso en el desarrollo de este proyecto. Aún así, en la fase de diseño siempre se tuvo en cuenta los requisitos y compromisos que el *software* planeado para ser reutilizable requiere.

El objetivo del proyecto es crear un sistema que permita que los usuarios puedan conocer el sitio arqueológico de Calakmul. Para lograr que el usuario pueda conocer el sitio, se necesita que el sistema muestre modelos 3D de los diferentes edificios del lugar, además se necesita que las representaciones de éstos tengan el detalle suficiente que permita la identificación de cada uno. La necesidad de desarrollar un sistema con tales características surge debido al relativamente reciente descubrimiento de este sitio arqueológico y la consecuente falta de rutas de acceso a éste. Por lo que, al considerar la belleza e importancia arqueológica del mismo, se decidió por el uso de un viaje "virtual" como posible medio de difusión del mismo. Además, se necesita que el recorrido, experimentado por el usuario a través del sistema, sea capaz de interesar y de mantener la atención del usuario. De ahí que se propusiera su desarrollo como un sistema de historias interactivas en computadora. Y por la necesidad de presentar gráficamente los acontecimientos de la historia, es necesario que el enfoque del sistema sea hacia el drama interactivo. Recuérdese que al hablar de drama [1.1.3.], no se implica la presentación de hechos trágicos. Si no que la presentación de los acontecimientos se haga de manera directa, como en una obra de teatro y no como en una narración, con la intención de mantener interesado al usuario.

El problema que el sistema de Calakmul virtual resuelve es la difusión y divulgación de un importante sitio arqueológico recién descubierto y por ende mal comunicado. Además de resolver esos problemas, el sistema no debe requerir *hardware* extremadamente caro o especializado para funcionar. Ya que los recursos de los museos o exposiciones donde se utilice el sistema pueden ser reducidos.

Al analizar los requisitos del sistema, y conociendo la problemática existente en el área de las historias interactivas, se decidió por el desarrollo de un API capaz de agilizar el desarrollo del proyecto Calakmul virtual y de proveer a los desarrolladores de otros sistemas de historias interactivas con una base para facilitar el desarrollo de sus propios proyectos.

6.1.2. Funciones del Sistema Calakmul virtual y el API de HIC.

La función del sistema Calakmul es presentar el sitio arqueológico de manera que parezca que se está realizando una visita guiada. El sistema debe presentar un guía virtual que lleve a los usuarios a los sitios de interés y les explique todo lo relativo a éstos y a los objetos que ahí se encuentran. Obviamente que la base para la implementación del guía debe de ser un agente con estado interno. Además, el sistema debe de permitir al usuario la interacción con los sitios, los objetos y los personajes que se encuentren dentro de la historia. Esta interacción debe ser suficientemente amplia como para que el usuario se sienta interesado en lo que le presenta el sistema, que no sienta que se trata de una simple visita donde el guía le arroja gran cantidad información acerca de cada lugar que pasan, pero lo suficientemente acotada como para hacer factible el desarrollo del sistema.

Es necesario subrayar que el sistema además de la función de educar, función realizada principalmente mediante los diálogos del guía virtual, tiene la función entretener al usuario. Función que concuerda con las propuestas de la pedagogía moderna, las cuales proclaman la necesidad de que la actividad educativa sea adoptada por los educandos como algo apetecible. Y no hay mejor manera de volver a la educación algo apetecible que él volverla algo entretenido. Cabe remarcar a la necesidad del sistema de cumplir esta función como la principal responsable de la decisión de implementar el sistema de Calakmul virtual a través de un sistema de historias interactivas. Decisión que se tomó con la expectativa de tornar una visita en la que el usuario solo sigue al guía mientras que recibe información de manera pasiva, en una experiencia dinámica y cambiante donde la interacción con los objetos y personajes le permite conocer mas acerca de la historia y descubrir lugares hasta ese momento ocultos.

De la funcionalidad presentada por el sistema y que esta implementada en el API se puede mencionar rápidamente: La habilidad de seguir caminos del agente que controla al guía virtual, la capacidad de convertir texto a voz y que está implementada en el API gracias a la inclusión de una librería de Microsoft, la obvia capacidad de mostrar y animar con una gran calidad gráfica y detalle a los modelos 3D implementada gracias a la inclusión del motor de juegos Nebula, la detección de colisiones entre los personajes y los objetos que también está gracias al motor de juegos, etc.

6.1.3. Experiencia en el desarrollo de Aplicación al Usar el API de HIC.

La experiencia de desarrollar el sistema de Calakmul virtual utilizando el API ha permitido distinguir las ventajas de utilizar la arquitectura abierta en el desarrollo de sistemas complejos y en particular en los sistemas que requieren de equipos de desarrollo altamente multidisciplinarios. Al usar el API en este proyecto, se descubrió que tener toda la funcionalidad necesaria accesible a través de una interfaz clara permitía que el equipo de desarrolladores se enfocara mas en el aspecto creativo. Además, este cambio de enfoque facilitó el funcionamiento del equipo ya que redujo los roces que surgen cuando la composición de éste incluye la misma cantidad de programadores que de artistas. El resultado de este cambio fue que la mayoría del talento y esfuerzo se le dedico al desarrollo de contenido del proyecto, mientras que solo se requirió de un poco de atención a la programación de los scripts de comportamiento y de respuesta de los agentes.

En conclusión, la experiencia de desarrollo de Calakmul virtual utilizando el API ha sido buena. Conforme avance el desarrollo del proyecto, porque en el momento en que esto es escrito el proyecto Calakmul continua en desarrollo, será interesante ver si las previsiones y propuestas realizadas durante el análisis y diseño del API permiten que el desarrollo continué de frente y sin inconvenientes.

6.2. Trabajo Futuro.

Como la intención al desarrollar este API no solo fue que se usara en el proyecto de Calakmul, sino que se espera que sea utilizado en otros proyectos. A continuación se muestra una serie de características que ampliaría de manera notoria el campo de aplicación del API. Que características no fueran implementadas en este API obedece principalmente a dos razones; que la complejidad y falta de algoritmos estándar hace que éstas sean muy difíciles de implementar y que su uso en el proyecto Calakmul virtual era dispensable.

6.2.1. Animación basado en Cinemática Inversa.

El sistema que presenta el API actualmente permite la animación de los modelos mediante un esqueleto controlado por animaciones prefabricadas. Estas animaciones no pueden cambiar en tiempo de ejecución para adaptarse a la situación particular en la que son utilizadas, ya que solo pueden creadas en la fase de diseño por un modelista u obtenidas mediante la captura de movimiento. Cabe mencionar que este sistema satisface las necesidades de animación de una gran cantidad de proyectos, incluyendo el de Calakmul virtual.

Aún así, la flexibilidad que provee un sistema de animación basado en cinemática inversa no tiene rival. En situaciones donde la animación de los modelos esta sujeta a restricciones de espacio, a interacción con objetos de posición variable o a colisiones con amientes dinámicos; ésta es la opción mas viable para obtener animaciones correctas. La animación basada en cinemática inversa proporciona control directo sobre la colocación final de los componentes del modelo, resolviendo de manera automática los ángulos para las rotaciones que pondrían a los huesos del esqueleto que controla al modelo en la posición deseada. Por ejemplo, esta capacidad se podría aplicar a la animación usada para mostrar que el modelo toma un objeto pequeño con posición variable. El sistema calcularía los ángulos necesarios para colocar la mano en la posición del objeto, mientras que en el sistema actual el desarrollador se tienen que conformar con que el modelo solo tome los objetos que están en la posición adecuada o que el objeto sea tomado aún cuando la animación nunca haya colocado la mano sobre el objeto.

Sin lugar a dudas, que si el API contara con esta característica, esto lo haría mucho mas interesante y usable en un rango mas amplio de situaciones. De cualquier modo hay que indicar que la cinemática inversa es una técnica muy compleja de aplicar para estructuras tan complejas como el cuerpo humano. Y que sus resultados, en cuanto a realismo e integridad dinámica dejan que desear en comparación con la captura de movimiento. Para información mas acerca de este tema véase [DUA02].

6.2.2. Expresión Facial.

La posibilidad de que los modelos tuvieran expresiones faciales es sin duda una de las características faltantes mas importantes dentro de la actual implementación del API. Y es que tratándose de un API enfocado principalmente al desarrollo de dramas interactivos, en los cuales la expresividad de los actores virtuales es vital, su ausencia es muy adversa. Pero hay una muy buena excusa para esto, sólo la complejidad de implementar esta característica en un modelo es suficiente para un tema de doctorado. Simplemente al considerar la cantidad de músculos que intervienen para formar una expresión, sin olvidar la gran cantidad de expresiones que tiene un rostro humano y sus posibles combinaciones, es que nos damos cuenta de lo complejo de esta labor.

Sin lugar a dudas, si esta característica es agregada al API algún día, será una adición que aumentara la calidad de los actores virtuales en varios ordenes de magnitud. Aumento de calidad en los actores que indudablemente favorecería la calidad de toda la obra dramática, por lo mencionado en [1.1.3]. Mas información acerca de la creación y composición de expresiones faciales consúltese en [PER97].

6.2.3. El Servidor de Historias.

Una última característica que podría agregarse al API sería la de incluir una clase que proveyera la funcionalidad de un servidor de historia. Cuya responsabilidad sería la de crear, de acuerdo con las acciones del usuario y con una serie de parámetros indicados por el autor, el flujo y los eventos de la historia con un nivel muy elevado de abstracción. La responsabilidad de ésta clase estaría delimitada por el grado de automatización con que se implemente.

En el caso completamente automatizado, el servidor simplemente generara la historia de acuerdo con un concepto muy general que el autor le indique. Mientras en un caso menos automatizado, el servidor controlara el flujo de la historia mediante la selección de escenas antes construidas por el autor.

En realidad, más que ser una característica que el API tenga que incluir, éste es uno de los campos donde se puede aprovechar el API para acelerar su desarrollo. Ya que el API provee a los investigadores de una base funcional sobre la cual empezar sus investigaciones.

6.3. Pruebas y Resultados.

Debido a que el proyecto Calakmul virtual se encuentra en una etapa inicial de desarrollo, todavía no se puede dar un veredicto definitivo acerca de la ayuda que el API de historias interactivas le proporcionó. Por lo tanto, las pruebas a las que se puede someter el API sólo evalúan cada componente de programa por separado. Aunque el propósito de este proyecto es construir un API, a final de cuentas, se tiene que construir una pequeña aplicación que permita realizar estas pruebas. También es importante señalar, que debido a la naturaleza de varios de los requisitos de *software*, varias de las pruebas que se apliquen tendrán un carácter cualitativo.

A continuación se presentan las pruebas realizadas a los diferentes componentes de programa. Se da una pequeña descripción de ella y se compara con el resultado que el requisito indicaba. Como se mencionó anteriormente, la naturaleza de varias de estas pruebas es únicamente cualitativa.

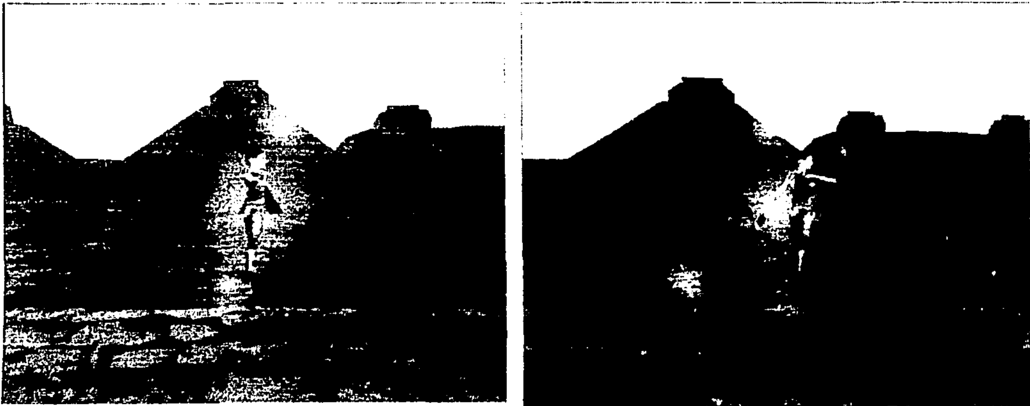
Prueba 1. Desempeño de los estilos de cámara de la Clase nIstCamera.

Esta prueba pretende comprobar la forma como la clase nIstCamera cumple con los diferentes tipos de tomas que se le encargan. La comparación se realizará contra los requisitos que cada uno de ellos debe de cumplir y que se mencionaron anteriormente. A continuación se presentan los resultados de comparar el desempeño de la cámara en cada estilo contra lo mencionado en los requisitos.

El estilo estacionario debe de mantener la cámara en una misma posición mientras que cambia su orientación para poder seguir a algún objeto presente en la escena. Para probar el desempeño en este estilo se llamó al método:

```
void nIstCamera::SetStyleStationary(n3DNode *target,n3DNode *camer);
```

Como muestra la secuencia de imágenes, el desempeño es satisfactorio y cumple con lo requerido.



TESIS CON
FALLA DE ORIGEN

Figura 6.1. Secuencia de imágenes. Cámara estilo estacionario.

El estilo de persecución debe de mantener la cámara a una cierta distancia del objetivo, mientras que cambia su orientación según la entrada que del usuario a través del ratón. Para probar el desempeño en este estilo se llamó al método:

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

La secuencia de imágenes muestra un desempeño es satisfactorio y cumple con los requisitos al siempre mantener una distancia, mientras que el ángulo con respecto a la horizontal cambia de acuerdo a la entrada del usuario.

TESIS CON
FALLA DE ORIGEN

Figura 6.2. Secuencia de imágenes. Cámara estilo persecución.

El estilo de persecución fija es un caso particular del estilo de persecución, pero en él que se fija el ángulo con la horizontal y con la vertical. El método que se llamó es el siguiente:

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

Como muestra la secuencia de imágenes la cámara persigue al objetivo indicado mientras mantiene su orientación y posición relativa a éste. Por lo que también cumple con lo requerido.

TESIS CON
FALLA DE ORIGEN

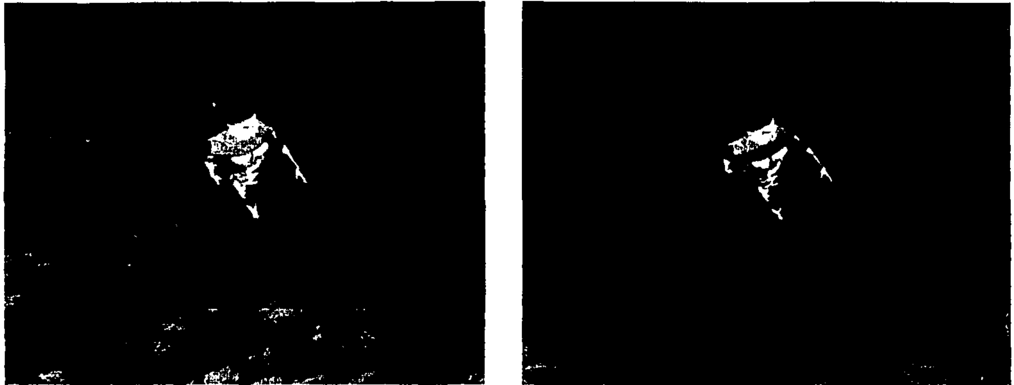
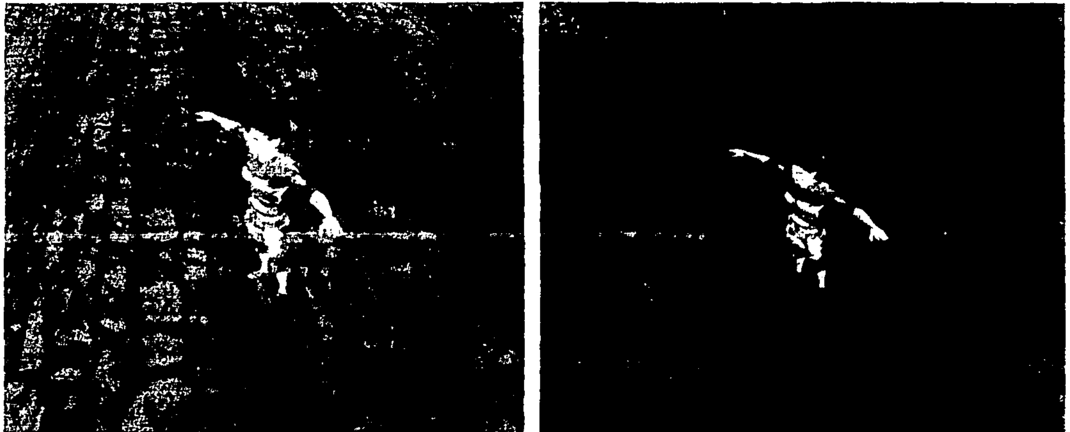


Figura 6.3. Secuencia de imágenes. Cámara estilo persecución fija.

El estilo de seguir una ruta apuntando requiere que la cámara se mueva a lo largo de un camino mientras que apunta hacia un objetivo determinado. El método para activar este estilo es el siguiente:

```
void nlstCamera::SetStyleAimPath(n3DNode *target,const char *orig,const char *dest,float len);
```

Con la secuencia de imágenes se observa que la cámara siempre apunta al objetivo indicado mientras cambia su posición.



TESIS CON
FALLA DE ORIGEN

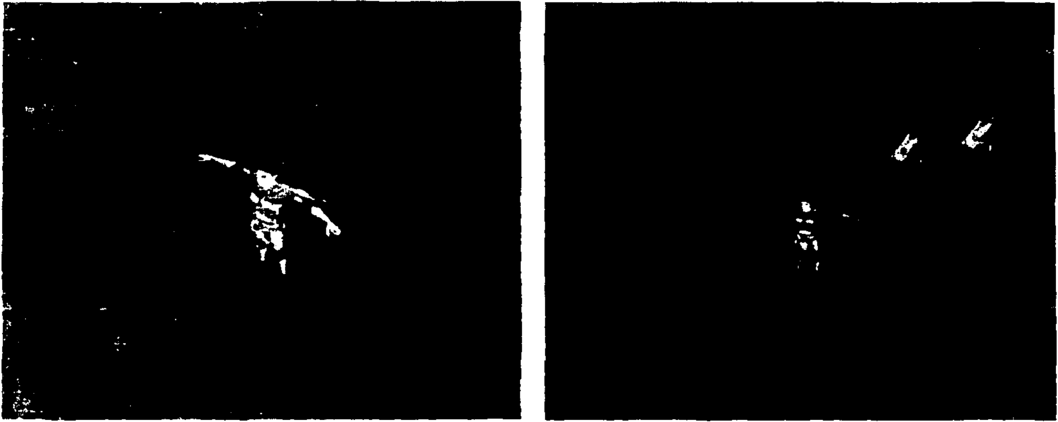
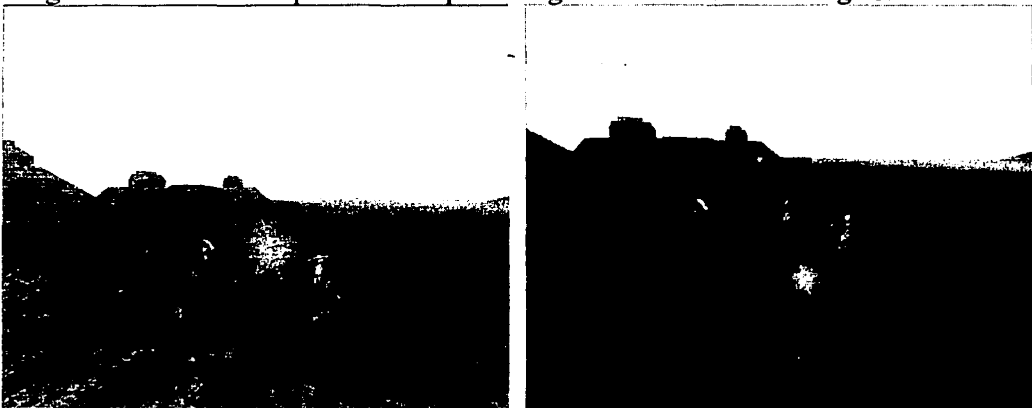


Figura 6.4. Secuencia de imágenes. Cámara estilo apuntando y siguiendo una ruta.

El estilo de ruta con orientación determinada requiere que la cámara se mueva a lo largo de un camino mientras que la orientación de la cámara cambia según valores predefinidos. La llamada para activar este estilo es el siguiente:

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

El desempeño de este estilo no cumple con todo lo que se espera. Y aunque en la mayoría de las ocasiones la orientación de la cámara es la correcta, siguiendo los valores predefinidos e interpolando correctamente entre ellos. Hay veces en las que la cámara gira en sentido opuesto del deseado, problema atribuible a la interpretación de los ángulos negativos, evitando que se logre el efecto de cámara deseado. En la siguiente secuencia de imágenes se muestra este problema. Y que sin lugar a dudas debe ser corregido.



TESIS CON
FALLA DE ORIGEN

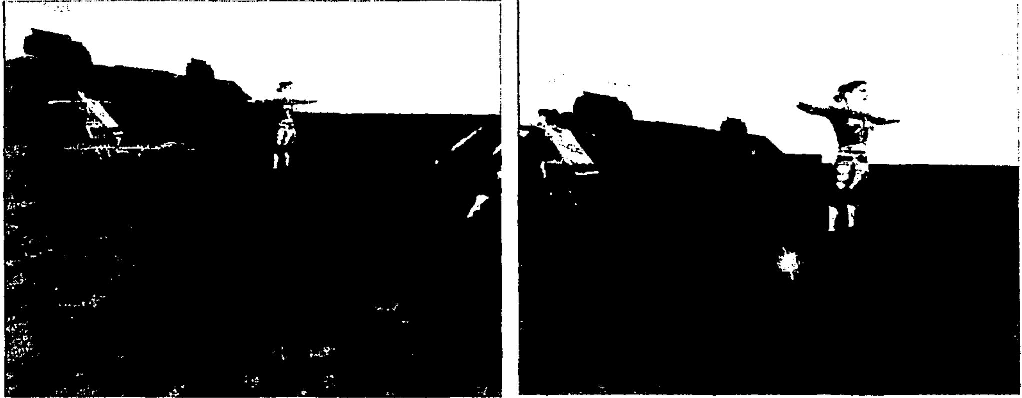
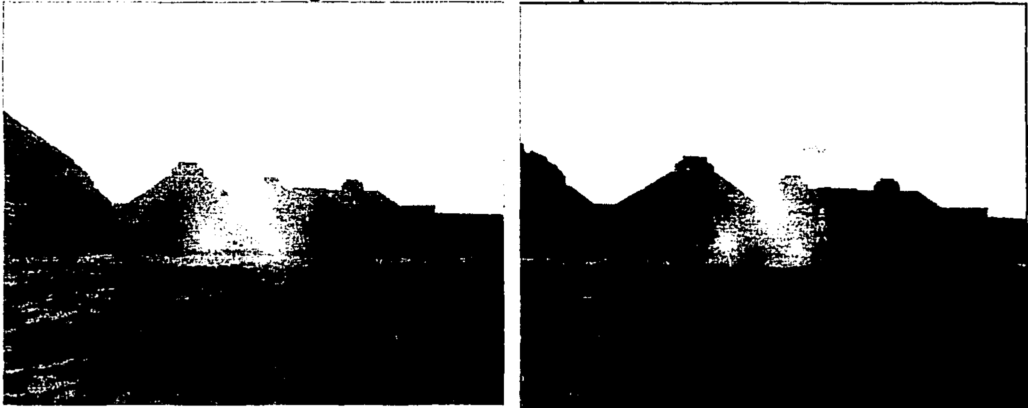


Figura 6.5. Secuencia de imágenes. Cámara estilo orientación predefinida y siguiendo ruta.

El estilo de vista en primera persona debe colocar a la cámara en el punto de vista de algún personaje que se encuentre en la escena. Obviamente que también debe imitar cualquier cambio en la orientación del mismo. Para probar el desempeño en este estilo se llamó al método:

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

La secuencia de imágenes muestra un desempeño adecuado.



TESIS CON
FALLA DE ORIGEN

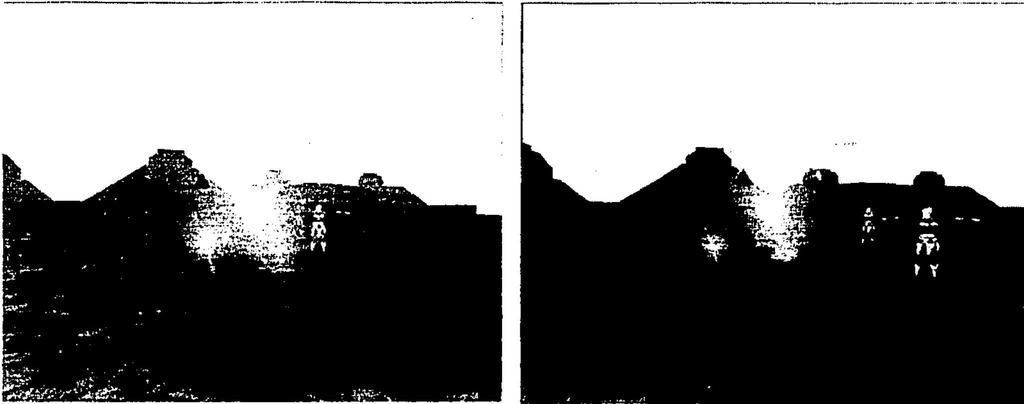
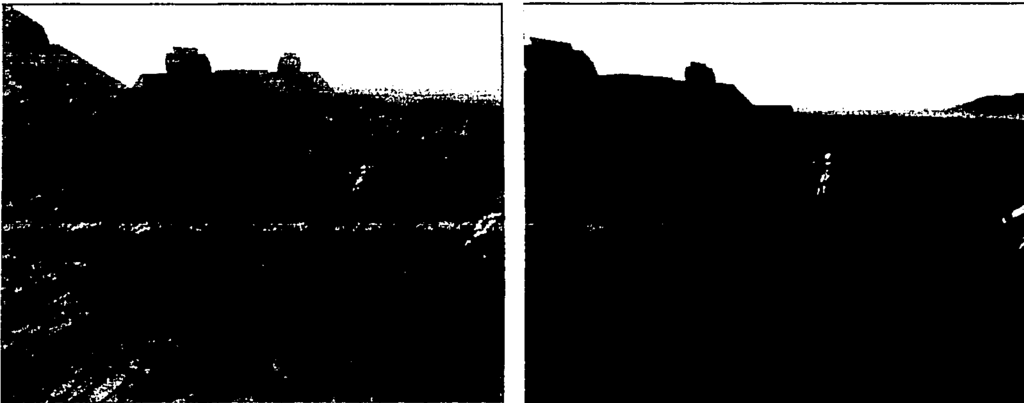


Figura 6.6. Secuencia de imágenes. Cámara estilo primera persona.

El estilo de cámara libre requiere que ésta cambie su posición y orientación de acuerdo con las entradas que el usuario haga a través del ratón. Para activar este estilo se realiza la llamada del siguiente método:

```
void nlstCamera::SetStyleFree();
```

La secuencia de imágenes muestra la respuesta de la cámara a las entradas del usuario, cabe mencionar que la respuesta fue satisfactoria con las entradas del usuario.



TESIS CON
FALLA DE ORIGEN

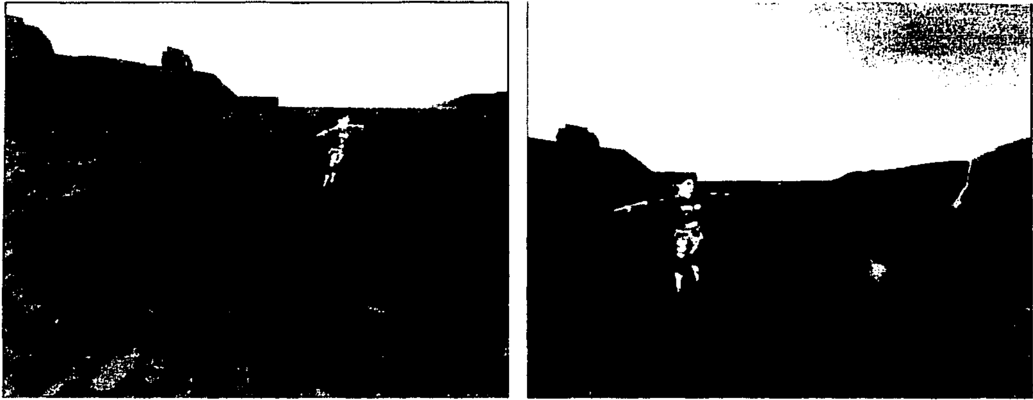


Figura 6.7. Secuencia de imágenes. Cámara estilo libre.

Prueba 2. Animación de la Clase nCal3DModel.

Esta prueba consiste en comprobar de manera cualitativa la calidad de las animaciones que presentan los modelos graficados gracias a la inclusión del paquete nCal en el API de historias interactivas. Es necesario realizar estas pruebas para comprobar que lo proclamado por los desarrolladores del paquete es cierto. En la primer serie de figuras se muestra al modelo cuando se llamó al método de ciclo de animación:

```
nCal3DModel::NewCycle (int id, float weight, float delay);
```

En esta serie se puede notar que el modelo realiza la animación en un ciclo, capacidad sin lugar a dudas requerida para movimientos repetitivos como el caminar. En términos simples, se puede concluir que la calidad de la animación mostrada por el modelo es buena.

Figura 6.8. Secuencia de imágenes. Animación cíclica correr.

TESIS CON
FALLA DE ORIGEN

A continuación se realiza una prueba al llamar el método de animaciones sencillas, que son ejecutadas sólo una vez, y que es utilizado principalmente para los movimientos de los brazos. El método que llama una animación sencilla es:

void nCal3DModel::**Action** (int id, float delayIn, float delayOut);

En este caso la animación llamada fue una que muestra al modelo realizando un tiro con arco. El arco y la flecha no se colocaron para permitir una mayor visibilidad. Al comparar cualitativamente la animación modelada con lo mostrado gráficamente, también se puede decir que el resultado es satisfactorio, tal como se puede ver en la siguiente secuencia.

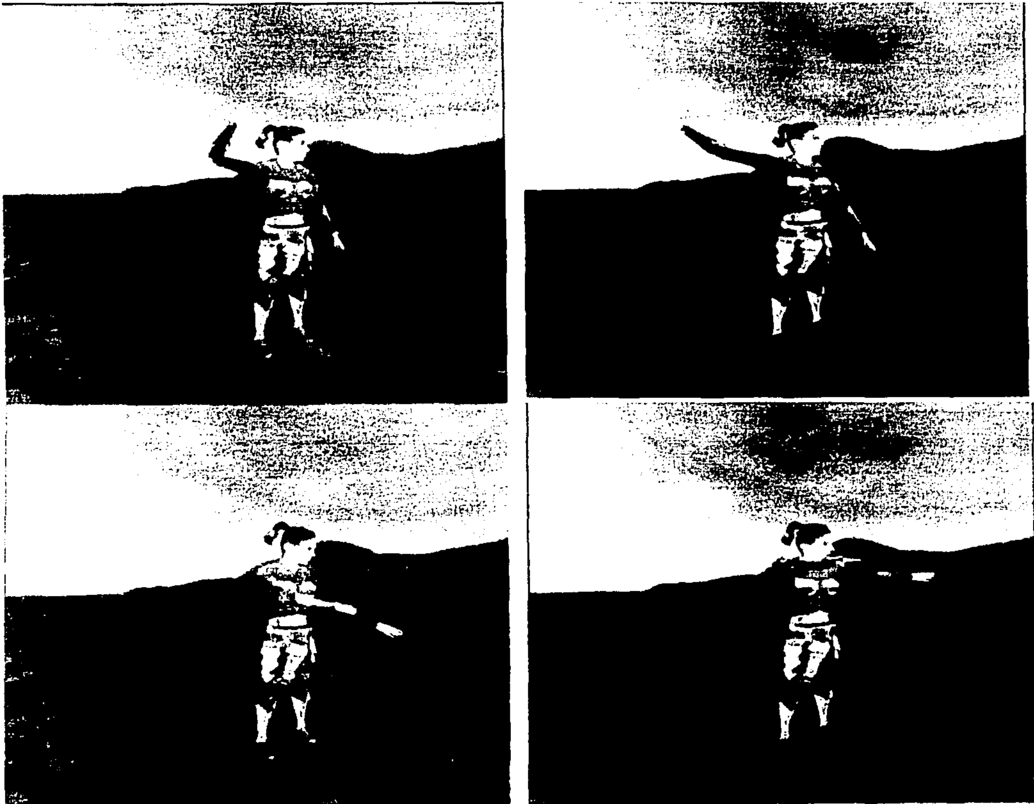


Figura 6.9. Secuencia de imágenes. Animación sencilla de disparo con arco.

FALLA DE ORIGEN

Para finalizar esta serie de pruebas, se comprobaron las capacidades de mezclado de animaciones que es proclamado por los desarrolladores del paquete, para realizar esto, primero se llamó al método de animación en ciclo y luego al de animación sencilla. La siguiente serie de imágenes muestra el resultado. Se puede ver que los resultados fueron buenos.

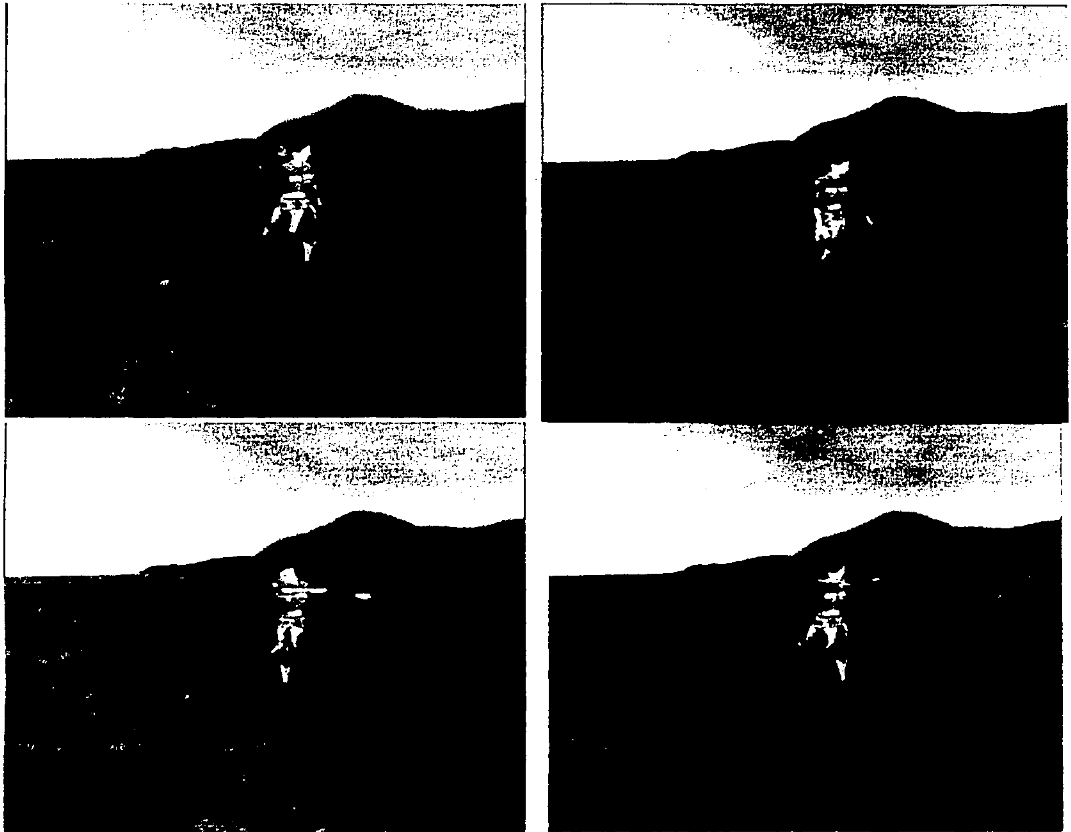


Figura 6.10. Secuencia de imágenes. Mezcla de animaciones: Disparar y caminar.

TESIS CON
FALLA DE ORIGEN

Prueba 3. Error en las Rutas de la Clase nPath.

En esta prueba se desea medir el error absoluto y el relativo que existe entre los puntos originales y los que regresa la clase nPath. También se desea obtener el error promedio. Esta prueba se realiza con la intención de medir precisamente como la curva nurbs modela la curva original que recorre el camino. El siguiente cuadro muestra los puntos originales por los que pasa el camino, los puntos que regreso la clase nPath y los errores que éstos tuvieron. La curva que se modelo fue una parábola que se descrie mediante la siguiente función:

$$z = 450 - 0.05*(x - 200)^2$$

Cuadro de Comparación de la Clase nPath							
Coordenadas Originales para la Construcción del Camino		Coordenadas Obtenidas de nPath		Coordenadas Obtenidas a través de la función.		Error de lo Obtenido entre nPath y la función.	
x base	**z base**	**x de nPath**	**y de nPath**	**x de función**	**y de función**	**Error absoluto**	**Error relativo**
0	-1550	0	-1550	0	-1550	0	0
20	-1170	38.333	-865	38.333	-856.8109	-8.189055	0.00946712
40	-830	60	-536.667	60	-530	-6.667	0.01242297
60	-530	76.667	-313.333	76.667	-310.5514	-2.781555	0.00887731
80	-270	83.333	-233.33	83.333	-230.5594	-2.770555	0.01187398
100	-50	96.666	-86.666	96.666	-83.89577	-2.770222	0.03196435
120	130	103.333	-19.999	103.333	-17.22544	-2.773555	0.13868471
140	270	120	123.333	120	130	-6.667	-0.0540569
160	370	140	263.333	140	270	-6.667	-0.0253177
180	430	159.999	363.333	159.999	369.996	-6.662999	-0.0183385
200	450	180	423.333	180	430	-6.667	-0.0157488
220	430	200	443.333	200	450	-6.667	-0.0150383
240	370	220	423.333	220	430	-6.667	-0.0157488
260	270	239.999	363.333	239.999	370.004	-6.670999	-0.0183605
280	130	260	263.333	260	270	-6.667	-0.0253177
300	-50	280	123.333	280	130	-6.667	-0.0540569
320	-270	300	-56.666	300	-50	-6.666	0.11763668
340	-530	320	-276.666	320	-270	-6.666	0.02409403
360	-830	340	-536.666	340	-530	-6.666	0.01242113
380	-1170	361.666	-865	361.666	-856.7947	-8.205222	0.00948581
400	-1550	400	-1550	400	-1550	0	0

Cuadro 6.1. Error en las coordenadas de la clase nPath.

TESIS CON
FALLA DE ORIGEN

Prueba 4. Desempeño en la Simulación de la Física.

La prueba que a continuación se muestra, tiene como intención calificar de manera cualitativa el desempeño, así como el realismo, de la simulación de las leyes físicas. En particular, esta prueba se enfoca a las leyes de la gravedad y de la inercia. Ésta consiste en una observación detallada del comportamiento de los cuerpos en situaciones que permitan corroborar la apariencia de correcta simulación de las leyes físicas mencionadas.

En esta primer secuencia se observa la simulación de la gravedad en una situación muy obvia. Cabe remarcar que el objeto no solo cae, sino que se acelera conforme sigue cayendo, por lo que en general, se puede decir que cumple con la simulación.

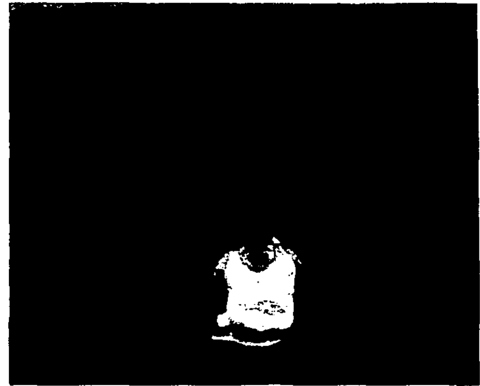


Figura 6.11. Secuencia de imágenes. Simulación de la gravedad.

T.. .. CON
FALLA DE ORIGEN

En esta otra situación, se busca probar la apariencia de realismo en la simulación de la inercia. En la secuencia de imágenes se puede observar que el personaje al tocar la pirámide continua subiendo por un breve lapso de tiempo, aún cuando el personaje deje de caminar. Esto se debe a la velocidad que el personaje tiene en el momento de chocar con la pirámide, un comportamiento resultado de la simulación de inercia. Por lo que también se considera como un desempeño satisfactorio.

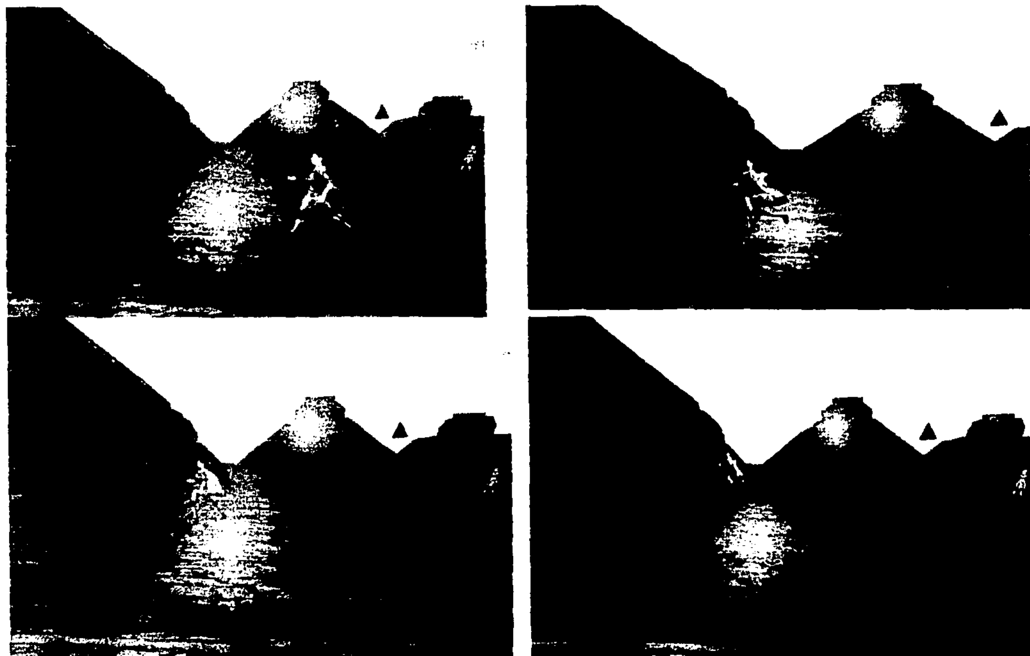


Figura 6.12. Secuencia de imágenes. Simulación de la inercia.

TESIS CON
FALLA DE ORIGEN

En esta última secuencia se puede observar otra simulación física que se logra correctamente. La fuerza de reacción que el suelo aplica al personaje cuando éste choca con él. Se puede decir que esta simulación también es realista, por lo que se puede calificar como un desempeño satisfactorio.

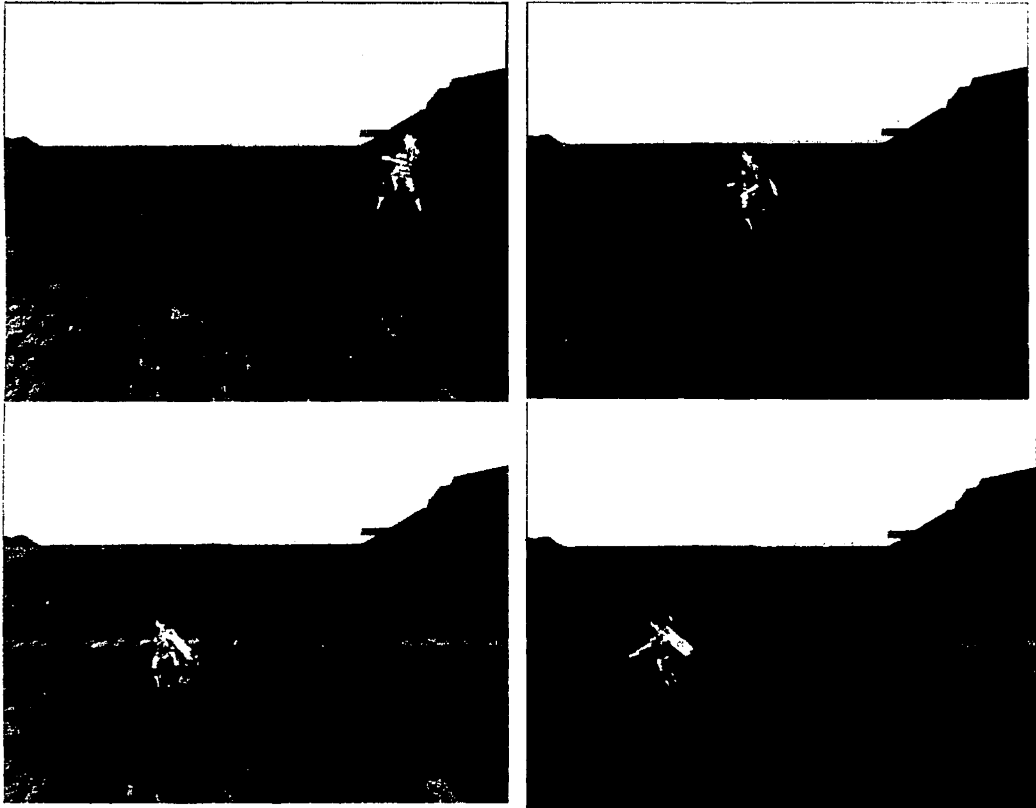


Figura 6.13. Secuencia de imágenes. Simulación de las fuerzas de reacción en estructuras.

TESIS CON
FALLA DE ORIGEN

Prueba 5. Desempeño de la Ejecución de Script activada por Colisiones.

En esta sencilla prueba, se desea corroborar la correcta ejecución de un *script* cuando el personaje controlado por el usuario entra en contacto con un personaje controlado por el sistema. La prueba simplemente consiste en una verificación visual del comportamiento del personaje. Sin embargo, que esta función del API se pruebe de manera tan sencilla, no quiere decir que se trate de una función intrascendente. Todo lo contrario, se trata de una de las funcionalidades que darán mas flexibilidad al desarrollador. De ahí la importancia de su prueba. En la secuencia de imágenes se puede notar como el personaje realiza la animación de tiro de arco y después sigue un camino, tal como el script que ejecuta lo indicaba. Por lo que se considera que el desempeño de esta función es adecuada.

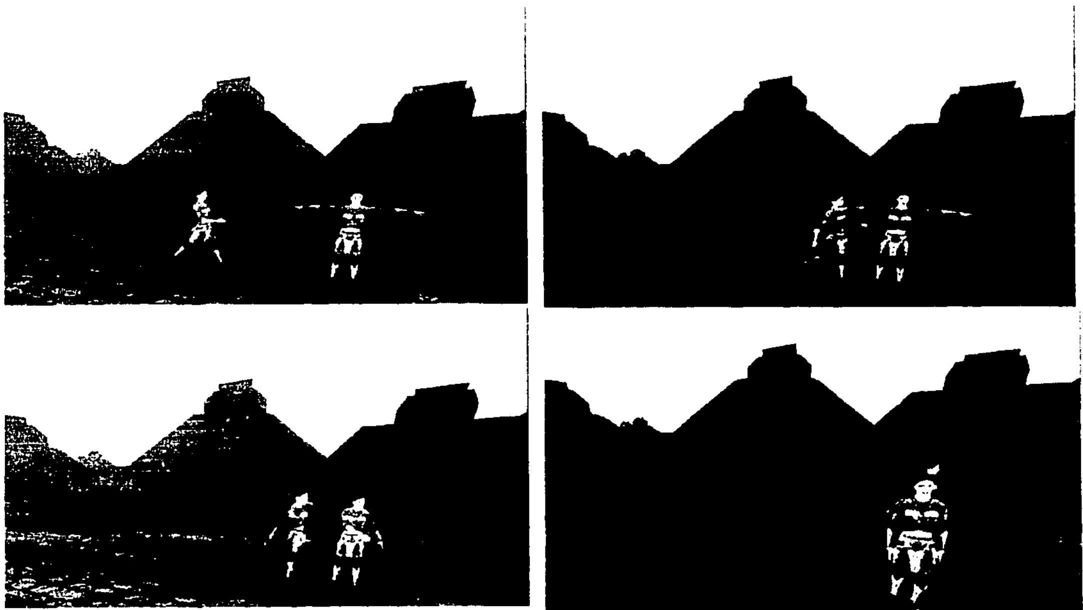


Figura 6.14. Secuencia de imágenes. Ejecución de un script activado por colisión.

TESIS CON
FALLA DE ORIGEN

Conclusión.

Después de realizar el proceso de ingeniería de *software* orientado a objetos (OO), de haber implementado las clases que se reconocieron en el proceso de análisis y se precisaron en el diseño orientado a objetos y de comparar los componentes de programa mas relevantes frente los requerimientos que se tenían, se puede afirmar que el objetivo de este trabajo se ha cumplido. Se desarrolló un API que facilita la creación de sistemas que permiten al usuario interactuar con la historia que el desarrollador concibió, o al menos definió, si se trata de un sistema automatizado de generación de historias. El API permite que esa historia se implementa fácilmente en el sistema, mediante una característica trascendental de éste, la ejecución de *scripts* que pueden controlar cualquier objeto dentro de la historia.

Las pruebas realizadas muestran que los objetos diseñados cumplen con sus responsabilidades de manera adecuada. La creación de una pequeña aplicación para realizar estás pruebas muestra que es factible utilizar esas clases y sus relaciones para la creación de aplicaciones funcionales. En realidad, la última prueba que este proyecto tendría que pasar para decir que cumplió relativamente bien con su objetivo, es la de su utilización en el desarrollo de un sistema real de historias interactivas.

La aplicación de un proceso de ingeniería de *software* orientado a objetos, además, ayudó a desarrollar el API de manera mas formal, en especial tratándose de un campo tan nuevo y con metodologías de diseño poco probadas. El rigor del análisis OO, al menos,

permitió el reconocer clases que de otra forma posiblemente no se habrían creado, en especial, si se pondera la poca experiencia que se tenía en el campo.

En cuanto a la base teórica de las historias interactivas, mediante lo expuesto en los capítulos 1 y 3, se puede notar que todavía no hay conceptos mayoritariamente aceptados acerca de que es un sistema de historias interactivas. O pero aún, no hay un consenso acerca de la necesidad de crear tales sistemas. Lo mismo se puede decir de la metodología de diseño aplicable a este tipo de sistemas. Los pasos, las capas y las estructuras que éstas presentan todavía son muy difusas. Aunque a favor del área, se puede afirmar que últimamente han existido varias iniciativas para resolver estos problemas e indefiniciones. Para hacer una comparación con otras áreas de la ciencia de la computación, el estado de las historias interactivas se parece a aquél en que se encontraba la inteligencia artificial entre la década de los 50s y los 60s, donde el repentino aumento en el poder de las computadoras causó un optimismo que hacia creer que para ellas nada era imposible. Es posible que en algún punto del desarrollo de las historias interactivas alguien se pregunte si en realidad será factible la creación de un sistema que pueda generar historias realmente interesantes, o esa labor tendrá que continuar en manos del desarrollador humano. Como una de tantas labores que los desarrolladores de IA de esa época optimista pensaron que rápidamente pasarían a la lista de lo que una computadora puede hacer.

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 de historias interactivas; en realidad, ese nunca fue el propósito de éste. El enfoque que este trabajo trata de darle al desarrollo de historias interactivas es uno que discretiza y reduce el espacio del problema. Esto al formar una base mediante la cual se pueden luego probar ideas mas innovadoras y atrevidas. Al fin de cuentas, es mejor avanzar paso a paso; y la decisión de desarrollar un API en el que se basen otros proyectos es, sin lugar a dudas, un paso en la dirección correcta.

En esta parte final del trabajo, es necesario admitir que la posibilidad de que este API sea utilizado en otros proyectos diferentes a los que se tenia previsto es reducida. De cualquier manera, este ejercicio permitió la presentación y el análisis de diversos conceptos relacionados con las historias interactivas. Además, lo poco conocido del campo, implicó la aplicación de un proceso de ingeniería de *software* para el desarrollo de un sistema de mediana complejidad, labor pocas veces realizada durante el recorrido de un ingeniero en computación a lo largo de las aulas de su facultad. Además, la creación de este tipo de herramientas a manos de la propia industria del entretenimiento digital es algo que se verá pronto, sin lugar a dudas. Esto se puede prever debido a la creciente necesidad de aplicar los conceptos de las HIC en el área de los videojuegos. Compañías especializadas desarrollan motores de juegos con lo último en tecnología para sus propios juegos, además

estas compañías licencian sus motores para recuperar los costos de investigación y diseño, mientras que benefician a compañías mas pequeñas a las que les sería imposible desarrollar algo tan complicado y que al licenciar el motor obtienen tecnología de punta. No hay nada que indique que no es posible que este patrón se repita en el área de las historias interactivas, cuando ésta área esté mas desarrollada.

La contribución de este trabajo es proponer una base de desarrollo para otros sistemas, 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 del API para HIC.

Generado mediante el programa doxygen.

1 API de Historias Interactivas por Computadora Índice jerárquico

1.1 API de Historias Interactivas por Computadora Jerarquía de la clase

Esta lista de herencias está en orden alfabético

nRoot 177
  nIstEntity 141
    nIstCamera 129
    nIstCursor 136
    nIstNPC 148
    nIstObject 155
    nIstPlayer 158
    nIstStruct 166
  nIstWorld 168
  nPath 172
  nVoiceServer 187
    nSAPIVoiceServer 185

2 API de Historias Interactivas por Computadora Índice de clases

2.1 API de Historias Interactivas por Computadora Lista de componentes

Lista de las clases, estructuras, uniones e interfaces con una breve descripción:

nIstCamera (Clase encargada de controlar el punto de vista desde el que se presenta la escena) 129
nIstCursor (Clase encargada de controlar un pequeño objeto que permite conocer las coordenadas de su posición) 136
nIstEntity (Clase base de la que debe heredar todo objeto que se agrega al mundo de la historia) 141
nIstNPC (Clase responsable de controlar a los personajes de acuerdo con las acciones del usuario) 148
nIstObject (Clase encargada de representar a los objetos que existen en la historia) 155
nIstPlayer (Clase responsable de recibir las entradas del usuario y de controlar a su personaje) 158
nIstStruct (Clase encargada de representar a las estructuras que existen en la historia) 166
nIstWorld (Clase encargada de contener a las entidades de la historia) 168
nPath (Clase que sirve de envolvente para utilizar la funcionalidad de nurbcurve en nebula) . 172
nRoot (Root class for Nebula classes) 177
nSAPIVoiceServer (Clase que implementa el servidor de voz mediante el Speech SDK 5.1 de MS) 185
nVoiceServer (Clase base que sirve para crear distintas implementaciones de servidor de voz) 187

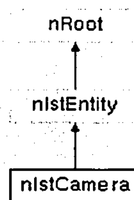
3 API de Historias Interactivas por Computadora Documentación de clases

3.1 Referencia de la Clase nIstCamera

Clase encargada de controlar el punto de vista desde el que se presenta la escena.

```
#include <nistcamera.h>
```

Diagrama de herencias de nIstCamera



TESIS CON FALLA DE ORIGEN

3.1.1 Tipos públicos

enum Style { STATIONARY, LOCKED_CHASE, CHASE, AIM_PATH, LOOK_PATH, FPS, FREE }
Valores de los diferentes estilos de cámara.

Métodos públicos

nIstCamera ()
Método constructor.

virtual **~nIstCamera** ()
Método destructor.

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

const matrix44 & **GetTransform** () const
Regresa la matriz de orientación y posición 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)

Usado en las colisiones para agregar impulso.

void **SetPosition** (float x, float y, float z)
Coloca a la cámara en la posición. (Solo en estilo estacionario).

void **SetStyleStationary** (vector3 targetPos, float angleX, float angleY, float angleZ, float dist)
Activa el estilo estacionario.

void **SetStyleStationary** (n3DNode *target, float angleX, float angleY, float angleZ, float dist)
Activa el estilo estacionario.

void **SetStyleStationary** (vector3 targetPos, vector3 cameraPos)
Activa el estilo estacionario.

void **SetStyleStationary** (n3DNode *target, n3DNode *camer)
Activa el estilo estacionario.

void **SetStyleChase** (n3DNode *target, float height, float prefDist)
Activa el estilo persecución.

void **SetStyleLockedChase** (n3DNode *target, float angleX, float angleY, float angleZ, float dist)
Activa el estilo persecución fija.

void **SetStyleAimPath** (n3DNode *target, const char *orig, const char *dest, float len)
Activa el estilo de apuntar y seguir un camino.

void **SetStyleLookPath** (const char *origPos, const char *destPos, const char *origAng, const char *destAng, float len)
Activa el estilo de orientación predefinida y seguir un camino.

void **SetStyleFPS** (n3DNode *target, float height)
Activa el estilo de primera persona.

void SetStyleFree ()
Activa el estilo de cámara libre.

void **handleInput** (nInputEvent *ie)
Maneja las entradas del usuario. Cuando el estilo de cámara lo requiere.

quaternion **getQ** (void)
Regresa el cuaternión de la matriz de orientación.

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

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

Atributos públicos estáticos

nKernelServer * **kernelServer**
Apuntador a nKernelServer.

nClass * **loacl_cl**
Apuntador a nClass.

3.1.2 Descripción detallada

Clase encargada de controlar el punto de vista desde el que se presenta la escena.

Esta clase mantiene información acerca del punto de vista y la orientación desde la cual se presenta las escenas. Esa información es guardada en una matriz de 4x4. Soporta varios estilos de cámara, para seguir o apuntar a ciertos objetivos. También pude manejar las entradas del usuario para los estilos que así lo requieran.

Creada por Enrique Larios

3.1.3 Documentación de las enumeraciones miembro de la clase

enum nlstCamera::Style

Valores de los diferentes estilos de cámara.

Valores de la enumeración:

STATIONARY
LOCKED_CHASE
CHASE
AIM_PATH
LOOK_PATH
FPS
FREE

3.1.4 Documentación del constructor y destructor

nlstCamera::nlstCamera ()

Método constructor.

nlstCamera::~nlstCamera () [virtual]

Método destructor.

3.1.5 Documentación de las funciones miembro

void nlstCamera::AddImpulse (vector3 *addv*) `[virtual]`

Usado en las colisiones para agregar impulso.

Parámetros:

addv Vector que contiene el impulso que será aplicado al objeto.

Reimplementado de **nIstEntity** (*p.144*).

void nlstCamera::Collide (void) `[virtual]`

Maneja el comportamiento del objeto cuando éste colisiona.

Reimplementado de **nIstEntity** (*p.144*).

quaternion nlstCamera::getQ (void) `[inline]`

Regresa el cuaternión de la matriz de orientación.

El cuaternión contiene toda la información necesaria respecto a la orientación de la cámara.

Devuelve:

Cuaternión con la información de la matriz de orientación.

vector3 nlstCamera::getT (void) `[inline]`

Regresa la posición de la cámara.

Devuelve:

Vector con la información de posición.

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

Regresa la matriz de orientación y posición de la cámara.

Esta matriz normalmente es requerida por la clase escena para conocer el punto desde el cual se observa.

Devuelve:

Matriz de posición y orientación de la cámara.

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

Maneja las entradas del usuario. Cuando el estilo de cámara lo requiere.

Si no es el caso, el evento se deja desatendido. Actualmente solo el estilo libre y el estilo de persecución requieren de entradas del usuario.

Parámetros:

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

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

Fija la matriz de orientación mediante un cuaternión.

Parámetros:

x Componente x del cuaternión.
y Componente y del cuaternión.
z Componente z del cuaternión.
w Componente w del cuaternión.

bool nlstCamera::SaveCmds (nFileServer * *fs*) `[virtual]`

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

Parámetros:

fileServer writes the nCmd object contents out to a file.

Devuelve:

success or failure

Reimplementado de **nlstEntity** *(p.146)*.

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

Coloca a la cámara en la posición. (Solo en estilo estacionario).

Parámetros:

x Fija la coordenada x de la posición de la cámara.
y Fija la coordenada y de la posición de la cámara.
z Fija la coordenada z de la posición de la cámara.

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

Activa el estilo de apuntar y seguir un camino.

Parámetros:

target Objetivo al que apunta la cámara.
orig Denominación del punto de origen del camino.
dest Denominación del punto de destino del camino.
len Tiempo requerido para recorrer el camino.

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

Activa el estilo persecución.

Parámetros:

target Objeto de escena al que persigue la cámara.
height Altura del objeto de escena al que persigue la cámara.
prefDist Distancia a la se mantiene la persecución.

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

Activa el estilo de primera persona.

Parámetros:

target Objeto del cual se toma la vista de primera persona.
height Altura a la que se quiere colocar la cámara.

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

Activa el estilo de cámara libre.

Estilo totalmente controlado por las entradas del usuario mediante el ratón.

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

Activa el estilo persecución fija.

Parámetros:

target Objetivo al que persigue la cámara.
angleX Orientación en eje x de la cámara con respecto al objetivo.
angleY Orientación en eje y de la cámara con respecto al objetivo.
angleZ Orientación en eje z de la cámara con respecto al objetivo.
dist Distancia de la cámara con respecto del objetivo.

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

Activa el estilo de orientación predefinida y seguir un camino.

Parámetros:

origPos Denominación del punto de origen del camino.
destPos Denominación del punto de destino del camino.
origAng Denominación del punto de origen de la ruta de cámara.
destAng Denominación del punto de destino de la ruta de cámara.
len Tiempo requerido para recorrer el camino.

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

Activa el estilo estacionario.

Parámetros:

targetPos Objeto de escena al que apunta la cámara.
cameraPos Objeto de escena con la posición de la cámara.

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

Activa el estilo estacionario.

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 estilo estacionario.

Parámetros:

target Objetivo al que apunta la cámara.
angleX Orientación en eje x de la cámara con respecto al objetivo.
angleY Orientación en eje y de la cámara con respecto al objetivo.
angleZ Orientación en eje z de la cámara con respecto al objetivo.
dist Distancia de la cámara con respecto del objetivo.

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

Activa el estilo estacionario.

Parámetros:

targetPos Posición del objetivo.
angleX Orientación en eje x de la cámara con respecto al objetivo.
angleY Orientación en eje y de la cámara con respecto al objetivo.
angleZ Orientación en eje z de la cámara con respecto al objetivo.
dist Distancia de la cámara con respecto del objetivo.

void nlstCamera::Trigger (float *dt*) `[virtual]`

Actualiza el estado del objeto.

Parámetros:

dt Delta de tiempo desde la vez anterior en la cual se llamó a este método.

Reimplementado de **nIstEntity** (*p.146*).

3.1.6 Documentación de los datos miembro

nKernelServer* nlstCamera::kernelServer `[static]`

Apuntador a nKernelServer.

Reimplementado de **nIstEntity** (*p.148*).

nClass* nlstCamera::loacl_cl `[static]`

Apuntador a nClass.

La documentación para esta clase fue generada a partir de los siguientes archivos:

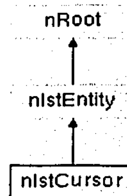
nistcamera.h
nistcamera_cmds.cc
nistcamera_main.cc

3.2 Referencia de la Clase nIstCursor

Clase encargada de controlar un pequeño objeto que permite conocer las coordenadas de su posición.

```
#include <nIstCursor.h>
```

Diagrama de herencias de nIstCursor



3.2.1 Métodos públicos

nIstCursor ()
Método constructor.

virtual **~nIstCursor** ()
Método destructor.

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

void **StartFW** ()
Coloca al cursor en el estado de moverse hacia adelante.

void **StartBW** ()
Coloca al cursor en el estado de moverse hacia atrás.

void **StartUP** ()
Coloca al cursor en el estado de moverse hacia arriba.

void **StartDW** ()
Coloca al cursor en el estado de moverse hacia abajo.

void **StartL** ()
Coloca al cursor en el estado de moverse hacia izquierda.

void **StartR** ()
Coloca al cursor en el estado de moverse hacia derecha.

void **Stop** ()
Detiene al cursor de moverse.

vector3 * **GetPosition** (void)
Regresa la posición del cursor en las coordenadas de la escena.

void **SetN3DNodeTar** (const char *path)
Fija el nodo que contiene la figura del cursor.

const char * **GetN3DNodeName** () const
Regresa la ruta del nodo que contiene la figura del cursor.

void **SetRunSpeed** (float rs)
Fija la velocidad al moverse.

float **GetRunSpeed** (void)
Regresa a velocidad al moverse.

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

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

void **UpdatePosition** (float dt)
Actualiza la posición del objeto.

virtual void **AddImpulse** (vector3 addv)

3.2.2 Atributos públicos estáticos

nKernelServer * **kernelServer**
pointer to nKernelServer

3.2.3 Descripción detallada

Clase encargada de controlar un pequeño objeto que permite conocer las coordenadas de su posición.

Esta clase se puede utilizar con una aplicación de construcción de escenarios, donde se requiere saber las coordenadas de la posición de un objeto.

Creada por Enrique Larios.

3.2.4 Documentación del constructor y destructor

nIstCursor::nIstCursor ()

Método constructor.

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

Método destructor.

3.2.5 Documentación de las funciones miembro

void nIstCursor::AddImpulse (vector3 *addv*) `[virtual]`

En esta clase en particular, este método no hace nada.

Reimplementado de **nIstEntity** *(p.144)*.

void nIstCursor::Collide (void) `[virtual]`

Maneja el comportamiento del objeto cuando éste colisiona.

En esta clase en particular, este método no hace nada, ya que no interesa que el cursor este chocando.

Reimplementado de **nIstEntity** *(p.144)*.

const char * nIstCursor::GetN3DNodeName () const `[inline]`

Regresa la ruta del nodo que contiene la figura del cursor.

Devuelve:

Nombre del nodo que contiene la figura.

vector3 * nIstCursor::GetPosition (void) `[inline]`

Regresa la posición del cursor en las coordenadas de la escena.

Devuelve:

Vector que contiene la posición del cursor.

float nIstCursor::GetRunSpeed (void) `[inline]`

Regresa a velocidad al moverse.

Devuelve:

Velocidad al moverse en m/s.

bool nIstCursor::SaveCmds (nFileServer * *fs*) `[virtual]`

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

Parámetros:

fileServer writes the nCmd object contents out to a file.

Devuelve:

success or failure

Reimplementado de **nIstEntity** *(p.146)*.

void nIstCursor::SetN3DNodeTar (const char * *path*) `[inline]`

Fija el nodo que contiene la figura del cursor.

Parámetros:

path Ruta en la jerarquía donde está el nodo que contiene al modelo.

void nIstCursor::SetRunSpeed (float *rs*) `[inline]`

Fija la velocidad al moverse.

Parámetros:

rs Velocidad al moverse en m/s.

void nIstCursor::StartBW () `[inline]`

Coloca al cursor en el estado de moverse hacia atrás.

void nIstCursor::StartDW () `[inline]`

Coloca al cursor en el estado de moverse hacia abajo.

void nIstCursor::StartFW () `[inline]`

Coloca al cursor en el estado de moverse hacia adelante.

void nIstCursor::StartL () `[inline]`

Coloca al cursor en el estado de moverse hacia izquierda.

void nlstCursor::StartR () `[inline]`

Coloca al cursor en el estado de moverse hacia derecha.

void nlstCursor::StartUP () `[inline]`

Coloca al cursor en el estado de moverse hacia arriba.

void nlstCursor::Stop () `[inline]`

Detiene al cursor de moverse.

void nlstCursor::Trigger (float *dt*) `[virtual]`

Actualiza el estado del objeto.

En esta clase en particular, este método mantiene el movimiento del cursor.

Parámetros:

dt Delta de tiempo desde la vez anterior en la cual se llamó a este método.

Reimplementado de **nIstEntity** (*p.146*).

void nlstCursor::UpdatePosition (float *dt*) `[virtual]`

Actualiza la posición del objeto.

En esta clase en particular, este método no hace nada.

Parámetros:

dt Delta de tiempo desde la vez anterior en la cual se llamó a este método.

Reimplementado de **nIstEntity** (*p.147*).

3.2.6 Documentación de los datos miembro

nKernelServer* nlstCursor::kernelServer `[static]`

pointer to nKernelServer

Reimplementado de **nIstEntity** (*p.148*).

La documentación para esta clase fue generada a partir de los siguientes archivos:

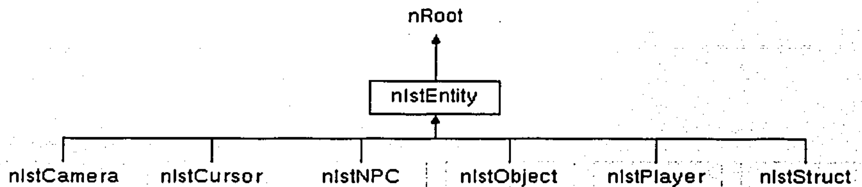
nIstCursor.h
nIstCursor_cmds.cc
nIstCursor_main.cc

3.3 Referencia de la Clase nIstEntity

Clase base de la que debe heredar todo objeto que se agrega al mundo de la historia.

`#include <nIstEntity.h>`

Diagrama de herencias de nIstEntity



3.3.1 Métodos públicos

nIstEntity ()
Método constructor.

virtual ~**nIstEntity** ()
Método destructor.

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

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

const char * **GetName** ()
Método que regresa el nombre del objeto.

nIstWorld * GetIstWorld ()
Regresa un apuntador al objeto mundo que contiene a este objeto.

void **SetMass** (float m)
Fija la masa del objeto.

float **GetMass** (void)
Regresa la masa del objeto.

void **SetCollideClass** (const char *coll_class)
Fija la clase de colisión a la que pertenece este objeto.

const char * **GetCollideClass** () const
Regresa la clase de colisión a la que pertenece este objeto.

void **SetCollideShape** (const char *coll_shapeName, const char *coll_shapeFile)
Fija la forma con la que se representa este objeto para las colisiones.

const char * **GetCollideShape** () const
Regresa el nombre la forma que representa este objeto para las colisiones.

const char * **GetCollideShapeFile** () const
Regresa el nombre del archivo donde se guarda la forma de este objeto para las colisiones.

bool GetDynamicColl (void)
void **SetDynamicColl** (bool b)
virtual void **Collide** ()
Maneja el comportamiento del objeto cuando éste colisiona.

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

virtual void **AddImpulse** (vector3 addv)
Usado en las colisiones para aplicar un impulso al objeto.

virtual void **AddForce** (vector3 addf)
Usado en las colisiones para aplicar una fuerza al objeto.

virtual void **UpdatePosition** (float dt)
Actualiza la posición del objeto.

vector3 **GetForceVector** (void)
Regresa el vector de velocidad del objeto.

vector3 **GetVelVector** (void)
Regresa el vector de fuerza del objeto.

3.3.2 Atributos públicos

bool DynamicColl

3.3.3 Atributos públicos estáticos

nKernelServer * **kernelServer**
Apuntador a nKernelServer.

3.3.4 Atributos protegidos

nRef< nIstWorld > refWorld
nAutoRef< nChannelServer > **refChannelServer**
nString collideClass

nString collideShape
nString collideShapeFile
nCollideObject * **collideObject**
Referencia a collide object.

vector3 velVector
vector3 forVector
vector3 accForVector
vector3 aclVector
float **mass**
float invMass

3.3.5 Descripción detallada

Clase base de la que debe heredar todo objeto que se agrega al mundo de la historia.

Mediante la funcionalidad base y los métodos de esta clase, se realizan todas las interacciones comunes entre los objetos, como la colisión y la actualización.

Creada por Enrique Larios.

3.3.6 Documentación del constructor y destructor

nIstEntity::nIstEntity ()

Método constructor.

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

Método destructor.

3.3.7 Documentación de las funciones miembro

void nIstEntity::AddForce (vector3 *addf*) `[virtual]`

Usado en las colisiones para aplicar una fuerza al objeto.

Este método normalmente es llamado desde los métodos de colisión de los objetos con que colisiona.

Parámetros:

addf Vector que contiene la fuerza que será aplicada al objeto.

void nlstEntity::AddImpulse (vector3 *addv*) `[virtual]`

Usado en las colisiones para aplicar un impulso al objeto.

Este método normalmente es llamado desde los métodos de colisión de los objetos con que colisiona.

Parámetros:

addv Vector que contiene el impulso que será aplicado al objeto.

Reimplementado en **nlstCamera** (*p.132*), **nlstCursor** (*p.138*), **nlstNPC** (*p.151*), **nlstObject** (*p.157*), **nlstPlayer** (*p.161*), y **nlstStruct** (*p.167*).

void nlstEntity::Collide (void) `[virtual]`

Maneja el comportamiento del objeto cuando éste colisiona.

Reimplementado en **nlstCamera** (*p.132*), **nlstCursor** (*p.138*), **nlstNPC** (*p.152*), **nlstObject** (*p.157*), **nlstPlayer** (*p.162*), y **nlstStruct** (*p.167*).

const char * nlstEntity::GetCollideClass () const

Regresa la clase de colisión a la que pertenece este objeto.

Devuelve:

El objeto clase de colisión al que este objeto pertenece.

const char * nlstEntity::GetCollideShape () const

Regresa el nombre la forma que representa este objeto para las colisiones.

Devuelve:

El objeto forma de colisión al que este objeto pertenece.

const char * nlstEntity::GetCollideShapeFile () const

Regresa el nombre del archivo donde se guarda la forma de este objeto para las colisiones.

Devuelve:

El nombre del archivo que contiene la forma de colisión de este objeto.

bool nlstEntity::GetDynamicColl (void) `[inline]`

vector3 nlstEntity::GetForceVector (void) `[inline]`

Regresa el vector de velocidad del objeto.

Devuelve:
El vector de fuerza resultante.

nlstWorld * nlstEntity::GetlstWorld () [inline]

Regresa un apuntador al objeto mundo que contiene a este objeto.

Devuelve:
El objeto mundo que contiene a este objeto y que asigna en el método inicializar.

float nlstEntity::GetMass (void) [inline]

Regresa la masa del objeto.

Devuelve:
Masa del objeto.

const char * nlstEntity::GetName (void) [inline]

Método que regresa el nombre del objeto.

Devuelve:
El nombre del objeto en la jerarquía de Nebula.

Reimplementado de **nRoot** (*p.181*).

vector3 nlstEntity::GetVelVector (void) [inline]

Regresa el vector de fuerza del objeto.

Devuelve:
El vector de velocidad resultante.

regresa el vector de velocidad resultante

void nlstEntity::Initialize (void) [virtual]

Inicializa diversos valores de la clase.

Encuentra el objeto padre **nIstWorld** en algún lugar arriba en la jerarquía y se liga. Además crea su nodo de scriptlets.

Reimplementado de **nRoot** (*p.182*).

Reimplementado en **nIstNPC** (*p.153*).

bool nIstEntity::SaveCmds (nFileServer * *fs*) `[virtual]`

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

Parámetros:

fileServer writes the nCmd object contents out to a file.

Devuelve:

success or failure

Reimplementado de **nRoot** (*p.183*).

Reimplementado en **nIstCamera** (*p.133*), **nIstCursor** (*p.139*), **nIstNPC** (*p.153*), **nIstObject** (*p.157*), **nIstPlayer** (*p.163*), y **nIstStruct** (*p.167*).

void nIstEntity::SetCollideClass (const char * *coll_class*)

Fija la clase de colisión a la que pertenece este objeto.

Es importante que esta clase ya antes haya sido dada de alta en el servidor de colisiones.

Parámetros:

coll_class Nombre de la clase de colisión.

void nIstEntity::SetCollideShape (const char * *coll_shapeName*, const char * *coll_shapeFile*)

Fija la forma con la que se representa este objeto para las colisiones.

Parámetros:

coll_shapeName Nombre de la forma de colisión.
coll_shapeFile Nombre del archivo donde se almacena la forma de colisión.

void nIstEntity::SetDynamicColl (bool *b*) `[inline]`

void nIstEntity::SetMass (float *m*) `[inline]`

Fija la masa del objeto.

Masa que es utilizada en la simulación de la física.

Parámetros:

m Masa del objeto.

void nIstEntity::Trigger (float *dt*) `[virtual]`

Actualiza el estado del objeto.

Parámetros:

dt Delta de tiempo desde la vez anterior en la cual se llamó a este método.

Reimplementado en **nIstCamera** (*p.135*), **nIstCursor** (*p.140*), **nIstNPC** (*p.155*), **nIstObject** (*p.158*), **nIstPlayer** (*p.165*), y **nIstStruct** (*p.168*).

void nIstEntity::UpdatePosition (float *dt*) `[virtual]`

Actualiza la posición del objeto.

Parámetros:

dt Delta de tiempo desde la vez anterior en la cual se llamó a este método.

Reimplementado en **nIstCursor** (*p.140*), **nIstNPC** (*p.155*), y **nIstPlayer** (*p.165*).

3.3.8 Documentación de los datos miembro

vector3 nIstEntity::accForVector `[protected]`

vector3 nIstEntity::aclVector `[protected]`

nString nIstEntity::collideClass `[protected]`

nCollideObject* nIstEntity::collideObject `[protected]`

Referencia a collide object.

nString nIstEntity::collideShape `[protected]`

nString nIstEntity::collideShapeFile `[protected]`

bool nIstEntity::DynamicColl

vector3 nIstEntity::forVector `[protected]`

float nIstEntity::invMass `[protected]`

nKernelServer* nIstEntity::kernelServer `[static]`

Apuntador a nKernelServer.

Reimplementado en **nIstCamera** (*p.135*), **nIstCursor** (*p.140*), **nIstNPC** (*p.155*), **nIstObject** (*p.158*), **nIstPlayer** (*p.165*), y **nIstStruct** (*p.168*).

float nIstEntity::mass `[protected]`

nAutoRef<nChannelServer> nIstEntity::refChannelServer `[protected]`

nRef<nIstWorld> nIstEntity::refWorld `[protected]`

vector3 nIstEntity::velVector `[protected]`

La documentación para esta clase fue generada a partir de los siguientes archivos:

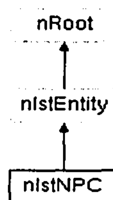
- **nIstEntity.h**
- **nIstEntity_cmds.cc**
- **nIstEntity_main.cc**

3.4 Referencia de la Clase nIstNPC

Clase responsable de controlar a los personajes de acuerdo con las acciones del usuario.

```
#include <nIstNPC.h>
```

Diagrama de herencias de nIstNPC



3.4.1 Métodos públicos

nIstNPC ()

Método constructor.

virtual **~nIstNPC** ()

Método destructor.

virtual void **Initialize** ()

Inicializa diversos valores de la clase.

virtual bool **SaveCmds** (nFileServer *fileServer)

Método encargado de la persistencia, salva en 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 objeto.

void **AddImpulse** (vector3 addv)

Usado en las colisiones para aplicar un impulso al objeto.

void **AnimWalk** ()

Lama a la animación "caminar" del modelo que representa al personaje.

void AnimStrut ()

Lama a la animación "girar" del modelo que representa al personaje.

void **AnimIdle** ()

Lama a la animación "parado" del modelo que representa al personaje.

void **SetRotSpeed** (float rs)

Fija la velocidad de Giro.

void **SetRunSpeed** (float rs)

Fija la velocidad al correr.

float GetRotSpeed ()

Regresa la velocidad de Giro.

float GetRunSpeed ()

Regresa la velocidad al correr.

void **SetN3DNodeTar** (const char *path)
Fija el nodo que contiene el modelo del personaje.

const char * **GetN3DNodeTar** () const
Regresa el nombre del nodo que contiene el modelo del personaje.

void **SetNCalModTar** (const char *path)
Fija el modelo que representa al personaje.

const char * **GetNCalModTar** () const
Regresa el nombre del modelo que representa al personaje.

void **SetCollServ** (const char *path)
Fija el servidor de colisiones.

const char * **GetCollServ** () const
Regresa el servidor de colisiones.

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

void DisableEventColl ()
Deshabilita la ejecución de scripts con eventos.

nIstEntity * GetLastCollEntity (void)
Regresa la última entidad con la que colisionó.

bool **SetStatePathSeeking** (const char *orig, const char *dest, double duration)
Coloca al personaje en el estado seguir camino.

3.4.2 Atributos públicos estáticos

nKernelServer * **kernelServer**
Apuntador a nKernelServer.

3.4.3 Métodos protegidos

bool **BeginPathSeeking** (const char *orig, const char *dest, double duration)
vector3 * **SeekPath** (float dt)

.

3.4.4 Descripción detallada

Clase responsable de controlar a los personajes de acuerdo con las acciones del usuario.

Mediante las acciones del usuario, representadas en su personaje, esta clase responde a las acciones del usuario mediante scripts. Controla y actualiza la animación y la posición del modelo que representa al personaje. Además puede seguir caminos.

Creada por Enrique Larios.

3.4.5 Documentación del constructor y destructor

nIstNPC::nIstNPC ()

Método constructor.

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

Método destructor.

3.4.6 Documentación de las funciones miembro

void nIstNPC::AddImpulse (vector3 *addv*) `[inline, virtual]`

Usado en las colisiones para aplicar un impulso al objeto.

Este método normalmente es llamado desde los métodos de colisión de los objetos con que colisiona.

Parámetros:

addv Vector que contiene el impulso que será aplicado al objeto.

Reimplementado de **nIstEntity** (*p.144*).

void nIstNPC::AnimIdle () `[inline]`

Lama a la animación "parado" del modelo que representa al personaje.

void nIstNPC::AnimStrut () `[inline]`

Lama a la animación "girar" del modelo que representa al personaje.

void nIstNPC::AnimWalk () `[inline]`

Lama a la animación "caminar" del modelo que representa al personaje.

bool nlstNPC::BeginPathSeeking (const char * *orig*, const char * *dest*, double *duration*) `[protected]`

void nlstNPC::Collide (void) `[virtual]`

Maneja el comportamiento del objeto cuando éste colisiona.

En esta clase en particular, este método se encarga de llamar a la ejecución de scripts. Si este evento está habilitado.

Reimplementado de **nlstEntity** (*p.144*).

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

Deshabilita la ejecución de scripts con eventos.

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

Habilita la ejecución de scripts con eventos.

Parámetros:

collScriptFile Nombre del archivo que contiene el script a ejecutar.

const char * nlstNPC::GetCollServ () const `[inline]`

Regresa el servidor de colisiones.

Devuelve:

Ruta en la jerarquía donde está el servidor de colisiones.

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

Regresa la última entidad con la que colisionó.

Devuelve:

Último objeto con él que colisionó.

const char * nlstNPC::GetN3DNodeTar () const `[inline]`

Regresa el nombre del nodo que contiene el modelo del personaje.

Devuelve:

Nombre del nodo que contiene al modelo.

const char * nIstNPC::GetNCalModTar () const `[inline]`

Regresa el nombre del modelo que representa al personaje.

Devuelve:
Nombre del modelo del personaje.

float nIstNPC::GetRotSpeed () `[inline]`

Regresa la velocidad de Giro.

Devuelve:
Velocidad de giro en grados/s.

float nIstNPC::GetRunSpeed (void) `[inline]`

Regresa la velocidad al correr.

Devuelve:
Velocidad al correr en m/s.

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

Inicializa diversos valores de la clase.

En esta clase en particular, este método carga todos los scripts de comportamiento.

Reimplementado de **nIstEntity** (*p.145*).

bool nIstNPC::SaveCmds (nFileServer * *fs*) `[virtual]`

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

Parámetros:
fileServer writes the nCmd object contents out to a file.

Devuelve:
success or failure

Reimplementado de **nIstEntity** (*p.146*).

vector3 * nIstNPC::SeekPath (float *dt*) `[protected]`

void nIstNPC::SetCollServ (const char * *path*) `[inline]`

Fija el servidor de colisiones.

Parámetros:

path Ruta en la jerarquía donde está el servidor de colisiones.

void nlstNPC::SetN3DNodeTar (const char * *path*) `[inline]`

Fija el nodo que contiene el modelo del personaje.

Parámetros:

path Ruta en la jerarquía donde está el nodo que contiene al modelo.

void nlstNPC::SetNCalModTar (const char * *path*) `[inline]`

Fija el modelo que representa al personaje.

Parámetros:

path Ruta en la jerarquía donde está el modelo.

void nlstNPC::SetRotSpeed (float *rs*) `[inline]`

Fija la velocidad de Giro.

Parámetros:

rs Velocidad de giro en grados/s.

void nlstNPC::SetRunSpeed (float *rs*) `[inline]`

Fija la velocidad al correr.

Parámetros:

ms Velocidad que puede alcanzar el personaje al correr en m/s.

bool nlstNPC::SetStatePathSeeking (const char * *orig*, const char * *dest*, double *duration*)

Coloca al personaje en el estado seguir camino.

Parámetros:

orig Denominación del punto de origen del camino
dest Denominación del punto de destino del camino
durarion Tiempo en recorrer el camino

Devuelve:

Resultado. En especial si encuentra un camino que lo lleve de origen a destino.

void nIstNPC::Trigger (float *dt*) `[virtual]`

Actualiza el estado del objeto.

En esta clase en particular, este método mantiene las animaciones y el movimiento del personaje. Por ejemplo, cuando el personaje sigue un camino.

Parámetros:

dt Delta de tiempo desde la vez anterior en la cual se llamó a este método.

Reimplementado de **nIstEntity** (*p.146*).

void nIstNPC::UpdatePosition (float *dt*) `[virtual]`

Actualiza la posición del objeto.

Parámetros:

dt Delta de tiempo desde la vez anterior en la cual se llamó a este método.

Reimplementado de **nIstEntity** (*p.147*).

3.4.7 Documentación de los datos miembro

nKernelServer* nIstNPC::kernelServer `[static]`

Apuntador a nKernelServer.

Reimplementado de **nIstEntity** (*p.148*).

La documentación para esta clase fue generada a partir de los siguientes archivos:

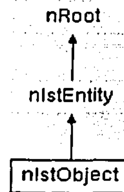
nIstNPC.h
nIstNPC_cmds.cc
nIstNPC_main.cc

3.5 Referencia de la Clase nIstObject

Clase encargada de representar a los objetos que existen en la historia.

`#include <nIstObject.h>`

Diagrama de herencias de nIstObject



3.5.1 Métodos públicos

nIstObject ()
Método constructor.

virtual **~nIstObject** ()
Método destructor.

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

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

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

void **AddImpulse** (vector3 addv)
Usado en las colisiones para aplicar un impulso al objeto.

void **CreateModel** (const char *meshfile, const char *texfile, bool lightEn, float x, float y, float z)
Método optimizado para crear la representación gráfica del objeto.

3.5.2 Atributos públicos estáticos

nKernelServer * **kernelServer**
Apuntador a nKernelServer.

3.5.3 Descripción detallada

Clase encargada de representar a los objetos que existen en la historia.

Esta clase puede ver como un contenedor que mantiene una referencia a la representación gráfica de un objeto. Además de presentar un método con la funcionalidad de cargar de manera sencilla la representación gráfica del objeto.

3.5.4 Documentación del constructor y destructor

nlstObject::nlstObject ()

Método constructor.

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

Método destructor.

3.5.5 Documentación de las funciones miembro

void nlstObject::AddImpulse (vector3 *addv*) `[virtual]`

Usado en las colisiones para aplicar un impulso al objeto.

Este método no realiza nada en esta clase.

Parámetros:

addv Vector que contiene el impulso que será aplicado al objeto.

Reimplementado de **nlstEntity** *(p.144)*.

void nlstObject::Collide (void) `[virtual]`

Maneja el comportamiento del objeto cuando éste colisiona.

Reimplementado de **nlstEntity** *(p.144)*.

void nlstObject::CreateModel (const char * *meshfile*, const char * *texfile*, bool *lightEn*, float *x*, float *y*, float *z*)

Método optimizado para crear la representación gráfica del objeto.

Parámetros:

meshfile Nombre del archivo que contiene la geometría de la representación gráfica.
texfile Nombre del archivo que contiene la textura de la representación gráfica.
lightEn Habilita la iluminación en el sombreado.
x Posición en x del objeto.
y Posición en y del objeto.
z Posición en z del objeto.

bool nlstObject::SaveCmds (nFileServer * *fs*) `[virtual]`

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

Parámetros:
fileServer writes the nCmd object contents out to a file.

Devuelve:
success or failure

Reimplementado de **nIstEntity** *(p.146)*.

void nIstObject::Trigger (float *dt*) `[virtual]`

Actualiza el estado del objeto.

En esta clase en particular, este método mantiene a la malla y la geometría de colisión en la misma posición.

Parámetros:
dt Delta de tiempo desde la vez anterior en la cual se llamó a este método.

Reimplementado de **nIstEntity** *(p.146)*.

3.5.6 Documentación de los datos miembro

nKernelServer* nIstObject::kernelServer `[static]`

Apuntador a nKernelServer.

Reimplementado de **nIstEntity** *(p.148)*.

La documentación para esta clase fue generada a partir de los siguientes archivos:

nIstObject.h
nIstObject_cmds.cc
nIstObject_main.cc

3.6 Referencia de la Clase nIstPlayer

Clase responsable de recibir las entradas del usuario y de controlar a su personaje.

`#include <nIstPlayer.h>`

Diagrama de herencias de nIstPlayer

3.6.1 Métodos públicos

nIstPlayer ()
Método constructor.

virtual **~nIstPlayer** ()
Método destructor.

virtual bool **SaveCmds** (nFileServer *fileServer)
Método encargado de la persistencia, salva en 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 objeto.

void **AddImpulse** (vector3 addv)
Usado en las colisiones para aplicar un impulso al objeto.

void **StartFW** ()
Coloca al personaje en el estado de caminar hacia adelante.

void **StartBW** ()
Coloca al personaje en el estado de caminar hacia atrás.

void **StartRL** ()
Coloca al personaje en el estado de girar a la izquierda.

void **StartRR** ()
Coloca al personaje en el estado de girar a la derecha.

void **StopFW** ()
Detiene al personaje de caminar hacia adelante.

void **StopBW** ()
Detiene al personaje de caminar hacia atrás.

void **StopRL** ()
Detiene al personaje de girar a la izquierda.

void **StopRR** ()
Detiene al personaje de girar a la derecha.

void **ActShoot** ()
Llama a la animación de acción.

void **SetRotSpeed** (float rs)
Fija la velocidad de Giro.

void **SetMaxRunSpeed** (float ms)
Fija la máxima velocidad que se alcanza al correr.

void **SetRunAccel** (float ac)
Fija la aceleración al correr.

float GetRotSpeed ()
Regresa la velocidad de Giro.

float GetMaxRunSpeed ()
Regresa la máxima velocidad que se alcanza al correr.

float GetRunAccel ()
Regresa a aceleración al correr.

void **SetN3DNodeTar** (const char *path)
Fija el nodo que contiene el modelo del personaje.

const char * **GetN3DNodeTar** () const
Regresa el nombre del nodo que contiene el modelo del personaje.

void **SetNCalModTar** (const char *path)
Fija el modelo que representa al personaje.

const char * **GetNCalModTar** () const
Regresa el nombre del modelo que representa al personaje.

void **SetCollServ** (const char *path)
Fija el servidor de colisiones.

const char * **GetCollServ** () const

Regresa el servidor de colisiones.

3.6.2 Atributos públicos estáticos

nKernelServer * **kernelServer**

Apuntador a nKernelServer.

3.6.3 Descripción detallada

Clase responsable de recibir las entradas del usuario y de controlar a su personaje.

Mediante las entradas el usuario, que se utilizan en el servidor de entrada para llamar a los métodos iniciadores de esta clase, esta clase controla y actualiza la animación y la posición del modelo que representa al usuario.

Creada por Enrique Larios.

3.6.4 Documentación del constructor y destructor

nlstPlayer::nlstPlayer ()

Método constructor.

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

Método destructor.

3.6.5 Documentación de las funciones miembro

void nlstPlayer::ActShoot () `[inline]`

Llama a la animación de acción.

void nlstPlayer::AddImpulse (vector3 *addv*) `[inline, virtual]`

Usado en las colisiones para aplicar un impulso al objeto.

Este método normalmente es llamado desde los métodos de colisión de los objetos con que choca.

Parámetros:
addv Vector que contiene el impulso que será aplicado al objeto.

Reimplementado de **nIstEntity** *(p.144)*.

void nIstPlayer::Collide (void) [virtual]

Maneja el comportamiento del objeto cuando éste colisiona.

En esta clase en particular, este método se encarga de la física de colisión del personaje. Por ejemplo, que este no atraviese las paredes.

Reimplementado de **nIstEntity** *(p.144)*.

const char * nIstPlayer::GetCollServ () const [inline]

Regresa el servidor de colisiones.

Devuelve:
Ruta en la jerarquía donde está el servidor de colisiones.

float nIstPlayer::GetMaxRunSpeed () [inline]

Regresa la máxima velocidad que se alcanza al correr.

Devuelve:
Máxima velocidad al correr en m/s.

const char * nIstPlayer::GetN3DNodeTar () const [inline]

Regresa el nombre del nodo que contiene el modelo del personaje.

Devuelve:
Nombre del nodo que contiene al modelo.

const char * nIstPlayer::GetNCalModTar () const [inline]

Regresa el nombre del modelo que representa al personaje.

Devuelve:
Nombre del modelo del personaje.

float nIstPlayer::GetRotSpeed () [inline]

Regresa la velocidad de Giro.

Devuelve:

Velocidad de giro en grados/s.

float nlstPlayer::GetRunAccel () [inline]

Regresa a aceleración al correr.

Devuelve:

Aceleración al correr en m/s^2.

bool nlstPlayer::SaveCmds (nFileServer * *fs*) [virtual]

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

Parámetros:

fileServer writes the nCmd object contents out to a file.

Devuelve:

success or failure

Reimplementado de **nIstEntity** *(p.146)*.

void nlstPlayer::SetCollServ (const char * *path*) [inline]

Fija el servidor de colisiones.

Parámetros:

path Ruta en la jerarquía donde está el servidor de colisiones.

void nlstPlayer::SetMaxRunSpeed (float *ms*) [inline]

Fija la máxima velocidad que se alcanza al correr.

Parámetros:

ms Máxima velocidad que puede alcanzar el personaje al correr en m/s.

void nlstPlayer::SetN3DNodeTar (const char * *path*) [inline]

Fija el nodo que contiene el modelo del personaje.

Parámetros:

path Ruta en la jerarquía donde está 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 jerarquía donde está el modelo.

void nlstPlayer::SetRotSpeed (float *rs*) `[inline]`

Fija la velocidad de Giro.

Parámetros:

rs Velocidad de giro en grados/s.

void nlstPlayer::SetRunAccel (float *ac*) `[inline]`

Fija la aceleración al correr.

Parámetros:

ac Aceleración del personaje al correr en m/s^2.

void nlstPlayer::StartBW () `[inline]`

Coloca al personaje en el estado de caminar hacia atrás.

void nlstPlayer::StartFW () `[inline]`

Coloca al personaje en el estado de caminar hacia adelante.

void nlstPlayer::StartRL () `[inline]`

Coloca al personaje en el estado de girar a la izquierda.

void nlstPlayer::StartRR () `[inline]`

Coloca al personaje en el estado de girar a la derecha.

void nlstPlayer::StopBW () `[inline]`

Detiene al personaje de caminar hacia atrás.

void nlstPlayer::StopFW () `[inline]`

Detiene al personaje de caminar hacia adelante.

void nIstPlayer::StopRL () `[inline]`

Detiene al personaje de girar a la izquierda.

void nIstPlayer::StopRR () `[inline]`

Detiene al personaje de girar a la derecha.

void nIstPlayer::Trigger (float *dt*) `[virtual]`

Actualiza el estado del objeto.

En esta clase en particular, este método mantiene las animaciones y el movimiento del personaje.

Parámetros:

dt Delta de tiempo desde la vez anterior en la cual se llamó a este método.

Reimplementado de **nIstEntity** (*p.146*).

void nIstPlayer::UpdatePosition (float *dt*) `[virtual]`

Actualiza la posición del objeto.

La posición se obtiene de integrar el vector de aceleración y luego del de velocidad.

Parámetros:

dt Delta de tiempo desde la vez anterior en la cual se llamó a este método.

Reimplementado de **nIstEntity** (*p.147*).

3.6.6 Documentación de los datos miembro

nKernelServer* nIstPlayer::kernelServer `[static]`

Apuntador a nKernelServer.

Reimplementado de **nIstEntity** (*p.148*).

La documentación para esta clase fue generada a partir de los siguientes archivos:

- **nIstPlayer.h**
- **nIstPlayer_cmds.cc**
- **nIstPlayer_main.cc**

3.7 Referencia de la Clase nIstStruct

Clase encargada de representar a las estructuras que existen en la historia.

`#include <nIstStruct.h>`

Diagrama de herencias de nIstStruct

3.7.1 Métodos públicos

nIstStruct ()

Método constructor.

virtual **~nIstStruct** ()

Método destructor.

virtual bool **SaveCmds** (nFileServer *fileServer)

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

void **Collide** ()

Maneja el comportamiento del objeto cuando éste colisiona.

void **Trigger** (float dt)

Actualiza el estado del objeto.

void **AddImpulse** (vector3 addv)

Usado en las colisiones para aplicar un impulso al objeto.

3.7.2 Atributos públicos estáticos

nKernelServer * **kernelServer**

Apuntador a nKernelServer.

3.7.3 Descripción detallada

Clase encargada de representar a las estructuras que existen en la historia.

Esta clase puede ver como un contenedor que esta en **nIstWorld** y que contiene una liga a la representación gráfica de la estructura.

3.7.4 Documentación del constructor y destructor

nIstStruct::nIstStruct ()

Método constructor.

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

Método destructor.

3.7.5 Documentación de las funciones miembro

void nIstStruct::AddImpulse (vector3 *addv*) `[virtual]`

Usado en las colisiones para aplicar un impulso al objeto.

Parámetros:

addv Vector que contiene el impulso que será aplicado al objeto.

Reimplementado de **nIstEntity** *(p.144)*.

void nIstStruct::Collide (void) `[virtual]`

Maneja el comportamiento del objeto cuando éste colisiona.

Reimplementado de **nIstEntity** *(p.144)*.

bool nIstStruct::SaveCmds (nFileServer * *fs*) `[virtual]`

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

Parámetros:

fileServer writes the nCmd object contents out to a file.

Devuelve:

success or failure

Reimplementado de **nIstEntity** (*p.146*).

void nIstStruct::Trigger (float *dt*) [virtual]

Actualiza el estado del objeto.

Parámetros:

dt Delta de tiempo desde la vez anterior en la cual se llamó a este método.

Reimplementado de **nIstEntity** (*p.146*).

3.7.6 Documentación de los datos miembro

nKernelServer* nIstStruct::kernelServer [static]

Apuntador a nKernelServer.

Reimplementado de **nIstEntity** (*p.148*).

La documentación para esta clase fue generada a partir de los siguientes archivos:

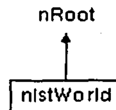
nIstStruct.h
nIstStruct_cmds.cc
nIstStruct_main.cc

3.8 Referencia de la Clase nIstWorld

Clase encargada de contener a las entidades de la historia.

`#include <nIstWorld.h>`

Diagrama de herencias de nIstWorld



3.8.1 Métodos públicos

nIstWorld ()

Método constructor.

virtual **~nIstWorld** ()

Método destructor.

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

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

vector3 **GetGravity** (void)
Regresa la gravedad del mundo.

void **SetGravity** (vector3 g)
Fija el valor de la gravedad del mundo.

nCollideContext * **GetCollideContext** ()
Regresa el contexto de colisión.

nCollideServer * **GetCollideServer** ()
Regresa el servidor de colisiones.

nCollideShape * **NewCollideShape** (const char *name, const char *file)
Crea una nueva forma de colisión.

nCollideObject * **NewCollideObject** (nIstEntity *entity)
Crea un nuevo objeto de colisión.

void **AddCollideBSPNode** (const char *pathBSPNode, const char *map_dir)
Agrega una estructura BSP al servidor de colisiones.

3.8.2 Atributos públicos estáticos

nKernelServer * **kernelServer**
Apuntador a nKernelServer.

3.8.3 Descripción detallada

Clase encargada de contener a las entidades de la historia.

La responsabilidad de esta clase es la de contener y actualizar a todas las entidades del mundo, además también contiene el contexto de colisión en el que están todas ellas.
Enrique Larios

3.8.4 Documentación del constructor y destructor

nIstWorld::nIstWorld ()

Método constructor.

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

Método destructor.

3.8.5 Documentación de las funciones miembro

void nIstWorld::AddCollideBSPNode (const char * *pathBSPNode*, const char * *map_dir*)

Agrega una estructura BSP al servidor de colisiones.

Parámetros:

pathBSPNode Ruta del nodo donde están las estructuras.
map_dir Directorio donde están los archivos que contienen la geometría de las estructuras.

nCollideContext * nIstWorld::GetCollideContext () `[inline]`

Regresa el contexto de colisión.

Devuelve:

Regresa una referencia al contexto de colisiones.

nCollideServer * nIstWorld::GetCollideServer () `[inline]`

Regresa el servidor de colisiones.

Devuelve:

Regresa una referencia al servidor de colisiones.

vector3 nIstWorld::GetGravity (void) `[inline]`

Regresa la gravedad del mundo.

Devuelve:

Vector de gravedad del mundo en m/s^2

nCollideObject * nIstWorld::NewCollideObject (nIstEntity * *entity*)

Crea un nuevo objeto de colisión.

Crea un objeto de colisión de la clase y forma dada. Es un método que usualmente sólo es llamado por **nIstEntity**.

Parámetros:

Entidad relacionada con ese objeto de colisión.

nCollideShape * nIstWorld::NewCollideShape (const char * *name*, const char * *file*)

Crea una nueva forma de colisión.

Crea una nueva forma de colisión. Particularmente llamado por nIstEntity::SetCollideShape() para agregar una geometría a una forma.

Parámetros:

name Nombre con el que se conocerá a la nueva forma.
file Nombre del archivo que contiene la geometría de la forma de colisión.

bool nIstWorld::SaveCmds (nFileServer * *fs*) `[virtual]`

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

Parámetros:

fileServer writes the nCmd object contents out to a file.

Devuelve:

success or failure

Reimplementado de **nRoot** (*p.183*).

void nIstWorld::SetGravity (vector3 *g*) `[inline]`

Fija el valor de la gravedad del mundo.

Parámetros:

Vector de gravedad del mundo en m/s^2

void nIstWorld::Trigger (float *t*) `[virtual]`

Actualiza el estado del objeto.

En esta clase en particular, este método actualiza al mundo y todas las entidades que contiene. Mediante el step_number se puede definir en cuantos pasos se hace la actualización.

Parámetros:

t Tiempo desde el inicio de ejecución del programa.

3.8.6 Documentación de los datos miembro

nKernelServer* nlstWorld::kernelServer `[static]`

Apuntador a nKernelServer.

La documentación para esta clase fue generada a partir de los siguientes archivos:

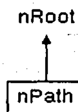
nlstWorld.h
nlstWorld_cmds.cc
nlstWorld_main.cc

3.9 Referencia de la Clase nPath

Clase que sirve de envolvente para utilizar la funcionalidad de nurbcurve en nebula.

`#include <npath.h>`

Diagrama de herencias de nPath



3.9.1 Métodos públicos

nPath ()
Método constructor.

virtual ~**nPath** ()
Método destructor.

void **SetOrigName** (const char *o)
Fija el nombre del punto origen.

void **SetDestName** (const char *d)
Fija el nombre del punto destino.

const char * **GetOrigName** (void) const
Regresa el nombre del punto origen.

const char * **GetDestName** (void) const
Regresa el nombre del punto destino.

void BeginAddHPoint (void)
Inicia el estado de agregar puntos.

void **BeginAddHPoint** (const char *or)
Inicia el estado de agregar puntos.

void **AddHPoint** (float x, float y, float z)
Agrega un punto en 3D.

void **AddHPoint** (float x, float y, float z, float w)
Agrega un punto en 4D.

void **EndAddHPoint** (void)
Termina el estado de agregar puntos.

void **EndAddHPoint** (const char *de)
Termina el estado de agregar puntos.

void CreateNurbCurve (void)
Inicializa y crea la curva nurbs.

vector3 * **PointAt** (float u)
Regresa el punto 3D correspondiente a la coordenada homogénea u.

quaternion * **HPointAt** (float u)
Regresa el punto 4D correspondiente a la coordenada homogénea u.

vector3 * **GetHPointIndex** (int i)
Regresa el punto 3D correspondiente al índice entero i.

quaternion * **GetHPointIndexW** (int i)
Regresa el punto 4D correspondiente al índice entero i.

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

3.9.2 Atributos públicos estáticos

nKernelServer * **kernelServer**
Apuntador a nKernelServer.

3.9.3 Descripción detallada

Clase que sirve de envolvente para utilizar la funcionalidad de nurbcurve en nebula.

Permite agregar a la jerarquía de Nebula una curva nurbs que funciona como camino.

3.9.4 Documentación del constructor y destructor

nPath::nPath ()

Método constructor.

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

Método destructor.

3.9.5 Documentación de las funciones miembro

void nPath::AddHPoint (float *x*, float *y*, float *z*, float *w*)

Agrega un punto en 4D.

Parámetros:

x Coordenada x del punto a agregar.
y Coordenada y del punto a agregar.
z Coordenada z del punto a agregar.
w Coordenada w del punto a agregar.

void nPath::AddHPoint (float *x*, float *y*, float *z*)

Agrega un punto en 3D.

Parámetros:

x Coordenada x del punto a agregar.
y Coordenada y del punto a agregar.
z Coordenada z del punto a agregar.

void nPath::BeginAddHPoint (const char * *or*)

Inicia el estado de agregar puntos.

Parámetros:

or Nombre del punto de origen.

void nPath::BeginAddHPoint (void)

Inicia el estado de agregar puntos.

void nPath::CreateNurbCurve (void)

Inicializa y crea la curva nurbs.

void nPath::EndAddHPoint (const char * *de*)

Termina el estado de agregar puntos.

Parámetros:

de Nombre del punto de destino.

void nPath::EndAddHPoint (void)

Termina el estado de agregar puntos.

const char * nPath::GetDestName (void) const `[inline]`

Regresa el nombre del punto destino.

Devuelve:

El nombre del punto destino.

vector3 * nPath::GetHPointIndex (int *i*)

Regresa el punto 3D correspondiente al índice entero i.

Parámetros:

i Índice entero de los puntos.

Devuelve:

Vector 3D correspondiente al índice i.

quaternion * nPath::GetHPointIndexW (int *i*)

Regresa el punto 4D correspondiente al índice entero i.

Parámetros:

i Índice entero de los puntos.

Devuelve:

Cuaternión correspondiente al índice i.

const char * nPath::GetOrigName (void) const `[inline]`

Regresa el nombre del punto origen.

Devuelve:

El nombre del origen.

quaternion * nPath::HPointAt (float *u*)

Regresa el punto 4D correspondiente a la coordenada homogénea u.

Parámetros:

u Coordenada homogénea de longitud de cuerda.

Devuelve:

Cuaternión correspondiente a la coordenada u.

vector3 * nPath::PointAt (float *u*)

Regresa el punto 3D correspondiente a la coordenada homogénea u.

Parámetros:

u Coordenada homogénea de longitud de cuerda.

Devuelve:

Vector 3D correspondiente a la coordenada u.

bool nPath::SaveCmds (nFileServer * *fs*) `[virtual]`

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

Parámetros:

fileServer writes the nCmd object contents out to a file.

Devuelve:

success or failure

Reimplementado de **nRoot** (*p.183*).

void nPath::SetDestName (const char * *d*) `[inline]`

Fija el nombre del punto destino.

Parámetros:
d Nombre del destino.

void nPath::SetOrigName (const char * *o*) [inline]

Fija el nombre del punto origen.

Parámetros:
o Nombre del origen.

3.9.6 Documentación de los datos miembro

nKernelServer* nPath::kernelServer [static]

Apuntador a nKernelServer.

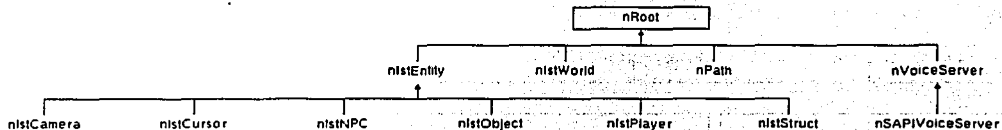
La documentación para esta clase fue generada a partir de los siguientes archivos:
npath.h
npath_main.cc

3.10 Referencia de la Clase nRoot

Root class for Nebula classes.

`#include <nroot.h>`

Diagrama de herencias de nRoot



3.10.1 Métodos públicos

nRoot (void)
virtual **~nRoot** (void)
void **AddObjectRef** (nRef< nRoot > *)
void **RemObjectRef** (nRef< nRoot > *)
nList * **GetRefs** (void)
void InvalidateAllRefs (void)
virtual void **Initialize** (void)
Will be called by the kernel after the object has been constructed and linked into the name space.

virtual bool **Release** (void)
Decrement the reference count and destroy the object if no references remain.

long **AddRef** (void)
Increment the reference count.

long **GetRefCount** (void)
Return the reference count.

void **SetFlags** (int)
void **UnsetFlags** (int)
int **GetFlags** (void)
virtual void **SetClass** (nClass *)
Set the class object that this object is an instance of.

virtual nClass * **GetClass** (void)
Return the class object that this object is an instance of.

virtual bool **IsA** (nClass *)
Test whether or not this object inherits from the class `cls` *or not.*

virtual bool **IsInstanceOf** (nClass *)
Return whether or not the object is a direct instance of the class `cls`.

virtual const char * **GetVersion** (void)
virtual bool **Save** (void)
virtual bool **SaveAs** (const char *name)
virtual nRoot * **Clone** (const char *name)
virtual bool **Parse** (const char *name)
virtual bool **SaveCmds** (nFileServer *fs)
virtual bool **Import** (const char *name)
virtual bool **Dispatch** (nCmd *)
void **GetCmdProtos** (nHashList *)
void **SetName** (const char *str)
char * **GetFullName** (char *buf, long sizeof_buf)
Get the full name within the hierarchy of the nRoot object.

char * **GetRelPath** (nRoot *other, char *buf, long sizeof_buf)
Return shortest relative path leading from 'this' to 'other' object.

nRoot * **Find** (const char *str)
void **AddHead** (nRoot *n)
void **AddTail** (nRoot *n)
nRoot * **RemHead** (void)
nRoot * **RemTail** (void)
void **Remove** (void)
void **Sort** (void)

Sort children objects alphabetically.

nRoot * **GetParent** (void)
Returns the parent object in hierarchical namespace.

nRoot * **GetHead** (void)
Returns the first child object in hierarchical namespace.

nRoot * **GetTail** (void)
Returns the last child object in hierarchical namespace.

nRoot * **GetSucc** (void)
Returns the succeeding object in hierarchical namespace.

nRoot * **GetPred** (void)
Returns the preceding object in hierarchical namespace.

const char * **GetName** (void)
Returns the name of the object in hierarchical namespace.

3.10.2 Atributos públicos estáticos

nClass * **local_cl**
pointer to instance class

nKernelServer * **ks**
pointer to kernel server

3.10.3 Tipos protegidos

enum { N_FLAG_SAVEUPSIDEDOWN = (1<<0), N_FLAG_SAVESHALLOW = (1<<1) }

3.10.4 Atributos protegidos

nList ref_list
list of references pointing to me

nClass * instance_cl
pointer to instance class object

nRoot * **parent**
parent object in hierarchical name space

nHashList * **child_list**
list of child objects in hierarchical name space

ushort ref_count
ushort root_flags
char * import_file
nRoot::Import() stores filename here...

3.10.5 Descripción detallada

Root class for Nebula classes.

nRoot defines the basic functionality and interface for the Nebula class hierarchy:

reference counting
RTTI a class is identified by a string name
object serialization
script interface
linkage into the hierarchical name space

Rules for subclasses:

only the default constructor is allowed
never create or destroy nRoot objects directly, use nKernelServer::New() to create an object and the objects **Release()** method to destroy it

3.10.6 Documentación de las enumeraciones miembro de la clase

anonymous enum `[protected]`

Valores de la enumeración:

N_FLAG_SAVEUPSIDEDOWN
N_FLAG_SAVESHALLOW

3.10.7 Documentación del constructor y destructor

nRoot::nRoot (void)

nRoot::~nRoot (void) `[virtual]`

3.10.8 Documentación de las funciones miembro

void nRoot::AddHead (nRoot * *n*)

void nRoot::AddObjectRef (nRef< nRoot > *)

long nRoot::AddRef (void)

Increment the reference count.

void nRoot::AddTail (nRoot * *n*)

virtual nRoot* nRoot::Clone (const char * *name*) `[virtual]`

bool nRoot::Dispatch (nCmd * *cmd*) `[virtual]`

nRoot * nRoot::Find (const char * *str*)

nClass * nRoot::GetClass (void) `[virtual]`

Return the class object that this object is an instance of.

void nRoot::GetCmdProtos (nHashList * *cmd_list*)

int nRoot::GetFlags (void)

char * nRoot::GetFullName (char * *buf*, long *sizeof_buf*)

Get the full name within the hierarchy of the nRoot object.

nRoot * nRoot::GetHead (void) `[inline]`

Returns the first child object in hierarchical namespace.

const char * nRoot::GetName (void) `[inline]`

Returns the name of the object in hierarchical namespace.

Reimplementado en **nIstEntity** (*p.145*).

nRoot * nRoot::GetParent (void) `[inline]`

Returns the parent object in hierarchical namespace.

nRoot * nRoot::GetPred (void) `[inline]`

Returns the preceding object in hierarchical namespace.

long nRoot::GetRefCount (void)

Return the reference count.

nList* nRoot::GetRefs (void)

char * nRoot::GetRelPath (nRoot * *other*, char * *buf*, long *sizeof_buf*)

Return shortest relative path leading from 'this' to 'other' object.

This is a slow operation, unless one object is the parent of the other (this is a special case optimization).

nRoot * nRoot::GetSucc (void) `[inline]`

Returns the succeeding object in hierarchical namespace.

nRoot * nRoot::GetTail (void) `[inline]`

Returns the last child object in hierarchical namespace.

const char * nRoot::GetVersion (void) `[virtual]`

virtual bool nRoot::Import (const char * *name*) `[virtual]`

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

Will be called by the kernel after the object has been constructed and linked into the name space.

Reimplementado en **nIstEntity** (*p.145*), y **nIstNPC** (*p.153*).

void nRoot::InvalidateAllRefs (void)

bool nRoot::IsA (nClass * *cls*) [virtual]

Test whether or not this object inherits from the class cls or not.

bool nRoot::IsInstanceOf (nClass * *cls*) [virtual]

Return whether or not the object is a direct instance of the class cls.

virtual bool nRoot::Parse (const char * *name*) [virtual]

bool nRoot::Release (void) [virtual]

Decrement the reference count and destroy the object if no references remain.

nRoot * nRoot::RemHead (void)

void nRoot::RemObjectRef (nRef< nRoot > *)

void nRoot::Remove (void)

nRoot * nRoot::RemTail (void)

virtual bool nRoot::Save (void) [virtual]

virtual bool nRoot::SaveAs (const char * *name*) [virtual]

virtual bool nRoot::SaveCmds (nFileServer * *fs*) [virtual]

Reimplementado en **nIstCamera** (*p.133*), **nIstCursor** (*p.139*), **nIstEntity** (*p.146*), **nIstNPC** (*p.153*), **nIstObject** (*p.157*), **nIstPlayer** (*p.163*), **nIstStruct** (*p.167*), **nIstWorld** (*p.171*), y **nPath** (*p.176*).

void nRoot::SetClass (nClass * *inst_cl*) [virtual]

Set the class object that this object is an instance of.

void nRoot::SetFlags (int *f*)

void nRoot::SetName (const char * *str*)

void nRoot::Sort (void)

Sort children objects alphabetically.

Not particularly optimized for runtime.

void nRoot::UnsetFlags (int *f*)

3.10.9 Documentación de los datos miembro

nHashList* nRoot::child_list `[protected]`

list of child objects in hierarchical name space

char* nRoot::import_file `[protected]`

nRoot::Import() stores filename here...

nClass* nRoot::instance_cl `[protected]`

pointer to instance class object

nKernelServer* nRoot::ks `[static]`

pointer to kernel server

Reimplementado en **nSAPIVoiceServer** (*p.187*), y **nVoiceServer** (*p.190*).

nClass* nRoot::local_cl `[static]`

pointer to instance class

Reimplementado en **nSAPIVoiceServer** (*p.187*), y **nVoiceServer** (*p.190*).

nRoot* nRoot::parent `[protected]`

parent object in hierarchical name space

ushort nRoot::ref_count `[protected]`

nList nRoot::ref_list `[protected]`

list of references pointing to me

ushort nRoot::root_flags `[protected]`

La documentación para esta clase fue generada a partir de los siguientes archivos:

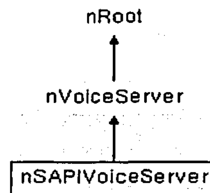
nroot.h
nroot_main.cc
nrootchild.cc

3.11 Referencia de la Clase nSAPIVoiceServer

Clase que implementa el servidor de voz mediante el Speech SDK 5.1 de MS.

`#include <nsapivoiceserver.h>`

Diagrama de herencias de nSAPIVoiceServer



3.11.1 Métodos públicos

nSAPIVoiceServer ()
Método constructor.

virtual ~nSAPIVoiceServer ()
Método destructor.

virtual void **Trigger** (void)
Actualiza el estado del servidor.

virtual bool **SetVoice** (int voiceIndex)
Fija la voz del servidor de acuerdo al índice.

virtual bool **SetVoice** (const char *voiceName)
Fija la voz del servidor de acuerdo al nombre.

virtual bool **Speak** (const char *speech)
Convierte de texto a voz.

3.11.2 Atributos públicos estáticos

nClass * **local_cl** = NULL

Apuntador a nClass.

nKernelServer * **ks** = NULL
Apuntador a nKernelServer.

3.11.3 Descripción detallada

Clase que implementa el servidor de voz mediante el Speech SDK 5.1 de MS.

Esta clase hereda de **nVoiceServer** y que implementa la funcionalidad necesaria mediante un conjunto de herramientas de Microsoft. Por lo tanto esta clase sólo es compilable en el sistema operativo Windows.

Creada por Enrique Larios.

3.11.4 Documentación del constructor y destructor

nSAPIVoiceServer::nSAPIVoiceServer ()

Método constructor.

También aquí se realiza la iniciación de COM.

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

Método destructor.

También aquí se cierra COM.

3.11.5 Documentación de las funciones miembro

bool nSAPIVoiceServer::SetVoice (const char * *voiceName*) `[virtual]`

Fija la voz del servidor de acuerdo al nombre.

Permite seleccionar entre las voces incluidas en el SAPI 5.1 de Microsoft.

Parámetros:

voiceName Nombre que identifica a la voz que se quiere fijar.

Reimplementado de **nVoiceServer** (*p.189*).

bool nSAPIVoiceServer::SetVoice (int *voiceIndex*) `[virtual]`

Fija la voz del servidor de acuerdo al índice.

Permite seleccionar entre las voces incluidas en el SAPI 5.1 de Microsoft.

Parámetros:

voiceIndex Índice que identifica a la voz que se quiere fijar.

Reimplementado de **nVoiceServer** (*p.190*).

bool nSAPIVoiceServer::Speak (const char * *speech*) `[virtual]`

Convierte de texto a voz.

Parámetros:

speech Texto que se convierte a voz.

Reimplementado de **nVoiceServer** (*p.190*).

void nSAPIVoiceServer::Trigger (void) `[virtual]`

Actualiza el estado del servidor.

Reimplementado de **nVoiceServer** (*p.190*).

3.11.6 Documentación de los datos miembro

nKernelServer * nSAPIVoiceServer::ks = NULL `[static]`

Apuntador a nKernelServer.

Reimplementado de **nVoiceServer** (*p.190*).

nClass * nSAPIVoiceServer::local_cl = NULL `[static]`

Apuntador a nClass.

Reimplementado de **nVoiceServer** (*p.190*).

La documentación para esta clase fue generada a partir de los siguientes archivos:

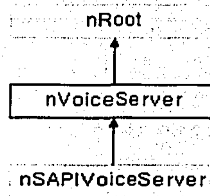
nsapivoiceserver.h
nsapivoiceserver_init.cc
nsapivoiceserver_main.cc

3.12 Referencia de la Clase nVoiceServer

Clase base que sirve para crear distintas implementaciones de servidor de voz.

`#include <nvoiceserver.h>`

Diagrama de herencias de nVoiceServer



3.12.1 Métodos públicos

nVoiceServer ()
: *Método constructor.*

virtual ~nVoiceServer ()
: *Método destructor.*

bool **Ready** (void)
: *Regresa verdadero si el servidor esta listo para convertir de texto a voz.*

int GetNumVoices (void)
: *Regresa el número de diferentes voces que tiene el servidor.*

virtual void **Trigger** (void)
: *Actualiza el estado del servidor.*

virtual bool **SetVoice** (int voiceIndex)
: *Fija la voz del servidor de acuerdo al índice.*

virtual bool **SetVoice** (const char *voiceName)
: *Fija la voz del servidor de acuerdo al nombre.*

virtual bool **Speak** (const char *speech)
: *Convierte de texto a voz.*

3.12.2 Atributos públicos estáticos

nClass * **local_cl** = NULL
: *Apuntador a nClass.*

nKernelServer * **ks** = NULL
: *Apuntador a nKernelServer.*

3.12.3 Atributos protegidos

nAutoRef< nInputServer > **ref_is**
nAutoRef< nGfxServer > **ref_gs**
int numVoices
bool **ready**

3.12.4 Descripción detallada

Clase base que sirve para crear distintas implementaciones de servidor de voz.

Esta se debe de tomar como una clase abstracta, y heredar de ella las clases que realmente implementan la funcionalidad de servidor de voz.

Creada por Enrique Larios.

3.12.5 Documentación del constructor y destructor

nVoiceServer::nVoiceServer ()

Método constructor.

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

Método destructor.

3.12.6 Documentación de las funciones miembro

int nVoiceServer::GetNumVoices (void) `[inline]`

Regresa el número de diferentes voces que tiene el servidor.

Devuelve:

Número de voces que soporta el sistema.

bool nVoiceServer::Ready (void) `[inline]`

Regresa verdadero si el servidor esta listo para convertir de texto a voz.

Devuelve:

Si está listo el sistema para convertir.

bool nVoiceServer::SetVoice (const char * *voiceName*) `[virtual]`

Fija la voz del servidor de acuerdo al nombre.

Este método se debe implementar en una subclase para agregarle la funcionalidad.

Parámetros:

voiceName Nombre que identifica a la voz que se quiere fijar.

Reimplementado en **nSAPIVoiceServer** (*p.186*).

bool nVoiceServer::SetVoice (int *voiceIndex*) [virtual]

Fija la voz del servidor de acuerdo al índice.

Este método se debe implementar en una subclase para agregarle la funcionalidad.

Parámetros:

voiceIndex Índice que identifica a la voz que se quiere fijar.

Reimplementado en **nSAPIVoiceServer** (*p.186*).

bool nVoiceServer::Speak (const char * *speech*) [virtual]

Convierte de texto a voz.

Este método se debe implementar en una subclase para agregarle la funcionalidad.

Parámetros:

speech Texto que se convierte a voz.

Reimplementado en **nSAPIVoiceServer** (*p.187*).

void nVoiceServer::Trigger (void) [virtual]

Actualiza el estado del servidor.

Este método se debe implementar en una subclase para agregarle la funcionalidad.

Reimplementado en **nSAPIVoiceServer** (*p.187*).

3.12.7 Documentación de los datos miembro

nKernelServer * nVoiceServer::ks = NULL [static]

Apuntador a nKernelServer.

Reimplementado de **nRoot** (*p.184*).

Reimplementado en **nSAPIVoiceServer** (*p.187*).

nClass * nVoiceServer::local_cl = NULL [static]

Apuntador a nClass.

Reimplementado de **nRoot** (*p.184*).

Reimplementado en **nSAPIVoiceServer** (*p.187*).

int nVoiceServer::numVoices [protected]

bool nVoiceServer::ready [protected]

nAutoRef<nGfxServer> nVoiceServer::ref_gs [protected]

nAutoRef<nInputServer> nVoiceServer::ref_is [protected]

La documentación para esta clase fue generada a partir de los siguientes archivos:

- **nvoiceserver.h**
- **nvoiceserver_init.cc**
- **nvoiceserver_main.cc**

Apéndice B

Código Fuente del API para HIC.

Debido a la gran cantidad de espacio que requeriría mostrar todo el código, se ha optado por solo presentar el código de las cuatro clases mas importantes: nIstCamera, nIstNPC, nIstPlayer y nIstWorld.

1. Archivos de nIstCamera

1.1. nistcamera.h

```
#ifndef N_ISTCAMERA_H
#define N_ISTCAMERA_H
//------------------------------------------------------------------------------
/**
   @class nIstCamera

   @brief Clase encargada de controlar el punto de vista desde el que se presenta la escena.

   Esta clase mantiene información acerca del punto de vista y la orientación
         desde la cual se presenta las escenas. Esa información es guardada en una
         matriz de 4x4. Soporta varios estilos de cámara, para seguir o apuntar a
         ciertos objetivos. También pude manejar las entradas del usuario para los
         estilos que así lo requieran.

         Creada por Enrique Larios

  Debido a la zxzcv nistcamera
*/

#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"

//------------------------------------------------------------------------------

class nInputEvent;
class nPath;
class nInputServer;


class N_PUBLIC nIstCamera : public nIstEntity
{
public:

        //Valores de los diferentes estilos de cámara.
  enum Style
  {
```

```
    STATIONARY,
    LOCKED_CHASE,
            CHASE,
            AIM_PATH,
            LOOK_PATH,
            FPS,
            FREE
  };

public:
  /// Método constructor
  nIstCamera();
  /// Método destructor
  virtual ~nIstCamera();
  /// Método encargado de la persitencia, salva en un archivo de comandos.
  virtual bool SaveCmds(nFileServer* fileServer);

  /// Regresa la matriz de orientación y posición de la cámara.
  const matrix44& GetTransform() const;


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

        /// Maneja el comportamineto del objeto cuando éste colisiona.
        void Collide(void);


        /// Usado en las colisiones para agregar impulso.
        void AddImpulse(vector3 addv);


  /// Coloca a la cámara en la posición. (solo en estilo estacionario).
  void SetPosition(float x, float y, float z);

        /// Activa el estilo estacionario.
        void SetStyleStationary(vector3 targetPos,float angleX,float angleY,float angleZ,float dist);

        /// Activa el estilo estacionario.
        void SetStyleStationary(n3DNode *target,float angleX,float angleY,float angleZ,float dist);

        /// Activa el estilo estacionario.
        void SetStyleStationary(vector3 targetPos,vector3 cameraPos);

        /// Activa el estilo estacionario.
        void SetStyleStationary(n3DNode *target,n3DNode *camer);

        /// Activa el estilo persecución.
        void SetStyleChase(n3DNode *target,float height, float prefDist);

        /// Activa el estilo persecución fija.
        void SetStyleLockedChase(n3DNode *target,float angleX,float angleY,float angleZ,float dist);

        /// Activa el estilo de apuntar y seguir un camino.
        void SetStyleAimPath(n3DNode *target,const char *orig,const char *dest,float len);

        /// Activa el estilo de orientación predefinida y seguir un camino.
        void SetStyleLookPath(const char *origPos,const char *destPos,const char *origAng,const char
*destAng,float len);

        /// Activa el estilo de primera persona.
        void SetStyleFPS(n3DNode *target,float height);

        /// Activa el estilo de cámara libre.
        void SetStyleFree();

        ///Maneja las entradas del usuario. Cuando el estilo de cámara lo requiere.
        void handleInput(nInputEvent *ie );

        /// Regresa el cuaternión de la matriz de orientación.
        quaternion getQ(void);
```

```
        /// Regresa la posición de la cámara.
        vector3 getT(void);

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


    /// Apuntador a nKernelServer
    static nKernelServer* kernelServer;

        ///Apuntador a nClass
        static nClass *loacl_cl;

private:
        nAutoRef<nInputServer>      ref_is;
        n3DNode *cameraNode;

    matrix44 cameraMat;
        matrix44 targetMat;
        n3DNode *myTarget;

        nPath *actualPath;
        bool seekingBack;

        float pathLength;
        float pathTime;

        nPath *actualAngPath;
        bool seekingAngBack;

        float fpsHeight;

        float chaseCamHeight;
        float chaseDistance;
        float chaseAngleRad;

    nlstCamera::Style cameraStyle;
        nlstCamera::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;

        float distance;

        matrix44 Rx,Ry;

        void handleViewer(void);
        void handleChaseViewer(void);

        vector3 *SeekPath(float dt);
        quaternion *SeekAngPath(float dt);

};

//-------------------------------------------------------------------------
/**
   Esta matriz normalmente es requerida por
```

```
        la clase escena para conocer el punto desde el cual se observa.
    @return          Matriz de posición y orientación de la cámara.
*/
inline
const matrix44& nlstCamera::GetTransform() const
{
    return cameraMat;
}
//-----------------------------------------------------------------------
/**
    Estilo totalmente controlado por las entradas del usuario mediante el ratón.
*/
inline
void nlstCamera::SetStyleFree()
{
        this->lastCamStyle=this->cameraStyle;
        if(lastCamStyle == nlstCamera::FPS) myTarget->SetActive(true);
        this->cameraStyle=nlstCamera::FREE;
}
//-----------------------------------------------------------------------
/**
    El cuaternión contiene toda la información necesaria respecto a la orientación de la cámara.
    @return          Cuaternion con la información de la matriz de orientación.
*/
inline
quaternion nlstCamera::getQ(void)
{
        return cameraMat.get_quaternion();
}
//-----------------------------------------------------------------------
/**
    @return          Vector con la información de posición.
*/
inline
vector3 nlstCamera::getT(void)
{
        return cameraMat.pos_component();
}
//-----------------------------------------------------------------------
/**
    @param     x   Componente x del cuaternión.
        @param     y      Componente y del cuaternión.
        @param     z      Componente z del cuaternión.
        @param     w      Componente w del cuaternión.
*/
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
```

1.2. nistcamera_main.cc

```
#define N_IMPLEMENTS nIstCamera
//-------------------------------------------------------------------------
// Creado. Enrique Larios.
//-------------------------------------------------------------------------
#include "node/n3dnode.h"
#include "input/ninputserver.h"
#include "ist/nistcamera.h"
#include "path/npath.h"
#include <string>
nNebulaScriptClass(nIstCamera, "nroot");
vector3 upVector(0,2,0);

//-------------------------------------------------------------------------
/**
*/
nIstCamera::nIstCamera() :
   cameraMat(),
          targetMat(),
          myTarget(),
          cameraStyle(nIstCamera::FREE),
          lastCamStyle(nIstCamera::FREE),
          ref_is(kernelServer, this),
          seekingBack(false),
          seekingAngBack(false),
          actualPath(NULL),
          actualAngPath(NULL),
          fpsHeight(50)
{

   cameraMat.ident();
          targetMat.ident();

          //Usados en handleInput
          cur_up    = false;
          cur_down  = false;
          cur_left  = false;
          cur_right = false;
          ctrl      = false;
          shift     = false;
          lmb = false;
          mmb = false;
          rmb = false;
          normalize = false;
          old_x = 0;
          old_y = 0;
          act_x = 0;
          act_y = 0;
          pathLength=0;
          pathTime=0;
}
//-------------------------------------------------------------------------
/**
*/
nIstCamera::~nIstCamera()
{
}

void nIstCamera::Collide(void)
{
}
//-------------------------------------------------------------------------
/**

 @param  addv   Vector que contiene el impulso que será aplicado al objeto.

*/
void nIstCamera::AddImpulse(vector3 addv)
{

}
```

```
//---------------------------------------------------------------------------
/**

  @param  x    Fija la coordenada x de la posición de la cámara.
  @param  y    Fija la coordenada y de la posición de la cámara.
  @param  z    Fija la coordenada z de la posición de la cámara.

*/
void nlstCamera::SetPosition(float x, float y, float z)
{
        if(cameraStyle == nlstCamera::STATIONARY || cameraStyle == nlstCamera::FREE )
                cameraMat.translate(x,y,z);
}
//---------------------------------------------------------------------------
/**

        @param  dt          Delta de tiempo desde la vez anterior en la cual se llamó a este método.

*/
void nlstCamera::Trigger(float dt)
{

   switch (cameraStyle)
   {
     case nlstCamera::STATIONARY:
        // No hagas nada
      break;

     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=nlstCamera::FREE;
		}

}
break;

	case nlstCamera::FPS:
{
		if(myTarget)
		{
			cameraMat=myTarget->GetM();
			vector3 pos =cameraMat.pos_component();
			pos.y+=fpsHeight;
			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.mult_simple(qSwp);

			cameraMat.translate(pos);

		}

	}
break;
```

```
                case nlstCamera::FREE:
    {
                        this->handleViewer();
    }
    break;
  }
}

//------------------------------------------------------------------------
/**
  Si no es el caso, el evento se deja desatendido.
      Actualmente solo el estilo libre y el estilo de persecución requieren
      de entradas del usuario.

      @param  ie        Evento de entrada que debe de ser generado por el servidor de entradas.

*/
void nlstCamera::handleInput(nInputEvent *ie )
{
        if(this->cameraStyle != nlstCamera::FREE  && this->cameraStyle != nlstCamera::CHASE) 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:
      {
        old_x = act_x = ie->GetAbsXPos();
        old_y = act_y = ie->GetAbsYPos();
        switch (ie->GetButton()) {
          case N_KEY_LBUTTON:  lmb=true; break;
          case N_KEY_RBUTTON:  rmb=true; break;
          case N_KEY_MBUTTON:  mmb=true; break;
          default: break;
        }
      }
      break;
    case N_INPUT_BUTTON_UP:
      {
        old_x = act_x = ie->GetAbsXPos();
        old_y = act_y = ie->GetAbsYPos();
        switch (ie->GetButton()) {
          case N_KEY_LBUTTON:  lmb=false; break;
          case N_KEY_RBUTTON:  rmb=false; break;
          case N_KEY_MBUTTON:  mmb=false; break;
          default: break;
        }
      }
      break;
    case N_INPUT_MOUSE_MOVE:
      {
        act_x = ie->GetAbsXPos();
```

```
            act_y = ie->GetAbsYPos();
        }
        break;
    default: break;
    }
}

//-----------------------------------------------------------------
void nlstCamera::handleViewer(void)
{
    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 -= 0.25f;
        if (cur_down)  rx += 0.25f;
        if (cur_left)  ry -= 0.25f;
        if (cur_right) ry += 0.25f;
    } else {
        if (ctrl) {
            if (cur_up)    ty -= 1.0f;
            if (cur_down)  ty += 1.0f;
        } else {
            if (cur_up)    tz -= 1.0f;
            if (cur_down)  tz += 1.0f;
            if (cur_left)  tx -= 1.0f;
            if (cur_right) tx += 1.0f;
        }
    }
    if (rmb) {
        ry += ((float)(old_x-act_x)) * 0.05f;
        rx += ((float)(old_y-act_y)) * 0.05f;
    }
    if (lmb) {
        tz += ((float)(old_y-act_y) * 1.0f);
        tx += ((float)(old_x-act_x) * 1.0f);
    }

    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);

    // Translacion de la matriz
    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 nlstCamera::handleChaseViewer(void)
{
	float rx=0;

	if (rmb) {
   chaseAngleRad += ((float)(old_y-act_y)) * 0.05f;

		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;
  }

	old_x = act_x;
  old_y = act_y;
}

//-------------------------------------------------------------------------------
/**

	@param  targetPos	Posición del objetivo.
	@paramangleX	Orientación en eje x de la cámara con respecto al objetivo.
	@paramangleY	Orientación en eje y de la cámara con respecto al objetivo.
	@paramangleZ	Orientación en eje z de la cámara con respecto al objetivo.
	@param  dist	Distancia de la cámara con respecto del objetivo.

*/
void nlstCamera::SetStyleStationary(vector3 targetPos,float angleX,float angleY,float angleZ,float dist)
{
	this->lastCamStyle=this->cameraStyle;
	if(lastCamStyle == nlstCamera::FPS) myTarget->SetActive(true);
	this->cameraStyle=nlstCamera::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;
}
//-------------------------------------------------------------------------------
/**
	@param  target	Objetivo al que apunta la cámara.
	@paramangleX	Orientación en eje x de la cámara con respecto al objetivo.
	@paramangleY	Orientación en eje y de la cámara con respecto al objetivo.
	@paramangleZ	Orientación en eje z de la cámara con respecto al objetivo.
	@param  dist	Distancia de la cámara con respecto del objetivo.

*/
void nlstCamera::SetStyleStationary(n3DNode *target,float angleX,float angleY,float angleZ,float dist)
{
	vector3 pos;
	target->GetT(pos.x,pos.y,pos.z);
	this->SetStyleStationary(pos,angleX,angleY,angleZ,dist);
}

//-------------------------------------------------------------------------------
/**

	@param  targetPos	Vector con la posición del objeto al que apunta la cámara.
	@paramcameraPos	Vector con la posición de la cámara.
```

```
*/
void nlstCamera::SetStyleStationary(vector3 targetPos,vector3 cameraPos)
{
        this->lastCamStyle=this->cameraStyle;
        if(lastCamStyle == nlstCamera::FPS) myTarget->SetActive(true);
        this->cameraStyle=nlstCamera::STATIONARY;
        cameraMat.ident();
        cameraMat.translate(cameraPos);
        cameraMat.lookat(targetPos,upVector);
}

//--------------------------------------------------------------------------
/**
        @param  targetPos       Objeto de escena al que apunta la cámara.
        @paramcameraPos         Objeto de escena con la posición de la cámara.

*/
void nlstCamera::SetStyleStationary(n3DNode *target,n3DNode *camer)
{
        this->lastCamStyle=this->cameraStyle;
        if(lastCamStyle == nlstCamera::FPS) myTarget->SetActive(true);
        this->cameraStyle=nlstCamera::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);
}
//--------------------------------------------------------------------------
/**

        @param  target   Objeto de escena al que persigue la cámara.
        @paramheight     Altura del objeto de escena al que persigue la cámara.
        @paramprefDist   Distancia a la que mantener la persecución.

*/
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;

}

//--------------------------------------------------------------------------
/**

        @param  target   Objetivo al que persigue la cámara.
        @paramangleX     Orientación en eje x de la cámara con respecto al objetivo.
        @paramangleY     Orientación en eje y de la cámara con respecto al objetivo.
        @paramangleZ     Orientación en eje z de la cámara con respecto al objetivo.
        @param  dist     Distancia de la cámara con respecto del objetivo.

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

}

//---------------------------------------------------------------------------
/**

        @param  target   Objetivo al que apunta la cámara.
        @paramorig       Denominación del punto de origen del camino.
        @paramdest       Denominación del punto de destino del camino.
        @paramlen                 Tiempo requerido para recorrer el camino.

*/
void nlstCamera::SetStyleAimPath(n3DNode *target,const char *orig,const char *dest,float len)
{
        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->lastCamStyle=this->cameraStyle;
        if(lastCamStyle == nlstCamera::FPS) myTarget->SetActive(true);
        this->cameraStyle=nlstCamera::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);

}

//------------------------------------------------------------------------
/**

        @param origPos  Denominación del punto de origen del camino.
        @param destPos  Denominación del punto de destino del camino.
        @param origAng  Denominación del punto de origen de la ruta de cámara.
        @param destAng  Denominación del punto de destino de la ruta de cámara.
        @param len              Tiempo requerido para recorrer el camino.

*/
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 == nIstCamera::FPS) myTarget->SetActive(true);
        this->cameraStyle=nIstCamera::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);


		//Fijando la nueva cameraMat
		this->cameraMat.ident();
		this->cameraMat.mult_simple(matrix44(*cameraQ));
		this->cameraMat.translate(*cameraPos);

}


//------------------------------------------------------------------------
/**

	@param  target   Objeto del cual se toma la vista de primera persona.
	@param height  Altura a la que se quiere colocar la cámara.

*/
void nlstCamera::SetStyleFPS(n3DNode *target,float height)
{
	if(target)
	{
		this->lastCamStyle=this->cameraStyle;
		if(lastCamStyle == nlstCamera::FPS) myTarget->SetActive(true);
		this->cameraStyle=nlstCamera::FPS;
		myTarget=target;
		myTarget->SetActive(false);
		fpsHeight=height;

	}
}

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 *nlstCamera::SeekAngPath(float dt)
{
	if( cameraStyle==nlstCamera::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;
}
```

2. Archivos de nIstNPC

2.1. nistnpc.h

```
//==============================================================================
// ist/nIstNPC.h
// 2002 Enrique 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 nIstNPC

    @brief Clase responsable de controlar a los personajes de acuerdo con las acciones del usuario.

    Mediante las acciones del usuario, representadas en su personaje, esta clase responde a las acciones
        del usuario mediante scripts. Controla y actualiza la animación y la posición del modelo que
        representa al parsonaje. Además puede seguir caminos.
        Creada por Enrique Larios.
```

```
*/
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;             // en m/s
        float rot_speed;   // en grados/s

        //Atributos al seguir caminos
        double pathDur;
        bool seekingBack;
        double pathTime;
        nPath *actualPath;

        //Estados
        bool StatePtSk;

        //Habilita acciones en eventos.
        bool EnEvColl;

        nIstEntity *lastCollEntity;

        nString snpcnode;
        nString snpcmodel;

protected:
        bool BeginPathSeeking(const char *orig,const char *dest,double duration);
        vector3 *SeekPath(float dt);

public:
        /// Método constructor.
    nIstNPC();

        /// Método destructor.
    virtual ~nIstNPC();

        ///Inicializa diversos valores de la clase.
        virtual void Initialize();
        /// Apuntador a nKernelServer
    static nKernelServer* kernelServer;
        /// Método encargado de la persitencia, salva en un archivo de comandos.
        virtual bool SaveCmds(nFileServer* fileServer);
        /// Actualiza el estado del objeto.
        void Trigger(float dt);
        /// Maneja el comportamiento del objeto cuando éste colisiona.
        void Collide(void);
        /// Actualiza la posición del objeto.
        void UpdatePosition(float dt);
        /// Usado en las colisiones para aplicar un impulso al objeto.
        void AddImpulse(vector3 addv);
        /// Lama a la animación "caminar" del modelo que representa al personaje.
        void AnimWalk();
        /// Lama a la animación "girar" del modelo que representa al personaje.
        void AnimStrut();
        /// Lama a la animación "parado" del modelo que representa al personaje.
        void AnimIdle();
```

```
        /// Fija la velocidad de Giro.
        void SetRotSpeed(float rs);
        /// Fija la velocidad al correr.
        void SetRunSpeed(float rs);
        /// Regresa la velocidad de Giro.
        float GetRotSpeed();
        /// Regresa la velocidad al correr.
        float GetRunSpeed();

        /// Fija el nodo que contiene el modelo del personaje.
        void SetN3DNodeTar(const char *path);
        /// Regresa el nombre del nodo que contiene el modelo del personaje.
        const char * GetN3DNodeTar() const;
        /// Fija el modelo que representa al personaje.
        void SetNCalModTar(const char *path);
        /// Regresa el nombre del modelo que representa al personaje.
        const char * GetNCalModTar() const;
        /// Fija el servidor de colisiones.
        void SetCollServ(const char *path);
        /// Regresa el servidor de colisiones.
        const char * GetCollServ() const;

        /// Habilia la ejecución de scripts con eventos.
        void EnableEventColl(const char *collScriptFile);
        /// Deshabilia la ejecución de scripts con eventos.
        void DisableEventColl();
        /// Regresa la última entidad con la que colisionó.
        nlstEntity *GetLastCollEntity(void);
        /// Coloca al personaje en el estado seguir camino.
        bool SetStatePathSeeking(const char *orig,const char *dest,double duration);

};

//---------------------------------------------------------------------------
/**
        @return Nombre del nodo que contiene al modelo.
*/
inline
const char * nlstNPC::GetN3DNodeTar() const
{
        return ref_npcnode.getname();
}
//---------------------------------------------------------------------------
/**
        @return Nombre del modelo del personaje.
*/
inline const char * nlstNPC::GetNCalModTar() const
{
        return ref_npcmodel.getname();
}

//---------------------------------------------------------------------------
/**
        @paramrs Velocidad de giro en grados/s.
*/
inline void nlstNPC::SetRotSpeed(float rs) {
        this->rot_speed=rs;
}
//---------------------------------------------------------------------------
/**
        @paramms Velocidad que puede alcanzar el personaje al correr en m/s.
*/
inline void nlstNPC::SetRunSpeed(float rs) {
        this->run_speed=rs;
}
//---------------------------------------------------------------------------
/**
        @return Velocidad de giro en grados/s.
*/
inline float nlstNPC:: GetRotSpeed() {
        return rot_speed;
```

```
}
//---------------------------------------------------------------------------
/**
          @return Velocidad al correr en m/s.
*/
inline float nlstNPC:: GetRunSpeed() {
          return run_speed;
}

//---------------------------------------------------------------------------
/**
          @param path      Ruta en la jerarquía donde está el nodo que contiene al modelo.
*/
inline void nlstNPC::SetN3DNodeTar(const char *path) {
          n_assert(path);
          snpcnode=path;
          this->ref_npcnode=path;
          this->npcnode = ref_npcnode.get();
}
//---------------------------------------------------------------------------
/**
          @param path      Ruta en la jerarquía donde está el modelo.
*/
inline void nlstNPC::SetNCalModTar(const char *path) {
          n_assert(path);
          snpcmodel=path;
          this->ref_npcmodel=path;
          this->npcmodel = ref_npcmodel.get();
}
//---------------------------------------------------------------------------
/**
          @param path      Ruta en la jerarquía donde está el servidor de colisiones.
*/
inline
void nlstNPC::SetCollServ(const char *path)
{
          n_assert(path);
          this->ref_collide=path;
}
//---------------------------------------------------------------------------
/**
          @return Ruta en la jerarquía donde está el servidor de colisiones.
*/
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);
}
//---------------------------------------------------------------------------
/**
          Este método normalmente es llamado desde los métodos de colision de los objetos con que
colisiona.

          @param  addv    Vector que contiene el impulso que será aplicado al objeto.

*/
```

```
inline void nIstNPC::AddImpulse(vector3 addv)
{
	velVector+=addv;
}

//------------------------------------------------------------------------
/**
	@param collScriptFile   Nombre del archivo que contiene el script a ejecutar.
*/
inline void nIstNPC::EnableEventColl(const char *collScriptFile)
{
	this->EnEvColl=collScriptlet->ParseFile(collScriptFile);
}
//------------------------------------------------------------------------
/**
	@return Último objeto con él que colisionó.
*/
inline
nIstEntity *nIstNPC::GetLastCollEntity(void)
{
	return this->lastCollEntity;
}
//------------------------------------------------------------------------
inline
void nIstNPC::DisableEventColl()
{
	this->EnEvColl=false;
}
//------------------------------------------------------------------------
#endif
```

2.2. nistnpc_main.cc

```
#define N_IMPLEMENTS nIstNPC
//==========================================================================
// ist/nistnpc_main
// Enrique 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";
}
//------------------------------------------------------------------------
nlstNPC::~nlstNPC()
{
        collScriptlet->~nTclScriptlet();
}
//------------------------------------------------------------------------
/**
        En esta clase en particular, este método carga todos los scripts de comportamiento.
*/
void nlstNPC::Initialize()
{
        nlstEntity::Initialize();

        std::string scriptPath="/sys/share/scriptlets/";

        scriptPath+=GetName();
        scriptPath+="/";
        this->collScriptlet= (nTclScriptlet *) kernelServer->New("ntclscriptlet",(
scriptPath+"collscriptlet").c_str() );
}
//------------------------------------------------------------------------
/**
        En esta clase en particular, este método mantiene las animaciones y el movimiento del personaje.
        Por ejemplo, cuando el personaje sigue un camino.

        @param  dt      Delta de tiempo desde la vez anterior en la cual se llamó a este método.
*/
void nlstNPC::Trigger(float dt)
{
        this->collideObject->Transform(0.0F, npcnode->GetM() );

        nlstEntity::Trigger(dt);      //Maneja fierzas y gravedad

        if(StatePtSk)
        {
                if(pathTime<pathDur )
                {
                        //Obtiene posición actual.
                        vector3  actPos;
                        npcnode->GetT(actPos.x,actPos.y,actPos.z);

                        //Obtiene posición nueva del camino.
                        vector3 *newPos=SeekPath(dt);


                        //Vector de dirección
                        vector3 * newDir=new vector3(newPos->x-actPos.x,0.0F,newPos->z-actPos.z);
                        newDir->norm();
                        newDir->y=0;
```

```
				//Si el loop es muy rápido y no hay diferencia.
				if(newDir->len() >0)
				{
					vector3 w;
					npcnode->GetR(w.x,w.y,w.z);			//giro anterior

					if(newDir->x>=0)
						w.y = n_rad2deg( float( acos( newDir->z )  ) );
					else
						w.y = -n_rad2deg( float( acos( newDir->z  )  ) );

					//girando
					npcnode->Rxyz(w.x, w.y, w.z);

					//fija nueva posición.
					npcnode->Txyz(newPos->x,newPos->y,newPos->z);
				}
			}
			else
			{
				this->StatePtSk=false;
				this->AnimIdle();
			}
	}//end StatePtSk
	//Actualizando posición de collideObject
	collideObject->Transform(0.0F, npcnode->GetM() );
}
//-----------------------------------------------------------------------------
/**
	En esta clase en particular, este método se encarga de llamar a la ejecución de scripts. Si este
	evento está habilitado.
*/
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;

				if(StatePtSk )
				{
					(*contact_normal).norm();
					(*contact_normal).y=0;
					velVector += (*contact_normal)*(-run_speed)*1.5;


				}
```

```
					else
					{
						other->AddImpulse(-(*contact_normal)*(-run_speed)*1.5);
					}
					if(EnEvColl)
						collScriptlet->Run();
				}
		}
}
//-------------------------------------------------------------------------
/**
	@param  dt	Delta de tiempo desde la vez anteriror en la cual se llamó a este método.
*/
void nlstNPC::UpdatePosition(float dt)
{
}
//-------------------------------------------------------------------------
/**

	@param orig	Denominación del punto de origen del camino
	@param dest	Denominación del punto de destino del camino
	@param durarion Tiempo en recorrer el camino
	@return Resultado. En especial si encuentra un camino que lo lleve de origen a destino.
*/
bool nlstNPC::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 nlstNPC::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 *nlstNPC::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;
}
```

3. Archivos de nIstPlayer.

3.1. nistplayer.h

```
//==========================================================================
// ist/nIstPlayer.h
//  Enrique 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 usado para fijar la escala de las coordenadas del motor con las distancias reales que representan.
#define SPACECOORD_SCALE 45.0F
//--------------------------------------------------------------------------
/**
   @class nIstPlayer

   @brief Clase responsable de recibir las entradas del usuario y de controlar a su personaje.

   Mediante las entradas el usuario, que se utilizan en el servidor de entrada para llamar a los métodos
        iniciadores de esta clase, esta clase controla y actualiza la animación y la posición del modelo que
        representa al usuario.

        Creada por Enrique Larios.

*/
```

```
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;            // en m/s^2
        float max_speed;
        float rot_speed;   // en grados/s

        bool isFW;
        bool isBW;
        bool isRL;
        bool isRR;
        bool isSTPFW;
        bool isSTPBW;

        nString splayernode;
        nString smodel;
        //auxiliar
        bool animWalk;

public:
        /// Método constructor.
   nIstPlayer();
        /// Método destructor.
   virtual ~nIstPlayer();

        /// Apuntador a nKernelServer
   static nKernelServer* kernelServer;
        /// Método encargado de la persitencia, salva en un archivo de comandos.
        virtual bool SaveCmds(nFileServer* fileServer);

        /// Actualiza el estado del objeto.
        void Trigger(float dt);
        /// Maneja el comportamiento del objeto cuando éste colisiona.
        void Collide(void);
        /// Actualiza la posición del objeto.
        void UpdatePosition(float dt);
        /// Usado en las colisiones para aplicar un impulso al objeto.
        void AddImpulse(vector3 addv);

        /// Coloca al personaje en el estado de caminar hacia adelante.
        void StartFW();
        /// Coloca al personaje en el estado de caminar hacia atras.
        void StartBW();
        /// Coloca al personaje en el estado de girar a la izquierda.
        void StartRL();
        /// Coloca al personaje en el estado de girar a la derecha.
        void StartRR();

        /// Detiene al personaje de caminar hacia adelante.
        void StopFW();
        /// Detiene al personaje de caminar hacia atras.
        void StopBW();
        /// Detiene al personaje de girar a la izquierda.
        void StopRL();
        /// Detiene al personaje de girar a la derecha.
        void StopRR();
        /// Llama a la animación de acción.
        void ActShoot();
        /// Fija la velocidad de Giro.
        void SetRotSpeed(float rs);
        /// Fija la máxima velocidad que se alcanza al correr.
```

```
        void SetMaxRunSpeed(float ms);
        /// Fija la aceleración al correr.
        void SetRunAccel(float ac);
        /// Regresa la velocidad de Giro.
        float GetRotSpeed();
        /// Regresa la máxima velocidad que se alcanza al correr.
        float GetMaxRunSpeed();
        /// Regresa a aceleración al correr.
        float GetRunAccel();
        /// Fija el nodo que contiene el modelo del personaje.
        void SetN3DNodeTar(const char *path);
        /// Regresa el nombre del nodo que contiene el modelo del personaje.
        const char * GetN3DNodeTar() const;
        /// Fija el modelo que representa al personaje.
        void SetNCalModTar(const char *path);
        /// Regresa el nombre del modelo que representa al personaje.
        const char * GetNCalModTar() const;
        /// Fija el servidor de colisiones.
        void SetCollServ(const char *path);
        /// Regresa el servidor de colisiones.
        const char * GetCollServ() const;
};

//------------------------------------------------------------------------
/**
        @return Nombre del nodo que contiene al modelo.
*/
inline const char * nlstPlayer::GetN3DNodeTar() const
{
        return splayernode.Get();
}

//------------------------------------------------------------------------
/**
        @return Nombre del modelo del personaje.
*/
inline const char * nlstPlayer::GetNCalModTar() const
{
        return smodel.Get();
}
//------------------------------------------------------------------------
/**
        @paramrs Velocidad de giro en grados/s.
*/
inline void nlstPlayer::SetRotSpeed(float rs) {
        this->rot_speed=rs;
}
//------------------------------------------------------------------------
/**
        @paramms Máxima velocidad que puede alcanzar el personaje al correr en m/s.
*/
inline void nlstPlayer::SetMaxRunSpeed(float ms) {
        this->max_speed=ms * SPACECOORD_SCALE;
}
//------------------------------------------------------------------------
/**
        @paramac        Aceleración del personaje al correr en m/s^2.
*/
inline void nlstPlayer::SetRunAccel(float ac) {
        this->run_accel=ac * SPACECOORD_SCALE ;
}
//------------------------------------------------------------------------
/**
        @return Velocidad de giro en grados/s.
*/
inline float nlstPlayer:: GetRotSpeed() {
        return rot_speed;
}
//------------------------------------------------------------------------
/**
        @return Máxima velocidad al correr en m/s.
*/
```

```
inline  float nIstPlayer::GetMaxRunSpeed() {
        return max_speed;
}
//-----------------------------------------------------------------------
/**
        @return  Aceleración al correr en m/s^2.
*/
inline float nIstPlayer::GetRunAccel() {
        return run_accel;
}
//-----------------------------------------------------------------------
/**
        @parampath      Ruta en la jerarquía donde está el nodo que contiene al modelo.
*/
inline void nIstPlayer::SetN3DNodeTar(const char *path) {
        n_assert(path);
        splayernode=path;
        this->playernode =(n3DNode *) kernelServer->Lookup(path);
}
//-----------------------------------------------------------------------
/**
        @parampath      Ruta en la jerarquía donde está el modelo.

*/
inline
void nIstPlayer::SetNCalModTar(const char *path) {
        n_assert(path);
        smodel=path;

        this->model = (nCal3DModel *) kernelServer->Lookup(path);
}
//-----------------------------------------------------------------------
/**
        @parampath      Ruta en la jerarquía donde está el servidor de colisiones.
*/
inline void nIstPlayer::SetCollServ(const char *path)
{
        n_assert(path);
        this->ref_collide=path;
}
//-----------------------------------------------------------------------
/**
        @return Ruta en la jerarquía donde está el servidor de colisiones.
*/
inline const char * nIstPlayer::GetCollServ() const
{
        return ref_collide.getname();
}
//-----------------------------------------------------------------------
/**
*/
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::StartRR() {
        isRR=true;
        if( model )
                model->NewCycle(3,0.5F,0.2F);
}
```

```
//------------------------------------------------------------------------
inline void nlstPlayer::StartRL() {
	isRL=true;
	if( model )
		model->NewCycle(3,0.5F,0.2F);
}
//------------------------------------------------------------------------
inline void nlstPlayer::StopFW() {
	isFW=false;
	isSTPFW=true;
	if( model && !isBW && !isRR && !isRL )
		model->NewCycle(4,0.5F,0.2F);			//Agregar la animación de alto
}
//------------------------------------------------------------------------
inline void nlstPlayer::StopBW() {
	isBW=false;
	isSTPBW=true;

	if( model && !isFW && !isRR && !isRL)
		model->NewCycle(4,0.5F,0.2F);		//Agregar la animación de alto
}
//------------------------------------------------------------------------
inline void nlstPlayer::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 nlstPlayer::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 nlstPlayer::ActShoot() {
	if( model )
		model->Action(2,0.1F,0.2F);
}
//------------------------------------------------------------------------
/**
	Este método normalmente es llamado desde los métodos de colisión de los objetos con que choca.
	@param  addv   Vector que contiene el impulso que será aplicado al objeto.
*/
inline void nlstPlayer::AddImpulse(vector3 addv)
{
	velVector+=addv;
}
//------------------------------------------------------------------------
#endif
```

3.2. nistplayer_main.cc

```
#define N_IMPLEMENTS nIstPlayer
//===========================================================================
// ist/nistplayer_main
// 2002 Enrique 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()

{
   // servidores
          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";

          //estado
          isFW=false;
          isBW=false;
          isRL=false;
          isRR=false;
          isSTPFW=false;
          isSTPBW=false;

          max_speed = 5.0F * SPACECOORD_SCALE;                         //en m/s
          run_accel = 2.0F * SPACECOORD_SCALE;                      //en m/s^2
          rot_speed = 60.0F;                                     //en grados/s

}
//---------------------------------------------------------------------------
nIstPlayer::~nIstPlayer()
{
}
//---------------------------------------------------------------------------
/**

         En esta clase en particular, este método mantiene las animaciones y el movimiento del personaje.
         @param  dt       Delta de tiempo desde la vez anterior en la cual se llamó a este método.
*/
void
nIstPlayer::Trigger(float dt)
{
         //Actualiza la posición de colisión.
```

```
collideObject->Transform(0.0F, playernode->GetM() );
nlstEntity::Trigger(dt);   //fuerzas y gravedad
//Se encarga del estado
if(isRR)
{
        if( !isSTPFW || !isSTPBW)
        {
                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);
        }
}

if(isRL)
{
        if( !isSTPFW || !isSTPBW)
        {
                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);
        }
}

if (isFW)
{
        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)                    //comienza a correr
        {
                model->BlendCycle(1,0.5F,0.2F);
                animWalk=false;
        }
        else if(velVector.len() <= this->max_speed/2  && !animWalk)  //comienza a caminar
        {
                model->BlendCycle(4,0.5F,0.2F);
                animWalk=true;
        }
}
if(isBW)
{
        vector3 r;
        matrix44 m;

        m= playernode->GetM();

        r = m.y_component();
        r.norm();

        //integrating
        r *= run_accel;

        accForVector += r*mass;
}
if(isSTPFW)
{
        vector3 r(velVector.x,0.0,velVector.z);

        if(r.len()>4)                          //valor dependiente de la escala 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); //limpia de caminar
				model->ClearCycle(1,0.2F);		//limpia de correr
				if(isRR || isRL)
					model->NewCycle(3,0.5F,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)
			{
				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); //limpia de caminar
				model->ClearCycle(1,0.2F);		//limpia de correr
								if(isRR || isRL)
					model->NewCycle(3,0.5F,0.2F);
				else
					model->NewCycle(0,0.5F,0.2F);

				isSTPBW=false;
			}
		}

}
//-------------------------------------------------------------------------
/**
	En esta clase en particular, este método se encarga de la física de colisión del personaje.
	Por ejemplo, que este no atraviese las paredes.
*/
void nIstPlayer::Collide(void)
{
		if (NULL == collideObject)
	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 )
                    {
                            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 );
                            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");
                    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
                    {
                            vector3 impulse=other_contact_normal*( -1.0F*comp );
                            this->velVector+=impulse;
                    }

            } //end else if niststruct
        } //end for
}       //end nistplayer collide
//---------------------------------------------------------------------------
/**
        La posición se obtiene de integrar el vector de aceleraión y luego del de velocidad.

        @param  dt        Delta de tiempo desde la vez anteriror en la cual se llamó a este método.
*/
void nIstPlayer::UpdatePosition(float dt)
{
        this->forVector+=accForVector;
        this->accForVector.set(0.0F,0.0F,0.0F);                         //limpiando
        this->aclVector=this->forVector*invMass;                //Obteniendo la aceleración.
        vector3 velXZComp(velVector.x,0.0,velVector.z);
        if(velXZComp.len() > this->max_speed )                  // horizontal max vel
        {
                velXZComp.norm();
                velXZComp*=max_speed;
                this->velVector.set(velXZComp.x,velVector.y,velXZComp.z);
        }
        //integrando
        this->velVector+=aclVector*dt;  //v1=v0+dv
        vector3 p;
        playernode->GetT(p.x,p.y,p.z);
        //integrando
        p+=this->velVector*dt;                  //X1=X0+dX
        //fija la nueva posición.
        playernode->Txyz(p.x,p.y,p.z);
}
```

4. Archivos de nIstWorld.
4.1. nistworld.h

```
//==========================================================================
// IST World Class
// 23-10-2002 Enrique Larios
//--------------------------------------------------------------------------

#ifndef N_ISTWORLD_H
#define N_ISTWORLD_H
//--------------------------------------------------------------------------
/**
   @class nIstWorld

   @brief Clase encargada de contener a las entidades de la historia.

   La responsabilidad de esta clase es la de contener y actualizar a todas las entidades del
          mundo, además también  contie el contexto de colisión en el que están todas ellas.

   Enrique Larios

*/

// includes
#ifndef N_ROOT_H
#include "kernel/nroot.h"
#endif

#ifndef N_STRING_H
#include "kernel/nstring.h"
#endif

#ifndef N_AUTOREF_H
#include "kernel/nautoref.h"
#endif

#ifndef N_MATRIX_H
#include "mathlib/matrix.h"
#endif

#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 "node/nmeshnode.h"
#include <string.h>

#include "ist/nIstEntity.h"
#include "ist/nIstPlayer.h"

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

#define INIT_GRAV_VAL -9.81F
#define SPACECOORD_SCALE 45.0F

//--------------------------------------------------------------------------
class n3DNode;
class nChannelServer;
class nCollideContext;
class nCollideObject;
class nCollideServer;
class nCollideShape;
class nIstEntity;
```

```
class nGfxServer;
//---------------------------------------------------------------------------
class N_PUBLIC nIstWorld : public nRoot
{
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;
        nCollideContext* collideContext;
        // time
        float last_time;
        float current_time;
        float dt;
        int step_number;
        vector3 gravity;
        virtual void UpdateEntities();
public:
        /// Método constructor.
    nIstWorld();
        /// Método destructor.
    virtual ~nIstWorld();
        /// Apuntador a nKernelServer
    static nKernelServer* kernelServer;
        /// Actualiza el estado del objeto.
        virtual void Trigger(float t);

        /// Método encargado de la persitencia, salva en un archivo de comandos.
        virtual bool SaveCmds(nFileServer* fileServer);

        /// Regresa la gravedad del mundo.
        vector3 GetGravity(void);
        /// Fija el valor de la gravedad del mundo.
        void SetGravity(vector3 g);

        /// Regresa el contexto de colisión.
    nCollideContext* GetCollideContext();
        /// Regresa el servidor de colisiones.
    nCollideServer* GetCollideServer();
        /// Crea una nueva forma de colisión.
    nCollideShape* NewCollideShape(const char* name, const char* file);
        /// Crea un nuevo objeto de colisión.
    nCollideObject* NewCollideObject(nIstEntity* entity);
        /// Agrega una estructura BSP al servidor de colisiones.
        void AddCollideBSPNode(const char *pathBSPNode,const char *map_dir);
};
//---------------------------------------------------------------------------
/**
        @return Regresa una referencia al servidor de colisiones.
*/
inline nCollideServer* nIstWorld::GetCollideServer()
{
    return ref_collide.get();
}
//---------------------------------------------------------------------------
/**
        @return Regresa una referencia al contexto de colisiones.
*/
inline nCollideContext* nIstWorld::GetCollideContext()
{
    return collideContext;
}
//---------------------------------------------------------------------------
/**
        @return Vector de gravedad del mundo en m/s^2
*/
inline vector3 nIstWorld::GetGravity(void)
{
        return this->gravity;
}
```

```
//--------------------------------------------------------------------------
/**
        @param Vector de gravedad del mundo en m/s^2
*/
inline void nIstWorld::SetGravity(vector3 g)
{
        this->gravity=g*SPACECOORD_SCALE;
}
//--------------------------------------------------------------------------
#endif
```

4.2. nistworld_main.cc

```
#define N_IMPLEMENTS nIstWorld
//=============================================================================
// ist/nIstWorld_main.cc
// 23-10-2002 Enrique Larios
//--------------------------------------------------------------------------

// includes
#include "ist/nIstWorld.h"
#include "ist/nIstEntity.h"

#include "kernel/nenv.h"
#include "kernel/ntimeserver.h"
#include "node/n3dnode.h"

#include "collide/ncollideobject.h"
#include "collide/ncollideserver.h"
#include "gfx/nchannelserver.h"

nNebulaScriptClass(nIstWorld, "nroot");

//--------------------------------------------------------------------------
nIstWorld::nIstWorld()
      : ref_gs(kernelServer,this),
        ref_is(kernelServer,this),
        ref_ss(kernelServer,this),
        ref_sg(kernelServer,this),
        ref_con(kernelServer,this),
        ref_collide(kernelServer,this),
                        gravity(0.0F,INIT_GRAV_VAL*SPACECOORD_SCALE,0.0F)
{
  // servidores
        this->ref_gs   = "/sys/servers/gfx";
  this->ref_is   = "/sys/servers/input";
  this->ref_ss   = "/sys/servers/script";
  this->ref_sg   = "/sys/servers/sgraph";
  this->ref_con  = "/sys/servers/console";
        this->ref_collide="/sys/servers/collide";
        this->collideContext = ref_collide->NewContext();
        current_time = 0;
        last_time = 0;
        step_number=10;
}

//--------------------------------------------------------------------------

nIstWorld::~nIstWorld()
{
}
//--------------------------------------------------------------------------
/**
-       En esta clase en particular, este método actualiza al mundo y todas las entidades que contiene.
        Mediante el step_number se puede definir en cuantos pasos se hace la actualización.
        @param  t       Tiempo desde el inicio de ejecución del programa.
*/
void nIstWorld::Trigger(float t)
{
        last_time = current_time;
        current_time = t;
```

```
        float step_time=current_time-last_time;
        dt=step_time/(float)step_number;
        for(int i=0;i<step_number;i++)
                UpdateEntities();
}
//------------------------------------------------------------------------
void nlstWorld::UpdateEntities(void)
{
        nlstEntity* entity;             //auxiliar
        for (entity = (nlstEntity*)this->GetHead(); entity != NULL; entity = (nlstEntity*)entity->GetSucc())
                entity->Trigger(dt);

        if (collideContext->Collide() != NULL)
     for (entity = (nlstEntity*)this->GetHead(); entity != NULL; entity = (nlstEntity*)entity->GetSucc())
        entity->Collide();

        for (entity = (nlstEntity*)this->GetHead(); entity != NULL; entity = (nlstEntity*)entity->GetSucc())
                entity->UpdatePosition(dt);
}
//Métodos de colisión------------------------------------------------------------------------
/**
  Crea una nueva forma de colisión. Particulamente llamado por nEntity::SetCollideShape()
        para agregar una geometría a una forma.

        @paramname      Nombre con el que se conocerá a la nueva forma.
        @paramfile      Nombre del archivo que contiene la geometría de la forma de colisión.
*/
nCollideShape* nlstWorld::NewCollideShape(const char* name, const char* file)
{
  nCollideShape* shape = ref_collide->NewShape(name);
  if (false == shape->IsInitialized())
     shape->Load((nFileServer2*)kernelServer->Lookup("/sys/servers/file2"), file);
  return shape;
}
//------------------------------------------------------------------------
/**
  Crea un objeto de colisión de la clase y forma dada.
  Es un método que usualmente sólo es llamado por nlstEntity.

        @param Entidad relacionada con ese objeto de colisión.
*/
nCollideObject* nlstWorld::NewCollideObject(nlstEntity* entity)
{
  nCollideObject* object = collideContext->NewObject();
  collideContext->AddObject(object);
  object->SetClientData(entity);

  return object;
}

//------------------------------------------------------------------------
/**
        @param pathBSPNode Ruta del nodo donde estan las estructuras.
        @param map_dirDirectorio donde estan los archivos que contienen la geometría de las estructuras.
*/
void nlstWorld::AddCollideBSPNode(const char *pathBSPNode,const char *map_dir )
{
        nRoot *bspNode;
        char buffy[100];
        n_assert(pathBSPNode);
        bspNode = (nRoot *) kernelServer->Lookup(pathBSPNode);

  n_printf("Entering  reading bsp meshes in:\n");
        n_printf("%s\n",pathBSPNode);

        nRoot* auxobj = NULL;
        char id;

  for (auxobj = (nRoot*)bspNode->GetHead(),id='a'; auxobj != NULL; auxobj = (nRoot*)auxobj->GetSucc())
  {
                strcpy(buffy,auxobj->GetName() );
```

```
                buffy[4]=NULL;
                if( strcmp(buffy,"mesh") ==0 )
                {
                        strcpy(buffy,pathBSPNode);

                        int end=strlen(buffy);
                        if(buffy[end-1]!='/') {
                                buffy[end]='/';
                                buffy[end+1]=NULL;
                        }

                        strcat(buffy,auxobj->GetName());
                        strcat(buffy,"/mesh");
                        n_assert(buffy); //borra después de probar
                        nMeshNode *mesh= (nMeshNode *) kernelServer->Lookup(buffy);

                        nCollideObject* object = collideContext->NewObject();
                        collideContext->AddObject(object);
                        char auxBuffy[20]="/world/iststruct";
                        auxBuffy[15]=id++;
                        auxBuffy[16]=NULL;

                        nRoot* iststructobject = kernelServer->New("niststruct",auxBuffy);
                        // hack
                        object->SetClientData( iststructobject );
                        object->SetCollClass( ref_collide.get()->QueryCollClass("nistcollstruct") );

                        char buff[10], *aux;
                        strcpy(buff,mesh->GetFilename() );

                        for(aux=buff;*aux!='.' && *aux!=NULL ;aux++);
                        if(*aux=='.')
                                *aux=NULL;

                        nCollideShape* shape = ref_collide->NewShape(buff);

                        if ( shape->IsInitialized() == false )
                        {
                                char nf[30];
                        nFileServer2 *fs=(nFileServer2*) kernelServer->Lookup("/sys/servers/file2");
                                strcpy(nf,map_dir);
                                int en=strlen(nf);
                                if(nf[en-1]!='/') {
                                        nf[en]='/';
                                        nf[en+1]=NULL;
                                }

                                strcat(nf,mesh->GetFilename());
                          shape->Load(fs,nf);
                        }
                        object->SetShape(shape);
                        matrix44 *mat_coll= new matrix44();
                        mat_coll->ident();
                        //gira -90 grados en eje x.
                        mat_coll->rotate_x(-1.570796326F);

                        object->Transform(0.0,*mat_coll);
                } //end if  mesh objects
        } //end for   mesh objects
}
```

Bibliografía.

[CRA02] Crawford, Chris. Artists and engineers as cats and dogs: implications for interactive storytelling. Computer Graphics, 2002; (26)1.

[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.

[GAR88] De Asís Garrote, María Dolores. Formas de Comunicación en la Narrativa. Ed. Fundamentos, Madrid España, 1988, pp. 18-86.

[GEI03] Geist. http://www.zgdv.de/zgdv/departments/z5/Z5Projects/Geist_1/index_html_en

[IZA03] Información Tocable. http://www.zgdv.de/zgdv/departments/z5/Z5Presse/Presse_2002_02/

[LAW01] Lawrence, Deborah. Social Dynamincs of storytelling: Implications for story base design. 2001.

[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).

[PER97] Perlin, K. Layered compositing of facial expression. ACM SIGGRAPH Conference, Los Angeles, 1997.

[PRE97] Pressman, Roger. Ingeniería del software: Un enfoque práctico, McGraw Hill, 4ta ed., México, 1997, pp. 367-424.

[PRO58] Propp, Vladimir. Morphology of the folktale, International Journal of American Linguistics, 24(4), 1958.

[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

[SKO02] Skov, Mikael. Designing interactive narrative systems: is object-orientation useful?, Computer Graphics, 2002; (26)1.

[SZI99] Szilas, Nicolas. Interactive drama on computer: beyond linear narrative. In: AAAI Fall Symposium. Menlo Park, CA: AAAI Press 1999. pp 150-156.

[ULR02] Spierling, Ulrike. Setting the scene: playing digital director in a interactive storytelling and creation. Computer Graphics, 2002; (26)1.

[WEH97] Wehling, Jasón. Aproveche las noches con Java, Prentice Hall, México, 1997, pp. 2-18.

[WOO96] Woo, Mason. OpenGL programming guide: the official guide to learning OpenGL, Addison Wesley, 2da ed., EE.UU., 1996, pp. 2-26.