Universidad Nacional Autónoma de México

Posgrado en Ciencia e Ingeniería de la Computación

Entorno de ejecución de guiones en ambientes virtuales

T E S I S

que para obtener el grado de:

Maestra en Ciencias
(Computación)

P R E S E N T A

Martha Angélica Nakayama Cervantes

Director: Dr. Jesús Savage Carmona

Ciudad Universitaria, México D.F., 2009

Resumen

Los recientes avances tecnológicos han aumentado el desarrollo de aplicaciones con Ambientes Virtuales, siendo la Realidad Aumentada una de las áreas más explotadas. La Realidad aumentada es una tecnología que permite sobreponer objetos virtuales en el mundo real.

Con esta tecnología se han creado los llamados "Libros aumentados"los cuales son libros comunes pero con marcas especiales. Utilizando técnicas de visión computacional, estas marcas son detectadas usando una cámara y con ellas se generan gráficos 3D en tiempo real sobre el video, permitiendo la mezcla entre la realidad virtual y el mundo real.

Este tipo de aplicaciones son muy útiles en la educación ya que puede ayudar enormemente al proceso de aprendizaje. Sin embargo, no es muy conocida ni utilizada en México dado que existen pocas herramientas y equipo necesario para crear y ejecutar aplicaciones de Realidad Aumentada y las pocas que existen son difíciles de utilizar o requieren bastos conocimientos técnicos.

El objetivo de este trabajo es poner al alcance de más personas ésta tecnología, simplificando la creación de aplicaciones de libros aumentados para que personas con poco conocimiento técnico puedan crearlos sin lidiar con las complicaciones de entender completamente la tecnología y así dedicar más tiempo al diseño de contenidos.

Para tales fines, se desarrolló un Framework con el cual se pueden crear libros aumentados con sólo crear un guión (documento XML), en el cual se especifican personajes, escenarios, comportamientos, secuencias de acciones, etc. Se desarrolló en lenguaje C/C++, utilizando herramientas como ARToolkit para el manejo de la realidad aumentada, XPAT para XML, y OpenGL para los modelos gráficos.

Los resultados mostraron ser buenos ya que el tiempo y conocimientos requeridos fueron disminuidos significativamente, además de que se tiene la posibilidad extender el Framework por desarrolladores con un poco más de experiencia.

Índice general

Índice de figuras

Introducción

Desde siempre los seres humanos se han preocupado en inventar herramientas que les ayuden a mejorar su percepción del mundo y aumentar sus capacidades físicas y hoy en día, gracias a la tecnología, se han logrado avances que antes eran incluso difíciles de imaginar.

Estos avances se pueden observar en campos como la *ciencia, la medicina, la educación, la industria (comercio, publicidad, entretenimiento, etc).* Y actualmente debido al auge que ha tenido la computación en todas estas áreas, es común encontrar software que facilita diversas tareas simplificando el trabajo y mejorando los resultados.

Ahora es común ver aplicaciones computacionales que involucran ambientes virtuales, por lo que su desarrollo e investigación ha ido ganando terreno y popularidad. Hay que tener presente que la estimulación visual es muy importante para los usuarios, ya que que para ellos es más atractivo el uso de interfaces de fácil manejo e interacción; dando como resultado que las aplicaciones gráficas sean de las más solicitadas.

La **realidad aumentada** consiste en una combinación de *objetos virtuales* y ambientes reales generando un efecto de convivencia entre los dos mundos de manera natural. Entre las aplicaciones que utilizan las bondades de esta tecnología, se puede mencionar al *desarrollo de juegos interactivos y visuales*, destinados tanto al entretenimiento como a la pedagogía. Sin embargo, estos conocimientos no solamente se

pueden aplicar en la industria de los videojuegos si no también se puede pensar en aplicaciones médicas, educativas y algunas otras.

En el campo de la *pedagógia* por ejemplo, si un niño pequeño que aún no sabe leer tuviese un libro, el cual además de contener el texto de la historia, pudiera mostrar con gráficos tridimensionales la escena que se esta describiendo; para el niño esta experiencia sería más divertida y educativa que un libro convencional ya que incluso no necesitaría la participación directa de un adulto en la lectura de la historia [DH07].

Se debe tener en cuenta que una adecuada técnica de enseñanza en conjunto con esta tecnología, aportaría grandes beneficios por ejemplo si el niño en cuestión estuviera aprendiendo a leer, ya que podría facilitársele esta tarea.

"El verdadero aprendizaje es experimental", los seres humanos aprendemos mejor al hacer que sólo al leer o escuchar clases. Entre más sentidos son involucrados *(vista, oído, tacto, etc.)* más poderosa es la experiencia del aprendizaje. Por ejemplo, en lugar de leer sobre la guerra de independencia en un libro de texto, se podrían observar las batallas, las conversaciones o las acciones que se llevaron a cabo en ese momento, como si le estuvieran representando una obra teatral [Ada04] .

Descripción del problema

Es fácil imaginar que la creación de dichos *libros aumentados* no es una tarea sencilla ya que se requiere de varias disciplinas como *el diseño, la didáctica, el arte gráfico y el desarrollo técnico* entre otras cosas.

Sin embargo, **son pocas las herramientas con las que se cuenta para el desarrollo de este tipo de sistemas** y las pocas que existen no siempre pueden ser utilizadas por personas sin muchos conocimientos técnicos de programación, por lo que los artistas y diseñadores, se ven limitados en el uso de esta tecnología [Bil02] .

Para poder brindar a estos y a muchas otras personas en la misma situación la oportunidad de acercarse y beneficiarse de las tecnologías de realidad aumentada en aplicaciones de libros aumentados, **se pretende hacer una herramienta que permita la creación de dichos libros de una manera mas sencilla.** Para lograr esto se desarrolló un *entorno de ejecución* que tiene como entrada un *guión o script* que representa la escena de la historia así como los personajes o actores, sus acciones y en general todo el flujo de la misma y genera como salida la historia en un contexto de realidad aumentada.

Objetivo

El objetivo de este trabajo es construir un *entorno de ejecución* con el cual las personas interesadas en hacer *libros con realidad aumentada* puedan interactuar fácilmente y obtengan los resultados deseados en un tiempo menor al que les llevaría aprender la tecnología y aplicarla, además de no necesitar *amplios conocimientos técnicos o de programación*, ya que dicho entorno recibirá por parte del usuario un guión o script y con la información contenida en él generará una representación gráfica de las escenas deseadas asignándolas a un *libro aumentado*.

Con esto los usuarios podrán dedicar más tiempo en el modelado de los personajes, las escenas, el desarrollo de la historia y no necesitarán lidiar con los contratiempos de aprender a programarlo usando realidad aumentada y todo lo que implica.

Contribución y relevancia

La tecnología de realidad aumentada ha sido utilizada para desarrollar diversos tipos de aplicaciones educacionales y de entretenimiento. Los *libros aumentados* han llamado la atención no solo de investigadores sino de educadores como un medio de *mejorar los libros* con visualización y simulación interactiva, animación, gráficos en 3-D y sonidos. Tales características añadidas a un libro pueden *mejorar el aprendizaje* por medio de la exploración activa y la manipulación del medio [DH07].

El uso de estas herramientas y métodos de presentación conducen a un mejor entendimiento de procesos dinámicos complejos o estructuras 3-D y **superar las limitaciones de los medios de educación convencionales**. En lugar de solo visualizar e interactuar con el contenido de un libro en una computadora, un libro hecho con la tecnología de realidad aumentada permite la integración de interacción tangible, lo cual mejora el aprendizaje y la colaboración entre usuarios.

El aprendizaje y la comprensión son apoyados por la interacción, la actitud autodidacta, la exploración y la colaboración. **"La experiencia es el mejor maestro"**, sin embargo los estudiantes rara vez tienen la oportunidad de experimentar directamente lo que están aprendiendo. Al incorporar nuevos medios a la educación se puede mejorar el aprendizaje y la experiencia de la lectura (Fig. 1).

Figura 1: Niños interactuando con un libro aumentado [DH07]

Desafortunadamente en **México** esta tecnología no ha sido muy difundida aún, lo cual en parte se debe a que no existen muchas herramientas para la creación de estas aplicaciones y las pocas que existen han sido desarrolladas en otros países por lo que muchas veces es difícil tener acceso a ellas, además de que no siempre son fáciles de utilizar por personas con poco conocimiento técnico.

Con el desarrollo del *entorno de ejecución de guiones*, se logrará acercar a más personas a esta tecnología, además de que al contar con herramientas que faciliten a los desarrolladores de libros aumentados la creación de éstos, se podrá dedicar más tiempo y enfocarse más al diseño de los contenidos, modo de interacción y técnicas de didáctica aplicada, y se podrán llevar sus ideas a la práctica de una manera más sencilla sin tener que preocuparse en cómo aplicar la tecnología de realidad aumentada. Con esto se pretende aumentar la creación de aplicaciones de libros aumentados y poner al alcance de más personas el uso de tecnologías de realidad aumentada para la educación y estimulación temprana en nuestro país.

Organización de la tesis

- **Revisión de antecedentes y técnicas:** Conocer los conceptos básicos y las herramientas existentes para el desarrollo de aplicaciones de realidad aumentada, las técnicas para diseñar este tipo de aplicaciones y en general las características que debe tener un entorno enfocado a aplicaciones de realidad aumentada.
- **Análisis y selección de tecnologías:** Con base en lo investigado, se realizó un análisis del problema y la solución a desarrollar, así como también se seleccionaron las tecnologías para llevar a cabo la programación, el modo de almacenamiento de información, la lógica de funcionamiento y las herramientas a utilizar adecuadas para la creación del entorno.
- **Diseño:** Una vez teniendo las tecnologías a utilizar, se diseñó una metodología de desarrollo, así como el flujo de la información y la interacción final con las aplicaciones que podrán ser desarrolladas con este entorno.
- **Desarrollo:** Se llevó a cabo la programación e implementación con base en el análisis y el diseño realizados anteriormente, procurando pulir los detalles que pudieran haberse obviado en las etapas anteriores.
- **Desarrollo de un demo sobre el entorno:** Para poder cerciorarse de que el entorno cumple con el objetivo deseado, se desarrolló una aplicación tipo **demo** con la que se mostraron las características más importantes y se comprobó su utilidad en el caso de estudio de los libros aumentados.
- **Conclusiones y trabajo futuro:** Finalmente se corroboró si se cumplieron los objetivos planteados incialmente, se dan conclusiones y se mencionan los cambios o modificaciones que se pueden realizar para mejorar y ampliar el entorno.

Capítulo 1

Estado del Arte, Graficación y Realidad Aumentada

Con el auge que ha ganado en los últimos años la *realidad aumentada*, día con día surgen nuevas ideas para utilizar esta tecnología en todos los campos del conocimiento ya que a pesar de no tener muchos años de existir, ya cuenta con muchos seguidores.

Se puede pensar en todo tipo de aplicaciones, todas muy interesantes, no sólo en las áreas relacionadas con la computación, sino también en otras como *la médicina, la construcción, la industria, la educación, el arte,* por nombrar algunas.

Sin embargo en nuestro país aún no es tan popular dado que no se cuenta con muchas herramientas que faciliten el trabajo para realizar aplicaciones usando realidad aumentada y las que existen en la actualidad aún requieren que los usuarios tengan *bastos conocimientos de programación, graficación por computadora y visión computacional*, además de ser especialistas en el campo para el cual quieren desarrollar su aplicación.

Es por eso, que surgió la idea de desarrollar un **entorno de ejecución de guiones en realidad aumentada**, para ayudar a las personas que desean involucrarse en la creación de *libros aumentados* (una de las muchas aplicaciones de esta tecnología), teniendo como objetivo simplificar su tarea a solamente crear los contenidos de dichos libros sin tener que aprender a fondo como funciona la realidad aumentada, es decir, para aprovechar al máximo las habilidades de los *diseñadores, creadores de ambientes, escritores y artistas.*

Con esto se pretende aportar nuevas herramientas para la creación de libros aumentados, los cuales pueden contribuir enormemente al aprendizaje de niños en edad escolar o incluso a adultos, además de poner al alcance de más personas esta tecnología con el objetivo de comenzar a difundirla más en nuestro país.

1.1. Antecedentes

En los últimos años la *realidad aumentada* ha pasado a ser una disciplina de investigación versátil y madura [Led04]. La realidad aumentada es una **mezcla de realidad y ficción**, es decir, los elementos del mundo real conviven con los elementos virtuales, los cuales sirven para aportar información acerca de los primeros.

Se trata de una tecnología totalmente innovadora, ligada a la realidad virtual aunque diferente en varios aspectos, ya que en la **realidad virtual** el usuario **no** puede ver el mundo real a su alrededor si no que lo simula reemplazando todo lo que hay a su alrededor y la realidad aumentada complementa la visión real del usuario, sin reemplazarla [Azu97].

La realidad virtual pretende la inmersión del usuario en un mundo donde todo aquello que percibe ha sido generado por computadora, se encuentra en un mundo distinto, aislado del mundo real, rodeado de objetos que no existen pero puede interactuar con ellos.

En cambio, la **realidad aumentada** no pretende aislar al usuario del mundo real, sino **complementar** éste mediante objetos e imágenes generadas por computadora. Así pues, la realidad aumentada consiste en *añadir gráficos virtuales o información en tiempo real al campo de visión de una persona*. Supone una inmersión por parte del usuario en un mundo que une los objetos del mundo real con los objetos animados del mundo virtual (Fig. 1.1).

Figura 1.1: Diagrama de Realidad Mezclada donde se observa la relación de la Realidad Aumentada con el Mundo Virtual y el Real

Aunque el entorno de la realidad aumentada existe desde hace una década aproximadamente, apenas en los últimos años es que su progreso se ha hecho más evidente. Pero, *¿Por qué es interesante la realidad aumentada? ¿Para que puede ser útil combinar la realidad con objetos virtuales en 3-D?*

La realidad aumentada mejora la percepción y la interacción del usuario con el mundo real. Los objetos virtuales muestran información que el usuario no puede detectar directamente con sus propios sentidos. La información transmitida por dichos objetos ayuda al usuario a realizar tareas en el mundo real [KD07].

1.2. Estado del Arte

La realidad aumentada avanza y perfecciona la interacción del usuario con los objetos virtuales y la realidad mostrando información que no esta disponible de manera común. Por esta razón, actualmente el uso de sistemas de realidad aumentada en diversas áreas está aumentando, por ejemplo:

Comunicación espacial para sistemas espaciales autónomos [GRSS07].

Es decir, comunicación entre humanos y sistemas espaciales autónomos como *robots, satélites de observación etc.*

Este enfoque del diálogo espacial proporciona mejores medios de comunicación entre sistemas espaciales autónomos y humanos al proporcionar una base espacial común para tareas conjuntas con un modelo espacial y un sistema de

diálogo espacial que brinda una manera de ver y clarificar tareas interactivamente durante etapas de ejecución (Fig. 1.2).

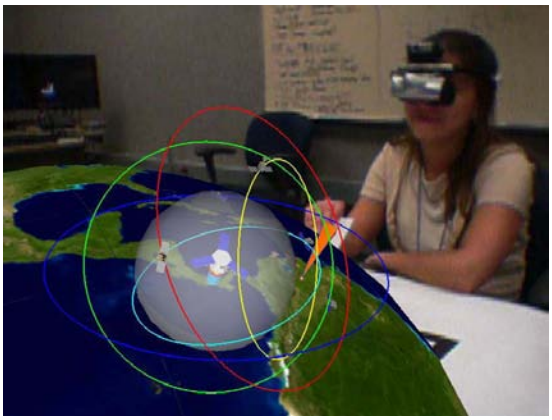

Figura 1.2: Comunicación espacial entre humanos y sistemas espaciales autónomos utilizando un sistema de Realidad Aumentada [GRSS07]

Edición de mallas en 3-D utilizando teléfonos celulares [HB07].

Esto es, usando el enfoque de realidad aumentada se ha desarrollado un sistema que permite al usuario seleccionar uno o más vértices en una malla poligonal de tamaño arbitrario y trasladarlos o rotarlos libremente al trasladar y rotar el dispositivo (Fig. 1.3).

Figura 1.3: Edición de varios vértices en una malla 3-D usando un teléfono celular [HB07]

Diseño Urbano con interfaces tangibles [SS05]

Se han creado interfaces basadas en realidad aumentada que permiten a los diseñadores trabajar en un ambiente tanto real como virtual, teniendo así diversas vistas y una mejor percepción espacial, además de poder trabajar varias personas a la vez (Fig. 1.4).

Figura 1.4: Estudiantes trabajando en un modelo generado con Realidad Aumentada [SS05]

AR Tennis [HBO06]

Un juego de tenis en teléfonos celulares el cual aprovecha la cámara del teléfono para convertirlo en un dispositivo de realidad aumentada y así mostrar objetos virtuales sobre el video y crear la ilusión de estar jugando en una pequeña cancha de tenis donde el teléfono actúa como raqueta para golpear la pelota, además esta aplicación se puede jugar entre dos personas conectando por bluetooth sus teléfonos (Fig. 1.5).

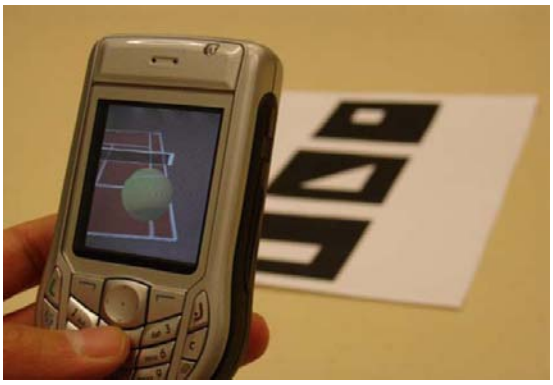

Figura 1.5: Cancha de Tenis virtual utilizando marcas de Realidad Aumentada para calcular posiciones [HBO06]

Mixed Reality Book [GDSB07]

También llamados libros aumentados, los cuales son libros tradicionales de papel pero mejorados digitalmente añadiéndoles gráficos en 3-D e incluso sonidos, con los cuales el usuario puede interactuar de distintas maneras, ya sea al mover el libro, cambiando de página o utilizando cubos con marcas con los que puede poner o quitar información a la página (Fig. 1.6).

Figura 1.6: Libro aumentado, escena con gráficos del océano agregado con realidad aumentada [GDSB07]

1.3. Graficación por computadora

Debido a que el presente proyecto involucra temas como la *graficación por computadora* es necesario entender esta diciplina así como conocer las herramientas con las que se cuenta para el desarrollo.

Las interfaces gráficas han sustituido a las interfaces textuales como el medio habitual para la interacción usuario-computadora. Los gráficos también se han convertido en una tecnología fundamental para comunicar ideas, datos y tendencias en la mayoría de las áreas (*comerciales*, *científicas*, *de ingeniería*, *educativas* y otras). Con los gráficos podemos **crear mundos artificiales o virtuales** (Fig. 1.7) constituyendo cada uno de ellos un área de exploración para el estudio de objetos y fenómenos en una forma natural e intuitiva que aproveche nuestras altamente desarrolladas habilidades de reconocimiento de patrones visuales [FVDFH97].

Figura 1.7: Escena generada con gráficos por computadora

Hasta finales de la década de 1980, muchas de las aplicaciones de graficación por computadora tenían que ver con objetos *bidimensionales*; las aplicaciones *tridimensionales* eran relativamente raras, ya que el software de este tipo es más complejo que el bidimensional y porque se requiere mucho poder de cálculo para producir imágenes más realistas. Es por esto que por mucho tiempo la interacción en tiempo real entre el usuario y los modelos tridimensionales e imágenes realistas sólo era factible en costosas estaciones de trabajo de alto rendimiento que usaba hardware gráfico de propósito especial [FVDFH97].

Gracias al *rápido avance tecnológico* en los últimos años, con el mejoramiento de los microprocesadores, la disminución del costo de la memoria de las computadoras y la evolución de las tarjetas gráficas, el desarrollo de interfaces de cómputo basadas en gráficos ha aumentado considerablemente, así como también se han hecho espectaculares avances en cuanto a las *animaciones tridimensionales en tiempo real* (Fig. 1.8), dado que ya no es necesario contar con equipo especializado y costoso para tener este tipo de aplicaciones, ya que el software y hardware gráfico necesario ya es más accesible.

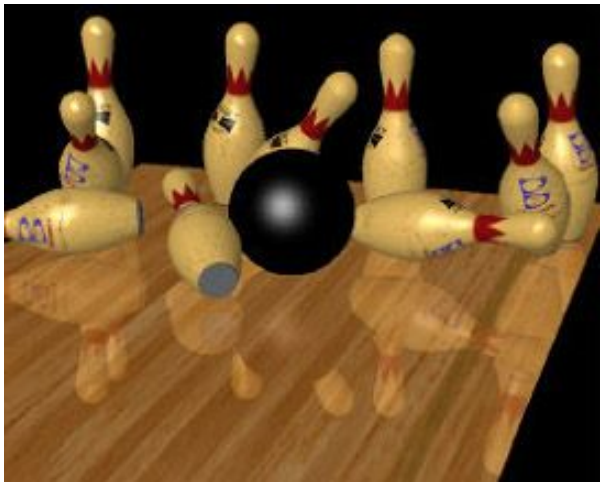

Figura 1.8: Gráficos en 3D generados por computadora

La graficación por computadora es la rama de las ciencias de la computación que se encarga del estudio, diseño y trabajo del *despliegue de imágenes* en la pantalla de una computadora a través de las herramientas proporcionadas por *la física, la óptica, la térmica, la geometría* y otras disciplinas.

Al observar esta definición se puede notar que el *campo de la graficación por computadora* es bastante complejo, esto debido a que muchas disciplinas están involucradas. Para poder lograr representaciones realistas en una computadora, lo primero que se necesita es *tratar* de comprender como sucede lo que se quiere representar en el mundo real. Por ejemplo, se debe entender los principios básicos de como interactúa la luz con los objetos y para esto se requieren *cálculos matemáticos* y algunos conceptos de *física*.

Para poder representar escenas del **mundo real** usando graficación por computa-

dora, es necesario contar con conocimientos de muchas disciplinas como *la geometría, el algebra vectorial, la física, la optica*, por nombrar algunas. Todo esto debido a que la computación gráfica se compone de muchos conceptos importantes sin los cuales no podríamos llegar a tener escenas realistas o animaciones. Algunos de los temas involucrados son:

- Puntos, líneas, planos
- Sistemas de coordenadas, vectores, matrices
- Transformaciones geométricas
- Estructuras de datos
- Algoritmos
- Modelos de iluminación y sombreado
- Modelado de objetos bidimensionales y tridimensionales
- Modelos de color
- Visión (percepción humana)
- Modelado jerárquico
- Texturas
- Procesamiento de imágenes
- Representación de cámaras virtuales
- Animación basada en física
- Rendering en tiempo real

Entre otros, además de hacer uso de conceptos y conocimientos de computación y programación.

1.3.1. Proceso Gráfico

Para tener una idea un poco más clara del proceso de *creación de gráficos* por computadora se necesita comprender el proceso que se sigue desde que se tiene una especificación de objetos hasta su despliegue en pantalla (existen otros procesos pero el que se muestra a continuación es el más básico).

Como se observa en el siguiente diagrama (Fig. 1.9), el proceso para generar gráficos por computadora funciona de la siguiente manera:

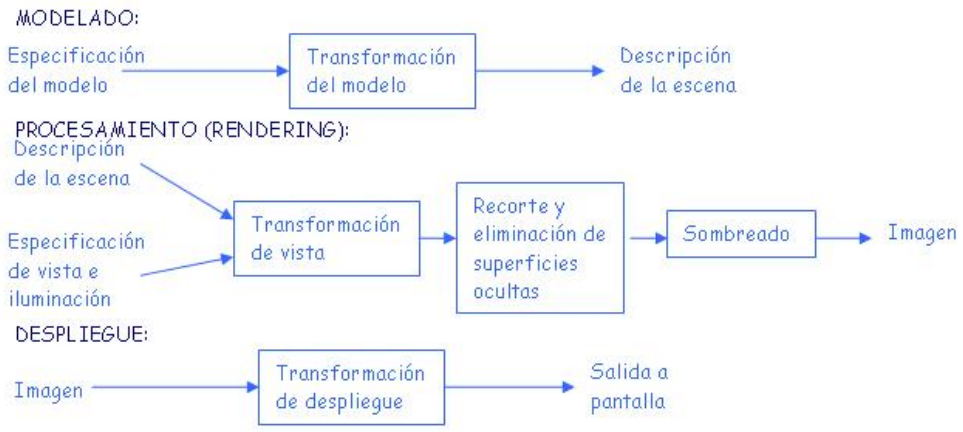

Figura 1.9: Proceso Gráfico básico

Se comienza con la etapa de **modelado**, en la cual se tiene una *especificación del objeto o modelo* a representar, esta especificación no es más que datos provenientes de alguna fuente, ya sea un sistema médico, un artista o cualquier otro origen de datos. Esta información será posteriormente usada para *crear una imagen* (**Rendering**) la cual será finalmente **desplegada** en la pantalla.

Es decir, en el proceso de generacion de imágenes realistas, existen 3 etapas principales: *modelado, procesamiento (rendering) y despliegue*[MC06].

La etapa de *modelado* crea una representación interna de los objetos en la escena, la etapa de *rendering* convierte la descripción de la escena en una imagen de la escena y la etapa de *despliegue* muestra la imagen resultante en el dispositivo de salida o pantalla.

Esto puede ser considerado como una analogía con la vida real, por ejemplo: "la luz proviene de muchas fuentes ambientales o artificiales y refleja en la superficie de los objetos en diferentes direcciones, dependiendo de la posición, orientación y material del que están hechos dichos objetos. Estos reflejos llegan a los ojos donde son convertidos en señales que el sistema visual y el cerebro procesan y finalmente interpretan."

1.4. Programación gráfica con OpenGL

Cada computadora, sistema operativo y procesador de gráficos tiene su propio conjunto de comandos para controlar la salida en la pantalla de la computadora. Tiempo atrás esto hacía que los programas de gráficos no fueran muy portables, frecuentemente los programadores desarrollaban *su propio conjunto de rutinas* que representaban las *operaciones básicas de graficación* y ocultaban dentro de estas rutinas los detalles de implementación particular del *sistema operativo*.

Cuando los programas eran usados en un nuevo sistema, un programador tenía que actualizar esas rutinas. Con el tiempo, los programadores se dieron cuenta de que contar con un **conjunto de rutinas estándar** para todas las plataformas ayudaría enormemente a los programadores de gráficos, así como también ayudaría a la implementación de técnicas de graficación por computadora. Desde entonces se han desarrollado *diversos estándares*, entre los cuales se encuentra **OpenGL** (uno de los más populares).

OpenGL (Open Graphics Library) es una **especificación estándar** que define una *Interfaz de Programación de Aplicación (API)* multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. La interfaz consiste en **más de 250 funciones** diferentes que pueden usarse para generar escenas tridimensionales complejas a partir de *primitivas geométricas* simples, especificar objetos y operaciones necesarias para producir aplicaciones interactivas (Fig. 1.10).

OpenGL es una interfaz **independiente del hardware** que puede ser implementada en varias plataformas distintas. Para lograr esto no se tienen comandos relacionados con tareas de ventanas ni para obtener datos de entrada del usuario, para tal propósito se utilizará el sistema de manejo de ventanas adecuado para cada plataforma [SW04].

Figura 1.10: Escena generada con OpenGL

Del mismo modo, OpenGL no proporciona comandos de alto nivel para describir modelos de objetos tridimensionales. Con OpenGL se pueden construir los modelos partiendo de un pequeño conjunto de primitivas geométricas como *puntos, líneas y polígonos.*

Existe una biblioteca más elaborada que proporciona algunas de estas utilidades que pueden ser construidas sobre OpenGL; **GLU (OpenGL Utility Library)** proporciona muchas herramientas de modelado como superficies cuadráticas y curvas y superficies NURBS. *GLU es una parte estándar* de las implementaciones de OpenGL.

Para lidiar con el manejo de ventanas e interacción con el usuario existen una serie de bibliotecas para cada sistema operativo, pero una de las más populares e independiente del sistema operativo es **GLUT (OpenGL Utility Toolkit)**. Utilizando esta biblioteca se puede lograr que las aplicaciones sean más portables entre plataformas.

Las principales operaciones que OpenGL realiza para generar imágenes en la pantalla son básicamente:

- Construir formas con primitivas geométricas, utilizando descripciones matemáticas de los objetos (OpenGL considera puntos, líneas, poligonos, imágenes y mapas de bits como primitivas).
- Arreglar los objetos en el espacio tridimensional y aplicar el punto de vista del observador (composición de la escena).
- Calcular los colores finales de todos los objetos. Los colores deben ser explícitamente asigandos por la aplicación, determinados por las condiciones de iluminación especificadas, obtenidos al ponerle una textura a los objetos o con la combinación de estas acciones.
- Convertir las descripciones matemáticas de los objetos y sus colores asociados en información de pixels en la pantalla (Rasterización).

Durante estas etapas OpenGL puede realizar otras operaciones como eliminar partes de los objetos que estan ocultas por otros objetos. Además después de que cada escena es rasterizada pero antes de que sea dibujada en la pantalla se pueden realizar algunas operaciones en los datos de pixel si se desea.

1.5. Realidad Aumentada

Hasta ahora se ha hablado sobre *realidad aumentada* y sus aplicaciones, pero dado que el presente proyecto esta involucrado con esta tecnología, es necesario comprender *¿qué es?, ¿cómo funciona?, ¿cuál es la diferencia con la realidad virtual? y ¿con qué herramientas se cuenta para poder generarla?.*

La **Realidad Aumentada** consiste en *añadir gráficos virtuales*, en tiempo real, al *campo de visión* de una persona. Supone una inmersión por parte del usuario en un mundo que resulta la unión entre el mundo real y el mundo virtual.

La diferencia entre la realidad aumentada y la más popular realidad virtual es que ésta última pretende la inmersión del usuario en un mundo totalmente virtual, donde todo aquello que percibe visualmente ha sido generado por computadora.

El usuario se encuentra en un mundo distinto, prácticamente aislado del mundo real, rodeado de objetos virtuales que no existen en la realidad, pero puede interactuar con ellos como si de verdad existieran (Fig. 1.11).

Figura 1.11: Ejemplo de una aplicación de Realidad Virtual

En cambio, **la realidad aumentada** no pretende aislar al usuario del mundo real, sino *complementar* éste mediante objetos virtuales e imágenes generadas por computadora. El usuario se encuentra inmerso en un mundo que tiene a la vez elementos *virtuales* y elementos *reales* con los que puede *interactuar* (Fig.1.12).

Algunos investigadores definen a la realidad aumentada de manera que se requiere el uso de *Head Mounted Displays (HMD)* pero para no limitar la realidad aumentada a una tecnología específica se puede considerar que los sistemas de realidad aumentada son aquellos que cumplen con las siguientes características [Azu97]:

- Combinan el mundo real y el mundo virtual.
- Son interactivos en tiempo real.
- Se registran en 3 dimensiones.

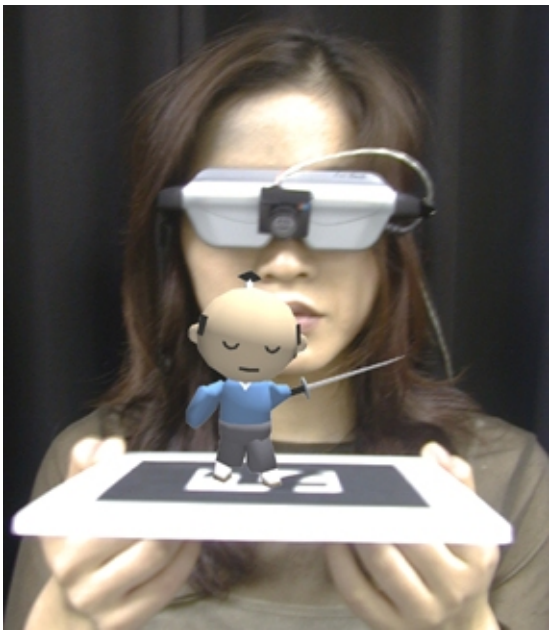

Figura 1.12: Ejemplo de realidad aumentada

Entonces, un sistema de realidad aumentada necesita contar con un *medio de captura de imágenes* de mundo real (como por ejemplo una cámara de video), una máquina capaz de *crear imágenes virtuales* y *procesar las imágenes reales* añadiéndoles esta información (un procesador y un software específico para esto) y un *medio para desplegar* la imagen resultante (una pantalla).

La Realidad Aumentada ofrece infinidad de nuevas posibilidades de interacción, que hacen que esté presente en muchos y varios ámbitos, como son *la arquitectura, el entretenimiento, la educación, el arte, la medicina*, por nombrar algunos. Por ejemplo:

Proyectos educativos. Actualmente la mayoría de aplicaciones de Realidad Aumentada para proyectos educativos se usan en museos, exhibiciones, parques de atracciones temáticos, etc (Fig. 1.13).

Entretenimiento. Teniendo en cuenta que la industria de los juegos es un mercado que mueve unos 30,000 millones de dólares al año en los Estados Unidos, es comprensible que se esté apostando mucho por la Realidad Aumentada en este campo puesto que ésta puede aportar muchas nuevas posibilidades a la manera de jugar (Fig. 1.14 y 1.15).

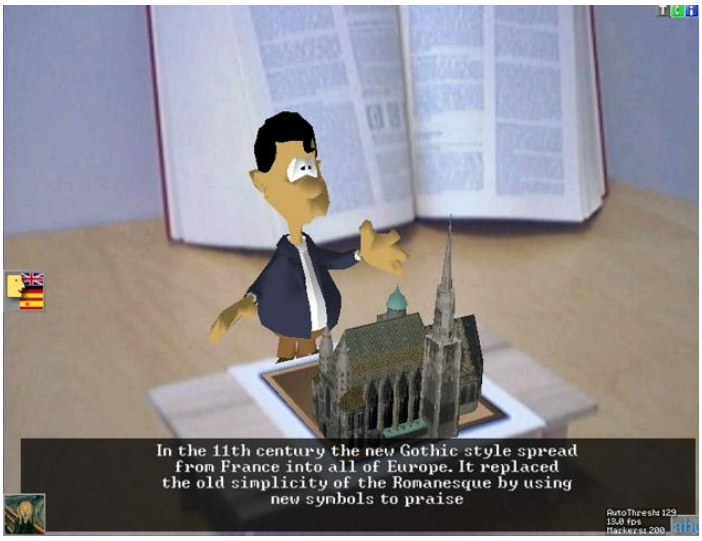

Figura 1.13: Personaje virtual en un libro de historia del arte [WBS06]

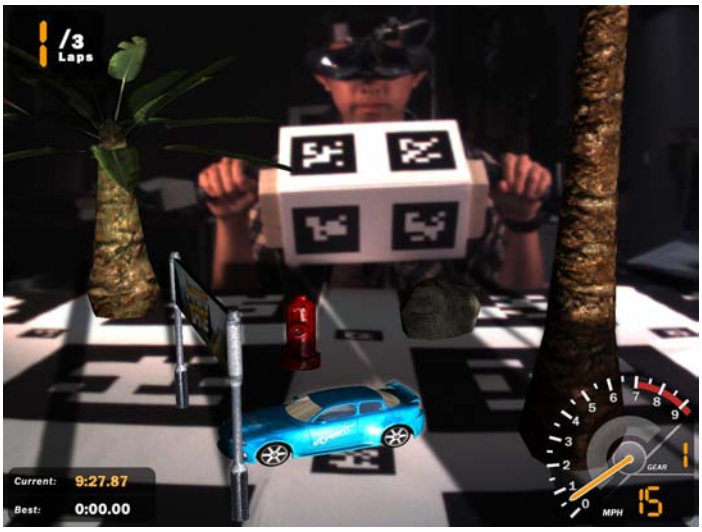

Figura 1.14: Videojuego con realidad aumentada

Figura 1.15: Videojuego de PlayStation 3 con Realidad Aumentada

Servicios de emergencias y militares. En caso de emergencia la Realidad Aumentada puede servir para mostrar instrucciones de evacuación de un lugar. En el campo militar, puede mostrar mapas, localización de los enemigos, o cualquier otra información útil (Fig. 1.16).

Figura 1.16: Instrucciones en Realidad Aumentada

Arquitectura. La Realidad Aumentada es muy útil a la hora de resucitar virtualmente edificios históricos destruidos, así como proyectos de construcción que todavía están bajo plano (Fig. 1.17).

Figura 1.17: Utilización de realidad aumentada en la Arquitectura

Medicina. Los médicos pueden utilizar Realidad Aumentada como ayuda en la visualización y entrenamiento para cirugía, para asistir en tratamientos y procedimientos o incluso para la enseñanza de la medicina (Fig. 1.18).

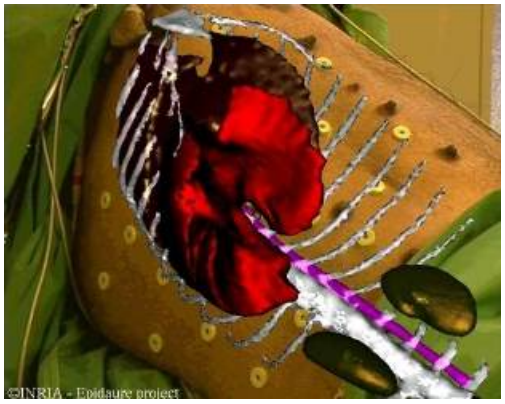

Figura 1.18: Cirugia con realidad aumentada

A pesar de que el entorno de la Realidad Aumentada existe **desde hace una década** aproximadamente, apenas en los últimos años su progreso se ha hecho más evidente, debido a que el costo del hardware y software necesario para este tipo de aplicaciones ha disminuido y ahora es más accesible y común.

1.5.1. Head Mounted Displays

En aplicaciones de Realidad Aumentada generalmente se utilizan *HMDs (Head Mounted Displays)* como dispositivos de visualización, aunque actualmente también se pueden utilizar otros dispositivos como *cámaras de video*. Los HMDs son dispositivos que se usan *sobre la cabeza* y cuentan con un dispositivo óptico pequeño colocado frente a uno o ambos ojos. Un HMD típico cuenta con uno o dos pantallas pequeñas con lentes y espejos semitransparentes montados en el casco, lentes o visor. Los HMDs permiten al usuario **ver una señal de video generada por computadora** (Fig. 1.19).

Figura 1.19: Algunos tipos de Head Mounted Display (HMDs)

Muchos investigadores están apostando también a la utilización de pequeños dispositivos móviles como interfaz de los sistemas de Realidad Aumentada. *Teléfonos móviles con cámaras y Asistentes Personales Digitales (PDAs)* ya están siendo probados para esta tarea (Fig. 1.20).

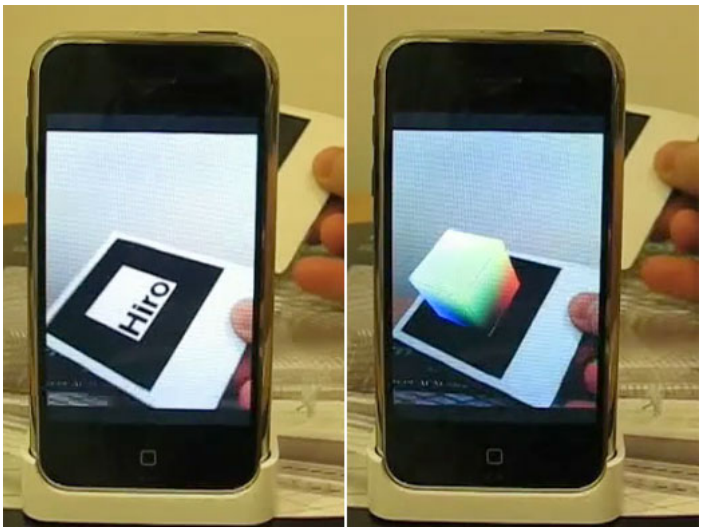

Figura 1.20: Realidad Aumentada en Iphone

1.6. ARToolkit

ARToolkit es un *conjunto de bibliotecas* para la creación de aplicaciones en **C/C++** de realidad aumentada la cual fué desarrollada en 1999 por **Hirokazo Kato y Mark Billinghurst** y que actualmente se encuentra en su versión 2.72.1, liberada en febrero de 2007 [KB07].

Estas librerías proporcionan una serie de funciones para la captura de vídeo y para la *búsqueda de ciertos patrones* en las imágenes capturadas, mediante técnicas de visión por computadora.

Un punto importante en las aplicaciones de realidad aumentada es el hecho de calcular el punto de vista de la cámara para así poder realizar las operaciones necesarias sobre los objetos virtuales y lograr que estos *se integren correctamente en el mundo real.*

ARToolKit utiliza algunas técnicas de *visión por computadora* para calcular la posición de la cámara y la orientación relativa a una plantilla, lo cual permite al desarrollador sobreponer *objetos virtuales* sobre las plantillas.

Es decir, si se desea mostrar objetos virtuales de modo que el usuario realmente se crea que existen en el mundo real, se deben realizar transformaciones sobre esos objetos de modo que el usuario los vea (a través de la cámara o dispositivo de captura

utilizado) en la posición, tamaño, orientación e iluminación en que esos objetos serían percibidos por el usuario en el mundo real en caso de que *realmente* estuvieran ahí.

Para esto se utilizan unas plantillas de forma cuadrada, que se componen de *un cuadrado color negro con otro cuadrado blanco en su interior con área cuatro veces más pequeña y en el centro de éste un dibujo sencillo* (Fig. 1.21). La aplicación será capaz de detectar estas plantillas en las imágenes de vídeo capturadas utilizando las funciones y herramientas proporcionadas por ARToolKit.

Figura 1.21: Ejemplo de plantillas de ARToolkit

Una vez *detectada* una plantilla en una imagen, estudiando la *orientación, posición y tamaño* de la plantilla, la aplicación es capaz de *calcular* la posición y orientación relativa de la cámara respecto a la plantilla, y usando esta información podrá pasar a *generar el objeto* correspondiente sobre la imagen capturada mediante librerías externas a ARToolKit (por ejemplo GLUT y OpenGL), de modo que el objeto aparezca sobre la plantilla en la posición, orientación y tamaño correspondiente al punto de vista de la cámara (Fig.1.22).

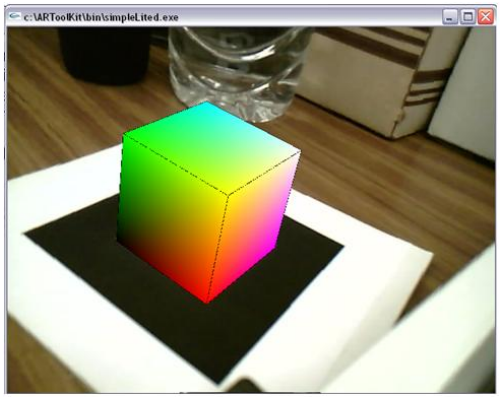

Figura 1.22: Ejemplo de una aplicación con ARToolkit

1.6.1. Principios de ARToolkit

El *funcionamiento básico* de una aplicación de ARToolKit es el siguiente:

- Primero se **captura el video** mediante la cámara y se manda a la computadora.
- El video se convierte en una serie de imágenes tomadas a intervalos de tiempo muy pequeños
- A continuación cada imagen se **convierte en una imagen binaria**, esto se hace utilizando un umbral (threshold), de forma que los píxeles cuya intensidad supere este valor son transformados en píxeles de color negro. El resto se transforman en píxeles blancos.
- **Se buscan y encuentran todos los cuadrados** negros (como los de la plantilla) existentes en la imagen (en realidad al umbralizar la imagen del cuadrado exterior aparece blanco y el cuadrado interior aparece negro) (Fig. 1.23).

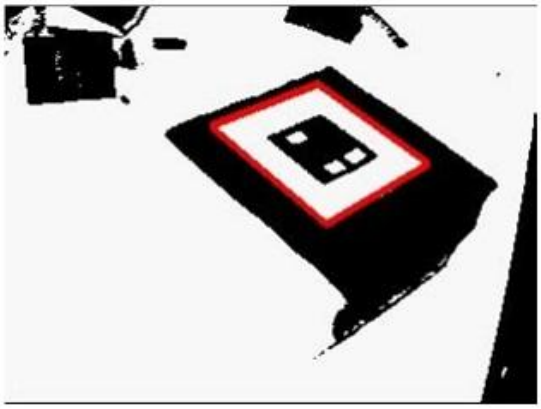

Figura 1.23: Imagen binaria con una plantilla detectada

- **Se compara el interior del cuadrado** con plantillas que fueron previaente almacenadas y configuradas en la aplicación.
- Si la forma de la plantilla analizada y la plantilla almacenada coincide, **se utiliza la información de tamaño y orientación** de la plantilla almacenada para compararla con la plantilla que se ha detectado y así poder calcular la posición y orientación relativas de la cámara a la plantilla y se guarda la matriz de transformación asociada.

- Se utiliza esta matriz para **establecer la posición y orientación de la cámara virtual** (la cámara que se maneja en graficación), lo que equivale a una transformación de las coordenadas del objeto a dibujar (transformación de vista).
- Al haber puesto la cámara virtual en la misma posición y orientación que la cámara real, **el objeto virtual se dibuja sobre la plantilla** y se muestra la imagen resultante que contiene la imagen del mundo real y el objeto virtual superpuesto, alineado sobre la plantilla.
- Se realiza el mismo proceso con el resto de las imágenes en el video.

El siguiente diagrama muestra el funcionamiento que se acaba de describir (Fig. 1.24):

Figura 1.24: Funcionamiento del proceso de incorporar un objeto virtual en una escena real a través de patrones con ARToolKit

1.6.2. Limitaciones

Existen algunas *limitaciones* en los sistemas de Realidad Aumentada (RA) basados en visión computacional.

Naturalmente los objetos virtuales *sólo aparecen cuando las plantillas son visibles*, esto puede limitar el tamaño, resolución o movimiento de los objetos virtuales además de que si el usuario *cubre una parte de la plantilla* con las manos u otra cosa, el objeto virtual desaparece.

Entonces, se debe tener cuidado que el video contenga las plantillas completas para un mejor desempeño de la aplicación.

También se debe considerar que entre *más grande* físicamente sea la plantilla, mayor será la distancia a la que esta puede ser detectada. Además el rango de detección puede verse afectado por la *complejidad del patrón* de la plantilla, entre *más sencillo* sea éste mejor, los patrones con mayores regiones blancas y negras son los más efectivos. Otros factores importantes que pueden afectar el desempeño son la *orientación relativa de la cámara* con respecto a la plantilla y las *condiciones de iluminación*.

1.6.3. La infraestructura de ARToolkit

ARToolKit es un conjunto de **herramientas** (*ToolKit*) para desarrollo de software, al igual que OpenGL éste contiene una serie de funciones predefinidas. En una aplicación de realidad aumentada, dichas funciones son llamadas en un orden específico. Sin embargo, también es posible utilizar las diferentes herramientas del ToolKit por separado.

ARToolKit utiliza **OpenGL** para la parte del despliegue de gráficos (*Rendering*), **GLUT** para el manejo de *Eventos/Ventanas* y bibliotecas de video dependientes del hardware y un API estándar para cada plataforma.

1.6.4. Estructura

ARToolKit se compone de 4 módulos:

- **Módulo de RA:** módulo base con rutinas de seguimiento, calibración y una colección de parámetros.

- **Módulo de video:** colección de rutinas para captura de video. Son envolventes (Wrapper) de las rutinas de captura de video del SDK (Software Development Kit) estándar.
- **Módulo Gsub:** colección de bibliotecas de gráficos basadas en OpenGL y GLUT.
- **Módulo Gsub Lite:** módulo de reemplazo de Gsub con una colección de bibliotecas de gráficos más eficientes, independientes del sistema de ventanas.

A continuación se muestran las estructuras jerárquicas de ARToolKit y su relación con las bibliotecas de las cuales depende (Fig. 1.25 y 1.26).

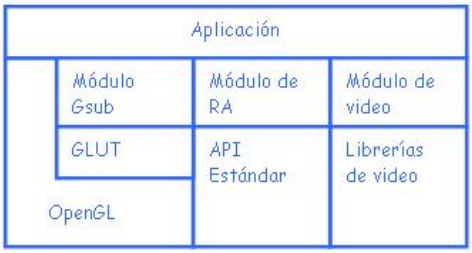

Figura 1.25: Estructura jerárquica de ARToolKit usando el módulo Gsub

Figura 1.26: Estructura jerárquica de ARToolKit usando el módulo Gsub Lite

Capítulo 2

Framework, Guiones e Inteligencia Artificial

Existen otros conceptos importantes que es necesario establecer para la mejor comprensión del presente proyecto, por ejemplo, se debe comprender que un **Framework** tiene como finalidad *facilitar el desarrollo de software,* por lo tanto lo que interesa construir es un framework que, en este caso, será útil para crear **libros aumentados**. Además, como se desea que los usuarios de este framework no requieran muchos *conocimientos técnicos*, es necesario encontrar la manera de simplificarles la tarea de pasar de una historia a un lenguaje que la computadora pueda entender.

También es necesario poder realizar **planeación de acciones y movimientos** de los personajes dentro de la escena para darle un poco más de *realismo e interactividad* a la historia, para esto se debe contar con técnicas de planeación y por supuesto con **herramientas** que ayuden a la aplicación de todos los conceptos antes mencionados.

2.1. Framework

En el desarrollo de software, un **framework** es una estructura de soporte definida en la cual otro proyecto de software puede ser organizado y desarrollado. Típicamente, un framework puede incluir soporte de *programas, bibliotecas y un lenguaje de guión* para ayudar a desarrollar y unir los diferentes componentes de un proyecto.

Un framework representa una *arquitectura de software* que modela las relaciones generales de las entidades del dominio. Provee una **estructura y una metodología de trabajo** la cual extiende o utiliza las aplicaciones del dominio.

Los Frameworks son diseñados con el intento de facilitar el desarrollo de software, permitiendo a los diseñadores y programadores pasar más tiempo identificando requisitos de software que tratando con los tediosos detalles de bajo nivel para proveer un sistema funcional.

Fuera de las aplicaciones en la informática, un framework puede ser considerado como el *conjunto de procesos y tecnologías* usados para resolver un problema complejo. **Es el esqueleto sobre el cual varios objetos son integrados para una solución dada**.

2.2. Lenguajes de Guión

Un **script** es un **guión** o conjunto de instrucciones. Permiten la automatización de tareas creando pequeñas utilerías. Son ejecutados por un *intérprete* y usualmente son archivos de texto. **Un lenguaje de guiones** es un lenguaje de programación de *alto nivel* que es interpretado por otro programa **en tiempo de ejecución** en lugar de ser compilado como los lenguajes comunes. Los lenguajes de guiones representan un estilo de programación muy diferente a los lenguajes de programación comunes. Los lenguajes de guiones son diseñados para unir aplicaciones, al utilizarlos se logra el desarrollo de aplicaciones de manera *más rápida* que con los lenguajes convencionales [Ous98].

Los lengajes de guiones están diseñados para diferentes tareas que los lenguajes de programación de sistemas, por esta razón existen *diferencias fundamentales* entre ellos. Los lenguajes de programación fueron diseñados para *construir estructuras de datos, algoritmos, etc.* en contraste, los lenguajes de guiones fueron diseñados para *unir*, asumen la existencia de un conjunto de componentes poderosos y principalmente **conectan estos componentes**.

Los lenguajes de guiones y los lenguajes de programación **son complementarios** y la mayoría de las plataformas han contado con *ambos* desde los años 60. Estos lenguajes típicamente se se usan juntos en frameworks, donde los componentes son creados con lenguajes de programación y unidos con lenguajes de guiones.

Sin embargo las recientes tendencias en la tecnología, tal como máquinas más rápidas, mejores lenguajes de guiones, el aumento de la importancia de las interfaces gráficas y el crecimiento de las infraestructuras de comunicación tales como la internet, han aumentado la aplicabilidad de los lenguajes de guiones, estas tendencias continuarán con más y más aplicaciones escritas *completamente en lenguajes de guiones* y los lenguajes de programación serán usados principalmente para crear componentes.

2.3. XML

XML, siglas en inglés de **Extensible Markup Language** (lenguaje de marcas extensible), es un metalenguaje extensible de etiquetas desarrollado por el *World Wide Web Consortium (W3C)*. Proviene de un lenguaje inventado por IBM en los años setenta, llamado *GML (Generalized Markup Language)*, que surgió por la necesidad que tenía la empresa de almacenar grandes cantidades de información [GL95].

Este lenguaje gustó a la *ISO*, por lo que en 1986 trabajaron para normalizarlo, creando *SGML (Standard Generalized Markup Language)*, capaz de adaptarse a un gran abanico de problemas. A partir de él se han creado otros sistemas para almacenar información. En el año 1989 Tim Berners Lee creó *la Web*, y junto con ella el lenguaje *HTML*. Este lenguaje se definió en el marco de *SGML* y fue la aplicación más conocida de este estándar.

Los *navegadores Web* sin embargo siempre han puesto pocas exigencias al código HTML que interpretan y así las páginas Web son *caóticas* y no cumplen con la sintaxis. Estas páginas Web dependen fuertemente de una forma específica de lidiar con los errores y las ambigüedades, lo que hace a las páginas más frágiles y a los navegadores más complejos.

Otra *limitación* de SGML es que cada documento pertenece a un vocabulario fijo, establecido por la *DTD (Document Type Definition)*. No se pueden combinar elementos de diferentes vocabularios. Asimismo es imposible para un intérprete (por ejemplo un navegador) analizar el documento sin tener conocimiento de *su gramática* (la DTD).

Los navegadores resolvieron esto incluyendo lógica *ad hoc* para el HTML, en vez de incluir un analizador genérico. Ambas opciones, de todos modos, son muy complejas para los navegadores.

Por todo esto se buscó entonces definir un subconjunto del SGML que permitiera:

- **Mezclar elementos de diferentes lenguajes.** Es decir que los lenguajes sean extensibles.
- **La creación de analizadores simples**, sin ninguna lógica especial para cada lenguaje.
- Empezar de cero y hacer hincapié en que **no se acepte** nunca un documento con **errores de sintaxis**.

Para hacer esto XML deja de lado muchas características de SGML que estaban pensadas para facilitar la escritura manual de documentos. XML en cambio está *orientado a hacer las cosas más sencillas* para los programas automáticos que necesiten interpretar el documento. XML es una **simplificación y adaptación** del SGML y permite definir la gramática de lenguajes específicos. Por lo tanto XML no es realmente un lenguaje en particular, sino *una manera de definir lenguajes* para diferentes necesidades.

XML no ha nacido sólo para su aplicación en Internet, sino que se propone como un **estándar para el intercambio de información** estructurada entre diferentes plataformas. Se puede usar en *bases de datos, editores de texto, hojas de cálculo* y casi cualquier cosa imaginable.

XML es una tecnología sencilla que tiene a su alrededor otras que la complementan y la hacen mucho más grande y con unas posibilidades mucho mayores. Tiene un papel muy importante en la actualidad ya que permite la **compatibilidad entre sistemas para compartir la información** de una manera segura, fiable y fácil.

Ventajas del XML

- Es **extensible**, lo que quiere decir que una vez diseñado un lenguaje y puesto en producción, es posible extenderlo con la adición de nuevas etiquetas de manera de que los antiguos consumidores de la vieja versión todavía puedan entender el nuevo formato.
- El analizador es un componente **estándar**, no es necesario crear un analizador específico para cada lenguaje. Esto posibilita el empleo de uno de los tantos

disponibles. De esta manera se evitan bugs y se acelera el desarrollo de la aplicación.

- Si un tercero decide usar un documento creado en XML, es sencillo entender su estructura y procesarlo. Mejora la **compatibilidad** entre aplicaciones.

Estructura de un documento XML

La tecnología XML busca dar solución al problema de **expresar información estructurada** de la manera más *abstracta y reutilizable* posible. Que la información sea estructurada quiere decir que se compone de partes bien definidas y que esas partes se componen a su vez de otras partes. Entonces se tiene un árbol de fracciones de información. Estas partes se llaman **elementos** y se las señala mediante *etiquetas*. Formalmente un documento XML consiste en marcas o etiquetas y datos. Una *marca o etiqueta* es simplemente el texto que transmite *información sobre la estructura* lógica del documento, con información o datos que están siendo marcados.

En XML una marca se denota con el texto encerrado en paréntesis angulares (los símbolos de menor que y mayor que) y texto con la información.

Estructuralmente un documento XML tiene dos partes básicas: el **prólogo** y el **cuerpo** del documento. El prólogo proporciona la información general sobre el documento pero no representa la estructura real del documento. La otra parte principal son los datos y las marcas que componen el cuerpo del documento.

Documentos XML bien formados

Los documentos denominados como *"bien formados"* (del inglés *well formed*) son aquellos que cumplen con todas las definiciones básicas de formato y pueden, por lo tanto, ser analizados correctamente por cualquier *analizador sintáctico (parser)* que cumpla con la norma XML.

- Los documentos han de seguir una **estructura estrictamente jerárquica** con lo que respecta a las etiquetas que delimitan sus elementos. Una etiqueta debe estar correctamente incluida en otra, es decir, las etiquetas deben estar correctamente anidadas. Los elementos con contenido deben estar correctamente cerrados.

- Los documentos XML sólo permiten **un elemento raíz** del que todos los demás sean parte, es decir, solo pueden tener un elemento inicial.

- Los valores atributos en XML siempre deben estar **encerrados entre comillas** simples o dobles.
- El XML es **sensible a mayúsculas y minúsculas**. Existe un conjunto de caracteres llamados espacios en blanco (*espacios, tabuladores, retornos de carro, saltos de línea*) que los procesadores XML tratan de forma diferente en el marcado XML.
- Es necesario **asignar nombres** a las estructuras, tipos de elementos, entidades, elementos particulares, etc. En XML los nombres tienen alguna característica en común.
- Las construcciones como *etiquetas, referencias de entidad y declaraciones* se denominan **marcas**; son partes del documento que el procesador XML espera entender. El resto del documento entre marcas son los datos *"entendibles"* por las personas.

Partes de un documento XML

Un documento XML está formado por el prólogo y por el cuerpo del documento.

Prólogo

Aunque no es obligatorio, los documentos XML pueden empezar con unas líneas que describen la versión XML, el tipo de documento y otras cosas.

El prólogo contiene:

- **Una declaración XML.** Es la sentencia que declara al documento como un documento XML.
- **Una declaración de tipo de documento.** Enlaza el documento con su DTD, o la DTD puede estar incluida en la propia declaración o ambas cosas al mismo tiempo.
- **Uno o más comentarios** e instrucciones de procesamiento.

Cuerpo

A diferencia del prólogo, el cuerpo no es opcional en un documento XML, el cuerpo debe contener un único un elemento raíz, característica indispensable también para que el documento esté bien formado.

Elementos

Los elementos XML pueden tener contenido (más elementos, caracteres o ambos) o bien ser elementos vacíos.

Atributos

Los elementos pueden tener atributos que son una manera de incorporar características o propiedades a los elementos de un documento. Los atributos se encierran entre comillas.

Entidades predefinidas

Entidades para representar caracteres especiales para que, de esta forma, no sean interpretados como marcas en el procesador XML.

Secciones CDATA (Character Data)

Es una construcción en XML para especificar datos utilizando cualquier carácter sin que se interprete como marcado XML. Solo se utiliza en los atributos. (No confundir con #PCDATA que es para los elementos).

Comentarios

Comentarios a modo informativo para el programador los cuales son ignorados por el procesador.

Validez

Que un documento sea *"bien formado"* solamente habla de su estructura sintáctica básica, es decir, que se componga de elementos, atributos y comentarios como el estándar XML requiere que sean escritos. Ahora bien, cada aplicación de XML, es decir cada lenguaje definido con esta tecnología, necesitará especificar cuál es exactamente la relación que debe verificarse entre los distintos elementos presentes en el documento.

Esta relación entre elementos se especifica en un **documento externo o definición** (*DTD Document Type Definition o XSchema*). Crear una definición equivale a crear un nuevo lenguaje de marcado para una aplicación específica.

Document type definition (DTD)

La **DTD** define los tipos de *elementos, atributos y entidades permitidas* y puede expresar algunas limitaciones para combinarlos. Los documentos XML que se ajustan a su DTD se denominan *válidos*.

Declaraciones tipo elemento

Los elementos deben ajustarse a un tipo de documento declarado en una DTD para que el documento sea considerado como válido.

Modelos de contenido

Un modelo de contenido es un patrón que establece los sub-elementos aceptados y el orden en que se aceptan.

Declaraciones de lista de atributos

Los atributos se usan para añadir información adicional a los elementos de un documento.

Tipos de atributos

- Atributos CDATA (Character Data) y NMTOKEN (Name Token)
- Atributos enumerados y notaciones
- Atributos ID e IDREF

Declaración de entidades

XML hace referencia a objetos que no deben ser analizados sintácticamente según las reglas XML mediante el uso de entidades. Las entidades pueden ser:

- Internas o externas
- Analizadas o no analizadas
- Generales o parametrizadas

Espacios de nombres

Los espacios de nombres XML permiten separar semánticamente los elementos que forman un documento XML.

XML Schemas (XSD)

Un Schema es algo similar a una DTD, define qué elementos puede contener un documento XML, cómo están organizados y qué atributos y de qué tipo pueden tener sus elementos.

Herramientas para trabajar con documentos XML

Cualquier procesador, que sea capaz de producir archivos de texto simple es capaz de generar XML, aunque en los entornos de desarrollo (como *Eclipse o Visual Studio* y algunos otros) ésto se facilita, ya que reconocen los formatos y ayudan a generar un XML bien formado.

2.4. Planeación de acciones

Planear es una habilidad importante en los sistemas inteligentes, aumenta su autonomía y flexibilidad con la construcción de secuencias de acciones para lograr sus objetivos. Esto ha sido un área de investigación de la Inteligencia Artificial por más de tres decadas [Gha04].

Las técnicas de planeación han sido aplicadas a diversas áreas como *robotica, planeación de procesos, agentes autónomos* por nombrar algunas. La planeación involucra *la representación de acciones, modelos del mundo, razonamiento sobre los efectos de las acciones y técnicas para buscar eficientemente el espacio de los posibles planes.*

Un planeador típico toma tres entradas: una **descripción inicial** del estado del mundo, una **descripción del objetivo** deseado y un **conjunto de posibles acciones**. El planeador produce una secuencia de acciones que van desde el estado inicial al estado objetivo.

La planeación de acciones puede ser vista como una *búsqueda espacio estado*, en cada estado se cuenta con una representación de las condiciones actuales del mundo. Nuevos estados son producidos por reglas generales. **Una regla** está determinada por precondiciones las cuales activan a esta, una lista aditiva la cual indica los hechos o condiciones nuevas del sistema y una lista de borrado que elimina hechos o condicones viejas.

2.5. Dependencia conceptual

Creada en 1972 por *Roger Schank*, la dependencia conceptual (**DC**) es una teoría sobre la representación del significado del lenguaje natural. Originalmente fue desarrollada para representar la *adquisición del conocimiento* por medio del lenguaje natural [AGF95].

Los objetivos principales de la dependencia conceptual son:

- Facilitar la realización de inferencias a partir de frases
- Independencia del idioma y del uso

La dependencia conceptual tiene dos axiomas básicos:

- Para dos frases que son de significado idéntico, no importando el lenguaje, debe de haber solo una representación.
- Cualquier información que esté implícita en una frase, debe ser explícita en la representación del significado de la frase (Representación no ambigua).

A partir de estos axiomas, Schank construyó una teoría basada en un conjunto de primitivas conceptuales, las cuales se dividen en dos grupos: **activos y de estados**.

Primitivas conceptuales activas o Acciones:

Una ACCION es realizada por un ACTOR sobre un OBJETO con alguna DIRECCION

Primitivas conceptuales de estado o Estados:

Un OBJETO esta en un ESTADO con un VALOR

También existen relaciones entre estados, incluyendo ligas estáticas que son utilizadas para definir el significado de las frases y ligas causales las cuales son utilizadas para inferir nuevas DC.

Ligas Estáticas o Relaciones:

Una RELACION está definida entre una DC FUENTE y una DC DESTINO

Ligas Causales o Inferencias:

Una LIGA CAUSAL se define entre una DC FUENTE y una DC DESTINO.

ACCIONES PRIMITIVAS

Se supone que cualquier acción es reducible a una o más acciones primitivas:

Acciones físicas

PROPEL aplicar fuerza a un objeto físico (empujar)

MOVE mover una parte del cuerpo por su dueño (patear)

INGEST un objeto animado ingiere algo (comer)

EXPEL un objeto animado expulsa algo (llorar)

GRASP agarrar un objeto (empuñar)

Acciones que provocan cambios de estado

ATRANS transferencia abstracta, transferir una relación abstracta (dar)

PTRANS transferencia física, acción que cambia la posición de un objeto (ir)

Acciones instrumento

SPEAK producir sonido (hablar)

ATTEND focalizar un órgano sensorial hacia un estímulo (escuchar)

Acciones mentales

MTRANS transferencia de información mental (decir)

MBUILD construcción mental nueva a partir de información anterior (decidir)

ACCIONES PRIMITIVAS ACTs:

Nos sirven para expresar *"todos"* los verbos. Los distintos matices hay que representarlos mediante atributos. Además, hay que representar a los actores y objetos que participan en la acción junto con sus atributos

Categorías conceptuales

ACTs acciones (actions)

PPs objetos (picture producers)

AAs modificadores de acciones (action aiders)

PAs modificadores de objetos (picture aiders)

Estados

Los estados son un conjunto abierto que puede ser constantemente aumentado. Los estados comparten un formato común: un OBJETO en un ESTADO con algún VALOR.

Relaciones

Las primitivas se usan para definir relaciones de dependencia conceptual Describen estructuras semánticas mediante combinación de conceptos. Son reglas sintácticas conceptuales: constituyen una gramática de relaciones semánticas y pueden usarse para construir una representación interna de una frase.

2.6. Representación del conocimiento

¿Cómo es que los seres humanos organizan todo el conocimiento que deben tener para poder entender? ¿Cómo saben cual es el comportamiento adecuado para una situación particular?

Las personas saben como actuar apropiadamente debido a que tienen conocimientos sobre el mundo en el que viven. Pero, *¿cuál es la naturaleza de este conocimiento? ¿cómo está organizado? ¿cuándo es apropiado utilizarlo?*

Se pueden reconocer dos clases de conocimientos que las personas pueden utilizar durante el proceso de entendimiento: **el conocimiento general y el conocimiento específico** [Sch77].

El conocimiento general permite a una persona entender e interpretar las acciones de otra persona simplemente porque la otra persona es un ser humano con ciertas necesidades estándar que vive en un mundo que tiene ciertos métodos estándar para conseguir satisfacer esas necesidades. Por ejemplo, si alguien pide un vaso de agua, no es necesario preguntar por que lo quiere.

Se utiliza *el conocimiento específico* para interpretar y participar en eventos en los que se ha participado muchas veces. El conocimiento específico y detallado sobre una situación permite hacer menos procesamiento y cuestionarse menos sobre eventos que se experimentan frecuentemente. Por ejemplo, no es necesario preguntar por qué una persona desea ver el boleto de otra en la entrada de un cine. El conocimiento de situaciones específicas, como ir al cine, permite interpretar las acciones que las personas generalmente realizan en los cines.

En el caso de las historias, los lectores parecen no tener problemas al extraer fácilmente las características de la situación que el escritor desea describir. Por ejemplo:

«Una noche en Le Cafe mientras la mesera tomaba su orden , un hombre con aspecto de mafioso se acercó a Juan.»

En esta oración se establece el contexto de la escena en un restaurante, lo cual es un gran repositorio de conocimiento específico del mundo. Por lo tanto el lector no pone atención a pensar quién es *"la mesera"* o por qué Juan estaba hablando con ella, sin embargo, rápidamente nota que Juan probablemente esta en una mesa, en el acto de ordenar en un restaurante francés.

Tal conocimiento específico existe en detalle para las personas que han estado varias veces en un restaurante y están familiarizadas con la situación específica. El proceso de inferir dichos hechos, resulta en **una cadena de eventos causales conectados**, esta cadena de eventos causales es útil para representar cualquier flujo de eventos secuenciales. Dado que ciertas secuencias de eventos frecuentemente ocurren en un orden específico las personas deben desarrollar mecanismos especiales para lidiar con

ellos. Esto es, existen ciertos agrupamientos de cadenas causales que son considerados como unidades conceptuales.

Es necesario poder distinguir **una secuencia de eventos específicos**, es decir **un script**, de eventos aislados. Un script es una estructura que describe una secuencia apropiadad de eventos en un contexto particular. Aunque es posible entender una historia sin usar un script, los scripts son una parte importante para la comprensión de la misma. Lo que hacen es permitir obviar los detalles tediosos cuando se está hablando o escribiendo y llenar estos huecos en la historia mientras se escucha o se lee.

2.7. Guiones o Scripts

Son una extensión de las DCs, las cuales representan sucesos aislados y son un mecanismo para representar conocimiento sobre secuencias habituales de sucesos que permite anticipar una sucesión de situaciones.

Un guión es una estructura que describe *una secuencia estereotípica de sucesos en un contexto particular.* Es una técnica basada en estudios psicológicos o de patrones de comportamiento. Existen evidencias de que los seres humanos organizan cierto tipo de conocimiento en estructuras que se corresponden con situaciones típicas [Sch77].

También resuelven ambigüedades en función del contexto particular de la historia que leen o de la conversación que mantienen. Los sucesos tienden a ocurrir en secuencias o patrones que se repiten porque existen relaciones causales entre los sucesos. Los sucesos descritos por un guión forman *una cadena causal.*

Principio de la cadena: Se refiere al conjunto de condiciones de entrada que permiten que los primeros sucesos del guión tengan lugar.

Final de la cadena: Es el conjunto de resultados que pueden permitir que otros sucesos o secuencias de sucesos (por ejemplo otro guión) tengan lugar.

Parte intermedia: Son sucesos conectados con sucesos previos que los hacen posibles y con sucesos posteriores que posibilitan.

2.7.1. Componentes de un guión

Condiciones de entrada (precondiciones) Condiciones que deben cumplirse, en general, para que puedan tener lugar los sucesos representados en el guión. Por ejemplo: *restaurante abierto, cliente hambriento*

Resultados (postcondiciones) Son condiciones que, en general, se cumplirán después de que tengan lugar los sucesos del guión. Por ejemplo: *cliente lleno y con menos dinero, dueño del restaurante con más dinero.*

Props Representan objetos que intervienen en los sucesos del guión. Aunque no aparezcan pueden ser inferidos. Por ejemplo: se supone que un restaurante cuenta con *mesas, sillas, menús* salvo que se especifique otra cosa.

Roles Personas que intervienen en los sucesos del guión. También pueden ser inferidos si no aparecen explícitamente. Por ejemplo: *camareros, clientes.*

Escenas Secuencias de sucesos que tienen lugar. Los sucesos se representan utilizando DCs. Por ejemplo: *entrar, pedir, comer.*

Track Variación específica de un patrón más general representada por el guión. Los guiones constituyen una jerarquía de especialización: Diferentes tracks del mismo guión comparten algunos componentes pero no todos.

2.7.2. Uso de guiones

Los guiones pueden resultar de utilidad para:

- La predicción de sucesos no mencionados explícitamente
- Indicar la relación entre sucesos

Funcionamiento:

- Selección o activación de un guión apropiado para la situación
- Comprobación de condiciones de entrada
- Uso de otras cabeceras: props, roles...
- Proceso dinámico: el guión seleccionado puede descartarse después

2.7.3. Aplicación del guión

- Inferir información no explícita en las frases analizadas
- La secuencia de eventos es una cadena causal
- Cada uno es consecuencia del anterior y posibilita la realización del siguiente
- Generación de interpretaciones coherentes acerca de una cierta situación, para poder contestar preguntas

2.7.4. Razonamiento por guiones:

Los guiones se activan por coincidencia de *nombre, precondiciones, papeles, etc.*

Objetivo:

Inferir, por medio de razonamiento por defecto, conocimiento que no ha sido dado de forma explícita.

Ejemplo de guiones:

- NOMBRE: Cine
- ROLES: Cinéfilo, Taquillero, Portero, Acomodador
- CONDICIONES DE ENTRADA: Cinéfilo desea ver película
- PROPIEDADES: Pelicula, Butaca, Dinero, Entrada
- ESCENAS:
  - Obtener entrada
    - Cinéfilo MTRANS “Deme boleto” a Taquillero
    - Cinéfilo ATRANS Dinero a Taquillero
    - Taquillero ATRANS Entrada a Cinéfilo
  - Entrar en sala
    - Cinéfilo ATRANS Entrada a Portero
    - Portero ATRANS Entrada a Cinéfilo
    - Cinéfilo PTRANS Cinéfilo a Sala

- Acomodarse...
- Ver Película...
- Salir de sala...

- RESULTADOS:
  - Cinéfilo ha visto Película
  - Taquillero tiene más Dinero
  - Cinéfilo tiene menos Dinero

2.8. CLIPS

CLIPS es un acrónimo de **C Language Integrated Production System** *(Sistema de Producción Integrado en Lenguaje C)*. Es una herramienta que provee un ambiente de desarrollo para la producción y ejecución de sistemas expertos.

Los *sistemas basados en reglas* trabajan mediante la aplicación de reglas, comparación de resultados y aplicación de las nuevas reglas basadas en situación modificada. También pueden trabajar por inferencia lógica dirigida, bien empezando con una evidencia inicial en una determinada situación y dirigiéndose hacia la obtención de una solución, o bien con hipótesis sobre las posibles soluciones y volviendo hacia atrás para encontrar una evidencia existente (o una deducción de una evidencia existente) que apoye una hipótesis en particular.

Características principales de CLIPS:

Representación del conocimiento: CLIPS permite manejar una amplia variedad de conocimiento, soportando varios paradigmas de programación. La programación lógica basada en reglas, la cual permite que el conocimiento sea representado como reglas heurísticas que especifican las acciones a ser ejecutadas dada una situación, la programación orientada a objetos (POO), que permite modelar sistemas complejos como componentes modulares y la programación imperativa la cual permite ejecutar algoritmos de la misma manera que en C, LISP y otros lenguajes.

Portabilidad: CLIPS fue escrito en C con el fin de hacerlo más portable y rápido, y ha sido instalado en diversos sistemas operativos *(Windows 95/98/NT, MacOS X,*

Unix) sin ser necesario modificar su código fuente. CLIPS puede ser ejecutado en cualquier sistema con un compilador ANSI de C/C++. El código fuente de CLIPS puede ser modificado en caso que el usuario lo considere necesario, con el fin de agregar o quitar funcionalidades.

Integrabilidad: CLIPS puede ser integrado en código imperativo (invocado como una sub-rutina) con lenguajes como C, Java, FORTRAN y otros. CLIPS puede ser extendido por el usuario mediante el uso de protocolos definidos.

Desarrollo interactivo: La versión estándar de CLIPS provee un ambiente de desarrollo interactivo y basado en texto; este incluye herramientas para la depuración, ayuda en línea, y un editor integrado. Las interfaces de este ambiente tienen menús, editores y ventanas que han sido desarrollados para MacOS, Windows 95/98/NT, X Window, entre otros.

Verificación / Validacion: CLIPS contiene funcionalidades que permiten verificar las reglas incluidas en el sistema experto que está siendo desarrollado, incluyendo diseño modular y particionamiento de la base de conocimientos del sistema, verificación de restricciones estático y dinámico para funciones y algunos tipos de datos, y análisis semántico de reglas para prevenir posibles inconsistencias.

Documentación: En la página Web oficial de CLIPS se encuentra una extensa documentación que incluye un Manual de Referencia y una Guía del Usuario.

Bajo costo: CLIPS es un software de dominio público.

CLIPS probablemente es el sistema experto más ampliamente usado debido a que es rápido, eficiente y gratuito. Aunque ahora es de dominio público, aún es actualizado y mantenido por su autor original, *Gary Riley* [GR94].

2.9. Planeación de movimientos

Además de planear las acciones, también es necesario planear las rutas o los movimientos que se realizarán. En la planeación de movimientos se requiere **encontrar rutas posibles** de movimientos de un punto origen a un punto destino. Una vez obtenidas dichas rutas se debe escoger la mejor en base a un criterio de optimización.

El problema de búsqueda básico es, dado un punto inicial o nodo, un punto objetivo y un mapa de nodos y conexiones (Fig. 2.1-a), encontrar una ruta o la mejor ruta que encuentre el punto objetivo (Fig. 2.1-b) y finalmente recorrer la ruta seleccionada.

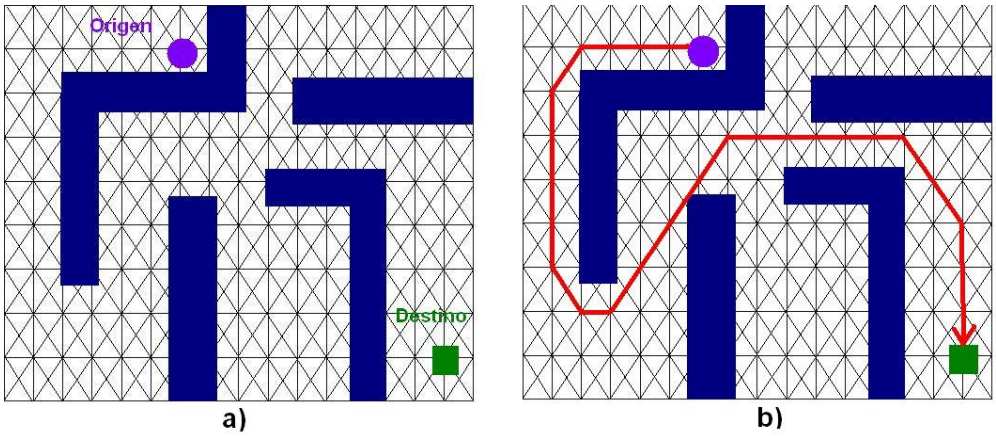

Figura 2.1: Nodo inicial, nodo destino y mejor ruta encontrada en el mapa de nodos

2.10. Dijkstra

La idea básica en la búsqueda de rutas es dada una **gráfica** *(del inglés graph)*, y un **punto inicial**, explorar los nodos adyacentes hasta que se encuentre el **nodo final**. El objetivo es generalmente obtener el **camino más corto** hasta el nodo destino. Existen varios algoritmos de búsqueda, pero uno de los más eficientes y rápidos es el *algoritmo de Dijkstra.*

El algoritmo de Dijkstra también llamado *algoritmo de caminos mínimos* es un algoritmo para la determinación del camino más corto dado un vértice origen al resto de los vértices en una *gráfica dirigida* con pesos en cada arista [Mil06]. Su nombre se

refiere a **Edsger Dijkstra**, quien lo describió por primera vez en 1959. La idea subyacente en este algoritmo consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices; cuando se obtiene el camino más corto desde el vértice origen, al resto de vértices que componen el grafo, el algoritmo se detiene.

Dada una gráfica dirigida *G=(V,E)* y un nodo inicial *s* se requiere encontrar el mínimo camino de *s* a todos los nodos *v* de la gráfica.

V es el conjunto de nodos (vertex) de la gráfica y *E* es un conjunto de aristas (edge) cada una representada por un par ordenado de nodos *(u,v)* representando la conexión del nodo *u* al nodo *v* , *s* es un nodo origen que pertenece a *G* y *w(u,v)* es el costo de ir del nodo *u* al nodo *v*.

Entonces se crea un conjunto *S* de nodos cuyas rutas mínimas ya han sido calculadas, el algoritmo selecciona repetidamente un nodo *u* que pertenece a *V-S* con la distancia mínima, agrega *u* al conjunto *S* y recalcula los costos de todas las aristas desde *u* [Cor90]. En pseudocódigo el algoritmo de Dijkstra queda de la siguiente manera (fig. 2.2):

```
Dijkstra(Grafica, origen)
    para cada nodo v en la Gráfica:      // Inicializaciones
        dist[v] = infinito               // Distancia infinita desde el origen a v
        previo[v] = indefinido           // Nodo previo en la ruta optima desde el origen
    dist[origen] = 0                     // Distancia del origen al origen
    Q = conjunto de todos los nodos de la Gráfica
    // Todos los nodos en la Gráfica que no estan optimizados entran al conjunto Q
    mientras Q no esta vacia:            // El ciclo principal
        u := nodo en Q con la menor dist[]
        si dist[u] = infinito:
           break                         // todos los demás nodos son inaccesibles
       quitar u de Q
       para cada nodo v adjacente a u:   // donde v no ha sido aún eliminado de Q
           alt = dist[u] + dist_entre(u, v)
           si alt < dist[v]:             //se actualizan los valores
               dist[v] = alt
               previo[v] = u
   return previo[]
```

Figura 2.2: Pseudocódigo del algoritmo de Dijkstra para rutas mínimas

De una forma más simple, el algoritmo funciona de la siguiente manera:

1. Crear una lista de distancias, una lista de vértices previos, una lista de visitados y un vértice actual.

2. Todos los valores en la lista de distancias se inicializan con valor infinito excepto el vértice inicial el cual se inicializa con valor cero.

3. Todos los valores en la lista de visitados se inicializan con *falso*.
4. Todos los valores en la lsita de vértices previos se inicalizan como *nulos*.
5. El vértice actual se inicializa con el vértice inicial.
6. Se marca como visitado el vértice actual.
7. Se actualizan las distancias y la lista de vértices previos basándose en los vértices que pueden ser alcanzados directamente desde el vértice actual.
8. Actualizar el vértice actual con el vértice no visitado que se encuentra a menor distancia del vértice inicial.
9. Repetir desde el paso 6 hasta que todos los vértices sean visitados.

Al aplicar este algoritmo se obtienen las rutas más cortas del nodo inicial a cada uno de los nodos, entonces tomado un nodo destino, se tendrá solamente una **ruta mínima del origen al destino**.

Capítulo 3

Análisis y diseño del entorno de ejecución del guiones

Ahora que ya se tienen las bases de conocimiento suficientes para comprender el problema y poder buscar una solución adecuada, el siguiente paso es hacer un **análisis detallado** de lo que se desea realizar, es decir, plantear los requisitos que se deben cubrir, que es lo que se desea que los usuarios puedan hacer con el entorno, como serán las entradas y salidas de información y demás funcionalidades que tendrá.

Una vez hecho este análisis es necesario **diseñar una estrategia** para lograr cumplir con todos los requisitos planteados anteriormente la cual también involucrará herramientas a utilizar y tecnologías seleccionadas para el desarrollo. Todo esto con el objetivo de que posteriormente el proceso de desarrollo sea *más fácil*, *bien estructurado* y que los *errores* de diseño sean los *mínmos* posibles.

3.1. Análisis de los Requisitos del entorno

Dado que el principal *objetivo* de este trabajo es obtener un **framework** que ayude a la **creación de libros aumentados** se deben analizar las características propias de un libro como son e*scenas, personajes, comportamientos, y flujo de la historia.* Por otro lado, se deben considerar también las características necesarias para una aplicación de realidad aumentada, es decir, *marcas, posiciones y modelos gráficos.* Todo esto debe poder representarse como información de entrada en un formato que sea *sencillo de entender y usar* para una persona con poco conocimiento técnico.

Definición de escenas:

Para definir una escena se necesitará la siguiente información:

- **Nombre de la escena.** Ya que un libro puede contener una o más escenas, con éste nombre se podrá identificar una escena determinada.
- **Dimensiones.** Para su representación en Realidad Aumentada, se deben establecer las dimensiones que tendrá la escena en cuestión, esto en unidades entendibles para la representación gráfica.
- **Textura, imágen o color** que tendrá el escenario, esto para que tenga un efecto más realista.
- **Objetos que componen la escena** (por ejemplo: *paredes, árboles, edificios*) cada uno con su posición específica.
- **Marca de realidad aumentada** con la que será ligada.
- **Personajes** que aparecerán en la escena (igualmente con su posición inicial y además su comportamiento o guión de acciones).

Definición de personajes:

Cada uno de los personajes que aparecerán en el libro deben contar con:

- **Nombre del personaje.** Sirve para identificarlo dentro del libro.
- **Modelo 3D** con el cual se generará el gráfico que representará al personaje.
- **Acciones** que podrá realizar. (como *moverse, girar, correr, detenerse*).

- **Comportamientos**, los cuales serán regidos por un guión (acciones que realizará cuando ciertas condiciones se cumplan).
- **Tamaño.** Indica el tamaño que tendrá el personaje en sus 3 dimensiones.

Definición de comportamientos:

Los comportamientos son acciones que uno o varios personajes ejecutarán en caso de cumplirse ciertos hechos. La información necesaria para los comportamientos serán:

- **Hechos o reglas** que los activarán.
- El o los **personajes** que serán afectados
- La **características propias** de cada comportamiento, es decir, para un comportamiento de *caminar*, se deberá especificar *el origen y el destino, la velocidad, etc*.

Flujo de la historia (guión):

El flujo de la historia o **guión** es una **serie de acciones o comportamientos sucesivos** que darán forma a la historia. En este caso se necesita una secuencia de comportamientos, cada uno con su información propia y en un orden determinado.

Configuración del sistema de Realidad Aumentada:

Se desea que los libros puedan ser representados en Realidad Aumentada (RA), entonces, se requiere que exista un **mecanismo de configuración de RA**, es decir, poder establecer las marcas que participarán en la historia, los hechos que desencadenarán, sus características como tamaño, posición relativa a la escena y otras.

Representación de la información:

Es necesario contar con un *repositorio* para toda la **información** que hasta el momento se ha analizado, además se requiere que dicha información tenga un **formato fácil de entender y utilizar** que además sea **estándar** para poder utilizar herramientas ya desarrolladas para la posterior creación de aplicaciones.

3.2. Selección de tecnologías

Hasta ahora se tiene un *panorama general* de todos los puntos que se desea cubrir, entonces, es necesario **plantear una propuesta** que combine los requerimientos con aspectos *técnicos* que puedan ser implementados en la etapa de desarrollo y que cumplan con todos los requerimientos mencionados.

3.2.1. Representación del libro:

Por las características solicitadas anteriormente se puede observar que es necesario un *lenguaje de definición de atributos estándar, sencillo y muy difundido*, además si se considera que con dicho lenguaje se deberá poder representar todos los elementos que componen a un libro aumetado será apropiado utilizar una *representación jerárquica* dado que en dichos elementos se encuentra la siguiente relación (Fig. 3.1):

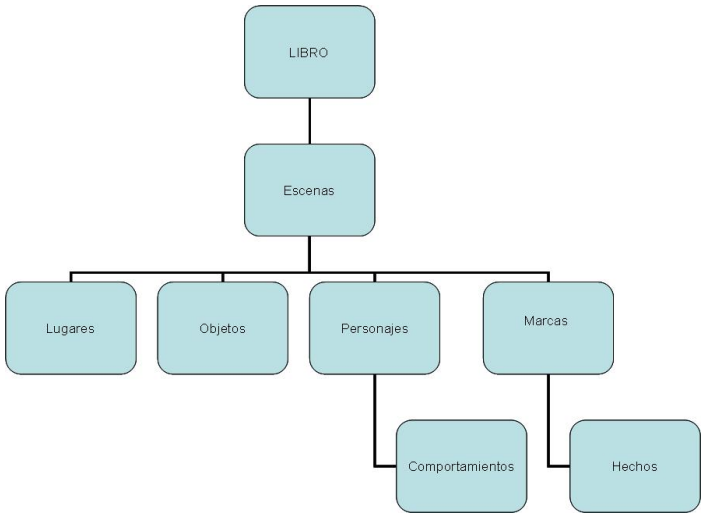

Figura 3.1: Jerarquía de los elementos que contiene un libro aumentado

Dado que **XML** es un *meta-lenguaje* que esta siendo convirtiéndose en un *estándar* que además es *jerárquico* y funciona adecuadamente para la definición de atributos, **se seleccionó XML** como lenguaje de scripting para contener la información de entrada requerida.

3.2.2. Reaidad Aumentada

Un punto sumamente importante en este proyecto es la Realidad Aumentada, por esta razón se debe contar con alguna *herramienta* que permita el desarrollo de aplicaciones de este tipo. Para este propósito **se seleccionó ARToolkit**, una herramienta para construir aplicaciones de RA, que proporciona un *algoritmo de detección de marcas* y posicionamiento de los objetos virtuales en base al patrón detectado con un mínimo costo, ya que básicamente sólo se requiere una *webcam* y una *computadora*.

3.2.3. Lenguaje de Programación

ARToolkit es una biblioteca escrita en **lenguaje C** y debido a que se pretende utilizar dicha *herramienta*, el lenguaje de programación propuesto inicialmente fué C, pero considerando que actualmente el *desarrollo orientado a objetos* está tomando cada vez más fuerza y convirtiendose en el paradigma de programación más utilizado, se buscó una manera de *combinar ambas necesidades* en un sólo lenguaje, por lo que finalmente se optó por utilizar **lenguaje C++** *(orientado a objetos)* el cual no tiene problemas para *convivir con C* y así poder utilizar las bibliotecas de RA (*ARToolkit*) y utilizar las bondades de la *orientación a objetos* para facilitar la modularización de código y aplicaciones.

Para la parte gráfica **se seleccionó OpenGL**, ya que son bibliotecas para generar gráficos utilizando *lenguaje C* lo cual se acopla muy bien a las necesidades del proyecto, además de que es ampliamente conocido y utilizado por desarrolladores de gráficos en 3D.

3.3. Diseño del framework

3.3.1. Toolkits y Frameworks

A menudo una aplicación incorpora *clases de una o más librerías* de clases predefinidas llamadas *toolkits*. Un **toolkit** es un conjunto de clases relacionadas y reusables diseñadas para proveer de funcionalidades de propósito general.

Los toolkits no imponen un diseño particular en las aplicaciones, solo proporcionan funcionalidades que ayudan a las aplicaciones a hacer su trabajo. Permiten a

los desarrolladores *evitar recodificar* funcionalidades comunes. Los toolkits enfatizan el *reuso de código*. Son el equivalente de las *bibliotecas* en orientación a objetos.

Un **framework** es un conjunto de *clases cooperativas* que hacen posible un diseño reutilizable para algún tipo de software específico [Deu89] [JF88]. Por ejemplo, un framework puede estar orientado a *construir editores gráficos* para diferentes dominios como *dibujo artistico, composicion musical, CAD* [VL90] [Joh92].Otro framework puede ayudar a *construir compiladores* para diferentes lenguajes de programacion y plataformas. Otro más puede ayudar a la *construccion de aplicaciones de modelos finacieros* [BE93].

Un framework se *personaliza* para una aplicación particular creando subclases o clases abstractas especificas para dicha aplicación. El framework **dicta la arquitectura** de las aplicaciones, define la estructura total, su partición en clases y objetos, las responsabilidades principales del mismo, cómo colaboran las clases y objetos y el flujo de control.

Un framework *predefine los parámetros de diseño* para que el desarrollador de la aplicación pueda concentrarse en los aspectos específicos de su aplicación. El framework captura las *decisiones de diseño* que son comunes a su dominio de aplicación. Los frameworks entonces enfatizan el **reuso de diseño** más que el reuso de codigo, por tanto un framework generalmente incluirá subclases concretas que se pueden utilizar directamente para trabajar.

El reuso a este nivel lleva a un *mejor control* entre la aplicación y el software en el cual se basa. Cuando se utiliza un Toolkit (o una biblioteca convencional) se escribe el cuerpo principal de la aplicación y *se hacen llamadas al código* que se desea reusar. Cuando se utiliza un framework, *se reusa el cuerpo principal* y se escribe el código al que se hace la llamada. Se tienen que escribir operaciones con nombres específicos y convenciones de llamadas pero se reducen las desiciones de diseño que se tienen que hacer.

No sólo se pueden **construir aplicaciones más rápidamente** como resultado, si no que también las aplicaciones tendrán estructuras similares. Con esto son *más fácil de mantener* y son *más consistentes* para los usuarios. Por otro lado, se pierde algo de libertad creativa, ya que muchas de las decisiones de diseño ya han sido tomadas.

3.3.2. Arquitectura del framework

Dado que lo que se pretende construir es un framework, es necesario contar con un *diseño adecuado* que además permita cumplir con todos los requerimientos planteados anteriormente. Para lograr esto, se propone la siguiente arquitectura (Fig. 3.2):

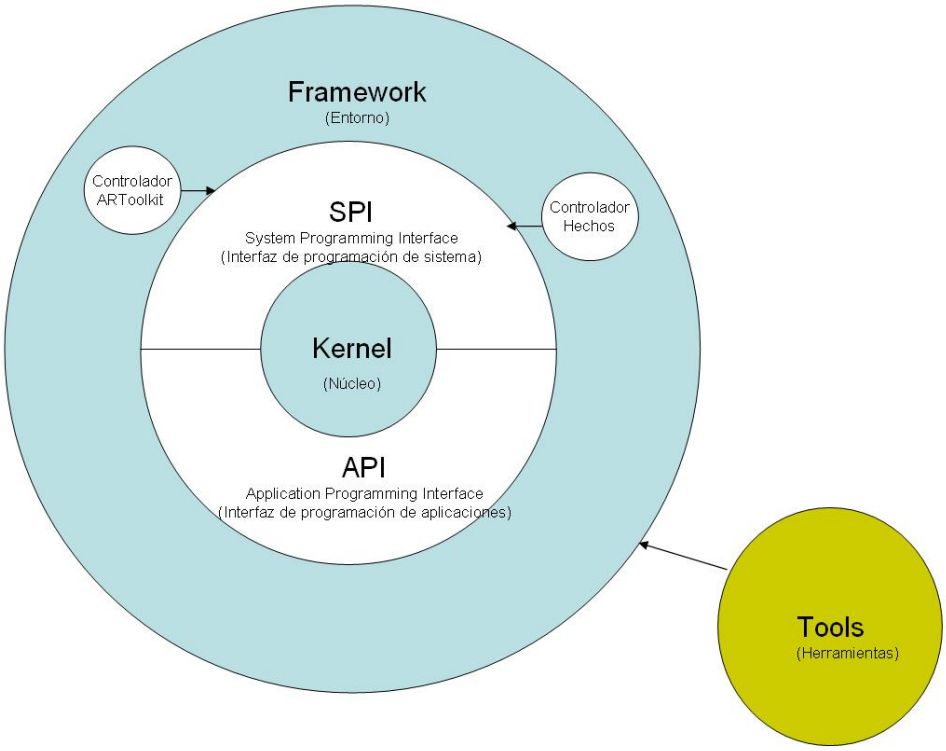

Figura 3.2: Arquitectura del entorno de desarrollo

Kernel: Es la parte central o principal del Entorno de Ejecución, es la encargada de coordinar las operaciones y llevar el flujo de la operación. Contiene los motores de hechos y acciones entre otras cosas.

SPI (System Programming Interface): Es la interfaz que presenta el Kernel a través de la cual se puede extender o implementar.

API (Application Programming Interface): Es la interfaz que presenta el Kernel a través de la cual se puede utilizar, son una serie de clases abstractas o inter-

faces que abstraen conceptos como el render, ejecución de reglas, descripción de objetos y comunicación entre estos.

Controladores (drivers): Son los encargados de comunicarse con el Kernel a través del SPI y son la implementación del Kernel.

Controlador ARToolkit: Este driver o controlador implementa el Kernel con funciones de Realidad Aumentada y despliegue visual con OpenGL.

Controlador Hechos: Este driver o controlador implementa el Kernel con funciones propias para manejo de hechos.

Framework: Es la parte que se comunica con el Kernel por medio de API y proporciona las funciones a las que se puede acceder para desarrollar apliaciones.

Tools: Son las herramientas proporcionadas al usuario del Framework para acceder a él, es decir la *UI (User Interface o Interfaz de Usuario)*.

3.4. Estrategias de programación

Si las aplicaciones son difíciles de diseñar, los toolkits son aún más difíciles y *los frameworks son los más dificiles de todos*. Un disañador de frameworks pretende que una arquitectura funcione para todas las aplicaciones del dominio. Cualquier cambio sustantivo al diseño del framework reducirá sus beneficios considerablente ya que la principal contribución del framework a una aplicacion es *la arquitectura que define*.

Entonces es imperativo diseñar el framework para ser *lo más flexible y extensible posible*. Un framework que utiliza *patrones de diseño* podrá alcanzar más altos niveles de reuso de diseño y código que un framework que no los utiliza. Los frameworks maduros generalmente incorporan *varios* patrones de diseño, los patrones ayudan a hacer la arquitectura del framework adecuada para diversas aplicaciones sin tener que rediseñar [Gam04].

Por tal motivo, para el desarrollo de este proyecto se utilizaron un par de patrones de diseño: *Composite y Command*.

3.4.1. Patrones de Diseño tipo Estructural (Structural)

Los *patrones estructurales* están enfocados en como se componen las clases y los objetos para *formar estructuras mas grandes*. Los patrones estructurales usan herencia para crear interfaces o implementaciones. Como ejemplo, se puede considerar cómo la *herencia* múltiple mezcla dos o más clases en una sola. El resultado es una clase que combina las propiedades de sus clases padre. Este patrón es particularmente útil para hacer que clases de bibliotecas desarolladas independientemente *puedan trabajar juntas*.

En lugar de componer interfaces o implementaciones, los patrones estructurales describen formas de **componer objetos para obtener nuevas funcionalidades**. La flexibilidad agregada con la composicion de objetos se obtiene gracias a la habilidad de **modificar la composicion en tiempo de ejecución**, lo que sería imposible con una *composicion estática* (composición de clases común).

3.4.1.1. Patrón de Diseño COMPOSITE

Composite es un ejemplo de *patrones estructurales*. Describe como construir una *jerarquía de clases* utilizando dos tipos de objetos: *primitivos y compuestos*. Los objetos compuestos permiten componer a su vez a primitivos y otros objetos compuestos resultando en *estructuras arbitrariamente complejas*.

El propósito de este patrón es componer los objetos en una *estructura tipo árbol* para representar jerarquías parte-todo. *Composite* permite que los usuarios traten a los objetos individuales y a los objetos compuestos uniformemente.

Por ejemplo, las aplicaciones gráficas como los editores y los sistemas de captura permiten a los usuarios crear diagramas complejos usando componentes simples. El usuario puede agrupar los componentes para formar componentes más grandes, los cuales a su vez pueden ser agrupados para formar componentes más grandes aún.

Se podría pensar en una implementación simple de las clases que representan a las primitivas como linea, texto, más otras clases que actúen como contenedores de dichas primitivas, pero al hacer esto, surge un pequeño inconveniente, se deben tratar los objetos de las primitivas y a los contenedores de manera diferente a pesar de que el usuario los utilice por igual.

Entonces hacer la distinción entre estos objetos hace que las aplicaciones *se vuelvan*

más complejas. *El patrón Composite* describe como usar *composición recursiva* para que los usuarios no tengan que hacer dicha distinción.

La clave en el patrón *Composite* es **una clase abstracta** que represente tanto a las primitivas como a los contenedores. Para el ejemplo de los gráficos, esta clase es *Graphic* (Fig. 3.3). *Graphic* declara operaciones como *Draw* que son específicas para los objetos gráficos, también declara operaciones que comparten todos los objetos compuestos, como las operaciones para acceder y manejar clases hijas.

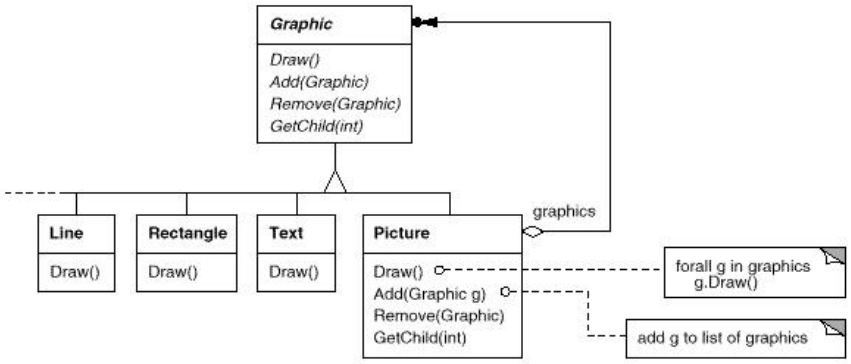

Figura 3.3: Ejemplo de estructura jerárquica composite [Gam04]

Las subclases *Line, Rectangle y Text* (Fig. 3.3) definen objetos gráficos primitivos. Estas clases implementan *Draw* para dibujar líneas, rectángulos y texto respectivamente. Dado que los gráficos primitivos no tienen objetos gráficos hijos, ninguna de estas subclases implementan las operaciones relacionadas con clases hijas.

La clase *Picture* (Fig. 3.3) define un conjunto de objetos gráficos. *Picture* implementa *Draw* para llamar a *Draw* de cada uno de sus hijos y también implementa operaciones relacionadas con ellos. Dado que la interfaz *Picture* conforma a la interfaz *Graphic*, entonces objetos de la clase *Picture* pueden componer otros objetos *Picture* recursivamente. De manera más general, la estructura de este patrón se puede representar de la siguiente manera (Fig. 3.4):

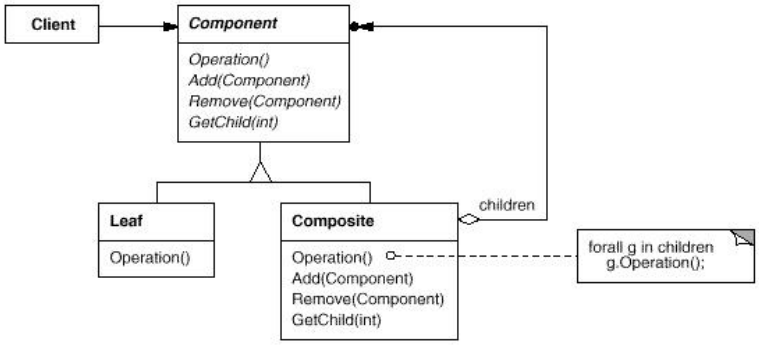

Figura 3.4: Estructura del patrón ***Composite*** [Gam04]

Un ejemplo de utilización de este patrón en el presente proyecto se puede observar en las clases *Behavior, ModelObjectBehavior y BehaviorSequence* (Fig. 3.5).

```
class Behavior{
   public:
     enum BehaviorState{ACTIVE,DONE};
     virtual BehaviorState step()=0;
};

class ModelObjectBehavior:public virtual Behavior{
  protected:
    ModelObject* modelObject;
  public:
     ModelObjectBehavior(ModelObject*);

};

class BehaviorSequence:public virtual Behavior{
     protected:
         vector<ModelObjectBehavior*> sequence;
         int position;
     public:
         BehaviorSequence();
         void addBehavior(ModelObjectBehavior *);
         Behavior::BehaviorState step();
         void stepNoReturn();
};
```

Figura 3.5: Clases Behavior, ModelObjectBehavior y BehaviorSequence implementando el patrón *Composite*

En este ejemplo la clase *Behavior* será la clase abstracta, *ModelObjectBehavior* será una clase primitiva y *BehaviorSequence* será la clase compuesta, ya que contiene a su vez un vector de objetos de la clase *ModelObjectBehavior*.

3.4.2. Patrones de Diseño tipo Comportamiento (Behavioral)

Los patrones *comportamiento* se enfocan en algoritmos y la asignación de responsabilidades entre los objetos. Los patrones *comportamiento* describen no sólo patrones de objetos o clases si no también los patrones de comunicación entre ellos. Estos patrones se utilizan cuando *el control de flujo es complejo* y por tal razón dificil de seguir en tiempo de ejecución. Permiten olvidarse del control de flujo y concentrarse solamente en cómo estan interconectados los objetos.

Los patrones de objetos *comportamiento* usan la composición más que la herencia. Algunos describen como un par de objetos cooperan para realizar una tarea que un objeto simple no podría realizar solo. Otros están relacionados con encapsular un comportamiento en un objeto y delegar las peticiones a este. El patron **Command**, encapsula una petición en un objeto para que pueda ser pasado como un parámetro, almacenado en un historial, o manipulado de alguna otra manera.

3.4.2.1. Patrón de diseño COMMAND

Command permite solicitar una operación a un objeto sin conocer realmente el contenido de esta operación, ni el receptor real de la misma. Para ello se *encapsula la petición* como un objeto, con lo que además se facilita la parametrización de los métodos.

Algunas veces es necesario lidiar con peticiones a objetos sin saber nada sobre la operacion que será ejecutada o el receptor de la misma. Por ejemplo, los toolkits de interfaces de usuario incluyen objetos como botones y menús que llevarán a cabo peticiones en respuesta a la interacción del usuario. Pero el toolkit no puede implementar la peticion explicitamente en el botón o menú porque sólo las aplicaciones que utilicen el toolkit sabrán que se debe hacer en cada caso.Los diseñadores del toolkit no cuentan con alguna forma de saber quién será el receptor de la petición o las operaciones que efectuará.

El patrón *Command* permite hacer peticiones de objetos de aplicación no específicas al convertir las peticiones en un objeto. Este objeto puede ser almacenado y pasado como cualquier otro objeto. La *clave de este patrón* es **una clase abstracta *Command***, la cual declara una interfaz para **ejecutar operaciones**.

De forma simple, esta interface incluye una operacion abstracta *Execute*. Las subclases de *Command* especifican un par *receptor-acción* al almacenar el receptor como una variable de instancia e implementar *Execute* para invocar la petición. El receptor tiene el conocimiento requerido para llevar a cabo dicha petición.

Con este patrón se podrá tener una interfaz común que permita invocar las acciones de forma uniforme y extender el sistema con nuevas acciones de forma más sencilla. Es muy útil para implementar *CallBacks*, especificando que órdenes se desea que se ejecuten en ciertas situaciones de otras órdenes. Es decir, un parámetro de una orden puede ser otra orden a ejecutar, también se pueden desarrollar sistemas utilizando órdenes de alto nivel que se construyen con operaciones sencillas (primitivas).

La estructura general de este patrón se puede representar de la siguiente manera (Fig. 3.6):

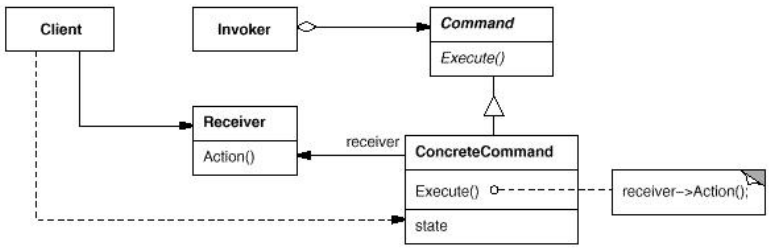

Figura 3.6: Estructura del patrón ***Command*** [Gam04]

La clase abstracta *Command* (Fig. 3.6) ofrece una interfaz para la ejecución de una operación. La clase *ConcreteCommand* define un vínculo entre el objeto receptor y una acción, además de implementar *Execute* al invocar las operaciones correspondientes en el receptor. La clase *Client* (cliente o aplicación) crea un objeto *ConcreteCommand* y establece su receptor, mientras que la clase *Invoker* pide al *command* que efectúe la petición. La clase *Receiver* sabe como ejecutar las operaciones asociadas con llevar a cabo la petición. Cualquier clase puede servir como receptor.

En el presente proyecto, un ejemplo del patrón Command sería (Fig.3.7):

```
class Command {
    public:
        virtual void execute() = 0;
};

template<class Receiver> class SimpleCommand: public Command {
    public:
        typedef void (Receiver::* Action)();
        SimpleCommand(Receiver* r, Action a): _receiver(r), _action(a){}
        void execute();
    private:
        Action _action;
        Receiver* _receiver;
};

template <class Receiver> void SimpleCommand<Receiver>::execute () {
         (_receiver->*_action)();
}
```

Figura 3.7: Clases Command y SimpleCommand

La clase *Command* (Fig. 3.7) es la interfaz y la clase *SimpleComand* es el equivalente a *ConcreteCommand* del diagrama anterior (Fig. 3.6). Para ejemplificar el uso de estas clases, se tiene el método setDrawSceneCallback de la clase ARToolkitBind, que en su definición es de la forma ***void setDrawSceneCallback(Command*)*** y para realizar la petición se hace como en la Fig. 3.8, donde el receptor es la clase *Scene* y la acción es el método *Draw* de dicha clase.

```
void setDrawSceneCallback(Command*);

ARToolkit->setDrawSceneCallback(new SimpleCommand<Scene>(scene,&Scene::draw));
```

Figura 3.8: Utilización de las clases *Command*

Capítulo 4

Desarrollo del entorno de ejecución de guiones

Una vez analizado el problema y habiendo diseñado una solución óptima, se procede a la **implementación de dicho diseño**. Para lograr ésto de manera exitosa se deben considerar algunos puntos importantes en el proceso de desarrollo de software, por ejemplo, se debe seguir un *ciclo de desarrollo* que no necesariamente será lineal, ya que generalmente el diseño original se modificará o ajustará a lo largo del desarrollo por lo que se deberá ir desarrollando por partes, una vez finalizada y probada cada una de estas partes se deberá rediseñar o ajustar algunos detalles nuevos o no considerados originalmente para proseguir con la siguiente parte. Otro punto importante es realizar *diagramas de clases*, los cuales proporcionarán un panorama más amplio de todo el desarrollo y posteriormente facilitarán la implementación de dichas clases.

Una vez concluido el desarrollo es necesario *probar* que la implementación funcione como se esperaba, para esto se diseñará y realizará *una pequeña aplicación* utilizando el entorno o framework para corroborar que se cumplió el objetivo.

4.1. Modelo de Desarrollo

La Ingeniería del Software, se vale de una serie de modelos que establecen y muestran las distintas etapas y estados por los que pasa un producto software, desde su concepción inicial, pasando por su desarrollo, puesta en marcha y posterior mantenimiento, hasta la retirada del producto. A estos modelos se les denomina *"Modelos de Ciclo de Vida del Software"*. El primer modelo concebido fue el más comunmente conocido como *Cascada o "Lineal Secuencial"*. Este modelo establece que las diversas actividades que se van realizando al desarrollar un producto de software se suceden de forma lineal. Sin embargo, actualmente estamos en un punto en el desarrollo de software donde los métodos de diseño estructurado *top-down* no son suficientes para lidiar con la complejidad de los posibles problemas actuales.

Desarrollo en cascada y desarrollo iterativo

El antiguo paradigma de tratar el proceso de desarrollo de software como una línea de ensamble ya no es válida. Tomando en consideración el tradicional modelo de desarrollo de software en cascada, se puede notar que en este modelo, *el análisis, diseño, codificación, pruebas y mantenimiento* forman cinco pasos discretos, cada uno con una entrada y salida bien definida(Fig. 4.1). En cada transición las salidas son entregables que pemiten a los administradores evaluar el progreso del proyecto [Rie62].

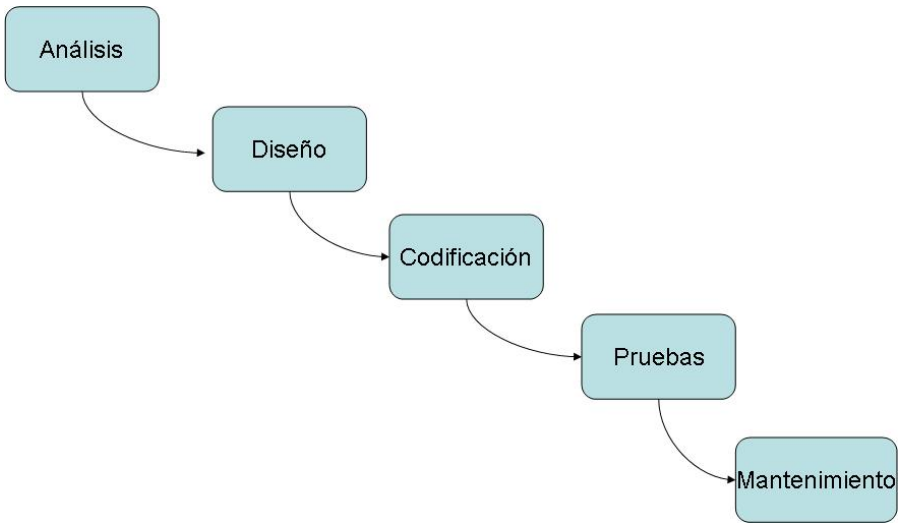

Figura 4.1: Modelo de desarrollo de software en Cascada

Con el tiempo se empezó a notar que *el proceso de desarrollo de software* no era exactamente un proceso de producción en serie o línea de ensamble. Por ejemplo, en el mejor curso de acción en una línea de producción si uno de los eslabones en la cadena se retrasa, retrasa el resto de la línea, es decir, si en una línea de producción estan construyendo muñecas y la persona que pone los brazos esta alentando la línea, *¿cuál sería la solución mas obvia para este caso?*

La respuesta sería reemplazar a la persona en la línea para poner los brazos. Esto eliminará el cuello de botella. Sin embargo si se intenta hacer lo mismo en el caso de la ingeniería de software los resultados son generalmente *desastrosos*. Si la codificación esta atrazada, un administrador de proyecto **no puede simplemente agregar algunos miembros al equipo de desarrollo**, ya que los miembros actuales del equipo verán un decremento en su productividad debido a que necesitan tomarse tiempo para entrenar a los nuevos.

El modelo iterativo de desarrollo de software es basicamente un modelo de cascada solo que se permite a los desarrolladores moverse en dos direcciones a lo largo del flujo del proyecto (Fig. 4.2). Es decir, si se detecta una falla mientras se esta codificando se puede regresar al diseño y corregirlo, o bien si se detecta la necesidad de un nuevo requerimiento del sistema mientras se está probando la aplicación se puede regresar al análisis y reparar el problema [Rie62]. Este modelo es adecuado para proyectos en los que se tienen claros los objetivos finales pero no todos los detalles de implementación están definidos inicialmente.

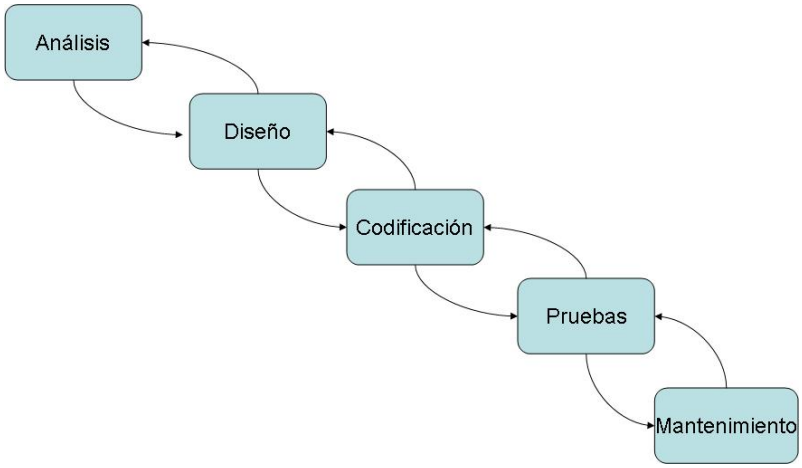

Figura 4.2: Modelo de desarrollo de software Iterativo

4.2. Descripción de la implementación

La implementación del proyecto se realizó con **lenguaje C++** utilizando **OpenGL** (para el manejo de gráficos), **ARToolkit** (para realidad aumentada) y **EXPAT** (para el uso de XML). Este proyecto se compone de una serie de clases intecomunicadas y agrupadas para realizar distintas tareas, por ejemplo, existen clases encargadas de hacer el manejo de realidad aumentada, otras encargadas de generar los modelos en 3D utilizando OpenGL y otras más relacionadas con los comportamientos que se ejecutarán sobre algún modelo 3D.

4.2.1. Diagrama de clases

Dado que el desarrollo se llevó a cabo utilizando el paradigma de programación orientado a objetos, se requiere contar con un diseño también orientado a objetos que se adapte adecuadamente a las necesidades del proyecto. Para la mejor comprensión de dicho diseño se tienen los diagramas de clase, estos *diagramas de clase* además de ser de uso extendido, también están sujetos a la más amplia gama de conceptos de *modelado*.

Un diagrama de clases describe los *tipos de objetos* que hay en el sistema y las diversas clases de *relaciones estáticas* que existen entre ellos. Los diagramas de clase también muestran los *atributos y operaciones* de una clase y las restricciones a las que se ven sujetos, según la forma en que se conectan los objetos [Fow99].

Al ser un proyecto que involucra diversos tópicos, es necesario separar en partes su desarrollo, estas partes son:

- Realidad aumentada
- Gráficos
- Comportamientos
- Hechos y reglas
- XML

4.2.1.1. Clases para realidad aumentada

Dado que la realidad aumentada juega un papel sumamente importante en el proyecto, se cuenta con una serie de clases que se encargan de hacer posible el uso de esta tecnología.

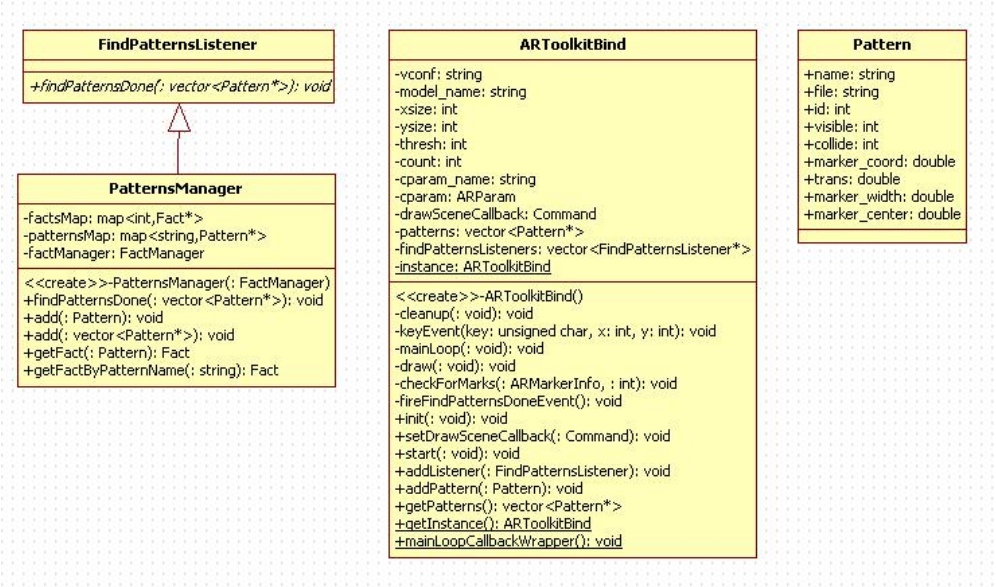

Figura 4.3: Clases para el manejo de ARToolkit

La clase ***ARToolkitBind*** (fig. 4.3) es una clase creada para poder hacer uso de las funciones de ARToolkit pero con el paradigma de orientación a objetos, esto significa que a pesar de que ARToolkit fue creado para utilizarse con programación estructurada (*lenguaje C*) es posible utilizarlo con un enfoque diferente. Para lograr esto se deben hacer algunas consideraciones a la hora de hacer la migración entre paradigmas.

Por ejemplo, ARToolkit utiliza *callbacks* los cuales en orientación a objetos no tienen un simil, entonces habrá que hacer algunas maniobras para poder hacer el cambio de programación estructurada a programación orientada a objetos.

Por otro lado, también se requiere tener libertad para manejar patrones o marcas para realidad aumentada, por lo que se cuenta con la clase ***Pattern*** (fig. 4.3) para representar la información que ARToolkit necesita para considerarlo como un patrón válido.

Además se cuenta con una interfaz ***FindPatternsListener*** y una clase ***Patterns-Manager*** (fig. 4.3) las cuales además de interactuar con *ARToolkitBind* son la liga con las clases que manejan los hechos o eventos que posteriormente desencadenarán los comportamientos correspondientes.

4.2.1.2. Clases para gráficos

Para poder generar un efecto de realidad aumentada, es necesario poder generar gráficos de modelos 3D, los cuales serán sobrepuestos al video capturado en la posición indicada por las clases de realidad aumentada. Para lograr esto, se requieren algunas clases que se encarguen del manejo de los elementos gráficos.

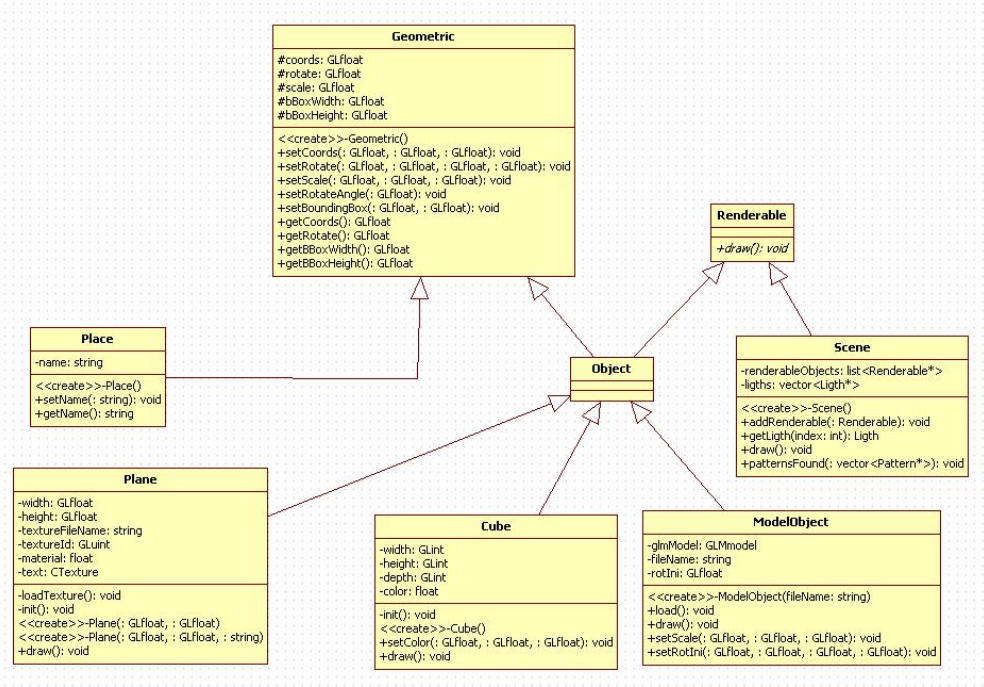

Figura 4.4: Clases para manejo de gráficos

La clase ***Geometric*** (fig. 4.4) es una clase creada para representar todos aquellos valores requeridos por un objeto geométrico como su posición en el mundo virtual, rotación y cambio de escala.

La interfaz ***Renderable*** (fig. 4.4) será implementada por cualquier clase que requiera ser dibujada en el mundo virtual.

Al combinar ambas clases *(Geometric y Renderable)* se forma la clase ***Object*** (fig. 4.4) de la cual heredarán las clases que podrán ser representadas y desplegadas en el mundo virtual.

Un ejemplo de estas clases son *Plane*, *Cube* y *ModelObject* (fig. 4.4), las dos primeras sirven para representar y dibujar planos, cubos y la última se encarga de generar los modelos 3D utilizando información de vértices, normales y texturas contenidos en archivos con formato *OBJ*. (Esta clase posteriormente puede ser modificada para utilizar otro tipo de archivos, inclusive formatos de modelos con animación).

4.2.1.3. Clases para comportamientos

Para representar y ejecutar los comportamientos que podrá realizar cada uno de los personajes en la escena se tienen las clases que se encargan de modelar y manejar dichos comportamientos.

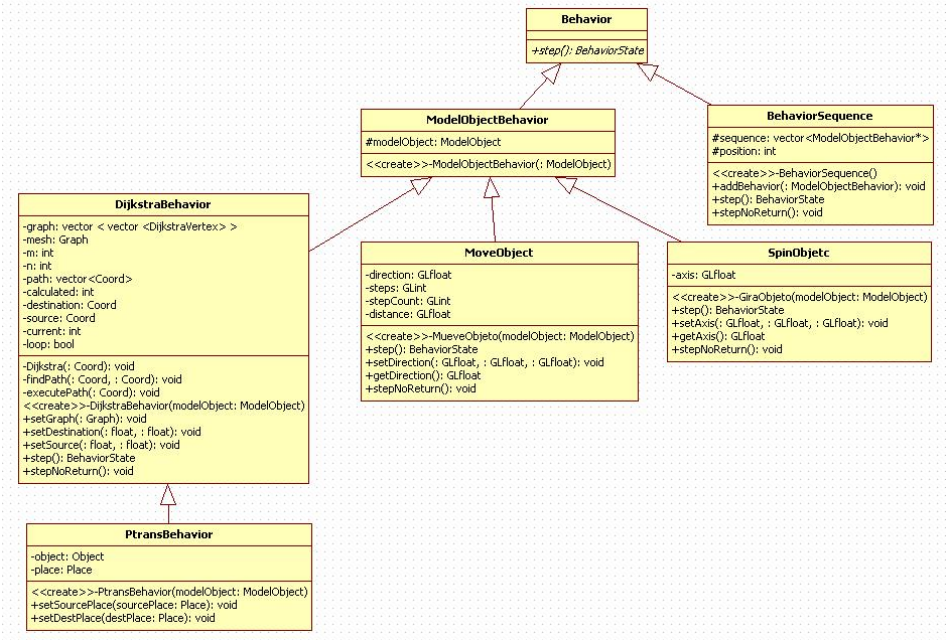

Figura 4.5: Clases involucradas en los comportamientos (Behavior)

La iterfaz ***Behavior*** (fig. 4.5) es la clase base de la que heredarán todas las clases que pretendan implementar comportamientos.

Para poder relacionar dichos comportamientos con los personajes o modelos 3D existe la clase ***ModelObjectBehavior*** (fig. 4.5) la cual liga un objeto de la clase *ModelObject* con la clase *Behavior*. Como la mayoría de los comportamientos afectarán a un personaje o modelo en específico, muchas de las clases de comportamientos heredarán de *ModelObjecBehavior*.

Se cuenta también con las clases ***MoveObject*** y ***SpinObject*** (fig. 4.5) que sirven para mover un objeto de un punto origen a un punto final la primera y para rotar sobre un eje al objeto la segunda.

Estas clases son una muestra de los comportamientos que pueden ser implementados posteriormente dependiendo las necesidades del desarrollador, es decir, se pueden crear nuevos comportamientos siguiendo el diseño propuesto en estas clases.

Otro ejemplo de los comportamientos que pueden ser creados son las clases ***DijkstraBehavior*** (fig. 4.5) con la cual se crean comportamientos que utilizan el algoritmo de Dijkstra para encontrar rutas y mover al personaje por esas rutas esquivando obstáculos y ***PtransBehavior*** (fig. 4.5) que deriva de *DijkstraBehavior* para poder también hacer uso de las rutas pero con el estilo de *PTRANS* de dependencia conceptual.

Por otro lado, uno de los objetivos es poder ejecutar guiones o secuencias de acciones, para esto, se tiene la clase ***BehaviorSequence*** (fig. 4.5), que deriva de *Behavior* y contiene a su vez elementos *ModelObjectBehavior* que serán ejecutados uno tras de otro en secuencia.

4.2.1.4. Clases para Hechos y Reglas

Dado que se desea contar un motor de hechos (similar al que se tiene en CLIPS pero un poco más simplificado) se tienen clases encargadas de representar hechos y manejarlos de acuerdo a expresiones y reglas que pueden a su vez desencadenar otros hechos.

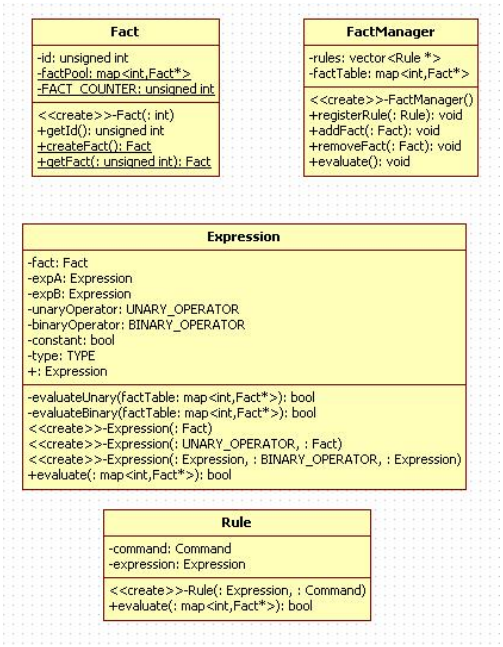

Figura 4.6: Clases para el manejo de Hechos y Reglas

La clase ***Fact*** (fig. 4.6) representa un hecho (que puede estar ligado a un patrón de realidad aumentada por ejemplo) el cual tiene asignado un identificador o *id* numérico y se encuentra en un *pool* de hechos, es decir, un repositorio o banco de hechos al cual se recurre cuando se necesita un hecho.

La clase ***Expression*** (fig. 4.6) sirve para representar expresiones las cuales pueden ser unarias o binarias, estas expresiones podrán ser utilizadas con los operadores *NOT, AND y OR* para generar un resultado *booleano* (verdadero o falso).

Con estas expresiones se pueden definir reglas específicas utilizando la clase ***Rule*** (fig. 4.6), por ejemplo, se puede tener una regla que sea de la forma:

Si HechoA y HechoB entonces ejecuta acción

Para poder lograr esta funcionalidad, la clase *Rule* se apoya de la clase ***Command*** (Fig. 4.7), con la cual se aplica el patrón de diseño del mismo nombre, entonces la regla es evaluada mediante la expresión que define la regla y si resulta verdadero se ejecuta el comando asociado al objeto de la clase *Command*.

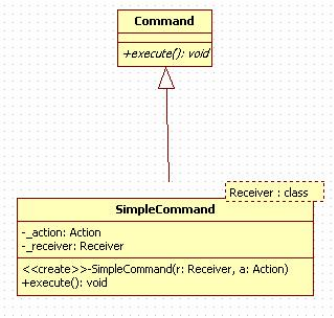

Figura 4.7: Clases para implementar el patrón Command

Finalmente, la clase ***FactManager*** (fig. 4.6) es la encargada de configurar y manejar todos los hechos y reglas que se utilizan, esta clase agrega y quita hechos del *pool*, registra nuevas reglas y evalúa si con los hechos contenidos se pueden cumplir las reglas para entonces ejecutar las acciones correspondientes.

4.2.1.5. Clases para XML

Para poder utilizar la información proveniente de un documento XML se cuenta con la clase *XMLConfig* (Fig. 4.8), la cual como su nombre lo indica sirve para configurar los datos obtenidos desde el XML indicado. Esta clase se encarga de hacer el parseo del XML y convertirlo en objetos de las clases correspondientes del entorno para generar una aplicación.

Figura 4.8: Clase para configuracion de XML

4.2.2. Iteraciones en el proceso de desarrollo

Como se mencionó anteriormente, es conveniente dividir el desarrollo en varias partes para ir construyendo la siguiente parte sobre algo que ya esta funcionando correctamente, esto para evitar que se detecten errores cuando ya se estén efectuando las pruebas.

En el presente proyecto se dividió el desarrollo en *etapas o iteraciones*, las cuales tienen un objetivo particular y para poder comenzar la siguiente se debe haber concluido y probado la anterior, el proceso **técnico** de dichas iteraciones se describen a continuación:

- **ARToolkit en C++**

  El objetivo de esta iteración es lograr migrar ARToolkit *de un paradigma estructurado (lenguaje C) a un paradigma orientado a objetos (lenguaje C++)* conservando todas sus funcionalidades.

  Para lograr este primer objetivo, el principal obstáculo con el que se tiene que lidiar es que las bibliotecas de ARToolkit estan programadas con un enfoque de programación estructurada (lenguaje C) y se desea pasarlo a orientación a objetos (lenguaje C++).

  Esto por una parte tiene algunas ventajas ya que C y C++ además de ser compatibles son muy parecidos, pero también se tiene algunas desventajas como el hecho de que en orientación a objetos los llamados *Callback* o apuntadores a función no se utilizan como tal, por lo cual es necesario aplicar algunas*"técnicas"* para poder conservar la funcionalidad pero con el enfoque requerido.

  En este caso es necesario hacer una función miembro estática que actúe como *wrapper* o *envolvente* y con ella simular los *callbacks* que ARToolkit normalmente utilizaría, esto se puede observar en el siguiente fragmento de código (fig. 4.9):

```
class ARToolkitBind{
    private:
        string vconf;
        string model_name;
        int xsize;
        int ysize;
        int thresh;
        int count;
        string cparam_name;
        ARParam cparam;
        Command* drawSceneCallback;
        vector<Pattern*> patterns;
        vector<FindPatternsListener*> findPatternsListeners;
        static ARToolkitBind* instance;
        ARToolkitBind();
        void cleanup(void);
        void keyEvent( unsigned char key, int x, int y);
        void mainLoop(void);
        void draw( void );
        void checkForMarks(ARMarkerInfo *, int);
        void fireFindPatternsDoneEvent();
    public:
        void init(void);
        void setDrawSceneCallback(Command*);
        void start(void);
        void addListener(FindPatternsListener*);
        void addPattern(Pattern *);
        vector<Pattern*>* getPatterns();
        static ARToolkitBind* getInstance();
        static void mainLoopCallbackWrapper();
};
```

Figura 4.9: Clase ARToolkitBind

Para lograr lo anterior se hace uso del patrón de diseño *Command* (fig. 4.9-1) y se declara un miembro estático de la misma clase llamado *instance* (fig. 4.9-2) los cuales servirán para que sólamente se tenga una instancia de ARToolkit ejecutandose, tal como pasaría con un callback.

Posteriormente se utilizan los métodos *getInstance* (fig. 4.9-4) para crear una instancia de la clase y asignarla al miembro estático (*instance*), en caso de que ya exista una instancia devolverá esta misma y *mainLoopCallbackWrapper* para ejecutar el método *mainLoop* de la instancia.

La implementación de dichos métodos se puede observar en la fig. 4.10, en donde también se aprecia que se debe efectuar una modificación en el método *start* para que además de iniciar la captura de video también asocie el *mainLoop* de ARToolkit con el *mainLoopCallbackWrapper* creado.

```
ARToolkitBind* ARToolkitBind::getInstance(){
    if(instance==NULL)
        instance=new ARToolkitBind();
    return instance;
}

void ARToolkitBind::setDrawSceneCallback(Command* callback){
     drawSceneCallback=callback;
}

void ARToolkitBind::mainLoopCallbackWrapper(){
       instance->mainLoop();
}

void ARToolkitBind::start(void){
    arVideoCapStart();
    argMainLoop( NULL, NULL, ARToolkitBind::mainLoopCallbackWrapper );
}
```

Figura 4.10: Implementación de algunos métodos de la clase ARToolkitBind

Finalmente para poder manejar la generación de gráficos se utiliza el patrón *Command* con el método *setDrawSceneCallback* (fig. 4.9-3), el cual servirá para dibujar la escena al hacer la llamada al método *execute* dentro del metodo *draw* (Fig. 4.11).

```
void ARToolkitBind::draw( void )
{
    double gl_para[16];
    argDrawMode3D();
    argDraw3dCamera( 0, 0 );
    glClearDepth( 1.0 );
    glClear(GL_DEPTH_BUFFER_BIT);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    if(patterns[0]->visible==1){
        argConvGlpara(patterns[0]->trans, gl_para);
        glMatrixMode(GL_MODELVIEW);
        glLoadMatrixd( gl_para );
        drawSceneCallback->execute();
    }
    glDisable( GL_DEPTH_TEST );
}
```

Figura 4.11: Uso de Command en la clase ARToolkitBind

Gracias a la implementación del patrón *Command* se logra que el método *draw* pueda a su vez llamar al método *execute* el cual tendrá el código requerido para cada caso, es decir, no está ligado a la clase *ARToolkitBind* ni a la clase *Command*, como se explicó anteriormente.

Una vez implementado todo lo anterior, la ejecución resultante no tiene mucha diferencia con una aplicación de *ARToolkit* (Fig. 4.12) pero el paradigma utilizado ahora es orientado a objetos como se deseaba inicialmente.

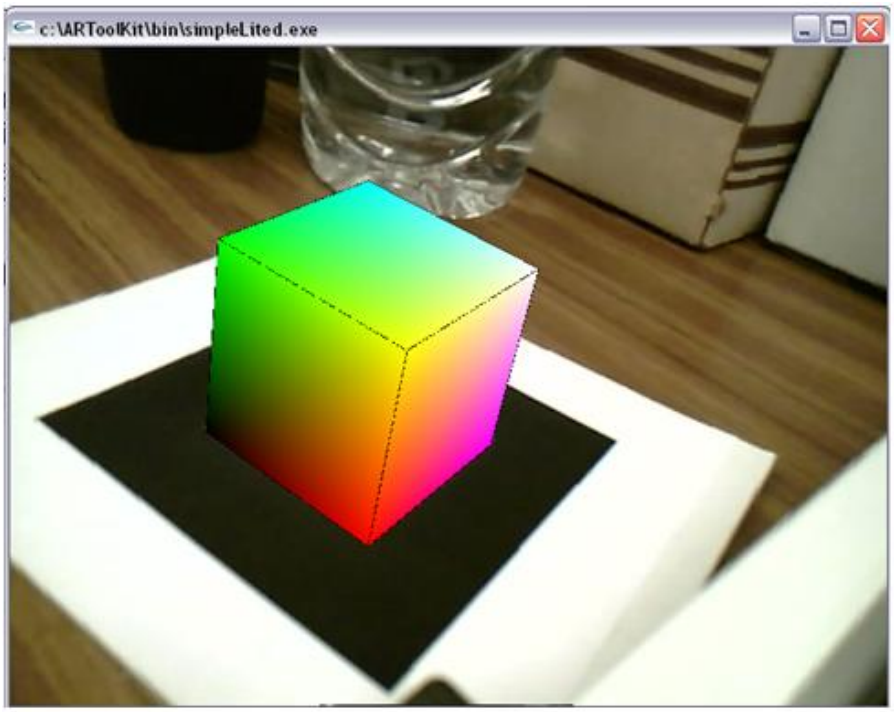

Figura 4.12: Ejemplo de ejecución una vez implementado ARToolkit con C++

- **Agregar clases para patrones ARToolkit**

  Ahora que ya se tiene ARToolkit funcionando con orientación a objetos, se requiere que también los patrones puedan ser representados como clases, entonces se procede a identificar la información que ARToolkit requiere para un patrón y con esto se crea la clase correspondiente (Fig. 4.13).

```
class Pattern{
    public:
        string name;
        string file;
        int id;
        int visible;
        int collide;
        double marker_coord[4][2];
        double trans[3][4];
        double marker_width;
        double marker_center[2];
};
```

Figura 4.13: Implementación de los patrones con la clase Pattern

Después se procede a integrarlo con la clase *ARToolkitBind*, esto se logra mediante la implementación de un vector de objetos de la clase *Pattern* (Fig. 4.14), el cual servirá para el reconocimiento de los patrones que hace ARToolkit (Fig. 4.15).

Al terminar esta iteración la ejecución continúa sin cambios aparentes, sólo se puede notar la mejora en la parte estructural de la programación.

```
class ARToolkitBind{
    private:
        string vconf;
        string model_name;
        int xsize;
        int ysize;
        int thresh;
        int count;
        string cparam_name;
        ARParam cparam;
        Command* drawSceneCallback;
        vector<Pattern*> patterns;
        vector<FindPatternsListener*> findPatternsListeners;
        static ARToolkitBind* instance;
        ARToolkitBind();
        void cleanup(void);
        void keyEvent( unsigned char key, int x, int y);
        void mainLoop(void);
        void draw( void );
        void checkForMarks(ARMarkerInfo *, int);
        void fireFindPatternsDoneEvent();
    public:
        void init(void);
        void setDrawSceneCallback(Command*);
        void start(void);
        void addListener(FindPatternsListener*);
        void addPattern(Pattern *);
        vector<Pattern*>* getPatterns();
        static ARToolkitBind* getInstance();
        static void mainLoopCallbackWrapper();
};
```

Figura 4.14: Uso de la clase Pattern en ArtoolkitBind

```
void ARToolkitBind::checkForMarks(ARMarkerInfo *marker_info, int marker_num){
     for(int i = 0; i < patterns.size(); i++ ) {
        int k = -1;
        for(int  j = 0; j < marker_num; j++ ) {
            if( patterns.at(i)->id == marker_info[j].id) {
                if( k == -1 ) k = j;
                else
                    if( marker_info[k].cf < marker_info[j].cf ) k = j;
            }
        }
        if( k == -1 ) {
            patterns.at(i)->visible = 0;
            continue;
        }
        if( patterns.at(i)->visible == 0 ) {
            arGetTransMat(&marker_info[k],
                          patterns.at(i)->marker_center,patterns.at(i)->marker_width,
                          patterns.at(i)->trans);
        }
        else {
            arGetTransMatCont(&marker_info[k], patterns.at(i)->trans,
                          patterns.at(i)->marker_center, patterns.at(i)->marker_width,
                          patterns.at(i)->trans);
        }
        patterns.at(i)->visible = 1;
    }
}
```

Figura 4.15: Utilización del vector de objetos Pattern en ARToolkitBind

- **Cargar modelos OBJ**

  Una vez que ARToolkit y los patrones funcionan correctamente con el paradigma orientado a objetos, el siguiente objetivo es *cargar modelos 3D* generados con la información contenida en archivos formato *obj*. Para esto se pueden utilizar bibliotecas estándar en lenguaje C que ya han sido probadas para no tener problemas técnicos.

  En este caso se utilizó una biblioteca (*glm*) y se adaptó código para el manejo de texturas y para poder funcionar con orientación a objetos con C++. Si posteriormente si desea se puede utilizar alguna otra biblioteca para cargar modelos 3D con otros formatos diferentes al *obj* e incluso formatos de modelos animados en 3D, ésto da *mayor flexibilidad* a las aplicaciones que serán desarrolladas utilizando este framework.

  Para lograr que los modelos se visualicen en realidad aumentada y que se puedan configurar en cuanto a *posición, tamaño y orientación*, es necesario implementar la clase ***Geometric***, la interfaz ***Renderable*** y la clase ***Object*** como se explicó anteriormente (Fig. 4.16).

```
class Geometric{
   protected:
      GLfloat coords[3];
      GLfloat rotate[4];
      GLfloat scale[3];
      GLfloat bBoxWidth;
      GLfloat bBoxHeight;
   public:
      Geometric();
      void setCoords(GLfloat,GLfloat,GLfloat);
      void setRotate(GLfloat,GLfloat,GLfloat,GLfloat);
      void setScale(GLfloat,GLfloat,GLfloat);
      void setRotateAngle(GLfloat);
      void setBoundingBox(GLfloat,GLfloat);
      GLfloat* getCoords();
      GLfloat* getRotate();
      GLfloat getBBoxWidth();
      GLfloat getBBoxHeight();
};

class Renderable{
  public:
      virtual void draw()=0;
};

class Object:public Geometric, public Renderable{
};
```

Figura 4.16: Implementación de las clases base para el manejo de gráficos

Una vez hecho esto, se implementa la clase *ModelObject* (Fig. 4.17), la cual utiliza *MiGlm* (modificación de la biblioteca glm).

```
class ModelObject: public Object{
      GLMmodel* glmModel;
      string fileName;
      GLfloat rotIni[4];

  public:
      ModelObject(string fileName);
      void load();
      void draw();

      void setScale(GLfloat,GLfloat,GLfloat);
      void setRotIni(GLfloat,GLfloat,GLfloat,GLfloat);
};
```

Figura 4.17: Implementación de la clase ModelObject para el uso de modelos 3D con formato obj

Para poder visualizar modelos 3D que no sean generados a partir de archivos *obj* se crean clases con jerarquía al mismo nivel que *ModelObject* con el código requerido para generar dichos modelos.

Como muestra se crearon las clases *Plane* y *Cube* las cuales están asociadas con un plano con textura la primera y un cubo de color la segunda. Si se desea se puede implementar alguna otra clase por ejemplo una clase *Sphere* que genere esferas, pero se debe cuidar que dicha clase herede de *Object* para poder ser visualizada y configurada correctamente.

Al finalizar esta iteración, se puede observar la ejecución del código de demostración como sigue (Fig. 4.18):

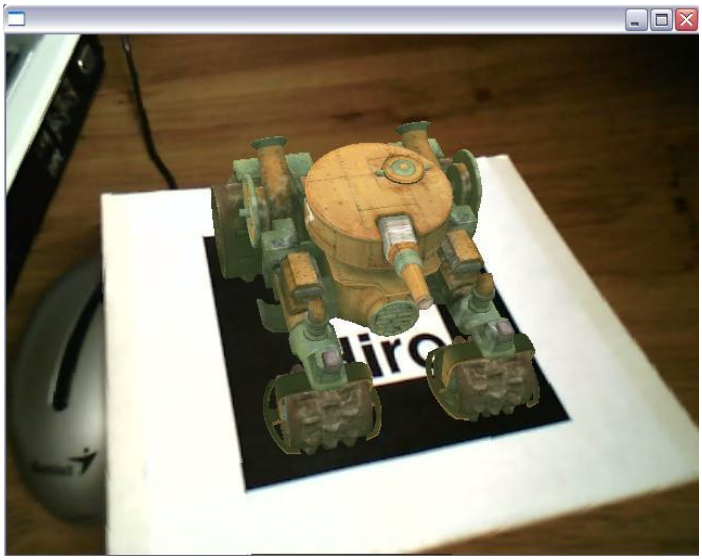

Figura 4.18: ARToolkit con orientación a objetos y modelo OBJ funcionando

- **Primera configuración desde XML**

  Ahora ya se cuenta con ARToolkit y modelos 3D funcionando con la estructura propuesta y el paradigma orientado a objetos, sin embargo, toda la información necesaria para configurar *patrones, archivos obj, posiciones* y demás características se encuentra en *código duro.* Esta situación hace que no se cumpla totalmente el objetivo de este proyecto, entonces se deben efectuar modificaciones para poder *obtener toda la información* necesaria para la *configuración* de una manera más flexible y externa al framework.

  Entonces se crea una clase que realiza la *conversión o parseo* de un **documento XML** y utiliza la información contenida para crear objetos de las clases que se han desarrollado hasta el momento.

Este archivo de configuración (Fig. 4.19) será mejorado posteriormente.

```
<?xml version="1.0" encoding="utf-8"?>
<book name="La bella durmiente">

  <mark name="Hiro" file="Data/patt.hiro" width="80.0" centerX="0.0" centerY="0.0"></mark>

  <character name="Tank" file="Tank.obj" scaleX=".3" scaleY=".3" scaleZ=".3" rotAng="90.0"
   rotX="1.0" rotY="0.0" rotZ="0.0" posX="0.0" posY="0.0" posZ="0.0" bboxW="40.0" bboxH="40.0">

  </character>

  <scene name="primera" file="pasto.bmp" width="1" height="1"  posX="0" posY="0" posZ="0" scaleX="1"
   scaleY="1" rotAng="0" rotX="0" rotY="0" rotZ="0" mark="Hiro">

  </scene>

</book>
```

Figura 4.19: Archivo de configuración XML inicial

Utilizando este archivo se puede fácilmente modificar la escena que será representada con realidad aumentada, ahora ya se pueden agregar más modelos, posicionarlos en un lugar específico o girarlos por dar un ejemplo. (Fig. 4.20)

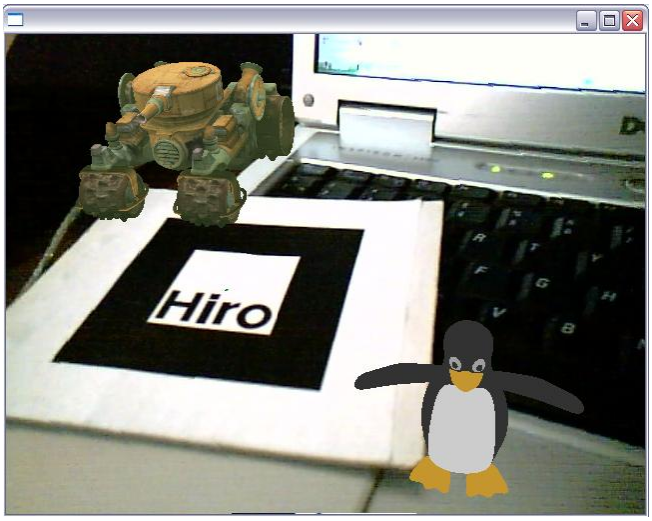

Figura 4.20: Configuración de modelos con XML

- **Agregar hechos y expresiones**

  Hasta el momento se ha logrado incorporar todos los elementos involucrados con la realidad aumentada, los modelos 3D y configurar todo desde un XML, lo siguiente es controlar *los hechos* que estarán relacionados con los patrones de ARToolkit y los *comportamientos* que dichos hechos activarán.

La clase ***Fact*** (Fig. 4.21) representará los hechos, los cuales estarán contenidos en un repositorio o *pool* de donde serán obtenidos cada vez que se necesite un hecho. Los hechos están plenamente identificados gracias a un atributo *id* el cual es único y se asigna al momento de su creación.

```
class Fact{
    private:
        unsigned int id;

        static map<int,Fact*> factPool;
        static unsigned int FACT_COUNTER;

        Fact(int);
    public:
        unsigned int getId();
        static Fact* createFact();
        static Fact* getFact(unsigned int);
};
```

Figura 4.21: Implementación de la clase Fact

Para poder manejar los hechos como expresiones unarias o binarias (como AND, OR y NOT) existe la clase ***Expression*** (Fig. 4.22), ésta se encuentra ampliamente relacionada con la clase *Fact* para poder llevar a cabo expresiones del tipo:

"Si hecho1 y hecho2 entonces acción1" o bien *"Si no hay hecho4 entonces acción2"*.

```
class Expression{
    public:
        enum UNARY_OPERATOR{NONE,NOT};
        enum BINARY_OPERATOR{AND,OR};
    private:
        enum TYPE{CONSTANT,UNARY,BINARY};
        Fact * fact;
        Expression * expA;
        Expression * expB;
        UNARY_OPERATOR unaryOperator;
        BINARY_OPERATOR binaryOperator;
        bool constant;
        TYPE type;

        bool evaluateUnary(map<int,Fact*> *factTable);
        bool evaluateBinary(map<int,Fact*> *factTable);
    public:
        Expression(bool);
        Expression(Fact *);
        Expression(UNARY_OPERATOR,Fact *);
        Expression(Expression *,BINARY_OPERATOR,Expression *);
        bool evaluate(map<int,Fact*>*);
};
```

Figura 4.22: Implementación de la clase *Expression*

Para tal efecto, cada una de estas expresiones se encuentra relacionada con un hecho en particular, es decir, cada objeto de la clase *Expression* contiene un atributo de la clase *Fact* que será el *hecho* al cual la expresión estará ligada en el caso de una expresión unaria. Cuando se trate de una expresión binaria se contará con dos atributos de la clase *Expression* que funcionarán como los dos operandos de la expresión a evaluar, además de contar con un operador binario que las relacione.

Esta clase también cuenta con un método para evaluar si la expresión se cumple, para hacer esto, se busca en una *tabla de hechos* el hecho que está asociado con la expresión, si se encuentra, la expresión se considera como verdadera.

```
class Rule{
    private:
        Command* command;
        Expression * expression;
    public:
        Rule(Expression *,Command*);
        bool evaluate(map<int,Fact*>*);
};

class FactManager{
    private:
        vector<Rule *> rules;
        map<int,Fact*> factTable;

    public:
        FactManager();
        void registerRule(Rule *);
        void addFact(Fact *);
        void removeFact(Fact *);
        void evaluate();
};
```

Figura 4.23: Implementación de la clase Rule y FactManager

La clase ***Rule*** (Fig. 4.23) se utiliza para asociar el cumplimiento de una expresión con una acción o comando, lo cual se servirá para la ejecución de un comportamiento (utilizando el patrón de diseño *Command*). Es decir, si la evaluación de la expresión asociada resulta verdadera entonces se ejecuta el comando asociado.

Por último la clase ***FactManager*** (Fig. 4.23) se encarga de hacer todo el manejo de los hechos, expresiones y reglas. Esta clase se encarga del *registro de hechos* en la tabla de hechos, así como de registrar las reglas que estarán disponibles para evaluar o ejecutar.

- **Ligar hechos con patrones**

  Una vez implementado el manejo de hechos, expresiones y reglas, se procede a *relacionar hechos con patrones* de ARToolkit, es decir, ahora un hecho estará relacionado con el reconocimiento por parte de ARToolkit de uno de los patrones configurados inicialmente. Entonces, en esta iteración se desea que **la aparición de un patrón sea considerado como un hecho** y manejado como tal por las clases correspondientes.

  Para tal efecto, se cuenta con la clase ***PatternsManager***(fig. 4.24) que es la encargada de manejar la relación entre hechos y patrones, cuenta con un mapa de hechos y un mapa de patrones relacionados entre sí.

```
class FindPatternsListener{
    public:
        virtual void findPatternsDone(vector<Pattern*>*)=0;
};

class PatternsManager:public FindPatternsListener{
    private:
        map<int,Fact*> factsMap;
        map<string,Pattern*> patternsMap;
        FactManager* factManager;
    public:
        PatternsManager(FactManager*);
        void findPatternsDone(vector<Pattern*>*);
        void add(Pattern*);
        void add(vector<Pattern*>*);
        Fact* getFact(Pattern*);
        Fact* getFactByPatternName(string);
};
```

Figura 4.24: Implementación de la clase PatternsManager y la interfaz FindPatternsListener

Por otro lado, la clase ***ARToolkitBind*** se encargará de detectar los patrones en el video de la forma habitual y al terminar enviará a la clase ***PatternsManager*** un conjunto de patrones encontrados, estos a su vez, al estar ligados con los hechos iniciarán el proceso de evaluación de las reglas para identificar si con los patrones detectados se cumple alguna o varias de las reglas definidas, si esto sucede se ejecutará entonces el comando o acción específicada en la regla correspondiente.

- **Agregar comportamientos**

  Ahora que ya se cuenta con el manejo de reglas y su relación con los patrones detectados, se requiere contar con **acciones o comportamientos a ejecutar** una vez que se cumple con los hechos definidos en las reglas.

Se tiene entonces la clase ***Behavior*** (Fig. 4.25) que es la clase base de la cual tendrá que heredar cualquier clase que desee implementar un comportamiento específico.

En este caso se desea que los comportamientos estén asociados a modelos 3D que representarán a los personajes o actores de la historia, para eso se crea la clase ***ModelObjectBehavior*** (Fig. 4.25) que relaciona el comportamiento con el modelo que será el que ejecutará dicho comportamiento.

```
class Behavior{
   public:
     enum BehaviorState{ACTIVE,DONE};
     virtual BehaviorState step()=0;
};

class ModelObjectBehavior:public virtual Behavior{
  protected:
    ModelObject* modelObject;
  public:
     ModelObjectBehavior(ModelObject*);

};
```

Figura 4.25: Implementación de las clases Behavior y ModelObjectBehavior

Gracias a esta clase ahora se pueden crear otras clases con comportamientos específicos como la clase *MoveObject* y *SpinObject*, las cuales heredan de *ModelObjectBehavior* para contar con todas las características de un comportamiento y poder ser ligadas con un modelo 3D.

En cada una de estas clases, se configuran parámetros específicos para poder ejecutar las acciones que componen al comportamiento, estas acciones se encuentran definidas en el método *step* de cada clase.

Step es un método que será ejecutado una y otra vez mientras esté activo el comportamiento en cuestión, es decir, cada que se desee ejecutar el comportamiento se hará una llamada al método *step* mientras se tenga un estado *Activo*, cuando se llega al final del comportamiento este estado cambia a *Terminado*.

También se pueden implementar comportamientos más complejos como la clase ***DijkstraBehavior*** (Fig. 4.26), la cual implementa una planeación de rutas utilizando el algoritmo de *Dijkstra* evadiendo obstáculos y buscando caminos mínimos, o la clase ***PtransBehavior*** (Fig. 4.26) que utiliza *DijkstraBehavior* para implementar un comportamiento similar a *PTRANS* de dependencia conceptual.

```
class DijkstraBehavior:public ModelObjectBehavior{
      private:
        vector < vector <DijkstraVertex> > graph;
        Graph *mesh;
        int m,n;
        vector<Coord> path;

      int calculated;
      Coord destination;
      Coord source;
      int current;
      bool loop;
      void Dijkstra(Coord);
      void findPath(Coord,Coord);
      void executePath(Coord);

      public:
       DijkstraBehavior(ModelObject *modelObject);
       void setGraph(Graph *);
       void setDestination(float,float);
       void setSource(float,float);
       Behavior::BehaviorState step();
       void stepNoReturn();
};

class PtransBehavior:public DijkstraBehavior{
      private:
         Object *object;
         Place *place;
      public:
         PtransBehavior(ModelObject *modelObject);
         void setSourcePlace(Place *sourcePlace);
         void setDestPlace(Place *destPlace);
};
```

Figura 4.26: Implementación de las clases DijkstraBehavior y PtransBehavior

Siguiendo este esquema, se pueden implementar tantas clases de comportamientos como se requieran, siempre y cuando dichas clases hereden de la clase *Behavior* e implementen el método *step* con sus correspondientes acciones.

- **Ejecutar secuencia de comportamientos**

  Cabe recordar que se desea que los comportamientos se ejecuten en secuencia, es decir, uno después de otro como en un *guión*. Para esto se crea la clase ***BehaviorSequence*** (Fig. 4.27), la cual hereda de *Behavior* y que a su vez esta compuesta por un conjunto de objetos *ModelObjectBehavior* que serán los comportamientos que se ejecuten en secuencia.

Esta clase simplemente se encargará de recopilar los comportamientos que conforman la secuencia y ejecutarlos en orden, para esto se utiliza el método *step* que por ser método de Behavior deben tener todas las clases que heredan de ésta.

En el caso de *BehaviorSequence*, al ejecutar el método *step*, este mandará ejecutar el método *step* del comportamiento en turno, esto se seguirá haciendo hasta que dicho comportamiento pase de estado *Activo* a *Terminado* y al llegar a dicho estado se comenzará con el *step* del siguiente comportamiento en la secuencia y nuevamente se inicará el ciclo hasta que se hayan ejectutado todos los comportamientos de la secuencia.

```
class BehaviorSequence:public virtual Behavior{
     protected:
          vector<ModelObjectBehavior*> sequence;
          int position;
     public:
          BehaviorSequence();
          void addBehavior(ModelObjectBehavior *);
          Behavior::BehaviorState step();
          void stepNoReturn();
};
```

Figura 4.27: Implementación de la clase BehaviorSequence

- **Última configuración de XML**

Finalmente, se debe poder configurar también desde XML los hechos y comportamientos, así como la secuencia de comportamientos. Para tal efecto se modifica la clase ***XMLConfig*** (Fig. 4.28) para agregar la información necesaria para hacer el *parseo* del documento XML y generar los objetos correspondientes a las clases involucradas con hechos y comportamientos.

Al finalizar esta etapa, la estructura del documento XML queda como se muestra en la Fig. 4.29.

```
class XMLConfig{
   private:
      Scene *myScene;
      char * fileName;
      ARToolkitBind *myARTBind;

      FactManager *myFactManager;
      PatternsManager *myPatternsManager;
      Fact *myFact;
      Command* myCommand;
      Expression* myExpression;
      Rule *myRule;

      map<string,ModelObject *> characterMap;
      map<string, ModelObjectBehavior *> behaviorMap;
      map <string,Fact *> factsMap;

      void leerLibroXML(char *nombreArchivo);
      void addToScene();
      void addToARToolkit();
      void addToFacts();
      void addToBehaviors();

   public:
      XMLConfig();
      void setFileName(char *);
      void setScene(Scene *);
      void setARTBind(ARToolkitBind *);
      void setFactManager(FactManager *);
      void setPatternsManager(PatternsManager *);
      void parse();
};
```

Figura 4.28: Implementación de la clase XMLConfig

```
<?xml version="1.0" encoding="utf-8"?>
<book name="La bella durmiente">
 <mark name="Hiro" ... ></mark>
 <character name="Duke" ...></character>
 <object name="cube1" ...></object>
 <scene name="primera" ...>
   <grid ...></grid>
   <obstacle type="character" ..></obstacle>
 </scene>
 <place name="esquina" ...></place>
 <fact name="Pat1Fact" ...></fact>
 <behavior name="gira"... ></behavior>
 <sequence name="seq1" ...>
  <behavior name="dijkstra" ...></behavior>
 </sequence>
</book>
```

Figura 4.29: Ejemplo de la estructura del documento XML de configuración

4.3. Probando el entorno (DEMO)

Para probar que todos los objetivos principales del framework han sido cubiertos con las iteraciones de desarrollo, se crea una *aplicación DEMO* utilizando la estructura y las clases desarrolladas. La idea básicamente es **generar un libro aumentado utilizando el framework** de manera rápida y sencilla.

Lo primero que se debe hacer es definir la escena donde se desarrollará la acción, es decir, configurar tamaño de la escena, objetos dentro de la escena y lugares clave. Por otro lado también se deben definir los personajes o actores que intervendrán en dicha escena.

Posteriormente se crea un guión en el cual se describen las acciones que cada actor o personaje realizará cuando se presente algún evento como la aparición de marcas específicas.Todo esto en una primera etapa se hace en lenguaje natural, es decir, de manera coloquial, por ejemplo:

«***El Pingüino busca su tanque de juguete, lo busca debajo del escritorio, luego debajo de la cama, finalmente lo busca en un rincón, al encontrarlo lo empuja para jugar con el y lo sigue, cuando llega a alcanzarlo se pone muy feliz y gira de alegría.***»

En una forma más técnica se puede decir:

Actores: Pingüino, tanque de juguete

Lugares clave: Escritiorio, cama, esquina superior derecha, esquina inferior izquierda

Comportamientos:

El pingüino va del escritorio a la cama

El pingüino va de la cama a la esquina superior derecha

El tanque va de la esquina superior derecha a la esquina inferior izquierda

El pingüino va de la esquina superior derecha a la esquina inferior izquierda

El pingüino gira al llegar

Una vez hecha esta separación del guión en pequeñas acciones, se puede representar con dependencia conceptual de la siguiente manera:

(PTRANS (actor pingüino) (object pingüino) (from escritorio) (to cama))

(PTRANS (actor pingüino) (object pingüino) (from cama) (to esquina))

(PTRANS (actor tanque) (object tanque) (from esquina) (to otraEsquina))

(PTRANS (actor pingüino) (object pingüino) (from esquina) (to otraEsquina))

(MOVE (actor pingüino) (object cuerpo) (from quieto) (to gira))

Para terminar de transformar este script en un documento XML se puede utilizar algún editor de texto que pueda manejar expresiones regulares y con ello generar el siguiente fragmento de XML (Fig. 4.30):

```
- <sequence name="seq1" fact="KanjiFact">
    <behavior name="ptrans1" type="PtransBehavior" character="Tux" object="Tux" source="escritorio" dest="cama" />
    <behavior name="ptrans2" type="PtransBehavior" character="Tux" object="Tux" source="cama" dest="esquina" />
    <behavior name="ptrans3" type="PtransBehavior" character="Tank" object="Tank" source="esquina" dest="otraesquina" />
    <behavior name="ptrans4" type="PtransBehavior" character="Tux" object="Tux" source="esquina" dest="otraesquina" />
    <behavior name="gira" type="GiraObjeto" character="Tux" axisX="0" axisY="0" axisZ="1" />
  </sequence>
```

Figura 4.30: Script convertido a XML

Finalmente se debe convertir todo a **un documento XML** que representará esta información, al hacer esto dicho documento queda como se muestra en la figura 4.31.

Con esta sencilla configuración del documento XML se logra generar la escena descrita anteriormente con comportamientos agredados a los personajes que aparecen en ella, además de que se tiene interacción con el usuario ya que la aparición de un patrón en escena desencadenará un comportamiento, ésto será interpretado por el usuario como interacción con la escena en cuestión.

```
<?xml version="1.0" encoding="utf-8" ?>
<book name="La bella durmiente">
  <mark name="Hiro" file="Data/patt.hiro" width="80.0" centerX="0.0" centerY="0.0" />
  <mark name="Kanji" file="Data/patt.kanji" width="80.0" centerX="0.0" centerY="0.0" />
  <mark name="Pat1" file="Data/patt.sample1" width="80.0" centerX="0.0" centerY="0.0" />
  <mark name="Pat2" file="Data/patt.sample2" width="80.0" centerX="0.0" centerY="0.0" />
  <character name="Duke" file="Duke.obj" scaleX="20" scaleY="20" scaleZ="20" rotAng="90.0" rotX="1.0" rotY="0.0" rotZ="0.0"
    posX="0.0" posY="-30.0" posZ="0.0" bboxW="40.0" bboxH="40.0" />
  <character name="Escritorio" file="memex.obj" scaleX="20" scaleY="20" scaleZ="20" rotAng="90.0" rotX="1.0" rotY="0.0"
    rotZ="0.0" posX="70.0" posY="-70.0" posZ="0.0" bboxW="20.0" bboxH="20.0" />
  <character name="Tux" file="Tux2.obj" scaleX="20" scaleY="20" scaleZ="20" rotAng="90.0" rotX="1.0" rotY="0.0" rotZ="0.0"
    posX="80.0" posY="-120.0" posZ="0.0" bboxW="40.0" bboxH="40.0" />
  <character name="Tank" file="Tank.obj" scaleX=".20" scaleY=".20" scaleZ=".20" rotAng="90.0" rotX="1.0" rotY="0.0" rotZ="0.0"
    posX="100.0" posY="90.0" posZ="0.0" bboxW="40.0" bboxH="40.0" />
  <character name="Cama" file="cama.obj" scaleX="1" scaleY="1" scaleZ="1" rotAng="90.0" rotX="1.0" rotY="0.0" rotZ="0.0"
    posX="-70.0" posY="-200.0" posZ="0.0" bboxW="20.0" bboxH="20.0" />
  <character name="Maceta" file="pote_flor_56.obj" scaleX="1" scaleY="1" scaleZ="1" rotAng="90.0" rotX="1.0" rotY="0.0"
    rotZ="0.0" posX="100.0" posY="-220.0" posZ="0.0" bboxW="20.0" bboxH="20.0" />
  <object name="cube1" type="Cube" scaleX="20" scaleY="20" scaleZ="20" rotAng="0.0" rotX="0.0" rotY="0.0" rotZ="0.0"
    posX="0.0" posY="0.0" posZ="20.0" bboxW="40.0" bboxH="40.0" colorR="0" colorG="0" colorB="1" />
- <scene name="primera" file="Piso4.bmp" width="5" height="5" posX="0" posY="0" posZ="0" scaleX="50" scaleY="50" rotAng="0"
    rotX="0" rotY="0" rotZ="0" mark="Hiro">
    <grid m="11" n="11" d="20.0" offsetX="-100.0" offsetY="-100" />
    <obstacle type="character" name="Duke" />
    <obstacle type="object" name="cube1" />
    <obstacle type="character" name="Escritorio" />
  </scene>
  <place name="esquina" posX="100.0" posY="100.0" posZ="0.0" />
  <place name="otraesquina" posX="-100.0" posY="-100.0" posZ="0.0" />
  <fact name="Pat1Fact" mark="Pat1" />
  <fact name="KanjiFact" mark="Kanji" />
  <fact name="Pat2Fact" mark="Pat2" />
  <behavior name="gira" type="GiraObjeto" character="Duke" fact="Pat1Fact" axisX="0" axisY="0" axisZ="1" />
- <sequence name="seq1" fact="KanjiFact">
    <behavior name="dijkstra" type="DijkstraBehavior" character="Tux" sourceX="80.0" sourceY="-100.0" destX="-100.0"
      destY="80.0" />
    <behavior name="dijkstra2" type="DijkstraBehavior" character="Tux" sourceX="-100.0" sourceY="80.0" destX="100.0"
      destY="80.0" />
    <behavior name="ptrans1" type="PtransBehavior" character="Tank" object="Tank" source="esquina"
      dest="otraesquina" />
    <behavior name="ptrans2" type="PtransBehavior" character="Tux" object="Tux" source="esquina" dest="otraesquina" />
    <behavior name="gira" type="GiraObjeto" character="Tux" axisX="0" axisY="0" axisZ="1" />
  </sequence>
</book>
```

Figura 4.31: Documento XML para configuración del DEMO

Para probar la aplicación que será generada con el XML anterior, se necesita contar con una cámara web, las marcas de ARToolkit y ejecutar la aplicación indicando el nombre del archivo xml como se muestra en la fig. 4.32.

Figura 4.32: Configuración del equipo para ejecutar la aplicación DEMO

Al ejecutarse esta aplicación se visualiza inicialmente como en la fig. 4.33. En esta primera imágen, al ser detectado el patrón base de la escena, se generan los gráficos de los personajes y los objetos de la escena tal cual fueron definidos en el XML, en la posición correspondiente con respecto al video que se esta capturando en tiempo real.

Figura 4.33: Ejecución del Demo: Patrón base detectado y escena generada

Al detectar un patrón que tiene un comportamiento ligado, éste comportamiento comienza a ejecutarse, para el caso de las figuras 4.34, 4.35, 4.36, 4.37 y 4.38 al detectarse el patrón encerrado en el rectángulo azul, se ejecuta un comportamiento de tipo *BehaviorSequence* el cual corresponde al guión que se describió anteriormente, con esto se van efectuando los comportamientos uno después de otro.

En el caso de la figura 4.39 al detectar otro patrón (también encerrado en el rectángulo azul) se ejecutará el comportamiento de tipo *GiraObjeto* que fue ligado con dicho patrón.

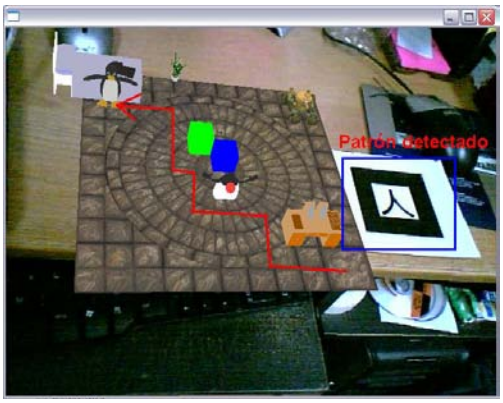

Figura 4.34: Ejecución del Demo: Patrón kanji detectado, inicia ejecución de la secuencia de comportamientos *BehaviorSequence* (script)

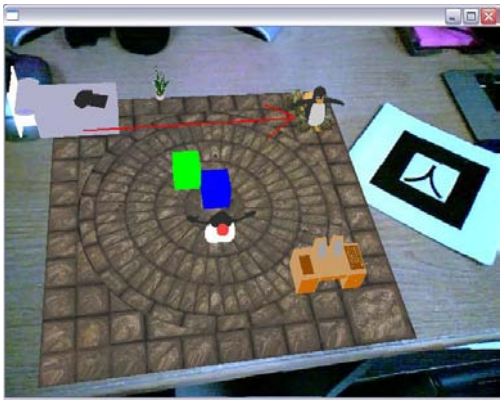

Figura 4.35: Ejecución del Demo: Segundo comportamiento de la secuencia

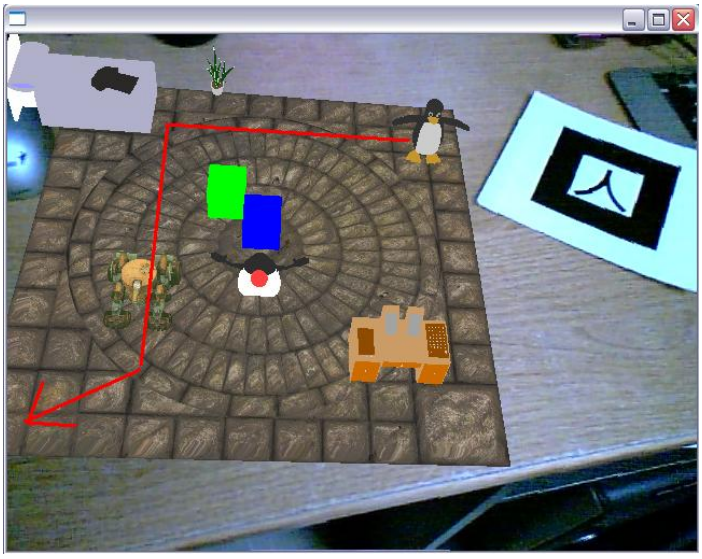

Figura 4.36: Ejecución del Demo: Tercer comportamiento de la secuencia

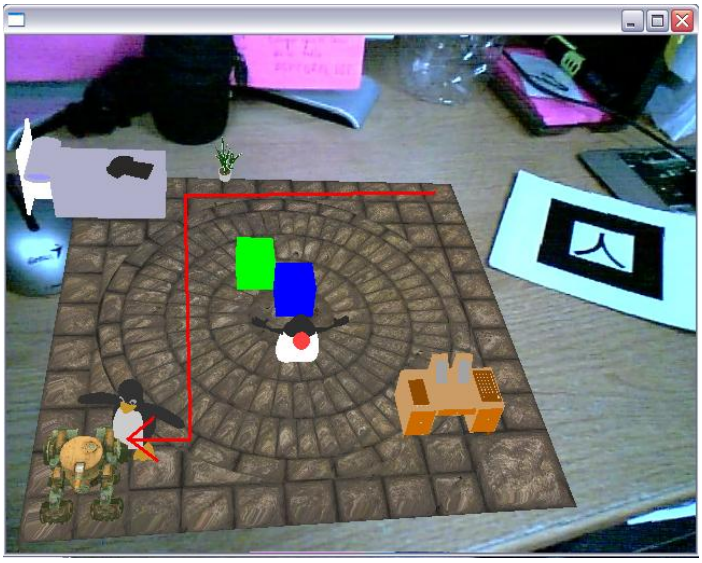

Figura 4.37: Ejecución del Demo: Cuarto comportamiento de la secuencia

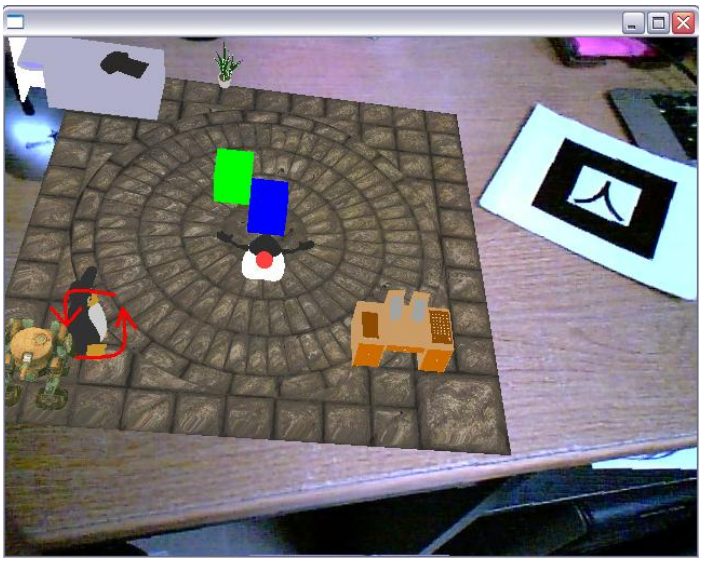

Figura 4.38: Ejecución del Demo: Quinto comportamiento de la secuencia

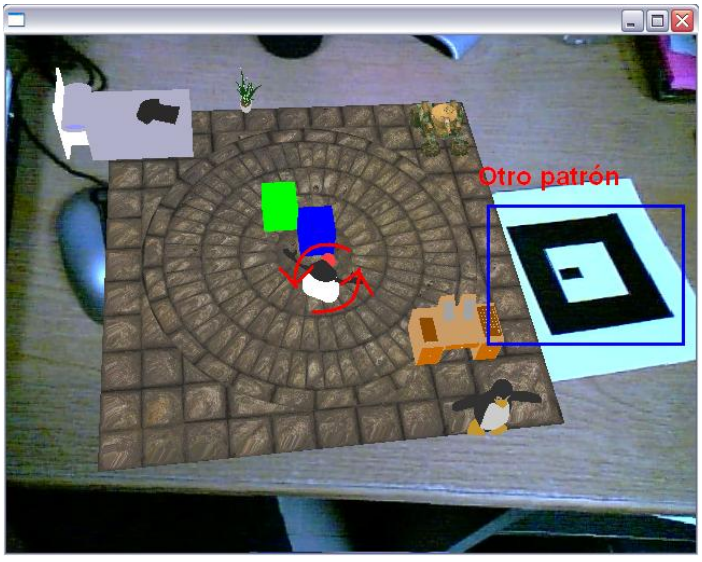

Figura 4.39: Ejecución del Demo: Otro patrón detectado, ejecuta comportamiento GiraObjeto

Conclusiones y Trabajo Futuro

En este momento ya se tiene terminado el desarrollo del entorno y se ha creado una aplicación DEMO con el fin de mostrar la utilidad del Framework, ahora sólo falta analizar *si los objetivos iniciales del proyecto fueron cubiertos y de que manera.* Por otro lado, también se debe pensar en las *mejoras o complementos* que se pueden desarrollar en el futuro con el objeto de facilitar y promover su utilización para generar más y mejores aplicaciones con esta tecnología.

Conclusiones

Al desarrollar la *aplicación DEMO* se pudo observar como con el uso de este framework, la creación de libros aumentados es *muy sencilla y rápida* lo cual es uno de los objetivos principales de este trabajo. Con el uso de este framework sólo se requiere dar valores a los parámetros de un *documento XML*, entonces una persona sin conocimientos en programación puede generar estas aplicaciones con solo crear dicho *XML* sin tener que saber como funciona la tecnología de realidad aumentada ni como se realiza el proceso gráfico. Esta misma persona no habría sido capaz de lograr esto sin la ayuda de programadores experimentados en el campo lo cual involucraría mas tiempo y recursos.

Con el desarrollo de este framework se pone la tecnología de los libros aumentados a dispocición de muchas más personas de incluso otras áreas como por ejemplo a *diseñadores gráficos*, *escritores de cuentos*, *pedagogos*, *etc.* ya que al simplificar la creación de estos libros es más factible que todas estas personas se involucren en la creación de aplicaciones sin toparse con la barrera tecnológica que muchas veces entorpece el uso de tecnologías innovadoras.

Por otro lado, *la mayoría de las aplicaciones gráficas* se desarrollan sin utilizar *patrones de diseño* o técnicas similares, entonces sólo los programadores con conocimientos de graficación por computadora son los únicos que pueden crear este tipo de aplicaciones siendo complicado que otros programadores puedan también crear estas aplicaciones. Con este framework se logra bastante *independencia* dado que su utilización es muy sencilla y pueden crearse clases que extiendan las funcionalidades del mismo con lo que tiene mucha flexibilidad, además de que no es necesario tener grandes conocimientos de computación gráfica para poder lograr aplicaciones con *modelos 3D*.

La *Realidad Aumentada* no es muy utlizada en nuestro país ya que las bibliotecas de las que se dispone son un poco complicadas de entender y si no se tiene el conocimiento técnico necesario no es posible crear aplicaciones con esta tecnología, negando con esto la posibilidad a *pedagogos*, *escritores de cuentos*, *diseñadores* y en general las personas involucrads con los cuentos, la posibilidad de auxiliarse y beneficiarse de ella. Ahora, con la creación de el framework es posible que todos ellos puedan utilizarla.

Además, el *tiempo* que se tiene que invertir en el desarrollo de un libro aumentado, en el aspecto *técnico*, se reduce a generar un *documento XML*, por lo que la mayor parte del tiempo que se invierte se destina principalmente al *proceso creativo y artístico*. Ahora los creadores de aplicaciones tendrán ahora más tiempo para diseñar modelos 3D para representar personajes, planear las escenas, crear una historia o script y con ello crear mejores aplicaciones.

En conclusión se puede decir que el objetivo principal *se cumplió* ya que ahora esta tecnología podrá ser utilizada con mucho mayor facilidad por personas que no poseen gran conocimiento técnico de como funciona la realidad aumentad y la computación gráfica, ni de como programarla. Por otro lado los desarrolladores de aplicaciones pueden extender el framework y crear nuevas funcionalidades de acuerdo a las necesidades de los usuarios ya que por su diseño y estructura bien definida se facilita esta tarea.

Trabajo Futuro

Ahora que se ha mostrado la utilidad del framework, se puede pensar en una variedad de **mejoras y complementos** que se pueden incluir posteriormente, esto con el objetivo de simplificar y facilitar aún más su uso y acercarlo a una mayor cantidad de *desarrolladores, diseñadores y usuarios.*

Algunas de estas mejoras y complementos son:

- ***GUI para generar el XML***

  Desarrollar una *GUI (Graphic User Interface*) con cualquier lengueje de programación que permita configurar el documento XML de manera más fácil al diseñador de libros aumentados. Esta interfaz gráfica de usuario generará la información requerida para el libro, lo único que requerirá será que el usuario *llene campos* con valores válidos. Una vez hecho esto, dicha información se transformará en un *XML* con la estructura necesaria para la creación de dicho libro aumentado utilizando el framework creado.

- ***Liga con reconocedor de lenguaje natural***

  Se puede utilizar algún programa ya desarrollado que reconozca *lenguaje natural* y lo convierta en *dependencia conceptual* para generar *scripts* que posteriormente sean convertidos a XML con el formato requerido, esto con el objetivo de que la creación de libros aumentados sea una tarea que pueda ser realizada sin tener que escribir los valores en una computadora, si no que simplemente le sean dictadas las historias en las que estan basados y genere el XML que las represente.

- ***Añadir más comportamientos***

  Se pueden *crear nuevas clases* para lograr que las acciones que realizan los personajes en los *scripts* sean más variadas o que incluso exista mayor interacción entre ellos o con el usuario. Debido al diseño con el que cuenta el framework, éste *soporta ser exitendido* creando nuevas clases de comportamientos siempre y cuando se respete la estructura y el diseño.

- ***Añadir modelos 3D con otros formatos y/o animación***

  Otra mejora que se le puede hacer al framework es crear clases para generar *modelos 3D* con información contenida en archivos con otros formatos distintos al *OBJ* o incluso *formatos de animación (como VRML).* Con esto se logrará mejorar la apariecnia y la compatibilidad del framework con otras aplicaciones de modelado 3D y las animaciones harán que el efecto de los comportamientos sea mucho más fluido y entendible.

- ***Utilización de CLIPS***

  Para que el manejo de hechos y reglas soporte *mayores y más complejas reglas* se pueden crear clases para utilizar el *motor de hechos de CLIPS* en vez del propio, para esto pueden utilizarse bibliotecas o clases ya desarrolladas para este fin y simplemente adaptarlas con *clases envolventes* para que la interacción entre estas clases y el framework sea transparente y sencilla.

Con todas estas mejoras se logrará que el uso de la tecnología de libros aumentados sea más facilmente aceptada y utilizada por la mayor cantidad de *desarrolladores*, *diseñadores*, *animadores*, *escritores*, *pedagogos*, *profesores*, o *simples usuarios* que deseen experimentar los libros aumentados y pueda llegar a una mayor audiencia, llevando con esto a una mejor comprensión de las historias representadas en los libros apoyando al aprendizaje.

Bibliografía

[Ada04] Mike Adams. Top ten technologies: #3 augmented reality. Natural News, Sitio de internet en línea. Disponible en `http://www.naturalnews.com/001333.html`, julio 2004.

[AGF95] Jr Anthony G. Francis. A pocket guide to cd theory. Technical report, College of Computing Georgia Insitute of Technology, `http://www.cc.gatech.edu/computing/classes/cs3361_96_spring/Fall95/Notes/cd.html`, 1995.

[Azu97] Ronald T. Azuma. A survey of augmented reality. *Presence: Teleoperators and Virtual Environments*, 6(4):355–385, Agosto 1997.

[BE93] Andreas Birrer and Thomas Eggenschwiler. Frameworks in the financial engineering domain - an experience report. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 21–35, London, UK, 1993. Springer-Verlag.

[Bil02] Mark Billinghurst. Augmented reality in education. New Horizons for Learning, Sitio de internet en línea. Disponible en `http://www.newhorizons.org/strategies/technology/billinghurst.htm`, Diciembre 2002.

[Cor90] Thomas H. Cormen. *Introduction to Algorithms*. Mc Graw Hill, second edition, 1990.

[Deu89] L. P. Deutsch. Design reuse and frameworks in the smalltalk-80 system. In *Software reusability: vol. 2, applications and experience*, pages 57–71, New York, NY, USA, 1989. ACM.

[DH07] Andreas Dünser and Eva Hornecker. *Technologies for E-Learning and Digital Entertainment*, chapter An Observational Study of Children Interacting with an Augmented Story Book, pages 305–315. Lecture Notes in Computer Science, 2007.

[Fow99] Martin Fowler. *UML gota a gota*. Addison Wesley Longman de México, 1999.

[FVDFH97] James D. Foley, Andries Van Dam, Steven K. Feiner, and John Hughes. *Computer Graphics: Principles and Practice. Second edition in C*. The Systems Programming Series. Addison Wesley, Estados Unidos, 1997.

[Gam04] Erich Gamma. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 2004.

[GDSB07] Raphael Grasset, Andreas Dünser, Hartmut Seichter, and Mark Billinghurst. The mixed reality book: a new multimedia reading experience. In *CHI '07: CHI '07 extended abstracts on Human factors in computing systems*, pages 1953–1958, New York, NY, USA, 2007. ACM.

[Gha04] Malik Ghallab. *Automated Planning Theory and Practice*. Morgan Kauffmann Publishers, Mayo 2004.

[GL95] Ian Graham and Quin Liam. *XML Specification Guide*. Wiley Computer Publishing, U.S., 1995.

[GR94] Joseph Giarratano and Gary Riley. *Expert Systems: Principles and Programming*. PWS Publishing Company, second edition, 1994.

[GRSS07] Scott Green, Scott Richardson, Vadim Slavin, and Randy Stiles. Spatial dialog for space system autonomy. In *HRI '07: Proceedings of the ACM/IEEE international conference on Human-robot interaction*, pages 341–348, New York, NY, USA, 2007. ACM.

[HB07] Anders Henrysson and Mark Billinghurst. Using a mobile phone for 6 dof mesh editing. In *CHINZ '07: Proceedings of the 7th ACM SIGCHI New Zealand chapter's international conference on Computer-human interaction*, pages 9–16, New York, NY, USA, 2007. ACM.

[HBO06] Anders Henrysson, Mark Billinghurst, and Mark Ollila. Ar tennis. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 13, New York, NY, USA, 2006. ACM.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented*, 1(2):22–35, 1988.

[Joh92] Ralph E. Johnson. Documenting frameworks using patterns. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 63–76, New York, NY, USA, 1992. ACM.

[KB07] Hirokazo Kato and Mark Bllinghurst. Artoolkit, sitio de internet en línea. disponible en `http://sourceforge.net/projects/artoolkit/`. 2007.

[KD07] Hannes Kaufmann and Andreas Dünser. *Virtual Reality*, chapter Summary of Usability Evaluations of an Educational Augmented Reality Application, pages 660–669. Lecture Notes in Computer Science, 2007.

[Led04] Florian Ledermann. An authoring framework for augmented reality presentations. Master's thesis, Institut für Softwaretechnik der Technischen Universität Wien, 2004.

[MC06] Jeffrey J. Mc. Connell. *Computer Graphics Theory into practice*. Jones and Bartlett Publishers, 2006.

[Mil06] Ian Millington. *Artificial Intelligence for Games*. Morgan Kaufmann Publishers, 2006.

[Ous98] John K. Ousterhout. Scripting: Higer level programing for the 21st century. *IEEE Computer Magazine*, 31:23–30, Marzo 1998.

[Rie62] Arthur J. Riel. *Object-Oriented design heuristics*. Addison Wesley, 1962.

[Sch77] Roger Schank. *Scrpts Plans Goals and Understanding*. Lawrence Eralbaum Associates Publishers, 1977.

[SS05] Harmut Seichter and Marc Aurel Schanabel. Digital and tangible sensation: An augmented reality urban design studio. In *Proceedings of The Tenth Conference on Computer-Aided Architectural Design Research in Asia (CAADRIA 2005)*, pages 191–202, New Dehli, India, 2005.

[SW04] Dave Shreiner and Mason Woo. *OpenGL Programming Guide*. Pearson Education, 2004.

[VL90] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific. *ACM Transactions on Information Systems*, July 1990.

[WBS06] Daniel Wagner, Mark Billinghurst, and Dieter Schmalstieg. How real should virtual characters be? In *ACE '06: Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology*, page 57, New York, NY, USA, 2006. ACM.