UNIVERSIDAD NACIONAL
AUTÓNOMA DE MÉXICO
Facultad de Ingeniería

VISUALIZACIÓN DE OBJETOS EN 3D EMPLEANDO REALIDAD AUMENTADA Y *SHADERS* PROGRAMABLES

TESIS
Que para obtener el Título de
INGENIERO EN COMPUTACIÓN

Presenta
Javier Jiménez Juárez

Director: Ing. Emmanuel Hernández Hernández

México, D.F., 2004

DEDICATORIA

A mis padres, que siempre han estado cerca de mi, apoyándome en todo.

Es poco para pagarles todo lo que han hecho por su hijo, pero es algo que he realizado con base en mi esfuerzo y que les dedico con todo mi corazón

AGRADECIMIENTOS

He aquí el trabajo que representa el fin de un ciclo. Un ciclo no solo de cinco años o más en la escuela, en la Universidad. Un periodo de tiempo en el cual conocí muchas personas e hice amistades, periodo en el que la vida algunas veces me trató bien y otras me dio duros golpes. Espacio temporal en el que aprendí muchas cosas, cambié mi punto de vista sobre algunas y madura sobre otras.

Esta tesis lleva al título mi nombre, pero el realizador no he sido solo yo. En el desarrollo de ella se encuentran involucradas muy valiosas personas: amigos, parientes, amigos que se han convertido en verdadera familia. Decir que yo solo he podido escribir, programar e hilar ideas al respecto sería soberbio. Ahora que escribo las hojas iniciales de la tesis, que en realidad son las últimas, pasan por mi mente momentos y tiempo transcurrido al lado de cada uno de ustedes. Instantes de alegría, de preocupaciones por exámenes y situaciones de la vida, instantes de confusión, de dudas, de decisiones. Cada uno de ellos tiene una marca especial y una etiqueta única, que hacen inconfundibles e inolvidables el tiempo pasado, que sin lugar a dudas, ha valido la pena porque en él han estado siempre ustedes acompañándome.

Todo toma un rumbo y todos tomamos un camino. Tal vez nuestros caminos se diverjan ahora, pero tengan por seguro que todos y cada uno de ustedes estarán por siempre presentes en mí. Podremos estar lejos, sin vernos, incluso sin hablar o tener noticias de nosotros pero la huella y los lazos que nos unen jamás nos abandonarán, simplemente existirán y existiremos por la amistad que nos liga.

Tengo tantas cosas que decir y tan poco espacio en el que escribir. Saben lo que significan para mí y que nunca podré terminar de agradecer a Dios por permitirme haberlos conocido. Mejor comenzaré con algunos agradecimientos individuales.

A Rodolfo Jiménez Martínez y Dominga Juárez Alonso, mis padres y a quienes debo la vida. No solo la vida, debo todo, mi educación, los valores que me han dado, la capacidad de decisión sobre esos momentos importantes en la vida y libertad como ser humano. Por la confianza que siempre han depositado en mí y el orgullo que sienten tanto por mi como por mi hermano. Menciono esa confianza pues no perdieron la fe a pesar de que fui necio y no quise ir al kinder. Los momentos de enseñanza en los que tu mamá, me enseñabas pacientemente a escribir, a leer, las tablas de multiplicar y mis $100 pesos condicionados para ir a jugar a las maquinitas. A ti papá por todos esos momentos en que me ayudabas y enseñabas con calma y hasta que entendía química, física, matemáticas, dibujo técnico e innumerables cosas. Gracias por todo lo que me han dado.

A Rodolfo Jiménez Juárez, mi hermano. Tu sabes lo importante que eres para mi. Si no hubiera sido por ti, no habría incursionado en ninguna actividad deportiva. Las pocas o muchas mañas que conozco del fútbol te las debo a ti. Gracias por aquellos momentos en los que me explicabas pacientemente anatomía y me ayudabas a preparar mi examen final en la prepa.

A Rocío, Rodrigo, Emilio y Diego, mi cuñada y mis tres sobrinos. Gracias porque con ustedes he aprendido a valorar aun mas lo que es una familia y la felicidad que puede traer consigo un niño, o dos o tres. Gracias también por permitirme ser padrino de ese niño geniudo pero lindo, Emilio.

A mis abuelos, tíos, tías, primos y primas. Gracias por todos el tiempo compartido, por la confianza depositada en mí y el cariño que me tienen. Su nieto y sobrino consentido los recuerda siempre a pesar de que a veces les quede mal.

Amigos de la prepa y de la Universidad: Juan Eduardo González, Iván Urzúa, Luis Godínez, Gerardo Caravantes, Armando Ávila, Sankarsana Romero, Uriel Nava, Carlos Soster. Cuantas fiestas memorables, juegos en los que nos fue bien y otros en lo que nos fue mal. Las fracturas por las que han pasado algunos. Gracias pos su amistad, apoyo y aguante durante todo este tiempo.

Aquí es difícil enumerar, pues he pasado al punto donde nombraré la gente de la Facultad, y es que son tantos. Quiero y debo agradecer a Juan Monter, Miguel Troncoso, Pedro León, Nadir Castañeda, Alberto Escalante, Enrique y José Larios por todas esas tardes en las que estudiábamos y pretendíamos realizar series. Y digo pretendíamos porque uno la hacia y los demás copiábamos. Bueno, no siempre pero en ocasiones así era. También por esos días en los que no dormíamos y trabajábamos como negros en vísperas de la entrega de proyectos. Como olvidar también los geniales comentarios de Nadir, gracias a él tengo mi cuenta de correo.

En otro grupo están los industriales, electrónicos y algunos otros de telecomunicaciones. Coral Donají, Rodrigo Vera, Carina Bentrup, Marcos González, Verónica Villanueva, Helia González, Eric Ríos, Arely Vergara, Ricardo Carmona, Yahvé Ledezma, Jorge Pérez. Recuerdo bien ese viaje a San Antonio, algunos fuimos a Veracruz y nos divertimos en los rápidos. ¿Y que tal ese viaje a Chiapas? Que buenos momentos. Con algunos otros las muy buenas cáscaras de fútbol en el anexo o algunas partidas de XBox.

Agradezco de manera individual a Alejandro Bustamante por ser mi mejor amigo. Por todos esas horas, minutos, segundos en los que ha estado junto a mí, siempre apoyándome. Por aguantarme aun en los momentos en que mas necio me ponía, por soportar y darme ánimos cuando algo me hacía sentir mal o me pasaba algo. Gracias Alejandro, gracias Bus. Tu sabes que no necesito decir nada mas, pues no existen palabras con las cuales agradecer tu amistad.

Quiero también agradecer a mi otro mejor amigo, más reciente pero no por ello menos mejor amigo Miguel Ángel Guillén Torres. Gracias por todo ese tiempo compartido, saberme escuchar y quitarme el tiempo. No olvidaré esas tardes completas "invertidas" en ir a comer con el chino o que te acompañará a Perisur, o que aquí, allá o acullá. En serio, gracias porque has demostrado ser alguien en quien se puede confiar y platicar de cosas de real importancia, sin preocuparse del que dirán, de críticas o prejuicios. Gracias por tenerme tanta confianza y ser tan buen amigo.

Otra persona a quien debo agradecer es a Hector Yuen, de quien recibí gran ayuda en la última etapa de la tesis (en buscar sinodales, trámites, etc). También por ser el tipo de persona que es, de ser siempre alegre y más mexicano que chino.

Mara y Paola. Ustedes no se conocen pero tienen algo en común. Tal vez no hemos pasado mucho tiempo juntos. A ti Paola cuanto tiene que no te veo y tu Mara, realmente nuestra amistad tiene poco pero ha sido de lo más sincera y natural que se puede gestar. Las dos merecen que les agradezca por todo su cariño y amistad que me han otorgado durante el tiempo que nos conocemos. Gracias.

Agradezco también a mi mejor amiga, mi hermana Ishtar. Fue realmente curiosa la manera en que nos conocimos. Han pasado tantas cosas y seguirás pasando tantas otras. Solo quiero decir que el tiempo que ha transcurrido en el que nos hemos conocido y convivido bien ha valido la pena porque tu siempre has estado ahí escuchándome, apoyándome, haciéndome ver las cosas con otro cristal. En ocasiones, jalándome las orejas para que recapacite sobre lo que estoy haciendo. No tengo más que decir Ishtar, solamente quiero agradecerte todo lo que has hecho por mi y tu amistad sincera y desinteresada. El tiempo vivido junto a ti ha valido realmente la pena.

Nallely Lara, si, Nallelyta, ¿cómo crees que me podría haber olvidado de ti? No puedo hacerlo. Tu mereces especial mención en estos agradecimientos. Si, tú que nunca de los nuncas dejaste de creer en mí, tú que siempre me has tenido en una estima muy alta. Muchas gracias por todos estos años en los que he disfrutado de tu generosa y sincera amistad. Muchas gracias por todo ese cariño que me demuestras cada vez que nos vemos o hablamos. Sin tu confianza depositada en mí, habría sido muy difícil concluir esta tesis. Jamás dejaste que me cayera, siempre me dabas palabras de aliento y me impulsabas a seguir adelante, no solo con la tesis sino con cada uno de los proyectos que me he propuesto. No puedo decirte otra cosa más que gracias y estar agradecido con Dios por haberte conocido.

Por último me queda agradecer al staff técnico, es decir, a las personas que se encuentran detrás de la realización de esta tesis. En primero lugar está el Dr. Jesús Savage Carmona que fue mi tutor mientras estaba en la Facultad y que ahora es mi jefe pero mas que eso, es alguien a quien puedo considerar un amigo. Gracias por permitirme laborar, aprender y continuar aprendiendo de usted en todos los aspectos.

En segundo lugar agradecer a Emmanuel Hernández Hernández, mi director de tesis y amigo. Alguien de quien he recibido ayuda, consejos, en fin. Tu sabes que mas que jefe o director de tesis eres en primer lugar un amigo. Muchas gracias por todo lo que has hecho por mí.

Quiero también agradecer al personal de laboratorio de Interfaces Inteligentes, Laura, Andy, Sergio, Humberto y Gabriel. De todos ustedes he aprendido algo, digamos que desde cine hasta artes ocultas y magia negra.

Alguien a quien no puedo dejar de agradecer es a Dios, quien siempre me ha ayudado. Gracias por darme confianza y ánimo cada vez que me he sentido derrotado. Gracias por escucharme siempre.

Finalmente, agradezco infinitamente al Universidad Nacional Autónoma de México, mi alma mater. Máxima casa de estudios en México y América Latina. Me siento muy orgulloso de pertenecer a ella y comprometido a dar lo mejor de mí cada día para llevar su nombre muy en alto. Gracias por todo el conocimiento que he recibido dentro y fuera de las aulas, por la diversidad de gente y por ser lo más grande que hasta ahora he conocido.

ÍNDICE

Índice de figuras v
Índice de tablas ix
Introducción xi

Capítulo I: Conceptos Generales

1.1. Interfaces Hombre-Máquina (*HCI*) 3
1.2. Realidad Virtual 5
1.2.1. Breve historia 6
1.2.2. Clasificación de mundos virtuales 7
1.2.3. Dispositivos de salida usados en aplicaciones de Realidad Virtual 8
1.2.3.1. Dispositivos de presentación 8
1.2.3.2. Dispositivos de audio 9
1.2.3.3. Otros dispositivos de salida 10
1.2.4. Dispositivos de entrada usados en aplicaciones de Realidad Virtual 10
1.2.4.1. Dispositivos de localización 11
1.2.4.2. Dispositivos de control 11
1.2.5. Creación de un mundo virtual 12
1.3. Realidad Aumentada 12
1.3.1. Aplicaciones de la AR 13
1.3.1.1. Medicina 13
1.3.1.2. Entretenimiento 13
1.3.1.3. Milicia 14
1.3.2. Sistema típico de *AR* 14
1.3.3. Rendimiento de un sistema de *AR* 15

1.3.4. Tecnologías de despliegue en *AR* 15
1.4. Visualización 17
1.5. Reconocimiento de patrones 18
1.5.1. Aplicaciones muy eficientes en campos muy concretos 19
1.6. Sombreadores programables (*Shaders*) 19
Resumen 21

Capítulo II: *DirectX*

2.1. *DirectX Graphics* o *Direct3D* 26
2.1.1. *Direct3D HAL* 26
2.1.2. Dispositivos de software conectable 27
2.1.3. *Reference Rasterizer* 28
2.1.4. Modelo de Componentes de Objetos (*COM*) 28
2.1.5. *Pipeline* de *Direct3D* 28
2.2. *DirectShow* 32
2.2.1. Arquitectura de *DirectShow* 33
2.2.2. Manejador de filtros y *FilterGraph* 35
2.3. *Shaders* programables para *DirectX Graphics* 36
2.3.1. Unidad de *Vertex Shader* 37
2.3.1.1. Arquitectura de un *Vertex Shader* 36
2.3.2. Unidad de *Píxel Shader* 39
2.3.2.1. Arquitectura de un *Píxel Shader* 40
Resumen 41

Capítulo III: Editor de *MagicBooks*

3.1. ¿Por qué un editor de contenido para el *MagicBook*? 45
3.2. Características del editor: *BooX' Wizard* 46
3.2.1. Área principal de despliegue de modelos 47
3.2.2. Menú 47
3.2.3. Controles para la carga de modelos 48
3.2.4. Selección del tipo de render 48
3.2.5. Trabajo actual 49
3.2.6. Selección de patrones 49
3.2.7. Selección de *shader* 50
3.3. Funcionamiento del editor 50
3.3.1. Inicialización de *Direct3D* 51
3.3.2. Inicialización del entorno 51

3.3.3. Procedimiento de *render* 53
3.3.4. Interfaz de control de transformaciones mediante el ratón 56
3.3.4.1. Esquema general de la clase *CD3DarcBall* 56
3.3.4.2. Escalamiento 57
3.3.4.3. Rotaciones 58
3.3.4.4. Traslaciones 60
Resumen 61

Capítulo IV: *MagicBook*

4.1. Cambios realizados en comparación con la versión de *OpenGL* 65
4.2. Captura de video 69
4.3. Empleo de las rutinas del *ARToolKit* 71
4.4. Proceso de carga de libros 72
4.5. Ajustes efectuados a las transformaciones 73
4.6. *Render* de la escena 78
4.6.1. Reconocimiento de los patrones 78
4.6.2. Video 79
4.6.3. Modelos 80
Resumen 82

Capítulo V: *Shaders*

5.1. Especificación de un *shader para Direct3D* 85
5.1.1. Verificación de soporte en Hardware para *Vertex* y *Pixel Shader* .. 86
5.1.2. Estructuras de almacenamiento y definición de un vértice 87
5.1.3. Declaración de la interfaz para *Vertex Shaders* 88
5.1.4. Creación de un *Vertex Buffer* 89
5.1.5. Fuentes de vértices (*Vertex Streams*) 90
5.1.6. Creación, ensamblaje y definición de constantes de un *shader* 90
5.1.7. Dibujo de la escena mediante *shaders* 91
5.2. Ensamblador de las unidades de *Píxel* y *Vertex Shader* 91
5.3. *Shaders* en formato *Fx* (HLSL) 92
5.4. Creación de los *shaders* de la aplicación 93
5.4.1. Transformaciones comunes mediante *shaders* 93
5.4.2. *Crystal shader* 97
5.4.3. Iluminación y sombreado Gooch 100
5.4.4. Transición entre dos texturas (*multitexturing*) 103
Resumen 107

Pruebas al sistema
Soporte de modelos y tipo de *render* ... 111
Uso de *shaders* y selección de patrones ... 112
Crystal Shader ... 112
Gooch *Lighting* ... 113
Simple Transformations ... 114
Texture Transition ... 114
Creación de un libro ... 115
Carga de un libro para el *MagicBook* ... 117

Conclusiones 125

Apéndice A: Uso de *Cal3D*
A.1 Preparación del sistema ... 129
A.2 Creación del sistema *biped* ... 130
A.3 Modificador *Physique* ... 132
A.4 Exportando al personaje ... 134

Apéndice B: Uso de las bibliotecas del *ArToolKit*
B.1 Creación del proyecto ... 139
B.2 Esquema general para el desarrollo de aplicaciones mediante *ARToolKit* ... 140

Apéndice C: Cuaterniones
C.1 Bosquejo histórico ... 147
C.2 Álgebra y definición de un cuaternión ... 147
C.3 Representación de rotaciones usando cuaterniones ... 149
C.4 Demostración de que un cuaternión define una rotación ... 149
C.5 Representación matricial para la multiplicación de cuaterniones ... 151
C.6 Ángulos de Euler ... 153

Bibliografía ... 155

Glosario de términos ... 159

ÍNDICE DE FIGURAS

Figura 1 Esquema de un sistema de *AR* 14
Figura 2 Tecnología de "Ventana del Mundo" para aplicaciones de *AR* 16
Figura 3 Tecnología '*see-through*' para sistemas de *AR* 17
Figura 4 Curvas de Lissajous proyectadas 18
Figura 5 Integración de *Direct3D* con el sistema 27
Figura 6 *Pipeline* de *Direct3D* 29
Figura 7 *Frustum Clipping* 30
Figura 8 Arquitectura de *DirectShow* 34
Figura 9 Acceso de una aplicación al manejador de filtros 35
Figura 10 Diagrama de filtros para reproducir un archivo *AVI* 35
Figura 11 Registros de un *Vertex Shader* 37
Figura 12 Arquitectura de un *Vertex Shader* 38
Figura 13 Operaciones que ejecuta un *Píxel Shader* en paralelo 39
Figura 14 Arquitectura de un *píxel shader* 40
Figura 15 Interfaz del editor: *Boox' Wizard* 46
Figura 16 Ventana principal de dibujo 47
Figura 17 Menú 47
Figura 18 Carga de modelos 48
Figura 19 Tipo de *render* 48
Figura 20 Controles del trabajo actual 49
Figura 21 Selección de patrones 49
Figura 22 Selección de *shader* 50
Figura 23 Procesos involucrados con la inicialización de *Direct3D* 51
Figura 24 Pasos efectuados al inicializar el entorno 52
Figura 25 Especificación de dibujo de una escena en *Direct3D* 53

Figura 26 Elección de las rutinas de *render* 54
Figura 27 Procesos llevados a cabo al seleccionar algún *shader* 55
Figura 28 Matrices de escalamiento 58
Figura 29 Conversión del puerto de vista al plano de tamaño 2x2 situado en el origen 59
Figura 30 Cálculo del cuaternión 60
Figura 31 Acciones tomadas para efectuar una traslación 61
Figura 32 Relación entre algunas de las clases del *VisonSDK* de *Microsoft* 65
Figura 33 Ventana de tamaño definible por el usuario 66
Figura 34 Modelos soportados en ambas versiones 67
Figura 35 Diálogo para la carga de libros 68
Figura 36 *Shaders* en el *MagicBook* 68
Figura 37 Inicialización general de *DirectShow* en la aplicación 69
Figura 38 Etapa de selección de dispositivo de video (*CaptureGraph*) 70
Figura 39 Relación entre las clases *CTextureRenderer* y *CCustomPresentation* ... 70
Figura 40 Inicialización y reconocimiento de patrones usando el *ARToolKit* 71
Figura 41 Procesos involucrados en la carga de un elemento del *MagicBook* 72
Figura 42 Errores en la posición del sistema de referencia respecto del puerto de vista 75
Figura 43 *Skew* 76
Figura 44 Orientación de la cámara y como sería visto el objeto por ella 76
Figura 45 Planos de corte invertidos 77
Figura 46 Efecto en el volumen de vista al invertir los planos de corte 77
Figura 47 Identificación de patrones en el dibujo de la escena 78
Figura 48 Mapeo de *RGB24* a *RGB32* por el objeto *TextureRenderer* 80
Figura 49 Dibujo de un modelo en el *MagicBook* 81
Figura 50 Creación y *render* usando *shaders* 86
Figura 51 Especificación y uso de un *shader* para *Direct3D* 93
Figura 52 Iluminación difusa 95
Figura 53 Captura de pantalla del shader: *Simple Transformations* 97
Figura 54 Transformación a coordenadas de cámara 98
Figura 55 *Crystal Shader* 99
Figura 56 Transición fría-cálida en una ilustración de tipo no fotorealista 100
Figura 57 Creación de tonos 101
Figura 58 Acabado logrado mediante sombreado de Gooch 101
Figura 59 Gooch *Shading* 103

Figura 60 Texturas necesarias para el *shader*: *Texture Transition* 104
Figura 61 *Texture Transition* 106
Figura 62 Archivos *X* y *Cal3D* 111
Figura 63 Tipo de *render* 112
Figura 64.a Crystal Shader... 113
Figura 64.b Iluminación de Gooch 113
Figura 64.c Transformaciones usuales 114
Figura 64.d Transición entre dos texturas.. 115
Figura 65 Selección de patrones 115
Figura 66.a Avión militar F5e con *shader* de oxidación 116
Figura 66.b R2D2 con *shader* sustituto de *pipeline* por *default*........................... 116
Figura 66.c Modelos de *Cal3D*: *Skeleton* 116
Figura 66.d Modelos de *Cal3D*: *Cally* 116
Figura 66.e Cráneo con apariencia de radiografía 117
Figura 66.f Escudo deportivo de la UNAM en acabado de cristal 117
Figura 67 Interfaz para selección de libro 117
Figura 68.a Avión militar F5e... 118
Figura 68.b Despliegue del R2D2 118
Figura 68.c Modelos de *Cal3D* desplegándose en el *MagicBook*:
Cally con sombreado Gooch 119
Figura 68.d Modelos de *Cal3D* desplegándose en el *MagicBook*: Esqueleto 119
Figura 68.e Efecto de cristal en el MagicBook: Cráneo con apariencia
de radiografía 119
Figura 68.f Efecto de cristal en el MagicBook: Escudo de PUMAS
en apariencia de cristal 119
Figura 69 Animación y *tracking* en modelos *Cal3D* 120
Figura A.1 Estructura de directorios de *3DStudio Max*.................................... 130
Figura A.2 Personaje que será exportado al formato *Cal3D* 130
Figura A.3 Sistema *biped* en *3DStudio Max* 131
Figura A.4 Colocación del centro de masa acorde a la pelvis del personaje.......... 131
Figura A.5 Lado izquierdo de *biped* escalado y colocado acorde a las
dimensiones del personaje 132
Figura A.6 Botones de copiado y pegado para el biped.................................... 132
Figura A.7 Opciones del modificador *Physique* 133
Figura A.8 Envolvente de un brazo del personaje 133
Figura A.9 *KeyFrames* del personaje 135

Figura B.1 Propiedades de proyecto 139
Figura B.2 Directorios de encabezados y bibliotecas para Visual C++................. 140
Figura C.1 Base ortonormal que origina a R^3 146
Figura C.2 Eje de rotación y vector a rotar 149
Figura C.3 Sistema ortogonal formado por v_1, v_2 y v_3 150

ÍNDICE DE TABLAS

Tabla 1 Registros de un *Vertex Shader* 1.1 y posteriores 39
Tabla 2 Registros de un píxel shader 41
Tabla 3 Longitudes focales estándar de lentes, *FOV* y nombres 75
Tabla 4 Mapeo de registros para un *Vertex Shader* 89
Tabla C.1 Tabla de multiplicación para los número ***i***, ***j*** y ***k*** 147

INTRODUCCIÓN

En la actualidad, las gráficas por computadora (*CG*) son parte de la vida cotidiana. Carteles publicitarios, películas y software bastante elaborado hacen uso de ellas, sin mencionar la manera intensiva en que incurren las ramas de la ciencia y del entretenimiento en ellas. Incluso se podría decir que estás dos áreas son quienes las emplean más extensivamente.

En la industria del entretenimiento, las *CG* permiten crear efectos especiales, animaciones, personajes realistas para el cine, publicidad, televisión o videojuegos. No es necesario ir muy lejos para enumerar una serie de películas en las que los personajes digitales han participado con gran éxito: *Shrek* (2001), *Final Fantasy: The Spirits Within* (2001), las animaciones de *Pixar Animation Studios* (*A Bug's Life*, *Toy Story*, *Monsters Inc*., etc.) y algunos pioneros como *TRON* (1982) y *Blade Runner* (1982) que si bien no contaban con personajes creados por computadora, fueron los primeros en hacer uso de ellas para crear ciertos efectos especiales.

Simuladores, equipos de visualización numérica, de superficies o de fenómenos físicos, químicos, etc. es trabajo diario para las *CG*. Para geofísica, por ejemplo, después de efectuar una serie de estimaciones sobre la corteza terrestre, es posible que los investigadores requieran de un sistema que les permita simular y visualizar en tiempo real los estudios realizados. Para ello necesitan poder de computo para procesar todos sus datos y las *CG* les ayudan a visualizar sus datos, por ejemplo, el caso en el que se desee observar la forma en que se mueven las placas tectónicas durante un sismo.

En todas las áreas son útiles las gráficas por computadora. Sin embargo, una que no ha sido explorada es la de la educación. Considérese la enseñanza de historia durante los primeros años escolares. A lo largo del tiempo han transcurrido sucesos y situaciones que tienen repercusión hasta nuestros días, son una base de conocimiento de la humanidad muy fuerte. A pesar de ello, el estudio resulta tedioso e incluso aburrido, pues dicha actividad requiere de gran concentración y mucho leer, elementos carentes en los niños. La respuesta es captar su atención y entregar el conocimiento de una manera mas digerida y con una presentación que sea atractiva.

Haciendo uso de las *CG* se puede lograr lo comentado, ya que todo aquello que es visual, representa información más rica en contenido, además de que a la niñez de hoy en día, le fascina este tipo de cosas. Si se diseña una herramienta interactiva que permita al niño explorar un mundo o ser participe de ciertos eventos mientras se desarrolla el evento histórico, el aprendizaje será mayor pues su atención es total sobre lo que observa.

La pregunta a responder es la siguiente: ¿cómo resolver los retos presentes en cada una de las áreas presentadas? En el caso del entretenimiento, la respuesta mas simple es la de los videojuegos. Los mas recientes, mundos completamente tridimensionales en los que el jugador puede interactuar. Para el caso de las películas, existen diversos sistemas que permiten diseñar el mundo 3D, a los personajes, sus animaciones así como generar los cuadros. Uno de los mas populares es *Maya* de *Alias Wavefront*, usado en un sinnúmero de películas.

Cabe mencionar que gracias a la industria del entretenimiento, ha sido posible desarrollar cada vez mas el hardware y software relativo a las CG, ya que cuando en el mercado no se encuentra la aplicación que resuelva un problema, existe un grupo de desarrolladores encargado de crear la aplicación de acuerdo a las necesidades del trabajo.

Cuando se habla de visualización, se piensa en ambientes de simulación puramente virtuales y en tres dimensiones. Por ello, el área de interés que cubre las necesidades planteadas es la de la realidad virtual (*VR*), que permite que un individuo se desenvuelva en un ambiente tridimensional. Por aplicaciones de *VR* se tiene a todos los juegos de video en 3D, los paseos o recorridos virtuales que se pueden encontrar por Internet, los recorridos en un sitio de completa inmersión en los que se cuenta con un visor y un dispositivo que permita interactuar con el ambiente, entre otros.

La realidad aumentada (*AR*) es un subconjunto de la *VR*. Difiere en el aspecto de dar información adicional a un entorno real, esto es, en lugar de desenvolverse en un mundo virtual, el lugar de acción es el mundo real al cual se le agrega cierta información extra. Supóngase el caso de un piloto militar. La cabina del avión está repleta de controles e indicadores que debe de memorizar para poder maniobrar, disparar, comunicarse, etc. ¿Qué pasaría si algunos de esos controles o indicadores se colocan directamente en un visor que lleve en el casco?

Para el caso de los indicadores, estos serían de mayor utilidad, ya que en función de lo que requiera, estos podrían ser enviados mediante un sistema autónomo o bien mediante un operativo. En este caso se trataría de AR ya que a través de su visor percibe el mundo real, pero se le esta realimentado con información adicional, a manera de letras, en un par de lentes que lleva y que gracias a ellos logra tener los nuevos datos.

El caso de la realidad aumentada es el cubierto en el presente trabajo, motivado por una aplicación ya existente llamada *MagicBook*[1] que permite visualizar objetos tridimensionales a partir de patrones impresos en hojas comunes. Con el *MagicBook* es posible recrear cuentos, teniendo la versión impresa y su correspondiente virtual.

La aplicación funciona mediante una cámara web, los patrones impresos y el software, que fue creado usando una serie de bibliotecas de programación llamadas *ARToolKit*[2]. Tales bibliotecas están dedicadas a la adquisición y procesamiento de video, el reconocimiento de patrones, estimación de la posición de los objetos virtuales a partir del video adquirido y despliegue gráfico mediante la *API* de *OpenGL*.

Ahora bien, la industria de las gráficas ha seguido creciendo en software y hardware. En el caso del segundo, una de los avances mas grandes que ha tenido es el de poseer unidades programables de *shaders* en hardware. Los *shaders* son pequeños

---

[1] *MagicBook* es una aplicación desarrollada en el *Human Interface Technology Laboratory (HITL)* de la Universidad de Washington.

[2] *ARToolkit* es un conjunto de bibliotecas libres desarrolladas en el *HITLab* de la Universidad de Washington para desarrollar aplicaciones de Realidad Aumentada.

programas escritos en un lenguaje específico (como puede ser *Renderman* de *Pixar*), que están destinados a dar una apariencia mas realista a los objetos sintéticos (creados por computadora) y eliminar el acabado de plástico mate que los ha caracterizado por tanto tiempo.

En un inicio, los *shaders* se procesaban por software, siendo una tarea de mucha carga y tiempo de procesador aunque con resultados visuales agradables. Por supuesto, era imposible pensar en obtener escenas en tiempo real y el procesamiento requería de clusters de computadoras e incluso días enteros para llevar a cabo el dibujo de algunas cuantas.

Con el desarrollo de unidades en hardware que se dediquen a estas tareas, se ha incrementado la capacidad de procesamiento conduciendo a tener gráficas en tiempo real con una buena aproximación de acabado realista. En la actualidad, diversos juegos hacen uso de estas unidades (contenidas en las tarjetas aceleradoras de video de mas reciente generación) para mostrar diversos elementos. Entre ellos los mas impresionantes son el agua (transparente, con reflexión y refracción), el pasto (que ya no se crea mediante planos superpuestos y con transparencia, sino con vértice verdaderos), efectos de fuego, simulación de las propiedades intrínsecas de diversos materiales como pueden ser la roca, el metal, pintura metálica, etc.

Retomando el punto del presente trabajo, se desarrollará un nuevo visualizador que tenga la capacidad de dibujar usando *shaders*. Se requiere de uno nuevo, ya que para poder implementar los efectos se deben de usar la *API* de *Direct3D*, que a partir de sus distribuciones 8.1 y 9 soporta *shaders* programables en hardware. *OpenGL* ya cuenta con su respectivo lenguaje de *shading*, pero aun no hay distribución de las herramientas necesarias para compilarlos.

El nuevo visualizador será rehecho por completo en la parte gráfica, es decir, adquisición de video y dibujo de la escena. Las rutinas de las bibliotecas del *ARToolKit* para el reconocimiento de patrones y estimación de la posición no serán reemplazadas sino que se usarán tal y como si se emplearán en otro sistema de realidad aumentada.

En adición a la nueva versión del *MagicBook*, se creará un editor de libros. Esto para permitir a un usuario con menor experiencia hacer libros que posteriormente pueda utilizar. El editor debe ofrecer funciones para carga de modelos, observarlos, ubicarlos en la posición deseada, seleccionar un *shader* y aplicarlo al modelo, así como asociarle un patrón. Al final se debe de generar un archivo que represente el libro recién creado y que sea cargado por el visualizador sin mayor interacción que seleccionarlo.

El trabajo se presenta en cinco capítulos en los que se abordarán conceptos básicos sobre los temas aquí tratados, una introducción a lo que es el conjunto de *APIs* de *DirectX* (en específico *Direct3D*, *DirectShow* y los *shaders* para *Direct3D*), el desarrollo del editor de libros, la implementación de la nueva versión del *MagicBook* para concluir con un capítulo destinado únicamente a los *shaders*, como especificarlos, los lenguajes en que se pueden escribir, implementación de algunos de ellos, etc.

Al final se incluyen tres apéndices que explican temas importantes durante el desarrollo de las aplicaciones: uso de las bibliotecas de programación del *ARToolKit*, creación de modelos de *Cal3D* (uno de los formatos soportados en la nueva versión) y álgebra de cuaterniones.

Capítulo I
Conceptos Generales

Las gráficas por computadora (*CG*) comenzaron con el despliegue de datos en *plotters* y pantallas de tubos de rayos catódicos (*CRT*[3]) poco después de su introducción. Las *CG* se han desarrollado hasta incluir la creación, almacenamiento y manipulación de modelos e imágenes de objetos; estos modelos provienen de diversas y amplias áreas de estudio, que incluyen estructuras físicas, matemáticas, ingenieriles, arquitectónicas, etc.

Hasta principios de los ochentas, las *CG* eran un campo pequeño y especializado a causa del hardware costoso y las aplicaciones basadas en gráficos no eran fáciles de usar. Fue entonces que aparecieron las primeras computadores personales (*PC*) con dispositivos de gráficos por barrido (*Raster Graphic Devices*) que hicieron populares el uso de mapas de bits (*bitmaps*[4]) para la interacción usuario-computadora. Cuando los *bitmaps* se hicieron accesibles, un advenimiento de aplicaciones gráficas sencillas de usar se hizo llegar.

A partir de ese momento, el desarrollo de los gráficos se aceleró de una manera impresionante y comenzaron a surgir *APIs*[5] de programación especiales, motores de juegos (*game engines*), sistemas de *CAD*[6], *GPUs*[7], nuevas técnicas de animación y dibujo (*render*[8]), efectos especiales en tiempo real, etc.

A continuación se presentarán diversos conceptos y definiciones básicas referentes a los temas principales tratados en el desarrollo del trabajo como son: Interfaces Hombre-Máquina, Realidad Virtual, Realidad Aumentada, Visualización, Reconocimiento de patrones y Sombreadores programables (*shaders*).

1.1 Interfaces Hombre-Máquina (*HCI*)

> *"Donde los bits y las personas se encuentran"*
>
> *Nicholas Negroponte*[9]

Una interfaz es el elemento que hace posible la comunicación entre dos entes (aparatos, personas, entidades u objetos) cuando no pueden comunicarse directamente, porque se expresan en lenguajes distintos o con señales de diferentes especificaciones[10]. Como ejemplo se puede mencionar el caso en que una persona hace traducción simultánea, siendo así la interfaz que permite comunicarse a otras dos personas que hablan idiomas diferentes. Otro caso es el de un módem, interfaz que permite comunicar líneas telefónicas (analógicas) con computadoras (digitales).

---

[3] *CRT:* Cathode Rays Tube. Tubo de Rayos Catódicos.

[4] *Bitmap*: Representación binaria del arreglo rectangular de puntos en pantalla.

[5] *API*: Application Programming Interface. Interfaz de Programación de Aplicaciones.

[6] *CAD*: Computer Aided Design. Diseño Asistido por Computadora.

[7] *GPU*: Graphic Processing Unit. Unidad de Procesamiento Gráfico.

[8] *Render*: Verbo del inglés proveniente del francés *rendre*, cuyo significado es el de 'producir' o 'rendir cuentas'. Tiene diferentes significados dependiendo del ámbito en el que se use. En términos de graficación por computadora es usado para referirse a dibujar, hacer un bosquejo, o el mostrar algo en papel u otro medio.

[9] Negroponte, Nicholas. *Ser digital*. México. Ed. Planeta. p. 107

[10] Sitio web http://www.interfaz.com.co/asesoria/respuestas.html

En la actualidad las Interfaces Hombre-Computadora, independientemente de la aplicación, implican el despliegue y cálculo de gráficas extensivas. Por lo regular, los sistemas operativos actuales presentan ventanas, menús, íconos, botones y dispositivos señaladores para ubicar el cursor en pantalla.

Las interfaces gráficas para usuario más comunes incluyen los sistemas XWindows, Windows, OsX, Gnome, KDE, etc. Estas interfaces son explotadas por una variedad de aplicaciones que comprenden el procesamiento de texto, hojas de cálculo, bases de datos, sistemas de administración de archivos, entre otras.

Algunos principios que toda *HCI* debe cubrir son los siguientes:[11]

1) Estado del sistema. El sistema siempre debe mantener informado a los usuarios de lo que ocurre, con la actualización correcta en un tiempo razonable.

2) Correspondencia entre el sistema y el mundo real. El sistema debe hablar el lenguaje de los usuarios, con palabras, frases y conceptos familiares para el mismo así como seguir convenciones del mundo real, permitiendo que la información aparezca en forma natural y lógica.

3) Control y libertad del usuario. Los usuarios frecuentemente eligen opciones por error y se debe indicar una salida para esas situaciones no deseadas, sin necesidad de pasar por extensos diálogos.

4) Consistencia y estándares. Los usuarios no tienen que adivinar, lo que las diferentes palabras, situaciones o acciones significan.

5) Evitar errores. Un diseño cuidadoso que prevenga la ocurrencia de problemas es mejor que mensajes de error.

6) Reconocimiento vs. Recuerdo. Hacer objetos, acciones, y opciones visibles. El usuario no tiene que recordar información de una parte a otra. Las instrucciones de uso del sistema deben ser visibles o fácilmente recuperables.

7) Flexibles y eficientes. Diseñar un sistema que pueda ser utilizado por un rango amplio de usuarios. Brindar instrucciones cuando sean necesarias para nuevos usuarios pero que no se entrometa en el camino de usuarios avanzados. Permitir a los usuarios avanzados ir directamente al contenido que buscan.

8) Diseño minimalista. No mostrar información que no sea relevante. Cada tramo de información extra, compite con la importante y disminuye su visibilidad relativa.

9) Ayuda a los usuarios para reconocer, diagnosticar y recuperarse de los errores. Los mensajes de error deben estar escritos en un lenguaje sencillo, indicar de manera precisa el problema y constructivamente indicar la solución.

10) Ayuda y documentación. El mejor sistema es aquel que se puede usar sin documentación, pero siempre facilita una ayuda o documentación. Esta información debe ser accesible fácilmente, dirigida a las tareas de los usuarios, listar los pasos concretos para realizar las acciones mas comunes y no ser muy extensa.

[11] Preece, Jenny, Rogers Yvonne, Sharp Helen Benyon, David. *Human-Computer Interaction,* p.138 1994

Interesan las cuestiones psicológicas inherentes a los dispositivos que se usan o a las pantallas que se observan. Dichas cuestiones son las que determinan si una interfaz está bien diseñada o es eficiente. Esto es porque los usuarios difieren en competencias o capacidades como son: la comprensión de las instrucciones escritas, el recordar o memorizar instrucciones, la paciencia o entusiasmo para aprender formas nuevas de hacer las cosas. Asimismo, la interpretación de símbolos dibujados en los botones. Por ejemplo, ¿qué significa, hacia delante o hacia atrás?[12]

Es decir, el éxito de una HCI depende del entrenamiento, las estrategias y uso de habilidades previas que poseen los usuarios. A pesar de que los usuarios tienen similitudes en ejecución y procesamiento de información existen diferencias en competencias y/o capacidades, haciendo que una interfaz adecuada para algunos no lo sea para otros. Las ineficiencias en el uso de la *HCI* pueden deberse a problemas físicos o fisiológicos pero también a limitaciones psicológicas en memoria, aprendizaje, o falta de atención. La mayoría de estas limitantes pueden solventarse llevando a cabo un buen diseño que cumpla con las recomendaciones mencionadas.

1.2 Realidad Virtual

"El nivel de intimidad de un interfaz hombre-máquina
es el ancho de banda de la comunicación.
En el extremo superior del espectro
se encuentran los sistemas "inmersivos" de R.V."

Nicholas Negroponte[13]

Atendiendo al significado de las palabras que componen al término Realidad Virtual se puede encontrar que *realidad* se refiere a la cualidad o estado de ser real o verdadero mientras que *virtual* es aquello que existe o resulta en esencia o efecto, pero no como forma, nombre o hecho real.

Para lograr una definición adecuada, es necesario introducir la concepción del hombre y tecnológica del término: un sistema con un ambiente interactivo, tridimensional, generado por computadora que pretende manipular los sentidos humanos mediante la simulación del conjunto completo de datos sensoriales que constituyen la *experiencia real*.

El resultado de dicha simulación es un ambiente artificial que no existe desde el punto de vista físico. Dentro de este ambiente, uno o varios participantes acoplados de manera adecuada al sistema interactúan de forma rápida e intuitiva. Por lo tanto, un mundo virtual es un ambiente similar a la realidad que se crea utilizando un medio artificial; es una simulación de un ambiente real o imaginario que provee una experiencia en tiempo real proporcionando estímulos parecidos a los de la realidad.

[12] Sitio Web: http://www.um.es/docencia/agustinr/pca/textos/LandsdHCI.htm
[13] Negroponte, Nicholas. *Ser digital*. México. Ed. Planeta. p. 165

1.2.1 Breve historia

Los primeros dispositivos que intentaron recrear situaciones reales en un modo artificial para uso humano, fueron los primeros simuladores de vuelo patentados en 1930, los cuales intentaban reproducir los movimientos de una nave para generar la sensación de vuelo. Estos dispositivos eran en su totalidad mecánicos y sin dispositivo gráfico o interfaz parecida.

A mediados de los sesenta, el término Realidad Virtual no existía. Éste surge cuando el Dr. Iván Sutherland publica un artículo titulado "*The Ultimate Display*", en el cual establece los principios y conceptos básicos de la realidad virtual.

En dicho escrito se menciona la posibilidad de proporcionar una interfaz que funcione con cualquier simulación, que produjera no sólo la experiencia de volar en un avión, sino la de estar en cualquier espacio artificial, en donde la computadora tiene control total de los eventos.

Para 1966, Sutherland había ya creado el primer casco visor de VR montando tubos de rayos catódicos en un armazón de alambre. A tal instrumento se le conoce por "*espada de Damocles*" ya que el aparato requería de un sistema de apoyo que pendía del techo.

Durante 1968 Ivan Sutherland y David Evans crean el primer generador de escenarios con imágenes tridimensionales, datos almacenados y aceleradores. Además, fundan la sociedad Evans & Sutherland. Su sistema final consistía en muchos aceleradores de hardware para improvisar el funcionamiento del sistema gráfico y la generación de imágenes estereoscópicas en lugar de monoscópicas.

Al inicio de los setentas se crean los primeros simuladores de vuelo computarizados y de ésta manera, se marca lo viable y práctico de los sistemas virtuales. Esta década presenta importantes avances en investigaciones y desarrollo sobre despliegue, presentaciones gráficas e interfaces hombre-máquina.

En los ochenta, William Gibson publica su novela de ciencia ficción *Neuromancer,* en la que se emplea por vez primera el término *Ciberespacio,* para referirse a un mundo alternativo al de las computadoras.

Durante tal periodo Thomas Furness III desarrolla la *cabina virtual* y poco después presenta el simulador más avanzado contenido por completo en un casco y diseñado para el ejército de los Estados Unidos. Asimismo se tuvieron avances notables en los dispositivos de entrada y salida como son los visores a color, sonido tridimensional, guantes de control y sistemas más avanzados de posicionamiento. Fue a finales de ésta década cuando se exhiben los primeros sistemas de *VR* completos.

En la actualidad, los sistemas de realidad virtual son empleados en tareas científicas (cómo la medicina y la astronomía) y en diversos sistemas de entrenamientos como los de la NASA.

1.2.2 Clasificación de mundos virtuales

Un mundo virtual puede ser clasificado de diferentes maneras de acuerdo a sus características como son la interacción con el usuario, si los sentidos humanos son involucrados, la cercanía con la realidad, etc.

Así pues, por la interacción con el usuario se pueden clasificar en:

- Pasivos. Permiten explorar el lugar pero sin interactuar con el medio, además de que no existe comportamiento alguno para los objetos.
- Reactivos. Permiten al usuario obtener reacciones ante las acciones que éste lleve a cabo en el mundo.
- Activos. Los objetos tienen comportamiento sin necesidad de que haya interacción con el usuario y además tienen comportamientos reactivos.

Ahora bien, por la forma en que se presenta la información del mundo virtual se tienen los siguientes:

- Inmersivos. Aquellos en los que el usuario tiene la sensación de estar realmente en el ambiente.
- Proyección. Pretenden proporcionar la misma sensación inmersiva, pero dentro de un espacio cerrado en el que el usuario es introducido.
- Sobremesa. No buscan la inmersión, además de que el usuario no es asilado de su mundo circundante.

Para el caso de sistemas específicos de Realidad Virtual, se tienen los siguientes:

- Cabina de simulación. Recrea el interior del dispositivo o máquina que se desea simular en donde las ventanas de la misma son reemplazadas por monitores en los que se recrea el mundo exterior. Además, existen bocinas estereofónicas que brindan el sonido ambiental y que pueden estar fijas o sobre ejes móviles. El programa controlador es diseñado para que responda en tiempo real a los estímulos que el usuario envía a través de los controles de la cabina.
- Realidad proyectada. Una imagen en movimiento del usuario es proyectada junto con otras imágenes en una extensa pantalla donde el usuario puede verse a sí mismo como si estuviese en escena. Básicamente, los usuarios se ven proyectados en el mundo virtual, como ejemplo se tiene a los escenarios virtuales que se utilizan en ciertos programas de televisión.
- Realidad Aumentada[14]. Se logra cuando la persona tiene como contexto el mundo real y utiliza visores de cristal transparente o algún otro medio inmersivo para aumentar la realidad por medio de la superposición de esquemas, texto o y/o modelos en 3D.
- Telepresencia. El sistema proporciona al usuario la sensación de estar físicamente en otro lugar por medio de una escena creada por computadora. El resultado deseado solo se obtiene cuando el sistema es tan completo como para engañar a la psique del usuario de que se encuentra realmente en ese lugar.

---

[14] Véase sección 1.4 para una explicación con mayor detalle.

1.2.3 Dispositivos de salida usados en aplicaciones de Realidad Virtual

A los dispositivos de salida que se usan para este tipo de aplicaciones se les puede dividir básicamente en dos: los dispositivos de presentación y los dispositivos de audio.

Los dispositivos de presentación proporcionan al usuario la imagen del mundo virtual mientras que los de audio son auxiliares del realismo de la aplicación.

1.2.3.1 Dispositivos de presentación

Estos dispositivos tienen por objeto proporcionar al usuario la imagen del mundo virtual, es decir información de carácter gráfico acerca de los objetos que componen el ambiente, sus posiciones y características.

Los dispositivos que comúnmente se encargan de realizar éstas tareas son los monitores y las pantallas de proyección, sin embrago, existen otros dispositivos como los *HMD* (*Head Mounted Display*/Cascos de *VR*).

Los cascos y los sistemas binoculares son empleados únicamente en sistemas que son inmersivos con motivo de reforzar en el usuario la sensación de realidad, ya que aíslan al usuario por completo del mundo exterior. Sin embargo, este aislamiento trae consigo desventajas. Una de ellas es que el usuario queda imposibilitado para emplear algunos dispositivos de entrada como es el caso del teclado.

En los sistemas proyectivos y de sobremesa se emplean con mayor frecuencia dispositivos convencionales (monitores por ejemplo) lo cual brinda una excelente resolución y calidad de imagen además de no impedir el uso de otras interfaces físicas. A cambio, estos sistemas son no inmersivos, el ángulo de visión es pequeño y la imagen no es estereoscópica.

Un dispositivo muy comúnmente empleado, es el *HMD*[15] (*Head Mounted Display*). Los *HMD* son dispositivos en forma de casco o visera que el usuario lleva mientras se encuentra dentro del mundo virtual. En dicho casco, es incorporado un par de pantallas de cristal líquido pasivas en color lo cual permite una visión estereoscópica de la escena. Además, cuenta con un par de auriculares estereofónicos para proporcionar sonido y un sensor de posición (*tracker*) encargado de determinar la ubicación y orientación del usuario.

La visión estereoscópica es lograda mostrando en cada una de las pantallas imágenes ligeramente diferentes, que corresponden a las imágenes que cada ojo percibiría en la realidad. El efecto de visión estereoscópica, aunado al trabajo del *tracker* son los responsables de que el usuario tenga inmersión en el mundo virtual.

Una característica importante de los *HMD* es que son muy ligeros y poco voluminosos, requisitos indispensables desde el momento que el usuario debe llevar puesto el dispositivo. Las pantallas tienen lentes de aumento para agrandar el tamaño de las imágenes. El ángulo de visión que se consigue es de 110º en la horizontal y de 90º en la vertical.

[15] *HMD*: Head Mounted Display. Cascos de Realidad Virtual.

Este tipo de dispositivos tiene la inconveniente de presentar problemas de brillo y contraste, lo que provoca que las imágenes generadas sean de baja resolución y mala calidad.

1.2.3.2 Dispositivos de audio

El sonido dentro de las aplicaciones de Realidad Virtual tiene funciones muy importantes. Entre ellas se encuentran la informativa, la cuál tiene como finalidad indicar al usuario que cierto evento ha ocurrido, como ejemplo la pulsación de un botón.

Otra función es la metafórica, esto es que el sonido puede emplearse para traducir una serie de datos a un formato fácilmente entendible por el usuario, por ejemplo, en visualización científica y para el caso de una aplicación que tenga que ver con los niveles de un campo magnético, el volumen se incrementa a medida que el campo es más fuerte.

La función artística es también importante, ya que mediante la adición de música de fondo se puede lograr una mejor ambientación, se incremente el atractivo de la aplicación e incluso influye sobre el estado de ánimo del usuario.

Por último queda la función descriptiva, a la que corresponden los efectos especiales (*FX*) ya que describen sucesos que tienen lugar dentro del mundo virtual como puede ser el sonido de un vaso al romperse.

Para poder crear estas secuencias sonoras existen diversos métodos, los más usados son la carga de una secuencia previamente grabada que se reproduce, o bien, crear la secuencia empleando un sintetizador.

El primer método ofrece una buena producción de sonido así como buena calidad, pero no se pueden ejercer modificaciones sobre de ella. En cambio un sintetizador provee la flexibilidad necesaria para generar determinada nota con los parámetros (volumen, duración, etc) e instrumentos deseados.

Con los sonidos obtenidos, resta el proceso de integración al mundo y que el usuario sea capaz de escucharlos, identificarlos y ubicarlos en un espacio tridimensional. Para esto, se requiere de tarjetas de sonido 3D que ofrezcan la posibilidad de que el usuario se percate de la proximidad de los objetos u ocurrencia de eventos asociados a objetos que caen fuera de su campo de visión.

La técnica más efectiva que se emplea es la de localización del sonido creada en 1988 por Scott Foster. La técnica se basa en la transmisión de sonido a los oídos en diferentes tiempos y con intensidades ligeramente diferentes. Con esto, se logra que el sonido llegue con mayor rapidez al oído situado más cerca de la fuente que lo originó que al otro.

Este pequeño retardo de tiempo es suficiente para realizar una primer discriminación acerca de la ubicación de la fuente de sonido. Además, los pliegues de la oreja permiten filtrar la señal sonora de una forma que dependa de la dirección en que ésta provenga. Por lo tanto, con hacer escuchar al usuario mediante un par de auriculares, éste será capaz de determinar la dirección de la fuente virtual de sonido.

1.2.3.3 *Otros dispositivos de salida*

Además de los dispositivos mencionados anteriormente, existen otros que se integran a los sistemas de realidad virtual con la finalidad de aumentar el realismo de las aplicaciones y la naturalidad de la interfaz con la computadora. Algunos de ellos son las plataformas móviles y los dispositivos táctiles.

Las plataformas móviles tienen por objeto simular los movimientos de navegación para conseguir mayor realismo y para compensar en parte las causas productoras de mareo. Las cabinas son las plataformas que se usan con mayor frecuencia, ya que dependiendo de los movimientos que el usuario realice, el sistema habrá de inclinar la plataforma.

Es importante mencionar que las plataformas móviles no simulan las traslaciones por el mundo virtual, sino solamente las posiciones que adopta el usuario. Por ejemplo, supóngase que un usuario se encuentra volando a ras de suelo a través del mundo. Mientras no efectúe una maniobra de ascenso, la plataforma no se moverá. En el momento en que el sistema reciba la orden de ascenso por parte del usuario, la plataforma se inclinará hacía atrás para reflejar la nueva posición que toma el usuario y se mostrará la imagen correspondiente a un ascenso. Lo mismo sucederá cuando el usuario se mueva a los lados o bien hacía abajo.

Existe la posibilidad adicional de simular la inercia de un cuerpo a través de su peso. Tomando el ejemplo anterior, si el usuario volando a ras de suelo experimenta un incremento de velocidad entonces la plataforma puede moverse hacía atrás en proporción al peso del usuario pero sin afectarse la imagen. Esto se logra debido a que el cerebro del usuario interpreta una aceleración al momento de sentir su cuerpo que va hacía atrás, mientras que la imagen se mantiene igual.

Los dispositivos táctiles son también muy importantes en las aplicaciones de realidad virtual, ya que permiten sentir al usuario el contacto con algún objeto virtual cercano e incrementar el realismo. Desafortunadamente este tipo de dispositivos tiene dos defectos principales:

- Su gran tamaño y baja frecuencia de activación, lo que impide simular las rugosidades finas de las superficies.
- La sensación de tacto es algo irreal, ya que no se tienen realimentaciones de tipo cinético como podrían ser la resistencia del objeto, su peso, la viscosidad, etc.

1.2.4 Dispositivos de entrada usados en aplicaciones de Realidad Virtual

Los dispositivos de entrada usados en los sistemas de *VR* juegan un papel muy importante ya que permiten al usuario interactuar con el sistema trasmitiendo órdenes referentes a la posición y orientación del usuario. Por lo tanto, los dispositivos de entrada pueden ser clasificados en dispositivos de localización y dispositivos de control.

1.2.4.1 Dispositivos de localización

La función primordial de los dispositivos de localización es la de obtener la posición y la orientación de cualquier objeto dentro del mundo virtual.

En el caso de la orientación del usuario, ésta debe de ser procesada para mantener la correspondencia entre lo que el usuario pretende ver con sus acciones y lo que el dispositivo de salida entrega. Para los sistemas inmersivos, el control de la orientación debe ser automático, reflejando un cambio de orientación dentro del mundo virtual cuando el usuario ejerza un movimiento en su cabeza. En los sistemas no inmersivos, un cambio de orientación dentro del mundo virtual, corresponde a un cambio de orientación de la ventana (la cabeza del usuario siempre mira hacía el monitor).

En cuanto a la posición, conocerla es necesaria puesto que de esta manera se pueden conocer los desplazamientos que el usuario realiza y actualizar la misma dentro del mundo virtual acorde a ellos.

1.2.4.2 Dispositivos de control

Tienen como objeto permitir que el usuario interactúe con el mundo virtual. Las órdenes que se pueden dar mediante estos dispositivos tienen que ver con comandos de navegación, con los cuales el usuario indica al sistema su deseo de desplazarse dentro del espacio virtual.

Asimismo, se tienen comandos de interacción y manipulación, que permiten al usuario tomar decisiones, realizar acciones y afectar al mundo de manera determinada.

Algunos ejemplos de este tipo de dispositivos son los siguientes:

- Guante. Los guantes son dispositivos empleados para determinar la posición de la mano del usuario. Suelen ser realizados de nylon o lycra e incorporan una serie de sensores que permiten determinar el grado de flexión de cada uno de ellos así como la separación existente entre los dedos.

  Incorporando un dispositivo de localización al guante, se puede complementar dicha información con la relativa a la posición absoluta de la mano.

- Ratones y *Joysticks*[16] 3D. Los ratones 3D son una extensión de los ratones convencionales que surgieron como solución ante el problema de navegación en un mundo tridimensional. Es de esta manera que permiten controlar el movimiento de cualquier objeto en un espacio virtual.

Por su parte, los *joysticks* 3D combinan la función de un ratón tridimensional además de medir la fuerza que el usuario aplica al movimiento, logrando de esta manera calcular la dirección y velocidad del mismo.

[16] *Joystick*: Dispositivo de control usado en juegos de computadora. Obtuvo este nombre a partir del control empleado por los pilotos de aeronaves para controlar los alerones y elevadores de las mismas.

1.2.5 Creación de un mundo virtual

Un mundo virtual pasa por distintas etapas que determinan su ciclo de vida de la misma forma que cualquier otro software. Las etapas son las siguientes:

- Propuesta de ambiente: Específica lo que se espera del mundo virtual justificando su creación y la necesidad de usar la realidad virtual.
- Objetivo: Indica cuales son las metas y propósitos del ambiente. Con base en el objetivo se crea una lista de las limitaciones (tanto prácticas como de diseño) que determinan el ámbito del problema y el alcance del sistema.
- Análisis: Su objetivo es el de obtener los requerimientos del sistema (sabiendo de antemano los recursos con que se cuenta) así como de proponer la metodología de solución al problema.
- Diseño abstracto: Consta de la realización de diagramas que especifican el funcionamiento del mundo virtual y sus elementos.
- Diseño concreto: Aterriza los conceptos dados por los diagramas llevando estos a un contexto más práctico, es decir se trata del proceso de preparación previo a la implementación del mundo en software.
- Implementación: Consiste en toda la programación necesaria para crear el mundo virtual.
- Pruebas: Esta fase tiene como objetivo verificar el correcto funcionamiento del mundo así como establecer si los puntos que al inicio se plantearon han sido cubiertos.
- Mantenimiento: Una vez constatado que el sistema funciona correctamente, se debe de asegurar su funcionalidad. Para esto es necesario el mantenimiento, etapa en la que aquellos errores y *bugs*[17] no encontrados durante la fase de pruebas sean corregidos.

1.3 Realidad Aumentada

La Realidad Aumentada (*AR*) es un área de la Realidad Virtual que se encuentra en crecimiento. Ésta responde a la necesidad de añadir un extra de información o clarificar percepciones correspondientes al mundo real.

El medio ambiente que nos rodea, provee de gran cantidad de información que es casi imposible representarla en un sistema computacional. Esto es puesto en evidencia al observar los mundos usados en ambientes virtuales. Dichos mundos pueden tratarse de representaciones muy simples (usadas generalmente en juegos y para el entretenimiento inmersivo) o bien ser sistemas con ambientes muy realistas pero muy costosos (como es el caso de los sistemas de simuladores de vuelos).

Como se menciono anteriormente, un sistema de AR genera una vista compuesta para el usuario, a partir de la escena real y una virtual creada mediante la computadora y que dota al usuario de información adicional sobre la misma.

La meta final es crear un sistema tal que el usuario sea incapaz de notar la conjunción de elementos virtuales con la realidad, dando la sensación de observar a una sola escena.

[17] *Bug*: Error de programación en la codificación de una pieza de software.

1.3.1 Aplicaciones de la AR

Apenas, hace relativamente poco tiempo, las capacidades del procesamiento de imágenes provenientes de video en tiempo real, sistemas de gráficos por computadora, y nuevas tecnologías de despliegue han convergido para hacer posible el despliegue de una imagen virtual correctamente localizada con una vista en 3D del ambiente que rodea al usuario.

Los investigadores que trabajan con este tipo de sistemas, los han propuesto como soluciones en diversos campos, tales como el militar o el medico por citar algunos. A continuación se comentarán algunos de los campos de aplicación en relación a esta tecnología.

1.3.1.1 Medicina

Ya que la tecnología es un elemento muy persuasivo en el campo médico, no es sorprendente que este campo sea visto como uno de los más importantes para los sistemas de *AR*.

En su mayoría, las intervenciones médicas tienen que ver con cirugía asistida mediante imágenes. Con esto se refiere a todos los estudios efectuados al paciente previamente como puede ser el caso de un ultrasonido, una tomografía, radiografías, etc. con el fin de planear la cirugía.

En el caso de la extracción de un tumor, por ejemplo, lo primero es crear la representación 3D a partir de los estudios previamente realizados. La *AR* puede emplearse entonces para que el equipo de cirugía cuente con información precisa acerca de las radiografías, tomografías, etc. en el instante que efectúa la operación.

Otra aplicación es la del ultrasonido, en donde el técnico podría ver una imagen volumétrica del feto dentro del abdomen de la paciente.

1.3.1.2 Entretenimiento

Un tipo sencillo de *AR* ha sido usado en los negocios del entretenimiento y noticias por ya algún tiempo.

Cuando se observa el reporte del meteorológico, el reportero es mostrado enfrente de mapas regionales que cambian cada determinado tiempo. La situación real en el estudio, es que el reportero se encuentra parado frente a una pantalla de color verde o azul. Esta imagen real es entonces aumentada con mapas generados por computadora empleando una técnica llamada *chroma-keying*[18].

Otro caso es el de la publicidad virtual que se coloca en ciertos eventos transmitidos, como son juegos de baseball, basketball, football, soccer, etc. Por ejemplo, mientras que un juego de baseball se transmite, el sistema sería capaz de colocar un anuncio en la imagen de TV de forma tal que se tenga la apariencia de que el diamante contiene al mencionado anuncio.

---

[18] *Chroma Keying*: Esta técnica (también conocida como de la pantalla azul) consiste en fotografiar o tomar cuadros de video a un sujeto frente un fondo azul, para después superponer cualquier fondo que se necesite sin afectar al personaje. Ha sido empleada principalmente en el cine, fines publicitarios y ambientes virtuales usando video.

1.3.1.3 Milicia

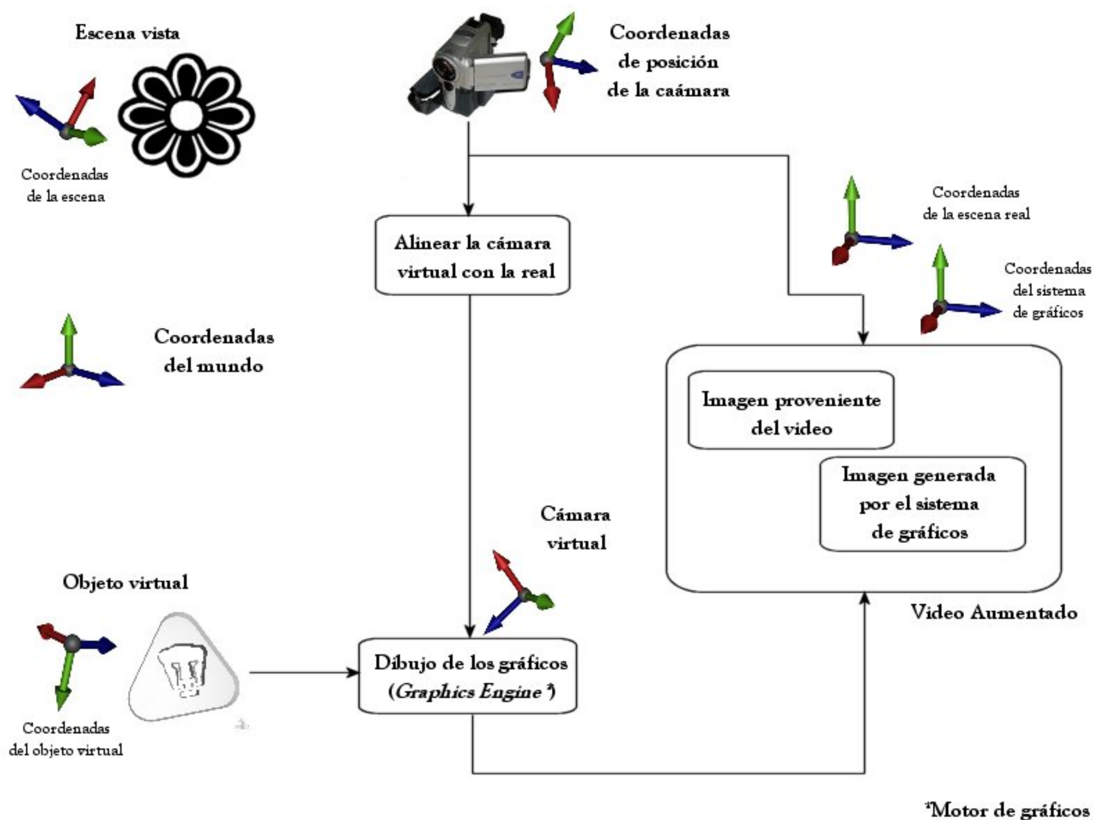
La milicia se ha caracterizado por usar pantallas en simuladores de vuelo que muestran información adicional al piloto en el vidrio de la cabina de vuelo o bien en el casco que porta.

1.3.2 Sistema típico de *AR*

A pesar de que existen diferentes y muy diversas aplicaciones, un sistema de este tipo tiene componentes comunes.

Un sistema estándar de *AR* busca la completa inmersión del usuario en un ambiente generado por computadora. Dicho ambiente es mantenido por el sistema en un marco de referencia registrado mediante el sistema de gráficos por computadora que hace el dibujo del mundo virtual. Para que la inmersión sea efectiva, el marco de referencia entre el sistema psicomotor del individuo y la referencia del mundo virtual deben de ser concordantes. Esto requiere que los movimientos o cambios efectuados por el usuario en el ambiente real, sean reflejados en el virtual.

Ya que no existe una conexión directa entre los mundos, ésta debe ser creada. La tarea entonces consiste en crear un marco de referencia real, de acuerdo al virtual con que se cuenta. La siguiente figura (Figura 1) muestra las diferentes etapas relacionadas en un sistema de *AR*.



Esquema de un sistema de *AR*
Figura 1

La escena es vista por un dispositivo de imágenes, en este caso se trata de una cámara de video. La cámara produce una proyección en perspectiva del mundo 3D en una imagen 2D. Los parámetros intrínsecos (longitud focal y distorsión del lente) y extrínsecos (posición y pose) del dispositivo determinan exactamente lo que se muestra en el plano de proyección.

La creación de la imagen virtual es realizada con un sistema estándar de gráficos por computadora. Los objetos son modelados con base en su propio marco de referencia. Este sistema requiere de información concerniente a la escena real para que pueda dibujar correctamente los objetos. Dicha información controlará la cámara 'sintética' o virtual que es usada para obtener las imágenes de los objetos.

La imagen generada es entonces combinada con la imagen de la escena real para crear la imagen en *AR*.

1.3.3 Rendimiento de un sistema de *AR*

Con el fin de que el usuario pueda moverse libremente dentro de la escena y obtener una imagen de AR adecuada, el sistema de *AR* debe de ser ejecutado en tiempo real. Ésta característica agrega dos criterios de rendimiento en el sistema, los cuales son:

- Actualización constante y continua para generar la imagen aumentada.
- Exactitud en la conformación de la imagen aumentada (concordancia entre las imágenes real y virtual).

Visualmente, la restricción del tiempo real se manifiesta cuando el usuario observa la imagen aumentada y en ésta existe una falta de sincronía al ser dibujada. Para que dicha imagen aparezca sin saltos, una regla ya estándar es que el sistema sea capaz de dibujar la escena al menos 10 veces por segundo.

La regla mencionada anteriormente es valida para sistemas de gráficos por computadora que requieren dibujo no foto-realista. Si lo que se busca es un dibujo de tipo foto-realista, entonces un motor de gráficos más poderoso es requerido.

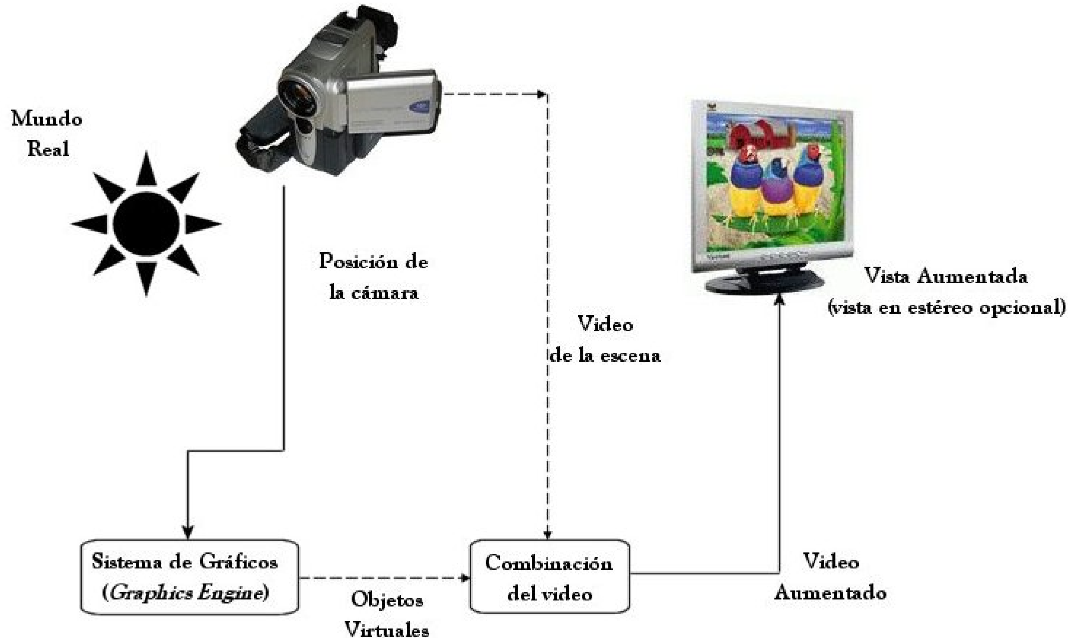
Las fallas en el segundo criterio pueden deberse a dos cosas. La primera es una falta de sincronización por ruido en el sistema. La posición y pose de la cámara con respecto a la escena real debe ser medida. Cualquier falla en la medición tiene el potencial de ser mostrado como errores de sincronización entre las imágenes real y virtual. La fluctuación entre valores mientras que el sistema es usado causará pequeños saltos en las imágenes generadas. El efecto final es que el objeto virtual no se posiciona correctamente en la escena real.

La segunda causa de falta de sincronización es el retraso que pueda tener el sistema. Si existen retrasos al calcular la posición de la cámara o la alineación de la cámara virtual, entonces los objetos aumentados presentarán un retraso al presentarse movimientos de la escena real.

1.3.4 Tecnologías de despliegue en *AR*

La combinación de imágenes reales y virtuales provee de nuevos retos para los diseñadores de sistemas de *AR*. La pregunta de cómo hacer que dos imágenes se mezclen es básica. Ante tal cuestionamiento, se presentan los siguientes esquemas.

El primero y más sencillo de implantar es aquel conocido como "Ventana del Mundo"[19] o "Realidad Virtual de Pecera"[20]. Mediante esta tecnología, el usuario tiene poca sensación de inmersión en el mundo virtual que se crea. Un diagrama que muestra dicha tecnología se muestra en la Figura 2.



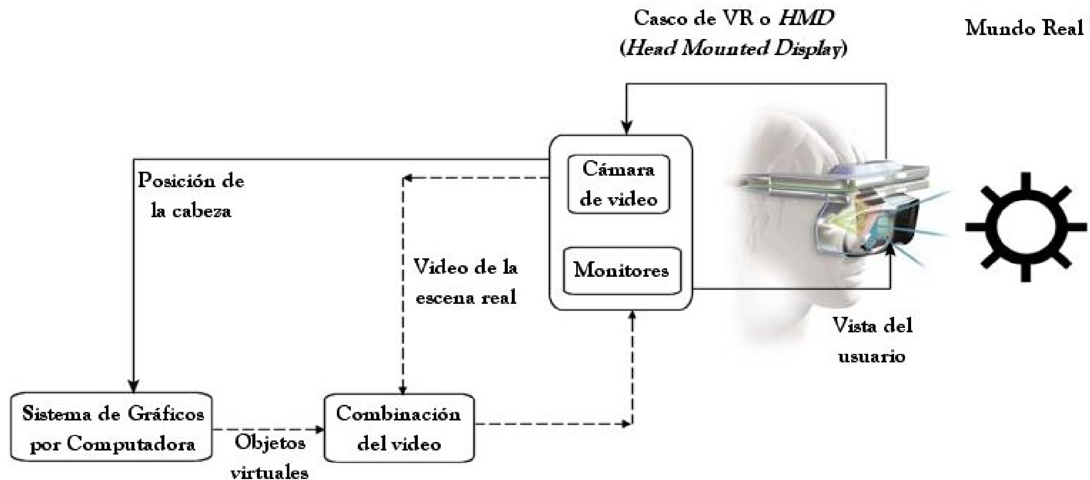
Tecnología de "Ventana del Mundo" para aplicaciones de *AR*
Figura 2

Para incrementar la sensación de presencia, otras tecnologías de despliegue son necesarias. Los *HMD* son usados ampliamente en los sistemas de *VR*. En el caso de aplicaciones de *AR*, los *HMD* empleados son del tipo "*see-through*"[21]. Esto ya que es necesario percibir el ambiente real a diferencia del estándar usado en aplicaciones de VR y que aísla al usuario del medio que lo rodea. Un esquema de un sistema de *AR* usando HMD "*see-through*" es mostrado a continuación (Figura 3).

[19] Feiner, Steven, Blair MacIntyre, Marcus Haupt, Eliot Solomon. *Windows on the World: 2D Windows for 3D Augmented Reality*. ACM Symposium. Atlanta Georgia, noviembre de 1993. p145-155

[20] Arthur, Kevin W., Kellog S. Booth, Colin Ware. *Evaluating 3D Task Performance for Fish Tanks Virtual Worlds*. ACM Transactions on Information Systems, julio de 1993. p239-265

[21] Tecnología '*see-through*' Los *HMD* empleados se encuentran compuestos por un par de monitores en los que se muestra el video adquirido de la escena real.



Tecnología 'see-through' para sistemas de *AR*
Figura 3

1.4 Visualización

La Real Academia Española define al verbo *visualizar* como:[22]

- Representar mediante imágenes ópticas fenómenos de otro carácter; p. ej., el curso de la fiebre o los cambios de condiciones meteorológicas mediante gráficas, los cambios de corriente eléctrica o las oscilaciones sonoras con el oscilógrafo, etc.
- Formar en la mente una imagen visual de un concepto abstracto
- Imaginar con rasgos visibles algo que no se tiene a la vista.
- Hacer visible una imagen en un monitor.

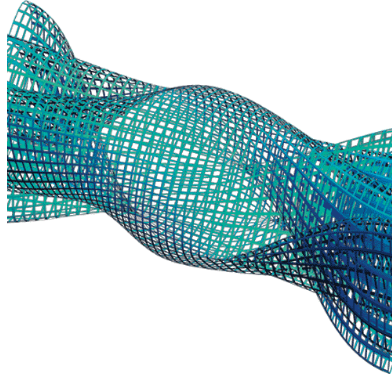
La definición que interesa y que mejor se adapta a las necesidades del presente documento es la última, referente al aspecto informático.

Científicos, ingenieros, personal médico y otros con frecuencia necesitan analizar grandes cantidades de información o estudiar el comportamiento de ciertos procesos. Las simulaciones numéricas que se efectúan en supercomputadoras a menudo producen archivos de datos que contienen miles e incluso millones de valores de datos. De modo similar, cámaras vía satélite y otra fuentes acumulan grandes archivos de datos mas rápido de lo que se pueden interpretar. El rastreo de estos grandes conjuntos de números para determinar tendencias y relaciones es un proceso tedios e ineficaz. Pero si se convierten los datos en forma visual, es frecuente que se perciban de inmediato las tendencias y los patrones.

Existen muchas clases de conjuntos de datos y los esquemas de visualización efectivos dependen de las características de los datos. Una compilación de datos contiene valores escalares, vectores, tensores de orden superior o cualquier combinación de estos tipos de datos. Los conjuntos pueden ser bidimensionales o tridimensionales. La codificación por colores es una de las diversas formas de mostrar los datos.

[22] Sito Web: http://www.rae.es/

Otras técnicas incluyen trazos, gráficas, diagramas de contorno, presentaciones de superficie y de interiores de volúmenes. Además de que se combinan las áreas del procesamiento digital de imágenes y la de las gráficas por computadora para crear las ya mencionadas visualizaciones de datos. Un ejemplo de visualización de datos es el mostrado en la Figura 4, en donde se muestra una superficie tridimensional generada a partir de una expresión analítica.



Curvas de Lissajous proyectadas
Figura 4

1.5 Reconocimiento de patrones

"1.- Las matemáticas son el lenguaje de la naturaleza
2.- Todo puede ser representado y ser comprendido con números
3.- Al graficar cualquier sistema surgen patrones
Por lo tanto, existen patrones en toda la naturaleza"[23]

Mucha de la información que se maneja en la vida real se presenta en forma de patrones complejos: caras, textos escritos, enfermedades, música, flores, piezas industriales, etc.

Para la Psicología, el problema central en el ámbito del reconocimiento de patrones el es estudio de los mecanismos por los que las señales externas estimulan los órganos sensoriales y se convierten en experiencias perceptuales significativas, o dicho de otra forma, cómo se realiza la clasificación de estos estímulos tan complejos asignándoles un nombre. Estos procesos continúan siendo desconocidos en su mayor parte y no se ha encontrado un modelo concluyente sobre cómo el sistema nervioso realiza este reconocimiento. No obstante, se admite que esta tarea debe realizarse siguiendo un esquema general como el que se detalla a continuación. Antes del *reconocimiento*, un patrón debe ser *percibido* por los órganos sensoriales. Además, el mismo patrón o alguno similar (de la misma clase) debe haberse percibido y *recordado* previamente. Finalmente, debe establecerse alguna *correspondencia* entre la percepción actual y lo recordado.

En resumen, esta aproximación al reconocimiento de patrones se centra en *el estudio del mecanismo de reconocimiento presente en los seres vivos*.

[23] PI, el orden del caos.

El uso intensivo de computadoras u otros dispositivos electrónicos en los últimos años ha impulsado el estudio y aplicación de técnicas de reconocimiento de patrones. En particular los dispositivos de adquisición de datos (sensores y transductores) y los convertidores A/D hacen que una computadora pueda ser "alimentada'' con datos (observaciones) del mundo real, utilizada para almacenar y recuperar información y para establecer la correspondencia entre observaciones pasadas y actuales. La aproximación al reconocimiento de patrones que comúnmente se adopta es la del *estudio de las teorías y técnicas de reconocimiento implementables en un sistema basado en computadoras*. En este sentido, las fuentes de este área son las Matemáticas (fundamentalmente la Estadística) y la Inteligencia Artificial, en sus aproximaciones algorítmicas a la resolución de problemas.

1.5.1 Aplicaciones muy eficientes en campos muy concretos

Aunque la aplicabilidad de las técnicas resulta muy amplia, no hay un método que sea la panacea. Diversas razones hacen que los sistemas de reconocimiento de formas operativos sean muy específicos del problema a resolver:

1) La naturaleza de los patrones: caracteres escritos, símbolos, dibujos, imágenes biomédicas, objetos tridimensionales, firmas, huellas dactilares, espectrogramas, imágenes de Teledetección, cromosomas, etc.
2) Los requerimientos del sistema, especialmente en tiempo de respuesta hace que algunos métodos de reconocimiento, aún siendo superiores en éxito no sean aplicables en la práctica.
3) Factores económicos: un sistema equipado con diferentes sensores y equipos de procesamiento muy potentes pueden dar resultados muy satisfactorios pero no pueden ser asumidos por los usuarios.

Estos factores hacen que un sistema adecuado para un problema sea inaplicable para otro, lo que posibilita el estudio y desarrollo de nuevas técnicas.

Debe considerarse, además, que el reconocimiento de patrones no constituye un campo de estudio cerrado sino que las técnicas relacionadas con este campo pueden encontrase en otras ramas de la Ciencia y de la Tecnología. De ahí que encontrar una definición formal para un campo tan diversificado sea imposible.

1.6 Sombreadores programables (*Shaders*)

Un *shader* (sombreador) describe los efectos que la iluminación causa sobre una superficie mediante la manipulación de sombras y de algunas cualidades de su textura[24] como son la tersura, el brillo e incluso volumen.

Por lo regular, un *shader* se presenta como un *script*[25] en el cual se describen los efectos especiales que serán aplicados sobre una textura al momento de ser dibujada.

El proceso de *render* puede ser en tiempo no real o bien en tiempo real. En el primer caso, los cálculos son hechos en su totalidad por el procesador en cooperación con la memoria RAM. Memoria en donde son almacenados los resultados de todos los cálculos intermedios necesarios.

---

[24] Textura: Representación digital de la superficie de un objeto. Además de sus propiedades bidimensionales (como color y brillantez), una textura es también codificada con propiedades tridimensionales como las capacidades de transparencia y reflexión del objeto.

[25] *Script*: Una lista de comandos que pueden ser ejecutados sin interacción del usuario.

Por más de una década el lenguaje de programación para gráficos *PhotoRealistic RenderMan* de *Pixar Animation Studios* ha sido la elección de los profesionales para enfrentar el problema del realismo digital (imágenes y/o video foto-realista).

La capacidad de programación (*programabilidad*) que permite *RenderMan*, le ha permitido evolucionar al parejo de las nuevas técnicas de *render* ya que no impone límites estrictos en cálculos y operaciones. Esto le da a los programadores una gran flexibilidad para crear y diseñar. Sin embargo, ésta misma capacidad condiciona al lenguaje a implementaciones en software, es decir, a un proceso de *render* en tiempo no real.

Cuando se trata de tiempo real, el cálculo de todas las operaciones es tan costoso (en lo que a tiempo de procesador cómo número de instrucciones se refiere), que es necesario hacer uso de hardware específico cómo lo son las tarjetas aceleradoras de video.

A pesar de que el rendimiento de las tarjetas aceleradoras de video ha aumentando considerablemente en cuanto a una más rápida ejecución de gráficos desde el lanzamiento de las primeras tarjetas 3dfx Voodoo en 1995, no se ha logrado que los gráficos ejecutados sean de mayor calidad.

La principal limitante de dichas tarjetas es que son de funciones fijas. Dado término refiere que los diseñadores del *GPU* han programado de manera permanente e inalterable ciertos algoritmos en los *chips* gráficos, obligando a los programadores a usar únicamente las rutinas previamente programadas.

Ahora bien, el hardware de bajo costo (en lo que a tarjetas aceleradoras de video se refiere) ha alcanzado el punto en el que pueden ser implementadas las bases del sombreado programable en una manera similar al lenguaje *RenderMan* pero con ejecución en tiempo real.

Las *APIs* principales 3D (*DirectX*[26] y *OpenGL*[27]) han evolucionado al mismo ritmo que el hardware para gráficos, de manera tal que una de las más importantes características nuevas de *DirectX* es la adición de una etapa de *pipeline*[28] que provee una interfaz de lenguaje ensamblador para las transformaciones e iluminación (llamado *Vertex Shader*) y el de píxel (*Píxel Shader*). Dicho etapa da al programador una mayor libertad para crear efectos que nunca se habían visto en aplicaciones de ejecución en tiempo real.

---

[26] DirectX: Conjunto de APIs desarrolladas por Microsoft para crear aplicaciones multimedia. Conjunto avanzado de APIs multimedia para sistemas operativos Windows. DirectX provee de una plataforma estándar de desarrollo para PCs basadas en Windows, permitiendo a los desarrolladores acceder a hardware especializado sin necesidad de escribir código que sea específico del hardware

[27] OpenGL: API de gráficos por computadora mas popular en el área. Desarrollada por *Sillicon Graphics Industries* y destinada al uso de operaciones específicas y objetos para producir aplicaciones interactivas en 3 dimensiones.

[28] *Pipeline*: Serie de etapas en la que se que efectúa una tarea determinada en varios pasos, al igual que en una línea de ensamblaje de una fábrica.

Resumen

A lo largo de este capítulo se han explicado los conceptos mas importantes y necesarios que se tratarán durante el desarrollo del presente trabajo. Las interfaces hombre-computadora son importantes ya que conforman el medio de comunicación entre el usuario y la máquina. Ambos hablan idiomas diferentes, por lo que se requiere de un intermediario que permita la interacción de ellos.

Hablar en estos días de interfaces requiere de varios términos como son ventana, botón, menú, etc., mismos que tienen una cosa en común: todos requieren de una representación gráfica en el monitor de la computadora para poder ser concebidos como lo que son. Es importante mencionar que una buena interfaz debe de cubrir diversos aspectos como son informar al usuario de lo que internamente sucede, comunicarse con el usuario de una manera que se le pueda comprender, se concretos y precisos en los mensajes e información que entreguen, flexibles en su uso, entre otros.

El éxito de una interfaz depende directamente de la habilidad, la experiencia y el entrenamiento que posea el humano, por lo tanto, se puede decir que la limitante del principal del tema tratado, el éxito, es proporcional a la capacidad que tenga el usuario frente a una computadora.

Por la necesidad de interfaces, se desarrollaron cada vez más y más las gráficas por computadora y poco a poco se fue trabajando en el ámbito hasta que se llego a definir lo que se conoce ahora por Realidad Virtual. Un concepto referente a la cualidad del ser verdadero en contraposición con la existencia en esencia. En otras palabras es el enfrentamiento de lo tangible con lo intangible o: que el conciente humano sea capaz de reconocer algo tangible pero que no existe, sino que solo resida en la conciencia.

La Realidad Virtual puede considerarse como un ambiente sintético con apariencia muy similar a la de la realidad, que sumerge al usuario en un lugar interactivo y, en ocasiones, reactivo a él, pero cuya representación especial y respuesta a estímulos está definida completamente por expresiones lógicas.

Los ambientes, como se acaba de mencionar, pueden ser reactivos o pasivos, inmersivos o no inmersivos. En los pasivos no existe interacción, solo una serie de datos que son desplegados ante los ojos del espectador mientras que en los interactivos, algunas acciones del espectador repercuten en el ambiente o en la historia de lo que se esté proyectando. La inmersión se especifica mediante la percepción que tenga el usuario del mundo virtual y el verdadero, si éste no ve mas que el ambiente virtual, se dice que se encuentra inmerso en el mundo. Caso contrario es cuando el usuario puede reconocer y discriminar los mundos real y virtual, entonces se dice que carece de inmersión.

Un ambiente virtual también puede ser clasificado por el tipo de aplicación para el que es destinada. Si se requiere que el usuario navegue libre por un mundo, entonces puede usarse una cueva. Para fines de simulación existen las cabinas en donde el interior de la cabina está completamente ligado con un mundo externo que puede existir. Otro caso interesante, es para fines de exploración o de investigación, por llamarlo de algún modo. Con esto se busca referirse al área de la Realidad Aumentada, conjunción de un entorno virtual con el mundo verdadero.

Supóngase que se sobrevuela por el desierto de Egipto, en específico sobre el área de las pirámides. Las personas que hagan el vuelo estarían interesadas en obtener información sobre las mismas para hacer mas ameno el viaje. ¿Cómo lograrlo? La empresa aérea ha contratado un equipo de desarrollo de software al que le ha solicitado una aplicación mediante la cual el usuario al ver las pirámides pueda obtener información de ellas e incluso algún otro tipo de representación como la manera de construcción, etc.

El equipo, al final del proceso, entrega una aplicación de Realidad Aumentada que consta de una cámara y un par de lentes de Realidad Virtual. Cuando el usuario se coloca los lentes y ve hacia las pirámides, en cuanto se distingue la esfinge al lado de la Gran Pirámide, aparecen en los lentes datos únicos sobre la construcción como la altura, antigüedad, para que faraón fue construida, etc. A su vez, si el usuario así lo desea, puede ver una animación sencilla y pequeña sobre como los hebreos las construían.

Una aplicación de gran interés y fascinante por todo lo que puede lograr, pero, ¿cuáles son los procesos involucrados? Se requiere construir toda una base gráfica para poder hacer la animación y desplegar todo el texto explicativo. A su vez, se debe de procesar el video adquirido, para posteriormente hacer un reconocimiento de cierto patrón en la imagen, que servirá de guía para seleccionar la información y colocar las estructuras gráficas necesarias.

Por lo tanto, para desarrollar aplicaciones de Realidad Aumentada, es necesario conocer los conceptos planteados en este capítulo. Ninguno es menos importante que otro, la interfaz humano-computadora, el entorno virtual, el reconocimiento de patrones, etc. Se incluyó el aspecto de los *shaders*, ya que ofrecen nuevas características de procesamiento que permiten tener acabado mas realista y de esta manera, intentar que la conjunción de lo virtual y lo real sea mejor, caracterizando lo virtual lo más posible a lo real.

Capítulo II
DirectX

DirectX es un conjunto de *APIs* de bajo nivel para el desarrollo de aplicaciones gráficas y multimedia de alto desempeño. Incluye soporte para gráficos bidimensionales y tridimensionales, efectos de sonido, dispositivos de entrada y funciones para operación en red. Dichas *APIs* contienen los siguientes componentes.

- *DirectX Graphics*. Combina los componentes *DirectDraw* y *Direct3D* (encontrados en distribuciones anteriores de *DirectX*) en una sola *API* que puede usarse para todo tipo de programación con gráficos. El componente incluye la biblioteca con las extensiones de *Direct3D* (D3DX), que simplifican muchas tareas involucradas en la programación con gráficos.
- *DirectInput*. Provee soporte para una variedad de dispositivos de entrada incluyendo la tecnología *force-feedback*[29].
- *DirectPlay*. Soporte para el uso de la aplicación en red.
- *DirectSound*. Puede ser usado en el desarrollo de aplicaciones de audio que reproducen y graban audio en formato *WAV*[30].
- *DirectMusic*. Aporta una solución completa para pistas de audio musicales y no musicales basadas en formas de onda (*WAV*), sonidos *MIDI*[31], entre otros formatos.
- *DirectShow*. Empleado para la reproducción, captura y procesamiento de video de alta calidad.
- *DirectSetup*. Una *API* que permite la instalación de componentes de *DirectX* mediante una sola llamada.
- *DirectX Media Objects*. Permiten escribir y usar objetos que transmiten un flujo de datos, incluyendo codificadores de video y audio, decodificadores y efectos.

A continuación se tratarán con mayor detalle las *APIs* de *DirectX Graphics* y *DirectShow*, ya que han sido usadas para la implementación del software desarrollado en esta tesis.

[29] *Force-feedback*: Realimentación de fuerza. Permite la percepción directa de objetos 3D y acopla las entradas y salidas entre la computadora y el usuario. Es una adición poderosa en los gráficos por computadora, ya que ayuda a resolver problemas que involucran el entendimiento de estructuras 3D, percepción de formas y reacción ante fuerzas en el mundo.

[30] *WAV*: Formato de audio creado por Microsoft e IBM.

[31] *MIDI*: Musical Instrument Digital Interface. Interfaz Digital de Instrumento Musical. Estándar adoptado por la industria electrónica de música para controlar dispositivos, como sintetizadores y tarjetas de sonido que emiten música.

2.1 *DirectX Graphics* o *Direct3D*

Cómo la mayoría de las *APIs* de *DirectX*, *DirectX Graphics* ha sido diseñado para ofrecer el máximo de velocidad en diferente hardware, una interfaz de programación común y compatibilidad con versiones anteriores. Por lo tanto, era necesario satisfacer los siguientes puntos:

- La *API* necesita de consistencia para ser viable.
- Si alguna característica no es soportada por hardware, tiene que haber un mecanismo que lo emule o bien desactive.
- La curva de aprendizaje para que los programadores comiencen a desarrollar aplicaciones, debe de mostrar resultados en poco tiempo.
- Asegurar el correcto funcionamiento de todas las aplicaciones desarrolladas para versiones anteriores de *DirectX* en sus futuras versiones.

Las respuestas ante tales demandas son las conocidos como Capa de Abstracción de Hardware (*HAL*[32]) o dispositivos de software conectable y el Modelo de Componentes de Objetos (*COM*[33]).

Las aplicaciones creadas con *DirectX Graphics* son diseñadas para obtener el máximo rendimiento en plataformas que contengan una tarjeta aceleradora de video. A pesar de esto, la *API* tiene soporte de software, con lo que se asegura la ejecución de la aplicación (con un rendimiento menor) siempre y cuando esta no emplee características únicas del hardware.

Ante este problema se implementó una emulación en software de dichas caracterísiticas. Hasta la versión 7.0 de *DirectX* se conoció por Capa de Emulación de Hardware (*HEL*[34]) y a partir de la versión 8.0 se le llamó por Dispositivos de Software Conectable. Mientras que DirectX 7.0 dotaba de una *HEL*, los nuevos dispositivos de software conectable son generalmente creados por el desarrollador si es que pretende la ejecución de su software en máquinas carentes de aceleración por hardware. La *HAL* es de gran utilidad desde el primer momento que se comienza a usar *Direct3D*, ya que es una pieza de software (provista por el creador del hardware acelerador) que permite el acceso a todas las características del mismo.

Cuando la aplicación emplea ciertos atributos no encontrados en el hardware se tienen dos posibilidades. La primera consiste en verificar las capacidades de la *HAL* con la que se cuenta en la PC y desactivar aquellas con que no se cuenta. La segunda opción es la de proporcionar al usuario una manera de poder cambiar de la *HAL* al dispositivo de software conectable.

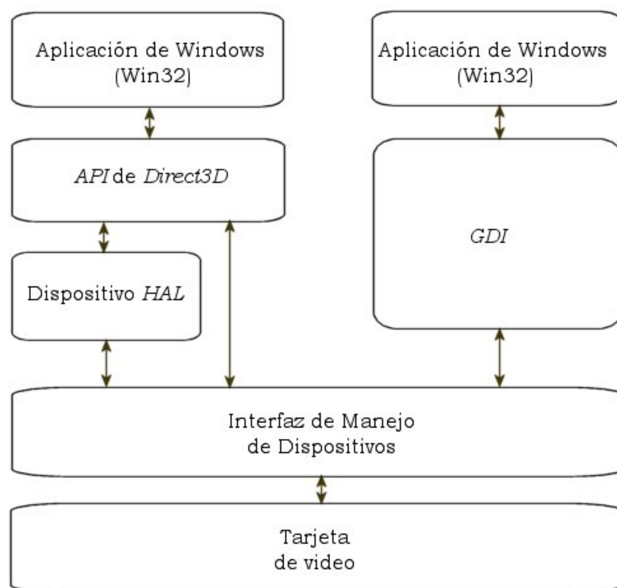
En adición a la *HAL* y a los dispositivos de software conectable, existe otra opción conocida como *Reference Rasterizer*[35]. Dichos dispositivos serán tratados a continuación.

2.1.1 *Direct3D HAL*

En la Figura 5 se pueden observar las relaciones existentes entre el *API* de *Direct3D*, la *HAL*, y la interfaz de dispositivos gráficos (*GDI*[36]).

---

[32] *HAL*: Hardware Abstraction Layer. Capa de Abstracción de Hardware. Véase sección 2.1.1
[33] *COM*: Component Object Model. Modelo de Componentes de Objetos. Véase sección 2.1.4
[34] *HEL*: Hardware Emulation Layer. Capa de Emulación de Hardware. Véase sección 2.1.2
[35] *Reference Rasterizer*. Véase Sección 2.1.3
[36] *GDI*: Graphics Device Interface. Interfaz de Dispositivos Gráficos. Es el estándar de MS Windows para representar objetos gráficos y transmitirlos a los dispositivos de salida, como monitores e impresoras.



Integración de *Direct3D* con el sistema
Figura 5

Como se puede observar, la *HAL* es empleada por *Direct3D* para tener acceso a la tarjeta de video a través de la Interfaz de Manejo de Dispositivos (*DDI*[37]). La *HAL* es el tipo de dispositivo principal que soporta aceleración por hardware y procesamiento de vértices por software y hardware. Siempre que la tarjeta de video del sistema soporte *Direct3D* se contará con una *HAL*.

Todo el código escrito para determinada aplicación que pretenda usar las capacidades de una *HAL*, debería funcionar con los dispositivos de emulación y de referencia sin modificaciones. Sin embargo, es conveniente realizar una revisión de las capacidades del hardware con que se cuenta antes de la ejecución de la aplicación.

2.1.2 Dispositivos de software conectable

Si el hardware en donde se prueba la aplicación no provee de todas las operaciones 3D necesarias para la ejecución, el software debe emular el hardware 3D mediante el empleo de un dispositivo de software conectable. Este tipo de dispositivos no soportan 256 colores ni modos de color de 8 bits, pero toma ventaja de aquellas instrucciones especiales soportadas por el microprocesador para aumentar el rendimiento de la aplicación, incluyendo las instrucciones *AMD 3D Now!* Así como de las *MMX/SIMD* de los procesadores Intel.

Actualmente, la emulación de hardware debe ser provista por el desarrollador de software, mientras que en las primeras versiones de *DirectX* era provisto por Microsoft en la forma de la *HEL*. Debido a esto, es posible que diversas capacidades de *Direct3D* no sean soportadas.

---

[37] *DDI*: Device Driver Interface. Interfaz de Manejo de Dispositivos.

2.1.3 *Reference Rasterizer*

Este dispositivo soporta todas las características de *Direct3D*. Debe ser usado solo para propósitos de prueba de aquello no soportado por hardware. Este dispositivo está optimizado para precisión y no para velocidad.

2.1.4 Modelo de Componentes de Objetos (*COM*)

En términos generales, un componente de software es simplemente una porción de código que ofrece servicios a través de interfaces. Una interfaz es un grupo de métodos relacionados. Un *COM*, por lo general, es un archivo *DLL*[38] que puede ser accedido en una manera bien definida y consistente.

Las ventajas de utilizar objetos *COM* cuando se programa usando *Direct3D* son:

- Las interfaces nunca cambian. Las aplicaciones que hacen uso de ellas no necesitan ser compiladas nuevamente cuando una interfaz cambia, porque los objetos COM no proveen una manera estándar de comunicación con otros objetos cuando la interfaz de un objeto es modificable.
- Son independientes del lenguaje de programación. No importa el lenguaje que se emplee, los objetos *COM* son accesibles desde diversos lenguajes y compiladores. La importancia de esta independencia recae en la flexibilidad de programación ofrecida a los desarrolladores. Esto es que la aplicación puede ser creada por diferentes personas en diferentes lenguajes y se garantice la funcionalidad de la misma.
- Solamente se puede acceder a los métodos, nunca a sus datos. Esto es un buen diseño de objetos, pues permite el encapsulamiento[39] de los mismos.

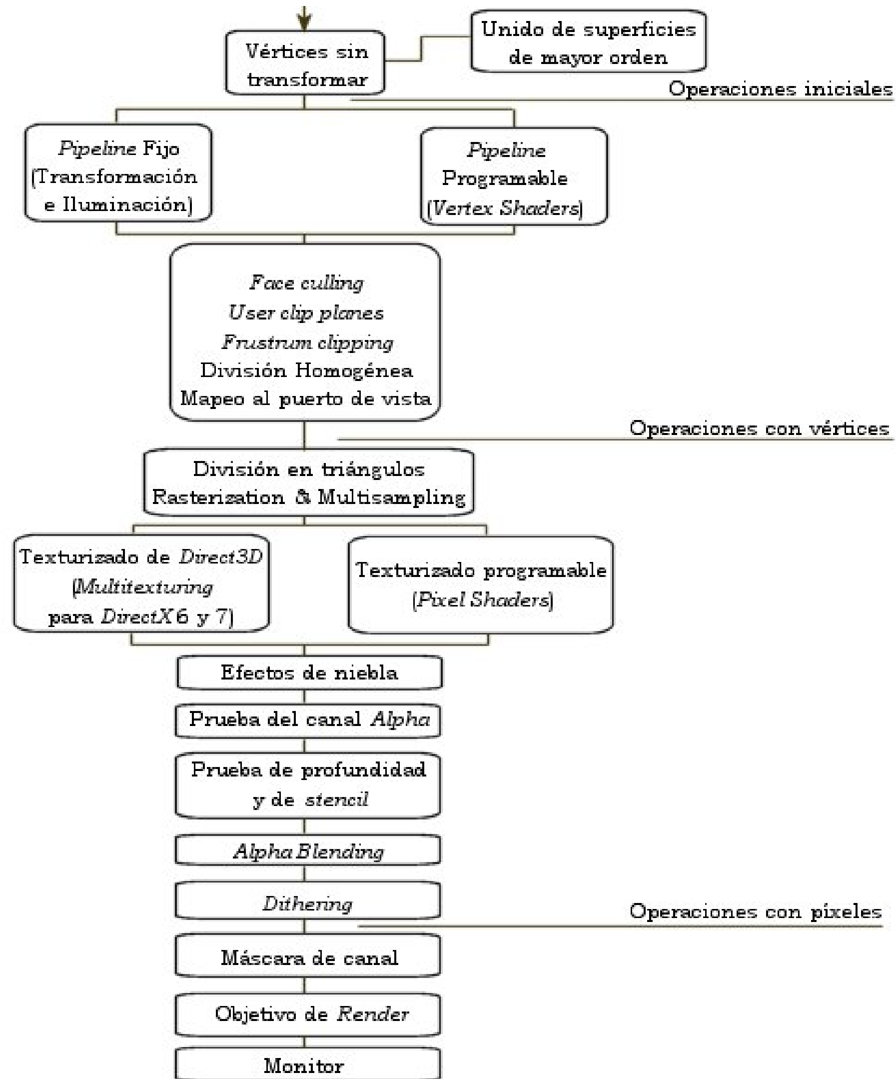
Los dispositivos que *Direct3D* permite utilizar han sido tratados. Es turno de analizar la manera en que esta *API* lleva a cabo el proceso de dibujo.

2.1.5 *Pipeline* de *Direct3D*

A continuación se muestra un esquema simplificado del proceso de *pipeline* de *Direct3D* (Figura 6).

[38] *DLL*: Dynamic Link Library. Bibliotecas de Enlace Dinámico. Bibliotecas que proveen de una o más funciones particulares y un programa tiene acceso a ellas mediante la creación de una liga estática o dinámica al *DLL*.

[39] Encapsulamiento: Proceso de combinar elementos para crear una entidad nueva. Por ejemplo, una función es un tipo de encapsulamiento porque combina una serie de instrucciones de computadora.



Pipeline* de *Direct3D
Figura 6

En el nivel base, los vértices son unidos. Este es el módulo donde son procesadas las primitivas de alto orden y en donde se trabaja para unir a las mismas. Estas primitivas incluyen los *N-Patches*, curvas de Bézier, *B-splines* y *patches* rectangulares y triangulares.

La siguiente etapa cubre las operaciones que se efectúan sobre los vértices. Se cuenta con dos formas para procesarlos:

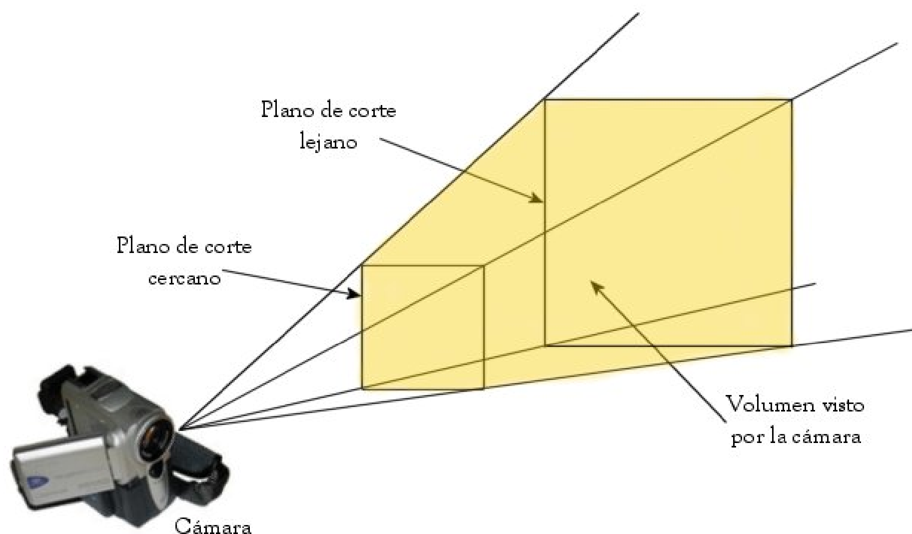
1. El *pipeline* estándar de Transformación e Iluminación (*T&L*[40]), donde el procesamiento es controlado por el programador seleccionando los estados de *render*, las matrices de transformación, las luces y los materiales.
2. *Vertex Shaders*. Es el nuevo mecanismo introducido a partir de la especificación 8.0 de *DirectX*. En lugar de seleccionar los parámetros de control, se escribe un programa de *vertex shader* que es ejecutado en el hardware (tarjeta de video).

[40] *T&L*: Transformation and Lighting. Transformación e Iluminación.

En la sección 2.3 se tratará con mayor detalle a los *vertex* y *pixel shaders*. Por ahora se continuarán analizando las siguientes etapas del proceso.

Una vez que el vértice ha sido procesado, éste abandona la etapa como un vértice con color y transformado. Entonces pasa por la etapa de *Face Culling*[41] en la que son removidos todos los triángulos que no se encuentran frente a la cámara. Por omisión, se trata de los vértices agrupados en sentido antihorario. Esta etapa es necesaria para reducir el tiempo en el que son dibujados los vértices ya que, en promedio, la mitad de los triángulos totales que componen la escena no se verán en cualquier instante.

Los planos de corte de usuario son definidos por el desarrollador con el fin de eliminar aquellos triángulos que no son necesarios (por ejemplo aquellos que estén muy alejados) y así reducir el número de cálculos. El *Frustum Clipping*[42] es otro tipo de recorte en donde aquellas primitivas que se encuentran parcial o totalmente fuera de pantalla son recortadas a la región definida por el puerto de vista, que es una representación en perspectiva y 3D. Esta representación es una pirámide en donde la cámara se encuentra en la punta. Por consiguiente, el interior del volumen aunado a los planos de corte definen lo que la cámara verá.



Frustum Clipping
Figura 7

La división homogénea tiene como fin el dar la apariencia de profundidad real a la escena. Esto es, las componentes *x*, *y, z* de las coordenadas de posición de los vértices transformados, son dividas por *w*. De esta manera se logra observar que los objetos grandes se ven más cercanos mientras que los pequeños aparentan lejanía, de la misma manera que se esperaría en la realidad. Las coordenadas en espacio de recorte son también conocidas como Coordenadas de Dispositivo Normalizadas (*NDC*[43]). Estas coordenadas son transformadas y mapeadas a la pantalla. Esto es lo que se conoce como mapeo a puerto de vista (*Viewport Mapping*).

[41] *Face Culling*: Etapa en la que se lleva evalúa cuales son las caras cuya normal tiene componente en dirección de la cámara. Útil para ocultar y no dibujar aquellos polígonos que no se vean.
[42] *Frustum Clipping*: Recorte de la escena vista en perspectiva.
[43] *NDC*: Normal Device Coordinates. Coordenadas de Dispositivo Normalizadas.

La siguiente etapa es la de triangularización, en donde se computan los parámetros (triángulo por triángulo) requeridos para delinear y rellenar los triángulos. En esta misma etapa se lleva a cabo el *Multisampling*[44], el cual solamente afecta a los triángulos y grupos de triángulos y no a las líneas. Mediante esto se logra incrementar la resolución de las aristas de los polígonos.

A continuación se encuentran los niveles de operaciones con píxeles. Cuando se emplea la unidad por *default* de *Direct3D* (*Multitexturing DX6/7*), el desarrollador trabaja de manera similar a como se trabajó con la unidad de *T&L*. Es decir, únicamente tiene que activar, desactivar o elegir estados de *render* y etapas de textura y seleccionar las texturas que se vayan a usar. Cuando se emplean los llamados *píxel shaders*, se tiene acceso a aquellas características programables del hardware gráfico. Un píxel shader toma los píxeles provenientes de la etapa anterior con color e información de textura. Los píxeles sombreados mediante sombreado de Gouraud[45] o plano (*Flat*[46]), pueden ser combinados en este dispositivo con la componente especular de color y color proveniente de la textura. Las unidades de *píxel shaders* serán explicadas con mayor detalle en la sección 2.3.

En la etapa de niebla, un factor es calculado y aplicado al píxel usando una operación de mezclado para así combinar la cantidad de niebla (color) y el color del píxel ya sombreado, tomando en cuenta la distancia a la que se encuentra el objeto. La distancia es calculada a partir de la componente en *z*, *w* o bien usando un valor de atenuación que mide la distancia ente la cámara y el objeto. Si la niebla se calcula por vértice, ésta es interpolada en cada triángulo usando sombreado de Gouraud.

La prueba de canal alfa[47] consiste en eliminar a los píxeles con un valor específico de dicha componente. Ésta es una forma de elaborar degradados con una textura mapeada que contiene componente alfa. Cabe mencionar que el llevar a cabo esta operación no es causa de procesamiento extra, incluyendo el caso en el que todos los píxeles pasan la prueba. Si el valor adecuado de alfa es elegido correctamente, y se evita dibujar todos aquellos píxeles transparentes, el rendimiento de la aplicación se verá favorecido en términos de velocidad. EL *Stencil Test*[48] enmascara[49] el píxel con el contenido del *stencil buffer*[50]. Esto es útil cuando se pretende obtener un degradado, delinear o construir sombras con volumen.

[44] *Multisampling*: Es un método de *antialiasing* en el que un almacenamiento de píxeles adicionales, se mantiene como parte de los *buffers* de color, profundidad y *stencil*; lo anterior con el fin de establecer ciertas máscaras que ayudarán en el proceso de mantener la información del almacenamiento de píxeles actualizada.

[45] *Gouraud Shading*: Uno de los algoritmos más usados para sombreado suave. El nombre proviene del apellido de su creados, el francés Henri Gouraud. Esta técnica es basada en la interpolación de color a lo largo de las caras de todos los polígonos para determinar el color de cada píxel.

[46] *Flat shading*: También llamado sombreado constante. Para dibujar, asigna un solo color a toda las superficie del polígono. Es la técnica de sombreado con más baja calidad, en donde la superficie del objeto aparenta estar compuesta por muchos bloques.

[47] *Alpha channel*: La componente alfa del color nunca es mostrada. Únicamente es empleada para controlar las mezclas entre colores. En otras palabras, establece el nivel de transparencia del color.

[48] *Stencil Test*: Prueba de Stencil. Esta prueba es usada para enmascarar regiones, tapar sólidos geométricos y generar transparencias entre polígonos.

[49] Enmascarar: Aplicación de un operador lógico sobre datos para extraer los bits de importancia y operar con ellos. La máscara mas empleada es la operación AND con los bits de interés encendidos.

[50] *Stencil Buffer*: Memoria que es usada para hacer pruebas por fragmentos en conjunto con el *buffer* Z.

La prueba de profundidad o *Depth Test*[51] determina cuando un píxel es visible comparando su valor de profundidad con el valor guardado. Una aplicación puede especificar las *z* mínima y máxima, propias para el *buffer z*[52]. Ésta prueba es una operación lógica efectuada píxel por píxel en donde se pregunta si el píxel actual se encuentra atrás de otro en ese lugar. Si se responde que sí, el píxel es descartado, si la respuesta es no, entonces ese píxel continuará avanzando por el *pipeline* y se actualizará el *buffer z*.

La etapa de *Alpha Blending*[53] mezcla los datos del píxel actual con los del píxel que se tiene ya en el objetivo de *render*. Se usó principalmente para mezclar diferentes texturas, pero ahora esto se lleva a cabo mediante la unidad de *píxel shader*. También es usado para simular diferentes niveles de transparencia. El *Dithering*[54] intenta engañar al usuario pretendiendo que existen mas colores de los que realmente se presentan. Esto se logra colocando píxeles de diferente color cerca de otro para crear un color compuesto que el usuario apreciará. Por ejemplo, el usar amarillo cerca de azul generará la sensación de verde.

Finalmente, el objetivo de *render* es por lo regular el *Backbuffer*[55] de la aplicación, en el cual es dibujada la escena, pero podría ser también la superficie de una textura. Los datos en el *Backbuffer* son enviados al monitor para su despliegue y fin del *pipeline*.

2.2 *DirectShow*

DirectShow es una arquitectura que se emplea para el manejo de archivos multimedia bajo la plataforma Windows. Entre sus características principales destacan la captura y reproducción de archivos multimedia de alta calidad, su amplia variedad de formatos soportados (*ASF*[56], *AVI*[57], *MPEG*[58], *MP3*[59] y *WAV*) así como la compatibilidad con *WDM*[60] y dispositivos de video anteriores para *Windows*.

---

[51] *Depth Test*: Para eliminar superficies ocultas, el *buffer* z guarda la z más próxima al usuario en cada píxel. El valor de cada fragmento nuevo usa el dato almacenado para hacer una comparación y decidir si ese píxel es o no dibujado.

[52] *Buffer Z*: Memoria que almacena el valor de profundidad de cada píxel.

[53] *Alpha Blending*: Es una técnica para generar transparencia, en donde el color resultante de cada píxel es una combinación de los colores principales y de fondo.

[54] *Dithering*: Útil para lograr colores con calidad de 24 bits en *buffers* que solo son de 8 o 16 bits. Consiste en la utilización de dos colores para simular un tercero, dándole una apariencia suave.

[55] *Backbuffer*: Es un espacio en memoria de video donde son dibujados los gráficos antes de ser enviados al FrontBuffer para ser mostrados al usuario.

[56] *ASF*: Advance Systems Format. Es un formato extensible diseñado para almacenar datos multimedia sincronizados. Soporta el flujo de datos sobre una gran variedad de redes y protocolos además de permitir la reproducción de manera local.

[57] *AVI*: Audio Video Interleaved. Es un caso especial del formato *RIFF* (Resource Interchange File Format) pero definido por Microsoft. Es el formato más común de video en la PC.

[58] *MPEG*: Moving Pictures Expert Group. Uno de los formatos pertenecientes a la familia de estándares para compresión de video desarrollados por el grupo. *MPEG* generalmente brinda una calidad de video que otros formatos competidores, tales como Video for Windows, Indeo o QuickTime. La decodificación puede llevarse a cabo mediante software o hardware.

[59] *MP3*: MPEG Layer 3. Es uno de tres esquemas de codificación para la compresión de señales de audio. Usa codificación de audio perceptual y compresión psico–acústica para remover información superflua para el oído humano. Además implementa un filtro basado en la transformada discreta de coseno modifica con lo que se incrementa la resolución en frecuencia hasta 18 veces mas que en Layer 2. El resultado es música digital con calidad de CD con una compresión de un factor de 12.

[60] *WDM*: Windows Driver Model. Una tecnología desarrollada por Microsoft para la creación de *drivers* y cuyo código es compatible con Windows 98, 2000, Me y XP. Trabaja canalizando parte del trabajo del *driver* del dispositivo en porciones de código que son integradas en el sistema operativo. Estas porciones de código manejan todas las funciones de memoria de bajo nivel como son el *DMA* y *Plug and Play*.

Además de esto, *DirectShow* es integrable con otras tecnologías de *DirectX* lo cual ofrece una flexibilidad para la creación de aplicaciones que necesiten la reproducción de archivos multimedia y, por ejemplo, escenas tridimensionales generadas en tiempo real. También ofrece un buen rendimiento ya que soporta aceleración por hardware siempre y cuando éste lo ofrezca.

El trabajar con archivos multimedia presenta varios retos, entre ellos destacan los siguientes:

- Contienen grandes cantidades de datos, los cuales deben ser procesados muy rápido.
- El audio y el video debe ser sincronizado, de tal manera que inicien y se detengan al mismo tiempo y sean reproducidos a la misma tasa.
- Los datos provienen de muy diversas fuentes, incluyendo archivos locales, redes de computadoras, transmisiones televisivas y video cámaras.
- Existe una amplía variedad de formatos en que se presentan los datos. Estos pueden ser *AVIs*, *ASFs*, *MPEGs* o bien video digital (*DV*[61]).
- El programador no sabe el hardware con el que cuenta el usuario final.

De esta manera, *DirectShow* ha sido diseñado para responder a tales retos, aislando a las aplicaciones de las complejidades del transporte de datos, las diferencias de *hardware* y la sincronización. Para resolver la necesidad de transmisión de audio y video, esta *API* hace uso de otras *APIs* de *DirectX* como son *DirectDraw* y *DirectSound*. El hacerlo así, permite el uso eficiente del hardware con que cuenta el usuario (tarjetas de sonido y video).

Para cubrir la amplia variedad de formatos y hardware, se emplea una arquitectura modular en la que la aplicación mezcla y hace corresponder diferentes componentes de software llamados filtros[62]. Los filtros que provee *DirectShow* soportan dispositivos de captura y sintonía basados en *WDM*, filtros para las ya antiguas tarjetas con soporte de *VFW*[63] y *codecs*[64] escritos para las interfaces del manejador de compresión de audio (*ACM*[65]) y del manejador de compresión de video (*VCM*[66]).

2.2.1 Arquitectura de *DirectShow*

El diagrama de la Figura 8 muestra las relaciones entre la aplicación, los componentes de *DirectShow* y algunos de los componentes en hardware y software que dicha *API* soporta.

---

[61] *DV*: Digital Video. Se refiere a la captura, manipulación y almacenamiento de video en formato digital. Una cámara de video digital, por ejemplo, es una cámara de video que captura y almacena imágenes en un medio digital como tarjetas de memoria o cintas magnéticas.

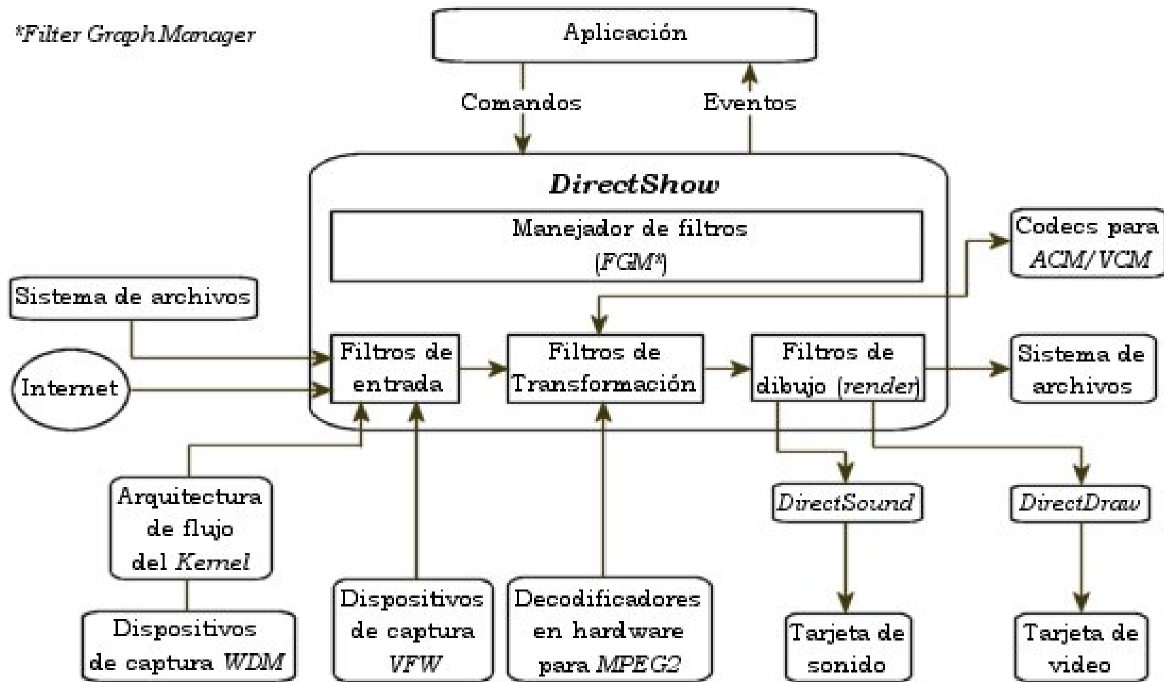
[62] Véase sección 2.2.2

[63] *VFW*: Video For Windows. Formato desarrollado por Microsoft para almacenar información de audio y video. El formato empleado es el *AVI*, estando limitado a una resolución de 320 x 240 píxeles y 30 cuadros por segundo. Ninguna de las especificaciones permite la reproducción a pantalla completa.

[64] *Codec*: Compressor/Decompressor. El término *codec* se emplea para referirse a cualquier tecnología para comprimir y descomprimir datos. Pueden ser implementados en software, hardware o ambos.

[65] *ACM*: Audio Compression Manager. Permite que una aplicación convierta datos entre diferentes formatos. Por sí mismo, el ACM no es capaz de hacer las conversiones, por ello se cuenta con *drivers* especiales que permitan su funcionamiento. Estos *drivers* son llamados *codecs*, conversores o filtros.

[66] *VCM*: Video Compression Manager. Permite el acceso a la interfaz empleada por los *codecs* para el manejo de datos en tiempo real.



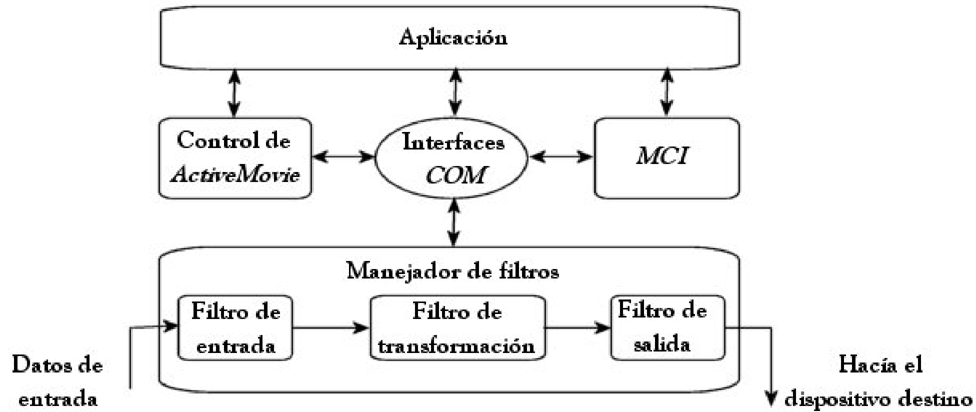
Arquitectura de *DirectShow*
Figura 8

La arquitectura de *DirectShow* define la manera en que los flujos de datos multimedia son controlados y procesados mediante el uso de componentes modulares llamados filtros. Los filtros tienen *pines*[67] de entrada, salida o ambos y son conectados entre ellos en una configuración llamada *FilterGraph*. Es así que las aplicaciones usan un objeto llamado manejador de filtros para ensamblar el *FilterGraph* y transmitir datos a través de él.

Por *default*, este manejador automáticamente se encarga del flujo de datos; por ejemplo, inserta el *codec* cuando éste se necesita y, de la misma manera, conecta un filtro de transformación a un *pin* del filtro de salida. Si no se desea emplear la configuración por *default*, es posible especificar una propia.

El manejador de filtros provee un conjunto de interfaces *COM* para que las aplicaciones puedan tener acceso al *FilterGraph*. Las aplicaciones pueden directamente llamar las interfaces mencionadas para controlar el flujo de información, conocer eventos de los filtros o usar la aplicación *Windows Media Player* para controlar la reproducción de archivos. Por lo tanto, es posible tener acceso a *DirectShow* por medio de una interfaz *COM*, el control de *Windows Media Player* o las interfaces de control para multimedia (*MCI*) tal y como es mostrado en la Figura 9.

[67] *Pin*: Puntos de conexión en un diagrama. El término es principalmente usado en el area eléctrica, pero debido a su generalidad es usado también en el ámbito de la computación para indicar entradas y/o salidas.



Acceso de una aplicación al manejador de filtros
Figura 9

2.2.2 Manejador de filtros y *FilterGraph*

Se trata de un objeto *COM* encargado de controlar los filtros y que lleva a cabo diferentes funciones entre las que se encuentran:

- Coordinar los cambios de estado entre los filtros. Estos cambios deben de llevarse a cabo en un orden particular. Por lo tanto, la aplicación no emite comandos de cambio de estado directamente a los filtros, sino que genera un comando para el manejador de filtros, quien distribuye el comando a cada uno de los filtros en uso.
- Establecer un reloj de sincronía. Todos los filtros que componen al diagrama usan el mismo reloj para asegurar que todos los flujos de datos se encuentren sincronizados. Al tiempo en donde un cuadro de video o señal de audio deben ser emitidos se le conoce por tiempo de presentación, el cual es medido con respecto del reloj de sincronía.
- Informar sobre eventos a la aplicación principal. El manejador de filtros emplea una cola de eventos que informa a la aplicación de eventos que ocurren en el *FilterGraph*. Este mecanismo es similar al ciclo de mensajes de Windows.
- Proveer de métodos a las aplicaciones para construir el *FilterGraph*. Permite la construcción, conexión y desconexión de filtros pero no se encarga de la transmisión de datos de un filtro a otro. Dicha transmisión es llevada a cabo por los filtros mismos a través de las conexiones entre sus pines.

Ahora bien, se ha hablado del manejador y cual es el papel que éste desempeña. Es turno de hablar acerca de los filtros y del diagrama de filtros o *FilterGraph*. Con motivo de explicar de una manera más sencilla el funcionamiento de un filtro y como es que se estructura una pequeña aplicación de reproducción, se tomará el caso de reproducir un video en formato *AVI*. Obsérvese la siguiente figura (Figura 10):

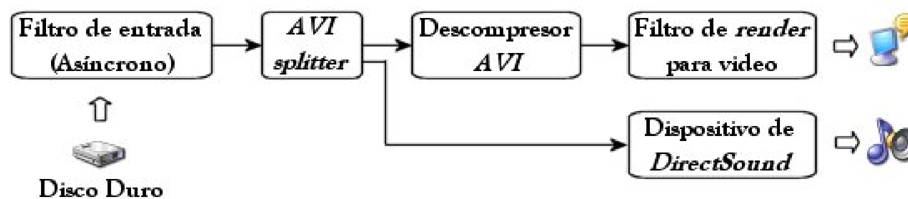


Diagrama de filtros para reproducir un archivo *AVI*
Figura 10

En la figura se pueden observar diversos bloques funcionales. Cada uno de ellos representa un filtro que hace una tarea en específico. De esta manera, la secuencia de eventos que tienen lugar para reproducir un video en formato *AVI* es la siguiente:

- Se leen los datos del archivo en formato binario (Filtro de entrada o *File Source Filter*).
- Se revisa la sintaxis de las cabeceras del *AVI* y se analiza la información contenida en los cuadros de video y las muestras de audio (*AVI Splitter Filter*).
- Se decodifican los cuadros de video (el número de filtros de decodificación depende del tipo de compresión que se haya usado cuando se creo el archivo).
- Se dibujan los cuadros de video (*Video Renderer Filter*).
- Se envían las muestras de audio a la tarjeta de sonido (*DirectSound Device Filter*).

De esta manera, cada filtro es conectado a uno o más filtros. Los puntos de conexión son también objetos COM, llamados pines. Los filtros usan pines para transferir datos de un filtro al siguiente. Las flechas mostradas en la Figura 8 representan la dirección en la que los datos viajan. Para *DirectShow*, un conjunto de filtros conectados es conocido como *FilterGraph* o Diagrama de filtros.

Los filtros pueden tener tres estados: ejecución, pausa y detenido. Cuando un filtro se encuentra en ejecución, los datos son procesados. Si está detenido, el procesamiento termina. El estado de pausa es usado para preparar los datos antes de la ejecución.

2.3 *Shaders* programables para *DirectX Graphics*

Los *shaders* han sido creados con el fin de reemplazar la apariencia de plástico mate que ha sido una constante de las gráficas por computadora usadas para algunos programas y juegos en los últimos años. Por tal razón, mediante su creación y utilización se pretende dar una mayor libertad a los artistas y programadores de forma tal que sean capaces de generar estilos únicos para las aplicaciones por venir. Además, los *shaders* han sido diseñados para ser ejecutados en hardware y en tiempo real, controlando solo aquello que el programador y/o artista hallan decidido afectar. Utilizando este nuevo concepto se puede lograr:

- Efectuar transformaciones geométricas básicas.
- Animar la geometría (*blend*[68] y *skinning*[69]).
- Generar información de color (componentes especular y difusa).
- *Tweeninig*[70] de vértices entre matrices de transformación.
- Generar y transformar coordenadas de textura.
- Usar un modelo de iluminación definido por el programador.
- Crear mapeos de ambiente[71] y de *bump*[72].

---

[68] *Blend*: Técnica basada en la interpolación de cuadros de animación. Mediante esta técnica es posible mezclar dos animaciones diferentes, causando que sean mostradas al mismo tiempo y, en caso que se desee, que tengan la misma duración.

[69] *Skinning*: Una técnica para producir animaciones fluidas y suaves basadas en cuadros de animación por huesos.

[70] *Tweening*: Técnica de interpolación entre dos o mas cuadros para generar animaciones.

Las opciones de creación tienen como limitante el ingenio y, principalmente, el tamaño del *buffer* para el *shader*, lo que limita la complejidad que pueda tener.

Escribir un *shader* para *DirectX Graphics* es como programar en lenguaje ensamblador, lo que convierte a la tarea en algo simple, obvia y en ocasiones muy tediosa. Una de las bondades de escribir en ensamblador, es que se sabe lo que ha de ocurrir, haciendo el proceso de búsqueda de errores más sencillo así como el desecho de todas aquellas instrucciones o líneas de código que no aporten al desempeño ni objetivo del *shader*.

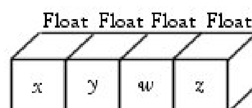
A continuación se describirán las unidades programables de Vertex y Píxel Shader una a una y les proseguirá una descripción acerca del lenguaje ensamblador para cada una de las unidades programables.

2.3.1 Unidad de *Vertex Shader*

Las unidades de *Vertex Shader* reemplazan a la etapa de *T&L* del *pipeline* de *Direct3D*, enfocándose en realizar la misma tarea que dicha unidad: tomar un conjunto de vértices como entrada, procesarlos y emitir un conjunto de vértices transformados como salida. Ya que existe una correspondencia uno a uno entre el vértice de entrada y el transformado, un programa para *vertex shader* no puede cambiar el número de vértices porque dicho programa es llamado una vez por vértice. A continuación se explicará la arquitectura y funcionamiento de una de estas unidades.

2.3.1.1 Arquitectura de un Vertex Shader

Estas unidades pueden ser vistas como un procesador *SIMD*[73] ya que una sola instrucción es efectuada sobre un conjunto de hasta 4 variables de 32 bits cada una (ver Figura 11).



Registros de un *Vertex Shader*
Figura 11

Este formato de registro es ideal para los gráficos por computadora, debido a que la mayoría de los cálculos de transformación e iluminación son hechos usando matrices o cuaterniones de 4x4. Las instrucciones son simples y sencillas de entender.

---

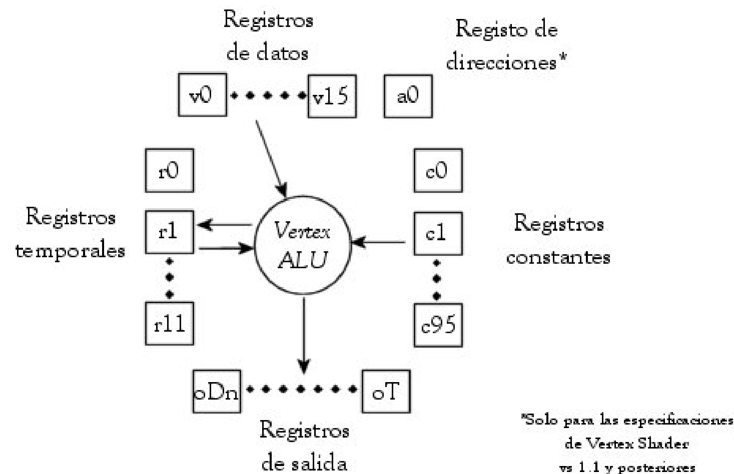
[71] *Environment Mapping*: Mediante el uso de esta técnica se logra un efecto en el que la superficie afectada refleja el medio que lo rodea.

[72] *Bump Mapping*: Técnica de mapeo de textura que logra un efecto de volumen a las superficies. Se logra variando la iluminación de cada píxel de acuerdo con los valores en la textura de *bump* o *Height Map*. Con cada píxel que se dibuja, se lleva a cabo una búsqueda en una textura que simula el relieve de la superficie (*Height Map*). Los valores en dicha textura son usados para modificar las normales del píxel en cuestión. Posteriormente, los cálculos de color e iluminación se basan en la normal modificada teniendo como resultado final una superficie con textura que aparenta volumen.

[73] *SIMD*: Single Instruction Flow, Multiple Data Flow. Arquitectura de computadoras paralelas que caracteriza a la mayoría de las computadoras vectoriales. Una instrucción causa que mas de un procesador realice la misma operación de manera síncrona, aun con diferentes datos.

Entre ellas no se encuentran ciclos ni saltos condicionales o incondicionales, haciendo que el programa sea ejecutado de manera lineal. En *DirectX* el número máximo de instrucciones en un programa de *vertex shader* se encuentra limitado a 128 y la posible combinación de ellos para transformar con uno e iluminar con otro es imposible. Lo anterior se debe a que solo se puede usar un programa a la vez y este se debe de encargar de todos los datos requeridos por vértice.

Véase la siguiente figura (Figura 12) para comprender la arquitectura de una de estas unidades.



Arquitectura de un *Vertex Shader*
Figura 12

Un *vertex shader* contiene 16 registros de entrada (v0-v15) para tener acceso a los datos de entrada. Cada registro es de 128 bit, espacio en el que se almacena la posición del vértice y, considerando que existen 16 registros, se pueden almacenar además otras características de cada uno los vértices como son la normal, las componentes de color difusa y especular, valores de niebla e inclusive las coordenadas de textura.

El CPU carga los datos contenidos en los registros constantes antes de que la unidad de *vertex shader* comience a ejecutar los parámetros definidos por el programador. Estos registros son usados para mantener datos como la posición de la luz, matrices de transformación, datos para efectos de animación, datos de interpolación de vértices para hacer *morphing*[74] de objetos, entre otros. Dichas constantes pueden ser usadas varias veces en el programa y hacerles referencia mediante un direccionamiento indirecto con la ayuda del registro de direcciones (a0.x), pero solamente una constante puede ser usada por instrucción. Si una instrucción requiere de dos o mas datos contenidos en los registros constantes, ellos deben ser cargados con anterioridad en los registros temporales. En el caso de las tarjetas *Nvidia GeForce 3 Ti/4 Ti* se cuenta con 96 registros constantes (c0-c95) mientras que en las tarjetas *ATI Radeon 8500* y superiores se tiene 192 (c0-c191).

Se cuenta con 12 registros temporales. Estos son de lectura/escritura por lo que pueden ser empleados para almacenar y/o guardar cálculos intermedios. Dependiendo del hardware con el que se cuente, pueden existir hasta 13 registros de salida. Los nombres de estos registros siempre comienzan con una 'o' y solo permiten la escritura en ellos. El resultado que es entregado a estos registros es un vértice ya transformado.

[74] *Morphing*: Transformación animada en la que un objeto se convierte en otro. Esto se logra mediante una deformación de los vértices del primero hasta que coinciden algunos de ellos con los del segundo.

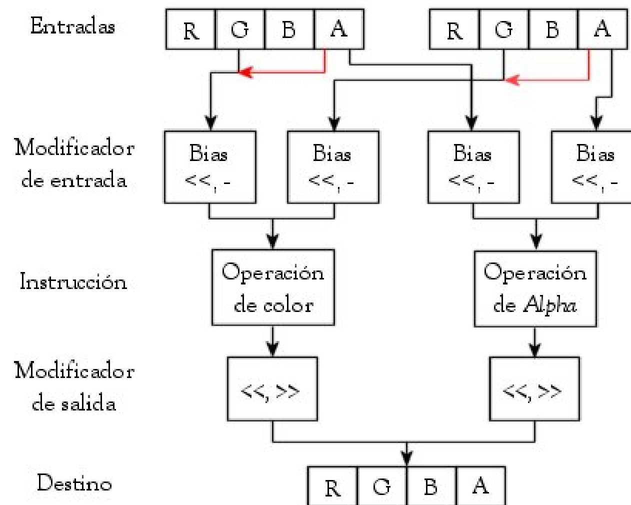
En la siguiente tabla (Tabla 1) se muestran algunos de los registros con que se cuenta.

Registros	Número de registros	Propiedades
Entrada (v0-v15)	16	Lectura
Salida (o*)	*GeForce 3/4Ti*: 9, *Radeon 8500*: 11	Escritura
Constantes	*GeForce 3/4Ti*: 92 (c0-c91), *Radeon 8500*: 192 (c0-c191)	Lectura
Temporales (r0-r11)	12	Lectura/Escritura
Direcciones (a0.x)	1	Escritura

Registros de un *Vertex Shader* 1.1 y posteriores
Tabla 1

2.3.2 Unidad de *Píxel Shader*

Mientras que los *vertex shaders* sustituyen la etapa de *T&L* en el *pipeline* de *render* tradicional, los *píxel shaders* reemplazan la sección de multitextura. Para entender como es que operan estas unidades, se debe de estar familiarizado con la naturaleza dual del *pipeline*. De manera tradicional, dos trayectorias son ejecutadas paralelamente en el hardware. Una de ellas es la de color (o *vector pipe*) y la otra es la de alfa (o *scalar pipe*). Estas dos trayectorias se encargan de todas las operaciones referentes al color y alfa de la unidad de texturizado. Usualmente, el programador establece un conjunto de parámetros para el color y otro diferente para el alfa. Al final, los resultados son combinados en valor *RGBA*[75] resultante. Esto puede ser observado en la Figura 13.



Operaciones que ejecuta un *Píxel Shader* en paralelo
Figura 13

A pesar de que el *pipeline* tradicional permite especificar una secuandia de operaciones sobre texturas, los *píxel shaders* permiten especificarlas de manera más general. La dualidad es conservada, pero el lograr que las operaciones de color y alfa sean ejecutadas paralelamente en el hardware lleva mas tiempo de desarrollo y a la larga, un mejor desempeño.

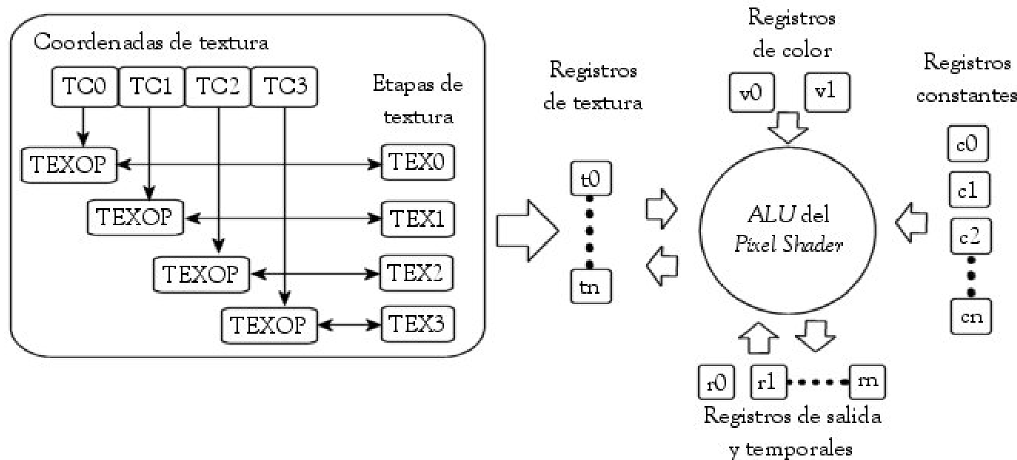
Otra dualidad de esta unidad es que posee dos tipos de operaciones: aritméticas y de textura. Las operaciones aritméticas son usadas para generar o modificar los valores de color o bien las coordenadas de textura. Las operaciones de textura sirven para traer las coordenadas de textura u obtener muestras de ellas

[75] *RGBA*: Formato de representación para color de 32-bit. Cada componente (roja, verde, azul y el canal alfa) está compuesta por 8 bit.

usando algunas de las coordenadas. Sin importar el tipo de operación, la salida de un *píxel shader* es siempre un valor en formato *RGBA*.

2.3.2.1 Arquitectura de un *Píxel Shader*

Como se mencionó anteriormente, un *píxel shader* toma las texturas, coordenadas de textura y color como entrada y entrega un valor en formato *RGBA* como salida. Desde el interior de una de estas unidades se pueden buscar coordenadas de textura, modificarlas, mezclarlas, etc. Contiene dos registros de color por entrada, algunos registros constantes y temporales. Esto puede ser observado en la Figura 14.



Arquitectura de un píxel shader
Figura 14

En la mitad derecha del diagrama, la ALU del píxel shader se encuentra rodeada de cuatro tipos de registros. Los registros de color controlan el flujo de datos concerniente al color de los vértices provenientes. Los registros constantes sirven para almacenar constantes que serán empleadas en el programa escrito. Los registros temporales se encargan de almacenar cálculos intermedios. Es importante notar que el registro r0 siempre debe especificarse como salida al terminar el *shader*.

Las coordenadas de textura pueden ser suministradas como parte del formato del vértice o pueden ser cargadas a partir del mapa de textura y pueden usarse hasta 4 (especificación ps.1.1 – ps.1.3) o bien hasta 6 texturas (especificación ps.1.4). Las etapas de textura mantienen la referencia a los datos de textura que pueden ser de una dimensión (como el caso del *Cell-Shading*[76]), bidimensionales o tridimensionales (texturas de volumen o *cube maps*[77]). Cada dato de textura es llamado *texel*[78], los cuales generalmente almacenan valores de color aunque pueden contener cualquier tipo de dato incluyendo vectores normales para *bump mapping*, sombras entre otros. La Tabla 2 muestra información relevante acerca de los registros mencionados.

---

[76] *Cell-Shading*: Técnica de *render* que mediante la cual los objetos tienen apariencia de *comic*. Es decir, en lugar de presentar un sombreado degradado en su superficie (tal y como en la vida real), el objeto es delineado y sombreado con base en tonalidades, claramente al estilo de un dibujo animado.

[77] *Cube-mapping*: Técnica común de *environment mapping* para objetos reflejantes en donde es calculado un vector de reflexión (ya sea por píxel o por vértice) y posteriormente usado para apuntar dentro de un cubo que tiene mapeada la imagen del mundo.

[78] *Texel*: Abreviatura de Elementos de Textura (*TEXture ELements*) que por lo general se refiere a una componente de textura en formato RGB o ARGB.

Registro	Nombre	ps.1.1	ps.1.2	ps.1.3	ps.1.4	Atributos
Constante	c	8	8	8	8	Lectura
Textura	t	4	4	4	6	Lectura/Escritura. En ps.1.4 solo lectura
Temporales	r	2	2	2	6	Lectura/Escritura
Color	v	2	2	2	2	Lectura

Registros de un píxel shader
Tabla 2

Los registros constantes (c0-c7) contienen 4 canales de flotantes (de igual manera que un *vertex shader*) y son únicamente de lectura por lo que son usados como fuente de datos estáticos. La aplicación principal puede hacer lecturas y escrituras a estos registros antes de que el programa de *píxel shade*r sea ejecutado. El rango de valores se encuentra entre [−1, 1] y el valor es truncado cuando no pertenece al intervalo. Además, para las especificaciones ps.1.1 - ps.1.3 las instrucciones de textura no pueden hacer uso de estos.

Los registros temporales son usados para almacenar resultados de cálculos intermedios. El registro r0, es de salida y es el argumento destino para la instrucción del programa. Al igual que los registros constantes, las instrucciones de textura no pueden hacer uso de ellos. En el caso de los registros de color, estos contienen valores de color por vértice en el rango [0, 1] y por lo regular se almacenan las componentes especular[79] y difusa[80] del material para después operar con ellas.

Resumen.

Sin lugar a dudas, las *APIs* de *DirectX* constituyen una ayuda importante para el desarrollador de aplicaciones ya que cubren unas serie de puntos complejas entre el hardware dedicado y las tareas que debería efectuar el programador. Para el desarrollo de esta tesis, se seleccionó *Direct3D* como motor gráfico de aplicación, por el hecho de ser la *API* con mayor documentación y herramientas estables para la elaboración de *shaders*, capacidad requerida desde el momento de definición de la aplicación. *DirectShow* fue elegido como herramienta de procesamiento y adquisición de video a causa de ser perfectamente integrable con la parte gráfica.

Por parte de *Direct3D*, se sabe que es una *API* dedicada a gráficos en 3D con diversas características importantes como son su *HAL*, *HEL*, *Reference Rasterizer* y *PSD* (*Pluggable Software Drivers*). Cada una de ellas creada para satisfacer diferentesa necesidades como acceder a todas las características del hardware disponible (*driver* del fabricante), ofrecer una emulación de ciertas características de hardware (provistas por Microsoft), permitir al desarrollador explotar todas las características de la API en software (aunque se perjudique el rendimiento) y dotar al programador del poder para crear manejadores virtuales que le permitan acceder a características no presentes. Todo lo anterior en orden respectivo con las características antes presentadas.

Esta *API* trabaja mediante clases *COM*, que no son mas que piezas de software que ofrecen servicios a través de interfaces. Por lo regular se trata de bibliotecas de enlace dinámico, *DLL*. Programar mediante interfaces tiene algunas ventajas como son su nulo cambio, la independencia del lenguaje así como el encapsulamiento que presentan, por lo que solo se puede acceder a los métodos pero no a los datos de la clase.

---

[79] Componente especular: Esta componente establece la capacidad que tiene el material para reflejar luz.
[80] Componente difusa: Indica el color que posee el material.

Respecto del pipeline, este ha sido creado de manera que se pueda elegir el modo de procesamiento: fijo (mediante las unidades de *T&L* y *Multitextura*) o programable (usando *vertex* y *píxel shaders*). El caso de los shaders es puramente de poder en el procesamiento, control y despliegue de gráficas. En un inicio fueron pensados para eliminar el aspecto de plástico mate característico del periodo de inicio. A su vez, ofrecen libertad a los artistas y programadores para crear nuevos y diferentes estilos a los ya conocidos. De esta manera, el uso de *shaders* permite efectuar transformaciones básicas, uso de diversas técnicas de animación, generar colores y coordenadas de textura para el fin deseado, iluminar mediante diferentes modelos e incluso dar efectos muy reales como son los llamados *environment* y *bump mapping*.

DirectShow es la *API* que permitió manejar video y mezclarlo con gráficas en 3D. Entre sus características destacan la capacidad para procesar y reproducir una amplia gama de formatos y de *codecs* disponibles. El esquema de trabajo presentado es hasta cierto punto simple, ya que se basa en conexiones de filtros de una manera como se acostumbra hacerlo en el área de procesamiento de señales e, incluso, cuando debe de aterrizarse a un dispositivo físico. El encargado de llevar el control de los filtros y de la especificación de los mismos es el llamado *FilterGraph*.

Por último, ambas *APIs* elegidas ofrecen el poder necesario para resolver la tarea planteada de construir una aplicación que permita visualizar objetos tridimensionales a partir de la Realidad Aumentada y con el uso de *shaders* programables en hardware. La parte de evaluación involucrada en la Realidad Aumentada será concretizada mediante *DirectShow* para la parte de captura y procesamiento de video (además del uso de una serie de bibliotecas de programación conocidas por *ARToolKit* y que en su momento se explicarán) mientras que *Direct3D* se usará para efectuar todo el despliegue de información gráfica. En los siguientes capítulos se tratarán ambos aspectos.

Capítulo III
Editor de *MagicBooks*

Cuando se emplea el término editor dentro de un contexto informático, se debe de entender por aquella aplicación que permite desarrollar contenido para un fin en específico. Estos fines pueden ir desde la creación de textos generales, de carácter científico e incluso del entretenimiento (editores de niveles para juegos de video).

El presente capítulo abordará la necesidad de un editor para el *MagicBook*, así como algunas de las características que éste presenta, la manera en que fueron creados algunos módulos y su correspondiente funcionamiento.

3.1 ¿Por qué un editor de contenido para el *MagicBook*?

Como se mencionó anteriormente, existe una amplia variedad de editores, cada uno con diversos fines. El motivo de crear un editor de contenido en este trabajo, nace ante la necesidad de hacer un libro 'mágico' con mayor rapidez.

En la versión del *MagicBook* para *OpenGL*, la carga del libro es llevada a cabo mediante un archivo de configuración. Este archivo especifica cuales son los modelos a mostrar, así como el patrón asociado y algunos otros parámetros referentes al tamaño del patrón y transformaciones a efectuarse sobre el modelo. Tal creación del archivo involucra, por lo tanto, conocimientos sobre transformaciones geométricas (rotaciones, traslaciones y escalamientos) haciendo un poco más complicado el uso correcto del *MagicBook* ya que los objetos pueden dibujarse de manera incorrecta y el usuario no conocer el porque.

A su vez, en ocasiones era necesario editar directamente el código para hacer que los objetos se orienten de acuerdo a las necesidades del usuario, pues las transformaciones especificadas en el archivo de configuración, no eran suficientes para lograr el efecto deseado.

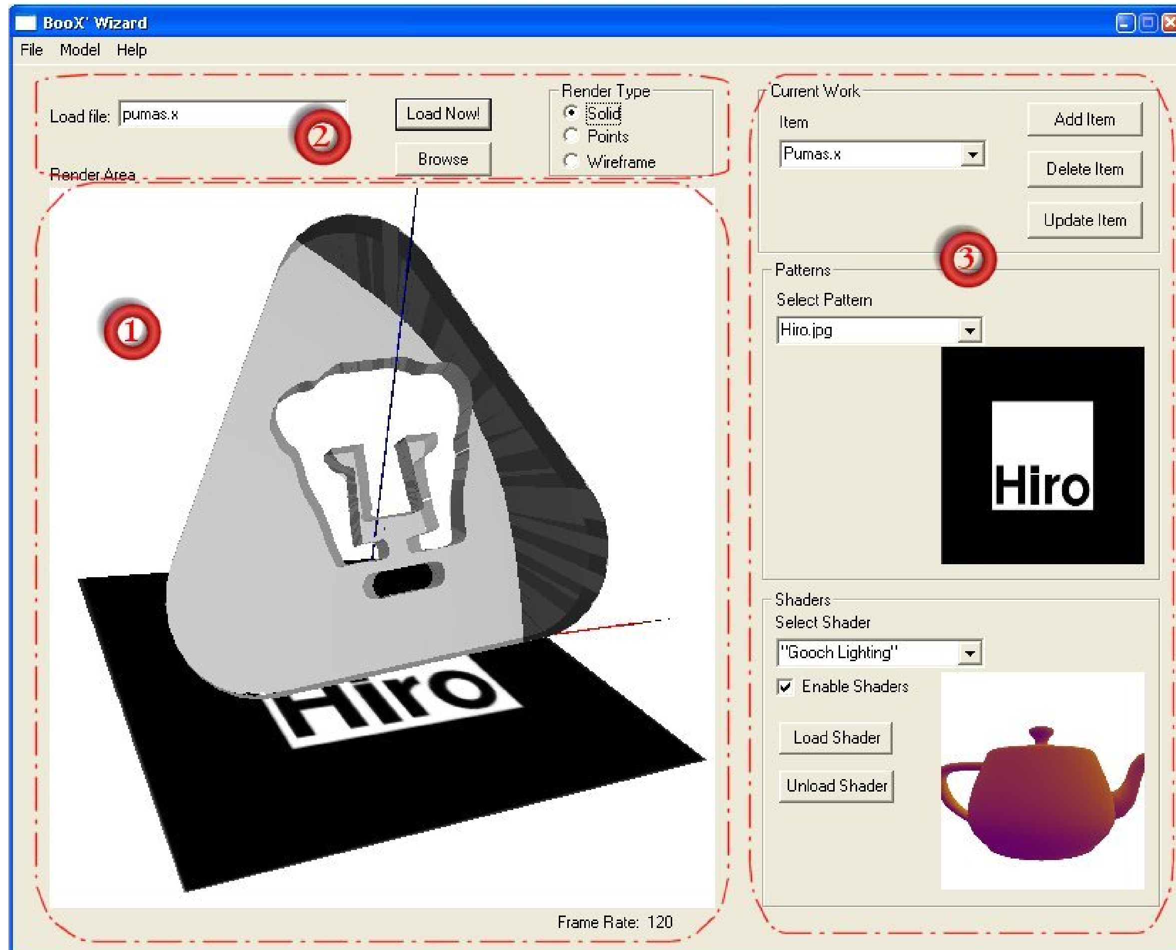
Al realizar un editor de contenido, el trabajo que representa hacer un libro se lleva a un punto mas simple ya que el usuario observa el modelo que va a dibujar, con escala real respecto de la que aparecería al usar el visualizador y usando tanto el patrón elegido, como el efecto elegido. Aún más importante, el usuario no debe poseer muchos conocimientos de computación para poder hacer un libro. El punto referente a los efectos es la mayor innovación que presenta la versión actual y es porque permite visualizar los objetos con otra apariencia.

Si bien el editor ofrece al usuario una representación grafica y en tiempo real de lo que será visualizado en el *MagicBook*, este no serviría sino es capaz de generar un archivo de configuración para el mismo. Es por esto que permite guardar diferentes tipos de archivos, uno de ellos es el mencionada con anterioridad y otro es el que permite crear trabajos preliminares o pendientes antes de elaborar lo que sería el libro final.

A continuación se comentarán algunas de las características que el editor presenta y posteriormente se explicará el funcionamiento de las principales piezas (módulos de software) que le dan vida.

3.2 Características del editor: *BooX' Wizard*

El editor es una interfaz 3D con la que el usuario puede cargar modelos, verlos y operar sobre ellos en un ambiente tridimensional, aplicar efectos y visualizarlos así como guardar archivos de configuración para el *MagicBook* y trabajos inconclusos desarrollados en el editor. En entorno se presenta en la Figura 15.



Interfaz del editor: *Boox' Wizard*
Figura 15

El contenido del editor puede ser dividido en 3 partes principales. La primera es el área principal de despliegue. Aquí es donde se presenta el modelo cargado, el efecto seleccionado y aplicado, el patrón asociado y un sistema de ejes coordenados que sirven como referencia al usuario para colocar los modelos en la posición en que desea que sean vistos al hacer uso del *MagicBook*.

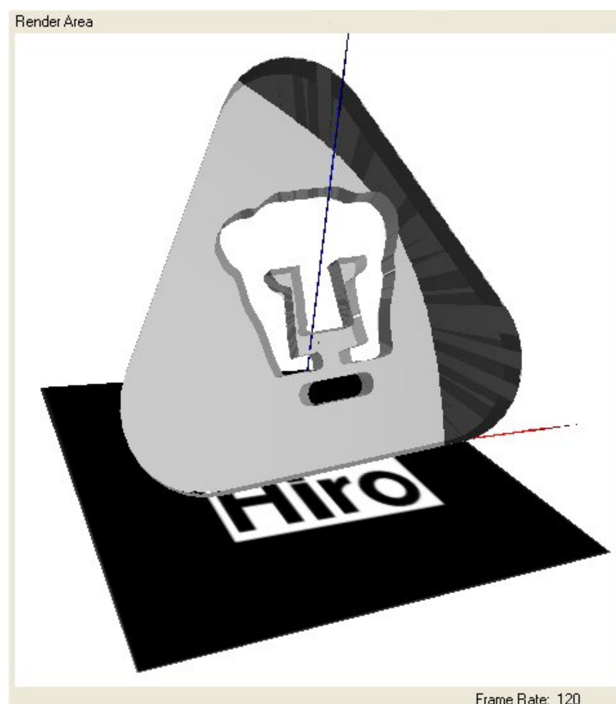
La segunda parte tiene que ver con los controles relacionados a la carga de modelos y el despliegue de los mismos, si es en *wireframe*[81], sólida o representación por puntos. Finalmente, la tercer parte se encuentra formada por aquellos controles que permiten al usuario modificar el trabajo actual, el patrón elegido y el *shader* en uso. En los siguientes puntos se cubrirán la funcionalidad y una breve descripción de cada elemento presente en la interfaz de usuario.

[81] *Wireframe*: Representación por modelo de alambres. Todos los vértices por los que se encuentra formado el modelos son unidos con líneas que le dan una apariencia de estar formado con alambre entretejido.

3.2.1 Área principal de despliegue

Esta área es en la que se dibuja el modelo cargado, el sistema de referencia del mundo 3D y el patrón asociado.

En esta área (Figura 16), se llevará a cabo el dibujo de los modelos cargados, así como de un sistema de referencia y el patrón asociado, ubicados en el origen del sistema coordenado. Sobre esta misma ventana se cuenta con la posibilidad de efectuar diversas transformaciones (traslaciones, rotaciones y escalamientos) sobre la cámara[82] y/o el modelo cargado. Por omisión el modelo es mostrado en la manera en que fue creado en el software de diseño.



Ventana principal de dibujo
Figura 16

3.2.2 Menú

Mediante el menú de aplicación (ver Figura 17), el usuario tiene control sobre la mayoría de las acciones contenidas en la interfaz gráfica. En el caso particular del *BooX' Wizard*, permite guardar y cargar trabajos, seleccionar el color de fondo del área de despliegue, salir de la aplicación, cargar modelos, seleccionar el tipo de dibujo para ellos, etc.

Menú
Figura 17

[82] Cámara: Para poder crear una escena tridimensional, es necesario especificar una cámara virtual que sea la que determine el volumen visto. Esto es, las *APIs* de programación para gráficos como *OpenGL* y *Direct3D*, permiten al programador definir una cámara y un volumen de recorte (ver *Frustum Clipping* en el glosario) con base en parámetros propios de una cámara real, aunados a algunos propios de las virtuales. Véase el glosario para mayor información al respecto.

Existen dos opciones para guardar archivos. Éstas son las conocidas como trabajo (*Work*) o bien libro (*Book*), ambas accesibles a partir del menú *File* y eligiendo *Save* del submenú emergente.

La primera permite crear un archivo con la información requerida para cargar los modelos, patrones y efectos usados hasta ese momento en una ocasión posterior. La elección de *Book*, conduce a la creación de un archivo de configuración para el *MagicBook*, en donde se encuentran los datos necesarios para desplegar el trabajo realizado asociado con los patrones empleados en el mismo.

Mediante el menú *Model*, el usuario es capaz de cargar modelos dentro del área principal de despliegue (acción *Load*), elegir el estilo con el que quiere que sea visto (*Solid*, *Wireframe* o *Points*). El estado de los *shaders* también puede ser controlado mediante este menú en la opción *Shaders*. Dicho estado puede ser cambiado usando las opciones *Enable* para habilitarlos o *Disable* para deshabilitarlos.

Finalmente se tiene al menú *Help*, en donde el usuario podrá consultar una ayuda rápida para el uso de la aplicación así como información referente a la misma (*About*).

3.2.3 Controles para la carga de modelos

La carga de modelos puede ser llevada a cabo mediante el campo *"Load file"*, el botón *"Browse"* o bien, como ya se mencionó, usando el menú *"Model"* en la opción *"Load"*. En caso de usar el campo de texto *"Load file"*, se debe de incluir la extensión del nombre del archivo, puesto que es lo que hace la diferencia entre los dos tipos de formato soportados, archivos de *Cal3D*[83] y en formato *X*[84]. La Figura 18 ilustra el control asociado a la acción descrita.

Carga de modelos
Figura 18

3.2.4 Selección del tipo de *render*

Cómo ya se mencionó en el punto 3.2.2, es posible dibujar los modelos de diferentes maneras. Los botones mostrados en la Figura 19, permiten cambiar la forma en que el modelo es visto.

Tipo de *render*
Figura 19

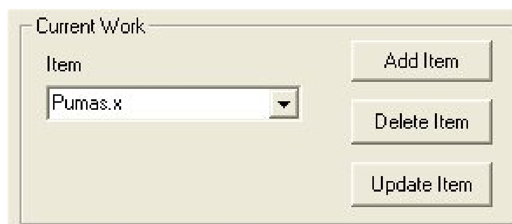
Esto es, si se oprime el botón *Wireframe*, el modelo será mostrado en modo de alambres. Al seleccionar *Points*, solamente se dibujan los puntos del modelo. Finalmente, la opción *Solid* dibuja al modelo tal y como se ha diseñado, usando las diversas texturas y materiales que han sido empleados durante su construcción.

---

[83] Formato *Cal3D*: Formato de archivo que permite tener modelos animados basados en animación por esqueleto.
[84] Formato X: Formato de archivo para modelos 3D definido por Microsoft y propio de la *API* de *Direct3D*.

3.2.5 Trabajo actual

Los botones y controles mostrados en la Figura 20, muestran y controlan el trabajo que se ha desarrollado, o bien, alguno que se haya efectuado en otra ocasión y que ha sido cargado para continuar con él.



Controles del trabajo actual
Figura 20

El área se compone por tres botones y un *combo box*[85]. Una vez que se ha cargado un modelo, se ha asignado un patrón y ha sido o no usado un *shader*, el usuario puede guardar ese elemento pulsando el botón *Add Item* y continuar creando lo que se podría llamar otra "*página mágica*", dado que ese modelo y patrón serán usados posteriormente en el visualizador *MagicBook*.

Los otros botones, *Delete Item* y *Update Item*, permiten al usuario borrar alguna *página* creada o bien, actualizarla. El decir actualizarla, se refiere a poder cambiar alguna de las propiedades; es decir, el modelo, el patrón el uso de *shaders* y/o cual sería esté.

3.2.6 Selección de patrones

El área 6 de la interfaz (Figura 21) contiene únicamente un *combo box* y un área de despliegue del patrón que se está usando.

Selección de patrones
Figura 21

En el *combo box* se muestran todos los patrones que pueden ser usados por *default*. Cuando se hace un archivo de configuración para el *MagicBook* o, dicho de otra forma, un libro, este patrón será el asociado al momento de usar la aplicación.

[85] *Combo box*: Control conformado por una lista desplegable al momento de pulsar sobre ella.

3.2.7 Selección de *shader*

En la figura 22 se observan los controles que el usuario puede emplear para seleccionar un *shader*, así como habilitarlo y ver una presentación preliminar del mismo.

Selección de *shader*
Figura 22

En el *Combo box* se despliegan los diversos *shaders* con que se cuenta y que son elegibles. Cuando alguno es seleccionado, éste es cargado y su efecto mostrado. Al seleccionar el botón *Enable Shaders*, el efecto es aplicado sobre el modelo que el usuario ha cargado y visualizado en la ventana principal de *render*. Si se vuelve a oprimir el botón *Enable Shaders*, el efecto desaparecerá de la ventana principal.

3.3 Funcionamiento del editor

En la sección 3.2 se han explicado las características y los elementos que componen al editor, pero, ¿de qué manera se llevan a cabo las tareas comentadas? Precisamente es en esta sección donde se abordará la respuesta a esta pregunta, por lo que se comentarán algunos de los aspectos técnicos en la realización de los módulos mas importantes de la aplicación.

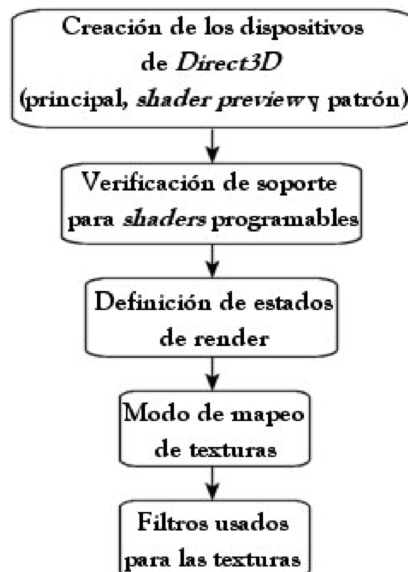
Las primeras etapas a tratar, son inicialización de variables e interfaces de *Direct3D* y la inicialización del entorno. La parte de inicialización de *Direct3D* tiene que ver con la definición y creación del número de dispositivos que se crearán, las ventanas a las que se encontrarán asociadas, pruebas en 3D (como son la del *buffer Z*, *stencil*, canal alfa, etc) y estados asociados con el manejo del dispositivo (iluminación, modo de sombreado, *culling*[86], etc). Con el término entorno se pretende incluir a todas aquellas variables y estructuras de datos necesarias para el funcionamiento de la aplicación.

En los siguientes puntos se trata con mayor detalle a cada una de las etapas para posteriormente analizar la manera en que es realizado el *render* de la aplicación, que es la parte fuerte y más importante del editor.

[86] *Culling*: Es el proceso de eliminar la cara frontal o posterior de un polígono de manera que no sean dibujadas.

3.3.1 Inicialización de *Direct3D*

La Figura 23 ilustra los procesos llevados a cabo en esta etapa. La primer parte consta de la creación de cada uno de los dispositivos empleados por la aplicación. Estos son los correspondientes a la presentación preliminar del *shader*, del patrón actual en uso y del área principal de despliegue.



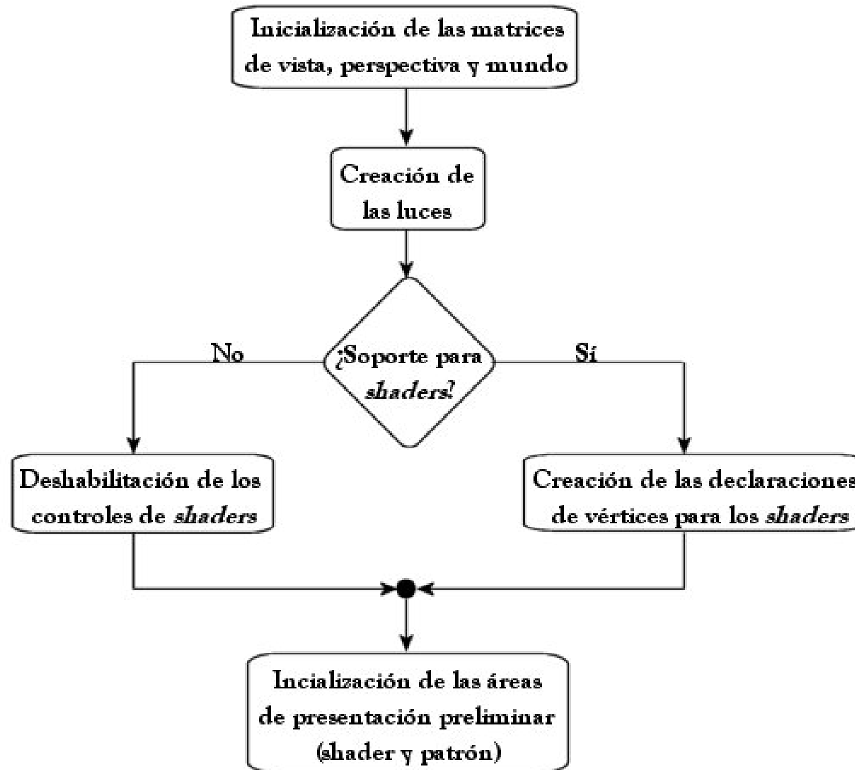
Procesos involucrados con la inicialización de *Direct3D*
Figura 23

La segunda parte tiene como objeto saber si el hardware sobre el cual se está ejecutando la aplicación permite el uso de *shaders* programables. En caso contrario, la aplicación desactiva los controles concernientes al uso de efectos para evitar la presencia de un error de tipo fatal para la aplicación. Los tres últimos estados, tienen que ver con los estados de *render* y texturizado del dispositivo 3D, esto es, la elección de algunas pruebas generales como son la del *BufferZ*, el mapeo de textura (*wrap*[87], espejo, *clamp*[88], etc) y los filtro para las texturas que pueden ser lineares, bilineares o anisotrópicos[89].

3.3.2 Inicialización del entorno

Una vez que se ha concluido con éxito la parte relacionada con la inicialización de *Direct3D*, toma lugar la inicialización del resto de las variables globales y de algunos estados asociados con los dispositivos creados. En la Figura 24 puede observarse el flujo de las acciones efectuadas.

[87] *Wrap*: Modo de mapeo de textura en donde esta es dibujada tantas veces como quepa en los vértices que se índico se mapeara.
[88] *Clamp*: Modo de mapeo de textura en donde esta es estirada hasta abarcar el área que debe cubrir.
[89] Filtro anisotrópico: Las técnicas usuales de filtrado no compensan la anisotropía, efecto de elongación de los píxeles cuando son mapeados a espacio de textura. El resultado es ver borrosa la textura o con *aliasing*, dependiendo del nivel de detalle y conformación de la textura. Para observar texturas mas claras y nítidas se emplea este tipo de filtrado, proceso que involucra un núcleo elíptico cuya forma y orientación depende de la proyección del píxel en la textura.



Pasos efectuados al inicializar el entorno
Figura 24

Primero se inicializan las matrices que permiten crear y observar al mundo tridimensional, dichas matrices son las de mundo, perspectiva y vista. La matriz de mundo es la que lleva control sobre la escena que se dibuja y es sobre la que se opera para posicionar al modelo de la manera deseada. La matriz de perspectiva define lo que es el volumen de vista, esto es (véase Figura 7) el volumen definido por la pirámide truncada creada a partir de un ángulo de apertura y dos planos de corte. La matriz de vista define el sistema de referencia para la cámara, con base en una terna de vectores, *Up*, *At* y *Eye* que definen hacia donde se encuentra 'arriba', la dirección a la que se mira y el punto de observación, respectivamente.

Posteriormente se crean las luces necesarias, cada una con sus diversas características de tipo, intensidad, color, etc.

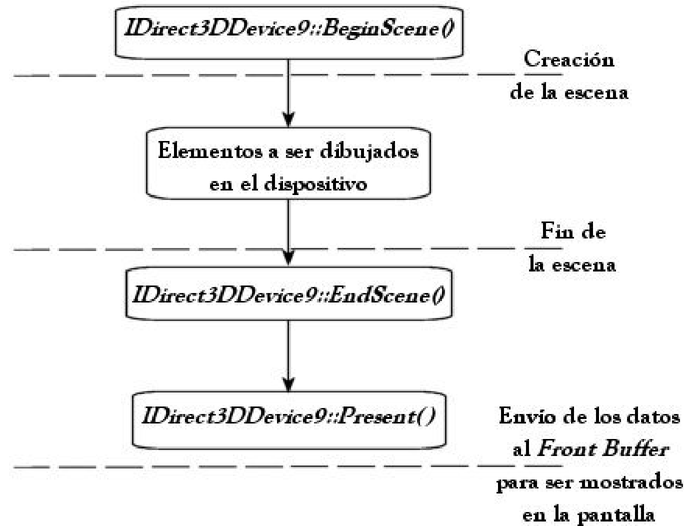
Si el hardware en el que se ejecuta la aplicación soporta *shaders* en hardware[90], entonces son creadas las declaraciones para los mismos. Al decir 'declaraciones', considérese las propiedades de los vértices con las que operaran los efectos, esto es si emplean coordenadas transformadas, sin transformar, coordenadas de textura (*UV* o *UVW*), normales, componentes difusa y/o especular del material, etc. En caso contrario, entonces se deben de desactivar los controles asociados al uso de efectos.

Finalmente toca el turno a la inicialización de las áreas de presentación preliminar. Para el caso del *shader*, se crea el modelo de presentación preliminar así como los recursos que el *shader* actual emplee. En el caso del área de patrones, se lee del directorio por *default* todas las texturas que se encuentren en el mismo y se colocan en la lista desplegable del *combo box*. Asimismo, se despliega un plano (creado en esta misma parte) con la primer textura encontrada en el directorio.

[90] Nota: En el capítulo 5 serán tratados con mayor detalle los *shaders* programables.

3.3.3 Procedimiento de *render*

El proceso de dibujo de la aplicación es lo más importante en ella. Gracias a estas rutinas es posible observar los objetos en la manera que se han enviado a dibujar. Para dibujar una escena en *D3D* (Figura 25) se hace una llamada al método *BeginScene* para crear a la misma, mediante *EndScene* se finaliza y es pasada al *FrontBuffer* usando el método *Present* (todos pertenecientes a la interfaz *IDirect3DDevice9*).



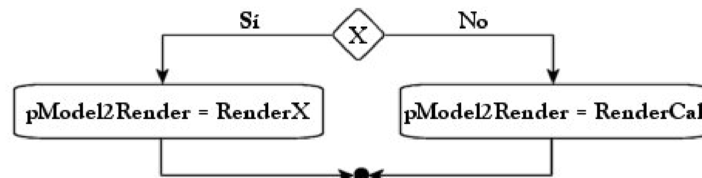
Especificación de dibujo de una escena en *Direct3D*
Figura 25

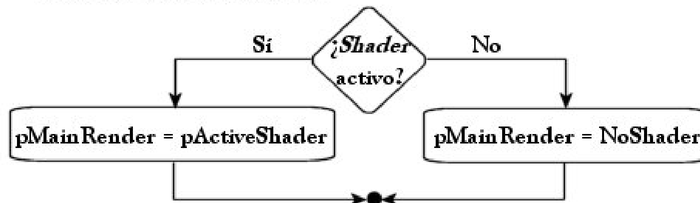
Dentro de los dos primeros métodos se especifican los estados y parámetros de *render*, iluminación, transformaciones, texturas empleadas, etc. Esto se hace de acuerdo a la escena que se esté creando, es decir, acorde a cada una de las geometrías que se empleen. Ahora bien, en el editor es posible cargar dos diferentes tipos de modelos (*Cal3D* y *X*) y cada uno de ellos se dibuja de diferente manera debido a las propiedades, como el tipo de vértices, que cada uno de ellos maneja. Asimismo, el editor soporta el dibujo de modelos por medio de *shaders* programables y cada uno de ellos se especifica de manera diferente. Lo anterior debido a que los efectos son determinados con un cierto número de *shaders* y pasadas[91] de ellos, así como diferentes estados y propiedades de *render* en cada una de las mismas.

Por la situación comentada, se decidió efectuar el *render* de los objetos mediante apuntadores a funciones. Si bien no es la manera más óptima en cuanto a velocidad, ofrece la flexibilidad necesaria en el desarrollo de los procesos de *render* ya que, de otra manera, se habría tenido que definir un solo proceso de dibujo que involucrara todos los casos de los *shaders* que se desearan implementar. Lo anterior involucraría la evaluación de demasiadas condiciones para esta habilitando y deshabilitando opciones, lo cual, a su vez, termina por repercutir en la velocidad con que se desempeña la aplicación.

Un apuntador a función determina el tipo de dibujo: sombreado por defecto o sombreado usando *shaders*. Otro apuntador a función indica el tipo de modelo a dibujar. Es así que se logra tener la flexibilidad requerida para que la aplicación pueda crecer en un futuro. La Figura 26 ilustra la elección de las funciones de dibujo.

[91] Pasada: Entiéndase este término como las veces que necesitan ser aplicadas las transformaciones e iluminación sobre los vértices que componen una geometría antes de ser dibujada.





Elección de las rutinas de *render*
Figura 26

De esta manera es seleccionado el dibujo del modelo. *pModel2Render* y *pMainRender* son los dos apuntadores a función que se mencionó con anterioridad mientras que *pActiveShader* es otro apuntador a función que indica cual es el *shader* que está siendo empleado (el elegido del *combo box* de selección de *shader*). Las funciones principales de dibujo son *RenderX*, *RenderCal*, *NoShader* y aquella a la que apunta *pActiveShader*. Las dos primeras rutinas tienen que ver únicamente con el *render* de los modelos en donde solo se especifica la manera en que serán unidos los vértices (si es que están indexados), por medio de que primitiva se hará el tejido así como algunas propiedades específicas como es el *FVF*[92] y otras pruebas como el *Alpha Blending*. El siguiente pseudocódigo ilustra un esquema para la realización de funciones de *render* mendiante *shaders*:

Limpiar dispositivo de *render*
Calcular matrices de transformación
ModelView = Matriz de Mundo * Matriz de Vista
ModelViewProjection = Matriz de ModelView * Matriz de Proyección
Cargar en los registros del *Vertex Shader* (de acuerdo a la especificación realizada en el mismo) los valores de ambas matrices[93]
Trasponer Matriz ModelView y cargarla en los registros asociados
Trasponer Matriz ModelViewProjection y cargarla en los registros asociados
Cargar en algún(os) registro(s) la(s) luz(ces) involucrada(s)
Iniciar la escena de *Direct3D* mediante el método *BeginScene*

FOR Cada pasada que el efecto necesite
Cargar otras variables necesarias para lograr el efecto (algún color, textura, etc) en los registros del *Vertex Shader*
Cargar las variables necesarias en los registros del *Píxel Shader*
Habilitar y deshabilitar los estados de *render* pruebas y etapas de textura involucrados en el funcionamiento del *shader*
Indicar a *D3D* cual es la declaración de vértices correspondiente
Indicar a *D3D* cual es el *Vertex Shader* activo
Indicar a *D3D* cual es el *Píxel shader* activo
Dibujar de acuerdo al apuntador a función de *render* de modelo

---

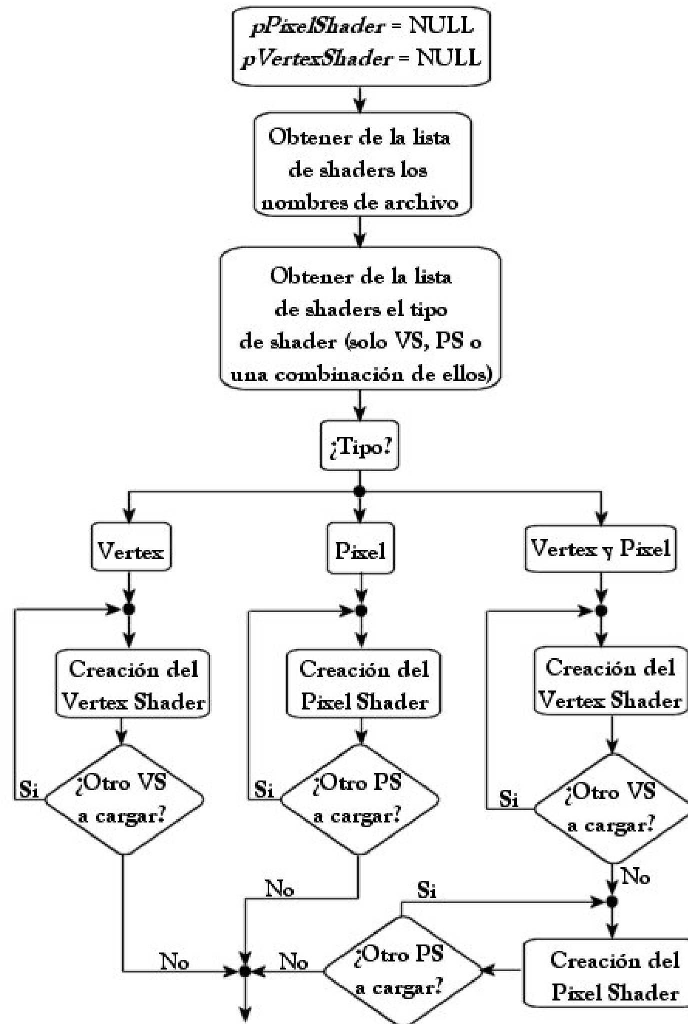
[92] *FVF*: Flexible Vertex Format. Una variable de tipo *double word* (*DWORD*) en la que se dice por cuales componentes se encuentra formado un vértice. Es decir, si el vértice está especificado en coordenadas XYZ sin transformar, ya transformadas, si cuenta con coordenadas de textura (bidimensionales o tridimensionales), la normal, componentes difusa, especular, etc.

[93] Nota: Dado que las unidades de *Píxel* y *Vertex Shader* son de tipo vectoriales, es necesario introducir los datos de las matrices de manera traspuesta, ya que las operaciones efectuadas son post-multiplicación.

Deshabilitar la declaración de vértices
Deshabilitar *Vertex Shaders*
Deshabilitar *Píxel Shaders*

Terminar el dibujo de la escena con el método *EndScene*
Mandar al *FrontBuffer* los vértices transformados, iluminados, con color y/o texturas usando el método *Present*

Ahora bien, cuando se elige un *shader* este es mostrado en su área de presentación preliminar. El proceso interno que lleva desde la elección hasta el despliegue es el ilustrado en la Figura 27.



Procesos llevados a cabo al seleccionar algún *shader*
Figura 27

Como se puede observar, lo primero en realizarse es hacer que no exista basura en los apuntadores a interfaces de *Píxel* y *Vertex Shader*, esto con el fin de que no se direccione erróneamente a una localidad en la memoria. Enseguida, se obtienen de la lista de *shaders* los nombres de los archivos que componen al efecto así como el tipo. Por tipo se ha denominado a la manera en que vienen compuestos los efectos, si se basan únicamente en *vertex shaders*, *píxel shaders* o una combinación de ambos, que es lo más natural.

Una vez que se conoce el tipo de efecto, entonces se hace la creación de los recursos. Esto es, por cada archivo de *shader* (ya sea *píxel* o *vertex*) ensamblado previamente, se crea un manejador de *Vertex* o *Píxel Shader*, según sea el caso. Esto conduce a que está función, cuyo nombre es *SetupShaders*, trabaje con apuntadores a la Interfaz *IDirect3DVertexShader9* y *IDirect3DPixelShader9*. Al final del proceso, si ningún error ocurrió, el apuntador a función de *shader*: *pActiveShader*, apunta a la función especificada por la selección hecha en el combo box y con los recursos necesarios para dibujar generados.

Entendido ya el proceso de *render* y comentada la forma en que son seleccionados los *shaders*, es turno de explicar la manera en que las transformaciones sobre el modelo y la cámara son llevadas a cabo.

3.3.4 Interfaz de control de transformaciones mediante el ratón

Si bien el editor *BooX' Wizard* permite visualizar modelos 3D, escoger un patrón y un efecto asociado al modelo, ¿de qué manera es posible ver al objeto desde otro punto de vista? Para esto, es necesario introducir una interfaz que permita controlar la ubicación del modelo de una manera sencilla y eficaz.

Dentro de los periféricos con los que se cuenta en una PC, el ratón es de los más usados gracias a su versatilidad para resolver tareas. Sirve para apuntar, marcar, seleccionar, abrir, etc. Diversas funcionalidades que han surgido a partir de un problema que resolver y enfocados a la aplicación que planteo el problema. De esta manera se tiene que para un editor de textos, el ratón permite seleccionar texto, abrir algún menú, seleccionar de entre las herramientas gráficas a manera de íconos, etc.

En el caso particular de una interfaz 3D, el ratón es de gran utilidad. Con sus botones es posible realizar las transformaciones necesarias (traslaciones, rotaciones y escalamientos) de una manera sencilla. Además, el hecho de poder arrastrar el ratón en una cierta dirección, repercute en que los movimientos sean mas intuitivos, casi tal y como si el objeto fuese movido por la mano.

De esta manera es que se ha decidido emplear el ratón como interfaz y se ha desarrollado una clase que controla las transformaciones llamada *CD3DArcBall*. Dicha clase se encarga de las rotaciones, traslaciones y escalamientos con base en los movimientos del ratón, el estado de los botones del mismo y la rueda (en caso de contar con ella).

3.3.4.1 Esquema general de la clase CD3DArcBall

Esta clase está compuesta por un conjunto de matrices, cuaterniones y apuntadores a ambos. Los cuaterniones son empleados para llevar a cabo las rotaciones pero generando al final una matriz de rotación, mientras que las matrices llevan control de las traslaciones y escalamiento. Se emplean a su vez apuntadores a ellos por dos motivos principales.

1. Posibilidad de actuar sobre el objeto directamente o bien de actuar sobre la cámara. Esto es, se pueden efectuar transformaciones de manera independiente sobre ambas matrices ofreciendo la posibilidad de observar la escena desde diferentes puntos de vista y al modelo en diversas posiciones con respecto del sistema de referencia.

2. Cada que se guarda una '*página mágica*' y esta es vuelta a cargar, el editor debe ser capaz de dibujar el modelo en la posición y vista tal y como fue guardado. Por tal motivo, basta con que la aplicación pase las localidades de memoria de las matrices para que se opere directamente sobre de ellas. La ganancia: flexibilidad y simplicidad para el programador, al volver a cargar un elemento que ha sido guardado.

De esta manera, los apuntadores ofrecen la flexibilidad necesaria para llevar a cabo las dos tareas planteadas de una manera eficaz ya que permiten cambiar entre las matrices que llevan el estado del mundo y de la cámara y operar únicamente con las direcciones de memoria. Lo anterior permite que se ejecute, además, con mayor rapidez.

A continuación se tratará de manera independiente el funcionamiento interno de cada operación comenzando por el escalamiento.

3.3.4.2 Escalamiento

Para llevar a cabo el escalamiento, la clase *CD3DArcBall* involucra dos matrices, una que es la que se entrega a la aplicación principal para efectuar la serie de transformaciones finales y otra que lleva el estado interno, es decir los incrementos que se han presentado desde el momento en que se comenzó a usar.

Esta transformación está asociada con la rueda del ratón (*mouse wheel*) ya que se ofrece una forma cómoda y rápida para efectuarlas. Un inconveniente es la precisión que puede obtenerse. Sin embargo, esto no es una restricción ya que el editor fue pensado para permitir que el usuario cree con rapidez el material necesario para mostrar su libro mágico, dentro de estos lineamientos no es una prioridad la precisión.

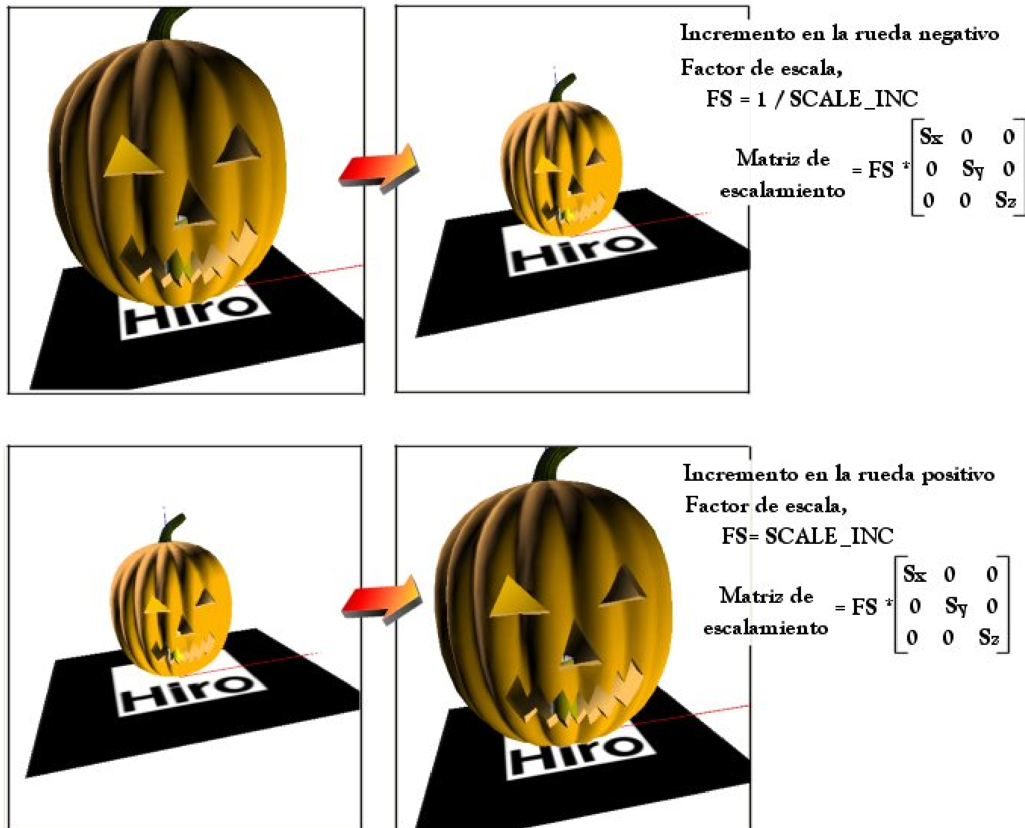
La rueda del ratón es un dispositivo incremental, esto es que basa su funcionamiento en los deltas[94] que se generan al rodarla hacia delante o hacia atrás. Cuando se efectúa un movimiento hacia delante, el incremento es positivo. En el caso contrario es negativo.

Con estos datos es posible generar el escalamiento uniforme[95] mediante un factor de escala previamente definido y operándolo como una multiplicación por él (en el caso de aumentar) o por su recíproco (para disminuir). Véase la Figura 28 para una mejor comprensión.

---

[94] Delta: Diferencia entre un estado final y uno inicial.

[95] Escalamiento uniforme: El modelo o puerto de vista es escalado pro igual en los tres ejes de referencia, dando la sensación de que cambio de tamaño pero no la proporción. El escalamiento puede realizarse de esta manera, o bien sobre cualquiera de los ejes dando una apariencia de deformidad del objeto.



Matrices de escalamiento
Figura 28

3.3.4.3 Rotaciones

Las rotaciones pueden ser implementadas de diversas maneras. Una de las mas usuales es mediante ángulos sobre los ejes de referencia para los cuales ya se tiene las expresiones calculadas (ángulos de Euler). Así, las rotaciones se llevarían a cabo de una manera jerárquica, es decir una por una y consecutivas para lograr el movimiento deseado. Por otro lado, las multiplicaciones entre matrices, en términos de costo computacional, tienen un costo elevado además de resultar en ocasiones lentas. Es por ello que otro método, a menudo empleado, es el de rotar por medio de cuaterniones.

"A pesar de que los cuaterniones fueron inventados en 1843 por Sir William Rowan Hamilton como una extensión a los números complejos, no fue sino hasta 1985 que Ken Shoemake los introdujo al área de los gráficos por computadora en SIGGRAPH. De esta manera, los cuaterniones extienden el concepto de rotación en 3D a 4D y pueden representar cualquier rotación en el espacio alrededor de un eje. Es conveniente emplearlos porque consumen menor espacio que las matrices y algunas operaciones (como las interpolaciones entre ellos) producen un efecto visual más agradable."[96] Un cuaternión[97] puede ser descrito de la siguiente manera:

$$q = \left(\cos\left(\frac{\theta}{2}\right), X_A \sin\left(\frac{\theta}{2}\right), Y_A \sin\left(\frac{\theta}{2}\right), Z_A \sin\left(\frac{\theta}{2}\right) \right)$$

[96] Engel, Wolfgang F., Amir Geva. *Beginning Direct3D Game Programming*. Estados Unidos. Prima Tech. 2001 p.92-95

[97] Véase Apéndice C para una información mas detallada.

donde:

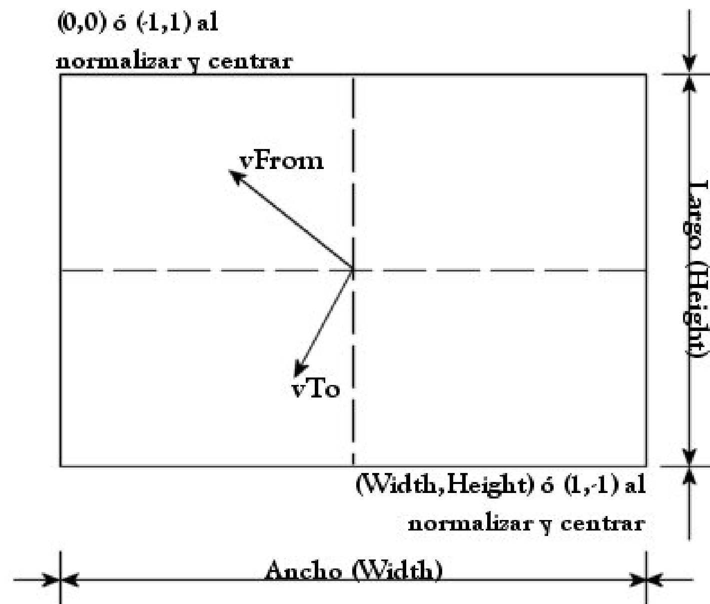
- A es el eje de rotación con componentes $A(X_A, Y_A, Z_A)$
- θ es el ángulo

El procedimiento para calcular el cuaternión asociado a la rotación es el mostrado a continuación. Cuando el botón del ratón es pulsado, son tomados los píxeles (coordenadas *x*, *y*) de la posición actual del puntero. Posteriormente, conforme se arrastra el botón, se van adquiriendo las posiciones por las que pasa el puntero. Con estos datos se calculan dos vectores, *vFrom* y *vTo* que indican las posiciones origen y destino y que son útiles para calcular el cuaternión.

El siguiente paso es ubicar a los vectores dentro de un plano de dimensión 2 situado en el origen, es decir, limitado por los vértices (-1,1), (1,1), (1,-1) y (-1, -1). Esto con el fin de operar sobre una esfera de radio 1, ya que el eje de rotación con el que se crea el cuaternión debe ser unitario. Por lo tanto, se hace uso de las siguientes transformaciones:

$$x = \frac{sx - Width/2}{Width/2} \qquad y = \frac{sy - Height/2}{Height/2}$$

En la Figura 29 se puede observar el plano descrito con anterioridad.



Conversión del puerto de vista al plano de tamaño 2x2 situado en el origen
Figura 29

Una vez que se cuenta con los vectores dentro del espacio mencionado, entonces se lleva acabo la evaluación del cuaternión. Para ello se emplean los vectores vA y vB ($vA = vFrom/\|vFrom\|$, $vB = vTo/\|vTo\|$) y se calcula un vector que los divida por la mitad, vHalf ($vHalf = (vA + vB)/\|vA + vB\|$). Con lo anterior se logra obtener el ángulo de rotación θ y, para los fines con que se operan los cuaterniones, la mitad del ángulo $\theta/2$. Del producto cruz entre *vA* y *vHalf* (*vRAxis*) es obtenido el eje de rotación, perpendicular al plano que forman los mismos y sobre el cual se debe girar.

Por consiguiente, el cuaternión generado sería:

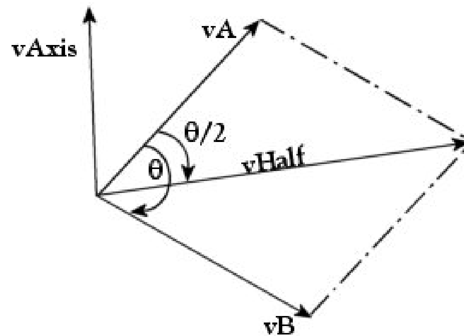
$$qN = (vA \cdot vH, vRAxis.x, vRAxis.y, vRAxis.z)$$

donde las componentes de *vRAxis* corresponden al eje de rotación y se colocan directamente sobre los términos complejos del cuaternión. Lo anterior debido a que el producto cruz entre dos vectores representa el producto del seno del ángulo que se forma entre ellos por la norma de los vectores. De manera similar, el término real del cuaternión es equivalente al producto interno entre los vectores descritos pues dicho producto representa el producto del coseno del ángulo entre ellos por su norma. Como se mencionó anteriormente, ambos vectores (*vA* y *vHalf*) se encuentran normalizados, esto es:

$$\|vA\|\,\|vHalf\|\,sen\theta = \|vA \times vHalf\|, \qquad \|vA\|\,\|vHalf\| = 1 \qquad \therefore \qquad \|vA \times vHalf\| = sen\theta$$

$$\|vA\|\,\|vHalf\|\cos\theta = vA \quad vHalf, \qquad \|vA\|\,\|vHalf\| = 1 \qquad \therefore \qquad vA \cdot vHalf = \cos\theta$$

La Figura 30 ilustra las operaciones realizadas.

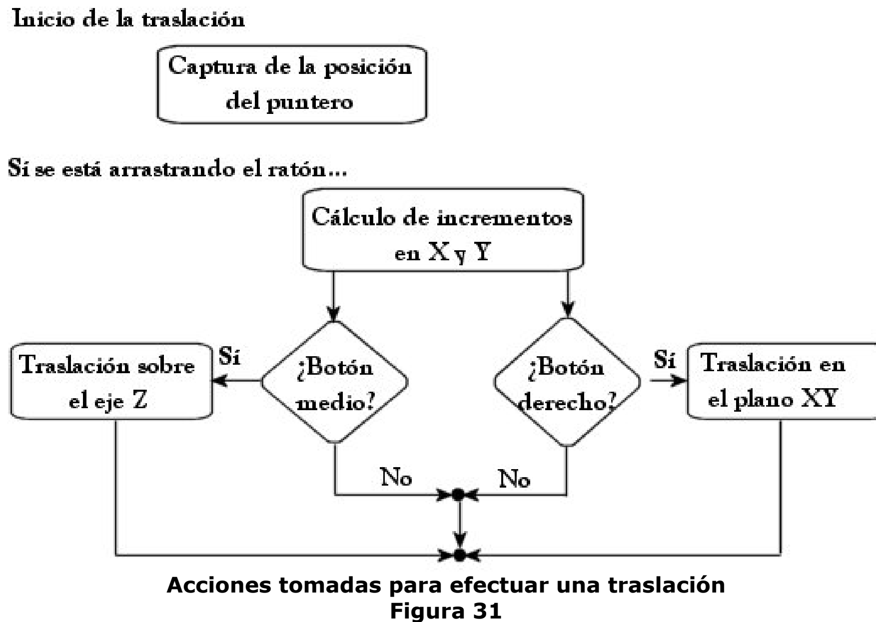


Cálculo del cuaternión
Figura 30

Con el cuaternión que representa la rotación calculado, falta obtener su correspondiente forma matricial, para que pueda ser operado con el resto de las matrices de transformación. En el Apéndice C se trata con mayor detalle el álgebra de cuaterniones, una pequeña justificación del porque representan rotaciones y su equivalente matricial.

3.3.4.4 Traslaciones

Las traslaciones se llevan a cabo presionando ya sea el botón derecho o el medio del ratón y para calcularlas, se obtienen deltas entre la posición en que fue capturado el puntero del ratón cuando se comenzó el movimiento y la posición que ocupó al final del mismo. Visto en un diagrama lo anterior, se tiene que (ver Figura 31):



Acciones tomadas para efectuar una traslación
Figura 31

El cálculo de incrementos no es mas que una diferencia entre la posición actual del puntero y la posición donde se inició el desplazamiento. Dicha diferencia es multiplicada por un factor (que indica cuantas unidades será desplazado) y dividida por el ancho (en el caso del eje X) o por el alto (en el caso del eje Y). Si se va a desplazar sobre el plano XY, entonces se toman ambas diferencias mientras que si se traslada en el eje Z, se considera solamente el delta en Y.

Resumen

Los editores de contenido para las diversas aplicaciones existentes son de importancia para el usuario final. La razón: el usuario final debe de tener una manera sencilla de generar contenido sin necesidad de conocer la estructura interna de un programa. Esto es, para el caso de los editores de texto, al usuario final no le interesa como es que se lleva a cabo el despliegue de una letra sobre la hoja y como es que dicha letra es modificable en tamaño, espesor, etc., que posteriormente pueda guardarse en un archivo e incluso ser impresa. Lo que a el le importa es que sea capaz de escribir sus textos, poder ver lo que se encuentra desarrollando, guardarlo e imprimirlo, no más.

De la misma manera se comporta un editor de niveles de juego. Al artista que está involucrado en el diseño del nivel, no le importa si es que usa *shaders* en tiempo real, si tiene capacidad de filtrado anisotrópico, etc. Lo que le interesa es poder seleccionar efectos que se vean bien, contar con las herramientas necesarias para construir su mundo y, si se ve un poco mal, borroso o con *aliasing*, poder eliminarlos mediante una opción que aparezca por ahí.

Por las razones anteriores se ha creado un editor de libros. De esta manera se pretende que gente ajena al área de la graficación por computadora sea capaza de hacer uso de un software como lo es el *MagicBook* y que pueda crear historias. El editor presentado tiene características que ofrecen la visualización de objetos, el patrón que se requiera, así como de un efecto que puede o no ser empleado por el *MagicBook*.

Asimismo, se han presentado las características que presenta así como el porque de la elección de las mismas. Sin duda, lo más importante que presenta dicho editor es la característica de *shaders* programables en hardware y, hablando a nivel técnico, el hecho de que se empleen apuntadores a funciones como métodos de dibujo. Lo último es flexibilidad para el(los) desarrollador(es) porque no es necesario hacer grandes cambios estructurales para diseñar nuevos módulos destinados a tareas específicas.

Respecto a la manera en que se efectúan las transformaciones, es importante resaltar el uso de cuaterniones, estructuras matemáticas no usadas a menudo y con poder realmente notable. Cabe mencionar simplemente el costo computacional que involucra una multiplicación de matrices respecto una de vectores.

A nivel de características importantes de usuario, se han creado controles para que quien desarrollé con el editor se sienta cómodo. Si bien la interfaz no es muy avanzada e incluso no muy atractiva, es una buena aproximación a un software que pueda considerarse producto final.

Capítulo IV
MagicBook

En el capítulo anterior se explicó el porque de la creación y el funcionamiento del editor *BooX' Wizard*, pero ¿cómo visualizar en un ambiente de realidad aumentada (*RA*) el trabajo realizado? Para poder observar el contenido de uno de los libros creados, existe la aplicación *MagicBook*. Dicha aplicación permite al usuario ver una escena aumentada a partir de una cámara de video-conferencia y un libro previamente realizado.

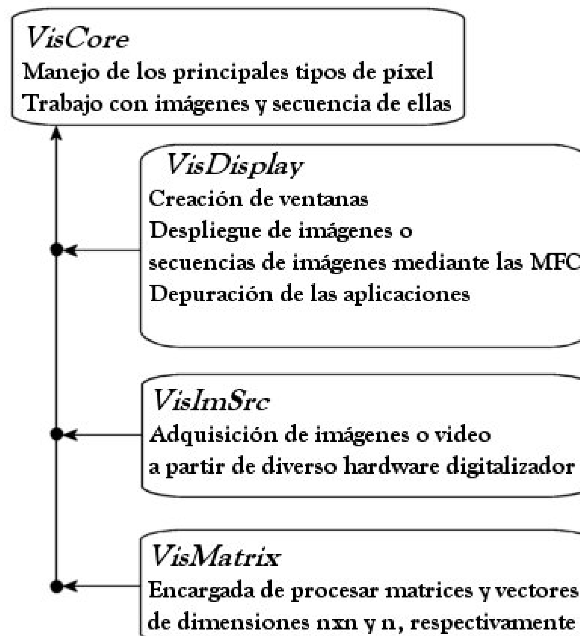
De esta manera el usuario tiene una experiencia de *RV* con una sencilla exploración de una escena tridimensional, permitiéndole ver los objetos dibujados en casi cualquier posición, siempre y cuando no se pierda de vista al patrón asociado al objeto.

En las próximas páginas se tratarán algunos puntos acerca de la creación del visualizador, su funcionamiento, así como los cambios y ajustes necesarios que debieron realizarse para que este funcionara como su versión original de *OpenGL*.

4.1 Cambios realizados en comparación con la versión de *OpenGL*

El visualizador creado en el presente trabajo, contiene diversos cambios respecto de su antecesor. El primero de ellos pero no menos significativo es la manera en que se captura video. Mientras que la versión anterior adquiría video mediante el *VisionSDK* de *Microsoft*, la versión de *DirectX* lo hace mediante *DirectShow*.

Las anteriores bibliotecas de programación del *VisionSDK* fueron pensadas para tener compatibilidad con el antiguo formato *Video For Windows* (*VFW*), encontrado principalmente en aplicaciones de *Windows 3.1* y *95*, por lo tanto, son un poco lentas comparadas con las nuevas. En este cambio se encuentra una mejora, ya que el flujo de datos de video es superior al que se tenía antes, teniendo como consecuencia un despliegue de video con mas cuadros por segundo. Asimismo, dicho SDK se encuentra formado por un conjunto de clases que cubre diversas necesidades de los programadores que se enfrentan al reto de trabajar con imágenes y video. Dichas clases son las mostradas en la Figura 32.



Relación entre algunas de las clases del *VisonSDK* de *Microsoft*
Figura 32

A partir de la estructura anterior, es posible observar lo mencionado: las clases por las que se encuentra formado este SDK cubren un conjunto de necesidades involucradas en el desarrollo de una pieza de software. A pesar de que este es un buen enfoque, para fines de velocidad y compatibilidad no es apropiado, ya que una envoltura de tal nivel debe de hacer diversas evaluaciones que consumen mas tiempo de procesador. En cambio, si la aplicación es construida a la medida de las necesidades, esta será eficaz y con velocidad adecuada para los fines perseguidos puesto que no se deben de evaluar diferentes estados que van a ser descartados desde un inicio. Esta situación es la que invita a adquirir el video de una manera más dedicada y específica para la aplicación. Construir una aplicación desde el inicio con *DirectShow*, permite al programador elegir y programar de acuerdo a los requerimientos sin necesidad de evaluar estados o considerar eventos que no ocurrirán.

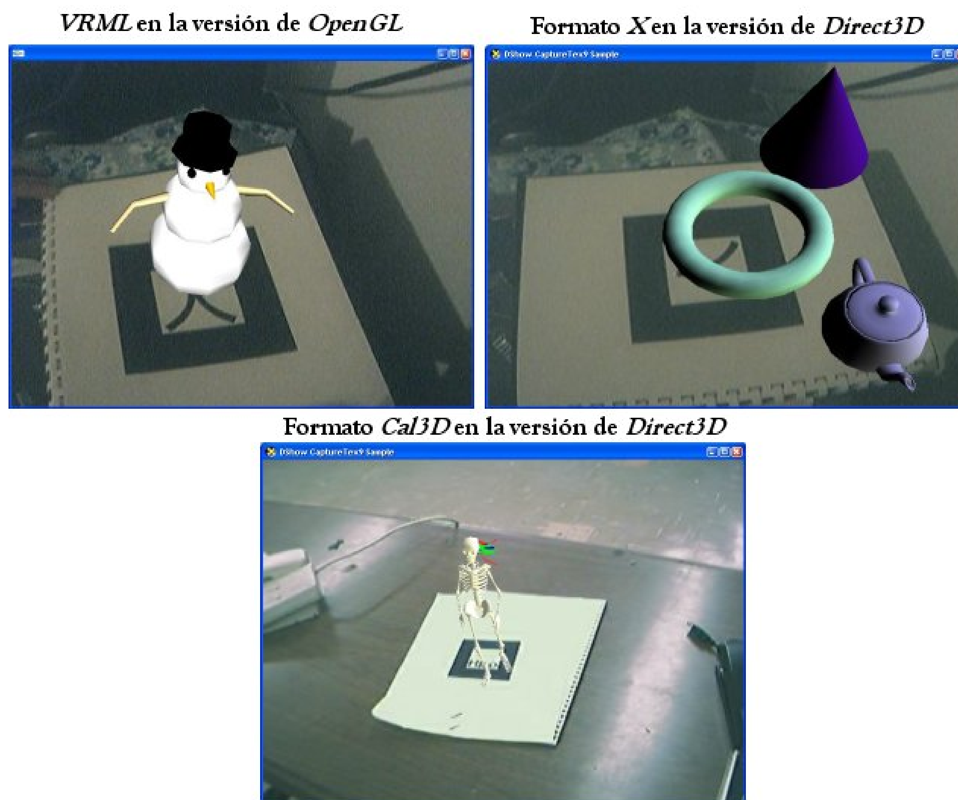
Otra diferencia presente, es el área de video que puede ser mostrada. Mientras que la versión de *OpenGL* permitía ejecutarse a pantalla completa o bien, hacer el área de despliegue mas grande, esto debía especificarse en tiempo de compilación. Para la nueva versión esto ha cambiado, permitiendo al usuario especificar el tamaño de la ventana tal y como se tratará de una ventana común de *Windows*. Esto es una ventaja, ya que el video es adquirido con la resolución de la cámara (por lo general es de 320x240 píxeles) y visualizar una escena de *RA* con una resolución tan pobre, repercute en escasa claridad al ver los objetos virtuales.

Tamaño Original...

Al hacer más grande la ventana...

Ventana de tamaño definible por el usuario
Figura 33

Otro cambio es el referente con el soporte de modelos para ser desplegados. En la versión original, solamente era posible desplegar modelos creados en *VRML*[98]. En la nueva versión el soporte ha sido trasladado a modelos en formato *X* y *Cal3D*. Los archivos *X* son el formato por *default* de *Direct3D* mientras que el segundo formato pertenece a modelos animados por esqueleto. Esto es, el modelo puede ser creado, por ejemplo, en *3DStudio Max* y mediante ciertos *plug-ins*[99], ser exportado a cada uno de los formatos. Para el caso particular del formato *Cal3D*[100], el modelo debe ser asociado a un modificador para crear el esqueleto, especificar los puntos de control sobre éste y ser creada la animación.



Modelos soportados en ambas versiones
Figura 34

Cabe mencionar que la versión original desplegaba los modelos de *VRML* con animación, siempre y cuando se contara con ella. En el caso de la versión para *Direct3D*, por ahora solo son soportadas animaciones en los archivos *Cal3D* mientras que los *X* (a pesar de que contengan alguna) se presentan de manera estática.

Una opción bastante útil, que no se tenía en la versión original, corresponde a un diálogo para seleccionar el libro que se desee cargar. En la versión original, el *MagicBook* cargaba un archivo de configuración por *default* (que contenía la especificación del contenido del libro), dejando al usuario la tarea de modificarlo cada vez que quisiera cambiar el contenido. Para esta versión, se ha incluido un diálogo para la selección del libro. Esto es, si el usuario ha creado varios libros por medio del editor, se podrán cargar mediante dicha ventana.

---

[98] *VRML*: *Virtual Reality Model Language*. Especificación para el diseño e implementación de un lenguaje independiente de plataforma que permita describir escenas de Realidad Virtual.
[99] *Plug-in*: Pieza de software que, añadida a cierto paquete, permite hacer nuevas acciones que originalmente no estaban presentes.
[100] Véase el Apéndice A para mayor información.

Diálogo para la carga de libros
Figura 35

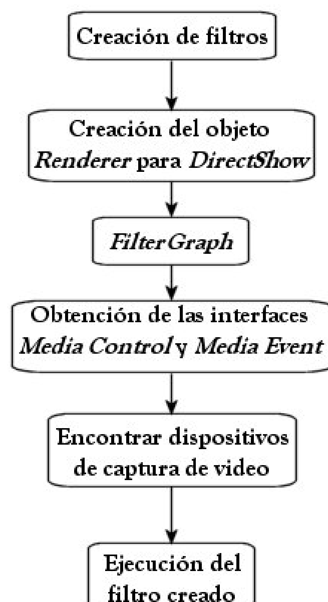
Sin lugar a dudas, el cambio más importante con respecto del antecesor es la capacidad de mostrar los modelos con un efecto o *shader* programable. Esta adición presenta un nuevo abánico de posibilidades, ya que el programador o usuario del *MagicBook* será capaz de mostrar ciertas características o darle un significado gráfico a los objetos mostrados con solo aplicar ciertos *shaders*. Para versiones a futuro, incluso se puede pensar en una escena de *RA* en la que cada objeto que se encuentre en ella tenga un efecto diferente, provocando una experiencia visual más rica y apetecible al sentido humano de la vista.

Shaders* en el *MagicBook
Figura 36

Ahora que se conocen los cambios de la versión de *DirectX* con respecto de la versión de *OpenGL*, se explicarán algunos puntos importantes en la creación del visualizador.

4.2 Captura de video

Como se mencionó anteriormente, la captura de video mediante el *VisionSDK* ha sido reemplazada por *DirectShow*. De esta manera[101], el proceso es llevado a cabo en diversas etapas que van desde la creación de los filtros, la definición de los *pines* de salida, el grafo que representa las conexiones entre los filtro, etc. A continuación se explicará el funcionamiento de este proceso, así como algunos temas relacionados con su implementación.



Inicialización general de *DirectShow* en la aplicación
Figura 37

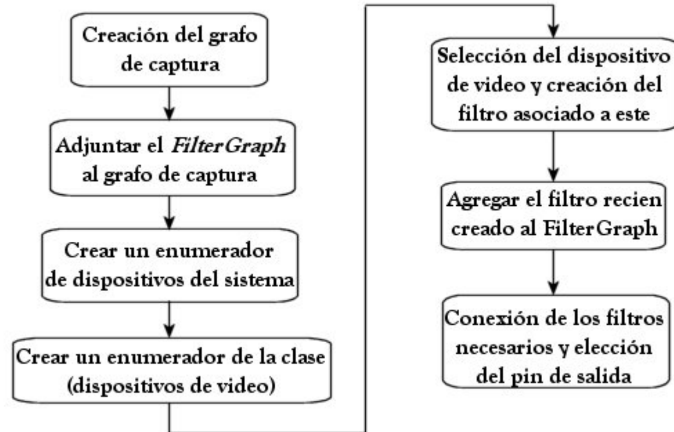
La Figura 37 ilustra los procesos efectuados en la inicialización de *DirectShow* para los fines perseguidos. El primer paso se refiere a la especificación y creación de los filtros. Estos filtros servirán tanto para adquirir el video, como de intermediarios entre el muestreo de imágenes y el mapeo a *Direct3D*. Una vez definidos los filtros a usar, se crea su grafo controlador (*FilterGraph*). Dicho grafo es el componente central de cualquier aplicación de *DirectShow*, ya que se emplea primero para la construcción y posteriormente para el control de los filtros. Asimismo, se encarga de la sincronización, notificación de eventos y algunos otros aspectos.

Obtener las interfaces *Media Control* y *Media Event* a partir del *FilterGraph* es la meta posterior. La primer interfaz provee de métodos para controlar la ejecución, pausa y detención del grafo, mientras que la segunda es útil para obtener las notificaciones de eventos y poder especificar un propio manejo de los mismos.

Una vez que se cuenta con ambas interfaces, entonces corresponde crear el objeto que servirá de intermediario entre la imagen muestreada por *DirectShow* y la textura asociada en *Direct3D*. El objeto recién creado representa al filtro de dibujo, que debe ser agregado al *FilterGraph*. Dicho objeto se encarga de tareas tales como la verificación del formato de video que se adquiere, conversión del tipo, mapear la imagen muestreada como textura entre otras. Estás tareas se abordaran al final del presente apartado.

[101] Consúltese el apartado 2.2.1 Arquitectura de *DirectShow* para observar el modo de operación de la *API*.

Con los filtros creados, casi se tiene preparado al grafo para su uso y comenzar a adquirir video pero, ¿cómo se define el dispositivo? Para ello se debe de encontrar algún dispositivo de video. Una vez mas, debe crearse otro grafo, para especificar la manera en que se captura video. A este grafo, llamado *CaptureGraph* es agregado el *FilterGraph*, ya que es el encargado de controlar el flujo de datos (véase Figura 38).

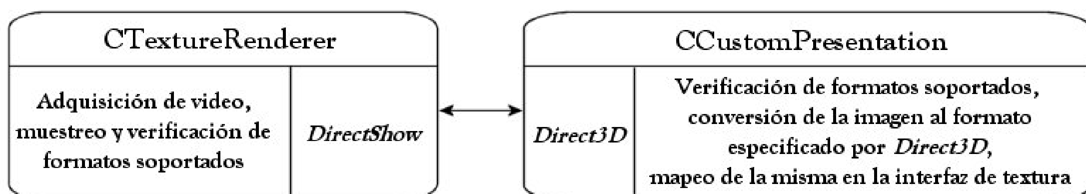


Etapa de selección de dispositivo de video (*CaptureGraph*)
Figura 38

Las dos enumeraciones posteriores sirven para obtener información acerca de los dispositivos (en particular los de video) que el sistema posee. Con dicha lista es posible seleccionar uno de los dispositivos que se encuentren presentes, por *default* en esta versión se ha elegido el primero que se encuentra. Después de estos procesos, se crea el filtro asociado al dispositivo y se agrega al *FilterGraph*, con la finalidad de que cuando se ejecute dicho grafo, todas los eventos sean referidos al dispositivo elegido. La conexión de los filtros y el *pin* de salida tienen que ver con la forma en que internamente son manipulados y procesados los datos provenientes de la cámara.

Una vez que se han creado el *FilterGraph*, *CaptureGraph*, los filtros asociados al dispositivo, etc, es necesario ejecutar al grafo principal para comenzar con la captura de video. Ahora bien, falta mencionar las funciones de los objetos que sirven de intermediarios entre *DirectShow* y *Direct3D*, ya que mediante ellos es mostrado el video. Dichos objetos corresponden a instancias de las clases *CTextureRenderer* y *CCustomPresentation*.

Mientras que la clase *CTextureRenderer* se encuentra mas ligada con *DirectShow*, la otra lo está con el dispositivo de *render* y las interfaces de desarrollo de *Direct3D*. Es decir, el primer objeto se encarga de procesar el video, reconocer su formato y reservar memoria necesaria para almacenar la imagen muestreada. El segundo tiene como tareas la verificación del formato de video para su posterior conversión a un formato soportado por *Direct3D* en sus texturas. Asimismo, se encarga de la creación de la textura y el posterior procesamiento de la imagen de acuerdo al formato que presente.



Relación entre las clases *CTextureRenderer* y *CCustomPresentation*
Figura 39

De esta manera es realizada la captura de video, con algunos de los procesos que involucra la misma. Por otro lado, en la sección 4.6.2 se retomarán estos puntos para explicar la intervención de las clases mencionadas en el *render* del video como parte de una escena 3D de Realidad Aumentada.

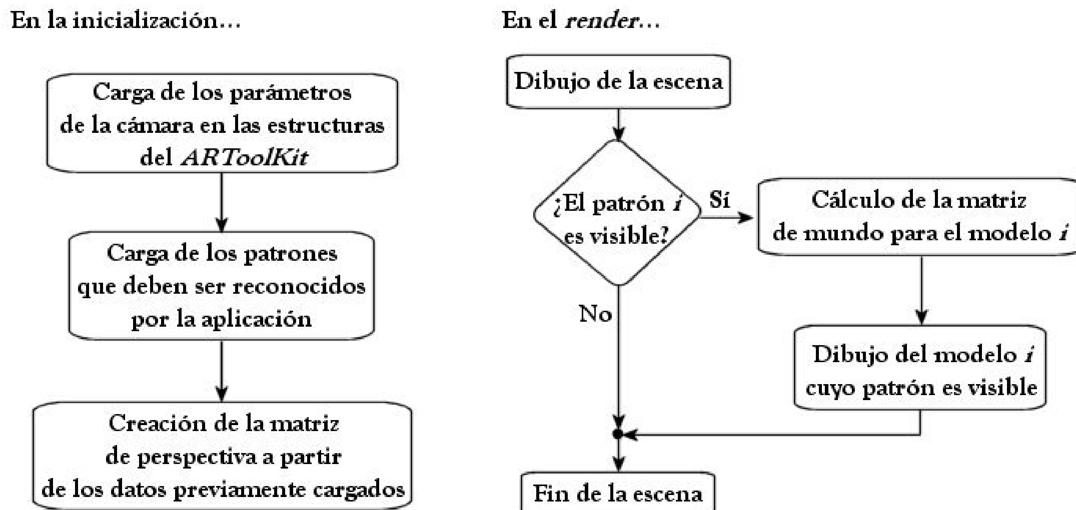
4.3 Empleo de las rutinas del *ARToolKit*

Una pieza importante para la construcción del visualizador, es saber cuando y cuales rutinas llamar para efectuar el procesamiento de la imagen y posterior reconocimiento de patrones. El *ARToolKit* ofrece estas rutinas y más. Dentro de él se han especificado procesos de reconocimiento, operaciones matemáticas, adquisición de video y despliegue de gráficos. Dado que en el presente trabajo se han sustituido por completo la parte gráfica y de video, entonces interesa saber como es que se deben de emplear las demás rutinas.

El primer paso es cargar los parámetros de la cámara ya calibrada en la aplicación, en sus respectivas estructuras de datos. Esto es muy importante ya que con estos datos se calculará la matriz de perspectiva de la escena, misma que definirá el volumen de vista y, por ende, lo que será y no será dibujado. Una vez cargados los datos de la cámara, se invoca a las rutinas que se encargan de llenar las estructuras de datos necesarias para el reconocimiento de patrones.

Una vez concluido el proceso, es necesario cargar los patrones que serán reconocidos en el libro. Para ello, cuando se indica cual es el libro a ser empleado, se carga el archivo de configuración en donde se especifican diversos parámetros, siendo uno de ellos el nombre del patrón asociado con el modelo que se requiera desplegar. Al finalizar la carga de patrones, son ejecutadas algunas rutinas que tienen por objeto la construcción de la matriz de perspectiva para el mundo 3D, creada a partir de los parámetros de la cámara previamente cargados.

Los procesos anteriores son efectuados como parte de la inicialización del *ARToolKit*. Por su parte, el reconocimiento de los patrones es llevado a cabo cada que se dibuja la escena, es decir, cada vez que es llamada la rutina de *render*. Si se encontró que en la imagen actual existe un patrón válido para el libro, entonces es evaluada la matriz de mundo mediante algunas funciones del *ARToolKit*. La Figura 40 ilustra de manera esquemática como son efectuados los procesos aquí mencionados.



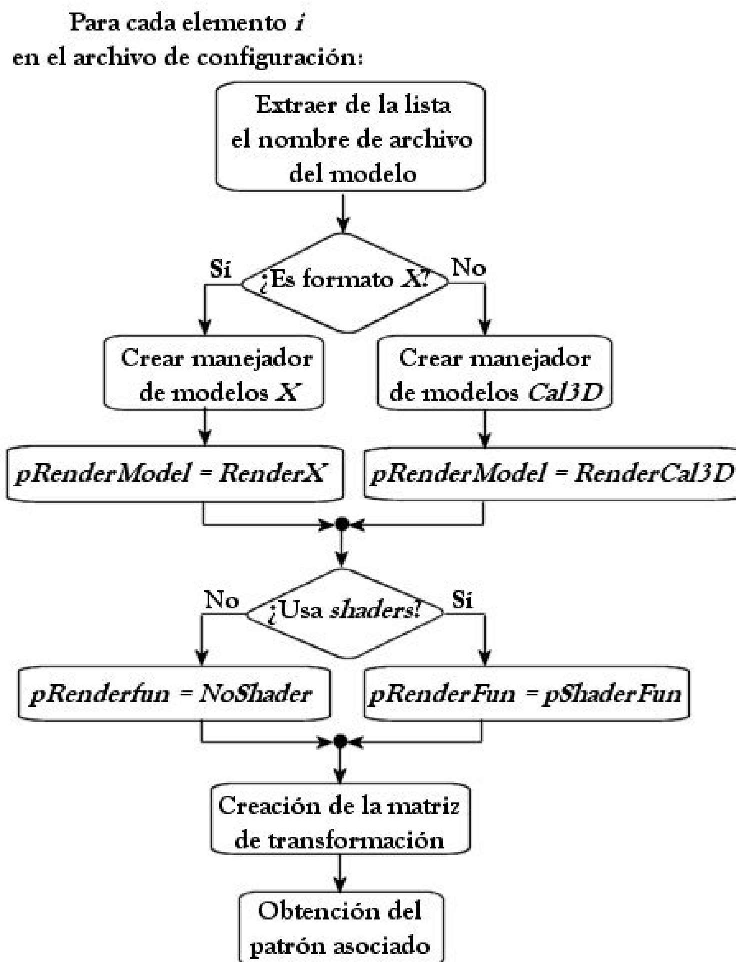
Inicialización y reconocimiento de patrones usando el *ARToolKit*
Figura 40

4.4 Proceso de carga de libros

Con el video inicializado y sabiendo la manera en que son llamadas las rutinas del *ARToolKit*, es conveniente explicar la serie de procesos que intervienen al momento de cargar un libro.

El primer paso es la creación de una lista con la información que contiene el archivo de configuración elegido. Esta lista contiene datos útiles para la creación del modelo (nombre del archivo), si usa un efecto y cual es éste, el patrón asociado y, finalmente, una matriz de transformación para colocar al objeto en la posición que el usuario definió mediante el editor.

Si el archivo fue cargado con éxito, entonces la estructura que maneja los datos contenidos entrega el número de elementos por utilizarse. Con esta información se crea una nueva lista que contendrá los recursos necesarios para controlar la carga y creación de los modelos. El proceso que se lleva a cabo es el mostrado por la Figura 41.



Procesos involucrados en la carga de un elemento del *MagicBook*
Figura 41

En la sección 4.6.3 se explicará el porque del enfoque empleado en el *render*. Dicho enfoque es el que determina la creación de los modelos y su posterior envoltura en una estructura de datos. Por el momento es conveniente entender y enfocarse únicamente al proceso de carga.

Para cada modelo, es creado su manejador correspondiente (ya sea *X* o *Cal3D*), es indicada (mediante un apuntador a función) cual es la función encargada del dibujo, se crea su matriz de transformaciones, los recursos empleados por el *shader*, si es que está asociado a alguno y se índica cual es el patrón asociado al modelo. En caso de que se emplee un *shader*, el apuntador de función de *render* principal es puesto al procedimiento que se indica en el archivo de configuración.

4.5 Ajustes efectuados a las transformaciones

Tal como se mencionó en la sección 4.3, las rutinas del *ARToolKit* son las encargadas de llenar las estructuras de datos referentes al control que es efectuado sobre el reconocimiento de patrones y en la generación de la matriz de perspectiva de la escena.

Si bien, el empleo de dichas rutinas debiese ser integro en la realización de una aplicación que las use, esto no se presentó al implementar la presente versión del *MagicBook*. La documentación[102] índica que para cada cámara que se use se debe efectuar su correspondiente calibración. Sin embargo, en el foro de discusión[103] del sitio, el Dr. Kato (creador del *ARToolKit*) menciona que no es completamente necesaria la calibración, ya que el *ARToolKit* en sus estimaciones lleva compensación ante errores. La anterior situación y una serie de pruebas realizadas con la versión de *OpenGL* usando diferentes versiones de libros y cámaras, condujeron a pensar que en realidad no era necesaria la calibración.

Por tal motivo se creó el proyecto pensando que las rutinas del *ARToolKit*, en conjunto con el archivo existente de calibración eran correctos y la aplicación debiera funcionar como su antecesora. Una vez que se hubo terminado el proyecto y se comenzaba a verificar que el *tracking*[104] y reconocimiento de patrones fueran correctos, se observó que mientras que el reconocimiento era adecuado, el *tracking* de los objetos era malo.

Los errores registrados no se presentaban solamente respecto de las traslaciones sino también sobre las rotaciones. En una primer instancia se pensó que el problema original dependía de la matriz de transformación del modelo, llevando a reflexionar que no era posible una corrección directa a la matriz puesto que es una combinación de las matrices de rotación (en los tres ejes) y de traslación. Es decir, de la matriz compuesta es imposible obtener las rotaciones sobre cada eje para poder aplicarlas en el orden y jerarquía mas conveniente. Considérense las matrices de rotación sobre cada uno de los ejes.

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_y = \begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Recordando que la multiplicación entre matrices no es conmutativa, es obvio encontrar que se cuenta con 6 posibilidades de rotaciones. Se podría pensar entonces en resolver los sistemas para cada combinación y encontrar los valores de los ángulos de rotación para cada uno de los ejes, pero ¿qué ocurre si antes de alguna rotación se efectuaron traslaciones o escalamientos?

---

[102] Refiérase al sitio web: http://www.hitl.washington.edu/artoolkit/ para mayor información

[103] Consúltese el sitio web del foro: http://www.hitl.washington.edu/artoolkit/mail-archive/search.php

[104] *Tracking*: Seguimiento de un objeto a partir de marcas o rastros que lo identifican. En este caso, se refiere al hecho de orientar al objeto de acuerdo a la pose con que lo observa la cámara.

En el caso donde se presenta la situación anterior, es imposible obtener las rotaciones a partir de la matriz compuesta. A causa de esta situación se descartó la posibilidad de encontrar los ángulos de rotación sobre cada eje.

Posteriormente, se llevaron a cabo una serie de pruebas en las que se modificó, por fuerza bruta, dicha matriz y se obtuvieron resultados de un *tracking* en traslaciones aproximado al correcto, pero incurriendo en los mismos problemas respecto de las rotaciones.

Otra matriz que es entregada por las rutinas del *ARToolKit* es la de perspectiva. Gracias a que se cuenta con esta matriz, fue posible abordar el problema de otra manera, ya que de esta matriz depende el volumen de recorte del mundo tridimensional. Dicha matriz define lo que sería el campo de visión, la apertura del lente de la cámara y los planos imaginarios que delimitan el área a ser dibujada. Variando estos parámetros, es posible crear una matriz de perspectiva que contemple la deformación en la imagen causada por la cámara.

La creación de la matriz de perspectiva, utilizando el *ARToolKit*, tiene que ver con los parámetros de configuración de la cámara, evaluados en la calibración de la misma. Entre los comentarios del foro de discusión, se encontró la siguiente información referente a los parámetros de construcción de la matriz:

```
"…Camera parameters are follows.
typedef struct {
    int             xsize, ysize;
    double          mat[3][4];
    double          dist_factor[4];
} ARParam;
xsize and ysize represents image size(ex. 640x480).
mat[3][4] represents perspective transformation matrix of the camera.
It is like
            fx  s   x0  0
            0   fy  y0  0
            0   0   1   0
(fx,fy) is focal length.
s means skew.
(x0,y0) is image center.
…
dist_factor[4] is a distortion parameter…"
```
[105]

En las pruebas que se realizaron durante el desarrollo del software, se encontró que uno de los defectos principales era, como ya se comentó, el desplazamiento sobre un plano que presentaba el modelo virtual. Por ello se decidió dibujar un sistema de referencia derecho (como el empleado por *OpenGL*) para determinar la orientación de la cámara en la escena virtual y, de inicio, se encontró que el origen no correspondía con el centro de la pantalla. Dicha situación, impulsaba que todos los modelos presentaran un desplazamiento respecto de la posición en que debieran aparecer.

Con la información recabada, se manipularon directamente los datos de la matriz de perspectiva (la posición inicial x_0, y_0) para ajustar manualmente el centro del sistema al centro de la ventana. De esta manera se consiguió que los desplazamientos en que se incurría fueran menores, tendiendo a ser casi nulos.

[105] Hirokazu Kato en respuesta a la pregunta en: http://www.hitl.washington.edu/artoolkit/mail-archive/message-thread-00255-Re--FW--Camera-Calibrati.html

Errores en la posición del sistema de referencia respecto del puerto de vista
Figura 42

La deformación que presenta el sistema de ejes de la Figura 42 puede ser ocasionada por los demás factores como la longitud focal del lente y el *skew*[106]. El campo de visión (*Field Of View*) es el ángulo descrito por un cono imaginario, cuyo vértice coincide con la ubicación de la cámara. Dicho ángulo es determinado por la longitud focal del lente en cuestión[107]. Los lentes con longitud focal corta permiten tener un gran ángulo de visión, provocando que los objetos de la escena aparenten estar alejados unos de otros, por lo que los objetos que aparecen por el horizonte son casi imperceptibles mientras que los elementos cercanos aparentan ser inmensos. La Tabla 3 muestra algunas de las longitudes focales estándar de lentes.

Lente	Campo de visión	Tipo de lente
10[mm]	132.01º	Ojo de pez
15[mm]	112.62º	Extra angular
28[mm]	77.57º	Gran angular
35[mm]	65.47º	Gran angular medio
50[mm]	48.45º	Estándar/Normal
135[mm]	18.93º	Teleobjetivo/Larga
500[mm]	5.15º	Superteleobjetivo/Extra larga

Longitudes focales estándar de lentes, *FOV* y nombres
Tabla 3

En el caso de las rutinas del *ARToolKit* se tienen 2 valores de longitud focal, uno en *x* y otro en *y*, esto por la manera en que se representa una cámara virtual. Si bien se trata de seguir el modelo de cámara real, en ocasiones es posible que el programador o el diseñador de la escena quiera presentar una deformación de la misma y especificar el área de alguno de los planos de recorte. La definición de dicho plano de recorte es equivalente a especificar dos ángulos de apertura de la cámara, uno en *x* y otro en *y*. Las rutinas involucradas en la creación de la matriz de perspectiva del *ARToolKit* siguen la segunda convención, comprobándolo de manera cualitativa al observar el sistema de referencia en la Figura 42.

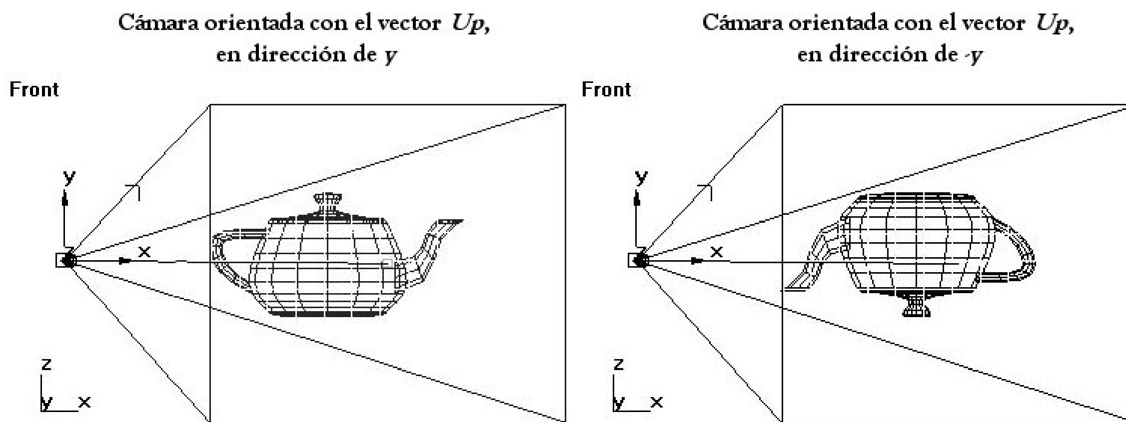
Además de los valores de longitud focal, se tiene el parámetro *skew*. Este término se refiere a la deformación efectuada sobre la proyección de la escena, es decir, el efecto que causaría estirar o encoger un dibujo en papel sobre una dirección cualquiera (Véase Figura 43).

[106] *Skew*: Posición oblicua de un objeto cualquiera.
[107] Jones, Angie, Sean Bonney, Brandon Davis, Sean Miller y Shane Olsen. *3D Studio Max 3 Animación Profesional*. Madrid, España. Ed. Prentice Hall, 1998. p445

Skew
Figura 43

Para la realización de este trabajo, las correcciones logradas a partir de la modificación del centro (x_0, y_0) fueron suficientes para los fines del proyecto. En trabajos a futuro, deben ser cubiertos los puntos referentes a la longitud focal del lente y el parámetro de *skew* con el objeto de llegar a un software que permita efectuar una buena calibración de la cámara con bases teóricas firmes.

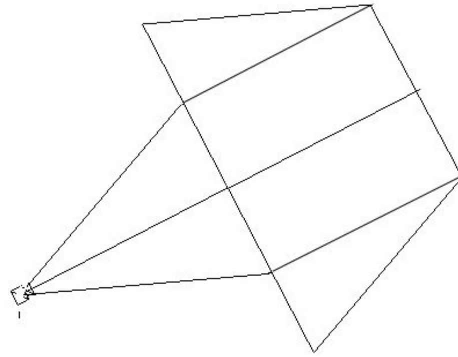
Una vez resuelto el problema de traslaciones, se trató el relacionado con las rotaciones. Esto necesitó de un análisis más a fondo sobre como se encuentra orientada la cámara en el mundo virtual tridimensional. La Figura 44, ilustra la orientación de la cámara, así como la posición en la que sería observado el objeto, con respecto de ella, en el mundo 3D.



Orientación de la cámara y como sería visto el objeto por ella
Figura 44

Con la figura anterior se intenta explicar el primer problema a resolver: los objetos se dibujaban con el eje *Y* o vector *Up* de la cámara invertido. Al crear una cámara con dicho vector corregido, las rotaciones mejoraron pero aun se presentaban problemas respecto de algunos ejes; además, algunas traslaciones del objeto se efectuaban en el sentido contrario.

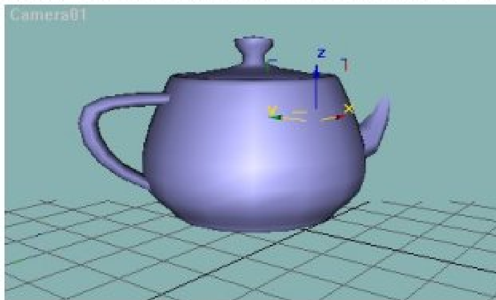
Si se considera la manera en que es creada una matriz de perspectiva (véase la Figura 7 – *Frustum Clipping*), se recordará que es necesario definir un par de planos que sirven para acotar el volumen de vista y definir lo que es capaz de ver la cámara. Dichos planos se definen como plano cercano y plano lejano de corte y por lo general (como sus nombres lo indican) el cercano se encuentra antes, más próximo a la cámara y el lejano después, más distante de la cámara. Ahora bien, considérese la situación en donde estos planos invierten su lugar (Figura 45).



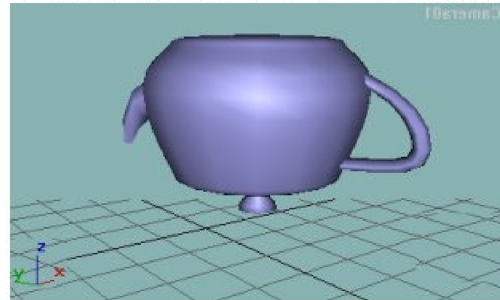
Planos de corte invertidos
Figura 45

En la lógica de las *APIs* de programación, no existe sentido común. Es por ello que si se define un plano de corte cercano como el más alejado de la cámara y viceversa, las operaciones de recorte de la escena se seguirán realizando respecto del orden con que fueron definidos. Pero, ¿cuál es el efecto que se tiene para el espectador? Véase la Figura 46.

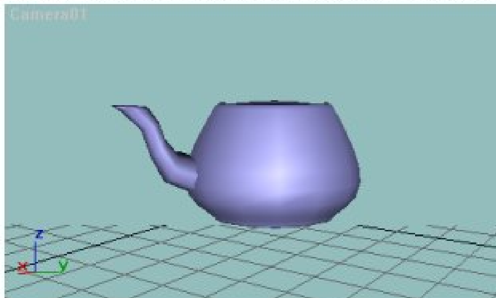
Recorte de la escena con el vector Up de la cámara en dirección de Y y planos de corte adecuados



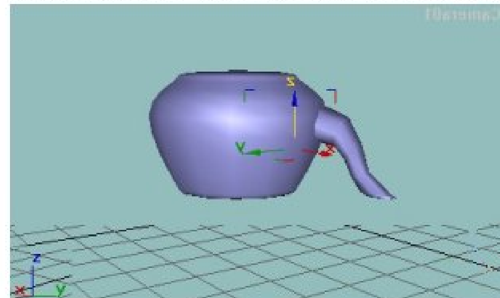
Recorte de la escena con el vector Up de la cámara en dirección de -Y y planos de corte adecuados



Recorte de la escena con el vector Up de la cámara en dirección de Y y planos de corte invertidos



Recorte de la escena con el vector Up de la cámara en dirección de -Y y planos de corte invertidos



Efecto en el volumen de vista al invertir los planos de corte
Figura 46

Cuando se invierten los planos de corte, el primer efecto que se nota es: mientras que el objeto se traslada de un plano hacia el otro y corta con el primero, la escena es proyectada sobre él y comienza a dibujarse. Esto provoca que el objeto se dibuje de atrás hacia delante. Cuando corta con el plano lejano, entonces el efecto es el esperado de ver que se dibuja de la parte anterior del objeto hacia la posterior del mismo.

Este efecto es el que provoca que las rotaciones con respecto de algunos ejes se ejecuten en sentido contrario. Es decir, la matriz de perspectiva que regresa el *ARToolKit* especifica los planos invertidos. Para corregir esto, es necesario invertir la dirección en la que mira la cámara, con el fin de hacer que los objetos se corten en la manera esperada, es decir, el plano que realmente funge como plano de corte cercano. Si la cámara creada observa en la dirección de $-z$, entonces hay que hacer que vea en dirección de z y viceversa.

Las modificaciones hechas corrigen en buena manera el *tracking* de los objetos, sin embrago, tal y como se mencionó anteriormente, es necesario desarrollar un software que calibre de manera correcta la cámara para los fines aquí buscados.

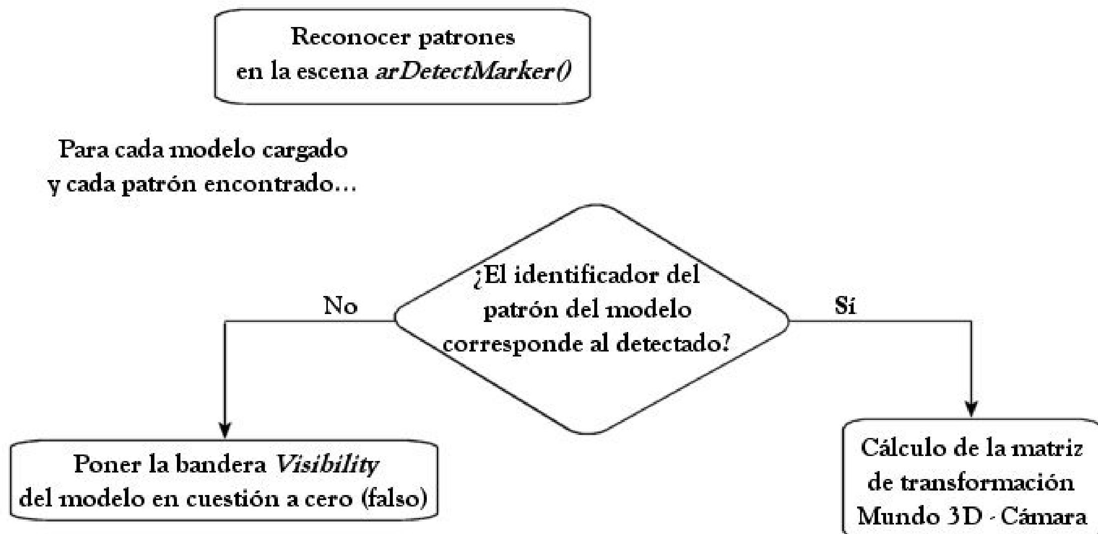
4.6 *Render* de la escena

El dibujo del mundo aumentado es realizado en tres etapas. La primera de ellas concerniente al reconocimiento del patrón en la escena, la segunda referente al dibujo del video como fondo y, finalmente, el *render* de los objetos virtuales en caso de que deba de mostrarse alguno de ellos. A continuación se explicarán cada una de estas etapas así como el porque del enfoque de programación adoptado en ciertas etapas.

4.6.1 Reconocimiento de los patrones

La evaluación sobre la imagen adquirida para conocer si existe un patrón válido en ella, debe de hacerse cada que se vaya a dibujar la escena o cuando se haya muestreado una nueva imagen. El primer esquema presenta algunos inconvenientes ya que las rutinas del *ARToolKit*, por la cantidad de cálculos matemáticos que involucran, requieren de un alto costo computacional. Es por ello que el evaluar el reconocimiento en cada cuadro que se vaya a dibujar es hasta cierto punto absurdo, pues se están perdiendo velocidad.

Si se sigue el segundo camino, el asunto es completamente diferente, ya que se sabe que la imagen acaba de ser adquirida y por lo tanto, requiere su correspondiente análisis para determinar si existe un patrón válido. Por lo tanto, este es el modo a elegir. El proceso a seguir es el mostrado en la Figura 47.



Identificación de patrones en el dibujo de la escena
Figura 47

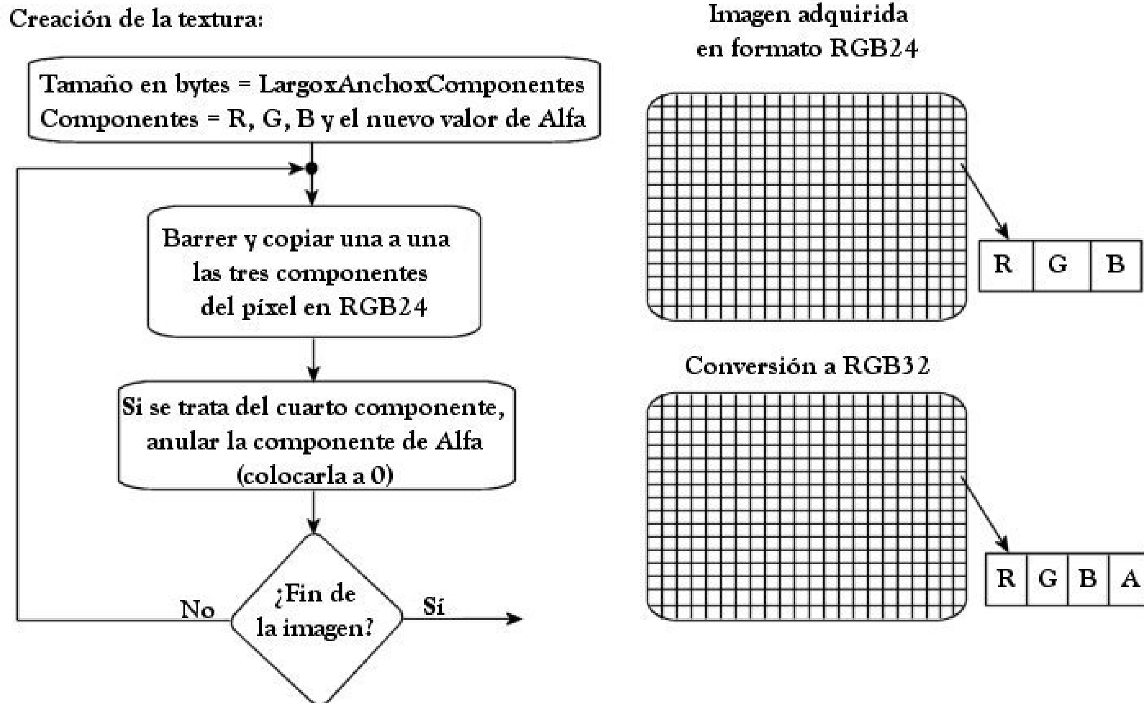
Primero se debe de efectuar el reconocimiento de los patrones que se encuentren en la escena mediante la rutina *arDetectMarker*(), la cual entrega la información necesaria para saber cuantos y cuales fueron los patrones detectados. Enseguida, se debe de analizar, para cada patrón por el que se encuentra formado el libro, si éste fue identificado. En caso afirmativo se procede a calcular la matriz de transformación del modelo. Si el resultado es negativo, entonces una bandera que indica si el objeto es visible o no, es puesta a cero, indicando que el objeto no debe ser dibujado. Esta es la parte concerniente al reconocimiento que se debe de efectuar, posteriormente, esta información será empleada durante el *render* de los modelos (sección 4.6.3).

4.6.2 Video

Mediante los filtros creados con *DirectShow* se lleva a cabo la captura de video de la cámara web. Dentro de dichos filtros, como ya se mencionó en la sección 4.2, se incluyen los correspondientes a lo que sería el reproductor de video y el muestreo del mismo para obtener imágenes. Si se recuerda que *DirectShow* solamente es útil para los fines de adquisición de video y este es necesario incluirlo en una escena 3D, es obvio pensar que el dibujo del mismo será hecho mediante las funcionalidades que presenta *Direct3D*. Como se mencionó anteriormente, las clases *CTextureRenderer* y *CCustomPresentation* son las intermediarias entre las *APIs* mencionadas. La Figura 39 muestra la relación que existe entre ambas clases.

Revisando primero la clase *CTextureRenderer*, el proceso de verificación de tipo de video tiene que ver con la manera en que vienen ordenados las componentes de color de cada píxel. Esto es, solamente se permiten dos tipos de video, el que se adquiere en formato RGB24 (8 bits para cada componente de color) y el formato YUY2. La conversión del tipo (efectuada por *CCustomPresentation*) tiene que ver con la adecuación del formato de píxel al usado por *Direct3D* en mapas de textura. Esto es, si el video se conforma por arreglos en RGB24, entonces la imagen muestreada deber ser transformada a un arreglo de RGBA32. En otras palabras, se debe incluir 8 bits adicionales para especificar el canal alfa de cada uno de los píxeles que conforman la imagen. De esta manera, la imagen que se crea como mapa de textura, tendrá 8 bits mas de información por píxel.

Finalmente, esta imagen nueva es copiada en el espacio de memoria destinado para la textura que será mapeada como el fondo de la escena y que representa el mundo real. La Figura 48 representa la manera en que esto se realiza.



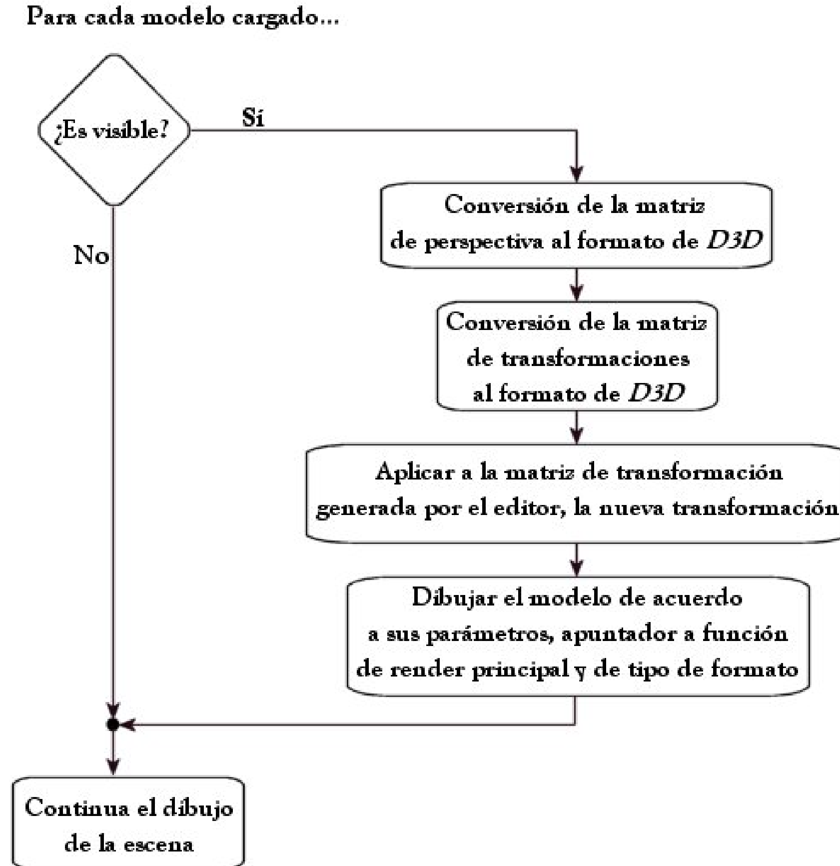
Mapeo de *RGB24* a *RGB32* por el objeto *TextureRenderer*
Figura 48

4.6.3 Modelos

Hasta ahora se ha explicado la manera en que el reconocimiento de patrones y el dibujo de las imágenes muestreadas es llevado a cabo, es decir, se cuenta con la escena real y el conocimiento de cuales son los objetos virtuales a ser dibujados. Es turno de analizar el proceso de *render* de los modelos.

Recordando, los modelos, o mejor dicho, los recursos que contienen la información de ellos, se encuentran en una lista que fue creada durante la inicialización de la aplicación, justo después de cargar el libro. En dicha lista se encuentran sus manejadores y algunas otras características como son las matrices de transformación (la generada por el editor y la calculada por las rutinas del *ARToolKit*). Estos dos aspectos son los que interesan durante el proceso de *render*. Los manejadores son empleados para indicar que se va a dibujar, los apuntadores a función indican la manera en que los modelos serán dibujados y, finalmente, las matrices proporcionan la orientación, posición y tamaño de los mismos.

En el capítulo anterior, se trató y explicó el porque usar apuntadores a función para los diversos procesos de *render* que se tendrán en la aplicación. Para realizar el dibujo de esta aplicación, se retomará el enfoque previamente planteado y será de esa manera, ya que los apuntadores a función ofrecen la flexibilidad necesaria para que cada modelo tenga su propio modo de dibujo y sin depender de los demás. Esto es, que la forma de dibujar los modelos únicamente dependa de su formato y el efecto que se le haya aplicado, no más. El proceso que se sigue para dibujar un modelo es el siguiente (véase Figura 49):



Dibujo de un modelo en el *MagicBook*
Figura 49

Primero se debe determinar si el objeto es visible, en caso afirmativo se procede a realizar una serie de cálculos, de lo contrario se continua con el dibujo de la escena. Los cálculos a realizar son respecto de las matrices que permitirán observar el mundo. Para ambos casos, matrices de perspectiva y transformación, es necesario convertirlas al formato de *D3D*. El *ARToolKit* mantiene una convención de postmultiplicación por un vector columna mientras que *Direct3D* efectúa una premultiplicación por un vector renglón. Tal motivo origina que se deban trasponer ambas matrices y colocarlas en una estructura matricial de *D3D*, un elemento de la clase *D3DXMatrix*.

Con las matrices adecuadas al funcionamiento interno de Direct3D, es hora de generar una matriz de transformación final. Dicha matriz es creada a partir de la combinación de las matrices de transformación generada por el editor y la entregada por el *ARToolKit*. Teniendo lista esta serie de parámetros, el modelo está completamente listo para ser dibujado por las rutinas de *render* previamente seleccionadas al momento de la creación de la lista de modelos.

Resumen

A lo largo de este capítulo se han comentado los puntos más importantes involucrados en el desarrollo de la aplicación *MagicBook* para *Direct3D*. Uno de los principales era rehacer las partes gráfica y de adquisición de video, para no emplear las bibliotecas de programación con que proveen el *ARToolKit* y el *VisionSDK* de *Microsoft*. Ya que se planeó usar *shaders* para *Direct3D*, *DirectShow* fue elegido para el video a causa de la compatibilidad inmediata entre las *APIs* de *DirectX* y de *Microsoft*.

Las modificaciones hechas con respecto de la versión anterior consistieron en mejoras de presentación y funcionalidad. Dentro de las primeras se pueden encontrar el tamaño de la venta definible por el usuario, el soporte de modelos en formatos *X* y *Cal3D*, selección entre diversos libros previamente creados y la adición mas importantes: el uso de *shaders* programables en hardware.

Respecto del video, cabe mencionar que tanto la captura de video como el proceso de muestrear imágenes se hizo a la medida de la aplicación, es decir, solamente se crearon los filtros necesarios para abrir un dispositivo de video y utilizarlo para adquirir video proveniente del mundo exterior. Con este filtro y las características de la clase que lo controla, fue posible acceder a los cuadros que lo componen y obtener una copia que serviría a las rutinas de reconocimiento del *ARToolKit*.

El entorno 3D está basado en su totalidad en Direct3D. El dibujo de cada uno de los elementos (fondo y modelos) tienen bases distintas, mientras que el caso del fondo es un elemento 2D, los modelos interactúan por completo en un mundo 3D. Por ello es conveniente trabajar con un plano como fondo, sobre el que simplemente es mapeado el video entrante como textura. Para determinar los objetos tridimensionales, es necesario efectuar previamente el reconocimiento de patrones en el cuadro adquirido, en este momento entran en juego las diversas rutinas del *ARToolKit* que permiten hacerlo. En caso de que se haya detectado, se procede a calcular su matriz de transformación para dibujarlo en la pose adecuada, respecto del punto de vista de la cámara.

Como se mencionó, se presentaron diversos problemas que impedían la correcta visualización de los objetos, mismos que fueron resueltos por fuerza bruta, ya que la documentación con la que se cuenta es pobre y no especifica ni el uso, ni la función que realizan determinadas rutinas. Debido a ello, hubo que efectuar numerosas pruebas, con diferentes objetos y métodos de orientación, para entonces ubicar en que parte ocurría el problema y así conocer la posible solución. Esta fue la manera de encontrar el porque del mal comportamiento y dar una posible solución.

Capítulo V
Shaders

En los capítulos anteriores se han tratado puntos referentes a la construcción del software de edición y despliegue para los libros, sin embargo, la principal mejora y aumento al *MagicBook* se ha planteado únicamente de manera teórica, sin saber a ciencia cierta como operan o como son construidos los *shaders*. La sección 1.6 describe qué es un *shader* así como un poco de su historia. En el capítulo 2, mientras se trataban a los mismos, se explicó la arquitectura de cada una de las unidades de *Vertex* y *Píxel Shader*, así como el porque emplearlos y algunas de sus funciones.

En este capítulo se tratará con mayor profundidad el aspecto lógico. De esta manera, se cubrirán los aspectos concernientes a la definición de un *shader*, algunos puntos importantes respecto del lenguaje ensamblador con que deben ser programados así como la más nueva definición de *shaders* para *Direct3D*, su lenguaje de *shading* de alto nivel (HLSL) reflejado en el formato *Fx*. Finalmente, se hará la especificación de algunos de los *shaders* creados en conjunto con la aplicación.

5.1 Especificación de un *shader* para *Direct3D*

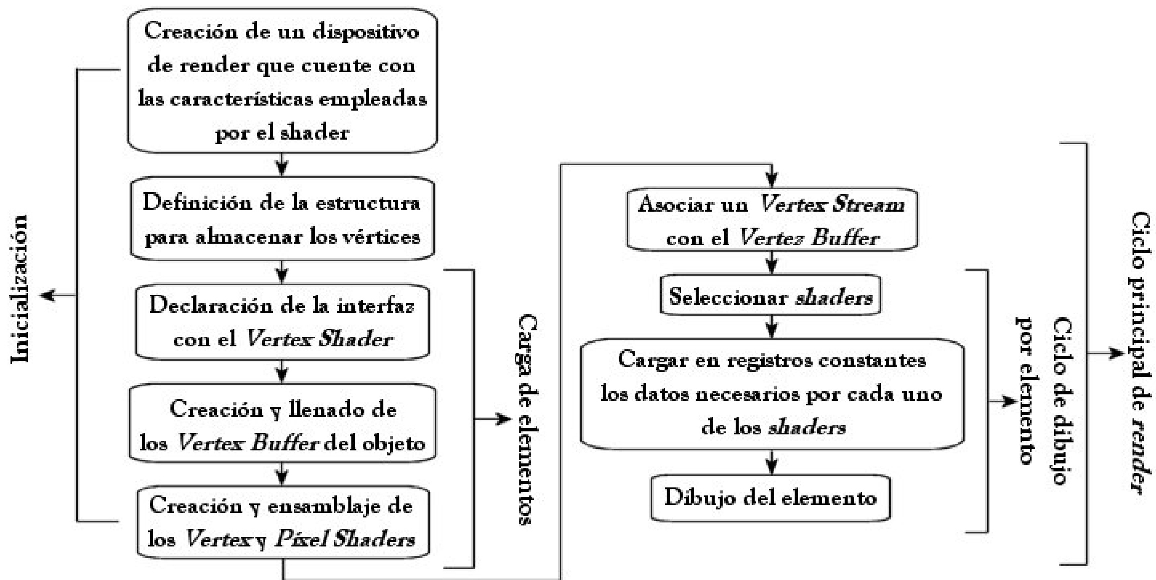
Para comenzar a trabajar con *vertex* y *píxel shaders* se deben de considerar dos casos. Cuando se emplea un *shader* en conjunto con el *pipeline* de *Direct3D* (*FFP*), deben ser consideradas las operaciones a emplearse en orden de ejecución. Esto es, las operaciones tales como el *culling* de caras, planos de corte definidos por el usuario, f*rustum clipping*, etc. son fijas y efectuadas en etapas posteriores a las del *vertex shader*. Para el caso de un *píxel shade*r, las etapas posteriores del *FFP* involucradas son las referentes a efectos de niebla, pruebas de canal alfa y *stencil* buffer, etc. Consúltese la Figura 6 (*Pipeline* de *Direct3D*) para ver con mayor claridad cada una de las etapas así como el orden en que cada una de ellas es aplicada.

El segundo caso corresponde a omitir toda funcionalidad del *FFP*, de esta manera el programador tiene completo control sobre lo mostrado en su aplicación. Este enfoque obliga a que el programador haya definido previamente las constantes y variables que servirán para realizar diversas pruebas codificadas, ya sea en un *vertex* y/o *píxel shader*, para lograr efectos similares o mejorados a los obtenidos mediante el *FFP* de *D3D*.

Para crear un *shader* que funcione en conjunto con un dispositivo de *Direct3D*, se deben de cumplir con las siguientes tareas.

1. Tener un dispositivo de *render* que soporte las características que el programador necesita al momento de implementar sus *shaders*.
2. Crear las estructuras que almacenarán los vértices.
3. Hacer la declaración de la interfaz de *Vertex Shader*.
4. Llenar los *buffers* que mantendrán los vértices sin transformar (*Vertex Buffers*).
5. Crear y ensamblar los *vertex* y *píxel shader*s.
6. Asociar una fuente de vértices (*Vertex Stream*) con los *Vertex Buffers*.
7. Seleccionar los *shaders* que serán usados.
8. Cargar en los registros constantes, los datos necesarios para el funcionamiento del *shader*.
9. Dibujar la escena.

Los primeros 6 puntos son llevados a cabo en tiempo de inicialización ya que no necesitan estarse repitiendo cada vez que se dibuja. Sin embargo, en los casos donde es menester el uso de otro *shader* se deben de repetir los pasos 3 – 7 cada que se cargue un modelo que emplee un efecto diferente al anterior. El ciclo de dibujo está compuesto por los últimos 3 puntos, los cuales, en caso de que se utilice mas de un *shader*, forman un ciclo que se ejecuta tantas veces como efectos se tenga. Para una mejor comprensión, véase la Figura 50.



Creación y *render* usando *shaders*
Figura 50

A continuación se tratarán cada uno de estos puntos con el fin de mostrar y explicar los pasos involucrados en la creación de un *shader* en hardware.

5.1.1 Verificación de soporte en Hardware para *Vertex* y *Pixel Shader*

El programador a creado un dispositivo de *Direct3D* con el cual construir su escena y dibujar todo aquello que necesite mostrar en la misma. Ha decidido usar *shaders* programables en hardware para mostrar efectos determinados sobre ciertos elementos compositivos y demostrativos de la escena. Antes de que comience a desarrollar su trabajo, es importante que verifique que el hardware de su computadora soporte las características que ha decidido emplear. Además, efectuar esta verificación mientras su aplicación es ejecutada, permitirá no incurrir en errores al ser usada en equipos que no soporten *shaders* programables.

La verificación corre a cargo del programador de la aplicación, para ello es necesario valerse de ciertas rutinas y estructuras informativas de *Direct3D* sobre las capacidades del dispositivo. Mediante el siguiente código, se ilustrará como saber las versiones soportadas (tanto de *vertex* como de *píxel shader*) en el hardware, en caso de existir soporte para dichas características.

```
// Device capabilities strucuture
D3DCAPS9 d3dcaps;
g_pd3dDevice->GetDeviceCaps( &d3dcaps );
// At the beginning, suppose there is support for both, pixel and vertex shader
SupportedShad->bPSSupport = true;
SupportedShad->bVSVer = true;
```

```
// Check that the pixel shader supported is at least, compliant with 1.2 definition
if( d3dcaps.PixelShaderVersion < D3DPS_VERSION(1,4) )
{
        if( d3dcaps.PixelShaderVersion < D3DPS_VERSION(1,2) )
        SupportedShad->bPSSupport = false;   //Pixel Shader support disable
        else
                SupportedShad->iPSVer = PS12;          // Ok 1.2 version
}
else            SupportedShad->iPSVer = PS14;          // Ok 1.4 version
// Check that the vertex shader supported is at least, compliant with 1.1 definition
if( d3dcaps.VertexShaderVersion < D3DVS_VERSION(1,1) )
        SupportedShad->bVSVer = false;                  // Vertex Shader support disable
```

De esta manera, para que la aplicación use *shaders* sin mayor problema, se evalúa que el hardware tenga soporte para los mismos y además, que sus versiones sean al menos 1.2 para *Píxel Shader* y 1.1 para *Vertex Shader*. La diferencia entre versiones, tiene que ver con el conjunto de instrucciones disponibles para programar así como las macros que se puedan emplear.

5.1.2 Estructuras de almacenamiento y definición de un vértice

Para tener control sobre los vértices que componen las geometrías de la escena, es necesario que se defina cuales son las características que cada uno de ellos tendrá. Es decir, deben ser definidos los datos que proporcionan alguna información, principalmente aquellos que tienen que ver con los cálculos involucrados durante la transformación e iluminación así como la presentación misma de la escena. De esta manera, se cuenta con diversos datos como son la posición, la normal asociada al vértice, componentes de color (difusa, especular), coordenadas de textura, entre otros.

Podría pasarse por alto una definición para un vértice en particular y generar una global que cubriera a cualquier tipo de vértice. Con lo anterior se tendría mayor flexibilidad en la definición, pero se sacrificaría tiempo y costo de procesador, al igual que ancho de banda en memoria por el envió de datos adicionales. Por ello es conveniente definir cada vértice de acuerdo a la información relevante del mismo para su transformación, iluminación y despliegue. El pseudocódigo presentado a continuación ilustra la manera de realizar la definición correspondiente.

```
struct MyVertexData
{
        Vector3D        Position;       // Vertex position
        Vector3D        Normal;         // Vertex Normal
        Color           Diffuse;        // Diffuse component
        Color           Specular;       // Specular component
        Tex2            TexCoord;       // Texture Coordinates
        …
};
```

Una vez que esta estructura ha sido definida y si se emplea el FFP, debe definirse el formato que especifica la información contenida en el vértice. Para ello, se define una variable que no es más que una combinación de banderas (definidas por *Direct3D*) que describen cada una de las propiedades que puede tener un vértice. A esta variable se le conoce como *Custom FVF*[108]. Un ejemplo es el mostrado a continuación:

[108] Custom *FVF*: Custom Flexible Vertex Format. Formato de Vértice Flexible definido por el usuario como una unión de las máscaras provistas por *Direct3D*.

```
DWORD D3DFVF_MyVertexData = D3DFVF_XYZ |          // Untransformed vertex
                            D3DFVF_NORMAL |       // Vertex Normal
                            D3DFVF_DIFFUSE |      // Diffuse component
                            D3DFVF_SPECULAR |     // Specular component
                            D3DFVF_TEX2;          // Texture Coordinates
```

5.1.3 Declaración de la interfaz para *Vertex Shaders*

Ya es posible conocer si el dispositivo soporta las características necesarias así como definir un vértice acorde a las necesidades del trabajo. Ahora se debe realizar la declaración de interfaz para un *Vertex Shader*. Pero, ¿para qué?

El motivo de esta declaración es informar al *driver*[109] acerca del formato con que cuenta cada vértice y de esta manera declarar el mapeo entre los *Vertex Buffer* y las fuentes de vértices (*Vertex Streams*). Es importante mencionar que cuando se usa el *FFP*, esta declaración no es hecha, el *Custom FVF* es empleado y el mapeo a los registros del *vertex shader* es realizado automáticamente.

Es así que la declaración de la interfaz permite definir un arreglo de *tokens*[110] (de la Tabla 5) asociados con el formato de vértice. Además, son asociados el número de fuente (*Vertex Stream*) y los elementos ordenados tal y como se definen en la estructura asociada. Para definir una declaración se emplea la estructura de *Direct3D* D3DVERTEXELEMENT9 como en el siguiente ejemplo:

```
// Vertex Shader Declarator
D3DVERTEXELEMENT9 MyVSDecl[] =
{
        // stream, offset, type, method
                // usage, usage index
        {0, 00, D3DECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                D3DDECLUSAGE_POSITION, 0},
        {0, 12, D3DECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                D3DDECLUSAGE_NORMAL, 0},
        {0, 24, D3DECLTYPE_FLOAT1, D3DDECLMETHOD_DEFAULT,
                D3DDECLUSAGE_DIFFUSE, 0},
        {0, 28, D3DECLTYPE_FLOAT1, D3DDECLMETHOD_DEFAULT,
                D3DDECLUSAGE_SPECULAR, 0},
        {0, 32, D3DECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
                D3DDECLUSAGE_TEXCOORD2, 0},
        D3DDECL_END()
};
// Validate and get the tokenized version
IDirect3DVDecl9 *pDecl;
g_pDirect3DDevice->CreateVertexDeclaration( MyVSDecl, &pDecl);
// Set the declaration
g_pDirect3DDevice->SetVertexDeclaration( pDecl );
```

Con la anterior definición, se indica el tipo de elemento a ser cargado en el registro especificado del *vertex shader*, así como el tamaño de cada uno de ellos. Por otro lado, los métodos *CreateVertexDeclaration* y *SetVertexDeclaration* de la interfaz *IDirect3DDevice9* son los encargados de crear y seleccionar la declaración.

[109] *Driver*: Pieza de software que indica a la computadora (al sistema operativo) la manera de controlar un dispositivo externo.

[110] *Token*: Unidad básica e indivisible de un lenguaje. Es la unidad independiente más pequeña y con significado en un programa, especificada por un analizador léxico o sintáctico. Un *token* puede ser datos, una palabra reservada del lenguaje, un identificador u otras partes que conforman la sintaxis del lenguaje.

FVF	NAME	VERTEX SHADER
D3DFVF_XYZ	D3DVSDE_POSITION	v0
D3DFVF_XYZRHW	D3DVSDE_BLENDWEIGHT	v1
D3DFVF_XYZB1-5	D3DVSDE_BLENDINDICES	v2
D3DFVF_NORMAL	D3DVSDE_NORMAL	v3
D3DFVF_PSIZE	D3DVSDE_PSIZE	v4
D3DFVF_DIFFUSE	D3DVSDE_DIFFUSE	v5
D3DFVF_SPECULAR	D3DVSDE_SPECULAR	v6
D3DFVF_TEX1-8	D3DVSDE_TEXCOORD0-7	v7-v14
	D3DVSDE_POSITION2	v15
	D3DVSDE_NORMAL2	v16

Mapeo de registros para un *Vertex Shader*
Tabla 4

5.1.4 Creación de un *Vertex Buffer*

Un *Vertex Buffer* es una interfaz de *Direct3D* en donde son copiados los vértices del (de los) elemento(s) a ser dibujado(s). Para su creación, deben seguirse los pasos enumerados a continuación.

1. Definición de la estructura de almacenamiento (características del vértice).
2. Crear un arreglo de estructuras de manera que cubra el número de vértices que vaya a contener el elemento en cuestión y llenarlo con los datos.
3. Crear un *Vertex Buffer* y copiar los datos de la estructura dentro de él.
4. Asociar un *Vertex Stream* con la información contenida dentro del *Vertex Buffer*.

El motivo de copiar los datos contenidos en el arreglo de estructuras hacia el *Vertex Buffer* es para tenerlos en memoria de dispositivo. De esta manera se garantiza el procesamiento más rápido posible. La manera de realizar esto se muestra a continuación:

```
// Space for 300 elements
int iVertexArraySize = 300;
MyVertexData *pVertexArray = new MyVertexData[iVertexArraySize];
// Code to fill the array
// Vertex Buffer creation
IDirect3DvertexBuffer9 *pVertexBuffer;
g_pD3DDevice->CreateVertexBuffer(   iVertexArraySize*sizeof(MyVertexData),
                                    0,                      // Usage
                                    D3DFVF_MyVertexData,// Custom format
                                    D3DPOOL_MANAGED,    //  D3D managed
                                    &pVertexBuffer,         // Pointer to handle
                                    NULL );
// Fill it with the data
BYTE **pLocalVB;
pVertexBuffer->Lock(  0,              // Offset in vertex data
                      0,              // Lock entire buffer
                      &pLocalVB,      // Pointer to the VB
                      0);
memcpy( pLocalVB, pVertexArray, iVertexArraySize * sizeof (MyVertexData) );
pLocalVB->Unlock();
// If Vertex Array is not needed anymore, it can be deleted because there is a copy in
// the video card's memory
```

El proceso anterior es efectuado una sola vez, si es que se cuenta con geometrías en las que se agrupan todos los datos, o bien cada vez que se deba generar una nueva geometría.

5.1.5 Fuentes de vértices (*Vertex Streams*)

Los *streams* indican el lugar de donde han de ser tomados los vértices a procesar. Han sido diseñados para permitir que el *driver* escriba en paralelo sobre múltiples buffers, mientras que adquiere datos en forma lineal. Dicha escritura debe ser rápida, por lo que es hecha vía *DMA*[111].

Además, ya que estas fuentes definen el lugar de donde provienen los vértices, se encuentran relacionadas estrechamente con el accionar de la declaración de interfaz de *Vertex Shaders*, así como con los formatos de vértices flexibles (*FVF*). Si se recuerda, la declaración de *VS* debía ser ordenada en lo que se refiere a los elementos a ser considerados. Este orden, llevado a un nivel mas inferior, permite que se incremente la coherencia de datos en el caché. Esto es, contar con datos contiguos en la memoria brinda la oportunidad de que permanezcan en caché y no se deba acceder a memoria de sistema o del GPU para obtener el siguiente dato, repercutiendo en una mayor velocidad de *render*.

Tanto en *DirectX8* como en *DirectX9*, el máximo de *streams* soportados es de 16. Si bien es buena idea emplear diversas fuentes, en términos de rendimiento no es conveniente, ya que realizar mas transacciones involucra enviar datos extras y esto a su vez una posible perdida de información en el caché.

5.1.6 Creación, ensamblaje y definición de constantes de un *shader*

En los puntos anteriores se ha explicado como preparar la aplicación para trabajar con *shaders* programables, pero no se ha mencionado como crear uno de ellos. Los siguientes pasos ilustran la definición de un *shader* y su puesta a punto.

1. Cargar el *shader* en memoria ya sea a partir de un archivo o bien, tenerlo cargado en la memoria de aplicación.
2. Ensamblar el *shader* y generar el archivo objeto con la colección de *tokens*.
3. Crear el manejador del *shader* ensamblado.
4. Seleccionar el *shader*.
5. Definir los valores constantes y cargarlos en los registros constantes.

Los primeros dos pasos pueden ser efectuados en tiempo de carga o bien precompilados. Por ejemplo, si se planea distribuir comercialmente la aplicación, es conveniente tener los datos previamente ensamblados, con motivos de protección de propiedad intelectual. Si se trata de una aplicación de tipo demostrativa o que va a requerir de modificaciones continuas, entonces tal vez sea mejor definir el código del efecto dentro del código de la aplicación y que se ensamble cada vez que sea compile.

La creación, selección y carga en los registros constantes de un *shader* se debe realizar siempre en tiempo de ejecución y, si se va a usar mas de un*o*, entonces es necesario realizar los dos últimos puntos tantas veces como efectos se tengan. Este proceso tiene que ser hecho antes de llamar a dibujar. Para fines de rendimiento y costo de procesador, el elegir texturas es más caro que seleccionar *shaders*, es por ello que deben de efectuarse en ese orden, primero la(s) textura(s) y enseguida los *shaders*.

[111] *DMA*: Direct Memory Access. Acceso Directo a Memoria. Es un método para transferir datos de una memoria a otra sin tener que pasar por la CPU. Las computadoras con canales DMA pueden transferir datos entre dispositivos con mayor velocidad que aquellas carecientes, en donde se debe pasar forzosamente por la CPU.

Para ensamblar los *shaders*, puede hacerse a partir de un archivo objeto previamente ensamblado o bien, en tiempo de compilación de la aplicación mediante el método *D3DXAssembleShader()*. Si se elige el primer caso, deben de ser programadas las rutinas necesarias para cargar el efecto ya ensamblado en memoria. Finalmente, para el mapeo de las constantes a los registros constantes es importante conocer que existen 3 tipos de ellas para *Vertex Shader*: enteros (I), boléanos (B) y flotantes (F). Son cargadas mediante el método *SetVertexShaderConstant*()* donde * representa el tipo de dato. En el caso de los *Píxel Shaders* solo existen valores flotantes y el método asociado es *SetPixelShaderConstant()*.

5.1.7 Dibujo de la escena mediante *shaders*

Teniendo ya todo preparado para comenzar el dibujo de la escena, éste es hecho como se describe a continuación. Primero el *vertex shader* es llamado por cada vértice que se encuentra en el *Vertex Buffer* para ser transformado, iluminado y especificar estados que serán tomados en cuenta por el *píxel shader*. Cuando un triángulo ha pasado, el proceso de *raster*[112] comienza y se generan los píxeles, quienes pasan al *píxel shader*, donde son evaluadas todas las operaciones que tienen que ver con el color del píxel que va a ser dibujado.

Algunos datos, como las coordenadas de textura, serán interpoladas a partir de los vértices antes de pasar por el *píxel shader*. Esto dependiendo de los estados de *render*. Además, para obtener mayor velocidad de dibujo, se deben de ordenar primero los objetos que comparten texturas de manera que se dibujen al mismo tiempo. A continuación deben ordenarse nuevamente los objetos, pero esta vez de acuerdo al *shader* que puedan compartir.

5.2 Ensamblador de las unidades de *Píxel* y *Vertex Shader*

El ensamblador empleado por estas unidades, fue la primer implementación de lenguaje que permitía trabajar con *shaders* programables en hardware. Este lenguaje (entregado por primera ocasión para la versión 8 de *DirectX*) es el que se ha adoptado durante la realización de este trabajo, ya que cuando se comenzó a desarrollar, únicamente existían dos posibilidades: ensamblador para *DirectX* y el recién lanzado compilador *Cg* de *Nvidia*.

Por cuestiones de documentación e información, el *Cg* de *Nvidia* no era una opción muy buena a seguir. Apenas se contaba con la ayuda que venía con la distribución y muy pocos ejemplos de los desarrolladores. Por su parte, la distribución de Microsoft fue apoyada por casas editoriales y gente de programación, quienes editaron los primeros libros basados en artículos alguna vez publicados en *SIGGGRAPH* o en sitios de desarrollo como *GameDev* y *GamaSutra*[113]. Además, en sitios de internet dedicados a la programación, entusiastas de las gráficas comenzaron a montar tutoriales sobre el uso del lenguaje ensamblador de *DirectX*, ayudando a principiantes a comenzar con su uso.

---

[112] *Raster*: El proceso de dibujar una imagen usando la tecnología con la que funciona la televisión. Involucra dibujar píxel por píxel de la imagen barriendo su totalidad, primero de manera horizontal y enseguida vertical.

[113] Nota: Estos sitios contienen una amplia variedad de artículos concernientes al área de la graficación por computadora. Tratan puntos básicos, avanzados, clásicos y recientes del área. Para mayor información, consultar los sitios: www.gamedev.net y www.gamasutra.com

Es importante conocer que cada unidad posee cierto número de instrucciones por versión. Las versiones 1.0 y 1.1 de *VS* pueden contener hasta 128, mientras que las versiones 2.0 y 3.0 de *DirectX9* soportan hasta 256. Para el caso de un *Píxel Shader* esto es un tanto diferente, ya que depende de dos tipos de instrucciones: las aritméticas y las de textura. A lo largo de las diferentes versiones, se ha visto un cambio en el número de instrucciones. Para las versiones 1.0 a 1.3 se pueden emplear hasta 8 instrucciones aritméticas y 4 de textura. La versión 1.4 permite el uso de 6 instrucciones de textura y 8 aritméticas, sin embargo, mediante el mnemónico '*phase*'[114] es posible duplicar el número de instrucciones. Finalmente, en *shaders* para la versión 2.0 puede haber hasta un total de 32 instrucciones de textura y 64 aritméticas sin importar el orden y sin usar '*phase*', que ha sido eliminado en dicha versión.

Para concluir, no se abordarán ejemplos en esta sección sobre como se escribe un *shader* en ensamblador, ya que la sección 5.4 trata precisamente tal aspecto. En la mencionada sección se especificarán las bases algorítmicas y matemáticas del funcionamiento del *shader*, así como su respectiva implementación en lenguaje ensamblador de *Vertex* y/o *Píxel Shader* según sea el caso.

5.3 *Shaders* en formato *Fx* (HLSL)

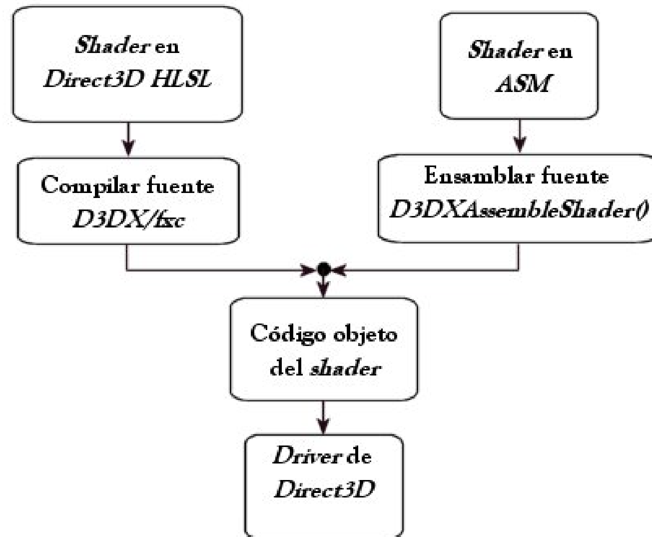
El Lenguaje de *Shading* de Alto Nivel (HLSL por sus siglas en inglés) propuesto por Microsoft a través de la última entrega de *DirectX9*, es una de las nuevas e innovadoras características de la distribución. A medida que se ha continuado explotando el uso de *shaders* en tiempo real, estos se han ido volviendo más poderosos y complejos. Dicha situación, ha originado que la codificación en lenguaje ensamblador de los algoritmos necesarios sea mas compleja.

La interfaz propuesta entre el entorno de trabajo de *D3DX Fx* y el HLSL ofrece una flexibilidad de desarrollo. Esto porque los archivos *Fx* están basados en una sintaxis similar a la del lenguaje C, facilitando el aprendizaje para desarrolladores de dicho lenguaje. Además, contiene una pre-selección de tipos de variables, haciendo más sencillo la declaración y empleo de escalares, vectores y matrices.

El desarrollo de *shaders* mediante el HLSL de *DirectX9* tiene por objetivos la simplicidad, rendimiento y extensibilidad. La programación debía hacerse más simple, ya que la mayoría de los efectos son creados mediante prueba y error, tratando de ver poco a poco el resultado y efectuando cambios hasta llegar al resultado deseado. Con rendimiento y extensibilidad se busca que, a pesar de que se programe en un lenguaje de mayor nivel, se obtenga un rendimiento como si se programara en ensamblador. La extensibilidad tiene que ver con dar soporte a *GPUs* modernos así como a los primeros que contaron con unidades programables.

El proceso seguido para utilizar *shaders* escritos en el HLSL de *DirectX9* es el siguiente (Figura 51):

[114] *Phase*: Permite descomponer el *shader* en dos secciones, con la finalidad de poder usar un mayor número de instrucciones.



Especificación y uso de un *shader* para *Direct3D*
Figura 51

La tarea es dividida en dos partes, la primera compila el *shader* especificado en el lenguaje de alto nivel y entrega un código objeto, equivalente al ensamblado cuando se utiliza el ASM propio. Posteriormente, este código es cargado por el *driver* de *Direct3D* de la tarjeta de video. Como puede ser visto en la figura, un *shader*, ya sea especificado por completo en ensamblador o bien en el HLSL, es tratado de la misma manera después de ensamblar o compilar, según sea el caso.

5.4 Creación de los *shaders* de la aplicación

Ahora que se cuenta con las bases necesarias para comenzar a desarrollar *shaders*, se presentarán los que han sido implementados durante este trabajo. Cada uno contará con la descripción del efecto que realiza, su base teórica así como el código que lo implementa.

5.4.1 Transformaciones comunes mediante *shaders*

El primer efecto en realidad no se debe de considerar como tal, ya que simplemente se encarga de realizar la tarea más básica del *pipeline* de *Direct3D*, que es transformar e iluminar vértices y colorear píxeles. Para ello se hará uso de un *vertex shader* que a continuación es descrito.

El primer paso es transformar e iluminar los vértices de acuerdo con las propiedades y acabado que interese. Comiéncese por realizar la transformación de los vértices. En la aplicación de *Direct3D* se debe de llevar control sobre las transformaciones realizadas sobre el objeto. Estas transformaciones son acumuladas en la llamada matriz de transformación de mundo. Cuando se opera en programas creados para *OpenGL*, dicha matriz es conocida por MODELVIEW mientras que en *Direct3D* es la que afectaría al modelo, es decir WORLD. Para obtener la proyección es necesario que se cuente con la matriz de perspectiva (PROJECTION) y la que representa la cámara (VIEW). Este arreglo de matrices servirá para obtener la posición del vértice en el espacio de recorte.

Si se recuerda que las unidades de *vertex* y *píxel shader* son de tipo vectorial, es necesario pasar en cierto orden el arreglo matricial, de manera que se efectúen el mínimo de operaciones para transformar al vértice. Existe una macro llamada **m4x4** que efectúa un producto punto acumulado de un vector por otros cuatro. La manera de operación es mostrada a continuación.

```
m4x4  r0, v0, c[MATRIX]          dp4   r0.x, v0, c[MATRIX+0]
                                 dp4   r0.y, v0, c[MATRIX+1]
                                 dp4   r0.z, v0, c[MATRIX+2]
                                 dp4   r0.w, v0, c[MATRIX+3]
```

La instrucción **dp4** es en realidad el producto interno de los vectores v0·c[MATRIX+i]. La expansión que se tendría es la siguiente:

$$
\begin{aligned}
v0 \bullet c[MATRIX + i] = {} & v0.x \cdot c[MATRIX + i].x + \\
& v0.y \cdot c[MATRIX + i].y + \\
& v0.z \cdot c[MATRIX + i].z + \\
& v0.w \cdot c[MATRIX + i].w
\end{aligned}
$$

La manera de transformar un vector mediante una matriz X de 4x4 es realizada de la siguiente manera:

$$
dest = v0 \cdot WorldMatrix = \begin{bmatrix} v0.x & v0.y & v0.z & v0.w \end{bmatrix} \begin{bmatrix} wmA & wmB & wmC & wmD \\ wmE & wmF & wmG & wmH \\ wmI & wmJ & wmK & wmL \\ wmM & wmN & wmO & wmP \end{bmatrix}
$$

$$
\begin{aligned}
dest.x &= v0.x \cdot wmA + v0.y \cdot wmE + v0.z \cdot wmI + v0.w \cdot wmM \\
dest.y &= v0.x \cdot wmB + v0.y \cdot wmF + v0.z \cdot wmJ + v0.w \cdot wmN \\
dest.z &= v0.x \cdot wmC + v0.y \cdot wmG + v0.z \cdot wmK + v0.w \cdot wmO \\
dest.w &= v0.x \cdot wmD + v0.y \cdot wmH + v0.z \cdot wmL + v0.w \cdot wmP
\end{aligned}
$$

Del análisis anterior, es posible darse cuenta que al emplear la instrucción **m4x4** se debe operar con la matriz de transformación transpuesta para obtener el resultado deseado. De lo contrario, se estaría operando con las filas de la matriz en lugar de ser operadas las columnas de la misma.

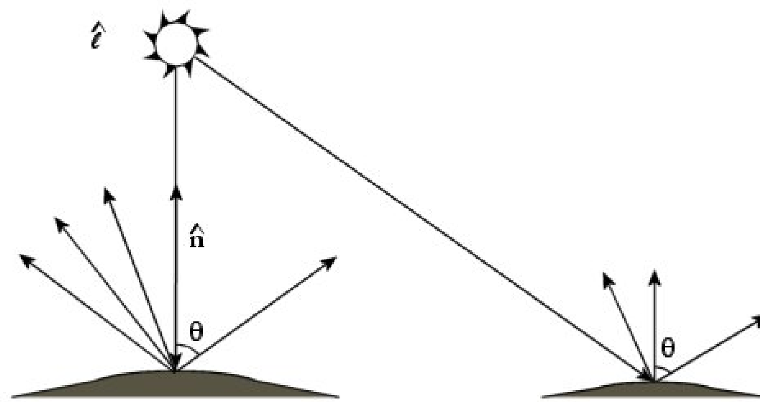
Ahora bien, si en la aplicación (cómo es el caso de la actual) se efectúan escalamientos, los vectores normales sufren una deformación en dirección y magnitud, por lo que no representan correctamente las normales en cada uno de los vértices. Ya que de estos vectores normales depende la iluminación del vértice, es necesario mantenerlos siempre en su dirección correcta. Para ello es utilizada la matriz transpuesta de la inversa de la transformación[115].

El código que a continuación se presenta implementa lo mencionado y toma como requisitos que se coloque la transpuesta de la matriz de transformación en espacio de recorte (Model·View·Projection) y la matriz transpuesta de la inversa de la transformación del mundo en una serie de registros constantes, además de definir la posición del vértice en el registro **v0** y la normal en **v3**.

[115] Nota: Para mayor información sobre las matemáticas involucradas en la transformación de normales para matrices no ortogonales refiérase a: Turkowski, Ken. *"Properties of Surface-Normal Transformations" en Graphics Gems I*. Ed. Academic Press. Estados Unidos 1990. p539

```
// Transform vertex position
m4x4   r0, v0, c[TRANS_MVP]
// Put the transformed vertex into the output position register
mov    oPos, r0
// Recalculate normal
m3x3   r1, v3, c[INV_TRANS_M]
// Normalize the normal
dp3    r1.w, r1, r1
rsq    r1.w, r1.w
mul    r1, r1, r1.w      // r1 =  n normalized
```

Una vez transformado el vértice, es necesario iluminarlo. Se utilizará el modelo de iluminación difusa, el cual, únicamente toma como base la componente difusa de la luz, la modula con la misma componente del material y el factor de iluminación calculado, tomando como base la dirección de la luz y el vector normal. El modelo físico es el mostrado en la Figura 52.



Iluminación difusa
Figura 52

Este modelo de iluminación representa la luz que es absorbida por una superficie y que es reflejada en todas direcciones. La intensidad reflejada depende únicamente de la dirección con la que la luz incide en la superficie y el material de ella. Por lo tanto, este tipo de iluminación es una función del ángulo existente entre la luz y la superficie[116]. Recordando, un *vertex shader* opera sobre cada vértice de la escena, es por ello que se emplean también vectores normales por vértice y no por cara[117]. De esta manera, para calcular la intensidad con la que es iluminado un vértice se emplea la siguiente expresión:

$$i_d = (\hat{n} \bullet \hat{l})(m_d \cdot s_d)$$

donde $\hat{n}$ y $\hat{l}$ representan los vectores normal al vértice y de dirección de la luz, mientras que m_d y s_d son las componentes difusa del material y de la luz respectivamente.

[116] Nota: A la iluminación difusa se le conoce también como Sombreado de Lambert (*Lambertian Shading*) después de que Lambert estableció su ley del coseno. Dicha ley establece que la intensidad de la luz reflejada es proporcional al coseno entre la normal del vértice y de la dirección de la luz.

[117] La diferencia entre usar normales por vértice y por caras, recae en el tipo de sombreado que se tendrá. Si se emplea normal por vértice se hace un sombreado de Gouraud mientras que la utilización de normales por cara generará un sombreado de tipo plano o *Flat Shading*.

Es importante recordar que el producto internos de dos vectores en R^n es igual al producto de sus normas con el coseno del ángulo que forman entre ellos: $vA \cdot vB = \|vA\| \|vB\| \cos\theta$. Si el ángulo entre los vectores es de 0º, entonces se tiene máxima iluminación mientras que, por el contrario, si el ángulo es de 90º el vértice no es iluminado. En el caso de que formen 180º, el coseno es de −1, por lo tanto, el vértice no debe ser iluminado. Para ello, es conveniente hacer un acotamiento en la evaluación del producto interno de los vectores al rango [0, 1].

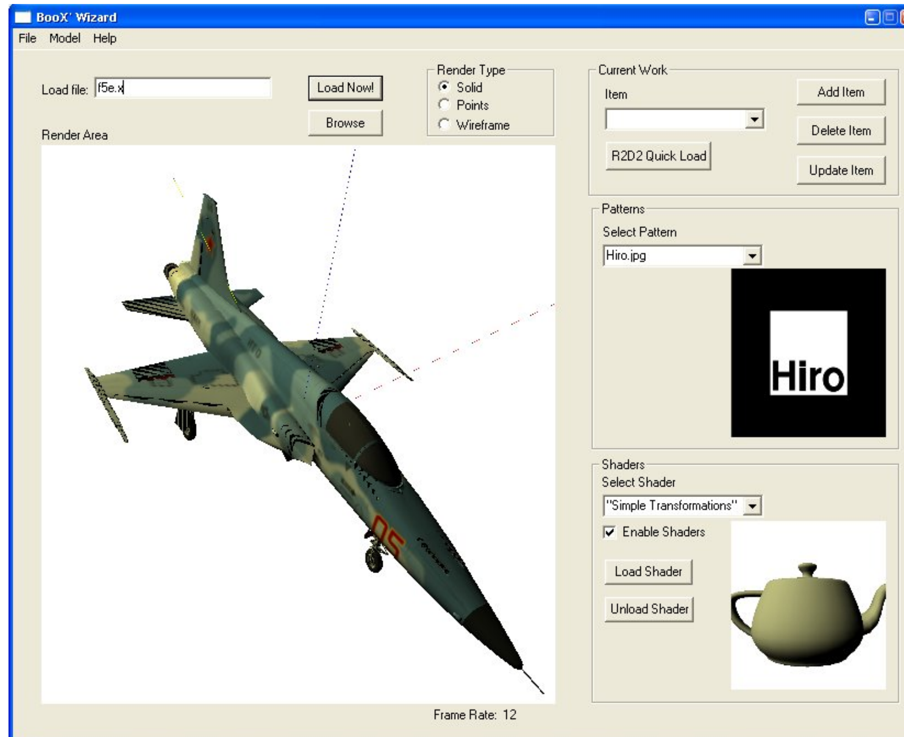
El código encargado de producir lo anterior es el que se muestra a continuación. También se le ha agregado la componente respectiva para lograr iluminación ambiente, la cual no requiere mayor explicación ya que solo consiste en modular la componente ambiente del material con la de la luz.

```
// Compute n·l
mov     r2, -c13            // Change light direction
dp3     r2.w, r2, r2        // Normalize the light
rsq     r2.w, r2.w
mul     r2, r2, r2.w        // r2 = l normalized
dp3     r3.w, r1, r2
// Modulate vertex' diffuse component
mov     r4, c28             // Vertex' diffuse component
mul     r5, r4, c12         // Light's diffuse component
mul     oD0, r5, r3.wwww    // Modulate with vertex intensity
// Modulate vertex' ambient component
mov     r6, c14             // Get light’s ambient component
mul     oD1, r6, c29        // Modulate against vertex’ ambient component
// Output texture coordinates
mov     oT0, v7
```

Para ejecutar este *shader* y obtener los resultados esperados es necesario habilitar ciertas banderas concernientes a las etapas de textura, las cuales son:

```
pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,   D3DTOP_MODULATE2X );
```

las cuales indican que debe modularse la componente difusa del vértice con la textura (si es que posee) y además, aplicarle un pequeño aumento a la intensidad del color. Lo último no es absolutamente necesario, pero, si se recuerda que los valores se obtendrán a partir del coseno del ángulo entre los vectores normal al vértice y el director de la luz (en el rango [0, 1]), entonces se comprenderá que se están opacando los colores. En otras palabras, solamente es necesario para que el objeto a dibujar se vea más claro. El resultado de este *shader* puede ser apreciado en la Figura 53.



Captura de pantalla del shader: *Simple Transformations*
Figura 53

5.4.2 *Crystal Shader*

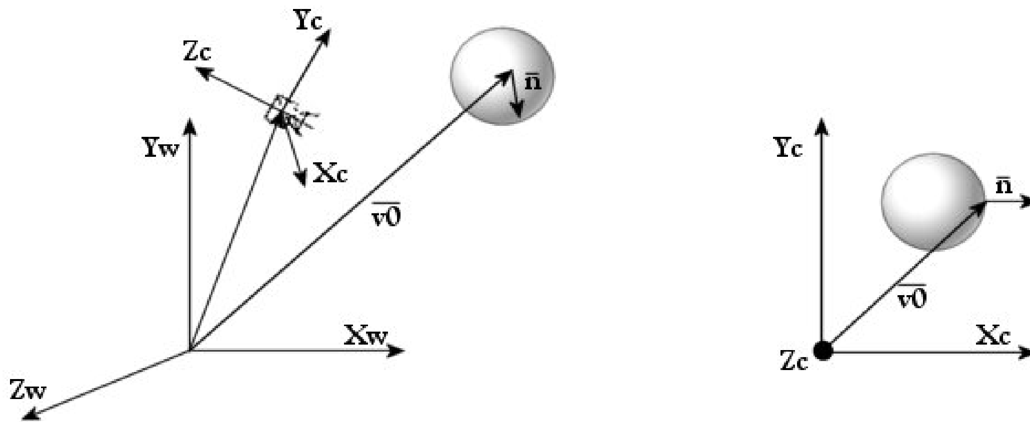
El *shader* descrito a continuación genera un efecto que muestra los objetos como si se encontraran construidos por cristal. Como requerimientos se tienen la matriz transpuesta de transformación a espacio de recorte (TransMVP_Matrix), la matriz transpuesta de transformación a coordenadas de cámara (TransMV_Matrix) y el color con que se va a delinear el objeto.

A su vez, mediante la declaración del *shader*, la posición del vértice se lleva en el registro **v0** y la normal en **v3**. El código en ensamblador aquí mostrado es el que se encarga de generar el efecto. Primero se mostrará el código y enseguida su explicación.

```
// Vertex Shader 1.1
vs.1.1
// c0   – Color and clamp values (0.0, 0.5, 1.0, 2.0)
// c10  – TransMV_Matrix        c14 – TransMVP_Matrix
dcl_position0   v0              // Vertex position
dcl_normal0     v3              // Normal
// Transform vertex into camera coordinates and normalize it
m4x4    r0, v0, c10
dp3     r0.w, r0, r0            // Normalize position
rsq     r0.w, r0.w
mul     r0, r0, r0.w
// Transform normal into camera coordinates and normalize it
m3x3    r1, v3, c10
dp3     r1.w, r1, r1            // Renormalize normal
rsq     r1.w, r1.w
mul     r1, r1, r1.w
// Find the angle between position and normal vertex' vectors
dp3     r3.x, r0, r1
// Emit projected position
m4x4    oPos, v0, c14
```

```
// Clamp to [0, 1] and emit as diffuse alpha
mad     oD0.w, r3.x , c0.y , c0.y
// Black Silhouette
mov     oD0.xyz, c0.x
```

Primeramente, la posición y la normal del vértice son calculadas en coordenadas de cámara. Con dicha transformación, se pretende saber cuales son los vértices que, en espacio de cámara o vista, en conjunto con su normal conforman la periferia del objeto. A continuación, es necesario renormalizarlos pues es casi seguro que ambos vectores sufrieron un escalamiento en su correspondiente longitud.



Transformación a coordenadas de cámara
Figura 54

Una vez renormalizados ambos vectores se calcula el producto interno entre ellos ($vA \cdot vB = \|vA\| \|vB\| \cos\theta$). El motivo es para generar el termino alfa de la componente difusa del vértice. Pero, ¿por qué el producto interno? Cuando los vectores con los que se opera son unitarios, esta operación representa el coseno del ángulo entre ellos. Si bien el ángulo no importa demasiado, el coseno del mismo es de utilidad ya que entrega resultados dentro del rango [-1, 1]. Por lo general, las componentes alfa van de 0 a 1, siendo la primera opaca y la segunda 100% transparente.

De esta manera, se cuenta con un valor dentro del intervalo [-1, 1] que si se multiplica y suma por 0.5, entonces pertenece al intervalo deseado para componentes de canal alfa: [0, 1]. Lo anterior es evaluado por las siguientes instrucciones:

```
// Find the angle between position and normal vertex' vectors
dp3     r3.x, r0, r1
// Clamp to [0, 1] and emit as diffuse alpha
mad     oD0.w, r3.x , c0.y , c0.y          // n Ε [-1, 1] , n*0.5+0.5 -> n Ε [0, 1]
// Black Silhouette
mov     oD0.xyz, c0.x
```

La última instrucción, indica que el vértice será de color negro, en el supuesto que deba ser pintado. Cabe mencionar que para que este *vertex shader* funcione de la manera planteada, deben de habilitarse las siguientes etapas de textura y estados de render:

```
// Set up for silhouette shader
pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,    D3DTOP_MODULATE);
pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP,    D3DTOP_DISABLE );
```

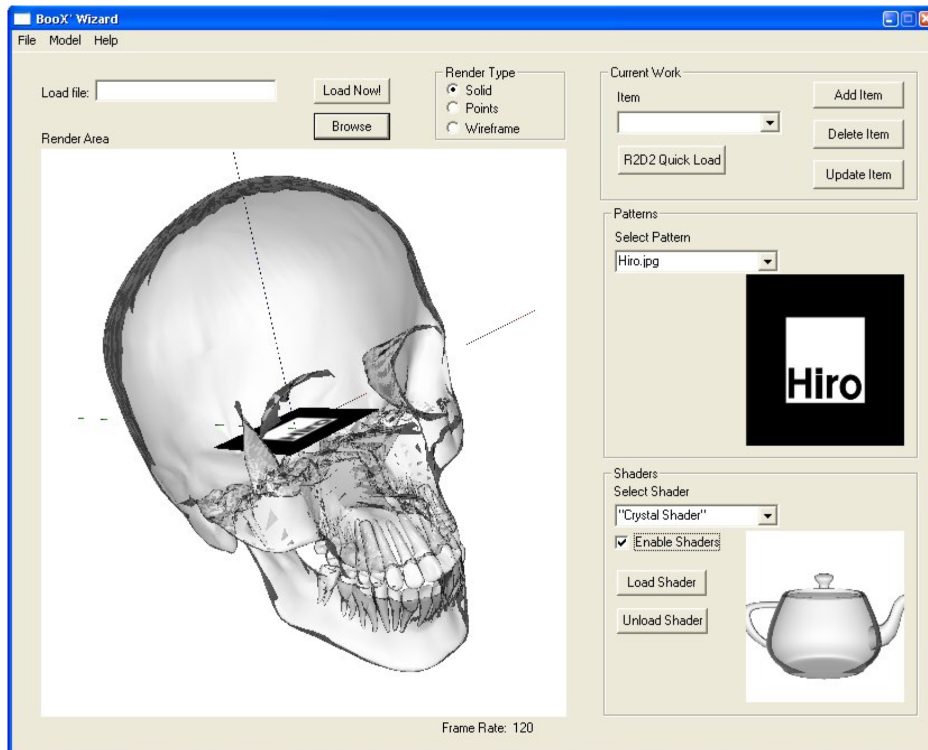
```
// Manage Blending
pd3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE,  TRUE );
pd3dDevice->SetRenderState( D3DRS_SRCBLEND,          D3DBLEND_INVSRCALPHA);
pd3dDevice->SetRenderState( D3DRS_DESTBLEND,         D3DBLEND_SRCALPHA);
pd3dDevice->SetRenderState( D3DRS_ALPHATESTENABLE,   TRUE );
pd3dDevice->SetRenderState( D3DRS_ALPHAFUNC,         D3DCMP_GREATEREQUAL );
pd3dDevice->SetRenderState( D3DRS_ALPHAREF,          0x50);
```

De esta manera se indica que se debe de usar como fuente de color, aquel modulado por la textura y el material del objeto. Para determinar que elementos son dibujados y cuales no, se usa el *Alpha Blending*, que es una técnica que controla la manera en que el color de una primitiva se combina con píxeles existentes en el *frame buffer* para emitir un color de píxel final.

Cuando se efectúa dicha técnica, dos colores son mezclados: uno fuente y otro destino. El primero de ellos pertenece al objeto transparente mientras que el segundo corresponde al color del píxel actual. En otras palabras, el color de destino es el resultado de dibujar algún otro objeto que se encuentra tras del transparente, esto es, el objeto que será visible a través del transparente. Esta técnica emplea una formula para controlar la proporción entre los objetos fuente y destino.

Final Color = ObjectColor * SourceBlendFactor + PixelColor * DestinationBlendFactor

Donde *ObjectColor* representa la contribución de la primitiva que es dibujada mientras que *PíxelColor* corresponde a la del *frame buffer*. Es por ello que con el grupo de instrucciones arriba mostradas, se prueban el canal alfa de todos los píxeles para saber si son transparentes o no y se les invierte el color, produciendo el resultado que se muestra a continuación.



Crystal Shader
Figura 55

5.4.3 Iluminación y sombreado *Gooch*

Entre los diferentes modelos de iluminación existentes, existen dos tendencias: la del foto-realismo y la no foto-realista. Mientras que en la primera se encuentran diversas técnicas que van desde las más sencillas (cómo es el caso de la iluminación difusa), iluminación *phong* hasta algunas basada en física como el *RayTracing*, el segundo caso tiende más hacía el aspecto artístico de la imagen generada.

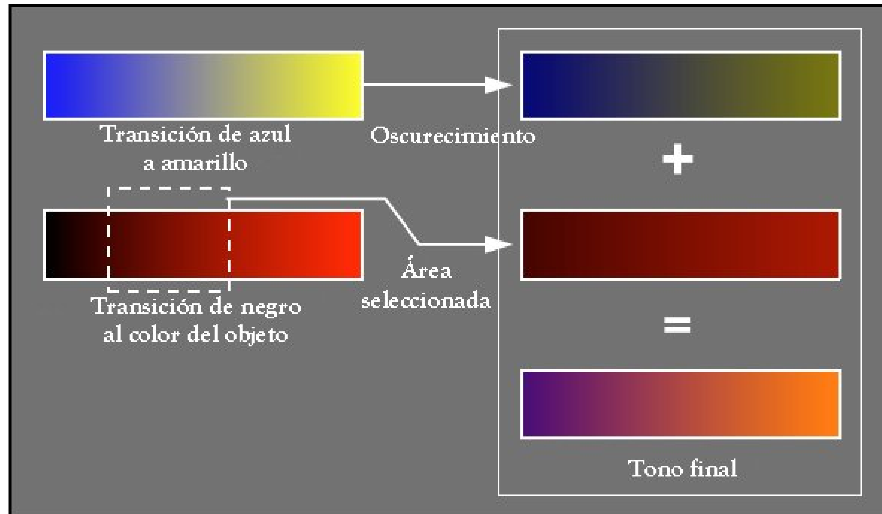
El caso de la iluminación y sombreado de Gooch, es el de imagen no foto-realista. Sus autores, Amy y Bruce Gooch, lo desarrollaron con miras a automatizar las llamadas ilustraciones técnicas. Ellos titulan por ilustración técnica a las imágenes que componen manuales e instructivos de equipo técnico, como puede ser el de un auto o un estéreo, por mencionar algunos. La Figura 56 muestra una imagen que puede considerarse dentro de este esquema.

Transición fría-cálida en una ilustración de tipo no fotorealista[118]
Figura56

En su artículo: "*A Non-Photorealistic Lighting Model For Automatic Technical Illustrations*" publicado en 1998, los hermanos Gooch proponen un modelo de iluminación que sea capaz de lograr tales efectos. Para ello, se basan en cambios de tonos, transiciones entre colores fríos y cálidos, características que fueron determinadas después de realizar observaciones sobre diferentes ilustraciones.

Mientras que el modelo de iluminación tradicional (iluminación difusa) se basa únicamente en la componente difusa k_d del material, el modelo de Gooch presenta algunas variantes al ser un sombreado con base en transiciones de tonos. Cuando un artista hace su obra, por ejemplo, en lápiz, éste genera un sombreado con base en tonalidades de gris, haciendo un barrido desde el blanco hasta el negro. Si se hace un dibujo a color y se le agregan grises, se logra lo que se conoce como *sombras* (en el caso de aplicar negro) y *tintes* (en el caso de blanco). Cuando una escala de color es creada mediante la añadidura de gris a cierto color, entonces dicha escala es conocida como tono. La Figura 57 representa lo aquí mencionado.

[118] Ilustración extraída del artículo: "*A Non-Photorealistic Lighting Model For Automatic Technical Illustrations*". Gooch, Amy, Bruce Gooch, Peter Shirley, Elaine Cohen. Universidad de UTAH.



Creación de tonos
Figura 57

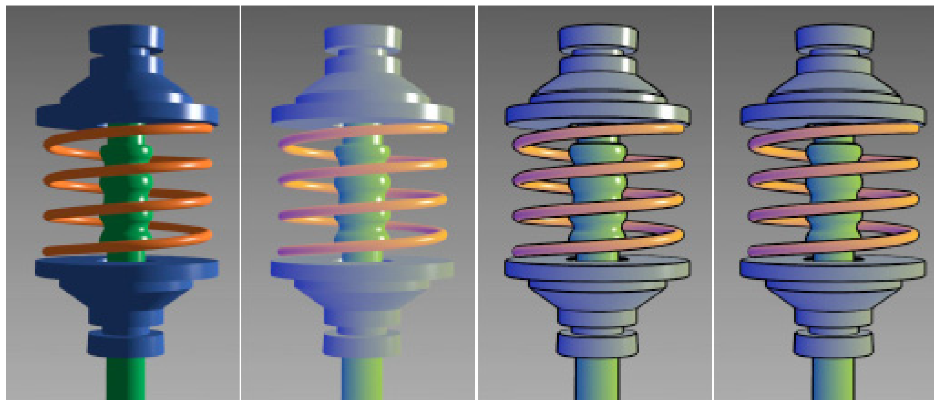
El modelo que presentan fue obtenido a partir de una generalización del modelo clásico de sombreado para gráficas por computadora, con el motivo de experimentar con tonos. Esto se hace mediante el término del coseno de $n \cdot l$ para mezclar dos colores RGB: k_{cool} y k_{warm}.

$$I = \left(\frac{1 + l \cdot n}{2}\right) k_{cool} + \left(1 - \frac{1 + l \cdot n}{2}\right) k_{warm}$$

Para simular tonos intermedios e involucrar la componente difusa del objeto, las constantes k_{cool} y k_{warm} pueden calcularse de la siguiente manera:

$$k_{cool} = k_{blue} + \alpha k_d$$
$$k_{warm} = k_{yellow} + \beta k_d$$

En el caso de estudio de Gooch, también se incluyó un *shader* para determinar los bordes y hacerlos visibles, así como los reflejos que puedan existir en el material. Para el caso del presente trabajo, solamente se ha implementado el sombreado, dejando a un lado la correspondiente detección de contorno y de brillos. Un ejemplo del acabado que se logra mediante este tipo se iluminación es el mostrado en la Figura 58.



Acabado logrado mediante sombreado de Gooch
Figura 58

Este efecto es implementado en dos partes: un *vertex shader* y un *píxel shader*. El *vertex shader* se encarga de hacer las correspondientes y usuales transformaciones con el vértice, la normal y el vector de iluminación y disponerlos en los registros de textura para su uso posterior en el *píxel shader*.

Primero analícese el caso del *vertex shader*. Es necesario colocar las matrices de transformación (Model·View y Model·View·Projection) en los registros constantes **c0** y **c4** respectivamente y proveer de la posición del vértice y normal en los registros de entrada **v0** y **v3**.

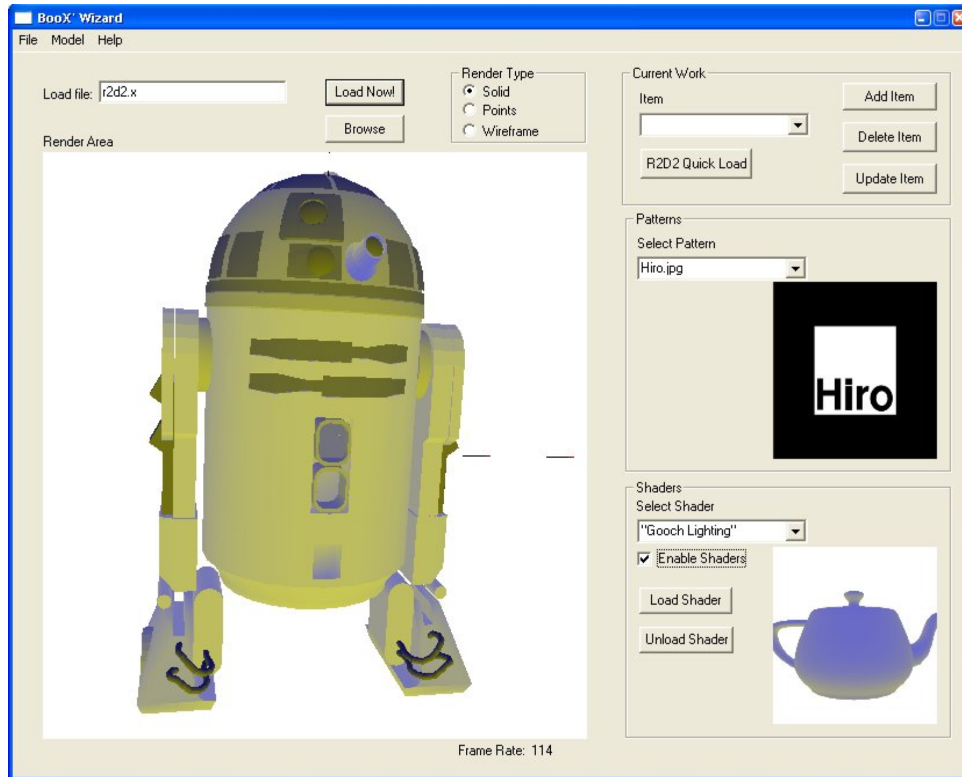
```
// Vertex Shader 1.1
vs.1.1
dcl_position    v0          // Vertex position
dcl_normal      v3          // Vertex normal
dcl_texcoord    v7          // Texture Coordinates
// Transform vertex into clip-space
m4x4   oPos, v0, c0
// compute the normal
mov     r0, v3
dp3     r0.w, r0, r0
rsq     r0.w, r0.w
mul     r0, r0, r0.w
// Compute light vector
sub     r1, c8, v0
dp3     r1.w, r1, r1
rsq     r1.w, r1.w
mul     r1, r1, r1.w
// Outputs
mov     oT0, r0             // Normal -> oT0
mov     oT1, r1             // Light   -> oT1
```

Es así que primero se transforma el vértice en espacio de recorte. Luego se renormaliza la normal al igual que el vector de dirección de la luz. La parte final tiene como objetivo colocar en los registros de textura, los datos que posteriormente serán usados por el *píxel shader.* Dicho *shader* se muestra a continuación.

```
// Pixel Shader 1.4
ps.1.4
texcrd    r0.xyz, t0              // Normal -> Register 0
texcrd    r1.xyz, t1              // Light  -> Register 1
// Compute n·l
dp3       r3, r0, r1
add_d2    r3, r3, c5              // clamp to [0, 1]
// Compute kc (Cool Factor)
mul_sat  r0, c4, c0
add_sat  r0, r0, c2
mul_sat  r0, r0, r3
// Compute kw (Warm Factor)
mul_sat  r1, c4, c1
add_sat  r1, r1, c3
mad_sat r0, r1, 1-r3, r0
```

Este shader tiene por requerimientos que los datos del valor de alfa, beta, azul y amarillo (necesarios para calcular las constantes k_{cool} y k_{warm}) sean colocados previamente en los registros constantes **c0**, **c1**, **c2** y **c3** respectivamente. Primero carga en registros temporales las componentes de la normal y del vector de dirección de la luz. Enseguida es efectuado el cálculo de $n \cdot l$ y acotado al rango [0, 1].

Finalmente, son calculados los factores k_{cool} y k_{warm}, multiplicados por su respectivo factor de iluminación y colocados en el registro de color **r0**. El resultado obtenido es el que se muestra a continuación (véase Figura 59).



Gooch Shading
Figura 59

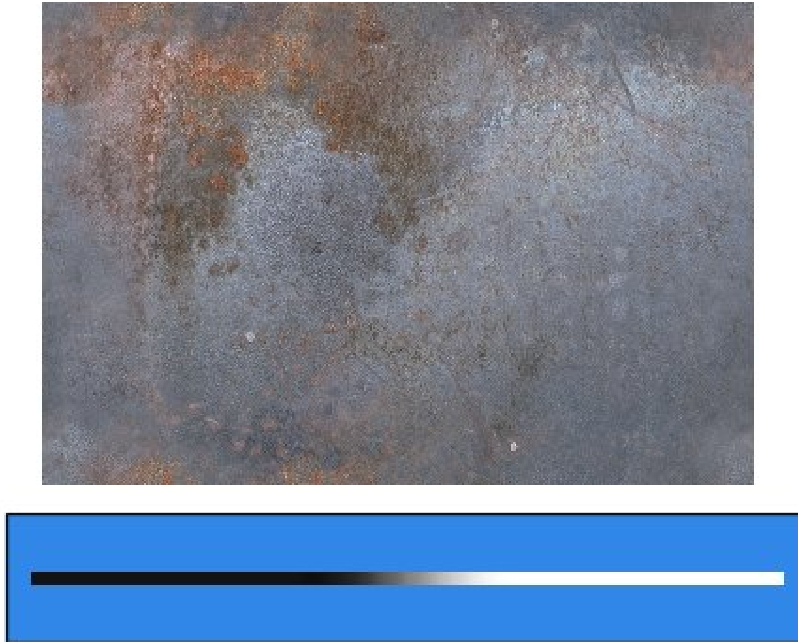
5.4.4 Transición entre dos texturas (*multitexturing*)

En ocasiones, es interesante ver efectos de transición entre materiales. Por ejemplo, en diversos juegos y películas se han visto objetos biomecánicos, los cuales, al perder un elemento constitutivo de su ser, dejan a la vista el otro u otros materiales por los que está compuesto. Es de está manera que es posible ver objetos que tengan una apariencia orgánica (como la piel) pero a su vez tener una composición de metal. Otro efecto de estos puede ser la transición que sufre determinado material, personaje, elemento de una escena al ser transformado por un poder mágico en piedra, el resultado: partes de piel, plástico, metal cubiertos por piedra, pero no la totalidad del mismo.

El *shader* propuesto a continuación tiene como finalidad mostrar un efecto parecido con base en el manejo de su textura base y una adicional del material compositivo. Para ello, se ha decidido implementar un efecto que represente una descomposición de un objeto cualquiera en metal oxidado, *shader* que permitirá simular partes de metal que se encuentran oxidadas.

Para lograr dicha meta, el método propuesto es el de combinar dos texturas: la base del objeto (la que indica su origen, su composición) y una adicional que es la de metal oxidado. Además de este par de texturas, se hará uso de una tercera que se utilizará como gradiente para determinar cual es el color del píxel dominante.

La Figura 60 muestra las dos últimas texturas usadas. La primer ilustra el material de transición, el metal oxidado. La segunda es el gradiente empleado, que no es más que una escala de color del blanco al negro puros.



Texturas necesarias para el *shader*: *Texture Transition*
Figura 60

A continuación se tratará la creación del *shader*, comenzando por algunos puntos necesarios a considerar. Primero, se debe recordar que los *píxel shaders* operan sobre cada píxel constitutivo de la escena, es decir, se aplican en una etapa posterior al del proceso de *raster*. Dentro de *Direct3D*, se cuenta con diversas etapas de *render*, cada una de ellas asociada a un registro de textura y temporal de la unidad de *píxel shader*[119]. La textura base del modelo se encuentra asociada por *default* a la etapa cero y por ende a los registros **r0** y **t0**. Por dicha situación, la textura de metal será asociada a la etapa 1, mientras que el gradiente a la etapa2. La porción de código mostrada a continuación, permite crear las texturas y hacer el mapeo con sus respectivas etapas de *render*.

```
ArbitraryTex = NULL;
Gradient = NULL;
D3DXCreateTextureFromFile(g_pd3dDevice, ARBTEX, &ArbitraryTex);
D3DXCreateTextureFromFile(g_pd3dDevice, GRADIENT, &Gradient);
g_pd3dDevice->SetTexture( 1, ArbitraryTex );  // Bind to Texture Stage 1
g_pd3dDevice->SetTexture( 2, Gradient );       // Bind to Texture Stage 2
```

Una vez especificado lo anterior, es conveniente explicar el código del *vertex shader*. Cómo en los anteriores efectos, este código permitirá efectuar las transformaciones necesarias así como el mapeo necesario en los registros de textura del *píxel shader* con los datos que sean necesarios. Los requerimientos para este *vertex shader* son los siguientes: matriz compuesta de transformación en espacio de recorte (transpuesta), vector de dirección de la luz, posición del vértice en cuestión, su vector normal y las coordenadas de textura del mismo.

```
// Vertex Shader 1.1
vs.1.1
// c0  - MVP Matrix                c4  - MV Matrix
// c8  - Light Dir
```

[119] Consúltese la sección 2.3.2.1 así como la Figura 14.

```
dcl_position    v0              // Vertex Position
dcl_normal      v3              // Vertex Normal
dcl_texcoord0   v7              // Texture coordinates
// Transform vertex
m4x4            oPos, v0, c0    // Transformed vertex
// Normalize normal
mov             r0, v3
dp3             r0.w, r0, r0
rsq             r0.w, r0.w
mul             oT1, r0, r0.w   // Normal renormalized
// Normalize light vector
mov             r1, c8
dp3             r1.w, r1, r1
rsq             r1.w, r1.w
mul             oT2, r1, r1.w   // Light renormalized
//Outputs
mov             oT0.xy, v7      // Tex coordinates
```

Como de costumbre, se transforma el vértice y se coloca en su registro de salida. También se renormalizan los vectores normal y de dirección de la luz para efectuar su sombreado. Dado que se usará un *píxel shader*, es necesario pasar en algunos registros dichos vectores normalizados, para que sea calculado el factor de iluminación. Finalmente, las coordenadas de textura son pasadas también en uno de los registros de textura.

Ahora es el turno del *píxel shader*. Este código es el que se encarga de lograr el efecto de transición entre dos materiales. Como primer paso, es necesario cargar los datos de color para operar con ellos. Para ello se emplea la instrucción **texld**, que permite cargar el color del píxel en cuestión de la textura asociada a la etapa *i* usando las coordenadas de textura definidas por el registro de textura *j*. Por ejemplo, la instrucción: **texld r0, t0** carga en r0 el color del píxel actual usando las coordenadas de textura que se han colocado en el registro de texturas 0. De igual manera, **texld r1, t0** y **texld r2, t0** cargan los colores del píxel actual (en **r1** y **r2** respectivamente) empleando las coordenadas de textura del registro **t0**. Mediante **texcrd** se cargan directamente los datos contenidos en los registros especificados (en este caso particular, **t1** y **t2**)

```
texld   r0, t0          // First Tex   -> Register 0
texld   r1, t0          // Second Tex  -> Register 1
texld   r2, t0          // Gradient    -> Register 2
texcrd  r3.xyz, t1      // Normal      -> Register 3
texcrd  r4.xyz, t2      // Light       -> Register 4
```

Una vez con los datos necesarios en los registros, es necesario calcular que tan intenso es el color de cada textura. Para ello se hace uso del gradiente. La textura que se usa como gradiente es un degradado de negro hacia blanco. En otras palabras, los datos (considerando el color como la combinación de valores flotantes RGB) se encuentran dentro del rango [0,1]. Por ello, es posible hacer una modulación del color con estos datos, de manera que el resultado final siga perteneciendo al mismo intervalo. Ahora, ¿por qué es necesario hacer esto?

Para crear una transición entre las dos texturas, se debe de saber cuando se pinta una, cuando la otra e, incluso, cuando se pintan ambas. Dependiendo de la configuración elegida será la transición. Para este caso, se ha creado una escala que únicamente en el centro tiene tonos de grises, los extremos son color puro ya sea blanco o negro.

El siguiente punto es, ¿cómo controlar el degradado de los colores? El gradiente es el que permite crear la transición, por lo tanto una de las otras texturas debe ser operada en una dirección mientras que la restante, en la dirección contraria.

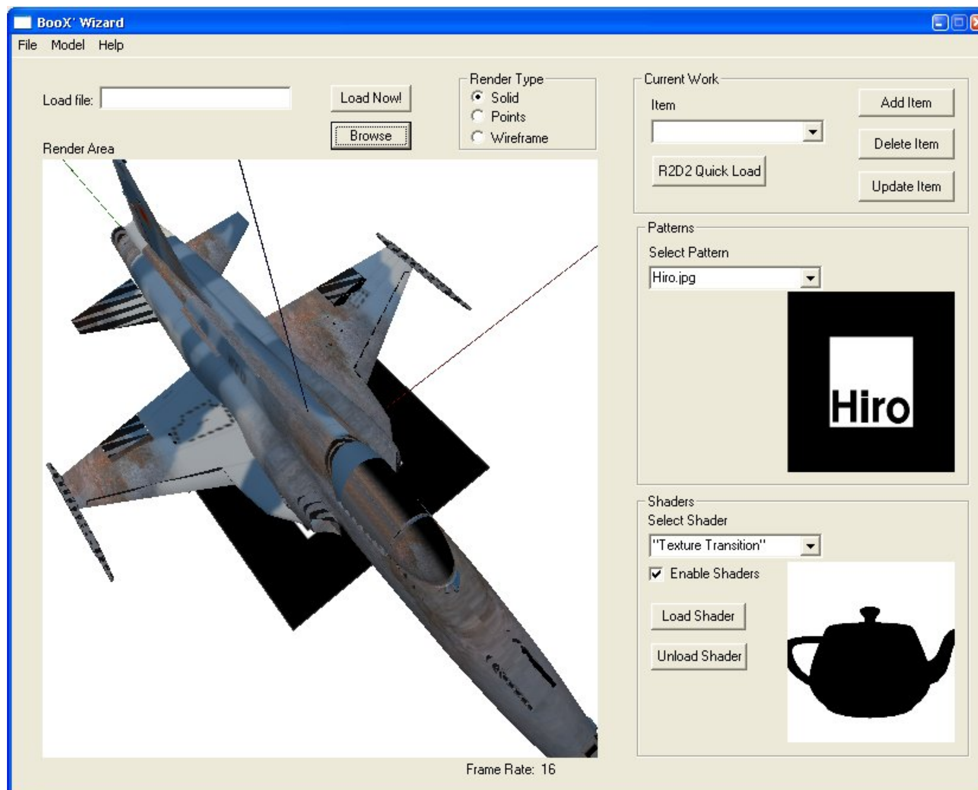
Dado que la escala del gradiente se encuentra en el intervalo [0, 1], basta con que se le reste 1 al valor actual de color de gradiente, de esa manera la dirección del mismo es invertida. Efectuando lo anterior, se puede observar que únicamente se pintara la textura en donde el gradiente es 1 y se tendrá una mezcla de ambas en donde es diferente de 0 y 1. Las siguientes instrucciones se encargan de realizar lo aquí comentado.

```
// Compute colors (Gradient)
mul        r0, r0, r2        // First Texture Color
mul        r1, r1, 1-r2      // Second Texture Color
```

Por ultimo, resta calcular el factor de iluminación del píxel en cuestión. El siguiente código calcula dicho factor, calcula el color del vértice (basado en la suma de los colores de las texturas), lo modula con el color de la luz y, finalmente, es atenuado con el factor de iluminación calculado.

```
// Compute n·l
dp3        r2, r3, r4
add_d2     r2, r2, c1        // Clamp to [0, 1]
// Compute final pixel color
mul        r2, r2, c0
add        r0, r0, r1
mul        r0, r0, r2
```

El resultado que se obtiene mediante este shader es el mostrado en la Figura 61. Es importante mencionar que para que se obtenga el efecto correcto, el modelo debe de tener coordenadas de textura y una textura mapeada, de lo contrario, se tendrán resultados no deseados. Además, si se desea cambiar de degradado, basta con especificar otra textura (diferente a la de metal oxidado) y se obtendrá el mismo resultado: una transición del material compositivo del objeto al especificado.



Texture Transition
Figura 61

Resumen

En este capítulo se ha tratado con profundidad el tema referente a los *shaders*. Se ha explicado la definición de uno de ellos, que comienza con la especificación del formato de vértice, la creación del *vertex buffer*, su llenado, asociación a un *vertex stream* y culminando con las definiciones de la interfaz para declaraciones de vértices y creación de los recursos del(os) *shader*(s) en cuestión.

Es importante recordar que la elección de la interfaz para declaraciones de vértices está ligada y determinará el correcto funcionamiento del *shader* en cuestión, ya que en ésta se encuentra toda la información que indicará cuales son las propiedades que poseen los vértices, así como los *vertex streams* y registros asociados a cada una de ellas.

Otro punto importante en la preparación de la aplicación, es la comprobación de soporte en hardware para shaders. Esto es un paso importante, ya que de ello depende evitar *bugs*, errores o incluso el paro de la ejecución del programa. El soporte de *píxel shaders* es el más complejo de controlar, ya que se puede tener el mismo efecto programado para diferentes versiones y sin embargo no funcionar correctamente a causa de que no existe soporte en hardware para dicha versión.

Se han descrito brevemente los lenguajes existentes para programar *shaders*. En particular, para la versión de *DirectX* se cuenta con la especificación mediante lenguaje ensamblador y el lenguaje de alto nivel (HLSL). El modo de operación a nivel de hardware es el mismo, la diferencia se tiene en la capa del programador. A medida que aumentan en complejidad los efectos que se deseen programar, la tarea de definirlo en lenguaje ensamblador aumenta también. Es en dichos casos cuando es conveniente hacer uso del HLSL puesto que otorga mas capacidad de abstracción y por lo tanto, le facilita un poco el trabajo al desarrollador.

La sección final de este capítulo, incluye la implementación de los *shaders*. Fueron desarrollados cuatro y de ellos, el que hace más uso de instrucciones es el correspondiente al efecto *Texture Transition*. Dicho *shader* tiene la particularidad de emplear un *píxel shader* que evalúa el color que poseerá el píxel en cuestión. En su especificación se puede observar como se manejan las coordenadas de textura de un modelo para extraer la información del color de la textura y operarlo como cualquier tipo de dato. En el efecto *Gooch Lighting* también se emplean registros de textura, pero los datos contenidos en ellos son escritos mediante un *vertex shader*, es decir, mediante dicho ejemplo se ilustra la manera de relacionar *vertex* y *píxel shaders* para lograr determinado fin.

Grande es el poder que ofrecen los *shaders*, pero para hacer buen uso de ellos es necesario tener conocimientos firmes sobre las matemáticas que involucran la implementación de un modelo de iluminación, transformaciones geométricas, animación de mallas, etc. Emplear el HLSL dota al programador con nuevas posibilidades al ser un lenguaje de alto nivel en el que no se debe de preocupar tanto por el acomodo a nivel bajo, sino en mayoría de la algorítmica involucrada en sus efectos.

Pruebas
al sistema

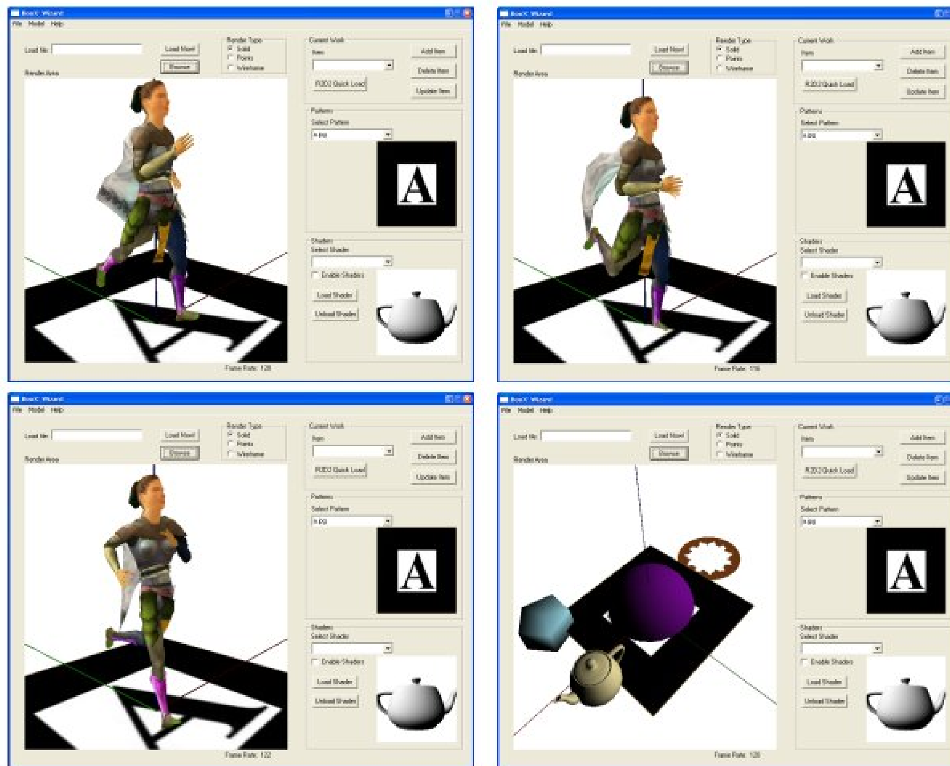
Para verificar el funcionamiento de las aplicaciones creadas (*BooX Wizard* y *MagicBook*), se han evaluado las siguientes características en el editor y el visualizador:

- Carga de ambos tipos de modelos soportados, *X* y *Cal3D*
- Cambio del tipo de *render* entre los modos *wireframe*, *solid* y *points*
- Selección y dibujo mediante los *shaders* desarrollados (*Crystal Shader*, *Gooch Lighting*, *Simple Transformations* y *Texture Transition*)
- Selección de patrones
- Creación de un archivo de configuración para el *MagicBook* (libro)
- Despliegue de los modelos de acuerdo al libro cargado
- Uso del *shader* asociado a cada modelo en caso de haberse hecho
- Posición y *tracking* del objeto de acuerdo a la posición que guarde la cámara

A continuación se abordará cada una de estas características, comenzando con los relativos al editor *BooX Wizard* y se mostrarán imágenes que ejemplifiquen el punto en evaluación.

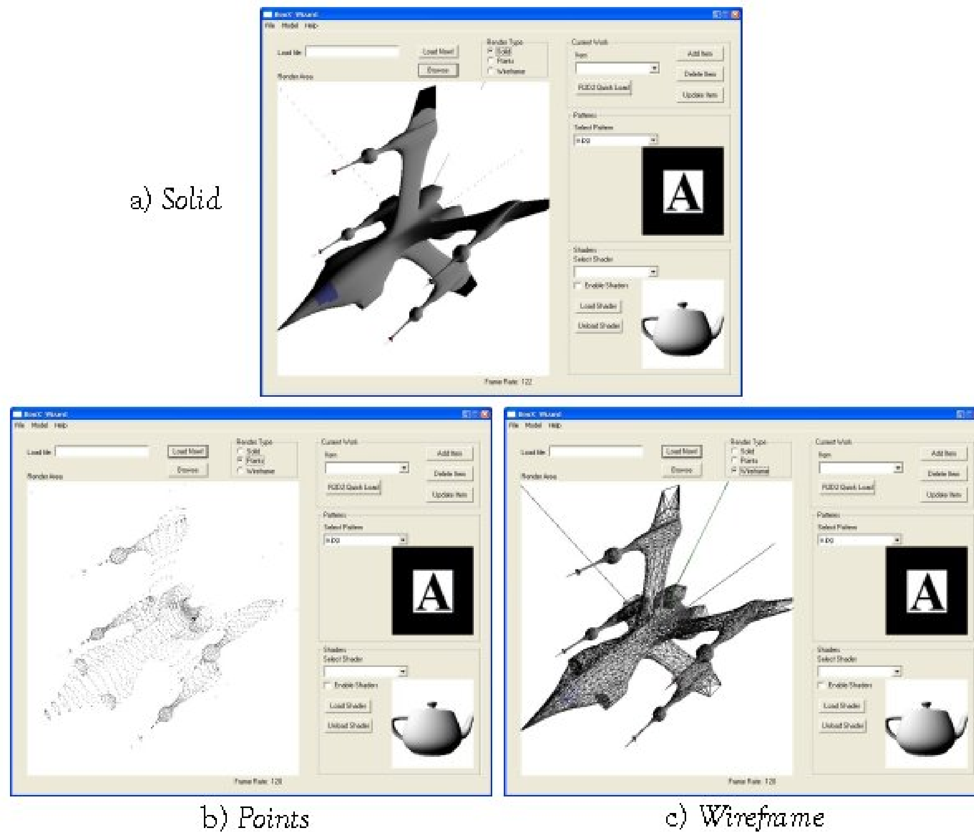
Soporte de modelos y tipo de *render*

El soporte de modelos *X* y *Cal3D* se muestra en la siguiente imagen (Figura 62). Los archivos de *Cal3D* son cargados mediante un archivo de configuración que especifica las partes a cargar (mallas, animaciones, materiales, etc.) con extensión *.cfg*, mientras que los *X* contienen extensión *.x*. Además, en la figura se han agregado tres cuadros para ilustrar la animación del personaje.



Archivos *X* y *Cal3D*
Figura 62

La Figura 63 muestra el cambio de dibujo de la escena según la elección hecha en el control "*Render Type*". En ella se pueden observar tres entornos con el mismo modelo: una nave en los tres estados propuestos (*solid*, *points* y *wireframe*). Mediante estas opciones, es posible observar la densidad de polígonos que contiene el modelo y decidir si es conveniente usarlo dentro del *MagicBook*.



Tipo de *render*
Figura 63

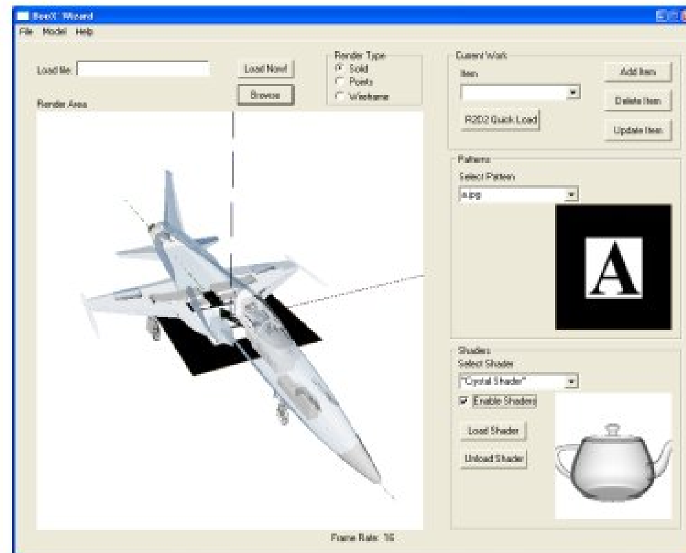
Uso de *shaders* y selección de patrones

La verificación del dibujo mediante *shaders* viene dado en la siguiente serie de figuras (Figuras 65.a, 65.b, 65.c y 65.d). En ellas se efectúo una prueba de cargar el modelo de un avión F5e y aplicarle los efectos desarrollados (véase Capítulo V, sección 5.4). El resultado es mostrado a continuación.

- *Crystal Shader*

  El efecto de la Figura 64.a, llamado *Crystal Shader*[120], permite volver transparente al modelo actual, dejando a la vista todos los elementos compositivos del mismo. Es de utilidad cuando se requiere visualizar objetos que contengan piezas dentro de ellos, o bien elementos complejos como un cráneo por ejemplo[121]. En el caso del cráneo, si se trata de un objeto detallado que cuenta con dientes y estructura ósea basada en una anatomía verdadera, sirve incluso de radiografía del mismo, pues dejará a la vista los dientes incluyendo su raíz. Este efecto detecta los bordes del objeto y modifica la componente alfa de los vértices. Mediante la prueba de canal alfa, algunos son descartados y otros son dibujados semi-transparentes o sólidos, presentando el color del material o textura que posean. Los vértices dibujados son aquellos cuya normal es casi o totalmente perpendicular a la cámara. En la figura se puede observar como el avión se vuelve transparente, dejando a la vista los elementos por los que se encuentra construido (como lo es la cabina del piloto), además se puede apreciar como se delinean los bordes el dibujo es de un color azul, propio de la textura que tiene mapeada el modelo.

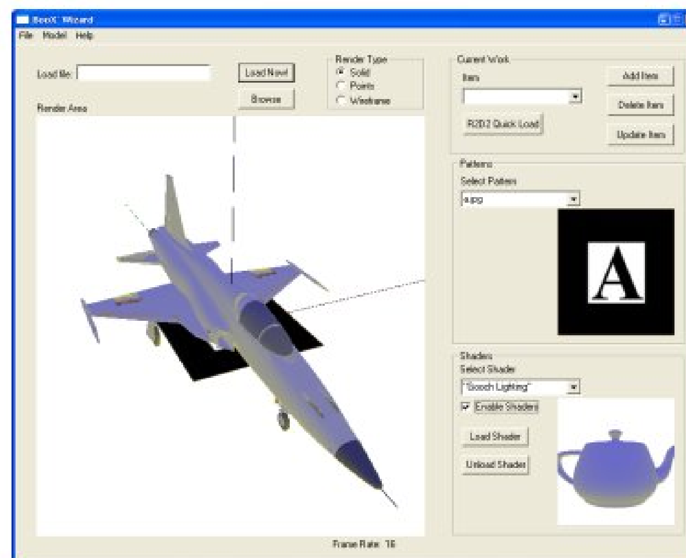
[120] Refiérase a la sección 5.4.2 del capítulo V para una descripción completa del efecto.
[121] Véase la figura 55 para observar al cráneo citado.



Crystal Shader
Figura 64.a

- *Gooch Lighting*

  El caso de iluminación de Gooch[122] (Figura 64.b) es diferente. Dicha iluminación fue concebida con fines de acabado no foto-realista, tratando de imitar al sombreado que producen los ilustradores de libros técnicos o manuales, en donde la figura en lugar de ser sombreada, es coloreada con tonos cálidos y fríos, generando una impresión visual característica del tipo de libros mencionado. El modelo de iluminación es similar al empleado por la iluminación difusa, su cambio principal radica en usar la banda tonal entre los colores definidos como cálido y frío. La transición entre las tonalidades se puede observa principalmente en la cola del avión, donde se observan las transiciones entre el morado (tonalidad fría) y el amarillo (tonalidad cálida).

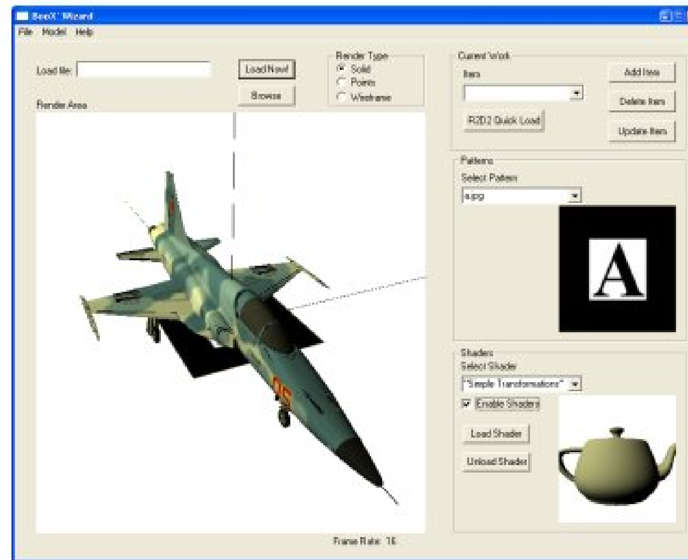


Iluminación Gooch
Figura 64.b

[122] En la sección 5.4.3 se trata con mayor detalle el sistema de iluminación de Gooch.

- *Simple Transformations*

  Este efecto es un sustituto del *pipeline* por *default* de *Direct3D* en donde lo único que se hace es mostrar el modelo y aplicarle un sombreado mediante iluminación difusa. Dicho modelo es evaluado por un *vertex shader* que toma los vértices y los transforma en espacio de recorte. En seguida se realiza el cálculo de la iluminación de cada vértice, tomando la normal asociada y el vector de luz (normalizados) para después modular dicho factor con el color del vértice en cuestión.

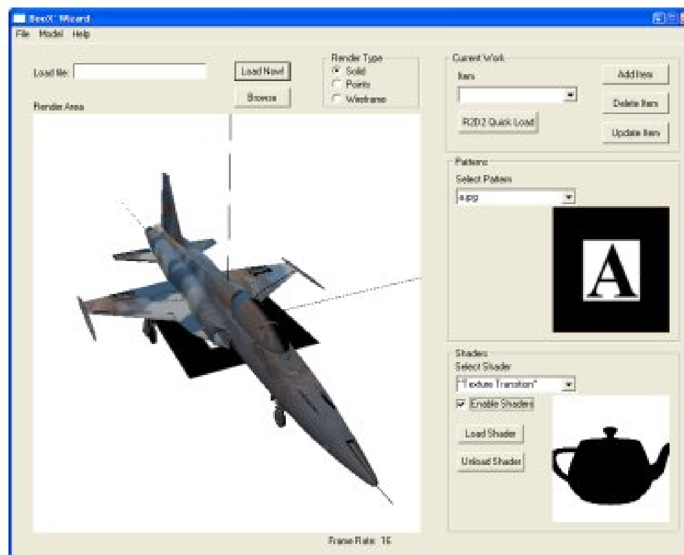


Transformaciones usuales
Figura 64.c

- *Texture Transition*

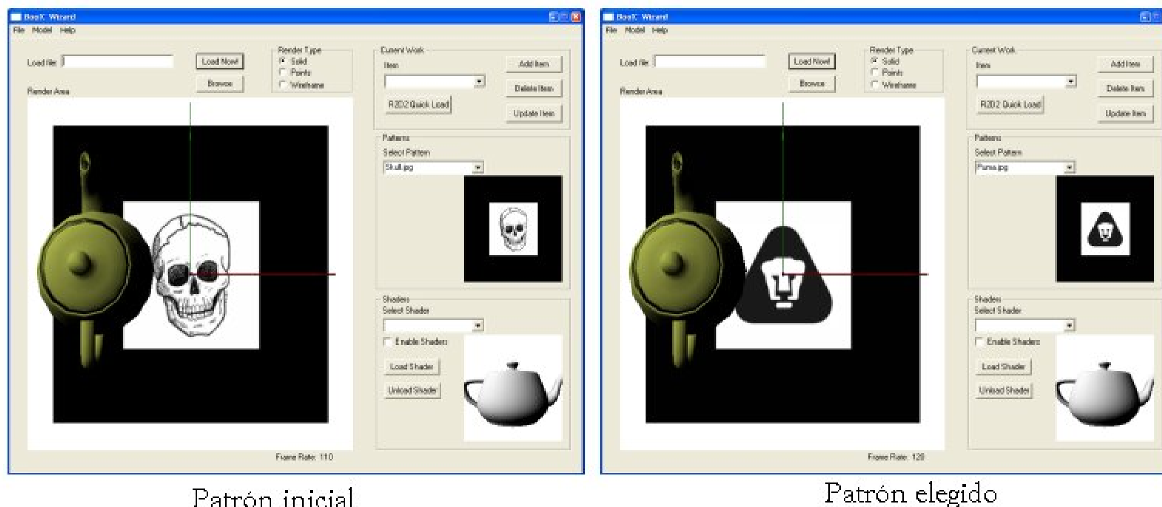
  El caso de este shader es un poco más elaborado y pensado para dibujar objetos con dos texturas, una base y otra a manera de efecto secundario. El caso presentado es de oxidación, por lo que el modelo del avión funciona bien. De esta manera, el *shader* mezcla ambas texturas (la base y la de óxido) con base en un gradiente (provisto a manera de imagen en blanco y negro) generando el resultado mostrado. Cabe mencionar que todos los cálculos involucrados (exceptuando los de iluminación) son realizados a partir del color proveniente de las coordenadas de textura provistas por el modelo. La operación realizada para determinar cual es el color final de píxel se logra mezclando los colores provenientes de las texturas base y secundaria, previamente multiplicadas por su gradiente (valores entre [0, 1]). El gradiente es invertido para la segunda textura. Mediante tales acciones, lo que se tiene son dos texturas a ser sumadas, pero cada una de ellas se comporta como el gradiente por el que fue multiplicado, es decir que donde la textura gradiente tenía blanco, la textura en cuestión poseerá su color original, en donde haya niveles de gris el color será opacado, mientras que en donde se tiene negro no existirá color[123]. El resultado es el mostrad a continuación, en donde se pueden observar zonas que contienen la textura original y zonas que tienen la textura de óxido, rindiendo una apariencia de metal oxidado al fuselaje del avión.

[123] El funcionamiento de este *shader* se explica en la sección 5.4.4



Transición entre dos texturas
Figura 64.d

La selección de patrones está controlada por el *combo box* "*Select Pattern*", mismo que contiene la lista de nombres de los patrones encontrados en el directorio por *default* (carpeta *Patterns* dentro del directorio del *MagicBook*). Cuando se selecciona alguno, tanto el patrón del área de presentación preliminar como el del área principal de *render* son actualizados. La Figura 65 ilustra lo anterior.



Selección de patrones
Figura 65

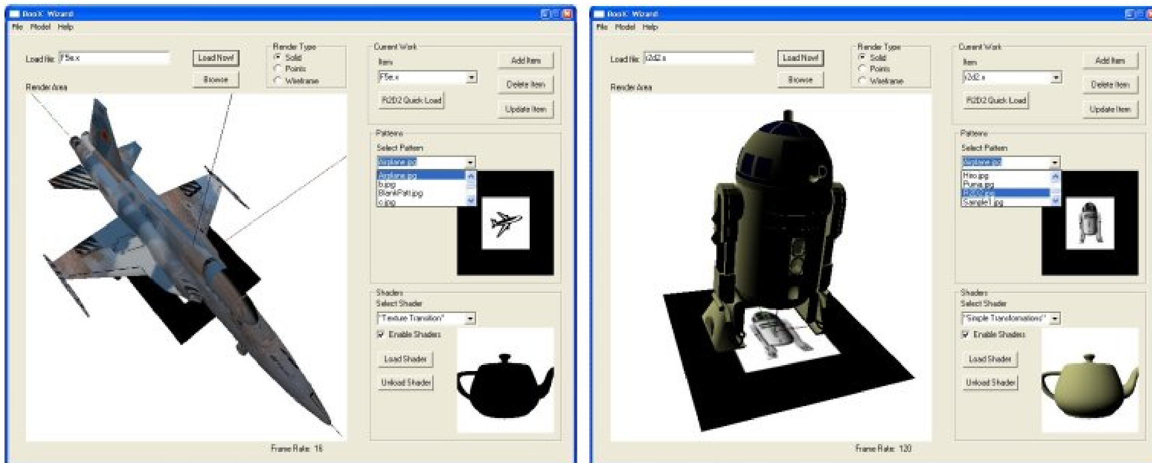
Creación de un libro

En este punto se tratarán los diversos aspectos involucrados en la elaboración de un libro, donde las imágenes ejemplificarán a cada una de las 'páginas' creadas, es decir, mostrarán al modelo, el patrón asociado y el *shader* seleccionado.

El libro constará de los siguientes modelos, *shaders* y patrones:

1. Avión del ejercito F5e (F5e.x) con *shader* de transición entre texturas para dar efecto de oxidación (*Texture transition*). El patrón asociado será aquel que contenga un avión en el área blanca.

2. Modelo de R2D2 de *Star Wars* con *shader* sustituto del *pipeline* por *default* de *Direct3D* (*Simple Transformtations*), Patrón del mismo personaje: R2D2.

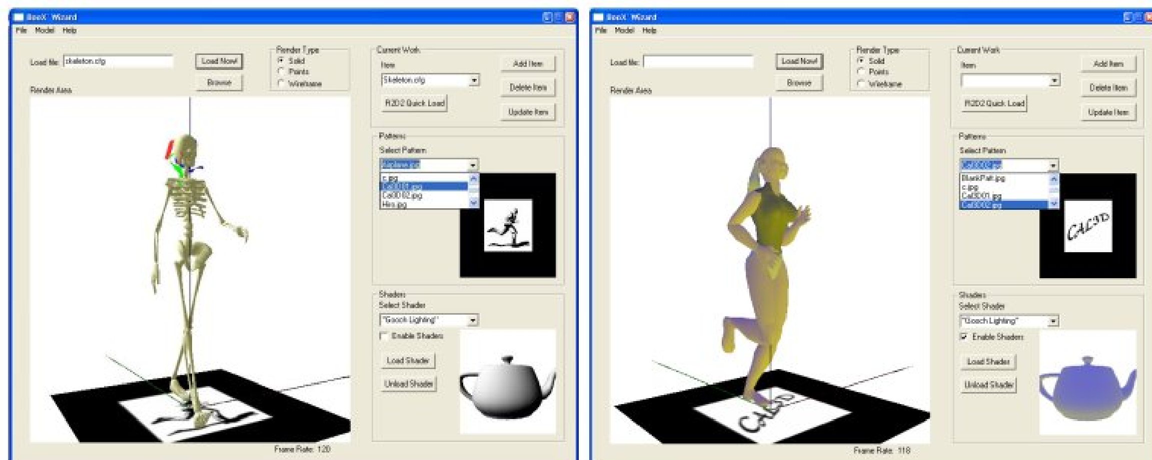


a) Avión militar F5e con *shader* de oxidación

b) R2D2 con *shader* sustituto del *pipeline* por *default*

Avión militar F5e y R2D2
Figura 66.a y 66.b

3. Esqueleto en formato de *Cal3D* con animación de caminar. No tiene ningún *shader* asignado y su patrón corresponde a la figura de una persona corriendo.
4. Modelo *Cally* con animación de correr. Se dibujará con sombreado Gooch (*Gooch Lighting*) y el patrón es la leyenda "*Cal3D*".

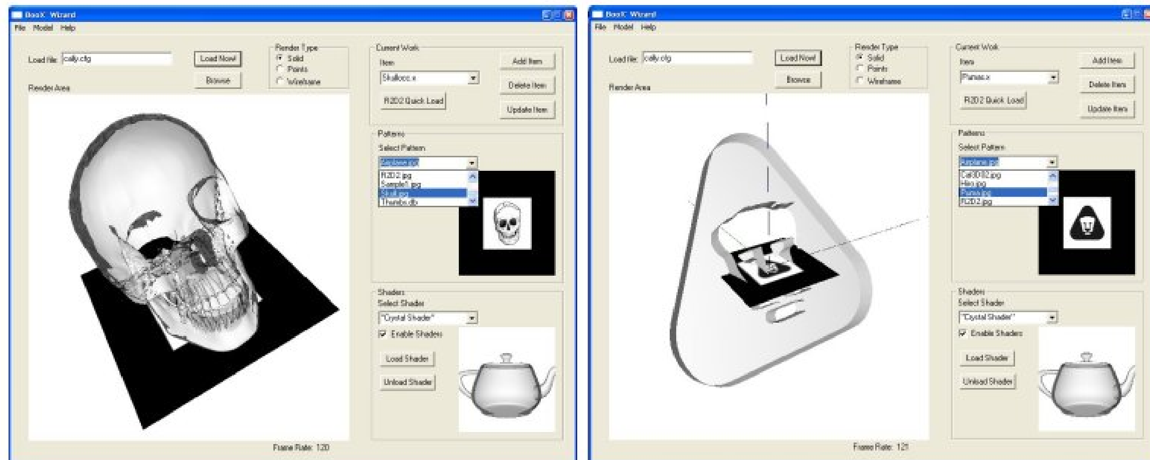


c) Esqueleto de *Cal3D*

d) Modelo *Cally*

Modelos de Cal3D
Figura 66.c y 66.d

5. Cráneo en formato *X*. Se utilizará para demostrar el efecto de tipo radiografía que se logra mediante el efecto *Crystal Shader*. El patrón asignado es el correspondiente a una calavera.
6. Escudo deportivo de los PUMAS de la UNAM. Se utilizará el mismo *shader* que dota a los objetos de una apariencia cristalina con motivo de mostrar que efecto es totalmente transparente. El patrón asociado es el logotipo de un puma.



e) Cráneo con apariencia de radiografía

f) Escudo deportivo de la UNAM en acabado de cristal

Shader con acabado de cristal
Figura 66.e y 66.f

Una vez cargados los modelos y guardado el libro, el siguiente paso es cargar el trabajo en el visualizador de *MagicBooks*.

Carga de un libro para el *MagicBook*

La Figura 67 ilustra la interfaz que permite seleccionar y cargar un libro previamente creado. Con el proceso anterior se cargará el libro realizado y se mostrará una ventana mostrando el mundo real que es adquirido en tiempo real.

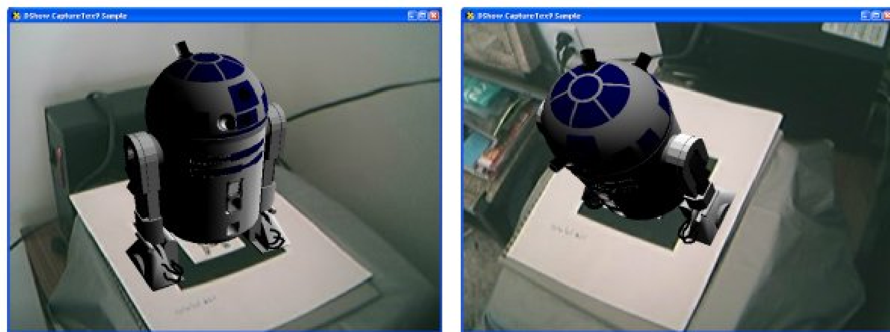
Interfaz para selección de libro
Figura 67

Cuando se observa con la cámara cada uno de los patrones, se muestra a cada uno de los modelos. Comenzando por el avión militar y el R2D2 se tiene que (Figura 68.a y 68.b):

a) Avión F5e

Avión militar F5e
Figura 68.a

El avión militar se dibuja correctamente ya que aparece sobre el patrón ubicado en la misma posición y aproximadamente con la misma escala que en el editor (Figura 66.a). El *shader* especificado es aplicado tal y como se observó en el editor. Además, el *tracking* responde de manera adecuada puesto que el modelo conserva su alineación y posición respecto de la orientación del patrón con la cámara. Uno de los problemas que se presentó en la alineación es que el patrón elegido es pequeño y casi simétrico, tal simetría origina que el avión se puede orientar en sentido contrario.



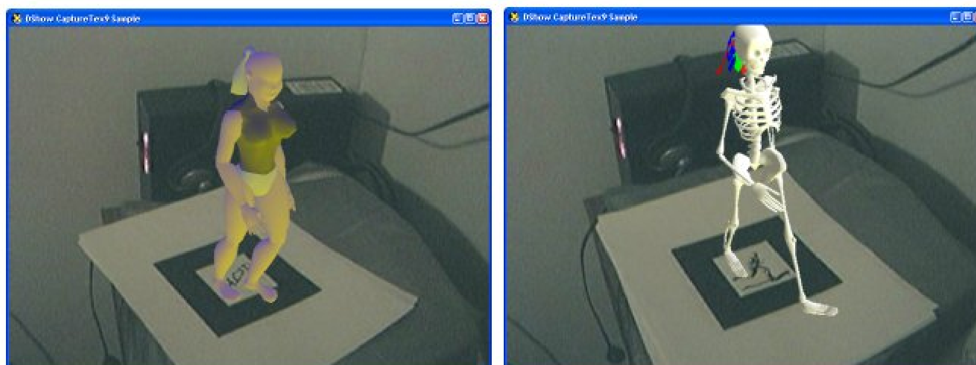
b) R2D2

Despliegue del R2D2
Figura 68.b

Por parte del R2D2, éste no se dibuja exactamente igual a como lo hizo el editor (Figura 66.b), esto se debe a que el editor tiene definida una luz adicional para dicho *shader*. En una próxima versión del MagicBook, se puede incluir especificación y soporte para más de una luz (en el archivo de configuración) y de esa manera producir el mismo o mejores efectos. A pesar de lo mencionado, no existe problema mayor, ya que el *shader* se está empleando y se están obteniendo los resultados esperados. El *tracking* del modelo es bueno, sin embargo existen problemas de reconocimiento asociados a la figura del mismo. El dibujo del R2D2 empleado posee muchos niveles de grises, lo cual repercute en que muchos de esos grises se vuelvan negros cuando las rutinas del *ARToolKit* hacen la conversión a imagen blanco y negro (al aplicar el *threshold*). La solución es hacer una figura menos elaborada y únicamente en lbanco y negro, sin grises.

En las Figuras 68.c y 68.d se observan los modelos *Cal3D* cargados, con la animación en ejecución y con el *shader* usado. El modelo *Cally* es quien presenta el *shader* de iluminación Gooch, tal y como puede ser observado en la imagen correspondiente. En el reconocimiento de estos dos modelos es donde se tuvieron los mayores problemas. Primero, el modelo del hombre corriendo no es muy claro y presenta muchas partes con color negro, por lo cual al momento de hacer la

conversión a blanco y negro se presentan mediciones erróneas. En el caso de la leyenda *"Cal3D"* del modelo *Cally*, el problema es causado por el tipo de letra seleccionado y el tamaño de las mismas. Entre mas cercanas estén unas a otras, mayor será el error en que se incurra.

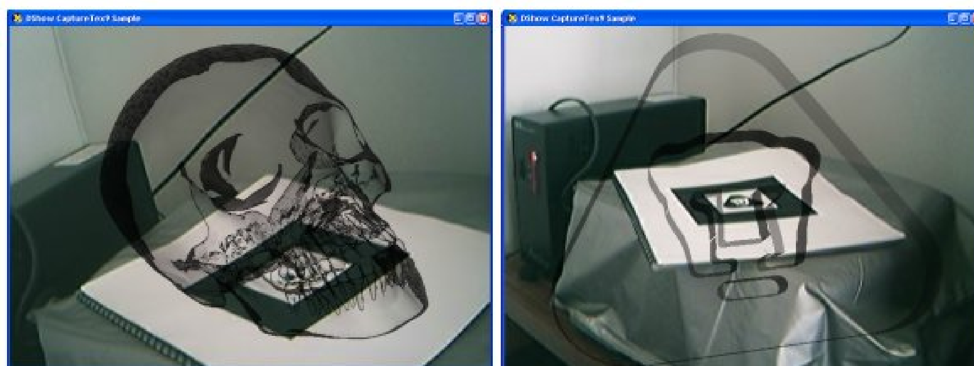


c) Cally con sombreado Gooch d) Esqueleto en formato *Cal3D*

Modelos de *Cal3D* desplegándose en el *MagicBook*
Figura 68.c y 68.d

Por último, se presentan las imágenes correspondientes al *shader* de cristal. Los objetos que fueron cargados son el cráneo (compuesto por un gran número de polígonos) y el escudo deportivo de los PUMAS de la UNAM. Comentando un poco acerca del cráneo, mediante esta toma se puede observar como deja al descubierto la composición interior del mismo, provocando el efecto de una radiografía, tal y como se describió en la sección 5.4.2 del Capítulo V.

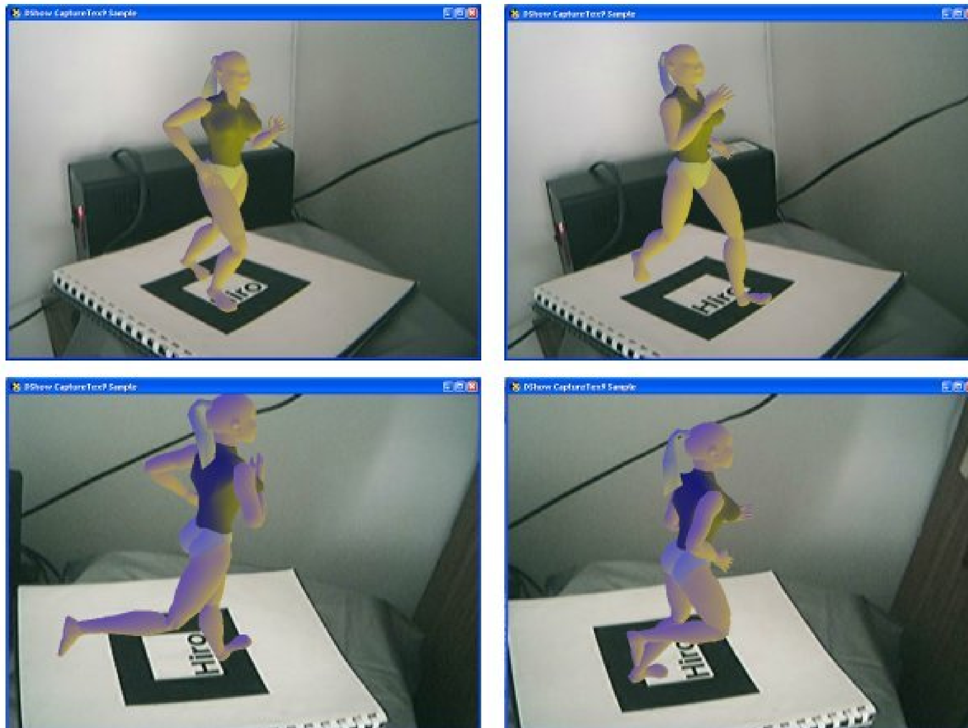
En el caso del logotipo de los PUMAS, cabe mencionar que la transparencia a través del mismo es notoria, abriendo la posibilidad de crear diferentes objetos como vasos, ventanas, vitrales, etc. Sin embargo, la calidad de transparencia que se logra con este *shader*, es total. Es decir, para poder simular con verosimilitud al cristal, vidrio, plástico, etc. se deben de tomar en cuenta otros factores como son la distorsión que es causada por la refracción, brillos por la luz, entre otras.



e) Cráneo con apariencia de radiografía f) Escudo de PUMAS en apariencia de cristal

Efecto de cristal en el MagicBook
Figura 68.e y 68.f

La figura 69 se presenta para mostrar el comportamiento del *tracking* y el desarrollo de la animación de los modelos de *Cal3D*. En ella, se puede observar al modelo Cally corriendo y que esta animación no se pierde cuando se hacen movimientos de la cámara. Además, es posible observar como conserva su orientación y posición respecto del patrón que lo representa.



Animación y *tracking* en modelos *Cal3D*
Figura 69

Después de describir y analizar el funcionamiento de las aplicaciones *BooX Wizard* y *MagicBook vD3D*, se puede concluir que las pruebas realizadas han sido satisfactorias. Se ha generado un libro mediante el editor, especificando los modelos, efectos y patrones requeridos. El mismo ha sido cargado en el visualizador y se ha comportado de la manera esperada, cargando todos los modelos sin omitir alguno y dibujando con los *shaders* especificados. A su vez, se han comentado los problemas encontrados en el reconocimiento de patrones, los cuales deben de cumplir con el requerimiento de ser muy simples en sus figuras y, preferentemente, en blanco y negro. Se ha observado que al incorporar imágenes en blanco y negro, pero con tonos de grises, el reconocimiento se ha visto empobrecido, entregando malos resultados que van desde el nulo despliegue de modelos hasta confundir con algún otro.

Respecto del *tracking*, se puede decir que la aplicación se comporta de manera adecuada. Existen algunos problemas debidos a la calibración de la cámara que causaban traslaciones y rotaciones en sentido o incluso en direcciones incorrectas, mismas que procuraron ser corregidas mediante las modificaciones realizadas[124]. Pero, del *tracking* que se tenía sin las modificaciones, al que se obtuvo después, es realmente notable la diferencia. Ahora el mayor problema en el que se incurre, son una serie de traslaciones alrededor del patrón, pero nada comparable con los iniciales relacionadas a la rotación, ubicación y traslaciones que se presentaban.

Con los programas desarrollados y analizando el desempeño de *DirectX*, se puede decir que se ha logrado un mejor rendimiento de aplicación (cualitativamente hablando), ya que el video se muestra con mayor fluidez y la calidad de objetos (como el avión y el cráneo que tienen una alta densidad de polígonos) es más alta en comparación con los modelos de *VRML*. Además de ello, durante la implementación de

[124] Consúltese la sección 4.5 para mayor información

shaders, es posible especificarlos mediante los dos métodos existentes (ensamblador y HLSL) puesto que se trata de aplicaciones nativas de *Direct3D*.

Cabe mencionar que la posición y el tracking en el MagicBook se evalúan de manera cualitativa ya que no existen parámetro numéricos con los cuales hacer una comparación. Las rutinas del *ARToolKit* mantienen completamente envuelto al proceso involucrado en el *tracking* de los objeto virtuales, no entregan información certera sobre la relación entre los mundos virtual y real.

En un sentido estricto, se podría obtener la relación mencionada pero ello involucraría la introducción de un elemento virtual con dimensiones específicas (con unidades ya sea en [*mm*] o [*cm*], por ejemplo). Mediante dicho elemento se tendrían que efectuar diversas mediciones en el mundo real y virtual, para encontrar cuales son los parámetros que pudieran estar incorrectos. Lo anterior provocaría que la aplicación se comportara adecuadamente pero sigue siendo una forma de corrección, para ello es mejor desarrollar una calibración seria y con bases firmes para la cámara y trabajar con configuraciones correctas. Lo anterior es una mejora propuesta para ser desarrollada como trabajo a futuro.

Conclusiones

A lo largo de este trabajo se ha presentado el desarrollo de dos aplicaciones: el editor *BooX Wizard* y el visualizador *MagicBook* en su versión de *Direct3D*. La primera consiste en una interfaz 3D que permite al usuario visualizar modelos tridimensionales y posteriormente incluirlos en un ambiente de realidad aumentada. El visualizador es quien crea dicho ambiente y donde se observa el contenido creado mediante el editor.

Hablando primero del editor, se puede decir que cubre las necesidades planteadas en un inicio como son:

- Carga de diversos formatos de modelos (se soportan *X* y *Cal3D*)
- Dibujo de la escena mediante *shaders* programables
- Despliegue del modelo colocado respecto del patrón, tal y como se presentaría en el *MagicBook*
- Selección y despliegue de patrones
- Posibilidad de guardar trabajos pendientes y volverlos a cargar para continuar con la realización del libro
- Generación de un archivo de configuración para el *MagicBook*

De esta manera, el trabajo de creación y edición de un libro se lleva a niveles de seleccionar, cargar, aplicar y guardar. Una interfaz que disminuye en gran medida las tareas implicadas en el desarrollo de contenido para la versión anterior de *OpenGL*. A su vez, es una herramienta útil cuando se necesita visualizar objetos en 3D y que estén en alguno de los formatos soportados.

Algunos problemas que se presentan son referentes al soporte en hardware de *shaders*, ya que se pudo haber creado un libro en una máquina donde la verificación era aprobatoria pero, al llevarlo a un hardware diferente y sin soporte, dado que el proceso de carga es independiente a la verificación, podrían presentarse problemas en el dibujo. Estos problemas podrían presentarse como la nula aparición de objetos, dibujarse en negro o bien, problemas asociados con las normales o el *culling* de los polígonos, aparentando verse hacia adentro.

El visualizador creado satisface también las expectativas hechas antes de su desarrollo. Carga libros creados mediante el editor y los muestra en una ventana de tamaño modificable. Además, la característica principal que se requería era la de mostrar objetos usando *shaders* programables, la cual es cubierta en su totalidad para los efectos creados en este trabajo.

Como contratiempos y problemas que se tuvieron en el desarrollo del *MagicBook* (versión de *Direct3D*) se tiene la poca y mala documentación referente a las bibliotecas de programación del *ARToolKit*. Por tal motivo, no se tienen los mismos resultados de su antecesor en cuanto al *tracking* de los objetos, de hecho, hubo que analizar gran cantidad de código de las rutinas y mensajes en el foro de discusión para comprender donde pudiera presentarse el error. Después de dichas experiencias, fue posible efectuar ciertas modificaciones que mejoraron en gran medida el *tracking*, dejando que fuese similar al de la versión original.

Cabe mencionar que no se da soporte en ninguna de las aplicaciones para modelos de *VRML*, mismos que eran usados en la versión de *OpenGL*. Para cubrir el rubro referente a modelos con animación, se incluyó soporte para el formato *Cal3D*. Dicho formato presenta la característica de animar con base en esqueletos, es decir, una estructura jerárquica con la que pueden ser construidos modelos más complejos; principalmente humanoides, animales y otros modelos que puedan ser definidos mediante una jerarquía de eslabones y articulaciones.

En términos de rendimiento y de manera cualitativa, el visualizador basado en *Direct3D* y *DirectShow* presenta un mayor número de cuadros por segundo que su antecesor. Solo basta con observar las animaciones de los modelos de *Cal3D* o algún modelo en formato *X* y mover la cámara para tener diferentes ángulos de la escena en cuestión para notar como tanto el *tracking* como la animación son mas fluidos.

En el caso del editor, está provisto de un indicador de número de cuadros por segundo que se dibujan y, ejecutándose la aplicación en una computadora con una tarjeta GeForce 3 Ti200 se logran cifras de hasta 120 fps[125]. Dicha cifra es real para los modelos *Cal3D* empleados y algunos *X* que tienen una cantidad considerable de polígonos. En el caso de modelos que contienen demasiadas texturas, cargarlas repercute en el rendimiento de la aplicación, disminuyendo el número de cuadros por segundo.

Ambas aplicaciones pueden ser aumentadas y mejorarlas. Entre ellas se pueden considerar las siguientes:

- Soporte de otros formatos populares para modelos 3D como el OBJ y el 3DS por mencionar algunos
- Despliegue de animaciones contenidas en los archivos *X* para tener otra posibilidad de modelos animados
- Carga de más de un modelo en el editor y de diferentes tipos (combinaciones de los soportados)
- En caso de que exista mas de un modelo dentro de la escena, implementar detección de colisiones entre ellos para controlar animaciones

En específico, para el visualizador una mejora bastante útil consiste en cambiar el reconocimiento de patrones de manera que se desplieguen modelos sin necesidad de que se encuentre el total del cuadro negro en el área de visión de la cámara. Esto porque en ocasiones se perdía apenas una arista del cuadrado y de inmediato al objeto también. Asimismo, otro cambio que es conveniente es realizar una calibración mas seria de la cámara, con el fin de evitar configuraciones y comportamientos malos del software de reconocimiento y posterior *tracking*.

[125] FPS: Frames per second. Número de cuadros por segundo que son mostrados.

Apéndice A
Uso de *Cal3D*

Cal3D es un conjunto de bibliotecas de programación que permiten utilizar modelos 3D en aplicaciones de graficas por computadora. Dichos modelos tienen la particularidad de contener animaciones de buena calidad, ya que son basadas en esqueleto, lo que permite generar casi cualquier modelo que contenga una estructura jerárquica con huesos, dígase humanoides, animales, objetos inorgánicos como mangueras, etc.

El paquete es código libre y los principales colaboradores del proyecto son: Bruno Heidelberger (*beosil*), Justin Hare (*jahare*) y Laurent Desmecht (*maxun*), todos miembros de la comunidad de desarrolladores de *SourceForge*[126]. Ha sido escrito de manera que sea independiente de plataforma y de API gráfica.

Se puede considerar que el paquete está compuesto por dos partes: las bibliotecas de programación, escritas en C++ y el exportador. El exportador permite transformar el modelo (previamente creado en algún software de diseño) en los diferentes archivos que especifican al formato *Cal3D*. Las bibliotecas de programación proveen al programador de la aplicación, de las funcionalidades necesarias para cargar los archivos exportados, construir el modelo, ejecutar animaciones y acceder a los datos necesarios para ser dibujados en el API gráfica elegida.

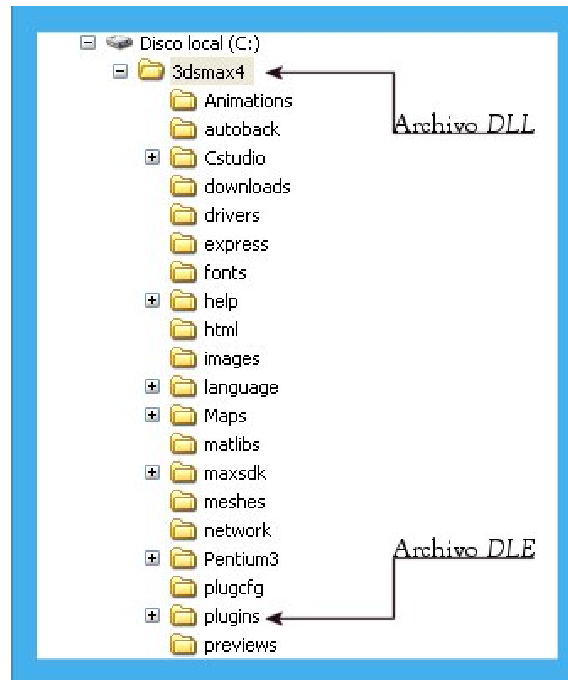
Cuando se descargan las bibliotecas, se debe descargar también el exportador. Dicho *plug-in* está disponible para dos paquetes de diseño gráfico: *3DStudio Max* (versiones 3.1, 4.2 y 5.x) y *MilkShape3D*. El primer exportador es el mas empleado, esto debido a la gran expansión que tiene dicho software en el ámbito del diseño. A nivel amateur, *3DStudio Max* también es popular, ya que el empleo de sus herramientas no es tan complicado como el de otros paquetes, tales como *Maya* de *Alias-Wavefront*.

Es importante mencionar que el hecho de que el exportador este disponible para *3DStudio Max* y no para otros paquetes de diseño, no repercute en la utilización de los mismos para crear los modelos. Es decir, se pueden emplear otros programas de diseño como *Maya*, *LightWave*, *Rhinoceros*, etc. para producir los modelos 3D y posteriormente exportarlos al formato 3DS (propio de *discreet* para *3DStudio*). Dicho formato permite al usuario importar el modelo creado dentro del ambiente de edición de *3DStudio Max* y, por lo tanto, usar el exportador de *Cal3D* para generar el correspondiente modelo y montarse en alguna aplicación que haga uso de dichas bibliotecas de programación.

A.1 Preparación del sistema

Una vez comentado qué es, cual es la utilidad y el *software* de diseño compatible con las bibliotecas de programación, es turno de tratar los aspectos previos al uso del exportador. El paquete puede descargarse del sitio web de *Cal3D*, ya sean los binarios o el código fuente para ser compilado. El exportador consta de 2 archivos: el *DLL* de *Cal3D* y el *plug-in* para el software de diseño. En este caso, se tratará con el exportador para *3DStudio Max* (*plug-ins* con extensión *DLE*). El *DLL* debe de colocarse en la carpeta de sistema, es decir, la ruta de instalación de *3DStudio Max*, mientras que el *DLE* se colocará en la carpeta *plugins*. La Figura A.1 ilustra la estructura de directorios donde deben ser colocados ambos componentes.

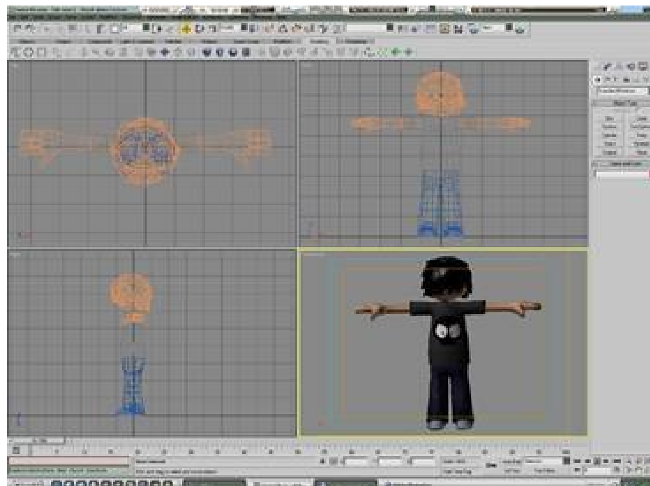
[126] *SourceForge* es una comunidad de internet para desarrolladores de aplicaciones. Todas las aplicaciones y código encontrado en dicha comunidad están bajo la licencia GPL o bien, algunas parecidas. El sitio correspondiente es: http://sourceforge.net. El paquete *Cal3D* se puede encontrar a su vez en: http://sourceforge.net/projects/cal3d.



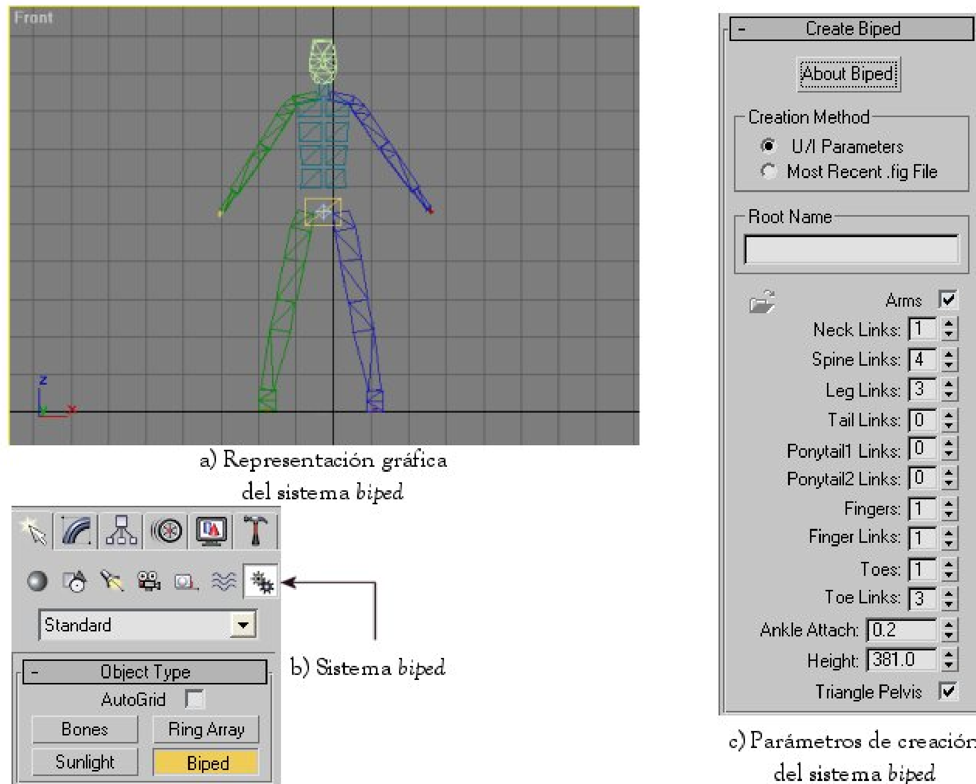
Estructura de directorios de *3DStudio Max*
Figura A.1

A.2 Creación del sistema *biped*

El siguiente paso es tener el modelo listo para ser animado. Para ello, se considerará que se tiene uno hecho y se procederá únicamente a animarlo. *3DStudio Max* contiene ciertas herramientas y modificadores que permiten crear huesos. Sin lugar a dudas, una de ellas es la que contiene el llamado *Character Studio*, que ya tiene un modelo de tipo humanoide con esqueleto y, por ende, estructura jerárquica necesaria para la animación. Dicha herramienta, es el modificador de *physique* para el sistema *biped*. Este último sistema (del cual provee al diseñador el *Character Studio*) es el esqueleto que servirá para crear las animaciones y que será adjuntado a la malla del modelo que se vaya a emplear. Las siguiente figura (Figura A.2 y A.3) muestra un modelo de ejemplo, así como los menús en donde se encuentran el sistema *biped* y el modificador *physique*.



Personaje que será exportado al formato *Cal3D*
Figura A.2



a) Representación gráfica del sistema *biped*

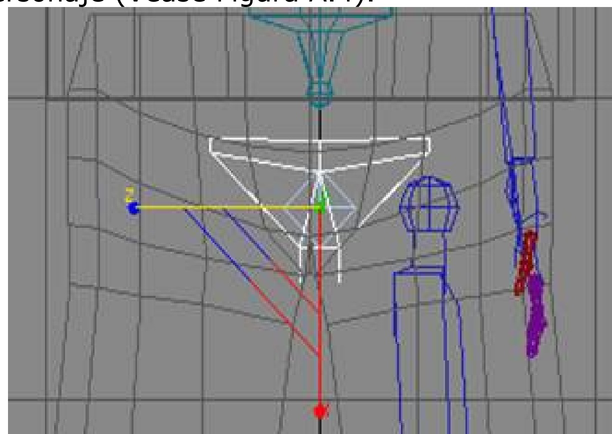
b) Sistema *biped*

c) Parámetros de creación del sistema *biped*

Sistema *biped* en *3DStudio Max*
Figura A.3

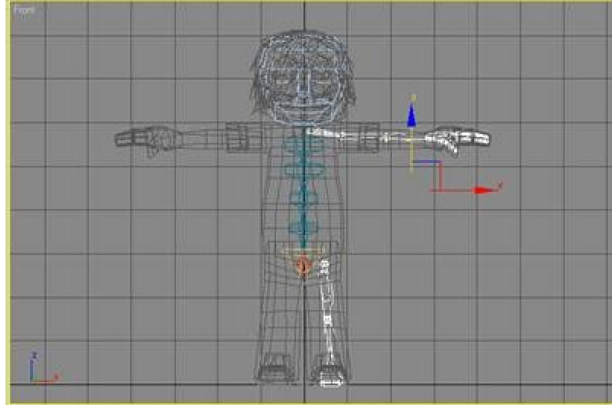
Dentro del menú de sistemas (botón con icono de engranes en la Figura A.3.b) se encuentra el deseado: *biped*. Al seleccionarlo, se expandirá una lista con los parámetros de creación del mismo, como son el número de eslabones que conformaran la espina dorsal, el cuello, número de dedos e, incluso, si es que se desea que tenga cola así como el número de extremidades que la crearán (Figura A.3.c).

El esqueleto ha sido creado de acuerdo a las características que se desea que presente y es turno de asociarlo a la malla que animará. Por lo tanto, se debe de hacer que el *biped* corresponda a la estructura fisiológica del personaje. Para ello es necesario seleccionar el rombo ubicado en la pelvis del *biped*. Este objeto representa en centro de masa del esqueleto, por lo que debe ser colocado exactamente arriba de la entrepierna del personaje (Véase Figura A.4).



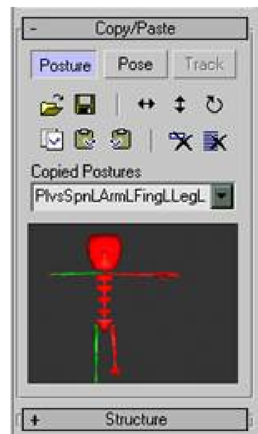
Colocación del centro de masa acorde a la pelvis del personaje
Figura A.4

Se continua ajustando el esqueleto, ahora sus extremidades. Es conveniente seleccionar la mano, el brazo, la clavícula, la pierna y el pie izquierdos y acomodarlos de manera correcta. Posteriormente, la pose y escala de las extremidades derechas serán copiadas a partir de sus simétricas. A continuación, se escalan cada una de estas extremidades, la cabeza, el cuello, los hombros, el tronco y la pelvis, de manera que se encuentren perfectamente en el interior del personaje (Figura A.5).



Lado izquierdo de *biped* escalado y colocado acorde a las dimensiones del personaje
Figura A.5

Con el lado izquierdo terminado y gracias al *Character Studio 4*, es posible emplear una herramienta que permite copiar la pose de manera simétrica. Dicha opción se encuentra en el menú *Copiar/Pegar* bajo el botón de postura (*Posture*). Se debe elegir copiar postura (*Copy Posture*) y presionar el botón de *Copy Posture Opposite*, lo que modificará el esqueleto del biped, haciendo que su parte derecha sea simétrica con la izquierda (Figura A.6). Ahora el esqueleto está listo para que se comience a delimitar las zonas que serán afectadas por el modificador de física *Physique*.

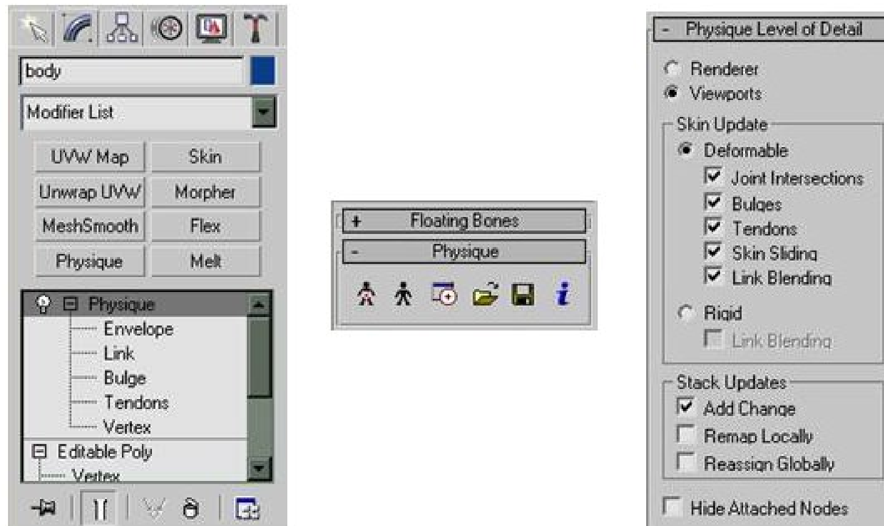


Botones de copiado y pegado para el *biped*
Figura A.6

A.3 Modificador *Physique*

Una vez terminado de escalar el esqueleto de manera que quepa dentro del cuerpo del modelo en cuestión, es necesario aplicar el modificador *Physique*, que permitirá asociar los movimientos del esqueleto con la malla que le circunda.

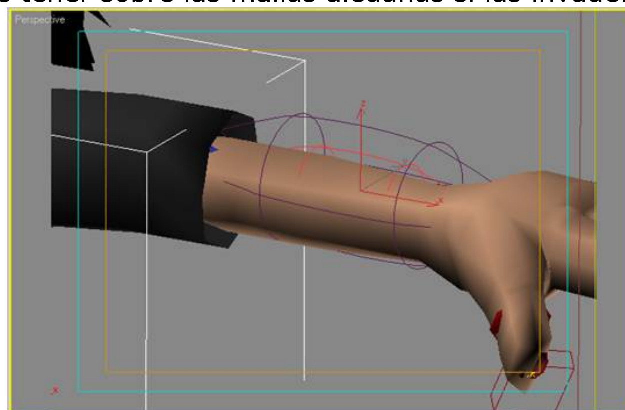
El modelo seleccionado está compuesto por lo que podría llamarse varias sub-mallas, es decir, la pierna izquierda es una sola geometría al igual que la mano y brazo derechos, etc. y por lo tanto, a cada una de ellas debe ser aplicado el modificador de *Physique*. Para ello se debe de seleccionar la pestaña de modificadores y seleccionarlo de la lista emergente. La figura A.7 ilustra las opciones del elemento en cuestión.



Opciones del modificador Physique
Figura A.7

Tal y como se mencionó anteriormente, a cada parte compositiva del modelo se debe de añadir un modificador *physique*. Después de hacerlo, es necesario seleccionar que se ligue al nodo raíz (*Link to Root Node*) y seleccionar el centro de masa del esqueleto. Cuando cada una de las sub-mallas cuenta con su modificador es necesario determinar su envolvente, que es la superficie de acción sobre la que el modificador tendrá efecto. Esto es, cuando se mueva un hueso del esqueleto, todos los vértices que se encuentren dentro de la envolvente serán modificados de acuerdo con la transformación hecha sobre el hueso.

Lo anterior se logra oprimiendo el botón de envolvente (*Envelope*) que mostrará una serie de líneas a manera de balón que corresponden a la superficie. Como puede observarse en la Figura A.8, la envolvente deber ser un poco mas grande. Es por ello que se emplea la herramienta de escalamiento radial (*Radial Scale*) que permitirá aumentar el radio de la envolvente. Un problema común cuando se efectúa esto, es la influencia que puede tener sobre las mallas aledañas si las invade.



Envolvente de un brazo del personaje
Figura A.8

El proceso se repite para cada circunferencia que compone la envolvente hasta que rodea por completo todos los vértices de la malla. En caso de que se hayan invadido mallas contiguas, es necesario eliminar los vértices afectados del eslabón que se está trabajando. Lo anterior se realiza mediante la selección de vértices (*Vertex Sub-Object Mode*) y la opción de *Remove from Link*.

El proceso anterior es repetido tantas veces como sub-mallas tenga el personaje. Es necesario efectuar ciertos intentos de animación para ver como se desenvuelven las articulaciones y verificar que no haya vértices dejados atrás o influenciados por algún hueso que no sea el correspondiente. Cuando se ha completado el proceso, entonces es hora de hacer la animación, trabajo que es dejado al diseñador del personaje y que no se tratará aquí, debido a las múltiples técnicas con que se cuenta para animar un modelo.

A.4 Exportando el personaje

Una de las partes más importantes durante la creación de un modelo para *Cal3D* es el *plug-in* que permite exportar el personaje a un formato en el que los programadores de aplicaciones gráficas puedan usarlo. Este proceso requiere de mucha atención, así que es conveniente seguir al pie de la letra las recomendaciones y pasos aquí señalados.

Antes de intentar exportar cualquier parte del modelo, se debe de modificar las texturas de acuerdo a las especificaciones de materiales y texturas de *Cal3D*. La correspondencia entre material del personaje y material de 3D Studio debe ser 1-1 y deben de ser llamados apropiadamente con un número o etiqueta comenzado en 0. Por ejemplo:

- **Pants[0]**
- **Shirt[1]**
- **Face[2]**
- **Arms[3]**
- **Shoes[4]**

Hecho lo anterior, el modelo está listo para ser exportado. Primero es necesario exportar el esqueleto. Para ello se debe de seleccionar el centro de masa del *biped* y que el mismo se encuentre en *Figura Mode* (sin animación). Posteriormente, seguir la ruta: *File->Export*, seleccionar: *Cal3D Skeleton Exporter* y elegir un nombre de archivo. De esta manera, aparecerá un cuadro de diálogo que mostrará todos los huesos contenidos dentro del *biped*.

El esqueleto es la pieza clave en el resto de la exportación. Ahora se puede exportar los materiales, mallas y animaciones en el orden que se quiera seleccionando el exportador adecuado. Cuando se haga lo anterior, el cuadro de diálogo preguntará por el archivo de esqueleto a utilizar, para lo que se deberá indicar el esqueleto previamente exportado.

Una nota importante para las animaciones, es asegurarse que el personaje no se encuentre en *Figure Mode*, ya que en caso afirmativo, la animación sería de solo 1 cuadro, permaneciendo el modelo estático. Para los materiales es conveniente remarcar que, si se cambia alguno de los empleados, estos deben ser exportados nuevamente al igual que las mallas que lo(s) ocupan.

Una vez que el esqueleto, materiales, mallas y animaciones han sido exportadas, es tiempo de crear el archivo de configuración para el modelo. Un sistema

adoptado dentro del *MiniViewer*[127] y del desarrollo de este trabajo es el de los archivos .cfg. Dichos archivos tienen la característica de ser de texto y son especificados de la siguiente manera:

```
# Cal3d cfg File

# Model's Scale
scale=0.5

# Skeleton
skeleton=Jason_skeleton.csf

# Meshes
mesh=arms_and_hands.cmf
mesh=body.cmf
mesh=Jason_head.cmf
mesh=shoes.cmf
mesh=slingshot.cmf
mesh=slingshot_band.cmf

#Animations
animation=jason_walk.caf
animation_strut=jason_strut.caf
animation_running=Jason_run.caf

#Materials
material=pants.crf
material=shirt.crf
material=jason_face.crf
material=brown_hair.crf
material=jason_arms.crf
material=puma_shoes.crf
```

La lista de materiales debe ser escrita en el orden en que fueron marcados, esto es, el material marcado con [0] es el primero en la lista, el siguiente es el marcado con [1] y así sucesivamente. El resultado final puede ser observado en el miniviewer provisto en el sitio web del proyecto, o bien en el editor de este trabajo (XLoader).

***KeyFrames* del personaje**
Figura A.9

[127] *Cal3D MiniViewer* es una aplicación creada por el equipo de desarrollo de *Cal3D* para permitir una rápida visualización del modelo recién creado.

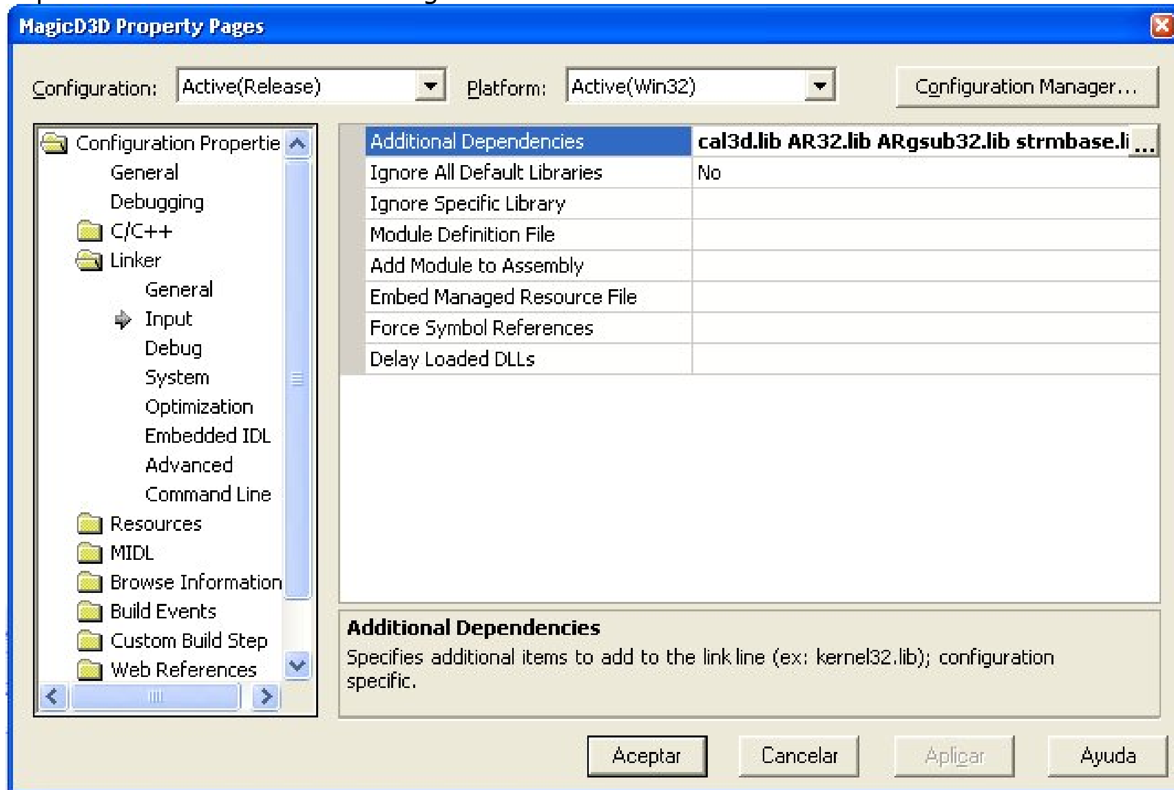
Apéndice B
Uso del *ARToolKit*

El uso de las bibliotecas de programación del ARToolKit requiere de ciertas características durante la creación, ligado y escritura del proyecto. En las siguientes líneas se presenta un esquema general de desarrollo de aplicaciones mediante el uso de las bibliotecas. Cabe mencionar que se tratará únicamente el aspecto de uso general, dejando la inicialización del entorno gráfico en manos del programador.

B.1 Creación del proyecto

La aplicación creada durante el presente trabajo fue desarrollada empleando el entorno de programación de *Visual Studio .NET*. Se trata de una aplicación para *Windows* de 32-bits que se comenzó desde cero, es decir, una aplicación nativa de *Windows* y nada más.

Por la razón anterior, la elección de configuración para compilación, ligado y uso de bibliotecas y archivos externos de programación fue efectuada por el programador. Es necesario explicar, principalmente, las directivas de ligado de esta aplicación en específico. Para ello véase la Figura B.1.

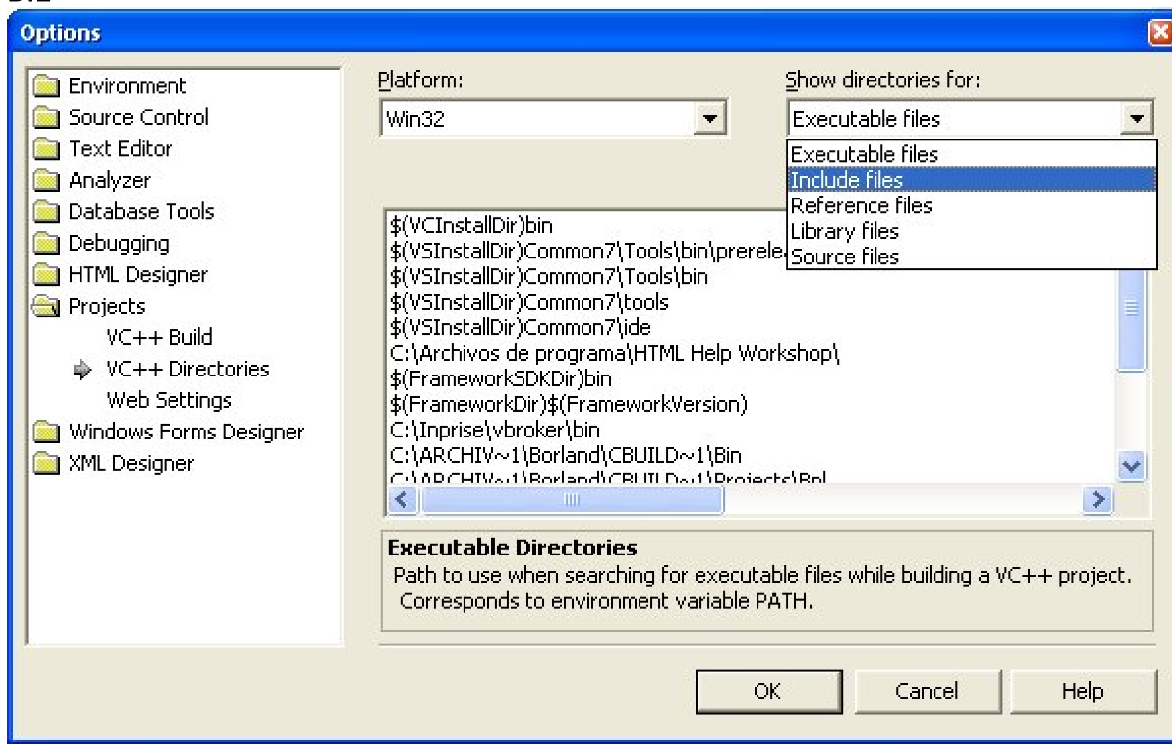


Propiedades del proyecto
Figura B.1

El cuadro de diálogo presenta las propiedades del proyecto en uso, mismo en el que deben ser especificadas las bibliotecas estáticas de programación adicionales. El campo "*Additional dependencies*" contiene los nombres de las bibliotecas con que se ligará al proyecto, paso primordial para la correcta compilación del mismo.

Cuando se emplea el *ARToolKit* tal como se ha hecho en el presente trabajo, es necesario indicar dos bibliotecas: *AR32.lib* y *Argsub32.lib*. La primera se encarga de inicializar lo que se puede considerar el motor de reconocimiento de patrones, mientras que la segunda lo hace con una serie de variables globales que tienen que ver con el ambiente gráfico y las matrices de transformación correspondiente.

Una vez indicadas los archivos extras de ligado, es necesario agregar a los directorios de trabajo del Visual Studio, las rutas donde se encuentran los encabezados y las bibliotecas de programación (archivos . h y .lib). Para ello obsérvese La Figura B.2



Directorios de encabezados y bibliotecas para Visual C++
Figura B.2

La opción "*Projects*" muestra las rutas en donde Visual Studio busca por *default* los encabezados, bibliotecas estáticas, ejecutables, código, etc. que sean especificados dentro de un proyecto. De está manera, seleccionando "*Include files*" primero y "*Library files*" después, se podrá indicar las rutas de los archivos necesarios del *ARToolKit*. Finalizados estos pasos, el proyecto está listo para comenzar a ser escrito.

B.2 Esquema general para el desarrollo de aplicaciones mediante *ARToolKit*

Las rutinas de programación de este grupo de herramientas, se podrían clasificar de la siguiente manera:

- Reconocimiento de patrones (*AR32.lib*)
- Inicialización y control del entorno gráfico (*ARgsub32.lib*)
- Soporte para el uso de modelos en formato *VRML* (*ARvrml.lib*, *libvrml97*.lib*)
- Inicialización, captura y procesamiento de video (*VideoWin32.lib*)

Como en el caso de este trabajo, cuando solamente se emplea el reconocimiento e inicialización de variables globales, el uso de las rutinas es mínimo, ya que la parte compleja de control de video y dibujo en 3D es dejada al programador. Tal motivo es el que provoca que únicamente se usen 9 funciones del conjunto disponible y que tiene por misión:

- Leer y cargar en memoria los datos de calibración de la cámara, *arParamLoad()*

- Adoptar los parámetros cargados de acuerdo con el tamaño de la ventana de aplicación, *arParamChangeSize()*
- Inicializar las estructuras de datos necesitadas por la biblioteca estática *AR32.lib*, *arInitCparam()*
- Cargar en memoria el patrón especificado para su posterior procesamiento, *arLoadPatt()*
- Convertir la matriz de perspectiva (calculada por el *ARToolKit*) a un formato de 4x4 que pueda ser interpretado como un vector de 16 elementos, *argConvGLcpara()*
- A partir de la imagen adquirida, verificar si existe algún patrón reconocido dentro de la misma, *arDetectMarker()*
- Calcular la matriz de transformación del objeto virtual, es decir, la relación existente entre el patrón reconocido y la cámara real, *arGetTransMat()* y *arGetTransMatCont()*
- Convertir la matriz de transformación a un formato de coordenadas homogéneas (matrices de 4x4 o bien, un vector de 16 elementos), *argConvGLpara()*

Además de las rutinas, es necesario la creación de determinadas variables que llevarán control sobre los eventos que se presenten. Generalmente se definen las siguientes:

```
// Pointers to the acquired images that will be processed
ARUint8       *pImAquired = NULL;
int           xsize, ysize;                          // Window size
int           thresh = 100;                          // Threshold
char          *cparam_name = "Data/camera_para.dat"; // Camera Data
ARParam       cparam, wparam;                        // Camera parameters
int           *patt_id;                              // Pattern's id
int           patt_width = 80;                       // Pattern's width [mm]
double        patt_center[2] = {0.0, 0.0};           // Pattern's center
ARMarkerInfo  *marker_info;
int           marker_num;                            // Number of patterns
```

en donde *ARUint8* no es más que una variable de tipo BYTE, *ARParam* es una estructura que almacena los datos de calibración de la cámara y, por último, *ARMakerInfo* es otra estructura utilizada para tener en memoria los datos asociados a los patrones cargados.

Ahora bien, el programa estará formado por al menos las siguientes rutinas o piezas destinadas al cumplimiento de cierta tarea:

- Rutina principal o *main*, en la que serán incluidas el resto
- Inicialización del entorno (gráfico, de video y lógico)
- Ciclo de dibujo (*render*) de la escena
- Destrucción de la aplicación y liberación de memoria

Cada una de estas etapas requiere de al menos una función del *ARToolKit*. Comenzando por la rutina principal, el primer paso es cargar los datos de la cámara. Sin ellos, no es posible realizar ninguna detección de patrones ni dibujar objetos de manera correcta acorde a lo percibido por la cámara. Un ejemplo de empleo de la rutina es la siguiente.

```
if( arParamLoad(cparam_name, 1, &wparam) < 0 )
{
        Msg( TEXT("Failed to load camera parameters."));
        return E_FAIL;
}
```

Con los parámetros cargados, el siguiente paso corresponde a la inicialización del resto de componentes, video, entorno gráfico, entorno de aplicación y rutinas del *ARToolKit*. El siguiente código ilustra la inicialización de las variables de la biblioteca estática *AR32.lib*.

```
// ARToolKit initializations
// Init and change camera parameters
arParamChangeSize( &wparam, xsize, ysize, &cparam );
arInitCparam( &cparam );
cparam.mat[0][2] = 139.5;           // Adjusts needed in this work
cparam.mat[1][2] = 170.5;           // Adjusts needed in this work
int i = 0;
patt_id = new int[g_iNumItems];     // Number of patterns to be recognized
if( !patt_id )
        return E_FAIL;
// Load every specified pattern
while( i < g_iNumItems )
{
        if( (patt_id[i] = arLoadPatt(g_pModelList[i].pPattern)) < 0 )
        {
                Msg(TEXT("pattern load error !!"));
                return E_FAIL;
        }
        i++;
}
```

Cómo se puede observar, además de la inicialización de la biblioteca estática, también son cargados en memoria cada uno de los patrones especificados por el campo *pPattern* de la estructura *g_pModelList*, estructura que ha sido definida para contener la información de cada uno de los elementos del libro que debe ser cargado.

Finalizado con el proceso de inicialización, es turno de analizar la manera en que operaría la función principal de dibujo. Para comenzar, debe verificarse si existe una imagen a ser procesada. En caso de que no la haya, no se debe de continuar con el proceso, puesto que las rutinas tomarán un apuntador a NULL e incurrirán en error. Enseguida es evaluada la visibilidad de objetos es decir, si se reconoce algún patrón en la imagen adquirida y calculada la matriz de transformación asociada al objeto virtual.

```
// Don't draw anything until an image has been acquired
if(pImAcquired == NULL)
        return E_FAIL;
ARParam gCparam;
gCparam = cparam;
for( i = 0; i < 4; i++ ) {
        gCparam.mat[1][i] = (gCparam.ysize-1)*(gCparam.mat[2][i]) –
                                                   gCparam.mat[1][i];
}
argConvGLcpara( &gCparam, AR_GL_CLIP_NEAR, AR_GL_CLIP_FAR, d3d_para );
// Recognition of the marker
if( arDetectMarker( pImAcquired, thresh, &marker_info, &marker_num ) < 0 )
        return E_FAIL;
// Check for object visibility
for( i = 0; i < g_iNumItems; i++ )
{
```

```
            k = -1;
            for( j = 0; j < marker_num; j++ )
            {
                    if( patt_id[i] == marker_info[j].id )
                    {
                            if( k == -1 ) k = j;
                            else
                                    if( marker_info[k].cf < marker_info[j].cf ) k = j;
                    }
            }
            if( k == -1 )
            {
                    g_pModelList[i].iVisibility = 0;
                    continue;
            }
            if( g_pModelList[i].iVisibility == 0 )
                    arGetTransMat( &marker_info[k], patt_center, patt_width,
                                    g_pModelList[i].MatTrans);
            else
                    arGetTransMatCont( &marker_info[k], g_pModelList[i].MatTrans,
                                    patt_center, patt_width, g_pModelList[i].MatTrans );
            g_pModelList[i].iVisibility = 1;
    }
```

Con el anterior procesamiento, todo está listo para ser dibujado. Primero el video y enseguida el (los) objeto(s) reconocido(s). Por lo tanto, se procede a dibujar la escena de la manera acostumbrada (según la *API* empleada) y cuando sea el turno del (de los) objeto(s) virtual(es) se evalúa el dibujo como se presenta en el código:

```
    // Render Visible Models
    for( i = 0; i < g_iNumItems; i++ )
    {
        if( g_pModelList[i].iVisibility == 0 )          continue;
        // Set Projection Matrix based on ARToolKit estimation
        TempMatrix._11=(float)d3d_para[ 0]; TempMatrix._12=(float)d3d_para[ 1];
        TempMatrix._13=(float)d3d_para[ 2]; TempMatrix._14=(float)d3d_para[ 3];
        TempMatrix._21=-(float)d3d_para[ 4]; TempMatrix._22=-(float)d3d_para[ 5];
        TempMatrix._23=-(float)d3d_para[ 6]; TempMatrix._24=-(float)d3d_para[ 7];
1       TempMatrix._31=(float)d3d_para[ 8]; TempMatrix._32=(float)d3d_para[ 9];
        TempMatrix._33=(float)d3d_para[10]; TempMatrix._34=(float)d3d_para[11];
        TempMatrix._41=(float)d3d_para[12]; TempMatrix._42=(float)d3d_para[13];
        TempMatrix._43=(float)d3d_para[14]; TempMatrix._44=(float)d3d_para[15];
        D3DXMatrixIdentity( &g_pModelList[i].Proj );
        D3DXMatrixMultiply( &g_pModelList[i].Proj, &g_pModelList[i].Proj, &TempMatrix );
        // Set Model Matrix
        argConvGlpara(g_pModelList[i].MatTrans, d3d_para);
        TempMatrix._11=(float)d3d_para[ 0]; TempMatrix._12=(float)d3d_para[ 1];
        TempMatrix._13=(float)d3d_para[ 2]; TempMatrix._14=(float)d3d_para[ 3];
        TempMatrix._21=(float)d3d_para[ 4]; TempMatrix._22=(float)d3d_para[ 5];
        TempMatrix._23=(float)d3d_para[ 6]; TempMatrix._24=(float)d3d_para[ 7];
2       TempMatrix._31=(float)d3d_para[ 8]; TempMatrix._32=(float)d3d_para[ 9];
        TempMatrix._33=(float)d3d_para[10]; TempMatrix._34=(float)d3d_para[11];
        TempMatrix._41=(float)d3d_para[12]; TempMatrix._42=(float)d3d_para[13];
        TempMatrix._43=(float)d3d_para[14]; TempMatrix._44=(float)d3d_para[15];
        D3DXMatrixIdentity(&g_pModelList[i].World );
        D3DXMatrixMultiply(&g_pModelList[i].World,&g_pModelList[i].Trans,&TempMatrix);
        // Render Model
        g_pModelList[i].pRender( g_pModelList+i, g_pModelList[i].pModel2Render );
    }
```

Lo primero que se debe de hacer es verificar la visibilidad del objeto mediante el campo *visibility* de la estructura *g_pModelList*. En caso de que sea visible, entonces se procede a dibujar el objeto mientras que en el negativo se procede a verificar la del siguiente elemento. El punto 1 ilustra el cálculo de la matriz de perspectiva, paso que se explicó con anterioridad.

Lo que se hace aquí es tomar una matriz temporal que servirá para copiar los elementos de la matriz de perspectiva y, posteriormente, de la matriz de transformación. Posteriormente (parte 2), se calcula la matriz de transformación, se le hacen los ajustes necesarios y, finalmente, se manda a dibujar el objeto en cuestión.

Cabe mencionar que el esquema aquí presentado es muy general y que el código usado para ejemplificar, corresponde a un caso particular que es el de la *API* de *Direct3D*. La parte del dibujo de la escena, corresponde al proceso de *render* empleado durante la elaboración de este trabajo y dependiendo del tipo de aplicación que se requiera, variará.

En caso de que sea necesario profundizar en el tema o, por ejemplo, conocer un esquema general para *OpenGL*, entonces se puede consultar la ayuda del *ARToolKit* y sus programas de ejemplo. Ambos pueden ser descargados desde la página web del software en: http://www.hitl.washington.edu/resarch/shared_space/download/

Apéndice C
Cuaterniones

La importancia del uso de los cuaterniones, radica en su bajo costo computacional para efectuar rotaciones en 3 dimensiones, en comparación con los ángulos de Euler. A continuación se tratarán diversos aspectos sobre como fueron creados, sus propiedades algebraicas, la manera en que se realiza una rotación y una justificación del porque puede representar a la misma.

C.1 Bosquejo histórico

El creador del cuaternión es Sir William Rowan Hamilton, matemático irlandés, equiparable a Newton, de gran nombre gracias a todos sus trabajos realizados. Con tan solo catorce años conocía al menos 8 idiomas, entre ellos italiano, francés, griego, latín, hebreo y sánscrito. A sus 16 años ya leía tratados completos de matemáticos de la época y su primer logro fue encontrar un error en el libro "La mecánica celeste" de Pierre-Simone Laplace.

En el año de 1843, Hamilton buscaba formas de extender los números complejos (que pueden interpretarse como puntos en un plano) a un número mayor de dimensiones. No pudo hacerlo para 3 dimensiones, pero para 4 dimensiones obtuvo los cuaterniones. Según una historia relatada por el propio Hamilton, la solución al problema que le ocupaba le sobrevino un día que estaba paseando con su esposa, bajo la forma de la ecuación: $\boldsymbol{i}^2 = \boldsymbol{j}^2 = \boldsymbol{k}^2 = \boldsymbol{ijk} = -1$. Inmediatamente, grabó esta expresión en el lateral del puente de Brougham, que estaba muy cerca del lugar.

Además, creía que este descubrimiento revolucionaría la Física y empleó el resto de su vida trabajando en los cuaterniones, escribiendo su primer libro en 1853 ("*Lectures in Quaternions*"), proyecto que abandonó al notar que no era didáctico para la enseñanza de la nueva teoría. Sin embargo insistió en la escritura (era un amante ferreo de la lectura y la escritura) y en 1866 publicó su principal obra: "*Elements of Quaternions*", con un total de 800 páginas.

Es de notar que dentro del álgebra y cálculo, introdujo el operador nabla, representado por ∇ (ya que dicho símbolo se asemeja a un antiguo musical hebreo del mismo nombre) definido como:

$$\nabla = i\frac{\delta}{\delta x} + j\frac{\delta}{\delta y} + k\frac{\delta}{\delta z}$$

C.2 Algebra y definición de un cuaternión

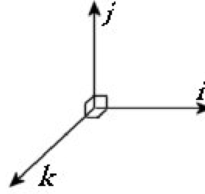
Los cuaterniones son una extensión de los números complejos. Mientras que los números complejos son una extensión de los reales por la adición de la unidad imaginaria $\boldsymbol{i}$, tal que $\boldsymbol{i}^2 = -1$, los cuaterniones son una extensión generada de manera análoga añadiendo las unidades imaginarias: $\boldsymbol{i}$, $\boldsymbol{j}$ y $\boldsymbol{k}$ tal que $\boldsymbol{i}^2 = \boldsymbol{j}^2 = \boldsymbol{k}^2 = \boldsymbol{ijk} = -1$.

	1	***i***	***j***	***k***
1	*1*	*i*	*j*	*k*
i	*i*	*-1*	*k*	*-j*
j	*j*	*-k*	*-1*	*i*
k	*k*	*j*	*-i*	*-1*

Tabla de multiplicación para los números *i, j y k*
Tabla C.1

Entonces un cuaternión es un número de la forma $a + b\boldsymbol{i} + c\boldsymbol{j} + d\boldsymbol{k}$, donde a, b, c, y d son números reales unívocamente determinados por cada cuaternión.

Si se analiza la Tabla C.1, la multiplicación de los cuaterniones no es conmutativa: ***ij*** = ***k***, ***ji*** = -***k***, ***jk*** = ***i***, ***kj*** = -***i***, ***ki*** = ***j***, ***ik*** = -***j***. Haciendo una analogía con el álgebra de vectores, la multiplicación de los números ***i, j*** *y* ***k*** es similar al producto cruz de vectores de una base ortonormal como lo es la de R^3 (base de vectores *i, j, k*). Véase la Figura C.1 para recordar como es una base de dicho tipo.



Base ortonormal que origina a R^3
Figura C.1

Es importante mencionar que la multiplicación es asociativa y todo cuaternión no nulo posee un único inverso. Esto es, dados tres cuaterniones,

$$\boldsymbol{q_1} = w_1 + x_1\boldsymbol{i} + y_1\boldsymbol{j} + z_1\boldsymbol{k}$$
$$\boldsymbol{q_2} = w_2 + x_2\boldsymbol{i} + y_2\boldsymbol{j} + z_2\boldsymbol{k}$$
$$\boldsymbol{q_3} = w_3 + x_3\boldsymbol{i} + y_3\boldsymbol{j} + z_3\boldsymbol{k}$$

se tiene que

$$(\boldsymbol{q_1 q_2})\boldsymbol{q_3} = \boldsymbol{q_1}(\boldsymbol{q_2 q_3})$$

y que el inverso de uno de ellos, ***q*** se obtiene haciendo:

$$\boldsymbol{1/q = q^{-1} = q'/(qq')}$$

en donde ***q'*** es el conjugado del cuaternión ***q*** y se calcula como,

$$\boldsymbol{q'} = w - x\boldsymbol{i} - y\boldsymbol{j} - z\boldsymbol{k}$$

Otra propiedad del conjugado de un cuaternión es:

$$(\boldsymbol{q'})' = \boldsymbol{q}$$
$$(\boldsymbol{q} + \boldsymbol{r})' = \boldsymbol{q'} + \boldsymbol{r'}$$
$$(\boldsymbol{qr})' = \boldsymbol{r'q'}$$

Los cuaterniones también tienen norma y se representa de la misma manera que la de un vector de los reales:

$$\|q\| = \sqrt{qq'} = \sqrt{w^2 + x^2 + y^2 + z^2}$$

Cuando se opera con cuaterniones de norma igual a uno, es decir unitarios, entonces su inversa es equivalente a su conjugado. Por último, estos elementos pueden ser expresados de diferentes maneras: como una combinación lineal de ***1***, ***i***, ***j*** y ***k***, un vector de cuatro elementos (compuestos por la combinación lineal de ellos) o bien, un escalar (factor del elemento ***1***) y un vector de tamaño tres con los elementos imaginarios.

$$\boldsymbol{q} = w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k} = [w\ x\ y\ z] = (\boldsymbol{s}, \boldsymbol{v})$$
$$\boldsymbol{s} = w$$
$$\boldsymbol{v} = [x\ y\ z]$$

Cuando se expresa en términos de un escalar y vector 'imaginario', la multiplicación entre dos cuaterniones se efectúa de la siguiente manera:

$$\boldsymbol{q_1} = (\boldsymbol{s_1}, \boldsymbol{v_1})$$
$$\boldsymbol{q_2} = (\boldsymbol{s_2}, \boldsymbol{v_2})$$
$$\boldsymbol{q_1\, q_2} = (\boldsymbol{s_1 s_2 - v_1 \cdot v_2},\ \boldsymbol{s_1 v_2 + s_2 v_1 + v_1} \times \boldsymbol{v_2})$$

C.3 Representación de rotaciones usando cuaterniones

Para representar una rotación sobre un vector (unitario) cualquiera **u**, un ángulo θ se tiene un cuaternión de la forma:

$$\boldsymbol{q} = (\boldsymbol{s}, \boldsymbol{v})$$
$$\mathbf{s} = \cos(\theta / 2)$$
$$\mathbf{v} = \mathbf{u} \sin(\theta / 2)$$

El punto en el espacio $\boldsymbol{p}$, que será rotado, puede expresarse en términos de un cuaternión $\boldsymbol{P} = (\boldsymbol{0}, \boldsymbol{p})$ y la rotación se define por:

$$P_{rotated} = qPq^{-1}$$

Asimismo, es conveniente normalizar el cuaternión q para emplear la propiedad de que su inversa q^{-1} es equivalente a su conjugado ($q^{-1} = q'$).

Cuando se requiere efectuar mas de una rotación sobre determinado objeto, las operaciones son concatenadas y los cuaterniones son útiles para tales objetivos. Supóngase que q_1 y q_2 son dos cuaterniones representando los ejes y ángulos de rotación y se busca rotar en el orden presentado. Para hacer esto, se rota sobre q_1 y, al resultado, se aplica q_2. Esto es:

$$P_{rotated} = q_2(q_1Pq_1^{-1})q_2^{-1}$$
$$P_{rotated} = (q_2q_1)P(q_1^{-1}q_2^{-1})$$

Si se emplea la propiedad $(qr)' = r'q'$, sabiendo que $|q_1| = |q_2| = 1$ y por consiguiente $q_1^{-1} = q_1'$ y $q_2^{-1} = q_2'$, se puede escribir

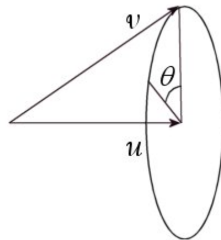
$$P_{rotated} = (q_2q_1)P(q_1'q_2')$$
$$P_{rotated} = (q_2q_1)P(q_2q_1)'$$
$$P_{rotated} = q_nPq_n'$$

Entonces, la concatenación de rotaciones mediante cuaterniones puede operarse paso a paso (procedimiento usual), o bien calcular el cuaternión compuesto que representa la pose final.

C.4 Demostración de que un cuaternión define una rotación

Las secciones anteriores indican que es un cuaternión, algunas de sus propiedades y la especificación de rotaciones a partir de ellos. Pero, ¿es verdad que representan rotaciones en un espacio tridimensional real al ser elementos del campo de los complejos y dimensión cuatro? En este apartado se realizará una pequeña demostración de ello.

Véase la Figura C.2. En ella aparecen dos vectores, $\boldsymbol{u}$ representa el eje de rotación (recordar que debe ser unitario) y $\boldsymbol{v}$ el vector que será rotado un ángulo θ alrededor de $\boldsymbol{u}$.



Eje de rotación y vector a rotar
Figura C.2

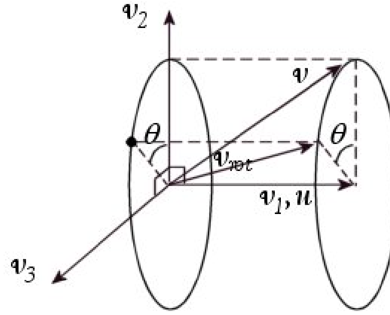
Para fines de la demostración, se calcularán otros vectores mostrados en la Figura C.3. Los vectores $\boldsymbol{v_1}$, $\boldsymbol{v_2}$ y $\boldsymbol{v_3}$ representan un sistema ortogonal en donde:

$\boldsymbol{v_1} = (\boldsymbol{u}\cdot\boldsymbol{v})\boldsymbol{u}$
$\boldsymbol{v_2} = \boldsymbol{v} - (\boldsymbol{u}\cdot\boldsymbol{v})\boldsymbol{u}$
$\boldsymbol{v_3} = \boldsymbol{u} \times \boldsymbol{v}$

y el vector rotado $\boldsymbol{v_{rot}}$ puede ser expresado en términos de dicha terna como

$$v_{rot} = v_1 + v_2\cos(\theta) + v_3\sin(\theta)$$

que representa al vector obtenido a partir de la geometría definida.



Sistema ortogonal formado por $\boldsymbol{v_1}$, $\boldsymbol{v_2}$ y $\boldsymbol{v_3}$
Figura C.3

Ahora constrúyanse los cuaterniones descriptores del vector a transformar y el eje de rotación. De esta manera se tendría que:

$$\boldsymbol{q} = (\cos(\theta/2), \boldsymbol{u}\sin(\theta/2))$$
$$\boldsymbol{V} = (0, \boldsymbol{v})$$
$$\boldsymbol{v_{rot}} = \boldsymbol{qVq^{-1}}$$

operando y sabiendo que $\boldsymbol{q^{-1}} = \boldsymbol{q'}$ porque el eje de rotación es unitario se encuentra que

$$v_{rot} = \left(\cos\left(\frac{\theta}{2}\right), u\sin\left(\frac{\theta}{2}\right)\right) * (0, v) * \left(\cos\left(\frac{\theta}{2}\right), -u\sin\left(\frac{\theta}{2}\right)\right)$$

$$v_{rot} = \left[\left(\cos\left(\frac{\theta}{2}\right), u\sin\left(\frac{\theta}{2}\right)\right) * (0, v)\right] * \left(\cos\left(\frac{\theta}{2}\right), -u\sin\left(\frac{\theta}{2}\right)\right)$$

$$v_{rot} = \left(-(u\cdot v)\sin\left(\frac{\theta}{2}\right), v\cos\left(\frac{\theta}{2}\right) + (u\times v)\sin\left(\frac{\theta}{2}\right)\right) * \left(\cos\left(\frac{\theta}{2}\right), -u\sin\left(\frac{\theta}{2}\right)\right)$$

$$v_{rot} = \begin{pmatrix} -(u\cdot v)\sin\left(\frac{\theta}{2}\right)\cos\left(\frac{\theta}{2}\right) + (u\cdot v)\sin\left(\frac{\theta}{2}\right)\cos\left(\frac{\theta}{2}\right) + (u\cdot(u\times v))\sin^2\left(\frac{\theta}{2}\right), \\ (u\cdot v)u\sin^2\left(\frac{\theta}{2}\right) + v\cos^2\left(\frac{\theta}{2}\right) + (u\times v)\sin\left(\frac{\theta}{2}\right)\cos\left(\frac{\theta}{2}\right) \\ -(v\times u)\sin\left(\frac{\theta}{2}\right)\cos\left(\frac{\theta}{2}\right) - ((u\times v)\times u)\sin^2\left(\frac{\theta}{2}\right) \end{pmatrix}$$

Rescribiendo $\boldsymbol{v_{rot}}$ como $\boldsymbol{v_{rot}} = (\boldsymbol{s_{rot}}, \boldsymbol{w_{rot}})$,

$$s_{rot} = -(u\cdot v)\sin\left(\frac{\theta}{2}\right)\cos\left(\frac{\theta}{2}\right) + (u\cdot v)\sin\left(\frac{\theta}{2}\right)\cos\left(\frac{\theta}{2}\right) + (u\cdot(u\times v))\sin^2\left(\frac{\theta}{2}\right)$$

y

$$w_{rot} = (u \cdot v)u \sin^2\left(\frac{\theta}{2}\right) + v\cos^2\left(\frac{\theta}{2}\right) - ((u \times v) \times u)\sin^2\left(\frac{\theta}{2}\right)$$
$$+ (u \times v)\sin\left(\frac{\theta}{2}\right)\cos\left(\frac{\theta}{2}\right) - (v \times u)\sin\left(\frac{\theta}{2}\right)\cos\left(\frac{\theta}{2}\right)$$

Para simplificar las expresiones se emplearán las siguientes identidades trigonométricas

$$\cos(\theta) = \cos^2(\theta/2) - \sin^2(\theta/2)$$
$$\sin(\theta) = 2\sin(\theta/2)\cos(\theta/2)$$
$$\cos(\theta) = 1 - 2\sin^2(\theta/2)$$

Comenzando con $\boldsymbol{s_{rot}}$, se observa con facilidad que el termino es igual a cero ya que los dos primeros factores se eliminan y el último corresponde al producto punto de vectores perpendiculares entre sí. Antes de abordar el caso de $\boldsymbol{w_{rot}}$, es conveniente remarcar que $\boldsymbol{u}\times\boldsymbol{v} = -(\boldsymbol{v}\times\boldsymbol{u})$ y que $(\boldsymbol{u}\times\boldsymbol{v})\times\boldsymbol{u} = \boldsymbol{v_2} = \boldsymbol{v} - (\boldsymbol{u}\cdot\boldsymbol{v})\boldsymbol{u}$ (véase la Figura C.3). Continuando,

$$w_{rot} = (u \cdot v)u \sin^2\left(\frac{\theta}{2}\right) + v\cos^2\left(\frac{\theta}{2}\right) - (v - (u \cdot v)u)\sin^2\left(\frac{\theta}{2}\right)$$
$$+ (u \times v)\sin\left(\frac{\theta}{2}\right)\cos\left(\frac{\theta}{2}\right) + (u \times v)\sin\left(\frac{\theta}{2}\right)\cos\left(\frac{\theta}{2}\right)$$
$$w_{rot} = 2(u \cdot v)u \sin^2\left(\frac{\theta}{2}\right) + v\left(\cos^2\left(\frac{\theta}{2}\right) - \sin^2\left(\frac{\theta}{2}\right)\right) + 2(u \times v)\sin\left(\frac{\theta}{2}\right)\cos\left(\frac{\theta}{2}\right)$$
$$w_{rot} = (u \cdot v)u(1 - \cos(\theta)) + v\cos(\theta) + (u \times v)\sin(\theta)$$
$$w_{rot} = (u \cdot v)u - (u \cdot v)u\cos(\theta) + v\cos(\theta) + (u \times v)\sin(\theta)$$
$$w_{rot} = (u \cdot v)u + (v - (u \cdot v)u)\cos(\theta) + (u \times v)\sin(\theta)$$

Finalmente

$$w_{rot} = v_1 + v_2\cos(\theta) + v_3\sin(\theta)$$

con lo que se demuestra que mediante cuaterniones se pueden expresar rotaciones en el espacio vectorial de R^3.

C.5 Representación matricial para la multiplicación de cuaterniones

Si bien estos elementos permiten especificar de una manera mas sencilla una rotación en el espacio y que las operaciones involucradas en el cálculo son menos costosas (en términos de poder de computo), al momento de involucrar otras transformaciones como escalamientos y traslaciones es necesario tener una representación matricial de la rotación en cuestión, para poder generar la matriz final de transformación. Es por ello que se presenta este apartado, en el que se muestra como se efectúan multiplicaciones entre cuaterniones de forma matricial.

Considérense los elementos $\boldsymbol{q_1}$ y $\boldsymbol{q_2}$ mostrados a continuación.

$$\boldsymbol{q_1} = x_1\boldsymbol{i} + y_1\boldsymbol{j} + z_1\boldsymbol{k} + w_1$$
$$\boldsymbol{q_2} = x_2\boldsymbol{i} + y_2\boldsymbol{j} + z_2\boldsymbol{k} + w_2$$

La multiplicación de los mismos está dada por:

$$
\begin{aligned}
\boldsymbol{q_1 q_2} = & \;(\;\; x_1w_2 + y_1z_2 - z_1y_2 + w_1x_2)\boldsymbol{i} \\
& + (-x_1z_2 + y_1w_2 + z_1x_2 + w_1y_2)\boldsymbol{j} \\
& + (\;\; x_1y_2 - y_1x_2 + z_1w_2 + w_1z_2)\boldsymbol{k} \\
& + (-x_1x_2 - y_1y_2 - z_1z_2 + w_1w_2)
\end{aligned}
$$

donde se observa que cada termino es una combinación lineal de las componentes de $\boldsymbol{q_1}$ y $\boldsymbol{q_2}$. Por lo tanto, es posible especificar el producto de dos cuaterniones mediante una multiplicación de matrices.

Si se considera que las combinaciones lineales que forman a cada uno de los elementos del nuevo cuaternión conforman las columnas de una matriz, entonces se puede decir que la multiplicación del "vector" $\boldsymbol{q_1}$ con la matriz $\boldsymbol{R_{row}(q_2)}$ es equivalente al producto de los cuaterniones $\boldsymbol{q_1q_2}$. La matriz $\boldsymbol{L_{row}(q_1)}$ se construye de igual manera, pero con los elementos del cuatermión $\boldsymbol{q_2}$.

Si se considera que las combinaciones lineales que forman a cada uno de los elementos del nuevo cuaternión conforman las columnas de una matriz, entonces se puede decir que la multiplicación del "vector" $\boldsymbol{q_1}$ con la matriz $\boldsymbol{R_{row}(q_2)}$ es equivalente al producto de los cuaterniones $\boldsymbol{q_1q_2}$. La matriz $\boldsymbol{L_{row}(q_1)}$ se construye de igual manera, pero con los elementos del cuatermión $\boldsymbol{q_2}$.

$$
q_1q_2 = q_1R_{row}(q_2) = \begin{bmatrix} x_1 & y_1 & z_1 & w_1 \end{bmatrix} \begin{bmatrix} w_2 & -z_2 & y_2 & -x_2 \\ z_2 & w_2 & -x_2 & -y_2 \\ -y_2 & x_2 & w_2 & -z_2 \\ x_2 & y_2 & z_2 & w_2 \end{bmatrix}
$$

$$
q_1q_2 = q_2L_{row}(q_1) = \begin{bmatrix} x_2 & y_2 & z_2 & w_2 \end{bmatrix} \begin{bmatrix} w_1 & z_1 & -y_1 & -x_1 \\ -z_1 & w_1 & x_2 & -y_1 \\ y_1 & -x_1 & w_1 & -z_1 \\ x_1 & y_1 & z_1 & w_1 \end{bmatrix}
$$

Con la representación matricial de la multiplicación entre cuaterniones, es posible calcular las rotaciones que representan. Considérese la matriz $\boldsymbol{L_{row}(q)}$ que multiplica por la izquierda (producto $\boldsymbol{qP}$):

$$
PL_{row}(q) = \begin{bmatrix} x_p & y_p & z_p & w_p \end{bmatrix} \begin{bmatrix} w_q & z_q & -y_q & -x_q \\ -z_q & w_q & x_q & -y_q \\ y_q & -x_q & w_q & -z_q \\ x_q & y_q & z_q & w_q \end{bmatrix}
$$

Para evaluar el producto $\boldsymbol{Pq}^{-1}$ se emplea una matriz derecha $\boldsymbol{R_{row}(q)}$. Además, recuérdese que dado que el eje de rotación es unitario, la inversa de $\boldsymbol{q}$ puede calcularse como $\boldsymbol{q'}$. Por lo tanto se tiene que:

$$
q^{-1} = q' = \begin{bmatrix} -x_q & -y_q & -z_q & w_q \end{bmatrix}
$$

$$PR_{row}\left(q^{-1}\right)=\begin{bmatrix}x_p & y_p & z_p & w_p\end{bmatrix}\begin{bmatrix} w_q & z_q & -y_q & x_q \\ -z_q & w_q & x_q & y_q \\ y_q & -x_q & w_q & z_q \\ -x_q & -y_q & -z_q & w_q \end{bmatrix}$$

Es de notar que la construcción de está matriz corresponde a la transpuesta de la que se obtuvo anteriormente y que puede verificarse realizando el mismo proceso de obtención. Continuando con el procedimiento de calculo de la matriz de rotación, ésta es obtenida como el producto de las matrices $\boldsymbol{R_{row}(q^{-1})\ L_{row}(q)}$,

$$\boldsymbol{qPq'} = \boldsymbol{q}(\boldsymbol{Pq'}) = \boldsymbol{q}(\boldsymbol{PR_{row}}(\boldsymbol{q'})) = (\boldsymbol{PR_{row}}(\boldsymbol{q'}))\boldsymbol{L_{row}}(\boldsymbol{q}) = \boldsymbol{P}(\boldsymbol{R_{row}}(\boldsymbol{q'})\boldsymbol{L_{row}}(\boldsymbol{q})) = \boldsymbol{PQ_{row}}(\boldsymbol{q})$$

obteniéndose:

$$Q_{row} = R_{row}\left(q^{-1}\right)L_{row}(q)=\begin{bmatrix} w_q & z_q & -y_q & x_q \\ -z_q & w_q & x_q & y_q \\ y_q & -x_q & w_q & z_q \\ -x_q & -y_q & -z_q & w_q \end{bmatrix}\begin{bmatrix} w_q & z_q & -y_q & -x_q \\ -z_q & w_q & x_q & -y_q \\ y_q & -x_q & w_q & -z_q \\ x_q & y_q & z_q & w_q \end{bmatrix}$$

$$Q_{row} =$$

$$\begin{bmatrix} w_q^2+x_q^2-y_q^2-z_q^2 & 2x_qy_q+2w_qz_q & 2x_qz_q-2w_qy_q & 0 \\ 2x_qy_q-2w_qz_q & w_q^2-x_q^2+y_q^2-z_q^2 & 2y_qz_q+2w_qx_q & 0 \\ 2x_qz_q+2w_qy_q & 2y_qz_q-2w_qx_q & w_q^2-x_q^2-y_q^2+z_q^2 & 0 \\ 0 & 0 & 0 & w_q^2+x_q^2+y_q^2+z_q^2 \end{bmatrix}$$

De esta manera, a partir de un cuaternión q se puede obtener la matriz correspondiente de rotación $\boldsymbol{Q_{row}}(\boldsymbol{q})$ y expresar el vector rotado como:

$$\boldsymbol{P_{rotated}} = \boldsymbol{P\ Q_{row}}(\boldsymbol{q})$$

C.6 Ángulos de Euler

En muchos campos, los ángulos de Euler se usan para efectuar rotaciones. Cualquier rotación puede descomponerse en series de rotaciones sobre los tres ejes que conformen al espacio vectorial.

Se puede simular una rotación arbitraria (en R^3) con una rotación sobre el eje X, una sobre Y y otra sobre Z. Por ejemplo, para un avión orientado en X y con Z hacia arriba de él. Cualquier pose que se requiera puede ser representada por los movimientos "*roll*", "*pitch*" y "*yaw*", cada uno de ellos rotaciones sobre los ejes X, Y y Z, respectivamente.

Esta representación de rotaciones es intuitiva y sencilla de usar en algunos casos pero presenta ciertos puntos en contra. Uno de ellos es que no existe un orden absoluto en el cual aplicar las transformaciones. Pueden efectuarse por ejemplo en z-y-z, x-y-z, x-z-x, etc. provocando efectos adversos sobre el objeto de hacer las rotaciones en la secuencia incorrecta.

A su vez, dado que cualquier rotación puede representarse por medio de estos ángulos o una matriz, puede calcularse una conversión entre ellos. Sin embargo el cálculo de dichos ángulos es costoso y puede introducir numerosos errores. Por otro lado, calcular una interpolación entre dos poses, provocará giros bruscos alrededor de los ejes canónicos (recordar que no se interpola un giro sino tres).

Para concluir, el uso de ángulos de Euler para realizar rotaciones es mas intuitivo y sencillo, pero para fines específicos (como interpolación de cuadros durante una animación) puede mostrar errores o efectos no deseados. Cuando se desea crear una animación mas compleja y basada, por ejemplo, en física, el uso de cuaterniones es sugerido pues pueden manejar de manera directa rotaciones alrededor de un eje arbitrario.

BIBLIOGRAFÍA

1. Arvo, James. *Graphics Gems II*. Estados Unidos, Ed. Morgan Kaufmann, 1994. 672p
2. Birn, Jeremy. *Técnicas de iluminación y render*. España, Ed. Anaya Multimedia, 2000. 304p
3. Engel, Wolfgang F., Amir Geva. *Beginning Direct3D game programming*. Estados Unidos, Ed. Prima Tech, 2001. 504p
4. ---------------------. *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*. Estados Unidos, Ed. Wordware Publishing, 2002. 500p
5. Foley, James D, Andries van Dam, Steven K. Feiner, John F. Hughes. *Computer graphics: principles and practice*. 2ª edición. Estados Unidos, Ed. Addison-Wesley, 1995. 1175 p
6. Fosner, Ron. *Real-Time Shader Programming, Covering DirectX 9.0*. Estados Unidos, Ed. Morgan Kaufmann Publishers, 2003. 406p
7. Hearn, Donald, M. Pauline Baker. *Gráficas por computadora*. 2ª edición. México, Ed. Prentice Hall, 1995. 686p
8. Jones, Angie, Sean Bonney, Brandon Davis, Sean Miller y Shane Olsen. *3D Studio Max 3 Animación Profesional*. Madrid, España. Ed. Prentice Hall, 1998. 724p
9. Möller, Thomas, Eric Haines. *Real-Time Rendering*. Estados Unidos. Ed. A K Peters, 1999. 482p
10. Negroponte, Nicholas. *Ser digital*. México, Ed. Planeta, 1995. 261p
11. Olano, Marc, John C. Hart, Wolfgang Heidrich, Michael MxCool. *Real-Time Shading*. Estados Unidos, Ed. A K Peters, 2002. 346p.
12. Paeth, Alan. Graphics Gems V. Estados Unidos, Ed. Morgan Kauffman, 1995. 438p
13. Petzold, Charles. *Programming Windows Fifth Edition*. Estados Unidos, Ed. Microsoft Press, 1999. 1479p
14. Preece, Jenny, Rogers Yvonne, Sharp Helen Benyon, David. *Human-Computer Interaction*. Estados Unidos, Ed. Addison-Wesley, 1994. 760p
15. Woo, Mason, Jackie Neider, Tom Davis. *OpenGL Programming Guide, Second Edition*. Estados Unidos, Ed. Addison-Wesley Developers Press, 1997. 650p

Artículos:

1. Arthur, Kevin W., Kellog S. Booth, Colin Ware. *Evaluating 3D Task Performance for Fish Tanks Virtual Worlds*. ACM Transactions on Information Systems, julio de 1993. p239-265
2. Feiner, Steven, Blair MacIntyre, Marcus Haupt, Eliot Solomon. *Windows on the World: 2D Windows for 3D Augmented Reality*. ACM Symposium. Atlanta Georgia, noviembre de 1993. p145-155
3. Gooch, Amy, Bruce Gooch, Peter Shirley, Elaine Cohen. "*A Non-Photorealistic Lighting Model For Automatic Technical Illustrations*". Departamento de Ciencias de la Computación, Universidad de UTAH.
4. Kato, Hirokazu, Mark Billinghurst, Ivan Poupyrev. "*ARToolKit version 2.33*". Human Interface Technology Laboratory, Universidad de Washington, noviembre de 2000. 44p
5. -----------------. "*Marker Tracking and HMD Calibration for a Video-based Augmented Reality Conferencing System*". Human Interface Technology Laboratory, Universidad de Washington. IWAR 1999.
6. Turkowski, Ken. "*Trasnformations of Surface Norma Vectors with applications to three dimensional computer graphics*". Advanced Technology Group, Graphics Software, Apple Computer Inc. Apple Technical Report No. 22. July the 6th 1990.

Sitios de Internet:

1. Allen, James. "*Creating Cal3D characters conwith 3DS Max*" En línea: < http://cal3d.sourceforge.net/modeling/tutorial.html> Marzo de 2004
2. Audio Video Affair. "*Audio Compression Manager*" En línea < http://www.audio-video-affair.com/audiocompessionmanager.html> Octubre de 2003.
3. Cadmus Professional Communications. "*CPC Glossary*". 2004 En línea < cjs.cadmus.com/da/glossary.asp > Enero de 2004.
4. Cortijo Bon, Francisco José. "*Introducción al reconocimiento de formas*". Octubre de 2001. En línea: <http://www-etsi2.ugr.es/depar/ccia/rf/www/tema1_00-01_www/tema1_00-01_www.html> Marzo de 2003.
5. Directron.org "*Glossary, Computer Video and Graphics*". 1997-2004. En línea: <http://www.directron.com/videoglossary.html> Octubre de 2003.
6. Documentación de Bison. "*Bison 1.28, B. Glossary*". En línea: < http://www.geekgadgets.org/docs/bison_14.html > Noviembre de 2003
7. Gorbatenko, Vadim. "*Class Wrapper for Video Compresión Manager (VCM)*". Noviembre 7 de 2000. <http://www.codeguru.com/Cpp/G-M/multimedia/video/article.php/c1591> Octubre de 2003.
8. Hower, Sean. "*Hokum Glossaries: Computer Graphics*" 2000-2004 en línea < http://hokum.freehomepage.com/Content/Glossary/Glossary_Graphics.html > Noviembre de 2003.
9. Hyperdictionay. "*Morphing: Dictionary entry and meaning*" 2000-2003. En línea < http://www.hyperdictionary.com/computing/morphing> Septiembre de 2003.
10. Interfaz.com "*¿Qué es una interfaz?*" 20 de septiembre de 200. En línea < http://www.interfaz.com.co/asesoria/respuestas.html> Marzo de 2003.

11. Laboratorio de informática de la Universidad Tecnológica Nacional, facultad Regional Mendoza, Argentina. "*Tutorial de Direct3D*" En línea <http://idam.ladei.com.ar/Tutoriales/Direct3D/index.html> Junio 2003.
12. LaCalle, Alberto. "*¿Qué es una interfaz?*" 1999 en línea < http://www.geocities.com/albertlacalle/hci/interfaz.htm> Marzo de 2003.
13. Landsdale, M. W., Ormerod T. C. "*Understanding interfaces. A handbook of human – computer dialog*". 1994 En línea: < http://www.um.es/docencia/agustinr/pca/textos/LandsdHCI.htm> Marzo de 2003.
14. Microsoft Windows Media 9. "*Advance System Format (ASF) Specification*". En línea: <http://www.microsoft.com/windows/windowsmedia/format/asfspec.aspx> Octubre de 2003.
15. Pendleton, C. Robert. "*Game programming and game glossary*". 1997. En línea: < http://www.gameprogrammer.com/glossary.html > Octubre de 2003.
16. Reyes, Juan. "*Reconocimiento de Patrones: Una aproximacion para Expresion de Instrumentos y Arte Interactiva*" Universidad de Stanford, 2003. En línea: <http://ccrma-www.stanford.edu/~juanig/articles/charlAndes/charlAndes.html> Marzo de 2003.
17. Sevo, Daniel. "*History Of Computer Graphics*". En línea: <http://hem.passagen.se/des/index.htm> Abril de 2003.
18. SGI Industries. "*Multisampling*" En línea, documentación de SGI para OpenGL. En línea: <http://www-etsi2.ugr.es/depar/ccia/rf/www/tema1_00-01_www/tema1_00-01_www.html> Junio de 2003.
19. Thomas R. Mark. "*XML GLossaries*". 19 de Octubre de 1999. En línea: < krypton.mnsu.edu/~spiral/eta/glossary/indxGlossOOxml.html > Octubre de 2003

GLOSARIO

ACM*.* Audio Compression Manager. Permite que una aplicación convierta datos entre diferentes formatos. Por sí misma, el ACM no es capaz de hacer las conversiones, es por eso que se cuenta con *drivers* especiales que le permiten el funcionamiento. Estos *drivers* son llamados *codecs*, conversores o filtros.

alpha blending*.* Es una técnica para generar transparencia, en donde el color resultante de cada píxel es una combinación de los colores principales y de fondo.

alpha channel. La componente alfa del color nunca es mostrada. Únicamente es empleada para controlar las mezclas entre colores. En otras palabras, establece el nivel de transparencia del color.

API*.* Application Programming Interface. Interfaz de Programación de Aplicaciones. Conjunto de rutinas y herramientas de desarrollo que conforman una interfaz entre una aplicación de alto nivel y servicios de bajo nivel (sistema operativo, manejadores de dispositivos, etc). Mediante un API es posible construir bloques des software y unirlos para generar una software final.

ASF*.* Advance Systems Format. Es un formato extensible diseñado para almacenar datos multimedia sincronizados. Soporta el flujo de datos sobre una gran variedad de redes y protocolos además de permitir la reproducción de manera local.

AVI. Audio Video Interleaved. Es un caso especial del formato RIFF (Resource Interchange File Format) pero definido por Microsoft. Es el formato más común de video en la PC.

backbuffer. Espacio en memoria de video donde son dibujados los gráficos antes de ser enviados al FrontBuffer para ser mostrados al usuario.

bitmap. Representación binaria del arreglo rectangular de puntos en pantalla. Representación digital de una imagen como un arreglo bidimensional de bits. Estrictamente hablando, un bitmap es siempre monocromático, de manera que cada píxel sea representado por un bit. Este término se ha generalizado y se usa ahora también para referirse a imágenes de color, teneiendo una representación de tipo RGB, RGBA, etc (cada componente de 8 bits).

blend. Técnica basada en la interpolación de cuadros de animación. Mediante esta técnica es posible mezclar dos animaciones diferentes, causando que sean mostradas al mismo tiempo y, en caso que se desee, que tengan la misma duración.

buffer z. Memoria que almacena el valor de profundidad de cada píxel (componente *Z*). Cuando los objetos se dibujan en un *frame buffer*

bidimensional, el motor gráfico deber remover las superficies ocultas.

bug. Error de programación en la codificación de una pieza de software. Dicho error puede causar mal funcionamiento, resultados incorrectos o bien, hacer que el programa falle de forma fatal. Como dato interesante, este término fue adoptado a raíz de que un insecto verdadero, dañará uno de los circuitos de la primer computadora digital: la ENIAC.

bump mapping*.* Técnica de mapeo de textura que logra un efecto de volumen a las superficies. Se logra variando la iluminación de cada píxel de acuerdo con los valores en la textura de *bump* o *Height Map*. Con cada píxel que se dibuja, se lleva a cabo una búsqueda en una textura que simula el relieve de la superficie (*Height Map*). Los valores en dicha textura son usados para modificar las normales del píxel en cuestión. Posteriormente, los cálculos de color e iluminación se basan en la normal modificada teniendo como resultado final una superficie con textura que aparenta volumen.

CAD. Computer Aided Design. Diseño Asistido por Computadora. El software de CAD es útil para producir dibujos, planos y diseños basados en especificaciones y requerimientos reales, etc. Este tipo de diseño es empleado en diferentes campos como la arquitectura, electrónica, industria aerospacial, naval, militar e incluso el diseño de autos. Originalmente, los sistemas de CAD se pensaron para diseño de dibujos y planos, sin embargo, el uso mas común en la actualidad es el de creación de contenido tridimensional para ambientes virutales.

cámara. Para poder crear una escena tridimensional, es necesario especificar una cámara virtual que sea la que determine el volumen visto. Esto es, las *API*s de programación para gráficos como *OpenGL* y *Direct3D*, permiten al programador definir una cámara y un volumen de recorte (ver *Frustum Clipping* en el glosario) con base en parámetros propios de una cámara real, aunados a algunos propios de las virtuales. Véase el glosario para mayor información al respecto.

cell-shading. Técnica de *render* que mediante la cual los objetos tienen apariencia de *comic*. Es decir, en lugar de presentar un sombreado degradado en su superficie (tal y como en la vida real), el objeto es delineado y sombreado con base en tonalidades, claramente al estilo de un dibujo animado.

chroma keying. Esta técnica (también conocida como de la pantalla azul) consiste en fotografiar o tomar cuadros de video a un sujeto frente un fondo azul, para después superponer cualquier fondo que se necesite sin afectar al personaje. Ha sido empleada principalmente en el cine, fines publicitarios y ambientes virtuales usando video.

clamp. Modo de mapeo de textura en donde esta es estirada hasta abarcar el área que debe cubrir.

codec. Compressor/Decompressor. El término *codec* se emplea para referirse a cualquier tecnología para comprimir y descomprimir datos. Pueden ser implementados en software, hardware o ambos.

COM. Component Object Model. Modelo de Componentes de Objetos. Lenguaje orientado a objetos basado en interfaces (estándar de Microsoft) que separa la interfaz de su implementación y, por lo tanto, precisa una manera de describir y definir interfaces y objetos.

componente difusa. Indica el color inherente del material. Por lo general se especifica en formato RGBA.

componente especular. Esta componente establece la capacidad que tiene el material para reflejar luz.

CRT*.* Cathode Rays Tube. Tubo de Rayos Catódicos. Los monitores

basados en CRT disparan un rayo de electrones que incide en la superficie de la pantalla cubierta con fósforo. El fósforo brilla cuando es excitado por el rayo y es de esta manera que se pueden observar la imagen.

cube-mapping. Técnica común de *environment mapping* para objetos reflejantes en donde es calculado un vector de reflexión (ya sea por píxel o por vértice) y posteriormente usado para apuntar dentro de un cubo que tiene mapeada la imagen del mundo.

culling. Es el proceso de eliminar la cara frontal o posterior de un polígono de manera que no sean dibujadas.

custom *FVF*. Custom Flexible Vertex Format. Formato de Vértice Flexible definido por el usuario como una unión de las máscaras provistas por *Direct3D*.

DDI. Device Driver Interface. Interfaz de Manejo de Dispositivos. Define las interfaces entre diferentes capas de manejadores de dispositivos en la jerarquía de manejadores. Típicamente se define un API para cada clase de dispositivo o bus, como parte del DDI.

delta. Diferencia numérica entre un estado final y uno inicial.

depth test. Para eliminar superficies ocultas, el *buffer* z guarda la z más próxima al usuario en cada píxel. El valor de cada fragmento nuevo usa el dato almacenado para hacer una comparación y decidir si ese píxel es o no dibujado.

DirectX. Conjunto de APIs desarrolladas por Microsoft para crear aplicaciones multimedia. Conjunto avanzado de APIs multimedia para sistemas operativos Windows. DirectX provee de una plataforma estándar de desarrollo para PCs basadas en Windows, permitiendo a los desarrolladores acceder a hardware especializado sin necesidad de escribir código que sea específico del hardware.

dithering. Útil para lograr colores con calidad de 24 bits en *buffers* que solo son de 8 o 16 bits. Consiste en la utilización de dos colores para simular un tercero, dándole una apariencia suave.

DLL. Dynamic Link Library. Bibliotecas de Enlace Dinámico. Bibliotecas que proveen de una o más funciones particulares y un programa tiene acceso a ellas mediante la creación de una liga estática o dinámica al *DLL*.

DMA. Direct Memory Access. Acceso Directo a Memoria. Es un método para transferir datos de una memoria a otra sin tener que pasar por la CPU. Las computadoras con canales DMA pueden transferir datos entre dispositivos con mayor velocidad que aquellas carecientes, en donde se debe pasar forzosamente por la CPU.

driver. Pieza de software que indica a al sistema operativo la manera de controlar un dispositivo externo.

DV. Digital Video. Se refiere a la captura, manipulación y almacenamiento de video en formato digital. Una cámara de video digital, por ejemplo, es una cámara de video que captura y almacena imágenes en un medio digital como tarjetas de memoria o cintas magnéticas.

encapsulamiento. Proceso de combinar elementos para crear una entidad nueva. Por ejemplo, una función es un tipo de encapsulamiento porque combina una serie de instrucciones de computadora.

enmascarar. Aplicación de un operador lógico sobre datos para extraer los bits de importancia y operar con ellos. La máscara mas empleada es la operación AND con los bits de interés encendidos.

environment mapping. Mediante el uso de esta técnica se logra un efecto en el que la superficie afectada refleja el medio que lo rodea.

escalamiento uniforme. El modelo o puerto de vista es escalado pro igual en los tres ejes de referencia, dando la

sensación de que cambio de tamaño pero no la proporción. El escalamiento puede realizarse de esta manera, o bien sobre cualquiera de los ejes dando una apariencia de deformidad del objeto.

face culling. Etapa en la que se lleva evalúa cuales son las caras cuya normal tiene componente en dirección de la cámara. Útil para ocultar y no dibujar aquellos polígonos que no se vean.

filtro anisotrópico. Las técnicas usuales de filtrado no compensan la anisotropía, efecto de elongación de los píxeles cuando son mapeados a espacio de textura. El resultado es ver borrosa la textura o con *aliasing*, dependiendo del nivel de detalle y conformación de la textura. Para observar texturas mas claras y nítidas se emplea este tipo de filtrado, proceso que involucra un núcleo elíptico cuya forma y orientación depende de la proyección del píxel en la textura.

flat shading. Sombreado plano. Para dibujar, asigna un solo por cara del polígono. Es la técnica de sombreado con más baja calidad, en donde la superficie del objeto aparenta estar compuesta por muchos bloques.

force-feedback. Realimentación de fuerza. Permite la percepción directa de objetos 3D y acopla las entradas y salidas entre la computadora y el usuario. Es una adición poderosa en los gráficos por computadora, ya que ayuda a resolver problemas que involucran el entendimiento de estructuras 3D, percepción de formas y reacción ante fuerzas en el mundo.

formato *Cal3D*. Formato de archivo que permite tener modelos animados basados en animación por esqueleto. Cabe mencionar que es libre y portable a diversas plataformas y lenguajes de programación.

formato X. Formato de archivo para modelos 3D definido por Microsoft y propio de la *API* de *Direct3D*. Este formato tiene dos especificaciones, una de texto y la otra binaria. Mientras que las binarias son convenientes para fines de distribución, los de texto permiten al desarrollador entender el formato y crear propios cargadores de archivos para ser implementados en otras plataformas o APIs.

frustum clipping. Recorte de la escena vista en perspectiva.

FVF. Flexible Vertex Format. Una variable de tipo *double word* (*DWORD*) en la que se dice por cuales componentes se encuentra formado un vértice. Es decir, si el vértice está especificado en coordenadas XYZ sin transformar, ya transformadas, si cuenta con coordenadas de textura (bidimensionales o tridimensionales), la normal, componentes difusa, especular, etc.

GDI. Graphics Device Interface. Interfaz de Dispositivos Gráficos. Es el estándar de MS Windows para representar objetos gráficos y transmitirlos a los dispositivos de salida, como monitores e impresoras.

Gouraud *shading*. Uno de los algoritmos más usados para sombreado suave. El nombre proviene del apellido de su creados, el francés Henri Gouraud. Esta técnica es basada en la interpolación de color a lo largo de las caras de todos los polígonos para determinar el color de cada píxel.

GPU. Graphic Processing Unit. Unidad de Procesamiento Gráfico. Encargada del proceso de transformación e iluminación, dejando que el CPU se enfoque a otro tipo de tareas.

HAL. Hardware Abstraction Layer. Capa de Abstracción de Hardware. Componente que da soporte a plataformas específicas de hardware como al *kernel* del sistema, al manejador de entrada-salida, los manejadores de dispositivos, etc. La HAL contiene rutinas que extraen características del hardware (caché, buses de I/O, interrupciones, etc). Asimismo, define una interfaz entre el

hardware de la plataforma y el sistema operativo.

HEL. Hardware Emulation Layer. Capa de Emulación de Hardware. Interfaz definida por *DirectDraw* que permite emular características de hardware no presente en software. Provoca que las aplicaciones se ejecuten de manera lenta y por ello es a menudo conocida como HELL.

HMD. Head Mounted Display. Cascos de Realidad Virtual. Periférico que coloca pequeñas pantallas ante cada ojo del usuario de manera que tenga inmersión en un mundo tridimensional.

joystick. Dispositivo de control usado en juegos de computadora. Obtuvo este nombre a partir del control empleado por los pilotos de aeronaves para controlar los alerones y elevadores de las mismas.

MIDI. Musical Instrument Digital Interface. Interfaz Digital de Instrumento Musical. Estándar adoptado por la industria electrónica de música para controlar dispositivos, como sintetizadores y tarjetas de sonido que emiten música.

morphing. Transformación animada en la que un objeto se convierte en otro. Esto se logra mediante una deformación de los vértices del primero hasta que coinciden algunos de ellos con los del segundo.

MP3. MPEG Layer 3. Es uno de tres esquemas de codificación para la compresión de señales de audio. Usa codificación de audio perceptual y compresión psico–acústica para remover información superflua para el oído humano. Además implementa un filtro basado en la transformada discreta de coseno modifica con lo que se incrementa la resolución en frecuencia hasta 18 veces mas que en Layer 2. El resultado es música digital con calidad de CD con una compresión de un factor de 12.

MPEG. Moving Pictures Expert Group. Uno de los formatos pertenecientes a la familia de estándares para compresión de video desarrollados por el grupo. *MPEG* generalmente brinda una calidad de video que otros formatos competidores, tales como Video for Windows, Indeo o QuickTime. La decodificación puede llevarse a cabo mediante software o hardware.

multisampling. Es un método de *antialiasing* en el que un almacenamiento de píxeles adicionales, se mantiene como parte de los *buffers* de color, profundidad y *stencil*; lo anterior con el fin de establecer ciertas máscaras que ayudarán en el proceso de mantener la información del almacenamiento de píxeles actualizada.

NDC. Normal Device Coordinates. Coordenadas de Dispositivo Normalizadas.

OpenGL. API de gráficos por computadora mas popular en el área. Desarrollada por *Sillicon Graphics Industries* y destinada al uso de operaciones específicas y objetos para producir aplicaciones interactivas en 3 dimensiones.

pasada. Entiéndase este término como las veces que necesitan ser aplicadas las transformaciones e iluminación sobre los vértices que componen una geometría antes de ser dibujada.

phase. Instrucción para *píxel shader* versión 1.4. Permite descomponer el *shader* en dos secciones, con la finalidad de poder usar un mayor número de instrucciones.

pin. Puntos de conexión en un diagrama. El término es principalmente usado en el area eléctrica, pero debido a su generalidad es usado también en el ámbito de la computación para indicar entradas y/o salidas.

pipeline. Serie de etapas en la que se que efectúa una tarea determinada en varios pasos, al igual que en una línea de ensamblaje de una fábrica.

plug-in. Pieza de software que, añadida a cierto paquete, permite hacer

nuevas acciones que originalmente no estaban presentes.

raster. El proceso de dibujar una imagen usando la tecnología con la que funciona la televisión. Involucra dibujar píxel por píxel de la imagen barriendo su totalidad, primero de manera horizontal y enseguida vertical.

render. Verbo del inglés proveniente del francés *rendre*, cuyo significado es el de 'regresar' o 'rendir cuentas'. Tiene diferentes significados dependiendo del ámbito en el que se use. En términos de graficación por computadora es usado para referirse a dibujar, hacer un bosquejo, o el mostrar algo en papel u otro medio.

RGBA. Formato de representación para color de 32-bit. Cada componente (roja, verde, azul y el canal alfa) está compuesta por 8 bit.

script. Una lista de comandos que pueden ser ejecutados sin interacción del usuario.

SIMD. Single Instruction Flow, Multiple Data Flow. Arquitectura de computadoras paralelas que caracteriza a la mayoría de las computadoras vectoriales. Una instrucción causa que mas de un procesador realice la misma operación de manera síncrona, aun con diferentes datos.

skew. Posición oblicua de un objeto cualquiera.

skinning. Una técnica para producir nimaciones fluidas y suaves basadas en cuadros de animación por huesos.

stencil buffer. Memoria que es usada para hacer pruebas por fragmentos en conjunto con el *buffer* Z.

stencil test. Prueba de Stencil. Esta prueba es usada para enmascarar regiones, tapar sólidos geométricos y generar transparencias entre polígonos.

T&L. Transformation and Lighting. Transformación e Iluminación. Proceso importante en el pipeline de dibujo, ya que es donde los vértices son transformados al espacio de recorte e iluminados.

tecnología *'see-through'*. Los *HMD* empleados se encuentran compuestos por un par de monitores en los que se muestra el video adquirido de la escena real.

texel. Abreviatura de Elementos de Textura (*TEXture ELements*) que por lo general se refiere a una componente de textura en formato RGB o ARGB.

textura. Representación digital de la superficie de un objeto. Además de sus propiedades bidimensionales (como color y brillantez), una textura es también codificada con propiedades tridimensionales como las capacidades de transparencia y reflexión del objeto.

token. Unidad básica e indivisible de un lenguaje. Es la unidad independiente más pequeña y con significado en un programa, especificada por un analizador léxico o sintáctico. Un *token* puede ser datos, una palabra reservada del lenguaje, un identificador u otras partes que conforman la sintaxis del lenguaje.

tracking. Seguimiento de un objeto a partir de marcas o rastros que lo identifican. En este caso, se refiere al hecho de orientar al objeto de acuerdo a la pose con que lo observa la cámara.

tweening. Técnica de interpolación entre dos o mas cuadros para generar animaciones.

VCM. Video Compression Manager. Manejador de Compresión de Video. Permite el acceso a la interfaz empleada por los *codecs* para el manejo de datos en tiempo real.

VFW. Video For Windows. Formato desarrollado por Microsoft para almacenar información de audio y video. El formato empleado es el AVI, estando limitado a una resolución de 320 x 240 píxeles y 30 cuadros por segundo. Ninguna de las especificaciones permite la reproducción a pantalla completa.

VRML. *Virtual Reality Model Language*. Especificación para el diseño e implementación de un lenguaje independiente de plataforma que permita describir escenas de Realidad Virtual.

WAV. Formato de audio creado por Microsoft e IBM.

WDM. Windows Driver Model. Una tecnología desarrollada por Microsoft para la creación de drivers y cuyo código es compatible con Windows 98, 2000, Me y XP. Trabaja canalizando parte del trabajo del driver del dispositivo en porciones de código que son integradas en el sistema operativo. Estas porciones de código manejan todas las funciones de memoria de bajo nivel como son el *DMA* y *Plug and Play*.

wireframe. Representación por modelo de alambres. Todos los vértices por los que se encuentra formado el modelos son unidos con líneas que le dan una apariencia de estar formado con alambre entretejido.

wrap. Modo de mapeo de textura en donde esta es dibujada tantas veces como quepa en los vértices que se índico se mapeara.