
UNIVERSIDAD DE BELGRANO

# Las tesinas de Belgrano

Facultad de Tecnología Informática
Ingeniería en Informática

Pipeline 3D para rendering en tiempo real
con capacidad de detección de movimiento

N° 468 Lucas Matías Daniel Madariaga

Tutora: Alicia Barón

Departamento de Investigaciones
2010

Universidad de Belgrano
Zabala 1837 (C1426DQ6)
Ciudad Autónoma de Buenos Aires - Argentina
Tel.: 011-4788-5400 int. 2533
e-mail: invest@ub.edu.ar
url: http://www.ub.edu.ar/investigaciones

# Índice

# 1 - Introducción al trabajo final de carrera

En esta sección introductoria se hablará del mundo de los videojuegos, los cuales motivan la creación de este proyecto, el valor académico de su estudio e implementación así como su importancia e influencia en la actualidad. Por último se definen los objetivos y alcance del trabajo final de carrera.

## 1.1 - Introducción

Los videojuegos son un campo de la informática que ha generado grandes debates, están aquellas personas que cuestionan la valía del producto final, aquellas que los consideran programas mal realizados que acaparan casi totalmente la capacidad computacional de un sistema, aquellos que opinan que son una vía de escapismo, un simple juguete para niños, etc.; y estamos las personas que creemos que es una forma de entretenimiento que está ganando la misma validez que las formas más clásicas.

Lo que no se puede negar, es que el desarrollo de videojuegos tiene un gran valor académico ya que, es un área de la informática de gran complejidad y amplitud. Con el avance continuo del que es testigo la industria se han pasado de simples juegos con una interfaz de texto, a gráficos en 2D y música midi, a superproducciones con gráficos producidos en tiempo real que por momentos pueden parecer foto-realistas y bandas de sonido orquestales que, realmente ponen a prueba el poder de procesamiento de nuestras PCs.

“Una computadora capaz de ejecutar juegos sofisticados es una computadora que es capaz de realizar casi cualquier tarea que se le asigne. Un buen juego de computadora demanda al máximo a nuestras computadoras, incluyendo su capacidad de procesar datos rápidamente, generar gráficos y animación, y crear efectos de sonido realistas. Solo las computadoras que, representan el estado del arte, pueden mantenerse a la par de los juegos de hoy; como lo son simuladores y juegos de acción en 3D. Es más hay pocas aplicaciones de negocios en existencia que requieran más poder computacional que un juego de computadora sofisticado” [1]

Ahora, ¿por qué se habla de valor académico? Un juego moderno abarca temáticas de computación grafica, desde fuentes, texturas a utilizar, creación de menús, hasta motores con capacidades 2D o 3D. Un juego reproduce sonido y música, contiene rutinas de inteligencia artificial, debe registrar y responder a los comandos del usuario, etc. Vemos que es un área de gran diversidad multidisciplinaria en la cual hoy en día se usan equipos de desarrollo formados por un gran número de personas para llevar a cabo un proyecto. A su vez la creación de juegos retroalimenta las habilidades en otros campos:
“Un programador que puede escribir juegos de computadora de calidad comercial puede escribir casi cualquier otro tipo de software, especialmente considerando el foco de hoy hacia gráficos y sonidos en aplicaciones.”[1]

“Los videojuegos son piezas de software extremadamente complejas. Es más, son sin lugar a dudas los programas más difíciles para escribir. Seguramente escribir algo como MS Word es más difícil que escribir un juego tipo Asteroids, pero escribir algo como Unreal, Quake Arena o Halo es más difícil que cualquier programa del que pueda pensar, incluyendo software militar para el control de armas!”[2]

Todos estos cambios evolutivos en la industria se han dado en un lapso no mayor a 40 años, estamos ante una industria con un ciclo de vida aceleradísimo en cuanto a mejoras técnicas, y un crecimiento realmente impactante en cuanto al público que atrae y la economía que maneja.

Para ejemplificar un poco estas afirmaciones Doug Lowenstein presidente de ESA (Entertainment Software Association) dice que la industria de videojuegos en EEUU tenía un valor de U$S 10.3 billones en 2004, y que generó un valor económico de U$S 7.7 billones. Aun así, estos números no muestran la influencia que la industria de los videojuegos tuvo en mejorar el rendimiento de la industria norteamericana en general. Por ejemplo los procesadores Cell creados conjuntamente entre Sony, IBM y Toshiba, usados en las consolas Playstation 3 también se utilizan en maquinas de mapeo de imágenes medicas. Los militares estadounidenses utilizan juegos a modo de entrenamiento, y también ayudan en el incremento de ventas de teléfonos móviles, banda ancha y redes hogareñas. Lowenstein dice “las compañías

como Verizon y Comcast aman la industria". Los videojuegos son como el rock 'n' roll para las nuevas generaciones: Halo y los Sims son como The Grateful Dead y los Rolling Stones. [3]

Visto el crecimiento e importancia que conlleva la industria de los videojuegos en la actualidad, ¿Cómo se crean éstos? ¿Qué es lo primero que se debe hacer o poseer para entrar en esta actividad? La respuesta es simple y muy compleja a la vez: un motor de juegos. Un engine de juegos es un sistema de software que abstrae los detalles de implementación de ciertas tareas comunes en juegos, algunas de estas son el rendering, que puede ser en 2D o 3D, detección de colisiones, sonido, scripting, animación, inteligencia artificial, streaming, manejo de memoria, comunicación en red, cálculos de física, threading, el manejo del input, etc. Con esto se logra que los desarrolladores se enfoquen en la tarea de crear el juego y no estancarse en detalles tales como comunicarse con los drivers de la placa de video o el mouse para lograr comunicarse con estos dispositivos. [4]

Los motores ofrecen componentes reutilizables lo que permite usar el mismo motor para la realización de diversos proyectos. Esto genera grandes ahorros en un estudio de desarrollo. Los engines proveen un conjunto de herramientas desarrollo visual como así también componentes de software,, que permiten el desarrollo veloz de juegos. Este tipo de engines a veces se llaman "middleware para juegos" ya que proveen una plataforma de software flexible y reusable con la funcionalidad necesaria para desarrollar una aplicación de juegos mientras se reducen costos, complejidades, y tiempo de mercadeo (time-to-market), todos factores críticos en la altamente competitiva industria de los video juegos.[5]

Vistos los componentes de un motor de videojuegos, se concluye que el mismo es una tarea demasiado compleja para desarrollar individualmente. Por lo tanto se decide implementar uno de los componentes más importantes:. el motor gráfico 3D.Se desarrollaran ciertas capacidades para responder a los comandos del usuario mediante dispositivos tradicionales, como el teclado, y con técnicas basadas en la captura de movimientos mediante una webcam.

Sin lugar a dudas la creación de un engine es una tarea muy vasta, una que se lleva a cabo desde la pasión, y que requiere extensas horas de trabajo. Pero que, a su vez, llena de satisfacción y orgullo cuando se pueden ver los frutos del trabajo. Para alguien como yo, que a lo largo de su carrera académica no ha tenido experiencia laboral, siento que este proyecto es la preparación final para afrontar el desafío de cualquier desarrollo que uno pueda enfrentarse en el futuro.

### 1.2 - Objetivos y Alcance

A partir de un gran interés personal por la temática de los videojuegos y del valor de la misma como herramienta de aprendizaje es que esta tesina tuvo sus inicios. Habiendo determinado que la creación de un motor para videojuegos es una tarea demasiado compleja. Se ha decidido que el objetivo de este trabajo sea la investigación e implementación de un pipeline gráfico 3D para videojuegos cuyo manejo se abstrae al usuario mediante un motor 3D. Como ya se ha mencionado el motor también podrá responder al input del usuario para garantizar la interactividad del mismo con métodos convencionales y otros de captura de movimientos.

Hablamos de pipeline 3D para videojuegos ya que este tipo de pipeline posee diferencias, algunas sutiles otras no tanto, con respecto a otras implementaciones de motores 3D. El pipeline 3D para videojuegos requiere realizar rendering en tiempo real, para lo cual se utilizara como medio de rendering un rasterizador. La rasterización es uno de los métodos menos precisos para generar imágenes a partir de la geometría de una escena. Aun si los resultados pueden ser visualmente convincentes, la matemática empleada utiliza simplificaciones y generalizaciones en pos de la velocidad. A la simplificación propia del dibujo mediante rasterización se le suman variadas simplificaciones que se llevarán a cabo en el motor especializado. El principal objetivo del motor es obtener un rendimiento que permita la interacción en tiempo real por lo tanto se llevarán a cabo simplificaciones y técnicas de optimización en varias etapas a fin de cumplir este objetivo. Una diferencia más sutil por ejemplo, es qué un motor utilizable en CAD implementa una proyección que no tiene en cuenta la perspectiva desde el punto donde se observa la escena. Esto es una característica deseable cuando se está generando un plano o prototipo de un objeto. En cambio, en un motor para películas o juegos se utiliza una proyección que si tiene en cuenta la perspectiva a modo de simular el punto de vista de una cámara. Otra posible situación se puede dar si ciertas aplicaciones

requieren que sus datos se almacenen en variables de mayor precisión a las utilizadas. Varias de estas diferencias son fácilmente solventables y podrán ser futuros métodos de expandir el motor así como que se puede dar más de un uso al mismo a pesar de las simplificaciones utilizadas en pos de la velocidad..

El motor 3D es un middleware que abstrae métodos para la creación de gráficos 3D, pero no se encarga de generar los modelos 3D que conformarán las escenas, salvo ciertas excepciones. Tampoco se encarga de la creación y manejo de ventanas pero si posee los métodos que configurarlas una vez creadas para poder dibujar en ellas. A su vez el motor no poseerá una interfaz gráfica ni modificación en tiempo real de los componentes de la escena, se podrá configurar la misma pero dentro del software del programa. El desarrollo del proyecto se ha basado en la creación de los algoritmos gráficos que permiten dibujar y visualizar las escenas pero no en la maquetación de las mismas. En consecuencia a los explicado anteriormente tampoco podrá ser almacenado el estado de la escena y de los objetos que lo componen durante la ejecución ni en el cierre de la misma pero, si se podrá interactuar con la escena utilizando el teclado, y la captura de movimientos vía webcam. Habiendo estipulado los limites de las capacidades del motor 3D se definen los objetivos de este trabajo final de carrera.

**Objetivo 1:** Debido a que la creación de un motor gráfico es un tema de gran alcance y complejidad, sobre el cual la mayoría de la bibliografía se encuentra en ingles. He decidido presentar en esta tesina todos los conceptos necesarios para que al finalizar la lectura el lector tenga una comprensión de que componentes son necesarios y como se comportan a la hora de crear un sistema grafico. Esto permitirá a los interesados en el tema tener una sólida base de conocimientos sobre las tareas que deberán realizar en caso de decidir implementar un sistema gráfico, con el valor agregado de tener esta información recopilada en su idioma nativo lo cual es en este ámbito de la informática es poco común. Los conceptos intentaran ser presentados de la manera más sencilla posible y tratando de evitar detalles matemáticos y de implementación.

**Objetivo 2:** Con los conocimientos adquiridos mediante la investigación, implementar un pipeline y un motor grafico propio y, unirlo a un sistema de control. El sistema de control además de contar con el clásico input por teclado, poseerá algoritmos de computación visual que permiten la captura de movimientos a fin de tener un método alternativo control.

# 2 - Marco Teórico

En el marco teórico se verá la definición del concepto de rendering y los tipos de rendering más comunes que se utilizan para generar imágenes 3D. Luego se verá con más detalle que es un pipeline gráfico y qué etapas conceptuales lo conforman. Seguido de esto se verá como las aceleradoras de hardware modernas implementan el pipeline gráfico. Se verá que existen ciertas aplicaciones que simplifican el uso del hardware gráfico llamadas APIs y se estudiarán las 2 más utilizadas: DirectX y OpenGL. Por último, una vez vistos los métodos y herramientas actuales utilizados para generar gráficos 3D en tiempo real, se presentará a la suite OpenCV cuyas capacidades permiten la implementación de medios alternativos para controlar aplicaciones. Por lo tanto se explicará qué es la computación visual y en qué consiste OpenCV. En base a los componentes que conforman un pipeline 3D y la suite de visión de computadoras se definirán las funcionalidades básicas que debe tener el motor a implementar y cuál es el aporte que se busca brindar con el proyecto.

## 2.1 - Rendering

El rendering es el proceso de generar imágenes a partir de modelos tridimensionales, la transformación de una escena 3D a una imagen 2D se lleva a cabo por el pipeline gráfico. Las operaciones de rendering pueden llevarse a cabo en tiempo real cuando se requiere interactividad con la escena presentada, el caso más común son los videojuegos. El proceso también puede durar una gran cantidad de tiempo cuando se utilizan técnicas de rendering computacionalmente intensivas como las necesarias para crear una película de animación 3D. A este tipo de rendering se lo conoce como pre-rendering o rendering off-line.

Idealmente para renderizar una escena se trazaría cada partícula de luz de la misma para que la misma este perfectamente iluminada. Esta técnica suele ser imposible de aplicar ya que requiere de un tiempo

de procesamiento decididamente alto. Es por ello que se han creado diferentes técnicas para modelar la iluminación de manera más eficiente, a continuación vemos las 4 más comunes.

### 2.1.1 Rendering por scanlines o rasterización

Las técnicas de rasterización se utilizan cuando renderizar pixel por pixel (orden de imagen) es poco práctico y demasiado lento para la tarea que se desea realizar. La rasterización se basa en la renderización primitiva por primitiva (orden de objeto), en la misma se recorren todas las primitivas determinando que pixeles de la imagen son afectados por las mismas y modificando a los pixeles en base a ello. Esta es la técnica utilizada por las tarjetas gráficas en la actualidad, ya que permite que las imágenes se procesen con suficiente rapidez como para permitir que el usuario interactúe con las imágenes generadas. Este incremento de velocidad se gana en sacrificio de precisión, los enfoques pixel por pixel son capaces de producir imágenes de mayor calidad además de ser más versátiles ya que no dependen tanto de las presunciones y simplificaciones de la imagen como lo hace la rasterización.[23]

### 2.1.2 Ray casting

En el ray casting la geometría de una escena es analizada pixel por pixel, línea por línea, desde el punto de vista de la cámara. Cuando se interseca a un objeto se determina el color del mismo. La determinación del color puede realizarse de varias maneras con distintos niveles de complejidad. Por ejemplo, se puede decidir que el color del polígono sea el color que ya tiene almacenado sin que el mismo sea afectado por otros componentes de la escena, o se puede realizar un cálculo con un factor de iluminación pero sin calcular la relación del objeto con las posibles fuentes de luz simuladas. El ray casting es utilizado principalmente en simulaciones de tiempo real, pero las imágenes generadas por los mismos suelen tener un aspecto plano si no se utilizan otras técnicas para simular detalles. [23]

### 2.1.3 Ray tracing

El Ray tracing busca simular el flujo natural de la luz, interpretando al mismo como partículas. Cuando se utiliza ray tracing usualmente se emiten múltiples rayos por cada pixel a dibujar y se traza el recorrido de los mismos no solo hasta la primer intersección con un objeto, sino también a través de una cantidad de rebotes basados en las layes de la óptica, las refracciones y la dureza de las superficies afectadas.

El trazado de rayos en general ha sido demasiado lento no solo para aplicaciones en tiempo real sino también para la creación de películas de animación 3D. Su principal uso ha sido en el área de efectos especiales, o publicidad. En la actualidad existen prototipos de aceleradoras de ray tracing por hardware para lograr realizar la técnica de trazado de rayos en tiempo real, estas aceleradoras a futuro reemplazarán a las tarjetas gráficas basadas en rasterización. [23]

### 2.1.4 Radiosidad

La radiosidad es un método que simula cómo superficies que son iluminadas directamente funcionan como fuentes de luz indirectas para otras superficies produciendo un sombreado más realista. Los cálculos de radiosidad son independientes de donde se observa la escena lo que incrementa la complejidad de los cálculos, pero permite utilizar distintos puntos de vista sin utilizar la necesidad de recalcular. Si no hay grandes cambios en una escena se pueden usar los valores ya calculados durante varios cuadros de animación, La radiosidad no se suele utilizar como técnica de rendering, sino que complementa a las otras técnicas para generar un modelo de iluminación más detallado y realista. [23]

## 2.2 El pipeline grafico

El pipeline grafico (graphics rendering pipeline) es el conjunto de funciones qué, provistas de una representación de una escena, a través de su listado de objetos, luces, cámaras, etc.; genera una ima-

gen raster de dos dimensiones como resultado. El pipeline es necesario tanto en pre-rendering como en rendering en tiempo real. El rendering en tiempo real se utiliza cuando se desea poder interactuar con el universo virtual creado, y su principal uso son los juegos de video. La tasa con la que se presentan las imágenes en pantalla generalmente se mide en cuadros por segundo se dice que una aplicación trabaja en tiempo real cuando en promedio, la misma dibuja 15 cuadros por segundo. El termino de pipeline grafico también se asocia a los métodos más actuales de rasterización soportados por el hardware grafico. Debido a su capacidad de abstraer al hardware gráfico dos de los modelos de pipeline grafico más utilizados son los incluidos en OpenGL y Direct3D.

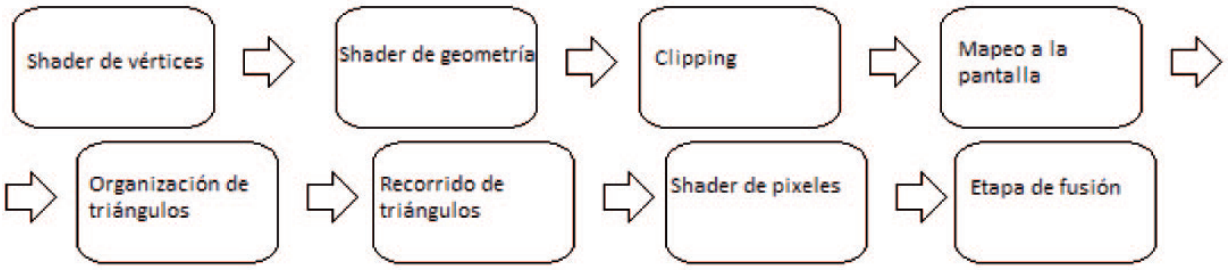
La arquitectura de un pipeline de tiempo real puede ser dividida en tres etapas conceptuales: aplicación, geometría y rasterización. La etapa de aplicación se lleva a cabo por software, en ella se suele configurar la escena, y cargar o crear los componentes que la conforman. También se implementa la lógica del programa, se lleva a cabo la animación si está presente, se responde al input del usuario, se utilizan algoritmos de aceleración, etc. Al finalizar se lleva a cabo la operación más importante de esta etapa alimentar la geometría a ser dibujada al pipeline. La geometría es el conjunto de primitivas que conforman la escena y que pueden eventualmente ser dibujadas en la imagen final.

En la fase de geometría se llevan a cabo la gran mayoría de las operaciones que se realizan sobre los polígonos o vértices, se computa lo que se debe dibujar, cómo se lo debe dibujar y donde se lo debe dibujar. Esta fase conceptual puede, a la hora de la implementación, estar conformada por una o múltiples etapas de acuerdo a como se decida llevar a cabo el desarrollo. Por último se encuentra la fase de rasterización donde una vez que las primitivas han terminado su recorrido por las etapas anteriores, se crea la imagen final para ser mostrada en pantalla. Es decir, se convierte a los elementos de la escena en pixeles y se les da color a los mismos. [2][11]

### 2.3 La unidad de procesamiento grafico

Históricamente el hardware de aceleración grafica implementaba la etapa última etapa conceptual del pipeline, la rasterización de triángulos. En las generaciones siguientes se han ido implementando cada vez más elementos de las etapas previas, tal es el caso que hoy en día ciertos algoritmos de alto nivel de la fase de aplicación se llevan a cabo por hardware. En los últimos años las unidades de procesamiento grafico (GPU) han pasado de ser implementaciones configurables de un pipeline con funcionalidades fijas, a procesadores altamente programables donde los desarrolladores pueden implementar sus propios algoritmos.

La programación de la GPU se lleva a cabo a través de varios tipos de shaders, el shader de vértices permite operaciones como transformaciones y deformaciones en cada vértice. Los shaders de pixeles permiten calcular complejas ecuaciones de sombreado sobre cada pixel individual. Por último los shaders de geometría permiten que la GPU cree y destruya primitivas geométricas sobre la marcha. Los valores calculados en estas etapas pueden ser almacenados en buffers de alta precisión y reutilizados como datos para vértices o texturas. Por razones de eficiencia algunas fases del pipeline siguen siendo configurables pero no programables aun así, la tendencia es hacia la flexibilidad y la programabilidad. [7][16] Las aceleradoras gráficas de la actualidad soportan las APIs DirectX, OpenGL como medio para simplificar la explotación de las capacidades de las GPUs. A continuación vemos el pipeline de la unidad de procesamiento grafico



El pipeline tiene la característica de estar dividido en etapas totalmente programables: los shaders de vértices, geometría y pixeles. Etapas configurables pero no programables: clipping y la etapa de fusión. Y por ultimo etapas que por cuestiones de rendimiento tienen su operación fijada: el mapeo a pantalla, la organización y recorrido de los triángulos. Cabe destacar que las etapas funcionales analizadas con anterioridad tienen correspondencia con el pipeline, la etapa de aplicación no aplica ya que como se ha mencionado se lleva a cabo por software y es anterior a las fases presentadas en esta sección. La etapa de geometría corresponde al shader de vértices, el shader de geometría y el clipping. Mientras que la etapa de rasterización corresponde a las fases restantes del motor: mapeo a pantalla, organización de triángulos, recorrido de triángulos, el shader de pixeles y la fusión. [11][16]

### 2.3.1 El shader de vértices

Una malla de triángulos está formada por un conjunto de vértices y la información que indica cuales vértices forman cada triangulo, el vertex shader es la primera etapa donde se procesa la malla de vértices. En esta fase solo se procesaran los vértices. para los cuales se pueden modificar, crear o ignorar los valores de sus normales, color, posición y coordenadas de textura. En general se transforman los vértices desde el espacio de modelo hasta un espacio de clipping homogéneo, esta es la operación mínima que debe realizar el shader de vértices.

En esta fase no se pueden crear ni destruir vértices y los resultados generados en un vértice no pueden ser utilizados por otro. Como cada vértice es tratado independientemente, se puede utilizar procesadores de shaders en paralelo dentro de la GPU para acelerar esta etapa. [11][16]

### 2.3.2 El shader de geometría

Esta etapa es una de las más recientes en ser implementadas, su uso es opcional pero, si se desea que la implementación este adherida a ciertos estándares modernos sobre el uso de shaders la misma debe ser utilizada. Los datos de entrada de esta fase son un solo objeto y su lista de vértices asociados. El shader de geometría procesa las primitivas del objeto y sus datos de salida son cero o más primitivas. Mediante este método la malla que conforma el objeto puede ser modificada selectivamente editando sus vértices, añadiendo nuevas primitivas y eliminando otras.

El shader de primitivas debe garantizar que los resultados obtenidos son entregados en el mismo orden que fueron recibidos, similar al comportamiento de un cola, esto afecta al rendimiento ya que si se utilizan una serie de shaders de geometría en paralelo los resultados deben ser almacenados y ordenados para respetar el orden. Como un compromiso entre capacidad y eficiencia se ha limitado el numero de valores generados por ejecución a 1024. Esto permite que se lleve a cabo una modificación programática de los datos recibidos así como realizar un número limitado de copias pero impide la replicación o amplificación masiva de estos datos. Es decir, en esta etapa no se puede utilizar un modelo de montaña para simular una cordillera ni se puede triangular a los objetos mediante teselación para añadir más detalle a los mismos. [11][16]

### 2.3.3 Stream output

El stream output es una funcionalidad añadida y no una etapa del motor en sí, la misma se puede utilizar luego de los shaders de vértices o geometría. El flujo normal de datos en un pipeline consiste en procesar los vértices, rasterizarlos y, luego trabajar con los pixeles resultantes mediante los shaders de pixeles. Antiguamente durante este recorrido por el pipeline los datos intermedios no podían ser accesados, el objetivo de esta funcionalidad es permitir el almacenamiento de los datos luego de las operaciones de shading de vértices o de geometría en un stream (arreglo ordenado). De esta manera los datos del stream pueden ser reenviados al pipeline permitiendo el procesamiento iterativo, lo cual es útil para la simulación de efectos de partículas , como el flujo del agua. Otra capacidad interesante que provee esta funcionalidad es que, al poder almacenar los datos, se puede desactivar el rasterizador transformando al pipeline en un procesador de datos sin capacidades graficas. Permitiendo de esta forma utilizar las aceleradoras gráficas como unidades de procesamiento general, este método se conoce como GPGPU (general purpose graphical procesing unit). [11][16]

### 2.3.4 Clipping

Solo las primitivas que se encuentren total o parcialmente dentro del volumen visual de la cámara necesitan ser enviadas a la etapa de rasterización. Las primitivas que estén parcialmente dentro del volumen visual deberán ser recalculadas para no salir del mismo. Si se traza la línea entre un vértice interno y otro externo de la primitiva se debe reemplazar el valor del vértice externo por el por el valor de la intersección entre la línea y el volumen visual. [11][16]

### 2.3.5 Mapeo a pantalla

Una vez obtenidas solo las primitivas que se encuentran en el volumen visual, las coordenadas x e y de las mismas son transformadas a coordenadas de pantalla. Mientras que las coordenadas en z permanecen iguales, ya que se pueden llegar a utilizar en cálculos de profundidad en etapas posteriores. El conjunto de coordenadas en pantalla y las coordenadas z se conoce como coordenadas de ventana. [11][16]

### 2.3.6 Organización de triángulos

En esta etapa se calculan los gradientes, pendientes y otros datos relacionados a las superficies de los triángulos. Estos datos serán utilizados para el recorrido de triángulos e interpolación de datos y colores en las etapas siguientes. [11][16]

### 2.3.7 Recorrido de triángulos

Esta fase analiza los triángulos que forman parte de la escena, determina qué pixeles serán influenciados por los triángulos analizados. Este proceso se lo conoce como recorrido de triángulos o análisis de conversión (scan conversión). Las propiedades de cada pixel formado por fragmentos de un triangulo son generadas de los datos interpolados entre los 3 vértices del triangulo así como de la profundidad del mismo y de los datos de sombreado de la etapa de geometría. [11][16]

### 2.3.8 El shader de pixeles

Durante las etapas anteriores no se determina el color final de los pixeles que aparecerán en pantalla sino que genera información sobre como los triángulos, de los cuales son derivados los pixeles, cubren cada pixel, la cobertura puede ser total o parcial. Esta información se utiliza junto a las características de sombreado obtenidas de la etapa de iluminación, la transparencia u opacidad del triangulo, el color del mismo y el color de su textura (si tiene una asociada). Para que el pixel shader derive el color final y la profundidad de cada pixel, estos valores sumados a otros serán utilizados en la etapa de fusión para determinar el valor final de cada pixel. Una de las principales operaciones que se realiza en esta etapa es la de aplicar texturas si están disponibles.

Al igual que los shaders de vértices los shaders de pixeles solo pueden operar sobre el pixel recibido, esto es más problemático en esta etapa ya que ciertos pixeles aledaños podrían compartir los mismos valores lo que permitiría reducir considerablemente los cálculos. Por lo tanto, cuando se interpolen los datos de los vértices, de las texturas, y de otras posibles constantes almacenadas, el resultado solo afectara a un único pixel. [11][16]

### 2.3.9 La etapa de fusión

En esta fase se combinan las profundidades y colores de los fragmentos individuales para almacenar el resultado final en el framebuffer. Se llevan a cabo las operaciones de los buffers de esténcil y de profundidad así como las operaciones de mezcla de colores para generar efectos de transparencia. La etapa de fusión opera de una manera intermedia entre las etapas de funcionalidad fija y las de shaders programables. Es decir, el funcionamiento de esta fase no es programable pero si altamente configura-

ble. Por ejemplo se puede configurar la operación de mezcla de colores para que se realicen múltiples operaciones en la misma como pueden ser: multiplicación, adición y substracción tanto de los colores como de los valores alfa. [11][16]

### 2.4 Interfaz de programación de aplicaciones

La interfaz de programación de aplicaciones o API, consiste en un conjunto de funciones y procedimientos que permiten al desarrollador abstraerse de los detalles de implementación de las mismas a la hora de utilizarlas. Esto permite al programador enfocarse en la aplicación que desea crear y no en los subsistemas que necesita dicha aplicación. Las interfaces de programación de aplicaciones graficas han ganado gran importancia ya que permiten al programador accesar al hardware gráfico de manera abstracta, las 2 principales son: OpenGL y Direct3D (subconjunto de DirectX para producir gráficos interactivos en 3D). [24][25]

#### 2.4.1 DirectX

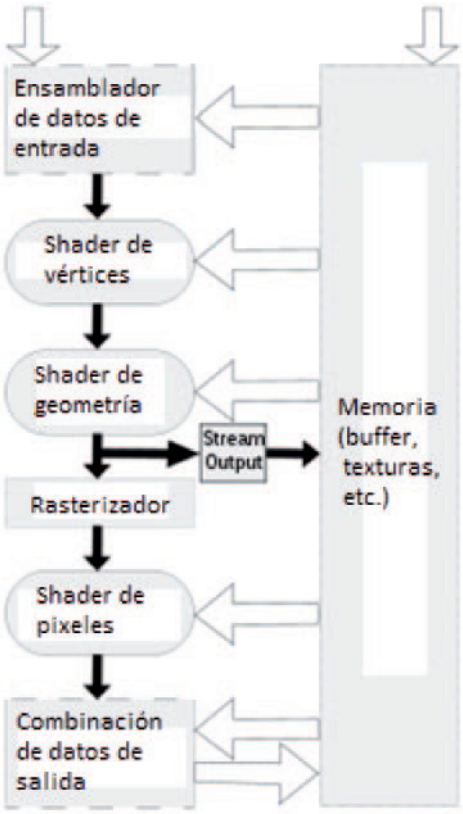
DirectX es un conjunto de APIs que facilitan la creación de aplicaciones multimedia, de video, de juegos de video, etc. Esta desarrollado por Microsoft y es compatible con los sistemas operativos Windows. Algunas de las APIs también son compatibles con las consolas de Microsoft Xbox y Xbox360. DirectX está formado por múltiples componentes que le permiten abstraer funcionalidades para la creación de gráficos 2D, 3D, utilizar múltiples monitores, responder al input del usuario y reproducir sonido, es decir que contiene los componentes necesarios para simplificar el proceso de creación de juegos. A través de sus iteraciones DirectX ha ido ganando funcionalidades en otros campos de la informática por ejemplo DirectCompute permite utilizar las aceleradoras de video como un procesador general, para acelerar aplicaciones cuyos cálculos son demasiado lentos en un CPU.

La última versión de DirectX disponible es la 11, la cual Microsoft promociona diciendo que permite crear videojuegos con mundos y personajes mas inmersivos y detallados. Para ello permite realizar teselación sobre la GPU, este método añade mas polígonos a los modelos dándole más detalle a los mismos. Se incluye una serie de instrucciones para simplificar el uso de multithreading en la CPU de modo que se aproveche al máximo la capacidad de los procesadores. Por último se puede utilizar DirectCompute para acelerar el procesamiento de juegos y aplicaciones, permitiendo que todos o parte de los cálculos se realicen sobre la placa de video. Es posible utilizar una tarjeta de video únicamente para acelerar programas que utilizan DirectCompute mientras otra tarjeta acelera los gráficos 3D. A continuación veremos con más detalle cómo funciona Direct3D que engloba el conjunto de funcionalidades para creación de aplicaciones 3D.[1]

##### 2.4.1.1 Direct3D

Direct3D ha sido desarrollado con el fin de abstraer la comunicación entre las aplicaciones gráficas y los drivers del hardware grafico. Trabaja en modo inmediato, por lo que provee una interfaz de bajo nivel a cada funcionalidad de la tarjeta gráfica entre ellas encontramos: transformaciones, clipping, iluminación, materiales, texturas, buffering de profundidad, anti aliasing, etc. El modo inmediato presenta la abstracción de 3 elementos principales: los dispositivos, recursos, y las cadenas de intercambio (swap chains). Los dispositivos son los encargados de dibujar la escena 3D, se proveen distintas capacidades según el tipo de dispositivo abstraído. Cada dispositivo utiliza datos, necesarios para el rendering, llamados recursos. Por último los datos rendereados se almacenan en un buffer trasero, el cual la cadena de intercambio presenta en pantalla cuando la imagen esta lista para mostrarse. [1]

#### 2.4.1.2 El pipeline de Direct3D




- Ensamblador de datos de entrada (Input Assembler): Lee de un buffer de vértices la información de la escena a dibujar. Estos vértices serán procesados en el pipeline.
- Shader de vértices: Opera sobre todos los vértices de la escena realizando operaciones como transformaciones, iluminación, y determinación de profundidad.
- Shader de geometría: procesa las primitivas provenientes del vertex shader permitiendo la generación de nuevas primitivas como líneas puntos y triángulos en base a las anteriores. Esto permite por ejemplo generar un modelo más complejo y detallado, evitando la transformación de vértices en la etapa anterior.
- Stream Output: Esta etapa se utiliza si se desea almacenar en memoria los resultados de las etapas anteriores, lo cual puede ser de utilidad para recircular datos por el pipeline.
- Rasterizer: Convierte los datos de las primitivas a pixeles, se puede realizar clipping en esta fase también.
- Pixel Shader: Determina el color final de los pixeles y calcula los valores de profundidad a ser almacenados en el buffer de profundidad.
- Combinación de datos de salida (Output Merger): La etapa final fusiona diferentes datos de salida (valores del pixel shader, valores de transparencia, valores de profundidad) construyendo la imagen final.

Las etapas del pipeline que utilizan shaders totalmente programables, la aplicación desarrollada debe proveer un programa con instrucciones para los shaders que describa las operaciones que se deben llevar a cabo. Varias de las etapas también son opcionales y pueden ser deshabilitadas si se desea. [1]

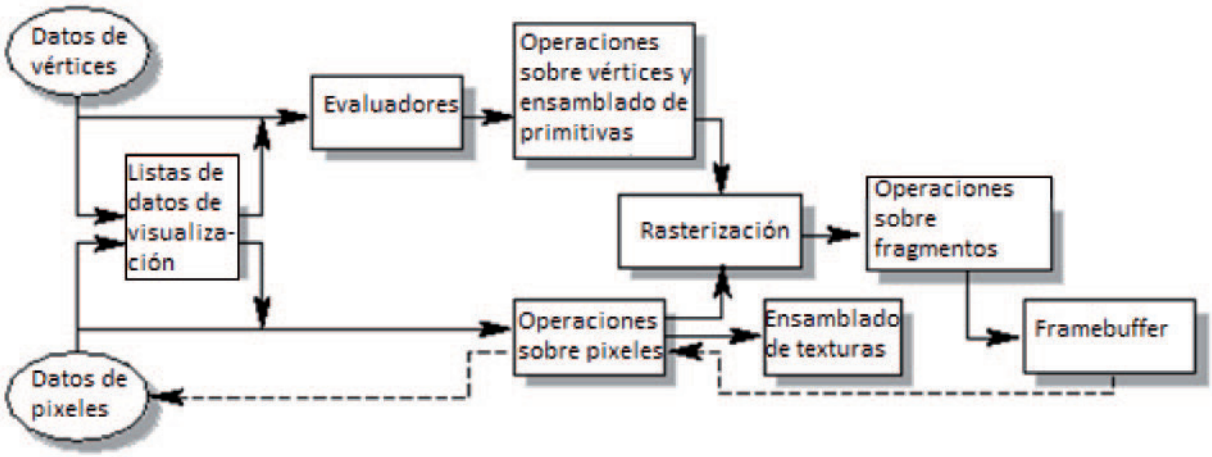
### 2.4.2 OpenGL

OpenGL (Open Graphics Library) es una interfaz de código abierto y multiplataforma para desarrollar aplicaciones graficas en 2 o 3 dimensiones. Posee más de 250 llamadas a funciones que permiten generar escenas tridimensionales complejas mediante el uso primitivas y llamadas a funciones simples.

Los principales objetivos de OpenGL son: esconder las complejidades de crear interfaces a diferentes aceleradoras 3D al presentar una interfaz única y uniforme al desarrollador. El otro objetivo es ocultar las diferencias entre distintas GPUs al requerir que el hardware compatible con OpenGL posea todas las funcionalidades que provee la interfaz.

La operación de OpenGL consiste en recibir primitivas, con el fin de convertirlas en pixeles, el pipeline que posibilita esto es conocido como “la máquina de estado de OpenGL”. Los comandos que el programador puede emitir permiten declarar primitivas y configurar las mismas para que sean procesadas por el pipeline. [8]

#### 2.4.2.1 El pipeline de OpenGL




- Listas de datos de visualizacion (Display Lists): Todos los datos, pixeles o geometría, se guardan en la lista para uso en el ciclo actual o posterior. Alternativamente se puede trabajar en modo inmediato donde los datos no se almacenan
- Evaluadores: Debido a que todas las primitivas geométricas eventualmente deben ser descriptas por sus vértices y, que ciertas estructuras como las curvas paramétricas y las superficies, pueden inicialmente estar descriptas por puntos de control y funciones polinómicas. Es que se utilizan los evaluadores para derivar vértices a partir de puntos de control.
- Operaciones sobre vertices: Se realizan las transformaciones sobre vertices. Si se usan funciones avanzadas es posible que los cálculos para aplicar texturas e iluminación se realicen en esta etapa.
- Ensamblado de primitivas: Se realiza el clipping, la eliminiacion de objetos y poligonos no visibles. Tambien se lleva a cabo la proyeccion de perspectiva y el mapeo a pantalla. A partir de estas operaciones se obtiene la lista completa de primitivas geométricas que se deben dibujar en pantalla, así como con su color y profundidad correspondiente ya definidos.
- Operaciones sobre pixeles: Las texturas a aplicar sobre los objetos se encuentran en memoria se leen los pixeles de las texturas. Los mismos son mapeados, escalados y modificados para coresponder a las posiciones de los objetos donde seran aplicadas en etapas posteriores. Los resultados vuelven a ser almacenados en memoria para aplicarlos usarlos en la etapa de rasterizacion.
- Ensamblado de texturas: Si se utilizan varias texturas en un objeto es conveniente almacenar las mismas en una estructura comun para poder intercambiar las mismas fácilmente.
- Rasterizacion: Es la conversión de los datos geométricos y de las texturas en fragmentos. Cada fragmento corresponde a un pixel en el framebuffer. En esta etapa se cargan los datos de las texturas para cada fragmento pero la asignacion final de color no se realizará hasta la etapa de operaciones sobre fragmentos ya que puede haber otras operaciones que influencien el color final.
- Operaciones sobre fragmentos: Antes de que los valores se almacenen efectivamente en el framebuffer un conjunto de operaciones se realizan que pueden alterar e incluso eliminar fragmentos. Todas estas operaciones pueden ser activadas o desactivadas. La primera operación que podemos encontrarnos es la aplicación de texturas. Luego se pueden aplicar cálculos de niebla, de transparencia, asi como

pruebas de profundidad y estencil. Tambien se pueden llevar a cabo operaciones lógicas contra una mascara de bits. Al finalizar la etapa tendremos el pixel final y su posición en pantalla.

- Framebuffer: se almacenan en la memoria del framebuffer los valores finales de cada pixel de la imagen y la misma es mostrada en pantalla.[8]

### 2.5 Sistemas de Input y OpenCV

Los sistemas de input se encargan de manejar las señales recibidas por los dispositivos de entrada por ejemplo: mouse y teclado. El motor deberá recurrir a uno de estos sistemas para poder captar las señales del teclado pero, utilizara a OpenCV para capturar imágenes de una webcam con el fin de utilizar partes del cuerpo u objetos cotidianos como medios de input alternativos. Antes de continuar con la presentación de OpenCV es necesario hablar de la computación visual explicar en que se basa y cuáles son los tipos de problemas que se encuentran en este campo de la informática.

#### 2.5.1 Visión de computadoras

La visión de computadoras se basa en transformar la información de una imagen o de cuadros de video en una decisión o una nueva representación, con intenciones de lograr algún fin en particular. A diferencia de los humanos cuando una computadora captura una imagen lo único que obtiene es una grilla de números, no tiene un contexto sobre que es la imagen o que ocurre en ella, ni que se debe hacer con la misma. Por ejemplo, si se captura una imagen de un avión, es tarea del desarrollador el implementar sistemas que ayudaran al equipo a interpretar ciertos valores de un mapa de bits como, el ala del avión.

Es fácil ver la complejidad de esta área de la informática, es más, no se puede crear una reconstrucción única de una señal tridimensional a partir del marco de referencia bidimensional que tenemos (foto, video). Esto permite decir formalmente que el problema es imposible de resolver, ya que no tiene una solución única ni definitiva, la imagen 2D puede representar un infinito numero de combinaciones de escenas 3D, aún si los datos capturados fuesen perfectos, cosa que no son ya que la imagen siempre contiene ruido y distorsiones generadas por el propio medio de captura y el ambiente.

Entonces, ¿qué métodos/información necesitamos para crear un sistema funcional? Para empezar, se puede añadir información contextual. Por ejemplo si se fuese a identificar un objeto sería útil poseer o poder inferir el tamaño del mismo así como un marco de referencia para identificar donde se suele encontrar el objeto y su tamaño relativo a otros objetos. A su vez, esta información se suele obtener de fotos en las que por lo general un fotógrafo centra y posiciona los objetos en formas que los humanos acostumbramos para identificar los objetos. Es por ello que se puede obtener mucha información implícita de este modo, lo que permite que permite que el sistema evite identificar erróneamente objetos que se encuentren en situaciones fuera de los contextos normales. Por ejemplo un sistema podría confundir un cartel publicitario con una imagen de una vaca por una vaca real, pero como la misma es mucho más grande que las vacas “convencionales” y se encuentra a varios metros del piso el contexto permite al sistema evitar el falso positivo.

Otro de los grandes problemas que nos encontramos es el ruido, el mismo se suele enfrentar con métodos estadísticos. Por ejemplo, puede ser imposible detectar un borde en una imagen comparándolo un punto a sus vecinos inmediatos, pero si tomamos las estadísticas de una regional local el proceso se ve altamente simplificado. Un borde será identificado como un conjunto de vecinos inmediatos, que tienen una orientación y características consistentes. Debido a que el ruido es un fenómeno variable, se puede compensar el mismo tomando estadísticas sobre un rango de tiempo. Otras técnicas en cambio, compensan el ruido construyendo modelos explícitos aprendidos de los datos disponibles, por ejemplo, es simple corregir la distorsión que generan los lentes ya que la misma es bien comprendida y por lo tanto se puede modelar y compensar. En resumen, nos encontramos en un campo complejo, con multitud de posibles aplicaciones, y con varios problemas los cuales deben ser considerados a la hora de implementar estos sistemas. Por lo tanto una buena regla a tener en cuenta es: cuanto más restringido sea el contexto de la computación visual, más podremos depender de esas restricciones para simplificar el problema, y más confiable será nuestra solución final. [21][22]

### 2.5.2 OpenCV

OpenCV es una librería de código abierto, programada en C y C++ aunque contiene interfaces para ser usada por otros lenguajes, la misma está disponible en Linux, Mac OS y Windows. Tiene el objetivo de proveer las herramientas básicas que son necesarias a la hora de resolver los problemas de computación visual, en muchos casos basta con utilizar las funciones de alto nivel provistas para lograr la solución de un problema. Y, aun cuando esto no es posible, el uso y concatenación de las funciones disponibles permitirán solucionar la gran mayoría de los problemas a resolver en la computación visual.

La librería está diseñada para ser eficiente, y para poder ser utilizada en aplicaciones que requieren funcionar en tiempo real. A su vez provee una infraestructura simple de utilizar, la cual permite crear aplicaciones sofisticadas rápida y fácilmente. Contiene más de 500 funciones de CV así como una librería para el aprendizaje de maquinas, el cual es un área íntimamente relacionada con la visión de computadoras. [21][22]

### 2.6 Síntesis

En este recorrido más profundo por los componentes que conforman a los engines 3D y la computación visual. Comenzamos a vislumbrar los componentes necesarios para el desarrollo del proyecto así como la complejidad real del mismo. Es evidente que no todos los sub-sistemas de un motor 3D de producción comercial serán implementados, ya que la complejidad computacional de las operaciones por pixel es demasiado grande dado que no se utilizará hardware para la ejecución de los algoritmos. A su vez el pipeline constará de etapas configurables y otras fijas pero, no de etapas programables. Esta capacidad por más de dotar al desarrollador con más posibilidades a la hora de utilizar el motor requeriría un conocimiento del mismo mucho más intimo. Por lo tanto preferimos priorizar la facilidad de uso y hacer que el motor tenga un funcionamiento similar al de un software tipo caja negra. Los componentes que se desarrollarán e implementarán son: el motor gráfico, una estructura lógica que deberán seguir los programas que deseen utilizar el motor, y el sistema de input.

Se vislumbran las capacidades que deberá poseer el motor: crear objetos y un mundo tridimensionales, con un sistema de iluminación para obtener un mayor grado de fidelidad grafica, más un sistema de control basado en el uso del teclado y en movimientos del propio usuario. El utilizar este tipo de control, permite un grado de interactividad y de inmersión imposible con los clásicos mouse y teclado. A su vez el estudio de la computación visual permite lograr comprender algunos principios y algoritmos que utilizan las grandes compañías como Sony y Microsoft en sus sistemas de motion control: PlayStation Move y Kinect.

Por lo tanto el aporte que busca dar esta tesina, es aunar en un texto los conceptos básicos que necesitan ser conocidos y comprendidos a la hora de crear un motor gráfico de modo que las personas que demuestren un interés en esta área puedan tener una introducción a la misma sin tener que leer libros de miles de páginas, la gran mayoría de ellos en inglés, para poder introducirse en el tema. A su vez también se buscara presentar un sistema que incluye un motor grafico 3D junto a un sistema de control innovador, el cual podría ser utilizado como la base para variadas aplicaciones dentro de la computación grafica.

## 3 - Conceptos teóricos: Sistemas de coordenadas y rasterización

La creación del motor grafico, supone implementar una serie de tareas y funcionalidades que en conjunto se denominan el pipeline 3D. Presentaremos los conceptos teóricos de las funcionalidades implementadas en el motor gráfico. Esto se lleva a cabo con el fin de que el lector tenga un grado de entendimiento aceptable sobre qué métodos se deben implementar y cómo.

En este capítulo se describen los sistemas de coordenadas a los cuales se transformará al objeto con el fin último de obtener las coordenadas que indiquen donde dibujarlo en pantalla. Se verá el porqué de la necesidad de cambiar de un sistema de coordenadas al otro y cómo realizar la transformación. Finalmente se explicará cómo se lleva a cabo el proceso de dibujo en pantalla. En caso de ser necesarios los conceptos de primitivas, objeto y transformaciones básicas, están presentados en el anexo 3.

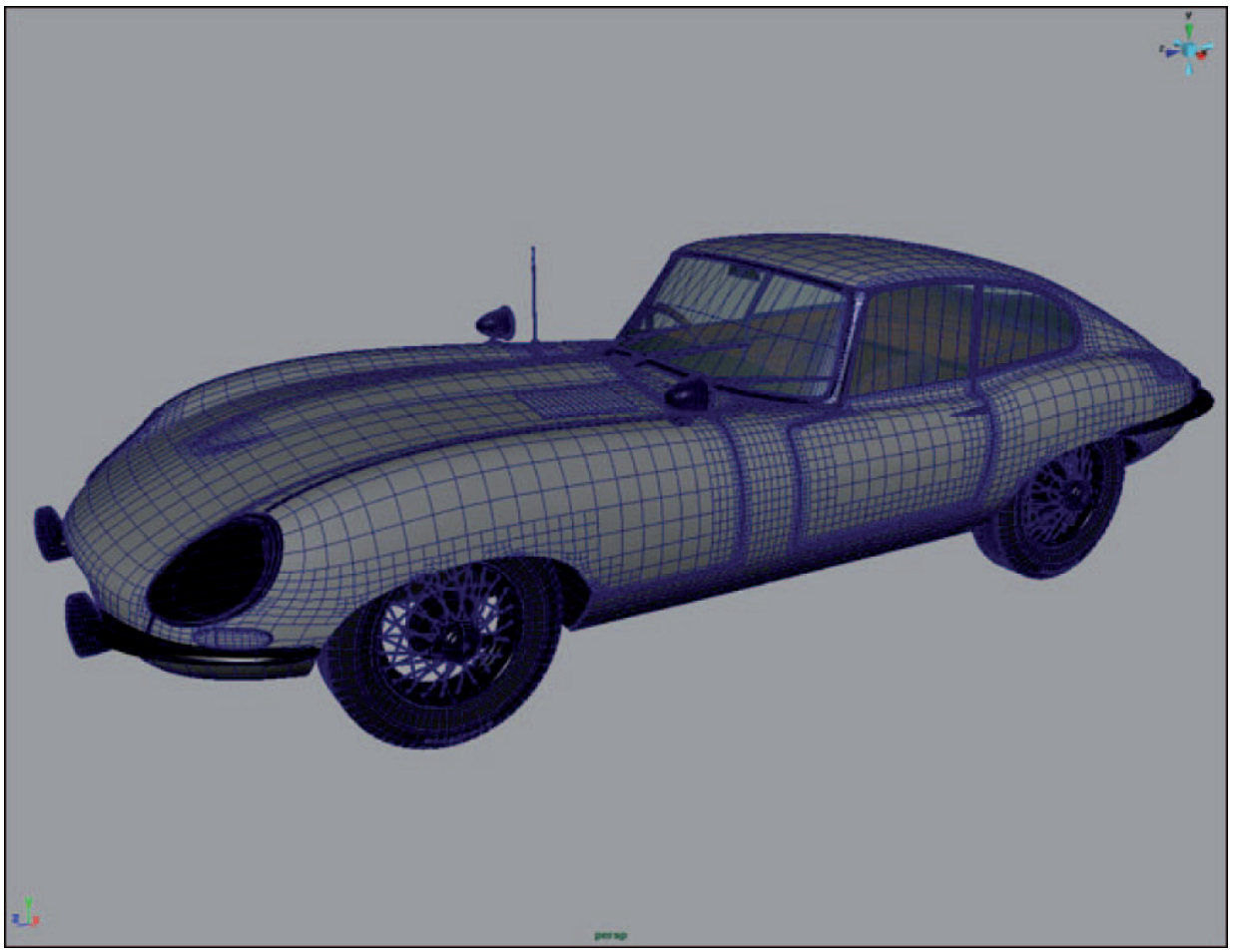
### 3.1 Sistemas de Coordenadas en 3D

Si estamos hablando de representaciones geométricas/graficas de objetos las mismas deben estar basadas en un sistema de coordenadas, a medida que vamos superando instancias en un motor 3D los datos se irán transformando desde su representación original hasta que los mismos son ubicados en su posición final en pantalla. Estas transformaciones requieren que los datos/modelos sean transportados de un sistema de coordenadas a otro, a continuación estudiaremos los sistemas de coordenadas por los que serán procesados los datos en nuestro motor 3D en camino a su representación final en pantalla. [13]

- Coordenadas de Modelo / Espacio Objeto
- Coordenadas del Mundo / Espacio Mundo
- Coordenadas de Cámara / Espacio Cámara
- Coordenadas de Perspectiva / Proyección al plano de visión
- Coordenadas de Pantalla / Mapeo al puerto visual

### 3.2 Coordenadas de Modelo

Cuando se crea un modelo tridimensional el mismo suele estar definido en base a su propio conjunto de ejes cartesianos, es en base a estos ejes que definimos los vértices y polígonos que conformarán el objeto, así como el centro del mismo, el cual suele encontrarse en la posición (0,0,0). Decimos que el centro del objeto suele encontrarse en (0,0,0) ya que esto no es una condición necesaria, en ocasiones es de nuestro interés posicionar el centro en otro punto distinto al origen de coordenadas.

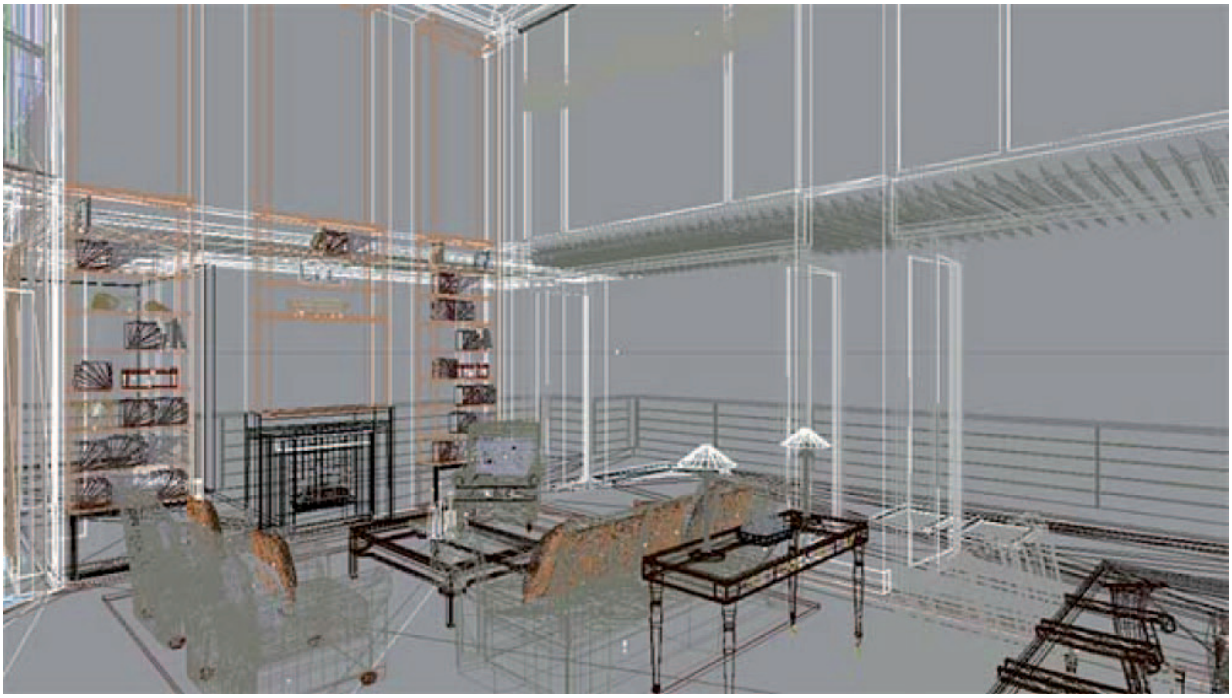


Modelo de un auto basado en una jerarquía de objetos
http://www.pullin-shapes.co.uk/page6.htm
Fecha de acceso: 30/05/2010

Si observamos el modelo del auto vemos que el mismo está conformado por un conjunto de modelos individuales, la idea de una jerarquía de objetos es poder representar los componentes individuales del objeto que buscamos representar en 3 dimensiones. Si crearemos el auto con una sola malla de polígonos, no podríamos simular el proceso de abrir y cerrar las puertas del mismo por ejemplo. Siguiendo este mismo ejemplo es donde volvemos a la utilidad de posicionar el centro del objeto en posiciones alternativas. Para simular la apertura de una puerta se utilizara una rotación, si fuésemos a posicionar el centro en el medio de la puerta la rotación obtenida sería incorrecta, lo que queremos hacer en este caso es posicionar el centro sobre la "bisagra" de la puerta para obtener la rotación correcta. Por último vemos una representación de un modelo en espacio objeto con los valores de sus vértices en concordancia con el sistema coordenado.[15]

### 3.3 Coordenadas del Mundo

El concepto de mundo en videojuegos y gráficos 3D se aplica al universo virtual donde se ubicarán, moverán y transformarán todos los modelos 3D que decidamos utilizar, aunando a los mismos bajo un único sistema de coordenadas. También es el espacio en donde el usuario interactuará con los mismos.



Una escena 3D: vemos como todos los modelos individuales se posicionan en base al mismo sistema coordenado
http://www.archiform3d.com/images/MACA-Liv-Still-A-wire.jpg
Fecha de acceso: 30/05/2010

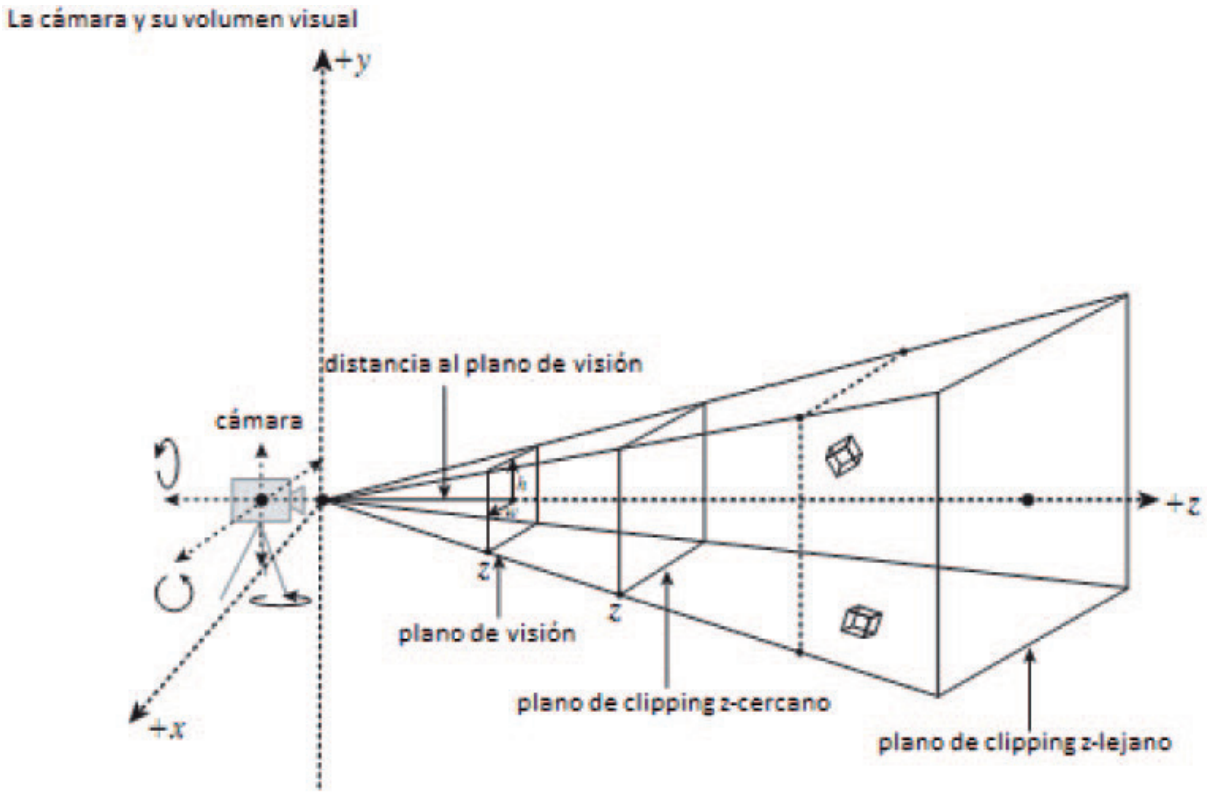
Vemos entonces que en las coordenadas de modelo nos ayudan a definir el objeto a representar pero luego debemos posicionar el mismo en el mundo es decir que debemos transformar el modelo de sus coordenadas locales a las coordenadas del mundo. A la hora de posicionar un modelo en el mundo debemos definir la posición que ocupará, así como su orientación/rotación y tamaño. Vemos cómo se aplican estas transformaciones para lograr la transición de espacio objeto a espacio mundo basado en el código desarrollado para el motor.

```
for (v = 0; v < obj->num_verts; v++)
{

        if(!(obj->vlist[v].vertex.est & VERTEX_EST_ACT))
                continue;

        // rotar
        mult_vec_mat4X4( &obj->vlist[v].vertex.v,
                         &obj->mrot,
                         &obj->tvlist[v].vertex.v);

        //escalo el modelo
        obj->tvlist[v].vertex.x = obj->tvlist[v].vertex.x * obj->escala.x;
        obj->tvlist[v].vertex.y = obj->tvlist[v].vertex.y * obj->escala.y;
        obj->tvlist[v].vertex.z = obj->tvlist[v].vertex.z * obj->escala.z;

        // traslado el modelo
        obj->tvlist[v].vertex.x = obj->tvlist[v].vertex.x + obj->tpos_glob.x;
        obj->tvlist[v].vertex.y = obj->tvlist[v].vertex.y + obj->tpos_glob.y;
        obj->tvlist[v].vertex.z = obj->tvlist[v].vertex.z + obj->tpos_glob.z;
        obj->tvlist[v].vertex.w = 1;

} // end for num_verts
```

El código realiza las transformaciones basadas en valores que ya se han definido en etapas previas de la ejecución, se comienza multiplicando los vértices del objeto, la lista obj->vlist[], contra una matriz de rotación que ya se encuentra calculada. Es necesario notar que el resultado de la rotación se almacena en la lista de vértices transformados obj->tvlist[], es esencial que las transformaciones no se realicen sobre la vlist ya que perderíamos los valores originales del modelo lo cual hará que en un nuevo ciclo de dibujo no se cuente con los valores originales del objeto. Una vez realizada la rotación, el modelo se escala, y se traslada en base a los cálculos presentados anteriormente, el escalado es una multiplicación de un vector por un escalar, y una traslación es la suma de 2 vectores. En estos casos no se utilizan matrices ya que es mas intuitivo para el desarrollo utilizar las ecuaciones presentadas y también debido a que las operaciones con matrices son más demandantes computacionalmente. Aun así, si se concatenaran varias transformaciones en la misma matriz es posible igual o superar el rendimiento actual, esto queda como futuro tema de investigación. [15]

### 3.4 Espacio Cámara

Hasta ahora tenemos un conjunto de objetos 3D posicionados en un mundo virtual pero no contamos con un medio para visualizarlos, la cámara es nuestra ventana al mundo virtual que hemos creado. La cámara enmarcará a través de su posición y orientación, la fracción del mundo virtual que será visible, esto se logra a través de los siguientes pasos: posicionar la cámara en el mundo, definir la orientación de la misma y su campo visual (Field of View). A partir de estos datos podemos proyectar una pirámide desde el “ojo” de la cámara la cual nos ayudara a determinar la región del mundo que será visible. A esta región encerrada por la pirámide se le llama frustum visual (view frustum), “un frustum es una porción de una figura geométrica (usualmente un cono o una pirámide) comprendida entre dos planos paralelos.” [12] Los elementos que se encuentren dentro del frustum serán visibles y el resto no, entonces ¿Cuales son los planos que definen al frustum?



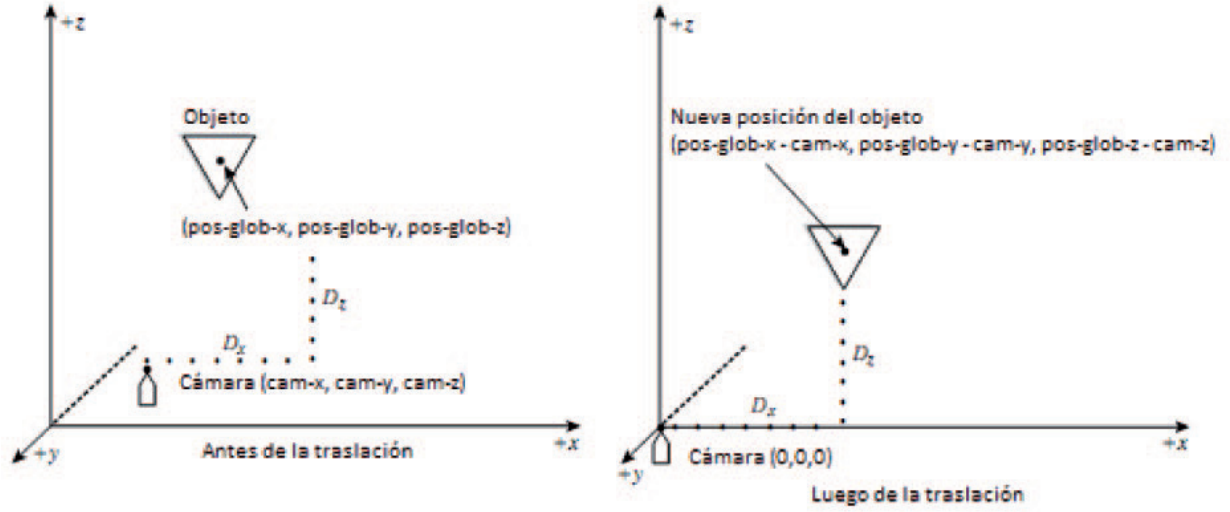

Tendremos 6 planos de clipping (el clipping es una operación que elimina componentes no visibles de la imagen) cada par de planos nos servirá para evaluar si un objeto es o no visible de acuerdo a su posición en x, en y o en z. De estos planos los más importantes a evaluar a son los planos de clipping en z. Llamamos a los mismos plano z-cercano y plano z-lejano, están definidos por las siguientes ecuaciones:

Plano de clipping en z cercano: 0*x + 0*y + 1*z = z = z_cercano
Plano de clipping en z lejano: 0*x + 0*y + 1*z = z = z_lejano

Estos planos nos indican que si un objeto estuviese por delante del plano z cercano, el mismo estaría "chocándose" contra la cámara a la hora de visualizarlo. A su vez, los objetos más allá del z lejano están tan alejados que no tiene sentido dibujarlos ya que serían pequeños puntos en la distancia, vemos entonces que no tiene sentido dibujar estos objetos. Finalmente por delante del frustum y por lo tanto de z cercano, nos encontramos con el view-plane el cual es el plano bidimensional donde, en una etapa posterior, se proyectará la imagen 3D que estamos visualizando para poder presentarla en pantalla.

Ahora bien, ¿en que se basa la transformación del espacio mundo al espacio cámara? En la idea de tener una relación simple entre los objetos en el espacio del mundo y la cámara. Si posicionamos a la cámara en el origen de coordenadas del mundo (0,0,0) y hacemos que la misma "mire" hacia z positivo a la vez que +y indique la dirección "arriba", los ángulos de rotación serán 0,0,0. Mediante la superposición de la cámara con el espacio mundo logramos tener una cámara y un frustum fácilmente definibles y representables, mediante los cuales es fácil ver y calcular que objetos serán visibles y cuáles no.

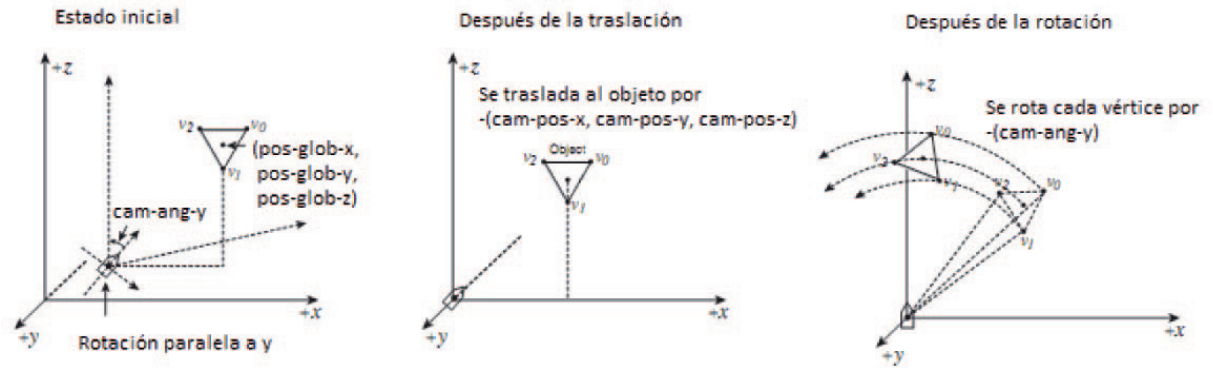
Debido a que la que la cámara suele estar siguiendo al modelo que representa al jugador o posicionada en algún lugar determinado por el desarrollador, la probabilidad de que la misma se encuentre en (0,0,0) con ángulos de rotación nulos es casi inexistente. Lo que debemos hacer es transformar a la escena para que la cámara sea posicionada en (0,0,0) mirando hacia +z. Esta es la transformación del espacio mundo al espacio cámara y está comprendida por una traslación seguida de una rotación. Observamos la siguiente figura para empezar a comprender como llevar a cabo la traslación.




En este modelo simplificado, consideramos que los ángulos de la cámara son 0,0,0 y su posición (cam-x, cam-y, cam-z), tenemos un objeto en (pos-glob-x, pos-glob-y, pos-glob-z). Si la cámara se traslada a (0,0,0) observamos que para mantener la relación existente entre la cámara y el objeto, el modelo debe trasladarse mediante (pos-glob-x, pos-glob-y, pos-glob-z) - (cam-x, cam-y, cam-z). De esta manera modificamos el marco de referencia del universo sin alterar la relación entre los objetos que lo componen. En resumen debemos trasladar a todos los componentes del universo, aplicando una traslación con el vector ( -cam-x, -cam-y, -cam-z). En el caso de querer utilizar matrices:

TCam^-1 = ( 1 0 0 0 )
( 0 1 0 0 )
( 0 0 1 0 )
( -cam-x -cam-y -cam-z 1 )

Supongamos ahora que tenemos el mismo ejemplo que estudiamos para la traslación pero la cámara tiene una orientación definida por (0,ang-y,0), esto significa que cuando posicionemos la cámara en (0, 0, 0) la misma no estará direccionada hacia +z. Nuestro objetivo será lograr orientar la cámara hacia el eje +z, el primer paso es realizar la traslación para que la cámara se ubique en (0,0,0). Una vez realizado esto, si nos detenemos a evaluar un momento nos daremos cuenta que para alinear la cámara al eje de nuestro interés debemos realizar una nueva operación inversa, en este caso rotamos la cámara alrededor del eje y con los ángulos (0, -ang-y, 0). Como es de suponer será necesario realizar esta operación en todos los vértices de la escena, es interesante notar que, a pesar de que toda la escena ha sido reposicionada la relación y la orientación de los objetos permanece inalterada. Las figuras mostradas a continuación ilustran el proceso.




En el caso de tener los ángulos genéricos (ang-x, ang-y, ang-z), solo se añaden las rotaciones relativas a los otros 2 ejes. Teniendo 3 ejes existe 3! = 6 posibles ordenes para realizar las rotaciones, los

más comunes son yxz y zyx, ya son los ordenes más naturales para las personas. Aun así cualquiera de las 6 posibilidades es utilizable y al final de la transformación es irrelevante que orden se selecciona.

Por fin nos encontramos en condiciones de poder realizar la transformación del mundo a la cámara, debido a que es la base de la implementación utilizada por nosotros recopilaremos a continuación las operaciones a llevar a cabo si deseamos usar matrices para lograr el cambio de sistema de coordenadas. Suponiendo que tomamos zyx como el orden de rotación, y teniendo en cuenta que debemos usar las matrices inversas a los ángulos de la cámara, la secuencias de multiplicaciones matriciales para lograr la transformación de espacio mundo a espacio objeto son:

EspCam = TCam^-1 * RCam-z^-1 * RCam-y^-1 * RCam-x^-1

Repasamos las matrices de rotación:

RCam-z^-1
( cos(-ang_z) sen(-ang_z) 0 0)
( -sen(-ang_z) cos(-ang_z) 0 0
( 0 0 1 0
( 0 0 0 1)

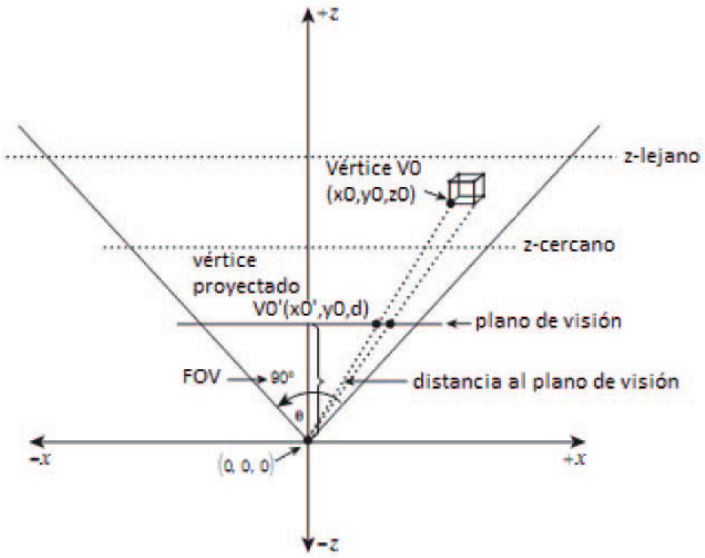
RCam-y^-1
( cos(-ang_y) 0 -sen(-ang_y) 0)
( 0 1 0 0)
( sen(-ang_y) 0 cos(-ang_y) 0)
( 0 0 0 1)

RCam-z^-1
(1 0 0 0)
(0 cos(-ang_x) sen(-ang_x) 0)
(0 -sen(-ang_x) cos(-ang_x) 0)
(0 0 0 1)

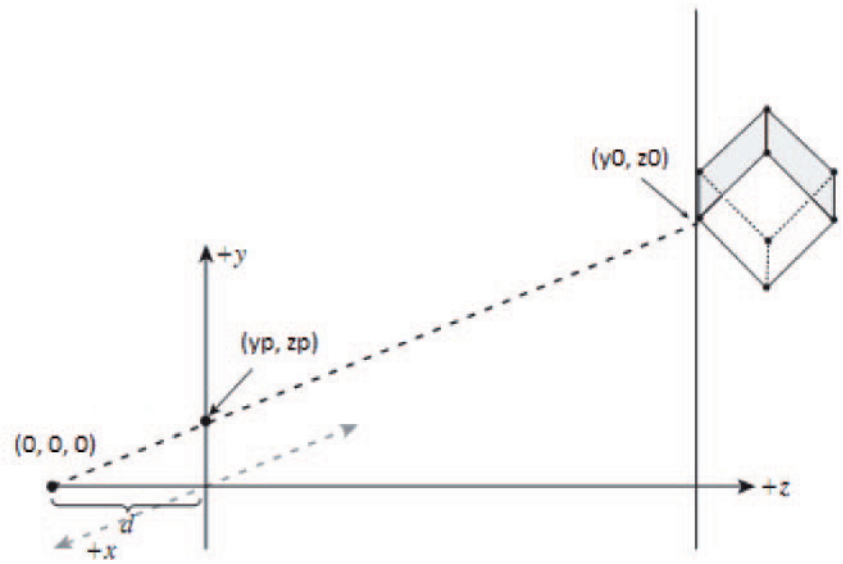
La implementación de la transformación de espacio mundo a espacio cámara se basa en el uso de estas matrices de traslación y rotación, las mismas son concatenadas y al final del proceso obtenemos la matriz por la cual multiplicaremos a los vértices de la escena. En este caso utilizar matrices y concatenarlas no es demasiado perjudicial ya que el cálculo de esta matriz se realiza una sola vez por ciclo de dibujo. [14][20]

### 3.5 Coordenadas de Perspectiva

Una vez determinados cuales son los elementos visibles de la escena, los elementos que se encuentran dentro del volumen visual de la cámara, los mismos deben ser proyectados a un plano bidimensional que será la imagen que se mostrara en pantalla. En primer lugar estudiaremos la proyección de perspectiva en base a un ejemplo con ciertos valores predeterminados, a partir de las conclusiones obtenidas se podrá realizar una derivación para casos generales.



Supongamos que tenemos una cámara posicionada en 0,0,0 con un campo visual de 90°. El valor del FOV nos permite concentrarnos en el sistema cartesiano formado por los ejes x & z, ya que existirá la misma relación entre y & z. En este sistema los componentes visibles del frustum son los planos de clipping en x, así como también los planos de clipping en z. Mientras que los planos en x están definidos matemáticamente en función del volumen visual, los planos en z pueden ser definidos arbitrariamente. Finalmente notamos la presencia de una variable de distancia 'd' que indica donde se posicionará el plano sobre el cual se proyectaran los objetos contenidos en el frustum. Una vez definida la posición del plano de visión, debemos proyectar los vértices de los objetos visibles a través del este plano. Esto se logra calculando la intersección de un rayo proyectado desde la posición de la cámara hasta el vértice del objeto a proyectar.




En la figura tenemos un punto p(y0,z0) el cual deseamos proyectar en el plano de visión cuya posición en z = d. Si utilizamos triángulos similares podemos decir que: d es a z0 como yp es a y0. Análogamente podemos llegar a la misma conclusión con el plano x-z.

Tenemos:
$d/z0 = yp/y0$ y $d/z0 = xp/x0$

Por lo tanto:
$xp = d*x0/z0$
$yp = d*y0/z0$

Generalizando:
$x\text{-pers} = d*x/z$
$y\text{-pers} = d*y/z$

Esta ecuación hace aparente la necesidad de poseer un plano de clipping cercano. Si z = 0 la ecuación tiende a infinito y si z < 0 la proyección se realizara pero los valores estarán invertidos, por lo tanto debemos tener un plano de clipping para que z no pueda tomar esos valores. Si quisiéramos tener una proyección ortográfica lo único que debemos hacer es eliminar el valor z del vértice y mantener los valores de x e y; este tipo de proyecciones se suelen utilizar en programas CAD donde no nos interesa tener perspectiva.

Antes de continuar debemos detenernos y analizar qué valor se le va a dar a la distancia de visión 'd'. Teóricamente 'd' se puede tener cualquier valor, pero al cambiar el valor cambiará la escala de las imágenes. Es decir, se incrementara o disminuirá la cantidad de perspectiva en las mismas. Aun si es

posible asignar a d cualquier valor, ciertos valores facilitan algunas tareas y cálculos. Si utilizamos d = 1, tendremos con un plano de visión normalizado, los valores en los ejes x e y estarán en el rango [-1, 1] y la formula de perspectiva será simplificada a :

x-pers = 1*x/z
y-pers = 1*y/z

Existen algunos posibles problemas con usar d = 1, por empezar será necesario transformar el rango normalizado a la resolución final de pantalla. Segundo obtendremos un plano de visión de forma cuadrada, si el puerto de visión (la pantalla donde se presentará la imagen) no es cuadrado, como suele ocurrir, entonces tendremos una imagen que se encuentra distorsionada en uno de sus ejes. Esto puede solucionarse con un posterior escalado del eje distorsionado. Considerando estos problemas es útil ver que otros valores alternativos podríamos utilizar a la hora de definir el plano de visión.

Una de estas alternativas es calcular d en base a la resolución de pantalla y campo de visión en x que deseemos utilizar. Por ejemplo podríamos utilizar 1024 x 768, con un FOV de 120° en x, y las proyecciones se encontraran dentro del rango deseado, debemos tener en consideración que x e y son de distinto tamaño y por lo tanto se necesitaría un d individual para cada eje, lo cual llevara a una distorsión a la hora de visualizar la imagen. La solución es utilizar el mismo d para ambas transformaciones de perspectivas, con lo cual el FOV en el eje y será un poco diferente al que seleccionamos para x.

Como vemos, las proyecciones a planos no-cuadrados son un poco más problemáticas, lo que debemos hacer para superar estos inconvenientes es recordar que existe un factor que relaciona a los ejes x e y; el mismo se llama: relación de aspecto (aspect ratio). La definición de esta relación es simple AR = ancho/alto, en caso de una resolución de 1024 x768, AR = 4:3 o 1.3333. Una vez que se conoce el aspect ratio se debe calcular 'd' la cual será: la mitad del ancho del plano de visión multiplicada por la tangente del ángulo del campo de visión dividido 2. [14][20]

Entonces la transformación de perspectiva para planos no cuadrados es:

d = 0.5 * ancho-pvision * tan (θ/2)
x-pers = d*x/z
y-pers = d*y/z

### 3.6 Coordenadas de Pantalla

Esta es la última etapa de transformaciones del pipeline, a partir de este punto no realizaremos mas transformación entre sistemas de coordenadas, solo nos quedará dibujar los triángulos en pantalla. Las operaciones que se realizan en esta etapa tienen el objetivo de transformar a los objetos a las coordenadas finales del puerto de visión (la pantalla). Como se ha visto la etapa de coordenadas de perspectiva puede estar íntimamente ligada a la etapa actual y dependiendo de qué método se haya utilizado para realizar el cálculo de perspectiva será el método que utilicemos para calcular los valores finales en pantalla. Veamos el primero de estos casos: plano de proyección cuadrado con FOV = 90° y d = 1. Como ya hemos mencionado en este caso obtenemos un plano normalizado de 2 x 2, con los valores de los ejes entre [-1, 1]. Si el plano es no-cuadrado en cambio sus valores son 2 x 2/ar, con x:[-1, 1] e y:[-1/ar, 1/ar]. Lo que debemos realizar es mapear estos valores a las coordenadas del puerto de visión. Para realizar esta transformación vamos a asumir que el buffer del puerto de visión tiene origen en (0,0) y las dimensiones del mismo están definidas por ancho-pantalla x alto-pantalla entonces la transformación debe tomar los valores x:[-1, 1] e y:[-1/ar, 1/ar] y mapearlos a x:[0, ancho-pantalla -1] e y:[alto-pantalla -1, 0]. Notemos que las coordenadas de y están en orden inverso porque él y de la pantalla esta invertido en comparación al eje y que veníamos utilizando.

La forma más fácil de realizar esta transformación es encontrar el centro de la pantalla y transformar a partir de esta, tenemos entonces:

x-pant = (x-pers+1) * ( (0.5 * ancho-pantalla) - 0.5)
y-pant = (alto-pantalla -1 ) - ( (y-pers+1) * ( (0.5 * alto-pantalla) - 0.5) )

Si evaluamos los bordes del plano de visión veremos que el mapeo es correcto ya que
(-1, 1) = (0, 0),
(1, 1) = (ancho-pantalla -1 , 0 ),
(1, -1) = (ancho-pantalla -1, alto-pantalla -1),
(-1, -1) = (0, alto-pantalla -1)

Analicemos ahora el caso en el que tenemos un campo de visión variable, y un d genérico, cuya fórmula es d = 0.5 * ancho-pvision * tan (θ/2), donde θ corresponde al FOV del eje mayor, x en nuestro caso. Teniendo en cuenta que el rango de la pantalla es [0, ancho-pantalla -1] modificamos el cálculo de para reflejar este hecho:

d = 0.5 * (ancho-pantalla -1) * tan (θ/2)

Utilizando esta ecuación es posible unir las transformaciones de proyección y de mapeo en pantalla. Calculamos los valores de x e y con el nuevo d.

x-pers = (d*x)/z
y-pers = (d*y)/z

Estos resultados ya están escalados al rango de valores de la resolución del plano de visión, pero debemos recordar invertir y.

y-pers = (alto-pantalla - 1) - y-pers
x-pers = xpers

Aun así todavía no hemos terminado de operar sobre estos valores ya que nos encontramos con otro problema, los valores en los ejes x e y están centrados en (0,0), es decir:
x : [ -( ancho-pantalla -1 ) / 2, ( ancho-pantalla -1 ) /2 ]
y : [ -( alto-pantalla -1 ) / 2, ( alto-pantalla -1 ) /2 ]

Por lo que debemos trasladar los valores para que el rango corresponda a:
x : [ 0, ancho-pantalla -1 ]
y : [ alto-pantalla -1, 0 ]

Esto se logra simplemente adicionando la mitad del rango para así trasladar los valores, vemos las ecuaciones finales.

x-pant = x-pers + ( (0.5 * ancho-pantalla) - 0.5)
y-pant = - y-pers + ( (0.5 * alto-pantalla) - 0.5) )

Finalmente si queremos simplificar todos los pasos vistos anteriormente, tenemos las siguientes ecuaciones:

x-pant = d*x-cam/z-cam + ( (0.5 * ancho-pantalla) - 0.5)
y-pant = -d*y-cam/z-cam + ( (0.5 * alto-pantalla) - 0.5)

La implementación que hemos utilizado se basa en fusionar las transformaciones de proyección y posicionamiento en pantalla a fin de ahorrar tiempo de procesamiento, precalculamos ciertos valores para evitar repetir las mismas cuentas una y otra vez en el ciclo. [17]

```
float dist_view_port = 0.5 * (cam->ancho_viewport - 1) * tabla_tan((int)(cam->fov/2));
float ancho = (0.5 * cam->ancho_viewport - 0.5);
float alto = (0.5 * cam->alto_viewport - 0.5);
float z;
```

```
for ( v = 0; v < obj->num_verts; v++)
{
        if(!(obj->tvlist[v].vertex.est & VERTEX_EST_ACT))
                continue;
        //calculo de perspectiva
        //transformación de cámara a plano de visión y pantalla
        z = obj->tvlist[v].vertex.z;
        x = (obj->tvlist[v].vertex.x * dist_view_port) / z;
        y = (obj->tvlist[v].vertex.y * dist_view_port) / z;

        //posicionamiento final en pantalla
        obj->tvlist[v].vertex.x = ancho + x  ;
        obj->tvlist[v].vertex.y = -y + alto ;

}//fin transformaciones
```

# 4 - Rasterización

La rasterización es la última etapa del pipeline gráfico que implementaremos, en la misma se tiene toda la geometría de la escena definida en base las coordenadas finales en pantalla, por lo tanto lo único que resta es dibujar cada triangulo visible. Esta operación de dibujo consiste en asignar valores de color a cada pixel que conformará la imagen raster a ser presentada en el dispositivo de salida.

En este capítulo se estudiaran los conceptos necesarios para comprender la etapa de rasterización así cómo los pasos necesarios para llevarla a cabo y unos ejemplos de código que implementan a la misma. Por último se verá que es provechoso utilizar esta etapa para llevar a cabo el proceso de clipping en dos dimensiones.

## 4.1 Introducción

La necesidad de llevar a cabo un proceso de rasterización radica en que los monitores presentan imágenes a través mediante una grilla rectangular de elementos emisores de luz, los pixeles, cuyos colores e intensidades pueden ser ajustados individualmente en cada imagen presentada. Esto significa que las imágenes a presentar necesitan ser discretizadas en una grilla rectangular cuyas celdas serán rellenadas por los colores de cada imagen a mostrar. El proceso de tomar la escena proyectada al puerto de visión y convertirla en la grilla de colores es llamado rasterización.

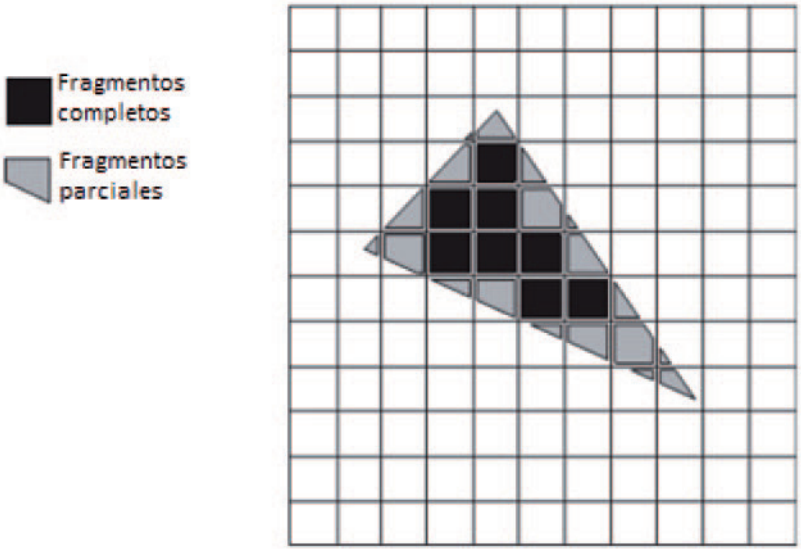
La etapa de rasterización es una de las más demandantes computacionalmente dentro del pipeline, en la misma se deben realizar operaciones sobre todos los pixeles de la imagen a mostrar. Esta es una diferencia notoria en comparación a las operaciones realizadas hasta el momento las cuales, se han llevado a cabo sobre objetos, polígonos y vértices. En motores implementados por software no es extraño observar que 80 a 90 por ciento del tiempo de rendering se consuma en esta etapa, demostrándose el porqué la rasterización fue la primera fase en ser acelerada por hardware. [18]

Las operaciones que se necesarias para rasterizar un triangulo son:

1. Determinar los pixeles visibles que cubre el triangulo. Esta etapa esta subdividida en:
   1.1 Determinar los pixeles cubiertos por un triangulo
   1.2 Determinar qué triángulos son visibles en cada pixel
2. Calcular el color del triangulo para cada pixel.
3. Establecer el color final de cada pixel y escribirlo al framebuffer.

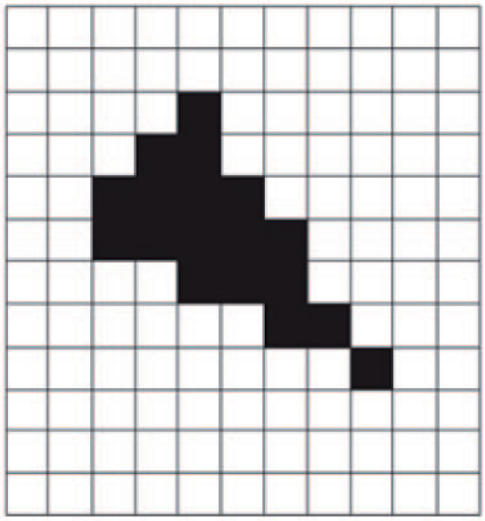
### 4.2 Fragmentos

El primer paso para generar la grilla es dividir a los polígonos en figuras cuya forma se corresponda mejor con los pixeles del framebuffer. La intersección entre el área de un pixel y los triángulos a dibujar es conocida como fragmento, se puede pensar en los mismos como fracciones de un triangulo del tamaño de un pixel. Cuando un triangulo ocupa el área de un fragmento íntegramente el mismo se llama fragmento completo. Mientras que cuando el triangulo no ocupa íntegramente el área del fragmento o el mismo está compuesto por múltiples fracciones de triángulos, se utiliza el término fragmentos parciales. Se puede decir que los fragmentos son las partes de un triangulo que tienen injerencia en un pixel, mientras que los pixeles son los contenedores para todos los fragmentos que ocupan su área. Los pixeles pueden contener fragmentos de distintos polígonos así como también pueden no contener fragmento alguno.[18]



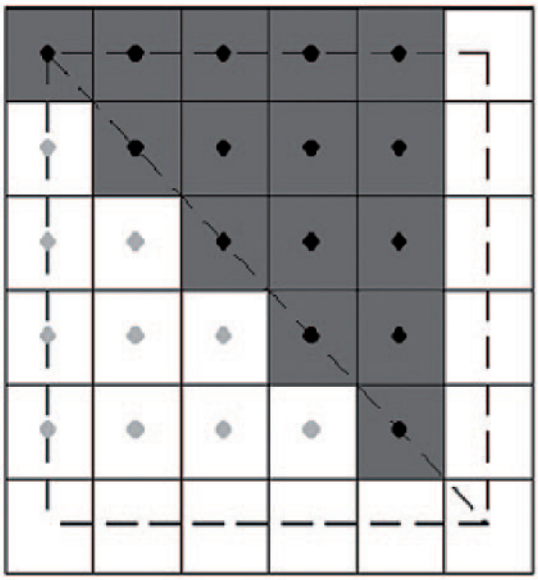

La cantidad de fragmentos en una escena puede ser mucho más pequeña o más grande que el numero de pixeles en pantalla por ejemplo, si solo un subconjunto de la pantalla está cubierta por geometría existe la posibilidad de que muchos pixeles no contengan fragmentos. A su vez si varios triángulos están solapados entonces muchos pixeles contendrán más de un fragmento. La relación entre pixeles en pantalla y fragmentos en una imagen se llama complejidad de profundidad o sobredibujo, esta relación representa cuantas pantallas completas de geometría abarca la escena.

A la hora de transformar los triángulos en fragmentos se aprovecha el hecho de que los triángulos intersecan a las filas horizontales de pixeles, también conocidas como scanlines, en segmentos contiguos. Esta propiedad también aplica para las líneas verticales, por lo que se pueden definir tanto los rangos de valores x-mínimo, x-máximo que ocupara el triangulo en cada scanline así como la extensión vertical del mismo. De esta manera se podrá recorrer verticalmente un triangulo rellenando los pixeles de cada scanline pero, antes de llevar a cabo esto se debe realizar una de las tareas más importantes del rasterizador es determinar cómo procesar los fragmentos parciales. Mientras que implementaciones por hardware el color final de un pixel será influenciado por todos los fragmentos parciales que contenga, existe una alternativa menos costosa computacionalmente. Esta alternativa es el método que utilizaremos en nuestra implementación, la misma consiste en evaluar los fragmentos parciales contra el pixel correspondiente y el que ocupe una posición predefinida en el mismo, generalmente el centro, será convertido en un fragmento completo, descartando a los demás. A este proceso se lo conoce como muestreo de puntos, ya que la comparación se realiza contra un único punto del pixel. Vemos un triangulo procesado y formado por fragmentos completos.[17][20]



### 4.3 Convenio de rellenado

Para llevar a cabo el muestreo de puntos, se debe utilizar un convenio de rellenado que indicará qué fragmento parcial de los que ocupan un pixel será convertido en fragmento completo. Esta determinación es necesaria para evitar que se dibuje un pixel varias veces con los colores de los fragmentos a descartar ahorrando tiempo de procesamiento y evitando posibles problemas gráficos que podrían ocurrir si se utilizan transparencias por tener un pixel sobredibujado. A su vez, si no se utiliza algún convenio de rellenado es posible que se encuentren “huecos” en bordes compartidos entre triángulos. Utilizaremos el método de tope-izquierdo (top-left) , el cuál es el método estándar a la hora de utilizar convenios de rellenado. Vemos dos triángulos que comparten un borde y como se los dibuja en base a este método.



Como se puede ver el triangulo gris ocupara 15 pixeles mientras que el triangulo blanco solo ocupara 10, esto ocurre ya que los pixeles del borde compartido contienen su esquina superior izquierda dentro del triangulo gris por ello se determina que él será quién ocupe el pixel. La implementación de este proceso es simple, antes de recorrer el triangulo se determinan los bordes del mismo y se los modifican para conformar a la convención mediante la función matemática techo, esta función convierte los valores de los bordes al valor entero superior inmediato por ejemplo, si se toma el valor 9.75, el resultado es 10. Esto debe realizarse en los valores máximo y mínimo en y así como para x en cada scanline. [17]

### 4.4 Implementación del rasterizador

El sistema de rasterización que será implementado contiene dos convenciones que ayudan a simplificarlo de modo de que el mismo sea ejecutable por software. Primero, todos los fragmentos son completos es decir, que un fragmento cubre íntegramente a un pixel. Segundo, todos los fragmentos son opacos, por lo que los fragmentos más cercanos a la cámara determinan el color del pixel ya que obstruyen a los más lejanos. Una vez determinado el color de un pixel solo se debe almacenar el resultado en el framebuffer. Estas dos convenciones permiten un incremento en la velocidad de procesamiento a partir de un sacrificio en la calidad de imagen, el no permitir fragmentos parciales hará que los polígonos puedan tener un aspecto escalonado conocido como aliasing. Mientras qué si se desea implementar un sistema de transparencias más adelante será necesario tener en cuenta la contribución de los pixeles obstruidos por el pixel más cercano a la cámara a la hora de determinar el color final que será mostrado.

Como ya se ha mencionado, el proceso de dibujar triángulos se basa en determinar y trazar los pixeles de los bordes del triangulo. Esto se logra mediante el cálculo del cambio de x con respecto a y, es decir 1/pendiente. Por cada línea en y del triangulo se ajustan los extremos de x en base al gradiente, si se desea rasterizar en modo wireframe solo se dibujan los bordes del triangulo. En cambio si se desea dibujar polígonos rellenos, por cada scanline se colorean los pixeles que se encuentran entre los extremos del eje x del triangulo. No es necesario utilizar el algoritmo de Bresenham ya que nos interesa encontrar donde interseca la línea con el pixel para cada intervalo entero en y. Veamos como es el algoritmo para dibujar un triangulo de base plana.

Para comenzar se debe calcular la relación dx/dy = 1/pendiente, para el lado izquierdo y derecho del triangulo. Basándonos en el triangulo del gráfico calculamos la altura del triangulo, dy = y2 - y0. La variación de x desde el punto en el ápice del triangulo hasta sus 2 vértices inferiores es dx_izq = x2 - x0, dx_der = x1 - x0. Por lo que finalmente tenemos:

grad_izq = dx_izq / dy = x2 - x0 / y2 - y0
grad_der = dx_der / dy = x1 - x0 / y2 - y0

Comenzando en (x0,y0) identificamos los extremos en x: xizq = x0, xder = x0, así como la posición inicial de y = y0. Añadimos los gradientes a los extremos de x, en cada scanline para encontrar la posición de los bordes del triangulo y dibujamos una línea entre los mismos. Descendemos y += 1, y repetimos el proceso hasta llegar al final del polígono.

El proceso para dibujar triángulos con tope plano es muy similar al ya explicado. Finalmente existen 2 métodos para dibujar triángulos que no tengan en caras planas el primero es dividir el mismo en 2 triángulos uno de base plana y otro de tope plano y dibujarlos individualmente. El otro proceso es similar pero, en lugar de dividir en 2 triángulos se tiene un condicional que busca el cambio de pendiente y, al encontrarla la calcula continuando con el proceso de rasterización. Vemos un extracto del código de un rasterizador para triángulos rellenos con color constante.

```
// aplicar convenio de rellenado

poly->vlist[0].x = (int)(poly->vlist[0].x+0.5);
poly->vlist[0].y = (int)(poly->vlist[0].y+0.5);

poly->vlist[1].x = (int)(poly->vlist[1].x+0.5);
poly->vlist[1].y = (int)(poly->vlist[1].y+0.5);

poly->vlist[2].x = (int)(poly->vlist[2].x+0.5);
poly->vlist[2].y = (int)(poly->vlist[2].y+0.5);


// ordenar vertices
if (poly->vlist[v1].y < poly->vlist[v0].y)
        {SWAP(v0,v1,temp);}
```

```
if (poly->vlist[v2].y < poly->vlist[v0].y)
        {SWAP(v0,v2,temp);}

if (poly->vlist[v2].y < poly->vlist[v1].y)
        {SWAP(v1,v2,temp);}

// testear para ver que tipo de triangulo es
if (FCMP(poly->vlist[v0].y, poly->vlist[v1].y))
        {
        // establecer tipo de triangulo
        tri_type = TRI_TYPE_FLAT_TOP;

        // ordenar vertices de izquierda a derecha
        if (poly->vlist[v1].x < poly->vlist[v0].x)
                {SWAP(v0,v1,temp);}

        } // end if
else
// now test for trivial flat sided cases
if (FCMP(poly->vlist[v1].y, poly->vlist[v2].y))
        {
        establecer tipo de triangulo
        tri_type = TRI_TYPE_FLAT_BOTTOM;

        // ordenar vertices de izquierda a derecha
        if (poly->vlist[v2].x < poly->vlist[v1].x)
                {SWAP(v1,v2,temp);}

        } // end if
else
        {

        tri_type = TRI_TYPE_GENERAL;

        } // end else

// extraer vertices para procesarlos
x0  = (int)(poly->vlist[v0].x);
y0  = (int)(poly->vlist[v0].y);

x1  = (int)(poly->vlist[v1].x);
y1  = (int)(poly->vlist[v1].y);

x2  = (int)(poly->vlist[v2].x);
y2  = (int)(poly->vlist[v2].y);


// extrer color constante
color = poly->color[0];

// valor de reinicio de interpolacion
yrestart = y1;

// determinar tipo de triangulo
if (tri_type & TRI_TYPE_FLAT_MASK)
        {
```

```
        // tope plano
        if (tri_type == TRI_TYPE_FLAT_TOP)
        {
        // calcular deltas
        dy = (y2 - y0);

        dxdyl = ((x2 - x0)   << FP16_SHIFT)/dy;
        dxdyr = ((x2 - x1)   << FP16_SHIFT)/dy;

        // caso sin clipping

        // valores iniciales para los bordes en x
        xl = (x0 << FP16_SHIFT);
        xr = (x1 << FP16_SHIFT);

        // y inicial
        ystart = y0;

        } // end tope plano
        else
        {
        // fondo plano

        // calcular deltas
        dy = (y1 - y0);
        dxdyl = ((x1 - x0)   << FP16_SHIFT)/dy;

        // caso sin clipping

        // valores iniciales para los bordes en x
        xl = (x0 << FP16_SHIFT);
        xr = (x0 << FP16_SHIFT);

        // y inicial
        ystart = y0;
        yend =y2

        } // end fondo plano

// caso sin clipping

// apuntar el buffer a la primer linea
screen_ptr = dest_buffer + (ystart * mem_pitch);
for (yi = ystart; yi<=yend; yi++)
{
        // calcular los bordes de x para el scanline
        xstart = ((xl + FP16_REDOND) >> FP16_SHIFT);
        xend   = ((xr + FP16_REDOND) >> FP16_SHIFT);

        // dibujar el scanline
        for (xi=xstart; xi<=xend; xi++)
        {
                // escribir pixel utilizar formato de color 5.6.5
                 screen_ptr[xi] = color;

        } // end for scanline
```

```
        // interpolar x
        xl+=dxdyl;
        xr+=dxdyr;

        // avanzar a la siguiente linea del buffer
        screen_ptr+=mem_pitch;

} // end for y
```

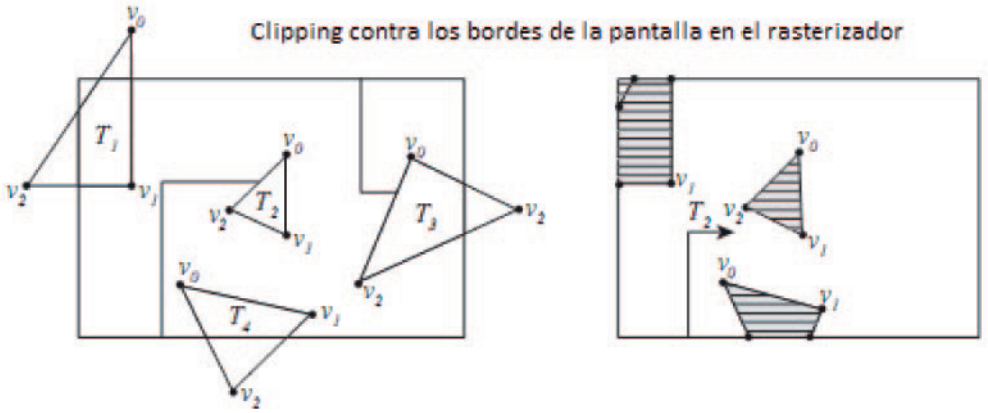
Como se puede observar la función debe calcular los gradientes y con los mismos dibujar el triangulo scanline por scanline, es una función cuya complejidad está más asociada con preparar los datos para llevar a cabo el recorrido del triangulo que, con el algoritmo en sí. La presencia de varios condicionales y ciclos for así como el procesamiento por pixeles son las razones de que esta función lleve mucho tiempo de procesamiento. A su vez, los mayores cambios para implementar rasterizadores con diferentes funcionalidades ocurren dentro de los ciclos anidados incrementando el tiempo de procesamiento de gran manera. Vemos un extracto de código de los ciclos for del rasterizador con soporte para texturas e iluminación Gouraud, se puede notar fácilmente cómo crece el grado de complejidad en los loops y cómo esto impactara en el rendimiento. [17] [20]

```
// apuntar el buffer a la primer linea
screen_ptr = dest_buffer + (ystart * mem_pitch);

for (yi = ystart; yi < yend; yi++)
        {
        // calcular los bordes de x para el scanline
        xstart = ((xl + FP16_REDOND) >> FP16_SHIFT);
        xend   = ((xr + FP16_REDOND) >> FP16_SHIFT);

        // calcular los valores iniciales para la interpolacion de colores y texturas  ui=ul+FP16_REDOND;
        vi = vl + FP16_REDOND;
        wi = wl + FP16_REDOND;

        si = sl + FP16_REDOND;
        ti = tl + FP16_REDOND;

                //calcular los gradientes de interpolacion para los colores y las texturas dentro del scanline
                if ((dx = (xend - xstart))>0)
                        {
                        du = (ur - ul)/dx;
                        dv = (vr - vl)/dx;
                        dw = (wr - wl)/dx;

                        ds = (sr - sl)/dx;
                        dt = (tr - tl)/dx;

                        } // end if
                else
                        {
                        du = (ur - ul);
                        dv = (vr - vl);
                        dw = (wr - wl);

                        ds = (sr - sl);
                        dt = (tr - tl);

                        } // end else
```

```
                    // dibujar el scanline
                    for (xi=xstart; xi < xend; xi++)
                            {

                            // obtener el elemento de textura
                            textel = textmap[(si >> FP16_SHIFT) + ((ti >> FP16_SHIFT) <<
                            texture_shift2)];

                        // extraer componentes rgb
                r_textel  = ((textel >> 11)       );
                g_textel  = ((textel >> 5)  & 0x3f);
                    b_textel =   (textel        & 0x1f);

                    // modular el elemento de textura con la iluminacion Gouraud                    r_
textel*=ui;
                    g_textel*=vi;
                b_textel*=wi;

                  // escribir el pixel modulando comprimiendo los valores para el formato de color 5.6.5
                        screen_ptr[xi] = ((b_textel >> (FP16_SHIFT+8)) +
                                ((g_textel >> (FP16_SHIFT+8)) << 5) +
                            ((r_textel >> (FP16_SHIFT+8)) << 11));

                        // interpolar los colores y las texturas
                        ui+=du;
                        vi+=dv;
                        wi+=dw;
                        si+=ds;
                        ti+=dt;

                } // end for scanline

                    // interpolate los bordes derecho e izquierdo
                    xl+=dxdyl;
                    ul+=dudyl;
                    vl+=dvdyl;
                    wl+=dwdyl;

            sl+=dsdyl;
            tl+=dtdyl;


            xr+=dxdyr;
            ur+=dudyr;
            vr+=dvdyr;
            wr+=dwdyr;
           sr+=dsdyr;
            tr+=dtdyr;


            // avanzar a la siguiente linea del buffer
            screen_ptr+=mem_pitch;

            } // end for y
```

### 4.5 Clipping en el rasterizador

El rasterizador es un lugar optimo para realizar clipping, el cual es el proceso de sólo dibujar los elementos que sean visibles por el usuario. A esta altura del pipeline cuando los triángulos sean cortados al superar los extremos de la pantalla, no es necesario crear nuevos triángulos a partir de las figuras que queden formadas en el puerto de visión. Simplemente dibujaremos los triángulos y, si encontramos que los mismos sobrepasan los límites del puerto de visión, sólo se dibujará la parte visible de los mismos lo que se puede lograr de forma casi trivial. Para llevar esto a cabo tendremos en el rasterizador un bloque de código que determinará, de acuerdo a los vértices del triangulo, si el mismo requiere clipping y en cuál de sus ejes.



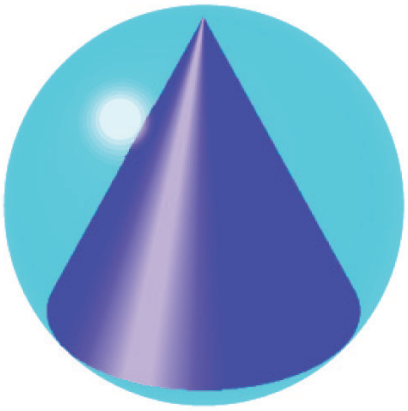

Si queremos realizar clipping en la parte superior de la pantalla, y = 0 , en este caso debemos ver si los vértices superiores del triangulo poseen y < 0. En caso afirmativo y = 0 y se debe calcular la/s nueva/s posición/es de x en base a la intersección de la línea entre los vértices e y = 0. En caso de estar realizando clipping en el fondo de la pantalla, y = alto pantalla, simplemente se deja de dibujar. Los casos para los lados izquierdo (x = 0) y derecho (x = ancho pantalla) son análogos a los casos en y. [18]

## 5 - Conceptos teóricos: Eliminación de componentes no visibles

Hemos visto como a lo largo del pipeline la geometría se transforma, ilumina, y dibuja. Además de estas operaciones existen otras que buscan identificar elementos de la geometría del mundo que por alguna razón no serán visibles en la escena con el fin de evitar que sean procesados. A estas pruebas se las suele clasificar dentro de la categoría remoción de superficies no visibles y las más comunes son: eliminación de caras traseras (back-face removal), prueba de las esferas circundantes (bounding spheres test) y clipping.

### 5.1 Prueba de las esferas circundantes

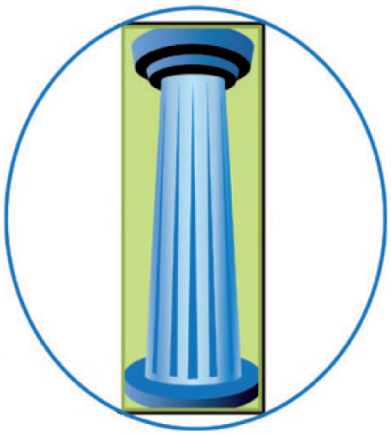
Esta es la primer prueba que se debe realizar ya que nos permitirá eliminar objetos enteros de la escena. Para ellos se crea una esfera que rodea a cada objeto que se encuentra en el espacio mundo. Luego se transforma la esfera a espacio cámara, para lo cual solo debemos transformar el centro de la misma ya que una esfera se define por su centro y diámetro. Si la misma se encuentra totalmente fuera del volumen visual entonces podemos descartar todo el objeto, en caso contrario seguiremos procesándolo. Como se puede ver este método nos permite eliminar objetos enteros, de forma sencilla y con muy pocas operaciones, pero no será útil en caso de estar trabajando con escenas en espacios cerrados donde la mayoría de la arquitectura está definida por una sola malla de polígonos.



Para crear una esfera que rodee a un objeto O, se recorren los vértices del mismo hasta determinar cuál es el más alejado del centro en otras palabras, se encuentra el radio máximo del mismo. Lo interesante es que este cálculo se puede realizar cuando se cargan los modelos antes de iniciar el motor por lo cual no es necesario el cálculo en tiempo real. Veamos un pseudo-algoritmo para calcular la esfera:

```
radio_max = 0;
radio_actual;
for desde vert = 0 hasta vert < num_verts hacer vert++
{
        x = obj[vert].x;
        y = obj[vert].y;
        z = obj[vert].z;

        si ( (radio_actaul = sqrt( x*x + y*y + z*z) ) > radio_max
        radio_max = radio_actual
}
```

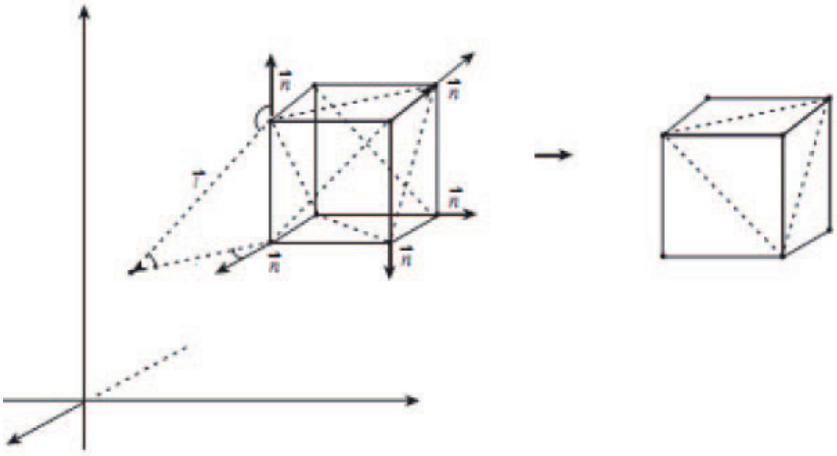
Un posible inconveniente al cual nos podemos enfrentar es que parte de la esfera que rodea al objeto se encuentre dentro del volumen visual, pero el objeto en si no lo haga. Esto ocurre cuando la esfera no es una buena representación del objeto a eliminar.



La figura ilustra este problema, en estos casos se podría utilizar otras figuras geométricas más adecuadas para rodear a los objetos. Lo importante a recordar en este caso es, que la prueba de las esferas circundantes puede no eliminar a todos los objetos no visibles dentro del volumen visual. [18]

### 5.2 Eliminación de caras traseras

Esta prueba se realiza en coordenadas de mundo a fin de evitar continuar transformando polígonos que no son visibles ya que se encuentran “mirando” en dirección opuesta a la cámara. Para eliminar las caras traseras es necesario que todos los polígonos del mundo estén ordenados consistentemente siguiendo las agujas del reloj o, en contra de ellas, es indistinto mientras se mantenga la consistencia. Para comenzar se calcula la normal para cada polígono de la escena y se calcula el ángulo formado entre las normales de los polígonos y el vector de visión de la cámara, si el ángulo obtenido es mayor a 90° el polígono no es visible. A partir del cálculo del ángulo, se hace evidente la necesidad de tener un orden consistente en los polígonos. El orden de un polígono hará que su normal apunte hacia “adentro” o “afuera”, por lo tanto si se debe calcular el ángulo entre la normal y otro vector es necesario que todas las normales de los polígonos sean consistentes y tengan la misma dirección. Para calcular el ángulo entre la normal y el vector de visión se usa el producto punto ya que si el resultado es u . v = 0, el ángulo entre los mismos es 90°, si es u . v > 0 el ángulo es menor a 90 y si u . v < 0 el ángulo es mayor a 90. Con esta prueba se puede eliminar hasta 50% de los polígonos de la escena. [18]



### 5.3 Clipping en espacio Cámara

La técnica para realizar clipping en el volumen visual de la cámara no se corresponde con los métodos normales de clipping, sino que busca anteponer el rendimiento y evitar la repetición de tareas. Como hemos visto cuando se rasterizan los triángulos se realiza clipping sobre los mismos en x e y para que las líneas no superen los márgenes del puerto de visión. En vez de repetir la misma operación con respecto al frustum, el clipping que realizaremos respecto a los ejes x, y será trivial, los polígonos serán eliminados si se encuentran íntegramente fuera del frustum. En el plano z-lejano realizaremos las mismas operaciones pero, debemos realizar clipping real en z-cercano ya que si un polígono se extiende a una posición z <= 0 al realizar la proyección del mismo este puede causara divisiones por 0 o proyecciones invertidas, haciendo peligrar la estabilidad del motor y causando errores gráficos.

Este método de clipping podría llevarse a cabo en otros sistemas coordenados y no solo en espacio cámara, pero se opto por utilizar el mismo ya que los polígonos quedan ordenados en relación a la cámara, la cual se encuentra en el origen. A su vez el volumen visual tiene sus planos paralelos y perpendiculares a los ejes coordenados simplificando las operaciones. Por último el clipping se realiza antes de la etapa de iluminación para evitar iluminar polígonos que no serán visibles en la imagen final. [18]

#### 5.3.1 Calculo del volumen visual para remoción de objetos

El cálculo del volumen visual ha sido abarcado con otro fin cuando se detallo la transformación al plano de visión. Si recordamos cuando deseamos calcular los planos x, y del volumen podemos tomar solo los

ejes x-z e y-z, obteniendo así 2 problemas en planos bidimensionales los cuales pueden ser resueltos con el principio de los triángulos similares.

Cuando tenemos un FOV de 90° el problema se simplifica aun mas y el plano x = z. Para realizar el clipping trivial en x debemos comparar los valores de los 3 vértices del triangulo contra el plano, es decir debemos comparar el componente x del vértice contra el componente z. [18]

Dado un vértice v(x, y, z) y FOV = 90

Si (x > z), v se encuentra fuera del plano de clipping derecho.
Si (x < z), v se encuentra dentro del plano de clipping derecho.
Si (x = z), v se encuentra sobre el plano de clipping derecho.

Si (x > z), v se encuentra dentro del plano de clipping izquierdo.
Si (x < z), v se encuentra fuera del plano de clipping izquierdo.
Si (x = z), v se encuentra sobre el plano de clipping izquierdo.

Los cálculos para y son iguales, solo se reemplaza y por x.

Habiendo visto el caso especifico FOV = 90°, es necesario generalizar para un campo de visión genérico. El cálculo ya se derivo con anterioridad obteniendo:
$\tan(\theta) = x/z \rightarrow x = \tan(\theta)*z$

Por lo que dado un vértice v(x, y, z) y FOV genérico

Si (x > tan(θ)*z), v se encuentra fuera del plano de clipping derecho.
Si (x < tan(θ)*z), v se encuentra dentro del plano de clipping derecho.
Si (x = tan(θ)*z), v se encuentra sobre el plano de clipping derecho.

Si (x > tan(θ)*z), v se encuentra dentro del plano de clipping izquierdo.
Si (x < tan(θ)*z), v se encuentra fuera del plano de clipping izquierdo.
Si (x = tan(θ)*z), v se encuentra sobre el plano de clipping izquierdo.

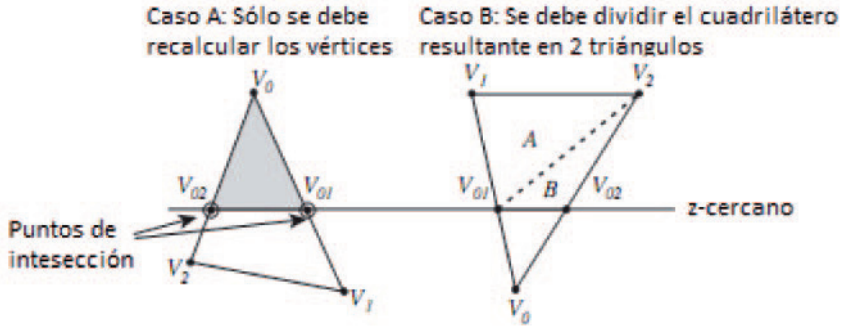
Los cálculos para y son iguales, solo se reemplaza y por x.

### 5.3.2 Clipping en Z

Determinar los planos cercanos y lejanos de clipping en z es una tarea trivial, ya al ser paralelos al eje x los mismos se definen arbitrariamente con una valor en z y no a través de una función matemática. Por lo tanto para eliminar polígonos trivialmente solo se debemos comparar los valores de los vértices en z contra los valores de los planos de z. La dificultad del clipping en z radica en que no podemos obviar a los polígonos que atraviesan parcialmente el plano z-cercano, debemos clippearlos contra el propio plano evitando de esta forma que el mismo sea atravesado. Veamos el pseudo código que evaluará a los polígonos en z antes de entrar en detalle de cómo realizar el clipping contra z-cercano.

```
Para cada polígono P de la escena
Comenzar
        Para cada vértice V de P
        Comenzar
        1. ¿Esta V dentro del frustum?
        2. ¿Esta V mas allá de z-lejano?
        3. ¿Esta V mas allá de z-cercano?
        Almacenar el resultado
        Fin
Si todos los vértices están más allá de z-cercano o z-lejano
        Eliminar P
Si algún vértice se encuentra dentro del frustum y algún vértice se encuentra más allá de z-cercano
        Clippear P a z-cercano
Fin
```

A continuación vemos los 2 posibles casos cuando un polígono atraviesa z-cercano




Caso A: Un vértice dentro del frustum y 2 exteriores. En este caso se calcula la intersección de los vértices con el plano y se reemplaza a los vértices viejos con los nuevos. En caso de tener mapas de textura, se debe volver a calcular las coordenadas de texturas de los mismos. También es posible que se necesite recalcular la normal del polígono, ya que el área del mismo ha cambiado, si es que la función de iluminación requiriere que las normales de los polígonos sean precalculadas.

Caso B: Dos vértices dentro del frustum y uno exterior. En este caso la figura que se obtiene al calcular los nuevos vértices es un cuadrilátero, el mismo debe ser dividido en 2 triángulos para mantener coherencia con las estructuras de datos que utiliza el motor. En este caso se debe generar una copia del triangulo antes de procesarlo y luego realizar el clipping, de esta forma uno de los triángulos generados será almacenado en el triangulo original y el otro en la copia del mismo.

Para calcular los puntos de intersección de los polígonos con z-cercano, se utiliza una línea paramétrica q representara las líneas formadas entre los vértices interiores y exteriores. Tenemos el polígono p con vértices v0, v1 y v2 y una ecuación con parámetro t:

p = v0 + (v1 – v0)*t

Desarmando la ecuación en sus componentes
x = v0.x + (v1.x – v0.x)*t
y = v0.y + (v1.y – v0.y)*t
z = v0.z + (v1.z – v0.z)*t

Como queremos que el nuevo valor de z sea z_cercano
z_cercano = v0.z + (v1.z – v0.z)*t

Despejamos el parámetro t el cual indicara el punto de intersección
t = z_cercano – v0.z/(v1.z – v0.z)

Finalmente una vez obtenido el parámetro el mismo se aplica en las ecuaciones de x e y para despejar los valores de los mismos en el punto de intersección. En caso de que el polígono posea texturas se deberá realizar el mismo procedimiento para calcular las nuevas posiciones de las coordenadas de texturas. [18]

# 6 - Desarrollo: Implementación del motor grafico y OpenCV

Se estudiará con más detalle la implementación del motor gráfico. Se analizará el pipeline desarrollado, explicando el porqué del orden de las operaciones realizadas en el mismo. Se observarán las etapas que componen a un programa que hace uso del engine, y se analizarán las estructuras de datos principales. Antes de tratar estos temas se discutirán los pro y contras de una las implementaciones por hardware y software.

Por último cabe destacar que el proyecto de desarrollo está basado en los lenguajes de programación C y C++, se utiliza el API DirectX y su SDK para tener acceso al dibujo de pixeles en pantalla, y por último se integran las librerías OpenCV de Intel para ayudar a la implementación de los algoritmos de computación visual. El entorno de desarrollo utilizado es Microsoft Visual Studio 2010 Ultimate, el sistema operativo es Windows 7 de 64bits.

### 6.1 - Implementación : Software versus Hardware

Al momento de comenzar la investigación para adquirir los conocimientos necesarios para poder desarrollar un motor gráfico, nos encontramos con dos corrientes de pensamiento entre los autores: desarrollar motores basados en llamadas a APIs gráficas para obtener aceleración de hardware e implementar las funcionalidades del motor en base a estas llamadas, o desarrollar motores basados íntegramente en software.

André LaMothe piensa que un enfoque puramente basado en software nos permite tener todo el conocimiento fundamental para escribir nuestro propio motor. Seremos familiarizados con todos los pasos necesarios para crear un motor 3D: desde la representación de datos hasta la rasterización final. Esencialmente estaríamos en condiciones de crear todo el API y los algoritmos necesarios que utiliza el hardware grafico a través de maquinas de estado. Por último dice: “No voy a enseñarte a usar APIs 3D como Direct3D u OpenGL, son geniales, pero si realmente quieres entender 3D, necesitas entender cómo escribir Direct3D u OpenGL por ti mismo, y eso es lo que me propongo. A fin de cuentas lo que quiero es que seas capaz de sentarte frente a cualquier computadora del planeta con nada más que acceso al buffer de video y ser capaz de escribir un engine 3D basado en software. Si puedes hacer eso, aprender cualquier API 3D será trivial.”[2]

Como se puede ver, la implementación propia de los algoritmos nos hace tener un insight mucho mayor en como los mismos funcionan. En qué problemas son los que se nos presentan a la hora de la implementación y del debugging, y que técnicas se pueden utilizar para agilizarlos. El grado de familiaridad que se logra es mucho mayor que con llamadas a funciones ya implementadas, por más que entendamos que es lo que hace la función cuando es llamada. [19]

### 6.2 Estructuras de datos

En esta sección veremos las principales estructuras de datos utilizadas en el motor, estudiaremos los componentes que las conforman esto nos dará una mayor comprensión de que datos son necesarios a la hora de crear un motor y como se utilizarán los mismos. Como punto de partida debemos destacar que el motor trabajara con objetos hasta que los polígonos estén en el espacio cámara y luego se copian los polígonos que se dibujaran a una renderlist, una lista de polígonos. Existen diversas formas de manejar los polígonos, hay motores que trabajan casi exclusivamente con listas de polígonos , y otros que utilizan objetos y luego cambian a listas de polígonos en diversos puntos del motor. Esto se debe a que ambas formas de almacenar los polígonos tienen beneficios y desventajas. Se ha decidido utilizar el almacenamiento en objetos durante la mayoría del motor ya que es más simple para el programador trabajar con el concepto de objetos individuales, que utilizar una lista de polígonos que pueden representar cualquier objeto en pantalla. Es evidente además que de esta manera es posible probar nuevos métodos y algoritmos en objetos individuales evitando sumar complejidad innecesaria en la etapa de pruebas. Por último es útil mantener el concepto de objetos hasta la etapa de iluminación ya que evita la repetición de muchos cálculos cuando se deben calcular las normales para los vértices en el sombreado Gouraud. A su vez utilizar renderlists simplifica el clipping en z-cercano ya que se pueden modificar los elementos sin problemas ya que cada uno de ellos es una entidad individual sin relación con los demás, cuando se trabaja con objetos se debe tener más cuidado en este sentido. Comencemos entonces estudiando la estructura OBJ.

```
typedef struct OBJ_TYP
{
        int estado;
        atr;
        int num_verts;
        int num_polys;
        int num_textcoords;
        int num_verts_orig; //almacena los valores cuando se degeneran con z clipping
        int num_polys_orig;
        int num_textcoords_orig;

        MAT4X4 mrot;

        //VECTOR3 centro_obj;
        VECTOR4 tpos_glob;
        VECTOR4 pos_glob;

        //ejes del obj
        VECTOR4 eje_x;
        VECTOR4 eje_y;
        VECTOR4 eje_z;

        VECTOR4 teje_x;
        VECTOR4 teje_y;
        VECTOR4 teje_z;

        //angulo de rotacion en enteros
        int ang_rot_x;
        int ang_rot_y;
        int ang_rot_z;

        //factor de escalado
        VECTOR3 escala;

        PNODOTEXT tlist;
        BITMAP_IMAGE_PTR skin;
        float rad_max;

        PNODOVERTEX vlist;
        PNODOVERTEX tvlist;
        PNODOPOLY plist;

} OBJ, *POBJ; //fin obj
```

Como se puede observar esta es una estructura compleja que depende de otras estructuras que veremos más en detalle a continuación. La variable estado indica si el objeto esta activo, inactivo, si fue eliminado por algún algoritmo de culling, o si fue clippeado en z ya que ciertos valores deberán restaurarse en tal caso. Las variables num indican el numero de vértices, polígonos y coordenadas de textura del objeto. Esto nos permite recorrer las listas de polígonos, vértices y coordenadas de textura, tanto como una lista: recorrer hasta encontrar el puntero siguiente = NULL o como un arreglo con un ciclo for. Esto nos permite más flexibilidad a la hora de tratar con las estructuras y se origina del hecho de que durante la mayor parte del desarrollo estas listas fueron arreglos pero como en ciertas ocasiones es necesario eliminar o añadir elementos se cambiaron a listas para que este proceso sea más simple. Las variables con el sufijo 'orig' justamente guardan los valores originales a fin poder restaurar las listas a su valores originales. La variable pos_glob contienen el centro del objeto en espacio de modelo, mientras que tpos_glob almacena el centro del objeto en los distintos sistemas de coordenadas a los que será transformado. El tipo de datos VECTOR4 es una estructura propia para almacenar vectores de 4 dimensiones.

Las variables eje y teje son análogas al pos y tpos pero para los ejes de coordenadas del objeto, de esta manera se puede saber la orientación del mismo en todas las etapas del motor.

Las variables ang_rot contienen los ángulos por los que será rotado el objeto en la transformación al espacio mundo, la matriz mrot almacena la matriz de rotación calculada en base a estos ángulos. El vector de 3 dimensiones almacena los valores por los que se escala al objeto, se utiliza un vector de 3 dimensiones para poder realizar escalados uniformes y no-uniformes. Si el objeto esta texturado skin contiene el puntero a la imagen que se utiliza como textura y tlist es la lista de coordenadas de textura para los vértices del objeto. Plist contiene la lista de polígonos, mientras que vlist contiene los vértices y tvlist contiene una copia de la lista de vértices que serán los que se modifiquen durante la ejecución del motor. Por ultimo rad_max contienen el radio del objeto para la eliminación de objetos contra FOV. La segunda estructura que presentaremos es la que conforma los polígonos, cada nodo de la lista de polígonos incluye esta estructura.

```
typedef struct POLY_TYP
{
        int estado;
        int atr;
        int color; //color original del poly
        int color_ilum[3]; // contiene los colores después de la etapa de iluminación si se usa ilum mono-
cromatica el valor esta en 0
                                                // si se usa rgb el valor esta en 0,1,2

        float x0,y0,z0;

        PNODOVERTEX vlist; //contiene la lista de vértices
        PNODOVERTEX tvlist; //contiene la lista de vértices transformados
        unsigned short vert_index[3]; // contiene los indices con las posiciones de los vértices en la vlist
del objeto
                                                // a la hora de crear una renderlist deberemos recorrer
estos valores para extraer los
                                                //vértices reales de la vlist en vert antes de seguir adelante
                                                // NO DEBERIA SER NECESARIO SI YO TENGO EL
PUNTERO A VLIST, SERA CUESTION DE INVESTIGAR
//      PBITMAP_IMG skin; // la textura que utilizara el objeto
        BITMAP_IMAGE_PTR skin;
        PNODOTEXT tlist; //contiene las coordenadas de las texturas
        unsigned short text_index[3]; // es el mismo caso que con los vértices pero la copia de los datos
se puede realizar
                                                 // antes ya que las textcoords no seran modificadas a
diferencia de los vértices
        VECTOR4 n; //normal
        VECTOR4 n_trans; //normal transformada
        float nl; //longitud de la normal
        float nl_trans; //longitud de la normal transformada

} POLY, *PPOLY; //fin poly
```

Los posibles estados de la estructura polígono indican, si el mismo está inactivo, activo, si ha sido iluminado, clippeado o removido por backface culling. Los atributos destacables que puede tener un polígono indican que tipo de sombreado se debe aplicar sobre los mismos en la etapa de iluminación, lo cual también ayuda a saber que rasterizador utilizar después. También indicia si un polígono es visible de ambas caras lo cual impide aplicar backface culling al mismo. El atributo color almacena el color original del polígono, y el arreglo color_ilum almacena el color del polígono luego de la etapa de iluminación. Las listas de vértices y coordenadas de texturas son punteros a las listas almacenadas en el objeto al que corresponde el polígono, por lo que vemos que los polígonos tienen una dependencia de los objetos que los contienen. Mientras que el arreglo text_index[] almacena las posiciones de la lista donde se almacenan los vértices correspondientes al polígono en particular. La otra estructura de datos interesante del objeto es vertex4:

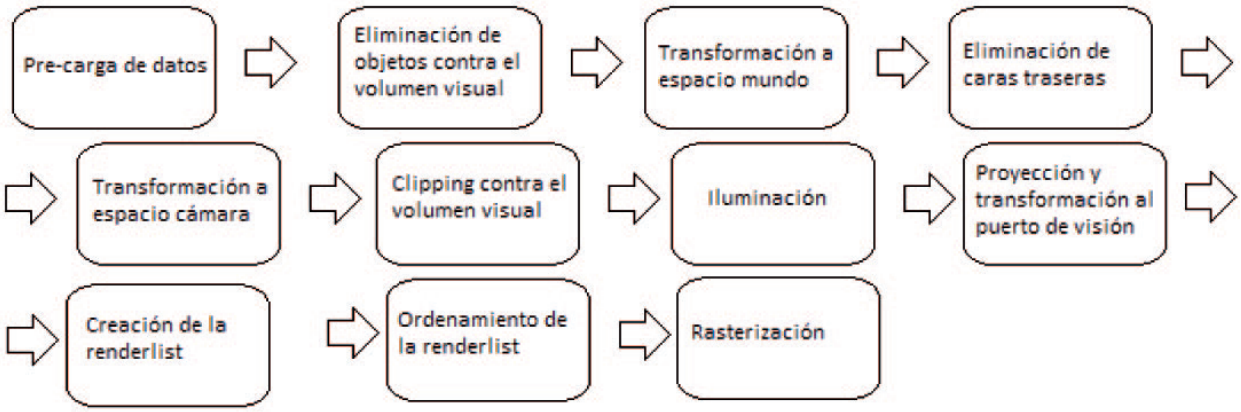
```
typedef struct VERTEX4_TYP
{
 union
        {
         float M[10];

         struct
         {
                 float x,y,z,w; // ubicacion
                 float nx, ny, nz, nw; // normales
                 float nl; //longitud de la normal
                 float nl_trans; //longitud de la normal transformada
                 float u0, v0; // text coords
                 float i; // intensidad final del vértice después de la iluminación
                 int atr; // atributos , text coords extra
                 int est; // estado , activo , inactivo, backface
         }; //fin low level struct
         struct
         {
                 POINT4 v; // el vertex
                 VECTOR4 n; // la normal
                 VECTOR4 n_trans; //la normal transformada
                 POINT2 t; // las text coords

         }; // fin high level struct

        }; //fin union
 } VERTEX4, *PVERTEX4;
```

Los vértices tiene estados y atributos similares a los polígonos, como vemos esta estructura es más simple que las vistas anteriormente. Contiene la posición del vértices así como el almacenamiento para las normales en caso de ser un vértice sometido sombreado Gouraud, también hay almacenamiento secundario para la normal por si la misma esta precalculada. Las variables u0 y v0 almacenan la posición en la lista de coordenadas de textura donde se encuentran las coordenadas correspondientes para el vértice . Quizás lo más interesante de esta estructura es la unión que nos permite acceder a la mayoría de los componentes de la misma desde las llamadas a los componentes básicos o a través de estructuras de mayor nivel. La última estructura que veremos es el renderpoly este es el polígono que conforma la renderlist, es decir una estructura que debe contener solo los datos necesarios para dibujar los polígonos en pantalla.

```
typedef struct RENDERPOLY_TYP
{
        int estado;
        VERTEX4 vlist[3]; //coordenadas en el espacio, también contiene las coordenadas de textura
        int color[3];
        BITMAP_IMAGE_PTR skin;
        int atr; // atributos para ver como se renderiza
} RENDERPOLY, *PRENDERPOLY;
```

Como vemos se trata de una estructura simple, lo más destacable de la misma es el arreglo que contiene a los 3 vértices, dentro de cada vértice esta almacenando el valor actual de las coordenadas y no un índice como antes. En el vértice también se almacena la posición real de las coordenadas de textura, lo que nos lleva al punto más ineficiente de la estructura. Cada polígono debe almacenar su copia de la textura a utilizar si se desea que el polígono sea totalmente autónomo. Si se sacrifica esta condición se podría crear una implementación intermedia la cual sería menos demandante en el consumo de memoria.

### 6.3 Estructura de programa y pipeline final




Existen numerosas maneras de crear un pipeline dependiendo de las funcionalidades que vaya a tener el motor así como del orden en que se desea que se implementen las operaciones. Habiendo presentado los conceptos teóricos para desarrollar el engine se presenta el pipeline que ha sido implementado en esta investigación y se detalla el porqué se decide realizar las operaciones en el orden seleccionado. [19]

Antes de que comience el ciclo de procesamiento tenemos una etapa de carga en el motor, en la misma se llevan a cabo operaciones que no necesitan ser realizadas en tiempo real. De esta manera se evita utilizar ciclos de cálculo en operaciones que no son críticas, veremos a continuación las operaciones que se realizan en la etapa de precálculo. Se configuran las ventanas para utilizar DirectX, se precalculan los valores de las funciones trigonométricas seno, coseno y tangente para los 360° a fin de evitar la llamada a estas funciones en tiempo real. Una de las operaciones más importantes es la carga de los modelos a utilizar en la escena, el motor permite a partir de funciones propias crear algunos modelos básicos.

El cargador utilizado está basado en un parser para archivos .cob (Caligari Truspace), el parser ya se encontraba desarrollado y por lo tanto solo se tuvo que adaptar a las estructuras de datos implementadas. Las funciones para crear objetos dentro del motor permiten crear un cubo y un rectángulo, estas funcionalidades permiten tener acceso rápido a objetos básicos con los cuales probar los métodos más simples del motor. La necesidad de un cargador de objetos se hace evidente cuando se desea tener objetos más detallados ya que definir los mismos manualmente seria una tarea altamente compleja y se necesitaría también implementar un algoritmo que ordene los vértices de los polígonos en CW o CCW. La necesidad de tener un orden coherentes para los polígonos proviene de las funciones de back-face culling e iluminación. En ambas se necesita calcular las normales de los polígonos, las mismas tendrán su dirección definida de acuerdo a si los polígonos se encuentran ordenados en CW o CCW, por lo tanto es necesario que todos los polígonos estén definidos con el mismo orden. Asimismo cuando se desea trabajar con texturas es necesario tener una lista de coordenadas de texturas que definen la región de la textura que será asignada a cada polígono. Se repite la situación: esta tarea es altamente compleja de realizar a mano salvo para los casos más básicos.

En la etapa de preloading también se crea la lista de luces que iluminará la escena así como las propiedades de las mismas: color, intensidad, orientación, posición, etc. Las luces se implementan como una lista a fin de que sea sencillo eliminar y añadir luces. Aun así, las luces poseen un atributo que indica si la misma esta o no activa a fin de simular el encendido o apagado, por lo tanto la eliminación de luces de la lista no es una operación común. La última tarea que se realiza en esta etapa es el cálculo de los radios de las esferas que se compararán contra el frustum visual con el objetivo de eliminar objetos. Como el radio del modelo solo podrá variar cuando es sometido a una función de escalado, se puede calcular su valor en base a las coordenadas en espacio objeto ya que al transformarlo al espacio mundo a lo sumo se requerirá someter al radio a una simple multiplicación.

Cabe destacar que en esta etapa también podrían precalcularse las normales de los polígonos pero se debe tener en cuenta que estas normales deberán ser transformadas junto al objeto del que son parte

ya que el valor de las normales cambiará ante la rotación y el escalado. Debido a esta necesidad de someter las normales a varias operaciones de transformación a lo largo de la etapa de procesamiento es que se decide no realizar el precálculo. Es probable que no se gane un beneficio notorio en rendimiento a partir de este método. Aun así podría ser una funcionalidad interesante a implementar, a fin de comparar técnicas de mejora de rendimiento.

Finalizada la etapa de precálculo se inicia el ciclo de procesamiento del motor. Aquí se definen las posiciones, tamaños y direcciones de los objetos de la escena, estas operaciones también podrían realizarse en la etapa de precálculo. Pero, a menos que los objetos sean estáticos, se necesitará redefinir estos valores durante la ejecución por lo que en la mayoría de los casos sólo se ahorrará el cálculo en el primer ciclo de ejecución algo totalmente trivial. También se pueden activar o desactivar luces, modificar sus posiciones, color, etc. Como se puede ver en esta etapa se define el comportamiento de los objetos, luces, cámaras de la escena definiendo el dinamismo y estética que se le quiere dar a la misma. Para modificar los valores de un objeto se utilizan funciones que asignan los valores al objeto seleccionado, por más que los objetos pueden ser accesados libremente ya que son estructuras C y por lo tanto no contienen una parte de datos privada. Se opta por utilizar funciones para abstraer al desarrollador de conocer la estructura interna de los objetos. A su vez estas funciones almacenan en la estructura los valores por los que será transformado el objeto pero no realizan las transformaciones directamente. Una vez finalizadas todas las modificaciones que se desean realizar a la escena, cada objeto debe llamar la función transformar_objetos(). Esta función es una de las piedras angulares del motor ya que toma los objetos en espacio de modelo y al finalizar la función los objetos tendrán sus valores en coordenadas de pantalla. La decisión de que cada objeto llame individualmente a la función se tomo ya que facilita las pruebas durante el desarrollo. Aun así durante la ejecución el motor almacena los objetos en un arreglo global objs, por lo tanto si se desea automatizar el procesamiento del arreglo basta con crear un ciclo for cuando se llama a funciones que reciban objetos como parámetros. La decisión de realizar todas estas transformaciones dentro de una sola función es en parte para evitar un gran número de llamadas a funciones lo cual ralentizaría la ejecución y en otra parte para abstraer al desarrollador del funcionamiento interno del motor.

Como vemos la gran mayoría de las etapas del engine se llevan a cabo dentro de la ya mencionada transformar_objetos(). La primera de estas etapas es la eliminación de objetos enteros con la prueba de las esferas circundantes. Por más que los objetos aun no hayan sido transformados al espacio mundo se tiene almacenada en la posición del centro de los objetos cuando sean transformados. Por lo tanto, es posible realizar la prueba en este momento logrando así eliminar objetos enteros cuando todavía se encuentran en el espacio de modelo evitando la transformación a espacio mundo. A continuación se realiza la transformación a espacio mundo, para ello se realiza primero la rotación, el escalado y por último la traslación. Este orden permite que la rotación no sea en base al centro del objeto sino a otra posición arbitraria, simplemente se reemplaza momentáneamente el centro del objeto por el nuevo valor y se realiza la rotación sin necesidad de las traslaciones necesarias si ya estuviésemos en espacio mundo. Se realiza la eliminación de caras traseras antes de la transformación al espacio cámara para evitar transformar geometría que no será visible desde la perspectiva de la cámara. Dado que se deben calcular las normales de los polígonos para llevar a cabo este algoritmo es posible almacenar las mismas para no repetir el cálculo en la etapa de iluminación. No obstante, si la iluminación se encuentra luego de la transformación a espacio cámara será necesario transformar las normales.

El siguiente paso es transformación de los objetos a espacio cámara, la matriz de transformación que incluye la traslación y rotación ha sido precalculada antes de ingresar a transformar_objetos. De esta manera se realiza el cálculo una sola vez por ciclo de ejecución, para llevar a cabo la transformación simplemente se multiplica la matriz por los vértices del objeto. Una vez en espacio cámara se realiza el clipping de polígonos contra el volumen visual. En el clipping se eliminaran los polígonos que se encuentren totalmente fuera del volumen para x,y,z. Existe una excepción a la regla, los polígonos que atraviesan parcialmente el plano z-cercano deben ser clipeados contra el plano. Si no se realiza el clipping, los polígonos que atraviesan el plano serán procesados erróneamente en la etapa de proyección y cabe la posibilidad de que se realice también una división por 0 en la misma etapa. El clipping contra el volumen visual se realiza antes de la etapa de iluminación para eliminar aun mas geometría y sólo iluminar los polígonos absolutamente necesarios. En caso de implementar el precálculo de normales antes de la etapa de iluminación, es necesario recalcular las normales de los polígonos que atraviesen z ya que el área de los mismos cambiara.

Finalizado el clipping contra el volumen visual se llega a la etapa de iluminación. En la misma se podrá iluminar a los objetos con luces ambientales, direccionales, puntuales, y spotlights, haciendo que los mismos sean sombreados de forma emisiva, facetada o por Gouraud. Los tipos de luces implementadas modelan a las fuentes de luz más comúnmente encontradas en la realidad y nos permiten simular la gran mayoría de las escenas que deseemos crear en el motor. Se ha utilizado un modelo simplificado para las spotlights ya que son un tipo de luz altamente demandantes desde el punto de vista computacional. Aun con estas simplificaciones no se recomienda utilizar una gran cantidad de spotlights en la escena. Cabe destacar que las luces solo poseen la componente de luz difusa, a modo de reducir la cantidad de cálculos a realizar. La funcionalidad para la luz especular está parcialmente implementada pero no ha sido probada, utilizar este tipo de luz ralentizaría al motor por lo que se decide no continuar con su implementación Los tipos de sombreado seleccionados son los más simples que se pueden modelar. Esto se permite realizar cálculos medianamente simples aun así, el sombreado Gouraud es un método relativamente complejo y demandante. Por lo tanto utilizar métodos de iluminación y sombreado más complejos sería demasiado demandante para un motor implementado por software. A pesar de todos los compromisos mencionados, el sistema de iluminación ayuda a mejorar considerablemente la calidad y el realismo de las imágenes generadas por el motor.

Concluida la etapa de iluminación se transforman los objetos visibles e iluminados de la escena a sus coordenadas finales en pantalla a través de una transformación que aúna la proyección al plano visual y el mapeo del plano visual al puerto de visión. Esta transformación permite que el FOV sea definido por el usuario y la distancia al plano de visión sea arbitraria, esto permite un alto grado personalización a la vez que reduce tiempo de procesamiento integrando dos transformaciones en una. Aquí finaliza la función transformar_objetos(), pero antes de llamar a la función de rasterización crear_renderlist() recorrerá el arreglo global de objetos, a fin de copiar sólo los polígonos de cada objeto que continúen activos en una lista de polígonos a la cual se llamará renderlist.

Podemos notar que una vez que los polígonos se encuentren en la renderlist se pierde el concepto de objetos individuales, solo tendremos el conjunto de polígonos que serán rasterizados. Para evitar la superimposición de ciertos elementos en la escena es necesario ordenar los polígonos de la renderlist en base a su posición en z, es decir a su profundidad en la escena. Existen varios métodos para realizar esto, el implementado es uno de los más simples y se llama algoritmo del pintor. Está basado en como un pintor pinta un cuadro en la vida real, primero pintando los objetos más lejanos para luego pintar los más cercanos. La renderlist es una estructura del tipo lista lo cual facilita la aplicación de los algoritmos de ordenamiento. El algoritmo utilizado es el ordenamiento de burbuja el cual es uno de los más simples y por lo tanto ineficientes, por lo que el motor podría beneficiarse de un incremento en rendimiento con la implementación de otro algoritmo La última función de importancia que se llamará en el ciclo de ejecución es la llamada al rasterizador. Esta se realiza mediante la función rasterizer_wrapper() la cual analiza las características de cada polígono a fin de llamar al rasterizador correspondiente. Si se desea que la escena se dibuje en wireframe se debe realizar el llamado directo a la función rasterize_wire, la misma no fue incluida en el wrapper ya que es una excepción que en general no se utiliza evitando así una comparación dentro de la función que en general no ocurrirá. Si se quiere utilizar wireframe simplemente se llama desde el ciclo de ejecución. [13][16][20]

### 6.4 OpenCV

A continuación veremos cómo se llevan a cabo las funciones de motion tracking del motor. En el caso del seguimiento de objetos veremos qué pasos son necesarios para llevar para implementar esta funcionalidad mientras que en el caso de detección de caras se estudiará con más detalle los principios teóricos en los que se basa ya que la implementación es básicamente la llamada a la función de detección.

Las funciones de detección trabajan con imágenes capturadas de una webcam o un video, OpenCV tiene una serie de funciones que permiten la captura de estas imágenes de una forma simple para el usuario. Se pueden capturar imágenes tanto con archivos de video como con cámaras web sin grandes modificaciones en el código. Para comenzar debemos declarar un puntero 'CVCapture * captura' que apuntara al stream de datos capturados y, un puntero a la estructura de datos que utiliza OpenCV para las imágenes 'IplImage * frame' donde se guardara la imagen capturada. Una vez declarados los punteros solo se debe llamar a 'captura = cvCaptureFromCAM(0)' este método encuentra las cámaras web

instaladas en el sistema, con el parámetro 0 se encarga de capturar las imágenes captadas por la primer cámara detectada. Finalmente en el ciclo de ejecución se utilizara ' frame = cvQueryFrame( captura )' para obtener la imagen que ha sido capturada en ese momento de la ejecución del programa. Esta imagen capturada y almacenada en frame será la imagen a procesar en los 2 métodos implementados. [21][22]

### 6.4.1 Detección de objetos en base a color

Una vez almacenada la imagen en la variable 'frame' se utiliza la función cvCvtColor() la cual convierte elementos de un espacio de color a otro. Se utiliza el parámetro 'CV_BGR2HSV' para indicar que la imagen será convertida a HSV, debido a que es más simple identificar los colores a filtrar a partir del tono, y la intensidad de los mismos. Se utiliza CvSmooth() para suavizar la imagen, esta es una operación común dentro del procesamiento de imágenes. Se la suele utilizar para reducir el ruido o los artefactos generados por la cámara en una imagen. Es posible realizar distintas operaciones de suavizado, de acuerdo del valor del parámetro 'smoothtype' de la función.

La primera de las posibles operaciones es CV_BLUR, en ella se calcula el promedio de todos los pixeles encuadrados por una ventana alrededor del pixel a suavizar. Esta operación puede ser sensible a imágenes ruidosas, en especial cuando el ruido está aislado y tiene un valor alto, ya que grandes diferencias entre valores tendrán un alto impacto en el cálculo del valor promedio. Otra operación de interés es CV_MEDIAN donde cada pixel es reemplazado por la mediana de un cuadrado de pixeles centrado en el pixel a modificar. Esta operación evita los problemas del método de suavizado más simple ya que selecciona los valores medios, ignorando así los grandes diferenciales que pueden existir entre los datos.

El método CV_GAUSSIAN podría decirse que es el más útil a cambio de no ser tan veloz como los anteriores. El filtrado gaussiano se realiza mediante una convolución de cada punto con un kernel gaussiano para luego sumarlos a fin de lograr la imagen resultante. El filtrado Gaussiano se basa en que los pixeles en una imagen deben variar lentamente en el espacio y, por lo tanto ser correlativos a sus vecinos. Mientras que el ruido en una imagen se puede suponer que variará ampliamente de un pixel a otro, es decir se preserva la señal mientras se suaviza el ruido.

Por último tenemos el filtrado bilateral, el cual es parte de una serie llamadas "suavizado con preservación de bordes". El filtrado bilateral se encarga de eliminar el problema del filtrado de Gauss, se suaviza una imagen pero sin suavizar los bordes de los elementos que la componen. El proceso construye un promedio con pesos para cada pixel y de los pixeles vecinos. El sistema de pesos utilizado tiene dos componentes el primero es el utilizado en el método gaussiano: basado en la distancia espacial desde el pixel a modificar y los vecinos. El segundo se basa en la diferencia en intensidad en comparación al pixel central. Esto nos permitirá encontrar los elementos de la imagen relacionados al pixel examinado así como el momento donde se pierde la relación, lo cual indica la presencia de un elemento nuevo y por lo tanto de un borde.

En nuestro caso utilizamos 'cvSmooth( frameHsv, frameHsv, CV_GAUSSIAN, 7, 7,0,0)' ya que el filtrado Gaussiano nos permite un buen compromiso entre la eliminación de ruido y rendimiento. Una vez filtrado el ruido de la imagen se utiliza cvInRangeS para filtrar la imagen capturada con los valores mínimos y máximos de color definidos por el usuario en la interfaz del programa. Se almacena la imagen filtrada en frameResultado y se prosigue a encontrar los componentes interconectados de la misma. Antes de comenzar la búsqueda de componentes se somete la imagen a las operaciones morfológicas de apertura y clausura a fin de eliminar aun más ruido y agrupar conjuntos de pixeles interconectados (para más detalles ver el anexo sobre operaciones morfológicas básicas).

Si se desea encontrar los elementos interconectados de una imagen se deben buscar contornos, los cuales son una secuencia de puntos que representan algún tipo de curva en el espacio de la imagen. Como estas secuencias de puntos se dan normalmente se han creado en OpenCV una serie de funciones para manipularlas, una de estas funciones se encarga de encontrar los contornos de una imagen 'cvStartFindContours()'. Para poder hacer uso de esta función se requiere una estructura de datos conocida como un scanner de contornos 'CvContourScanner scanner' en la cual se almacenara la lista de contornos hallados. Una vez obtenidos los contornos se recorre la lista de los mismos, a fin de crear una aproximación de su forma con polígonos en base al algoritmo Convex Hull (cvConvexHull2() ).Una vez obtenidas las aproximaciones poligonales se calcula el centro de masa con cvMoments a fin de encontrar

el centro del objeto. Con esto datos se tiene toda la información necesaria para operar en base a los objetos encontrados. En el motor utilizamos el centro del objeto detectado como un punto de comparación, de acuerdo a la variación de posición del mismo, moveremos un elemento de la escena replicando el movimiento del objeto reconocido. [21][22]

### 6.4.2 Detección de caras

La detección de caras es un método que se encuentra dentro del campo conocido como aprendizaje automático (machine learning). El objetivo del machine learning es convertir datos en información, luego de entrenar a una maquina con un conjunto de datos se desea que la misma sea capaz de responder preguntas sobre los datos analizados. La extracción de reglas y patrones a partir de los datos es la manera en la que se obtiene información de los mismos. La detección de caras se basa en 'entrenar' a una computadora a partir de imágenes de caras para que pueda extraer su propio juego de reglas y patrones que le permitan lograr detecciones exitosas .

Cuando se entrena a una computadora, se procesan datos para obtener características que los definan. Por ejemplo si se toma una base de datos de 12.000 imágenes de caras, se puede aplicar un detector de bordes para juntar datos como las instancias donde se detectan bordes, la intensidad de los mismos, y la distancia entre estos y el centro de las caras. Se pueden obtener por ejemplo, 500 de estos valores para cada cara evaluada. Una vez recolectada esta información se utilizan técnicas para construir un modelo de los datos recolectados. Si se desea clasificar caras según sus características (anchas, angostas, etc.) se utilizará un algoritmo de agrupamiento (clustering), mientras que si se desea predecir la edad de una persona basado en los patrones de los bordes detectados en una cara, se utiliza un algoritmo de clasificación. Para lograr este tipo de detecciones los algoritmos de aprendizaje automático deben analizar las características recolectadas y, ajustar pesos, umbrales, y demás parámetros para maximizar las capacidades de detección de acuerdo al objetivo deseado. Este proceso es lo que se conoce como aprendizaje.

En el aprendizaje automático se puede trabajar con 2 tipos de datos, los que tienen etiquetas y los que no. Las etiquetas son una señal de aprendizaje que se usará para clasificar los datos procesados. Al aprendizaje con etiquetas se lo conocer como aprendizaje supervisado en el mismo se puede aprender a asociar un nombre a una cara, también se puede tener etiquetas numéricas las cuales pueden simplemente asignar un valor numérico o indicar un orden en los datos. Si los datos procesados se asocian en base a categorías se dice que se realiza una clasificación en cambio, si los datos son numéricos el proceso se llama regresión. Cuando se tienen datos no supervisados (sin etiquetas) existen algoritmos para estudiar si los datos caen naturalmente en grupos, los mismos se llaman algoritmos de agrupamiento. El objetivo de estos algoritmos es agrupar datos que se encuentran cercanos, la cercanía puede ser en base a parámetros predeterminados o aprendidos. Los datos no supervisados y agrupados suelen utilizarse para crear un vector de características para ser utilizado en un clasificador de mayor nivel. El clustering y la clasificación son dos de las tareas más comunes del aprendizaje automático, las mismas se solapan con dos de las tareas más comunes en la visión de computadora, la segmentación y el reconocimiento respectivamente. Para la detección de caras es necesario reconocer las mismas por lo tanto se emplea un clasificador Haar para lograrlo, este clasificador está basado arboles de decisión, donde se crea una cascada de rechazos acelerada (boosted rejection cascade). Si se utiliza otro conjunto de datos y no caras el clasificador podría ser entrenado para reconocer otros objetos.

El detector usado en OpenCV está basado en el detector de Viola - Jones extendido para usar características Haar. La suite contiene un conjunto reconocedores de objetos ya entrenados, y también permite el entrenamiento de nuevos detectores. El detector de Viola-Jones se basa en AdaBoost, un algoritmo para acelerar el rendimiento de los arboles de decisión basado en crear un clasificador fuerte a partir de varios clasificadores débiles, pero se organiza como una cascada de nodos de rechazos. Cada nodo está conformado un árbol de decisión de un solo nivel de profundidad, estos nodos solo pueden decidir condiciones del tipo: "es el valor V de una característica particular C mayor o menor a un umbral U", donde "Sí" indica una detección y "No" indica una no detección. Los nodos están diseñados para tener una tasa de detección cercana a 100%, es decir que tiene pocas instancias de caras no detectadas, pero con el costo de una tasa de rechazo cercana al 50% es decir una alta presencia de falsos positivos. Para cualquier nodo si se obtiene el resultado "no incluido", se termina el procesamiento de la cascada y se declara que no existe el objeto a detectar en esa posición. Para que se declare una detección real se

debe procesar la cascada entera. En escenarios donde las detecciones reales son pocas, una cara en una foto por ejemplo, las cascadas de rechazo reducen considerablemente el tiempo de procesamiento ya que la mayoría de las áreas de la imagen son rechazadas rápidamente por la cascada al no contener el objeto de interés.

Los nodos son organizados dentro de la cascada de detección del más simple al más complejo a fin de que su procesamiento sea más veloz. De esta forma se minimizan los cálculos cuando se rechazan regiones de la imagen. Como ya se ha mencionado estos nodos suelen tener un alto número de detecciones alrededor de 99% y un algo grado de falso positivos alrededor de 50%, pero al aumentar la cantidad de nodos, se mantienen valores aceptables para las detecciones minimizando los falsos positivos. Por ejemplo con 20 nodos se tiene una detección delo 98% y una detección de falsos positivos de 0.0001%. En la practica el 70–80% de los no-objetos son detectados en los primeros 2 nodos de la cascada de rechazo, este tipo de detección temprana reduce dramáticamente la velocidad de la técnica de detección. Veamos extractos de código para ver como se implementa la detección de caras.

```
CvHaarClassifierCascade* cascade = (CvHaarClassifierCascade*)cvLoad( “data/haarcascades/haarcascade_frontalface_alt_tree.xml” );
faces = cvHaarDetectObjects( frame, cascade, storage, 1.2, 1, CV_HAAR_FIND_BIGGEST_OBJECT , cvSize( 75, 75 ));
```

Como se puede ver la implementación es básicamente trivial, se utiliza la función cvLoad para cargar un archivo XML que contiene la cascada/árbol de clasificación y se utiliza la función cvHaarDetectObjects() para realizar la detección en base a la cascada seleccionada y la imagen capturada. Veamos en más detalle la función para detectar objetos, comenzando por su prototipo.

```
CvSeq* cvHaarDetectObjects( const CvArr* image, CvHaarClassifierCascade* cascade, CvMemStorage* storage, double scale_factor, int min_neighbors, int flags, CvSize min_size);
```

La función requiere que la imagen a analizar sea en escala de grises, y que se le indique qué cascada de clasificación utilizar. Se puede seleccionar una región de interés con lo cual solo se buscaran detecciones en la región correspondiente a fin de simplificar la búsqueda y minimizar el tiempo de procesamiento. La cvHaarDetectObjects analiza la imagen de entrada en todas las escalas para caras, se puede seleccionar el valor del parámetro ‘scale factor’ para indicar el salto entre las escalas a utilizar, a mayores saltos más velocidad en la detección a riesgo de pérdida de detecciones. El parámetro ‘min_neighbors’ ayuda a evitar las falsas detecciones, cuando se detecta una cara se tendrán varias detecciones de la misma en las áreas cercanas de la imagen. Este parámetro indica la cantidad de detecciones solapadas que deben darse para determinar que hay una cara presente. Mediante los flags se podemos configurar el comportamiento del algoritmo por ejemplo, para que se detenga luego de la primera detección. Por último con el parámetro ‘min_size’ se define el tamaño mínimo la región en la cual buscar caras, agrandar este valor hace q caras pequeñas no se encuentren, pero acelera el tiempo de procesamiento. Por último podemos ver que la función retorna una secuencia, por lo tanto se pueden utilizar las mismas funciones que ya hemos visto para detección de objetos en base a color, para encontrar la posición de la cabeza en una imagen. En este caso se modificarán los ángulos de rotación de la cámara en base a la variación de los movimientos de la cabeza del usuario. [21][22]

# 7 - Conclusiones y recomendaciones

Los objetivos planteados al inicio de este trabajo han sido cumplidos. Se ha escrito una tesis extensa donde se estudian todos los componentes fundamentales para generar un pipeline 3D sin ahondar en los detalles de implementación. El segundo objetivo de crear un motor 3D con capacidades de detección de movimiento también ha sido logrado. El desarrollo ha sido un proceso altamente complejo, que sirvió como herramienta de aprendizaje.

El desarrollo ha sido un proceso iterativo, donde primero se crearon las funciones y estructuras de datos necesarias para poder mostrar simples elementos geométricos en pantalla y se fue creciendo a partir de esto. Este tipo de metodología de trabajo lleva a que se realicen reiteradas revisiones sobre los componentes ya implementados. Debido a la interconexión de los distintos componentes la resolución de errores es un proceso difícil ya que los mismos ocurren en cualquier etapa del pipeline. Estos inconvenientes hacen que el programador a medida que avanza con el desarrollo tenga cada vez un mayor entendimiento de cómo funciona el pipeline 3D y como se interrelacionan sus componentes.

Sobre las posibles funcionalidades a agregar se recomienda eliminar la etapa de rasterización por software y utilizar una aceleradora grafica para esta tarea. El rasterizador es una de las etapas que más tiempo consume dentro del motor y su implementación en hardware es una de las más simples. Esto permitirá añadir nuevas funcionalidades gracias al rendimiento adquirido.

La inclusión de OpenCV en el proyecto ha hecho que el rendimiento celosamente cuidado durante el desarrollo del motor, sea poco notable. Al integrar el engine con algoritmos de la suite de computación visual, se integran 2 sistemas altamente complejos y computacionalmente caros, la CPU simplemente no puede responder a la demanda con tiempos óptimos. Aún así, el sistema no se ha vuelto inutilizable pero sí lento, por lo que el rendimiento dependerá de la complejidad de la escena que deba ser dibujada. Si no se utiliza OpenCV el motor funciona a velocidad óptima.

Cabe destacar que aún con el cuidado que se la ha dado a crear un motor veloz, solo se ha visto la punta del iceberg que es el área de optimización de código. Es posible realizar múltiples mejoras tanto desde el punto de vista algorítmico como del aprovechamiento de hardware por ejemplo, la utilización de instrucciones SIMD y el multithreading.

El motor presenta un grado de flexibilidad y facilidad de uso muy atractivos como para poder ser la base de futuros proyectos. Esto es lo más interesante de un proyecto como este: el mismo puede ser utilizado y expandido en infinidad de direcciones. Se pueden seguir desarrollando algoritmos gráficos aumentar la fidelidad de la imágenes generadas. Se puede unir a una gran variedad de sistemas por ejemplo: sistemas de animación, de física, de comunicación por red, de sonido. Todo depende de qué tipo de uso se le quiera dar al sistema, así que el mismo queda abierto a futuras expansiones.

# Glosario

**Aprendizaje automático:** es una disciplina basada en el diseño y desarrollo de algoritmos que permitan que las computadoras obtengan una serie de comportamientos basados en datos empíricos.

**Árbol de clasificación:** es un árbol de decisión utilizado como un modelo predictivo, mapea observaciones a conclusiones sobre el valor que adquirirá un elemento.

**Buffer:** un buffer de datos es una ubicación en memoria reservada para almacenamiento temporal.

**Cascada de rechazo:** es un árbol de clasificación con nodos débiles, se caracteriza por tener un alto grado de detección pero también un alto grado de falsos positivos.

**Clipping:** el cual es el proceso de sólo dibujar los elementos que sean visibles por el usuario.

**Computación visual:** es la ciencia de hacer que las maquinas puedan extraer información de una imagen que les permita resolver algún tipo de tarea designada.

**DirectX:** conjunto de APIs que simplifican el proceso de creación de videojuegos u otras aplicaciones de computación gráfica.

**Eliminación de caras traseras / Backface culling:** determina si un polígono es visible o no de acuerdo a su posición en relación a la cámara.

**Eliminación de objetos contra el volumen visual / Frustum culling:** Son técnicas para eliminar objetos enteros contra el volumen visual de la cámara.

**Escena: es el conjunto:** geometría, luces, texturas, cámara, etc. a ser dibujados por el motor 3D.

**Interfaz de programación de aplicaciones / API:** conjunto de funciones y procedimientos para ser utilizado por otra aplicación como una capa de abstracción.

**Framebuffer:** es un dispositivo de salida que genera imágenes las cuales suelen estar almacenadas como pixeles en un buffer de datos.

**Imagen raster:** las imágenes raster o mapas de bits son estructuras de datos formadas por una grilla rectangular de pixeles como medio para almacenar imágenes.

**Malla de polígonos:** es un conjunto de polígonos interrelacionados e interconectados.

**Modelo / Objeto:** es una representación matemática de un objeto o superficie a fin de poder modificar a la misma mediante cálculos. Está formada por una malla de polígonos y su conjunto de vértices.

**Morfología de imágenes:** es una teoría y un conjunto de técnicas para el análisis y procesamiento de estructuras geométricas, aplicado principalmente a imágenes digitales.

**OpenCV:** es una librería de funciones de programación para desarrollar aplicaciones de computación visual en tiempo real.
Plano de visión: es un plano bidimensional al cual se proyecta la escena 3D en base a la posición de una cámara virtual.

**OpenGL:** es una API gráfica que se utiliza para la creación de imágenes 3D.

**Operaciones de esténcil:** se utilizan como una máscara pixel por pixel para almacenar o descartar pixeles, su operación más común es la de añadir sombras en aplicaciones de gráficos 3D.

**Operaciones del buffer z / buffer de profundidad:** determina qué elementos de una escena renderizada son visibles y cuáles no. El objeto que tenga un valor z menor, es decir esta más cercano a la cámara

será visible mientras que los objetos con valores mayores no, esta operación se realiza pixel por pixel.

**Pixel:** es la unidad más pequeña que compone a un archivo de imágenes, así como el elemento más pequeño direccionable en una pantalla.

**Polígono:** es una figura plana definida por un camino cerrado, en la computación gráfica se suele llamar polígono a los triángulos los cuales suelen ser las componentes geométricas más simples de un modelo 3D.

**Primitiva:** son los objetos geométricos más simples que puede manejar un sistema, en general los mismos suelen ser: puntos, líneas y triángulos.

**Puerto de visión:** es un plano 2D cuyos valores corresponden a las coordenadas de la pantalla donde se mostrara la imagen 3D generada.

**Rasterización:** es el proceso de convertir una imagen descripta en gráficos vectoriales (matemáticamente) y convertirla en una imagen raster (mapa de bits).

**Renderlist:** es una lista que contiene los polígonos a dibujar en pantalla, en la misma se pierde el concepto de relación entre los polígonos que antes formaban parte de un mismo objeto.

**Vértice:** es un punto que describe las esquinas o puntos de intersección de las figuras geométricas. En la computación gráfica se lo utiliza para definir las esquinas de los polígonos.

**Volumen visual / view frustum:** es la región de una escena 3D que puede aparecer en pantalla, ya que corresponde al campo visual de la cámara el cual está definido por una pirámide rectangular.

**Rendering:** es el proceso de generar imágenes raster a partir de modelos tridimensionales.
**Shader:** es un conjunto de instrucciones utilizados para llevar a cabo distintas operaciones de rendering sobre el hardware grafico con un alto grado de flexibilidad.

# Bibliografía

[1] Teach Yourself Game Programming With DirectX in 21 Days – Clayton Walnum –Sams - 2002

[2]Tricks of the Windows Game Programming Gurus, Fundamentals of 2D and 3D Game Programming - André LaMothe - Sams - 1999

[3] Entrevista de Business Week con Doug Lowenstein - Fecha de acceso: 26/08/2009
http://www.businessweek.com/innovate/content/may2006/id20060511_715050.htm?campaign_id=rss_innovate

[4] Jeff Ward – Artículo "¿Qué es un motor de juegos?" – Fecha de acceso : 12/09/2009
http://www.gamecareerguide.com/features/529/what_is_a_game_.php

[5] Artículo de Wikipedia "Game Engine" – Fecha de acceso: 12/09/2009
http://en.wikipedia.org/wiki/Game_engine

[6] 3D Game Programming All In One – Kenneth Finney – Premier Press & Thomson – 2004

[7] The Direct3D 10 System, ACM Transactions on Graphics vol. 25 - David Blythe - ACM - 2006

[8] OpenGL Super Bible: Comprehensive Tutorial and Reference, Fourth Edition - Richard Wright Jr., Benjamin Lipchak, Nicholas Haemel - Addison Wesley - 2007

[9] Articulo con datos sobre motores de juegos comerciales – Fecha de acceso: 06/10/2009
http://artigames.com/blog/2009/08/motor-de-juego/

[10] Articulo sobre el costo de creación de videojuegos - Fecha de acceso: 06/10/2009
http://artigames.com/blog/2009/08/los-costos-enormes-en-la-produccion-de-un-videojuego/

[11]Real-Time Rendering - Tomas Akenine-Moller,et al – A K Peters Ltd. - 2008

[12] Articulo de Wikipedia "Frustum" – Fecha de acceso : 1/6/2010
http://es.wikipedia.org/wiki/Frustum

[13] Mathematics for Game Developers - Christopher Tremblay - Thomson - 2004

[14] Mathematics for 3D Game Programming and Computer Graphics, Second Edition - Eric Lengyel - Charles River Media - 2004

[15] Mathematics for Computer Graphics, Second Edition - John Vince - Springer - 2006

[16] Presentacion: Evolution of GPUs - Chris Seitz - NVidia - 2004

[17] Essential Mathematics for Games and Interactive Applications, A Porgrammer's Guide, 2nd Edition - James Van Verth, Lars Bishop - Morgan Kaufmann - 2008

[18] 3D Game Engine Desing, A Practical Approach to Real-time Computer Graphics - David Eberly - Morgan Kaufmann

[19] 3D Game Engine Programming - Stefan Zerbst, Oliver Duvel - Thomson - 2004

[20] 3D Math Primer for Graphics and GameDevelopment - Fletcher Dunn, Ian Parberry- Wordware Publishing, Inc.- 2002

[21] Learning OpenCV: Computer Vision with the OpenCV Library - Gary Bradski, Adrian Kaehler - O'Reilly - 2008

[22] OpenCV Wiki - Fecha de acceso: 05/08/2010 - http://opencv.willowgarage.com/wiki/

[23] Artículo de Wikipedia “Rendering (computer graphics)”- Fecha de acceso: 15/09/2010 http://en.wikipedia.org/wiki/Rendering_(computer_graphics)

[24] How to Leverage an API for Conferencing - Fecha de acceso: 16/09/2010 - http://comunication.howstuffwork.com/how-to-leverage-a-api-for-conferencing.htm

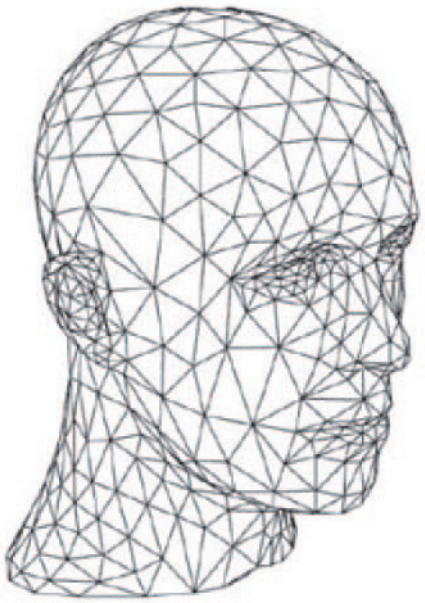
[25] Artículo de Wikipedia “Interfaz de programación de aplicaciones” - Fecha de acceso: 16/09/2010 - http://es.wikipedia.org/wiki/Interfaz_de_programación_de_aplicaciones

# Anexo 1 – Computación Gráfica

## A1.1 Primitivas y objetos

Cuando nos encontramos en el mundo de la computación gráfica es común escuchar la palabra “primitivas”, en este contexto la primitivas son las entidades geométricas más simples que es capaz de procesar nuestro sistema gráfico. En la mayoría del hardware grafico las mismas serán: puntos, líneas y triángulos; en nuestro motor utilizaremos los triángulos como la única primitiva que podemos dibujar directamente. En realidad para dibujar triángulos es necesario poder dibujar líneas y para dibujar líneas es necesario dibujar pixeles individuales así que las 3 funcionalidades se encuentran intrínsecamente, pero como usuarios solo tenemos acceso al dibujo de triángulos.

Las primitivas son de especial interés ya que los modelos u objetos que queremos representar son básicamente una colección de primitivas, es decir que usan las primitivas como las formas geométricas básicas a la hora de representar un objeto. [11]



Modelo de una cara con triangulos como primitivas
http://www.guru3d.com/article/nvidia-gf100-fermi-technology-preview/4
Fecha de acceso: 15/05/2010

## A1.2 Transformaciones geométricas

Las transformaciones son operaciones matemáticas que se aplican a entidades como puntos, vectores, normales, colores, etc. y cambian a los mismos en alguna manera significativa. Las transformaciones son muy importantes para el desarrollador, ya que con ellas el mismo puede posicionar, cambiar de forma y animar objetos, luces y cámaras y ubicar a los mismos en el mismo sistema de coordenadas. [11]

Existen diferentes tipos de transformaciones las mismas pueden describirse desde el tipo de modificación al que someterán al objeto (escalado, traslación, rotación), o también pueden ser descriptas bajo sus propiedades matemáticas. Desde el punto de vista de las propiedades matemáticas el primer tipo de transformación que estudiaremos es la transformación lineal. Este tipo de transformaciones preservan la suma vectorial y la multiplicación de un vector por un escalar:

$f(x) + f(y) = f(x + y)$
$kf(x) = f(kx)$

El caso de la multiplicación de un vector por un escalar es lo que se llama transformación de escala o escalado, ya que se modifica el tamaño/escala de un objeto. La rotación es otra transformación lineal que hace girar a un vector alrededor del origen.

La transformación de traslación se basa en la suma de 2 vectores de 3 elementos cada, uno indicará el vector que deseamos mover y el otro será un valor fijo que indicara la cantidad por la que el objeto será movido de lugar. Las transformaciones afines aplican una transformación lineal y una traslación, como se puede ver esta es una buena forma de combinar transformaciones. En caso de querer utilizar las transformaciones afines las mismas suelen almacenarse en matrices de 4 x 4 elementos que utilizan notación homogénea. Cuando se utiliza notación homogénea el cuarto elemento es 0 si se está trabajo con vectores que indican una dirección, por ejemplo v = [vx,vy,vz,0]. El elemento w será 1 si hablamos de un vector de posición, el cual también podemos decir que es un punto, el mismo se representa mediante p = [px, py, pz, 1].

La principal característica de una matriz afín es que se preserva el paralelismo entre las líneas, pero no necesariamente el tamaño o los ángulos de las mismas. Esta condición es ideal para nuestros propósitos ya que nos permite modificar los tamaños y ángulos de los objetos (escalado y rotación), pero, a su vez, mantiene la relación entre las líneas que conforman a los modelos evitando que los mismos se deformen. A su vez una transformación afín puede ser el resultado de una concatenación de transformaciones afines individuales, lo cual puede ser utilizado como una técnica de optimización ya que se pueden unir todas las transformaciones en una sola matriz y realizar un solo calculo. A continuación vemos con mayor detalle las 3 transformaciones básicas que utilizamos en el mundo de la computación grafica: traslación, rotación y escalado.[11]

### A1.3 Traslación

Cuando queremos mover un objeto de un lugar a otro utilizamos una traslación. Sumamos el vector de traslación t [tx,ty,tz] al punto que queremos mover:

$p' = p + t$

$[x', y', z'] = [x+tx, y+ty, z+tz]$

En caso querer utilizar una matriz de transformación la misma es:

$$T(t) = T(tx,ty,tz) = \begin{pmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Es interesante ver que la notación homogénea se comporta de manera adecuada cuando estamos frente a una traslación: En el caso de multiplicar la matriz de transformación por un punto p[x,y,z,1], el resultado será: p' = [x + tx, y + ty, z + tz, 1], mientras que si multiplicamos por v[x,y,z,0] el vector no es afectado por la multiplicación. Esto se debe a que los vectores de dirección no son afectados por la traslación, como vemos utilizar transformaciones afines nos puede ayudar evitar problemas, ya que de esta manera si inadvertidamente queremos trasladar un vector de dirección el mismo no será modificado aun si lo sometemos a la transformación. Por último, en caso de querer revertir la transformación aplicada la inversa de T(t) es T(-t). [11][13]

### A1.4 Rotación

Una transformación de rotación gira a un vector por un ángulo determinado sobre el origen del eje seleccionado. Este tipo de transformación nos permite definir la orientación de un objeto en el espacio, para así saber hacia dónde es "adelante" y "arriba" para cada objeto. Existen 3 matrices de rotación Rx, Ry, Rz, las cuales rotan φ radianes a un objeto alrededor del eje seleccionado.

$$Rx(\varphi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & -\text{sen}\,\varphi & 0 \\ 0 & \text{sen}\,\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

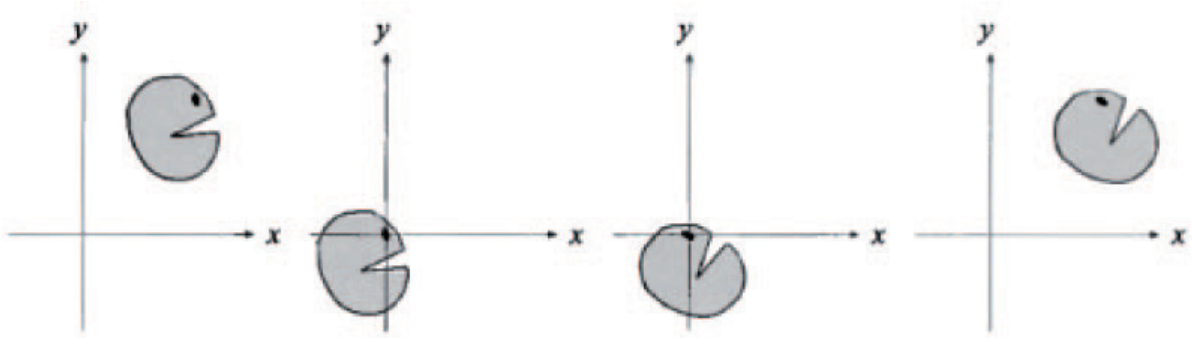
$$Ry(\varphi) = \begin{pmatrix} \cos\varphi & 0 & \text{sen}\,\varphi & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ -\text{sen}\,\varphi & 0 & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$Rz(\varphi) = \begin{pmatrix} \cos\varphi & -\text{sen}\,\varphi & 0 & 0 \\ \text{sen}\,\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Es interesante notar como la matriz de rotación sobre un eje no modifica los valores del vector en ese mismo eje, a su vez vemos como utilizando y, posiblemente concatenando, estas 3 matrices de rotación podemos rotar un objeto en cualquier dirección deseada. En caso de querer revertir la rotación la inversa de $R(\varphi)$ es $R(-\varphi)$.

Es posible que en este momento nos preguntemos, ¿qué hacer en caso de querer rotar un modelo alrededor de un punto determinado? La respuesta más simple a este problema es: teniendo en cuenta que si queremos realizar una rotación alrededor de un punto, el mismo no será modificado y que esto mismo ocurre con los puntos alineados a un eje cuando rotamos alrededor del mismo. Podemos concluir que debemos trasladar el punto para que coincida con el origen mediante una traslación, luego realizamos la rotación y volvemos a trasladar a la posición original del objeto.[11] [13]




Rotación de un objeto alrededor de un punto determinado. [11]

**A1.5 Escalado**

Como ya hemos señalado antes, una transformación de escalado se basa en la multiplicación de un escalar por un vector. Si k >1 entonces el modelo se agrandara, si 0> k < 1 el objeto será achicado, y k = 1 es la operación neutra. Existen 2 tipos de escalado: uniforme donde cada elemento del vector se multiplica por el mismo k, y no uniforme donde cada elemento es afectado por un k distinto. Vemos la matriz de escalado. [11] [13]

$$S(s) = S(sx,sy,sz) = \begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Si deseamos invertir la operación $S^{-1}(s) = S(1/sx, 1/sy, 1/sz)$.

# Anexo 2: Iluminación

En este momento tenemos los conocimientos necesarios para crear un motor grafico que cuenta con todas las capacidades necesarias para dibujar elementos en pantalla, aun así la representación no es muy realista ya que de momento solo conocemos como dibujar polígonos rellenos por un solo color, o en modo wireframe. Para poder aumentar el realismo de las escenas presentadas es necesario crear un sistema de iluminación, que se encargue de determinar como la intensidad de color de un objeto se verá afectada por las fuentes de iluminación presentes. Este sistema nos permitirá determinar así el brillo, color y sombreado de los objetos aumentando considerablemente la fidelidad grafica del motor.

La iluminación en computación grafica, estudia e intenta emular las interacciones entre la materia y los fotones. Los fotones son energía pura que presentan características de las ondas (frecuencia) y de las partículas (posición y velocidad). El color de la luz tiene que ver con la longitud de onda de la misma (frecuencia) y su intensidad con la frecuencia en que los fotones impactan contra una superficie. Como es de suponer al estar realizando un desarrollo por software no se intentará simular el comportamiento ni la física de los fotones o la materia pero, este conocimiento permite comprender que un sistema de iluminación requiere de operaciones que permitan determinar la intensidad y el color tanto de la luz como de los objetos de una escena.

Realizando observaciones en el mundo real, es simple notar que la luz interactúa de distintas maneras con los elementos afectado. Esto se debe al material que compone al objeto, el cual tiene una serie de propiedades que determinarán el tipo de interacción a llevarse a cabo. Por ejemplo parte de la luz puede ser absorbida, y parte reflejada, la luz reflejada como sabemos determina el color del objeto. Es posible que el objeto no sea completamente opaco y parte de la luz lo atraviese o se vea refractado en distintas direcciones, a su vez debido a la fosforescencia es posible que el mismo objeto emita luz al ser impactado por fotones. Es necesario tener presente las diferentes propiedades que pueden poseer los materiales, y que cuanto más complejidad se le dé al sistema de materiales, más realista será el comportamiento del modelo de iluminación. Para mantener un nivel de rendimiento aceptable en la implementación del motor, se considera que todos los objetos sólo poseen propiedades difusas es decir absorben los colores y reflejan el que representará su color final. Aún así, no sería altamente complejo implementar otras propiedades como especularidad y emisividad si se quisiera. [2]

### A2.1 Operaciones con colores

A la hora de hablar de colores en computación grafica es necesario saber que existen múltiples formas de representar y almacenar un color, por lo general se utilizará algún tipo de representación RGB para mantener una correspondencia con los colores usados por los monitores. En los formatos RGB cada canal de color se representa por un rango de valores por ejemplo de 0 a 1, 0 a 63, 0 a 255, etc. Las distintas representaciones tienen distintas utilidades, la representación decimal (0 a 1) puede facilitar operaciones matemáticas mientras que las representaciones en bytes pueden ser asignadas al buffer de video a la hora de dibujar los triángulos evitando realizar cálculos para convertir la representación decimal.

Es importante también conocer las operaciones que podemos aplicar a los colores y el significado de sus resultados. En muchas ocasiones tendremos luces de distintos colores afectando a los polígonos, y debemos saber cómo tratarlas para obtener el color final del polígono.

Comencemos con la adición de colores, si tenemos 2 colores RGB:
Ca (ra, ga, ba)
Cb (rb, gb, bb)

Cc = Ca + Cb = ( ra + rb , ga + gb, ba + bb)

En estos casos existe la posibilidad de que la suma de los componentes sea mayor al máximo valor permitido por el canal, por lo tanto se debe tener algún tipo de prueba para evitar que ocurran overflows. Otra operación de interés es la modulación de color, donde se multiplicará al color por un escalar.

Color-Modulado = k * C = (k * r, k * g, k * b)

El escalar puede ser cualquier numero de 0 a infinito, si k = 1 el color no es modificado, si k = 2 el color será el doble de brillante, si k = 0.10 el color tendrá un decimo de su brillo original, etc. Así como en la adición en este caso también se debe chequear que no ocurra un overflow. Existe otro fenómeno que debemos tener en cuenta cuando modulamos un color, si se modula un pixel hasta que uno de sus canales llega a su valor máximo y, se continúan incrementando los valores de los demás canales observaremos que los colores y la textura del modelo (si es que posee una) se distorsionarán. Esto se debe a que si se sigue incrementando los valores de los demás componentes RGB una vez que uno se ha saturado la relación entre los valores de los canales se pierde. Por lo tanto se debe dejar de modular el pixel una vez que se llega a la saturación de uno de sus componentes de color. Vemos ahora otra operación de modulación, en este caso la multiplicación entre 2 colores.

Color-Modulado = Ca * Cb = ( ra * rb , ga * gb, ba * bb)

Cuando modulamos colores es importante tener en cuenta que es lo que queremos lograr con la misma y el formato de los colores. Por ejemplo, si el formato de Ca es (0:1, 0:1, 0:1), lo que lograremos al multiplicar a Cb, sin importar en que formato RGB se encuentre, es modular su intensidad. Esto nos sirve por ejemplo si tenemos una textura monocromática la cual indica intensidades de luz, si la multiplicamos por un color estaremos modulando la intensidad del color haciéndolo más o menos brillante. Habiendo visto algunos conceptos básicos sobre los sistemas de iluminación y las operaciones que se realizan entre los colores , se continuará presentando los distintos tipos de luz y de iluminación que pueden encontrarse en una escena.[14] [17] [20]

### A2.2 Componentes de la luz

#### A2.2.1 Luz ambiental

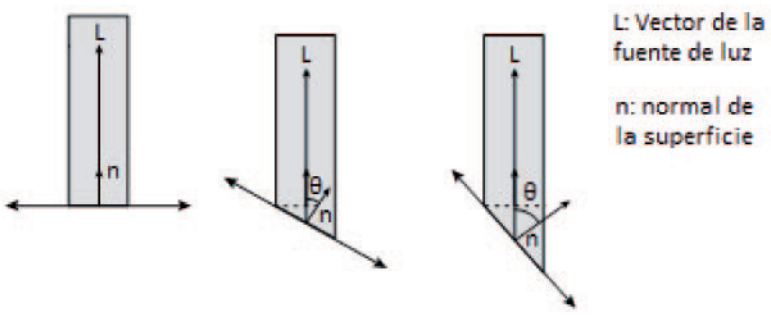
La luz ambiental es aquella que nos rodea, no tiene un origen definido sino que es el resultado de la sumatoria todas las fuentes de luz alrededor nuestro. Para representar el nivel de luz ambiental utilizaremos el producto de la reflectividad de color de la superficie a iluminar por la intensidad de luz ambiental. La luz ambiental se aplica a todos los polígonos de una escena sin importar la presencia o no de otras fuentes de iluminación, estableciendo así el nivel mínimo de luz en el mundo. Veamos la ecuación:

Intensidad-total-amb = RefColor-amb * Int-amb

Donde las reflectividades de colores se representan con RGB (0:1, 0:1, 0:1) y las intensidades con un escalar. [14] [17] [20]

#### A2.2.2 Luz difusa

La luz difusa es aquella que se dispersa de un objeto debido a la presencia de una fuente de luz y de la aspereza de la misma superficie. Este tipo de luz es invariante a la posición del observador y se dispersa en todas las direcciones de igual modo.




En la figura observamos como un haz de luz impacta con la superficie S, y la misma dispersa luz en todas direcciones, la cantidad de luz reflejada depende del ángulo que existe entre la superficie y la

fuente de luz .Podemos decir entonces que la intensidad de la luz difusa se define en función del ángulo θ que hay entre la normal de la superficie y el vector de la fuente de luz.

Intensidad ( n . l ) = cos θ donde n y l son vectores unitarios

Podemos comprobar que la ecuación es correcta observando que cuando la superficie tiende a volverse paralela a la fuente de luz el área de la misma disminuye, θ se acerca a 90° y la cantidad de energía reflejada es la mínima posible. Cuando la superficie tiende a estar perpendicular, la cantidad de dispersión crece, y θ tiende a 0. La mayor dispersión se da cuando la superficie y la fuente de luz son perpendiculares.

Los reflejos difusos están basados en el concepto de que las superficies están formadas por materiales que interactúan con la luz de diversas formas. Una de estas formas es esparcir la luz en todas direcciones debido detalles microscópicos que se encuentran orientados aleatoriamente en las superficies. Aun si este modelo no refleja la complejidad de la física del mundo real, es suficiente para desarrollar nuestro modelo de iluminación. Así como en el modelo de luz ambiental en este caso asumiremos que cada superficie tiene una reflectividad de color definida por RGB (0:1, 0:1, 0:1) a la cual llameros RefColor-dif. Observamos la ecuación de intensidad.

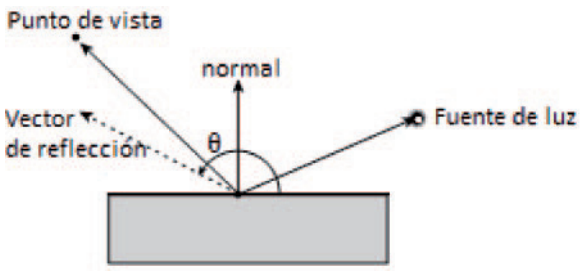
Intensidad-total-dif = RefColor-dif * Int-dif * ( n . l )

Para entender cómo se relacionan los distintos modelos de luz veremos a continuación la ecuación para un modelo con luz ambiental y difusa que tiene múltiples fuentes de luz. [14] [17] [20]

$$\text{Intensidad-total-amb- dif} = \text{RefColor-amb} * \text{Int-amb} + \text{RefColor-dif} * \sum_{i=0}^{n} [\text{Int-dif}(i) * (\text{normal- } i \,.\, \text{luz-}i)]$$

### A2.2.3 Luz especular

La iluminación especular es aquella que genera reflejos brillantes en los objetos que afecta, estos reflejos ocurren ya que un gran número de microfacetas en la superficie iluminada están alineadas en la misma dirección. El modelo de luz especular simula este efecto, el cual está relacionado con la fuente de luz, la normal de la superficie y la posición del observador.




Como podemos ver tenemos la superficie con su normal, el vector de punto de vista del espectador y el vector de reflexión. Cuando el vector de reflexión y el vector del espectador se alinean se verá el reflejo de la fuente de luz ya que la superficie se comportará como un espejo Es decir, cuando el ángulo θ entre los vectores tiende a 0 los reflejos se incrementan rápidamente, y cuando θ se incrementa los reflejos disminuyen rápidamente. Para definir la intensidad del reflejo se utilizara una variable llamada exponente especular que define la especularidad del material del objeto iluminado.

Intensidad-total-esp = RefColor-esp * Int-esp * MAX( r . e, 0) ˆee
donde r es el vector de reflexión, e es el vector del espectador y ee el exponente especular. [14] [17] [20]

### A2.2.4 Luz emisiva

La luz emisiva se basa en determinar cuanta luz emite una superficie, esto significa que la superficie se ilumina a sí misma y a su vez, actúa como una nueva fuente de luz que se debe añadir a los cálculos de iluminación. Existe una simplificación para este modelo donde simplemente utilizamos la emisividad para auto-iluminar la superficie ignorando que la misma es también una fuente de luz. Esto nos permite crear objetos que simulan ser una fuente de luz, visualmente parecen una luz y pueden ser “prendidos” y “apagados” pero, no son una fuente de luz en sí mismos.

Intensidad-total-emis= RefColor-emis

Como vemos la implementación de este tipo de luces es trivial y nos puede ayudar a simular una escena más realista sin agregar complejidad en los cálculos ya que podríamos tener un pequeño grupo de fuentes de luz reales y otro grupo de fuentes de luz emisivas para simular más detalle. Antes de dejar atrás los tipos de iluminación que podemos modelar y enfocarnos en hablar de las fuentes de luz que generan la iluminación, se deben dejar en claro ciertas cuestiones. Las fuentes de luz emiten luz con componentes de iluminación ambiental, emisiva, difusa y especular, es importante saber que las intensidades de cada componente pueden ser iguales o independientes. Esto quiere decir que podemos tener luces que solo afecten a ciertos componentes del modelo de iluminación, lo que puede simplificar los cálculos del modelo de iluminación si se prescinde de alguno de los componentes. Para finalizar veamos ahora como seria el cálculo de intensidad con todos los componentes de las luces y múltiples fuentes de iluminación. [14] [17] [20]

Intensidad-total-amb-emis-dif-esp=

$$\text{RefColor-amb} * \text{Int-amb} + \text{RefColor-emis} + \text{RefColor-dif} * \sum_{i=0}^{n} [\text{Int-dif}(i) * ( ni \cdot li )]$$

$$+ \text{RefColor-esp} * \sum_{i=0}^{n} [\text{Int-esp}(i) * \text{Máximo}( \text{reflex-i.vision-i}, 0)^{ee} * [( \text{normal-i} \cdot \text{luz-i} ) > 0 \ ? \ 1:0 ]]$$

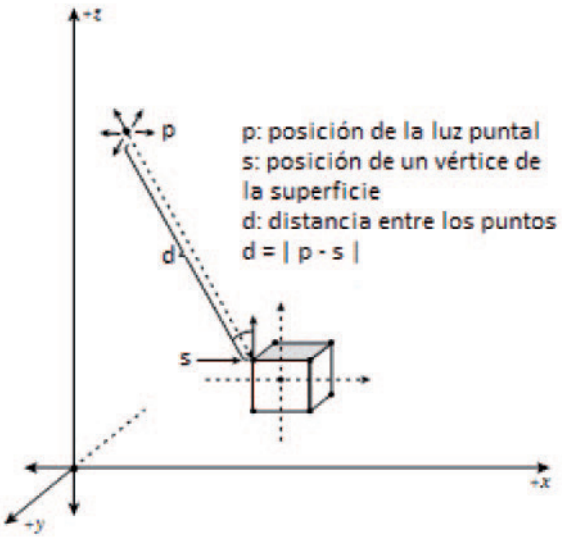
### A2.3 Tipos de fuentes de luz

### A2.3.1 Luz direccional

Habiendo detallado los posibles componentes que conforman a la luz es momento de ver las fuentes que generarán a la misma. El primer tipo de fuente de luz a estudiar es la luz direccional la cual esta infinitamente alejada de la escena por lo que podemos decir que no tiene una posición. Este tipo de luz posee una intensidad de valor constante es decir, la intensidad no es función de la distancia ya que luz se encuentra infinitamente alejada. Definimos una luz direccional con un color e intensidad iniciales, debemos diferenciar el color de las luces de los colores de reflectividad utilizados hasta el momento. Los colores de reflectividad indican el color que refleja el material de una superficie, mientras que ahora estamos definiendo el color de la luz emitida, es decir la que impacta contra la superficie. [14] [17] [20]

Intensidad-dir = Int0-dir * Col-dir

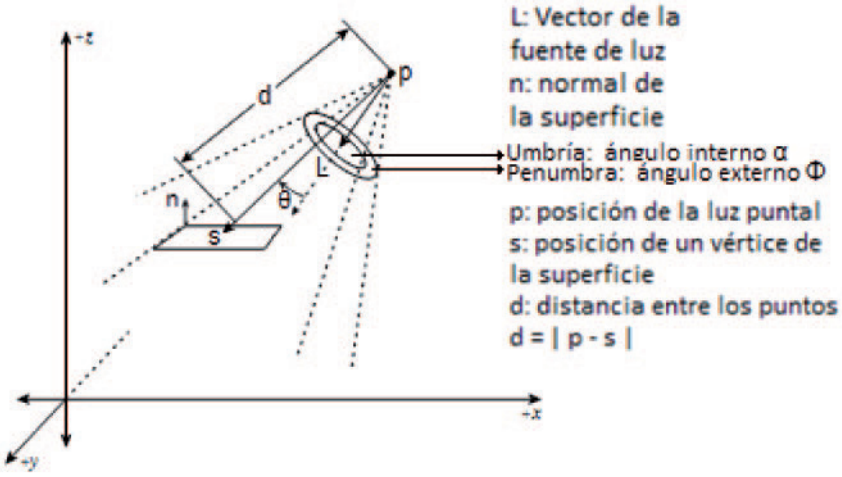
### A2.3.2 Luz puntual




Las luces puntuales se modelan como un punto en el espacio y su intensidad está definida en función de la distancia, a este fenómeno se lo conoce como atenuación. La atenuación normalmente se modela con 3 constantes: el factor de atenuación constante at-const, el factor de atenuación lineal at-lin, el factor de atenuación cuadrático at-cuad. Si tomamos estos factores y llamamos d a la distancia entre la luz posicionada en un punto p y un punto s en la superficie iluminada entonces obtenemos la ecuación para el cálculo de intensidad. Antes de presentar la ecuación eliminaremos el factor cuadrático ya que este no tiene demasiada injerencia en la atenuación. Evitando así cálculos innecesarios para obtener un resultado sigue satisfactorio. [14] [17] [20]

$$\text{Intensidad-punt} = \frac{\text{Int0-punt} * \text{Col-punt}}{\text{at-const} + \text{at-lin} * d} \qquad \text{con} \quad d = |p - s|$$

### A2.3.3 Spotlights




Las spotlights son un tipo de luz computacionalmente exigente por lo cual será necesario realizar una serie de simplificaciones para implementarlas por software, primero estudiemos su modelo matemático para luego lograr derivar una simplificación. Como se observa en la figura la spotlight se define con una

posición, una dirección y un cono que indica el área a ser iluminada en la superficie impactada. Este cono posee 2 regiones, la región interna llamada umbría está definida por el ángulo ά mientras que la región externa, la penumbra, se define con el ángulo Φ.

Dentro de la umbría la intensidad de la luz se puede decir que es constante, mientras que dentro de la penumbra la misma decae rápidamente. Como vemos si quisiéramos implementar este modelo deberíamos calcular el ángulo entre la fuente de luz y la superficie, si el mismo es mayor al cono de iluminación el punto en la superficie no se iluminara. Si el ángulo se encuentra dentro del cono interno se utilizará la intensidad máxima y en caso de que el ángulo se encuentre en la penumbra la intensidad se calculara en base a la función de atenuación basada en la distancia.

Es evidente que la implementación de las spotlights es compleja y por lo tanto se han buscado alternativas donde la relación costo computacional / fidelidad de imagen sean aceptables. En base a estas alternativas, utilizaremos un modelo que se basa en calcular la atenuación desde el centro de la spotlight en base al ángulo θ que existe entre la dirección de la spotlight y la superficie. Eliminando las pruebas condicionales para determinar que cálculo utilizar para la intensidad de la luz. El resultado del producto punto entre la normal de la superficie y la dirección de la luz, nos indica el valor del ángulo θ el cual exponenciaremos por una variable llamada factor de poder o de concentración (fp), que regula la intensidad de la luz. Por lo tanto el modelo simplificado es [14] [17] [20]:

$$\text{Intensidad-spot} = \frac{\text{Int0-spot} * \text{Col-spot} * \text{Max}[ ( n \cdot l), 0 ]^{pf}}{\text{at-const} + \text{at-lin} * d + \text{at-cuad} * d^2} \qquad \text{con} \quad d = |p - s|$$

### A2.4 Sombreado

El sombreado es una tecnica que añade a un modelo la sensacion de profundidad aplicando distintos niveles de oscuridad en el mismo. En la computacion gráfica esto se logra alterando el color de un modelo basado en su posicion y orientacion con respecto a una fuente de luz, con el objetivo de lograr un efecto fotorealista.

#### A2.4.1 Sombreado Constante

La etapa de sombreado estudia como las luces de la escena finalmente afectan a los polígonos, modificando el brillo y color de los mismos. Logrando de esta manera darles un aspecto más realista a los mismos ya que su aspecto variará de acuerdo a la luz que los rodea o la falta de la misma. El sombreado constante se caracteriza por ser el más simple de todos, a decir verdad no toma ningún cálculo de iluminación así como ninguna luz en consideración. Simplemente se generan polígonos sólidos, cuyo color ya está indicado dentro de la propia definición del polígono. Este tipo de sombreado nos permite también simular la iluminación emisiva, en la cual los objetos aparentan emitir luz, pero no lo hacen ya que no tienen una fuente de luz asociada. [14] [17] [20]

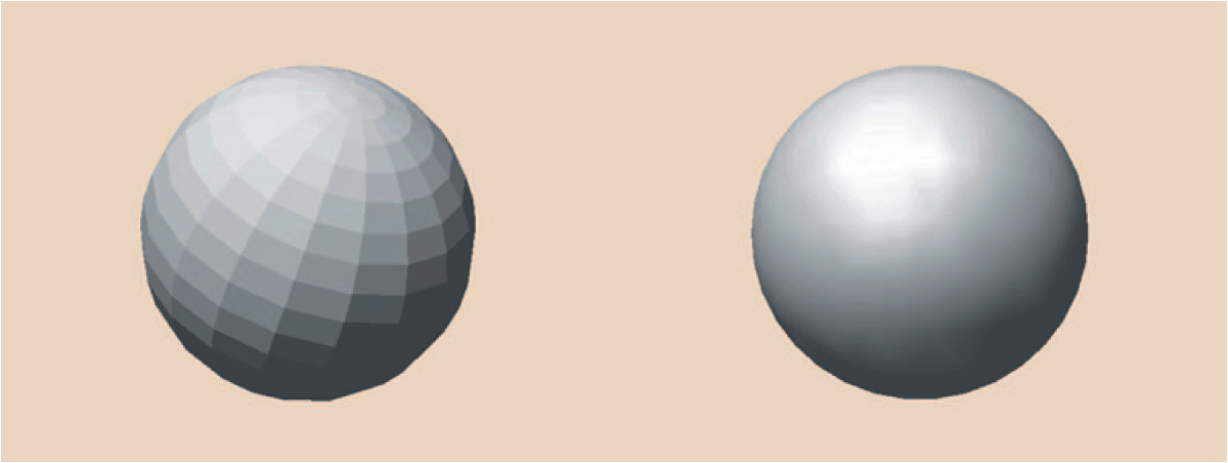
#### A2.4.2 Sombreado Plano/Facetado

Al realizar un motor en software es computacionalmente inviable determinar el color de cada pixel de un polígono, por lo cual se utilizan métodos como el sombreado facetado en donde se calcula la intensidad de color en un solo pixel del triangulo y se utiliza esta intensidad para colorearlo íntegramente. Aplicar este tipo de sombreado da buenos resultados para objetos formados por superficies planas, pero en objetos esféricos se hace evidente que los mismos que están conformados por superficies planas.

Para implementar el sombreado plano se debe calcular la normal del polígono, la cual se utilizará para definir el ángulo entre la superficie y las luces de la escena, determinando así si la luz afecta a la superficie y con cuanta intensidad lo hace. Finalmente se acumulan las intensidades de las luces que afectan al polígono, obteniendo el color final del mismo. [14] [17] [20]

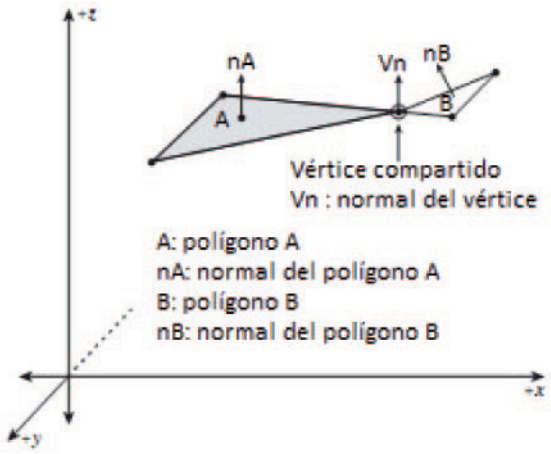
### A2.4.3 Sombreado Gouraud

Como ya se ha visto el sombreado facetado no permite genera una buena representación de como la iluminación afecta a los objetos esféricos, el sombreado Gouraud nos permitirá cumplir este propósito. Para lograr este efecto es necesario que el color que compone al objeto sea aplicado de manera suave y gradual sobre los polígonos del mismo. Se debe calcular el color y la intensidad de la luz en los 3 vértices de cada polígono, cuando varios polígonos comparten vértices, se promedian las normales de estos y se les da un peso acorde al área de cada polígono. Una vez calculados los colores de cada vértice se utiliza a los mismos como los colores entre los cuales interpolar colores intermedios cuando se recorre y dibuja el triangulo. Como se puede notar por lo dicho anteriormente el algoritmo de Gouraud tiene 2 partes. La primera consiste en calcular los colores e intensidades en los vértices cuando se lleva a cabo la etapa de iluminación. La segunda etapa donde se interpolan los colores se da en el rasterizador dado que la interpolación ya está siendo utilizada para dibujar los triángulos por lo que se aprovecha para interpolar no solo de líneas sino también de color.



Una esfera con sombreado facetado y sombreado Gouraud
Fuente:http://www.mactech.com/articles/mactech/Vol.14/14.11/PoorMansBrycePartII/index.html
Fecha de acceso: 11/08/2010

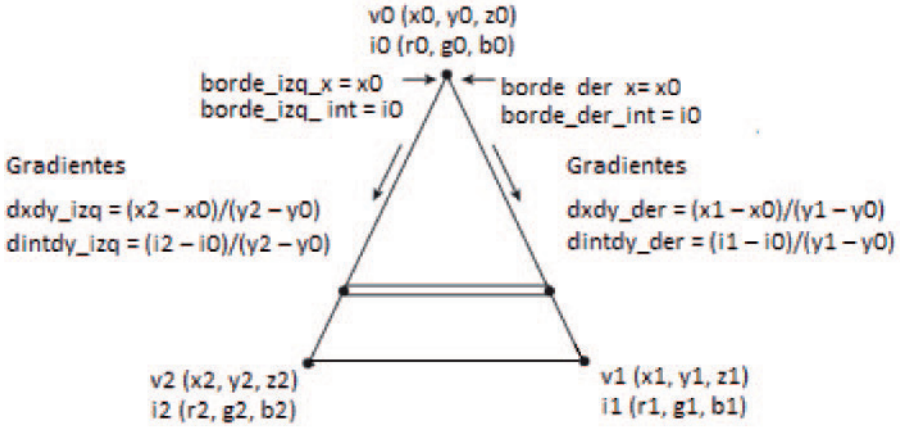
Para llevar a cabo el sombreado Gouraud se comienza con el cálculo de las normales de los vértices de un polígono. La normal de un vértice en términos de iluminación consiste en calcular el promedio de las normales de los polígonos que contienen al vértice. También se debe tener en consideración el área de estos polígonos ya que de acuerdo al área de las mismas la contribución de sus normales variará.




Como se puede observar en el ejemplo el área del polígono A es considerablemente mayor a la del polígono B por lo tanto su contribución debe ser mayor. Cuando se calcula el producto cruz de 2 vectores de un triangulo para calcular la normal del mismo, la longitud del producto cruz resultante es proporcional

al área del triangulo, por lo que la contribución de cada polígono a la normal del los vértices se obtiene fácilmente. Una vez que se poseen las normales de los vértices se procede a calcular la intensidad de la luz y el color de los mismos, de igual manera que se realiza en la iluminación facetada, solo en este caso se deben calcular tres colores por polígono en vez de uno.

La segunda parte de este algoritmo se lleva a cabo en la etapa de rasterización, donde será necesario crear un rasterizador que no solo interpole los bordes de los pixeles que conforman el triangulo y que dibuje las líneas entre extremo y extremo, sino que también debe interpolar colores de vértice a vértice y de línea a línea. Implementar esta funcionalidad es muy similar a la rasterización que ya hemos estudiado, en este caso en vez de calcular el cambio de x relativo al cambio en y, se calculará el cambio en los colores en función de y. A su vez por cada scanline también se interpolara la diferencia entre la intensidad de color del borde izquierdo al derecho de la misma.




Basados en un triangulo de base plana vemos como llevar a cabo la interpolación de colores. Para comenzar se definen las variables que indicaran la posición de los interpolantes en los extremos derecho e izquierdo, tanto para dibujar el triangulo como para calcular las intensidades de color:

borde_izq_x = x0 borde_der_x= x0
borde_izq_ int = i0 borde_der_int = i0

Calcular los gradientes para la rasterización y el coloreado. Es decir, los cambios de x en función de para ambos bordes del triangulo, así como los cambios entre los colores.

dxdy_izq = (x2 – x0)/(y2 – y0)
dxdy_der = (x1 – x0)/(y1 – y0)

dintdy_izq = (i2 – i0)/(y2 – y0)
dintdy_der = (i1 – i0)/(y1 – y0)

Se inicia la etapa de interpolación horizontal, para cada scan line se dibujan los pixeles entre los extremos del triangulo en x. El color de cada pixel es determinado por la interpolación de la intensidad de color entre los extremos. Para lograr esto se asignan los valores iniciales para la posición y la intensidad del borde izquierdo del triangulo. [14] [17] [20]

int = borde_izq_int
x = borde_izq_x

Se calcula el gradiente de la intensidad en función de x.
dintx = (borde_der_ int – borde_izq_ int) / (borde_der_x – borde_izq_x)

Se rasteriza la línea pixel por pixel interpolando el color.
Desde x = borde_izq_x, hasta borde_der_x

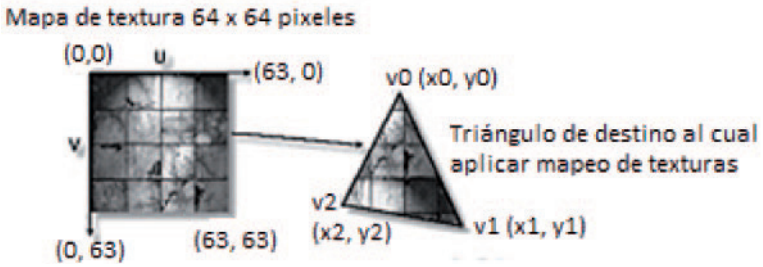
```
Comenzar
        dibujar_pixel(x,y, int)
        int = int + dintx
Fin
```

Finalmente se actualizan los interpolantes verticales para posición e intensidad, y se desciende en 1 la posición y.

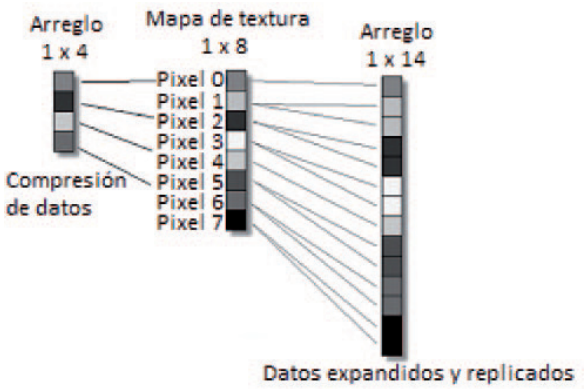
```
borde_izq_x  = borde_izq_x + dxdy_izq        borde_der_x = borde_der_x + dxdy_der
borde_izq_int = borde_izq_ int  + dintdy_izq       borde_der_ int = borde_der_ int + dintdy_der
```

### A2.5 Aplicación de texturas y teoría del muestro

Aplicar una textura a un modelo es un método para darle mayor de nivel de detalle al mismo. El proceso es comparable con forrar una caja con un papel estampado es decir, que se aplica un dibujo/diseño/patrón sobre un modelo para que su superficie aparente tener ciertas propiedades. Por ejemplo, una textura con ladrillos aplicada a un modelo de un rectángulo permitirá simular una pared más convincentemente que el mismo rectángulo si sólo estuviese pintado de rojo. Las texturas suelen estar almacenadas en mapas de bits y, deben tener una lista de coordenadas de texturas asociadas a los vértices del objeto para indicar que parte del mapa de bits corresponde a que parte del objeto. Implementar el mapeo de texturas implica reutilizar conocimientos ya adquiridos, por ejemplo a la hora de rasterizar el proceso es muy similar al sombreado Gouraud, la diferencia radica en que se interpolarán coordenadas de textura. Durante la interpolación del la textura, se usan los pixeles que la componen para definir el color de cada pixel del triangulo a dibujar. En general los mapas de textura son mapas de bits rectangulares de m x m pixeles con color en 8 o 16 bits y se quiere utilizar el mismo para dar textura a un triangulo que puede tener cualquier coordenada, por lo tanto es necesario tener un modo de considerar la rotación y el escalado al que puede ser sometido un triángulo.




En la imagen vemos el mapa de texturas con sus ejes u (horizontal) y v (vertical), un rango de (0,0) a (63,63) y el triangulo de destino con vértices v0, v1, v2. Los mapas de texturas suelen tener un rango 0 a 1 si son normalizados, o de 0 a cualquier potencia de 2. Observemos un ejemplo simple de cómo realizar la interpolación con el fin muestrear datos.



En este caso se tiene una textura de 1 x 8 pixeles la cual, debe ser mapeada a un polígono que también tiene 1 pixel de ancho pero puede tener cualquier numero de pixeles de alto. Si las dimensiones del polígono son 1 x 4 se toman los pixeles 0,2,4,6 de la textura y se los aplica a las posiciones disponibles del polígono. Para decidir que pixeles utilizar de la textura se utiliza la relación de muestreo (sampling ratio) que es simplemente un factor de interpolación.

rel_muestreo = altura_origen / altura_destino
En nuestro caso: rel_muestreo = 8/4 = 2.0

Esta relación nos indica que es necesario que por cada pixel del destino se mueva 2 pixeles la posición del origen, a esta técnica se la conoce como muestreo puntual . Si se evalúa un caso opuesto por ejemplo, el destino es 1 x 1. El factor de muestro será un numero decimal 8/14 = 0,57. En este caso los puntos de muestreo serán: 0,57 1,14 1,71 2,28 2,85 3,42 3,99 4,56 5,13 5,7 6,27 6,84 7,41 7,98. Trucando la parte decimal se obtiene el punto de muestreo final: 0, 1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6, 7, 7.

Como se ve el muestreo es otra forma de interpretar la interpolación, basada en utilizar información almacenada para obtener los colores del polígono a dibujar. La implementación de la interpolación de u & v en la etapa de rasterización es, en un 95% igual a la implementación de Gouraud. En este caso en lugar de calcular gradientes de intensidad de color, se calculan los gradientes y posiciones de u y v. La mayor diferencia se encuentra en el loop final, el que recorre el scanline. Donde de acuerdo a los valores de u y v interpolados, se accederá a la textura, almacenada en una matriz, y se utilizará el color del pixel en esa posición del mapa como el color para el pixel del triangulo a dibujar. [2][14]

```
Desde x = borde_izq_x hasta borde_der_x
Comenzar
        color_pixel = mapa_textura[ui][vi]
        dibujar_pixel( x,y, color_pixel)
        ui += du;
        vi += dv;
Fin
```

# Anexo 3 - Morfología de imágenes

La morfología de imágenes consiste en una serie de métodos para modificar imágenes a fin de eliminar ruido, aislar elementos individuales, unir elementos dispares, etc. La morfología también puede ser utilizada para encontrar picos o pozos de intensidad, así como los gradientes en imágenes. Las dos operaciones más básicas son llamadas dilatación y erosión.

La dilatación consiste en la convolución de una imagen A con un centro (kernel) B que puede tener cualquier tamaño o forma y un único punto de anclaje. El kernel puede considerarse como una máscara o un patrón, su efecto en la dilación es la de encontrar un máximo local, lo que hace que se incrementen las regiones brillantes en una imagen. La erosión es la operación contraria, encuentra el mínimo local en el área del kernel. En general la dilación expande y suaviza las concavidades de la imagen mientras que la erosión la reduce y suaviza las salientes.

La utilidad de la erosión radica en eliminar el tipo de ruido que se asemeja una mancha o punto, el principio es que este ruido es tan pequeño que se elimina mientras las regiones que tienen contenido visualmente significativo permanecen inalteradas. La dilatación en cambio sirve para encontrar componentes conectados, grandes regiones discretas que tienen pixeles de colores similares, mediante este proceso se evita que grandes regiones sean divididas en múltiples componentes debido a la presencia de ruido, sombras u otros efectos.

La morfología de imágenes suele ser suficiente para imágenes booleanas y mascaras, pero al trabajar con imágenes en escala de grises o color, existen una serie de operaciones que son de utilidad. Las operaciones de apertura y clausura son combinaciones de las operaciones de erosión y dilatación. La apertura consiste en realizar primero la erosión y luego la dilatación, es usada para separar imágenes en regiones las cuales tienen elementos en común, y así facilitar el conteo de estas regiones.

La clausura se utiliza en la mayoría de los algoritmos sofisticados para encontrar componentes que se encuentran interconectados. En general para encontrar los componentes se utiliza primero la clausura para eliminar los elementos surgidos del ruido y, luego se realiza una apertura para conectar regiones. Finalmente podríamos decir que el efecto de la apertura es encontrar y eliminar valores atípicos solitarios que son mayores a los valores vecinos. Mientras que el efecto de la clausura es eliminar los valores atípicos solitarios de valor menor al de sus vecinos. [21]

# Anexo 4 - Ejemplo: Main

A continuación veremos un extracto del código fuente del main de una aplicación que utiliza el motor 3D desarrollado con el fin de ver un poco del comportamiento del mismo, y como para el desarrollador el uso del mismo se asemeja al uso de un API. No se necesita tener mayor conocimiento de lo cómo funciona el motor para poder utilizarlo para crear aplicaciones que lo utilicen:

```
reiniciar_obj(&cubo);
cubo.tpos_glob.x = -50;
cubo.tpos_glob.y = 20;
cubo.tpos_glob.z = 120;
escalar_obj_3f(&cubo, 1.0, 1.0, 1.0);
rotar_obj(&cubo, x_ang, y_ang, z_ang);


reiniciar_obj(&cubo1);
cubo1.tpos_glob.x = 0;
cubo1.tpos_glob.y = 20;
cubo1.tpos_glob.z = 120;
escalar_obj_3f(&cubo1, 1.0, 1.0, 1.0);
rotar_obj(&cubo1, x_ang, y_ang, z_ang);

calcucular_camara_euler(cam, CAM_ZXY);

crear_mat_rot_obj(cubo.ang_rot_x, cubo.ang_rot_y, cubo.ang_rot_z, &cubo.mrot);
crear_mat_rot_obj(cubo1.ang_rot_x, cubo1.ang_rot_y, cubo1.ang_rot_z, &cubo1.mrot);

transformar_objetos(&cubo, cam, ILUM_NORMAL, CLIPPING_SIMPLE);
transformar_objetos(&cubo1, cam, ILUM_NORMAL, CLIPPING_SIMPLE);

objs[0] = cubo;
objs[1] = cubo1;

inicializar_renderlist();


if (renderlist != NULL)
{
        sort(renderlist);
        if (!(DDraw_Lock_Back_Surface()))
        return  (0);
        rasterizer_wrapper(back_buffer, back_lpitch);
        DDraw_Unlock_Back_Surface();
}
```

El extracto comienza con 2 objetos llamados cubo y cubo1 para los cuales se definen su posición en pantalla, el tamaño  y los ángulos de rotación de los mismos. La función reiniciar_obj() vuelve reinicia el estado de los objetos y los polígonos, el cual puede cambiar a lo largo del procesamiento del mismo. A continuación se llama a la función que calcula la matriz de transformación para la cámara, una vez que los objetos están en espacio del mundo los mismos son multiplicados por esta matriz para ser trasladados al espacio de la cámara, los parámetros de la función son la cámara para la cual realizar el cálculo y el orden de rotación de los ejes. Crear_mat_rot_obj() crea la matriz de rotación para hacer que un objeto rote en espacio objeto o mundo, recibe los ángulos a utilizar para crear la matriz y la propia matriz donde almacenar los datos.

Transformar_objetos() es la función que procesa el objeto transformando al mismo desde que se encuentra en espacio objeto hasta  sus coordenadas finales en pantalla, podríamos decir que es la función más importante del programa ya que, se encarga de transformar al objeto de espacio a espacio, eliminar

polígonos no visibles, e iluminarlo. Todo esto se realiza mediante llamadas a funciones internas, pero la idea era que este proceso sea transparente al usuario, el cual solo debe decir que objeto se quiere procesar y provee algunos flags para especificar cuáles funciones internas utilizar. Una vez transformados los objetos se guardan en el arreglo objs[], ya que esta será la estructura que se procesara a la hora de crear la renderlist a través de la función inicializar_renderlist(), luego si la renderlist no está vacía la misma se ordena de mayor a menor en base a los valores de z de los polígonos. Finalmente se realiza una llamada a DirectX para alistar la zona de memoria que corresponde a la “superficie” donde se dibujara la imagen a mostrar en pantalla y se llama a la función rasterizer_wrapper que procesa la renderlist y de acuerdo a las características de cada polígono llama a la función de rasterizing apropiada, otra vez este proceso es transparente al usuario.

# Anexo 5 - Datos de motores Comerciales 3D y Juegos Profesionales

Como último paso antes de realizar un planteo sobre los objetivos y las conclusiones obtenidas del marco teórico, veremos precios de algunos motores 3D comerciales de la pasada generación de videojuegos, es difícil obtener datos de los motores actuales ya que la mayoría de los precios se encuentran bajo convenios de no divulgación [9]. También se muestran datos de juegos comerciales como son: cantidad de líneas de código, cantidad de archivos generados, cantidad de programadores involucrados, etc. Esto nos da un marco de referencia sobre la envergadura y costos de producción de juegos y motores profesionales.

**A5.1 Motores [9]:**

| | |
|---|---|
| Nombre del motor | Unreal Engine 2 |
| Nombre de la empresa | Epic Games |
| Página web | http://www.unrealtechnology.com/ |
| Plataformas | Windows, Linux, MacOS, Xbox, Playstation, GameCube |
| API Gráfico | OpenGL, DirectX, Software |
| Licencia regalía libre | $750 000 para cualquiera de las plataformas disponibles, y $100 000 por cada plataforma adicional. |
| Licencia regalía condicional | $350 000 para cualquiera de las plataformas disponibles, y $50 000 por cada plataforma adicional. Más una regalía del 3% de las ventas del producto. |

| | |
|---|---|
| Nombre del motor | Auran Jet |
| Nombre de la empresa | Auran Pty Ltd |
| Página web | http://www.auran.com/jet/ |
| Plataformas | Windows |
| API Gráfico | OpenGL, DirectX |
| Licencia no comercial | $99 |
| Licencia comercial | $30 000 |

| | |
|---|---|
| Nombre del motor | Gamebryo Engine |
| Nombre de la empresa | Numerical Design Limited |
| Página web | http://www.ndl.com/gamebryo-engine.cfm |
| Plataformas | Windows, Xbox, Playstation, GameCube |
| API Gráfico | OpenGL, DirectX |
| Licencia | $50 000 por título, por plataforma |

| | |
|---|---|
| Nombre del motor | Eternity |
| Nombre de la empresa | TS Group Entertainment LLC. |
| Página web | http://www.tsgroup-inc.com/Eternity/index.htm |
| Plataformas | Windows |
| API Gráfico | DirectX, Glide |
| Licencia (1 producto) | $40 000 (Upgrades solo para los primeros 18 meses, 5 días de entrenamiento y 6 meses de soporte) |
| Licencia (productos ilimitados) | $99 000 (Upgrades ilimitados, 14 días de entrenamiento y 18 meses de soporte) |

**A5.2 Juegos [9]:**

| | |
|---|---|
| Nombre | Prince of Persia: The sands of time |
| Editor | Ubisoft Montreal |
| Número de desarroladores | 65 sin contra testeadores |
| Tiempo de desarrollo | 27 meses |
| Hardware de desarrollo | Dual AMD Athlon 2000. 1 GB RAM. Windows 2000. |
| Software de desarrollo | Microsoft Visual .NET 2003. Metrowerks CodeWarrior. PlayStation 2 Tuner. Incredibuild |
| Fecha de lanzamiento | Noviembre 2003 |
| Plataformas | PC, Xbox, GameCube, PlayStation 2, Game Boy Advance |
| Tamaño del proyecto | 4 188 archivos y 1 263 580 líneas de código |
| Total de errores | 14 613 |

Game Developer Magazine – Abril 2004 – Pág 46.

| | |
|---|---|
| Nombre | Ratchet & Clank |
| Editor | Sony |
| Número de desarroladores | 65 (mas dos contratistas) |
| Tiempo de desarrollo | 18 meses |
| Fecha de lanzamiento | Noviembre 2004 |
| Plataformas | PlayStation 2 |
| Tamaño del proyecto | 2 millones de líneas de código |

Game Developer Magazine – Febrero 2005 – Pág 29

| | |
|---|---|
| Nombre | Project Gotham Racing 2 |
| Editor | Microsoft Game Studios |
| Número de desarroladores | 40 |
| Tiempo de desarrollo | 2 años |
| Hardware de desarrollo | Pentium 600Hz-2.4GHz, 256-1024 RAM, GeForce 2-4 y ATI Radeon 9700 Pro |
| Software de desarrollo | Microsoft Visual Studio .NET, Microsoft SourceSafe, Alienbrain, Build-in 3D editor, Softimage XSI, Araxis Merge 2001, SoundForge |
| Fecha de lanzamiento | Noviembre 18 de 2003 |
| Plataformas | Xbox |
| Tamaño del proyecto | 37 174 archivos, 219 538 líneas de código, 41GB de data |

Game Developer Magazine – Marzo 2004 – Pág 47