Título Realtime Voxel Rendering

Tipo de Producto Material didáctico

Autores Banquiero, Mariano y Ortiz, Edgar

Código del Proyecto y Título del Proyecto

A16T11 - Realtime Voxel Rendering

Responsable del Proyecto

Banquiero, Mariano

Línea

Automatización y Procesamiento de Imágenes

Área Temática

Realtime Rendering- Computer Graphics

Fecha

2016

INTEC
Instituto de Tecnología

UADE FUNDACIÓN UADE

Realtime Vóxel Rendering

Parte A

Conceptos Básicos de 3D

Introducción a los gráficos por computadora

El Universo 3D

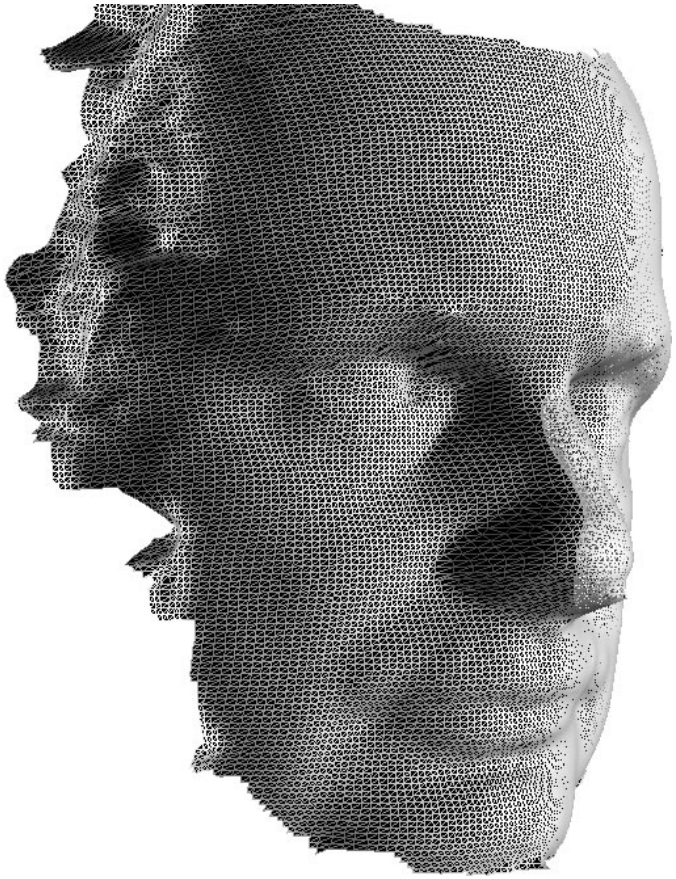


Ilustración 1

Para encarar la programación de gráficos en tres dimensiones utilizamos un modelo lógico, compuesto por un espacio geométrico. Este espacio geométrico contiene y representa todos los objetos que queremos terminar mostrando en pantalla.

Está compuesto por tres dimensiones: ancho, alto y profundidad, que se representan por medio de un sistema de tres ejes cartesianos: X, Y, Z.

Para representar un punto en este espacio geométrico necesitamos tres valores: un valor para X, un valor para Y, y un valor para Z.

Existen diversos tipos de sistemas de ejes. Los más utilizados son los siguientes:

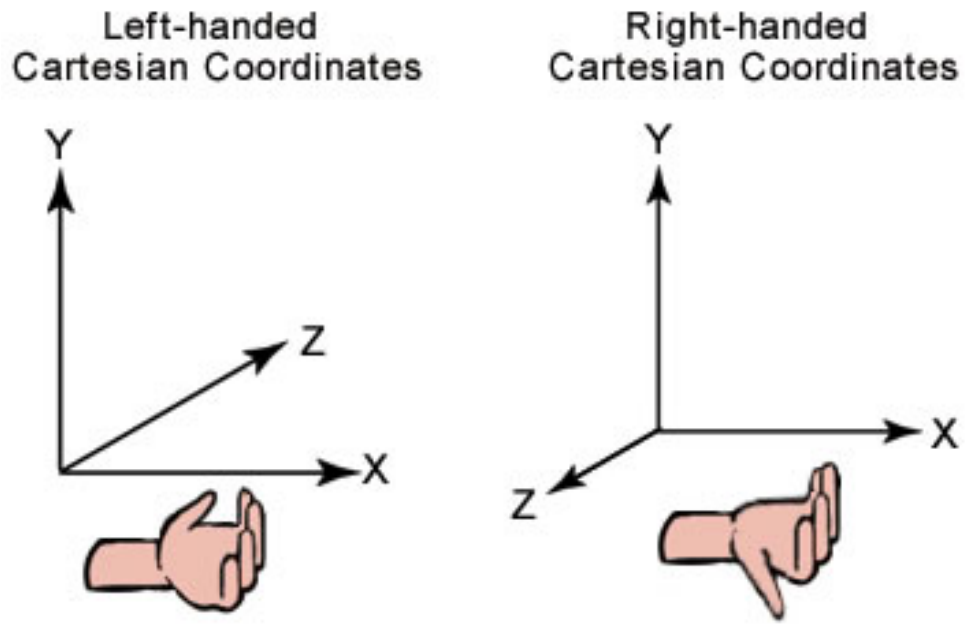


Ilustración 2

Ambos sistemas son similares, excepto por la dirección del eje Z.

Existen distintas formas o modelos lógicos de representar, computar y mostrar un universo en tres dimensiones en una computadora. Algunos ejemplos son:

- Modelo de Silicon Graphics
- Voxel Rendering
- Reyes rendering
- Nurbs

A continuación haremos un breve resumen del modelo de Silicon Graphics (SGi), puesto que es el más difundido, debido a que tiene soporte de hardware (GPU).

Primitivas

En el modelo de Silicon Graphics existe el concepto de primitiva. Una primitiva es la unidad elemental mediante la cual se pueden construir todos los elementos del universo 3D.

La unidad fundamental de este modelo es el triángulo.

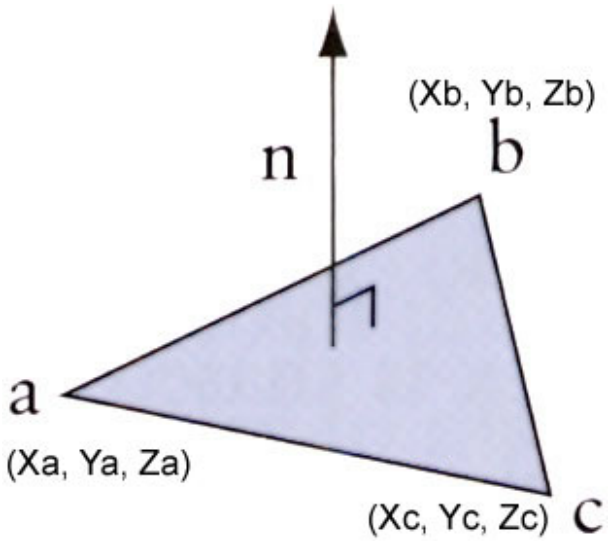


Ilustración 3

El triángulo es la unidad básica. Un triángulo está compuesto por 3 vértices: A, B y C.

Cada vértice especifica la posición de un punto en el espacio y, como se mencionó anteriormente, se necesitan tres valores para especificar una posición en un universo 3D (x, y, z).

El estándar es utilizar valores en punto flotante (float) para especificar el valor de una coordenada. Por lo tanto la estructura de datos para un triángulo podría ser la siguiente:

```
class Triangle
{
        Vertex a;
        Vertex b;
        Vertex c;
}

class Vertex
{
        float x;
        float y;
        float z;
}
```

Un triángulo está conformado por tres vértices, por lo tanto necesita nueve floats para poder ser representado correctamente. Por ejemplo:

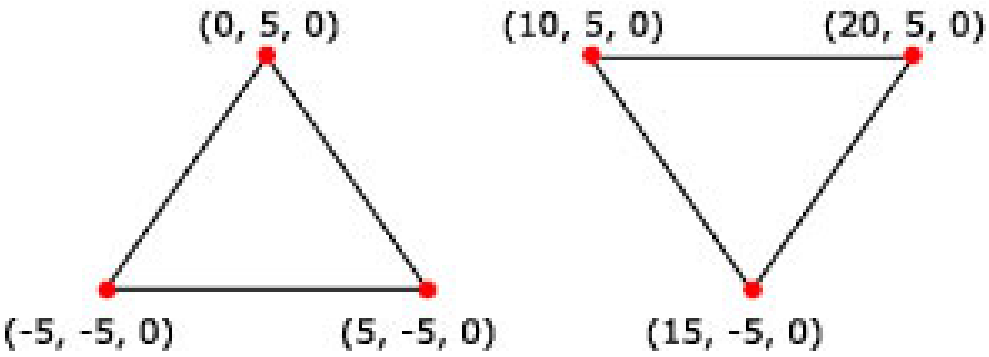


Ilustración 4

El triángulo es utilizado como la primitiva fundamental dado que es el cuerpo en tres dimensiones más simple de representar. Además un triángulo es coplanar, lo cual significa que todos sus puntos interiores (incluidos los vértices) se encuentran contenidos dentro de un único plano. Este permite muchas optimizaciones de implementación a la hora de dibujar triángulos en pantalla. Al ser la primitiva fundamental, a partir de él se puede crear cualquier otra figura. Por ejemplo, para confeccionar un rectángulo plano en 3D, es necesario especificar 2 triángulos:

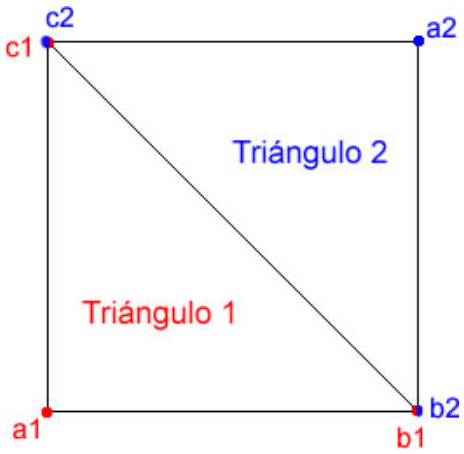


Ilustración 5

Cada uno de estos triángulos posee 3 vértices, por lo tanto la cantidad de vértices necesarios para especificar la figura son: 2 x 3 = 6 vértices, con un total de 18 valores floats para representarlo: 2 x 3 x 3 = 18 valores floats. Los vértices c1 y c2, al igual que b1 y b2 son iguales y, en su forma más sencilla de representación, se repiten al especificar la estructura geométrica del cuerpo.

Para confeccionar un cubo se necesitan 6 caras rectangulares, con 2 triángulos en cada cara, lo que nos da un total de 12 triángulos, 36 vértices y 108 valores floats:

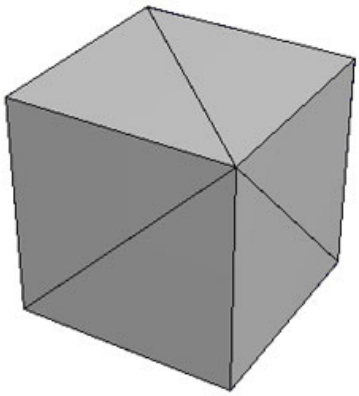


Ilustración 6

Existen otras primitivas pero en este apunte solo se tratarán los triángulos simples.

Básicamente cualquier figura en tres dimensiones puede ser modelada con triángulos. Algunos modelos requerirán poca cantidad de triángulos para poseer una buena apariencia y otros necesitarán una gran cantidad, en especial las superficies curvas:

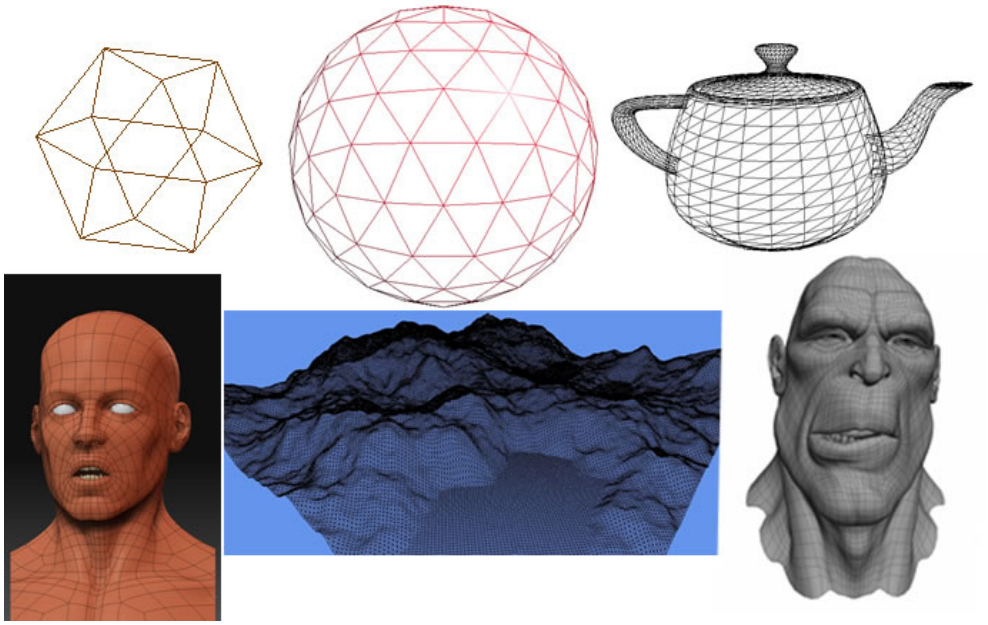


Ilustración 7

Hay formas geométricas simples como esferas, cubos y cilindros cuyas coordenadas pueden ser especificadas en forma programática desde una aplicación, pero otras formas requerirán de programas de diseño complejos para poder construirlas.

Una figura geométrica compuesta por triángulos se denomina *malla* (*mesh*). Un escenario en tres dimensiones estará compuesto básicamente por un conjunto de mallas las cuales, a su vez, estarán compuestas por un conjunto de triángulos.

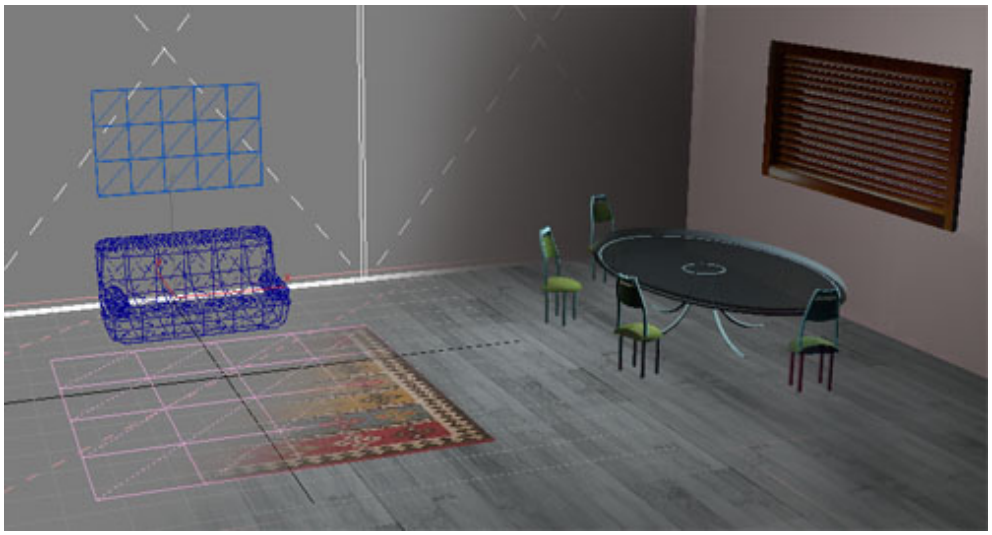


Ilustración 8

Elementos constitutivos de una Malla

Una malla en un universo 3D estará tendrá un esquema parecido al siguiente:

Malla:

- 1 Textura
- N triángulos
    - 1 normal (face-normal)
    - 3 vértices
        - 3 valores de Posición (por vértice)
            - X
            - Y
            - Z
        - 2 coordenadas de textura (por vértice)
            - U
            - V
        - 3 valores de vector normal (por vértice)
            - Nx
            - Ny

        - Nz
    - 4 valores de color (por vértice)
        - R
        - G
        - B
        - A

Esto es solo un esquema general. Algunas mallas podrán tener más elementos mientras que otros podrán obviar algunas, dependiendo de la aplicación en particular.

La estructura de datos para el esquema anterior podría ser la siguiente:

```
class Mesh
{
        Triangle[] triangles;
        int textureID;
}

class Triangle
{
        Vertex a;
        Vertex b;
        Vertex c;

        //direccion del vector normal del triangulo
        float nx;
        float ny;
        float nz;
}

class Vertex
{
        //posicion
        float x;
        float y;
        float z;

        //coordenadas de textura
        float u;
```

```
        float v;

        //color
        float r;
        float g;
        float b;
        float a;

        //direccion del vector normal del vertice
        float nx;
        float ny;
        float nz;
}
```

A continuación se explica brevemente cada elemento de una malla.

- Mesh: es el modelo 3D que queremos representar en el universo. Posee una lista de triángulos que definen su geometría. Además puede poseer una o más texturas asociadas (la estructura anterior muestra el caso para una única textura).
- Textura: es una imagen 2D que puede proyectarse sobre un triángulo para lograr un efecto de superficie, rugosidad, luminosidad, etc. Cada triángulo se mapea a una porción de la textura mediante las coordenadas de texturas de sus vértices.
- Triángulo: cada triángulo de una malla estará compuesto por 3 vértices. Además, un triángulo puede contar con un vector normal que indique la dirección de la cara, el cual se denomina “face-normal”. Como todo vector, estará compuesto por tres valores que especifican su dirección respecto de los tres ejes cartesianos (Nx, Ny, Nz).

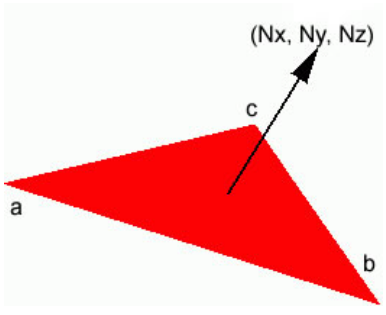


Ilustración 9

- Vértice: un triángulo, como se vio antes, posee su posición (X, Y, Z) que lo ubica en el espacio 3D. Además de esta posición puede poseer otros valores de utilidad.

- Coordenadas de textura: se especifican 2 por cada vértice de cada triángulo. Se expresan como un par ordenado (U, V). Representan una proyección del triángulo en una textura. Estas coordenadas permiten mapear una sección de una imagen 2D sobre la superficie del triángulo. Cada valor de una coordenada de textura puede tomar valores entre 0 y 1.

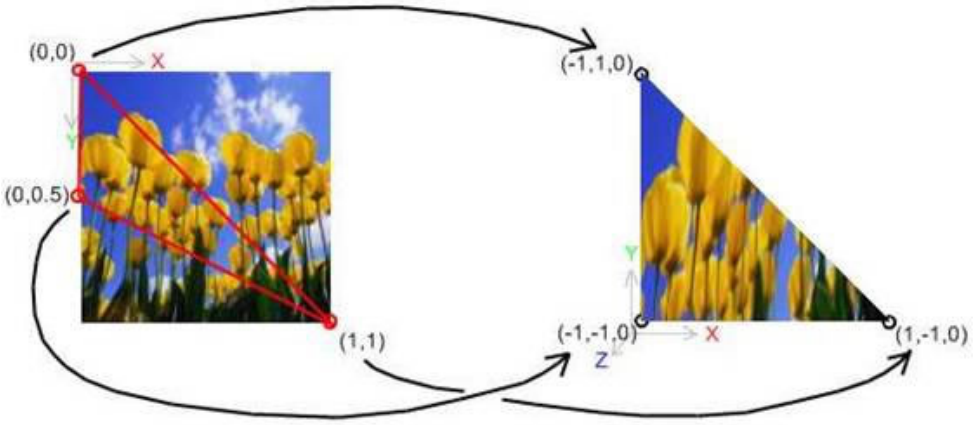


Ilustración 10

- Color: cara vértice puede especificar un color RGB o RGBA. Este color será utilizado para calcular la iluminación propia del triángulo. Esta iluminación es independiente de la textura que esté utilizando. Dos triángulos podrían estar mapeados a la misma textura, en la misma sección, pero uno estar iluminado en forma más brillante que el otro. Los adaptadores de video tomarán los colores de cada vértice y generarán colores interpolados para los pixeles de pantalla que se encuentren en el medio de estos colores. Cada coordenada de un color RGB puede tomar valores entre 0 y 1.

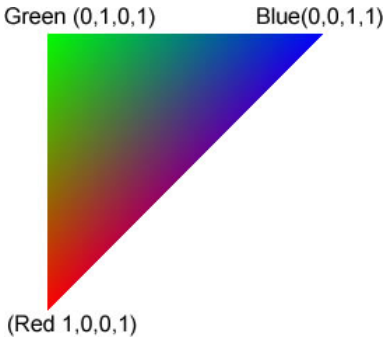


Ilustración 11

- Vector normal: cada vértice posee un vector normal que indica su orientación en el universo. A simple vista podría parecer innecesario que un vértice tenga una orientación, dado que se considera un punto en el espacio sin volumen. Pero estos vectores serán utilizados para calcular la iluminación correcta de un triángulo.

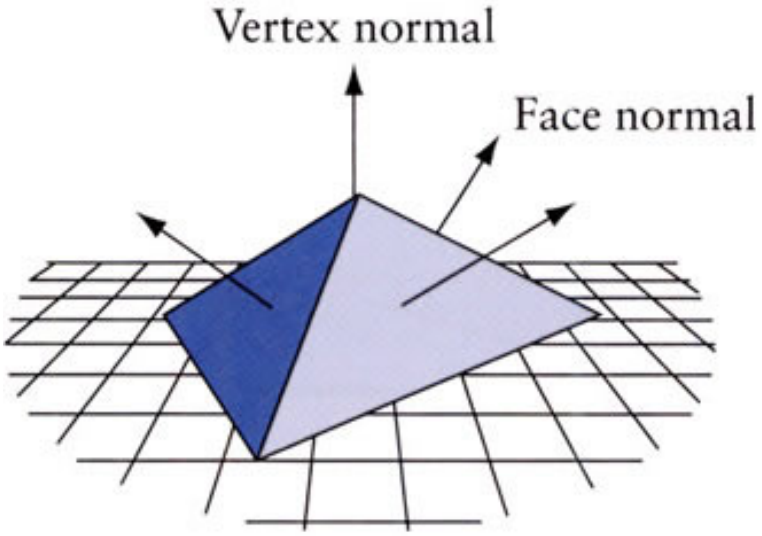


Ilustración 12

Rendering

Hasta ahora hemos definido como crear un universo 3D especificando las primitivas gráficas que definen cada malla. Pero este universo es simplemente un modelo lógico que existe en la memoria de nuestra aplicación. Para poder visualizar ese resultado, el modelo lógico deberá ser transformado a un modelo físico que sea desplegarse en una pantalla. Aquí es donde surge la principal diferencia: queremos representar un universo 3D pero las pantallas de los dispositivos gráficos solo trabajan con dos dimensiones:

Ilustración 13

Para lograr hacer visible nuestro universo 3D, el modelo lógico deberá ser proyectado a un modelo de dos dimensiones, compuesto por *píxels*. Este proceso se denomina *Rendering*.

En términos generales, el *rendering* consiste en generar una imagen de píxels a partir de un modelo. El modelo es una descripción tridimensional del objeto definida en una estructura de datos. Esta descripción, como se vio antes, contiene definiciones de geometría, texturas e

iluminación. Al finalizar el proceso de *Rendering*, se obtiene una imagen digital en dos dimensiones; una imagen de tipo *Raster*.

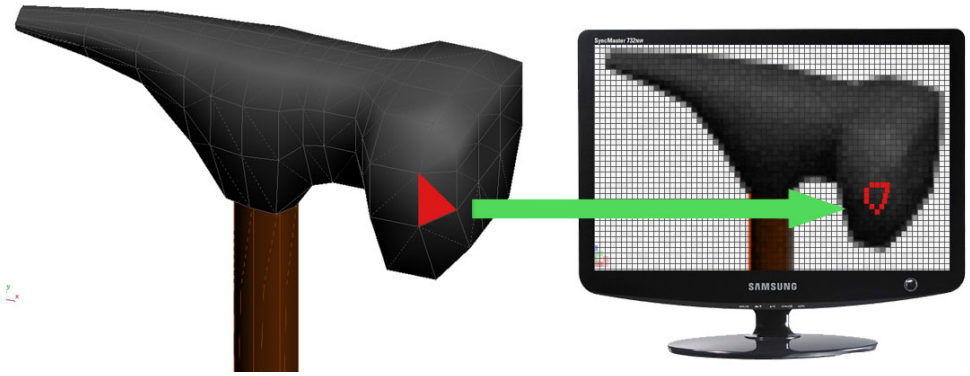


Ilustración 14

El universo 3D entero no es completamente observable en una pantalla 2D. Sino que solo una parte del mismo será visible, en un determinado momento, bajo una determinada posición y orientación de cámara.

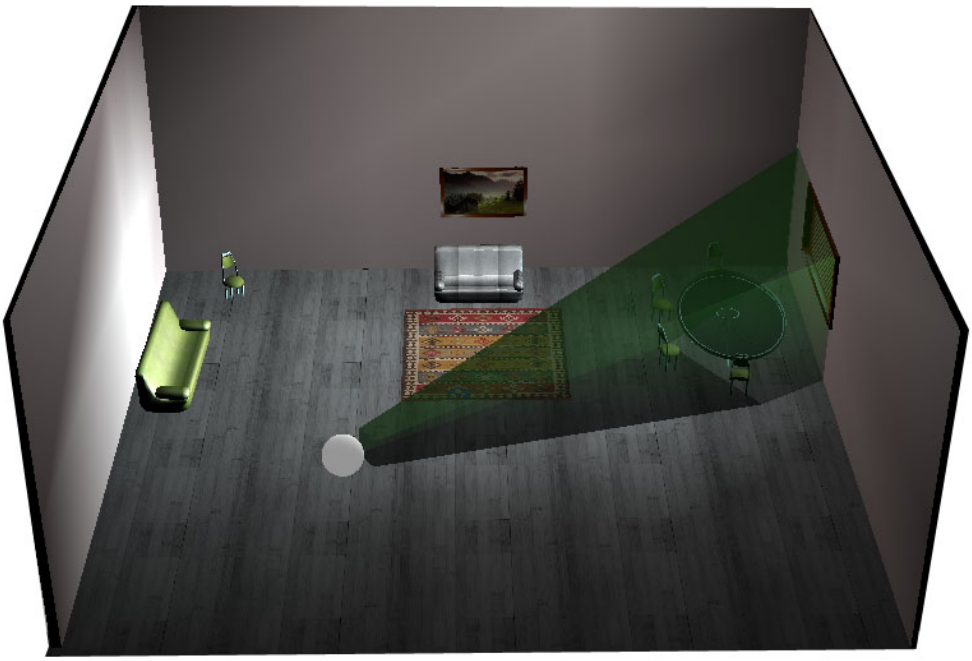


Ilustración 15

La cámara mencionada anteriormente representa el ojo desde el cual se está observando el universo 3D en un determinado momento, y se la denomina *Frustum*. El frustum es una figura geométrica de aspecto piramidal, que delimita la región de espacio del universo 3D que terminará siendo visible en pantalla.

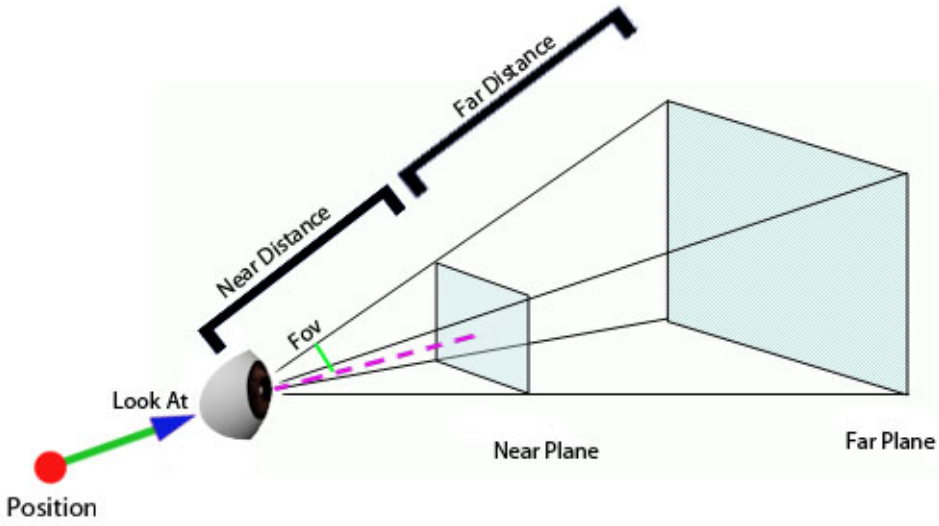


Ilustración 16

Consiste en una pirámide rectangular compuesto por seis planos: near plane, far plane, left plane, right plane, top plane y bottom plane. El volumen encerrado entre estos planos dependerá de los valores configurados de la cámara:

- Position: lugar en el espacio en donde se encuentra la cámara, en un momento determinado
- Look at: dirección hacia donde apunta la cámara en un momento determinado, partiendo desde la posición especificada anteriormente.
- Near distance: distancia que existe entre la posición de la cámara y el near plane del Frustum. Indica la distancia a partir de lo cual todo empieza a ser visible. Aquello que se encuentre entre la cámara y el near plane, es decir, demasiado cerca al ojo de la cámara, no será visible.
- Far distance: distancia máxima a la que un objeto será visible. Determina la posición del far plane del Frustum. Es análogo a la visibilidad máxima que puede ver el ojo.
- Aspect Ratio: es relación entre el ancho y el alto de la pantalla 2D sobre la que se va a proyectar el mundo 3D. Se calcula dividiendo el ancho de la pantalla por el alto de la misma. Ratio = width / height
- FOV: indica el ángulo de visión respecto del eje Y. Es lo que determina el ensanchamiento de la pirámide del Frustum, desde el near plane al far plane. Normalmente toma un valor entre 40° y 60°.

Todos los triángulos de todos los modelos del universo 3D que se encuentren dentro del volumen del frustum serán visibles en pantalla (exceptuando aquellos que se interponen ante otros). Aquellos triángulos que se encuentran parte dentro del frustum y parte afuera, serán recortados hasta dejar solo la parte de los mismos que es visible (triangle splitting). A este proceso de determinar que triángulos caen dentro del volumen de visión se lo denomina

Frustum Culling, y puede ser realizado en distintas etapas: a nivel de la aplicación o directamente por el adaptador de video.

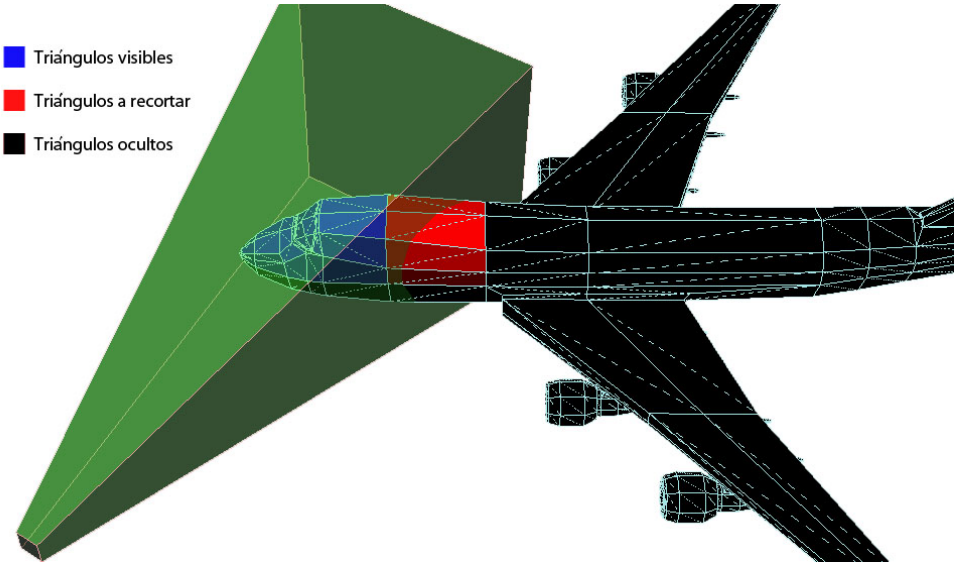


Ilustración 17

Una vez que se ha calculado los triángulos que se encuentran dentro del frustum, habrá que tener en cuenta que algunos estarán tapando a otros, en lo que respecta a al ojo de la cámara. Por lo tanto no todos los triángulos dentro del frustum terminarán convirtiéndose realmente en píxels de pantalla. A este proceso se lo denomina *Occlusion Culling*.

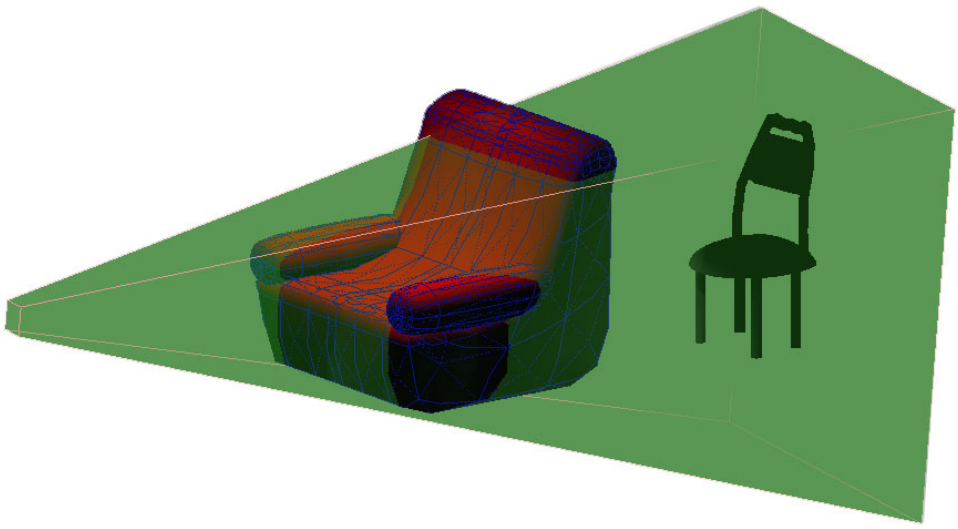


Ilustración 18

Por lo tanto el concepto de *Rendering* consiste en calcular, para una cámara determinada, dentro de un universo 3D determinado, que pixeles deberán ser dibujados en pantalla. El

proceso de rendering es bastante complejo e involucra varias etapas. Gracias a la existencia de APIs gráficas como *OpenGL* y *DirectX*, junto con el apoyo de los adaptadores de video, el programador normalmente no debe ocuparse de todo este proceso de transformación. Sino que se limita a especificar la geometría del mundo que quiere representar en el modelo lógico de 3D dimensiones, y luego la API gráfica en cuestión será la encargada de transformar ese mundo 3D en una matriz bidimensional de pixeles.

Ilustración 19

Renderizado RealTime vs Offline

Antes de continuar explicando el proceso de rendering, es necesario recalcar los dos tipos de situaciones en los que se puede estar ejecutando este proceso: *Offline* y *RealTime*.

En el renderizado Offline, se construye un universo 3D y a partir de ese universo se obtienen imágenes (*renders*) bajo determinadas posiciones de cámara. Pero lo importante de esta situación es obtener la mayor calidad posible de renders, sin importar demasiado el tiempo empleado para ello. Este concepto es utilizado para fotografía, edición digital de imágenes y cine de animación.

Por ejemplo, en una película de animación 3D, se necesitan 25 cuadros por segundo para mantener una buena continuidad de movimiento. Esto significa que son necesarios 25 renders por segundo. Para confeccionar una película entera de aproximadamente dos horas, son necesarios una cantidad importante de renders.

Ilustración 20

La cantidad puede parecer abismal, pero tiene una ventaja, es una cantidad fija y, una vez obtenidos todas las imágenes para cada cuadro, la película se compagina y se reproduce sin problemas. Computar cada imagen es bastante costoso en términos de potencia computacional, pero una vez que se tienen todas, reproducir la película solo se limita a mostrar la imagen necesaria para cada cuadro, con lo cual la reproducción no requiere gran potencia de cálculo (descartando los formatos de compresión de video como MPEG4, divX, etc).

La primera película largometraje de animación 3D fue ToyStory de Pixar en 1995. Los siguientes números cuantifican en cierta medida el esfuerzo de desarrollo empleado:

- 110 animadores para modelar todos los personajes y escenas en 3D.
- 300 procesadores fueron utilizados para renderizar toda la película.
- 114.200 renders, con un promedio de cálculo de entre 2 a 15 horas para obtener cada uno
- 800.000 horas fueron requeridas para computar todos los renders de todos los cuadros de la película, junto con la adición de efectos de sonido y música.

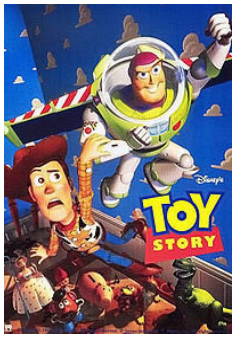


Ilustración 21

En el renderizado RealTime, la situación es completamente distinta. Pero primero es necesario entender en que consiste una aplicación en tiempo real.

Una aplicación en tiempo real consiste en un programa computacional que tiene una naturaleza crítica respecto del tiempo, es decir, una aplicación en la cual la adquisición de datos y la generación de una respuesta debe ser efectuada bajo restricciones fuertes de tiempo.

Dentro de las aplicaciones gráficas en tiempo real podemos encontrar: programas de diseño gráfico (3Ds MAX, AutoCAD), aplicaciones de simulación (simuladores de vuelo), realidad virtual, y video juegos (Quake, Unreal).

Ilustración 22

A diferencia del renderizado Offline, en esta categoría existe interacción por parte del usuario. En un programa de diseño el usuario va modificando constantemente el universo 3D a medida que modela una figura. En un videojuego el usuario cambia constantemente la dirección de la cámara de una forma impredecible.

Si comparamos el ejemplo de un videojuego con el de una película podemos notar grandes diferencias. La navegación de la cámara dentro de un universo 3D para una película es fija. La película se comportará de la misma forma y mostrará los mismos cuadros, por más que la veamos varias veces. En cambio en un programa de diseño o video juego, las imágenes (renders) que se van a estar mostrando en pantalla dependerán de la aleatoriedad de eventos que genere el usuario que interactúa con la aplicación.

Debido a esto, el esquema general para aplicación gráfica en tiempo real debe tener la siguiente forma:

```
while (condicion_corte)
{
        update();
        render();
}
```

En la etapa de update se analizan los eventos de entrada del usuario, se modifica la posición de la cámara, se actualizan los movimientos de las animaciones y se realizan otros cálculos internos.

En la etapa de render se procede a realizar la proyección del universo 3D actualmente visible a la pantalla 2D.

Se necesitan al menos treinta cuadros por segundo (30 FPS) en una aplicación computacional para mantener una buena fluidez de animación. Por lo tanto el bucle while deberá ejecutar al menos 30 veces por segundo. Esto significa que la etapa de renderizado tiene solo 1/30 segundos para ejecutar, sin que se degrade la performance de la aplicación. En la vida real se posee aún menos tiempo, dado que una aplicación hace mucho más que simplemente renderizar (computar la lógica propia de la aplicación en cuestión).

Esta restricción de tiempo marca claramente la diferencia con respecto al renderizado Offline. En una película, cada render puede ser calculado en horas. Lo único importante, dentro de límites razonables de tiempo, es lograr una imagen de la mayor calidad posible.

En cambio en una aplicación RealTime, es necesario ejecutar el proceso de renderización al menos 30 veces por segundo, por lo tanto la performance y no la calidad para a ser el foco principal de estas aplicaciones.

A medida que la tecnología avanza, cada vez hay más técnicas algorítmicas propias de mundo Offline que son utilizadas para aplicaciones RealTime. Pero las tecnologías Offline también avanzan por lo que siempre existirá una diferencia entre ambas.

Graphics Pipeline

Para efectuar un Render, es decir, lograr la transformación desde el modelo lógico 3D provisto por la aplicación al modelo físico 2D que se mostrará en pantalla, existen una serie compleja de pasos a efectuar que se denominan *Graphics Pipeline*.

Una aplicación 3D estará compuesta por los siguientes componentes básicos:

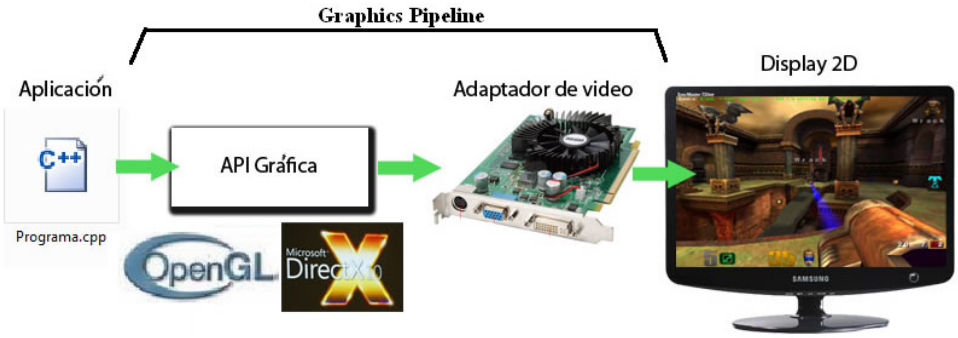


Ilustración 23

A muy grandes rasgos, los pasos de una aplicación 3D son los siguientes:

El programa específica a la API 3D la geometría, texturas e iluminación del mundo 3D que desea construir.

La API 3D, junto con la ayuda de los adaptadores de video, recibe toda la información de la aplicación, la procesa y la transforma para ser proyectada en el plano 2D de una pantalla.

Todo este proceso se denomina *Rendering* y para completarlo se debe pasar por todas las etapas del Graphics Pipeline.

El adaptador de video ya computó todos los píxels necesarios a mostrar en pantalla y los carga en el *Frame Buffer*, para que puedan ser visualizados en un dispositivo (monitor, proyector, etc.).

Las etapas del Graphics Pipeline son secuenciales. La información llega a una etapa, es procesada y luego envía su respuesta a la siguiente etapa.

La entrada inicial del Pipeline será la descripción del modelo lógico del universo 3D y la salida será la matriz de pixeles que se mostrará en pantalla.

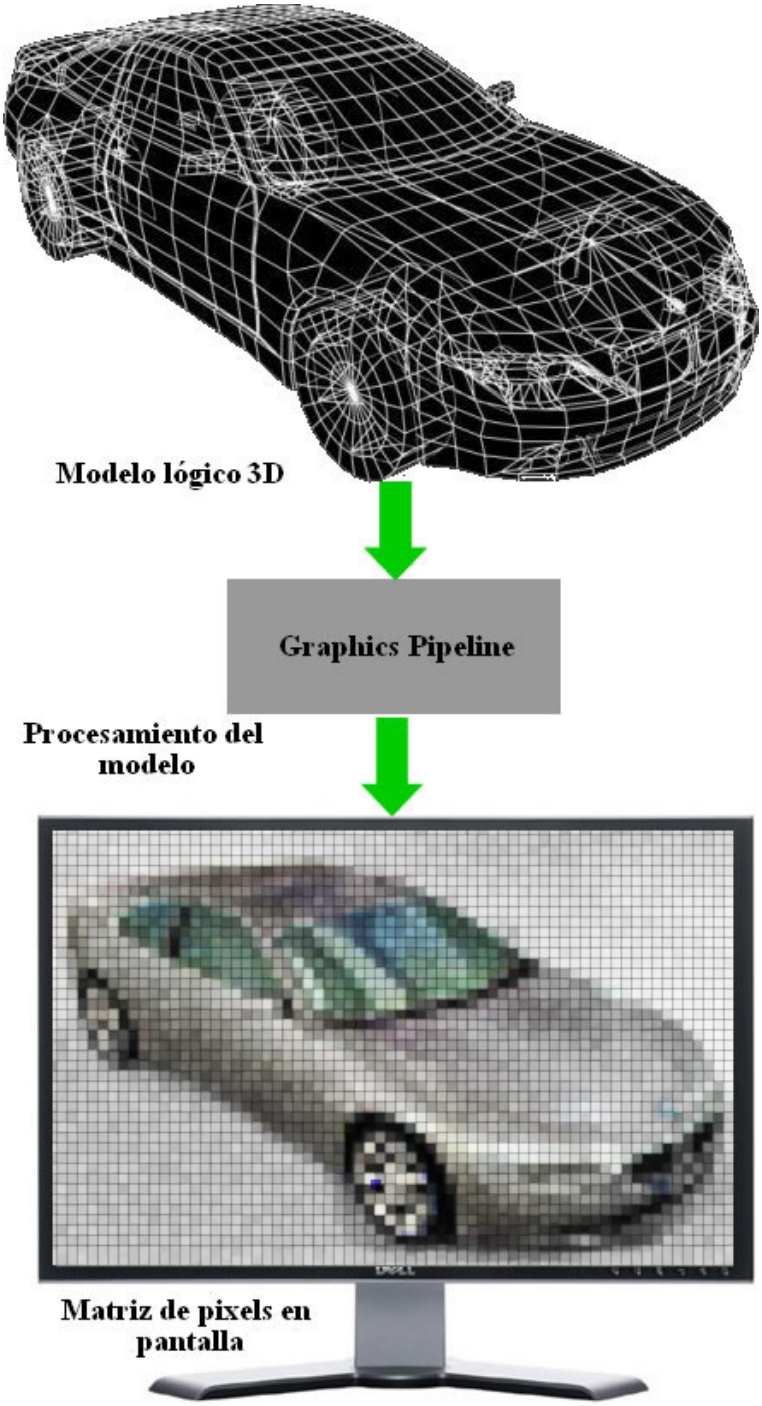


Ilustración 24

Las etapas completas de un Graphics Pipeline dependen de la API que se esté utilizando y del adaptador de video que la implementa. Muchas etapas no pueden ser del todo divisibles, sino que se encuentran entrelazadas. Otras no se encuentran en determinados fabricantes o se encuentran divididas en muchas secciones.

A grandes rasgos todas comparten los siguientes pasos:

- Application/Scene
- Geometry

- Scan-Line Conversion
- Rendering/Rasterization

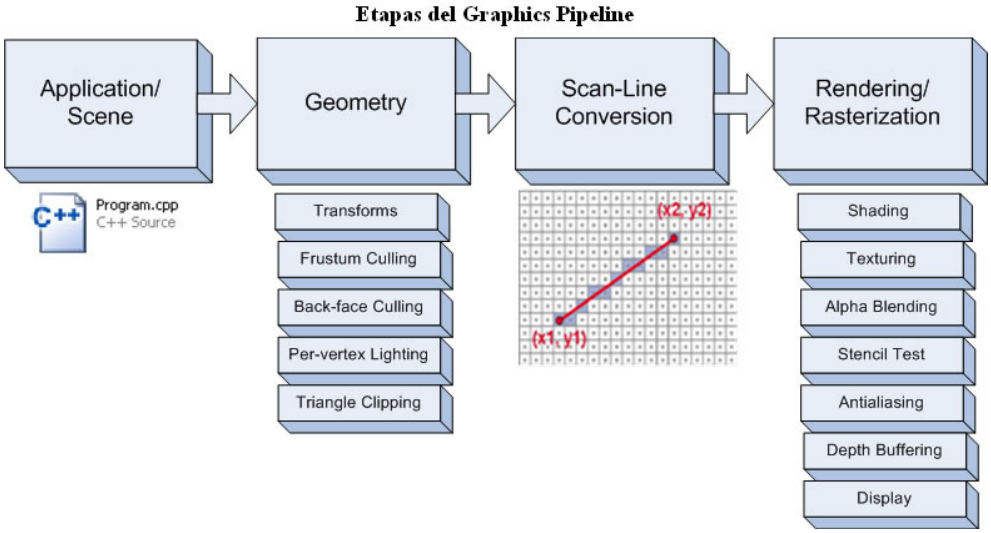


Ilustración 25

Consideración sobre el pipeline

Todo el Graphics Pipeline que se detalló anteriormente es una abstracción de lo que se denomina *Fixed Pipeline*. El nombre de *Fixed* (fijo) radica en el hecho de que las etapas del mismo están perfectamente establecidas y no pueden modificarse. El usuario provee a la API gráfica los datos necesarios de geometría y la misma procede a ejecutar etapa tras etapa del pipeline sin posibilidad de interrupción, con los propios parámetros que el usuario ya ha especificado. El usuario debe adaptarse a las etapas que posee el Pipeline, perdiendo el control de los datos que se generan, una vez que han sido enviados a la API gráfica.

Esta arquitectura de pipeline fijo ha ayudado a los fabricantes de adaptadores de video a optimizar sus chips cada vez más, pero limita al programador en muchos aspectos.

Por citar un ejemplo, la iluminación en tiempo real de todo universo 3D debe hacerse siempre con el método *Gouraud Shading*, el cual posee ciertas limitaciones. Si se quiere aplicar un nuevo método, el mismo deberá ser implementado en hardware en las nuevas aceleradoras de video. Este proceso de evolución es demasiado rígido.

Desde el lanzamiento de *DirectX 7*, el programador tiene la posibilidad de escribir pequeños programas que modifican o reemplazan el comportamiento de algunas etapas del Pipeline. A esta versión del pipeline se la denomina *Programmable Graphics Pipeline*, sus componentes principales son los *Shaders* y representan los últimos avances en materia de gráficos.

APIs gráficas

La evolución de los últimos años en materia de gráficos por computadora se debió en gran parte al mejoramiento continuo de los adaptadores de video. Estas piezas de hardware poseen instrucciones específicas para optimizar los cálculos necesarios para completar el

Graphics Pipeline. Pero existe una gran diversidad de dispositivos, marcas y modelos en el mercado y, lidiar con todos ellos o directamente con sus drivers sería un dolor de cabeza para cualquier programador.

Es por ello que existen las APIs gráficas que abstraen al programador de la interacción directa con la placa de video. Las APIs gráficas más conocidas son *DirectX* y *OpenGL*. Ambas poseen una versión similar del Graphics Pipeline explicado anteriormente. Como el Pipeline es relativamente fijo y estable, las empresas desarrolladoras de adaptadores de videos han ido optimizando cada vez más sus dispositivos para que provean todas las funciones de hardware necesario para acelerar el proceso de rendering.

Ilustración 26

A continuación se mencionan algunas características de las dos APIs más utilizadas:

OpenGL

OpenGL (Open Graphics Library) es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. La interfaz consiste en más de 250 funciones diferentes que pueden usarse para dibujar escenas tridimensionales complejas a partir de primitivas geométricas simples, tales como puntos, líneas y triángulos. Fue desarrollada originalmente por Silicon Graphics Inc. (SGI) en 1992 y se usa ampliamente en CAD, realidad virtual, representación científica, visualización de información y simulación de vuelo. También se usa en desarrollo de videojuegos.

Fundamentalmente OpenGL es una especificación, es decir, un documento que describe un conjunto de funciones y el comportamiento exacto que deben tener. Partiendo de ella, los fabricantes de hardware crean implementaciones, que son bibliotecas de funciones que se ajustan a los requisitos de la especificación, utilizando aceleración hardware cuando es posible.

Hay implementaciones eficientes de OpenGL para Mac OS, Microsoft Windows, Linux, varias plataformas Unix y PlayStation 3.

OpenGL tiene dos propósitos esenciales:

- Ocultar la complejidad de la interfaz con las diferentes tarjetas gráficas, presentando al programador una API única y uniforme.

- Ocultar las diferentes capacidades de las diversas plataformas hardware, requiriendo que todas las implementaciones soporten la funcionalidad completa de OpenGL (utilizando emulación software si fuese necesario).

El funcionamiento básico de OpenGL consiste en aceptar primitivas tales como puntos, líneas y polígonos, y convertirlas en píxels. Este proceso es realizado por una pipeline gráfica conocida como la máquina de estados de OpenGL.

El prefijo "*Open*" en este caso, indica que es una API de uso gratuito pero su código fuente no es abierto.

La API original se encuentra programada en lenguaje C, pero existen muchas versiones para diversos lenguajes, como Java, .NET, Python, etc., denominadas *Bindings*.

La última versión es OpenGL 4.1

Ilustración 27

DirectX – Direct3D

DirectX es una colección de APIs creadas y recreadas para facilitar las complejas tareas relacionadas con multimedia, especialmente programación de juegos y vídeo en la plataforma Microsoft Windows. Algunas de las APIs de las que consta son: Direct3D, DirectInput, DirectSound.

El objetivo de Direct3D es facilitar el manejo y trazado de entidades gráficas elementales, como líneas, polígonos y texturas, en cualquier aplicación que muestre gráficos en 3D, así como efectuar de forma transparente transformaciones geométricas sobre dichas entidades. Direct3D provee también una interfaz transparente con el hardware de aceleración gráfica.

Se usa principalmente en aplicaciones donde el rendimiento es fundamental, como los videojuegos, aprovechando el hardware de aceleración gráfica disponible en el adaptador de video.

Direct3D está disponible solo los sistemas operativos de Windows y es la API gráfica principal de las consolas Xbox y Xbox 360.

La última versión es DirectX 11 la cual se incluye en Windows 7.

Ilustración 28

Forma de interacción con APIs gráficas

Tanto OpenGL como DirectX poseen su propias interfaces de código con las cuales el programador tiene que tratar. La utilización de cada una de las APIs se encuentra fuera de los alcances de este documento pero en esta sección daremos un enfoque general de la forma en que se transmite la información.

Máquina de estado de OpenGL

OpenGL trabaja sobre el concepto de máquina de estado. El programador, en un momento determinado, configura un estado correcto en la API, suministra la información requerida y luego cierra el estado cargado. Toda la información recibida por la API entre la apertura y el cierre de un estado es procesada de una determinada forma.

El siguiente ejemplo muestra una versión simplificada de cómo crear un triángulo 3D con OpenGL:

```
public void render()
{
        //Limpiar la pantalla
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        //Cargar el estado "DIBUJAR TRIANGULOS", todo lo que viene a
        //continuación se considera un triángulo
        glBegin(GL_TRIANGLES);

        //Empezar a dibujar
        glColor3f(1.0f, 0.0f, 0.0f);  // Color del primer vértice
        glVertex3f(0.0f, 1.0f, 0.0f); // Coordenadas del primer vértice

        glColor3f(0.0f, 1.0f, 0.0f);     // Color del segundo vértice
        glVertex3f(-1.0f, -1.0f, 0.0f); // Coordenadas del segundo vértice

        glColor3f(0.0f, 0.0f, 1.0f);     // Color del tercer vértice
```

```
        glVertex3f(1.0f, -1.0f, 0.0f);  // Coordenadas del segundo vértice


        //Cerrar el estado "DIBUJAR TRIANGULOS"
        glEnd();
}
```

Las funciones "*glBegin*" y "*glEnd*" son las encargadas de cargar y quitar un estado determinado en la máquina de estado de OpenGL. Las funciones "*glColor3f*" y "*glVertex3f*" se encargan de enviar un color y las coordenadas de un vértice a la API gráfica. Estas funciones son llamadas tres veces en el ejemplo, una para cada vértice del triángulo. Si quisiéramos dibujar diez triángulos deberíamos llamar a estas dos funciones 30 veces (3 * 10). Todo lo enviado entre los bloques *glBegin*" y "*glEnd*" se considera parte de una lista de triángulos.

El resultado del código anterior dibuja el siguiente triángulo 3D en pantalla:

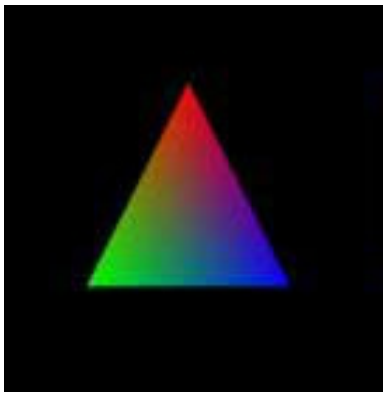


Ilustración 29

Estructuras de DirectX

DirectX se maneja con el concepto de estructuras de vértices. El programador crea un formato de vértice según sus necesidades, especifica este formato a la API y luego envía la geometría a dibujar.

```
public void render()
{
        //Limpiar la pantalla
        device.Clear(ClearFlags.Target, Color.CornflowerBlue, 1.0f, 0);


        //Crear un array de 3 vértices del formato Posición + Color
        CustomVertex.PositionColored[] verts = new CustomVertex.PositionColored[3];


        //Cargar color y coordenadas del primer vértice
```

```
            verts[0].Color = Color.Aqua.ToArgb();
            verts[0].Position = new Vector3(0.0f, 1.0f, 1.0f);

            //Cargar color y coordenadas del segundo vértice
            verts[1].Color = Color.Black.ToArgb();
            verts[1].Position = new Vector3(-1.0f, -1.0f, 1.0f);

            //Cargar color y coordenadas del tercer vértice
            verts[2].Color = Color.Purple.ToArgb();
            verts[2].Position = new Vector3(1.0f, -1.0f, 1.0f);

            //Marcar inicio de envío de información a la API
            device.BeginScene();

            //Especificar formato de vértice a dibujar
            device.VertexFormat = CustomVertex.PositionColored.Format;

            //Dibujar triángulo con array de vértices, indicando que
            //lo que se envía es un triángulo
            device.DrawUserPrimitives(PrimitiveType.TriangleList, 1, verts);

            //Marcar fin de envió de información
            device.EndScene();
        }
```

En el ejemplo, inicialmente se crea un array de vértices bajo un formato específico "*CustomVertex.PositionColored*", que indica que se enviarán a la API vértices con sus coordenadas en el espacio y un color RGB. Luego se procede a cargar el array con los dos valores para cada vértice.

Una vez que el array está cargado se procede a enviar la información hacia la API dentro del bloque limitado por las funciones "*BeginScene*" y "*EndScene*". La información se envía con la función "*DrawUserPrimitives*" que recibe como parámetro el array de vértices creado anteriormente. Previamente debe ser configurado el formato de vértices que se va a enviar a dibujar.

El resultado del ejemplo es el siguiente:

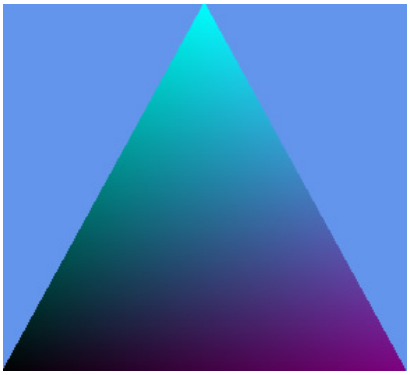


Ilustración 30

Vertex Buffers

En las primeras secciones de este documento se especifica la estructura modelo de una malla. En esta estructura, cada malla tiene sus triángulos y cada triángulo tiene sus vértices, junto con otros datos. Esta estructura de encapsulación de contenido resulta útil para entender la composición de un modelo 3D pero a la hora de trabajar con grandes volúmenes de datos presenta muchas indirecciones.

En una aplicación RealTime la performance es una de las características principales y esto se ve también reflejado en la forma de interactuar con las APIs gráficas.

Debido a esto, en las APIs gráficas se utiliza el concepto de *Vertex Buffer*, que consiste en un array que contiene una lista de vértices uno al lado del otro, en forma contigua.

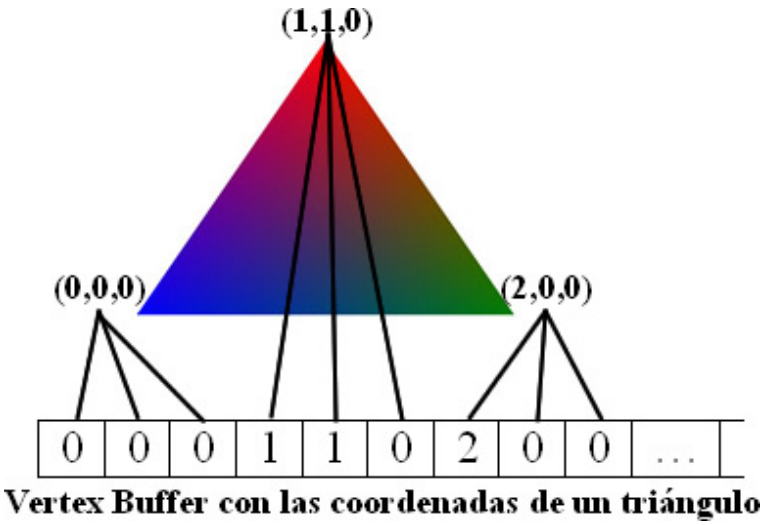
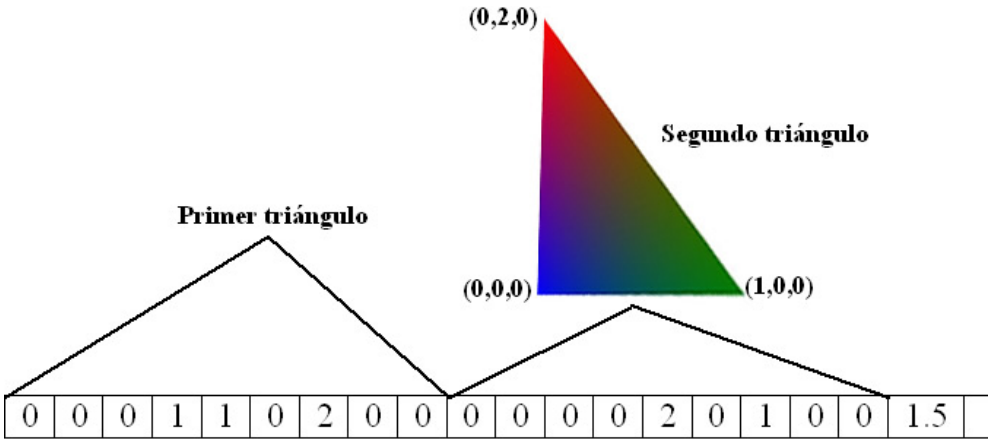


Ilustración 31

Si el usuario desea crear un malla con 10 triángulos no envía cada triángulo por separado, sino que deberá crear un array de floats con los tres valores (x, y, z) de cada vértice, para todos los triángulos, ubicados en forma continua. La API sabe que vamos a trabajar con triángulos como primitivas (se debe configurar previamente), por lo tanto tomará cada valor en forma secuencial y creará internamente la geometría deseada.

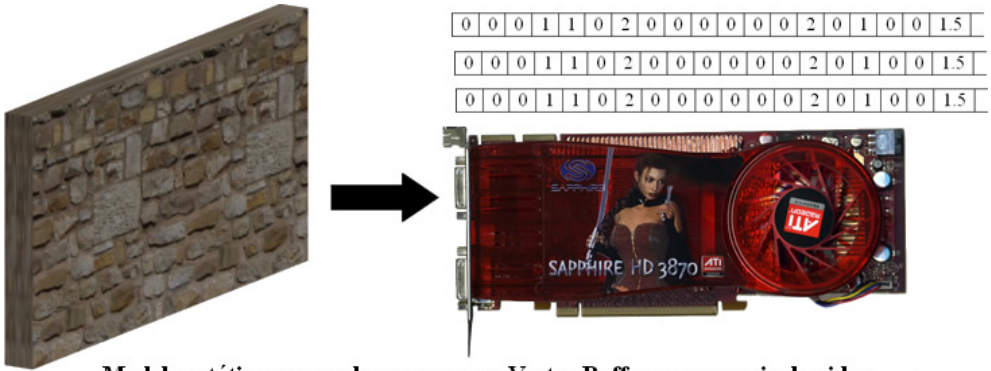


Intepretación de triángulos por parte de la API gráfica, a partir de un Vertex Buffer

Ilustración 32

Esta forma de construcción, que puede parecer incómoda en un principio, ayuda a reducir el overhead por llamada de funciones entre la aplicación, el usuario y la API gráfica. Hay que tener presente que la cantidad de vértices existentes en un universo 3D actual puede estar en el orden de los millones y una llamada a la API para configurar cada vértice sería prohibitivo.

Además esta forma de intercambio de información permite otras optimizaciones. Por ejemplo, aquellas mallas que no vayan a cambiar durante toda la interacción con la aplicación pueden ser marcadas como inmutables y su Vertex Buffer asociado podrá ser almacenado directamente en memoria de video. De esta forma se evita la saturación del BUS entre el CPU y el GPU.



Modelo estático que se almacena en un Vertex Buffer en memoria de video

Ilustración 33

GPGPU

La posibilidad de programar las etapas de *Transform & Lighting* con *Vertex* y *Fragment Shaders* se combinó con el aumento de la capacidad de memoria de las GPUs y con nuevos formatos de texturas que posibilitaron almacenar valores decimales con precisión de punto flotante. En poco tiempo, casi ninguna aplicación gráfica se limitaba a enviar los triángulos directamente a la placa de video. Las texturas se convirtieron en matrices de datos numéricos y

los *Pixel Shaders* se comenzaron a usar para efectuar todo tipo de cómputo sobre los mismos y no solo aquellos relacionados con la iluminación. Gracias a esto, se obtuvo una potencia de cómputo en paralelo sin precedentes a un costo bajo en comparación a hardware con capacidades similares. Toda una nueva gama de algoritmos fueron adaptados para ejecutar en las placas de video y se desarrollaron trucos para solventar las limitaciones de una arquitectura que estaba diseñada con otros fines. Debido a esto, las texturas se transformaron en datos y los pixeles en resultados. Un Pixel Shader podía computar, por ejemplo, el movimiento de un fluido íntegramente en la GPU varios cientos de veces más rápido que el mismo programa ejecutando en un hardware similar (del mismo costo) en CPU. A esto se lo llamo GPGPU (General purpose GPU).

Compute Shader

El Pixel Shader no estaba diseñado para cómputos de propósito general. Algunas de las limitaciones técnicas más importantes eran la falta de instrucciones para sincronizar las tareas y la imposibilidad de redirigir la salida del pixel. Desde el punto de vista del programador, era necesaria una sintaxis más apropiada a cómputos de propósito general. Debido a esto, con DirectX 10.1, surge el concepto de *Compute Shader* (CS). El CS permite sacar provecho de la gran cantidad de procesadores en paralelo de la GPU y consiste en un pequeño programa que será ejecutado en paralelo, usualmente, para una gran cantidad de datos. El CS toma como datos de entrada los buffers constantes de la GPU y un número de identificación llamado thread id, y como salida puede escribir a un buffer asociado a la misma. Si bien el concepto es muy similar a lo que los programadores ya hacían en Directx9 usando el Pixel Shader y texturas, este nuevo *shader* provee funciones para usar memoria compartida y para sincronizar las distintas tareas, las cuales se agrupan y estructuran de manera mucho más conveniente de acuerdo al problema a resolver. En el contexto de un CS, un pixel es llamado *thread*. Sin embargo, no es un thread como en un sistema operativo, ya que a diferencia de estos, los thread en el CS no tienen *instruction pointer individual*, ni sus propios registros, ni se organizan individualmente.

Realtime Vóxel Rendering

Parte B

Modelo de Vóxeles

Renderizado volumétrico

Se llama renderizado volumétrico (*VR*) al conjunto de técnicas para generar proyecciones bidimensionales a partir de datos discretos en 3D llamados vóxeles. El *VR* se utiliza en el diagnóstico por imágenes médicas. Se han desarrollado técnicas de visualización que producen imágenes con un alto grado de exactitud y permiten a los médicos sacar conclusiones a partir de los datos volumétricos. En el contexto del diagnóstico por imágenes es imprescindible una simulación que considere la absorción y emisión de energía para determinar el valor de iluminación o color de cada píxel.

Los métodos de renderizado directo de volúmenes generan imágenes en 3D de conjuntos de vóxeles sin extraer explícitamente superficies geométricas a partir de los datos. Estas técnicas usan un modelo óptico para mapear los datos (representados por muestras discretas organizadas en una matriz tridimensional) a propiedades ópticas como opacidad y color.

Vóxeles

En el modelo de vóxeles los objetos tridimensionales se representan con un conjunto de cubos de cierto tamaño (alto, ancho y profundidad) llamados vóxeles. El vocablo ingles voxel proviene de la contracción de los términos VO*lumen Element*. De esta forma el concepto de voxel es el equivalente al concepto de píxel (PIcture ELement) aplicado a 3 dimensiones.

Cada voxel se corresponde a una ubicación en el espacio y tiene uno o más valores asignados que pueden corresponder a distintas propiedades ópticas (opacidad). Los valores intermedios son inferidos por interpolación de los datos vecinos (reconstrucción).

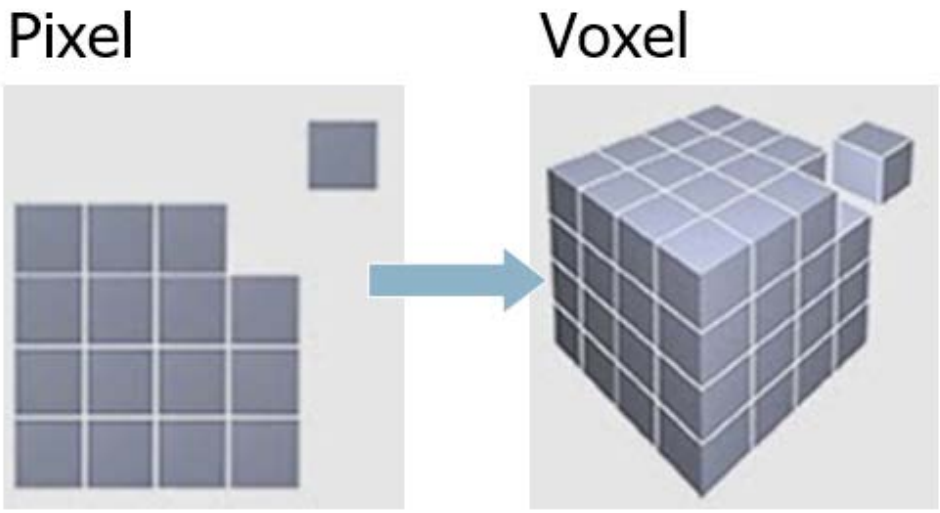


Ilustración 34

Ecuaciones discretas de Volume Rendering

En *Volume Rendering*, las imágenes son creadas muestreando una cierta cantidad de vóxeles (por ejemplo, a través de la proyección de rayos desde el plano de visión hacia la escena) y acumulando las propiedades ópticas resultantes. Para el modelo de emisión y absorción de luz, el color y la opacidad acumuladas son calculadas según ecuaciones discretas,

que son derivadas de ecuaciones continúas realizadas con integrales y por lo tanto difíciles de computar. Las ecuaciones discretas son:

$$C = \sum_{i=1}^{n} C_i \prod_{j=1}^{i-1} \left(1 - A_j\right)$$

$$A = 1 - \prod_{j=1}^{n} \left(1 - A_j\right)$$

Ecuación 1

Donde C_i y A_j son el color y la opacidad asignadas por la función de transferencia al valor del dato en la muestra i

La opacidad A_j aproxima la absorción, y el color de opacidad ponderada C_i aproxima la emisión y la absorción a lo largo del segmento del rayo entre las muestras i e $i+1$. Para el componente del color, el producto en la sumatoria representa la cantidad por la cual la luz emitida en la muestra i es atenuada antes de llegar al ojo. Esta fórmula es evaluada ordenando las muestras a lo largo del rayo de visión y calculando el color acumulado C y la opacidad A iterativamente. Como la ecuación es una aproximación numérica al modelo óptico continuo, la tasa de muestreo, que es inversamente proporcional a la distancia entre las muestras, influye significativamente en la precisión de la aproximación y la calidad del *rendering*

Funciones de transferencia

La función del modelo óptico es describir cómo las partículas del volumen interactúan con la luz. El modelo más común asume que el volumen está formado de partículas que simultáneamente absorben y emiten luz. Los parámetros ópticos del mismo son especificados por los datos de la imagen o son calculados aplicando funciones de transferencia a los datos. El objetivo de estas funciones es enfatizar o clasificar características de interés en los datos. En el caso de imágenes médicas, los scanner generan información de intensidad, es decir, un campo escalar de intensidades de gris y las funciones de transferencia intentan identificar con colores distintas áreas de intensidad. Las más simples usan tablas de sustitución, indicando un color para cada valor de intensidad. Pero existen funciones más elaboradas que tienen en cuenta información de vóxeles vecinos además de la del propio vóxeles para calcular el color.

Métodos de Volume Rendering

Texture Based Volume Rendering

Las placas de video actuales son capaces de almacenar grandes cantidades de datos en formato de imágenes llamados texturas. Están diseñadas para aplicarse sobre primitivas gráficas en el proceso llamado shading o texturing. Muchas placas de última generación tienen soporte nativo para texturas 3D que pueden usarse para almacenar datos volumétricos en forma directa. Las texturas volumétricas están compuestas por rebanadas o cortes 2D que se apilan

internamente formando un cubo y que pueden ser accedidas usando 3 coordenadas de texturas u, v, w.

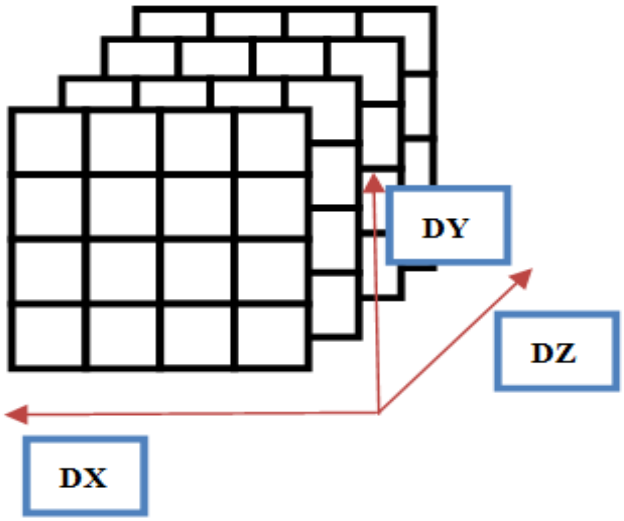


Ilustración 35 - Esquema de textura volumétrica. El volumen se corta en slices (rebanadas) 2D que luego se ordenan a lo largo del eje DZ.

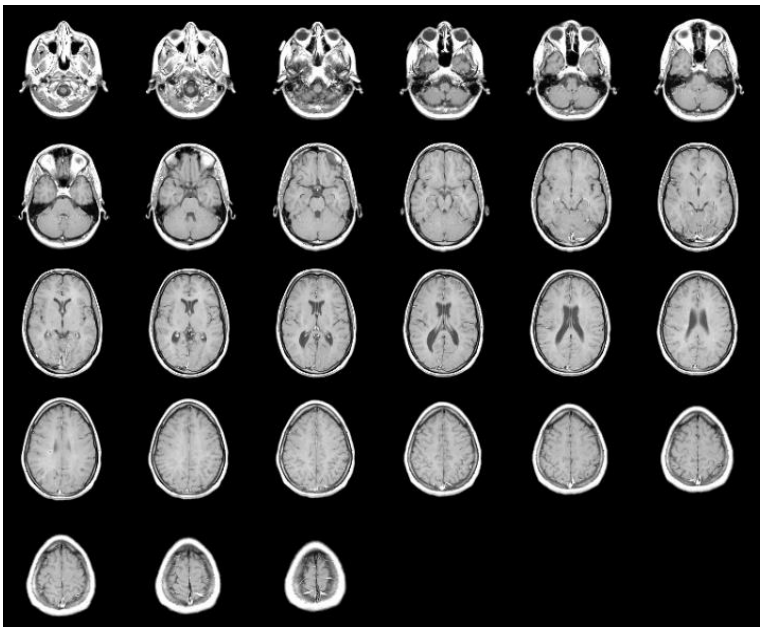


Ilustración 36 - Slices de una textura volumétrica representada en 2D

En general, los algoritmos de renderización de volúmenes basados en textura pueden ser divididos en 3 etapas:

- Inicialización
- Actualización
- Dibujado

La etapa de inicialización normalmente se realiza solo una vez. En cambio, las etapas de actualización y dibujado son ejecutadas cuando la aplicación recibe input del usuario, por ejemplo, cuando se cambia algún parámetro de renderización o visualización

Al comienzo de la aplicación, los volúmenes son cargados en la memoria de la CPU, procesados en caso de ser necesario, por ejemplo, procesar gradientes, y finalmente cargados en la memoria de la GPU. Algunas operaciones de procesamiento de datos pueden ser realizadas fuera de la aplicación

Después de la inicialización y cada vez que cambian los parámetros de visualización, se calcula el *proxy geometry* y se almacena en un arrays de *vertexs* (textura 3D). El *proxy geometry* consiste en un conjunto de polígonos, cortados a lo largo del volumen de manera perpendicular a la dirección de la vista. Los mismos son procesados primero intersectando los planos cortados con los bordes del volumen y luego uniendo (ordenando) los vértices resultantes en el sentido de las agujas del reloj o en el sentido contrario alrededor del centro de manera que resulte un polígono factible de triangular. Por cada *vertex*, la coordenada de la textura 3D correspondiente es calculada en la CPU, en un programa *vertex*, o a través de generación automática de textura-coordenada (*Automatic Texture-Coordinate Generation*)

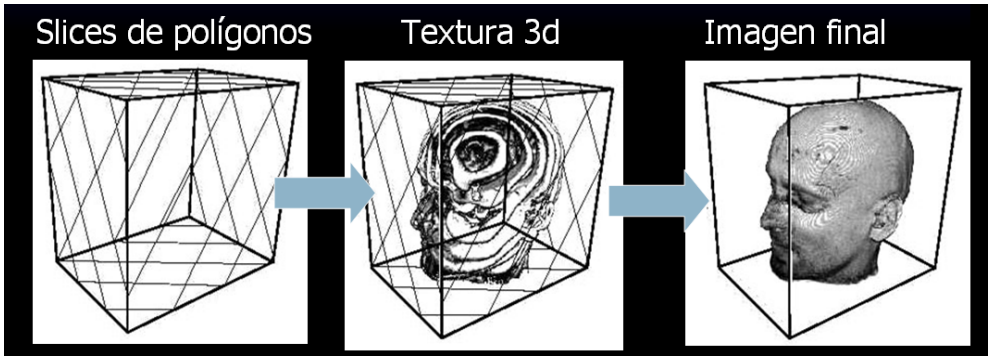


Ilustración 37

Cuando los datos son almacenados como un arreglo de *slices* de texturas 2D, los polígonos *proxy* son simples rectángulos alineados con los *slices*. A pesar de ser más rápido, este enfoque tiene varias desventajas:

- Requiere más memoria, porque los slices de datos necesitan ser replicados a lo largo de cada dirección principal. El uso de memoria puede ser disminuido a costa de performance, reconstruyendo los slices a medida que se necesitan
- La tasa de muestreo depende de la resolución del volumen. Esto se puede mejorar agregando slices intermedios y realizando interpolación trilineal con un fragment shader
- La distancia entre las muestras cambia con el punto de vista, resultando en variaciones de intensidad mientras la cámara se mueve y la aparición de artifacts al cambiar de un set de slices a otro

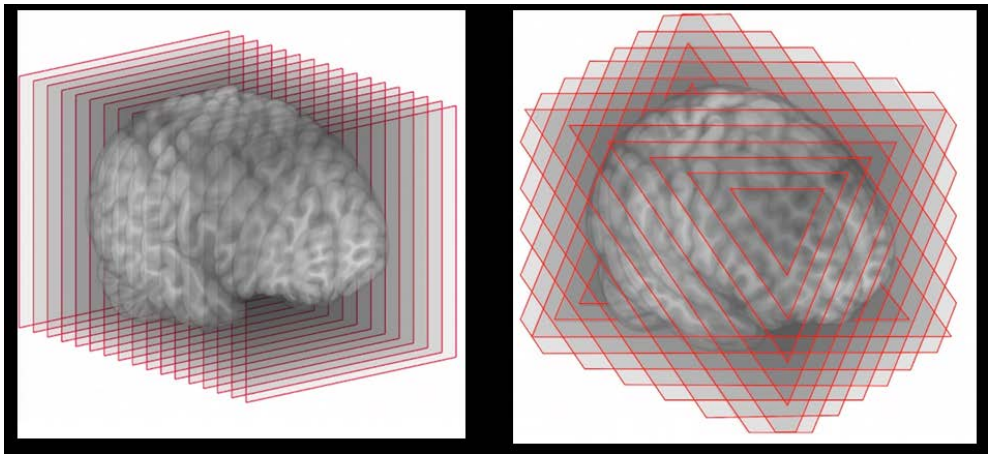


Ilustración 38

Durante la etapa de actualización, las texturas son actualizadas si el modo de renderización o los parámetros de la función de transferencia cambian. Además, la corrección de opacidad de las texturas de la función de transferencia es realizada si la tasa de muestreo cambia.

Antes que los slices de polígonos sean dibujados de manera ordenada, el estado de renderización debe ser configurado correctamente. Este paso comúnmente incluye deshabilitar los procesos de *lighting y culling* y configurar el *alpha blending*. Es muy importante que los *slices* se dibujen de manera ordenada, debido a que cada uno tiene efectos de transparencia, y las funciones integradas de *blend* de la placa de video deben acumular los distintos valores del color. El objetivo del *blend* es componer el color nuevo con el anterior, y la importancia del orden en el dibujado de los *slices* radica en que su ecuación no es conmutativa*:*

$$colorFinal = colorAnterior * (1 - alfa) + colorNuevo * alfa$$

Ecuación 2

Finalmente, luego que los *slices* son dibujados, el estado de renderización es restaurado de manera que no se afecte el dibujado de otros objetos en la escena.

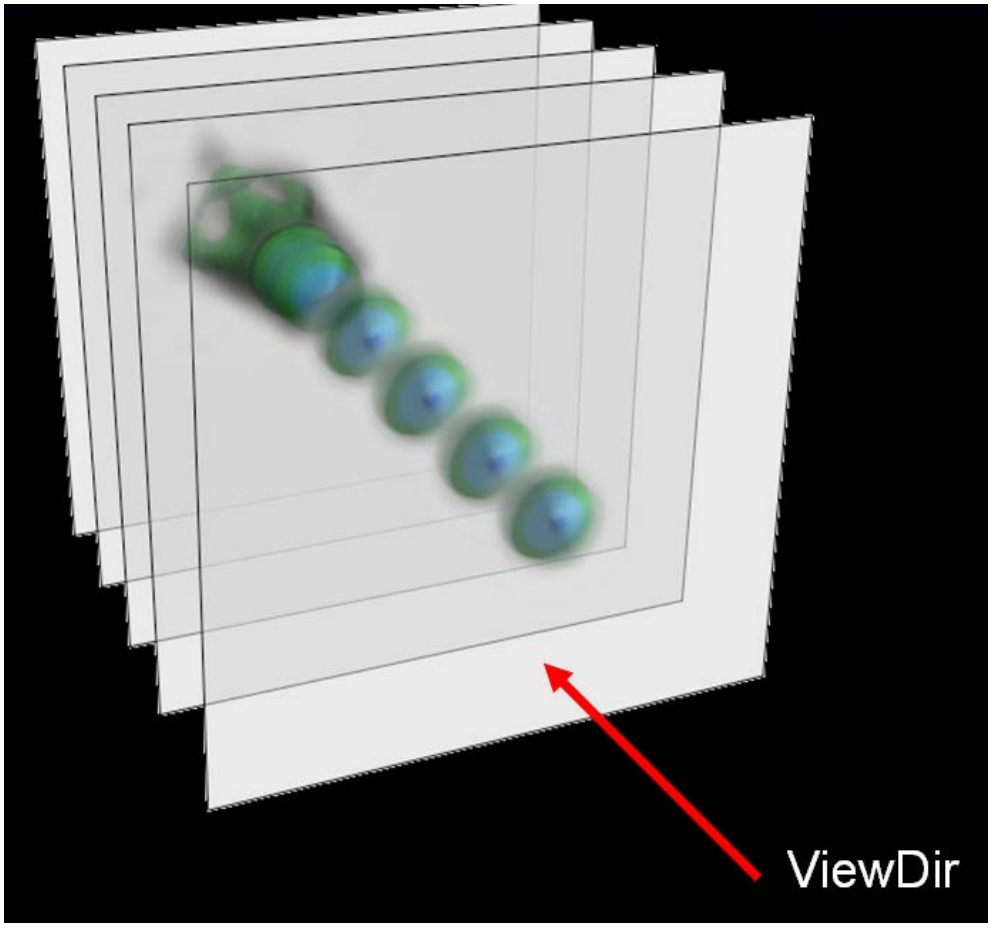


Ilustración 39

Implementando Texture Volume Rendering en Open GL

La implementación usual de texture VR en GPU se basa en dibujar una serie de *Quads* desde la aplicación usando la transparencia y ordenados de atrás hacia adelante.

Los quads representan una geometría intermedia de *slices* o rebanadas, pero que están fijas, es decir, siempre se dibujan en el mismo lugar de la pantalla y no cambian al mover el punto de vista:

```
void RenderEngine::TextureVR()
{
	glEnable(GL_ALPHA_TEST);
	glAlphaFunc(GL_GREATER, 0.05f);

	glEnable(GL_BLEND);
	glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

	glMatrixMode(GL_TEXTURE);
```

```
        glLoadIdentity();

        glTranslatef(0.5f, 0.5f, 0.5f);
        glTranslatef(-0.5f, -0.5f, -0.5f);
        glEnable(GL_TEXTURE_3D);
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_3D, tex.id());

        for (int fIndx = -1.0f; fIndx <= 1.0f; fIndx += 0.01f)
        {
                glBegin(GL_QUADS);
                int TexIndex = fIndx;
                float s = (1 - fIndx) / 4.0f * 0.1f;

                glTexCoord3f(0, 0, ((float)TexIndex + 1.0f) / 2.0f);
                glVertex3f(-1 + s, -1 + s, TexIndex);

                glTexCoord3f(1, 0, ((float)TexIndex + 1.0f) / 2.0f);
                glVertex3f(1 - s, -1 + s, TexIndex);

                glTexCoord3f(1, 1, ((float)TexIndex + 1.0f) / 2.0f);
                glVertex3f(1 - s, 1 - s, TexIndex);

                glTexCoord3f(0, 1, ((float)TexIndex + 1.0f) / 2.0f);
                glVertex3f(-1 + s, 1 - s, TexIndex);

                glEnd();
        }

}
```

Para que los *slices* o rebanadas, que están fijas y no se mueven con el punto de vista, reflejen la orientación del volumen (den la sensación que el mismo está rotando y que se pueda observar desde distintos ángulos), la transformación se hace en las coordenadas de textura. Es decir, en lugar de modificar la posición de los slices y a qué lugar de la textura volumétrica

están mapeados, se realiza el proceso inverso y se cambia las coordenadas de textura en tiempo real en un vertex shader:

```
varying vec3 vTexCoord;


void main()
{
    vTexCoord = (gl_TextureMatrix[0]*gl_MultiTexCoord0).xyz;
    gl_Position = gl_Vertex;
}
```

En este VS la matriz `gl_TextureMatrix` tiene la transformación lineal necesaria para que el volumen sea muestreado teniendo en cuenta el punto de vista del observador. El proceso es muchísimo menos costoso que re-computar los slices en cada frame.

```
    // Activo el modo gl_TextureMatrix
    glMatrixMode(GL_TEXTURE);
    glLoadIdentity();

    // Escalo y roto con respecto al centro del cubo
    glTranslatef(0.5f, 0.5f, 0.5f);
    transform = mat4::RotateX(t) * mat4::RotateY(t) * mat4::RotateZ(t);
    glMultMatrixd((const double *)transform.m());
    glTranslatef(-0.5f, -0.5f, -0.5f);
    glEnable(GL_TEXTURE_3D);
```

Finalmente un fragmente shader accede a la textura volumétrica y eventualmente puede aplicar alguna función sobre la intensidad para resaltar algunas partes del tejido sobre otras. Dichos efectos son implementados usualmente con funciones llamadas de transferencia, y permiten mapear un valor en el espacio de la intensidad (I) al espacio del color (RGBa).

En el siguiente ejemplo el fragment shader descarta los píxels con una intensidad menor a 10% y aplica una función de transferencia que mapea la intensidad a distintos tonos de azul.

```
void main()
{
    vec3 q = (vTexCoord-0.5)* 256;
```

```
    vec3 clr  = texture3D(s_texture0, vTexCoord).rgb;
    float k = clr.g;        // intensidad
       if(k<0.1)
              discard;

       // tejido normal
       if(k>0.9)
              gl_FragColor.a = 0.07;
       else
              gl_FragColor.a = k*k* 0.2;

       gl_FragColor.r = 45.0/255.0 * k * 1.5;
       gl_FragColor.g = 229.0/255.0 * k* 1.5;
       gl_FragColor.b = 237.0/255.0 * k* 1.5;
}
```

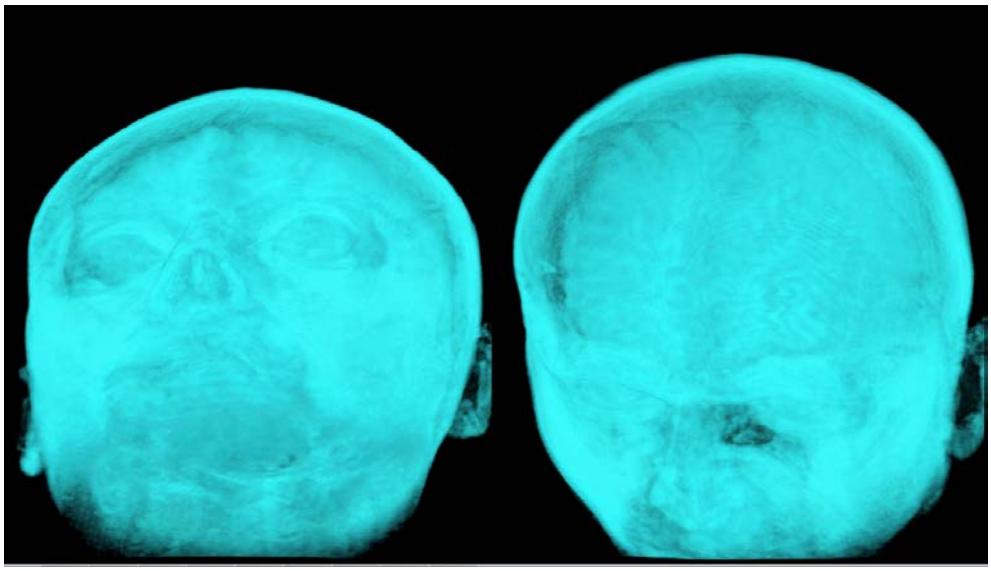


Ilustración 40

La función de transparencia aplica esta ecuación al canal alpha (o canal de transparencia):

```
              gl_FragColor.a = k*k* 0.2;
```

La función cuadrática aplicada sobre el canal de transparencia agrega contraste sobre los tejidos. A continuación se puede observar como varía el contraste / brillo y transparencia con funciones lineales, cuadráticas, cúbicas y cuadráticas:

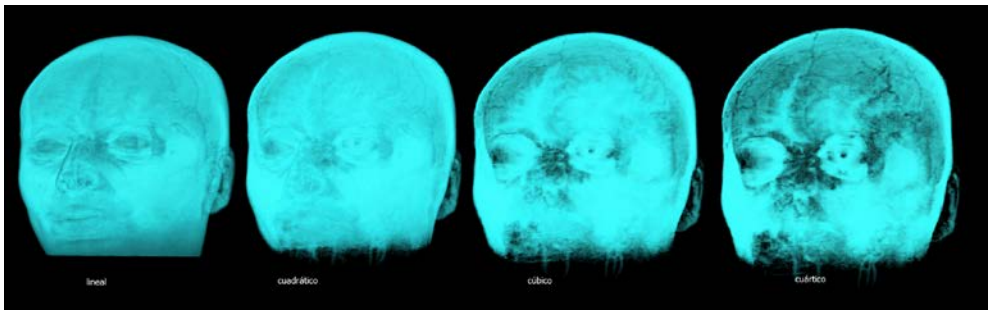


Ilustración 41

La cantidad de slices dibujados tiene un impacto directo en la velocidad de renderizado y en la calidad o precisión de la imagen. El número ideal de slices ronda la cantidad de vóxeles en la dirección de vista, así por ejemplo, si la textura es de 256 x 256 x 256 idealmente se deberían dibujar aproximadamente unos 256 slices.

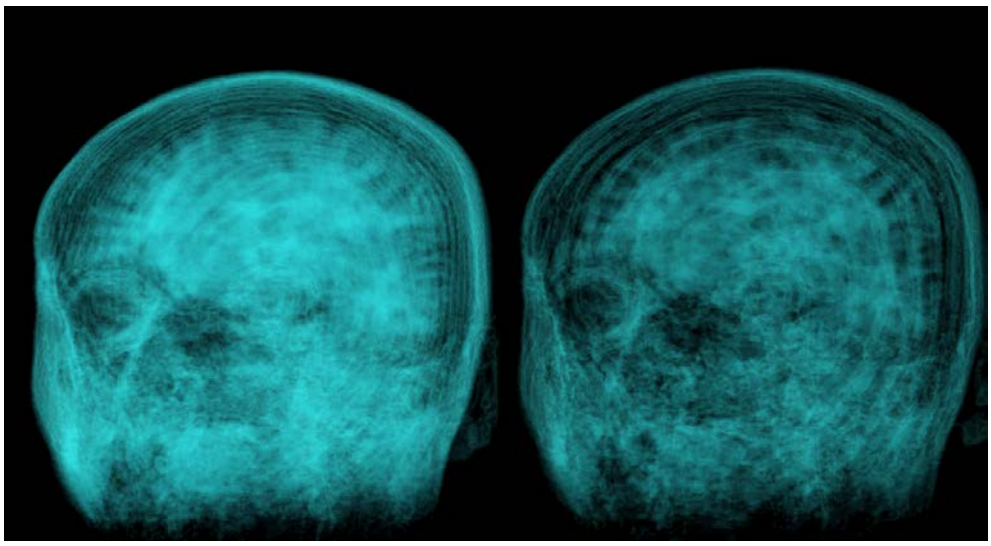


Ilustración 42 - En la figura de la izquierda se usaron el 50% de los slices y en la derecha solo un 25%.

Ray Tracing

La idea en la que se basa *Ray Tracing* es que las imágenes están compuestas por luz. Esa luz proviene de una fuente y llega a nuestros ojos a través de rayos de luz que rebotan en los objetos de la escena. El objetivo del algoritmo de ray tracing es generar imágenes 3D por computadora en una pantalla 2D mediante la simulación del trazado de rayos desde el ojo del observador hacia la escena.

El algoritmo estándar de ray tracing utiliza una gran cantidad de rayos para producir una escena en 3D. Por cada píxel se traza un rayo que va desde la cámara hacia la escena atravesando dicho píxel

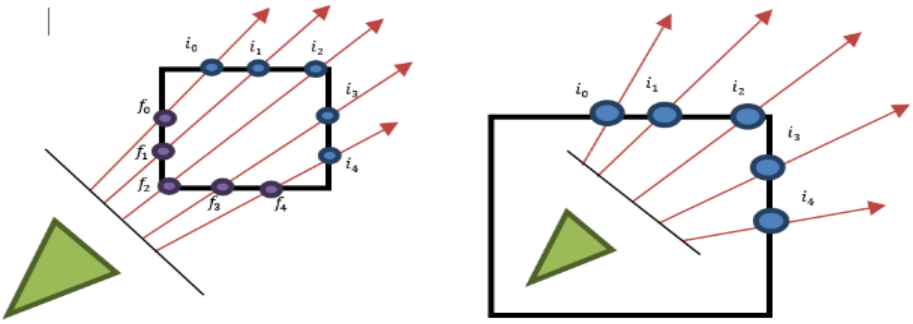


Ilustración 43 - Ray tracing a izquierda punto de vista afuera del volumen a renderizar y a izquierda adentro del mismo.

Ese rayo intersecta con todos los objetos de la escena y se conserva el punto de intersección más cercano. Ese punto, junto con las propiedades geométricas del objeto, se utiliza para calcular la iluminación. Usualmente se generan 3 rayos por cada píxel como mínimo para obtener información de gradiente (el rayo original, otro rayo desplazado levemente hacia abajo y otro levemente hacia la izquierda). Los gradientes se utilizan para acceder a texturas, MipMapping, etc. Esos rayos son llamados *directos* porque determinan el color del objeto original.

En la práctica, los algoritmos industriales y comerciales utilizan cientos de rayos por píxel. Por ejemplo, para simular los efectos de reflexión y refracción se trazan rayos recursivamente desde el punto de intersección que se está sombreando hacia la dirección reflejada o refractada, que depende del punto de incidencia y de la normal a la superficie en dicho punto. Para simular las sombras se lanzan rayos desde el punto de intersección hasta las fuentes de luz. Estos rayos se conocen con el nombre de *rayos de sombra*.

En la actualidad, el algoritmo de *ray tracing* es la base de otros algoritmos más complejos para síntesis de imágenes que son capaces de simular efectos de iluminación global complejos como la mezcla de colores (*color bleeding*)

Debido a sus características, ray tracing es mayormente utilizada en entornos offline como la generación de imágenes foto-realistas.

Ray Casting

Para aplicar las técnicas de ray tracing en entornos de tiempo real se busca reducir la cantidad de rayos por cada imagen mejorando la velocidad de renderizado, dando origen a la técnica llamada R*ay Casting*. En Ray Casting se utilizan muchos menos rayos para formar una imagen aproximada.

Uno de los primeros algoritmos de ray casting se utilizó en el Wolfenstein 3D, donde se utilizaba un solo rayo por cada columna de pixeles. Una vez que el rayo intersecaba una superficie, se generaba toda la línea de pixeles verticales correspondientes a esa dirección de vista realizando un cálculo de relación de alturas de triángulos semejantes. Esto quiere decir

que, por ejemplo, para una pantalla de 640 pixeles de alto y 480 pixeles de ancho, solo se utilizaban 480 rayos, y a partir de cada rayo se generaban los 640 pixeles de alto de la imagen correspondientes a esa dirección. Esta es la característica que hacia veloz al algoritmo (2,5D). Otros algoritmos de Ray Casting solo utilizan un rayo por píxel, sin utilizar rayos de sombra ni ningún otro adicional.

Los algoritmos de Ray Tracing y Ray Casting calculan la intersección de los rayos en forma geométrica. Implementan ecuaciones de intersecciones entre rayos y distintos objetos geométricos (triángulos, cubos, esferas, etc.). Por ejemplo, la intersección entre rayo y esfera, que es muy fácil de computar.

Ray Marching

La técnica de *Ray Marching* consiste en tomar muestras del volumen en intervalos regulares de una distancia predefinida a lo largo de todo el rayo de vista. Esta técnica puede producir distintas situaciones, como se puede observar en las figuras a continuación:

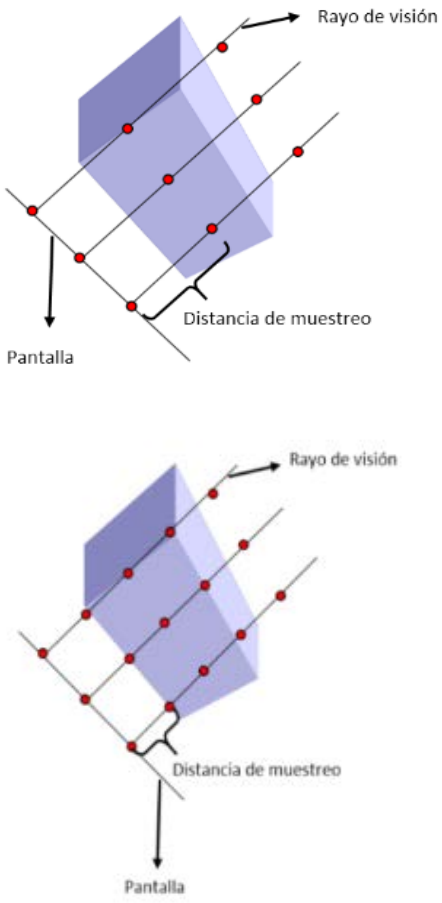


Ilustración 44 - Variación de la cantidad de pasos del algoritmo

En la ilustración, en la imagen superior, la cantidad de pasos es pequeña, hay detalles del objeto que no están siendo muestreados, resultando en una presentación de menor calidad pero más rápida. En la imagen inferior, la cantidad de pasos es mayor, se toma una mayor cantidad de muestras del objeto y la representación es de mayor calidad, pero el proceso es más lento.

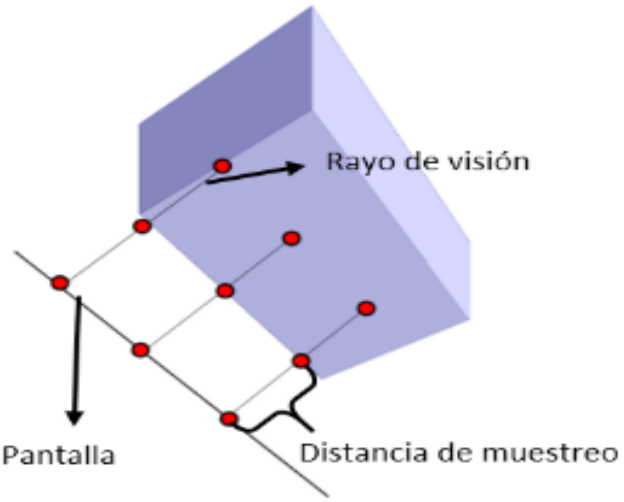


Ilustración 45 - Configuración con poco alcance

En la ilustración el alcance del rayo es pequeño, no se llega a muestrear el objeto completo e incluso puede haber objetos que queden fuera de alcance y no se muestren en absoluto.

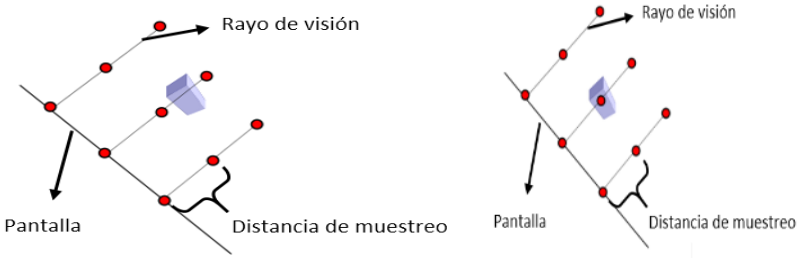


Ilustración 46 - Distancia entre pasos muy grande

En la ilustración se puede observar un artifact que genera esta técnica. Debido a la distancia entre los pasos, hay un objeto que no está siendo muestreado debido a la posición de las muestras y del objeto en sí. Pero al mover la cámara y variar la posición del rayo donde se toman las muestras, el objeto se muestrea y aparece dibujado en la escena. Esto produce que los objetos aparezcan y desaparezcan a medida que la cámara se mueve.

Para atacar el problema anterior existen dos posibles soluciones:

- Realizar Ray Marching con una alta precisión (mayor cantidad de pasos) hasta encontrar un voxel con un valor de opacidad mayor al 50%, y a partir de esa posición comenzar a tomar una cantidad N de muestras.
- Utilizar estructuras de aceleración para poder aumentar la cantidad de muestras a tomar a lo largo del rayo sin afectar la performance del método.

Distance Fields

Los campos de distancia (*Distance Fields*) son comúnmente utilizados para acelerar algoritmos basados en *Ray Casting*. Dado un conjunto de objetos sólidos en el espacio euclídeo, un campo de distancias continuo determina la distancia desde un punto arbitrario al punto más

cercano en la superficie del objeto representado. El *Distance Field* luego es utilizado para acelerar algoritmos que actúen sobre la dirección de un rayo. En él la técnica de *Ray Marching Distance Fields* los datos son muestreados usando un intervalo adaptativo que depende de los valores tomados del campo

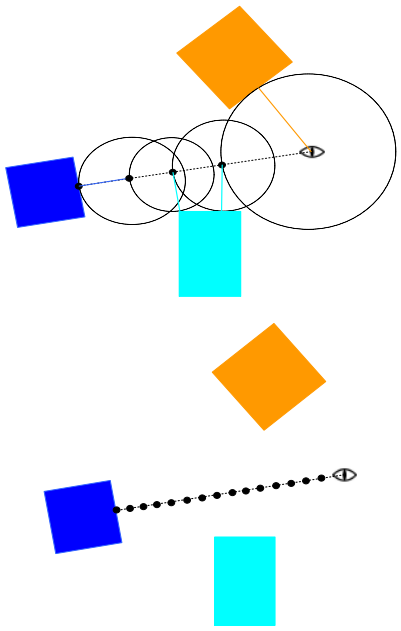


Ilustración 47 - Arriba ray-marching con distance fields, abajo técnica estándar de ray-marching

En las imágenes médicas la mayor parte de los vóxeles tienen como mínimo una cierta intensidad, o sea que prácticamente no hay vóxeles vacíos, por lo tanto, un Distance Field usual no tendría utilidad real. En consecuencia, se define un campo de variaciones S que almacena, en cada elemento de la textura volumétrica, la distancia máxima donde los vóxels vecinos no superan cierta variación con respecto al valor del voxel central.

$$S(p) = d \text{ / si } |p-q| < d \Rightarrow |I(p)-I(q)| < E$$

Ecuación 3

Dónde:

- d es la distancia en vóxeles almacenada en el mapa de variaciones S.
- I es la intensidad del volumen que se está representando.
- E es el valor máximo de variación permitido en el entorno del voxel.

Durante el Ray Marching se utiliza el mapa de variaciones para reemplazar N accesos a texturas por el valor del voxel. Es decir, en lugar de iterar N veces a intervalos regulares, directamente se utiliza el valor del voxel.

Algoritmo de Distance Field

El algoritmo comienza en el punto de partida *r0* (cuando *t* vale cero) y accede a la textura volumétrica para recuperar el valor del voxel actual y al mapa de variaciones que devuelve la distancia máxima sobre la cual no hay mayores variaciones de intensidad en los vóxeles vecinos. La idea principal es que al no haber variaciones significativas en un entorno de distancia d del punto actual, se pueden reemplazar los siguientes pasos por el mismo valor sin demasiada pérdida de precisión. Para determinar la cantidad de pasos que representa la distancia se divide por el paso fijo almacenado en el parámetro *step*. Esa cantidad, llamada *s* en el algoritmo, representa la cantidad de pasos ahorrados, así la distancia *t* se incrementa según el valor de *s* y a su vez la contribución del voxel, o sea el valor *I*, tiene que ser evaluado *s* veces, como si fuese muestreado varias veces.

```
1. C = 0
2. t = 0
3. i = 0
4. while(i<cant_steps)
5.{
6.      p = r(t)
7.      I = tex3D(p)
8.      d = S(p)
9.      s = d / step
10.     C = C + VR(I)*s
11.     t = t + d
12.     I = i + s
13.}
```

- C representa el color acumulado sobre cada rayo.
- t es la distancia recorrida sobre la ecuación del rayo, que es r(t) = r0 + t * rd.
- i es el paso actual.

La eficacia de la técnica depende de la elección del valor de épsilon E. Valores elevados permiten una gran aceleración, pero introducen una serie de artifact en la imagen, mientras que valores demasiados pequeños no generan beneficios. La elección del valor es un parámetro configurable de diseño.

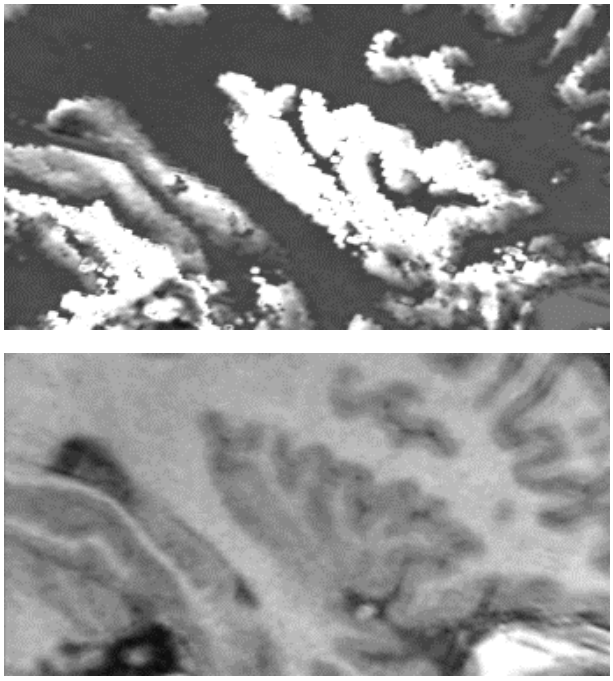


Ilustración 48 - Arriba: artifact por E demasiado grande (E=10px), abajo (E=4px)

Implementando Volume Ray casting en OpenGL

La implementación habitual de Ray Casting usando OpenGL requiere dibujar un quad que ocupa toda la pantalla (*Fullscreen Quad*) a los efectos de llamar a un Fragment Shader por cada píxel y así aprovechar la potencia de cómputo en paralelo de la GPU. Este tipo de quads se los suele llamar *Compute Quads* ya que básicamente es como simular un Compute Shader usando el Fragment Shader.

De esta forma desde la aplicación simplemente se dibuja el Fullscreen Quad con un Vertex Shader trivial (que no realiza ninguna transformación). Se agrega alguna información adicional a los efectos que el Fragment Shader pueda computar la dirección de cada rayo individual.

```
void RenderEngine::RayCasting()
{
    float fov = M_PI / 4.0f;
    float DX = fbWidth;
    float DY = fbHeight;
    float k = 2 * tan(fov / 2);
    vec3 Dy = U * (k*DY / DX);
    vec3 Dx = V * k;

    // Direccion de cada rayo
    // D = N + Dy*y + Dx*x;
```

```
    glMatrixMode(GL_TEXTURE);
    glLoadIdentity();
    glEnable(GL_TEXTURE_3D);

    // Full Screen Quad
    glBegin(GL_QUADS);

    glTexCoord2f(-1, -1);
    glVertex3f(-1, -1, 0);

    glTexCoord2f(1, -1);
    glVertex3f(1, -1, 0);

    glTexCoord2f(1, 1);
    glVertex3f(1, 1 - dy, 0);

    glTexCoord2f(-1, 1);
    glVertex3f(-1, 1 - dy, 0);

    glEnd();
}
```

El Vertex Shader entonces es trivial, simplemente propaga todos los datos hacia el fragment shader:

```
varying vec3 vTexCoord;
void main()
{
    vTexCoord = gl_MultiTexCoord0.xyz;
    gl_Position = gl_Vertex;
}
```

Toda la lógica del método está implementada en el Fragment Shader, que es donde se computa para cada píxel la dirección del rayo que pasa por el punto de vista y dicha posición de pantalla. Sobre ese rayo se muestrea la textura volumétrica y los valores se acumulan aplicando una versión simplificada de la ecuación de Volumen Render.

```
void main()
{
    vec2 uv = vTexCoord.xy*0.75;

    // computo la direccion del rayo
    // D = N + Dy*y + Dx*x;

    vec3 rd = normalize(iViewDir + iDy*uv.y + iDx*uv.x);
    vec3 ro = iLookFrom + rd*voxel_step0;
    vec3 S = vec3(0.0,0.0,0.0);

    // Ray marching
    for (int i = 0; i < cant_total; i++)
    {
        S += tex3d(ro)*k;
        ro += rd*voxel_step;
        k*=k;
    }
    gl_FragColor = vec4(S, 1.0);

}
```

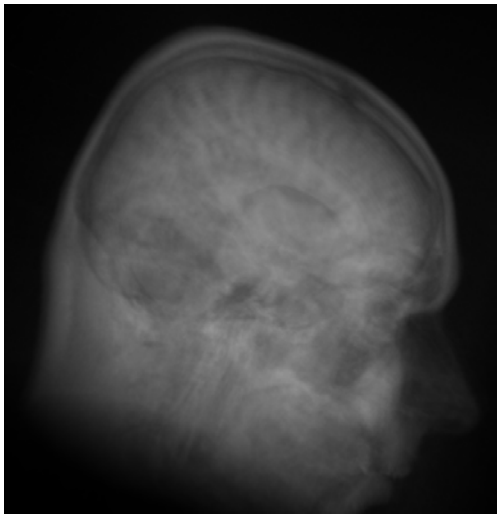


Ilustración 49

Funciones de transferencia

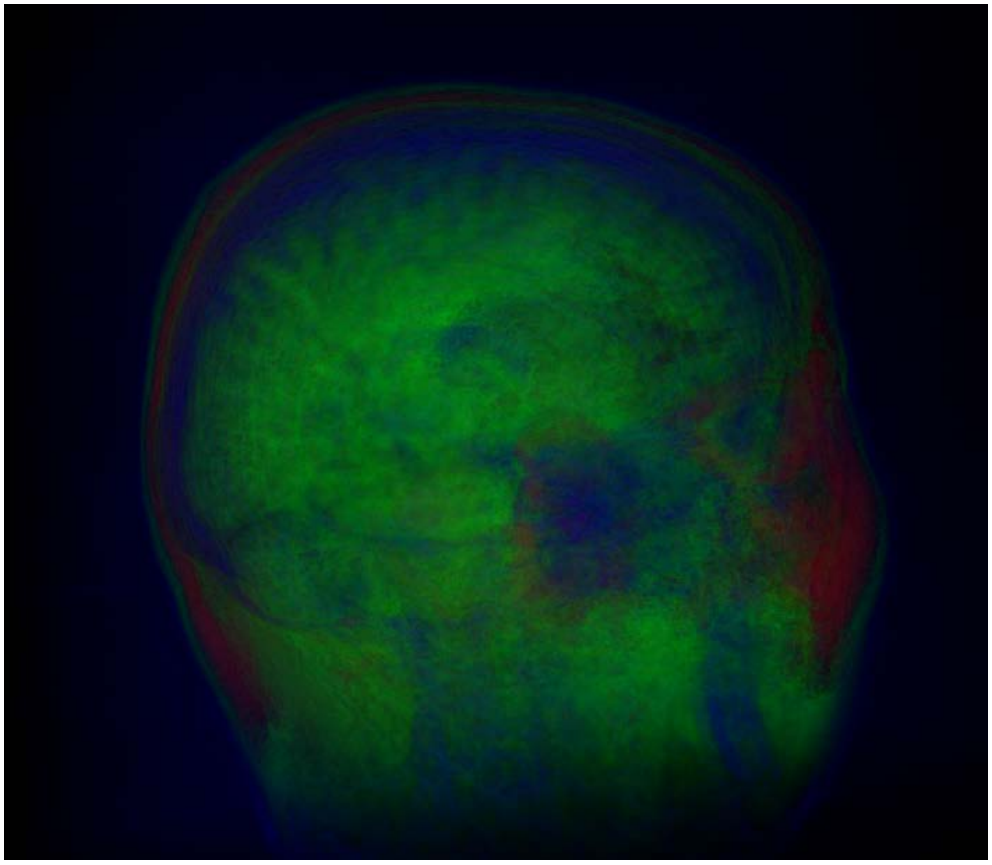
Las funciones de trasferencia sirven para resaltar o destacar distintas regiones del tejido de muestra. Ya que la información volumétrica solo tiene un canal para almacenar una cierta intensidad, en esencia carece de color, sin embargo se puede establecer una relación entre la intensidad computada para cada píxel y un cierto color de salida. Por ejemplo, una función simple puede identificar 3 tipos de intensidades (baja, media y elevada) y asignarles gradientes en los canales azul, verde y rojo respectivamente.

Usando este tipo de función las regiones coloreadas en rojo representarían las zonas de tejido que más energía absorben.

Se pueden diseñar funciones de transferencia mucho más complejas que asignen distintos colores en base a la intensidad de entrada, e inclusive las más elaboradas pueden requerir información de todo una región vecina al voxel evaluado.

```
// transfer function
vec4 transfer(float I)
{
	vec4 clr;
	float t0 = 0.3;
	float t1 = 0.7;
	if(I<t0)
		clr = vec4(0.0 , 0.0 , I/t0 , 1.0);
	else
	if(I<t1)
		clr = vec4(0.0 , (I-t0)/(t1-t0) , 0.0 , 1.0);
	else
		clr = vec4((I-t1)/(1-t1),0.0 , 0.0 , 1.0);
	return clr;

}
```



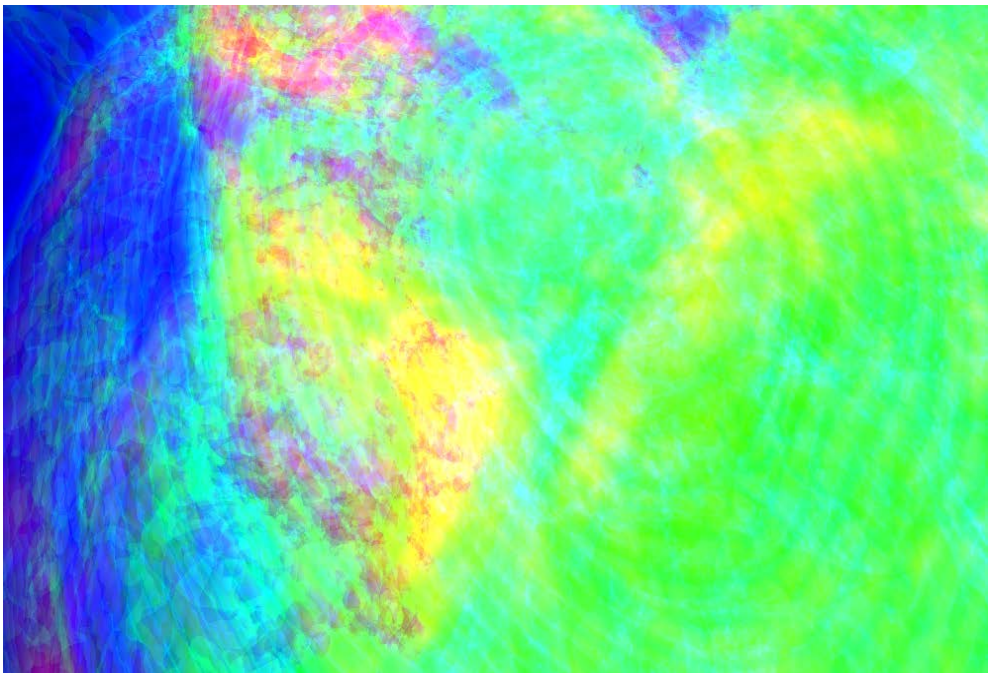


Ilustración 50 - Ejemplos de función de transparencia simple.

Muestras por rayo

La precisión de la imagen final depende linealmente de la cantidad de muestras tomadas a lo largo de cada rayo de visión. Cuantas más muestras se tengan en cuenta al evaluar la ecuación del volumen render mayor será la exactitud de la imagen final. Desafortunadamente, esto también tiene un impacto directo en la velocidad de procesamiento, con lo cual es preciso encontrar un equilibrio entre la resolución requerida que mantenga la tasa de refresco en valores normales (por ejemplo en 30 fps)

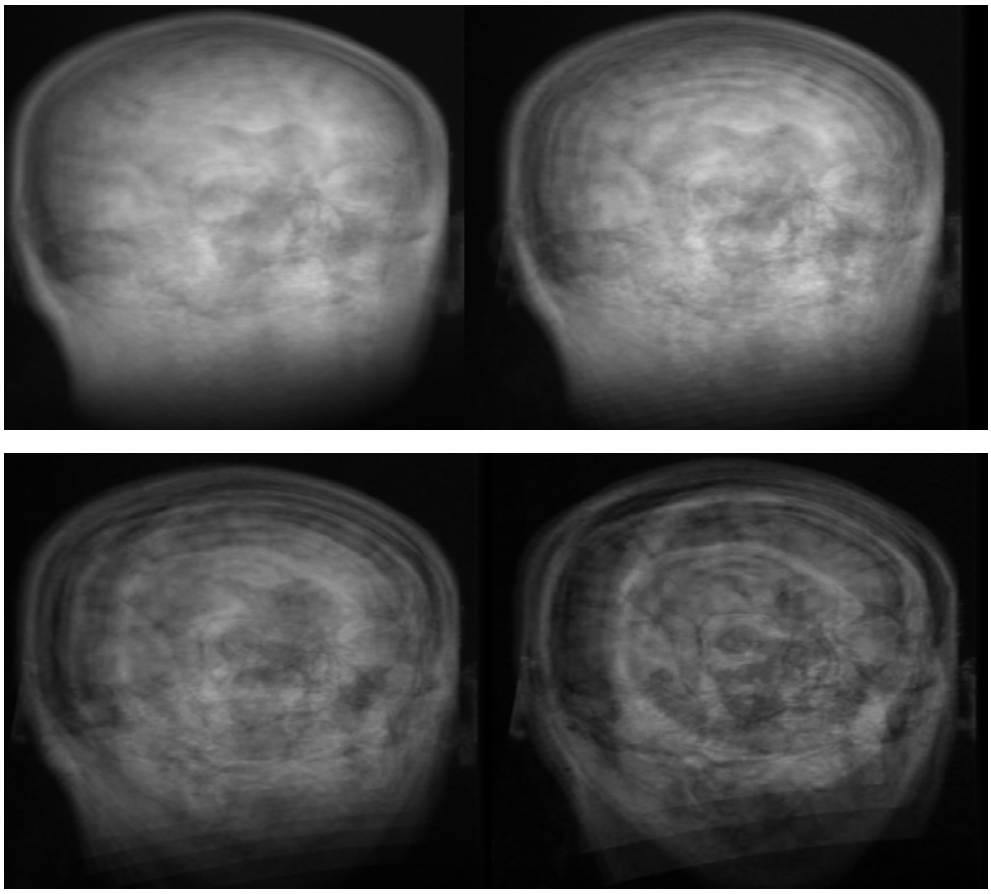


Ilustración 51 - Variación en la cantidad de muestras tomadas a lo largo del rayo. Arriba a la izquierda 100 muestras, a la derecha 75, abajo a la izquierda 40 muestras. La última imagen, con 20 muestras por rayo, exhibe artifacts notables en la parte inferior de la misma.

Bibliografía

- AMIGO, J.C., 2012. Diseño y desarrollo de un juego en WebGL [en línea]. S.l.: Universidad Politécnica de Catalunya. Disponible en: http://upcommons.upc.edu/bitstream/handle/2099.1/15478/82457.pdf?sequence=1&isAllowed=y.

- BASTOS, Thiago; CELES, Waldemar. GPU-accelerated adaptively sampled distance fields. En Shape Modeling and Applications, 2008. SMI 2008. IEEE International Conference on. IEEE, 2008. p. 171-178. 13

- BASTOS, Thiago; CELES, Waldemar. GPU-accelerated adaptively sampled distance fields. En Shape Modeling and Applications, 2008. SMI 2008. IEEE International Conference on. IEEE, 2008. p. 171-178. 13

- COLMENA DE CELIS, J.M., 2010. Un entorno virtual con clientes remotos sobre la plataforma XNA [en línea]. S.l.: Universidad Carlos III de Madrid. Disponible en: http://e-archivo.uc3m.es/bitstream/handle/10016/7585/PFC_Jose_Miguel_Colmena.pdf?sequence=1.

- ENGEL, Klaus, y otros, Real-time volume graphics. CRC Press, 2006.

- GE HEALTHCARE, 2015. Clinical Image Library - Optima NM/CT 640. [en línea]. [Consulta: 9 diciembre 2015]. Disponible en: http://www3.gehealthcare.com/en/products/categories/nuclear_medicine/spect_and_spect-ct/optima_nm-ct_640/optima_nm-ct_640_clinical_image_library#tabs/tab725DE12C6A0A4100955712C9C1BE8766.

- HADWIGER, Markus, y otros, High-quality volume graphics on consumer pc hardware. En IEEE Visualization. 2002.

- HONG, Sunpyo; KIM, Hyesoon. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. En ACM SIGARCH Computer Architecture News. ACM, 2009. p. 152-163.

- HOSPITAL UNIVERSITARIO AUSTRAL, [sin fecha]. Tomografía Axial Computada (TAC) Y Tomografía por Emisión de Positrones (PET). [en línea]. [Consulta: 13 noviembre 2015]. Disponible en: http://www.hospitalaustral.edu.ar/pruebas-diagnosticas/tomografia-axial-computada-tac-y-tomografia-por-emision-de-positrones-pet/.

- IKITS, Milan, y otros, Volume rendering techniques. GPU Gems, 2004, vol. 1.

- KESSENICH, John; BALDWIN, Dave; ROST, Randi. The opengl shading language. Language version, 2004, vol. 1.

- KHRONOS, 2011. What Is WebGL. [en línea]. [Consulta: 7 marzo 2016]. Disponible en: https://www.khronos.org/webgl/wiki/Getting_Started.

- LEFOHN, Aaron E., y otros, A streaming narrow-band algorithm: interactive computation and visualization of level sets. En ACM SIGGRAPH 2005 Courses. ACM, 2005. p. 243.

- LEVOY, M. 1988. "Display of Surfaces from Volume Data." IEEE Computer Graphics & Applications 8(2), pp. 29–37.
- LINDHOLM, Erik, et al. NVIDIA Tesla: A unified graphics and computing architecture. IEEE micro, 2008, vol. 28, no 2, p. 39-55.)
- MAX, N. 1995. "Optical Models for Direct Volume Rendering." IEEE Transactions on Visualization and Computer Graphics 1(2), pp. 97–108.
- MOZILLA DEVELOPER NETWORK. Tutorial Canvas [en línea]. 2016. S.l.: s.n. [Consulta: 1 Febrero 2016]. Disponible en: https://developer.mozilla.org/es/docs/Web/Guide/HTML/Canvas_tutorial
- NOON, Christian John. A volume rendering engine for desktops, laptops, mobile devices and immersive virtual reality systems using GPU-based volume raycasting. 2012.
- NOVO, Rodríguez Javier; PORTO, Díaz Iago; SUÁREZ, Casal Pedro; VALENCIA, Almansa José. Lenguajes de Shading de Alto Nivel, 2005.
- NVIDIA, 2007. The Cg Tutorial. [en línea]. [Consulta: 06 octubre 2016]. Disponible en: http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html
- REZK-SALAMA, Christoph, y otros, Interactive volume on standard PC graphics hardware using multi-textures and multi-stage rasterization. En Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware. ACM, 2000. p. 109-118.
- RHADAMÉS, C., 2015. Introducción al Rendering Directo de Volúmenes. [en línea]. Caracas: Disponible en: https://www.researchgate.net/profile/Rhadames_Carmona2/publication/272493263_Introduccin_al_Rendering_Directo_de_Volmenes/links/54e65fdf0cf2cd2e028ec239.pdf.
- SCHWABER, Ken. Scrum development process. In Business Object Design and Implementation, pp. 117-134. Springer London, 1997.
- SIEMENS, 2015. Magnetic Resonance Imaging - DICOM Images [en línea]. 2015. S.l.: s.n. [Consulta: 1 Febrero 2016]. Disponible en: http://www.healthcare.siemens.com/magnetic-resonance-imaging/magnetom-world/clinical-corner/protocols/dicom-images
- TOMCZAK, Lukasz Jaroslaw. GPU Ray Marching of Distance Fields. Technical University of Denmark, 2012. 8
- TOMCZAK, Lukasz Jaroslaw. GPU Ray Marching of Distance Fields. Technical University of Denmark, 2012. 8
- WEB3D CONSORTIUM, 2014. X3D Medical. [en línea]. [Consulta: 13 noviembre 2015]. Disponible en: http://www.web3d.org/wiki/index.php/X3D_Medical.
- WONG, Henry, et al. Demystifying GPU microarchitecture through microbenchmarking. En Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on. IEEE, 2010. p. 235-246 15