Título Realtime Volume Render en Videojuegos

Tipo de Producto Ponencia (texto completo)

Autores Banquiero, Mariano; Ortiz, Edgar y Rossi, Bibiana

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

Noviembre 2016

INTEC
Instituto de Tecnología

UADE
FUNDACIÓN UADE

Realtime Volumen Render en Videojuegos

Banquiero Mariano
Fundación UADE
Universidad Argentina de la Empresa
Lima 775, Ciudad de Buenos Aires
mbanquiero@uade.edu.ar

Rossi Bibiana
Fundación UADE
Universidad Argentina de la Empresa
Lima 775, Ciudad de Buenos Aires
birossi@uade.edu.ar

Ortiz Edgar
Fundación UADE
Universidad Argentina de la Empresa
Lima 775, Ciudad de Buenos Aires
eortiz@uade.edu.ar

Abstract

Se presentan un conjunto de técnicas para dibujar imágenes médicas en tiempo real en el contexto de un videojuego. En particular, a diferencia de las aplicaciones médicas usuales, se prioriza la velocidad de visualización a expensas de la exactitud en la representación. Se evalúan distintas adaptaciones de las técnicas de ray casting y sus estructuras de aceleración. También se evaluaron distintas representaciones para compactar los datos volumétricos. Por último se presentan cómo funcionan las adaptaciones en una implementación específica usando el poder de cómputo de la GPU y la tecnología OPENGL.

1. Introducción

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

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. [10]. 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 [11]. A cada uno de estos elementos se los llama vóxeles, contracción del término vóxel element, como el equivalente de un píxel en 3D. Cada vóxel se corresponde a una ubicación en el espacio y tiene uno o más valores asignados que pueden corresponder a distintas propiedades ópticas (usualmente la opacidad). Los valores en posiciones intermedias son inferidos por interpolación de los datos vecinos, en un proceso llamado reconstrucción.

1.1. Datos volumétricos

Existen numerosos formatos de datos volumétricos dependiendo del tipo de máquina (scanner, PET, etc.) y el tipo de uso. La mayoría define un tamaño en las 3 dimensiones (ancho, alto y profundidad) en vóxeles, y cada vóxel contiene un valor de opacidad o capacidad de absorción de energía en esa región del espacio. Algunos formatos especifican además la dimensión de cada vóxel, permitiendo distintas resoluciones en cada eje. La cantidad de bits por vóxel define el rango de valores posibles de intensidad, la mayoría de los formatos utilizan de 12 a 16 bits de resolución, mientras que otros pueden llegar a usar hasta 256 bits dependiendo la sensibilidad del escáner.

1.2. Ecuación de Volume Rendering

La ecuación de VR explica cómo la energía es absorbida por las diferentes capas de materia a medida que atraviesa el volumen a estudiar. Los parámetros ópticos son obtenidos directamente de los datos o bien computados a partir de una o varias funciones de transferencia sobre los mismos. Estas funciones tienen por objetivo resaltar o clasificar ciertas partes del volumen.

$$L(x) = \int_{x_A}^{x_B} e^{-\int_{x_A}^{x_B} \emptyset_t(x'')dx''} \in (x')dx'$$

Como la ecuación de VR no puede ser evaluada numéricamente se utilizan distintas aproximaciones como la ecuación de VR discreta [5]:

$$\begin{aligned} L(x) &= \sum_{i=0}^{n-1} c_i \cdot \prod_{j=0}^{i-1} 1-\alpha_j \\ &= c_0 + c_1(1-\alpha_0) + c_2(1-\alpha_0)(1-\alpha_1) + \cdots \\ &\quad + c_{n-1}(1-\alpha_0)\cdots(1-\alpha_{n-2}) \end{aligned} \tag{2}$$

2. Trabajo Previo

En la técnica de *ray-casting* el volumen es muestreado a lo largo de rayos de visión [2] [10] [12]. Las funciones de opacidad y emisión de color deben ser reconstruidas en distintos puntos de muestreo a lo largo de los rayos. Comúnmente, la reconstrucción 3D es realizada por interpolación tri-lineal de los valores de las funciones de opacidad y emisión de luz evaluados en los vértices de la grilla. Las muestras sucesivas a lo largo de un rayo son compuestas para producir el color final del rayo. Se han propuesto distintas técnicas de aceleración del algoritmo estándar. En la técnica conocida como *empty space skiping* las regiones vacías son salteadas utilizando distintas estructuras de organización espacial como *octtrees* [8]. Otro método llamado *early ray termination* establece que el rayo deja de ser computado si se determina que la contribución de los sucesivos vóxeles es menor a cierto umbral de corte [16].
El algoritmo de *ray-casting* es factible de implementarse eficientemente en GPU, ya que cada rayo es independiente y puede ser computado en paralelo aprovechando la arquitectura de multiprocesadores de los adaptadores de video [7]. En este tipo de implementaciones se utiliza una textura 3D o volumétrica y un *fragment shader* para cada dirección de rayo que es ejecutado en paralelo. El *fragment shader* accede a las muestras a lo largo de la dirección del rayo y computa su contribución al color final.
El acceso a memoria no es tan eficiente como los algoritmos basados en geometrías intermedias (*Shear-warp*, *Splatting* y *Texture-based Volume Rendering*). Debido a que las distintas direcciones no tienen coherencia espacial en el espacio de la textura, la mayor parte del tiempo de procesamiento se invierte en el cómputo de la posición a muestrear y no en el cálculo de ecuaciones de VR propiamente dichas.

En los últimos años y con los avances en el hardware de las placas de video, se han publicados numerosos *surveys* en técnicas de compresión de datos volumétricos y estructuras de aceleración para renderizado de imágenes médicas [4] [9], como T*ensor Representations* [14] o *Sparse Coding Methods* [6]. Los métodos soportados por hardware evolucionan y nuevas técnicas ganan el interés de los investigadores [3].

3. Motor de Visualización de Vóxeles

En esta sección se presenta cómo funciona el motor de visualización de vóxeles en tiempo real.

3.1. VR Basado en Texturas

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.

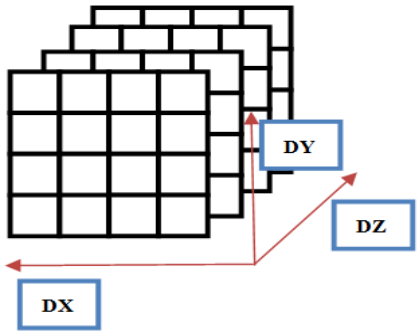

Figura 1. Esquema de textura volumétrica. El volumen se corta en *slices* (rebanadas) 2D que luego se ordenan a lo largo del eje DZ.

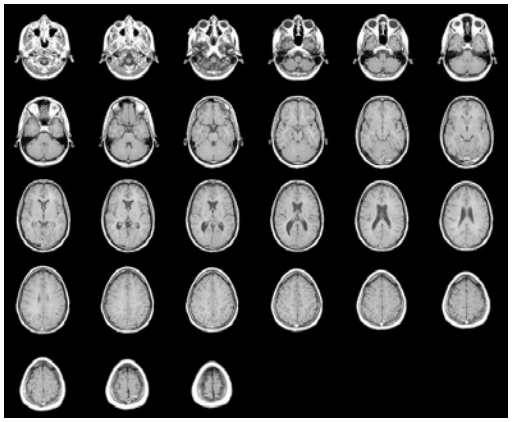

Figura 2. *Slices* de una textura volumétrica representada en 2D.

Las imágenes médicas suelen tener 12 o 16 bits de precisión para almacenar la intensidad de los vóxeles. Con el objeto de minimizar la cantidad de memoria utilizada en la implementación se usa el formato GL_RGBA8, que está compuesto de 4 canales de 8 bits por canal, pudiendo almacenar un total de 4 vóxeles por cada elemento de textura (4x). Considerando que el tamaño máximo de una textura es de 4096*4096 (en una placa genérica), la cantidad máxima de vóxeles es de 4096*4096*4 = 67108864 vóxeles, que pueden ser arreglados en una matriz de 1024*1024*64 o bien de 256*256*1024 dependiendo el tipo de escenario.
Las pruebas realizadas sugieren que no es necesario utilizar una precisión mayor a 8 bits por canal, sin embargo, compactar más vóxeles utilizando técnicas de operadores binarios y/o formatos de 4 bits por canal como GL_RGBA4, dan lugar a artifacts notables en la calidad de la imagen.
Usando un solo bit por vóxel (el vóxel está ocupado o vacío) se podrían lograr factores de compresión de 8*4 = 32, logrando cargar escenarios enteros en una sola textura. Sin embargo para acceder a los datos de un vóxel específico es necesario utilizar operadores del tipo bitwise, no tan eficientemente implementados por hardware, lo cual redunda en un impacto negativo en la velocidad de renderizado. La calidad de las imágenes obtenidas con esta técnica presenta errores notables aun en el contexto de un videojuego.
El formato interno GL_RGBA4 de OPENGL asigna 4 bits para los 4 canales, no hace falta el uso de operadores de bits para acceder a los datos almacenados en la textura. La imagen resultante tampoco tiene la calidad suficiente para un videojuego.

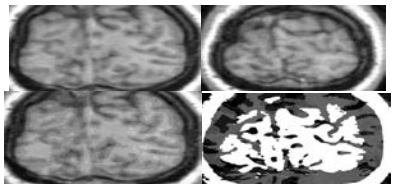

Figura 3. Distintas resoluciones de bits por vóxel. Arriba a la izquierda 8 bits por vóxel, arriba a la derecha 6 bits, abajo a la izquierda 4 bits y a la derecha 1 bits x vóxel.

3.2. Ray Casting

La técnica estándar de ray-casting consiste en generar rayos desde el punto de vista hacia todas las direcciones, luego evaluar, para cada rayo, cual es el punto de entrada y cual el punto de salida en el volumen a renderizar. Por último, iterar por la estructura de vóxeles desde el punto de entrada hasta el punto de salida y en cada paso evaluar la ecuación de VR.

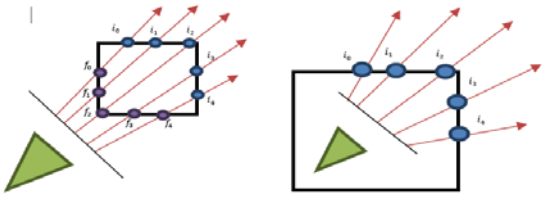

Figura 4. Ray casting de vóxeles. A la izquierda, el volumen es observado desde fuera. Se conocen los puntos de entrada y salida del rayo al objeto, y la distancia entre pasos de la técnica se puede calcular. A la derecha, el volumen es observado desde dentro. Se conoce el punto de salida del rayo desde el objeto, pero no el punto de entrada, por lo tanto se debe elegir una distancia arbitraria desde el ojo del observador para comenzar a muestrear el mismo.

En una implementación en OPENGL, el *fragment shader* se encarga de calcular, para cada píxel de pantalla, la dirección del rayo y los puntos de entrada y salida e iterar por el volumen computando en cada paso en la ecuación de VR. En este caso no existe en un *Vertex Shader*, sino que se dibuja una primitiva que ocupa toda la pantalla, llamada *fullscreen compute quad*, para ejecutar un *fragment shader* para cada píxel de pantalla en paralelo.
En este caso, el punto de vista se encuentra dentro del volumen que se quiere dibujar y la técnica estándar no es efectiva. A medida que el rayo se aleja del punto de partida, la contribución de cada vóxel que se evalúa decrece en forma exponencial como se observa en la ecuación de VR. Para un videojuego, sólo es necesario tomar algunos puntos de muestra en la dirección del rayo. Cuántas más muestras se computen más lento es el proceso, considerando que el acceso a textura es muy costoso en la arquitectura de GPU [17]. Los parámetros a configurar en el motor de visualización son:

- cantidad de pasos: cantidad de accesos a textura.
- paso: distancia en vóxeles a avanzar en la dirección del rayo en cada iteración.
- distancia inicial: resuelve el problema que el observador está dentro del volumen.

Variando estos parámetros se pueden producir varias situaciones como se puede observar en las figuras a continuación:

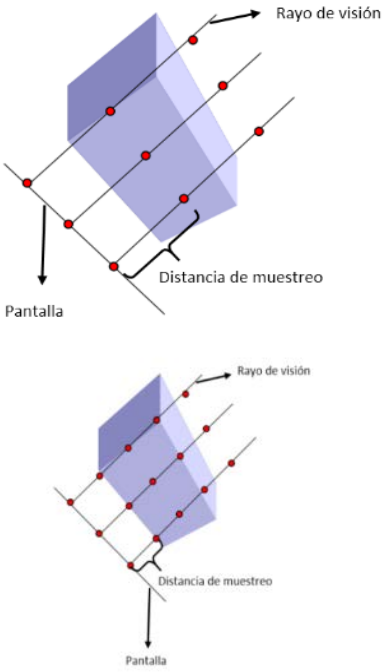

Figura 5. Variación de la cantidad de pasos del algoritmo

En la figura 5, 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 un proceso más lento.

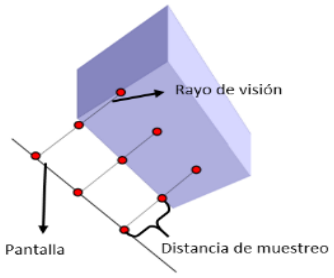

Figura 6. Configuración con poco alcance

En la figura 6, 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.

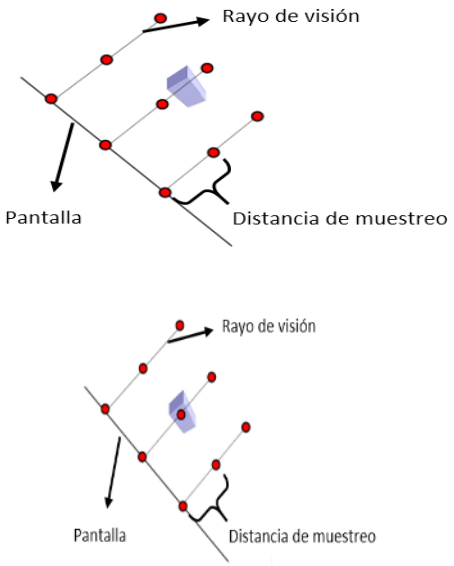

Figura 7. Distancia entre pasos muy grande

En la figura 7 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 de la figura 7 existen dos posibles soluciones:

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

3.3. Distance Field

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 [1]. El campo de distancia 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 [15].

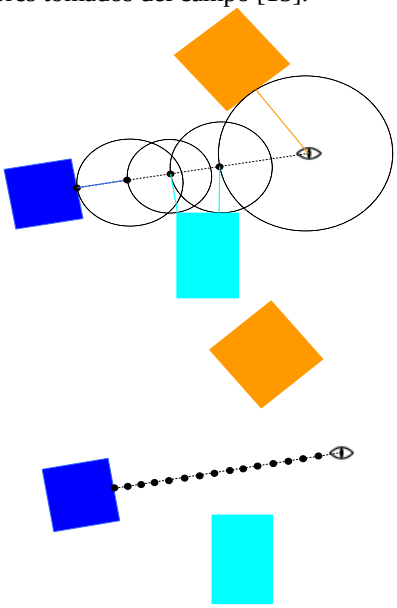

Figura 8. Arriba *ray-marching* con *distance fields*, abajo técnica estándar de *ray-marching*

En las imágenes médicas la mayor parte de los vóxeles tienen como mínimo una cierta intensidad, o sea que prácticamente no hay vóxeles vacíos, un campo de distancia 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 vóxeles vecinos no superan cierta variación con respecto al valor del vóxel central.

$$S(p) = d \ / \text{si } |p-q| < d \Rightarrow |I(p)-I(q)| < E \qquad (3)$$

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 vóxel.

Durante el *ray-marching* se utiliza el mapa de variaciones para reemplazar N accesos a texturas por el valor del vóxel. Es decir en lugar de iterar N veces a intervalos regulares, directamente se utiliza el valor del vóxel.

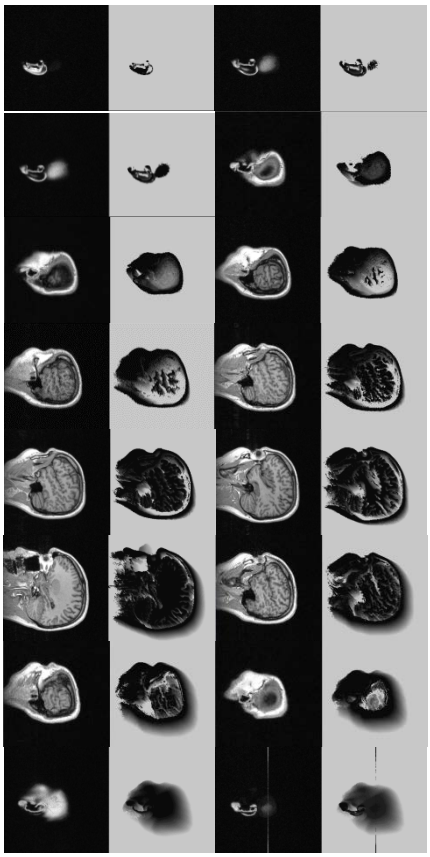

Figura 9. *Tiles* de vóxeles y sus correspondientes *distance fields* para algunos *slices* del volumen.

En la figura 9, el color claro en *distance map* que corresponde a las regiones vacías tiene la máxima distancia que se puede mover. En cambio los colores oscuros, que representan distancias menores, identifican los bordes en las regiones de la imagen.

El algoritmo de *Distance Field* comienza en el punto de partida r0 (cuando t vale cero) y accede a la textura volumétrica para recuperar el valor del vóxel 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 la 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 vóxel, 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 *artifacts* en la imagen, mientras que valores demasiados pequeños no generan beneficios. La elección del valor es un parámetro configurable de diseño.

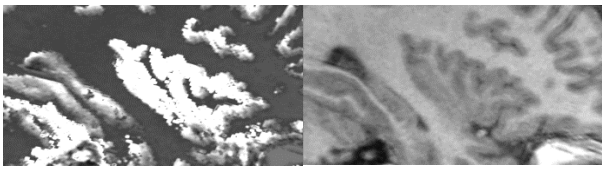

Figura 10. Izquierda: artifact por E demasiado grande (E=10px), a la izquierda (E=4px)

4. Implementación y resultados

Se ha implementado el motor de visualización usando OPENGL y testeado diferentes imágenes médicas reales usando una PC Intel Core Duo 2.40 Ghz con 4GB de RAM placa NVIDIA GForce GTX 660Ti. Las imágenes volumétricas más grandes que se utilizaron están compuestas de 256*256*256 vóxels, que es el tamaño máximo de un objeto de textura soportado por las placas de video genéricas y es el tamaño normal en que se obtienen las imágenes radiográficas. Como se puede observar en la Tabla 1, la cantidad de pasos influye drásticamente en la velocidad de Fotogramas Por Segundo (FPS). También se puede observar que con una cantidad de pasos de casi la mitad del volumen (120), la cantidad de FPS conseguidos es muy baja. En la Tabla 2, se puede observar que al utilizar *distance fields* la velocidad de fotogramas aumenta considerablemente, llegando en algún caso hasta casi el doble.

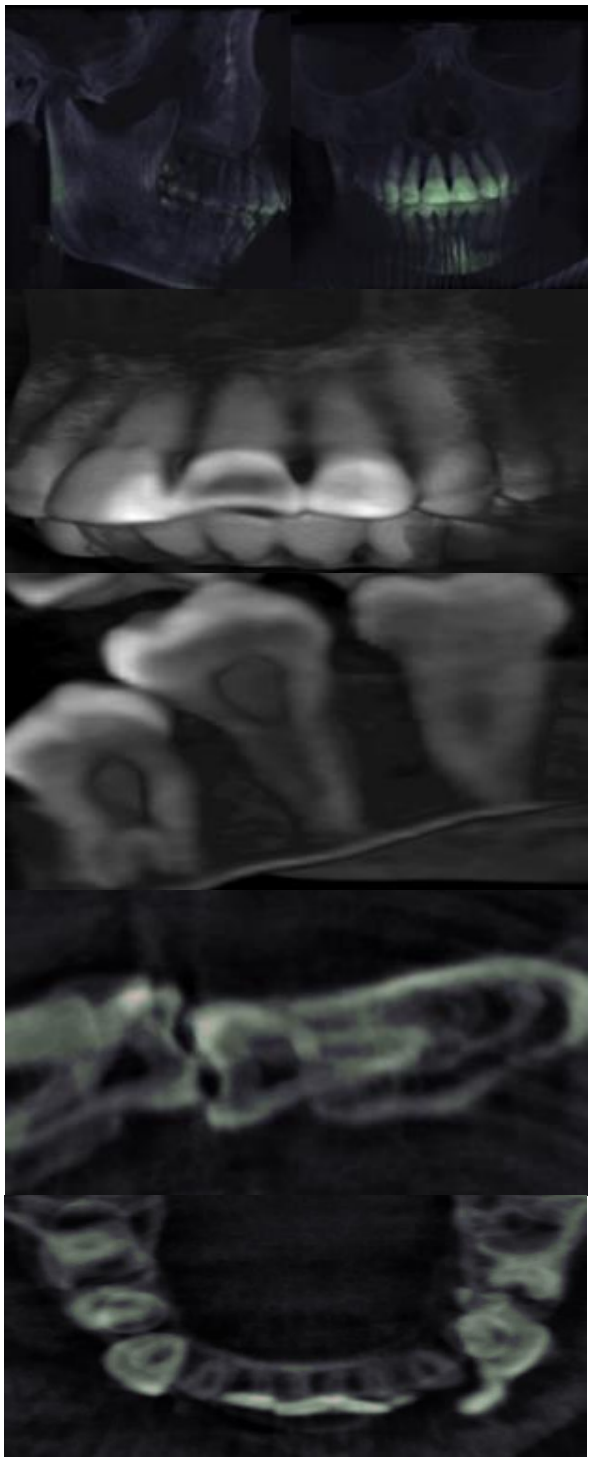

Figura 11. Distintas vistas de una imagen volumétrica de 256*256*256 vóxeles (SKULL)

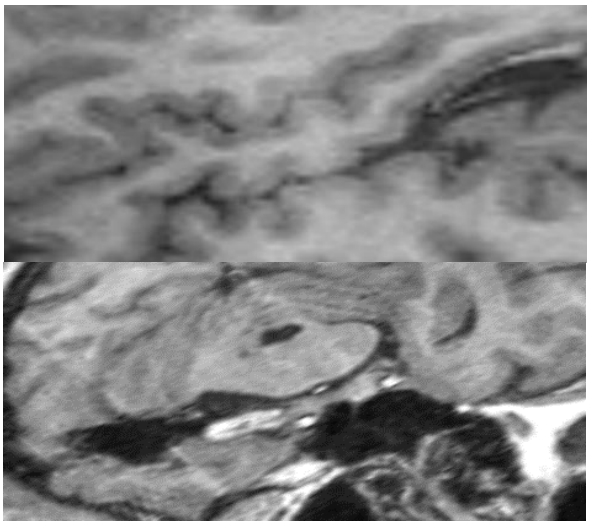

Figura 12. Distintas vistas de una imagen volumétrica de 256*256*256 vóxeles (MRI HEAD)

Tabla 1. Performance del algoritmo variando la cantidad de pasos

Volumen dataset	Tamaño	Pasos	FPS
SKULL	256 x 256 x 256	40	60
SKULL	256 x 256 x 256	60	59.8
SKULL	256 x 256 x 256	80	26.5
SKULL	256 x 256 x 256	120	11.1
MRI HEAD	256 x 256 x 256	40	60
MRI HEAD	256 x 256 x 256	60	54.8
MRI HEAD	256 x 256 x 256	80	23.2
MRI HEAD	256 x 256 x 256	120	10.3
VIS MALE	128 x 128 x 128	80	60
VIS MALE	256 x 256 x 256	120	32.1

Tabla 2. Performance de aceleración: RC = *ray casting*, DF = usando *distance fields*

Volumen dataset	Tamaño	FPS RC	FPS DF
SKULL	256 x 256 x 256	48	59
MRI HEAD	256 x 256 x 256	50	60
VIS MALE	128 x 128 x 128	45	>60
BONZAI	256 x 256 x 256	30	55
TOOTH	64 x 64 x 64	>60	>60

5. Trabajo Futuro

La limitación actual de texturas de 4K de las placas de videos genéricas restringe el tamaño de las imágenes a utilizar como escenario de un videojuego. Dicho tamaño aún es algo pequeño para los estándares actuales por lo que deben ser exploradas distintas técnicas para compilar escenarios más grandes (como tomografías de cuerpo entero) en varias texturas (*tiling*) y utilizar técnicas de *streaming* de datos para cargar las imágenes a medida que se requieren en el desarrollo del gameplay.

Como siguiente paso, para implementar el motor en la tecnología WEBGL será preciso simular las texturas volumétricas usando arreglos de texturas 2D, debido a que la API actual no tiene soporte nativo para texturas 3D. En este caso, un *fragment shader* puede simular la interpolación tri-lineal usando texturas 2D e interpolación bi-lineal acelerada por hardware.

6. Conclusiones

La implementación de la técnica de *ray-casting* en una GPU genérica, con una cantidad máxima de pasos de 60 y junto con adaptaciones puntuales, logra la velocidad de refresco necesaria para dar sensación de fluidez (aproximadamente 60 FPS), a la vez que la calidad de la imagen resulta suficiente para el propósito requerido.

7. Referencias

1. 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
2. Craig Upson and Michael Keeler. V-Buffer: Visible Volume Rendering. Proceedings of SIGGRAPH '88, Computer Graphics 22, 4. August 1988, pp. 59 - 64. 1
3. ELLIS S.: GL_KHR_texture_compression_astc_ldr. OpenGL (4.3 & ES 3) Registry, 2012. 2
4. ENGEL K., HADWIGER M., KNISS J. M., REZKSALAMA C., WEISKOPF D.: Real-time volume graphics. A K Peters, 2006. 3
5. FAST VOLUME RENDERING USING A SHEAR-WARP FACTORIZATION OF THE VIEWING TRANSFORMATION Philippe G. Lacroute Technical Report: CSL-TR-95-678, September 1995 Stanford, CA 94305-4055 16
6. GOBBETTI E., IGLESIAS GUITIÁN J., MARTON F.: Covra: A compression-domain output-sensitive volume rendering architecture based on a sparse representation of vóxel blocks. Computer Graphics Forum 31, 3pt4 (2012), 1315–1324. 4
7. Henning Scharsach. Advanced gpu raycasting. Proceedings of CESCG, 5:67–76, 2005. 5
8. J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In Proceedings of the 14th IEEE Visualization 2003 (VIS'03), VIS '03, pages 38–, Washington, DC, USA, 2003. IEEE Computer Society. 6
9. KOMMA P., FISCHER J., DUFFNER F., BARTZ D.: Lossless volume data compression schemes. In Proceedings of the Conference Simulation and Visualization (2007), p. 169–182. 7
10. Marc Levoy. Display of Surfaces From Volume Data. IEEE Computer Graphics and Applications 8, 3. March 1988, pp. 29 - 37. 9
11. MAX, Nelson. Optical models for direct volume rendering. Visualization and Computer Graphics, IEEE Transactions on, 1995, vol. 1, no 2, p. 99-108. 17

12. Paolo Sabella. A Rendering Algorithm for Visualizing 3D Scalar Fields. Proceedings of SIGGRAPH '88, Computer Graphics 22, 4. August 1988, pp. 51 - 58. 10
13. SIEMENS, 2015. Magnetic Resonance Imaging - DICOM Images [en línea]. 2015. S.l.: s.n. [Consulta: enero 2016]. Disponible en: http://www.healthcare.siemens.com/magnetic-resonance-imaging/magnetom-world/clinical-corner/protocols/dicom-images. 11
14. SUTER S. K., IGLESIAS GUITIÁN J. A., MARTON F., AGUS M., ELSENER A., ZOLLIKOFER C. P., GOPI M., GOBBETTI E., PAJAROLA R.: Interactive multiscale tensor reconstruction for multiresolution volume visualization. IEEE Transactions on Visualization and Computer Graphics 17, 12 (December 2011), 2135–2143. 12
15. TOMCZAK, Lukasz Jaroslaw. GPU Ray Marching of Distance Fields. Technical University of Denmark, 2012. 8
16. WEISS, Jakob. Efficient Rendering of Highly Detailed Volumetric Scenes with GigaVóxels. 2013 14
17. 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