Despliegue de Volúmenes Multi-resolución con Aceleración en GPU en el Cálculo de la Distorsión

K. López[1] y R. Carmona[1]
kijamve@gmail.com, rhadames.carmona@ciens.ucv.ve
[1]Centro de Computación Gráfica. Universidad Central de Venezuela.

RESUMEN

Para realizar el despliegue de volúmenes multi-resolución se asigna un nivel de detalle a cada área del volumen acorde a una prioridad. Para ello primero se establece una representación de múltiples niveles de detalle del volumen. Típicamente hay dos vertientes en la representación: utilizar una jerarquía de octree, o una jerarquía local en cada bloque del volumen. En este trabajo se desarrolla un sistema prototipo para la visualización de volúmenes que utiliza una representación por bloques, y se compara con un sistema previo basado en octree. Las pruebas realizadas sobre determinados volúmenes muestran los beneficios de la jerarquía por bloques sobre la jerarquía de octree, tanto en tiempo de respuesta y calidad en el despliegue. Para asignar las prioridades a las distintas áreas del volumen se utilizan 3 métricas: la distorsión de la representación multi-resolución, la distancia con respecto a un punto de interés y el espacio de memoria de textura a utilizar. El cálculo de la distorsión en particular es un proceso que puede tomarse varios segundos en su versión secuencial, por lo que se implementó un algoritmo paralelo basado en CUDA y otro basado en OpenMP para mejorar los tiempos de respuesta.

ABSTRACT

In order to perform a multi-resolution volume rendering, a level of detail is assigned to each volume area according to a priority. First, a hierarchical volume representation of multiple levels of detail is established. There are typically two approaches in the representation: using an octree hierarchy, or using a local hierarchy on each volume block. In this work, we develop a system prototype for volume rendering which uses the block based representation, and we compare this approach with a previous system which uses an octree. The tests over some volume datasets show the benefits of the block-based representation in response time and rendering quality. We use 3 metrics to assign priorities to the volume areas: multi-resolution distortion of the representation, distance to an interest point and amount of texture memory size to use. The distortion computation may take several seconds in its sequential version; thus, we implemented a CUDA-based and an OpenMP-based parallel algorithm to improve the response time.

Keywords: volúmenes multi-resolución, ray casting, textura atlas, jerarquía por bloques, distorsión basado en los datos, CUDA, OpenMP.

1. Introducción

La visualización de volúmenes hoy en día tiene una gran importancia en distintas áreas, incluyendo la áreas de la medicina, mecánica de fluidos, explotación de suelos, simulaciones en física y química, entre otras. Gracias al avance en la resolución de los equipos utilizados en estas áreas y en los diversos proyectos de investigación, los datos tridimensionales pueden llegar a resoluciones increíblemente altas. Las técnicas de visualización de volúmenes multi-resolución lidian con el problema de la limitación de memoria. En base a ciertos criterios, se asigna una prioridad (y así una resolución) a cada área del volumen a desplegar. Entonces, es necesario contar con una representación multi-resolución del volumen. Típicamente hay dos enfoques para representar al volumen en un esquema multi-resolución. Uno de ellos consiste en utilizar una estructura de octree (árbol de grado 8) [BNS01] [LDH*99]. Para ello, inicialmente se particiona el volumen original en *bricks* o ladrillos de igual tamaño (e.g. bricks de 16^3 muestras). Cada 8 de estos bricks se agrupan y se sub-muestrean en un simple brick (también de 16^3 muestras), generando su representación en un nivel de detalle inferior. Este proceso se repite recursivamente, hasta contar con un sólo brick que representa a todo el volumen (la raíz del octree). El otro enfoque (Ljung et al. [LWP*06]) consiste igualmente en dividir el volumen original en bricks de igual tamaño. Luego, cada brick es submuestreado independientemente para generar todos sus niveles de detalle (e.g. 16^3, 8^3, 4^3, 2^3, 1^3). En este caso, se crea una jerarquía local multi-resolución de bloques o *blocks* por cada brick.

En el Centro de Computación Gráfica de la UCV, se cuenta con un sistema de visualización multi-resolución de volúmenes que utiliza la estructura jerárquica de octree, Carmona et al. [CF*11]. Entre los inconvenientes de este sistema tenemos:

a) Al refinar un brick, este es reemplazado por sus 8 bricks hijos, donde la resolución resultante en algunos de estos 8 bricks podría ser innecesaria, haciendo un uso ineficiente de la memoria.

b) Durante el rendering, los bricks seleccionados son ordenados por profundidad, y desplegados independientemente, lo cual puede generar un cuello de botella en el envío de geometría al GPU cuando el número de bricks es grande (e.g. cientos de miles). Adicionalmente, al desplegar cada brick, los rayos deben ajustarse a la frontera de los mismos, generando artefactos visuales.

En este trabajo se implementa un sistema basado en la jerarquía por bloques para corregir las deficiencias del sistema previo. Los bloques seleccionados para visualización son almacenados en una gran textura 3D (atlas) similar a los trabajos de Ljung et al. [LWP*06] y Lux et al. [LB*09], de manera tal de realizar el ray casting en una pasada, reduciendo la cantidad de geometría y fragmentos a procesar. Así, ya no hay necesidad de ajustar los rayos a las fronteras de cada brick, sino únicamente a las fronteras del atlas. En las pruebas realizadas se comparan ambas implementaciones, mostrando las bondades de utilizar la jerarquía por bloques sobre la jerarquía de octree. El criterio utilizado para determinar el nivel de detalle de cada bloque toma en cuenta la distorsión que se comete al representar un área del volumen con un nivel de detalle inferior al original, en el dominio de la función de transferencia. Este cálculo requiere del recorrido de los vóxeles en distintos niveles de detalle, y considerando que estos volúmenes pueden contener cientos de millones de vóxeles, se pierde la interactividad mientras se edita la función de transferencia. Es por ello que se desarrolla un algoritmo paralelo utilizando el GPU (CUDA) o los núcleos del CPU (OpenMP) para reducir los tiempos de respuesta durante la edición de la función de transferencia.

2. Trabajos relacionados

Diferentes trabajos se han realizado en el área de visualización de volúmenes multi-resolución, desde que fue introducido en el año 1999 por LaMar et al. [LDH*99]. En particular, Ljung et al. [LWP*06], utilizan una jerarquía multi-resolución basada en bloques donde cada bloque a visualizar se almacena en una única textura y es indexada en otra textura de índices. En cada vóxel de la textura de índices se almacena la posición y el nivel de detalle de cada bloque a visualizar. De esta manera, al hacer ray casting sobre la textura de índices, se accede a la textura del atlas para obtener el valor de cada vóxel en la travesía de cada rayo. En el criterio de selección consideran la distorsión multi-resolución, específicamente el error en el espacio de color CIELuv (del francés, Commission Internationale de l'éclairage, l'espace colorimétrique L* u* v*, en español, Comisión Internacional de la Iluminación, espacio de color L* u* v*). Este error debe ser recalculado en cada bloque cada vez que el usuario haga un cambio en la función de transferencia.

Debido a que el cálculo del error en todos los bloques puede tomar decenas de segundos, Wang et al. [WGS07] utilizan un histograma 2D llamado *Summary Table* o tabla de sumarización para reducir el cálculo redundante del error en cada bloque. Se dieron cuenta que existe una correlación entre los vóxeles de los distintos niveles de detalle, por lo cual almacenan en cada entrada (i,j) de una tabla de tamaño 256^2 el número de veces en que un vóxel de valor i del bloque original es aproximado por un vóxel de valor j del bloque de menor detalle. Así, el error entre el vóxel i y el vóxel j es calculado una sola vez en el espacio CIELuv y multiplicado por su frecuencia en la tabla de sumarización. En sus pruebas, el tiempo en el cálculo del error es reducido de 43 segundos a 13 segundos.

Lux et al. [LF*09], implementan una forma similar a [LWP*06] de utilizar una única textura (atlas) para almacenar los bricks a visualizar, pero utilizando una representación multi-resolución de octree. Adicionalmente, soportan la visualización de múltiples volúmenes en una misma escena utilizando un árbol BSP (Binary Space Partitioning o Partición Binaria del Espacio).

Carmona et al. [CF*11] se basan en una jerarquía octree y en un despliegue utilizando ray casting de múltiples pasadas (una por brick). Los bricks seleccionados se ordenan por profundidad para luego ser desplegados con ray casting de manera independiente. Cada brick se almacena en una textura 3D individual. La selección de los niveles de detalle considera la distorsión multi-resolución similar a [WGS07], la distancia a un punto de interés y la capacidad de la memoria de textura.

Este trabajo utiliza una jerarquía de bloques similar a [LWP*06] y unos de sus propósitos es acelerar el tiempo de respuesta en el cálculo de la distorsión con el uso de GPU y OpenMP. Adicionalmente, se pretende realizar comparaciones de calidad y tiempo de respuesta con una implementación basada en octree [CF*11], mostrando así los pros y contras de la estructura de bloques sobre la estructura de octree.

3. Jerarquía multi-resolución

En este trabajo se utiliza una jerarquía multi-resolución basada en bloques. Consideremos una textura unidimensional 33 vóxeles como se puede ver en la Figura 1. Esta textura es dividida en sólo 2 bloques en el nivel de detalle original (nivel 0), de 17 vóxeles cada uno. El vóxel frontera (ver vóxel azul) entre los dos bloques es almacenado 2 veces, es decir, ambos bloques comparten dicho vóxel para garantizar una correcta interpolación entre bloques del mismo nivel de detalle [GHY98]. En cada bloque se genera una jerarquía multi-resolución local, en donde un vóxel de nivel de detalle i+1 representa el área de dos vóxeles en el nivel de detalle i. En este trabajo simplemente se genera el nivel de detalle i+1 a partir de los vóxeles en posiciones pares del nivel de detalle i (ver vóxeles grises en la Figura 1).

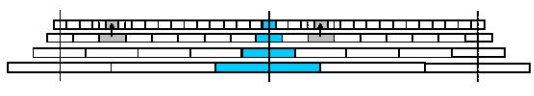

Figura 1: *Generación de la jerarquía multiresolución por bloques. Se puede observar en una representación 1D cómo interviene únicamente el píxel central del nivel d-1 para representar un píxel del nivel d que cubre exactamente 2 píxeles del nivel d-1. Adicionalmente en el área azul se muestra como los píxeles fronterizos se comparten entre los vecinos.*

4. Criterio de selección

Los criterios de selección considerados son los mismos utilizados en el trabajo de Carmona et al. [CF*11]: la distancia con respecto a un punto de interés, la distorsión de la representación multi-resolución en el dominio de la función de transferencia y la capacidad de memoria de textura.

El criterio basado en el punto de interés es calculado mediante la distancia de dicho punto al centro del bloque. Para darle mayor prioridad a los bloques de menor resolución se le suma la diagonal del bloque a dicha distancia. Debemos tener en cuenta que la diagonal es calculada en función al número de vóxeles del bloque.

La distorsión es determinada en función del error al utilizar un nivel de detalle más burdo con respecto a la data original de un bloque en el espacio CIELuv. Para combinar los criterios de la distancia a un punto de interés con el de la distorsión, hay que considerar que a mayor distorsión mayor prioridad de refinamiento y que a menor distancia al punto de interés también se requiere mayor prioridad. Por lo tanto la ecuación que describe ambos criterios puede escribirse como *(distancia+diagonal)/distorsión*.

Para determinar el nivel de detalle de cada bloque a visualizar, se utiliza una cola de prioridad. Inicialmente se insertan todos los bloques con su nivel de detalle más burdo, y ordenados por prioridad. Iterativamente, se toma el primero de la cola, y se refina (i.e. se reemplaza por su siguiente nivel de detalle). Luego el nodo refinado se reinserta en la cola con su nueva prioridad. El proceso se repite hasta alcanzar el tamaño de la memoria de textura, o hasta refinar un número determinado de bloques definido por el usuario. En el caso de detenerse por el criterio de número de bloques refinados, el refinamiento puede tomar varios fotogramas hasta converger a la selección deseada.

5. Cálculo de distorsión

Para realizar el cálculo de la distorsión en los bloques se requiere visitar los vóxeles del nivel de detalle más fino, y calcular el error de sus aproximaciones en los distintos niveles de detalle. Debido a que un volumen de gran tamaño puede contener miles de millones de vóxeles, esto requiere un tiempo considerable de cómputo. Para disminuir el tiempo de respuesta se introduce un algoritmo paralelo para hacer el cálculo de la distorsión. Usando como base a Ljung et al. [LLYM04], inicialmente la función de transferencia es convertida al espacio de color CIELuv. Posteriormente se procede a recorrer cada bloque visible para calcular su distorsión.

Además del algoritmo secuencial para calcular la distorsión, se implementaron dos algoritmos paralelos: uno utilizando el API de OpenMP y otro utilizando el API de CUDA. La distorsión es calculada como:

$$e(V, lod_i) = \frac{\sum_{v \in V} \left(\| f(s_0(v)) - f(s_i(v)) \| \right)}{n}$$

donde n es la cantidad de vóxeles del bloque en el nivel de detalle 0, f representa la función de transferencia en el espacio CIELuv y s_i es la interpolación tri-lineal del volumen en el nivel de detalle i.

OpenMP se basa en el modelo *fork-join*, paradigma que proviene de los sistemas Unix, donde una tarea muy pesada se divide en N hilos (fork) con menor peso, para luego "recolectar" sus resultados al final y unirlos en un sólo resultado (join). Para el cálculo de la distorsión, el algoritmo basado en el API de OpenMP es básicamente el siguiente:

```
#pragma omp parallel for num_threads(N)
For each bloque b del volumen original
      Si (bloques[b] es transparente)
            continuar;
      For each Nivel de detalle lod
            Bloques[b].distorsion[lod] =
            e(bloques[b], lod);
```

donde N es la cantidad de hilos utilizados. Cuando N=1 tenemos la versión secuencial del algoritmo. La lista *bloques* es la estructura de datos que contiene todo el volumen multi-resolución. El ciclo es ejecutado en N partes aplicando el esquema *fork-join* [URL01]. Cada parte es ejecutada por un procesador. Como cada bloque es totalmente independiente uno del otro, no hay que considerar problemas de memoria compartida ni secciones críticas. Por ende esta implementación no requirió de muchas adaptaciones al código secuencial.

El API de CUDA [NVI11] se constituye por un *host* (CPU) que se conecta a un *device* (GPU ó CPU). Este se encarga de realizar los cálculos que le asignó el *host* a través de los *kernels*. Un *kernel* es una función compilada, que en un inicio fue escrito en *C for CUDA*. Sólo puede ejecutarse un *kernel* a la vez. Una vez ejecutado, se instancian varios hilos (*threads)* de ejecución realizando las operaciones indicadas en el *kernel*. Se pueden crear miles de hilos en pocos ciclos de reloj, a diferencia del CPU. Los *threads* pueden ser agrupados en *grids* y *blocks*. Los *grids* pueden ser de una o dos dimensiones y los *blocks* de una, dos o tres dimensiones. La idea es compartir datos y sincronizar los hilos por conjuntos. En este trabajo nos referiremos a los bloques de CUDA como "*blocks*" y a los bloques de la jerarquía multi-resolución como "bloques".

El espacio de memoria del *host* y *device* es separado. El *host* maneja la memoria del *device*, como la asignación y liberación de memoria, la copia de datos entre el *host* y el *device* y entre *device* y *device*. El *host* puede hacer transferencia de datos a través de una memoria global que tiene el *device*. Los *blocks* tienen una memoria compartida y los *threads* tienen su propia memoria privada.

CUDA presenta distintas limitaciones que dependen de cada *hardware* gráfico. Las limitaciones vienen definidas en NVIDIA CUDA *Compute Capability* [NVI11]. A fines demostrativos supongamos el uso de la versión 1.0 del *capability* de CUDA (siendo la primera versión). Con esta versión se pueden crear a lo sumo 512 hilos de ejecución por *block*. La memoria local disponible para cada hilo es de 16KB, al igual que la memoria compartida.

Para el cálculo de la distorsión se requirió el uso adicional de cinco texturas: el bloque original (nivel de detalle 0), los otros tres niveles de detalle locales al bloque

y la función de transferencia. Para esto se utilizaron objetos de memoria de tipo *texture memory* que provee CUDA para el almacenamiento y el manejo de las texturas.

Una vez que se cargan las texturas en el GPU se inicia el *kernel* de CUDA para que se procese el bloque. La idea básica del algoritmo implementado es que cada hilo calcule el error de los tres niveles de detalle de un vóxel del bloque, posteriormente se suma el resultado de todos los hilos para obtener el error. Esto implica que para calcular el error de un bloque de 17^3 se requerirían un total 4913 hilos, en el caso de un bloque de 33^3 se requiere de 35937 hilos y para un bloque de 65^3 se necesitan 274625 hilos. Como la cantidad máxima soportada por *block* es de 512 se debe hacer una subdivisión del bloque.

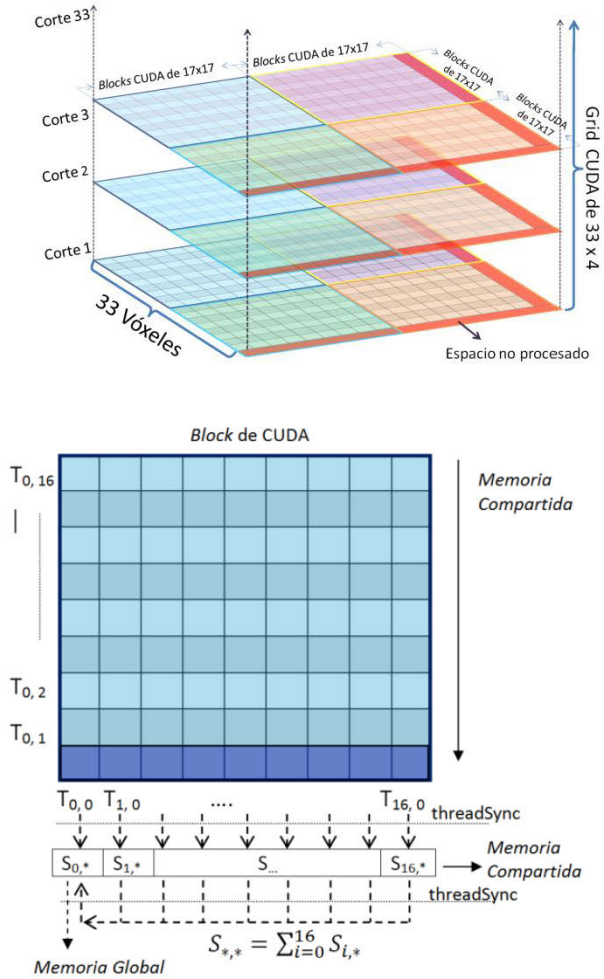

Figura 2: *En este ejemplo se puede observar cómo se subdividió un bloque de 33^3 vóxeles en un grid de 33x4, haciendo que cada block de CUDA tenga una dimensión de 17x17. Esto para distribuir de mejor forma el trabajo y no sobrepasar el límite de 512 hilos por block. Cada hilo almacena su resultado en una memoria compartida de tamaño NxN, donde N es la dimensión del block. Posteriormente los hilos $T_{i,0}$ suman su respectiva columna y lo almacenan en otra memoria compartida; por último se vuelve a sincronizar todo el block y el hilo $T_{0,0}$ suma los resultados de todas las columnas y lo almacena en un arreglo que se encuentra en la memoria global del GPU.*

En la Figura 2 se puede apreciar un ejemplo de cómo se forma el *grid* para un bloque de 33^3 muestras. El grid de CUDA será de 33x4, en donde 33 es el número de cortes del bloque, y 4 son las divisiones de un corte particular. Cada block del grid es un cuadrante de un corte de un bloque; esto es, 17x17 vóxeles (289 hilos). Al ejecutar un block, 289 hilos calculan la distorsión en un vóxel en sus distintos niveles de detalle. Una vez que cada hilo realiza este cálculo, los resultados se almacenan en una memoria compartida para que luego puedan sumarse. Primero los procesadores $T_{i,0}$ suman los resultados en su columna, y luego el procesador $T_{0,0}$ acumula las sumas parciales de dichas columnas. El resultado se almacena en un arreglo que se encuentra en memoria global. Para poder almacenar el error de cada block, el arreglo ubicado en la memoria global debe tener las dimensiones del *grid*, por ejemplo en la figura el arreglo global debe tener una dimensión de 33x4. Este arreglo es sumado posteriormente en el CPU cuando el kernel termina su ejecución debido a que CUDA no provee una sincronización a nivel de *grid*, solo es posible la sincronización de los *threads* de un *block*.

Hay que tener en cuenta que si los bloques del volumen son muy pequeños se desaprovecharía los recursos del GPU, pues la cantidad total de bloques generados para un volumen aumentaría y el nivel de paralelismo por bloque disminuiría. Esto hace que haya mayor cantidad de transferencia de datos al hardware gráfico y pocos cálculos por cada bloque. Debemos recordar que la tasa de transferencia hacia el GPU es limitada.

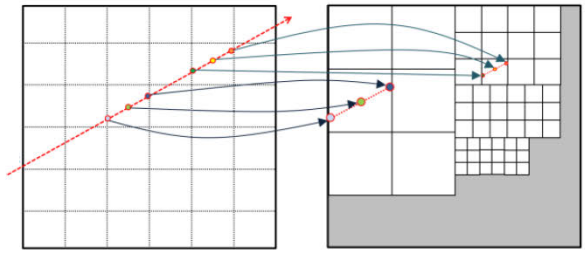

Figura 3: *Textura de índices y textura de atlas. Cada vóxel de la textura de índices tiene una tupla (Px, Py, Pz, Lod), representando la posición de inicio del bloque en el atlas, y su nivel de detalle Lod. Utilizando una representación 2D se puede observar la relación entre la textura de índice y el atlas en cada paso de un rayo por el volumen.*

6. Textura atlas

Utilizando el esquema propuesto por Lux y Ljung, se generó una textura atlas para el proceso de visualización. Este método consiste en tener los bloques que se van a mostrar almacenados en una sola textura e indexarlas en otra textura de índices (ver Figura 3). Al realizar ray casting acelerado por GPU, se despliegan las caras frontales de la textura de índices. Por cada fragmento generado de las caras frontales, se activa un programa de fragmento, en el que se genera un rayo que atraviesa la textura de índices. Mientras el rayo avanza por la textura de índices, estos índices son utilizados para muestrear el bloque respectivo en la textura de atlas. Así, la ecuación de composición volumétrica de va evaluando hasta que el rayo sale de la textura de índices.

7. Resultados

Debido a que se utiliza CUDA y OpenMP el requerimiento mínimo es utilizar una tarjeta gráfica NVIDIA GeForce 8 o superior y un CPU con múltiples procesadores. En total se utilizaron dos equipos diferentes para la realización de las pruebas. El equipo 1 consta de un Intel Core 2 Quad Q6600 de 2.40 HGz, 4GB de RAM, con una tarjeta GeForce GTX+9800 de 512MB de memoria y 128 núcleos de CUDA. El equipo 2 tiene un Intel Core i3 de 3.07GHz con 4 GB de RAM, con una tarjeta GeForce GTX 470 de 1280 MB de memoria y 448 núcleos de CUDA.

El sistema fue compilado en el entorno de desarrollo de Microsoft Visual Studio Professional 2008 en el lenguaje Visual C++. Se utilizó la librería Qt4 para la interfaz gráfica. Para la programación de GPU de propósito general se utilizó la librería CUDA Toolkit 4.0 y para la programación paralela en CPU se utilizó OpenMP 1.0.

Se consideraron tres volúmenes en las pruebas: dos de ellos tomados del proyecto del Humano Visible [URL02] y uno de una tomografía computarizada. Las imágenes mostradas a continuación fueron generadas con el algoritmo de ray casting de una pasada, con un viewport de 1024x768 y utilizando un atlas de 256MB de textura. En la Figura 4 se muestran las especificaciones de cada uno de estos volúmenes.

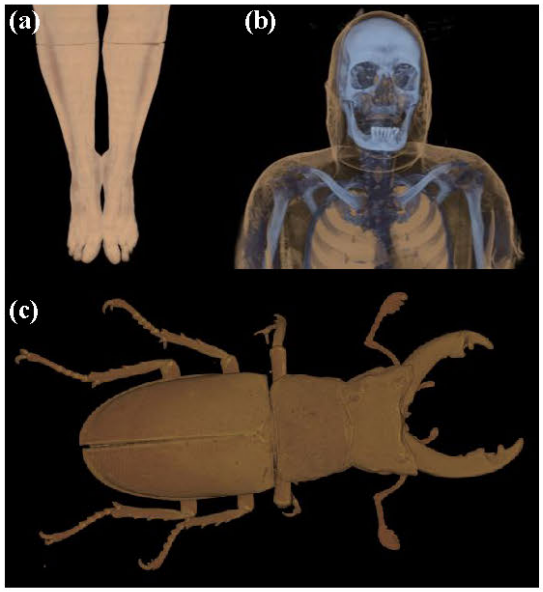

Figura 4: *(a) Mujer Visible en fotos grises 8bits, 890x890x2000 (1.47GB), (b) CT de la Mujer Visible 16 bits, 512x512x1734 (867MB), (C) CT de un Escarabajo 16 bits, 832x832x494 (652MB).*

En la fase de pruebas se utilizaron tres tamaños para la jerarquía basada en bloques y tres tamaños de atlas. La memoria requerida para almacenar todo el volumen multiresolución es dependiente al tamaño de los bloques. Mientras el tamaño de los bloques es más pequeño, mayor es la redundancia por compartir un vóxel fronterizo, y mayor es espacio requerido para almacenar el volumen multiresolución. Igualmente, el tiempo para generar los niveles de detalle aumenta mientras el tamaño de los bloques decrece.

7.1 Cálculo de la distorsión

Se realizaron una serie de pruebas de rendimiento utilizando 1 CPU, 4 CPUs y CUDA (ver Tabla 1). En el caso de CUDA se usó un block de tamaño 17x17 para todas las pruebas. Este tamaño fue el utilizado porque cumple con las limitantes de la cantidad de hilos permitidos en un block; en el caso de tener bloques de mayor tamaño que el block, los bloques se subdividen como se muestra en la Figura 2.

Volumen		Equipo 1			Equipo 2		
Versión		1 proc	4 proc	GPU	1 proc	4 proc	GPU
A	17^3	379,1	106,6	240,4	290,4	122,2	163,1
	33^3	341,7	94,0	55,5	259,2	108,3	24,3
	65^3	329,4	90,4	28,9	248,1	103,7	10,6
B	17^3	88,1	24,8	49,7	65,0	29,2	26,6
	33^3	86,4	24,0	12,4	63,3	28,0	5,8
	65^3	89,4	24,5	6,4	65,5	28,2	2,6
C	17^3	80,1	22,1	50,2	58,2	25,6	26,8
	33^3	75,4	19,3	12,0	54,8	24,1	5,7
	65^3	72,8	18,5	6,0	52,1	23,1	2,4

Tabla 1: *Tiempo en segundos para el cálculo de la distorsión de todos los bloques de cada volumen en 3 tamaños de bloques distintos (17^3, 33^3, 65^3).*

Observe que al utilizar un CPU, el tiempo de respuesta es mayor en todos los casos. En el caso de utilizar bloques de 17^3 muestras, utilizar 4 procesadores brinda mejores resultados que utilizar GPU. En el equipo 1 en particular, el tiempo de respuesta de 4 CPUs es alrededor del 50% del tiempo de GPU. En el caso de utilizar bloques de mayor tamaño (33^3 y 65^3) los resultados en CUDA toman hasta 10 veces menos tiempo que utilizando 4 CPUs en el equipo 2, y entre 2 y 3 veces menos en el equipo 1. Esto se debe a que es menor la cantidad total de bloques que se envían al GPU y cada bloque tiene mayor densidad de datos a procesar por cada llamada al kernel de CUDA. Con esto se aprovecha más el poder de cómputo ya que se puede distribuir más la carga de trabajo entre todos los procesadores del GPU.

Finalmente, al comparar la versión secuencial de 1 CPUs con la versión paralela de 4 CPUs encontramos una aceleración que oscila entre 3.6 y 3.9 con el equipo 1, y entre 2.2 y 2.9 con el equipo 2. Note además que el tiempo de respuesta varía ligeramente a medida que aumenta el tamaño del bloque con versión de 1 o 4 CPUs, pero disminuye drásticamente con la versión GPU.

7.2 Rendering

Como una primera prueba, se evaluó la diferencia en calidad entre utilizar bloques de 17^3 y bloques de 65^3 manteniendo fija la capacidad del atlas. Notamos que con bloques de 17^3 se logra un mejor nivel de detalle en las áreas importantes del volumen, puesto que hay un mejor uso de la memoria por contar con una partición más fina del mismo.

En la Figura 5b se puede notar más detalle que en la Figura 5a por utilizar bloques de tamaño 17^3.

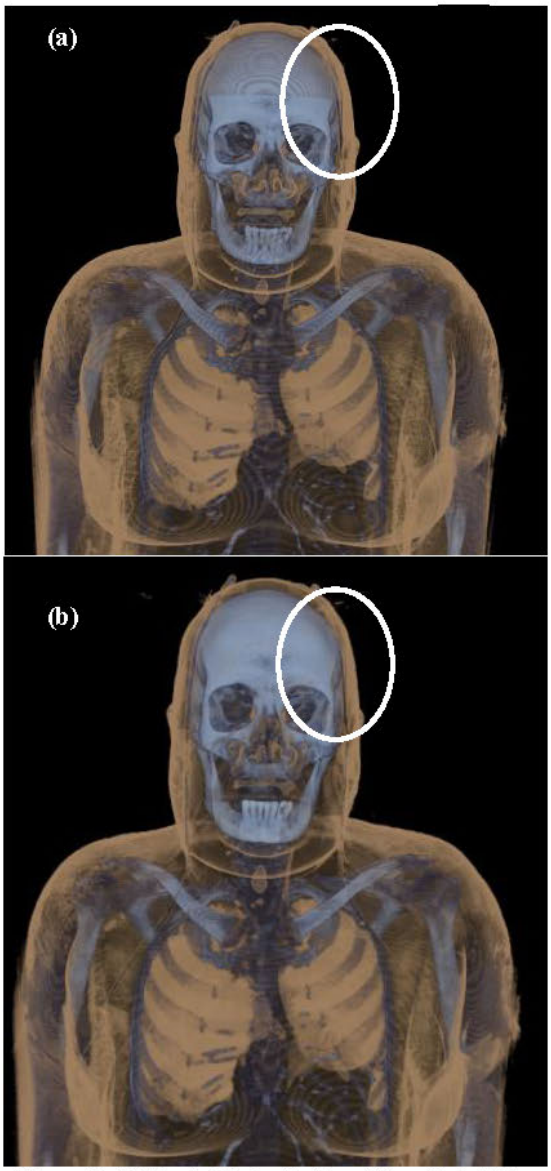

Figura 5: *CT de la Mujer Visible utilizando (a) bloques de 65^3 muestras y (b) bloques de 17^3 muestras.*

Figura	Fig. 6a		Fig. 6b		Fig. 6c	
Versión	Actual	Prev.	Actual	Prev.	Actual	Prev.
C.P.S.	9,47	3,74	4,61	5,44	6,43	3,32

Tabla 2: *Cuadros por segundo (Hz) generando las correspondientes imágenes para los sistemas prototipos previo y nuevo.*

En la siguiente prueba (ver Tabla 2) se compara la rata de cuadros por segundo del sistema actual (basado en bloques) con el sistema previo (basado en octree) para 3 ángulos distintos del volumen b (ver Figura 6). Cabe destacar que en esta prueba se está comparando únicamente el tiempo de rendering; así, la función de transferencia se mantiene constante entre cuadros para no requerir el cómputo de la distorsión multi-resolución. El tamaño de los bloques fue fijado en 17^3, y el de los bricks en 16^3. Podemos notar que el rendimiento depende significativamente del ángulo de visualización. Por lo general, el sistema actual tiene mejor tiempo de respuesta que el sistema previo. Esto se debe a que el sistema actual es de una pasada, y los fragmentos se generan una sola vez por cada píxel de la imagen. Hay casos particulares en donde el sistema previo tiene un mejor rendimiento. Notamos que estos casos tienen que ver con la cantidad de fragmentos "transparentes" generados por el sistema actual a partir de las caras frontales de la caja contenedora del volumen. Para la Figura 6b, el sistema actual desplegará las caras frontales de dicha caja abarcando prácticamente todo el viewport, y muchos de los fragmentos generados serán transparentes, desperdiciando así tiempo de rasterización. Para la misma imagen, el sistema previo va desplegar bricks individuales que se ajustan más a la silueta del volumen, desperdiciando así menos fragmentos y tiempo de rasterización. Para los otros ángulos de visualización (Figuras 6a y 6c), las caras frontales del volumen van a generar menos fragmentos transparentes en el sistema actual que en la Figura 6b, aprovechando mejor el paralelismo del GPU.

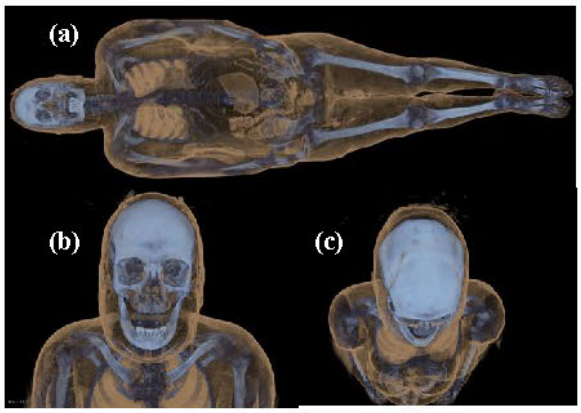

Figura 6: *Visualización de la tomografía computarizada de la mujer visible con 3 ángulos de visualización distintos. viewport de 1024x768 en cada caso.*

Adicionalmente se puede ver en la Figura 7a una superficie más uniforme y con mayor detalle que en la Figura 7b, debido a que se aprovecha mejor el espacio en la textura atlas. Esto se debe a que al refinar un bloque se aumenta el detalle en un área más pequeña del volumen que cuando se utiliza el octree. En el octree, al refinar un brick, este se reemplaza por sus 8 bricks hijos, pudiéndose refinar algunas áreas innecesariamente, haciendo un uso menos eficiente de la memoria de textura. Note además que en el zoom realizado en las Figuras 7d existen más artefactos visuales debido al ajuste de los rayos a las fronteras de cada brick.

8. Conclusiones y trabajos a futuro

En este trabajo se desarrolló un sistema prototipo para visualizar volúmenes de gran tamaño aplicando una jerar-

quía multi-resolución basada en bloques con un algoritmo de ray casting de una sola pasada. Esto fue posible gracias a la utilización de una textura atlas. En esta son almacenados los bloques que fueron previamente seleccionados para la visualización e indexados en otra textura de índices.

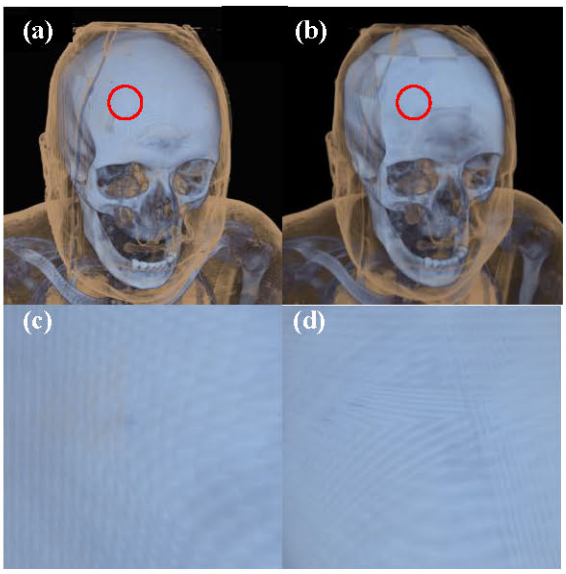

Figura 7: *Comparando calidad de imagen entre el sistema actual (a) y el sistema previo (b). Las imágenes (c) y (d) representan un acercamiento en el área de los círculos rojos en (a) y (b) respectivamente. En (b,d) se observan más artefactos que en (a,c) debido a la representación de ciertas áreas con menos detalle (b) y al clipping de los rayos a las fronteras de los bricks (d).*

Se pudo llegar a la conclusión de que el tamaño de los bloques y atlas son importantes en el tiempo de respuesta. Cuando los bloques son de menor tamaño, el espacio de la textura atlas es utilizado eficientemente debido a que la fragmentación origina que los espacios pequeños puedan ser utilizados, dando como resultado que la distorsión global del volumen disminuya. La desventaja es que se genera mayor cantidad de bloques y por lo tanto la cola de prioridad de refinamiento aumenta; similarmente, si se utiliza el GPU para el cálculo de distorsión se requieren mayor cantidad bloques a enviar disminuyendo el tiempo de respuesta. Como consecuencia, para bloques pequeños, el CPU obtiene un mejor rendimiento que el GPU en el cálculo de la distorsión.

En cuanto al cálculo de la distorsión encontramos que para una partición más fina del volumen (bloques de 17^3 muestras) la versión del GPU toma más tiempo que la versión paralela del CPU. Esto es debido a que se aprovecha poco el paralelismo del GPU debido a la transferencia de bloques pequeños de datos y exceso de llamadas a Kernels. Para particiones menos finas del volumen (33^3 y 65^3) el tiempo de respuesta del GPU es hasta 10 veces más rápido mejor que la versión paralela que corre en los CPUs.

En cuanto a la calidad de rendering, se comparó el sistema prototipo actual con el sistema prototipo previo realizado por Carmona et al. [CF*11]. En la prueba realizada se observó que una jerarquía por bloques ofrece un resultado de mejor calidad utilizando la misma cantidad de memoria, y tamaño similar en los bricks y bloques del nivel de detalle original. La razón principal es que la subdivisión del volumen en un octree no permite un refinamiento de un área específica dentro del brick, obligando que ciertas áreas no deseadas se refinen, haciendo un uso menos eficiente de la memoria. Adicionalmente se mostró que con el sistema nuevo se reduce el artefacto visual originado por las múltiples pasadas del sistema previo.

Ambos sistemas también fueron comparados en cuanto al tiempo de respuesta. El sistema nuevo es por lo general entre 2 y 3 veces mejor que el sistema previo; sin embargo, para ciertos ángulo de visualización, el sistema nuevo no logra superar al previo. Encontramos que en estos ángulos particulares, el sistema nuevo genera mayor cantidad de fragmentos transparentes, que ocupan tiempo importante de rasterización.

Aún cuando muchos autores han propuesto diversos métodos para la visualización de volúmenes de gran tamaño, siempre están presentes los artefactos visuales, producto de asignar distintos niveles de detalle a áreas adyacentes. A continuación se presentar algunas posibles mejoras sobre el sistema implementado, ya sea para reducir estos artefactos, o para acelerar el tiempo de respuesta:

- Estudiar el problema de fragmentación de la textura de atlas, lo cual sucede cuando se requiere actualizar continuamente los niveles de detalle a desplegar, debido a la interacción del usuario con el volumen y con la función de transferencia.
- Aplicar la interpolación entre niveles de detalle consecutivos para la reducción de artefactos entre bloques adyacentes [CF*09].
- Utilizar el muestreo adaptativo del atlas en función del nivel de detalle de los bricks, o de la opacidad acumulada en los rayos [CF*11].
- Optimizar el algoritmo implementado en el GPU para el cálculo de la distorsión. Para ello se sugiere agrupar varios bloques en un solo bloque para maximizar el uso de los núcleos del GPU. La solución implementada sólo permite el cálculo de un bloque a la vez.

Referencias

[BNS01] Boada I., Navazo I., Scopigno R.: Multiresolution volume visualization with a texture-based octree. *The Visual Computer* (2001), pp. 185-197.

[LDH*99] Lamar E., Duchaineau M., Hamann B., Joy K.: Multiresolution Techniques for Interactive Texture-based Volume Visualization. *Visualization* '99, California-USA (1999), pp. 355-361.

[LWP*06] Ljung P., Winskog C., Persson A., Lundstrom C., Ynnerman A.: Full Body Virtual Autopsies using a State-of-the-art Volume Rendering Pipeline. *IEEE Transactions on Visualization and Computer Graphics*, vol. 12 , 5 (Oct. 2006), pp. 869-876.

[CF*11] CARMONA R., FROEHLICH B. Error-Controlled Real-Time Cut Updates for Multi-Resolution Volume Rendering, *Elsevier Computers and Graphics* (Jan. 2011), pp. 931-944.

[LB*09] LUX C., FROHLICH, B.: GPU-Based Ray Casting of Multiple Multi-resolution Volume Datasets. *ISVC '09 Proceedings of the 5th International Symposium on Advances in Visual Computing* (2009), Part II, pp. 104-116.

[WGS07] WANG C., GARCIA A., SHEN H.-W.: Interactive Level-of-Detail Selection Using Image-Based Quality Metric for Large Volume Visualization. *IEEE Transactions on Visualization and Computer Graphics* (2007), pp. 122-134.

[GHY98] GRZESZCZUK R., HENN C., YAGEL R.: Advanced Geometric Techniques for Ray Casting Volumes. *ACM Siggraph* '98 (Julio 1998) notas de curso.

[LLYM04] LJUNG P., LUNDSTROM C., YNNERMAN A., MUSETH K.: Transfer Function Based Adaptive Decompression for Volume Rendering of Large Medical Data Sets. *IEEE Symposium on Volume Visualization and Graphics* (2004), pp. 25-32.

[URL01] OpenMP®: The OpenMP® API specification for parallel programming (2010). [Online]. http://openmp.org/.

[NVI11] NVidia®: NVIDIA CUDA C Programming Guide V4.0 (2011). *NVIDIA CUDA™*.

[URL02] The National Library of Medicine's. (1987) *Visible Human Project®*. [Online]. http://www.nlm.nih.gov/research/visible/visible_human.html.

[CF*09] CARMONA R., RODRIGUEZ G., FROEHLICH B. Reducing Artifacts Between Adjacent Bricks in Multi-resolution Volume Rendering. *Lecture Notes in Computer Science: Advances in Visual Computing* (2009). Vol. 5875. pp. 644-655.