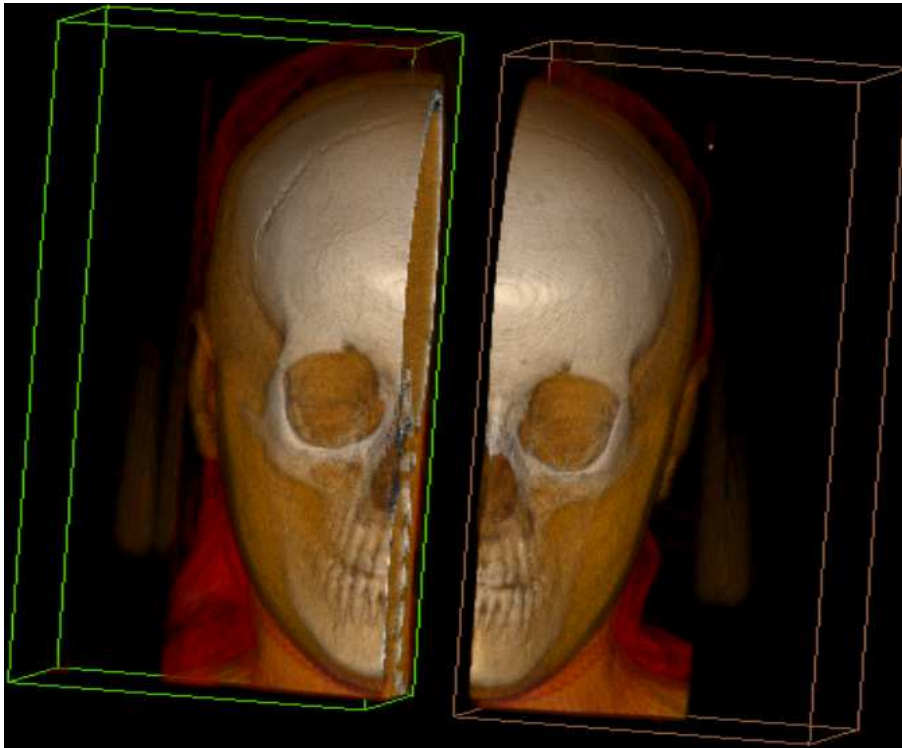
Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemes Informàtics
Informàtica Gràfica
Programa de Doctorat de Software

Visualización de grandes volúmenes vía bricking y texturas 3D

Héctor Yela Reneses

Tutora: Isabel Navazo

-



Barcelona, Junio 2007

Resumen

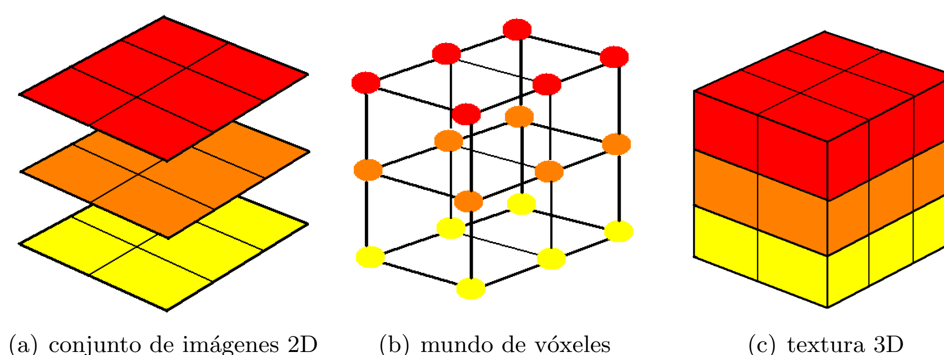
Este trabajo estudia la visualización de grandes volúmenes cuando se combinan las técnicas de renderizado mediante texturas 3D y de bricking. Los datos recibidos por los aparatos médicos son cada día mayores y nos encontramos con problemas a la hora de intentar visualizarlos interactivamente. A veces, la cantidad de memoria requerida por el volumen sobrepasa la ofrecida por la tarjeta gráfica y esto impide su visualización. De entre las distintas técnicas de visualización de volumen y de tratamiento de volúmenes especialmente grandes, nosotros decidimos implementar las texturas 3D y la técnica de bricking, con tal de evaluar el rendimiento en términos de calidad y velocidad de la combinación de ambas técnicas. Se escogen las texturas 3D porque es la técnica que más rendimiento extrae de la tarjeta gráfica, y se escoge bricking porque la visualización que se obtiene es la misma que se obtendría si el volumen cupiese en memoria. Dado que trabajamos con datos médicos queremos que las visualizaciones sean exactas. Los resultados obtenidos son esperanzadores e invitan a seguir trabajando por este camino, ya sea combinando nuestra solución con nuevas técnicas, ya sea depurando las existentes.

1. Introducción

La visualización de volumen tiene en las aplicaciones médicas una de sus utilidades más importantes. Los datos recibidos por estas aplicaciones son cada vez más grandes, debido al incremento en la sensibilidad de los aparatos de medición, que generan **volúmenes con mayor resolución** y con más detalles por unidad de volumen, vóxel. Estos volúmenes no siempre caben en la memoria que nos ofrecen las tarjetas gráficas y, por lo tanto, no permiten su **visualización interactiva**. La solución más intuitiva es el **bricking**, es decir, dividir el volumen en otros más pequeños que sí quepan en la GPU. En este informe se presenta una implementación y evolución de esta técnica en combinación con la visualización de volumen vía **texturas 3D**.

En la mayoría de las ocasiones, los volúmenes utilizados para la visualización médica se obtienen de disco, donde están almacenados como un conjunto de imágenes 2D, figura 1a. A partir de estas se construye un mundo de vóxeles, figura 1b, (generalización 3D del término píxel) residente en memoria RAM.

Esta estructura puede ser visualizada mediante distintas técnicas (texturas 3D, Splatting, Raycasting,...). En el caso de la técnica de texturas 3D, y de una manera similar con Raycasting, es necesario construir a partir del mundo de vóxeles una estructura que la tarjeta gráfica pueda reconocer y pintar. Esta estructura es la textura 3D, figura 1c, y una vez enviada a la tarjeta gráfica, esta la almacena en memoria interna de la GPU. Normalmente, el tamaño de estas memorias internas oscila entre los 128MB y los 512MB.



(a) conjunto de imágenes 2D (b) mundo de vóxeles (c) textura 3D

Figura 1: Estructuras que participan en el proceso. Dimensión común: 2x3x3.

Teniendo en cuenta que la resolución de un mundo de vóxeles suele ser de 512 x 512xDimZ, donde DimZ oscila entre 256 y 1024 dependiendo del número de imágenes médicas del modelo, y que en cada vóxel se guardan 1 ó 2 bytes de información referente al valor de propiedad devuelto por el escáner para dicho vóxel, un modelo con 300 imágenes (150MB) ya no podría ser visualizado por una tarjeta normal, menos aún un modelo como Visible Human, formado por 1871 imágenes axiales, de 2048 x 1216 píxeles

de resolución, que ocupa en total 14 GB, un tamaño que no cabe ni siquiera en memora RAM, lo que requiere el uso de otro tipo de técnicas, Out-Of-Core, para ser visualizado.

Limitando nuestro propósito a modelos que sí caben en memoria RAM, la técnica para poder mandarlos a la GPU y así visualizarlos es el bricking. Basado en la filosofía del "divide y vencerás", se trata de subdividir el mundo de vóxeles en bricks de menores dimensiónes, de manera que, a la hora de mandar a visualizar el volumen, en vez de tener una sola textura 3D correspondiente al mundo de vóxeles,de tamaño superior al que ofrece la GPU, tenemos tantas texturas 3D como bricks obtenidos a partir de la subdivisión, las cuales caben individualmente en la memoria interna de la GPU, porque nosotros hemos escogido las dimensiones apropiadas para este fin.Los bricks son almacenados en la memoria principal, y mandados a pintar en orden back-to-front, obteniendo finalmente la imagen deseada, que no es otra que la del volumen global.

El bricking no reduce la cantidad de memoria necesaria para representar los datos del volumen original, no hay compresión ninguna. Cada brick, o sea la textura 3d pertinente, tiene que ser enviado a la memoria local de la GPU antes de ser renderizado.El rendimiento del bricking está supeditado a la velocidad de transferencia entre la memoria principal y la memoria local a la GPU. Se puede ver este proceso como una jerarquía de cachés,figura 2, es decir, inicialmente se tiene el modelo en disco, si cabe en memoria principal se envía, si no hay que aplicar métodos Out-Of-Core. Una vez está en memoria principal se manda a memoria gráfica, si cabe, perfecto (visualización directa), en caso contrario se aplica bricking.

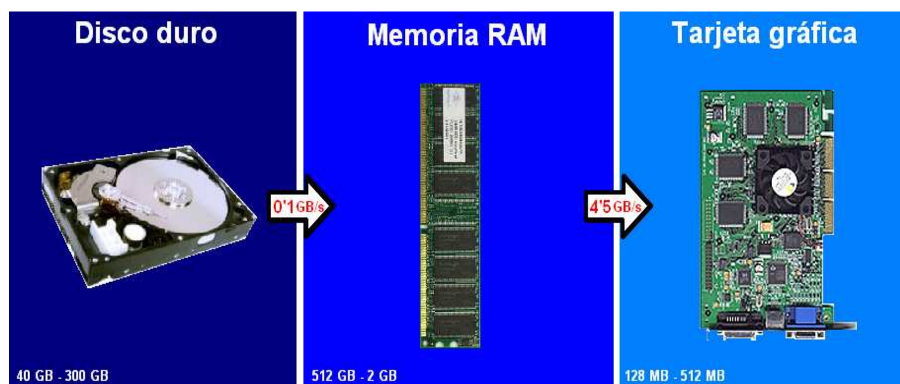


Figura 2: Esquema del flujo de datos entre los distintos dispositivos de almacenamiento

Para finalizar esta introducción explicar por qué se ha elegido bricking como método para tratar con volúmenes grandes y por qué texturas 3D como sistema de visualización. En las aplicaciones médicas se necesita la máxima precisión, ya que el mínimo detalle puede ser importante para el profesional a la hora de sacar conclusiones de lo

que está viendo. Por lo tanto, hay que descartar procedimientos que impliquen pérdida de precisión o detalle, tales como la compresión con pérdida. Otro factor determinante en la elección del bricking es la sencillez de la idea y la menor complejidad a la hora de implementarlo en comparación con otras técnicas existentes. Se escogió texturas 3D porque ya se tenía una implementación que permitía visualizar volúmenes que sí cabían en GPU, y ofrecía por un número menor de cambios en el código, un rendimiento y unos resultados mejores en comparación con una implementación nueva de Raycasting o de Splatting.

El informe está organizado en una breve *Introducción*, seguida de la sección *Trabajos relacionados*, donde se comentan y referencian los métodos más utilizados en la visualización de volumen y las técnicas más estandarizadas en lo referente a la gestión de grandes volúmenes. Después introducimos la *Visualización de volumen vía texturas 3D* y los algoritmos utilizados en nuestra solución bajo el epígrafe *Bricking*. Se presentan varios experimentos realizados en *Resultados* y, finalizamos con un análisis de los resultados obtenidos y una enumeración de las distintas opciones a estudiar en *Conclusiones y trabajo futuro*.

2. Trabajos relacionados

Hay muchos y muy buenos tutoriales sobre visualización de volumen[1], donde se introducen las técnicas más utilizadas, los métodos de aceleración más conocidos y se entra en detalle sobre muchos aspectos relacionados, como algoritmos de iluminación, procesos de segmentación de los datos y funciones de trasferencia. La mayor parte de la bibliografía y de los conocimientos necesarios para este proyecto han salido de ellos.

Existen varias técnicas a escoger a la hora de visualizar volúmenes. Nosotros utilizaremos texturas 3D por los motivos anteriormente citados y por ser la técnica que saca mayor rendimiento de la tarjeta gráfica, pero podríamos haber utilizado otros.

Entre estas otras técnicas, la más popular es el **Ray casting**, un algoritmo que trabaja en espacio imagen, la idea básica del cual consiste en evaluar directamente la integral de rendering a traves de rayos lanzados desde la cámara. Para cada pixel de la imagen lanzamos un rayo que atraviesa el volumen de adelante hacia atrás, de manera que va componiendo el color de dicho píxel conforme va progresando a través del mundo de vóxeles (la estructura que representa nuestro volumen). Es el método más importante para visualizaciones a partir de la CPU, se ha utilizado durante mucho tiempo y se han desarrollado muchos métodos de aceleración. Tambien hay implementaciones para GPU[5, 9] y junto con las texturas 3D son las técnicas más utilizadas en visualización de volumen.

Otra técnica bastante conocida es el **Splatting**[7], que antepone la velocidad de renderizado a la calidad de este. Consiste en proyectar cada elemento del volumen, cada vóxel, en el plano imagen, como si fueran bolas de nieve que se estrellan contra una pared. Es un algoritmo espacio objeto y la dirección en la que se recorre el mundo de vóxeles viene determinada por la dirección de visión.

Además del bricking, existen otros métodos que tratan de resolver el problema de los volúmenes que no caben en memoria gráfica. Ya hemos comentado que en el caso que el volumen tampoco quepa en memoria principal, éstos derivan de las técnicas Out-Of-Core, que nosotros pasaremos por alto para centrarnos en los que caben en memoria RAM.

El primero que presentaremos es el de **Multiresolución**[6]. La idea consiste en dibujar el modelo con menor o mayor resolución dependiendo de dónde se encuentre el observador. En caso de estar alejados, una simplificación del modelo bastará, en cambio necesitaremos mayor detalle si estamos haciendo un zoom de alguna de las zonas. Usamos un octree para simplificar/detallar las zonas del volumen de una manera eficiente y, así, navegar en tiempo real. Uno de los inconvenientes de este proceso es el tratamiento que necesitan las fronteras entre zonas de diferente resolución, ya que aparecen errores cuando se hace el blending de estas.

Otro enfoque es el proporcionado por las técnicas de **compresión de texturas** que ofrecen las tarjetas gráficas. Como punto positivo está el hecho de que es trasparente al código, es decir, el programador ha de modificarlo muy poco para activarlas y utilizarlas. Como negativo encontramos que están pensadas para datos RGB(A), que aparecen artifacts cuando los datos tienes gradientes pronunciados y que los ratios de compresión son moderados y con pérdida (y en medicina no nos lo podemos permitir).

La compresión vía **Wavelets**[8, 2] permite conservar las zonas con mayor detalle a la vez que ofrece ratios de compresión considerables. La técnica consiste en aplicar una transformada, conocida como "transformada Wavelet", sobre los datos del volumen. De esta manera, podemos escoger el nivel de detalle con el que visualizar el modelo, se comprime y, todo esto, de manera eficiente. Combinar esta técnica con la programación de tarjetas gráficas, puede ser la solución definitiva al problema de los volúmenes grandes.

Por último, mencionar las técnicas de **Packing**[4]. En un proceso previo se agrupan en texturas más pequeñas aquellas zonas que comparten valores de densidad similares y proximidad espacial. Así se reduce el tamaño de la textura del volumen a cambio de insertar un coste de traducción intermedio en la fase de acceso a texturas.

Resaltar que no hay ninguna técnica que predomine o que se haya impuesto como solución estándar al problema de los grandes volúmenes. Es un tema abierto y en el que, dependiendo de la aplicación que estemos buscando, en nuestro caso aplicaciones de visualización médica interactiva con posibilidad de interacción, podemos plantear distintas combinaciones de técnicas.

Además, hemos encontrado diferentes aproximaciones al Bricking[3, 10] que, si bien no coincidian con nuestra solución, nos han permitido corroborar detalles e inspirarnos a la hora de implementar nuestra técnica.

3. Introducción a la visualización vía texturas 3D

Supongamos que tenemos un volumen, la estructura que le representa en memoria es un mundo de vóxeles, una malla tridimensional donde podemos acceder al valor de cada vóxel de manera independiente. En nuestro caso particular, los valores almacenados en los vóxeles son valores de propiedad provinientes de captaciones producidas por aparatos médicos. Creamos una textura 3D a partir de nuestro mundo de vóxeles, y la rellenamos con valores calculados a partir de él, ya sean colores RGB, ya sea, como en nuestro caso, valores reales.

No existe un método que directamente permita visualizar las texturas, sino que se necesitan primitivas intermedias (usualmente polígonos) ubicadas en el interior del volumen que corresponde a la textura que se colorean de acuerdo con los elementos de la textura que intersectan. Estas operaciones pueden realizarse eficientemente utilizando programas (shaders) que se ejecutan en las tarjetas gráficas.

A partir de la caja contenedora del volumen y de la cámara que estamos utilizando, calculamos tantos rectángulos perpendiculares a la dirección de visión, incluidos dentro de esta caja contenedora, como queramos. Los rectángulos sí que son primitivas gráficas por lo tanto, sí podemos enviarlos a pintar y se les conoce como planos proxy, figura 3 .

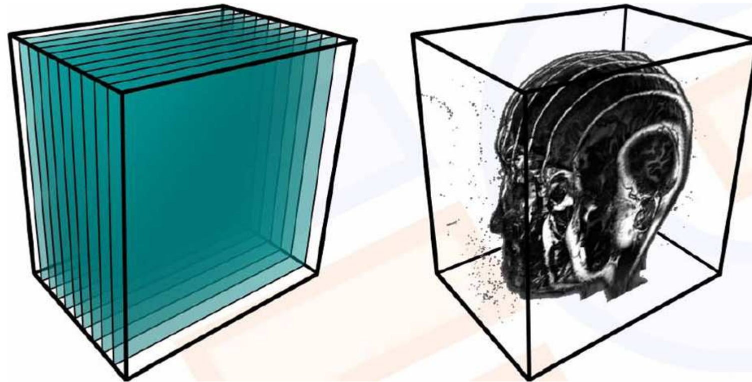


Figura 3: Planos proxy y resultado de la visualización con pocos planos

En estos rectángulos se mapea la textura 3D, de manera que al pintar todos los planos de atrás-hacia-adelante, recortarlos según la caja y aplicar blending, se compone la imagen del volumen almacenado en la textura 3D, figura 4. Como se puede deducir, a mayor cantidad de planos, mayor calidad en la imagen resultante y menor rendimiento del visualizador.

El aspecto positivo de este método es que aprovecha la interpolación trilineal que ofrece la tarjeta gráfica a la hora de mapear en los polígonos la textura 3D, obteniendo buena calidad a un precio no prohibitivo. Como puntos negativos, se observa que,

cuando crece el número de planos que hacen de geometría proxy, el rendimiento decae en igual medida y que la distancia de muestreado entre planos es constante, no como en el Raycasting.

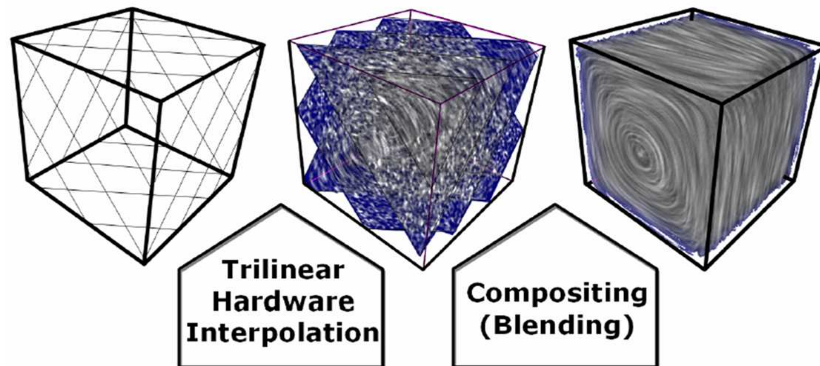


Figura 4: Rectángulos proxy recortados, cálculo de los valores mediante interpolación trilineal y blending.

Referente a los valores guardados en la textura 3D, nosotros guardamos un valor alpha, un real, que utilizamos a modo de índice de una tabla paleta que asigna un color y una opacidad de función de este valor alpha. De esta manera, para un mismo volumen ya cargado en memoria, basta con cambiar interactivamente la paleta para que cambien los colores y la imagen final, lo que es realmente útil en aplicaciones médicas, por ejemplo para hacer transparente un hueso, ya que los téxeles de este material tendrán valores de propiedad parecidos.

4. Bricking

La visualización vía texturas 3D de un volumen subdividido en bricks no dista demasiado de la de un volumen normal, hay que adaptar los algoritmos generales a la utilización del bricking, pero la estructura y la idea coinciden en ambos casos. Los cambios más importantes los encontramos a la hora de generar las texturas 3D que visualizaremos y en el algoritmo de renderizado, donde añadimos una etapa de clipping por cada brick.

Vamos a presuponer que tenemos el mundo de vóxeles cargado en memoria RAM, es decir, que ya hemos obtenido de disco una captación de los aparato médicos y la hemos transformado en una malla tridimensional, de la cual conocemos sus dimensiones. Como la textura 3D que obtendríamos no cabría en memoria interna de la tarjeta gráfica, tendremos que hacer bricking.

4.1. Generación de los bricks

Recibimos como entrada la dimensión que queremos que tengan los bricks y el mundo de vóxeles.Al igual que en la visualización estándar de texturas 3D teníamos que crear una textura 3D a partir del mundo de vóxeles, en el bricking crearemos N texturas 3D a las que llamaremos bricks. En nuestro caso, al trabajar con una tarjeta gráfica de NVidia, se permite cualquier tamaño de brick. En el supuesto de trabajar con ATI, nos veríamos limitados a dimensiones potencia de 2, ya que sus tarjetas no permiten trabajar con texturas que no cumplan dicha condición.

No basta con dividir el mundo de vóxeles en pequeños volúmenes disjuntos de la dimensión recibida y crear el brick, es más complejo. En primer lugar, calculamos si todos los bricks pueden tener las dimensiones introducidas. En caso negativo, hacemos que los de las fronteras más lejanas al origen tengan las dimensiones residuo, es decir, que el número de vóxeles en cada dirección sea el resultado de aplicar la operación residuo a la dimensión global por el tamaño deseado.

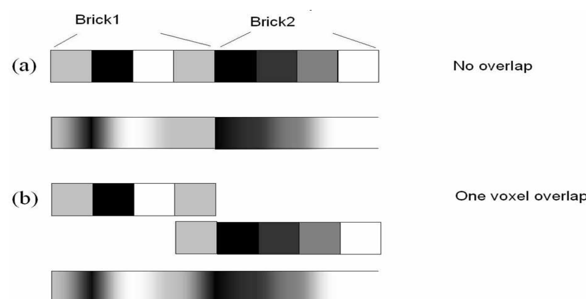


Figura 5: Visualización con bricks solapados y sin solapar

No basta con que los bricks sean disjuntos porque en ese caso, al visualizar los vecinos no se realizaría bien la interpolación y encontraríamos errores, figura 5. Si la visualización es sin iluminación, basta con incrementar en uno los vóxeles en cada dirección. En el caso contrario, es necesario incrementar en dos los vóxeles utilizados como offset. Esta diferencia se debe al cálculo del gradiente y a que sea necesario acceder al vóxel vecino para calcularlo. Si en la fórmula de calcular nuestro gradiente accedemos a vóxeles que distan N del tratado, el número de vóxeles a solapar será de N + 1.

Por lo tanto, los datos que necesita el brick para ser creado correctamente son los siguientes:

- Vóxel origen y dimensiones del brick.
- Número de vóxeles a solapar.
- Caja contenedora del brick completo, brick con offset.
- Caja contenedora de recorte, brick sin offset.

Una vez conocemos todos los detalles, figura 6, creamos una textura 3D (brick) a partir del mundo de vóxeles, que contenga los datos correspondientes a las dimensiones globales del brick (con offset). Enviamos a la GPU esta textura y guardamos su identificador para hacer referencia más tarde.

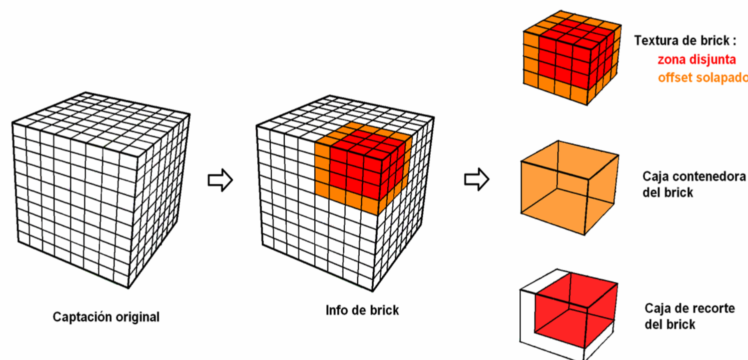
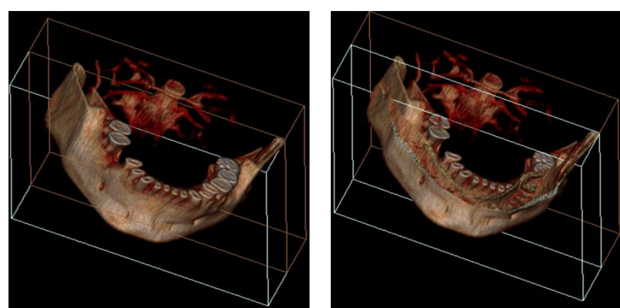


Figura 6: Obtención de un brick a partir del mundo de vóxeles

4.2. Visualización de los bricks

Llega el momento de visualizar el volumen. Como vamos a pintar los bricks en secuencia, primero se ordenan, ya que interesa que los más cercanos al observador sean los últimos en pintarse, en caso contrario los más alejados "machacarían"los píxeles de los primeros, figura 7. La medida de distancia utilizada es la del observador al centro del brick.



(a) Bricks ordenados (b) Bricks desordenados

Figura 7: Mandar a pintar los bricks mal ordenados genera errores

Una vez organizados, se pinta cada brick. La idea es que, con independencia del brick, se pinten siempre los mismos planos. Éstos, tal y como se ha comentado antes, son perpendiculares a la dirección de visión y tienen el tamaño especificado por la diagonal de la caja contenedora del volumen, de manera que cualquier rotación del volumen cae en el espacio compartido por la sucesión de planos. A la hora de mandar estos planos (en realidad se envian quads que les representan) se les aplica una matriz de transformación especial, no la modelview standard. Esta matriz viene definida por las siguientes comandas:

```
glMatrixMode (GL MODELVIEW);
glPushMatrix();
glTranslatef (mCenter.x(),mCenter.y(),mCenter.z());
glMultMatrixf (temp);
```

donde temp es la inversa para las rotaciones de la modelview actual, y mCenter el centro del volumen a visualizar. Haciendo una abstracción en 2D, el proceso vendría representado por el siguiente esquema, figura 8.

Lo que se está realizando es mandar siempre a pintar los mismos planos, en el ejemplo, un conjunto de líneas centradas en el origen y dirección horizontal. Aplicamos la matriz inversa a la modelview para hacer seguidamente una traslación en coordenadas de mundo y aplicarla para obtener las coordenadas de observador definitivas. Esto se realiza igual para cada brick, las operaciones sólo dependen del centro del mundo de vóxeles, común a todos los bricks, y de la cámara.

Lo que sí varía de brick a brick son las texturas que se envían a mapear en los planos proxy y las transformaciones realizadas para cada una de estas texturas mediante la matriz de texturas, figura 9. En el momento de crear cada brick, se crea una matriz de transformación para la textura del mismo.

Luego, durante el pintado del volumen, las transformaciones que se le aplican a la textura de cada brick son las siguientes:

```
glMatrixMode(GL TEXTURE);
glPushMatrix();
glLoadIdentity();
glTranslatef (1./(2.0),1./(2.0), 1./(2.0));
glTranslatef(
(boxGlobal.Center().x() - mVolCenter.x()) / mVolWidth,
(boxGlobal.Center().y() - mVolCenter.y()) / mVolHigh,
(boxGlobal.Center().z() - mVolCenter.z()) / mVolDepth
);
glScalef(
boxGlobal.GetDiagonal()/mVolWidth,
boxGlobal.GetDiagonal()/mVolHigh,
boxGlobal.GetDiagonal()/ mVolDepth
);
glMultMatrixf(temp);
```

El proceso de transformación de las coordenadas de textura consiste en enviar para cada brick las coordenadas centradas en el origen, para poder realizar correctamente el giro indicado por la inversa de la modelview. Acto seguido realizar el escalado correspondiente a la razón entre´las dimensiones del mundo de vóxeles original y las del brick. Por último se hacen las traslaciones pertinentes para que la textura ocupe su lugar relativo al mundo de vóxeles global. En definitiva se puede abstraer el proceso de transformación de coordenadas para los planos proxy y las texturas 3D, a la colocación de los planos en el lugar determinado por la cámara y la colocación de las texturas 3D para que se situen dentro de dichos planos en la dirección correcta.

Otro paso que se realiza por brick es el recortado, enviando al shader una caja de recorte, obtenida a partir de la textura sin offset de solapado. Con este paso se consigue que sólo se pinten los téxeles del brick que son disjuntos a otros bricks, recortando los téxeles offset que han servido para calcular el color correcto de los otros, pero que no deben contribuir a la imagen, porque lo hará el brick al que pertenezcan.

4.3. Ciclo de vida de la memoria RAM y de la memoria interna de la GPU

Uno de los mayores problemas que nos encontramos durante la implementación del proyecto fue el de la gestión de memoria.

El primer paso es cargar de disco el volumen y construir un modelo de voxeles en memoria RAM. Este paso no siempre se puede realizar directamente, los modelos cada vez son más grandes, llegando a ocupar hasta varias decenas de Gigabytes. La cantidad

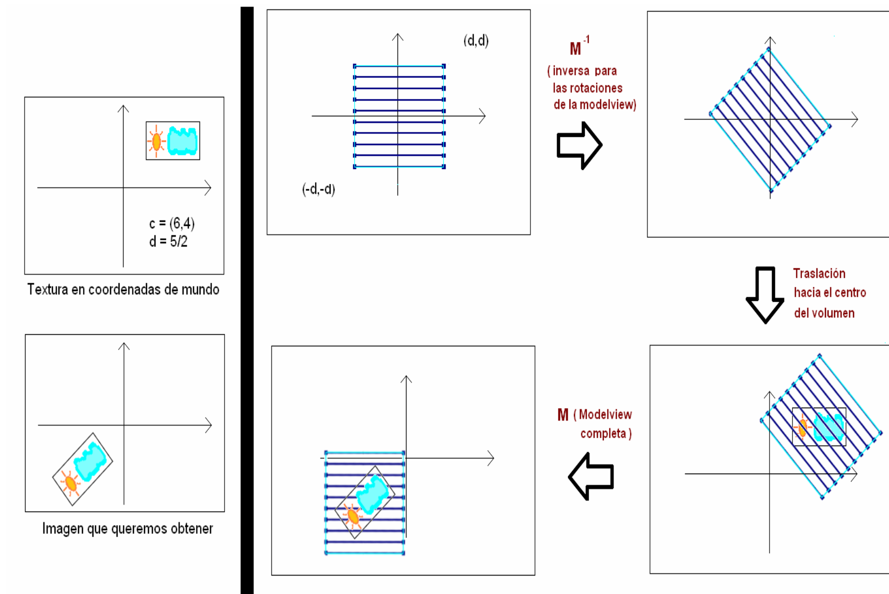


Figura 8: Proceso de transformación de las coordenadas de los planos

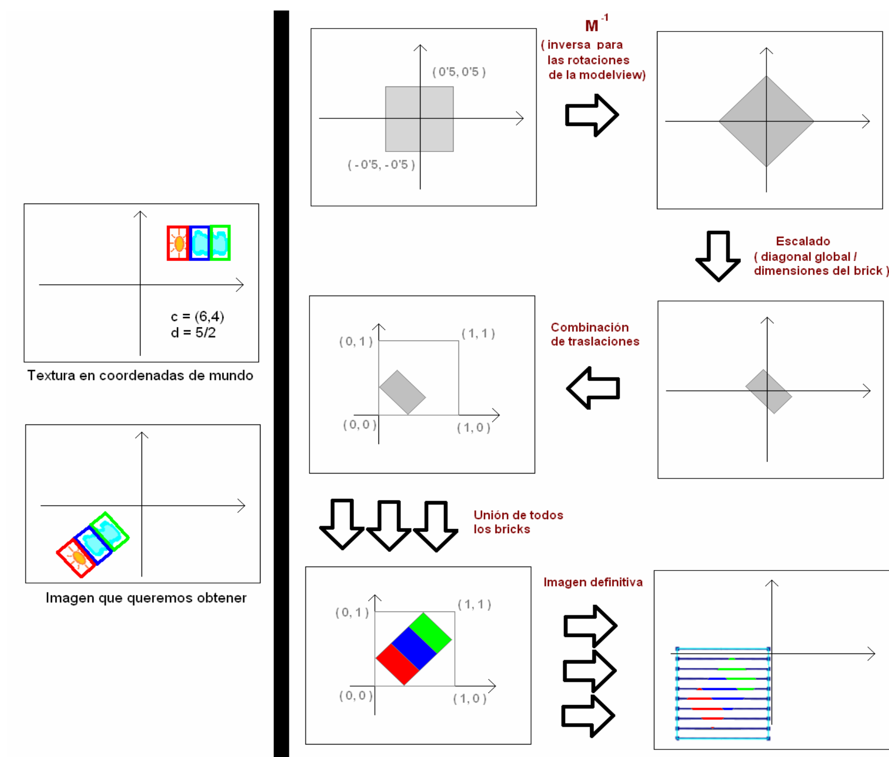


Figura 9: Proceso de transformación de las coordenadas de textura para cada brick

de memoria RAM que ofrecen los ordenadores hoy en dia va desde 512 MB hasta varios Gigabytes. En estos casos se pueden utilizar técnicas Out-Of-Core, que se encargan de manipular el modelo durante la etapa de preproceso y gestionar el envio de datos a la memoria RAM para que esta no se desborde.

En nuestro proyecto supusimos que el modelo cabia en memoria RAM, que el conflicto lo tendríamos con la capacidad de la memoria interna de la GPU, los valores de la cual oscilan entre 128 MB y 1 GB normalmente. Han ido surgiendo contratiempos a la hora de trabajar con la manera de gestionar la memoria por parte de las tarjetas gráficas. Las dos grandes compañias en este campo, ATI y NVIDIA, no ofrecen documentación apenas sobre este apartado. Tampoco es que exista ningún tutorial ni articulo que asegure su funcionamiento ni si existe diferentes comportamientos para tarjetas de la misma compañia. Por ejemplo, descubrimos que las tarjetas de Quadro de NVIDIA se encargaban de gestionar ellas mismas la memoria, es decir, podiamos ir enviandole texturas que cuando sobrepasaba el tamaño de la memoria interna ella misma hacia swapping al tener copias de las texturas en memoria RAM. En cambio, las tarjetas ATI que hemos probado no cargaban las texturas enviadas al soprepasar el máximo permitido.

Por lo tanto, a la hora de tratar con texturas, memoria interna de la GPU y memoria RAM, hay diversos factores a analizar. El primero es que cuando enviamos una textura a la tarjeta gráfica, al mismo tiempo que se carga en ella, se carga una copia de esta textura en RAM. Esta copia será la tarjeta quien la gestionará, ya sea para hacer swapping, ya sea para fines desconocidos y no documentados.

Por otra parte, la carga y descarga de texturas en la tarjeta por parte del programador, introduce un overhead considerable, por lo que en la medida de lo posible nos interesaría aprovechar al máximo las posibilidades que nos ofrezca la tarjeta gráfica en cuestión. En las tarjetas ATI nos veremos obligados a realizar manualmente el swapping o no podremo visaulizar el volumen entero.

Para finalizar, otro factor muy importante a tener en cuenta es la capacidad de la memoria RAM. Cuando cargamos un volumen de disco y creamos un mundo de vóxeles, este residirá en memoria RAM. Supongamos que este modelo ocupa 512 MB, que nuestra memoria RAM es de 1 GB, y que nuestra GPU tiene una memoria interna de 256 MB. Una vez cargado el modelo, en RAM tenemos algo más de 512 MB ocupados (el programa que esta siendo ejecutado, el sistema operativo, ..., en verdad se está ocupando bastante más de 512 MB) y que ahora tenemos que hacer bricking. Al trocear el mundo de vóxeles y enviar a pintar cada textura 3D (brick), la tarjeta está haciendo una copia en RAM, por lo que si no vamos descargando el brick una vez pintado esa memoria se desborda. Si por el contrario descargamos el brick para cada frame, necesitamos tenerlo guardado en RAM para poder volver a enviarlo en un frame posterior, por lo que estamos ocupando 512 MB en RAM del conjunto de los bricks. Durante todas estas situaciones hemos mantenido el mundo de vóxeles cargado en memoria por si teniamos q hacer bricking de nuevo con distintas dimensiones o por si necesitábamos acceder a los valores

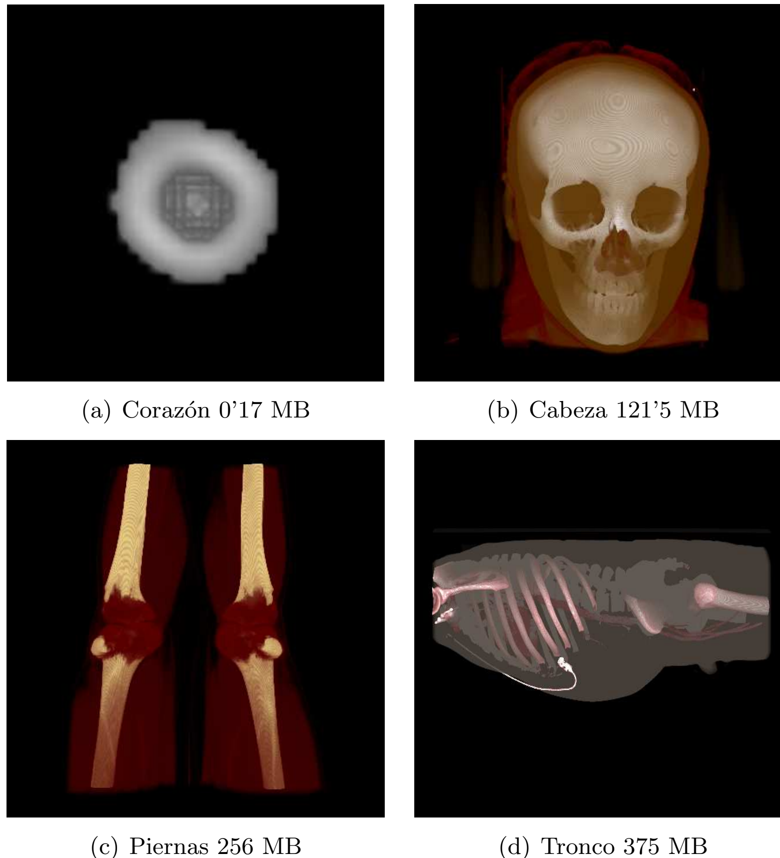
originales del modelo, ya fuera para hacer segmentación de los datos, o realizar procesos de interacción. En definitiva, es un problema complicado ya que la memoria RAM es finita y podemos superar su capacidad con facilidad al trabajar con este tipo de modelos, y una vez superado este limite el ordenador empieza a paginar y la aplicación deja de ser interactiva por completo.

5. Resultados

Hemos realizado una serie de experimentos con el programa para evaluar su rendimiento y extraer conclusiones. Los experimentos se han planteado sobre un Pentium® 4 3.4 GHz, con 1 GB de RAM, una NVIDIA Quadro FX 1400 como tarjeta gráfica (128 MB) y Windows XP Profesional Service Pack 2 como plataforma. La ejecución del programa fue en entorno Release.

Las variables que componen nuestro estudio son las siguientes:

- **Modelo utilizado**: Corazón, Cabeza, Piernas y Tronco.
- **Tipo de vista**: axial, sagital y coronal.
- **Número de planos proxy utilizados para la visualización**: standard y el 10 % de los standard.
- **Número de bricks utilizados**:1, 2, 8 y 128.



(a) Corazón 0'17 MB

(b) Cabeza 121'5 MB

(c) Piernas 256 MB

(d) Tronco 375 MB

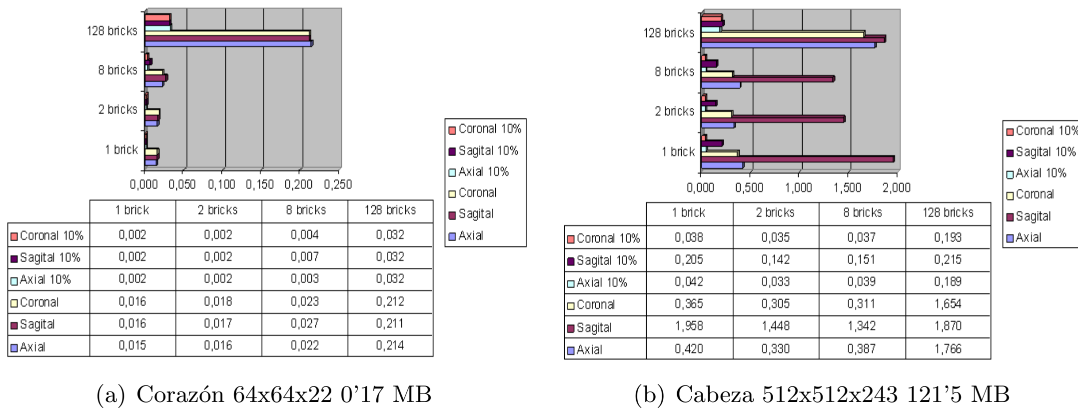
Figura 10: Modelos utilizados en nuestros experimentos

5.1. Experimentos con modelos pequeños

En este apartado vamos a analizar las pruebas realizadas sobre modelos que sí caben en memoria interna de la GPU. En principio, no sería necesario utilizar técnicas de bricking para su visualización, pero queremos ver como se ven afectados los tiempos de render por el uso de nuestros algoritmos.

En el caso del modelo Corazón, figura 11a, la textura es muy pequeña por lo que los tiempos no son realmente significativos. Lo único que podemos extraer es que no es conveniente dividir el volumen en muchísimos bricks, porque el coste añadido de gestionar y mandar a dibujar tantos, supera en mucho al de utilizar un número menor.

En el caso del modelo Cabeza, figura 11b, vemos una textura que casi alcanza el límite de la tarjeta, 128 MB. Se observa que el coste entre utilizar un 10 % de los planos es linealmente proporcional al de usar todos los planos, pero la calidad de la imagen se resiente mucho. También, se observa que la vista sagital es la que requiere más tiempo y que los mejores resultados se obtienen con 2 y 8 bricks. Por lo demás, a mayor número de bricks, sobre todo en el caso de 128, los costes por vista se igualan, lo que quiere decir que pesa más el número de bricks que el tipo de vista.



	1 brick	2 bricks	8 bricks	128 bricks
Coronal 10%	0,002	0,002	0,004	0,032
Sagital 10%	0,002	0,002	0,007	0,032
Axial 10%	0,002	0,002	0,003	0,032
Coronal	0,016	0,018	0,023	0,212
Sagital	0,016	0,017	0,027	0,211
Axial	0,015	0,016	0,022	0,214

(a) Corazón 64x64x22 0’17 MB

	1 brick	2 bricks	8 bricks	128 bricks
Coronal 10%	0,038	0,035	0,037	0,193
Sagital 10%	0,205	0,142	0,151	0,215
Axial 10%	0,042	0,033	0,039	0,189
Coronal	0,365	0,305	0,311	1,654
Sagital	1,958	1,448	1,342	1,870
Axial	0,420	0,330	0,387	1,766

(b) Cabeza 512x512x243 121’5 MB

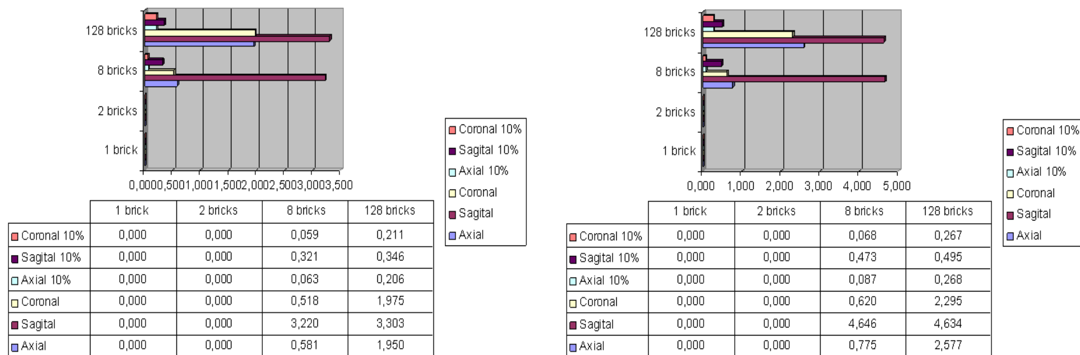
Figura 11: Resultados con modelos que caben enteros en memoria interna de la GPU.

5.2. Experimentos con modelos grandes

En este apartado vamos a analizar las pruebas realizadas sobre modelos que no caben en GPU, los cuales sin la técnica implementada en este proyecto antes no podiamos visualizar.

Los modelos estudiados, figura 12, ocupan el doble y más del triple de la memoria gráfica disponible, por lo que era imposible visualizarlos. Ahora, si los dividimos en más

de dos bricks podemos representarlos de una manera casi interactiva. En el caso de tener 8 (caso que mejores resultados está dando), comprobamos que, si la vista no es sagital, el frame rate obtenido es correcto. Ahora bien, en cuanto aparece la vista sagital o elevamos el número de bricks considerablemente, el rendimiento decae mucho. Como en el caso anterior, el número de planos es un factor determinante tanto del rendimiento como de la calidad de la imagen, por eso es interesante disminuirlo mientras se manipula el modelo y utilizar un número óptimo para renderizar el modelo parado.



	1 brick	2 bricks	8 bricks	128 bricks
Coronal 10%	0,000	0,000	0,059	0,211
Sagital 10%	0,000	0,000	0,321	0,346
Axial 10%	0,000	0,000	0,063	0,206
Coronal	0,000	0,000	0,518	1,975
Sagital	0,000	0,000	3,220	3,303
Axial	0,000	0,000	0,581	1,950

(a) Piernas 512x512x512 256 MB

	1 brick	2 bricks	8 bricks	128 bricks
Coronal 10%	0,000	0,000	0,068	0,267
Sagital 10%	0,000	0,000	0,473	0,495
Axial 10%	0,000	0,000	0,087	0,268
Coronal	0,000	0,000	0,620	2,295
Sagital	0,000	0,000	4,646	4,634
Axial	0,000	0,000	0,775	2,577

(b) Tronco 512x512x750 375 MB

Figura 12: Resultados con modelos que no caben enteros en memoria interna de la GPU.

5.3. Experimentos con iluminación local

Otro factor a tener en cuenta es el uso o no de algoritmos de iluminación local, que mejoran la visualización pero afectan al rendimiento.

Podemos ver la diferencia de tiempo entre visualizar un modelo con iluminación y sin ella, figura 13. Es más costoso con iluminación, porque para cada píxel el shader de pintado accede a los valores de textura vecinos para calcular el gradiente. Dado el alto número de píxeles para calcular el color y, por lo tanto, la elevada cantidad de accesos a texturas, el rendimiento se ve afectado. Además, en este experimento realizado sobre diferentes modelos y tipos de bricking, se aprecia cómo, a medida que el rendimiento de la visualización básica decrece, el de iluminación decrece de manera casi lineal.

Se observa también, teniendo en cuenta que los datos están ordenados de menor a mayor tiempo de render en el caso de iluminación, que la vista sagital es el peor de los casos, ya que en los tres modelos grandes es así, siendo para Piernas y Cabeza mayor el tiempo gastado por el modelo Tronco, realmente grande en las vistas Axial y Coronal.

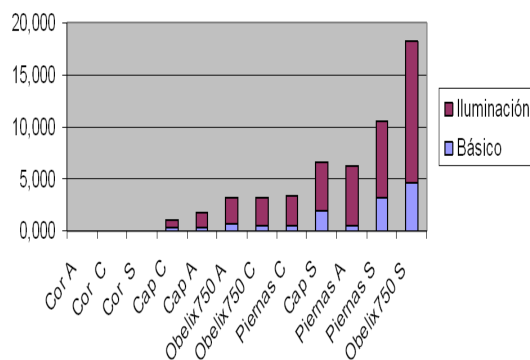
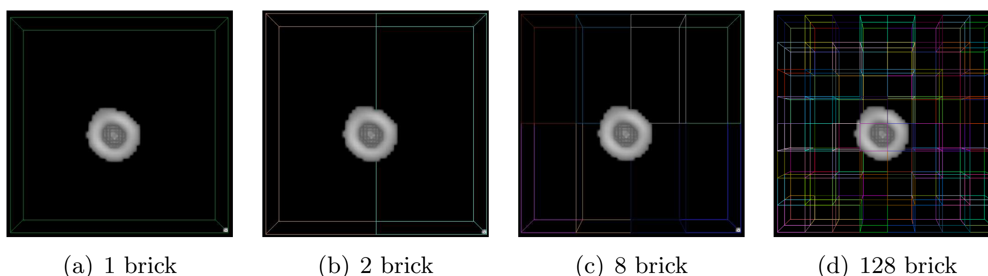


Figura 13: Resultados referentes al uso de técnicas de iluminación local

Por último recalcar que la imagen obtenida es exactamente la misma con cualquier configuración, es decir, que el número de subdivisiones en bricks hechas del volumen no influye para nada en la imagen, figura 14. El algoritmo utilizado lo que asegura es cualidad, se esté empleando iluminación local o no. Esto es importante sobre todo en el ámbito en el que estamos trabajando, donde no podemos permitir ningún tipo de error. Como se comenta en la sección Trabajos relacionados, no todas las técnicas existentes cumplen esta condición y en algunas, como en la Multiresolución, es, además, uno de los problemas que existen.



(a) 1 brick (b) 2 brick (c) 8 brick (d) 128 brick

Figura 14: Imágenes del modelo Corazón con distinta cantidad de subdivisiones.

Figura 15: Imagen del modelo Cabeza con iluminación local

6. Conclusiones y trabajo futuro

Tras el análisis de los resultados obtenidos, podemos concluir que con modelos que sí caben en memoria gráfica, el rendimiento del visualizador viene condicionado por el número de planos proxy y, en menor medida, por el tipo de vista. En caso de tener una cantidad elevada de bricks, el tipo de vista tiene menor influencia.

En el caso de ejemplos que no caben directamente en memoria de GPU, que antes éramos incapaces de visualizar, las conclusiones son un poco diferentes. En este caso, los aspectos que más inciden en el rendimiento de la herramienta, a parte del tamaño del volumen, serian el número de planos proxy obviamente, y después el tipo de vista. La vista sagital es con diferencia la que requiere de mayor tiempo de renderizado.

En definitiva, los tiempos obtenidos son buenos, teniendo en cuenta los tamaños con los que estamos trabajando, pero se pueden mejorar mucho. Además, el bricking nos permite aprovechar técnicas ya implementadas, adaptar algoritmos existentes y utilizar de nuevos, a un coste de recursos asequible. Otra cualidad del bricking es que permite su implementación en múltiples GPUs en paralelo.

Una conclusión que alcanzamos es que había que implementar el bricking y para entender realmente como gestionan la memoria las tarjetas gráficas. Debido a la escasa documentación sobre este tema, el tiempo consumido en la implementación y modificación de código relativo a la gestión de memoria y funcionamiento de las GPU ha sido mayor en comparación con ningún otro. Tras múltiples pruebas y experimentos podemos concluir que hay peculiaridades según la tarjeta sea de la compañia ATI o de la compañia NVIDIA. Las primeras no realizan el swapping automaticamente en caso de sobrepasar la memoria de la GPU, en cambio las NVIDIA probadas se encargaban ellas mismas. Ha sido estimulador intentar obtener el máximo rendimiento de la GPU y a la vez complicado al no disponer siempre de la documentación requerida.

Una posible solución al problema planteado en el apartado 4.3 es la de eliminar el mundo de vóxeles una vez construidos los bricks. En memoria tener solo las copias que hace la tarjeta de las texturas 3D enviadas, y que cada vez que realicemos el swapping manual obtengamos de la tarjeta la textura expulsada de la propia tarjeta, de esta manera mantenemos la memoria RAM lo más vacia posible pero en contra añadimos el overhead de copiar la textura saliente y de hacer el swapping manual. Es una de las posibilidades a implementar en un futuro para evaluar si el coste del overhead nos ofrece la visualización de volúmenes más grandes o si por el contrario el tiempo de visualización se hace demasiado grande.

Entre las futuras líneas de desarrollo de nuestra herramienta, consideramos que aplicar algún tipo de compresión sin pérdida o que modifique minimamente la visualización, últimamente hemos estado investigando en profundidad la compresión que ofrecen las tarjetas gráficas, utilizar técnicas de multiresolución, enviar a renderizar

simplificaciones del volumen para cámaras lejanas, o introducir el uso de estructuras jerárquicas. Lo que queda muy claro es que la solución a este problema se encuentra en la combinación de distintas técnicas y que los resultados obtenidos invitan a seguir trabajando por este camino.

Bibliografía

[1] Klaus Engel, Markus Hadwiger, Joe Kniss, Aaron Lefohn, Christof Rezk-Salama, and Daniel Weiskopf, *Real-time volume graphics*, 2004.

[2] Andrew S. Glassner, *Principles of digital image synthesis*, ch. Wavelet Transforms, pp. 243–298, Morgan Kaufmann, 1995.

[3] Joe Kniss, Patrick McCormick, Allen McPherson, James Ahrens, Jamie Painter, Alan Keahey, and Charles Hansen, *Interactive texture-based volume rendering for large datasets*, **21** (2001), no. 4, 52–61.

[4] Martin Kraus and Thomas Ertl, *Adaptive texture maps*, HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (Aire-la-Ville, Switzerland, Switzerland), Eurographics Association, 2002, pp. 7–15.

[5] Jens Krüger and Rüdiger Westermann, *Acceleration Techniques for GPU-based Volume Rendering*, Proceedings IEEE Visualization 2003, 2003.

[6] Eric C. LaMar, Bernd Hamann, and Kenneth I. Joy, *Multiresolution techniques for interactive texture-based volume visualization*, IEEE Visualization '99 (San Francisco) (David Ebert, Markus Gross, and Bernd Hamann, eds.), 1999, pp. 355–362.

[7] D. Xue R, Crawfish and C. Zhang, *The visualization handbook*, ch. Volume Rendering Using Splatting, Elsevier, 2005.

[8] Flemming Friche Rodler, *Wavelet based 3d compression with fast random access for very large volume data*, PG '99: Proceedings of the 7th Pacific Conference on Computer Graphics and Applications (Washington, DC, USA), IEEE Computer Society, 1999, p. 108.

[9] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser, *Smart hardware-accelerated volume rendering*, 2003.

[10] Daniel Ruijters and Anna Vilanova, *Optimizing gpu volume rendering*, Computer Graphics, Visualization and Computer Vision'2006 (2006).