Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

Despliegue de Volúmenes Multi-resolución Basado en Ray Casting de una Pasada

Trabajo especial de grado presentado por
Br. Kijam Enrique José López Berrocal

ante la ilustre
Universidad Central de Venezuela
para optar al título de
Licenciado en Ciencias de la Computación

Tutor:
Prof. Rhadamés Carmona

Caracas, Octubre de 2011

Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

ACTA DEL VEREDICTO

Quienes suscriben, Miembros del Jurado designado por el Consejo de la Escuela de Computación para examinar el Trabajo Especial de Grado, presentado por el Bachiller Kijam Enrique José López Berrocal C.I.: 17.979.410, con el título "Despliegue de Volúmenes Multi-resolución Basado en *Ray Casting* de una Pasada", a los fines de cumplir con el requisito legal para optar al título de Licenciado en Computación, dejan constancia de lo siguiente:

Leído el trabajo por cada uno de los Miembros del Jurado, se fijó el día 26 de Octubre de 2011 a las 3PM, para que su autor lo defendiera en forma pública, en el Centro de Computación Gráfica, lo cual este realizó mediante una exposición oral de su contenido, y luego respondió satisfactoriamente a las preguntas que les fueron formuladas por el Jurado, todo ello conforme a lo dispuesto en la Ley de Universidades y demás normativas vigentes de la Universidad Central de Venezuela. Finalizada la defensa pública del Trabajo Especial de Grado, el jurado decidió aprobarlo.

En fe de lo cual se levanta la presente acta, en Caracas el 26 de Octubre de 2011, dejándose también constancia de que actuó como Coordinador del Jurado el Profesor Tutor Rhadamés Carmona.

_____
Prof. Rhadamés Carmona
(Tutor)

_____
Prof. Otilio Rojas
(Jurado Principal)

_____
Prof. Héctor Navarro
(Jurado Principal)

Resumen

Con los avance de la tecnología en los últimos años ha ido incrementando el tamaño de los datos volumétricos; específicamente, esto es debido al mejoramiento de los equipos de captura de imágenes; la resolución y el detalle que es posible obtener en la actualidad son de tal magnitud que superan las capacidades del *hardware* gráfico, incluso llegan a superar la capacidad de la memoria principal de un computador convencional. Por lo tanto, se han ido incorporando nuevos métodos para poder procesarlos y visualizarlos en tiempo real.

En este trabajo se presenta un método de visualización que utiliza una jerarquía multi-resolución por bloques, donde cada bloque a visualizar es almacenado en una textura única denominada atlas. Esta textura es desplegada haciendo un sólo recorrido sobre ella aplicando el método de *Ray Casting* acelerado en GPU. El objetivo del despliegue de volúmenes multi-resolución es asignar un nivel de detalle a cada área del volumen acorde a una prioridad. Las prioridades utilizadas en este trabajo se basan en la distorsión de los datos con respecto a la representación más fina del volumen, en la distancia con respecto a un punto de interés y en la capacidad de la textura atlas. El cálculo de la distorsión basado en los datos es un proceso que puede tomarse un tiempo considerable; esto es se debe a la necesidad de visitar todos los vóxeles en cada nivel de detalle y compararlos con el nivel más fino. Para acelerar el cálculo de la distorsión se implementó un algoritmo paralelo basado en CUDA y otro basado en OpenMP.

Palabras Claves: volúmenes multi-resolución, *Ray Casting*, textura atlas, jerarquía por bloques, distorsión basado en los datos, CUDA, OpenMP.

Agradecimientos

Agradezco a Dios por estar presente en cada uno de los momentos de mi vida para conseguir todo los logros que he obtenido. Mamalita que desde el cielo me ayuda y me guía por el buen camino.

Agradezco a mis Padres por estar presentes en toda mi vida, darme todo su apoyo incondicional para lograr todas mis metas y ayudarme a surgir desde todo punto de vista. A mi Mamá le agradezco haberme dicho de una manera muy DIRECTA que aceptara el cupo de Matemáticas en un inicio, gracias a esta decisión pude estudiar lo que siempre había querido. Mi hermana que siempre está atenta a todo lo que hago (así tenga que llamarme 10 veces al día para saber donde estoy y a qué hora llego a la casa XD, cuaima cuaima). Mi hermano Kenito que siempre ha buscado la forma de ayudarnos en todo lo posible aún cuando lo tengamos muy lejos ☹. Cherry que siempre me ayuda en lo que ha podido y aunque me este fastidiando todo el tiempo, sé que siempre cuento con todo su apoyo. A mis sobrinitos Alejandro y Ángel siempre los tendré a ambos presentes en mi mente.

Agradezco a Rubén por su tiempo al darme clases para poder aprobar la prueba interna, espero que Dios siempre le dé lo mejor a él y a su familia. Carmencita gracias por ayudarme en la redacción y ortografía del seminario.

Agradezco a mí linda Novia Fiorella por estar presente en todo momento y darme todo su apoyo posible. Dios quiera que siempre estemos juntos y me permita darte el apoyo necesario en tu carrera para que seas una excelente médico (Te amo!).

Agradezco a Alex Pérez por ayudarme con el documento y el desarrollo de la aplicación, donde compartimos muchos conocimientos para la utilización de CUDA y, entre otras muchas cosas, por tener siempre la paciencia de explicarme en modo “pre-escolar” las materias teóricas XD.

Agradezco a mis amigos Pedro Caicedo, Adriana Urdaneta y Edgar Bernal (Alias los Vagos) por ayudarme en toda la carrera y por todo los buenos momentos que hemos pasado. Las salidas al cine, las parrillas en casa del vago y la mía en épocas de proyectos; los viajes a la playa, el paseo a San Cristóbal, Margarita, entre otras muchas cosas. Aún cuando no nos graduemos todos juntos, no hay que olvidar todas las cosas buenas que siempre hemos compartido. Y Adriana gracias por escuchar mis tonterías cuando necesitaba de una amiga.

Agradezco a mi tutor Rhadamés Carmona (alias Rix) por sus ideas y soluciones en la aplicación. Gracias por dar su “*best*” en corregir el documento a tiempo para graduarme ahorita en Diciembre.

Agradezco a todos los que conforman el Centro de Computación Gráfica (Profesores, Estudiantes, etc.) por prestar siempre una ayuda cuando se necesitaba. En especial a Karina Pedrique que me salvo la patria en más de una ocasión y por ayudarme con CUDA.

Agradezco a todos los que de alguna forma me hayan ayudado a conseguir mis metas y que no haya mencionado aquí.

Tabla de contenidos

Introducción

La visualización de volúmenes hoy en día tiene una gran importancia, tanto en el área médica como en áreas de la ciencia, en la 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. Como por ejemplo el proyecto del Humano Visible [**1**], donde en particular, la mujer visible tiene una resolución a full color de 2048 x 1216 x 5186 ocupando 36 *gigabytes* sin compresión.

Muchas técnicas se han desarrollado en los últimos tiempos para disminuir el problema de espacio y procesamiento de los datos. Esto es motivado a que el poder de cómputo del *hardware* gráfico, del procesador central y el ancho de banda de los buses del sistema, mejoran año tras año. Diversos investigadores han propuesto métodos para poder visualizar esta gran cantidad de datos. Algunas de las propuestas se basan en ir desplegando pequeños subvolúmenes hasta lograr visualizarlo completamente; otras en desplegar solamente un subvolúmen que pueda ser almacenado en la memoria gráfica, y la más destacada, el despliegue de volúmenes multi-resolución. En este último caso, el volumen es desplegado con distintos niveles de detalle dependiendo de las prioridades de las distintas áreas del volumen, que suelen basarse en la distancia a un punto de interés, limitaciones del *hardware* y en métricas de error en el espacio objeto o imagen.

Planteamiento del Problema

Pese a que la mayoría de las técnicas de visualización de volúmenes multi-resolución lidian con el problema de la limitación de memoria, el despliegue suele tener artefactos visuales porque los niveles de detalle no suelen ser seleccionados de manera correcta. Los criterios utilizados para determinar las zonas que deben ser refinadas usualmente no toman en cuenta la distorsión del volumen considerando la función de transferencia. La función de transferencia es la utilizada para asignarle un color una opacidad a cada elemento del volumen. Esto se debe a que este cálculo de la distorsión es un proceso que puede requerir de un tiempo considerable de cómputo, el tiempo requerido es relativo a la dimensión del volumen. Aunque hay técnicas que generan resultados aproximados muy certeros, por lo general es calculada al inicio de la aplicación y sin tener en cuenta los cambios que pueda tener la función de transferencia durante la ejecución. Adicional a esto, comúnmente para los niveles de detalles se utilizada una jerarquía *octree* para el despliegue (ver ***Sección 2.3.1***), el problema de utilizar esta técnica es que en el momento de refinar un nodo obligatoriamente tiene que subdividirse en ocho partes, dando como consecuencia que se refinan zonas que podrían no requerirlas.

La técnica de *Ray Casting* es la utilizada en la mayoría de las investigaciones para la visualización de volúmenes. En el área del despliegue multi-resolución esta técnica es aplicada a cada uno de los bloques que se van a visualizar, el problema es que se genera una sobre carga en el GPU ya que se deben generar polígonos y aplicar las respectivas texturas para cada uno de los bloques seleccionado. Esto sin considerar que se deben ordenar respecto a la posición de la cámara para aplicar correctamente la operación de mezcla, partiendo desde el bloque más lejano hasta el bloque más cercano. Por este motivo se dice que esta técnica requiere de múltiples pasadas.

Propuesta de Solución

En este trabajo especial de grado se plantea el desarrollo de una aplicación que utilice aceleración gráfica en el despliegue del volumen multi-resolución. En particular, se propone utilizar el algoritmo de *Ray Casting* acelerado por GPU. Para representar al volumen, se desea utilizar una jerarquía por bloques; en este sentido, el volumen original es primero dividido en bloques de igual tamaño, y cada bloque es representado en distintos niveles de detalles, generando una jerarquía multi-resolución local (una por cada bloque del volumen original).

El criterio de selección a utilizar se basa en la distancia con respecto a un punto de interés, la capacidad de la textura atlas y la distorsión del volumen multi-resolución considerando la función de transferencia. Para calcular la distorsión (basado en los datos) del volumen multi-resolución en tiempo real se plantea desarrollar un algoritmo paralelo utilizando el GPU (CUDA) o los *core's* del CPU (OpenMP) considerando los cambios de la función de transferencia. Finalmente se avaluará cuál de estas dos tecnologías ofrece resultados en menor tiempo.

Mediante un criterio de selección, se seleccionan los bloques a visualizar; estos serán almacenados en una sola textura de gran tamaño denominada atlas y esta a su vez es indexada en otra de menor tamaño llamada textura de índices. La ventaja de utilizar el atlas, es que la técnica de *Ray Casting* es aplicada a una sola geometría que tiene enlazada únicamente la textura de índices. A medida que el rayo de visualización recorre el interior de la geometría se van accediendo a los vóxeles que están en el atlas. Como se puede observar no se requiere múltiples pasadas por que solo se visualiza una única geometría y no cada bloque, adicionalmente no se requiere aplicar ningún tipo de ordenamiento, por lo tanto esta técnica disminuye la carga del GPU.

Objetivos Generales

Implementar un prototipo de visualización de volúmenes multi-resolución basado en *Ray Casting* de una pasada utilizando una textura atlas.

Objetivos Específicos

- Adaptar el sistema de despliegue de volúmenes multi-resolución realizado por *Carmona* [**2**] para manejar una jerarquía multi-resolución basada en bloques y una textura atlas.
- Desarrollar un algoritmo que genere y administre la textura atlas.
- Adaptar el algoritmo de *Ray Casting* convencional para realizar el despliegue en una sola pasada de la textura atlas.
- Establecer el criterio de selección basados en la distancia a un punto de interés, la capacidad de la textura atlas y la distorsión del volumen multi-resolución.
- Desarrollar los algoritmos necesarios para el cálculo de la distorsión en paralelo utilizando el GPU y el CPU.
- Realizar pruebas de rendimiento del algoritmo de *Ray Casting* de una pasada.
- Calcular los tiempos requeridos por el GPU y el CPU para calcular la distorsión y el tiempo total necesario para el refinamiento total del volumen.

Alcance de este Trabajo

- La implementación al requerir la arquitectura CUDA solo se puede ejecutar en dispositivos gráficos NVidia.
- Se restringe el tamaño de los archivos de datos volumétricos a un máximo 1.5GB de memoria.

El documento está dividido en varios capítulos. El primer capítulo presenta el marco teórico que introduce a la visualización de volúmenes. Como segundo capítulo se muestra las técnicas multi-resolución. Se plantean sus problemas, haciendo énfasis en los criterios propuestos en trabajos previos para la selección de los niveles de detalle y en las métricas de error utilizadas para el cálculo de la distorsión. En el tercer capítulo se muestran las tecnologías actuales para ejecutar aplicaciones en paralelo utilizando el CPU (OpenMP) y el GPU (CUDA). En el cuarto capítulo se describe los módulos del sistema sobre el cual se basa el presente trabajo especial de grado, así como los nuevos algoritmos que se implementaron, la jerarquía por bloques y el cálculo de la distorsión en tiempo real. En el quinto capítulo se presentan una serie de pruebas y resultados, mostrando el tiempo promedio requerido para la generación de un *frame*, haciendo énfasis en el tiempo requerido para el cálculo de la distorsión utilizando las distintas versiones desarrolladas (usando el CPU y usando el GPU). Por último, en el sexto capítulo se expondrán las conclusiones y los trabajos futuros.

Capítulo 1: Marco Teórico

El despliegue de volúmenes es una técnica que consiste en la generación de una imagen bidimensional, partiendo de datos discretos que pertenecen a un área tridimensional (volumen). La carga computacional que requiere el despliegue de un volumen depende directamente del tamaño, que puede llegar a ocupar en disco desde pocos *megabytes* hasta decenas de *gigabytes*. Los volúmenes de gran tamaño son aquellos que exceden la capacidad de la memoria del *hardware* gráfico y potencialmente la de la memoria principal de un computador. A continuación describiremos brevemente las técnicas utilizadas en la visualización de volúmenes.

1.1 Despliegue Directo de Volúmenes

El despliegue directo de volúmenes, también llamado *Direct Volumen Rendering,* es una técnica utilizada para la visualización de una secuencia de cortes seriados de un objeto (*data-set*), sin necesidad de utilizar reconstrucción 3D de superficies intermedias. Estos *data-sets* pueden ser obtenidos de una tomografía computarizada, ecografías 3D, resonancia magnética o cualquier medio de escaneo que genere una secuencia de imágenes. El *data-set* por lo general está almacenado en un arreglo tridimensional de escalares, llamados vóxeles.

De los datos volumétricos se pueden extraer contornos o también se puede generar una superficie poligonal [**3**] [**4**] [**5**]. En esta investigación sólo se considera el despliegue directo de volúmenes, donde no se reconstruyen formas poligonales, sino que se visualizan los datos volumétricos directamente teniendo en cuenta que la aceleración por *hardware* gráfico es posible. Con la visualización de volúmenes se logra estudiar la información que no está ni en los contornos ni en la superficie del volumen. Por comodidad, en el presente trabajo los términos despliegue directo de volúmenes y despliegue de volúmenes serán utilizados indistintamente.

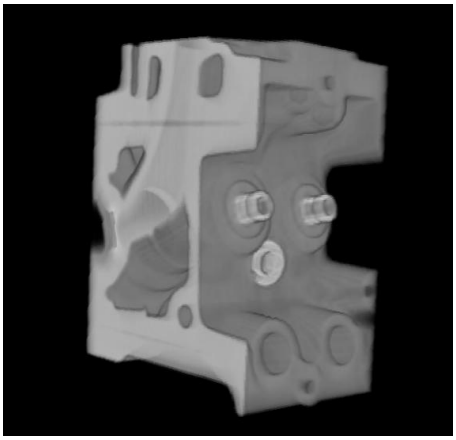


Figura 1.1*: imagen generada mediante el despliegue de volúmenes (Engine Block [**6**])*

El proceso de visualización de volúmenes simula la interacción de la luz al pasar un medio semi-transparente. Durante la interacción pueden ocurrir varios fenómenos:

- La luz es absorbida por los elementos del volumen.
- La luz se dispersa con los elementos del volumen.
- La luz se emite desde los elementos del volumen.

El proceso de dispersión de la luz se le denomina *scattering*. Aunque se pretenda simular todas las interacciones, la cantidad de cálculo involucrado es muy elevada. En un modelo más simplificado para el despliegue de volúmenes, solamente se utiliza la luz que es absorbida y emitida por los elementos del volumen. Este modelo óptico se representa por algunos trabajos mediante la siguiente ecuación matemática [**7**] [**8**] [**9**]:

$$C = \int_0^D c(\lambda)t(\lambda)e^{-\int_0^\lambda t(\lambda')d\lambda'}\,d\lambda \qquad \textit{Ec. 1.1}$$

donde C es el color resultante de la interacción de la luz con el volumen para un rayo de luz parametrizado con el parámetro λ, con $\lambda \in [0, D]$, mientras que 0 y D son los valores de λ que delimitan el rayo dentro del volumen. La emisión de la luz es representado por $c(\lambda)$ y la absorción como $t(\lambda)$. Por último el factor $e^{-\int_0^\lambda t(\lambda')d\lambda'}$ representa la extinción[1] de la luz o también se puede interpretar como la transparencia $T(\lambda)$ a una profundidad o distancia λ. Por otra parte, la opacidad se define como $1 - T(\lambda)$.

El proceso de visualización es llevado a cabo mediante el *hardware* gráfico, que contiene una serie de etapas para completar la visualización. A continuación se procede a explicar cada una de dichas etapas.

1.2 Hardware Gráfico

Antes de exponer las técnicas que existen para la visualización de volúmenes, primero se menciona el funcionamiento del *hardware* gráfico, tarjeta gráfica o GPU (Unidad de Procesamiento Gráfico). Este procesador gráfico está dedicado al procesamiento de polígonos u operaciones de punto flotante. Es utilizado comúnmente en los videojuegos y en aplicaciones 3D interactivas para aligerar la carga de trabajo de la unidad central de procesamiento (CPU).

Para el caso de despliegue de volúmenes, el *data-set* es enviado a la memoria de textura del GPU. El formato y dimensión de las texturas depende de la técnica de visualización que se utilice. El despliegue es realizado mediante el texturizado de polígonos, y es llevado a cabo por el *pipeline* gráfico. Al terminar todo el recorrido se genera una imagen similar a la ***Figura 1.1***. Más adelante se explica cada una de las técnicas que se pueden utilizar para este fin. El *pipeline* gráfico en general funciona de la siguiente manera:

[1] Extensión: Atenuación de la luz debido a la absorción y el *scattering.*

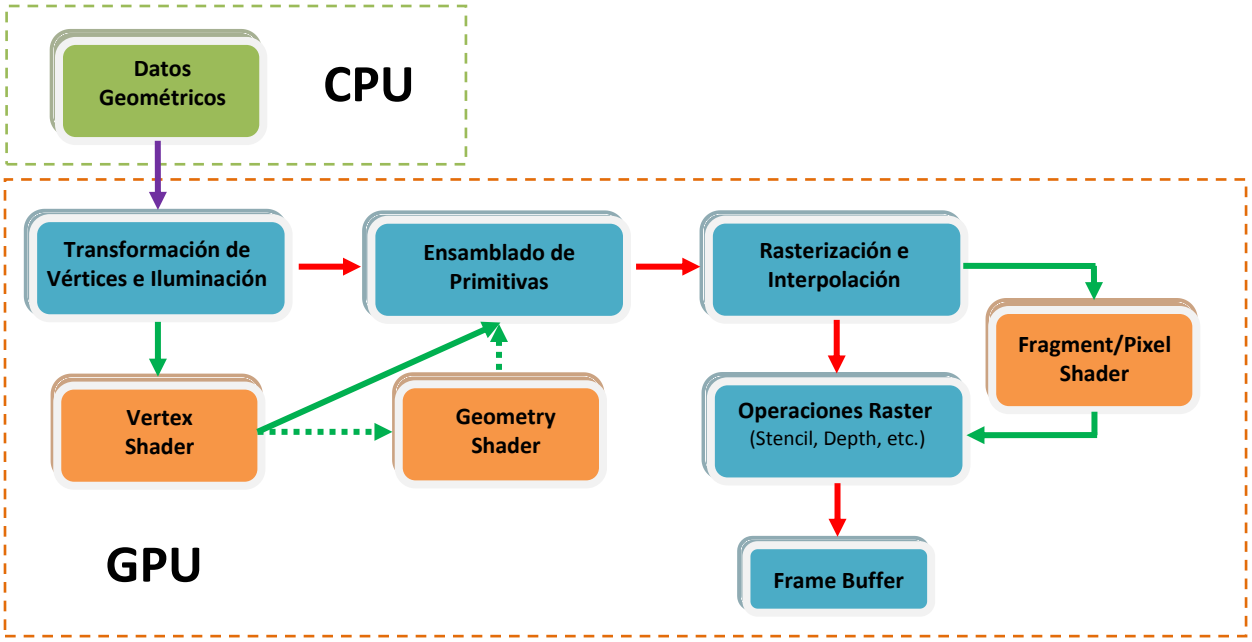


***Figura 1.2**: Pipeline y shaders del hardware gráfico.*

- *Procesamiento geométrico:* En esta etapa se aplica una serie de operaciones geométricas a los vértices. Éstas pueden ser proyecciones, transformaciones de las coordenadas de textura, de color y la aplicación de un modelo de iluminación. En el caso de utilizar un *vertex shader,* se puede re-programar todo este proceso para agregar o cambiar algunas funciones. Esto da libertad a los programadores de realizar diferentes efectos, como por ejemplo, deformar un objeto.

- *Ensamblado de Primitivas:* En esta etapa las primitivas pueden ser removidas o cortadas para generar nuevos vértices. Esto puede deberse a que el objeto está parcial o totalmente fuera de la pirámide de visualización. Posteriormente estas primitivas son transformadas a coordenadas de dispositivo o *viewport*. La utilización del *geometry shader* es algo relativamente nuevo. En el caso de las tarjetas gráficas basadas en chips NVidia [**10**], el *geometry shader* es soportado a partir de la serie 8. Aquí los programadores pueden generar nuevas primitivas a partir de los vértices obtenidos por el procesador geométrico.

- *Rasterización:* en este proceso, las primitivas son transformadas a fragmentos, donde cada fragmento corresponde a un píxel del buffer de imagen (*frame buffer*). En esta etapa se interpolaran los colores de los vértices, la profundidad, las coordenadas de textura y se calculan otros atributos que van a ser utilizado en las Operaciones Raster.

- *Operaciones Raster:* Está constituido por dos procesos. El primero es combinar el color con la textura y aplicar otras propiedades al fragmento para determinar su color final. El siguiente proceso constituye la aplicación de una serie de operaciones, que puede implicar la eliminación del fragmento. Esto ocurre si el fragmento es muy transparente o si no forma parte de la máscara del Esténcil (*Stencil*). Si el fragmento no es removido, se le aplica una operación de mezcla (*Blending*) con el fragmento que se encuentre en el buffer

de imagen. Mediante un *fragment shader* un programador puede hacer diversas transformaciones como cambiar la profundidad, trabajar con téxeles[2] y calcular efectos de iluminación con gran precisión.

Actualmente, el procesamiento de vértices y fragmentos es programable mediante programas de *shaders*. Los *shaders* constituyen una tecnología que han experimentado una gran evolución. Está destinada a proporcionar al programador una interacción con el GPU que antes no se podía. Existen lenguajes de alto nivel como Cg (*C for Graphics*) [**10**], HLSL (*High Level Shading Language*) [**11**] y GLSL (*OpenGL Shading Language*) [**12**], que permiten programar el *hardware* gráfico. Lo importante es que los desarrolladores sólo se preocupen por el funcionamiento de su aplicación y el uso correcto del API (*Application Programming Interface*) que se esté utilizando, mientras que los fabricantes del *hardware* se encarguen de la ejecución de las instrucciones y del rendimiento.

En esta investigación se considera el API de OpenGL [**12**] como base de los algoritmos de rendering.

1.3 Función de Transferencia

Normalmente el *data-set* no contiene la intensidad de luz por vóxel. Sólo incluye escalares que están representados comúnmente en 8*,* 12 o 16 *bits*. Al desplegar un volumen, el usuario decide qué materiales o estructuras visualizar. Para ello, se le asignan las propiedades ópticas de emisión *c* y absorción *t* a cada vóxel del volumen. A este proceso se le denomina clasificación.

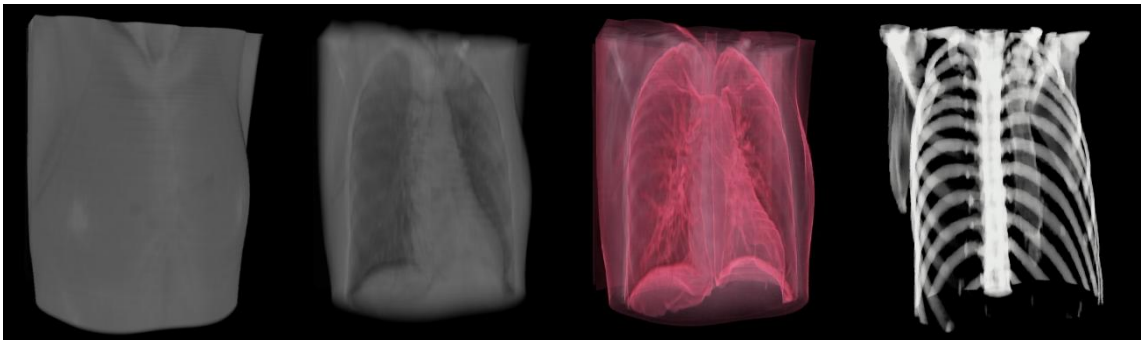
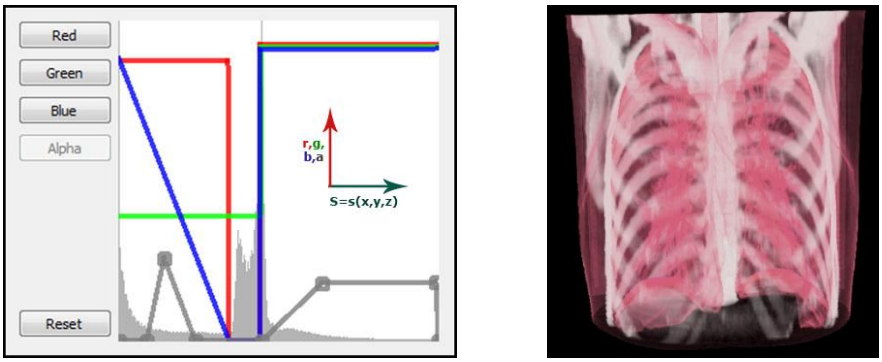


Figura 1.3*: un volumen desplegado con distintas funciones de transferencia. (Female Chest [**6**])*

Una forma de clasificar los vóxeles de un volumen es utilizando funciones, que al manipularlas se puedan resaltar las distintas áreas del volumen como en la ***Figura 1.3***. En el área de visualización de volúmenes se les denomina funciones de transferencia, que pueden ser de una dimensión o multidimensionales [**13**].

[2] Un téxel (contracción del inglés *texture element*, o también *texture pixel*) es la unidad mínima de una textura aplicada a una superficie. Similar a una imagen, se representa mediante arreglo bidimensional de píxeles, denominados téxeles.

En las funciones de transferencias unidimensionales el dominio está dado comúnmente por los valores de las muestras escalares $S = s(x, y, z)$. El rango de la función representaría la emisión $c(S)$ y absorción $t(S)$ de la luz. Una función de transferencia multidimensional puede requerir adicionalmente en el dominio la magnitud del gradiente, el producto punto contra la dirección de la luz, la normal del vóxel, etc. En esta investigación sólo se hará referencia a las funciones unidimensionales.



(a) (b)

***Figura 1.4**: En la figura (a) se encuentra la Función de transferencia teniendo de fondo el histograma de frecuencia y en la (b) un volumen desplegado utilizando dicha función. (Female Chest [**6**])*

En la ***Figura 1.4*** se puede observar los cuatro canales $RGBA$ ($R =$ Red/Rojo, $G =$ Green/Verde, $B =$ Blue/Azul y $A =$ Alpha) donde $c(S) = RGB$ y $t(S) = A$. Partiendo de la función identidad se van agregando puntos de control en los vóxeles donde el usuario quiere clasificar con un determinado color y una determinada absorción, dejando como resultado una función definida a trozos.

La clasificación del volumen puede ser aplicada de tres maneras. La diferencia entre cada una radica más que todo en el orden en que se realiza el proceso de clasificación e interpolación de las muestras[3].

1.3.1 Pre-Clasificación

La pre-clasificación consiste en aplicar la función de transferencia antes del despliegue. Así, todos los valores de los vóxeles del volumen van a ser sustituidos por su respectivo color y opacidad, acorde a la función de transferencia. Durante el despliegue se realiza la interpolación de los vóxeles ya clasificados. Como consecuencia, se pierden las altas frecuencias presentes en la función de transferencia [**14**], provocando una atenuación en la imagen.

[3] Interpolación de muestras: Es el proceso de reconstruir muestras que no existen partiendo de las que existen. Típicamente, se aplica una interpolación tri-lineal hasta entre 8 muestras existentes en el caso de volúmenes.

1.3.2 Post-Clasificación

Este método primero realiza la interpolación de las muestras escalares del volumen antes de la clasificación. En la **Ec. 1.1** no se tiene en cuenta el orden entre el proceso de muestreo y la clasificación. El color y absorción están en función de $\lambda \in [0, D]$, que es la parametrización de un rayo que atraviesa el volumen. Para ajustar la ecuación al proceso de post-clasificación hay que hacerle algunos ajustes:

$$C = \int_0^D c\left(s(x(\lambda))\right) t(s(x(\lambda))) e^{-\int_0^\lambda t\left(s(x(\lambda'))\right) d\lambda'} d\lambda \qquad \textit{Ec. 1.2}$$

Note que en esta ecuación, después de obtener por interpolación las muestras $s(x(\lambda))$, es que se le aplica la función de transferencia c y t.

Para facilitar la evaluación de la ***Ec. 1.2***, esta se discretiza. Primero se divide el rayo en secciones de longitud h. De esta forma, queda dividido en n segmentos, con $n = \left\lceil \frac{D}{h} \right\rceil$. Así:

$$C \approx \sum_{i=1}^{n} \int_{h_{i-1}}^{h_i} c(s(x(\lambda))) t(s(x(\lambda))) e^{-\int_0^\lambda t\left(s(x(\lambda'))\right) d\lambda'} d\lambda \qquad \textit{Ec. 1.3}$$

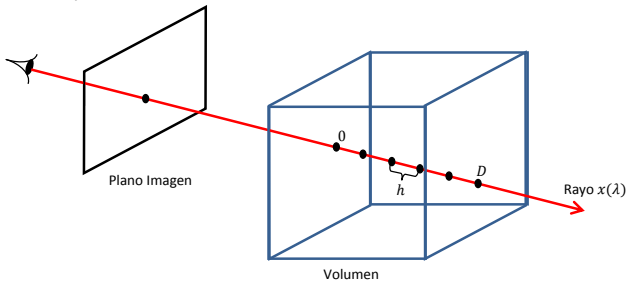


Figura 1.5*: Recorrido de un rayo de luz desde la cámara hasta salir del volumen ilustrando la* ***Ec. 1.3****. Se puede observar las muestras con una separación h entre cada una de ellas. Dicha separación comúnmente es constante.*

Comúnmente se asume que la emisión y la absorción son constantes en un intervalo $[h_i, h_{i+1}]$; por lo tanto se renombra $s(x(\lambda))$ como s_i. En este contexto $c(s_i)$ y $t(s_i)$ no dependen de la variable de integración λ. Siguiendo el trabajo de *Carmona* [**2**] se le hacen unos ajustes a la ecuación para que finalmente queda expresada de la siguiente manera:

$$C \approx \sum_{i=0}^{n-1} \prod_{j=0}^{i-1} e^{-ht(s_j)} c(s_i)(1 - e^{-ht(s_i)}) \qquad \textit{Ec. 1.4}$$

Si decimos que $c_i = c(s_i)$ y $\propto_i = 1 - e^{-ht(s_i)}$ la ecuación queda expresada de una manera más simple:

$$C \approx \sum_{i=0}^{n-1} \prod_{j=0}^{i-1} (1 - \propto_j) \propto_i c_i \qquad \textit{Ec. 1.5}$$

La ***Ec. 1.5*** puede ser evaluada utilizando el operador *under* u *over*. Se utiliza el operador digital *under* si se mezclan las muestras que se encuentran más cercanas a la vista hasta las más alejadas (*front to back*), en caso contrario se utiliza *over* (*back to front*).

Según *Ruijters et al.* [**15**] el operador *under* se define de la siguiente manera:

$$c := (1-\propto) \propto_i c_i + c$$
$$\propto := (1-\propto) \propto_i + \propto \qquad \textit{Ec. 1.6}$$

Y según *Lacroute et al.* [**9**] el operador *over* se define de esta forma:

$$c := \propto_i c_i + (1-\propto_i)c$$
$$\propto := \propto_i + (1-\propto_i) \propto \qquad \textit{Ec. 1.7}$$

En ambos casos c y $\propto$ son inicializados en 0. La técnica que se utiliza para explotar el *hardware* gráfico es texturizar polígonos con cortes del volumen. Comúnmente son desplegados desde el más lejano al más cercano al ojo utilizando el operador *over*, pero también hay técnicas donde el ordenamiento es inverso (operador *under*). Los operadores *over* y *under* están definidos en el API de OpenGL [**12**].

1.3.3 Clasificación Pre-Integrada

Al utilizar post-clasificación, primero se interpolan las muestras escalares antes de aplicar la función de transferencia. Pese a que está técnica no atenúa los colores como la pre-clasificación, es necesario aumentar drásticamente la frecuencia de muestreo del rayo para capturar todos los detalles de una función de transferencia que presenta altas frecuencias. Debido a que este aumento de la frecuencia de muestreo acarrea un tiempo de cómputo elevado, se suele muestrear el rayo a longitudes de por ejemplo ½ vóxel, generando bandas y puntos en lugar de una superficie continua [**14**].

Si se utiliza clasificación pre-integrada, en lugar de clasificar cada muestra individualmente, se clasifican en segmentos definidos entre cada par de muestras s_f, s_b. Para esto se pre-calcula la integral de la función de transferencia entre cada par de muestras potenciales (ver ***Figura 1.6a***). Este cálculo puede ser almacenado en una textura 2D, donde la coordenada (x, y) representaría la muestra (s_f, s_b) (ver ***Figura 1.6b***), es decir, el segmento que comienza en s_f y termina en s_b(ver ***Figura 1.6c***). El valor almacenado sería el color y opacidad obtenida de la integral. Si se realiza un muestreo adaptativo[4] se requerirá de una textura 3D para almacenar la tabla de pre-integración.

[4] Muestreo Adaptativo: La cantidad de muestras utilizadas puede variar en el trazo del rayo, es decir que la distancia h no es la misma en cada intervalo. El criterio utilizado para aumentar o disminuir la distancia entre cada par de muestra se determina por el nivel de detalle [**24**] [**23**] o por la opacidad acumulada [**2**].

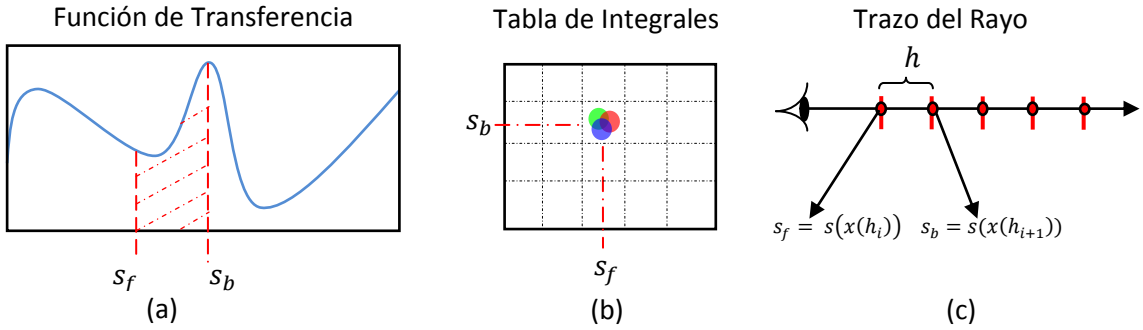


***Figura 1.6**: Despliegue de volúmenes utilizando clasificación pre-integrada. (a) Representaría el intervalo entre dos muestra en la función de transferencia. (b) son las integrales de cada canal (R, G, B, A) entre todos los posibles pares de muestras. (c) Una ilustración del rayo de luz en función de s_f, s_b.*

El objetivo de la pre-integración es ofrecer una solución numérica a la integral de la ***Ec. 1.3*** en cada segmento del rayo [**14**].

1.4 Técnicas de Despliegue de Volúmenes

Las técnicas de visualización más comunes actualmente son *Ray Casting*, *splatting*, *shear-warp,* planos alineados al objeto y planos alineados al *viewport*[5]. En todas ellas se puede utilizar clasificación pre-integrada, pre-clasificación o post-clasificación.

Ray Casting, *splatting* y *shear-warp* suelen ser implementadas por *software*, pero hay investigaciones donde se puede implementar utilizando el *hardware* gráfico [**16**]. A continuación se estudia la técnica de *Ray Casting*, la cual será implementada en este trabajo.

1.4.1 Ray Casting

Basándose en el trabajo de *Levoy* [**17**] esta técnica consiste en lanzar rayos que van desde el ojo del observador hacia cada píxel de la imagen, atravesando al volumen. Una vez que el rayo comienza la trayectoria por dentro del volumen, se muestrea el volumen a pasos de longitud constante (ver ***Figura 1.7***).

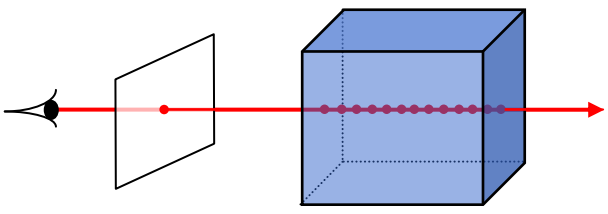


***Figura 1.7:** Ejemplo del trayecto de un rayo en Ray Casting.*

[5] *Viewport*: Se refiere al rectángulo 2D utiliza para proyectar la escena 3D.

El despliegue se puede acelerar mediante la utilización de las técnicas de terminación temprana del rayo y salto de espacios vacíos [**18**] [**19**]. La terminación temprana del rayo consiste en truncar la travesía del rayo cuando se acumula un umbral de opacidad definido por el usuario (comúnmente este umbral varía de 0.95 a 0.99). El resto se puede ignorar ya que el aporte a la imagen es mínimo. En el caso de saltos de espacios vacíos, se divide el volumen en bloques pre-calculando los valores mínimos y máximos de opacidad. Si un bloque es transparente, se salta al próximo bloque, pues no aporta información a la imagen final.

Esta técnica puede ser implementada utilizando el GPU mediante *Shaders*. Primero se crea un *bounding box*[6] proyectando las caras traseras y frontales en un buffer de color (ver ***Figura 1.8a***). Esto se hace para determinar la entrada y salida del rayo de visión en el volumen con respecto al observador, como se puede observar en la ***Figura 1.8b***. Teniendo por cada píxel de la imagen el punto de entrada y salida del rayo en el volumen, se calcula la dirección del rayo y se realiza el muestreo respectivo para evaluar la ecuación de visualización de volúmenes. Este último paso se realiza en el procesador de fragmentos [**16**].

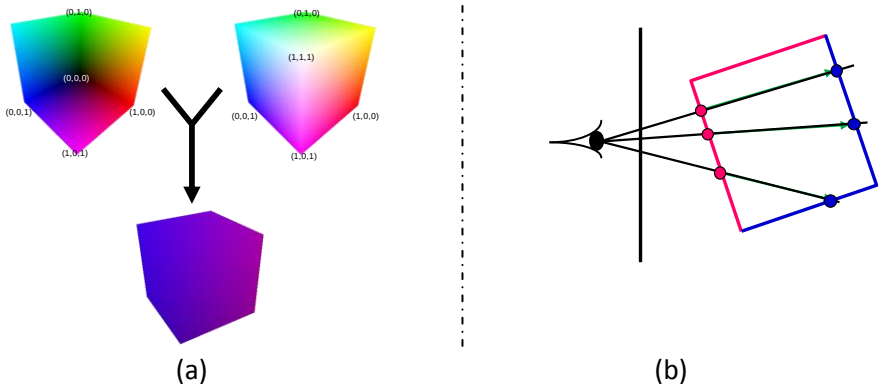


Figura 1.8: *Representación de Ray Casting basado en GPU. La figura (a) representa la resta de las caras traseras y delanteras del bounding box. Se obtiene como resultado una imagen que representaría el vector dirección del rayo de luz. En la figura (b) se puede observar el vector obtenido con las dos caras.*

[6] *Bounding box:* Define un volumen 3D en forma de una caja, cubriéndola completamente.

Capítulo 2: Despliegue de Volúmenes Multi-resolución

En los últimos años los dispositivos para capturar imágenes médicas han evolucionado considerablemente, generando volúmenes con resoluciones superiores a las capacidades del *hardware* gráfico. En ocasiones superan adicionalmente la capacidad de la memoria principal de un computador. A estos se le denominan volúmenes de gran tamaño. Existen proyectos como el del Humano Visible [**1**] y volúmenes producto de simulación [**20**] que cumplen estas características.

Para lidiar con estos volúmenes se han desarrollado diversas técnicas que permiten su despliegue en tiempo real. En esta investigación se discutirá brevemente algunas soluciones propuestas, tales como el particionamiento del volumen, técnicas para la visualización un área de interés y el enfoque multi-resolución.

2.1 Bricking

Es una técnica para particionar el volumen en pequeños subvolúmenes o ladrillos denominados *bricks*, los cuales, generalmente poseen el mismo tamaño y pueden ser desplegados por separado. Esto resulta de gran utilidad en volúmenes de gran tamaño que no pueden ser almacenados en la memoria de textura. A la hora de realizar el despliegue del volumen, lo que se hace por lo general es ordenarlos y desplegarlos desde el más lejano al más cercano al ojo de visión.

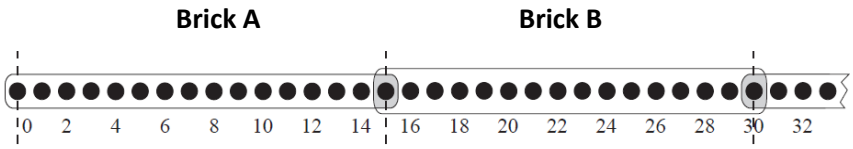


Figura 2.1: *Los puntos negros representan los píxeles. Se puede observar cómo se comparten en la frontera entre los bricks A y B*

Esta técnica tiene varias limitantes. Al momento de la interpolación en la frontera de los *bricks* se producen artefactos, ya que los bricks se despliegan de manera independiente, y todos los vóxeles requeridos para interpolación en la frontera no están disponibles. Para resolver este problema se comparte el píxel fronterizo entre bricks (ver ***Figura 2.1***). Así hay una consistencia en la interpolación. Esto tiene como desventaja que el tamaño del volumen aumenta y produce una sobre carga (*overhead*) en la memoria.

A pesar de que se resuelve la limitación de la memoria, esta técnica no puede ser utilizada en aplicaciones de tiempo real, debido a que el ancho de banda entre la memoria de textura y la memoria principal es limitado, empeora si el volumen es más grande que la memoria principal. Para aligerar este problema se evita cargar *bricks* vacíos, pero esto no garantiza todavía que los requerimientos de la memoria se satisfagan.

2.2 Despliegue de un Área de Interés

Esta técnica, también llamada *Volume Roaming* [**21**] consiste en mostrar solamente un área de interés que no sobrepase la capacidad total de la memoria de textura. Esta área puede ser seleccionada por el usuario de forma interactiva para luego cargarse y desplegarse. El área está definida comúnmente por un cubo, punto o un corte del volumen [**22**]. El problema de esta técnica es que al definir interactivamente el área de interés, se podría observar cierta lentitud provocada por la carga en memoria de textura del nuevo subvolúmen. Esto se puede disminuir utilizando *bricking* y coherencia *frame* a *frame*[7], así sólo son cargados en memoria los nuevos *bricks* que se necesiten y no todo el subvolúmen.

2.3 Técnicas de Multi-resolución

La mejor forma para visualizar volúmenes de gran tamaño en tiempo real es aplicando técnicas multi-resolución, que consiste básicamente en desplegar distintas áreas del volumen con distinto nivel de detalle. Se utiliza un determinado criterio para seleccionar los bricks a visualizar, de tal manera que no se exceda los requerimientos de memoria, pero cometiendo el menor error posible. A continuación, se estudia en qué consisten las técnicas multi-resolución y las implicaciones que trae: la representación del volumen, la compresión de datos para la generación de los niveles de detalle, el criterio para seleccionar el nivel de detalle y el proceso de despliegue del volumen.

El almacenamiento de los distintos niveles de detalles es una etapa fundamental en el despliegue de volúmenes de gran tamaño. En esta investigación se van a mencionar dos maneras de realizar este proceso, utilizando una jerarquía *octree*[8] y utilizando Bloques (también llamado *Flat Blocks* [**23**]*)*.

2.3.1 Jerarquía Octree

Consiste en representar el volumen jerárquicamente en una estructura *octree*, en donde el nodo raíz del árbol es un *brick* que representa todo el volumen y tiene el nivel de detalle más burdo. Cada 8 bricks de un nivel de detalle es aproximado por uno del siguiente nivel más burdo (ver ***Figura 2.2***); de esta forma las hojas del árbol representarían todo el volumen con su mayor nivel de detalle [**24**] [**25**].

[7] Frame: en español, cuadro de imagen, es una imagen en específico de todas las que componen una animación.
[8] *Octree*: es una estructura árbol de datos en la cual cada nodo interno puede tener como máximo 8 hijos.

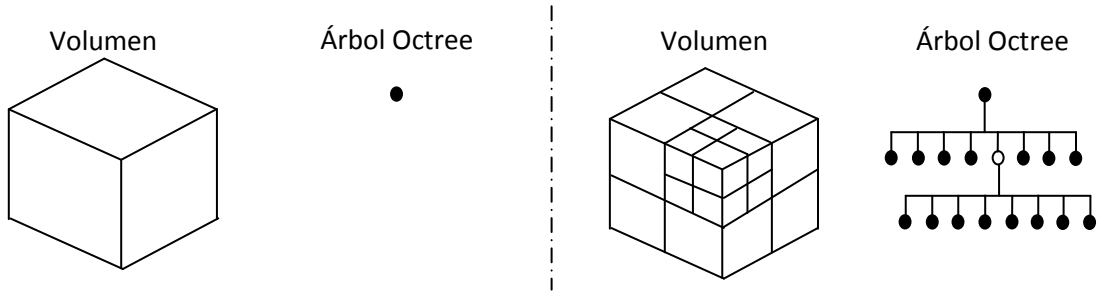


***Figura 2.2**: Representación de un mismo volumen, pero con dos representaciones. El de la izquierda representado por un brick con es el nivel más burdo, y el de la derecha con dos niveles de detalle.*

En esta técnica, la cantidad de vóxeles en cada *brick* es constante en todos los nodos del árbol, lo cual facilita la paginación. Los *bricks* de la frontera pueden requerir de algunos datos adicionales para completar el tamaño deseado.

2.3.2 Jerarquía por Bloques

Otra manera de realizar la representación es particionar el volumen en bloques [**26**] [**23**], donde cada bloque tiene su jerarquía local multi-resolución. Aquí, la cantidad de vóxeles varía por nivel de detalle (ver ***Figura* 2.3**). Caso contrario a una jerarquía *octree*, donde cada *brick* tiene la misma cantidad de vóxeles en todos los niveles del árbol.

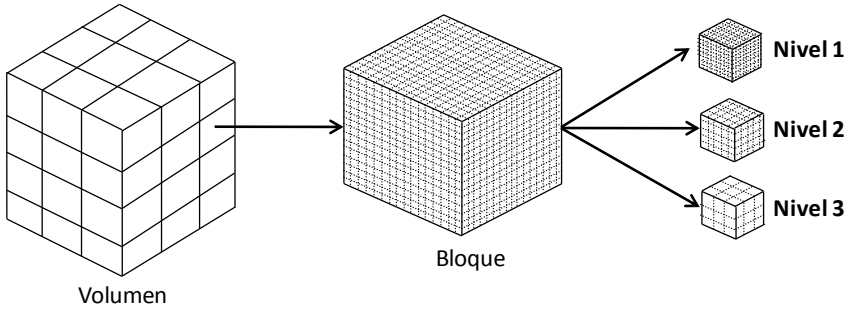


***Figura 2.3:** Ejemplo de una jerarquía por bloques. En un primer nivel tenemos la subdivisión en bloques del volumen, seguido se muestra los vóxeles de un bloque y tres niveles de detalles locales al bloque.*

Con esta representación se permite seleccionar efectivamente el nivel de detalle de los distintos bloques que van a ser mostrados, ya que cada bloque tiene su propia representación multi-resolución. En un árbol *octree,* se necesitaría cambiar el nivel de detalle en determinadas áreas, si se desea cambiar el refinamiento de un *brick*.

Para la generación de los niveles de detalle imaginemos una textura unidimensional de tamaño $n = 2^k \; para \; k \geq 1$, donde el nivel de detalle más fino es la textura original. El próximo nivel de detalle tiene $n/2$ píxeles, el siguiente $n/4$ píxeles y así sucesivamente hasta representar la

textura en un solo píxel teniendo en total $k + 1$ niveles de detalle, en la ***Figura 2.4*** se puede notar que con esta técnica el dominio de interpolación es cada vez más pequeño.

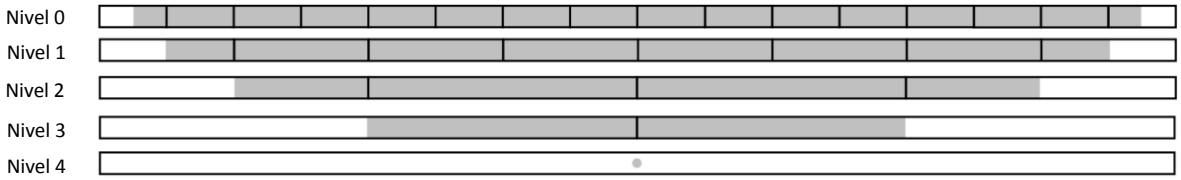


Figura 2.4: *Un ejemplo de los niveles de detalle de una textura 1D. El área gris representa el área de interpolación. El nivel 0 representa a la textura original y el 4 es el nivel de detalle más burdo. Así, cada píxel de un nivel de detalle representa dos píxeles de su nivel de detalle superior.*

En la ***Figura 2.5*** se puede notar que se alinearon los píxeles a los centros de las fronteras dando como consecuencia que un píxel de un nivel burdo representa más de dos píxeles del nivel de detalle superior dando como consecuencia la creación de un filtro muy complejo.

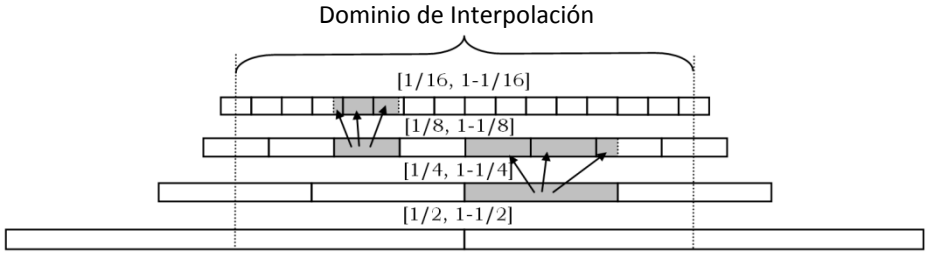


Figura 2.5: *Niveles de detalle alineados al dominio de interpolación, por cada píxel del nivel de detalle d intervienen 2 o 3 píxeles del nivel de detalle d–1.*

Para evitar pesados cálculos en el filtrado de píxeles, otra opción consiste en alinear las texturas agregando un píxel de holgura, en esta solución, la resolución de un nivel de detalle *d*, no es exactamente la mitad del nivel de detalle *d*–1 (ver ***Figura 2.6***).

$$tamNivel(i,n) = (n \; div \; 2^i + 1)^3 \qquad \textbf{Ec. 2.1}$$

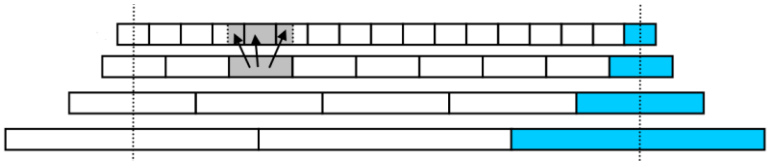


Figura 2.6: *Niveles de detalle agregando un píxel por cada nivel.*

En este caso, cada píxel del nivel *d* cubre un área de 2 píxeles del nivel *d*–1: el píxel central y dos medios píxeles (ver áreas grises). Los niveles 0, 1, 2 y 3 tienen 17, 9, 5 y 3 píxeles respectivamente siguiendo la ***Ec. 2.1***. En cada nivel de detalle *d* se utiliza el píxel extra (áreas azules) para el aporte del píxel fronterizo del nivel d–1 y adicionalmente es compartido con el primer píxel del siguiente bloque para el proceso de interpolación entre vecinos.

A pesar de que esta solución no requiere de muchos cálculos se requiere utilizar un píxel de holgura en cada nivel de detalle, para el caso 3d se puede ver en la ***Ec. 2.2*** que se requieren de $3 * n^2 + 3 * n + 1$ vóxeles de holgura por cada bloque.

$$\begin{gathered}(n+1)^3 - n^3 = \\ (n^3 + 3 * n^2 + 3 * n + 1) - n^3 = \\ 3 * n^2 + 3 * n + 1\end{gathered} \qquad \textit{Ec. 2.2}$$

Por ejemplo si se desea aplicar este esquema a un bloque de 32^3, se obtiene que se necesitan $3 * 32^2 + 3 * 32 + 1 = 3169$ vóxeles de holgura para el nivel más fino. Sumando los niveles de detalle, que para dicho ejemplo seria $17^3 + 9^3 + 5^3 = 5767$ vóxeles. Da como resultado que para un bloque de 32^3 se necesitan gastar 8936 vóxeles en ajustes, que es aproximadamente un 27% de memoria adicional.

2.4 Métricas de Error

A menudo, el criterio de selección de los niveles de detalle está determinado por varios parámetros especificados por el usuario, comúnmente basándose en cierta información que dependen de los datos [**27**] [**28**] [**29**], o que depende de la vista [**26**] [**24**] [**30**] o ambos [**31**] [**32**] [**33**] [**34**]. En general estas métricas pueden ser clasificadas en dos, basadas en los datos (*Data-based Metrics*) y basadas en la imagen (*Image-based Metrics*).

2.4.1 Basadas en los Datos

En la métrica basada en los datos se mide la distorsión de los datos del volumen en los distintos niveles de detalle. Los más utilizados son el error cuadrático medio y la norma [**27**] [**29**]. Estas métricas tienen un claro significado físico y son fáciles de calcular. Sin embargo, no suelen ser eficaces en la predicción de la calidad de las imágenes mostradas, debido a la falta de correlación entre los datos del volumen y la imagen resultante.

Laur et al. [**27**] generan una estructura que denominan pirámide. En ella se almacena el error medio de la aproximación. Este error es basado en los datos ya clasificados. Cada celda de la pirámide contiene el valor medio $RGBA$ de los hijos. Dicha pirámide es creada cada vez que se hace un cambio en la función de transferencia. Para calcular el error utilizaron la siguiente ecuación:

$$e(n_j) = \sqrt{\frac{\sum s_i^2}{n_j^l} - (\frac{\sum s_i}{n_j^l})^2} \qquad \textit{Ec. 2.3}$$

donde $e(n_j)$ es el error asociado al nodo j, n_j^l es el número de vóxeles de la región de nivel de detalle l, y s_i es el valor del vóxel clasificado. Este error se puede calcular eficientemente con sólo un recorrido de la pirámide.

Boada et al. [**28**] utilizan una jerarquía *octree*, donde en cada nodo se obtiene un error que ellos llaman error nodal. A medida que se calcula el error, desde las hojas hasta la raíz (recorrido

bottom-up), si un nodo padre puede representar a los hijos sin producir un error de aproximación, sus hijos serán descartados del despliegue. El cálculo del error lo realizan de la siguiente forma:

$$e(n_j) = \frac{\sqrt{\frac{1}{2^{3(l_{max}-k)}} \sum_{i=1}^{2^{3(l_{max}-k)}} (s_i(v) - f_{n_j}(v))^2}}{s_{max}} \qquad \textit{Ec. 2.4}$$

donde f_{n_j} es la interpolación trilineal de las esquinas del nodo n_j y $s_i(v)$ son los vóxeles de los nodos hijos, k es la altura del nodo en el árbol *octree*, l_{max} la altura máxima del árbol y s_{max} es la cantidad total de vóxeles. Cabe destacar que el error producido es muy elevado debido a que sólo se utilizan las 8 esquinas de un *brick*, sin tener en cuenta los demás vóxeles del *brick* que pueden aportar más información relevante.

Gao et al. [**29**] utilizaron de igual forma una jerarquía *octree*, aplicando compresión por *wavelets*. Basándose en el error cuadrático medio, ellos proponen un método para calcular el error de forma progresiva, recorriendo el árbol desde las hojas hasta la raíz. El error en el nodo n_j viene dado por:

$$e(n_j) = \frac{\sum_{i=0}^{7} (\sum_{v \epsilon V} (s_i(v) - f(v))^2)}{8n} + max_e \qquad \textit{Ec. 2.5}$$

donde v representa a un vóxel del volumen, $s_i(v)$ son los vóxeles de los nodos hijos, $f(v)$ es la muestra interpolada mediante *wavelets*, n es la cantidad de vóxeles por nodo y max_e es el máximo error entre los 8 hijos (ver ***Figura 2.7***). Inicialmente las hojas tienen un error $e = 0$.

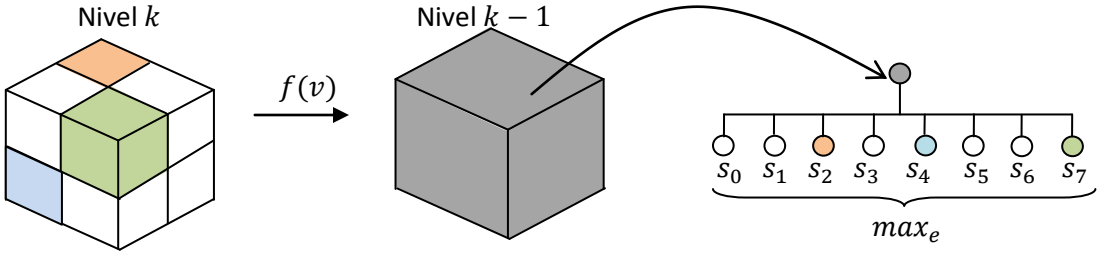


Figura 2.7*: Representación de una aproximación utilizando una jerarquía octree, ilustrando la* ***Ec. 2.5****.*

Ljung et al. [**34**], a diferencia de los trabajos anteriores, utilizan 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*) [**35**], y calculan el error cuadrático medio en base a la función de transferencia, es decir, a partir de los vóxeles ya clasificados. Esta técnica tiene como desventaja que debe re-calcularse el error cada vez que el usuario haga un cambio en la función de transferencia.

El error cuadrático medio se calcula entre los vóxeles de los bloques clasificados de cada par de niveles de detalle. Para reducir cálculos hicieron uso de un histograma de frecuencia de los vóxeles del bloque, reducido a solo 10 segmentos. Donde el eje x representa los vóxeles del volumen y el eje y representa la frecuencia del vóxel en el bloque. Así, para cada vóxel, se multiplica su distorsión por su frecuencia en la correspondiente entrada del histograma; con esto se elimina un cálculo redundante para hallar el error de todo el volumen.

Wang et al. [**36**], al igual que el *Ljung* [**34**], utilizaron un histograma de frecuencia pero de 256 segmentos; también tienen en cuenta la covarianza entre cada par de bloques e incluyeron el valor medio y la varianza en cada bloque. Adicionalmente para poder actualizar el error en cada bloque utilizaron otras tablas, incluyendo un histograma 2D llamado "*Summary Table*" o tabla de sumarización, y una tabla que almacena la diferencia en el espacio de color CIELuv entre cada par de muestras. Reportaron que realizar cambios en la función de transferencia requiere de varios segundos debido a la actualización de todas las tablas.

2.4.2 Basadas en la Imagen

Estas métricas se centran en la imagen final que percibe el usuario. Trata de captar la calidad perdida en las imágenes desplegadas. Esto es muy costoso ya que además de la percepción humana, hay que tener en cuenta el tiempo de ejecución, la proyección y la oclusión. Además hay que recordar el usuario puede hacer cambios en la función de transferencia o cambiar el punto de interés. Se han desarrollado diversas técnicas en esta área (*Li et al.* [**26**], *Guthe et al.* [**32**] y *Wang et al.* [**36**]), pero en el presente trabajo no se considero el espacio imagen.

La métrica de error utilizada es una etapa fundamental para poder obtener un criterio de selección adecuado, cada autor propone un criterio de selección diferente. A continuación se procederá a explicar brevemente algunos de ellos.

2.5 Criterio de Selección

El criterio de selección es una etapa importante. Aquí se determina cuales elementos van a requerir un mayor refinamiento. Esto debe hacerse de tal manera que no se pierda la interactividad con la aplicación, teniendo en cuenta las limitaciones del *hardware*. Adicionalmente, se debe tener un mayor refinamiento en el área de interés, pero sin provocar demasiada perdida de información en el resto del volumen. Existe una serie de técnicas para realizar este proceso, unos toman parámetros como la posición del ojo, otros un área o punto de interés dentro del volumen, algunos son basados en los datos, como la homogeneidad del *brick*. En las nuevas investigaciones han tratado de utilizar el espacio imagen como ya se ha visto anteriormente. A continuación se mencionarán con más detalles cada una de estas técnicas.

Chamberlain et al. [**37**] proponen un método de selección que se basa en la resolución proyectada para escenas estáticas complejas utilizando una jerarquía *octree.* Básicamente la técnica consiste excluir los nodos del *octree* que estén fuera de la pirámide de visualización (*frustum*) [**38**]. Si está dentro de la pirámide, bien sea parcial o totalmente, se proyecta el *bounding box*; si dicha proyección es menor a un determinado épsilon, se despliega con el color y opacidad de la celda contenida en el *bounding box*, y en caso contrario se subdivide y se aplica el criterio recursivamente. Se puede observar que no se tienen en cuenta la capacidad máxima de la memoria de textura.

LaMar et al. [**24**] utilizaron dos criterios. Se basaron en la distancia que hay con respecto a un punto de interés (ver ***Figura 2.8a***) y la relación entre el ángulo de visión con respecto a la diagonal proyectada del *brick*. Lo que se hace es recorrer el o*ctree* en pre-orden, si el nodo esta

fuera de la pirámide de visualización es excluido, de lo contrario, se verifica recursivamente hasta que se cumpla una de estas dos condiciones:

- La distancia desde el punto de interés al centro del *brick* debe ser mayor a la diagonal del *brick* y el ángulo proyectado del *brick* tiene que ser menor que la mitad del ángulo de visión (ver ***Figura 2.8b***).

- Se alcance un nodo hoja.

Si se observa, este refinamiento recursivo tampoco toma en cuenta la limitación de la memoria de textura.

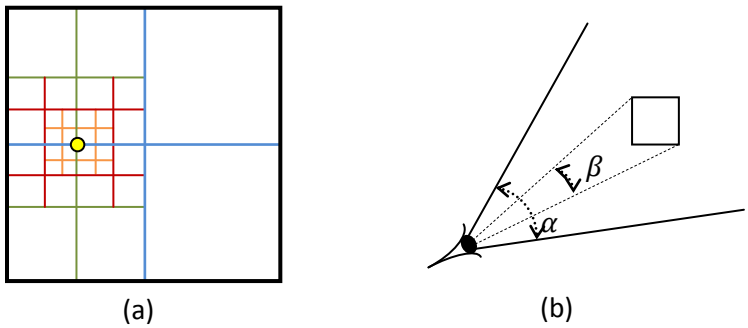


Figura 2.8*: Criterio de selección basándose en la distancia al punto de interés y el ángulo de proyección del brcik. En la imagen (a) se puede ver una representación de lo que sería un octree en 2D. En este ejemplo, se seleccionaron sólo 34 bricks, cuando su representación más fina es de 256 bricks. En la imagen (b) se puede ilustrar la representación del ángulo proyectado del brick β y el ángulo de visión α.*

Boada et al. [**25**] al igual que *Li et al.* [**26**], tienen en cuenta la importancia que define el usuario mediante un área de interés. También tienen en cuenta otros parámetros como la homogeneidad del bloque y la capacidad de la memoria de textura. La idea es buscar maximizar la cantidad de bloques homogéneos, tratando de obtener un bloque de alta coherencia espacial con el mayor tamaño posible. Esto es para disminuir la cantidad polígonos en el despliegue. Para seleccionar el nivel de detalle de cada bloque, utilizan la predicción de velocidad de despliegue, la opacidad, la distancia al punto de vista y el tamaño del bloque proyectado.

Guthe et al. [**33**] utilizan una cola de prioridad para limitar la cantidad de vóxeles que van a ser desplegados, esto con la finalidad de no superar la capacidad de la memoria de textura. Van a tener mayor prioridad los *bricks* que estén más cercano al ojo y que difieran más de su representación más fina (esto se determina utilizando la métrica de error vista anteriormente). El *brick* con la mayor prioridad será reemplazado por sus hijos. Este proceso se repetirá hasta alcanzar la capacidad máxima de la memoria de textura o hasta que no sea posible refinar más *bricks*.

Plate et al. [**39**] consideran la capacidad de la memoria de textura y la cantidad de *bricks* que se pueden cargar por cada *frame*. Primero se seleccionan los *bricks* basándose en la posición del visor; sólo se permite seleccionar *bricks* que difieran como máximo un nivel de detalle con respecto a sus vecinos. Posteriormente se determina cuántos *bricks* se cargan entre un *frame* y

otro. De esta manera se van cargando los niveles más burdos primero, para luego ir refinando hasta alcanzar el umbral deseado que no exceda la capacidad de la memoria de textura.

Carmona [**2**] se basó en un algoritmo utilizado inicialmente en terrenos multi-resolución. Este algoritmo voraz incremental, llamado algoritmo de *Split-and-Collapse*, utiliza la coherencia *frame* a *frame* para actualizar la representación multi-resolución del volumen. La cantidad de *bricks* que se transfieren al GPU en un *frame* dependerán del ancho de banda requerido. Utiliza una cola de prioridad que indica la próxima operación de refinamiento (*Split* o corte) y reducción (*Collapse* o colapso) a efectuar para lograr la representación deseada.

Carmona introdujo un algoritmo óptimo que determina la selección con mínimo error. Aunque el algoritmo puede requerir de varios minutos en ejecutarse por cada *frame*, se utilizó para demostrar que su algoritmo voraz genera resultados cercanos al óptimo. Es importante señalar que se toma en cuenta la distancia a un punto o área de interés, la distancia al visor, el error de los vóxeles clasificados con respecto a su representación más fina, y las limitaciones de *hardware,* como el ancho de banda y la capacidad de la memoria de textura.

Algunos trabajos obtuvieron mejores resultados que otros, pero de igual forma en cada trabajo se reporta que hay gran presencia de artefactos. Por lo tanto hay que buscar alguna forma de minimizarlo aun más.

2.6 Reducción de Artefactos en Fronteras

Una vez seleccionado los *bricks,* se pasa al proceso de despliegue. En esta etapa se interpolan las muestras para reconstruir todo el volumen. Los bordes de los *bricks* de distintos tamaños podrían representarse inadecuadamente, por lo tanto se producirá un error o artefacto visual por el cambio brusco del color y opacidad entre dos *bricks* de distinto nivel de detalle.

Una técnica propuesta por *Kurt et al.* [**40**] fue cambiar el vóxel de la frontera por el de la interpolación entre ambos *bricks*. De esta manera, cuando se interpolan las muestras entre niveles de detalle que difieran a lo sumo un nivel, se lograre ver una continuidad (ver ***Figura 2.9a***). Este procedimiento hay que aplicárselo a las seis caras del *brick. Guthe et al.* [**33**] proponen en su lugar utilizar sólo 3 caras (x_{max} , y_{max} , z_{max}). Como en cada *brick* se realiza el mismo proceso, todas las fronteras quedaran correctamente interpoladas (ver ***Figura 2.9b***).

Ljung et al. [**23**] proponen una técnica llamada *interblock interpolation* (interpolación entre bloques) que evita la duplicación o pre-cálculo de los vóxeles de frontera. Su técnica consta de dos etapas. En la primera se despliegan los vóxeles que no forman parte de la frontera del *brick,* esto se hace de la forma convencional. En la segunda etapa, para desplegar los vóxeles de la frontera utiliza un programa de fragmento complejo. En este se utilizan todas las muestras de la frontera de los vecinos para realizar una interpolación entre bloques (ver ***Figura 2.9c***). Reportan que su método es 4 veces más lento que la versión sin interpolación entre bloques.

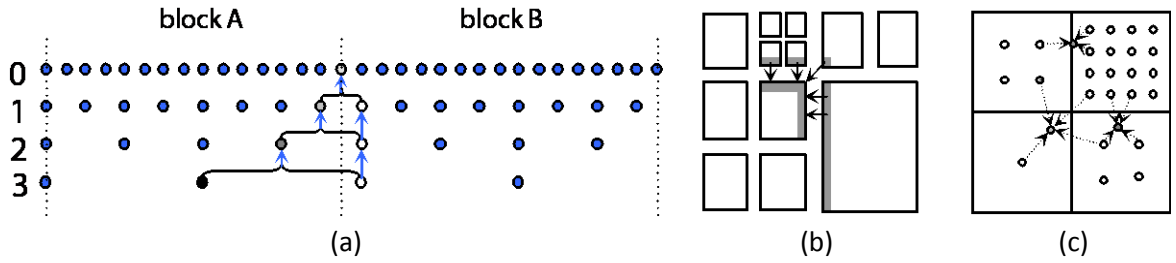


Figura 2.9: *Aquí se puede observar los ejemplos de las técnicas utilizadas por Kurt (a) [**40**], Guthe (b) [**33**] y Ljung (c) [**23**]. En la figura C, se señalan las muestras obtenidas por la interpolación de las otras muestras que pertenecen a la frontera de los bloques vecinos.*

LaMar et al. [**22**] utilizan una jerarquía *octree,* donde los niveles de detalle entre *bricks* adyacentes no varía en más de un nivel. Plantean trazar un plano de corte para posteriormente seleccionar los *bricks* que se intersectan. Por cada uno de los *bricks* que se van a desplegar, se calcula el polígono que se forma al intersectarse con el plano de corte. Luego se estudian los vecinos (caras, bordes y esquinas) del *brick*, marcando cada vértice de su *bounding box*. Si el vértice está en contacto con algún *brick* de menor resolución se marcara con un 0, caso contrario se marca con un 1. Los vértices del polígono que corta al *brick* coinciden obligatoriamente con alguno de los bordes o esquinas del *brick* (ver ***Figura 2.10a***).

A cada polígono se le asignan dos texturas, la de mayor y menor nivel de detalle. Utilizando interpolación lineal entre los vértices del *bounding box*, se ponderan los colores de los vértices del polígono con valores entre 0 y 1. Luego se realizará el despliegue de la textura del *brick* sobre el polígono, multiplicando los téxeles por los pesos de cada fragmento del polígono. Al darle valores entre 0 y 1 a los vértices, el efecto que se logra es atenuar el aporte final que tiene la textura cuando se está cerca de 0 y aumentarlo cuando esta cerca de 1.

Para el despliegue, primero se utiliza la textura con mayor resolución. Donde hay un 1 será totalmente opaco y donde exista un 0 será transparente. Luego, se hará lo mismo pero con la textura de menor resolución, invirtiendo la ponderación. De esta manera donde hay un 1 será totalmente transparente y donde hay un 0 será opaco. Las dos imágenes generadas son, finalmente, sumadas para formar la imagen final. Este será un cambio de nivel de detalle difuminado y suave entre *bricks* de niveles de detalle diferentes (ver ***Figura 2.10b***).

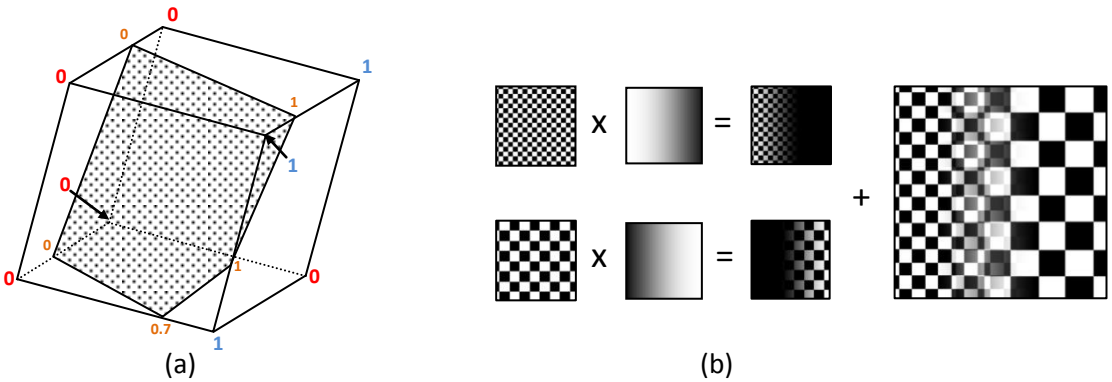


Figura 2.10: *Interpolación de niveles de detalle dentro del mismo brick. En la imagen (a) se muestra como son marcados los vértices del polígono que corta al brick, por interpolación lineal se consigue el valor de cada uno de ellos. En la imagen (b) se muestra como es el proceso de mezcla de las dos texturas existentes dentro de cada brick.*

Carmona et al. [**41**] generalizan este método para ser aplicado en el volumen, y no únicamente en un plano de corte. Esto es posible gracias a la capacidad de programación de las tarjetas gráficas. La idea es mezclar un brick fino con su padre p utilizando la siguiente ecuación (ver ***Figura 2.11***):

$$blend(\beta) = (1 - \beta) * x + \beta * p \qquad \textit{Ec. 2.6}$$

donde β es la interpolación trilineal de los pesos que se asignan a los vértices del *brick*, acorde al nivel de detalle de sus vecinos. Note que $blend(\beta)$ es una interpolación quatri-lineal, ya que es generada a partir de una interpolación tri-lineal de las coordenadas de textura x y p. Si $\beta = 0$ se obtendría que $blend(\beta) = x$, pero si $\beta = 1$ entonces $blend(\beta) = p$. Esto demuestra que se garantiza una transición suave entre bloques adyacentes. Sus resultados reportaron que se requiere un 20% de cómputo adicional y que hay una sobrecarga en memoria de textura del 10%.

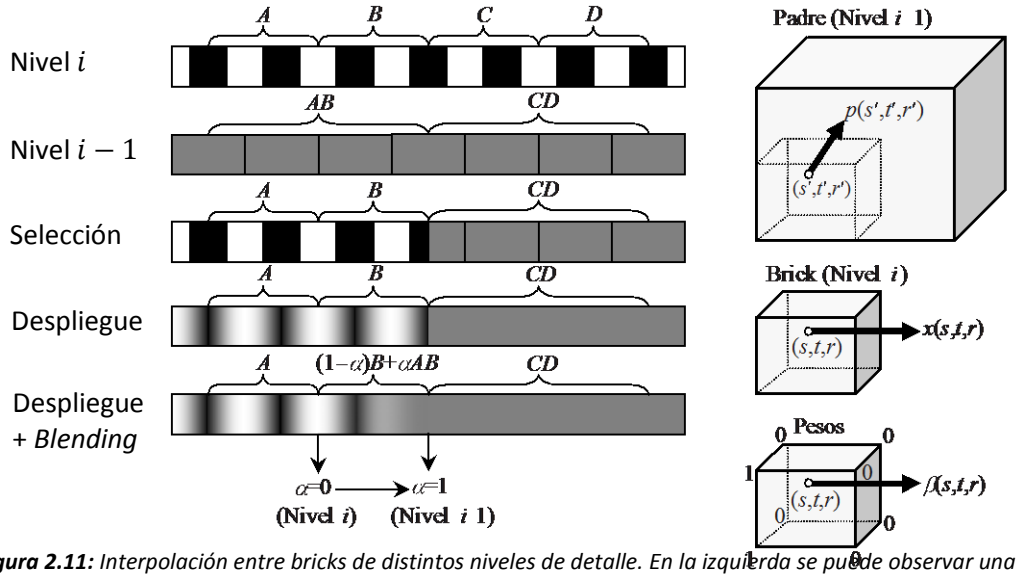


Figura 2.11: *Interpolación entre bricks de distintos niveles de detalle. En la izquierda se puede observar una representación 1D de dos niveles de detalle consecutivos y la selección para el despliegue. Sin blending se observa que hay un fuerte artefacto visual. Aplicando el blending entre dos niveles de detalles, se puede observar la suave transición. A la derecha se puede observar el caso 3D explicado en la* ***Ec. 2.6****.*

2.7 Proceso de Despliegue

En el proceso de despliegue, comúnmente, los *bricks* seleccionados son ordenados y desplegados en sentido *back to front.* Se mezclan utilizando el operador *over*. Posteriormente se suele aplicar la técnica de planos alineados al objeto, al *viewport*, conchas esféricas o *Ray Casting* basado en GPU. Para el caso de *Ray Casting* los *bricks* se despliegan en sentido *front to back* y se aplica el algoritmo en cada *brick* seleccionado por separado. Se puede aprovechar aplicar técnicas como terminación temprana del rayo y saltos de espacios vacios [**42**].

Lux et al. [**43**] implementaron una forma diferente de almacenar el conjunto de texturas multi-resolución que denominaron *atlas*. Su sistema implementa un árbol *BSP* (*Binary Space*

Partitioning o Partición Binaria del Espacio) para realizar el despliegue de varios volúmenes en una misma escena. En cada uno de los volúmenes utilizaron una jerarquía *octree*.

Se construye una sola textura que es almacenada en el GPU (*atlas*). En ella se organizan todos los *bricks* de todos los volúmenes seleccionados para el despliegue. Adicionalmente, crean una textura de índices para identificar en qué posición del *atlas* buscar un *brick* (ver ***Figura 2.12***). Posteriormente utilizaron *Ray Casting* basado en GPU, aplicando algunos ajustes para poder visualizar más de un volumen al mismo tiempo.

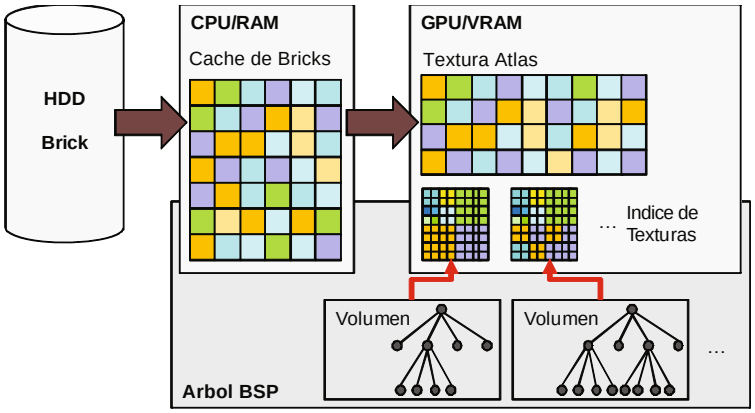


Figura 2.12*: El diseño de lo que sería una textura atlas.*

Cabe recordar que en una jerarquía *octree* cada *brick* ocupa el mismo tamaño en memoria sin importar el nivel de detalle que tenga. Por lo tanto no va haber segmentación en la memora de textura ni en memoria principal a la hora de reemplazar *bricks*.

Como se puede observar hay distintas formas de realizar el despliegue de volúmenes multi-resolución. Sólo hay que tener en cuenta cuáles son los resultados esperados y saber que cada técnica tiene sus requerimientos de procesamiento que genera distintos tiempos de respuesta. Adicionalmente, la calidad de los resultados finales puede variar técnica a técnica.

Capítulo 3: Programación Paralela

La mayoría de procesadores actuales incorporan varios núcleos de procesamiento y el número de núcleos es mayor conforme avanza el tiempo. Por otro lado, cada vez es más frecuente tener acceso a computadores de memoria distribuida tipo clúster, formadas por varios nodos o procesadores.

Para paralelizar una aplicación es necesario contar con un API de desarrollo que brinde las herramientas necesarias para esto. Dependiendo de la herramienta con que se cuente, se particiona el código en partes para que se ejecute en paralelo en varios procesadores. De de aquí surge el término de granularidad.

La granularidad es el tamaño de las partes en que se divide una aplicación. Dichas partes puede ser una sentencia de código, una función o un proceso en sí que se ejecutara en paralelo. Esta es categorizada como paralelismo de grano fino y paralelismo de grano grueso.

Paralelismo de grano fino es cuando el código se divide en una gran cantidad de partes pequeñas. Se le conoce además como Paralelismo de Datos.

Paralelismo de grano grueso es a nivel de subrutinas o segmentos de código, donde las piezas son pocas y de cómputo más intensivo que las de grano fino. También se le conoce como Paralelismo de Tareas. Usualmente para este tipo de paralelismo se usan equipos de memoria distribuida tipo clúster.

La programación paralela nos ofrece la posibilidad de aprovechar al máximo los recursos que los nuevos procesadores poseen. Adicionalmente, existen tecnologías como MPI (*Message Passing Interface*) y OpenMP (*Open Multi-Programming*) que nos ayudan a explotar al máximo todos estos núcleos del procesador.

La programación paralela actualmente también ha tenido un gran avance en las tarjetas gráficas. La programación de los procesadores gráficos, es comúnmente referido como GPGPU (*General-Purpose Computing on Graphics Processing Units*): este es un concepto reciente dentro de la informática que trata de estudiar y aprovechar las capacidades de cómputo de un GPU. Es famoso debido a sus operaciones de punto flotante que son increíblemente rápidos, y por su diseño paralelo [**10**] [**11**] [**12**] [**44**]. El GPU también tiene sus limitantes. La más importante es la transferencia desde la memoria principal a la memoria de textura del GPU, que es extremadamente lenta en comparación con la transferencia de datos dentro del GPU. Adicionalmente, la memoria en el GPU tiene una capacidad limitada, por lo tanto es usado comúnmente para atacar problemas de granularidad fina. Esto debe tenerse en cuenta para evitar cuellos de botella no deseados.

Actualmente hay una diversa cantidad de APIs para la utilización del GPU en aplicaciones de propósito general. Unas de ellas son OpenCL (Open Computing Language) del Grupo Khronos [**44**] y CUDA (Compute Unified Device Architecture) de NVidia [**10**].

A fines del alcance de este trabajo especial de grado a continuación solo se describirá OpenMP para realizar programar paralelos sobre máquinas con varios núcleos de procesadores, y CUDA, para la programación de GPUs.

3.1 OpenMP

OpenMP es una interfaz de programación de aplicaciones (API) para la programación multiproceso de memoria compartida en múltiples plataformas. Permite añadir concurrencia a los programas escritos en C, C++ y Fortran. Está disponible en muchas arquitecturas, incluidas las plataformas de Unix y Microsoft Windows. Se compone de un conjunto de directivas de compilador, rutinas de biblioteca, y variables de entorno que influencian el comportamiento de un programa en tiempo de ejecución [**45**].

Definido conjuntamente por un grupo de proveedores de hardware y de software, OpenMP es un modelo de programación portable y escalable que proporciona a los programadores una interfaz simple y flexible para el desarrollo de aplicaciones paralelas para las plataformas que van desde las computadoras de escritorio hasta las supercomputadoras.

3.1.1 Modelo de ejecución

OpenMP se basa en el modelo *fork-join* (ver ***Figura 3.1***), 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 solo resultado (*join*).

La sintaxis básica que nos encontramos en una directiva de OpenMP es:

```
#pragma omp <directiva> [cláusula [ , ...] ...]
```

Cuando se incluye una directiva OpenMP esto implica que se incluye una sincronización obligatoria en todo el bloque. Es decir, el bloque de código se marcará como paralelo y se lanzarán hilos según las características de la directiva. Al final de ella habrá una barrera para la sincronización de los diferentes hilos (salvo que implícitamente se indique lo contrario).

3.1.2 Contexto de Ejecución

Todo hilo tiene un contexto de ejecución, que consiste del espacio de direcciones que el hilo puede acceder. Este contexto de ejecución incluye variables estáticas, estructuras dinámicamente asignadas, y variables en la pila de ejecución. Una variable compartida tiene la misma dirección en el contexto de ejecución de cada hilo, las variables privadas tienen una dirección distinta en el contexto de ejecución de cada hilo. Un hilo no puede acceder las variables privadas de otro hilo.

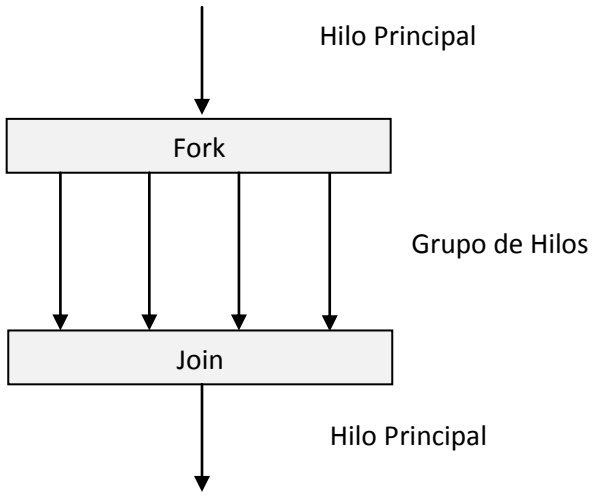


Figura 3.1: *Modelo fork-join. El programa comienza con un hilo de ejecución, el hilo inicial. Un grupo de hilos es creado por el hilo principal para trabajar en paralelo y al final de su ejecución se suspenden [**46**].*

3.1.3 Modelo de Memoria

En los modelos de programación paralela para memoria compartida las tareas tienen un espacio de memoria común, donde leen y escriben asincrónicamente. El acceso a memoria es controlado utilizando mecanismos tales como: semáforos, monitores, etc. Para estos modelos de programación no es necesario especificar explícitamente la comunicación de los datos entre las tareas productoras y las tareas consumidoras, por lo que el desarrollo de los programas se simplifica.

Las comunicaciones en los sistemas de memoria compartida tienen baja latencia y grandes anchos de banda. La portabilidad de estos modelos es pobre, pero resultan fáciles de programar comparados con los modelos de programación para memoria distribuida. OpenMP provee un modelo de memoria compartida como se muestra en la ***Figura 3.2***. Los hilos interactúan uno con otro mediante variables compartidas.

Hay que tomar en cuenta que el uso inadecuado de las variables puede originar secciones críticas. Para controlar esto es necesario el uso de sincronización para evitar el conflicto entre los datos. Sin embargo, la sincronización es costosa computacionalmente. Para minimizar la necesidad de sincronización hay que establecer la mejor forma de almacenar los datos, como lo es guardar datos de modo que sean locales al hilo que los accede o crear una estructura jerárquica para acceso paralelo de los datos.

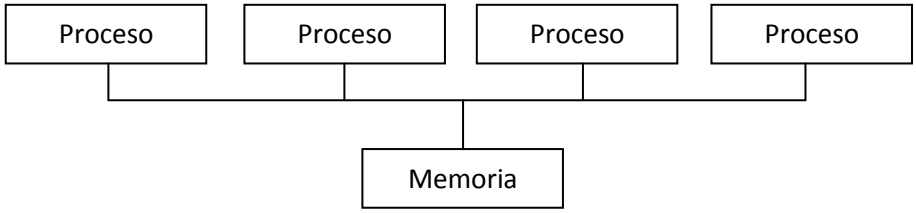


Figura 3.2: *Memoria compartida entre procesadores [**46**] .*

3.2 CUDA

CUDA son las siglas de *Compute Unified Device Architecture* (Arquitectura de Dispositivos de Cómputo Unificado), esta sólo está disponible para las ultimas tarjetas gráficas de NVidia superior o igual a la serie GeForce 8 [**10**] está basado en la sintaxis del lenguaje C [**47**] que ellos denominan *C for CUDA*. Proveen un SDK (*Software Development Kit*) y API para explotar las ventajas de las GPUs frente a las CPUs, utilizando el paralelismo que ofrecen sus múltiples núcleos, permitiendo ejecutar una gran cantidad de hilos de ejecución simultáneamente.

Si una aplicación está diseñada para utilizar numerosos hilos que realizan tareas independientes, un GPU podrá ofrecer un gran rendimiento. Permite la programación heterogénea; esto significa que las aplicaciones pueden usar tanto el CPU como el GPU para los algoritmos implementados. En la ***Figura 3.3*** se puede observar que CUDA se integra fácilmente a distintos lenguajes de programación, provee el soporte para el API de OpenCL y que tiene una serie de capas de abstracción para el manejo paralelo de los dispositivos gráficos NVidia.

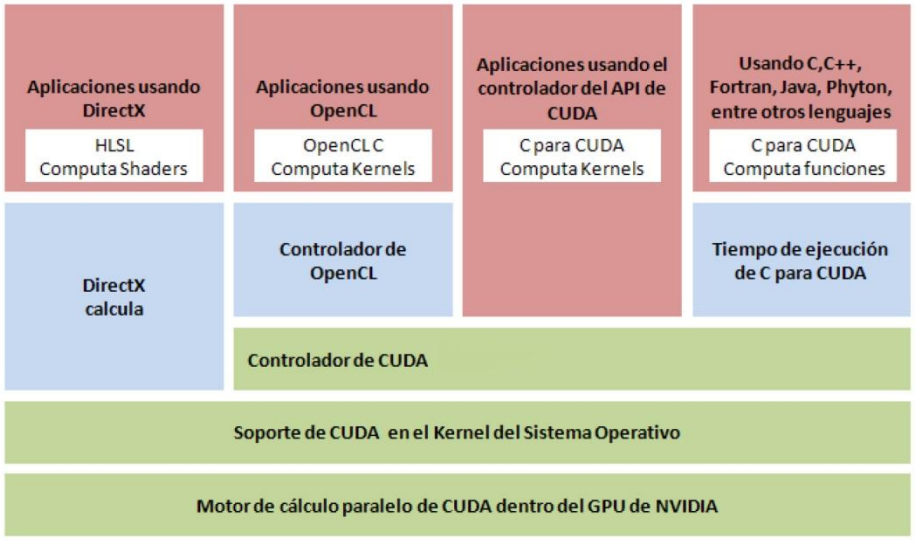


Figura 3.3*: Arquitectura CUDA [**48**].*

3.2.1 Procesamiento

El API de CUDA [**49**] se constituye por un *host* (CPU) que se conecta a un *device* (GPU ó CPU), este se encarga de realizar los cálculos que le asigno 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, casi sin costo alguno, 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 (ver ***Figura 3.4***). La idea es compartir datos y sincronizar los hilos por conjuntos. En este documento nos referiremos a los bloques de CUDA como "*blocks"* y a los bloques de la jerarquía multi-resolución como "bloques".

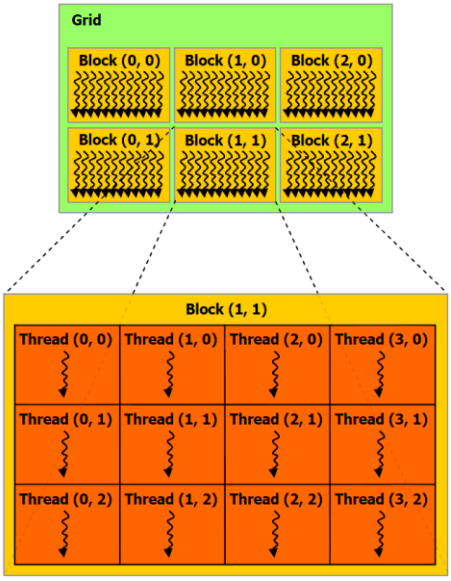


Figura 3.4*: Representación de los threads, blocks y grids [**49**]*

3.2.2 Manejo de la Memoria

El espacio de memoria del *host* y el *device* es separado. El *host* es el encargado de manejar 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* y los *threads* tienen su propia memoria privada (ver ***Figura 3.5***).

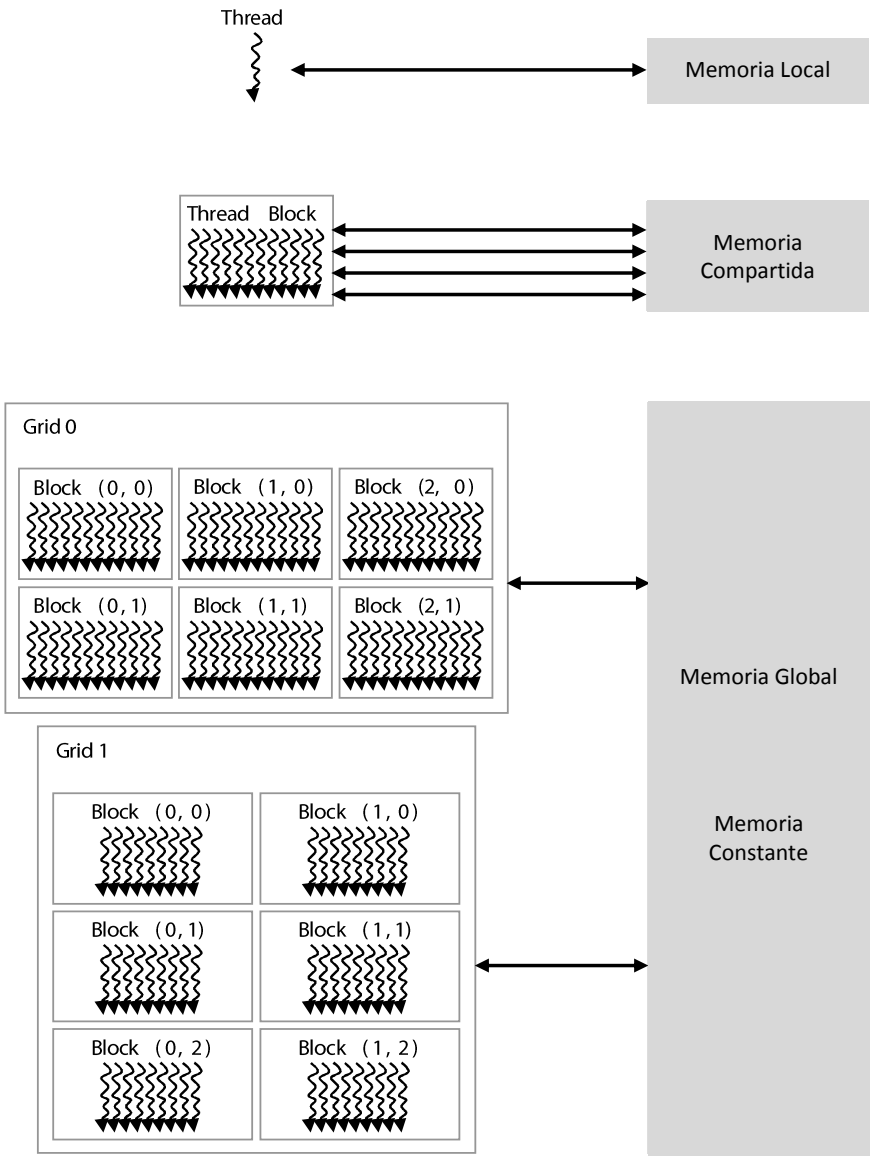


***Figura 3.5**: Arquitectura conceptual del manejo de memoria de CUDA [**49**].*

- **Memoria Global**: En esta, memoria se permite acceso de lectura/escritura a todos los *threads* en todos los *grids*. Los *threads* pueden leer y escribir cualquier elemento de un objeto de memoria de este tipo. Las lecturas y escrituras en la memoria global pueden ser almacenadas en caché dependiendo de las capacidades del *device*.

- **Memoria Constante**: En esta, los datos se mantienen constante durante la ejecución de un *kernel*. El *host* es el único que puede asignar e inicializa los objetos en esta memoria.

- **Memoria Compartida**: Esta memoria es local a un *Block*. Sólo puede ser usada para asignar variables que son compartidas por todos los *threads* en ese *block*. La velocidad de acceso a esta memoria desde el hilo es extremadamente más rápida que acceder a los datos que están almacenados en memoria global y constante.

- **Memoria Local**: Es una memoria privada para un *thread*. Las variables definidas en la memoria local de un *thread* no son visibles para otro *thread*. La velocidad de acceso a los datos que están en esta memoria es más rápida que la memoria compartida.

Adicionalmente, CUDA tiene soporte de dos tipos de objetos de memoria, *surface memory* y *texture memory*. Un *surface memory* almacena una colección unidimensional de elementos, mientras un *texture memory* es usado para almacenar texturas, *frame buffer* o imágenes de dos o tres dimensiones. Los elementos de un *surface memory* pueden ser datos escalares (enteros, flotantes), datos de tipo vectorial o una estructura de datos definida por el usuario.

La ventaja de utilizar un *surface memory* es que son almacenados en memoria de forma secuencial y pueden ser accedidos mediante punteros, en cambio, los *texture memory* sólo pueden ser accedido y modificados mediante las funciones que provee CUDA para este tipo de datos.

CUDA también incorpora un tipo llamado *CUDA arrays;* este es un espacio manejado sólo por funciones de CUDA, y al igual que *texture memory,* es un espacio que puede ser definido en una, dos o tres dimensiones y soporta enteros de 8, 16 y 32 *bits*, flotantes de 16 y 32 *bits*, con y sin signo.

Capítulo 4: *Ray Casting* Multi-resolución

En este capítulo se presentan los distintos módulos que conforman la solución al problema. Incluyendo los detalles de implementación. La ***Figura 4.1*** esquematiza los diferentes pasos que sigue el sistema de *Ray Casting* multi-resolución para realizar el despliegue.

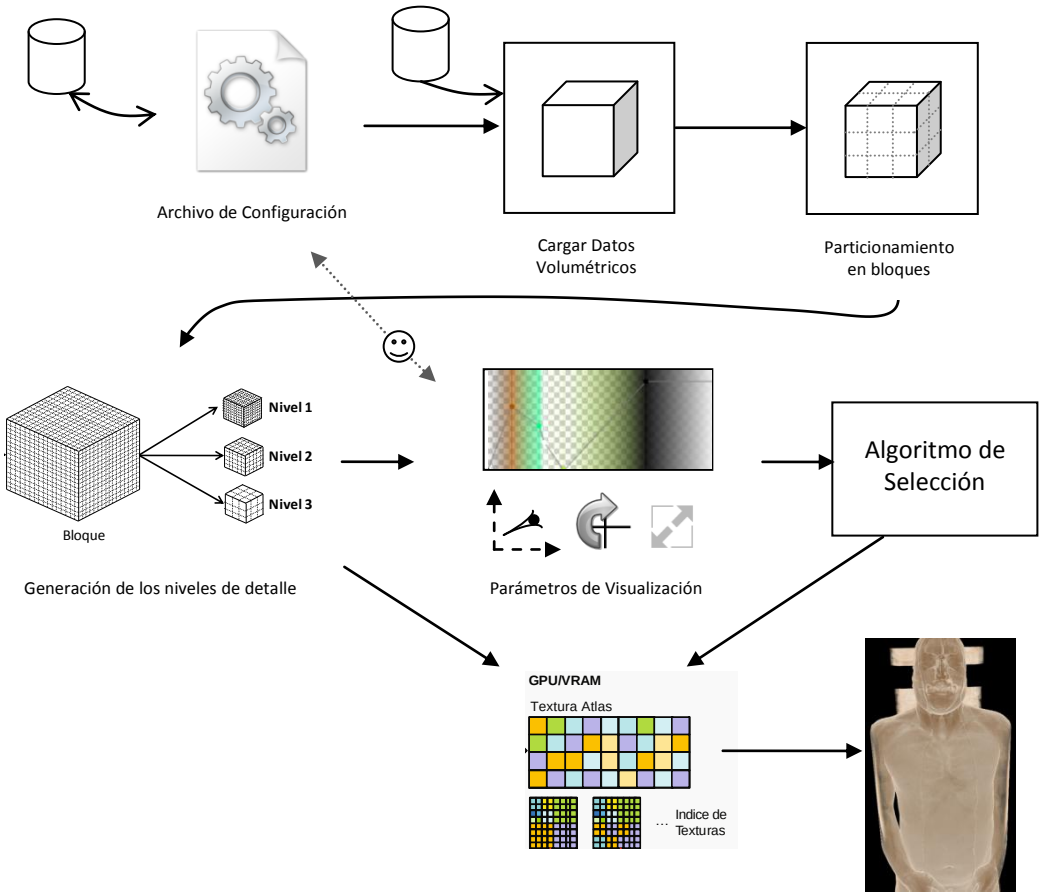


Figura 4.1: *En esta imagen se puede observar los distintos módulos que conforman la aplicación, comenzando por la carga del archivo de configuración hasta la visualización del volumen.*

Los pasos comienzan por la carga de la configuración del volumen. En este archivo se almacenan todos los parámetros de visualización (posición del ojo, rotaciones, escala, función de transferencia, la dimensión de los bloques y la capacidad que va tener la textura atlas) y la ruta del archivo que contiene los datos volumétricos. El usuario mediante una interfaz gráfica puede modificar algunos de los parámetros de visualización y la función de transferencia. Una vez cargado el volumen se pasa por una etapa de pre-procesamiento para poder utilizar una jerarquía por bloques. Aquí es generado cada bloque con sus respectivos niveles de detalle (ver ***Sección 2.3.2***) y adicionalmente aplicado el esquema visto en la ***Figura 2.6***. En este trabajo se utilizan solamente tres niveles de detalles para evitar sobrecargar la memoria del computador debido a los vóxeles de holgura que se requieren para esta técnica.

El algoritmo de selección se basa en la distancia con respecto a un punto de interés y, similar al trabajo realizado por *Ljung et al.* [**34**], se considera la distorsión del volumen multi-

resolución usando la función de transferencia. Debido a esto cada vez que el usuario la modifique se procede a calcular la nueva distorsión del volumen. En este trabajo no se utilizó un histograma de frecuencia como lo hace *Ljung,* sino más bien se aplicó un algoritmo acelerado por OpenMP y por GPU para tener el error local de cada vóxel.

Una vez que se tienen los parámetros necesarios para el proceso de selección se ejecuta un algoritmo que se basa en una cola de prioridad. De esta manera, los elementos más cercanos al ojo y/o con más distorsión pasaran a ser refinados con mayor prioridad. Los elementos posteriormente seleccionados son enviados a una textura de gran tamaño que se le denomina atlas, está a su vez es indexada en otra denominada textura de índices. Luego de cargados todos los boques se procede aplicar la técnica de *Ray Casting* sobre la textura de índices para poder finalmente visualizar el volumen; esta idea fue basada en el trabajo realizado por *Lux et al.* [**43**].

A continuación se presentara con más detalle todos los módulos mencionados anteriormente.

4.1 Jerarquía multi-resolución

En este trabajo especial de grado se utilizó una jerarquía basada en bloques para la representación del volumen multi-resolución. El bloque con mayor nivel de detalle tiene un tamaño de n^3, donde $n = 2^k + 1\ con\ k \geq 4$, los siguientes niveles son de tamaño $(n\ div\ 2^i + 1)^3\ con\ i = 1, 2, 3$. Para la generación de los mismos, a diferencia del ejemplo mostrado en la ***Figura* 2.6**, en este trabajo se utiliza únicamente el píxel central del nivel más fino sin aplicar ningún tipo de filtrado (ver ***Figura 4.2***).

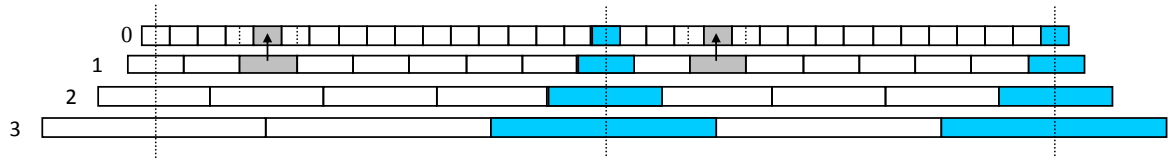


Figura 4.2: *Se puede observar en una representación 1D cómo interviene únicamente el píxel central del nivel d-1 para representar un píxel del nivel d que cubre exactamente 2 píxeles del dominio de interpolación. Adicionalmente en el área azul se muestra como los píxeles fronterizos se comparten entre los vecinos.*

De esta forma se busca evitar la perturbación de los datos, ya que si se observa la ***Figura 4.3a*** el píxel central perdió su intensidad original cuando se mezclo con sus vecinos. Esto da como consecuencia un cambio en los datos provocando una clasificación errónea a la hora de aplicar la función de transferencia. Este tipo de problemas se puede observar con mayor frecuencia entre los contornos de algún objeto. Por eso al no mezclar con los vecinos, como en la ***Figura 4.3b,*** en el proceso de filtrado no se perturbarían tanto los datos originales.

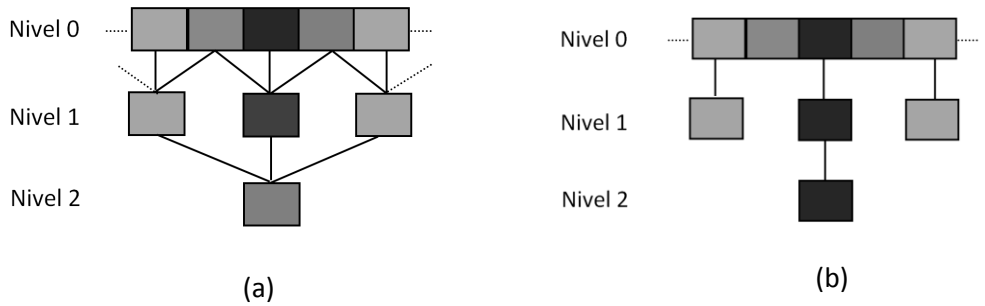


Figura 4.3*: En la figura (a) se puede observar la generación de los niveles de detalle considerando los 3 píxeles que intervienen en el solapamiento. En la (b) los niveles de detalles son generando considerando solamente el píxel central. Observe como el contorno central fue perdido por completo en la figura (a).*

La aplicación es compilada en 64bits para poder visualizar volúmenes que requieran más de 2gb de memoria principal. Este límite puede ser superado debido a que los niveles de detalles son almacenados por completo sin aplicar ningún tipo de paginación; en caso de requerir paginación se le encargará el trabajo al sistema operativo.

Una vez generados y cargados todos los bloques en memoria, se procederá a seleccionar los niveles de detalle aplicando un algoritmo de selección.

4.2 Criterio de selección

El criterio de selección utilizado fue la distancia con respecto a un punto de interés, la distorsión de los niveles de detalle considerando la función de transferencia y la unión de ambos. El usuario es el que decide cual de los tres criterios se utilizan mediante la interfaz gráfica (ver ***Figura 4.5***).

En un inicio los bloques son colocados en una cola de prioridad con el menor nivel de detalle. Los bloques están ordenados por prioridad, que puede basarse en la distorsión y/o la distancia al punto de interés. El proceso de refinamiento consiste en extraer el primer bloque de la cola, subirle un nivel de detalle y reinsertarlo en la cola acorde a su nueva prioridad. Esto se hace hasta que se agote el espacio de la textura atlas o hasta que el volumen quede totalmente refinado. Durante el proceso, aquellos bloques que alcanzan su mayor nivel de refinamiento se mueven a una lista especial de elementos, pues ya no pueden refinarse más.

El criterio basado en el punto de interés es calculado mediante la distancia de dicho punto al centro delo bloque. Considerando el trabajo realizado por *Carmona* [**2**], para darle mayor prioridad a los bloques de menor resolución se le suma la diagonal del bloque a dicha distancia (ver ***Figura 4.4***). Debemos tener en cuenta que la diagonal es calculada en función al número de vóxeles del bloque.

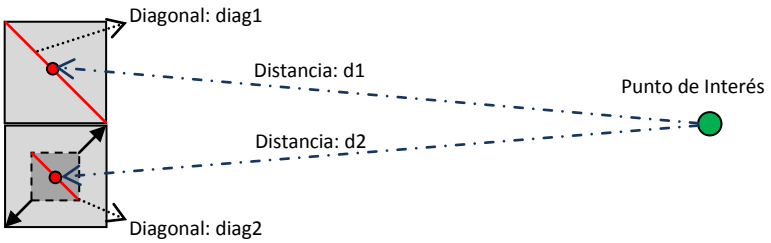


Figura 4.4: *En la imagen se puede observar que la suma diag1+d1 > diag2+d2, por lo tanto el bloque 2 tendrá mayor prioridad de refinamiento.*

La distorsión es determinada en función del error en el espacio CIELUV al utilizar un nivel de detalle más burdo con respecto al nivel más fino de un bloque, esto es una vez que ya es aplicada la función de transferencia. Más adelante se expondrá con detalle cómo se realiza dicho cálculo. También es posible aplicar la unión de ambos criterios, pero hay que considerar que a mayor distorsión mayor prioridad y que a menor distancia al punto de interés también se requiere mayor prioridad, por lo tanto la ecuación que describe ambos criterios es descrita de la siguiente forma:

$$prioridad = \frac{distancia + diag}{distorsion} \qquad \textbf{Ec. 4.1}$$

El refinamiento puede requerir un tiempo considerable de computo si la cola de prioridad contiene una gran cantidad de bloques; por este motivo se implemento un refinamiento *frame* a *frame* similar al trabajo realizado por *Carmona* [**2**]. Este método consiste en sólo refinar a lo sumo N bloques por *frame* hasta que el volumen quede totalmente refinado o se agote la memoria disponible (ver ***Figura 4.6***). A diferencia de *Carmona,* la cantidad de bloques que se refinan en un frame no es determinado por el ancho de banda requerido, sino es el usuario que establece en la interfaz gráfica un porcentaje de bloques que se van a refinar por *frame.*

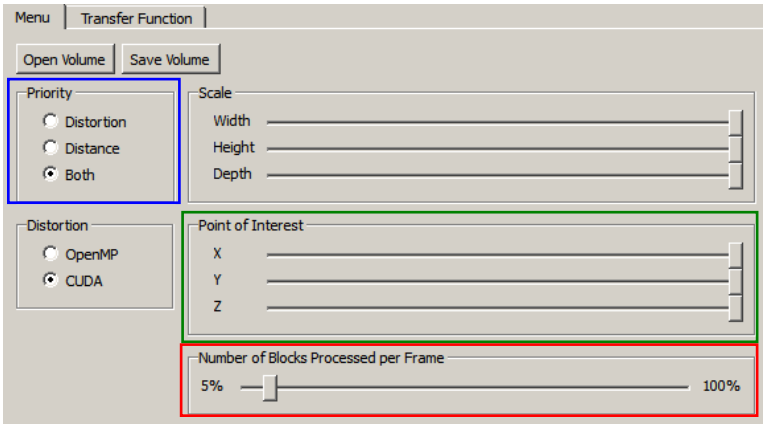


Figura 4.5: *Interfaz gráfica utilizada para el criterio de selección. En la región azul se puede apreciar como el usuario puede seleccionar el criterio que se va utilizar para el refinamiento. En la región verde se modifica la posición del punto de interés y en la región roja se aprecia el porcentaje de bloques que se pueden refinar en cada frame.*

Figura 4.6: *En esta imagen se puede observar como a medida que transcurre el tiempo, el volumen va siendo refinado [**1**].*

Para el criterio de selección también es calculada la opacidad acumulada, el valor del vóxel con menor y mayor intensidad por cada bloque (ver ***Figura 4.7***). Esto es con el fin de determinar si un bloque es totalmente transparente. Cuando un bloque es transparente puede ser omitido en el cálculo de la distorsión y del proceso de despliegue ya que este no hace ningún aporte al proceso de visualización.

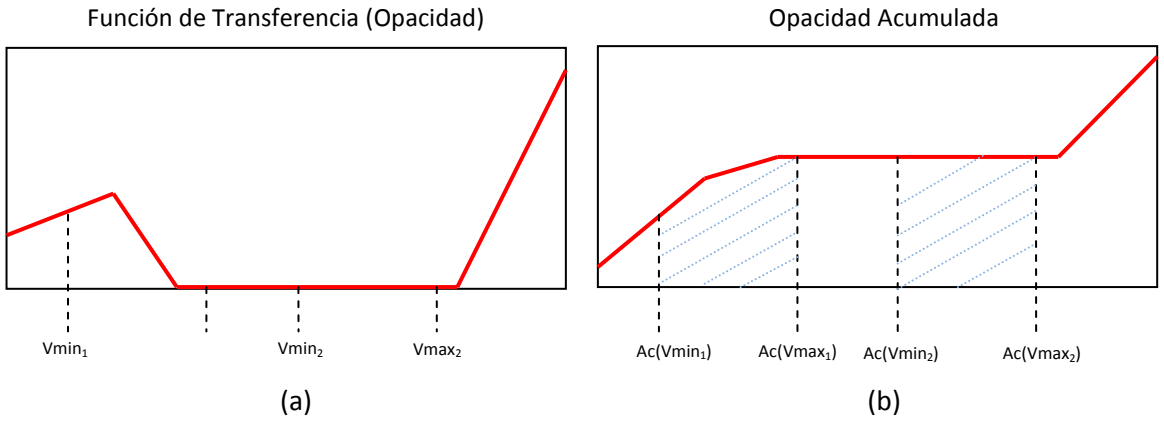


Figura 4.7: *En la figura (a) se puede observar el canal de opacidad de la función de transferencia; en la figura (b) se puede observar la acumulada de dicha función. Vmin y Vmax son los vóxeles de menor y mayor intensidad de todo el bloque. Para el caso del bloque 1 ($Vmin_1$ y $Vmax_1$) se dice que el bloque no es transparente ya que $Ac(Vmin_1) \neq Ac(Vmax_1)$. El bloque 2 ($Vmin_2$ y $Vmax_2$) se dice que es totalmente transparente ya que $Ac(Vmin_1) = Ac(Vmax_1)$. Esto ocurre si ningún vóxel en el intervalo ($Vmin_1$, $Vmax_1$) tiene una opacidad mayor a 0.*

Para realizar el cálculo de la distorsión del volumen multi-resolución se requiere visitar todos los vóxeles de cada nivel de detalle para compararlo con el más fino, por lo tanto esto requiere de un tiempo considerable de cómputo que a continuación mostramos una manera de solventar un poco este problema.

4.3 Cálculo de distorsión

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.* [**34**], inicialmente la función de transferencia es convertida al espacio de color CIELUV para obtener resultados más certeros. La función de transferencia ya convertida es almacenada en una textura 1D RGBA de tipo flotantes. Posteriormente se procede a recorrer cada bloque visible para calcular su distorsión. Se implementaron dos algoritmos, uno utilizando el API de OpenMP y otro utilizando el API de CUDA; el algoritmo puede ser seleccionado previamente por el usuario mediante la interfaz gráfica de la aplicación (ver ***Figura 4.8***). La distorsión es calculada en ambos algoritmos usando la siguiente ecuación:

$$e(V, lod_i) = \frac{\sum_{v \epsilon V}\left(||f\left(s_0(v)\right) - f(s_i(v))||\right)}{n} \qquad \textbf{Ec. 4.2}$$

donde n es la cantidad de vóxeles del bloque, f representa la función de transferencia, s_i es la interpolación tri-lineal del volumen en el nivel de detalle i. Esta ecuación fue obtenida usando como referencia el trabajo realizado por *Gao et al.* [**29**].

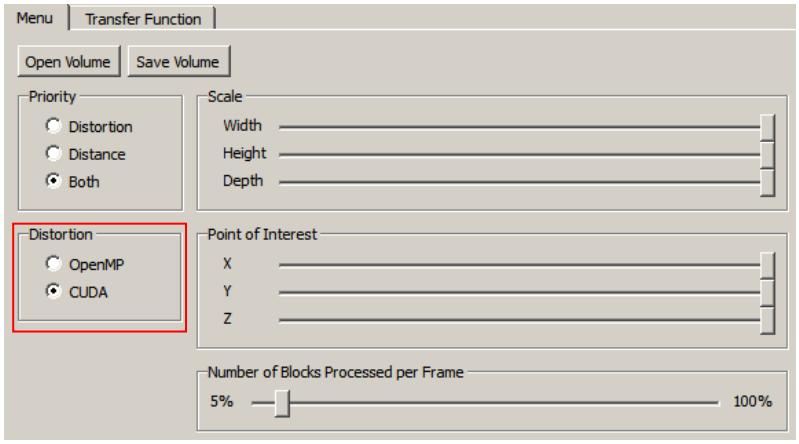


Figura 4.8: *En el área roja el usuario selecciona que tipo de algoritmo desea utilizar para calcular la distorsión del volumen multi-resolución.*

A continuación mostraremos la solución implementada en OpenMP y en CUDA.

4.3.1 Cálculo con OpenMP

El algoritmo utilizado en OpenMP es básicamente el siguiente:

```
#pragma omp parallel for num_threads(N)
Para cada bloque => index hacer
{
   Si(bloques[index].esTransparente) continuar;
   Para cada Nivel de detalle => lod hacer
   {
     bloques[index].distorsion[lod]=e(bloques[index].volumen, lod);
   }
}
```

Algoritmo 4.1: *Cálculo de la distorsión utilizando OpenMP, la función "e" es definida en la* ***Ec. 4.2****.*

donde N es la cantidad de hilos utilizados. Si se utilizara uno sólo este representaría el algoritmo secuencial. 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* [**45**]. 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[9], por ende esta implementación no requirió de muchas adaptaciones al código secuencial.

4.3.2 Cálculo con CUDA

CUDA presenta distintas limitaciones que dependen de cada *hardware* gráfico. Las limitaciones vienen definidas en NVIDIA CUDA *Compute Capability* [**49**]. 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 (ver ***Sección 3.2*** CUDA). Para calcular la cantidad total de hilos utilizados en una aplicación se usa la siguiente fórmula:

$$threadsTotales = threadsPorBlocks * cantidadDeBloks \qquad \textbf{Ec. 4.3}$$

Para el cálculo de la distorsión se requirió el uso adicional de cinco texturas: el bloque, los tres niveles de detalle locales al bloque y la función de transferencia. Para esto se utilizaron objetos de memoria de tipo *texture memory* para el almacenamiento y el manejo de las texturas, de igual forma que en OpenGL, pueden ser interpoladas de forma automática usando el filtro `cudaFilterModeLinear` [**49**].

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 un hilo calcula 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 total. 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

[9] Se denomina sección crítica, en programación paralela, a la porción de código de un programa en la cual se accede a un recurso compartido que no debe ser accedido por más de un hilo en ejecución.

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.

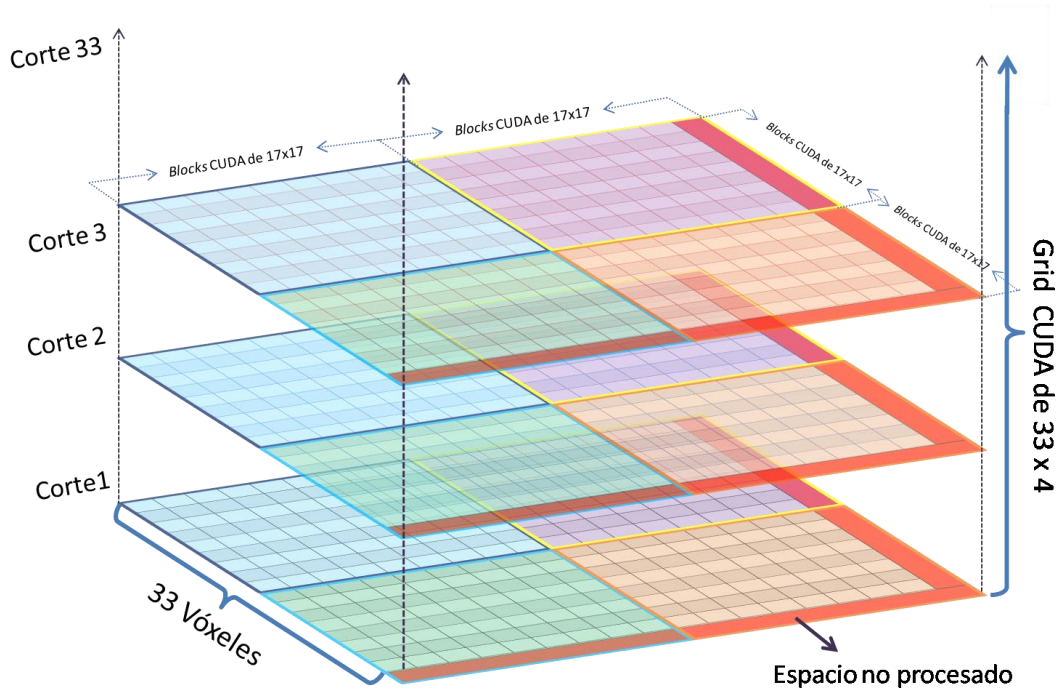


Figura 4.9: *En este ejemplo se puede observar cómo se subdividió un bloque de 33^3 en un grid de 33x4, haciendo que cada block de CUDA tuviera una dimensión de 17x17 para distribuir de mejor forma el trabajo y no sobre pasar el límite de 512 hilos.*

En la ***Figura 4.9*** se puede apreciar un ejemplo de cómo se forma el *grid*. Para este caso se puede observar los 33 cortes y los 4 cuadrantes en los que se subdivide cada uno. Note que hay un área roja que no es procesada en los *blocks* que pertenecen a la frontera. Esto se debe a que el esquema utilizado de $n^2 + 1$ no es un numero divisible, por lo tanto para este ejemplo se utilizaron 38148 hilos en total para procesar 35937 vóxeles, haciendo que 2211 queden inutilizados provocando un desaprovechamiento de recursos.

Una vez que cada hilo realiza su cálculo, estos lo almacenan en una memoria compartida para que luego puedan sumarse (ver ***Figura* 4.10**).

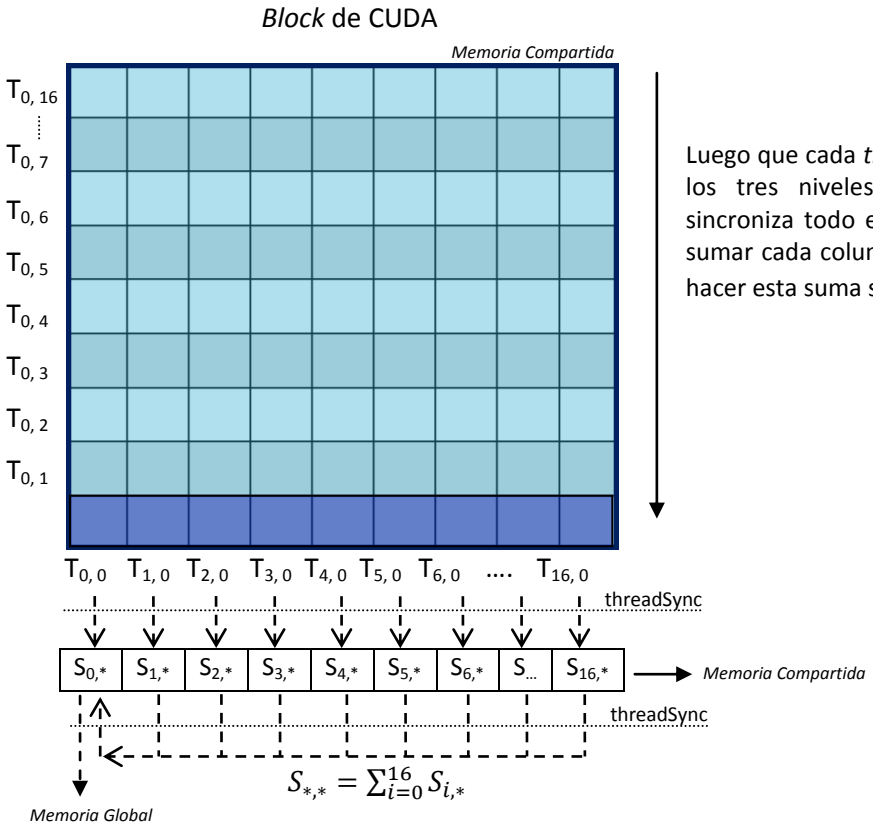


Figura 4.10: *Cada hilo almacena su resultado en una memoria compartida de tamaño NxN, donde N es la dimensión del Block. Posteriormente los hilos $T_{i,0}$ suman su respectiva columna y lo almacenan en otra memoria compartida, por último se vuelve a sincronizar todo el block y el hilo $T_{0,0}$ es el encargado de sumar los resultados de todas las columnas y lo almacena en un arreglo que se encuentra en la memoria global del GPU.*

El metodo utilizado consiste básicamente en sumar primero las columnas del *block* y luego sumar la fila resultante. El primer hilo de cada *block* es el encargado de calcular la suma total; a su vez este lo 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 4.9*** el arreglo global debe tener una dimensión de 33x4 (ver ***Algoritmo* 4.2**). 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*.

```
//Host CUDA

Funcion CalcularDistorsion(Textura bloques[4],Textura FTLuv)
{
    SubirTexturasGPU(bloques, FTLuv);
    Dim3 grid=CalcularGrid(bloques[0], SIZE_BLOCK_CUDA);
    Dim3 block(SIZE_BLOCK_CUDA, SIZE_BLOCK_CUDA, 1);
    float d_distorsion[3][grid.x][grid.y];
    float distorsionFinal[3];
    distortion<<<grid,block>>>(d_distorsion);
```

```
    cudaThreadSynchronize();
    Para cada Nivel de detalle => lod Hacer
    {
       Para cada Block => b Hacer
       {
         distorsionFinal[lod] += d_distorsion[lod][b.x][b.y];
       }
    }
    Retornar distorsionFinal;
}

//Kernel CUDA
__global__ void distortion(float errTotal[3][gridX][gridY])
{

      si(thread esta dentro del bloque){

            //Se obtiene el valor en LUV del voxel correspondiente de
              cada nivel de detalle.
            luv0  = tex1D(tex_ft_luv, getVoxel(lvl0));
            luv1  = tex1D(tex_ft_luv, getVoxel(lvl1));
            luv2  = tex1D(tex_ft_luv, getVoxel(lvl2));
            luv3  = tex1D(tex_ft_luv, getVoxel(lvl3));
            //Se calcula la diferencia entre el Nivel 0 y los niveles 1,2
              y 3
            eLocal[0][globalPosInBlock] = diffColor(luv0, luv1);
            eLocal[1][globalPosInBlock] = diffColor(luv0, luv2);
            eLocal[2][globalPosInBlock] = diffColor(luv0, luv3);

      }

      __syncthreads();

      si(thread esta dentro del bloque){

            si(threadIdx.y==0){
                  Para cada Nivel de detalle => lod Hacer
                  {
                      Para cada Fila => f Hacer
                     {
                        eSubTotal[lod][threadIdx.x]+= eLocal[lod][f];
                     }
                  }
            }

      }

       __syncthreads();

      si(thread esta dentro del bloque){

            si(threadIdx.x==0 && threadIdx.y==0){
                  Para cada Nivel de detalle => lod Hacer
                  {
                        Para cada Columna => c Hacer
                        {
                              errTotal[lod][blockIdx.x][blockIdx.y]+=
```

```
                                                    eSubTotal[lod][c];
                                }
                            }
                        }

                }

}
```

Algoritmo 4.2: *Kernel utilizado para el cálculo de la distorsión.*

La memoria compartida requerida para este esquema se calcula de la siguiente forma:

$$MemoriaCompartida = 3 * (SizeBlockCUDA^2 + SizeBlockCUDA) * sizeof(float) \quad \textbf{Ec. 4.4}$$

donde $SizeBlockCUDA^2$ es el espacio utilizado para almacenar el resultado de cada hilo, adicionalmente se le suma el espacio requerido para el arreglo que contiene la suma de cada columna. Esto es multiplicado por la cantidad de niveles de detalles y el tamaño en *bytes* que ocupa un tipo de dato *float*. En el ejemplo de la ***Figura 4.9*** la memoria compartida utilizada es de 7344 bytes, que equivalen aproximadamente a 7,17KB.

Hay que tener en cuenta que si el bloque del volumen es muy pequeño se desaprovecharía los recursos del GPU, pues la cantidad total de bloques generados para un volumen aumentaría. 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.

Una vez realizado el cálculo de la distorsión y aplicado el criterio de selección se procede a la carga de los bloques a la textura atlas. Esto se explica a continuación.

4.2 Textura atlas

Utilizando el esquema propuesto por *Lux et al.* [**43**] pero adaptándolo a una jerarquía por bloques 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.

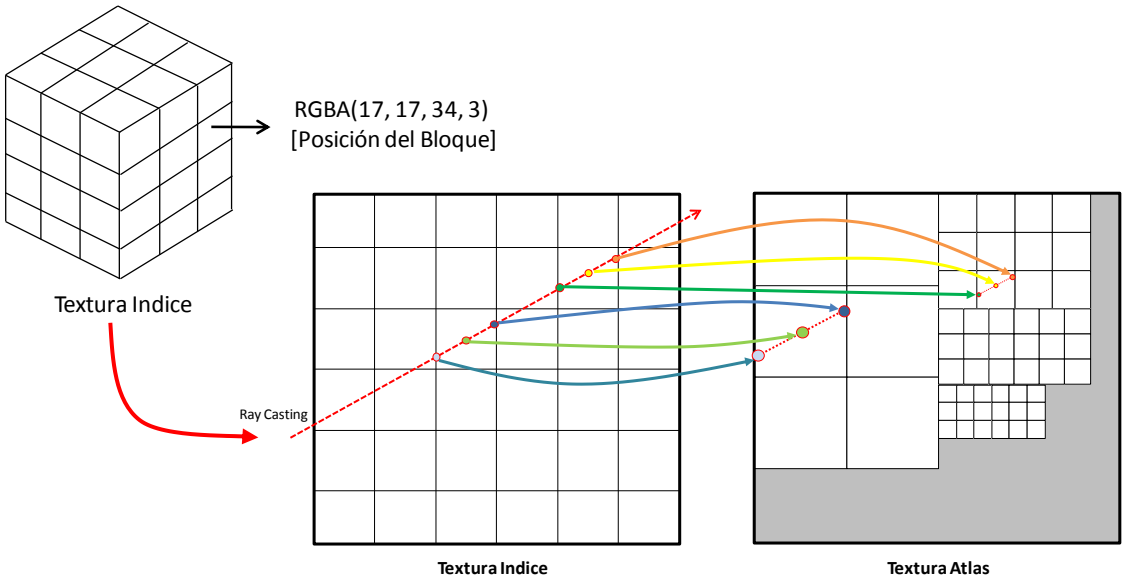


Figura 4.11: *Inicialmente* se *muestra cómo cada vóxel de la textura de índices tiene una tupla RGBA con el siguiente formato: (Px, Py, Pz, Lod). Utilizando una representación 2D se puede observar la interacción entre la textura de índice y el atlas en cada paso del rayo.*

La dimensión de la textura de índices es:

$$S(Tindice) = (\left\lceil \frac{Sx-1}{n-1} \right\rceil, \left\lceil \frac{Sy-1}{n-1} \right\rceil, \left\lceil \frac{Sz-1}{n-1} \right\rceil) \qquad \textit{Ec. 4.5}$$

donde (Sx, Sy, Sz) es el tamaño del volumen y n es el tamaño que va tener el bloque de mayor definición. Esta textura utiliza un formato RGBA de tipo flotantes para mapear el atlas: los canales RGB almacenan la ubicación del bloque en la textura y el canal *alpha* indica el nivel de detalle del bloque. Adicionalmente hay que utilizar la instrucción `gl_nearest [12]` para que no se interpolen los índices con los vecinos.

Una vez que todos los bloques son almacenados e indexados se le aplica la textura de índices a un cubo unitario y se renderizan las caras frontales para proceder a ejecutar el algoritmo de *Ray Casting* (ver ***Sección 1.4.1***) con clasificación pre-integrada (ver ***Algoritmo* 4.3** y ***Figura 4.11***). La importancia de este método de visualización radica en que sólo se necesita realizar una pasada del *Ray Casting* para visualizar todo el volumen multi-resolución; adicionalmente con esta técnica se pudo aplicar la terminación temprana del rayo y salto de bloques transparentes.

```
Para cada Rayo Hacer

  //Se obtiene la dirección del rayo normalizada.
  Vector3 direction=getDirection(Rayo);

  //Se obtiene la posición de entrada del rayo al volumen.
  Vector3 rayPos=getPos(Rayo);

  Mientras rayPos este dentro del Volumen Hacer
  {
    // Se busca la posición del bloque en la textura índice.
    Vector4 index = obtenerVoxelDelIndice(indexVolume, rayPos);
    Si no esTransparente(index)
    {
      //Se obtiene el valor del vóxel ubicado en el atlas en la posición
      //que haya indicado la textura de índice.
      voxel = obtenerVoxelDelAtlas(atlasVolume, rayPos, index);

      // Obtenemos el valor del color desde la tabla de pre-integración
      Vector4 color = preIntegracion(voxel);

      //Se procede a realizar la composición del color obtenido
      componerColor(color);
    }
    //Se actualiza la posición  del rayo.
    rayPos += rayStep*direction;
  }
```

Algoritmo 4.3: *Algoritmo de Ray Casting adaptada a una textura altas.*

La dimensión de la textura atlas es dinámica, el usuario puede establecer un tamaño en el archivo donde esta almacenada la información del volumen que se va desplegar. Esta debe adecuarse de manera correcta para que no exceda la memoria disponible del *hardware* gráfico.

Para administrar el espacio de la textura atlas se generó un algoritmo básico que consiste en dividirla en distintas áreas dependiendo de los requerimientos del criterio de selección. Cuando se refina un bloque se realiza una reservación de memoria con la dimensión del nuevo nivel de detalle y se liberar el espacio utilizado por el nivel burdo. Esto se describe mejor en el siguiente algoritmo:

```
Funcion reserveMemory(int size)
{
   //Cola de prioridad para las areas displonibles en el atlas
   Para cada freeArea => a Hacer
   {
      //Verificamos si el tamaño del Area es suficiente para el nuevo
      //bloque
      si(a.size>=size)
      {
        //Por cada subdivision se generan a lo sumo 3 areas nuevas
         Area nAreas[3] = generarNuevasAreas(a.size, a.pos, size);
```

```
         //Lista de las areas utilizadas en el atlas
          posicion=nonFreeArea.push_back(Area(a.pos, size));

          freeArea.eliminar(A);    // Eliminamos el Area Utilizada

          Para i=0 Hata 3 Hacer
             si(nAreas[i].size>=tamañoMinimoDelBloque){
                // Insertamos la nueva area disponible
                freeArea.insertar(nAreas[i]);
             }else{

//Contador en Bytes de la memoria libre inutlizable porque el tamaño es
//demasiado bajo para almacenar un bloque asi tenga el menor nivel de
//detalle
                countTrash += nAreas[i].blockSize;
             }
          }

          Retornar posicion; //Retornamos la posicion que se reservo
       }
    }

    Retornar -1; //Retornamos -1 si no hay espacio disponible
}
```

Algoritmo 4.4: *Función utilizada para administrar el espacio disponible en la memoria atlas.*

La desventaja de este algoritmo es que a medida que cambian los niveles de detalle del volumen se va generando mayor fragmentación en la textura atlas. Esto da como resultado que en ocasiones se desaproveche mucho el espacio cuando en la subdivisión resultan áreas muy pequeñas. A continuación se muestra una solución desarrollada para tratar de solventar un poco este problema.

4.2.1 Fragmentación del Atlas

En la ***Figura 4.12*** se puede notar un claro ejemplo de cómo el atlas se puede fragmentar con el algoritmo mostrado previamente; adicionalmente se observa que los bloques liberados también pasan a ser inutilizados si no hay mas bloques por refinar que puedan ser almacenados en este nuevo espacio disponible.

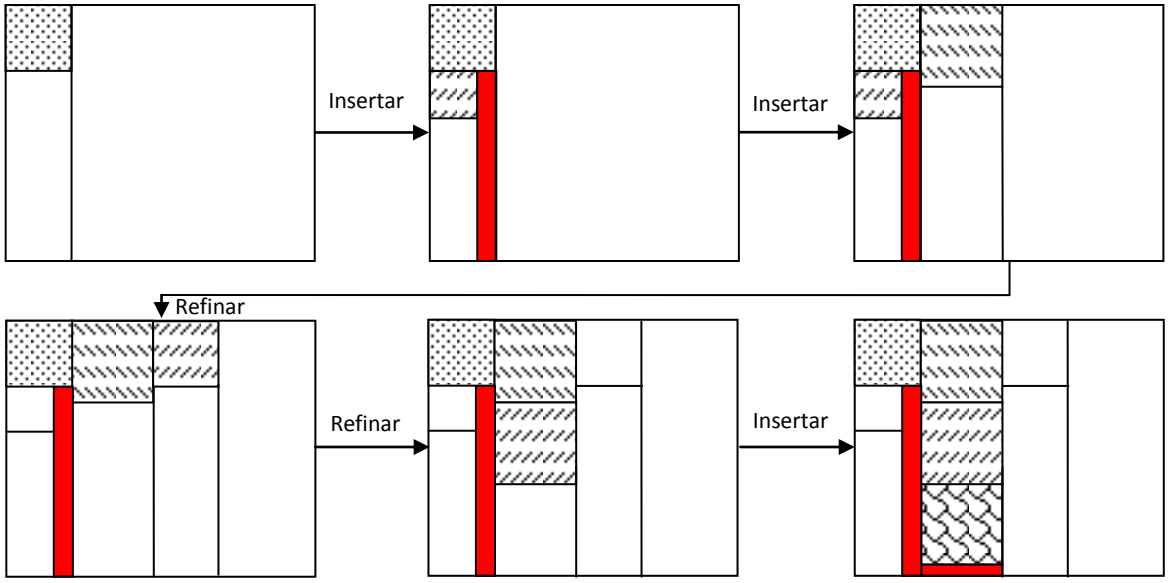


Figura 4.12: *Proceso de inserción y refinamiento de los bloques en la textura atlas. En la región roja se puede notar las áreas que son demasiado pequeñas como para almacenar un bloque, en cada subdivisión se generan 2 nuevas áreas vacías, en el caso 3D se generarían 3 nuevas áreas.*

El método desarrollado se basa en la siguiente premisa, si el volumen multi-resolución no pudo ser refinado por completo y el atlas tiene espacio disponible pero fragmentado, se realiza una desfragmentación. Para esto se reagrupan los bloques, esto significa que los bloques se ordenan por tamaño partiendo desde los más grandes hasta los más pequeños. Posteriormente se continúa el proceso de refinamiento. Esto se repite hasta que la cantidad de bloques que no pudieron ser refinados sea igual antes y después de una desfragmentación (ver ***Figura 4.13***). El problema de esta técnica radica en que se cada vez que se realiza una desfragmentación se vuelve a enviar todos los bloques a la textura atlas pero en sus nuevas posiciones.

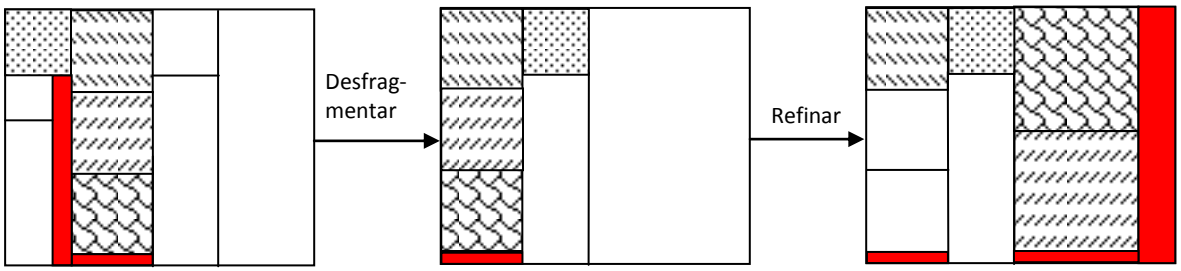


Figura 4.13: *En la primera imagen podemos observar que no hay espacio suficiente para poder refinar más los bloques que tienen mayor tamaño, aplicando el algoritmo de desfragmentación note que se acoplan los bloques permitiendo utilizar mejor el espacio restante del atlas.*

Habiendo mostrado el funcionamiento de todos los módulos desarrollados a continuación se mostrara un diagrama de clases de cómo se estructura este prototipo.

4.4 Diagrama de clases

El prototipo desarrollado se basó en el trabajo previo realizado *Carmona* [**2**]; a este se le eliminaron y cambiaron diversos módulos. Unos de los cambios más resaltantes fue el reemplazar la jerarquía *octree* por una jerarquía basada en bloques, añadirle el despliegue utilizando *Ray Casting* de una pasada y cambiar el criterio de selección.

La aplicación básicamente consta de 8 clases descritas en la ***Figura 4.14***. Ciertas convenciones fueron utilizadas en la nominación de las clases, estructuras, atributos, métodos y constantes. Estas fueron basadas en el *framework* de MFC (Microsoft Fundation Class) [**50**], donde las clases empiezan con la letra C, los atributos contienen el prefijo "m_", los métodos empiezan con letra mayúscula, las estructuras y constantes se nominan en letras mayúsculas. Ejemplos: CFlatBlock, CTextureObject, m_textureAtlas, GenBlocks(), PI.

La clase principal *CVolume* presenta los distintos métodos y atributos para el manejo de toda la aplicación, incluyendo la función de transferencia y la clase *CFlatBlock* que se encarga de manejar todo el funcionamiento de la jerarquía por bloques. *CMemoryArea* es la encargada de administrar la textura atlas; aquí se administra los espacios libres y se determina la posición en el que un bloque puede ser insertado. *CListToDisplay* es utilizado para el criterio de selección, esta es una cola de prioridad que se basa en el atributo *m_priority* para determinar cuál va ser el próximo bloque a refinar. *CBlock* es la clase que contiene la textura de un bloque y *CBlock_List* mantiene la traza de los niveles de detalle que están siendo desplegados; adicionalmente contiene el vóxel de mayor y menor intensidad del bloque y su posición global en la textura de índice. *CTextureObject* es la clase encargada de controlar todas las texturas utilizadas; adicionalmente hace las comunicaciones necesarias con el GPU para enviarlas, editarlas o eliminarlas.

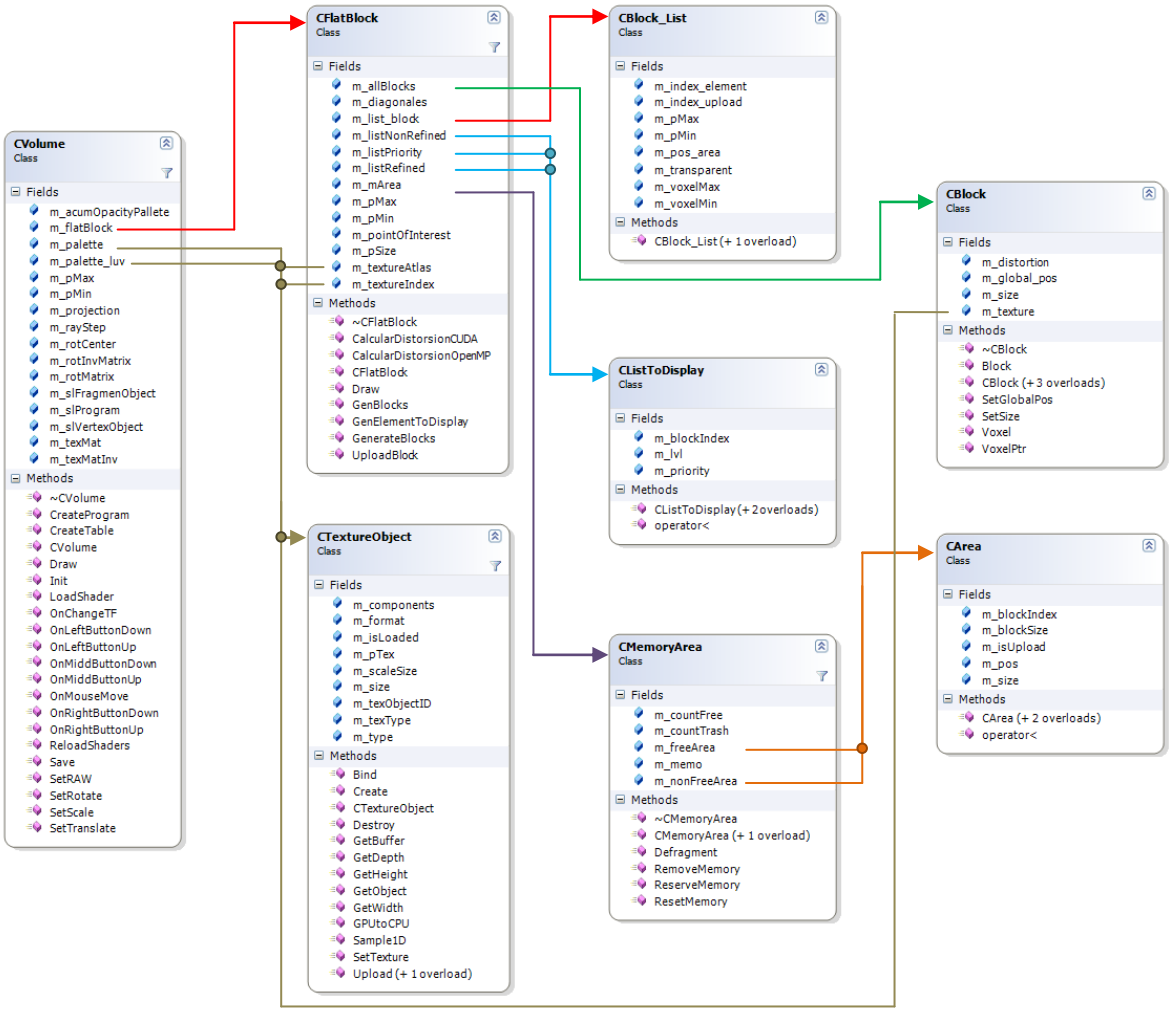


Figura 4.14: *Diagrama de clases del prototipo desarrollado*

Capítulo 5: Pruebas y Resultados

En el capítulo anterior se detalla la implementación del algoritmo *Ray Casting* multi-resolución en una pasada. Ahora procederemos a mostrar las pruebas y resultados del rendimiento y calidad de las imágenes generadas. En este capítulo también se especifica la plataforma, lenguaje, librerías y volúmenes utilizados para este fin.

5.1 Ambiente de pruebas

A continuación se exponen las especificaciones del ambiente de pruebas. Adicionalmente se especifican los volúmenes utilizados para cada una de las pruebas realizadas.

5.1.1 Especificaciones del hardware

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 (ver ***Tabla 5.1***).

Equipo	Procesador	Tarjeta Gráfica	Memoria RAM
Equipo 1	Intel Core 2 Quad Q6600 2.40 GHz	GeForce GTX+ 9800	DDR2 Dual Channel, 2 slots con 2 GB c/u, total 4 GB
Equipo 2	Intel Core i3 3.07GHz	GeForce GTX 470	DDR3 Single Channel, 1 slot con 4 GB

Tabla 5.1: *Descripción de los equipos utilizados para las pruebas*

Es importante señalar las especificaciones de las tarjetas gráficas utilizadas para saber la capacidad de la memoria de textura y conocer el NVIDIA CUDA *Compute Capability* [**49**] (ver ***Tabla 5.2***). También se mostraran las especificaciones de los procesadores utilizados (ver ***Tabla 5.3***), esto para conocer la cantidad de núcleos reales de procesamiento y la capacidad de la memoria cache.

Tarjeta Gráfica	Memoria	Núcleos de CUDA	Capability
GeForce GTX+ 9800	512 MB	128	1.1
GeForce GTX 470	1280 MB	448	2.0

Tabla 5.2: *Descripción de las tarjetas gráficas utilizadas para las pruebas*

Procesadores	Núcleos Reales	Nº de Thread/core	Cache
Intel Q6600	4	1	8 MB – 2 MB/Núcleo
Intel Core i3	2	2	4 MB – 2 MB/Núcleo

Tabla 5.3: *Descripción de los procesadores utilizadas para las pruebas*

5.1.2 Especificaciones del software

Los requerimientos de software son los siguientes:

- Sistema Operativo Windows de 64bits para abrir volúmenes de gran tamaño.
- Como API gráfica para realizar el despliegue se utilizó OpenGL® con soporte para la programación de los procesadores de vértices y fragmentos mediante el lenguaje GLSL. Para el manejo de eventos del teclado y mouse se utilizó GLUT.
- 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
- Para la programación paralela en CPU se utilizó OpenMP 1.0

El sistema fue compilado en el entorno de desarrollo de Microsoft Visual Studio Professional 2008.

5.1.3 Datasets

Para las pruebas se utilizaron tres volúmenes: dos volúmenes tomados del proyecto Humano Visible [**1**] y 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. A continuación mostraremos las especificaciones de cada uno de estos volúmenes.

A. Mujer Visible

Del proyecto Humano Visible se tienen los cortes fotografiados en RGB de la mujer visible convertidos a escala de grises. Por limitaciones de este trabajo, se utilizaron únicamente los primeros 1.47GB de los 12GB originales. Dicha zona corresponde a los miembros inferiores (Ver ***Figura 5.1***). La dimensión de este sub volumen es de 890x890x2000 donde cada vóxel tiene 8bits. De ahora en adelante nos referiremos a este como **"**Volumen A".

Figura 5.1: *Se puede observar el "Volumen A" que va ser utilizado en las pruebas. (a) Muestra la función de transferencia; (b) Ángulo posterior;(c) un ángulo anterior de los miembros inferiores.*

B. Tomografía computarizada de la Mujer Visible

También tomado del proyecto Humano Visible, este *dataset* tiene una dimensión de 512x512x1734 donde cada vóxel tiene 16bits y ocupa un total de 867 MB de memoria (ver ***Figura 5.2***). De ahora en adelante nos referiremos a este como **"**Volumen B".

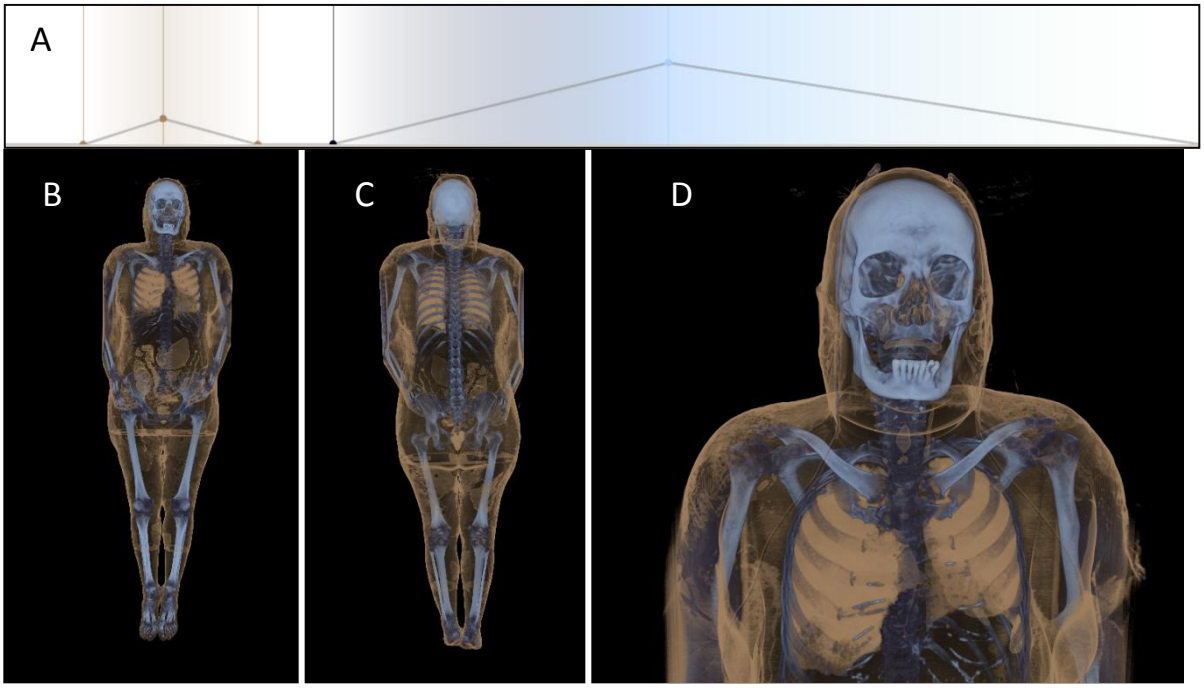


Figura 5.2: *En esta imagen podemos observar una vista previa del "Volumen B" que va ser utilizado en las pruebas. (a) Podemos observar la función de transferencia utilizada; (b) y (c) Podemos ver una vista global del volumen; (d) Acercamiento de la parte superior.*

*C. Escarabajo (Stag beetle [**51**])*

Este *dataset* contiene un escarabajo que fue tomado mediante una tomografía computarizada. Tiene una dimensión de 832x832x494 donde cada vóxel tiene 16bits y ocupa un total de 652MB de memoria (ver ***Figura 5.3***). De ahora en adelante nos referiremos a este como **"**Volumen C".

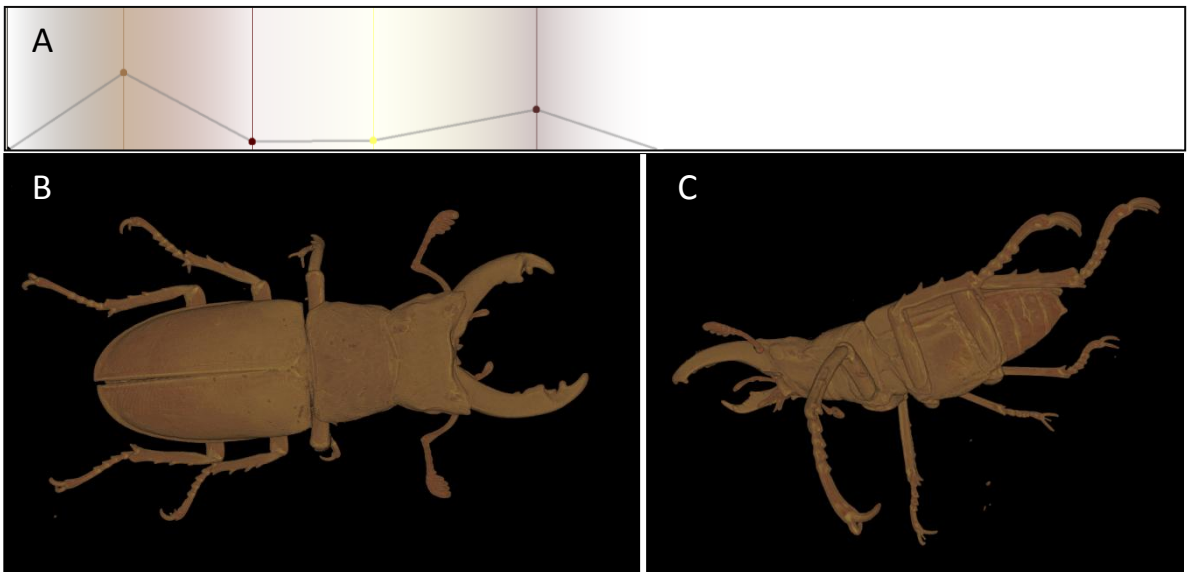


Figura 5.3: *En esta figura se tiene una vista previa del "Volumen C". En (a) podemos observar la función de transferencia utilizada; (b) una vista superior; (c) vista frontal.*

5.2 Resultados Cuantitativos

A continuación se hicieron una serie de mediciones de tiempo de procesamiento y consumo de memoria para los tres volúmenes vistos anteriormente. Cada prueba se ejecutó cinco veces para obtener resultados más precisos. El *viewport* y el ángulo de visualización de los volúmenes utilizados fue el mismo reflejado en la ***Figura 5.1c***, ***Figura 5.2d*** y ***Figura 5.3c***. En la fase de pruebas se utilizaron tres tamaños para la jerarquía basada en bloques y tres tamaños de atlas, con esto se pretenden determinar cual ofrece un mejor desempeño en el cómputo, requerimientos de memoria y da mejores resultados visuales.

5.2.1 Etapa de Pre-procesamiento

En esta etapa veremos una tabla donde se refleja la cantidad total de bloques que se generan, la memoria requerida para almacenar todo el volumen multi-resolución y el tiempo utilizado para la generación de los niveles de detalle (ver ***Tabla 5.4***).

		Volumen A			Volumen B			Volumen C		
		17^3	33^3	65^3	17^3	33^3	65^3	17^3	33^3	65^3
Bloques Generados		392.000	49.392	6.272	111.616	14.080	1.792	83.824	10.816	1.352
Memoria (MB) (incluye los niveles de detalle)		2.166	1.965	1.892	1.234	1.120	1.081	927	861	816
Tiempo requerido para generar los niveles de detalle (Segundos)	Equipo 1	88.3s	29.4s	26.7s	12.8s	8.5s	7.8s	9.3s	6.5s	5.9s
	Equipo 2	76,9s	24.6s	19.4s	8.5s	5.7s	5.3s	6.2s	4.4s	4s

Tabla 5.4: *Tabla de requerimientos de memoria y tiempo para la generación de la jerarquía por bloques.*

La memoria requerida para almacenar todo el volumen multi-resolución es dependiente al tamaño de los bloques, como se explico en la ***Sección 4.1*** Jerarquía multi-resolución; esto se debe al vóxel adicional utilizado para alinear las texturas y para el proceso de interpolación entre bloques. Adicionalmente observe que en esta tabla el Volumen A requiere un tiempo considerable en la etapa de pre-procesamiento comparado con los otros dos volúmenes debido a la gran cantidad de bloques que se generan, en el caso de los bloques de 17^3 se puede notar mas esta diferencia porque la cantidad de bloques es considerable.

Una vez cargado los bloques y generados los niveles de detalle, se procede a calcular la distorsión utilizando la función de transferencia y aplicar el algoritmo de selección. A continuación se mostrara una serie de pruebas realizadas en el cálculo de la distorsión.

5.2.2 Cálculo de distorsión

En el proceso del cálculo de la distorsión se utilizaron las funciones de transferencia mostradas en la ***Figura 5.1a***, ***Figura 5.2a*** y en la ***Figura 5.3a***. Aquí se pretende determinar la diferencia entre utilizar el GPU y utilizar el CPU en el tiempo de respuesta. Para esto se tomaron una serie de pruebas de rendimiento utilizando 1, 2, 3 y 4 procesadores y CUDA. En el caso de CUDA se uso 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 un bloque de mayor tamaño, este se subdivide como se muestra en la ***Figura 4.9****.*

		Volumen A			Volumen B			Volumen C		
		17^3	33^3	65^3	17^3	33^3	65^3	17^3	33^3	65^3
Bloques Visibles		381.020	47.243	6.272	83.967	11.058	1.463	83.824	10.816	1.352
Equipo 1	1 CPU	379,1s	341,7s	329,4s	88,2s	86,4s	89,4s	80,1s	75,4s	72,8s
	2 CPUs	202,1s	183,3s	177,1s	50,1s	47,5s	47,5s	40,3s	37,7s	36,5s
	3 CPUs	137,2s	123,4s	119,5s	32,2s	30,9s	31,5s	27,2s	25,5s	24,5s
	4 CPUs	106,6s	94s	90,4s	24,8s	24s	24,5s	22,1s	19,3s	18,5s
	CUDA	240,4s	55,5s	28,9s	49,7s	12,4s	6,4s	50,2s	12s	6s
Equipo 2	1 CPU	290,4s	259,2s	248,1s	65s	63,3s	65,5s	58,2s	54,8s	52,1s
	2 CPUs	174,7s	150,5s	140,2s	36,1s	35,2s	34,9s	32,4s	29,3s	28,1s
	3 CPUs	138,3s	120,5s	116s	33,1s	30,3s	31,2s	27,6s	25,7s	24,5s
	4 CPUs	122,2s	108,3s	103,7s	29,2s	28s	28,2s	25,6s	24,1s	23,1s
	CUDA	163,1s	24,3s	10,6s	26,6s	5,8s	2,6s	26,8s	5,7s	2,4s

Tabla 5.5: *Tabla comparativa entre el CPU y el GPU para el cálculo de la distorsión usando distintas dimensiones del bloque.*

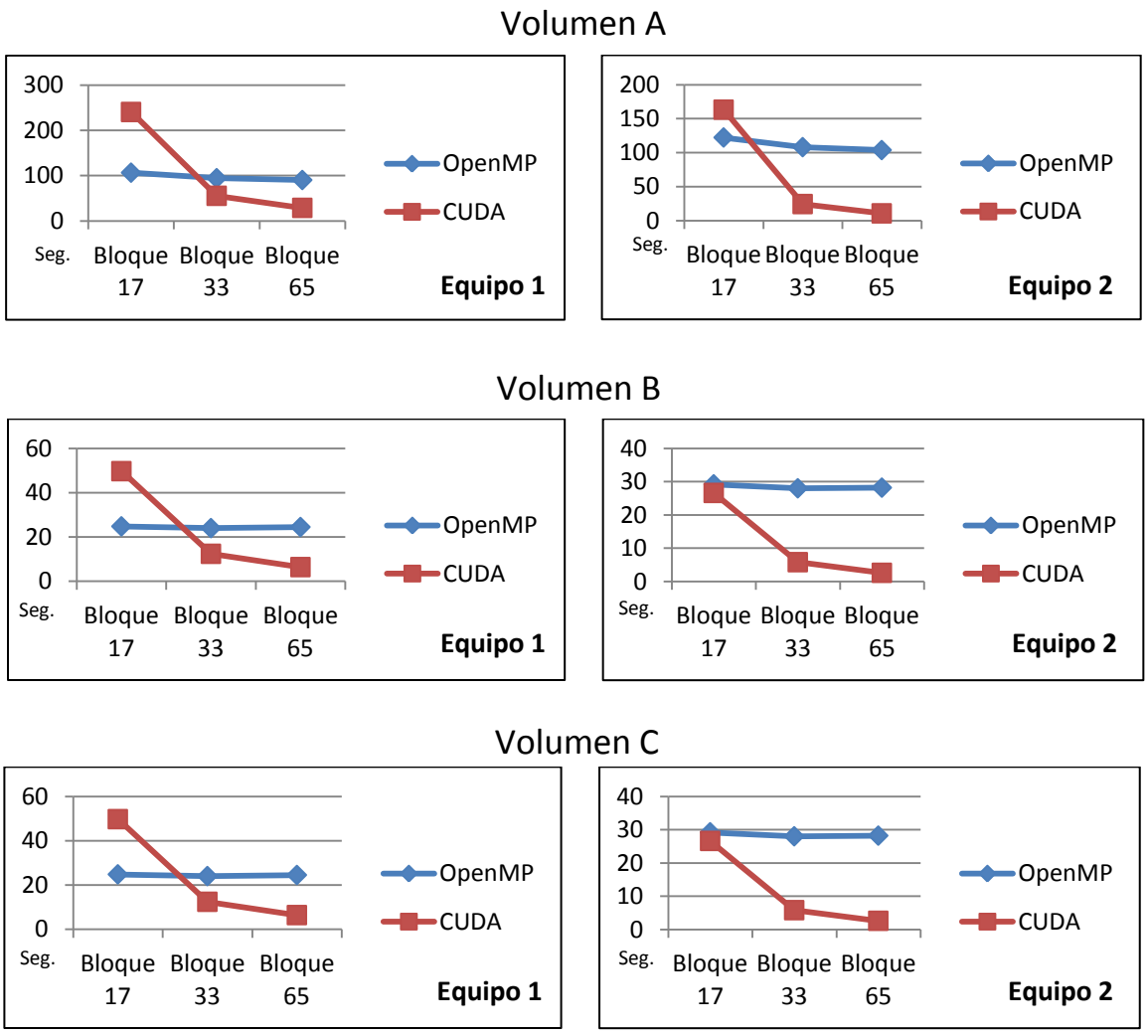


Figura 5.4: *Resumen gráfico de la* ***Tabla 5.5****; En los 3 volúmenes y en ambos equipos de prueba el cálculo realizado en CUDA es mucho más rápido cuando se utiliza un bloques de 33^3 y 65^3. OpenMP con 4 procesadores da mejores resultados en bloques de 17^3.*

En la ***Tabla 5.5*** y ***Figura 5.4*** se pudo observar que en el equipo 2 utilizar CUDA, en bloques de 65^3, puede llegar a ser aproximadamente 10 veces más rápido que usar 4 procesadores. En el equipo 1 la diferencia puede llegar a ser hasta 4 veces más rápido. Adicionalmente en bloques de 33^3 se pudo observar también una disminución del tiempo considerablemente. Caso contrario ocurre con los bloques de tamaño 17^3, donde el uso de 4 procesadores tiende a ser el que ofrece resultados más rápido. Esto se debe a que entre más pequeño es la dimensión del bloque, mayor es la cantidad de comunicaciones con el GPU. Recordemos que el GPU está diseñado para el procesamiento. OpenMP se mantiene constante en las pruebas debido a que la información siempre estará ubicada en la memoria principal del computador, por lo tanto el tiempo que se toma el CPU en recoger los datos es igual. En cambio en el GPU hay que hacer una actualización de las texturas y la invocación al *kernel* de CUDA generando mayor sobrecarga, por esto es que entre menos cantidad de instanciaciones de CUDA mejor es el tiempo de respuesta.

Otro detalle importante que se debe señalar es que, en el equipo 1, la tendencia del tiempo es disminuir hasta casi 1/4 con 4 procesadores. En cambio en el equipo 2 la tendencia es aproximadamente 2/5 del tiempo; esto se debe a que los Intel Core i3 solo tienen 2 procesadores reales y 2 simulados; por lo tanto la reducción del tiempo de respuesta está muy lejana a ser lineal con respecto al número de procesadores en este caso. Incluso si se compara, el equipo 1 con el equipo 2, el procesador Inter Core 2 Quad Q6600 ofrece resultados en menor tiempo en todas las pruebas que usan 4 procesadores. Adicional a esto cabe acotar que usar un sólo procesador no es una buena elección, debido a la gran diferencia que se tiene con respecto al resto de los resultados obtenidos.

Con esto se puede decir que en el caso de utilizar bloques de menor tamaño se debe considerar como mejor opción OpenMP para el paralelismo en el CPU, en el caso de que se desee utilizar bloques de mayor tamaño se puede considerar como mejor elección CUDA. A continuación se procede a dar los resultados obtenidos para el algoritmo de selección.

5.2.3 Algoritmo de Selección

En esta fase de pruebas se evaluó la distorsión promedio por bloques del volumen, para obtener esta métrica se calculó la distorsión total y esta se dividió entre la cantidad de bloques. Otras muestras obtenidas son la cantidad de veces que se realizó una desfragmentación, la memoria inutilizada del atlas y por último se tiene el tiempo total de refinamiento. Este último valor ya tiene sumado el tiempo requerido para la desfragmentación. Adicionalmente se utilizaron 3 tamaños para el atlas: 64mb, 128mb y 256mb. Esto con el fin de determinar la distorsión y los tiempos de procesamientos que se pueden llegar a tener para cada tamaño.

		Volumen A								
Tamaño del Bloque		17^3			33^3			65^3		
Tamaño Atlas (MB)		64	128	256	64	128	256	64	128	256
Distorsión promedio de bloques no refinado	Distancia	6,96	6,12	3,66	5,97	5,11	3,65	4,87	3,99	2,86
	Distorsión	3,77	2,18	0,65	3,42	2,23	1,23	2,95	1,94	1,35
	Distancia / Distorsión	4,27	3,15	2,67	5,21	4,24	2,53	3,46	2,12	1,43
Nro. de veces que se Desfragmento el Atlas		4	3	3	4	2	2	2	2	2
Memoria Inutilizada (MB) (Antes de Desfrag.)		28,8	72,6	92,3	27	62,2	78,4	23,1	60,28	96,8
Memoria Inutilizada (MB) (Después de Desfrag.)		2,4	2,5	7,4	6,4	6,13	24,3	14	15,1	55,9
Tiempo total de Refinamiento	Equipo 1	26,3s	41,3s	135,2s	3,3	3,5s	7,2s	0,7s	1,5s	2s
	Equipo 2	24,1s	28,8s	89,3s	1,14	2,3s	4,5	0,45s	0,9s	1,3s

Tabla 5.6: *Tabla de resultados del algoritmo de selección aplicado al Volumen A*

Para este volumen note que el tiempo de refinamiento es considerablemente alto en el bloque de menor tamaño, y aumenta más aún a medida que el atlas también lo hace. Recordemos que este volumen por su dimensión contiene una gran cantidad de bloques; esto hace que la cola de prioridad de refinamiento, el proceso de desfragmentación y el envió de datos al GPU se tomen un tiempo relativamente alto. Si se observa los casos de los bloques de mayor tamaño, los tiempos de refinamiento disminuyen debido a que es menor la cantidad de bloques que se visualizan.

En cuanto a la desfragmentación, note que aplicando la técnica desarrollada se logró utilizar desde un 30% hasta un 90% de la memoria que había quedado inutilizada. En los casos que se logró una baja utilización fue en los bloques de mayor tamaño; esto se debe a que los espacio inutilizados suelen ser muy pequeños por lo tanto es difícil almacenar bloques grandes en estas áreas.

		Volumen B								
Tamaño del Bloque		17^3			33^3			65^3		
Tamaño Atlas (MB)		64	128	256	64	128	256	64	128	256
Distorsión promedio de bloques no refinado	Distancia	25,55	12	16,85	22,92	9,84	9,13	16,83	10,58	6,58
	Distorsión	7,22	2,04	$6x10^{-4}$	7,53	4,61	0,18	7,83	4,99	2,74
	Distancia / Distorsión	15,10	8,16	0,001	11,24	5,12	3,52	12,07	8,20	4,97
Nro. de veces que se Desfragmento el Atlas		3	2	3	2	3	4	2	2	3
Memoria Inutilizada (MB) (Antes de Desfrag.)		24,3	29,25	42,73	30,6	39,7	74,5	31,6	44,9	96,49
Memoria Inutilizada (MB) (Después de Desfrag.)		1,52	6,1	8,1	9,1	10,6	15,1	17,8	22,6	74,18
Tiempo total de Refinamiento	Equipo 1	7,64s	10,8s	20,8s	0,8s	1,8s	4,7s	0,37	0,74s	1,3s
	Equipo 2	4,3s	7,3s	12,7s	0,6s	1,2s	3,1s	0,3s	0,4	0,6s

Tabla 5.7: *Tabla de resultados del algoritmo de selección aplicado al Volumen B*

En la ***Tabla 5.7*** podemos observar la misma tendencia que en el Volumen B en cuanto a la desfragmentación y el tiempo total de refinamiento. También podemos observar en estos dos

volúmenes que la distorsión promedio por bloque tiende a ser menor en algunos casos con bloques de menor tamaño. Esto es porque los bloques de menor tamaño se pueden enfocar en las zonas con mayor distorsión. Esta es una de las ventajas que ofrece también aplicar este tipo de jerarquía en lugar de un *octree.* Como se mencionó anteriormente, con un *octree* si se desea refinar un bloque obligatoriamente este debe subdividirse en máximo ocho nodos más finos, haciendo que ciertas zonas se refinen cuando no necesariamente lo requieran. En la ***Tabla 5.8*** este fenómeno se puede apreciar claramente debido a la anatomía que presenta el escarabajo.

		Volumen C								
Tamaño del Bloque		17^3			33^3			65^3		
Tamaño Atlas (MB)		64	128	256	64	128	256	64	128	256
Distorsión Final del Criterio de Selección	Distancia	0	0	0	21,95	0	0	13,4	10,7	0
	Distorsión	0	0	0	3,3	0	0	7,4	0,5	0
	Distancia / Distorsión	0	0	0	15,42	0	0	8,45	2,8	0
Nro. De veces que se Desfragmento el Atlas		2	0	0	2	0	0	2	1	0
Memoria Inutilizada (Antes de Desfrag.)		17,3	70,6	198,5	15,1	52,1	180,4	24,8	27,1	124,2
Memoria Inutilizada (Después de Desfrag.)		6,4	-	-	10,3	-	-	24,6	27,1	-
Tiempo total de Refinamiento (Con Desfrag.)	Equipo 1	3,1s	1,8s	1,8s	0,6s	0,4	0,4	1s	0,9s	0,3
	Equipo 2	1,7s	1s	1s	0,3	0,1	0,1	0,6	0,8s	0,2

Tabla 5.8: *Tabla de resultados del algoritmo de selección aplicado al Volumen C*

Note que usando bloques de tamaño 17^3 podemos almacenar todos los bloques visibles en solo 46 MB aproximadamente. Si se observa este volumen detalladamente (ver ***Figura 5.3c***) se aprecia que los miembros del escarabajo, que son de poco grosor, se extienden por todo el volumen. Debido a que estos miembros cubren un área muy pequeña, si se usan bloques pequeños no se almacena mucha información del espacio vacío que rodea al contorno (ver ***Figura 5.5***). A medida que se usan bloques más grandes se puede ver que 64MB no son suficientes.

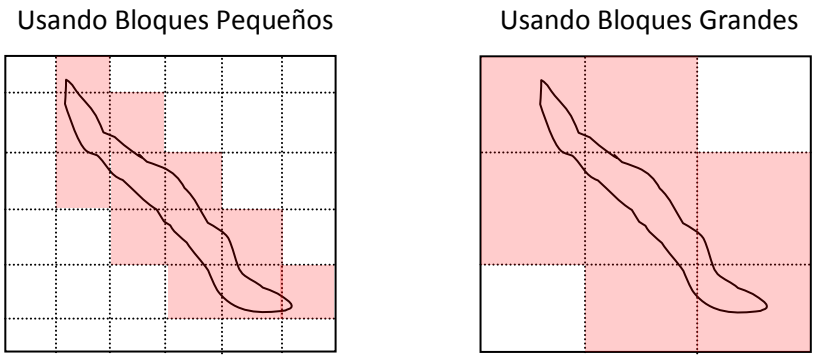


Figura 5.5*: Comparativa entre utilizar una subdivisión de menor tamaño contra una de mayor tamaño, observe que para este ejemplo usando bloques de mayor tamaño se necesita 7/9 del volumen y en el de menor tamaño se necesita 1/3 del volumen solamente.*

Ya se pudo ver que el tamaño de los bloques usados en la jerarquía es un factor importante, hasta los momentos influye en el tiempo de refinamiento y en el uso de la memoria. A medida que se usan bloques más pequeños se tiende a obtener menos distorsión y menos uso de memoria con determinados volúmenes que cubren áreas muy pequeñas, pero se pierde tiempo en el refinamiento y en el pre-procesamiento. A continuación veremos los resultados obtenidos para el despliegue del volumen multi-resolución.

5.2.3 Ray-Casting de una Pasada

Debido a que estos resultados son muy similares en los tres volúmenes se mostrará solamente los resultados del volumen B. Los resultados de los otros volúmenes se pueden ver en el ***Anexo 4*** y ***Anexo 5***. Para estas pruebas se realizaron una serie de mediciones, en el refinamiento *frame* a *frame* se procesaron desde el 10% (mínimo) hasta el 60% (máximo) de los bloques por cuadro desplegado. Esto con el fin de determinar el rendimiento a medida que el volumen se va refinando en el tiempo. La otra medida considerada es la cantidad cuadros por segundo (Hz) durante un despliegue sin refinamiento, donde ya se aplicó el criterio de selección, los bloques llegaron a su nivel de refinamiento idóneo y todos los bloques están almacenados en el atlas.

			Volumen B								
Tamaño del Bloque			17^3			33^3			65^3		
Tamaño Atlas (MB)			64	128	256	64	128	256	64	128	256
Equipo 1	Rango de tiempo de despliegue aplicando Refinamiento (Hz)	Max.	2,8	1,2	1,0	3,7	3,7	3,8	5,8	5,8	6,2
		Min.	0,5	0,1	0,2	1,6	1,3	0,9	3,2	3,1	3,1
	Tiempo de despliegue sin refinamiento frame a frame (Hz)		6,34	6,25	6,10	6,35	6,31	6,22	6,35	6,35	6,35
Equipo 2	Rango de tiempo de despliegue aplicando Refinamiento (Hz)	Max.	2,9	1,5	1,2	4,7	4,7	4,6	5,5	5,5	5,7
		Min.	0,6	0,2	0,3	2,5	1,9	1,2	3,3	2,9	2,8
	Tiempo de despliegue sin refinamiento frame a frame (Hz)		5,89	5,83	5,82	5,82	5,81	5,82	5,78	5,73	5,75

Tabla 5.9: *Tabla comparativa de los Hz (cuadros por segundos) utilizando distintas configuraciones. El rango mínimo y máximo hace referencia a refinar entre el 10% y el 60% de los bloques por cuadro respectivamente.*

Note que en la ***Tabla 5.9*** a medida que el tamaño del atlas aumenta, los cuadros por segundos disminuyen en el proceso de refinamiento. Esto sucede porque entre más espacio disponible mayor es la cantidad de bloques que se envían a la textura atlas que se encuentra en el GPU. Adicionalmente podemos observar que los cuadros por segundos durante un despliegue sin refinamiento se mantienen en un rango estable en cada prueba, solo hay una pequeña variación a medida que el tamaño del atlas aumenta debido a que el GPU accede a una textura de mayor tamaño generando una pequeña sobrecarga prácticamente despreciable. Aquí también se puede ver que el equipo 2 ofrece menor rendimiento respecto al equipo 1, a pesar de que este tiene GPU de última generación. La causa de esto pueden ser muchas, la más importante es que recordemos que el procesador Intel i3 puede ser una desventaja en casos muy específicos con respecto al otro

procesador, debido a que este utiliza dos procesadores reales y luego dos simulados. Adicionalmente hay que tener en cuenta que la memoria cache es menor y es compartida entre los dos hilos de procesamiento virtuales que tiene un núcleo; esto hace que se generan más fallos de memoria y haya una competencia por este recurso. Por lo tanto la ganancia que se obtiene en velocidad de procesamiento se puede perder en algunos casos debido a estas desventajas.

De aquí podemos concluir de igual forma que el tamaño del bloque influye mucho en el rendimiento de la aplicación y en la distorsión final del volumen. Debemos tener en cuenta que el proceso de refinamiento toma más tiempo a medida que los bloques, en los que se subdivide el volumen, sean de menor tamaño. También sucede cuando se cuenta con más memoria para el atlas. La ventaja de utilizar bloques pequeños es que se puede lograr resultados con menor distorsión dependiendo de la estructura presente en el volumen. La idea es que se debe saber elegir un balance entre la calidad de la imagen final y el tiempo de respuesta de la aplicación.

A continuación mostraremos los resultados cualitativos y realizaremos comparaciones a nivel visual con distintas configuraciones y con trabajos previos.

5.3 Resultados Cualitativos

Se tomaron una serie de muestras visuales para comparar los resultados utilizando distinta cantidad de memoria y con bloques de diferentes tamaños.

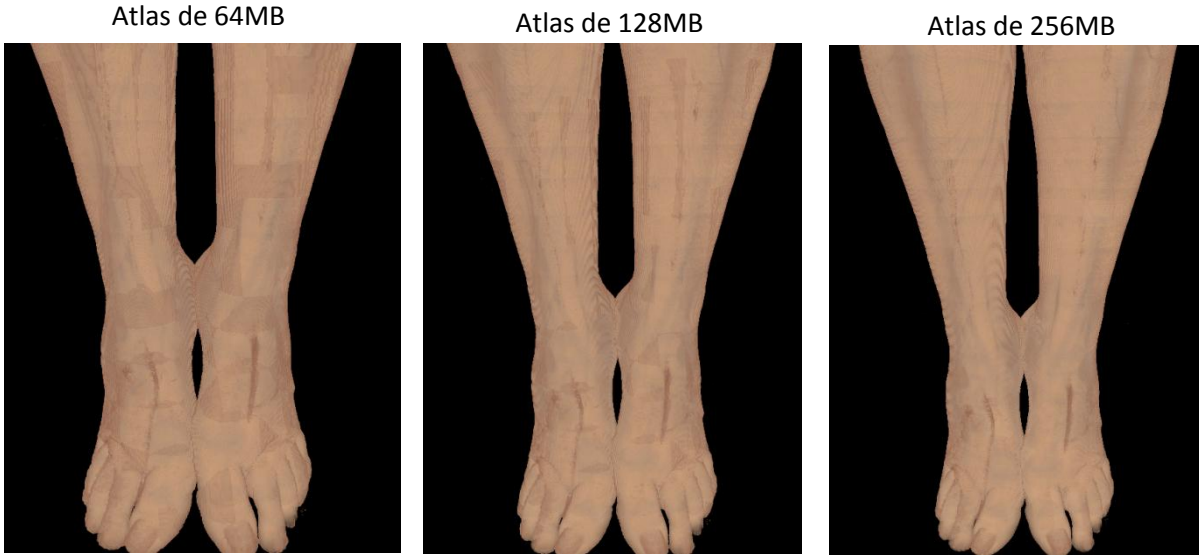


Figura 5.6: *Aquí se compara el volumen utilizando distintos tamaños del Atlas. Para esta prueba se utilizaron bloques de tamaño* 33^3

En la ***Figura 5.6*** se puede observar como la distorsión disminuye a medida que se cuenta con un atlas de mayor tamaño. Note que a pesar de que el volumen es relativamente grande, la distorsión final es casi imperceptible. Se utilizó un tamaño intermedio de bloque (33^3) para lograr un balance entre calidad de la partición y tiempo de respuesta.

En la ***Figura 5.7*** *y la* ***Figura 5.8*** se nota una diferencia importante cuando se usan bloques de distintos tamaños; utilizando bloques más grandes con poca memoria presenta mucha distorsión comparado con utilizar bloques más pequeños.

Bloque de 33^3 Bloque de 17^3

Figura 5.7: *Comparativa de utilizando bloques de distintos tamaños pero con la misma capacidad del atlas. Note como se distorsiona más el volumen por agotamiento del espacio utilizando un bloque de mayor tamaño para este volumen.*

Bloque de 65^3 Bloque de 33^3 Bloque de 17^3

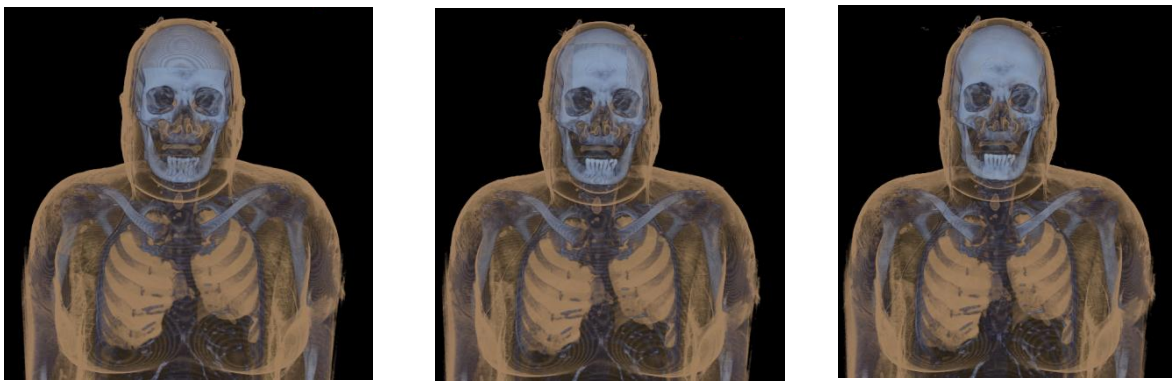


Figura 5.8: *Comparativa utilizando bloques de distintos tamaños pero con la misma capacidad del atlas. Note como los artefactos disminuyen a pesar que se cuenta con la misma cantidad de memoria.*

Para el proceso de refinamiento se realizó una prueba utilizando los tres criterios de selección, en la ***Figura 5.9*** podemos notar la diferencia entre cada criterio utilizado. Compare los resultados y note que el criterio basado en la distancia al punto de interés junto con la distorsión se puede tener un punto intermedio entre utilizar sólo uno de ellos.

Criterio de selección Basado solo en la Distancia al Punto de interés

Criterio de selección Basado solo en la distorsión

Criterio de selección Basado en la distorsión y la distancia

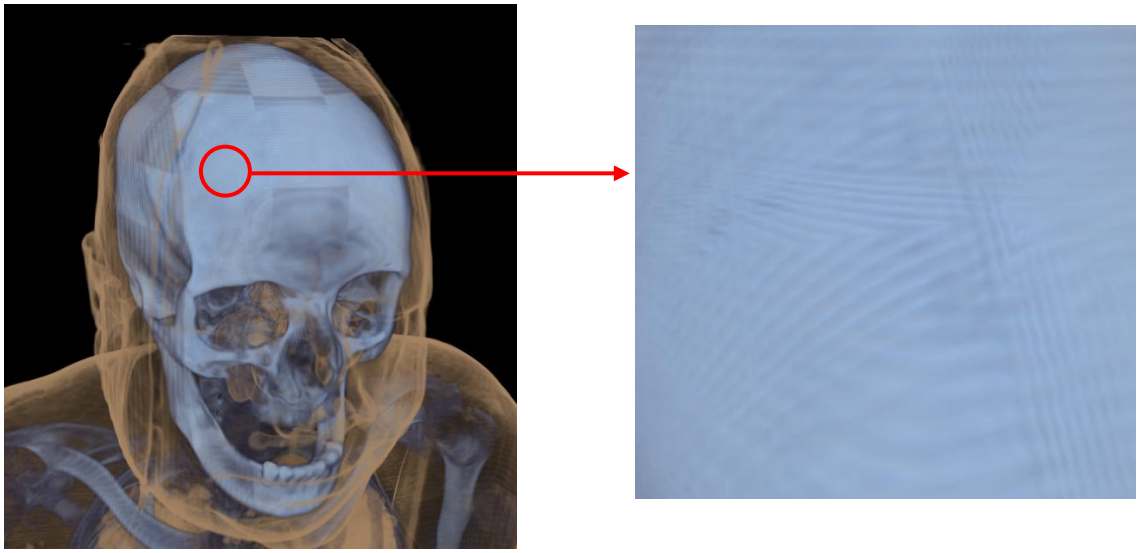
Figura 5.9: *Comparativa del criterio de selección, aquí se utilizó un atlas de 64MB y bloques de* 33^3.

Con respecto a las pruebas visuales del prototipo se necesitan hacer comparaciones con trabajos previos para ver la calidad de estos resultados obtenidos.

5.4 Comparaciones con trabajos previos

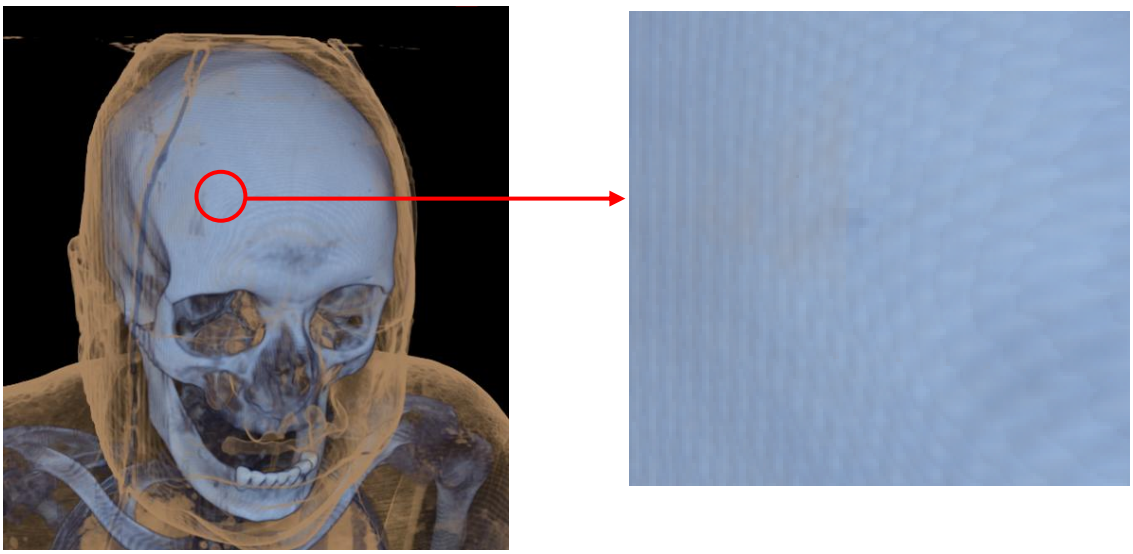
Para realizar comparaciones utilizamos el trabajo realizado por *Carmona* [**2**]. Como habíamos mencionado anteriormente, este se basó en una jerarquía *octree* y en un despliegue utilizando *Ray Casting* de múltiples pasadas. En la ***Figura 5.10*** se hizo una comparativa aplicando una configuración equivalente en ambos prototipos.

Bricks de 16^3 basado en un Octree, memoria de textura restringida a 64MB



(A)

Bloques de 17^3 basado en una jerarquía por bloques, memoria atlas restringida a 64MB



(B)

Figura 5.10: *Comparativa del criterio de selección, aquí se utilizó un atlas de 64MB y bloques de 33^3. Note que en el zoom realizado se puede ver otro tipo de artefacto visual debido a las múltiples pasadas del Ray Casting.*

En la ***Figura 5.10a*** se obtiene que el volumen final aplicando mismo criterio de selección presenta una gran cantidad de artefactos visuales; primordialmente esto se debe a la jerarquía de

tipo *octree*, donde el refinamiento de un bloque implica subdividirlo en otros ocho sub-bloques obligatoriamente dando como resultado que se refinen ciertas áreas no deseadas. Adicional a esto observe que se presenta otro tipo de artefacto visual originado por las múltiples pasadas del algoritmo de *Ray Casting*, debido al paso del rayo entre un bloque y otro. En el prototipo realizado (ver ***Figura 5.10b)*** se puede ver una superficie es más uniforme y con poca cantidad de artefactos logrando aprovechar un poco más la memoria disponible para este caso de prueba utilizado. A continuación mostraremos las conclusiones y los trabajos futuros a esta investigación.

Conclusiones y Trabajos Futuros

En este trabajo especial de grado se desarrolló una aplicación para visualizar volúmenes de gran tamaño aplicando una jerarquía 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.

Se pudo llegar a la conclusión de que el tamaño de los bloques y del atlas cumplen un factor muy importante. 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 de envió de datos 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.

Cuando se utilizan bloques muy grandes se llega a que la distorsión global del volumen puede ser muy elevada, pero debido a que la cantidad total de bloques es pequeña, el refinamiento es mucho más rápido y en el cálculo de la distorsión vimos que la mejor opción es la utilización del GPU, pues se aprovecha mas su poder de computo debido a que son menos la cantidad de llamadas al *kernel* dado a que en cada ejecución se procesan mas vóxeles.

En cuanto al rendering utilizando *Ray Casting* de una pasada, se observó que el tamaño del bloque no influye en el tiempo de rendering, mientras que el tamaño del atlas si pero ligeramente. El tamaño del atlas afecta solamente el tiempo utilizado en el criterio de selección cuando se utiliza la desfragmentación, pues debe reestructurarse todo el atlas para acoplar las áreas fragmentadas y puedan ser utilizables. Esto implica que se deben enviarse al GPU de nuevo todos los bloques, gracias a esto el tiempo de refinamiento incrementa.

Se comparó el resultado obtenido con el trabajo previo realizado por *Carmona* [**2**]. En la prueba realizada se vio que una jerarquía por bloques ofrece un resultado más fino usando la misma cantidad de memoria y el mismo tamaño de los bloques. La razón principal es que la subdivisión del volumen en un *octree* no permite un refinamiento de un área específica dentro del *brick* a refinar, obligando que ciertas áreas no deseadas se refinen, haciendo un uso menos eficiente de la memoria. Adicionalmente se mostró como se soluciona un artefacto visual originado por las múltiples pasadas del algoritmo de *Ray Casting*, pues se nota la transición del rayo entre un bloque y otro. Para el caso de utilizar una jerarquía por bloques, el artefacto visual ocurre solamente en la transición entre dos bloques que tengan distintos niveles de detalles.

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:

- Realizar una optimización al algoritmo utilizado para la desfragmentación, pues la solución utilizada siempre se deja una cierta cantidad memoria inutilizada. Cabe destacar que en la literatura no hay trabajos previos de desfragmentación de texturas de tres dimensiones, haciendo atractivo este punto para la investigación.

- Aplicar un despliegue adaptativo en el algoritmo de *Ray Casting*, donde el espaciado entre muestras varíe según el nivel de detalle del bloque.

- Adaptar una de las técnicas presentadas en la ***sección 2.6*** para la reducción de artefactos entre bloques adyacentes que tengan distintos niveles de detalle.

- Buscar una optimización al algoritmo implementado en el GPU para el cálculo de la distorsión. Se debe tratar de enviar múltiples bloques al GPU para calcular la distorsión en menor tiempo. La solución implementada solo permite el cálculo de un bloque a la vez.

Bibliografía

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

[2] Rhadames Carmona, "Visualización Multi-Resolución de Volúmenes de Gran Tamaño," Centro de Computación Gráfica, Universidad Central de Venezuela, Caracas, Venezuela, Tesis Doctoral 2008.

[3] Rhadamés Carmona and Omaira Rodríguez, "Cubos Marchantes: Una Implementación Eficiente," in *XXV Conferencia Latinoamericana de Informática*, Paraguay-La Asunción, 1999.

[4] Frank Goetz, Theodor Junklewitz, and Gitta Domik, "Real-Time Marching Cubes on the Vertex Shader," in *Research Group Computer Graphics, Visualization and Image Processing*, University of Paderborn, Germany, 2005.

[5] Max Nelson, "Optical Models for Direct Volume Rendering," in *Visualization in Scientific Computing*., 1995, pp. 35-40.

[6] Universität Erlangen-Nürnberg. (2010) The Volume Library. [Online]. http://www9.informatik.uni-erlangen.de/External/vollib/

[7] Paolo Sabella, "A rendering algorithm for visualizing 3D scalar fields," in *ACM SIGGRAPH '88*, vol. 22, 1988, pp. 51–58.

[8] Peter Williams and Nelson Max, "A volume density optical model," in *Workshop on Volume Visualization '92*, 1992, pp. 61–68.

[9] Philippe Lacroute and Marc Levoy, "Fast Volume Rendering Using Shear-Warp Factorization of the Viewing Transformation," in *SIGGRAPH '94*, Orlando, Florida, 1994, pp. 451-458.

[10] NVidia®. (2010) World Leader in Visual Computing Technologies. [Online]. http://www.nvidia.com

[11] Microsoft®. (2010) Reference for HLSL developed by Microsoft for use with the Microsoft Direct3D API. [Online]. http://msdn.microsoft.com/en-us/library/bb509638%28v=VS.85%29.aspx

[12] OpenGL®. (2010) The Industry Standard for High Performance Graphics. [Online]. http://www.opengl.org

[13] Joe Kniss, Gordon Kindlmann, and Charles Hansen, "Multidimensional Transfer Functions for Interactive Volume Rendering," in *IEEE Transactions on Visualization and Computer Graphics*, vol. 8, 2002, pp. 270-285.

[14] Engel Klaus, Kraus Martin, and Ertl Thomas, "High Quality Pre- Integrated Volume Rendering Using Hardware-Accelerated Píxel Shading," in *Siggraph/Eurographics Workshop on Graphics Hardware*, California-USA, 2001, pp. 9-16.

[15] D. Ruijters and A. Vilanova, "Optimizing GPU Volume Rendering," in *WSCG - Winter School of Computer Graphics*, vol. 14, 2006, pp. 9-16.

[16] J. Krüger and R. Westermann, "Acceleration Techniques for GPU-based Volume Rendering," in *Visualization '03*, Washington-USA, 2003, pp. 287-292.

[17] Marc Levoy, "Display of surfaces from volume data," in *IEEE Computer Graphics and Applications*, vol. 8:3, 1988, pp. 29 - 37.

[18] Marc Levoy, "Efficient Ray Tracing of Volume Data," in *ACM Transactions on Graphics*, vol. Vol.

9 - No. 3, 1990, pp. 245-261.

[19] John Danskin and Pat. Narran, "Fast Algorithms for Volume Ray," in *Workshop on Volume Visualization '02*, Boston-Massachussets-USA, 1992, pp. 91–98.

[20] A. A. Mirin et al., "Very High Resolution Simulation of Compressible Turbulence on the IBM-SP System," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, 1999, p. 70.

[21] P. Bhaniramka and Y. Demange, "OpenGL Volumizer: A Toolkit for High Quality Volume Rendering of Large Data sets," in *IEEE Symposium on Volume Visualization and Graphics*, 2002.

[22] Eric LaMar, Mark Duchaineau, Bernd Hamann, and Kenneth Joy, "Multiresolution Techniques for Interactive Texturebased Rendering of Arbitrarily Oriented Cutting Planes," in *Data Visualization 2000, The Joint Eurographics and IEEE TVCG conference on Visualization*, 2000, pp. 105-114.

[23] Patric Ljung, Claes Lundström, and Anders Ynnerman, "Multiresolution Interblock Interpolation in Direct Volume Rendering," in *Eurographics/IEEE-VGTC Symposium on Visualization*, 2006, pp. 259-266.

[24] Eric LaMar, Mark Duchaineau, Bernd Hamann, and Kenneth Joy, "Multiresolution Techniques for Interactive Texture-based Volume Visualization," in *Visualization '99*, California-USA, 1999, pp. 355-361.

[25] Imma Boada, Isabel Navazo, and Roberto Sopigno, "A 3D texture-based octree volume visualization algorithm," in *The Visual Computer*, vol. 17, 2000, pp. 185-197.

[26] Xinyue Li and Han-Wei Shen, "Time-Critical Multiresolution Volume Rendering Using 3D Texture Mapping Hardware," in *IEEE Volume Visualization and Graphics Symposium '02*, Boston-USA, 2002.

[27] David Laur and Pat Hanrahan, "Hierarchical splatting: a progressive refinement algorithm for volume rendering," in *SIGGRAPH Comput. Graph.*, 1991, pp. 285-288.

[28] Imma Boada, Isabel Navazo, and Roberto Sopigno, "Multiresolution volume visualization with a texture-based octree," in *The Visual Computer*, 2001, pp. 185-197.

[29] Jinzhu Gao Gao, Chaoli Wang, and Han-Wei Shen, "Parallel Multiresolution Volume Rendering of Large Data Sets with Error-Guided Load Balancing," in *Parallel Comput.*, 2005, pp. 185-204.

[30] Wang Chaoli, Garcia Antonio, and Shen Han-Wei, "Interactive Level-of-Detail Selection Using Image-Based Quality Metric for Large Volume Visualization," in *IEEE Transactions on Visualization and Computer Graphics*, 2007, pp. 122-134.

[31] Jane Wilhelms and Allen VanGelder, "Multi-Dimensional Trees for Controlled Volume Rendering and Compression," in *IEEE Visualization '02*, 2002, pp. 315-322.

[32] S. Guthe and W. Strasser, "Advanced Techniques for High-Quality Multi-Resolution Volume Rendering.ume Rendering and Compression," in *Computers & Graphics*, 2004, pp. 51-58.

[33] S. Guthe, M. Wand, J. Gonser, and W. Strasser, "Interactive Rendering of Large Volume Data Sets," in *IEEE Visualization '02*, 2002, pp. 53-60.

[34] Patric Ljung, Claes Lundstrom, Anders Ynnerman, and Ken Museth, "Transfer Function Based Adaptive Decompression for Volume Rendering of Large Medical Data Sets," in *IEEE Symposium on Volume Visualization and Graphics*, 2004, pp. 25-32.

[35] Deane B. Judd, "Hue saturation and lightness of surface colors with chromatic illumination," *Journal of the Optical Society of America*, vol. 3, no. 1, p. 2, 1949.

[36] Chaoli Wang, Antonio Garcia, and Han WeiShen, "Interactive Level-of-Detail Selection Using Image-Based Quality Metric for Large Volume Visualization," *IEEE Transactions on Visualization*

and Computer Graphics, vol. 13, no. 1, pp. 122-134, 2007.

[37] Bradford Chamberlain, Tony DeRose, Dani Lischinski, David Salesin, and John Snyder, "Fast rendering of complex environments using a spatial hierarchy," , 1996, pp. 132-141.

[38] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner., *OpenGL® Programming Guide: The Official Guide to Learning*, 3rd ed., 1999.

[39] Plate John, Tirsana Michael, Carmona Rhadamés, and Froehlich Bernd, "Octreemizer: A Hierarchical Approach for Interactive Roaming Through Very Large Volumes," in *Joint Eurographics - IEEE TCVG Symposium on Visualization '02*, 2002, pp. 53-60.

[40] Zimmermann Kurt, Westermann Rüdiger, Ertl Thomas, Hansen Chuck, and Weiler Manfred, "Level-of-Detail Volume Rendering via 3D Textures," in *Volume Visualization and Graphics, IEEE Symposium on*, Los Alamitos, CA, USA, 2000, pp. 7-13.

[41] Rhadamés Carmona, Gabriel Rodríguez, and Bernd Fröhlich, "Reducing Artifacts between Adjacent Bricks in Multi-resolution Volume Rendering," in *Advances in Visual Computing*.: Springer Berlin / Heidelberg, 2009, pp. 644-655.

[42] Patric Ljung, "Adaptive Sampling in Single Pass, GPU-based *Ray Casting* of Multiresolution," in *Eurographics/IEEE International Workshop on Volume Graphics*, Boston-USA, 2006, pp. 39-46.

[43] Christopher Lux and Bernd Frohlich, "GPU-Based *Ray Casting* of Multiple Multi-resolution Volume Datasets," in *ISVC '09 Proceedings of the 5th International Symposium on Advances in Visual Computing: Part II* , 2009, pp. 104-116.

[44] Khronos®. (2010) Open Standards for Media Authoring and Acceleration. [Online]. http://www.khronos.org/

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

[46] B. Hapman, G. Jost, and R. Van Der Pas, "Using OpenMP Portable Shared Memory Parallel Programming.," *The MIT Press*, 2008.

[47] ©ISO/IEC, "Programming Languages - C," International Standard ISO/IEC 9899, 1999.

[48] NVidia. (2010) NVIDIA ® CUDA™ Architecture. [Online]. http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf

[49] NVidia®, *NVIDIA CUDA C Programming Guide V4.0*.: NVIDIA CUDA™, 2011.

[50] Microsoft. Microsoft Foundation Class (MFC). [Online]. http://msdn.microsoft.com/en-us/library/d06h2x6e(vs.71).aspx

[51] Meister Eduard Gröller, Georg Glaeser, and Johannes Kastner. Stag beetle. [Online]. http://www.cg.tuwien.ac.at/research/publications/2005/dataset-stagbeetle/

[52] Sweeney J. and Mueller K., "Shear-warp Deluxe: The Shear-warp Algorithm Revisited," in *Joint Eurographics - IEEE TCVG Symposium on Visualization*, Barcelona-España, 2002, pp. 95 - 104.

[53] J. P. Schulze, M. Kraus, U. Lang, and T. Ertl, "Integrating Pre-Integration into the Shear-Warp Algorithm," in *Eurographics/IEEE TVCG Workshop on Volume Graphics*, Tokio-Japón, 2003, pp. 109-118.

[54] Jonathan Camacho, Larry Márquez, Rhadamés Carmona, and Robinson Rivas, "Implementación de Shear-warp distribuido utilizando hardware gráfico," in *VII Congreso Internacional de Métodos Numéricos en Ingeniería y Ciencias Aplicadas*, Táchira-Venezuela, 2004.

[55] R. Crawfis and N. Max, "Texture Splats for 3D Scalar and Vector Field Visualization," in *Visualization '93, IEEE*, 1993, pp. 261-265.

[56] J. Waage, "State of the Art Parallel Computing in Visualization using CUDA and OpenCL," in *INF358 Seminar in Visualization*, 2008.

[57] Wang Chaoli, Gao Jinzhu, and Shen Han-Wei, "Parallel Multiresolution Volume Rendering of Large Data Sets with Error-Guided Load Balancing," in *Eurographics Symposium on Parallel Graphics and Visualization*, 2004, pp. 23-40.

[58] Jung Pill Choi, Sung M. Kwon, and Jong B. Ra, "Efficient multiresolution volume rendering scheme," in *SPIE*, 2000, pp. 134-141.

[59] Y.T. Yang, H.S. Seah, and F. Lin, "Multi-Resolution Volume Rendering Based on Shear-Warp Factorization," in *International Workshop on Volume Graphics*, 1999, pp. 49–64.

[60] Khronos OpenCL Working Group, *The OpenCL 1.1 Specification*, 36th ed., Aaftab Munshi, Ed., 2010.

[61] Apple®. (2010) [Online]. http://www.apple.com/

[62] Extreme Programming: A gentle introduction. [Online]. http://www.extremeprogramming.org/

[63] Intel®. (2010) ©Intel Corporation. [Online]. http://intel.com

Anexos

```
for(int j=0;j<4;++j){

        const int sizeVol=FAST_INT_CEIL(size_block,(1<<j));
        const cudaExtent volSize = make_cudaExtent(sizeVol,sizeVol,sizeVol);

        if(firstTime) cudaMalloc3DArray(&(d_volArray[j]), &channelDescUC1, volSize);

        // copy data to 3D array
        cudaMemcpy3DParms copyParams = {0};
        copyParams.dstArray = d_volArray[j];
        copyParams.srcPtr   = make_cudaPitchedPtr((void*)(vol[j]), volSize.width*sizeof(uchar1),
                                                  volSize.width,volSize.height);
        copyParams.extent   = volSize;
        copyParams.kind     = cudaMemcpyHostToDevice;

        cudaMemcpy3D(&copyParams);

        if(firstTime){
                // bind array to 3D texture
                if(j==0) cudaBindTextureToArray(volume, d_volArray[0], channelDescUC1);
                if(j==1) cudaBindTextureToArray(lvl0,   d_volArray[1], channelDescUC1);
                if(j==2) cudaBindTextureToArray(lvl1,   d_volArray[2], channelDescUC1);
                if(j==3) cudaBindTextureToArray(lvl2,   d_volArray[3], channelDescUC1);
        }
}
if(changeFTinCuda){
        static cudaArray* cuArrayFT;

        if(firstTime) cudaMallocArray(&cuArrayFT, &channelDescF4, 256, 1);

        cudaMemcpyToArray(cuArrayFT, 0, 0, ft_luv, 256*4*sizeof(float),
                          cudaMemcpyHostToDevice);

        if(firstTime){

                tex_ft_luv.addressMode[0] = cudaAddressModeClamp;
                tex_ft_luv.filterMode = cudaFilterModeLinear;
                tex_ft_luv.normalized = true;

                cudaBindTextureToArray(tex_ft_luv, cuArrayFT, channelDescF4);
                checkErrorCuda("cudaBindTextureToArray 2");

        }
        changeFTinCuda=false;
}

const dim3 dimGrid(SIZE_BLOCK_CUDA,cBlock);
const dim3 dimBlock(SIZE_BLOCK_CUDA,SIZE_BLOCK_CUDA,1);

distortion<<<dimGrid,dimBlock>>>(distortion_remote, size_block);

cudaThreadSynchronize();

cudaMemcpy(error, distortion_remote, cBlock*3*SIZE_BLOCK_CUDA*sizeof(float), cudaMemcpyDeviceToHost);

for(int j=0;j<cBlock;++j){
        for(int i=0;i<SIZE_BLOCK_CUDA;++i){
                distortion_host[0]+= error[SIZE_BLOCK_CUDA*3*j + i*3 + 0]; // Error del nivel de
detalle 1
                distortion_host[1]+= error[SIZE_BLOCK_CUDA*3*j + i*3 + 1]; // Error del nivel de
detalle 2
                distortion_host[2]+= error[SIZE_BLOCK_CUDA*3*j + i*3 + 2]; // Error del nivel de
detalle 3
        }
}
```

Anexo 1: *Algoritmo ejecutado en el Host para cálculo de la distorsión en CUDA*

```
texture<uchar1, 3, cudaReadModeElementType> volume;
texture<uchar1, 3, cudaReadModeElementType> lvl0;
texture<uchar1, 3, cudaReadModeElementType> lvl1;
texture<uchar1, 3, cudaReadModeElementType> lvl2;
texture<float4, 1, cudaReadModeElementType> tex_ft_luv;

#define pX threadIdx.x
#define pY threadIdx.y
#define pZ blockIdx.x
#define b blockIdx.y
__global__ void distortion(float* err, unsigned int size_block)
{

        const unsigned char bX = b%cBlockAxis;
        const unsigned char bY = (b/cBlockAxis)%cBlockAxis;
        const unsigned char bZ = (b/(cBlockAxis*cBlockAxis))%cBlockAxis;

        const unsigned short pVX = bX*SIZE_BLOCK_CUDA + pX;
        const unsigned short pVY = bY*SIZE_BLOCK_CUDA + pY;
        const unsigned short pVZ = bZ*SIZE_BLOCK_CUDA + pZ;

        const unsigned int globalPosBlock=pY*SIZE_BLOCK_CUDA + pX;

        register float aux=0.0f;

        if(pVX<size_block && pVY<size_block && pVZ<size_block){

                float4 v;
                uchar1 voxel  = tex3D (volume, pVX,  pVY,  pVZ); // Obtener el voxel
                v.x=float(voxel.x)/255.0f;
                v.y=getVoxel(2.0f, pVX,  pVY,  pVZ, lvl0);
                v.z=getVoxel(4.0f, pVX,  pVY,  pVZ, lvl1);
                v.w=getVoxel(8.0f, pVX,  pVY,  pVZ, lvl2);

                float4 rgba  = tex1D (tex_ft_luv, v.x);  // Voxel -> CIELUV
                float4 rgba0 = tex1D (tex_ft_luv, v.y);
                float4 rgba1 = tex1D (tex_ft_luv, v.z);
                float4 rgba2 = tex1D (tex_ft_luv, v.w);

                eLocal[0][globalPosBlock] = diffColor(rgba,rgba0);
                eLocal[1][globalPosBlock] = diffColor(rgba,rgba1);
                eLocal[2][globalPosBlock] = diffColor(rgba,rgba2);

        }else{
                eLocal[0][globalPosBlock] = 0.0f;
                eLocal[1][globalPosBlock] = 0.0f;
                eLocal[2][globalPosBlock] = 0.0f;
        }

        __syncthreads();

        if(pVX<size_block && pVY<size_block && pVZ<size_block){
                if(pX==0){
                        for(register unsigned char i=0;i<3;++i){
                                aux=0.0f;
                                for(register unsigned short j=0;j<SIZE_BLOCK_CUDA;++j)
                                        aux+=eLocal[i][pY*SIZE_BLOCK_CUDA + j];
                                eSubTotal[i][pY]=aux;
                        }
                }
        }else{
                eSubTotal[0][pY] = 0.0f;
                eSubTotal[1][pY] = 0.0f;
                eSubTotal[2][pY] = 0.0f;
        }

        __syncthreads();

        if(pVX<size_block && pVY<size_block && pVZ<size_block){
                if(pX==0 && pY==0){
                        for(register unsigned char i=0;i<3;++i){
                                aux=0.0f;
                                for(register unsigned char j=0;j<SIZE_BLOCK_CUDA;++j)
                                        aux+=eSubTotal[i][j];

                                err[SIZE_BLOCK_CUDA*3*b + pZ*3 + i]=aux;
                        }
                }
        }else{
                err[SIZE_BLOCK_CUDA*3*b + pZ*3 + 0] = 0.0f;
```

```
                err[SIZE_BLOCK_CUDA*3*b + pZ*3 + 1] = 0.0f;
                err[SIZE_BLOCK_CUDA*3*b + pZ*3 + 2] = 0.0f;
        }
}
```

Anexo 2: *Algoritmo ejecutado en el Device para cálculo de la distorsión en CUDA*

```
// Textura de Indices
uniform sampler3D index;
// Textura de Atlas
uniform sampler3D volume;
// la tabla precomputada de integrales
uniform sampler3D integrals;
// minima coordenada de textura
uniform vec3      minTexCoord;
// máxima coordenada de textura
uniform vec3      maxTexCoord;
// distancia entre muestras en coodenadas de volumen
uniform float     ray_step;
// tamaño del volumen (cantidad de bloques * tamaño del bloque), en la posicion 4 se almacena el tamaño
de un bloque
uniform vec4      sizeVolumenBlocks;
// tamaño del atlas
uniform vec3     sizeTextureAtlas;


float GetSample(in vec3 pos)
{

  vec4 pmin = texture3D(index, pos);

  if(pmin[3]==-1.0) return 0.0;  //Bloque Transparente

  vec3 indexVoxel = mod(pos * sizeVolumenBlocks.xyz, sizeVolumenBlocks[3])
                    * pmin[3] * ((sizeVolumenBlocks[3]-1)/sizeVolumenBlocks[3]);

  return texture3D(volume, (pmin.xyz + 0.5 + indexVoxel)/sizeTextureAtlas).r;

}


void main()
{
        vec4  result=vec4(0.0, 0.0, 0.0, 1.0);
        vec4  col;
        float opacity = 0.0;
        vec4 posRay  = gl_TexCoord[0];
        vec4 view  = normalize(gl_TexCoord[0] - gl_ModelViewMatrixInverse * vec4(0.0, 0.0, 0.0, 1.0));
view.w = 0.0;

        vec3 coord;
        coord[1] = 0.0;
        coord[2] = 0.5;
        while (opacity<=OP_THRESHOLD)
        {
                coord[0] = coord[1];
                posRay += ray_step * view;
                vec3 signos = (maxTexCoord-posRay.xyz)*(posRay.xyz-minTexCoord);
                if (signos[0]>=0.0 && signos[1]>=0.0 && signos[2]>=0.0) //Determina si posRay esta
dentro del volumen
                {
                        coord[1] = GetSample(posRay.xyz);
                        col=clamp(texture3D(integrals, coord), 0.0, 1.0);
                }else break;


                result.rgb += result.a * col.rgb;

                result.a   *= 1.0-col.a;  // transparencia en el brick
                opacity += col.a*(1.0-opacity);              // good
        }
        result.a = 1.0 - result.a;
        if(result.a<0.05) discard;
        gl_FragColor = result;
}
```

Anexo 3: *Programas de fragmento para el despliegue de Ray Casting en una pasada*

			Volumen A								
Tamaño del Bloque			17^3			33^3			65^3		
Tamaño Atlas (MB)			64	128	256	64	128	256	64	128	256
Equipo 1	Rango de tiempo de despliegue aplicando Refinamiento (Hz)	Max.	1,2	0,7	0,2	1,8	2,3	1,1	3,3	4,5	4,9
		Min.	0,1	0,03	0,008	0,7	0,4	0,2	1,9	1,2	1,0
	Tiempo de despliegue sin refinamiento frame a frame (Hz)		5,21	5,12	4,8	5,32	5,27	5,20	5,39	5,33	5,33
Equipo 2	Rango de tiempo de despliegue aplicando Refinamiento (Hz)	Max.	1,8	1,1	0,3	3,5	2,3	1,9	4,5	5,0	5,0
		Min.	0,1	0,05	0,01	1,1	0,6	0,5	2,2	1,4	1,7
	Tiempo de despliegue sin refinamiento frame a frame (Hz)		5,00	5,07	5,04	5,12	5,08	5,05	5,07	5,04	5,09

Anexo 4: *Tabla comparativa en cuadros por segundos del despliegue aplicando refinamiento frame a frame en el volumen A*

			Volumen C								
Tamaño del Bloque			17^3			33^3			65^3		
Tamaño Atlas (MB)			64	128	256	64	128	256	64	128	256
Equipo 1	Rango de tiempo de despliegue aplicando Refinamiento (Hz)	Max.	1,8	3,5	3,5	2,8	3.3	3,3	4,3	4,4	4,4
		Min.	0,4	1,1	1,1	1,9	2.7	2,6	3,3	2,2	2,1
	Tiempo de despliegue sin refinamiento frame a frame (Hz)		4,43	4,50	4,48	4,55	4,51	4,49	4,59	4,59	4,59
Equipo 2	Rango de tiempo de despliegue aplicando Refinamiento (Hz)	Max.	1,5	3,4	3,4	3,1	4,1	4,1	4,1	4,1	4,1
		Min.	0,8	1,2	1,2	2,1	3,0	3,0	3,1	3,1	3,0
	Tiempo de despliegue sin refinamiento frame a frame (Hz)		4,21	4,23	4,24	4,19	4,18	4,18	4,20	4,20	4,22

Anexo 5: *Tabla comparativa en cuadros por segundos del despliegue aplicando refinamiento frame a frame en el volumen C*