UNIVERSIDAD CENTRAL DE VENEZUELA
FACULTAD DE CIENCIAS
POSTGRADO EN CIENCIAS DE LA COMPUTACIÓN

VISUALIZACIÓN MULTI-RESOLUCIÓN DE VOLÚMENES UTILZIANDO ATLAS DE TEXTURA

Trabajo de Grado de Maestría presentado ante la
Ilustre Universidad Central de Venezuela por el
Licenciado Francisco Sans, para optar al título de
Magister Scientiarum en Ciencias de la Computación

Tutor: Prof. Rhadamés Carmona

Caracas - Venezuela
Marzo 2018

UNIVERSIDAD CENTRAL DE VENEZUELA
FACULTAD DE CIENCIAS
COMISIÓN DE ESTUDIOS DE POSTGRADO

Comisión de Estudios de Postgrado

VEREDICTO

Quienes suscriben, miembros del jurado designado por el Consejo de la Facultad de Ciencias de la Universidad Central de Venezuela, para examinar el **Trabajo de Grado** presentado por: **Francisco Javier Sans Palacios, Cédula de identidad Nº** 18244557, bajo el título "**Visualización multi-resolución de volúmenes utilizando atlas de textura**", a fin de cumplir con el requisito legal para optar al grado académico de **MAGÍSTER SCIENTIARUM, MENCIÓN CIENCIAS DE LA COMPUTACIÓN** , dejan constancia de lo siguiente:

1.- Leído como fue dicho trabajo por cada uno de los miembros del jurado, se fijó el día 22 de Marzo de 2018 a las 09:00 AM., para que el autor lo defendiera en forma pública, lo que éste hizo en el auditorio Manuel Bemporad de la Escuela de Computación de la UCV, mediante un resumen oral de su contenido, luego de lo cual respondió satisfactoriamente a las preguntas que le fueron formuladas por el jurado, todo ello conforme con lo dispuesto en el Reglamento de Estudios de Postgrado.

2.- Finalizada la defensa del trabajo, el jurado decidió **aprobarlo**, por considerar, sin hacerse solidario con la ideas expuestas por el autor, que se ajusta a lo dispuesto y exigido en el Reglamento de Estudios de Postgrado.

Para dar este veredicto, el jurado estimó que el trabajo examinado muestra una exhaustiva revisión bibliográfica, una pulida presentación del documento, así como una clara defensa del trabajo y de los resultados obtenidos.

3.- El jurado por unanimidad decidió otorgar la calificación de EXCELENTE al presente trabajo por considerarlo de excepcional calidad con resultados preliminares ya publicados en un congreso internacional y una revista científica e indexada, y constituye un aporte novedoso en el área de visualización de volúmenes multi-resolución, en particular por proponer un esquema eficiente en el acceso de la memoria de la textura 3D, y un estudio comparativo de desempeño entre los esquemas de procesamiento paralelo gráfico utilizados.

POSTGRADO EN CIENCIAS DE LA COMPUTACIÓN
Universidad Central de Venezuela

En fe de lo cual se levanta la presente ACTA, a los 22 días del mes de Marzo del año 2018, conforme a lo dispuesto en el Reglamento de Estudios de Postgrado, actuó como Coordinador del jurado el Dr. Rhadamés Carmona.

El presente trabajo fue realizado bajo la dirección del Prof. Rhadamés Carmona.

Víctor Theoktisto / C.I. 5.223.604
Universidad Simón Bolívar

Walter Hernández / C.I. 13.246.769
Universidad Central de Venezuela

Rhadamés Carmona / C.I. 10804242
Universidad Central de Venezuela
Tutor

POSTGRADO EN CIENCIAS DE LA COMPUTACION
Universidad Central de Venezuela

RC 22/03/2018.

Resumen

La visualización de volúmenes es un área de suma importancia en la visualización científica. Con los avances tecnológicos, es cada vez más común ver volúmenes de mayor tamaño, los cuales muchas veces sobrepasan las capacidades de memoria de las tarjetas gráficas. Para solventar esto, se han desarrollado algoritmos de visualización multi-resolución que permiten visualizar diferentes áreas del volumen en distintas resoluciones. Actualmente una de las técnicas más utilizadas es la visualización con jerarquía de bloques y texturas atlas. Sin embargo, muchos de los cálculos asociados para este despliegue son altamente costosos, en donde cabe destacar, el manejo eficiente de la textura atlas. Uno de los problemas principales de la textura atlas es la dificultad de almacenar bloques de diferentes resoluciones y tamaños de manera compacta dentro de la misma, ya que al ir cambiando de resolución distintas áreas del volumen, se puede generar fragmentación. En este trabajo se realizó una comparación de un *ray casting* básico de volúmenes empleando *fragment shader*, *compute shader*, OpenCL y CUDA, para determinar que tecnología ofrece un mejor desempeño para este tipo de algoritmos. Además, se implementó un algoritmo de visualización de volúmenes multi-resolución utilizando jerarquía por bloques y la textura atlas. Esta implementación propone la optimización de diversas etapas necesarias para esta clase de algoritmos, utilizando cálculo paralelo con la GPGPU. Se propusieron optimizaciones para el proceso de carga y submuestreo del volumen, y para el cálculo de distorsión y otras métricas asociadas. También se elaboró un algoritmo de actualización de la textura atlas, evitando la fragmentación, y utilizando el direccionamiento obtenido por el uso del *Morton order*.

Palabras Claves: despliegue de volúmenes, volumenes multi-resolución, *compute shader*, OpenCL, CUDA, textura atlas, *Morton order*

Tabla de Contenidos

Índice de Figuras

Índice de Tablas

Introducción

La visualización de información volumétrica tiene una importancia significativa en la visualización científica, y es una herramienta de suma importancia en campos como la medicina, física, biología e ingeniería. Diversas técnicas se han desarrollado en el área de la computación gráfica que permiten la visualización de volúmenes, incluso en tiempo real. Sin embargo, los crecientes avances tecnológicos han permitido la adquisición de volúmenes de cada vez mayor tamaño y resolución, haciendo a muchas de estas técnicas insuficientes para el despliegue.

Actualmente los sistemas de computadores convencionales no tienen la capacidad de procesar volúmenes de gran tamaño en su representación más fina de forma eficiente. Para esto se han propuesto varios trabajos donde muestran soluciones al despliegue de volúmenes. Algunos de los trabajos se basan en ir desplegando pequeños subvolúmenes hasta lograr visualizarlo completamente [1]; otros en desplegar solamente un subvolumen que pueda ser almacenado en la memoria gráfica [2]; y entre los más destacados se encuentra 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 usando técnicas como jerarquías con *bricks* [3], jerarquía por bloques [4], jerarquías utilizando *wavelets* [5], entre otros. A pesar de hacer el despliegue del volumen, muchos de estos trabajos presentan problemas como la cantidad de artefactos en la imagen final, poca eficiencia y la gran cantidad de memoria utilizada.

Dentro del área del visualización de volúmenes multi-resolución, el trabajo realizado por López et al. [4] es uno de los más recientes. Los autores utilizan una técnica de despliegue de volúmenes multi-resolución usando *ray casting* de una pasada. Este algoritmo utiliza una jerarquía por bloque para almacenar el volumen y una textura denominada atlas para hacer el despliegue en el *fragment shader*. Además, se realiza el cálculo de la distorsión usando CUDA, teniendo un tiempo de respuesta menor a otros trabajos. Sin embargo, el trabajo presenta un proceso de desfragmentación ineficiente, artefactos visuales en las fronteras entre diferentes niveles de detalles y pocas optimizaciones en el cálculo de la distorsión debido a que solo se puede procesar un bloque a la vez en el GPU. Posteriormente, Fernández et al. [6] mejoraron el trabajo de López et al. [4], permitiendo una inserción eficiente dentro de la textura atlas al cambiar bloques de niveles de detalle, evitando la fragmentación de la memoria y aprovechando el máximo espacio posible. Sin embargo, se necesita de un esquema de direccionamiento que llaman *3D Z-order Strip*, que utiliza un esquema iterativo

ineficiente para calcular direcciones, cálculo que se realiza constantemente. Además, ignoran el valor de distorsión para el cálculo de las prioridades de los bloques, y no resuelven los artefactos visuales obtenidos en las fronteras de los bloques. En esta investigación se propone un algoritmo de despliegue de volúmenes multi-resolución basados en las propuestas de López et al. [4] y Fernández et al. [6], que se enfoque en mejorar las deficiencias encontradas en estos trabajos.

Objetivo General

Desarrollar un prototipo de sistema de despliegue de volúmenes multi-resolución, que utilice la técnica de *ray casting* de una pasada, en donde se quiere optimizar el cálculo de la distorsión y la actualización de la textura atlas, usando algún lenguaje de programación paralela.

Objetivos Específicos

- Comparar el desempeño del algoritmo de *ray casting* implementado en *fragment shader*, *compute shader*, OpenCL y CUDA.
- Aplicar una jerarquía multi-resolución con bloques, basado en el trabajo de Fernández et al. [6].
- Implementar un algoritmo de reducción de artefactos entre fronteras de bloques sin generar replicación de datos entre las fronteras, usando la interpolación entre bloque propuesta por Ljung [1].
- Optimizar el algoritmo de cálculo de la distorsión de los bloques en el GPU realizado por López et al. [4], para que pueda realizarse el cálculo de manera paralela en más de un bloque a la vez.
- Optimizar el algoritmo de actualización de la textura atlas presentado por Fernández et al. [6], utilizando de manera eficiente el *Morton order*.
- Realizar pruebas de rendimiento del algoritmo, incluyendo el cálculo de distorsión y actualización de la textura atlas.

Alcance del Trabajo de Grado

En esta investigación se realizará la implementación de un sistema prototipo para la visualización de volúmenes multi-resolución utilizando jerarquía de bloques. El sistema tomará como base el trabajo propuesto por Fernández et al. [6] y se modificará para solventar los problemas

de actualización de la textura atlas, mejorar la eficiencia del cálculo de la distorsión y eliminar artefactos visuales en las fronteras entre diferentes niveles de detalles.

El volumen a desplegar será cargado junto con un conjunto de valores correspondientes a los parámetros de visualización. Además, el sistema permitirá al usuario modificar estos parámetros a través de la interfaz gráfica. Una vez que el volumen sea cargado, se pasará a una etapa de preprocesamiento, donde se creará una jerarquía de bloques como la propuesta en [4]. Los bloques a ser usados para el despliegue serán almacenados en la textura altas, evitando la fragmentación basándose en el trabajo de Fernández et al. [6]. Además, no se utilizará un sistema de paginación con almacenamiento secundario, por lo que solo se permitirá la carga de volúmenes que quepan en la memoria principal junto con toda su jerarquía de bloques.

Para la elección del conjunto de bloques a ser desplegados, se utilizará un esquema similar al propuesto por López et al. [4], donde se hará uso del GPU para agilizar los cálculos. Con este criterio, se creará una cola de prioridad como la propuesta en [7], en donde los nodos con mayor prioridad serán aumentados de nivel de detalle, permitiendo mejoras progresivas en la resolución del volumen visualizado. También se reducirán artefactos entre fronteras implementando el algoritmo de interpolación entre bloques propuesto por Ljung et al. [1].

Finalmente, la visualización se efectuará usando la textura de índices en conjunto con la textura atlas, el cual es el método más utilizado actualmente [4] [6] [8] [9]. Este algoritmo de visualización se implementará en alguno de los APIs de programación paralela que serán estudiados en este trabajo: OpenCL, CUDA, *compute shader* o *fragment shader*. Se hará una comparación de estos APIs para seleccionar el que mejor se ajusta a la visualización de volúmenes usando *ray casting*.

Para el desarrollo de la solución se utilizará el paradigma de Programación Orientada a Objetos [10], y se usará Git [11] como repositorio y controlador de versiones. El lenguaje de programación a utilizar es C++ y se seguirá el estilo de programación sugerido por la guía de estilos propuesta por Google [12]. Adicionalmente, se usará el siguiente sistema para desarrollo y pruebas:

- Computadora con las siguientes características: Intel(R) Core(TM) i7-3770 CPU de 3.4 GHz, 12 GB de memoria principal, Windows 7 de 64 bits, y una Nvidia GeForce GTX 660 de 2 GB de memoria de video con una microarquitectura Kepler.
- Microsoft Visual Studio 2013.
- OpenGL 4.5, GLSL 4.5, CUDA con capacidad de cómputo 3.0 y OpenCL 1.2.
- Interfaz gráfica utilizando dear imgui (AKA ImGui) [13].
- Lenguaje de programación C++.

Se harán diversas pruebas sobre las implementaciones realizadas, donde se medirá y comparará la eficiencia y la calidad visual de las mismas. Los datasets a tomar en cuenta para la realización de las pruebas serán seleccionados para ofrecer una variedad de tamaño y tipo de volumen. Para las pruebas de comparación de desempeño de la visualización de volúmenes utilizando *ray*

casting, los datasets a seleccionar tendrán un tamaño cercano a los 600 MB, siendo volúmenes de un tamaño considerable, pero que no necesitan el uso de un algoritmo multi-resolución para desplegarlos. En el caso de la implementación del algoritmo multi-resolución, se utilizarán datasets de mayor tamaño, pero debido a que no se implementará un algoritmo de paginación del volumen en disco, se considerarán volúmenes de prueba de hasta 4 GB. De este modo, se asegura que el volumen y toda su representación multi-resolución puedan ser almacenados sin problemas en memoria principal.

Aportes de la Investigación

Como publicaciones realizadas en el contexto de este trabajo se tiene las siguientes:

- Presentación de una conferencia en la XLII Conferencia Latinoamericana en Informática 2016 (CLEI '16) [14]. La publicación presenta resultados preliminares obtenidos de la comparación de las implementaciones de *ray casting* con los diferentes APIs paralelos.
- El trabajo presentado en el CLEI 2016 fue posteriormente desarrollado como un artículo en extenso para el CELI Journal 2017 [15]. Allí se presentaron resultados más amplios que los obtenidos en [14], con pruebas, discusiones de los resultados y conclusiones más profundas. Los resultados expuestos en ese artículo se ven plasmados en este documento en la Sección 3.1.

Se tiene planificada la publicación de un tercer trabajo que contemple los resultados obtenidos de la implementación del visualizador de volúmenes multi-resolución. Esta publicación se centrará en divulgar los algoritmos propuestos para la aceleración del cálculo de la distorsión y del manejo eficiente de la textura atlas. Especialmente este último punto puede representar un avance significativo en esta área, ya que no se encuentra publicada información acerca de este manejo.

Organización del Documento

Este documento es presentado en 4 capítulos. El Capítulo 1 introduce los conceptos básicos sobre el tema a tratar y los estudios previos que sustentan el problema a resolver. El Capítulo 2 describe la solución propuesta y sus detalles de implementación. El Capítulo 3 muestra las pruebas y resultados obtenidos. Finalmente, el Capítulo 4 presenta las conclusiones y trabajos futuros.

Capítulo 1

Marco Teórico

La visualización de volúmenes es un área de la computación gráfica en la cual se han realizado diversos estudios. Especialmente en los últimos años, estos estudios se han enfocado en el despliegue de volúmenes de gran tamaño, buscando la mejor calidad visual junto con el mejor rendimiento. Por ello, en este capítulo se expondrán algunos de los trabajos más recientes, mostrando los soportes más destacados para la realización de la implementación de este trabajo. Primero, se condensará los conceptos básicos sobre la visualización de volúmenes, haciendo énfasis en el uso del algoritmo de *ray casting*, y resumiendo los estudios realizados acerca de las diferentes implementaciones paralelas del algoritmo. Después se expondrán las principales formas de representar volúmenes de gran tamaño para poder desplegarlos. Luego se explicará el uso de la textura atlas para el almacenamiento de las diferentes partes del volumen a desplegar en su correspondiente resolución, y se explicarán los trabajos actuales para la selección de la resolución utilizada. Posteriormente, se describirán algunas de las técnicas más utilizadas para la reducción de artefactos entre fronteras. Finalmente, se ejemplifica como es el proceso básico de despliegue de un volumen de gran tamaño utilizando una textura atlas.

1.1. Visualización de Volúmenes

La visualización directa de volúmenes consiste en el despliegue de un campo escalar tridimensional, proyectando sus muestras sin la necesidad de una reconstrucción intermedia de la data [16]. La idea es simular el paso de la luz a través de un medio participante, el cual es el volumen. Esta interacción puede ser modelada con la Ecuación 1.1.

$$C = \int_0^D c(s(x(\lambda)))\tau(s(x(\lambda)))e^{\int_0^\lambda \tau(s(x(\lambda')))d\lambda'} d\lambda \tag{1.1}$$

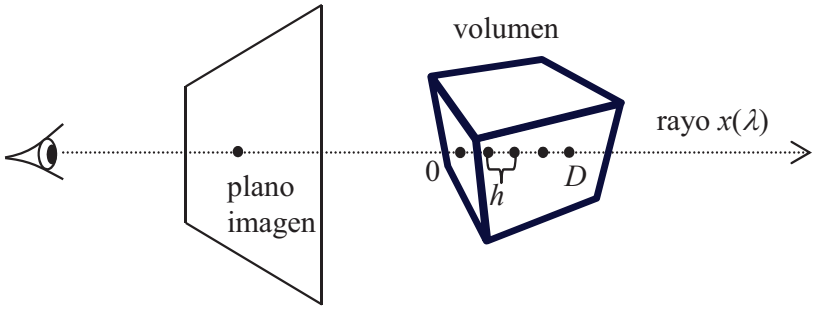


Figura 1.1: Cálculo del color del píxel de una imagen acumulando las contribuciones de color y opacidad de muestras equidistantes en el volumen.

En este modelo, el vector de visualización que atraviesa al volumen es parametrizado con λ en $[0, D]$, como puede observarse en la Figura 1.1. En la ecuación, la función $x()$ obtiene una coordenada 3D (x, y, z) evaluando la ecuación del rayo, y la función $s()$ obtiene una muestra del campo escalar representado por el volumen. Finalmente, en la ecuación, $c(s(x(\lambda)))$ corresponde a la emisión, y $\tau(s(x(\lambda)))$ a la absorción del volumen a una distancia λ del rayo [17]. El factor $exp\left(\int_0^\lambda \tau(s(x(\lambda')))d\lambda'\right)$ correspondiente a la extinción de la luz, se puede interpretar como la transparencia $T(s(x(\lambda)))$ del volumen a una profundidad o distancia λ. Basada en la transparencia, se puede calcular la opacidad α acumulada en la travesía del rayo a cualquier distancia λ, como se expresa en la Ecuación 1.2.

$$\begin{aligned} \alpha(\lambda) &= 1 - T(s(x(\lambda))) \\ \Rightarrow \alpha(\lambda) &= 1 - e^{\int_0^\lambda \tau(s(x(\lambda')))d\lambda'} \end{aligned} \tag{1.2}$$

Generalmente, el rayo es discretizado en distancias equidistantes a lo largo del volumen, lo que permite que, utilizando una aproximación conocida como la Aproximación de la Composición Volumétrica [18], la fórmula pueda escribirse de manera discreta, como es visto en la Ecuación 1.3. Aquí, la emisión y la opacidad se asumen constantes dentro de cada intervalo, pudiendo renombrar $s(x(\lambda))$ como s_i, para un segmento determinado del rayo. De esta manera, $c_i = c(s_i)$ es el color y $\alpha_i = 1 - e^{-h\tau(s_i)}$ es la opacidad de ese segmento de rayo de tamaño h. Detalles de esta simplificación pueden encontrarse en [7].

$$C \approx \sum_{i=0}^{n-1} \prod_{j=0}^{i-1} (1 - \alpha_j)\alpha_i c_i \tag{1.3}$$

Las propiedades ópticas del volumen generalmente son asignadas por medio de la función de transferencia [16], que es una función que para cada valor escalar del volumen hace una correspondencia con un color y una opacidad. Estos valores pueden ser escogidos arbitrariamente o por medio de algún proceso automático, como la segmentación. Debido a que el volumen

realmente es un campo escalar discreto, al tomar muestras a lo largo del recorrido del rayo, es posible que las muestras a tomar no correspondan con las muestras originales del volumen. Por ello, es necesario un proceso de interpolación tri-lineal para obtener el correspondiente valor a componer, para lo cual la muestra debe ser clasificada. Hay dos tipos básicos de clasificación: post-clasificación y pre-clasificación [7]. En la post-clasificación, los valores escalares primero son interpolados y posteriormente se les asignan sus valores ópticos a este valor interpolado. En la pre-clasificación, a los valores escalares primero se les asignan sus valores ópticos y posteriormente estos valores ya clasificados son interpolados.

Diversas técnicas se han desarrollado para evaluar computacionalmente la ecuación de despliegue de volúmenes, entre la que destaca el *ray casting* de volúmenes. El *ray casting* [19] es una técnica que evalúa directamente la ecuación de visualización volumétrica, lanzando rayos desde el ojo por cada píxel de la imagen a generar, como puede observarse en la Figura 1.1. Para cada rayo, se debe encontrar el punto de entrada y de salida con respecto al volumen, e ir tomando muestras a lo largo del recorrido del rayo a través del volumen, componiendo las muestras utilizando la Ecuación 1.3. El algoritmo básico puede observarse en el Código 1.1.

```
void rayCasting(framebuffer)
{
  //iterar sobre toda la imagen
  Para cada píxel (x,y)
    Definir el rayo r(t) que parte del ojo hacia el píxel (x,y)

    //verificar intersección del rayo con el volumen
    Si( el rayo intersecta al volumen)
      Obtener puntos de entrada y salida (a y b) del rayo en el volumen

      A = 1 //opacidad
      C = 0 //color
      D = length(b-a)  //longitud del rayo

      Redefinir el rayo como r(t) = a + (t/D)/(b-a)

      //realizar el recorrido del rayo
      Para (t = 0; t <= D; t = t+h)
        Muestrear el volumen en la posición r(t), obteniendo s = s(r(t))
        Clasificar la muestra s, obteniendo c(s) y a(s)

        //evaluar la función de composición volumétrica
        C = C + c(s) * a(s) * A
        A = A * (1 - a(s))


      framebuffer[x][y] = C
}
```

Código 1.1: *Ray casting* básico.

Debido a que el *ray casting* de volúmenes evalúa la ecuación de despliegue directamente, normalmente obtiene mejores resultados visuales que otras técnicas. Adicionalmente, esta técnica permite saltar zonas del volumen que se consideren vacías [18] (zonas transparentes del volumen), y hacer una terminación temprana del recorrido si la opacidad acumulada hace que la contribución de las siguientes muestras sean despreciables [20] (por ejemplo, con

una opacidad acumulada menor a $0{,}01$). Además, dado a que en esta técnica el cálculo es independiente para cada rayo, el proceso puede ser fácilmente paralelizado.

Para aprovechar el paralelismo del *ray casting*, generalmente se implementa en lenguajes de programación paralela como CUDA [21], OpenCL [22] o utilizando el *fragment shader* de OpenGL [23]. Además, existe el *compute shader* de OpenGL [23] en el cual no se encuentra una implementación publicada. En el Apéndice A se puede consultar teoría básica acerca de la programación paralela que puede ser de interés.

Aunque todos estos APIs pueden realizar cosas similares, hay diferencias que hacen que un API sea mejor a otro en ciertas circunstancias. Por ello, hay trabajos que se enfocan en estudiar el desempeño de estas tecnologías [24] [25] [26], y hacer comparaciones entre estos APIs paralelos [25] [27] [28] [29] [30] [31] [32] [33].

Nuestro interés principal es el desarrollo paralelo del despliegue de volúmenes utilizando *ray casting* y algunos autores han investigado al respecto. Schubert et al. [34] implementaron despliegue de múltiples volúmenes simultáneamente utilizando CUDA, obteniendo resultados en tiempo real. Además, diversos autores han comparado el desempeño entre implementaciones con: CUDA y OpenCL [35], y CUDA y *fragment shader* [36] [37] [38]. En estos estudios, CUDA siempre tiene un mejor desempeño.

Sin embargo, no hay ningún estudio hasta los momentos que utilice el *compute shader*. Esta puede ser una buena opción ya que posee características similares a CUDA y OpenCL, pero además está integrado dentro de OpenGL, el cual es necesario para el posterior despliegue del resultado del algoritmo de *ray casting*. Es de interés realizar una comparación de desempeño de una implementación de *ray casting* utilizando CUDA, OpenCL, *compute shader* y *fragment shader*.

1.2. Representación para el Despliegue de Volúmenes de Gran Tamaño

Según Bethel et al. [39], un volumen se considera de gran tamaño si son demasiado grandes para ser procesados: (1) en su totalidad, (2) todo el volumen a la vez, y/o (3) excede la memoria disponible. En este trabajo nos enfocaremos en los volúmenes que exceden la memoria disponible, de modo que no pueden ser desplegados en su totalidad, por lo que se deben buscar maneras de representarlo tal que puedan ser desplegadas partes significativas del volumen.

Una de las maneras de desplegar volúmenes de gran tamaño es mediante el despliegue de un área de interés o *volume roaming* [2]. Esta técnica se basa en el despliegue de una zona específica del volumen, substrayendo un subvolumen cuyo tamaño no sea mayor al de la memoria. El área de interés se especifica generalmente a través de un cubo, un punto o corte

del volumen. Sin embargo, los cambios del área de interés por interacción del usuario podrían crear un cuello de botella a la hora de actualizar el volumen a desplegar.

Por otro lado, el volumen también puede descomponerse en el espacio de objeto en sub-volúmenes denominados *bricks* (ladrillos) [40], como puede observarse en la Figura 1.2b, los cuales puede ser desplegados individualmente. Los *bricks* deben ser ordenados del más cercano al más lejano, de manera que pueda evaluarse la ecuación de composición volumétrica correctamente. Además, dado que los *bricks* se despliegan individualmente, se requieren de múltiples pasadas.

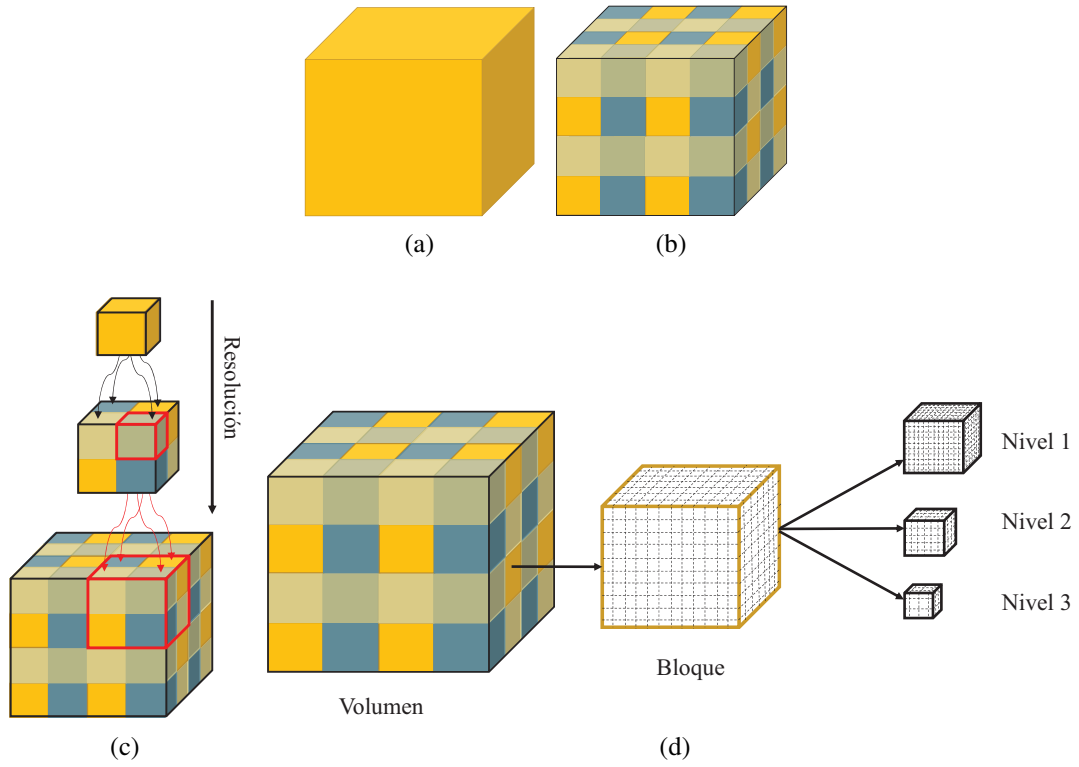


Figura 1.2: Diferentes maneras de representar el volumen en memoria. En (a), tenemos el volumen original. En (b), el volumen es particionado en subvolúmenes denominados *bricks*. En (c), el volumen es organizado en una jerarquía *octree* de *bricks*, en donde la raíz es un solo *brick* que contiene todo el volumen, y las hojas representan el volumen en su mayor nivel de detalle . Finalmente, en (d) se observa el volumen representando con una jerarquía de bloques. En un primer nivel tenemos la subdivisión en bloques del volumen, y seguidamente se muestran los vóxeles de un bloque y tres niveles de detalles locales al bloque.

En los últimos años ha cobrado mayor fuerza el uso de jerarquías multi-resolución como lo son el *octree* [1] y la jerarquía por bloques. En el caso del *octree* [7] [9] [8], cada nodo del árbol es un *brick* del volumen, donde la raíz representa el nivel de detalle más burdo y las

[1] Un *octree* es una estructura de datos de árboles en que cada nodo interno posee a lo sumo 8 hijos.

hojas el más fino (ver Figura 1.2c). Los nodos internos se obtienen mediante la reducción del nivel de detalle de sus hijos. A cada área del volumen se le asigna un nivel de detalle acorde a un prioridad que puede depender de muchas variables. En este tipo de jerarquías todos los *bricks* poseen el mismo tamaño en vóxeles, y por ejemplo, la raíz de la jerarquía contiene información correspondiente a todo el volumen. Actualmente, el recorrido del *octree* se puede realizar en una sola pasada de despliegue en el GPU [8][9][41]. En estos casos, los *bricks* a desplegar son almacenados en una textura en la memoria de video. Mientras el rayo va recorriendo un volumen virtual, se calcula la resolución en que debería desplegarse el área del volumen a procesar, y por medio de tablas de direccionamiento, se ubica el *brick* correspondiente a desplegar.

En el caso de la jerarquía por bloques, el volumen es particionado en bloques [42] [1] [4] [6]. Cada uno de los bloques es posteriormente submuestreado iterativamente, generando todos los niveles de detalle (ver Figura 1.2d). Esta jerarquía permite que cada bloque tenga un nivel de detalle asignado independientemente, pudiendo realizar cambios de nivel de detalle locales a cada bloque. Además, un bloque particular, y todos sus niveles de detalle, representan solamente una zona específica del volumen, donde para cada nivel de detalle inferior, la cantidad de vóxeles que contiene un bloque disminuye. Esta es una de las técnicas más utilizadas actualmente.

Otra opción de jerarquía es la propuesta por Gao et al. [5], en la que se construye una jerarquía de *wavelets*. A cada nodo de la jerarquía se le aplica una transformada *wavelet* 3D, con la que se produce un *brick* más pequeño con un filtro paso bajo, y unos coeficientes de paso alto. Con este *brick* más pequeño se puede realizar el mismo proceso y construir una jerarquía, donde por cada nodo solo se necesitarían guardar las componentes de alta frecuencia. Al momento de acceder a un nodo, se puede reconstruir su información utilizando la inversa de la transformada de *wavelet* 3D, juntando las componentes de alta frecuencia con el nodo padre, el cual contiene las bajas frecuencias. Como es posible que el padre también deba ser reconstruido, este proceso es recursivo. Este tipo de jerarquía representa una buena opción para el almacenamiento, pero para acceder un determinado nivel de detalle, varios niveles de la jerarquía deben ser accedidos, lo cual puede ser costoso.

Uno de los problemas al crear jerarquías de niveles de detalle, es la aplicación del submuestreo, lo cual genera pérdidas de altas frecuencias y una incorrecta aplicación de la función de transferencia. Para solventar estos problemas, Younesy et al. [43] proponen aproximar la distribución de los datos utilizando la media y la varianza de cada nivel de detalle. Posteriormente, Sicat et al. [44] introducen los volúmenes *pdf* esparcidos, en el cual se utiliza la función de densidad de probabilidad (*probability density function*, *pdf*) para poder aplicar consistentemente la función de transferencia independientemente del nivel de detalle.

Indiferentemente de la jerarquía a utilizar, se deben seleccionar el conjunto de *bricks* o bloques que van a ser desplegados. Este conjunto a desplegar es conocido como el *working set* (grupo de trabajo), el cual debe ser almacenado de alguna manera en memoria. El método más utilizado para almacenar el *working set* es el uso de la textura atlas, la cual será revisada con mayor detalle en la siguiente sección.

1.3. Textura Atlas

Cuando una textura no cabe en su totalidad en memoria de GPU, es posible fragmentarla y solo almacenar las partes que serán necesarias en un momento dado en una textura llamada atlas (también conocida como textura virtual o textura esparcida). Incluso se pueden almacenar diferentes secciones de una textura en diferentes niveles de detalle, como podemos observar en la Figura 1.3. Generalmente viene acompañada de una textura de índices, la cual funciona como un sistema de direccionamiento que permite muestrear en la posición correcta de la textura.

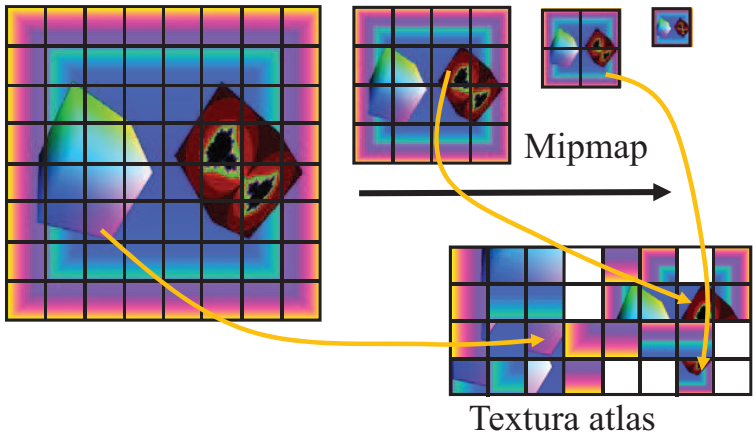


Figura 1.3: Textura atlas utilizada para almacenar diferentes fragmentos de una textura con diferentes niveles de detalle.

Esta misma idea puede ser utilizada para almacenar en una textura atlas tridimensional el *working set* de un volumen a desplegar. En el resto de este trabajo se tratarán texturas atlas tridimensionales para almacenar volúmenes, pero por simplicidad visual, serán mostradas en las figuras como texturas altas bidimensionales.

Lux et al. [45] proponen el uso de una textura atlas para almacenar los *bricks* de un *octree* a desplegar con sus diferentes resoluciones (ver Figura 1.4). Cada nodo del *octree* almacena la posición del *brick* que se encuentra contenido en la textura atlas. Como todos los *bricks* son del mismo tamaño, su ubicación en la textura altas puede realizarse de manera compacta sin que haya fragmentación dentro de la misma.

Posteriormente López et al. [4] adaptan esta idea para utilizarla con volúmenes multi-resolución por bloques. Igual que en el caso anterior, se posee una textura atlas que almacena todos los bloques a utilizar. Además, se utiliza una textura de índices la cual posee una tupla RGBA (px, py, pz, lod) con la posición (px, py, pz) del bloque dentro de la textura atlas y su correspondiente nivel de detalle (lod). Dado que el tamaño en vóxeles de los bloques varía según el nivel de detalle, el almacenamiento dentro de la textura atlas puede generar fragmentación,

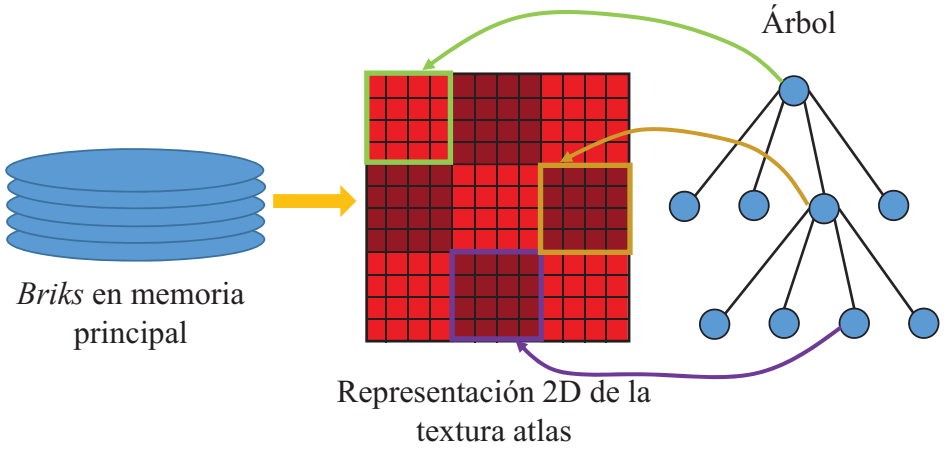


Figura 1.4: Enfoque de textura atlas con *bricks*, utilizado por Lux et al. [45].

lo que debe ser tratado con cuidado. López et al. [4] insertan de manera desordenada los bloques dentro de la textura atlas, ubicando cada bloque en el primer espacio libre disponible, generando fragmentación. Al momento de no poder insertar bloques nuevos, realizan un proceso de desfragmentación de todos los bloques, donde insertan de manera ordenada cada uno los bloques del *working set*. El proceso de desfragmentación es costoso, ya que requiere volver a ordenar toda la información.

Para evitar el costo del proceso de desfragmentación de la memoria, Fernández et al. [6] introdujeron un algoritmo eficiente para el manejo de la fragmentación de la textura atlas, que lo denominaron *3D Z-order Strip*. En primera instancia, utilizan una lista de bloques por nivel de detalle, estructura que puede verse en la Figura 1.5.

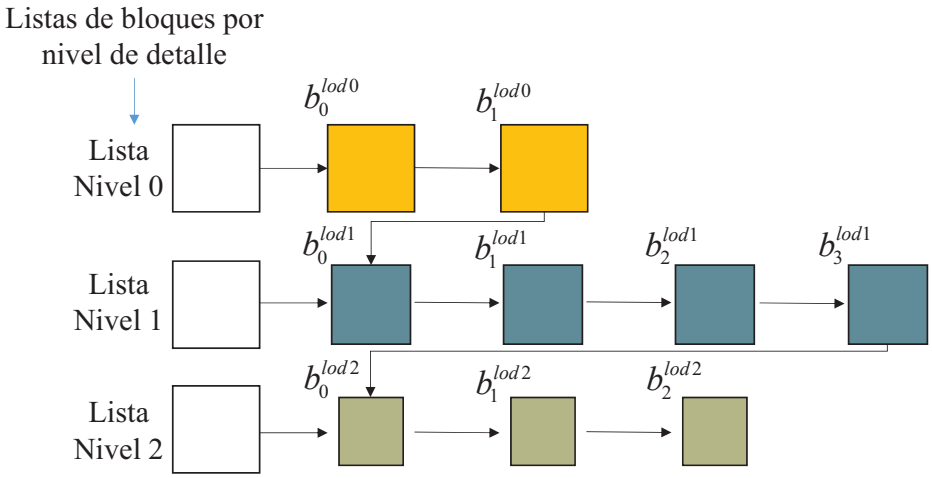


Figura 1.5: Descripción gráfica de las listas de bloques para su ordenamiento. La notación b_i^{lodx} representa el índice de cada bloque.

Utilizando esta lista, los bloques pueden ser indexados de manera lineal, tomando en cuenta su resolución y la cantidad de bloques anteriores en la lista. Luego, los bloques son insertados de manera ordenada en una estructura llamada *Strip*, en la que se utiliza un ordenamiento que es una aproximación al *Morton order* [46] (el *Morton order* será explicado en mayor detalle en la Sección 2.2.1). De esta manera, se realiza una correspondencia entre los índices lineales de la lista de bloques al *Strip*, lo cual permite almacenar los bloques ordenados por nivel de detalle, en forma consecutiva, asemejando una curva en forma de *Z*, como puede observarse en la Figura 1.6. En la figura, los bloques de mayor resolución se asumen de tamaño 32×32, y cada nivel de resolución menor es $1/4$ más pequeño que el anterior. Los índices presentes en la imagen son construidos a partir de la lista de bloques, y dependen del tamaño de los bloques anteriores. Con el algoritmo propuesto por Fernández et al. [6], a partir de estos índices lineales se obtiene una coordenada 2D dentro del *Strip*.

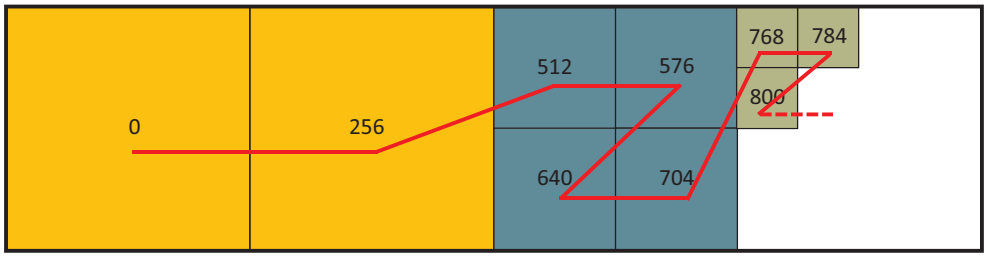


Figura 1.6: Estructura *Strip* con bloques ordenados por nivel de detalle. Bloques de mayor resolución se asumen de tamaño 32×32.

Con este orden, los bloques son almacenados eficientemente dentro de la textura atlas. Además, Fernández et al. [6] proponen movimientos eficientes de bloques dentro del *Strip*, lo que permite aumentar o disminuir el nivel de detalle de los bloques sin generar fragmentación. Los *bricks* o bloques almacenados dentro de la textura atlas deben ser escogidos utilizando un criterio de selección. En la siguiente sección serán expuestos algunos criterios de selección presentes en la literatura.

1.4. Criterio de Selección

El criterio de selección es la etapa en la que se determina el *working set*. Algunas de estas técnicas consideran parámetros basados en los datos, como la distorsión de representar un área del volumen con determinado nivel de detalle [47] [48] [49] [50] [51], homogeneidad del *brick* o bloque [48], entre otros. Otras técnicas consideran parámetros de visualización como la posición del ojo [7] [52], un punto o una región de interés dentro del volumen [3] [48] [42], descartes por el *frustum* [53] (pirámide truncada de visualización), guiados por los rayos de visualización [41] [9] [8], entre otros. Además, se pueden combinar estas técnicas de tal manera de poder tener un mejor criterio de selección.

Entre los autores que calculan distorsión, tenemos a Boada et al. [48], quienes obtienen el error medio de aproximación de cada nodo de un *octree* utilizando la siguiente fórmula:

$$e(n_j) = \frac{\sqrt{\frac{1}{2^{3(l_{max}-k)}} \sum_{i=1}^{2^{3(l_{max}-k)}} (s(v_i)) - f_{n_j}(v_i))^2}}{s_{max}}$$

donde f_{n_j} es la interpolación trilineal de los 8 valores escalares de las esquinas del nodo n_j, $s_i(v)$ son las muestras del volumen original asociados al área del nodo n_j, 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. Dado que solo se utilizan los vóxeles de las esquinas del *brick*, el error puede llegar a ser alto, ya que no toma en cuenta el aporte de los vóxeles interiores.

Utilizando también una jerarquía *octree*, Wang et al. [49] calculan una métrica de error, pero utilizando *wavelets*. El error viene dado por:

$$e(n_j) = \frac{\sum_{i=0}^{7} (\sum_{v \in B} (b_i(v) - f(v))^2)}{8n} + max_e$$

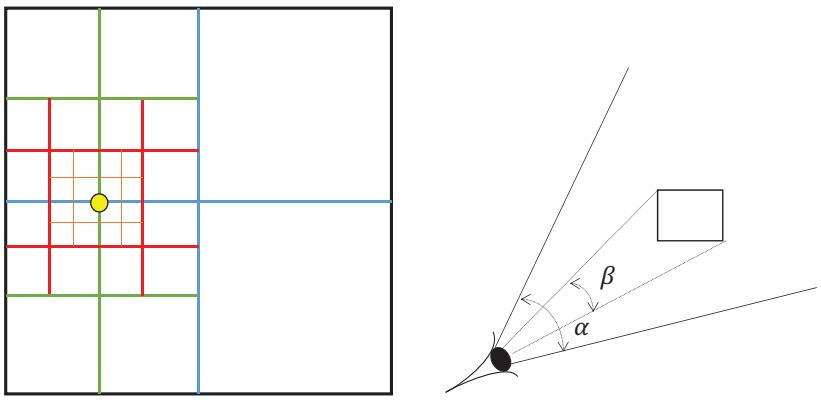
donde v representa a un vóxel del volumen, B representa un *brick*, $b_i(v)$ son las muestras de los nodos hijos obtenidos mediante el uso de transformadas *wavelets*, $f(v)$ es la muestra interpolada de los vóxeles originales del volumen, n es la cantidad de vóxeles por nodo y max_e es el máximo error entre los 8 hijos.

Posteriormente, Ljung et al. [50] proponen el cálculo del error utilizando el espacio de color CIELuv [54] (*Commission Internationale de l'éclairage, l'espace clorométrique L*u*v**) también utilizando *wavelets*. Dado que el error cuadrático debe ser calculado cada vez que se cambia la función de transferencia, los autores introducen un medio de aproximación del error, mediante un histograma de frecuencias por cada nodo, reducido a solo 10 segmentos. Así, para cada vóxel, se multiplica su distorsión por su frecuencia en la correspondiente entrada del histograma, de manera de reducir cálculo redundante. Una aproximación más precisa es considerada por Wang et al. [51], donde se utiliza un histograma de 256 entradas, con diversas tablas para acelerar los cálculos.

Estas métricas de error sobre los datos permiten determinar cual es el error de representar un *brick* o bloque con un determinado nivel de detalle. Sin embargo, estas técnicas suelen ser costosas ya que recorren todos los datos y pueden necesitar actualizar los valores de distorsión al modificarse la función de transferencia. Adicionalmente, podría haber *bricks* o bloques que no contribuyan a la imagen final y realizar errores de métricas sobre ellos sería innecesario. Por ello, otros criterios de selección han sido desarrollados para poder descartar *bricks* o bloques dependiendo de la contribución de los mismos en la imagen final. Igualmente podrían ser combinadas con una métrica de error para un mejor criterio de selección.

Por ejemplo, Chamberlein et al. [53] proponen descartar nodos de una jerarquía *octree* utilizando el *frustum*. Los nodos son proyectados a la pantalla de despliegue, y aquellos que estén completamente fuera del *frustum* son descartados. Si los nodos están parcial o totalmente dentro del *frustum*, se utiliza su caja envolvente proyectada para determinar que proporción de la pantalla ocupa. Si es menor a cierto umbral, se despliega una versión simplificada del nodo con un color y transparencia determinado. De lo contrario, se subdivide el nodo del *octree* y se aplica el mismo criterio recursivamente.

LaMar et al. [3] expanden esta idea, añadiendo otro criterio en el cual toman en consideración la distancia al punto de interés y la relación entre el ángulo de visión y el ángulo que genera la diagonal proyectada del *brick*. La jerarquía *octree* es recorrida en preorden, y si un nodo está fuera del *frustum* se descarta. En casa contrario, se selecciona si se cumple que: (1) la distancia del centro del *brick* al punto de interés es mayor que la diagonal del *brick*, o bien, si el nodo no puede ser refinado, por ejemplo, cuando es una hoja, y (2) el ángulo proyectado del *brick* es menor que la mitad del ángulo asociado al campo de visión. Si no se satisfacen los criterios, se evalúa el criterio recursivamente con cada uno de los hijos (ver Figura 1.7).



(a) (b)

Figura 1.7: Criterio de selección basándose en la distancia al punto de interés y el ángulo de proyección del *brick*. En la imagen (a) se puede ver una representación de lo que sería un *octree* en 2D. En este ejemplo, se seleccionaron solo 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. [48], y Li et al. [42] toman en cuenta la importancia que define el usuario mediante un área de interés, la homogeneidad del bloque y la capacidad de la memoria de textura. Se busca minimizar la cantidad de bloques homogéneos, para obtener menor cantidad de geometría a desplegar. Los bloques no homogéneos se representan con distintos niveles de detalle.

Similarmente, Guthe et al. [52] limitan la cantidad de bloques a desplegar a la capacidad de memoria de textura. Toman en cuenta el error cuadrático medio normalizado $E(b)$ entre el

valor de las muestras y las correspondientes muestras originales, y la distancia del vóxel más cercano del *brick* al ojo $z(b)$. De esta manera, se establece una prioridad para cada *brick* con la fórmula $P(b) = E(b)/z(b)$. Con este valor se construye una cola de prioridad, donde el nodo de mayor prioridad es reemplazado por sus hijos. De esta manera, se va refinando la selección iterativamente. El proceso es repetido hasta alcanzar la capacidad de la memoria de textura o no sea posible refinar más *bricks*.

Plate et al. [55] también consideran la capacidad de la memoria de textura, pero además toman en cuenta la cantidad de *bricks* que pueden ser cargados en cada *frame*. Usan como criterio la distancia al punto de vista y la limitación con la memoria de textura. Posteriormente, utilizan coherencia *frame* a *frame*, cargando una cantidad limitada de *bricks* por *frame*, y de esa manera no disminuir el rendimiento del sistema.

Carmona et al. [7] proponen un algoritmo similar tomando en cuenta la distancia al punto o área de interés, la distancia al visor, la distorsión multi-resolución de los vóxeles clasificados, y las limitaciones de hardware como ancho de banda y memoria de textura. Para la actualización del *working set*, utilizan un algoritmo incremental *frame* a *frame*, llamado *Split-and-Collapse*, en el cual se tienen dos colas de prioridad: una que indica la próxima operación de refinamiento de nivel de detalle (*Split*) y una que indica la próxima reducción de nivel de detalle (*Collapse*). Además, proponen un algoritmo óptimo que determina el *working set* con mínimo error. Con una idea similar Lopez et al. [4] y Fernández et al. [6], también proponen un esquema con coherencia *frame* a *frame* para la actualización de bloques dentro de una textura atlas.

Todas las técnicas descritas tienen la desventaja de que realizan el cálculo del criterio de selección sobre todos los datos, aunque hayan partes de esa data que no se vaya a desplegar, debido a que no es visible. Por ello, se han desarrollado técnicas actuales en las que el criterio de selección es guiado por los rayos de visualización. En este caso, los *bricks* o bloques, son añadidos en vez de descartados, ya que solo serán seleccionados los *bricks* o bloques que intersecten con los rayos de visualización.

Entre estas técnicas, tenemos el trabajo de Crassin et al. [41]. Hacen un despliegue con MRT (*Multi Render Target*) donde reportan los *bricks* que son visitados y no se tienen en el *working set*, durante el transcurso de un *frame*. Hadwiger et al. [9] presentan un enfoque parecido, pero utilizan la extensión de OpenGL *GL_shader_image_load_store*, la cual permite escribir sobre una textura no asociada al despliegue. Ampliando esta idea, Fogal et al. [8] utilizan esta extensión para escribir sobre una textura que utilizan como una tabla *hash*. Este proceso lo podemos observar en la Figura 1.8. Primeramente el rayo atraviesa el volumen y consulta en una tabla si el *brick* actual es: vacío, no es vacío y está presente, o no es vacío y está ausente. Si es vacío, simplemente se salta el *brick*. Si no es vacío y está presente se despliega. Y si no es vacío y está ausente, se registra en la tabla *hash*. Posteriormente, con esta tabla *hash* el CPU se encarga de buscar los *bricks* faltantes para agregarlos al *working set*. Finalmente, en *frames* posteriores se despliega el correspondiente *brick*. Cabe acotar, que en estos trabajos el nivel de detalle se selecciona dependiendo de la distancia del *brick* al punto de vista.

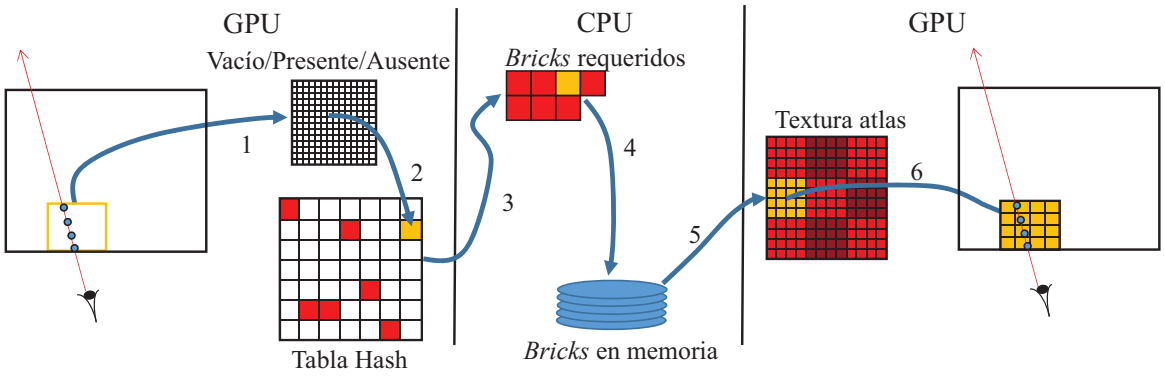


Figura 1.8: Reporte de bloques faltantes. Los bloques faltantes son registrados en una tabla *hash* (1, 2), la cual es utilizada para la carga de estos bloques (3, 4, 5), los cuales son desplegados en *frames* posteriores (6).

1.5. Reducción de Artefactos en Fronteras

En el momento de visualizar el volumen, se tiene que tener en consideración el despliegue de los vóxeles en la frontera entre *bricks* y bloques. Cuando los subvolúmenes adyacentes poseen la misma resolución, basta con compartir medio vóxel en las fronteras para realizar una correcta interpolación [3]. Sin embargo, si los subvolúmenes adyacentes poseen diferentes resoluciones, el manejo de las fronteras debe realizarse con cuidado, ya que se pueden producir errores o artefactos visuales debido a cambios bruscos del color y la opacidad en las fronteras.

Por ejemplo, Weiler et al. [56] reducen los artefactos entre niveles de detalles continuos en una jerarquía por bloque, en donde los bloques comparten los vóxeles de fronteras con sus vecinos utilizando ciertas consideraciones. Como puede observarse en la Figura 1.9a, los vóxeles en el borde izquierdo son copiados exactamente desde un nivel de detalle inferior al inmediatamente superior, mientras que el borde derecho lo calcula con la interpolación entre ambos bordes.

Seguidamente, Guthe et al. [52] utilizan una técnica para la interpolación de *bricks* en un *octree* como puede verse en la Figura 1.9b. En este caso, se ubican y se copian los valores de frontera de los *bricks* vecinos, solo tomando en cuenta tres de las seis caras del *brick*, reduciendo la cantidad de vóxeles duplicados a almacenar.

Ljung et al. [1] buscan realizar esta interpolación sin necesidad de compartir información entre bloques adyacentes. Toman en cuenta la interpolación entre bloques como puede ser visto en la Figura 1.9c. Para ello se deben ubicar todos los bloques adyacentes en sus diferentes resoluciones, tomar una muestra en cada uno de estos bloques, y proceder a interpolar estas muestras en el *fragment shader* manualmente.

Posteriormente, Beyer et al. [57] presentan una idea similar. Ellos proponen duplicación de vóxeles en las fronteras de los bloques, los cuales modifican dependiendo del nivel de detalle del vecino, tomando en cuenta solo dos niveles de detalle de diferencia. Los vóxeles

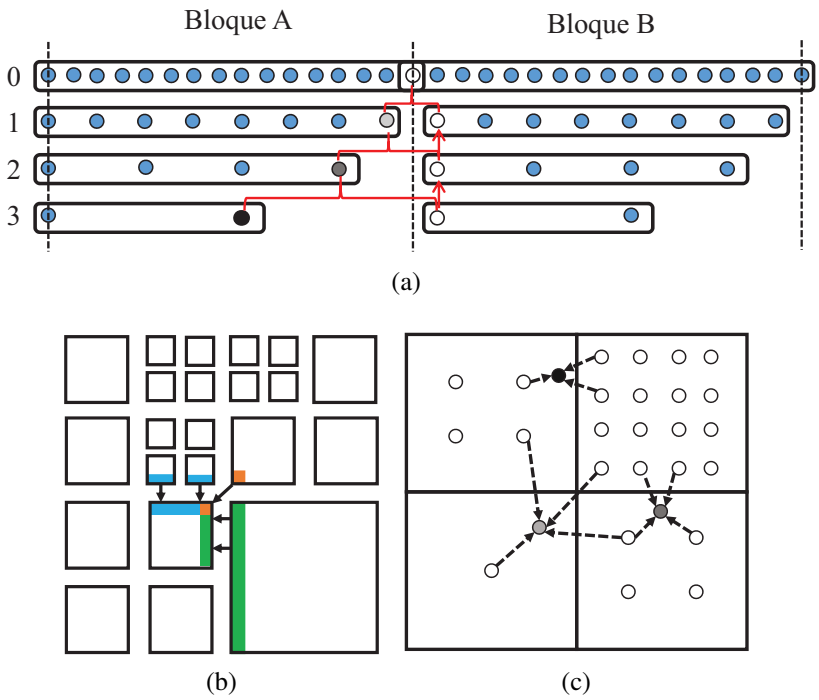


Figura 1.9: Reducción de artefactos en fronteras entre diferentes niveles de detalle. (a) Los vóxeles entre bloques vecinos son ajustados con los valores de frontera de los niveles de detalle inferior. (b) Se realiza una copia de los vóxeles de frontera de los *bricks*, solo considerando el vecino de la cara derecha, superior y trasera. (c) Interpolación entre *blocks*, lograda haciendo un promedio ponderado de los valores de frontera de los vóxeles vecinos.

de las fronteras de los bloques de alta resolución son ajustados utilizando los bloques de baja resolución adyacentes. Al momento del despliegue, los bloques de alta resolución utilizan funciones especiales que les permiten realizar una correcta interpolación entre los dos niveles de detalle.

Finalmente, Carmona et al. [58] proponen una interpolación entre *bricks* adyacentes que no solo toma en cuenta los vóxeles de la frontera. En su trabajo se realiza una mezcla entre un *brick* de una determinada resolución con su *brick* padre como puede observarse en la Figura 1.10. Aquí se tiene el volumen original en el nivel i y el nivel $i-1$ de menor resolución. La selección a desplegar son los bloques A, B y CD. En la frontera entre B y AB se aplica el proceso de mezclado entre las muestras obtenidas en B $(x(s,t,r))$ con la muestras obtenidas en AB $(p(s',t',r'))$, utilizando la fórmula $blend(\alpha) = (1-\alpha) * x + \alpha * p$. Esta técnica genera una transición más suave, pero obliga a mantener en el *working set* dos niveles de detalle por cada *brick*.

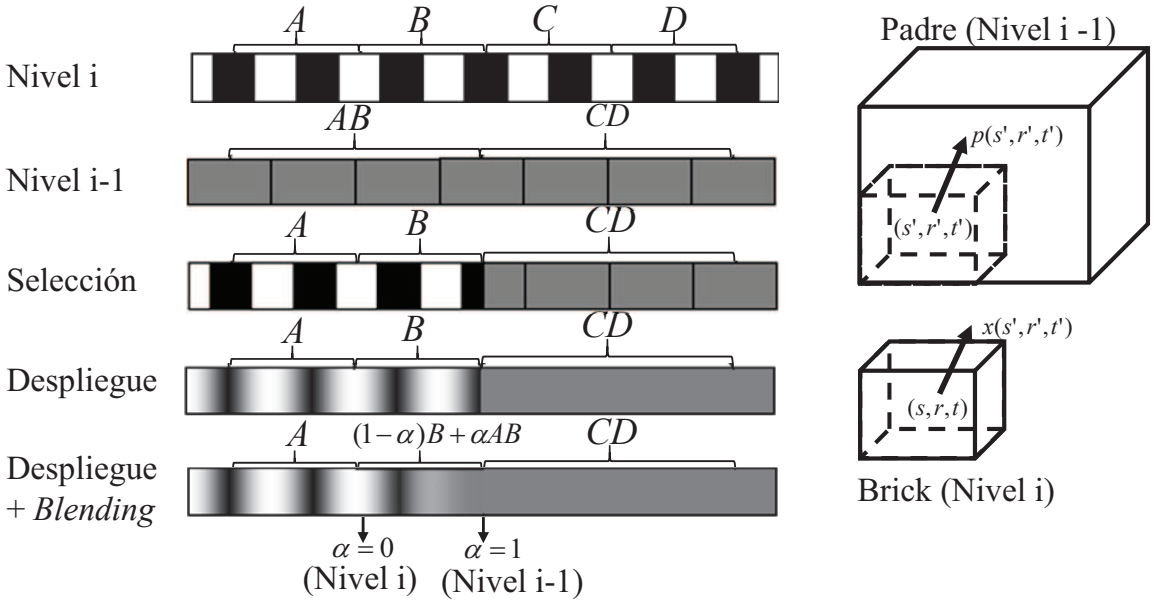


Figura 1.10: Mezcla entre dos *bricks* para una correcta interpolación entre diferentes niveles de detalle.

1.6. Proceso de Despliegue

A la hora de realizar el *ray casting* de un volumen de gran tamaño, actualmente lo más utilizado es el recorrido del rayo sobre la textura atlas en conjunto con una textura de índices [4] [6] [8] [9], como se puede observar en la Figura 1.11. Para realizar el proceso de despliegue, primero la textura de índices es aplicada a un cubo unitario (espacio $[0, 1]^3$), el cual representará virtualmente al volumen. El algoritmo de *ray casting* se aplica sobre este cubo, tomando muestras equidistantes de la textura de índice.

En los canales RGBA de cada vóxel de la textura de índices se almacenará la posición 3D del bloque o *brick* en la textura atlas, y en el canal alfa se almacenará el nivel de detalle en el cual está representado este bloque o *brick*. En la Figura 1.11 se muestra una representación 2D de la textura atlas, donde se puede observar como en la travesía del rayo, los vóxeles de la textura de índices redireccionan a diferentes zonas de la textura atlas. Si se tiene una jerarquía de bloques, éstos pueden tener diferentes tamaños, por lo que ocupan diferentes áreas dentro de la textura atlas. Por ello, al momento de realizar el muestreo, el rayo debe ser escalado y trasladado dependiendo del nivel de detalle. En la figura se destacan dos bloques sobre los que se toman muestras, en los que se puede observar que están representados en diferentes niveles de detalle, por lo que las coordenadas de muestreo deben ser transformadas acorde a su nivel.

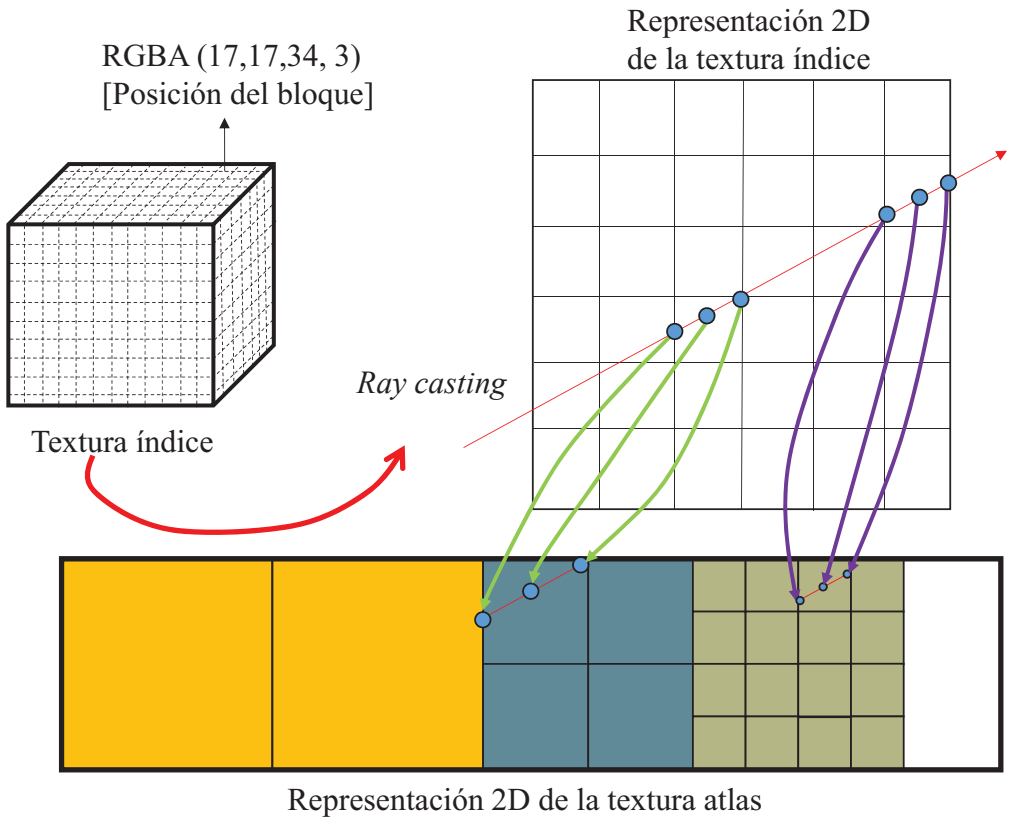


Figura 1.11: Recorrido de una textura atlas usado por López et al. [4] y Fernández et al. [6]. Cada vóxel de la textura de índices tiene una tupla RGBA (xp, py, pz, lod). Utilizando una representación 2D se puede observar la interacción entre la textura de índices y el atlas en cada paso del rayo.

Capítulo 2

Solución Propuesta

Este trabajo se desarrolló utilizando una metodología *ad-hoc* y se puede dividir en dos partes: las implementaciones de un *ray casting* básico con diferentes APIs paralelos, y la implementación de despliegue usando textura atlas. En este capítulo, primero se explicará en la Sección 2.1 las implementaciones de un *ray casting* básico con diferentes APIs paralelos. Posteriormente, se expondrán los detalles de implementación del despliegue de volúmenes por bloques utilizando la textura atlas en la Sección 2.2, la cual será utilizada para el despliegue de volúmenes de mayor resolución.

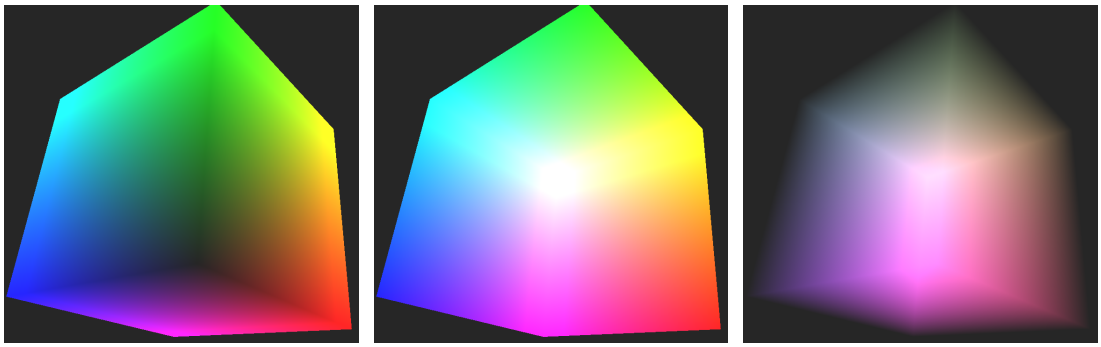
2.1. *Ray Casting* Básico de Volúmenes en GPU

En esta sección se exponen detalles de implementación de un *ray casting* básico utilizando *fragment shader*, *compute shader*, OpenCL y CUDA, las cuales serán utilizadas para medir el desempeño de estos APIs con diferentes escenarios y datasets. Primero, se tiene una breve descripción de los algoritmos para el cálculo de la intersección rayo/volumen, utilizando la rasterización y la intersección rayo/caja. Posteriormente, se muestran fragmentos de código con detalles específicos para cada una de las implementaciones paralelas a comparar. Finalmente, se describe la técnica utilizada para poder calcular la iluminación difusa en los volúmenes. En el Apéndice A se encuentra teoría básica sobre estos lenguajes de programación paralela que puede ser útil para la comprensión de estas implementaciones.

2.1.1. Cálculo de Intersección Utilizando la Rasterización

Una manera eficiente de obtener la intersección entre los rayos de visión y un volumen es utilizando la rasterización de OpenGL. La idea es utilizar el poder de la rasterización paralela de las tarjetas gráfica para generar la intersección rayo/volumen para cada uno de los píxeles del *framebuffer* [20]. Para ello, se despliega un cubo de color representando las fronteras

del volumen (observar Figura 2.1). Dado que las coordenadas de color y de textura están representadas en el mismo espacio unitario $[0, 1]^3$, cada color del cubo también representará una coordenada 3D de textura del volumen. De esta manera, el color de cada píxel rasterizado representará la coordenada de textura de la intersección de un rayo con el volumen. Este método es utilizado con la remoción de caras traseras (*back face culling*) para obtener la primera intersección de cada rayo con el volumen (punto de entrada, Figura 2.1a), y con la remoción de caras delanteras (*front face culling*) para obtener la última intersección (punto de salida, Figura 2.1b). Además, la dirección del rayo puede ser obtenida como la resta entre el punto de entrada menos el punto de salida (Figura 2.1c).



(a) (b) (c)

Figura 2.1: Imágenes obtenidas con rasterización, que contienen: (a) el primer punto de intersección del rayo con el volumen, (b) el último punto de intersección del rayo con el volumen, y (c) la dirección del rayo, la cual puede ser obtenida con la diferencia de la intensidad de colores entre las imágenes (b) y (a).

2.1.2. Cálculo de Intersección Rayo/Caja

La intersección de un rayo individual con el volumen también puede ser calculada analíticamente. Un algoritmo de enfoque divide y vencerás es comúnmente utilizado para este cálculo. Intersectar un rayo con una caja puede ser reducido a intersectar un rayo con cada corte (*slab*) del volumen [59]. En la Figura 2.2 se puede observar un ejemplo de este algoritmo en 2D. La idea es intersectar el rayo con las dos líneas verticales primero para obtener el $tmin_x$ y el $tmax_x$, y luego con las dos líneas horizontales para obtener el $tmin_y$ y el $tmax_y$. Finalmente, el punto de intersección $tnear$ es calculado como $max(tmin_x, tmin_y)$, y $tfar$ es calculado como $min(tmax_x, tmax_y)$, representando los puntos de entrada y de salida del rayo respectivamente.

Este algoritmo ha sido optimizado para trabajar en arquitecturas paralelas, evitando la divergencia entre hilos [60]. Una implementación del algoritmo paralelo puede ser vista en el

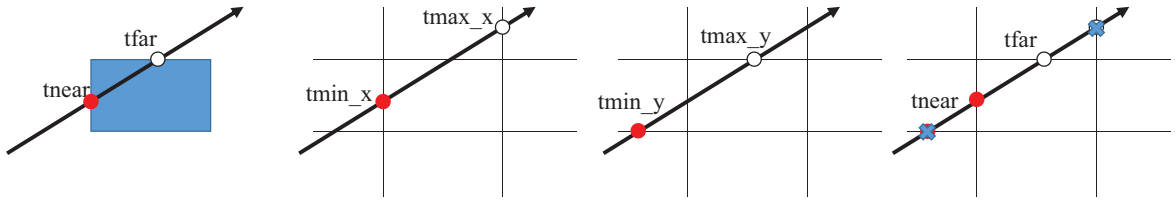


Figura 2.2: Pasos para la intersección rayo/caja.

Código 2.1. Aquí se aprovecha la aritmética vectorial de los APIs paralelos para realizar el cálculo sin necesidad de ciclos ni condicionales, los cuales pueden afectar considerablemente el desempeño del algoritmo.

```
bool intersectBox(Ray r, out float tnear, out float tfar)
{
  //caja envolvente unitaria del volumen
  vec3 boxmin = vec3(-.5f, -.5f, -.5f);
  vec3 boxmax = vec3(.5f, .5f, .5f);

  //calcular la intersección del rayo con los seis planos
  vec3 invR = 1.0f / r.d;
  vec3 tbot = invR * (boxmin - r.o);
  vec3 ttop = invR * (boxmax - r.o);

  //ordenar para encontrar el valor máximo y mínimo en cada eje
  vec3 tmin = vec3(100000.0f);
  vec3 tmax = vec3(0.0f);

  tmin = min(tmin, min(ttop, tbot));
  tmax = max(tmax, max(ttop, tbot));

  //encontrar el tmin más grande y el tmax más pequeño
  tnear = max(max(tmin.x, tmin.y), tmin.z);
  tfar = min(min(tmax.x, tmax.y), tmax.z);

  return tfar > tnear;
}
```

Código 2.1: Código para el cálculo de la intersección rayo/caja optimizado para evitar la divergencia entre hilos.

2.1.3. Implementaciones Paralelas

La implementación del *ray casting* de volúmenes en diferentes APIs paralelas tienen el mismo algoritmo básico. Sin embargo, cada API tiene sus propias limitaciones que vale la pena mencionar. En esta sección se presentan los detalles de las implementaciones del *ray casting* de volúmenes utilizadas en este trabajo, considerando el *fragment shader*, *compute shader*, OpenCL y CUDA. Solamente se muestran las implementaciones con el método de rasterización, debido a que el algoritmo de intersección rayo/caja fue mostrado previamente en el Código 2.1, y este algoritmo puede ser fácilmente adaptado a cada una de las APIs. Además,

esta sección presenta una implementación para añadir iluminación difusa al *ray casting* de volúmenes.

Fragment shader

Dado que es necesario desplegar alguna geometría con OpenGL para activar programas de fragmentos, podemos calcular la intersección de cada rayo con el volumen durante la rasterización, utilizando un cubo unitario. En este caso solo será necesario realizar dos pasadas. La primera pasada calculará la salida del rayo, y la segunda calculará la entrada del rayo, la dirección y evaluará numéricamente la ecuación de despliegue de volúmenes. Esta segunda pasada, que es la que posee el algoritmo de *ray casting*, puede observarse en el Código 2.2.

```
layout(binding = 1) uniform sampler2D lastHit;
layout(binding = 2) uniform sampler1D transferFunction;
layout(binding = 3) uniform sampler3D volume;
uniform float h;

in vec3 first;

#define opacityThreshold 0.99

layout(location = 0) out vec4 vFragColor;

void main(void){
  ivec2 tcoord = ivec2(gl_FragCoord.xy);
  vec3 last = texelFetch(lastHit, tcoord, 0).xyz;

  //obtener dirección del rayo
  vec3 direction = last.xyz - first.xyz;
  float D = length (direction);
  direction = normalize(direction);

  vec4 color = vec4(0.0f);
  color.a = 1.0f;

  vec3 tr = first;
  vec3 rayStep = direction * h;

  for(float t =0; t<=D; t += h){
    //muestrear el campo escalar y la función de transferencia
    vec4 samp = texture(transferFunction,
                texture(volume, tr).x).rgba;

    //calcular alfa
    samp.a = 1.0f - exp(-0.5f * samp.a);

    //acumular color y alfa usando el operador under
    samp.rgb = samp.rgb * samp.a;

    color.rgb += samp.rgb * color.a;
    color.a *= 1.0f - samp.w;

    //chequear si hay terminación temprana del rayo
    if(1.0f - color.w > opacityThreshold)     break;

    //incrementar el paso del rayo
    tr += rayStep;
```

```
  }

  color.w = 1.0f - color.w;
  vFragColor = color; //asignar el color final
}
```

Código 2.2: Implementación del despliegue de volúmenes utilizando el *fragment shader*.

En la primera pasada, las caras traseras del cubo unitario son desplegadas, y por cada píxel rasterizado, obtenemos el punto de salida del volumen, los cuales son almacenados en una textura. Durante la segunda pasada, se despliegan las caras frontales del cubo unitario, representando el punto de entrada del rayo en el volumen, los cuales se obtienen por la interpolación de la variable `first`. Las texturas requeridas para el despliegue son indicadas al shader a través de `samplers`; estas texturas representan respectivamente el punto de salida del rayo por píxel (`lasHit`), la función de transferencia (`transferFunction`), y el volumen propiamente (`volume`). El punto de salida del rayo (`lasHit`) es obtenido muestreando con la función `texelFetch`. Con `first` y `lasHit`, es posible calcular la dirección del rayo (`direction`) y su longitud (`D`). Posteriormente, el volumen es atravesado por el rayo, en un ciclo paso a paso. En cada paso, el volumen es muestreado en la posición actual del rayo y la posición es movida `h` unidades en la dirección del rayo en cada iteración. Para acceder las texturas de la función de transferencia y el volumen, se utiliza la función `texture` ya que queremos obtener téxeles y vóxeles interpolados respectivamente. El código está optimizado para realizar terminación temprana del rayo, lo que significa que cuando el rayo haya acumulado un valor de opacidad mayor a un umbral (`opacityThreshold`), el ciclo se termina. Posteriormente, el color resultante y la opacidad son almacenados en el *framebuffer*, el cual es indicado con la variable de salida `vFragColor`.

Compute shader

El Código 2.3 muestra la implementación del *ray casting* de volúmenes usando el *compute shader*. La lógica y el código es similar al presentado para el *fragment shader*. En esta implementación, el punto de entrada y salida del rayo son obtenidos mediante la rasterización de un cubo unitario con OpenGL, donde se almacenan estos valores en texturas para cada uno de los píxeles de la pantalla. Estas texturas son utilizadas por el *compute shader* mediante los `samplers` `firstHit` (punto de entrada) y `lastHit` (punto de salida). Aquí el acceso a las texturas se realiza de la misma manera que en el *fragment shader* usando `samplers`, donde la única diferencia es la textura de salida con la imagen final, la cual debe ser enlazada a una imagen OpenGL con la función `glBindImageTexture`. En contraste con las texturas, las imágenes en OpenGL permiten ser modificadas por un *shader*.

```
//almacenar en una textura
layout(binding = 0, rgba8) uniform writeonly image2D destTex;
layout(binding = 1) uniform sampler1D transferFunc;
layout(binding = 2) uniform sampler3D volume;
layout(binding = 3) uniform sampler2D lastHit;
```

```
layout(binding = 4) uniform sampler2D firstHit;

uniform float h, width, height, depth;
#define opacityThreshold 0.99

void main(){
  //leer la identificación global de este hilo
  ivec2 storePos = ivec2(gl_GlobalInvocationID.xy);

  //el tamaño de la imagen indica cuantos hilos en total se necesitan
  //size.x*size.y hilos en total, identificándolos de manera unívoca en cada dirección
  ivec2 size = imageSize(destTex);
  if(storePos.x < size.x && storePos.y < size.y){
    storePos.y = size.y - storePos.y;
    vec3 last = texelFetch(lastHit, storePos, 0).xyz;
    vec3 first = texelFetch(firstHit, storePos, 0).xyz;

    //obtener dirección del rayo
    vec3 direction = last.xyz - first.xyz;
    float D = length (direction);
    direction = normalize(direction);

    vec4 color = vec4(0.0f);
    color.a = 1.0f;

    vec3 tr = first;
    vec3 rayStep = direction * h;

    for(float t =0; t<=D; t += h){
      //muestrear el campo escalar y la función de transferencia
      float scalar = texture(volume,tr).x;
      vec4 samp = texture(transferFunc, scalar).rgba;

      //calcular alfa
      samp.a = 1.0f - exp(-0.5 * samp.a);

      //acumular color y alfa usando el operador under
      samp.rgb = samp.rgb * samp.a;

      color.rgb += samp.rgb * color.a;
      color.a *= 1.0f - samp.w;

      //chequear si hay terminación temprana del rayo
      if(1.0f - color.w > opacityThreshold) break;

      //incrementar el paso del rayo
      tr += rayStep;
    }

    color.w = 1.0f - color.w;
    imageStore(destTex, storePos, color);
  }
}
```

Código 2.3: Implementación del despliegue de volúmenes utilizando el *compute shader*.

Para lanzar el kernel paralelo, la función `glDispatchCompute` debe ser invocada, indicando el número total de hilos y el número de hilos por bloque. A diferencia del *fragment shader*, un kernel es ejecutado por cada hilo y la variable `gl_GlobalInvocationID` diferencia de manera unívoca un hilo de otro. Cada hilo se encargará de procesar un píxel de la pantalla, el cual será escogido dependiendo del identificador 2D del hilo. Tomando

muestras en las texturas `lastHit` y `firstHit` se calcula el rayo que pasa por ese píxel y se realiza todo el proceso del recorrido del rayo. Dado que la ejecución del kernel es asíncrono, la función `glMemoryBarrier` debe invocarse con el parámetro `GL_SHADER_IMAGE_ACCESS_BARRIER_BIT` para asegurar que el resultado es escrito en la textura antes de que pueda ser utilizado para el despliegue final. Por último, un cuadrado del tamaño de la pantalla es desplegado, y se texturiza con la imagen final obtenida anteriormente.

OpenCL

La implementación con OpenCL es presentada en el Código 2.4. En este caso, un API diferente a OpenGL es utilizado para calcular la imagen final, por lo que deben usarse algunas funciones que permitan la interoperabilidad para realizar la comunicación entre ambos APIs. Primero, todas las texturas OpenGL deben ser enlazadas a una imagen OpenCL utilizando `clCreateFromGLTexture`. Antes de invocar el kernel, OpenCL debe indicar a OpenGL cada textura a utilizar con `clEnqueueAcquireGLObjects`, y luego liberarlas después de usadas con `clEnqueueReleaseGLObjects`. La función `clEnqueueNDRangeKernel` debe ser invocada para lanzar un kernel, indicando el número total de hilos y el número total de hilos por bloque.

```
#define opacityThreshold 0.99

__kernel void volumeRendering(__write_only image2d_t d_output, float constantH, unsigned ↩
    int W, unsigned int H, __read_only image3d_t volume, __read_only image2d_t ↩
    transferFunc, sampler_t volumeSampler, sampler_t transferFuncSampler, __read_only ↩
    image2d_t firstTex, __read_only image2d_t lastTex, sampler_t hitSampler)
{

  int2 gid = (int2)(get_global_id(0), get_global_id(1));

  if (gid.x < W && gid.y < H){
    float2 Pos = (float2)((gid.x + 0.5f) / (float)W,
                (gid.y + 0.5f) / (float)H);

    float4 color;

    float4 first4D = read_imagef(firstTex, hitSampler, Pos);
    float3 first = (float3)(first4D.x, first4D.y, first4D.z);
    float4 last4D= read_imagef(lastTex, hitSampler, Pos);
    float3 last = (float3)(last4D.x, last4D.y, last4D.z);

    //obtener dirección del rayo
    float3 direction = last - first;
    float D = length(direction);
    direction = normalize(direction);

    color = (float4)(0.0f);
    color.w = 1.0f;

    float3 tr = first;
    float3 rayStep = direction * constantH;

    for (float t = 0; t <= D; t += constantH){
      //muestrear el campo escalar y la función de transferencia
```

```
      float4 scalar = read_imagef(volume, volumeSampler, (float4)(tr.x, tr.y, tr.z, 0.0f))↩
          ;
      float4 samp = read_imagef(transferFunc, transferFuncSampler, (float2)(scalar.x, 0.5f↩
          ));

      //calcular alfa
      samp.w = 1.0f - exp(-0.5 * samp.w);

      //acumular color y alfa usando el operador under
      samp.x = samp.x * samp.w;
      samp.y = samp.y * samp.w;
      samp.z = samp.z * samp.w;

      color.x += samp.x * color.w;
      color.y += samp.y * color.w;
      color.z += samp.z * color.w;
      color.w *= 1.0f - samp.w;

      //chequear si hay terminación temprana del rayo
      if (1.0f - color.w > opacityThreshold) break;

      //incrementar el paso del rayo
      tr += rayStep;
    }

    color.w = 1.0f - color.w;

    write_imagef(d_output, gid, color);
  }
}
```

Código 2.4: Implementación del despliegue de volúmenes utilizando OpenCL.

Como en el caso de *compute shader*, la función es ejecutada una vez por cada hilo y cada hilo es identificado de manera unívoca con la función `get_global_id`. Cada hilo se encargará de procesar un píxel de la pantalla, el cual será escogido dependiendo del identificador 2D del hilo. Tomando muestras en las texturas `lastTex` y `firstTex` se calcula el rayo que pasa por ese píxel y se realiza todo el proceso del recorrido del rayo. Finalmente, `clFinish` debe ser llamado para asegurar que la imagen final está lista para ser desplegada por OpenGL. Algunas funciones para la manipulación de los tipos de datos fueron agregadas a OpenCL para hacer el código más legible.

CUDA

Finalmente, la implementación utilizando CUDA puede observarse en el Código 2.5. Al igual que OpenCL, CUDA tiene funciones que permiten la interoperabilidad con OpenGL.

```
typedef unsigned char VolType;
#define opacityThreshold 0.99

texture<VolType, 3, cudaReadModeNormalizedFloat> volume;
texture<float4, 2, cudaReadModeElementType> transFunc;
texture<float4, 2, cudaReadModeElementType> texFirst;
texture<float4, 2, cudaReadModeElementType> texLast;
surface<void, cudaSurfaceType2D> surf;
```

```
__constant__ unsigned int constantWidth, constantHeight;
__constant__ float constantH;

__global__ void volumeRenderingKernel(){
  unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
  unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;


  if (x < constantWidth && y < constantHeight){
    float2 Pos = mfloat2(x, y);

    float4 first = tex2D(texFirst, x, y);
    float4 last = tex2D(texLast, x, y);

    //obtener dirección del rayo
    float3 direction = mfloat3(last) - mfloat3(first);
    float3 tr = mfloat3(first);

    float D = length(direction);
    direction = normalize(direction);

    float4 color = mfloat4(0.0f);
    color.w = 1.0f;

    float3 rayStep = direction * constantH;

    for (float t = 0; t <= D; t += constantH){
      //muestrear el campo escalar y la función de transferencia
      float scalar = tex3D(volume, tr.x, tr.y, tr.z);
      float4 samp = tex2D(transFunc, scalar, 0.5f);

      //calcular alfa
      samp.w = 1.0f - expf(-0.5f * samp.w);

      //acumular color y alfa usando el operador under
      samp.x = samp.x * samp.w;
      samp.y = samp.y * samp.w;
      samp.z = samp.z * samp.w;

      color.x += samp.x * color.w;
      color.y += samp.y * color.w;
      color.z += samp.z * color.w;
      color.w *= 1.0f - samp.w;

      //chequear si hay terminación temprana del rayo
      if (1.0f - color.w > opacityThreshold) break;

      //incrementar el paso del rayo
      tr += rayStep;
    }

    color.w = 1.0f - color.w;

    //escribir a la textura final
    uchar4 ucolor = muchar4(color.x * 255, color.y * 255, color.z * 255, color.w * 255);
    surf2Dwrite(ucolor, surf, x * sizeof(uchar4), y, cudaBoundaryModeClamp);
  }
}
```

Código 2.5: Implementación del despliegue de volúmenes utilizando CUDA.

Casi todas las texturas OpenGL deben ser enlazadas a arreglos de CUDA con las funciones

`cudaGraphicsGLRegisterImage` y `cudaGraphicsSubResourceGetMapped-Array`. Posteriormente el arreglo debe corresponderse a una textura CUDA con la función `cudaBindTextureToArray`. La única excepción a esta regla es la imagen resultante. CUDA no puede escribir a una textura, por lo que el arreglo correspondiente a la textura de salida debe corresponderse a una superficie CUDA con `cudaBindSurfaceToArray`. Esta correspondencia entre una textura OpenGL, y una textura o superficie CUDA solo tienen que realizarse una vez, y posteriormente la textura OpenGL va a tener los mismos espacios de memoria que la textura o superficie CUDA. De esta manera, CUDA va a tener acceso a las texturas de salida de OpenGL sin necesidad de ninguna operación extra.

Igual que en el caso del *compute shader* y OpenCL, la función debe ser ejecutada una vez por cada hilo y un hilo es identificado de manera unívoca con las variables `blockIdx`, `blockDim`, y `threadIdx`. Cada hilo se encargará de procesar un píxel de la pantalla, el cual será escogido dependiendo del identificador 2D del hilo. Tomando muestras en las texturas `texFirst` y `texLast` se calcula el rayo que pasa por ese píxel y se realiza todo el proceso del recorrido del rayo. Cuando el kernel es invocado, el número de hilos por bloque y el número de bloques debe ser indicado. Finalmente, `cudaDeviceSynchronize` debe ser llamado para esperar hasta que se termine la ejecución del kernel. Algunas funciones para la manipulación de los tipos de datos fueron agregadas a CUDA para hacer los código más legible.

2.1.4. Iluminación difusa

Para iluminar un volumen, la normal de su superficie debe ser aproximada para cada vóxel utilizando diferencias finitas. Las diferencias finitas en un campo escalar discreto como el volumen son calculadas como la resta del valor del vóxel actual con sus vecinos en cada eje, con lo cual se obtiene el vector gradiente. Este vector puede ser normalizado y utilizado como el vector normal del volumen en la correspondiente posición del vóxel [61]. El Código 2.6 muestra un código GLSL con una función para calcular la iluminación difusa. La misma lógica puede ser implementada para el *compute shader*, OpenCL y CUDA. Esta función debe ser llamada en el ciclo de evaluación del rayo, para cada muestra que se tome dentro del volumen. Los parámetros de entrada son: el color correspondiente a la muestra actual (`samp`), la coordenada dentro del volumen que se está muestreando (`tc`), y el valor escalar de esta muestra (`scalar`). Con estos valores es posible tomar muestras en los vóxeles vecinos, calcular el vector gradiente, y finalmente modificar el color de la muestra actual dependiendo del componente difuso de la luz, su posición y la orientación de la superficie.

```
uniform vec3 lightDir; //dirección de la luz en el espacio ojo
uniform vec3 diffColor; //color difuso de la luz
uniform vec3 voxelJump; //distancia entre vóxeles

//calcular la iluminación para el escalar muestreado
vec4 litSample(vec4 samp, vec3 tc, float scalar){
```

```
  //muestrear vecinos
  vec3 normal;
  vec3 displacement = vec3(voxelJump.x,0.0f,0.0f); //muestreo a la derecha
  normal.x = texture(volume, tc + displacement).x - scalar;
  displacement = vec3(0.0f,voxelJump.y,0.0f); //muestreo hacia arriba
  normal.y = texture(volume, tc + displacement).x - scalar;
  displacement = vec3(0.0f,0.0f,-voxelJump.z);  //muestreo con el vóxel trasero
  normal.z =  texture(volume, tc + displacement).x - scalar;

  //normalizar la normal
  normal = normalize(normal);

  //producto cruz con la luz
  float d = max(dot(lightDir, normal), 0.0f);

  //calcular contribución
  samp.xyz = samp.xyz * d * diffColor;
  return samp;
}
```

Código 2.6: Cálculo de la iluminación difusa de una muestra.

2.2. Visualizador Multi-Resolución Usando Textura Atlas

Para el despliegue de volúmenes de alta resolución se utilizó un esquema de jerarquía por bloques, junto a una textura atlas para el almacenamiento del *working set* a desplegar [4] [6] [8] [9]. La implementación sigue el esquema mostrado en la Figura 2.3. Como se puede observar, el primer paso es la carga del volumen de disco, y la creación de la jerarquía de bloques con cada uno de sus niveles de detalle. La información del volumen será leída de un archivo de entrada del cual se extraerán los vóxeles, y se agruparán de manera de obtener los bloques a utilizar. Cada bloque pasará por un proceso de submuestreo, con el cual se obtendrán resoluciones menores del volumen en esa área hasta obtener la jerarquía completa. Esta jerarquía es almacenada primeramente de forma completa en memoria principal, para tener acceso directo a todos los bloques sin necesidad de acceder a memoria secundaria, y obtener de manera más rápida la información. Sin embargo, esto posee la desventaja de que el tamaño del volumen se limita al tamaño de la memoria principal. El proceso de creación de la jerarquía de bloques se explicará en la Sección 2.2.2.

Debido a que en general el volumen en su mayor resolución será más grande que la memoria de GPU, se debe hacer una selección de nivel de detalle para cada bloque del volumen, de tal manera que la selección quepa en la memoria del GPU. Para ello, se utilizan los parámetros de visualización y métricas de error (función de transferencia, punto de vista y distancia al ojo), lo cual permitirá escoger el *working set*, el cual representará el conjunto de bloques a utilizar para desplegar el volumen en la resolución deseada. Este criterio de selección permitirá calcular una prioridad para cada bloque, con lo cual se podrá determinar un orden de importancia de los bloques, tratando de priorizar aquellas zonas que mejorarán de manera significativa la calidad visual del despliegue. La Sección 2.2.3 expone en mayor detalle el cálculo de las

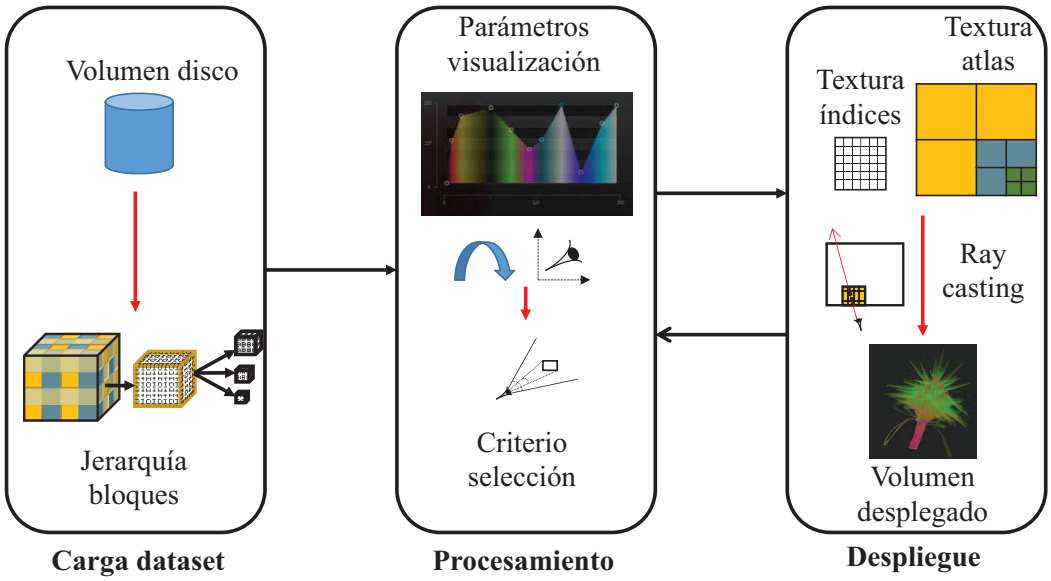


Figura 2.3: Esquema básico de funcionamiento del visualizador multi-resolución implementado.

prioridades y criterio de selección utilizados en este trabajo.

La mayoría de los parámetros de visualización pueden ser modificados por el usuario en tiempo real, por lo que el *working set* cambia constantemente a lo largo de la ejecución del programa. Además, el *working set* es almacenado en memoria de GPU dentro de la textura atlas, y el GPU tiene un ancho de banda limitado para subir información, por lo que la actualización de esta memoria puede ser costosa. Para disminuir el impacto de la actualización, se utiliza un enfoque progresivo *frame* a *frame* donde, de acuerdo a las prioridades, algunos bloques aumentan su nivel de detalle o se refinan, y algunos otros disminuyen su nivel de detalle o se colapsan. En cada *frame* se sube un número limitado de bloques, de manera que no se pierda la interactividad de la aplicación, y el criterio de selección implementado busca maximizar la reducción del error por byte transferido a la memoria de GPU.

Sin embargo, debido a que los bloques tienen diferente nivel de detalle, y por ende diferente tamaño en vóxeles, la organización de los mismos en la textura atlas es complejo, y más si esta se actualiza constantemente. Por ello, se desarrolló un algoritmo de organización y actualización de la textura atlas. La idea es mantener los bloques del *working set* organizados de manera lineal y agrupados por nivel de detalle, de manera que al querer refinar o colapsar un bloque, pueda encontrarse un espacio para el mismo, con relativamente pocos movimientos dentro de la textura, evitando así la desfragmentación constante de la textura atlas. Este algoritmo es explicado en la Sección 2.2.4. La principal ventaja de trabajar la organización del *working set* de manera lineal, es que simplifica sus movimientos y actualizaciones, lo cual en 3D sería extremadamente difícil de lograr. Esto además conlleva a que debe realizarse una conversión de esta organización lineal de los bloques al espacio 3D que representa la textura atlas, y para esto se utiliza el *Morton order* [46], el cual permite realizar estas conversiones de espacios de manera eficiente. La Sección 2.2.1 describe el *Morton order* y su uso en esta

implementación.

Finalmente, el despliegue del volumen, se basa en las ideas mostradas en la Sección 1.6, en donde se utiliza una textura de índices que almacenará la posición 3D del bloque dentro de la textura atlas y el actual nivel de detalle en el que se encuentra. El *ray casting* se realiza sobre esta textura de índices, donde para cada punto en el recorrido del rayo, primero se debe muestrear en la textura índice, la cual hace una referencia a una posición en la textura atlas, que almacena la información real del volumen. Debido al uso de esta técnica, se generan artefactos visuales en la fronteras de los bloques, por lo que se implementó el algoritmo de reducción de artefactos propuesto por Ljung et al. [1], para aumentar la calidad visual de los resultados. La Sección 2.2.5 contiene detalles sobre la implementación de este algoritmo de reducción de artefactos.

En esta propuesta se hace especial énfasis en las optimizaciones para la actualización y manejo eficiente de la textura atlas. La presente sección explica algunos detalles particulares de la implementación realizada, enfocándose en: el uso del *Morton orden* para el direccionamiento de las texturas, la creación de la jerarquía utilizando el GPU, el criterio de selección y cálculo de las prioridades usando GPU, el manejo de la textura atlas y la reducción de artefactos.

2.2.1. *Morton Order* para el Direccionamiento

Basándose en la idea propuesta por Fernández et al. [6] para realizar de manera eficiente el direccionamiento dentro de la textura atlas, se utilizó un orden denominado *Z-Order curve* o *Morton order* [46]. El *Morton order* es una función que hace una correspondencia entre un espacio multidimensional de datos y uno unidimensional preservando la localidad espacial. Esto permite en esta implementación tener estructuras de datos que mantengan una organización de los bloques en un espacio lineal, donde se realizan de manera eficiente todas las operaciones de actualización del *working set*, y luego poder hacer una correspondencia inmediata con el espacio 3D representado en la textura atlas encontrada en el GPU. Además, este orden también es utilizado para el direccionamiento eficiente del espacio 3D en tareas como la creación de la jerarquía y el cálculo de la distorsión.

En [6] para realizar la transformación entre una posición 1D y una 3D, era necesario un algoritmo iterativo y complejo, que al ser una transformación que se realiza constantemente, resulta ineficiente. En este trabajo se optimizó el algoritmo basándose en la técnica propuesta por Baert [62], donde los bits de las componentes de una dirección multidimensional son intercalados para obtener la dirección unidimensional, utilizando el siguiente método:

$$(x, y, z) = (x_n \dots x_1 x_0, y_n \dots y_1 y_0, z_n \dots z_1 z_0) \Rightarrow z_n y_n x_n \dots z_1 y_1 x_1 z_0 y_0 x_0 \tag{2.1}$$

donde (x, y, z) es la posición en el espacio tridimensional, y x_i, y_i y z_i son el bit i de las coordenadas x, y y z respectivamente.

Como ejemplo, teniendo la dirección 3D $(x, y, z) = (5, 9, 1) = (0101, 1001, 0001)$, se intercalan los bits de las componentes x, y, y z de izquierda a derecha para obtener la dirección unidimensional $010001000111 = 1095$. De manera inversa se pueden desintercalar los bits de un dirección unidimensional para obtener una dirección multidimensional. El Código 2.7 muestra funciones que utilizan operaciones de bit para poder hacer el cálculo de las direcciones usando el *Morton order*.

```
unsigned int juntar(unsigned int n){
  n &= 0x000007ff;                  //.... .... .... .... .... .a98 7654 3210
  n = (n ^ (n << 16)) & 0xff0000ff; //.... .a98 .... .... .... .... 7654 3210
  n = (n ^ (n << 8)) & 0x0700f00f;  //.... .a98 .... .... 7654 .... .... 3210
  n = (n ^ (n << 4)) & 0xc30c30c3;  //.... .a98 .... 76.. ..54 .... 32.. ..10
  n = (n ^ (n << 2)) & 0x49249249;  //.a.. 9..8 ..7. .6.. 5..4 ..3. .2.. 1..0
  return n;
}


unsigned int separar(unsigned int n){
  n &= 0x49249249;                  //.a.. 9..8 ..7. .6.. 5..4 ..3. .2.. 1..0
  n = (n ^ (n >> 2)) & 0xc30c30c3;  //.... .a98 .... 76.. ..54 .... 32.. ..10
  n = (n ^ (n >> 4)) & 0x0700f00f;  //.... .a98 .... .... 7654 .... .... 3210
  n = (n ^ (n >> 8)) & 0xff0000ff;  //.... .a98 .... .... .... .... 7654 3210
  n = (n ^ (n >> 16)) & 0x000007ff; //.... .... .... .... .... .a98 7654 3210
  return n;
}


unsigned int intercalar(unsigned int x, unsigned int y, unsigned int z){
  return juntar(x) | (juntar(y) << 1) | (juntar(z) << 2);
}


void desintercalar(unsigned int n, unsigned int &x, unsigned int &y, unsigned int &z){
  x = separar(n);
  y = separar(n >> 1);
  z = separar(n >> 2);
}
```

Código 2.7: Manejo de direccionamiento utilizando el *Morton order*.

Hay que tomar en cuenta que en la textura atlas podrán estar almacenados al mismo tiempo bloques en diversos niveles de detalle. Por lo tanto, lógicamente una dirección unidimensional significará diferentes posiciones dependiendo del nivel de detalle al cual nos estemos refiriendo. Esto puede ser observado en la Figura 2.4, en la que se muestra una textura atlas 2D para mayor comodidad. En la Figura 2.4a vemos como el espacio de 8×8 bloques es dividido en 4 bloques del mayor nivel de detalle (de 4×4). Aquí, para encontrar la dirección *Morton* del bloque 3 de mayor nivel de detalle, se debe multiplicar su posición unidimensional por la cantidad de vóxeles del bloque $3 \times 16 = 48$, y desintercalando los bits se obtiene su dirección 2D $(4, 4)$. Luego, la Figura 2.4b nos muestra como ese mismo espacio es dividido en el menor nivel de detalle. En este caso, una dirección unidimensional es multiplicada por el tamaño en vóxeles del bloque, el cual es 1.

Finalmente, en un escenario real, la textura atlas será direccionada con diferentes niveles de detalle al mismo tiempo como puede observarse en la Figura 2.4c. Aquí, hay que tomar

en cuenta que nuestra manera de posicionar los bloques en la textura atlas obliga a que los bloques se organicen de manera continua de mayor a menor nivel de detalle, agrupando los bloques con el mismo nivel de detalle. Por ello, la dirección unidimensional de cada bloque dependerá de la cantidad de vóxeles ocupados por los bloques de niveles anteriores, y los bloques anteriores de su mismo nivel de detalle. Así, el bloque ubicado en la posición $(6, 2)$, tiene la dirección *Morton* 28 debido a que hay 1 bloque de tamaño 16 y 3 de tamaño 4.

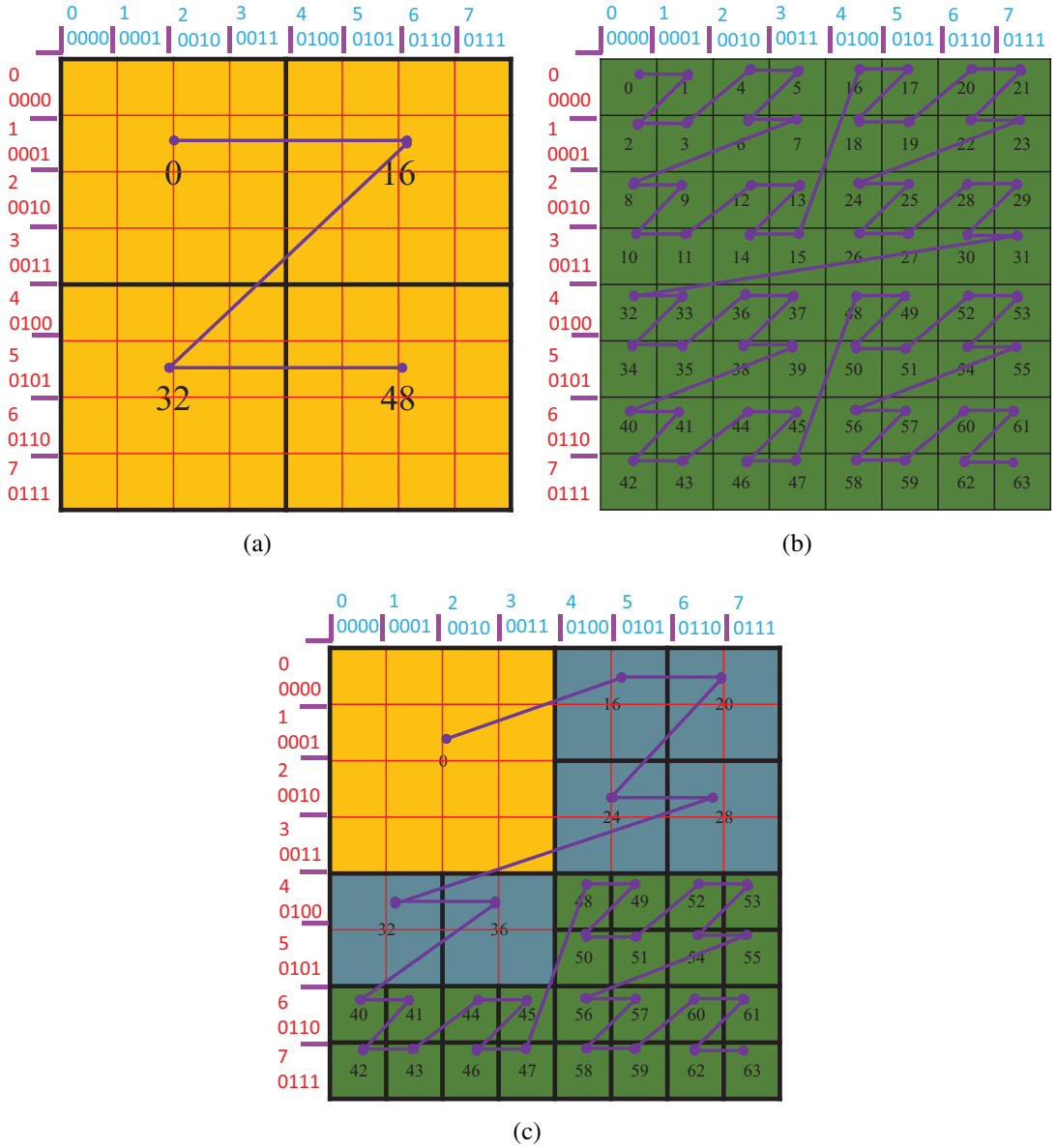


Figura 2.4: Imagen mostrando el direccionamiento con el *Morton order*, usando un espacio 2D de 8×8 y direccionando con: (a) objetos que ocuparían 4×4, (b) objetos que ocuparían 1×1, y (c) objetos con diferentes tamaños.

El uso de este direccionamiento contiene algunas desventajas que hay que tomar en cuenta. Debido al secuenciamiento de la data utilizando una forma de Z, obliga a que el espacio a direccionar deba tener las mismas dimensiones en al menos los ejes x y y, y el eje z debe ser de al menos la mitad de la dimensión de los otros dos ejes. Por otro lado, si se utiliza un entero sin signo para la representación de la dirección, se poseen 32 bits de precisión para la dirección unidimensional, y para el caso 3D esta dirección podrá ser desintercalada en 11 bits para el canal x, 11 bits para el canal y, y 10 bits para el canal z. De esta manera las dimensiones de la textura atlas estarían limitadas a $x, y \in [0, 2048]$ y $z \in [0, 1024]$, lo cual representaría un máximo de 4 GB.

2.2.2. Creación de la Jerarquía

Al momento de seleccionar el dataset a desplegar, es necesario ir cargando el volumen bloque a bloque y crear todos los niveles de detalle de cada bloque. La jerarquía completa es almacenada en memoria principal para poder tener acceso rápido a los diferentes niveles y poder actualizar la textura atlas de manera eficiente. Por ello, se asume que el volumen completo con todos sus niveles de resolución puede ser almacenado en la memoria principal, la cual en la mayoría de las veces tiene mayor capacidad que la memoria de video. Para obtener el volumen en sus diferentes resoluciones, se implementó tanto un algoritmo en CPU como en GPU.

En el caso del algoritmo en CPU se fue extrayendo bloque a bloque del archivo de entrada del volumen. Este bloque se encontraría en el mayor nivel de detalle (n vóxeles en total), y para obtener el nivel de detalle siguiente se deben recorren todos los vóxeles del bloque y se promedian los 8 vecinos más cercanos, obteniendo un bloque de tamaño $n/8$. El proceso se repite de manera iterativa hasta llegar a un bloque de tamaño 1, lo cual indica que ya se han obtenido todos los niveles de detalle de dicho bloque.

Para el caso del algoritmo en GPU, se utilizó la textura atlas como una textura auxiliar para realizar el submuestro en el GPU, la cual al momento de carga no es usada para el despliegue. Varios bloques son extraídos del archivo de entrada del volumen, llenando $7/8$ de la textura atlas, y utilizando el $1/8$ restante para ir almacenando los bloques submuestreados en sus diferentes niveles de resolución. Esto se debe a que si tenemos un bloque de tamaño w, el total de espacio que se necesitaría para almacenar todos sus niveles de detalles sería $\sum_{i=0}^{maxLoD} w/8^i$, y si tenemos una memoria total donde caben B bloques de tamaño w, almacenando inicialmente $B * 7/8$ bloques (ocupando un espacio de $B * w * 7/8$), la cantidad de memoria que necesitamos para almacenar los restantes niveles de detalles de esos bloques es $B * 7/8 * \sum_{i=1}^{maxLoD} w/8^i = B * w/8$, con lo cual $B * w * 7/8 + B * w/8 = B * w$ (tamaño total de la memoria).

En la Figura 2.5 se observa un ejemplo 2D de cómo sería este proceso. Para este caso, se muestra una textura atlas 2D de 2×4, por lo que se puede utilizar $3/4$ de la textura (6 bloques en máxima resolución) para almacenar bloques en su mayor nivel de detalle. De esta manera, en el $1/4$ restante (2 bloques en máxima resolución) se almacenan todas las resoluciones de los

primeros 6 bloques. Para el rápido almacenamiento, organización y ubicación de los bloques, se utiliza el *Morton order*, por lo que los bloques del mismo nivel de detalle se encuentran agrupados en este orden. En esta figura la etiqueta del bloque i simboliza el mismo bloque en los diferentes niveles de detalle.

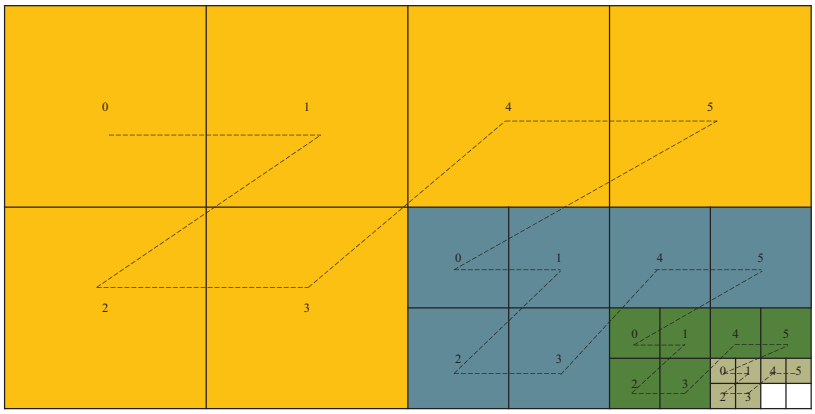


Figura 2.5: División de la memoria para el submuestreo en el GPU, mostrado para una textura 2D.

El proceso de submuestreo en el GPU se puede observar en la Figura 2.6. Por cada bloque del volumen a submuestrear se lanza un bloque de ejecución paralelo, el cual se encargará de obtener todos los niveles de detalle de dicha área del volumen. El bloque de ejecución paralelo es configurado para lanzar $8 \times 8 \times 8 = 512$ hilos, con lo que se pueden tener 4 bloques por *SM* [1] (tomando en cuenta la tarjeta de video utilizada para esta implementación). Inicialmente, con un bloque de tamaño n (nivel de detalle 0), $n/8$ hilos se encargarán de hacer de manera paralela el promedio de sus 8 vóxeles vecinos, obteniendo el siguiente nivel de resolución, el cual es un bloque de tamaño $n/8$ (nivel de detalle 1). Si el tamaño del bloque del volumen en su máxima resolución es mayor que el tamaño del bloque de ejecución paralela, entonces los hilos van a recorrer iterativamente el bloque del volumen, realizando el submuestreo por zonas, como puede observarse en la Figura 2.6. Para poder calcular el bloque de tamaño $n/16$ (nivel de detalle 2), todos los hilos del bloque de ejecución paralelo deben haber terminado sus procesos de cálculo de bloque de tamaño $n/8$ (nivel de detalle 1), por lo que se debe colocar una barrera de sincronización para que los hilos del bloque de ejecución paralelo se esperen entre ellos. Al calcular el bloque de tamaño $4 \times 4 \times 4$, $1/8$ de los hilos quedarán ociosos. Esto se repite para cada nivel de detalle menor a $4 \times 4 \times 4$, por lo que al ir calculando cada nivel de detalle nuevo, el número de hilos ociosos aumentará en $8\times$. Al mínimo nivel de detalle, solo un hilo se encontrará trabajando.

Una vez almacenado todos los niveles de detalle en la textura atlas, la misma debe ser traída desde la memoria de la GPU a la memoria principal y almacenar los diferentes niveles de detalle de los bloques en su correspondientes posiciones en la memoria principal. Esto conlleva

[1] Para mayor información sobre los *Stream Multiprocessor* y su ocupación, consultar el Apéndice A.

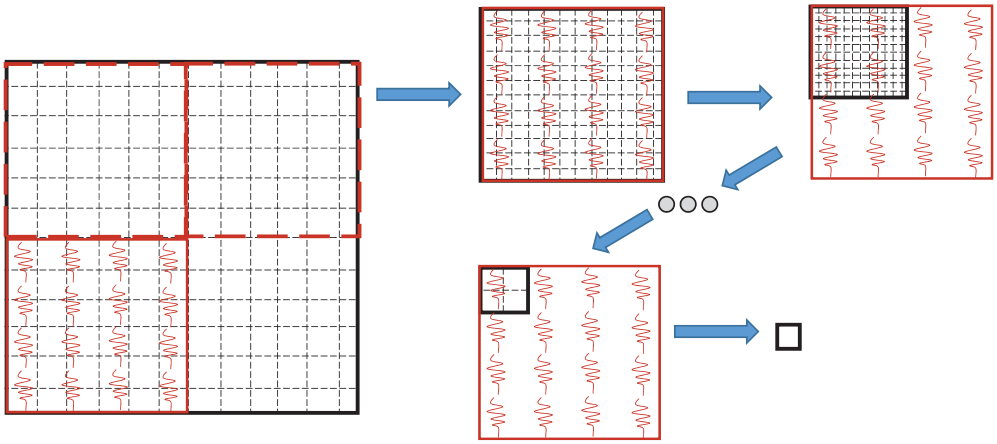


Figura 2.6: Pasos para el submuestreo en el GPU. En las primeras iteraciones el bloque del volumen puede llegar a ser más grande que el bloque de ejecución paralela, por lo que debe realizarse un recorrido iterativo para ir operando sobre subáreas del volumen. Posteriormente, el bloque del volumen será del mismo tamaño que el bloque de ejecución paralela, por lo que el submuestreo de ese nivel se realizará completamente de manera paralela. Luego, el bloque del volumen se irá haciendo progresivamente más pequeño hasta obtener el bloque en su menor resolución, por lo que ciertos hilos irán quedando ociosos.

a que tiene que haber un reposicionamiento de la memoria, la cual puede ser costosa. Sin embargo, debido al alto paralelismo de los cómputos de la creación de la jerarquía, es posible que el tiempo de reposicionamiento no sea un mayor inconveniente. Por ello, en este trabajo se realizaron diferentes pruebas para determinar cual de las dos versiones (CPU o GPU) presenta el mejor comportamiento.

2.2.3. Criterio de Selección y Cálculo de las Prioridades

Debido al limitado ancho de banda para subir información al GPU, y por ende actualizar la textura atlas, se utiliza un sistema de prioridades que maximiza la reducción del error por cada byte a ocupar en la textura. Al momento de decidir cuál es el bloque que se debe refinar y cuál debería ser colapsado, se toma en cuenta la distancia del bloque al ojo, su distorsión, y el número de bytes que necesitarían moverse para poder refinar o colapsar dicho bloque. Además, se define el error global como la suma de los errores de cada uno de los bloques representados en un nivel de detalle asignado. Si el error global es 0, quiere decir que el *working set* actual representa fielmente el volumen en su máxima resolución. Por ello, es de interés tratar de minimizar la función representada por el error global.

En el caso del refinamiento nos interesa maximizar la ganancia, lo cual representa la mayor disminución del error por byte a añadir en el *working set*. Por ello, el cálculo de la ganancia a obtener se realiza con la siguiente ecuación:

$$Ganancia = \frac{\frac{distor(l)}{dist+diag(l)} - \frac{distor(l-1)}{dist+diag(l-1)}}{tam(l-1) - tam(l)} \tag{2.2}$$

donde:

- l es el nivel de detalle actual del bloque.
- $l-1$ es el siguiente nivel superior de detalle del bloque.
- $dist$ es la distancia del bloque al ojo, la cual es independiente del nivel de detalle.
- $distor(i)$ es la distorsión que conlleva representar el bloque en el nivel de detalle i.
- $diag(i)$ es la tamaño de la diagonal del bloque en el nivel de detalle i.
- $tam(i)$ es el tamaño en bytes del bloque en el nivel de detalle i.

Con el término $distor(l)/(dist + diag(l))$ se determina el error de representar el bloque en el nivel de detalle l. Por ello, el error será directamente proporcional a la distorsión de representar el bloque en ese nivel de detalle, e inversamente proporcional a la suma de la distancia al ojo y la diagonal. Esto origina que bloques con mayor distorsión sean más propensos a ser refinados, siempre y cuando no estén lejos y su tamaño sea menor al de otros bloques cercanos. Restando este factor con el mismo factor pero con el bloque en el nivel detalle $l-1$ (mayor resolución) obtenemos la magnitud de la disminución del error de realizarse el refinamiento. Sin embargo, se agrega un factor extra en el cual se considera el tamaño en bytes de los bloques, penalizando aquellos bloques que requieran de introducir muchos bytes en la textura atlas para su refinamiento.

De manera similar, se plantea una fórmula para calcular la pérdida que significaría colapsar un bloque a un nivel de detalle menor. Con ello, se puede determinar cuál es el siguiente bloque a colpasar que minimiza la pérdida, lo que significa el menor aumento del error global por byte. Este valor puede ser calculado con la siguiente fórmula:

$$Perdida = \frac{\frac{distor(l+1)}{dist+diag(l+1)} - \frac{distor(l)}{dist+diag(l)}}{tam(l) - tam(l+1)} \tag{2.3}$$

donde $l+1$ es el siguiente nivel inferior de detalle del bloque. Aquí se quiere encontrar la diferencia de error más pequeña entre un bloque en un nivel de detalle l, y su nivel de detalle $l+1$, penalizando la cantidad de bytes que deben removerse para realiza este colapso.

El cálculo de las prioridades debe realizarse para todos los bloques una vez se tenga los valores calculados de la distancia, diagonal y distorsión. Además, cada vez que un bloque sea refinado o colapsado, su prioridad se cambiará. La diagonal es sencilla de calcular ya que solo depende del tamaño actual del bloque. Sin embargo, la distancia debe volverse a calcular cada vez que que el usuario rote o mueva el volumen y la distorsión debe volverse a calcular cada vez que la función de transferencia cambie. Si se tienen muchos bloques, ambos cálculos pueden ser

costosos. Para optimizar el cálculo de la distancia y la distorsión se implementaron algoritmos paralelos en GPU, lo que permite su aceleración.

Cálculo de la Distancia

Con este kernel es posible realizar la actualización de la distancia de los bloques en una sola llamada a la función cada vez que el usuario cambie la configuración de visualización del volumen. En este caso se lanzan la cantidad necesaria de bloques de ejecución paralelo para tener suficientes hilos, donde cada hilo se encargará de calcular de manera paralela la distancia de un bloque del volumen con respecto al ojo, y el resultado será almacenado en una textura del GPU. Además, se agregó un cálculo para determinar la visibilidad de los bloques, donde dos vértices opuestos de la caja envolvente del bloque son proyectados a espacio de *clipping*, con lo cual se calcula una esfera que envuelva a esta caja y se determina si el bloque está o no dentro del *frustum* de visualización. Debido a que este descarte se realiza en espacio de *clipping*, basta con conocer si existe una intersección entre la caja de dimensión $[-1, 1]^3$ que representa al *frustum* en este espacio, y la esfera envolvente del bloque previamente obtenida. Si no existe dicha intersección, colocamos la distancia al ojo en 0, y al momento de calcular prioridades, se determinará que su prioridad de actualización es 0. Posteriormente, la textura con los valores de la distancia debe ser traída a la memoria principal, donde se almacenará estas distancias en otra estructura residente en esta memoria, en la cual se pueda acceder de manera eficiente a estos valores.

Cálculo de la Distorsión

Para realizar este cálculo es necesario obtener la distorsión que conlleva de representar un bloque en una resolución l, tomando en cuenta los valores del bloque en su máxima resolución. La idea de obtener la distorsión y conocer que tanto difiere visualmente un bloque en un nivel de detalle cualquiera con respecto a su máxima resolución. El valor de distorsión es obtenido con la siguiente ecuación:

$$d(l) = \sum_{v \in V} \frac{||f(s_0(v)) - f(s_l(v))||}{n} \tag{2.4}$$

donde:

- V es el conjunto de vóxeles del bloque.
- n es el número total de vóxeles del bloque en el mayor nivel de detalle.
- f representa la función de transferencia.
- s_l es la interpolación trilineal del volumen en el nivel de detalle l.

- s_0 valores escalares correspondientes a las muestras originales del volumen.

La función de transferencia que se evalúa en esta fórmula representa los colores en el espacio CIEluv, debido a que diferencias de colores en este espacio son más fieles a diferencias visuales para el ojo [54]. El CIEluv representa un espacio de color perceptualmente uniforme, en donde mediciones y distancias en cualquiera de las componentes son igualmente perceptibles por el ojo humano, a diferencia del RGB, en donde el canal G se percibe con mayor detalle. Por ello, se mantiene una copia de la función de transferencia en este espacio para poder acceder a ella de manera eficiente en cualquier momento y poder calcular un valor de distorsión que se ajuste de mejor manera a lo que ve el ojo humano.

Se implementaron dos versiones para el cálculo de la distorsión, una en CPU y la otra en GPU. Debido a que se tiene que calcular valores de distorsión para todos los bloques y en todos sus niveles de detalle, se realizó un algoritmo progresivo, donde se calcula la distorsión de una cantidad de bloques por *frame* que puede ser modificada por el usuario.

En la versión en CPU se hace un recorrido secuencial de los bloques, y para cada nivel de detalle, se itera sobre todos los vóxeles del bloque al mayor nivel de detalle, acumulando la distorsión. Por cada vóxel del bloque al mayor nivel de detalle, se obtiene un vóxel interpolado trilinealmente en el nivel de detalle que se procesa actualmente, valor con el cual se calcula la distorsión. Debido a que todos los vóxeles se encuentran en la memoria principal, el acceso a ellos se realiza de manera inmediata.

Dado que el cálculo es completamente independiente para cada bloque y para cada nivel de detalle, se implementó un algoritmo en GPU para tratar de obtener mejores tiempos de procesamiento. Para ello, se necesita un espacio en la memoria del GPU donde se pueda almacenar los bloques y poder realizar este procedimiento. En este caso no se dispone de la textura atlas, ya que esta está siendo utilizada para el despliegue del volumen, por lo que se necesita crear una textura auxiliar donde almacenar los resultados. La implementación permite al usuario cambiar el tamaño de la textura auxiliar, variando así el cantidad de bloques a los cuales se le puede calcular la distorsión con una sola ejecución del kernel. Para poder realizar este cálculo, se deben tener todos los niveles de detalle de los bloques a procesar en la textura auxiliar, por lo cual se implementaron dos formas de subir esta información a el GPU. La primera consiste en subir todos los niveles de detalle a la textura auxiliar directamente desde la información ya contenida en la memoria principal. Esto implica enviar una cantidad de información considerable al GPU, lo cual puede ser costoso debido al limitado ancho de banda. La otra forma consiste en realizar el submuestreo directamente en el GPU, solo copiando la información de los bloques en su mayor nivel de detalle, y lanzando un kernel paralelo que realice el submuestreo de manera rápida.

Para realizar este submuestreo en el GPU, se utiliza el mismo algoritmo propuesto en la Sección 2.2.2, pero utilizado como memoria de almacenamiento la textura auxiliar. El resultado de este submuestreo no es descargado a la memoria principal, ya que la distorsión se calculará directamente en la memoria del GPU. En este caso se implementa la interpolación trilineal en el GPU, debido a que para almacenar los valores en esta textura auxiliar, OpenGL necesita

tratarla como una imagen en vez de una textura, y OpenGL no posee métodos de acceso con interpolación para las imágenes.

La distorsión se obtiene ejecutando un kernel, donde un bloque de ejecución paralela se encargará de procesar un bloque del volumen en un nivel de detalle específico. El bloque de ejecución paralela se lanza con una configuración de $32 \times 16 \times 1 = 512$ hilos, y se utiliza una memoria compartida de 32×32 para acelerar el acceso a la memoria y la acumulación al error. Esta configuración fue escogida para permitir tener 4 bloques de ejecución paralela por SM, y maximizar su ocupación. Este algoritmo posee 3 etapas como puede verse en la Figura 2.7. Primero, cada hilo se encarga de procesar de manera paralela una columna de vóxeles del bloque del volumen, calculando sus distorsiones y acumulando los resultados. En cada columna, un hilo irá recorriendo todos los vóxeles de la columna del bloque en su máxima resolución, al mismo tiempo que se obtiene una muestra en la misma área del bloque del volumen en la resolución que se está procesando, con lo que se calcula la distorsión de cada vóxel. Si la cantidad de columnas del bloque del volumen es mayor a 32×16 (número de hilos), entonces se irán procesando columnas de manera iterativa hasta obtener todos los resultados, los cuales se irán almacenando en la matriz de memoria compartida. Luego, después de una sincronización, se escogen 32 hilos (hilos cuyo $id.y == 0$) donde cada uno se encargará de totalizar cada una de las filas de la memoria compartida, almacenando los resultados al principio de cada fila de esta memoria. Finalmente, después de otra sincronización, un solo hilo se encargará de totalizar toda la primera fila de la memoria compartida y de obtener el promedio de distorsión final, el cual es almacenado en un arreglo de salida. Una vez terminada la ejecución paralela, el arreglo de salida es trasladado a la memoria principal para poder tener acceso rápido a los valores de distorsión.

Para optimizar aun más los cálculos, se almacenó un arreglo que contiene la opacidad acumulada ($A()$) para cada uno de los valores escalares de la función de trasferencia. La opacidad acumulada para el valor escalar k puede ser obtenida como $A(k) = \sum_{i=0}^{k} F(i)$, donde $F(i)$ representa la opacidad del valor escalar i. Además, se almacena el escalar mínimo ($minBlock$) y máximo ($maxBlock$) que contiene cada bloque en su máxima resolución. De esta manera, podemos calcular si un bloque es transparente de manera eficiente realizando la operación $A(maxBlock) - A(minBlock) < EPS$, como puede observarse en la Figura 2.8. Conociendo si un bloque es transparente podemos evitar el cálculo de su distorsión. Esta optimización es significativa ya que generalmente muchos bloques son transparentes.

Para agilizar el cálculo de la distorsión, algunos autores han propuesto el uso de tablas precalculadas e histogramas [7][51]. Por ejemplo, se podría almacenar una tabla 2D global a todos los bloques en la que se precalcula la diferencia en el espacio CIEluv entre un par de valores escalares, la cual solo se tendría que calcular una sola vez al momento de modificar la función de transferencia. Además, se podría calcular un histograma 2D para cada bloque en cada uno de sus niveles de detalle, almacenando cada par de valores a los cual se les deba calcular su distorsión. Este histograma puede ser calculado una sola vez al inicio del algoritmo, evitando así tener que realizar el recorrido de todos los bloques, mediante el uso de

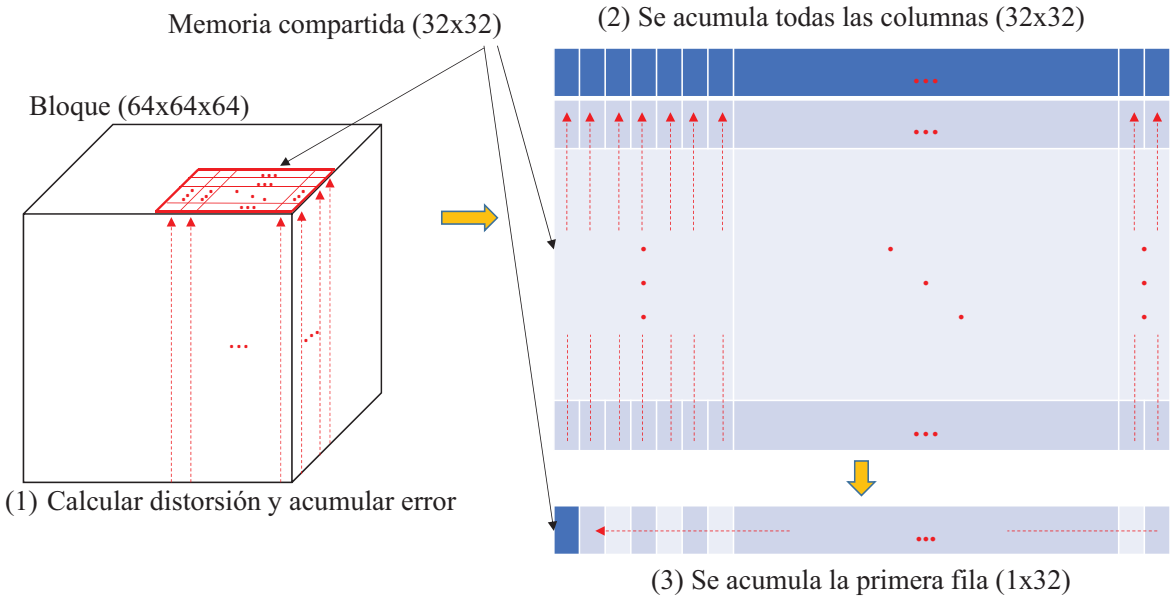


Figura 2.7: Cálculo de la distorsión en GPU. Primero, (1) un bloque de ejecución paralela de 32×16 hilos va recorriendo áreas de un bloque del volumen, donde por cada área, un hilo procesa una columna de vóxeles. Para cada vóxel de la columna, el hilo calcula la distorsión comparando cada vóxel del bloque del volumen en su máxima resolución, con la interpolación trilineal al muestrear el bloque del volumen en el nivel de detalle que se está procesando. Los hilos van acumulando los resultados en la memoria compartida. Posteriormente, (2) 32 hilos sumarán de manera paralela cada una de las columnas de la memoria compartida. Finalmente, (3) un solo hilo se encargará de calcular el resultado final.

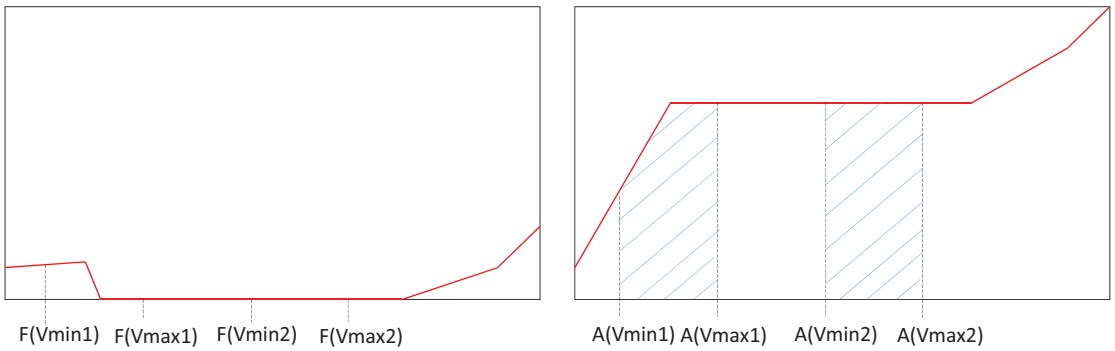


Figura 2.8: Función de opacidad acumulada para optimizar el cálculo de los bloques transparentes. En (a) se tiene la opacidad de la función de transferencia ($F(s)$) y en (b) la acumulada de la opacidad ($A(s)$). Considerando los valores escalares mínimo y máximo de dos bloques, podemos ver que el bloque 1 no es transparente ya que $A(Vmin1) != A(Vmax1)$, mientras que el bloque 2 si lo es debido a que $A(Vmin2) == A(Vmax2)$.

interpolación trilineal, cada vez que se modifique la función de transferencia. Sin embargo, esto solo funciona con volúmenes cuantizados a por ejemplo 8 bits, ya que hacer estas tablas para volúmenes con un mayor rango de valores escalares sería extremadamente pesado. Debido a que en este trabajo se quiere poder utilizar volúmenes de buena resolución sin cuantizarlos y el alto costo en memoria de estas tablas, no se implementaron ninguna de estas optimizaciones, siendo más lento pero con mayor precisión.

2.2.4. Manejo del Atlas

La textura atlas se encargará almacenar el *working set* o conjunto de bloques seleccionados en sus correspondientes niveles de detalle para poder desplegar el volumen. Sin embargo, este *working set* irá cambiando constantemente dependiendo de la prioridad que se le de a los bloques que lo componen y a la necesidad de refinar bloques, por lo que se requiere de constantes actualizaciones. El refinamiento y colapso de bloques conlleva a realizar movimientos dentro de la textura atlas, que se deben realizar de manera eficiente.

En este trabajo se realizó un algoritmo de manejo de la textura atlas tomando algunas ideas propuestas en [6]. Se busca evitar la fragmentación de la textura atlas insertando los bloques de manera ordenada, manteniéndolos en conjuntos del mismo nivel de detalle (y tamaño), y realizando actualizaciones que permitan mantener estos conjuntos unidos siempre (ver Figura 2.9). De esta manera, la textura atlas se organizará por nivel de detalle, donde cada uno de estos conjuntos se encontrarán de manera contigua. Así, la única fragmentación estará ubicada en espacios disponibles entre conjuntos de diferentes niveles de detalle, los cuales podrán ser utilizados de manera eficiente de ser necesario. Esto conlleva a que la textura atlas nunca llegará a un estado de desorden en el que se necesite realizar un proceso de desfragmentación completo que puede ser muy costoso.

Para la implementación de esta idea se definieron diversas estructuras de datos que permiten acelerar el manejo de la textura atlas de manera eficiente. Además, se plantearon algoritmos de movimientos utilizando estas estructuras para realizar el refinamiento y colapso de los bloques.

Estructuras de Datos

En la Figura 2.10 se muestran la diferentes estructuras de datos utilizadas para el manejo de la textura atlas. Primeramente se tiene una estructura tridimensional que almacena información diversa de cada uno de los bloques, la cual se llamó `blockInfo`. Contiene una entrada por cada bloque particionado del volumen, con información como el nivel de detalle actual, distorsión para todos los niveles de detalle, posición actual en la textura atlas, distancia con respecto al ojo, valor escalar máximo, valor escalar mínimo, un apuntador a la posición actual en las colas de prioridad (`queueToRefine` y `queueToCollapse`, flecha (a) de la

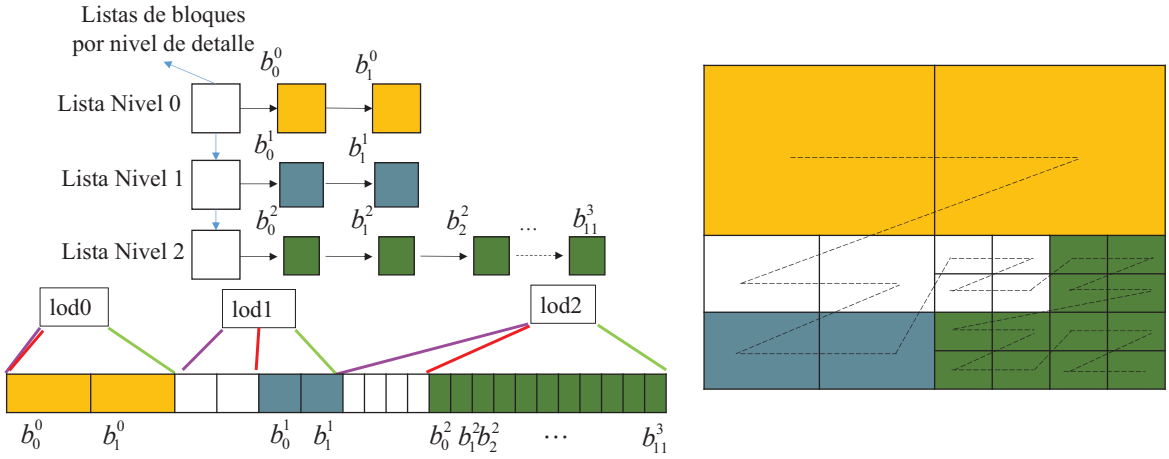


Figura 2.9: Lista por nivel de detalle, donde se observa una configuración posible de una textura atlas en 1D y su equivalente en 2D utilizando el *Morton order*. Para cada nivel de detalle se muestra un apuntador (dirección *Morton*) al primer espacio vacío (morado), al primer espacio lleno (rojo) y último espacio lleno (verde).

Figura 2.10) y un apuntador a la lista de ocupados (`firstEmptyFirstFill`, flecha (b) de la Figura 2.10).

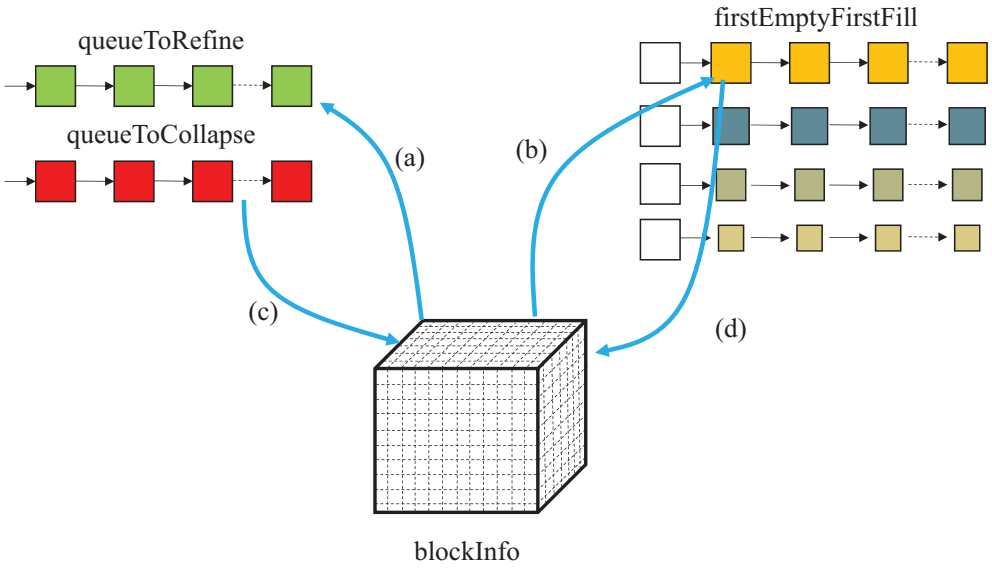


Figura 2.10: Estructuras de datos utilizadas para el manejo eficiente de la textura atlas.

Por otro lado, se tienen dos colas de prioridad que ordenarán los bloques a ser refinados y colapsados. Estás colas son implementadas como *multiset*, las cuales funcionan como un par clave-valor, ordenadas según la clave. En este caso, las claves corresponden a la ganancia por byte del bloque para ser refinado en la cola de refinamientos (`queueToRefine`), y la

pérdida por byte del bloque para ser colapsado en la cola de colapsos (`queueToCollapse`). El valor almacenado en cada uno de los elementos de la cola es la posición 3D del bloque en la matriz `blockInfo` (flecha (c) de la Figura 2.10). La cola de refinamientos es ordenada de mayor a menor ganancia, mientras que la cola de colapsos es ordenada de menor a mayor pérdida. De esta manera, por cada *frame* se quiere siempre refinar aquel bloque que más reduce el error global (mayor ganancia), y en caso de necesitar colapsar, se selecciona aquel bloque que menos aumente el error global (menor pérdida). Refinar en este orden, representa una estrategia *greedy* que nos acerca a la configuración óptima del *working set* [63] [64].

Además, para permitir movimientos eficientes dentro de la textura atlas se asume el siguiente conjunto de invariantes: (1) por cada nivel de detalle, en el espacio 1D primero se encuentran los bloques vacíos de manera contigua, y posteriormente se encuentran los bloques ocupados y (2) los niveles de detalle están ordenados en 1D desde el mayor nivel de detalle hasta el menor nivel de detalle.

Para ayudar a mantener estas invariantes, se tendrá un arreglo de listas por cada nivel de detalle, llamada `firstEmptyFirstFill`, la cual se presenta con mayor detalle en la Figura 2.9. Esta estructura es un arreglo con una posición por cada uno de los niveles de detalle disponible, y por cada entrada, contiene la dirección *Morton* de la primera posición vacía (línea morada), de la primera posición llena (línea roja), de la última posición llena (línea verde), y una lista ordenada de los bloques que se encuentran en ese nivel de detalle. La dirección de la última posición llena del nivel de detalle l siempre será igual a la dirección de la primera posición vacía del nivel de detalle $l + 1$, por lo que este arreglo va a hacer una correspondencia con todas las posiciones de la textura atlas. Además, cada nivel de detalle contiene una lista con los bloques que se encuentran representados en ese nivel de detalle. Debido a que se direcciona con el *Morton order*, esta estructura hace una referencia lineal de la textura atlas, donde se tendrán primero los espacios vacíos por cada nivel de detalle y posteriormente los espacios llenos. Cada una de las posiciones poseen una estructura que indica la dirección *Morton* donde se encuentra ese bloque en la textura atlas, y la dirección 3D del bloque en la estructura `blockInfo` (flecha (d) de la Figura 2.10). En la figura la textura atlas se representa en 2D, pero esta misma correspondencia se podría hacer con una textura 3D. La importancia de usar estas listas es que nos permiten manejar la organización de la textura en un espacio lineal, en vez de en el espacio 3D, simplificando considerablemente la complejidad del problema. Esta es la única estructura adicional propuesta por nuestro trabajo para poder mantener las invariantes mencionadas, y su tamaño depende de la cantidad de bloques necesarios para subdividir el volumen (*O(N)*).

Algoritmos de Refinamientos en el Atlas

El refinamiento de un bloque consiste en subir el nivel de detalle de un bloque que se encontraba representado en el nivel de detalle l (ocupando b bytes en la textura atlas), hacia el nivel de detalle $l - 1$ (ocupando ahora $b * 8$ bytes en la textura atlas). El algoritmo básico para el refinamiento de bloques por *frame* puede observarse en el Código 2.8. Los procedimientos de

este algoritmo se irán explicando con mayor detalle a lo largo de esta sección.

```
bool verificarEspacio(int tamañoBloque, int LoD)
{
  puedoMover = false;

  if(EspacioEnAtlas < EspacioNecesitado) puedoMover = false;
  else if(verificarEspacioEnLoD(tamañoBloque, LoD)) puedoMover = true;
  else if(moverALaDerecha(tamañoBloque * 8, LoD - 1)) puedoMover = true;
  else if(moverALaIzquierda(tamañoBloque, LoD)) puedoMover = true;


  return PuedoMover;

}

void refinar()
{
  bloquesRefinar = NUM_REFINAR_POR_CUADRO;

  while(bloquesRefinar > 0 && queueToRefine.size() > 0){
    bloqueARefinar = queueToRefine.begin();
    tamaño = bloqueARefinar.tamaño * 8; //nuevo tamaño del bloque
    nuevoLoD = bloqueARefinar.LoD - 1;  //vamos a refinar

    hayEspacio = verfificarEspacio(tamaño, nuevoLoD) //verificamos si puedo refinar

    if(!hayEspacio)
    {
      colapso(); //colapsamos uno o varios bloques
      hayEspacio = verfificarEspacio(tamaño, nuevoLoD) //verificamos otra vez
    }

    if(hayEspacio){
      refinarElBloque();  //finalmente, si hay espacio, refinar
    }

    bloquesRefinar = bloquesRefinar - 1;
  }
}
```

Código 2.8: Algoritmo básico para el refinamiento de bloques.

Primeramente, de manera iterativa se intenta refinar una cantidad de bloques fija por *frame*, mientras haya bloques para refinar. En caso de que haya bloques para refinar, lo primero es buscar si la textura atlas tiene suficiente espacio para almacenar el bloque en su nuevo nivel de detalle. Si hay espacio, pueden ocurrir 3 cosas: (1) que haya espacio libre en el nuevo nivel de detalle como para colocar el bloque nuevo, (2) no haya suficiente espacio pero moviendo bloques a la derecha se puede generar, o (3) no haya suficiente espacio pero moviendo bloques a la izquierda se puede generar. Si por el contrario la textura atlas no tiene suficiente espacio para almacenar el bloque en su nuevo nivel de detalle, se procede a colapsar uno o varios bloques hasta liberar el espacio necesario para poder subir de nivel de detalle el bloque.

Hay que tomar en cuenta, que todo movimiento de la textura atlas requiere de su actualización en el GPU, además de modificar la posición del bloque en la textura de índices que es utilizada para el despliegue. Esto conlleva a que se requiere llamar de manera constante la función

`glTexSubImage3D` de OpenGL para realizar las actualizaciones de las texturas.

A continuación se describen los movimientos principales que se deben realizar para poder mantener organizada la textura atlas:

- **Refinado**: en la Figura 2.11 podemos observar los movimientos de bloques necesarios para realizar un refinamiento. La figura muestra la representación de la operación en una textura atlas tanto en 1D (arriba y abajo) como en 2D (izquierda y derecha). Sin embargo, el mismo principio puede ser aplicado a 3D debido a la correspondencia obtenida con el *Morton orden*. El bloque de nivel de detalle l (azul) a refinar (bloque con borde rojo) debe ser insertado al principio de la lista de bloques de nivel de detalle $l-1$ (amarillo). Si este bloque no es el primero en la lista de bloques de nivel de detalle l, entonces el primero de la lista tomará su posición como lo muestra la figura (bloque azul con borde morado). Por lo tanto, a lo sumo se necesitará actualizar $b*8+b$ bytes de la textura atlas, siendo b el tamaño del bloque a actualizar en el nivel de detalle l.

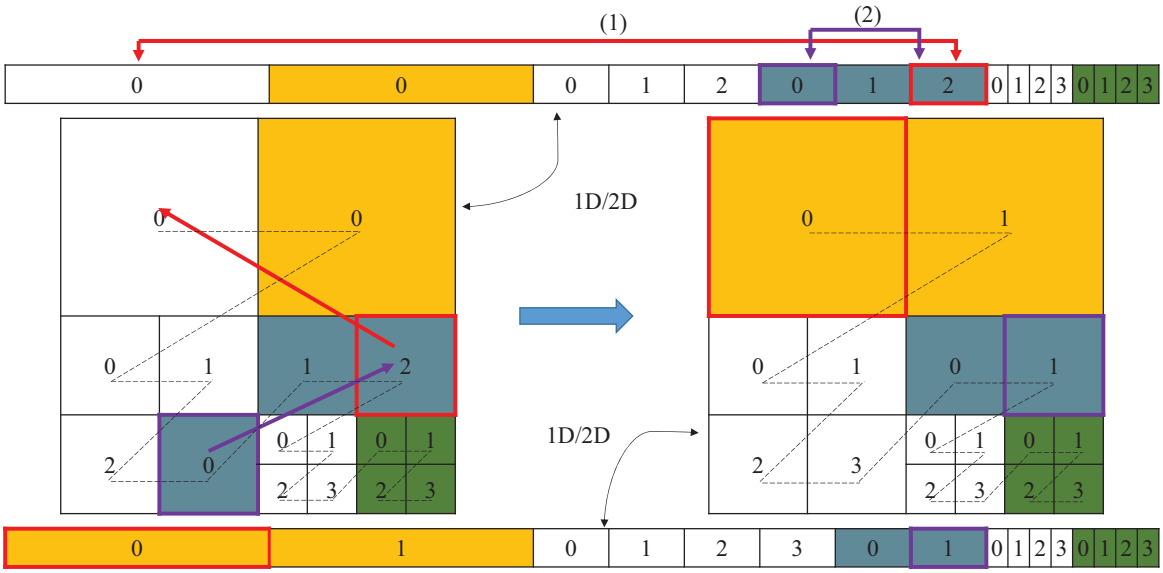


Figura 2.11: Refinamiento de bloques. En la parte superior (1D) e izquierda (2D) se encuentra la representación de la textura atlas antes de realizar la operación, y en la parte inferior (1D) y derecha (2D), se encuentra el resultado de la operación. En el refinamiento un bloque sube de nivel de detalle l a $l-1$ (en este caso de azul, con borde rojo, a amarillo), y posiblemente otro bloque del nivel de detalle l tiene que tomar la posición del bloque que se va a refinar (en este caso el bloque azul, con borde morado).

Al refinar el bloque, cambia su nivel de detalle, por lo que debe retirarse de las colas `queueToRefine` y `queueToCollapse`, calcular su nueva prioridad y volverse a insertar en las colas con su nueva prioridad. Además, se debe actualizar la estructura `firstEmptyFirstFill`, modificando los apuntadores y las listas correspondientes, y finalmente toda la información del bloque contenida en `blockInfo` debe ser

actualizada.

Si no hay espacio libre suficiente en el siguiente nivel de detalle para poder almacenar el bloque a refinar, es necesario ver si se puede reorganizar la textura atlas para obtener este espacio (movimientos hacia la izquierda o movimientos hacia la derecha), o bajando el nivel de detalle de algunos bloques para crear mayor espacio (colapsos).

- **Movimiento hacia la izquierda**: el objetivo de los movimientos hacia la izquierda buscan verificar si hay espacio suficiente a la izquierda para mover los bloques de tal manera que permitan realizar un refinamiento. Se dice que se mueve hacia la izquierda debido a que se hacen desplazamientos en esta dirección en la representación 1D. El algoritmo básico se muestra en el Código 2.9. Si el bloque se quiere refinar del nivel de detalle l al $l-1$, se comprueba recursivamente desde el nivel $l-2$ hacia la izquierda. Si se determina que no hay espacio suficiente a la izquierda para almacenar el bloque en la nueva resolución, no se realizará ningún movimiento. En caso de que en alguna llamada recursiva se comprueba que hay espacio suficiente, al volver de la recursión se realizan movimientos de bloques desde el final de la lista hacia el principio para todos los niveles de detalle, liberando el espacio suficiente para poder realizar el refinamiento.

```
bool moverIzquierda(int tamañoRequerido, int actualLoD)
{
  if(actualLoD < 0) return false; //si hemos verificado todos los niveles, y no hay ↩
      espacio, retornar falso

  nextLoD = actualLoD - 1;

  if(espacioEnEsteNivel(tamañoRequerido, actualLoD) || moverIzquierda(tamañ↩
      oRequerido * 8, nextLoD)) //si hay espacio en este nivel, o se creo suficiente↩
       espacio al mover hacia la izquierda los siguientes niveles
  {
    huboMovimiento = false;

    if(hayBloquesEnNivel(actualLoD)) //verificar si hay bloques
    {
      moverUltimoBloqueAlPrincipioLista(actualLoD); //mover un bloque desde el final↩
           de la lista hasta el principio
      huboMovimiento = true;
    }

    if(!huboMovimiento){
    {
      moverApuntadores(); //es posible que no haya más bloques que mover, pero si se↩
           necesiten mover apuntadores
    }
  }
}
```

Código 2.9: Algoritmo de movimientos hacia la izquierda.

En la Figura 2.12 se muestran algunos ejemplos de movimientos hacia la izquierda. La figura muestra la representación de la operación en una textura atlas tanto en 1D (arriba y abajo) como en 2D (izquierda y derecha). Sin embargo, el mismo principio puede

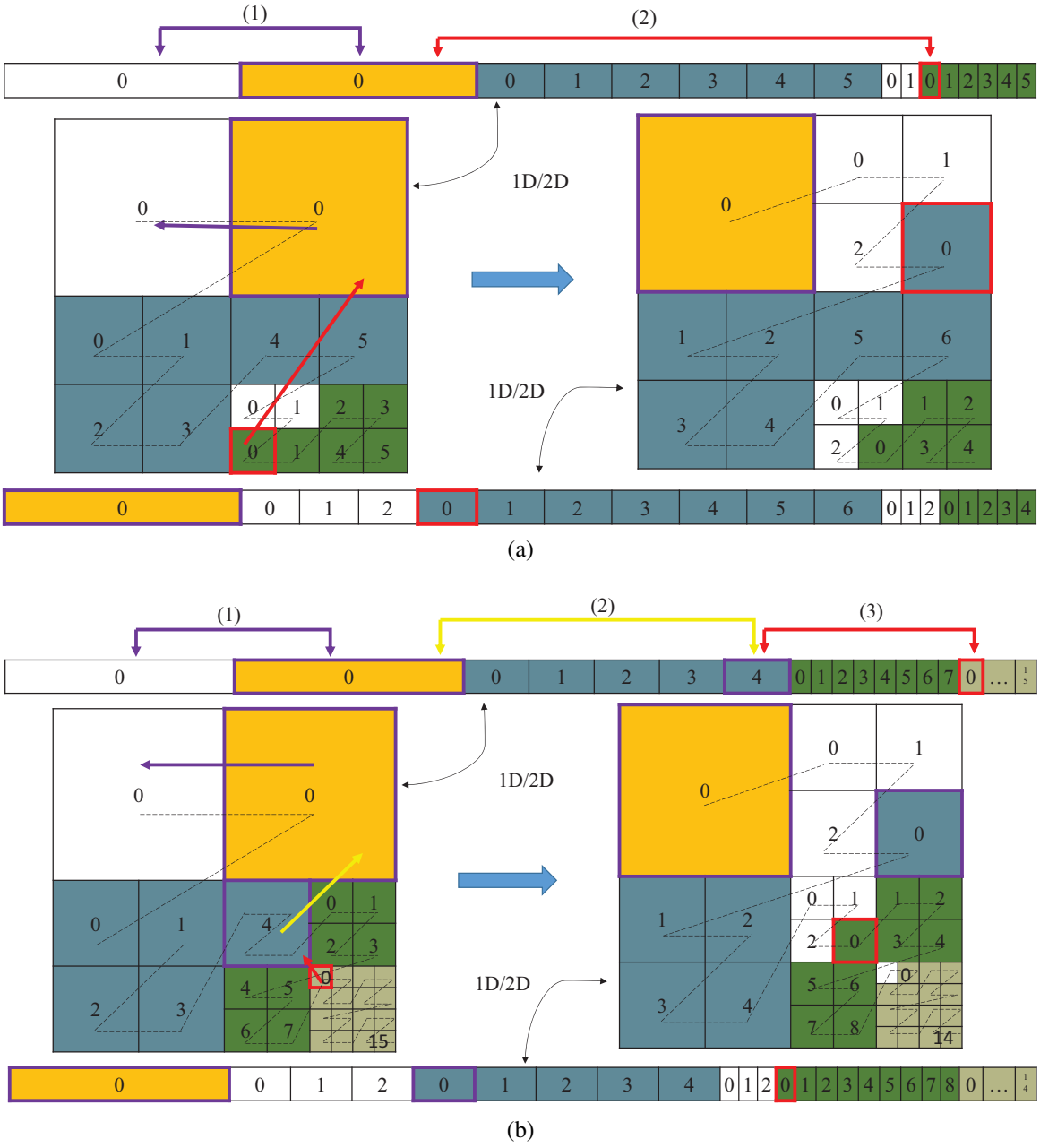


Figura 2.12: Movimientos hacia la izquierda, mostrando el mejor caso (a), y el peor caso (b). En la parte superior (1D) e izquierda (2D) se encuentra la representación de la textura atlas antes de realizar la operación, y en la parte inferior (1D) y derecha (2D), se encuentra el resultado de la operación. Para ambos casos el bloque bordeado por rojo sube de nivel de detalle, mientras los bloques con borde morado deben moverse para permitir este refinamiento y mantener las invariantes del algoritmo.

ser aplicado a 3D debido a la correspondencia obtenida con el *Morton orden*. En el mejor caso solo se deben mover $b * 64$ bytes hacia la izquierda (Figura 2.12a), siendo este el tamaño del bloque en el nivel $l - 2$. La figura muestra como el bloque amarillo

con borde morado se mueve hacia la izquierda, para generar el espacio suficiente para que el bloque verde con borde rojo se refine. En el peor caso, se deben realizar movimientos iterativos hasta llegar al máximo nivel de detalle, por lo que a lo sumo se deben mover $\sum_{i=0}^{l-2} 1 = l$ bloques, o $\sum_{i=0}^{l-2} 8^{maxLoD-i}$ bytes de memoria (Figura 2.12b), lo que significaría mover un bloque de cada nivel de detalle superior al nivel de detalle del bloque al cual se requiere refinar. En la figura se muestra como los bloques con borde morado van desplazándose hacia la izquierda para generar el espacio para que el bloque con borde rojo se refine. Primero se mueve el bloque amarillo (1), luego el azul (2), y finalmente se refina el bloque con borde rojo (3).

- **Movimiento hacia la derecha**: el objetivo de los movimientos hacia la derecha buscan verificar si hay espacio suficiente a la derecha para mover los bloques de tal manera que permitan realizar un refinamiento. Se dice que se mueve hacia la derecha debido a que se hacen desplazamientos en esta dirección en la representación 1D. El algoritmo básico se muestra en el Código 2.10. Si el bloque se quiere refinar desde el nivel de detalle l al $l-1$, se comprueba desde ese nivel de detalle hasta el nivel $l = minLoD - 1$ recursivamente, ya que en el último nivel no habrá espacio a la derecha. Si se determina que la suma de todos los espacios hacia la derecha son menores al espacio requerido, no se realiza ningún movimiento. En caso de que en alguna llamada recursiva se comprueba que hay espacio suficiente, al volver de la recursión se realizan movimientos de bloques desde el principio de la lista hacia el final, liberando el espacio suficiente para poder realizar el refinamiento.

```
bool moverDerecha(int tamañoRequerido, int actualLoD)
{
  if(actualLoD > minLoD) return false; //si hemos verificado todos los niveles, y no↩
       hay espacio, retornar falso

  nextLoD = actualLoD + 1;

  if(espacioEnEsteNivel(tamañoRequerido, actualLoD)) return true; // si hay espacio ↩
      en este nivel, podemos mover los bloques del nivel de detalle anterior a este ↩
      espacio

  espacioEnNivel = firstEmptyFirstFill[actualLoD].firstFilled - firstEmptyFirstFill[↩
      actualLoD].firstEmpty;
  espacioRequerido = tamañoRequerido - espacioEnNivel;

  if(moverDerecha(espacioRequerido, nextLoD))
  {
    espacioMovido = 0;

    while(espacioMovido < espacioRequerido) //mover bloques hasta que se obtenga el ↩
        espacio requerido
    {
      moverPrimerBloqueAlFinalLista(actualLoD); //mover un bloque desde el principio↩
           de la lista hasta el final

      espacioMovido += tamañoEnNivel(actualLoD);
    }

    if(espacioMovido < tamañoRequerido){
    {
```

```
      moverApuntadores(); //es posible que no haya más bloques que mover, pero si se↩
           necesiten mover apuntadores
    }
  }
}
```

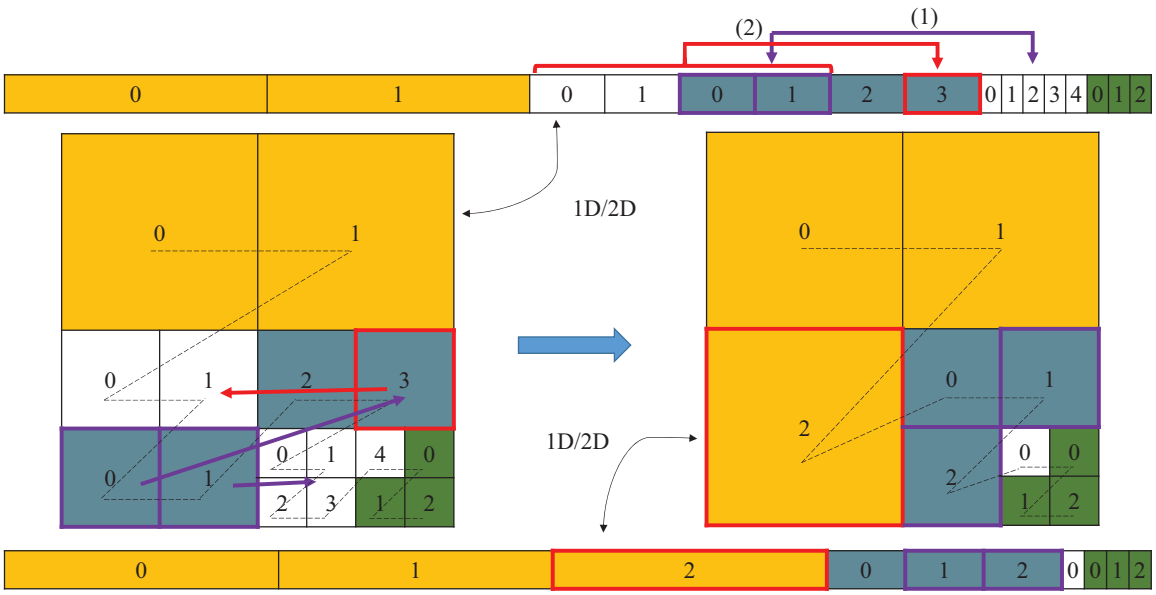
Código 2.10: Algoritmo de movimientos hacia la derecha.

En la Figura 2.13 se muestran algunos ejemplos de movimientos hacia la derecha. La figura muestra la representación de la operación en una textura atlas tanto en 1D (arriba y abajo) como en 2D (izquierda y derecha). Sin embargo, el mismo principio puede ser aplicado a 3D debido a la correspondencia obtenida con el *Morton orden*. En el mejor caso solo se deben mover b bytes hacia la derecha (Figura 2.13a), siendo este el tamaño del bloque en el nivel l. Aquí los bloques azules con borde morado deben desplazarse primero hacia la derecha para crear espacio suficiente para el bloque azul con borde rojo pueda ser refinado. En el peor caso, se deben realizar movimientos iterativos hasta llegar al menor nivel de detalle, por lo que a lo sumo se deben mover $8 + (7/8) * \sum_{i=l+1}^{minLoD-1} 8^{i-l+1}$ bloques, o $b * 8 + 7 * \sum_{i=l+1}^{minLoD-1} b$ bytes de memoria (Figura 2.13b). El ejemplo muestra como los bloques azules y verdes con borde morado deben ser desplazados hacia la derecha, de manera de dejar espacio al bloque azul con borde rojo espacio para ser refinado. Primero se mueven los bloques verdes (1), luego los bloques azules (2), y finalmente se realizar el refinamiento (3).

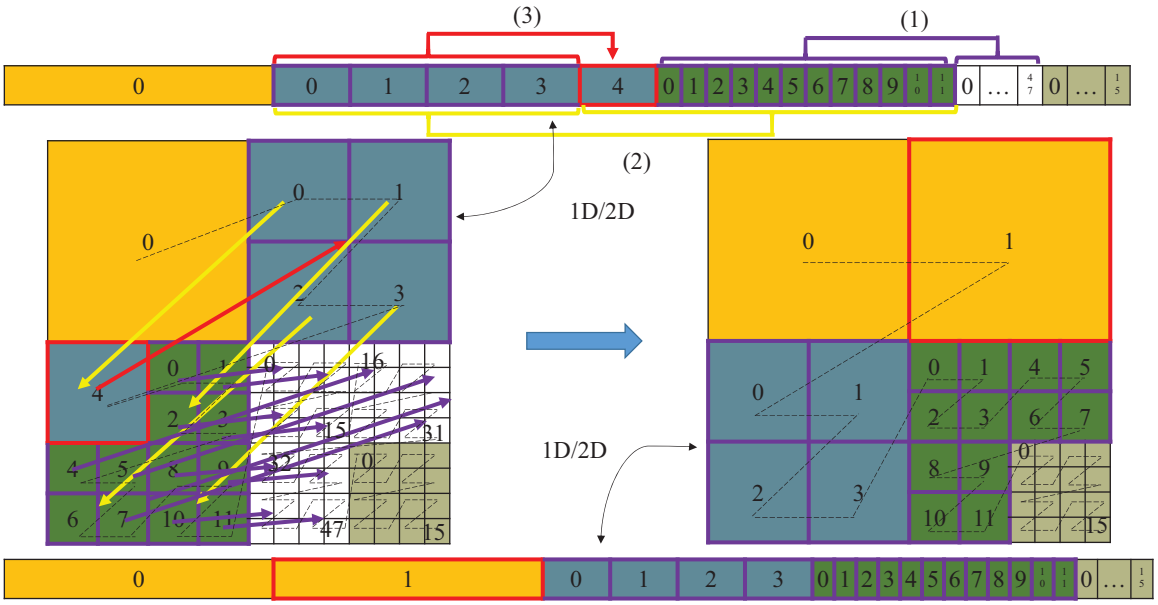
- **Colapso**: el colapso de un bloque siempre podrá ser realizado, ya que se pasaría a un bloque de nivel de detalle l a un nivel de detalle $l + 1$, pasando de ocupar b bytes, a $b/8$ bytes. Sin embargo, un colapso reduce la resolución del área del volumen correspondiente al bloque colapsado, bajando el nivel de detalle promedio en el *working set* y aumentando el error, por lo que nunca se debe realizar a menos que sea necesario. La aplicación se verá en la necesidad de colapsar en el momento en que el bloque con mayor prioridad no tenga espacio para ser refinado, por lo que un colapso podría generar el espacio requerido. El bloque a colpasar será seleccionado de la primera posición del $multiset$ `queueToCollapse`. No obstante, podría ocurrir que al colapsar un bloque, para luego refinar otro, el error de representar el volumen con este nuevo *working set* sea mayor al error que se tenía previamente. Para evitar esto, se almacena el error global acumulado de representar el volumen con el *working set* actual, y un colapso solo se realizará si esto conlleva a que el error global disminuya después del refinamiento posterior.

  Además, al momento de necesitar espacio para refinar un bloque, puede que un solo colapso no sea suficiente. Por ello, se realizó un algoritmo iterativo en el cual se van tomando del $multiset$ `queueToCollapse` posibles bloques a refinar, hasta crear un espacio suficiente para poder realizar el refinamiento. Igualmente, se va calculando el aumento del error global con cada uno de estos posibles colapsos, y solo se realizará si al final de todos los movimientos el error global va a disminuir.

  En la Figura 2.14 se puede observar el funcionamiento básico del colapso. La figura



(a)



(b)

Figura 2.13: Movimientos hacia la derecha, mostrando el mejor caso (a), y el peor caso (b). En la parte superior (1D) e izquierda (2D) se encuentra la representación de la textura atlas antes de realizar la operación, y en la parte inferior (1D) y derecha (2D), se encuentra el resultado de la operación. Para ambos casos el bloque bordeado por rojo sube de nivel de detalle, mientras los bloques con borde morado deben moverse para permitir este refinamiento y mantener las invariantes del algoritmo.

muestra la representación de la operación en una textura atlas tanto en 1D (arriba y abajo) como en 2D (izquierda y derecha). Sin embargo, el mismo principio puede ser aplicado a 3D debido a la correspondencia obtenida con el *Morton orden*. El bloque a colapsar debe ser retirado de la lista de bloques en el nivel l de la estructura `firstEmptyFirstFill`, e insertado al principio de la lista de bloques en el nivel $l+1$. En caso de que el bloque a colapsar no sea el último del nivel l, es necesario que otro bloque tome su lugar para mantener las invariantes, como lo muestra la Figura 2.14. Aquí, el bloque azul con borde rojo será colapsado, y para mantener la continuidad por nivel de detalle, el bloque azul con borde morado toma el lugar del bloque colapsado. Por ello, a lo sumo se deben actualizar $b + b/8$ bytes en la textura atlas al momento de realizar un colapso, debido a que en el peor caso solo se tendría que mover un bloque adicional para mantener las invariantes.

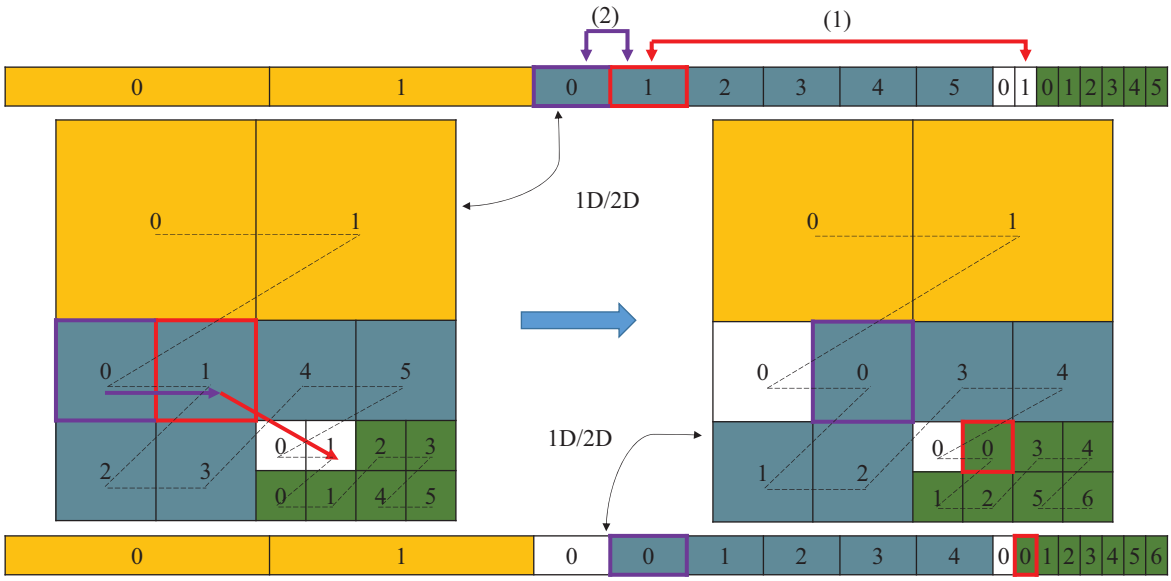


Figura 2.14: Colapso de bloques. En la parte superior (1D) e izquierda (2D) se encuentra la representación de la textura atlas antes de realizar la operación, y en la parte inferior (1D) y derecha (2D), se encuentra el resultado de la operación. El colapso de un bloque baja el nivel de detalle l a $l+1$, y y posiblemente otro bloque del nivel de detalle l tiene que tomar la posición del bloque que se va a colpasar.

Al colapsar el bloque, cambia su nivel de detalle, por lo que debe retirarse de las colas `queueToRefine` y `queueToCollapse`, calcular su nueva prioridad y volverse a insertar en las colas. Además, se debe actualizar la estructura `firstEmptyFirstFill`, modificando los apuntadores y las listas correspondientes, y finalmente toda la información del bloque contenida en `blockInfo` debe ser actualizada.

Hay que tomar en cuenta que al colapsar solamente cuando el error acumulado disminuye puede llevar a obtener un *working set* que no es el óptimo. Esto se debe a que es posible

que se necesite un conjunto de colapsos que aumenten el error global, para poder llegar en algún momento a un refinamiento que disminuya de manera considerable este mismo error. Por ello, el algoritmo, tratando de disminuir la función del error global, puede llegar a converger en mínimos locales que no obtengan el *working set* óptimo.

2.2.5. Reducción de artefactos

El uso de la jerarquía por bloques genera que zonas adyacentes del volumen puedan estar representadas con diferentes niveles de resolución, por lo que la cantidad de vóxeles que un bloque utiliza para representar dicha zona varía. Debido a que las muestras que se toman dentro del volumen no necesariamente se encuentran alineadas a la data, es necesario el uso de la interpolación trilineal para el cálculo correcto de la muestra. En el caso de que este proceso se realice dentro del área válida de interpolación del bloque (ver representación 2D en la Figura 2.15), se puede utilizar la interpolación trilineal provista por el hardware gráfico. Sin embargo, cuando se toman una muestra entre dos bloques se requiere de la realización de un proceso manual para el cálculo de la muestra interpolada, ya que estos bloques poseen distintas resoluciones, y se hace necesario algún proceso que balancee estas muestras dependiendo de las resoluciones de los mismos. Además, el uso de la textura atlas para el almacenamiento del *working set* genera que bloques vecinos del volumen puedan encontrarse distanciados en la memoria, por lo que se deben tomar muestras de diferentes zonas de esta textura. Si no se considera el nivel de detalle en el proceso de interpolación, se generan artefactos visuales en el muestreo de las áreas de las fronteras entre los bloques, que pueden ser perjudiciales para la calidad visual, por lo que es importante reducir estos artefactos.

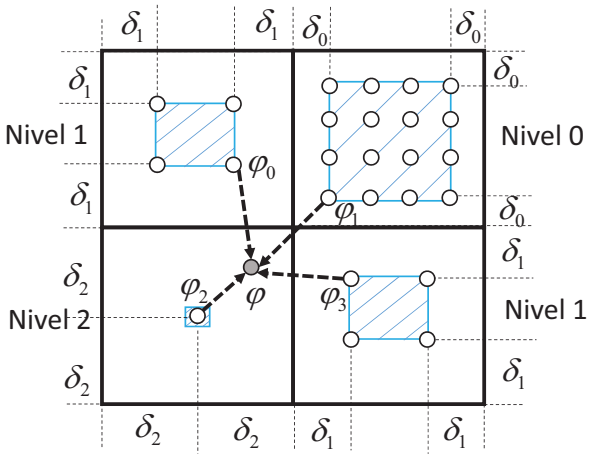


Figura 2.15: Detalles de la interpolación entre bloques. Se muestra el área válida de interpolación dentro de un bloque (rallada) y el espacio (δ_i) donde se debe realizar interpolación entre bloques, el cual dependerá del nivel de detalle del mismo. Se utiliza una representación 2D para simplificar su comprensión.

La reducción de artefactos en las fronteras de este trabajo se implementó utilizando la interpolación entre bloques propuesta por Ljung et al. [1]. La idea es tener dos casos básicos para la interpolación: interna al bloque, y entre bloques. El primer caso ocurre cuando se toma muestras del volumen en el área válida de interpolación, mientras que el segundo ocurre cuando se toman muestras fuera de esta área. En el caso del muestreo entre bloques, el valor de la muestra interpolada φ es calculada como la suma ponderada de las muestras tomadas en el punto más cercano en los bloques vecinos φ_b, utilizando la Ecuación 2.5. En la Figura 2.15 se evidencia en 2D como la muestra φ (color gris) es obtenida como un promedio ponderado de las muestras de los 4 bloques vecinos φ_b (en 3D serían 8 vecinos).

$$\varphi_{=}\frac{\sum_{b=1}^{8}\omega_b\varphi_b}{\sum_{b=1}^{8}\omega_b} \tag{2.5}$$

A la hora de tomar una muestra, lo primero es determinar si se necesita realizar interpolación interna del bloque o entre bloques. Si se asume que el tamaño de los bloques es unitario, entonces el área válida para el muestreo de un bloque de nivel de detalle l tiene un desplazamiento de $\delta(l) = 1/2^{1+(max\_level-l)}$ con respecto a los límites del bloque. En la Figura 2.15 podemos observar una representación de 4 bloques en 3 distintos niveles de detalle, donde el zona rayada representa el área válida de interpolación. A mayor nivel de detalle, mayor es esta área. Con esto, si se tiene una coordenada normalizada $c \in [0,1]^3$ dentro de un bloque, podemos obtener muestras válidas dentro del mismo con la siguiente fórmula:

$$c' = C_{\delta}^{1-\delta}(c) \tag{2.6}$$

donde $C_a^b(x)$ limita el valor x al intervalo $[a,b]$, y con lo cual se puede realizar muestreos válidos dentro de los bloques y desplegar el volumen solo realizando interpolación interna de los bloques.

Ahora, profundizando en los cálculos necesarios para obtener la interpolación entre bloques, primero tenemos que determinar la posición normalizada central de los 8 vecinos entre los cuales se va a calcular la interpolación. Suponemos que el volumen se encuentra definido en el rango $(0,0,0)$ a (N_r, N_s, N_t), donde N_p es el número de bloques en el eje p. Además, suponemos que las coordenadas locales a los 8 vecinos r,s,t, se encuentran en el rango $[-1/2, 1/2]^3$ (ver Figura 2.16). El centro de la vecindad se calcula tomando las coordenadas globales del volumen r_g, s_g, t_g, y haciendo de manera análoga para todos los ejes $r_0 = \lfloor C_0^{N_r-1}(r_g - 0,5) \rfloor$. La coordenada (r_0, s_0, t_0), indicará la coordenada del bloque frontal inferior izquierdo de los 8 vecinos, y las coordenadas relativas a esta vecindad pueden obtenerse análogamente para todas las dimensiones como $r = r_g - r_0 - 1$.

Una vez obtenidas estas coordenadas, se toma una muestra φ_b de cada uno de los bloques vecinos, haciendo ajustes en las coordenadas locales, y utilizando la interpolación interna de bloques. Debido a que cada bloque puede encontrarse en un nivel de resolución distinto,

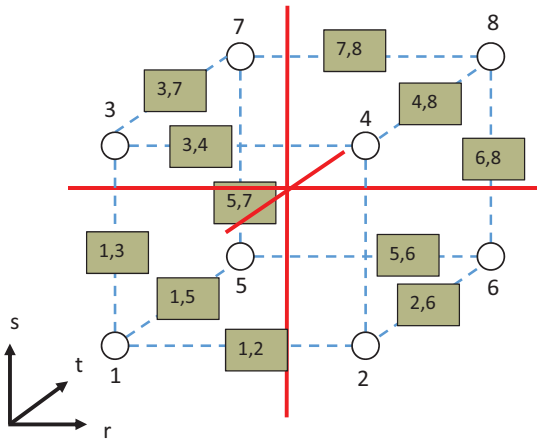


Figura 2.16: Se observa en 3D el sistema de coordenadas locales que se utiliza para realizar el cálculo de la interpolación entre bloques, donde cada nodo (del 1 al 8) representa un bloque a ser interpolado.

es necesario darle un peso ω_b a cada uno de los bloques. Para ello, se calcula un peso para cada una de las aristas de vecindad (cajas verdes en la Figura 2.16) entre bloques vecinos y posteriormente se utilizan estos pesos de aristas para determinar el peso de cada bloque.

Para manejar de manera más rápida las aristas de las vecindades, se introducen 3 conjuntos de etiquetas por cada uno de los ejes r, s, t, los cuales son:

$$\begin{aligned} E_r &= \{(1,2),(3,4),(5,6),(7,8)\} \\ E_s &= \{(1,3),(2,4),(5,7),(6,8)\} \\ E_t &= \{(1,5),(2,6),(3,7),(4,8)\} \end{aligned} \tag{2.7}$$

En el cálculo de los pesos de cada arista de vecindad $e_{i,j}$ (donde el par i, j pertenece a uno de los conjuntos por aristas), se utilizó la interpolación de distancia máxima expuesta por Ljung et al. [1]. En ella se realiza una interpolación lineal en un segmento sobre toda la distancia entre los bloques vecino, y puede expresarse como:

$$e_{i,j}(p) = C_0^1((p+\delta_i)/(\delta_i+\delta_j))) \tag{2.8}$$

Con estos pesos de aristas $e_{i,j}$, se determinan los pesos de los bloques ω_b de la siguiente manera:

$$
\begin{aligned}
\omega_1 &= (1{,}0f - e_{1,2}) * (1{,}0f - e_{1,3}) * (1{,}0f - e_{1,5}) \\
\omega_2 &= e_{1,2} * (1{,}0f - e_{2,4}) * (1{,}0f - e_{2,6}) \\
\omega_3 &= (1{,}0f - e_{3,4}) * e_{1,3} * (1{,}0f - e_{3,7}) \\
\omega_4 &= e_{3,4} * e_{2,4} * (1{,}0f - e_{4,8}) \\
\omega_5 &= (1{,}0f - e_{5,6}) * (1{,}0f - e_{5,7}) * e_{1,5} \\
\omega_6 &= e_{5,6} * (1{,}0f - e_{6,8}) * e_{2,6} \\
\omega_7 &= (1{,}0f - e_{7,8}) * e_{5,7} * e_{3,7} \\
\omega_8 &= e_{7,8} * e_{6,8} * e_{4,8}
\end{aligned}
\tag{2.9}
$$

Finalmente, los pasos generales para el cálculo de la interpolación entre bloques pueden resumirse en:

1. Determinar la posición de la vecindad de 8 bloques $r0$, $s0$, $t0$, y determinar el sistema de coordenadas locales r,s,t.
2. Tomar muestras φ_b de los bloques utilizando interpolación interna del bloque.
3. Calcular los pesos de las aristas de vecindad $e_{i,j}$.
4. Determinar los pesos de los bloques ω_b de 3 pesos de arista de vecindad.
5. Obtener la muestra φ, como la suma ponderada de ω_b y φ_b.

Debido al aumento en la cantidad de muestreos necesarios para realizar la interpolación, la disminución en el rendimiento de la aplicación al utilizar este método de despliegue es considerable. Por ello, también se implementó el despliegue sin realizar interpolación con los bloques vecinos, tomando en cuenta únicamente los vóxeles internos del bloque, y limitando el área de muestreo con la Ecuación 2.6.

Capítulo 3

Pruebas y Resultados

En este capítulo se muestran los resultados cualitativos y cuantitativos obtenidos a través de la prueba de las diferentes implementaciones realizadas en este trabajo de grado. En la Sección 3.1 se exponen los resultados obtenidos de la comparación de desempeño de las diferentes implementaciones del *ray casting* de volúmenes paralelo. Luego, la Sección 3.2 contiene las pruebas realizadas a la implementación del visualizador de volúmenes multi-resolución.

Para poder comparar los resultados obtenidos en las diferentes pruebas realizadas se utilizaron proporciones. Debido a que estas proporciones pueden interpretarse de varias maneras, es bueno aclarar como fueron usadas en este trabajo. Si se tienen dos valores A y B que pudieran medir tiempo, se expresó la relación entre estos dos valores como $C\times$, donde $C = A/B$. Si $A/B \geq 1$, entonces $C\times$ expresa que B es C veces más rápido que A. Por ejemplo, si tenemos que un proceso e tarda 35 segundos, y otro proceso f tarda 15 segundos, entonces decimos que f es $2,33\times$ veces más rápido que e. En la práctica esto indica que e tarda $tiempo_f + tiempo_f * 1,33$, ya que tarda $133\,\%$ (20 segundos) más que lo que tarda f. Esto se puede prestar a confusión para los casos en que $1 \leq A/B \leq 2$, donde una proporción de $1{,}33\times$ quiere decir que un proceso dura $33\,\%$ más de lo que dura el otro, y una proporción de $1{,}0\times$ indica que ambos procesos duran lo mismo ($0\,\%$ más de lo que duró el otro). Para los casos en que $A/B < 1$, simplemente se invierte la división (donde $B/A \geq 1$), y decimos que A es más rápido que B.

3.1. Pruebas Sobre el *Ray Casting*

En esta sección se presentan las pruebas utilizadas para realizar la comparación de desempeño de los diferentes APIs paralelos para la implementación del *ray casting*. Con estas pruebas se quiere determinar cual es la mejor tecnología a utilizar en el despliegue de volúmenes con *ray casting*, para así implementar el software de visualización multi-resolución tomando en cuenta

los resultados obtenidos. Primero se muestran los datasets utilizados, luego la metodología de prueba, seguida de los resultados y su discusión.

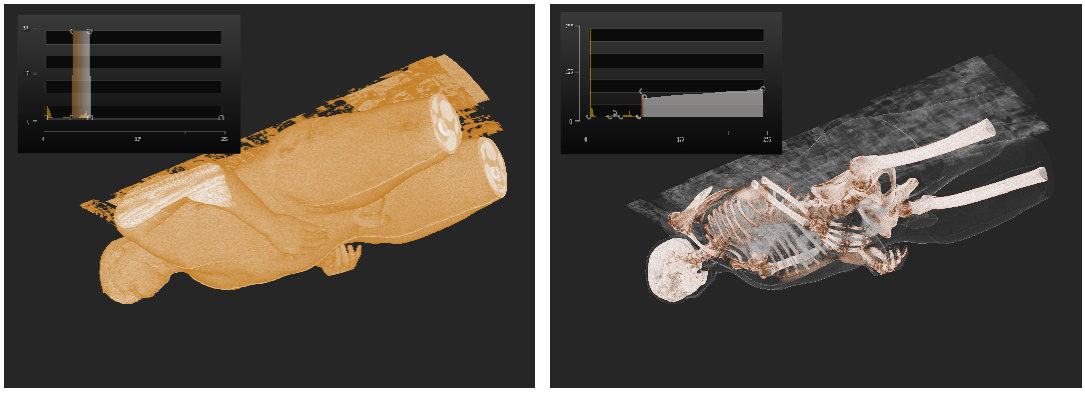
3.1.1. Datasets

Para esta prueba se seleccionaron 3 datasets diferentes: el Hombre Visible del Proyecto del Hombre Visible [65] (Figura 3.1), una ecuación explícita (Figura 3.2) y un escarabajo [66] (Figura 3.3). Para tener una mayor cantidad de datasets para realizar pruebas, también se consideraron los mismos volúmenes en 5 resoluciones reducidas como puede observarse en la Tabla 3.1. Todos los volúmenes considerados poseen 16 bits por muestra.

Tabla 3.1: Tamaño de los datasets.

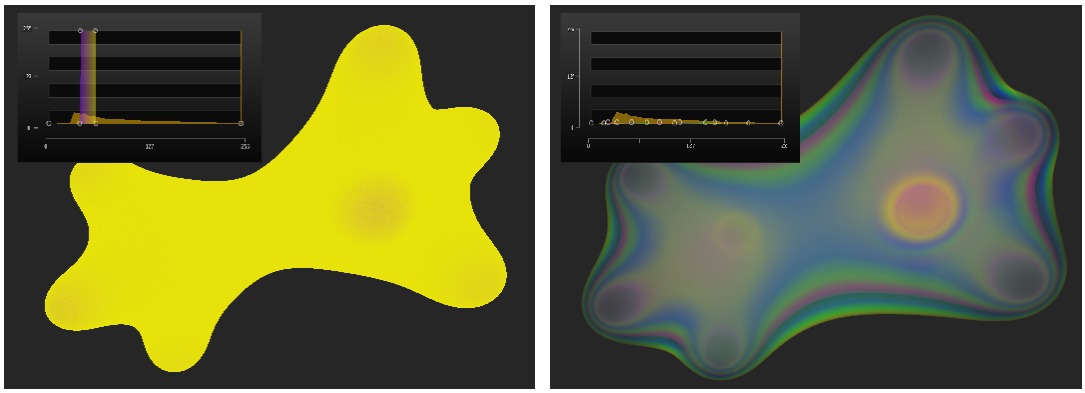
		Hombre	**Lemniscata**	**Escarabajo**
1	**Dimensión**	$512 \times 512 \times 1245$	$512 \times 512 \times 1058$	$832 \times 832 \times 494$
	Tamaño(MB)	622	529	652
2	**Dimensión**	$512 \times 256 \times 622$	$512 \times 256 \times 529$	$416 \times 416 \times 494$
	Tamaño(MB)	155	132	163
3	**Dimensión**	$256 \times 256 \times 311$	$256 \times 256 \times 264$	$416 \times 208 \times 247$
	Tamaño(MB)	38	33	40
4	**Dimensión**	$128 \times 128 \times 311$	$128 \times 128 \times 264$	$208 \times 208 \times 123$
	Tamaño(MB)	9	8	10
5	**Dimensión**	$128 \times 64 \times 155$	$128 \times 64 \times 132$	$104 \times 104 \times 123$
	Tamaño(MB)	2	2	2

Además, dos funciones de trasferencia fueron utilizadas para desplegar cada dataset. En el caso del Hombre Visible la primera función de transferencia (a) tiene como objetivo definir la piel como una superficie opaca (hombre TF1, Figura 3.1a), con lo cual algunos de los rayos se detendrán al chocar con la misma, y el proceso de despliegue terminará más rápido. Por otra parte, la segunda función de transferencia (b), define los tejidos de baja densidad como semi trasparentes, y los huesos opacos (hombre TF2, Figura 3.1b) obligando a los rayos a realizar un mayor recorrido hasta que choquen contra vóxeles opacos. Para la ecuación explícita también se escogieron dos funciones de transferencia: (a) opacidad máxima al chocar con una superficie particular (lemni TF1, Figura 3.2a), y (b) muchas superficies semitransparentes (lemni TF2, Figura 3.2b). Finalmente, el dataset del escarabajo fue desplegado con dos funciones de transferencia: (a) escarabajo con exoesqueleto opaco (escarabajo TF1, Figura 3.3a) y exoesqueleto semitransparente (escarabajo TF2, Figura 3.3b).



(a) Hombre TF1 (b) Hombre TF2

Figura 3.1: Despliegue del dataset del hombre con 2 funciones de transferencia.



(a) Lemni TF1 (b) Lemni TF2

Figura 3.2: Despliegue del dataset de la lemniscata con 2 funciones de transferencia.

3.1.2. Metodología de prueba

Para todas las pruebas los datasets fueron desplegados obligando a que los rayos atraviesen el volumen en diagonal, de manera tal que en el peor de los casos, un rayo necesite atravesar la longitud más larga posible del volumen. Las pruebas fueron realizadas para comparar el desempeño de las tecnologías de una forma justa. Primeramente, las implementaciones con *compute shader*, OpenCL, y CUDA fueron probadas con diferentes tamaño de bloque. Los tamaños de bloques fueron escogidos de tal manera que mantuvieran un número de hilos potencia de dos de la siguiente forma: 4, 8, 16, 32, 64, 128. Además, para cada número de hilos diferentes configuraciones fueron probadas considerando varias combinaciones de radio aspecto alto/ancho (mirar Tabla 3.2). Otros tamaños de bloques y configuraciones también fueron probadas (por ejemplo 16×16 y 32×16) en una manera menos exhaustiva, debido a que esas configuraciones no presentaron mejores resultados que los obtenidos por las configuraciones mostradas en la tabla. Se resumen las pruebas únicamente mostrando

(a) Escarabajo TF1 (b) Escarabajo TF2

Figura 3.3: Despliegue del dataset del escarabajo con 2 funciones de transferencia.

los mejores resultados para una configuración de bloque, para cada dataset y función de transferencia.

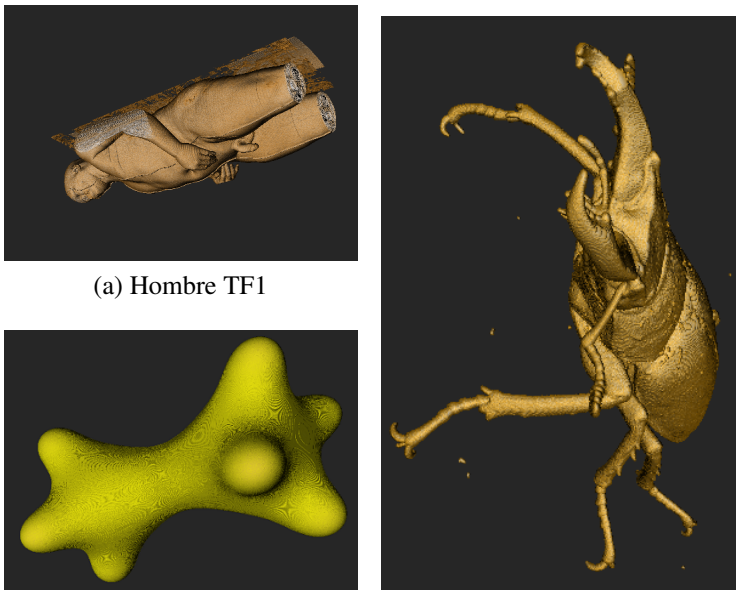
Tabla 3.2: Configuraciones de bloques utilizadas para las pruebas.

Número de hilos por bloque	**Configuraciones**
4	4×1, 2×2
8	8×1, 4×2
16	16×1, 8×2, 4×4
32	32×1, 16×2, 8×4
64	64×1, 32×2, 16×4, 8×8
128	128×1, 64×2, 32×4, 16×8

Además, las pruebas fueron realizadas utilizando la rasterización, y la prueba de intersección rayo/caja, para todos los datasets, todas las funciones de transferencia, y con una resolución de pantalla fija de 1024×768. Por cada píxel de la pantalla se ejecuta un hilo, y el tamaño y configuración del bloque indica como esos hilos son agrupados. Por ejemplo, si tenemos un bloque 4×4, entonces los hilos serán agrupados en una matriz de $1024/4 \times 768/4 = 256 \times 192$ bloques, con 16 hilos por cada bloque. Los resultados son también comparados con el *fragment shader*. Con el *fragment shader* no es necesario escoger entre configuraciones de bloques, ya que se ejecutará un programa de fragmentos por cada píxel de la pantalla, sin haber una agrupación explícita entre ellos.

Finalmente, una prueba adicional fue realizada añadiendo iluminación difusa al proceso del *ray casting*. La iluminación fue probada solamente con funciones de transferencias opacas (como puede observarse en la Figura 3.4), debido a que la iluminación de volúmenes con

funciones de transferencia transparentes conlleva resultados visuales incoherentes. Esta prueba fue realizada para el *fragment shader*, *compute shader*, OpenCL, y CUDA. Para obtener la iluminación es necesario realizar un conjunto de cálculos adicionales en el despliegue que suelen generar un impacto en el rendimiento.



(a) Hombre TF1

(b) Lemni TF1

(c) Escarabajo TF1

Figura 3.4: Despliegue de los datasets de prueba utilizando iluminación difusa con una luz blanca.

3.1.3. Resultados

Primeramente, en la Tabla 3.3 y la Figura 3.5, se muestran los tiempos obtenidos en milisegundos por *frame* para *fragment shader*, *compute shader*, OpenCL y CUDA. Estas pruebas fueron realizadas considerando dos tipos de intersección rayo/volumen: rasterización (R), e intersección rayo/caja (RB). Se resumen las pruebas únicamente mostrando los tiempos obtenidos por la mejor configuración del bloque, junto con su correspondiente configuración para *compute shader*, OpenCL y CUDA. En el Apéndice B se puede encontrar un ejemplo de una tabla que contiene todas las pruebas realizadas con todas las configuraciones de bloques para un solo dataset. Para poder realizar estas pruebas, se inició un reloj al principio del ciclo principal de despliegue, y se detuvo al desplegar todos los *frames*, promediando el tiempo obtenido por la cantidad de *frames* desplegados. También se realizaron las pruebas con ambos métodos (R y R/B), incluyendo y excluyendo el despliegue de la imagen final a la pantalla, para medir si esto ocasionaba una sobrecarga adicional. Esta sobrecarga siempre se encontraba entre $0,1$ ms y $0,5$ ms en los tiempos de ejecución para todos los casos.

Tabla 3.3: Comparación de desempeño en milisegundos de *fragment shader*, *compute shader*, OpenCL y CUDA.

			Fragment	Compute R	Compute RB	OpenCL R	OpenCL RB	CUDA R	CUDA RB
hombre	**TF1**	**Tiempo**	68,61	37,91	**36,99**	60,27	**60,25**	**47,28**	47,91
		Tamaño		4×4	4×4	8×8	8×8	4×4	4×4
$512 \times 512 \times 1245$	**TF2**	**Tiempo**	115,77	68,59	**68,20**	**108,57**	110,97	**84,24**	84,87
		Tamaño		4×4	4×4	8×8	8×8	4×4	4×4
hombre	**TF1**	**Tiempo**	16,79	11,75	**11,58**	26,41	**25,21**	**13,66**	13,68
		Tamaño		8×4	8×4	8×8	16×8	8×8	8×8
$512 \times 256 \times 622$	**TF2**	**Time**	31,33	21,11	**20,94**	44,43	**43,62**	**26,08**	26,38
		Tamaño		8×4	8×4	8×8	8×8	8×4	8×4
hombre	**TF1**	**Tiempo**	4,00	3,86	**3,85**	15,11	**14,98**	**3,85**	3,90
		Tamaño		8×8	8×8	8×8	16×8	16×8	16×8
$256 \times 256 \times 311$	**TF2**	**Tiempo**	7,55	6,44	**6,33**	23,21	**22,81**	**8,12**	8,28
		Tamaño		8×8	8×8	16×8	16×8	8×8	8×8
hombre	**TF1**	**Tiempo**	2,46	2,27	**2,14**	11,83	**11,40**	**2,41**	2,48
		Tamaño		16×8	16×8	16×8	16×8	16×8	16×8
$128 \times 128 \times 311$	**TF2**	**Tiempo**	3,46	3,88	**3,78**	17,84	**17,51**	**4,08**	4,13
		Tamaño		16×8	16×8	16×8	16×8	16×8	16×8
hombre	**TF1**	**Tiempo**	1,34	1,40	**1,19**	**7,65**	7,71	**1,55**	1,62
		Tamaño		32×4	16×8	8×8	16×8	16×8	16×8
$128 \times 64 \times 155$	**TF2**	**Tiempo**	1,81	1,79	**1,68**	11,43	**11,07**	**2,17**	2,22
		Tamaño		16×8	16×8	16×8	16×8	16×8	16×8
lemni	**TF1**	**Tiempo**	51,07	33,60	**33,57**	51,23	**50,77**	**43,18**	43,46
		Tamaño		4×4	4×4	8×8	8×8	8×4	8×4
$512 \times 512 \times 1058$	**TF2**	**Tiempo**	110,86	72,19	**71,70**	**109,61**	110,03	**91,06**	91,62
		Tamaño		4×4	4×4	8×8	8×8	4×4	4×4
lemni	**TF1**	**Tiempo**	11,02	**9,21**	9,22	23,85	**23,49**	9,93	**9,85**
		Tamaño		8×8	8×8	16×8	16×8	8×8	8×8
$512 \times 256 \times 529$	**TF2**	**Tiempo**	26,63	21,10	**20,87**	44,51	**43,65**	**21,97**	21,98
		Tamaño		8×4	8×4	16×8	16×8	8×8	8×8
lemni	**TF1**	**Tiempo**	3,84	3,53	**3,42**	**14,79**	14,90	**3,40**	3,45
		Tamaño		16×8	16×8	16×8	16×8	16×8	16×8
$256 \times 256 \times 264$	**TF2**	**Tiempo**	6,49	6,68	**6,57**	25,80	**25,52**	**7,52**	7,53
		Tamaño		8×8	8×8	16×8	16×8	16×8	16×8
lemni	**TF1**	**Tiempo**	2,57	2,10	**1,93**	11,29	**11,15**	**2,35**	2,38
		Tamaño		16×8	16×8	16×8	8×8	16×8	16×8
$128 \times 128 \times 264$	**TF2**	**Tiempo**	3,53	3,50	**3,39**	18,62	**18,46**	**3,91**	3,96
		Tamaño		16×8	16×8	16×8	16×8	16×8	16×8
lemni	**TF1**	**Tiempo**	1,46	1,33	**1,24**	8,17	**8,00**	**1,61**	1,69
		Tamaño		16×8	16×8	16×4	16×4	16×8	16×8
$128 \times 64 \times 132$	**TF2**	**Tiempo**	1,941	1,85	**1,75**	**12,02**	12,65	**2,31**	2,40
		Tamaño		16×8	16×8	16×8	8×8	16×8	16×8
escarabajo	**TF1**	**Tiempo**	104,93	**69,34**	69,45	144,01	**138,75**	84,58	**84,51**
		Tamaño		8×4	8×4	8×8	16×8	8×8	8×8
$832 \times 832 \times 494$	**TF2**	**Tiempo**	96,84	64,02	**64,01**	132,06	**128,63**	**78,13**	78,49
		Tamaño		8×4	8×4	8×8	16×8	8×4	8×4
escarabajo	**TF1**	**Tiempo**	37,60	**29,33**	**29,33**	80,10	**79,70**	32,52	**32,28**
		Tamaño		8×8	8×8	16×8	16×8	8×8	8×8
$416 \times 416 \times 494$	**TF2**	**Tiempo**	36,01	27,63	**27,61**	73,83	**73,48**	30,02	**29,94**
		Tamaño		8×8	8×8	16×8	16×8	8×8	8×8
escarabajo	**TF1**	**Tiempo**	10,38	10,22	**9,96**	54,74	**54,61**	**11,19**	11,36
		Tamaño		16×8	16×8	16×8	16×8	16×8	16×8
$416 \times 208 \times 247$	**TF2**	**Tiempo**	9,81	9,36	**9,23**	**50,20**	**50,20**	**10,33**	10,38
		Tamaño		16×8	16×8	8×8	8×8	16×8	16×8
escarabajo	**TF1**	**Tiempo**	5,50	5,18	**5,06**	33,65	**33,54**	**6,41**	6,43
		Tamaño		16×8	16×8	8×8	8×8	16×8	16×8
$208 \times 208 \times 123$	**TF2**	**Tiempo**	5,17	4,80	**4,68**	30,98	**30,70**	**5,91**	5,92
		Tamaño		16×8	16×8	8×8	8×8	16×8	16×8
escarabajo	**TF1**	**Tiempo**	3,24	3,13	**2,92**	20,89	**20,74**	**3,88**	3,89
		Tamaño		16×8	16×8	8×8	8×8	16×8	16×8
$104 \times 104 \times 123$	**TF2**	**Tiempo**	3,06	2,88	**2,69**	19,24	**19,10**	**3,60**	3,61
		Tamaño		16×8	16×8	8×8	8×8	16×8	16×8

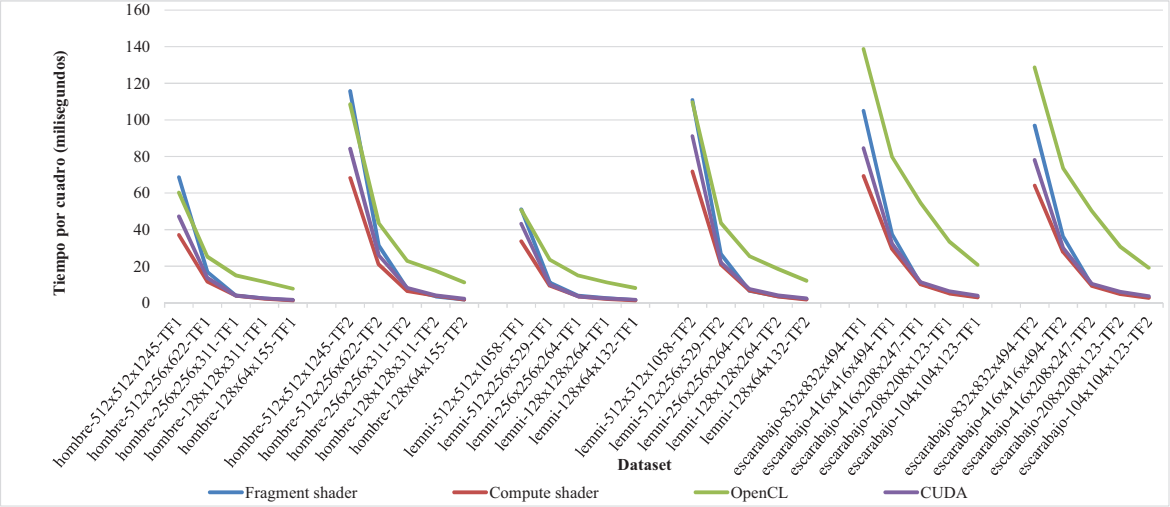


Figura 3.5: Comparación del tiempo de ejecución en milisegundos para *fragment shader*, *compute shader*, OpenCL y CUDA.

Con estos resultados se encontró que mientras más grande el dataset mayor es el tiempo de despliegue, debido a que los rayos deben atravesar áreas de volumen más largas. Lo mismo ocurre con funciones de transferencia transparentes, debido a que mientras más transparente sea, los rayos tendrán menos probabilidad de terminar tempranamente, y deben atravesar mayor cantidad de vóxeles.

En la tabla se marcaron algunos valores para indicar el mejor resultado entre R y RB, para cada implementación, dataset, y función de transferencia. En el caso de *compute shader*, RB mostró ser más rápido que R en el $96\,\%$ de los casos, en a lo sumo $0,92$ ms, representando un incremento de velocidad máximo de $14,5\,\%$. RB es más rápido en el caso de OpenCL en el $83\,\%$ de los casos, pero la diferencia entre R y RB es como máximo $5,26$ ms, representando un incremento de velocidad de a lo sumo de $4,97\,\%$. Sin embargo, en el caso de CUDA, R es la mejor opción en $86\,\%$ de los casos, siendo hasta $0{,}63$ ms más rápido que RB, representando un incremento de velocidad máximo de $5,14\,\%$.

En general, la mejor opción para el tamaño del bloque es 16×8 para los volúmenes pequeños (menos de 40 MB), 16×8 u 8×8 para volúmenes intermedios (cerca de los 150 MB), y 4×4 u 8×4 para los volúmenes grandes (cerca de los 600 MB), para la mayoría de los casos. Como puede observarse, los mejores resultados tienden a ser configuraciones de bloques con un radio aspecto ancho/alto de $1:1$ o $1:2$. Para los volúmenes más pequeños, seleccionar un mayor número de hilos por bloque ofrece un mejor rendimiento, mientras que para volúmenes más grandes, se obtienen mejores resultados con un menor número de hilos por bloque. Esto puede deberse a problemas de caché, idea que será discutida con mayor detalle posteriormente.

Ahora, tomando en cuenta los resultados obtenidos para *fragment shader*, y los mejores resultados obtenidos entre R y RB para el *comptue shader*, OpenCL y CUDA, podemos realizar comparaciones de interés. Para todos los datasets, *compute shader* siempre obtiene

los mejores resultados. Sus tiempos son de $1,03\times$[1] a $1,85\times$ más rápido que *fragment shader*, de $1,5\times$ a $7,1\times$ más rápido que OpenCL, y de $1,05\times$ a $1,3\times$ más rápido que CUDA. El segundo mejor rendimiento es obtenido con CUDA, el cual es de $1,1\times$ a $5,3\times$ más rápido que OpenCL. Sin embargo, la mejor opción entre *fragment shader* y CUDA depende del tamaño del volumen y la función de transferencia. CUDA es una mejor opción para volúmenes grandes y con funciones de transferencia opacas (TF1), mientras que el *fragment shader* presenta mejores resultados para volúmenes pequeños y funciones de transferencia semi-transparentes (TF2). Además, el peor desempeño es mostrado por OpenCL, el cual es solamente mejor que el *fragment shader* en algunos volúmenes en su resolución máxima.

Finalmente, una prueba adicional fue realizada para medir el desempeño de cada API paralelo añadiendo al *ray casting* iluminación difusa. Los resultados obtenidos son mostrados en la Tabla 3.4 y la Figura 3.6. Solamente se escogió utilizar la primera función de transferencia (TF1) para todos los volúmenes, debido a que esta función de transferencia despliega superficies opacas.

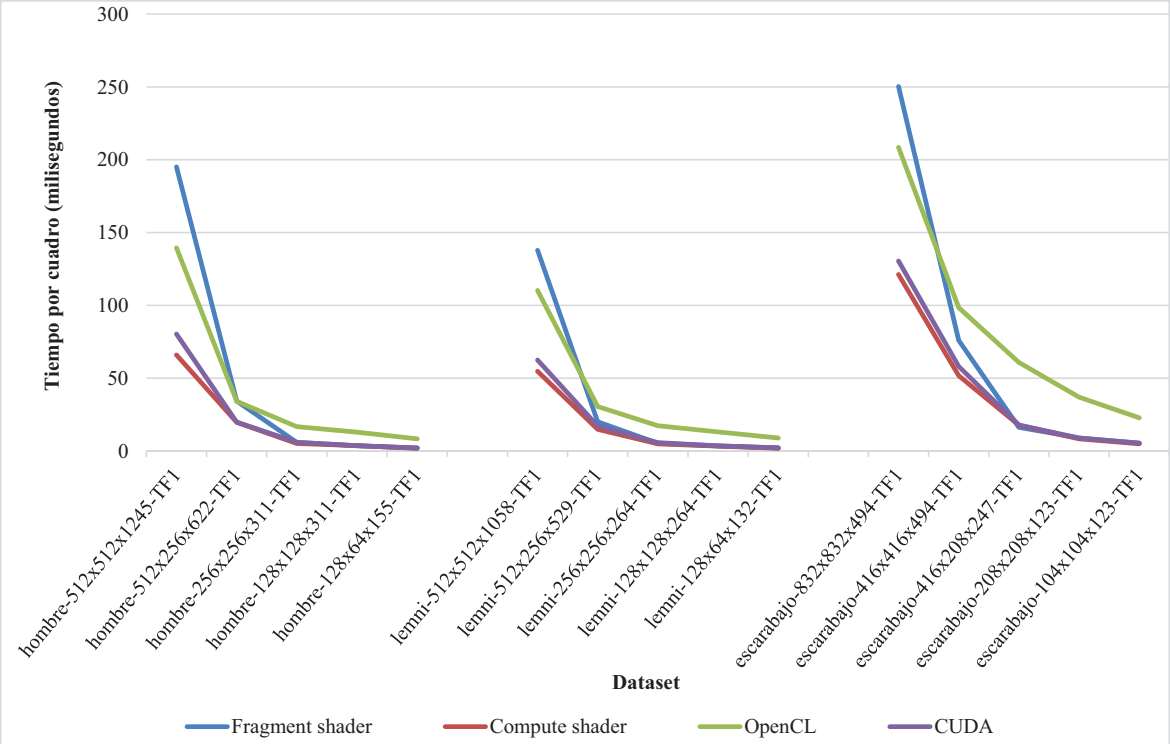


Figura 3.6: Comparación del tiempo de ejecución para *fragment shader*, *compute shader*, OpenCL y CUDA usando iluminación.

Los resultados de esta prueba muestran resultados similares a la prueba anterior. Al igual que en la prueba previa, algunos valores son remarcados para indicar el mejor resultado

[1]Para entender como interpretar estos resultados, referirse al principio del Capítulo 3

Tabla 3.4: Comparación de desempeño en milisegundos de *fragment shader*, *compute shader*, OpenCL y CUDA usando iluminación.

			Fragment	Compute R	Compute RB	OpenCL R	OpenCL RB	CUDA R	CUDA RB
hombre	**TF1**	**Tiempo**	195,01	66,34	**65,86**	139,81	**139,41**	**80,25**	81,15
$512 \times 512 \times 1245$		**Tamaño**		4×2	4×2	4×4	4×4	4×4	4×4
hombre	**TF1**	**Tiempo**	34,27	19,78	**19,61**	34,48	**34,09**	**19,82**	20,25
$512 \times 256 \times 622$		**Tamaño**		8×4	8×4	8×8	8×8	8×4	8×4
hombre	**TF1**	**Tiempo**	5,88	5,25	**5,09**	17,06	**16,65**	**5,92**	6,16
$256 \times 256 \times 311$		**Tamaño**		8×8	8×8	16×8	16×8	8×8	8×8
hombre	**TF1**	**Tiempo**	3,56	3,78	**3,59**	13,31	**12,84**	**3,58**	3,67
$128 \times 128 \times 311$		**Tamaño**		16×8	8×8	16×8	16×8	16×8	16×8
hombre	**TF1**	**Tiempo**	2,02	1,93	**1,78**	8,47	**8,30**	**2,00**	2,12
$128 \times 64 \times 155$		**Tamaño**		16×8	16×8	16×8	16×8	16×8	16×8
lemni	**TF1**	**Tiempo**	137,74	**54,62**	55,05	110,73	**110,19**	**62,44**	63,43
$512 \times 512 \times 1058$		**Tamaño**		4×4	4×4	8×4	8×4	4×4	4×4
lemni	**TF1**	**Tiempo**	20,06	15,21	**14,83**	30,73	**30,54**	**17,19**	17,48
$512 \times 256 \times 529$		**Tamaño**		8×4	8×4	8×8	8×8	8×4	8×4
lemni	**TF1**	**Tiempo**	5,25	5,00	**4,80**	17,64	**17,30**	**5,71**	5,80
$256 \times 256 \times 264$		**Tamaño**		8×8	8×8	16×8	16×8	8×8	16×8
lemni	**TF1**	**Tiempo**	3,46	3,35	**3,25**	13,41	**13,08**	**3,45**	3,54
$128 \times 128 \times 264$		**Tamaño**		16×8	16×8	16×8	16×8	16×8	16×8
lemni	**TF1**	**Tiempo**	2,10	1,92	**1,82**	9,19	**8,92**	**2,10**	2,23
$128 \times 64 \times 132$		**Tamaño**		16×8	16×8	16×8	16×8	16×8	16×8
escarabajo	**TF1**	**Tiempo**	250,27	122,03	**121,16**	209,04	**208,45**	**130,43**	131,68
$832 \times 832 \times 494$		**Tamaño**		4×4	4×4	8×8	8×8	8×4	8×4
escarabajo	**TF1**	**Tiempo**	75,90	51,97	**51,71**	98,64	**98,45**	**57,99**	58,18
$416 \times 416 \times 494$		**Tamaño**		8×8	8×4	8×8	8×8	8×8	8×8
escarabajo	**TF1**	**Tiempo**	16,17	18,05	**17,60**	**60,79**	60,80	17,85	**17,81**
$416 \times 208 \times 247$		**Tamaño**		8×8	8×8	16×8	16×8	16×8	16×8
escarabajo	**TF1**	**Tiempo**	8,88	8,39	**8,29**	**36,92**	37,08	**8,93**	9,10
$208 \times 208 \times 123$		**Tamaño**		16×8	16×8	16×8	16×8	16×8	16×8
escarabajo	**TF1**	**Tiempo**	5,37	5,10	**4,94**	22,76	**22,73**	**5,34**	5,45
$104 \times 104 \times 123$		**Tamaño**		16×8	16×8	16×8	16×8	16×8	16×8

entre R y RB, para cada implementación, dataset y función de transferencia. En el caso del *compute shader*, RB es más rápido que R en $93,3\,\%$ de los casos en a lo sumo $0,86$ ms, representando un incremento de velocidad máximo de $7,69\,\%$. RB es más rápido que R en OpenCL en $86,6\,\%$ de los casos de la tabla en a lo sumo $0,59$ ms, representando un incremento de velocidad máximo de $3,51\,\%$. Sin embargo, en el caso de CUDA, R es la mejor opción en el $93,3\,\%$ de los casos, siendo hasta $1{,}24$ ms más rápido que RB, representando un incremento de velocidad máximo de $5,94\,\%$.

Para volúmenes pequeños (menos de 40 MB), la mejor configuración de bloque es 16×8 para casi todos los casos. En el caso de volúmenes intermedios (cerca de 150 MB), las mejores configuraciones son 8×4, 8×8 o 16×8. Finalmente, para volúmenes más grandes (cerca 600 MB) las mejores opciones son 4×2, 4×4, u 8×4. La única excepción para los volúmenes grandes es en el caso de OpenCL, donde la mejor opción es 8×8. Parecido a la prueba anterior, las configuraciones de bloques con una relación radio aspecto de $1:1$ o $1:2$ tienden a ser la mejor opción, con un mayor número de hilos por bloque para pequeños volúmenes, y un menor número de hilos por bloque para volúmenes más grandes.

Comparando los mejores resultados entre R y RB para cada API paralelo y el *fragment shader*, encontramos que *compute shader* obtiene los resultados más rápidos entre todas las implementaciones con las excepciones del hombre $128 \times 128 \times 311$ con respecto al *fragment shader* y CUDA, y del escarabajo $416 \times 208 \times 247$ con respecto al *fragment shader*. Como antes, CUDA es la segunda mejor opción, siendo entre $1,5\times$ y $4,2\times$ más rápido que OpenCL, y más rápido que el *fragment shader* en algunos casos. El *fragment shader* muestra mejor rendimiento que OpenCL en casi todos los casos, excepto para volúmenes en su máxima resolución, y tiene mejor rendimiento que CUDA en algunos casos de volúmenes pequeños.

Finalmente, el incremento en el tiempo de ejecución debido a la adición del cálculo de la iluminación difusa depende del API. El incremento para el *fragment shader* es entre $1,3\times$ y $2,8\times$, para *compute shader* es entre $1,3\times$ y $1,7\times$, para OpenCL es entre $1,08\times$ y $2,3\times$, y para CUDA es entre $1,2\times$ y $1,7\times$. Como se puede observar, la sobrecarga para el cálculo de la iluminación es menor para *compute shader* y CUDA, que para *fragment shader* y OpenCL. Adicionalmente, se puede observar que en la mayoría de los casos el incremento en tiempo de ejecución para el dataset del hombre es mayor que los incrementos del dataset lemni. Del dataset del escarabajo no se pudo obtener resultados concluyentes comparándolo con los otros dos, ya que el resultados varían dependiendo del API y del tamaño del volumen.

Tomando en cuenta las pruebas realizadas tanto con iluminación como sin iluminación, podemos concluir que el *compute shader* representa, en general, la mejor opción para implementar un *ray casting* de volúmenes básico. También se determinó que no hay un ganador global en tiempo de ejecución entre los métodos de intersección rayo/volumen (rasterización y rayo/caja). Sin embargo, la intersección rayo/caja presenta mejor rendimiento para la mayoría de los casos con *compute shader* y OpenCL, mientras que la rasterización presenta mejores resultados para la mayoría de los casos con CUDA. Por lo tanto, los mejores resultados de todas las pruebas fueron obtenidos combinando *compute shader* con el uso de la prueba de intersección rayo/caja.

En nuestras pruebas se pudo observar que la mejor configuración del tamaño del bloque varía con el tamaño del volumen. Para volúmenes grandes una configuración de 4×4, 8×4, o 4×2 representan las mejores opciones. Vale la pena mencionar que para configuraciones de bloques de 4×4 o 4×2, el número de hilos por bloques es más pequeño que el tamaño del *warp* de CUDA (32 hilos). Teóricamente, con estas configuraciones, la mitad o más de la mitad de los hilos del *warp* permanecerán ociosos, por lo que habrá una alta cantidad de poder de procesamiento sin utilizar, lo cual debería ser ineficiente. Mientras el tamaño del volumen incrementa, una configuración con más hilos por bloque obtiene mejores resultados. Por ejemplo, 8×8 o 16×8 para volúmenes con tamaño cercano a los 150 MB. Configuraciones de bloques con un radio aspecto de $1 : 1$ o $1 : 2$ presentan mejores resultados que las configuraciones con radio aspecto rectangulares, lo que demuestra que las primeras opciones realizan accesos a las textura con mayor localidad espacial, y tienen menor divergencia entre bloques. Con configuraciones con radio aspecto rectangulares, debido a que los hilos toman muestras de data de diferentes áreas del volumen, que pueden representar diferentes materiales, es posible que un alto número de hilo hagan una terminación temprana del rayo, lo cual

produciría un alto número de hilos ociosos esperando a que los otros hilos en el mismo bloque culminen su ejecución. Además, para volúmenes grandes la mejor opción es una menor cantidad de número de hilos por bloques (por ejemplo 16 o 32), mientras que para volúmenes pequeños, la mejor opción es un alto número de hilos por bloque (por ejemplo 64 o 128). Este comportamiento puede estar relacionado con el rendimiento de la caché, debido a que grandes tamaños de bloques requieren acceder a áreas más grande del volumen al mismo tiempo. Para volúmenes pequeños, los vóxeles posiblemente se encuentran en la caché del GPU, mientras que para volúmenes grandes puede ser necesario un trabajo extra para la caché. El mismo problema de rendimiento de la caché podría observarse en configuraciones de bloques con menor cantidad de hilos que el tamaño del *warp* para volúmenes grandes. Con volúmenes grandes cada hilo muestrearía distintas áreas del volumen, generando fallos de caché, que pueden degenerar en una resolución serial de los muestreos. Por este motivo, incluso un número de hilos menor al tamaño del *warp* podría ser beneficioso, debido a que fallos masivos de la caché podrían prevenirse dentro de un *warp*.

3.2. Pruebas de la Implementación del Visualizador de Volúmenes Multi-Resolución

En esta sección se presentan las pruebas realizadas sobre el visualizador de volúmenes multi-resolución. Dados los resultados obtenidos en la Sección 3.1, se decidió realizar todas las implementaciones de códigos GPGPU utilizando el *compute shader*. Primero se muestran los datasets utilizados en estas pruebas. Posteriormente se tienen las pruebas y resultados obtenidos de los diferentes componentes de la implementación, entre los cuales tenemos pruebas de: carga y creación de la jerarquía por bloque, cálculo de las métricas de distancia y distorsión acelerado en GPU, actualización de la textura atlas y despliegue.

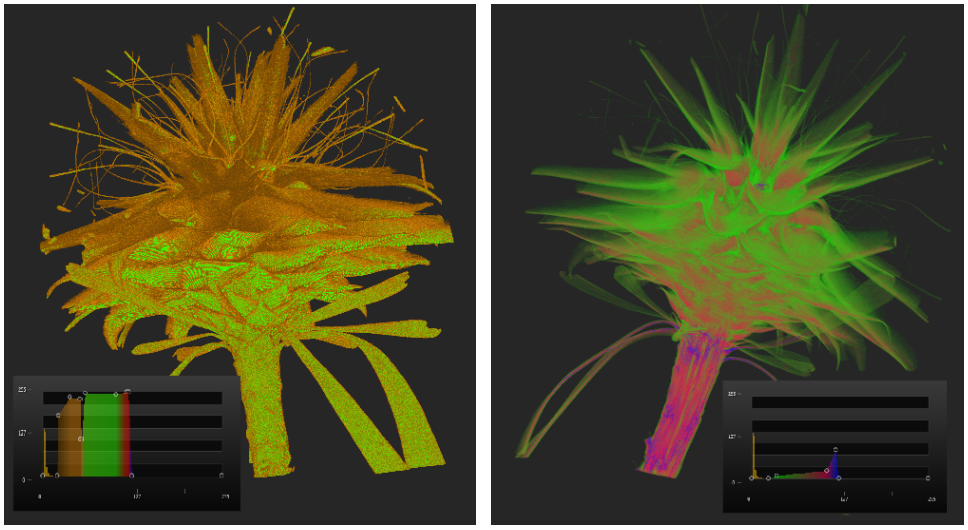
Por mayor comodidad, y debido a que los bloques siempre son seleccionados para ser cuadrados (misma dimensión en sus tres eje), en esta sección indicamos su tamaño con un solo valor que expresa su dimensión en cada eje. Es decir, si estamos utilizando un bloque de $32 \times 32 \times 32$, colocamos que su tamaño es 32. La misma idea será aplicada al referirnos a las dimensiones de la textura atlas y de la auxiliar: al siempre utilizarlas con la misma dimensión para sus tres ejes, se puede indicar su tamaño con un solo valor escalar. Recordemos que la textura atlas se utiliza para almacenar los bloques en sus diferentes niveles de detalles, y se usa la textura índice para indicar en que lugar del atlas se encuentra cada uno de los bloques, haciendo viable el despliegue. En el caso de la textura índice, su tamaño no puede ser variado para las pruebas, ya que debe poseer una entrada por cada bloque del volumen, y depende directamente de cuántos bloques se necesita para poder representar todo el volumen. Por otro lado, la textura auxiliar es un espacio de memoria reservado en el GPU que se utiliza para poder realizar los cálculos de distorsión de los bloques de manera paralela.

3.2.1. Datasets

En la realización de este conjunto de pruebas se seleccionaron otros 3 datasets diferentes: una flor (Figura 3.7), la misma ecuación explícita de la sección anterior con mayor resolución (Figura 3.8) y la Mujer Visible del Proyecto Del Hombre Visible [65] (Figura 3.9). Los tamaños de los datasets pueden observarse en la Tabla 3.5, tamaños con los cuales se quiso saturar la capacidad de memoria de la tarjeta de video. Únicamente el dataset de la flor puede caber en su totalidad en la memoria de la tarjeta de video seleccionada para las pruebas, ya que esta posee 2 GB de memoria de video. Sin embargo, se puede reducir el tamaño de la textura atlas, para simular un espacio más pequeño para la representación del volumen, y poder forzar la paginación de bloques de memoria principal a memoria de GPU.

Tabla 3.5: Tamaño de los datasets.

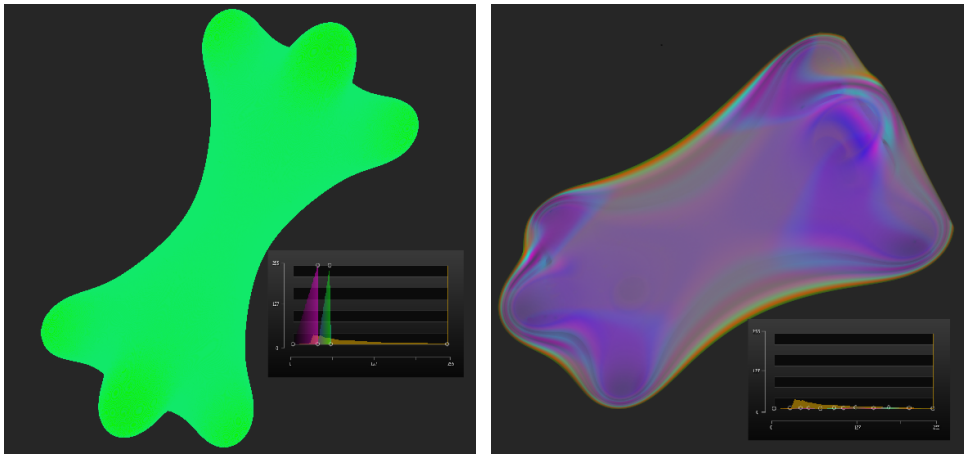
	Flor	**Lemniscata 2**	**Mujer**
Dimensión	$1024 \times 1024 \times 1024$	$960 \times 960 \times 1984$	$2048 \times 1216 \times 1729$
Bits por muestra	8	16	8
Tamaño(GB)	1	3,4	4



(a) Flor TF1 (b) Flor TF2

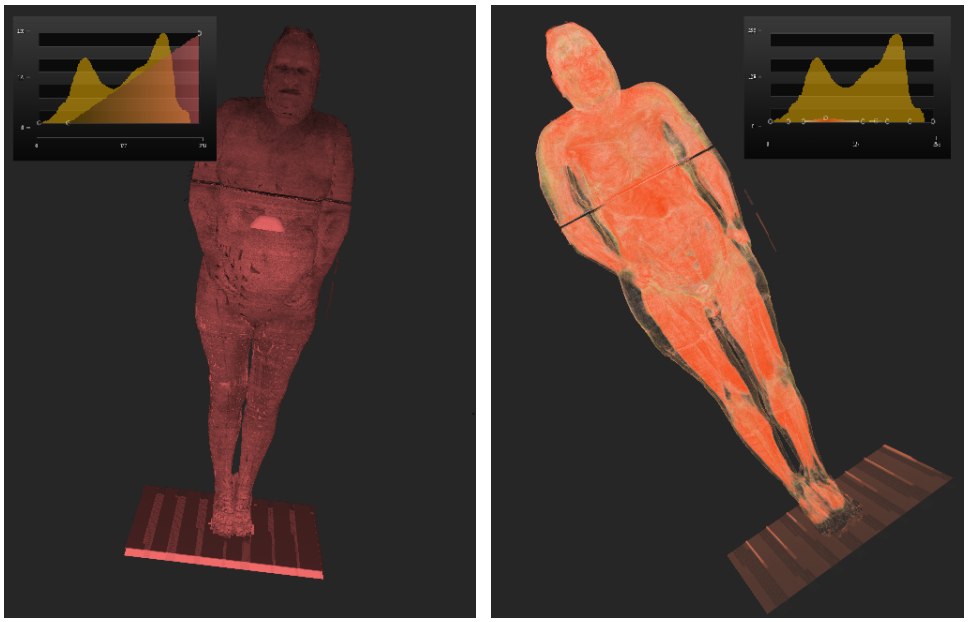
Figura 3.7: Despliegue del dataset de la flor con 2 funciones de transferencia, utilizando una textura atlas de $1024 \times 1024 \times 1024$.

Para cada dataset se utilizaron dos funciones de transferencia. En la primera función de transferencia (flor TF1, lemni2 TF1, mujer TF1), se tiene como objetivo tener superficies



(a) Lemni2 TF1 (b) Lemni2 TF2

Figura 3.8: Despliegue del dataset lemni2 con 2 funciones de transferencia, utilizando una textura atlas de $1024 \times 1024 \times 1024$.



(a) Mujer TF1 (b) Mujer TF2

Figura 3.9: Despliegue del dataset de la mujer con 2 funciones de transferencia, utilizando una textura atlas de $1024 \times 1024 \times 1024$.

opacas, para que aquellos rayos que choquen contra estas superficies, tengan una terminación temprana, y recorran pocos vóxeles del volumen (Figura 3.7a, Figura 3.8a, Figura 3.9a). Aquellos rayos que no choquen contra superficies si realizarán el recorrido completo, debido a que la implementación no consideró el salto de espacios vacío. La segunda función de

transferencia (flor TF2, lemni2 TF2, mujer TF2) representa un volumen semi-transparente que permite el paso de los rayos a través de las superficies, obligando a muestrear una mayor cantidad de vóxeles en el recorrido de los rayos (Figura 3.7b, Figura 3.8b, Figura 3.9b).

3.2.2. Pruebas de Carga y Creación de la Jerarquía por Bloques

El proceso de la creación de la jerarquía por bloques solo se realiza una vez al cargar el volumen desde el disco hacia la memoria principal. Para esta implementación, todo el volumen en su máxima resolución se carga en memoria principal, por lo que esta debe ser capaz de almacenarlo. En la Tabla 3.6 podemos observar la cantidad de bloques que se necesitan para cargar los diferentes modelos utilizando un tamaño de bloque máximo de 8, 16, 32 y 64. Dependiendo del tamaño de bloque máximo la textura de índice que se necesita para representar el objeto variará, siendo más pequeña cuando el tamaño del bloque máximo aumenta. Esto mismo ocurre con los bloques totales. Por ejemplo, si se tiene un volumen de tamaño $1024x768x512$, y tenemos un bloque de tamaño máximo de 16, se tendrá una textura de índices de $64x48x32$ (98304 bloques), mientras con un bloque de tamaño máximo de 64 bastará una textura de índices de $16x12x8$ (1536 bloques). La memoria ocupada nos indica la cantidad total en megabytes que ocupa la representación del dataset utilizando la jerarquía por bloques. Hay que tener presente que esta implementación cuenta también con otro conjunto de estructuras para el manejo de la textura atlas, que no fueron tomadas en cuenta en esta medición.

Tabla 3.6: Memoria ocupada por la jerarquía de bloques.

	Bloque	**Tamaño Textura Índice**	**Megabytes Ocupados**	**Total Bloques**
Flor	**8**	$128 \times 128 \times 128$	1170,00	2097152
	16	$64 \times 64 \times 64$	1170,25	262144
	32	$32 \times 32 \times 32$	1170,28	32768
	64	$16 \times 16 \times 16$	1170,28	4096
Lemni2	**8**	$120 \times 120 \times 248$	3984,74	3571200
	16	$60 \times 60 \times 124$	3985,59	446400
	32	$30 \times 30 \times 62$	3985,69	55800
	64	$15 \times 15 \times 31$	3985,71	6975
Mujer	**8**	$256 \times 152 \times 217$	4710,84	8443904
	16	$128 \times 76 \times 109$	4733,56	1060352
	32	$64 \times 38 \times 55$	4777,12	133760
	64	$32 \times 19 \times 28$	4863,99	17024

En la generación de la jerarquía por bloques del volumen se debe realizar un submuestreo de cada bloque, y se implementó un algoritmo en CPU y en GPU para realizar este proceso. En la Tabla 3.7 y la Figura 3.10 se muestra una comparación del tiempo necesario para la creación

de la jerarquía utilizando estos dos métodos. Los tiempos de carga no dependen de la función de transferencia utilizada. En el caso de la versión paralela basada en GPU, se utiliza la textura atlas para poder realizar el procesamiento paralelo. Por ello, se varía el tamaño de la textura atlas en 64, 128, 256, 512 y 1024, y se utilizó tamaños de bloque de 8, 16, 32 y 64. Ciertas combinaciones de tamaño Atlas/Bloque se encuentran vacías, por dos razones. Primero, en combinaciones como $64/8$ con la flor, o $64/16$ para lemni2 y mujer, poseen una textura atlas demasiado pequeña para poder contener el volumen en su mínima resolución, por lo que no es factible su despliegue. Por ejemplo, la flor de $1024x1024x1024$ con un tamaño de bloque de 8, tiene $1024/8x1024/8x1024/8 = 2097152$ bloques, los cuales no pueden almacenarse en una textura atlas de dimensión 8, ni siquiera en su mínima resolución. Segundo, la combinación $64/64$, posee una textura atlas muy pequeña comparada con el tamaño del bloque, por lo que esta textura no puede ser utilizada para hacer el submuestreo del bloque, ya que se deberían almacenar todos los niveles de detalle en esta memoria. En el caso de la versión de CPU, el tamaño de la textura atlas es irrelevante, debido a que el submuestreo se realiza directamente en la memoria principal, sin tener que utilizar la memoria de textura del GPU. Por ello, se colocaron los resultados solo utilizando una textura atlas de 512, y el resto de los casos fueron ignorados. Los tiempos se tomaron al empezar la carga y al finalizarla, y se repitió varias veces el procedimiento, promediando los resultados, para así obtener resultados más fidedignos.

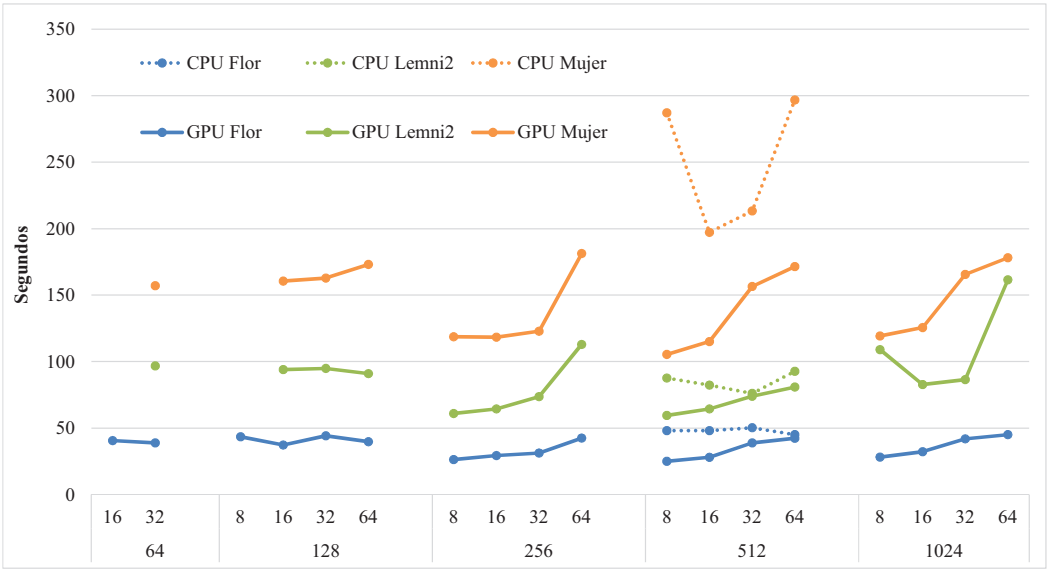


Figura 3.10: Comparación del tiempo de ejecución para la creación de la jerarquía de bloques utilizando CPU y GPU.

Como se puede observar, en la versión en CPU mientras mayor es el tamaño del volumen, el tiempo para la creación de la jerarquía es mayor. Este comportamiento no se mantiene igual para el aumento del tamaño del bloque, donde en muchas ocasiones un tamaño de bloque de 8 y de 64 presentan los peores resultados. Esto se debe a que un mayor tamaño

Tabla 3.7: Comparación del tiempo de ejecución en segundos para la creación de la jerarquía de bloques utilizando CPU y GPU.

Atlas	Bloque	CPU Flor	CPU Lemni2	CPU Mujer	GPU Flor	GPU Lemni2	GPU Mujer
64	**8**	-	-	-	-	-	-
	16	-	-	-	40,56	-	-
	32	-	-	-	38,91	96,72	157,07
	64	-	-	-	-	-	-
128	**8**	-	-	-	43,45	-	-
	16	-	-	-	37,21	94,03	160,49
	32	-	-	-	44,14	94,92	162,88
	64	-	-	-	39,68	90,96	173,14
256	**8**	-	-	-	26,36	60,93	118,73
	16	-	-	-	29,25	64,39	118,32
	32	-	-	-	31,16	73,67	122,86
	64	-	-	-	42,45	112,92	181,25
512	**8**	48,13	87,57	287,10	25,01	59,47	105,40
	16	48,13	82,28	197,20	27,96	64,46	115,09
	32	50,18	76,03	213,42	38,80	73,96	156,46
	64	45,04	92,62	296,73	42,27	80,90	171,56
1024	**8**	-	-	-	28,09	108,93	119,23
	16	-	-	-	32,23	82,66	125,54
	32	-	-	-	41,92	86,46	165,57
	64	-	-	-	45,03	161,60	178,12

de bloque, simboliza una menor cantidad de bloques para representar el volumen, pero se necesita una mayor cantidad de interpolaciones para generar los niveles de detalle para cada bloque. Por otro lado, un tamaño de bloque menor simboliza una mayor cantidad de bloques para representar el volumen, pero por cada uno hay que calcular una jerarquía más pequeña que en el caso anterior. Sin embargo, al momento de la creación de la jerarquía se necesita ir reservando el espacio en la memoria principal que se va a utilizar para almacenar cada uno de estos bloques en cada uno de sus niveles de detalle. Por ello, hay un balance con respecto a la cantidad de memoria que se debe reservar, y la cantidad de procesamiento que se deba realizar, haciendo que los extremos (8 y 64) en algunos casos presenten los peores resultados. En el caso de la versión en GPU, sucede algo similar, pero los resultados con un bloque de tamaño 64 se ven más afectados, ya que el kernel paralelo que hace el submuestreo está optimizado para funcionar con bloques más pequeños.

Finalmente, comparando la versiones de CPU y GPU podemos evidenciar que para el caso de los datasets de la flor y la mujer, el GPU siempre presenta mejores resultados, siendo hasta $1,9\times$[2] más rápido en el caso de la flor, y $2{,}7\times$ más rápido en el caso de la mujer. Para el dataset de lemni2 los resultados no son concluyentes, ya que en ciertas configuraciones la versión en GPU es más rápida que el CPU (usando atlas de 512), y en otras, es más rápida la versión en CPU (usando atlas de 1024). Esto puedo ocurrir, porque a pesar del paralelismo

[2]Para entender como interpretar estos resultados en proporciones, referirse al principio del Capítulo 3

que ofrece el GPU, el tiempo del traspaso de la información (en este caso representada por los bloques) desde el CPU al GPU es considerable, y en las pruebas se toma en cuenta este tiempo de carga de la información, y en el caso del lemni2 se tiene un volumen de 16 bits. No obstante, con el uso del GPU se puede observar que los mejores resultados son obtenidos utilizando un tamaño de textura atlas de 512, la cual es lo suficientemente grande para permitir realizar el submuestreo de una gran cantidad de bloques de manera paralela eficientemente.

3.2.3. Pruebas de Cálculo de la Distancia Paralela

El cálculo de la distancia de los bloques con respecto al ojo es calculado cada vez que el modelo es rotado, trasladado o acercado, y se implementó una versión en CPU y otra en GPU para realizar este cálculo. La versión en CPU fue acelerada utilizando OpenMP para obtener mejores resultados. Se comparó la eficiencia de ambos algoritmos y los resultados son mostrados en la Tabla 3.8 y Figura 3.11. Este cálculo no se ve afectado por la función de transferencia seleccionada, ni el tamaño de la textura atlas seleccionada. Solo depende del tamaño del bloque elegido, ya que esto variará la cantidad de bloques que se necesita para poder representar el volumen. Por lo tanto, la prueba fue realizada sobre todos los datasets, utilizando bloques de tamaño 8, 16, 32 y 64. Para obtener resultados de forma justa, el volumen era cargado siempre con las mismas transformaciones, y se ejecuta el algoritmo del cálculo una cantidad considerable de veces, finalmente promediando los tiempos obtenidos.

Tabla 3.8: Comparación del tiempo de ejecución en milisegundos para el cálculo de la distancia utilizando CPU y GPU.

	Bloque	**Flor**	**Lemni2**	**Mujer**
CPU	**8**	52,87	126,61	281,80
	16	5,75	10,79	24,74
	32	0,79	1,33	3,02
	64	0,13	0,23	0,45
GPU	**8**	48,32	88,40	237,38
	16	5,16	8,90	22,90
	32	0,54	0,78	2,16
	64	0,29	0,35	0,45

Los resultados muestran que mientras menor el tamaño del bloque, ambas implementaciones son más lentas, debido a que se necesita procesar una mayor cantidad de bloques, ya sea utilizando CPU o GPU. Lo mismo ocurre con el tamaño del volumen: a mayor tamaño, mayor cantidad de bloques que se necesita y mayor es el tiempo de procesamiento. Debido a que los tamaños de bloque probado se incrementaron en $2\times$ por cada dimensión ($8\times$ en tamaño total) en cada uno de los casos, la cantidad de bloques necesarios para representar el volumen

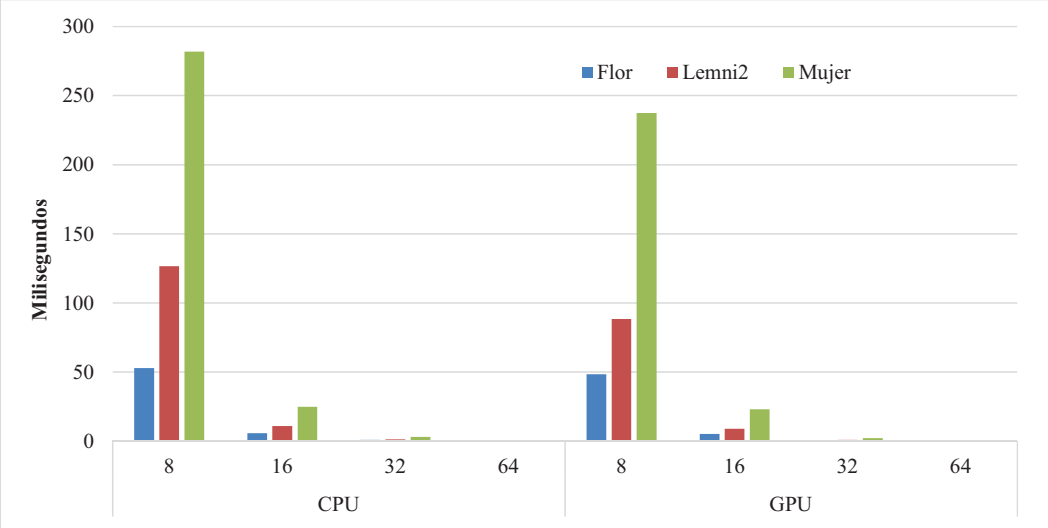


Figura 3.11: Comparación del tiempo de ejecución para el cálculo de la distancia utilizando CPU y GPU.

incrementa en $8\times$, y por ello, los incrementos de velocidad de cálculos de la distancia dado el incremento de tamaño de bloque siguen una proporción similar. Se puede observar que los incrementos en velocidad de cálculo para un tamaño de bloque determinado generalmente están entre $7,25\times$ y $11,39\times$ la velocidad de cálculo del tamaño de bloque inmediatamente inferior. Sin embargo, el incremento de velocidad de cálculo entre bloques de tamaño 32 y 64 es considerablemente menor, estando entre $1,8\times$ y $6\times$. Esto puede deberse a que para tamaños de bloque 64, la cantidad de bloques totales a procesar es muy pequeña, por lo que la velocidad de cálculo puede verse considerablemente retrasada por otros factores como la inicialización de OpenMP o el pase de información desde o hacia el GPU.

Para bloques de tamaño 8, 16 y 32 se observa que la implementación en GPU presenta mejores resultados que la implementación en CPU, siendo hasta un $1,7\times$ más rápido. Sin embargo, utilizando bloques de tamaño 64 el CPU presenta mejores resultados. Esto se debe a que con ese tamaño, se tiene una menor cantidad de bloques a los cuales calcular la distancia, por lo que realizar una llamada a un kernel paralelo en el GPU es más costoso que su cálculo en CPU.

3.2.4. Pruebas de Cálculo de la Distorsión Paralela

El cálculo de la distorsión se debe realizar cada vez que la función de transferencia se modifica, teniéndose que recalcular para todos los bloques en todos los niveles de detalle posible. Para ello se realizó una versión en CPU y dos versiones en GPU. En ambas versiones en GPU se utiliza una memoria auxiliar en el GPU para permitir el cómputo de manera paralela. En la primera versión, los bloques solo son cargados en su máxima resolución y un kernel se encarga

de calcular cada uno de los niveles de detalle. En la segunda versión (GPU2) los bloques en todos los niveles de detalle son cargados directamente desde la memoria principal.

En estas pruebas se comparó la eficiencia de cada uno de los algoritmos, utilizando todos los volúmenes. Para ello, primero se carga el volumen, y se realiza el cálculo de la distorsión de cada uno de los bloques sin interrupciones para el despliegue, midiendo el tiempo total que se requiere para realizarlo. El proceso es repetido varias veces y los valores obtenidos fueron promediados para obtener resultados más confiables. El tiempo que se tarda en calcular la distorsión no se ve afectado por la función de transferencia seleccionada, ni el tamaño de la textura atlas. En este caso, se depende del tamaño de la textura auxiliar elegida, y el tamaño del bloque seleccionado, ya que esto variará la cantidad de bloques que se necesita para poder representar el volumen. Por lo tanto, la prueba fue realizada sobre todos los datasets, utilizando bloques de tamaño 8, 16, 32 y 64, y se utilizó una textura auxiliar variando su tamaño en 16, 32, 64, 128, 256, y 512.

La Tabla 3.9 muestra los resultados obtenidos utilizando la implementación en CPU. En el caso de la implementación del CPU, todo se realiza en memoria principal, por lo que la textura auxiliar no es necesaria, y no se variará. Como podemos observar, mientras más grande sea el volumen, mayor cantidad de bloques son necesarios para representarlos y mayor la cantidad de tiempo para calcular la distorsión. Ahora, si comparamos el desempeño entre distintos tamaños de bloques podemos observar que mientras mayor el tamaño de bloque, mayor es el tiempo de procesamiento. Esto puede sonar contradictorio, ya que si el tamaño de bloque es mayor, se tiene menor cantidad de bloques a procesar. Sin embargo, para calcular la distorsión se tiene que hacer una comparación entre los vóxeles de cada bloque en su mayor resolución, y los vóxeles filtrados de cada uno de los bloques de menor resolución. Por lo tanto, con una máxima resolución de bloque mayor, se necesitan tomar mayor cantidad de muestras en cada uno de los bloques, ralentizando el cómputo total de la distorsión.

Tabla 3.9: Comparación del tiempo de ejecución en segundos para el cálculo de la distorsión usando CPU.

Bloque	**Flor**	**Lemni2**	**Mujer**
8	130,30	221,77	522,56
16	170,32	289,20	685,98
32	213,50	364,72	867,52
64	258,58	443,59	1070,83

Posteriormente, la Tabla 3.10 nos muestra la comparación entre las dos versiones en GPU (GPU y GPU2). Hay valores que no se muestran en la tabla, ya que la configuración entre el tamaño de la textura auxiliar y el tamaño del bloque es incompatible. Por ejemplo, un tamaño máximo de bloque de 32 necesita al menos una textura de tamaño 64 para que la distorsión pueda ser calculada en el GPU, ya que necesita ser la menos 8 veces más grande para poder contener todas las resoluciones de un mismo bloque al mismo tiempo. Igual que la versión de

CPU, mientras mayor es el volumen, mayor es el tiempo necesario para calcular la distorsión. Fijando el tamaño de la textura auxiliar, podemos ver que los tiempos se aceleran mientras mayor es el tamaño de bloque en el caso de 8, 16, y 32. Sin embargo, el tiempo decrementa al utilizar un bloque de tamaño de 64 con respecto al tamaño 32. Esto se debe a la manera en como está optimizado el kernel del cálculo de la distorsión. Recordemos que el código lanza bloques de 32×16 hilos, y si el tamaño del bloque es igual o mayor a 32, se obliga a que se tenga que hacer un recorrido del bloque por áreas de manera secuencial, disminuyendo el paralelismo y el rendimiento.

Tabla 3.10: Comparación del tiempo de ejecución en segundos para el cálculo de la distorsión usando GPU, con submuestreo en GPU y cargando todos los niveles de detalle desde la memoria principal (GPU2).

Auxiliar	Bloque	GPU Flor	GPU Lemni2	GPU Mujer	GPU2 Flor	GPU2 Lemni2	GPU2 Mujer
16	**8**	216,45	368,95	873,64	166,84	282,70	668,82
	16	-	-	-	-	-	-
	32	-	-	-	-	-	-
	64	-	-	-	-	-	-
32	**8**	42,48	70,70	170,73	61,58	105,64	247,70
	16	29,47	47,51	111,87	23,61	39,26	83,40
	32	-	-	-	-	-	-
	64	-	-	-	-	-	-
64	**8**	29,89	50,87	120,72	46,89	81,97	191,16
	16	12,30	20,28	46,60	13,65	22,95	53,03
	32	7,69	13,68	31,00	7,60	13,57	29,65
	64	-	-	-	-	-	-
128	**8**	28,06	47,91	113,76	42,80	73,64	187,31
	16	10,61	17,14	40,03	12,59	20,91	49,37
	32	5,82	10,34	23,42	6,31	11,23	24,38
	64	6,85	12,06	28,06	6,73	12,01	27,73
256	**8**	27,86	47,52	112,73	40,67	69,91	165,09
	16	10,45	16,79	39,10	12,37	20,75	48,60
	32	5,50	9,80	22,17	6,03	10,87	23,52
	64	6,00	10,48	24,59	6,01	10,53	24,58
512	**8**	27,56	47,15	112,16	39,56	69,04	163,42
	16	10,36	16,75	38,82	12,24	20,46	48,30
	32	5,52	9,81	22,07	6,07	10,95	23,47
	64	5,93	10,39	24,13	5,98	10,46	24,17

Además, se puede observar que si el tamaño de la textura auxiliar es pequeña con respecto al tamaño máximo del bloque, GPU2 tiende a ser mejor que GPU. GPU2 gana en combinaciones de tamaño Auxiliar/Bloque de $16/8$, $32/16$, $64/32$ y $128/64$, llegando a ser hasta $1,4\times$ más rápida que GPU. Para el resto de los casos, la versión GPU es más rápida que GPU2, llegando a ser hasta $1,6\times$ más rápida. Sin embargo, mientras las combinaciones tamaño Auxiliar/Bloque tienden a 1, esta ganancia se decrementa, llegando incluso a tener una ganancia mínima como en el caso $512/64$.

Por último, en las Figuras 3.12, 3.13 y 3.14, se muestran las gráficas juntando los valores obtenidos en la Tabla 3.9 y la Tabla 3.10, para los datasets de la flor, del lemniscata2 y de la mujer respectivamente. Los resultados obtenidos utilizando CPU se colocaron como si se utilizara una textura auxiliar de tamaño 128 para poder ser visualizados en la gráfica. Sin embargo, para esta versión no se necesita textura auxiliar. Como se puede evidenciar, para todos los volúmenes, todos los tamaños de bloques, y todos los tamaños de textura auxiliar, ambas versiones de GPU muestran un mejor rendimiento que la del CPU. La única excepción es utilizando la combinación de tamaño Atlas/Bloque de $16/8$ donde la versión en CPU es entre $1,2\times$ y $1,6\times$ más rápido que las versiones en GPU. Esto se debe a que en estos casos solo se procesan 8 bloques por cada llamada al kernel del cálculo de distorsión, disminuyendo considerablemente el paralelismo. En el resto de los casos, las versiones en GPU tienen un mejor rendimiento en hasta un $44,3\times$.

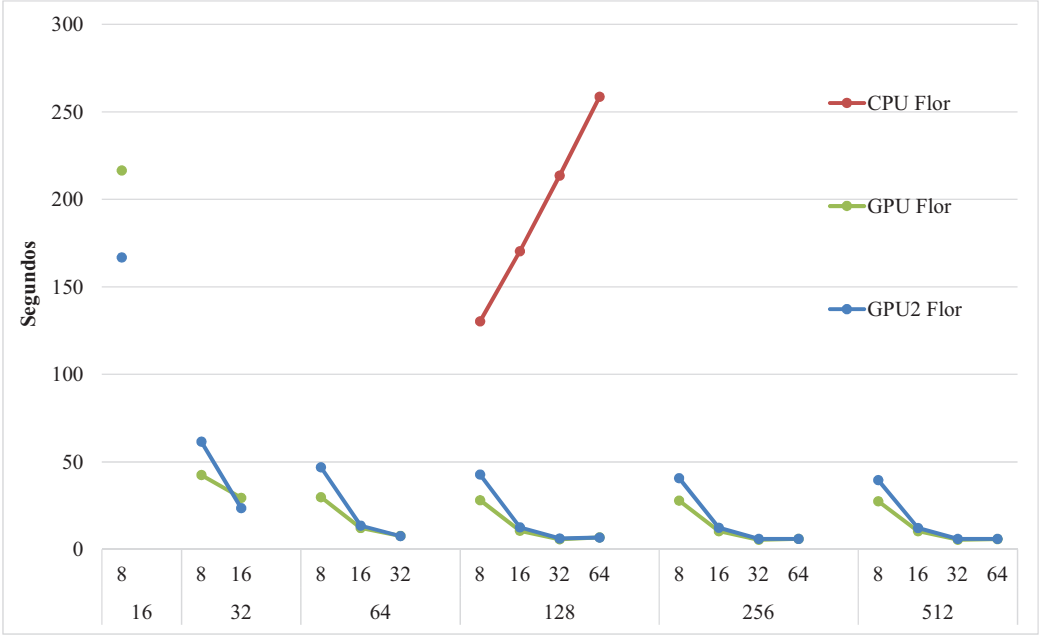


Figura 3.12: Comparación del tiempo de ejecución para el cálculo de la distorsión utilizando CPU, GPU con submuestreo en GPU y GPU cargando todos los niveles de detalle desde la memoria principal (GPU2), para el dataset de la flor.

Sin embargo, a pesar de la aceleración propuesta en GPU, el tiempo para el cálculo de la distorsión de todos los bloques es considerable, ya que se requiere de varios segundos. Como se comentó en la Sección 2.2.3, en otros trabajos se ha propuesto el uso de tablas precalculadas e histogramas para acelerar los cálculos de distorsión [7], pero debido a la necesidad de cuantizar la información y su alto costo de memoria, estas optimizaciones no fueron consideradas para esta implementación. No obstante, se considera de importancia

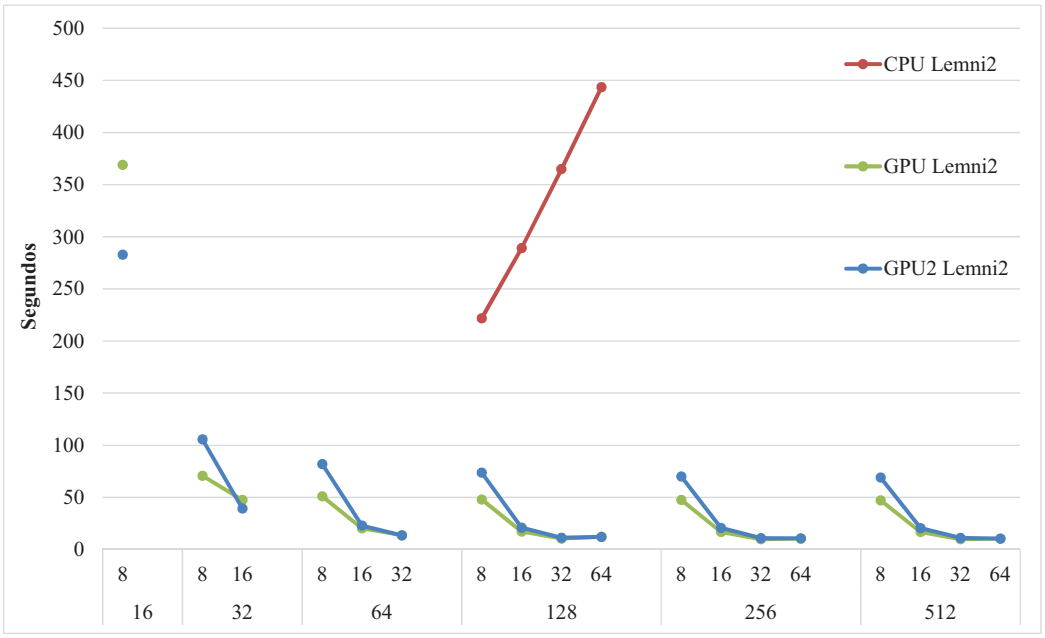


Figura 3.13: Comparación del tiempo de ejecución para el cálculo de la distorsión utilizando CPU, GPU con submuestreo en GPU y GPU cargando todos los niveles de detalle desde la memoria principal (GPU2), para el dataset lemniscata2.

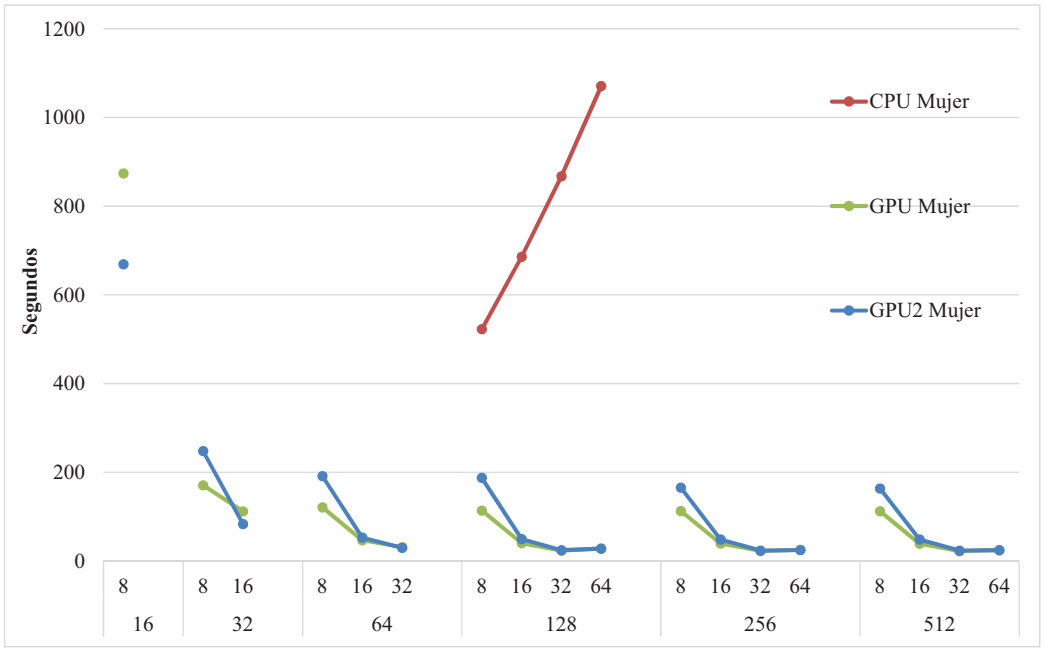
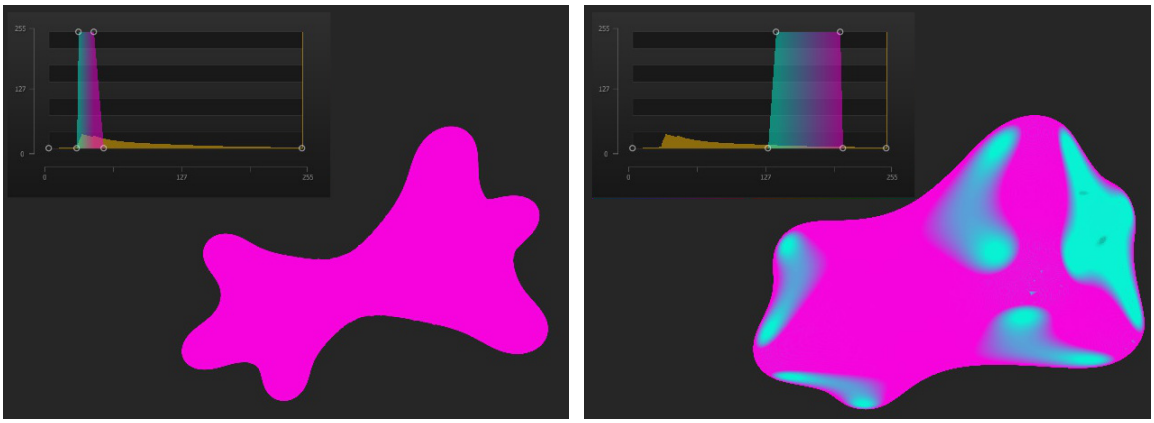


Figura 3.14: Comparación del tiempo de ejecución para el cálculo de la distorsión utilizando CPU, GPU con submuestreo en GPU y GPU cargando todos los niveles de detalle desde la memoria principal (GPU2), para el dataset de la mujer.

buscar otras formas de aumentar aún más la velocidad de estos cálculos manteniendo la precisión.

3.2.5. Pruebas de la Actualización de la Textura Atlas



(a) Lemni2 TF3 (b) Lemni2 TF4

Figura 3.15: Dataset del lemniscata2 con nuevas funciones de transferencia utilizando una textura atlas de tamaño 1024.

La idea de esta prueba es determinar la eficiencia de la actualización de la textura atlas. Se utilizó el dataset del lemni2 con dos nuevas funciones de transferencia (TF3 y TF4) como se muestran en la Figura 3.15, utilizando una textura atlas de 1024 y un tamaño máximo de bloque de 32. Para obtener los resultados, se cargó el volumen utilizando la TF3 (ver Figura 3.15a), y se corrió el algoritmo de refinamiento hasta que convergiera, lo cual implica que el error global del *working set* fue minimizado lo máximo posible, lo cual llamamos caso 1. Por cada *frame* se corrió una iteración del algoritmo, donde se permitía el refinamiento de hasta 10 MB, y se midió la cantidad de movimientos sobre la textura atlas, la cantidad de megabytes movidos, y el tiempo requerido para el cómputo de cada iteración. Una vez convergido, se cambió la función de transferencia para TF4 (ver Figura 3.15b) para probar el rendimiento del algoritmo de refinamiento una vez que ya la textura atlas estuviera llena de información, lo cual llamamos caso 2. Haciendo este cambio de función de transferencia se fuerza a modificar de manera considerable las prioridades de los bloques, provocando mayor cantidad de movimientos dentro de la textura atlas. Finalmente, una vez que el algoritmo de refinamiento haya convergido en el caso 2, se volvió a cambiar la función de transferencia a TF3, midiendo el rendimiento del algoritmo hasta su convergencia a la función de transferencia original, lo cual llamamos caso 3.

En la Figura 3.16 se muestra el comportamiento de la sobrecarga de megabytes movidos y el tiempo requerido por cada iteración, especificando el tiempo de refinamiento, el tiempo

de despliegue, y el tiempo combinado de refinamiento y despliegue. Para este dataset se necesitaron 336 iteraciones totales, donde el caso 1 comprende desde la iteración 1 hasta la 234, el caso 2 desde la iteración 235 hasta la 303, y el caso 3 desde la 304 hasta la 336, requiriendo de $24,6$ segundos totales para su ejecución. Por otro lado, la sobrecarga de bytes movidos contiene la suma de los megabytes movidos hacia la izquierda, hacia la derecha, los que son movidos cuando un bloque A debe sustituir a otro bloque B cuando B es refinado, y los que necesitan actualizarse debido a un colapso. En la Figura 3.16 se hace un desglose de cada uno de estos factores, indicando por cada iteración la cantidad de cada uno de los movimientos que se realizó, y los megabytes adicionales que se necesitaron mover.

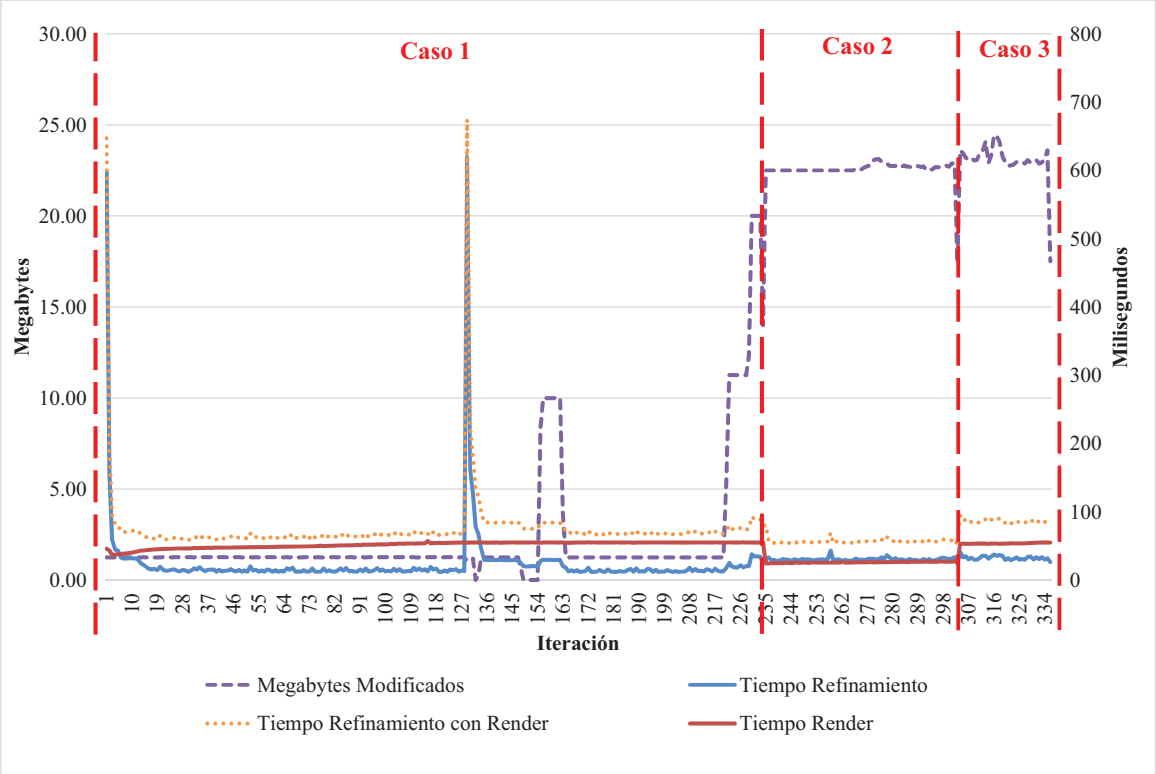


Figura 3.16: Tiempo requerido para el refinado de lemni2 (eje derecho), y cantidad de megabytes totales de sobrecarga actualizados (eje izquierdo), considerando los 3 casos.

En el caso 1 necesitó de $17,8$ segundos para converger, de los cuales $5,8$ segundos fueron utilizados solo para el refinamiento. En promedio, cada iteración necesitó de $51,55$ ms para el despliegue, y de $24,81$ ms para refinar, por lo que el refinamiento requirió de $26\,\%$ del tiempo necesario para generar el *frame*. Sin embargo, se puede observar en las figuras que cerca de las primeras iteraciones hay grandes aumentos en los tiempos necesarios para el refinamiento. Esto se debe a que en ese momento todos los bloques se encuentran en el menor nivel de detalle, y se necesitan realizar muchos refinamientos para poder alcanzar el límite establecido de 10 MB por iteración, conllevando al uso de muchos movimientos por sustitución, y tardando

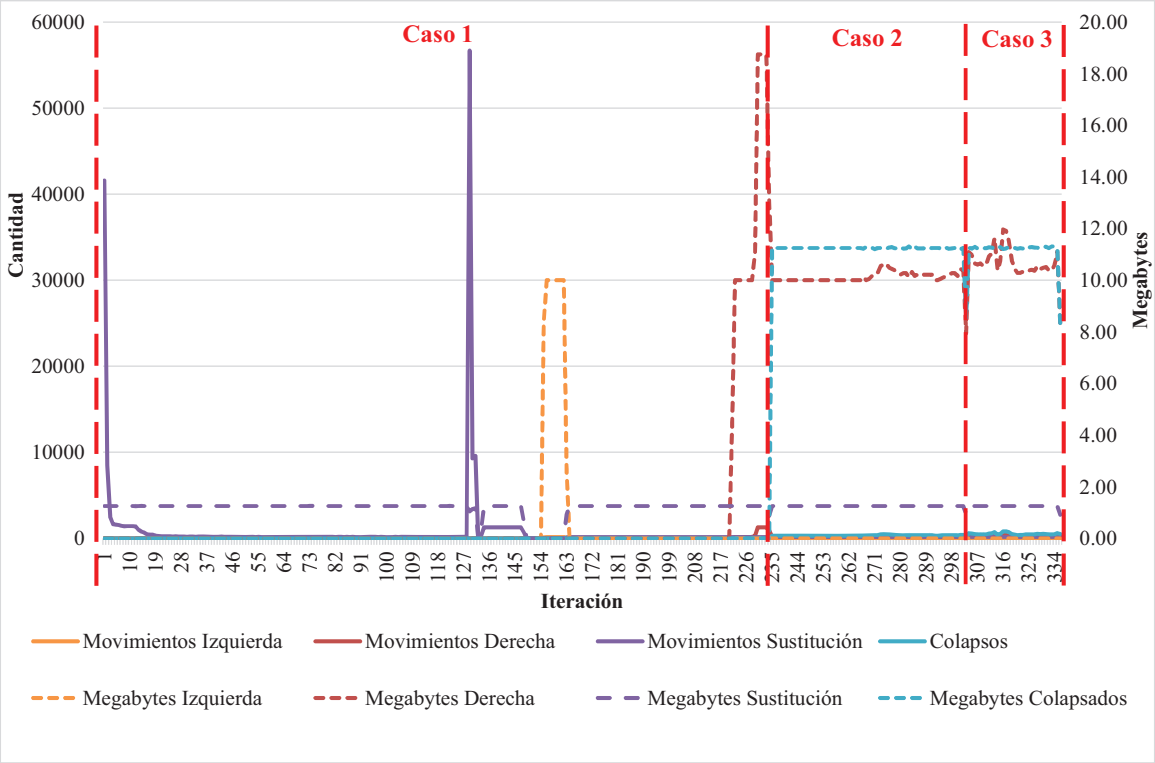


Figura 3.17: Cantidad de movimientos requeridos para el refinamiento (eje izquierdo), y cantidad de megabytes actualizados por estos movimientos (eje derecho).

aproximadamente 600 ms. Estos primeros bloques son refinados varias veces hasta llegar a un nivel de detalle alto cercano a la iteración 127, donde otra vez muchos bloques pequeños requieren refinamiento, emulando el comportamiento de las primeras iteraciones. Luego de esto, se empiezan a ver algunos movimientos a la izquierda y a la derecha, debido a que la textura atlas empieza a estar relativamente ocupada, con lo que el algoritmo necesita liberar espacio. Sin embargo, estos movimientos no representan una sobrecarga en tiempo especial sobre el algoritmo, debido a que los movimientos se realizan para mover bloques de niveles de detalle altos. En promedio por iteración se necesitó actualizar un 23 % de megabytes adicionales a los 10 MB que se quería refinar, es decir que se actualizó 2, 3 MB adicionales. Además, para este primer caso no hubo colapsos, debido a que el algoritmo siempre refina al bloque con mayor prioridad, por lo que al estar llena la textura atlas, el siguiente bloque a refinar no podrá disminuir más el error global obtenido hasta el momento.

Por otro lado, el caso 2 necesitó de 3, 91 segundos para converger, de los cuales 2 segundos fueron utilizados solo para el refinamiento. En promedio, cada iteración necesitó de 25, 95 ms para el despliegue, y de 30, 21 ms para refinar, por lo que el refinamiento requirió de 53 % del tiempo necesario para generar el *frame*. En este caso, la función de transferencia utilizada (TF4) crea una superficie opaca que obliga a los rayos realizar una terminación más temprana que utilizando la función de transferencia anterior (TF3). Por ello, el tiempo de despliegue

disminuye, por lo que el refinamiento tarda en promedio más que el despliegue. Además, debido a que para el inicio del caso 2 ya la textura atlas está llena, y las prioridades de los bloques se cambian, empiezan a ocurrir colapsos de bloques cuya prioridad bajó y cuyo nivel de detalle es muy alto. Como se puede observar en las figuras, la cantidad de movimientos aumenta considerablemente, y se mantiene relativamente constante durante el proceso de refinamiento, donde los colapsos y los movimientos a la derecha son los más utilizados, y los movimientos hacia la izquierda desaparecen totalmente. A pesar del aumento en la cantidad de movimientos y de megabytes adicionales que se necesitan mover, el tiempo promedio necesario no se vió extremadamente afectado en comparación con el caso 1. Sin embargo, el promedio por iteración de megabytes adicionales a actualizar para refinar si aumentó, siendo en promedio de $226\,\%$ ($22,6$ MB adicionales).

Finalmente, el caso 3 necesitó de $2,85$ segundos para converger, de los cuales $1,07$ segundos fueron utilizados solo para el refinamiento. En promedio, cada iteración necesitó de $53,63$ ms para el despliegue, y de $32,58$ ms para refinar, por lo que el refinamiento requirió de $38\,\%$ del tiempo necesario para generar el *frame*. El tiempo promedio de despliegue es similar al del caso 1, debido a que se utiliza la misma función de transferencia, pero el tiempo promedio de refinamiento es parecido al obtenido en el caso 2, debido a que el comportamiento en cantidad de movimientos es similar. Dado que la textura atlas se encontraba llena al inicio del caso 3, se necesitaron de colapsos y movimientos hacia la derecha para poder realizar los nuevos refinamientos, donde tampoco se encuentran movimientos hacia la izquierda. El promedio por iteración de megabytes adicionales a actualizar para refinar aumentó a $232\,\%$ ($23,2$ MB adicionales).

Dado los resultados obtenidos en esta prueba, podemos observar que el algoritmo de refinamiento propuesto presenta un tiempo de ejecución promedio aproximado de 30 ms por iteración, limitando la cantidad de megabytes a refinar a 10 MB, teniendo en cuenta que los tiempos esperados de algunas iteraciones serán mayor si hay un gran número de bloques pequeños a refinar. Por otro lado, a pesar de tener iteraciones con gran número de movimientos, la cantidad de megabytes adicionales a actualizar en la práctica es aceptable, siendo aproximadamente un $230\,\%$, y considerando que la sobrecarga máxima teóricamente es mucho mayor.

3.2.6. Pruebas de Despliegue

Para la visualización se realizaron pruebas cuantitativas y cualitativas. En la Tabla 3.11 y la Figura 3.18, se observa la comparación entre la interpolación intra bloques y entre bloques. Los datasets fueron probados con ambas funciones de transferencia, ya que sus tiempos de despliegue varían según su transparencia. En este caso, los volúmenes fueron cargados, y colocados en una posición específica de la pantalla. Una vez el algoritmo de refinamiento convergiera, se procedió a medir el tiempo de despliegue para varios *frames*, promediando sus

Tabla 3.11: Comparación del tiempo de despliegue en milisegundos utilizando interpolación intra bloque e inter bloque.

		Flor	**Lemni2**	**Mujer**
Intra Bloque	**TF1**	121,85	151,38	78,37
	TF2	345,47	176,98	134,93
Inter Bloque	**TF1**	640,64	398,05	493,61
	TF2	905,66	608,56	716,73

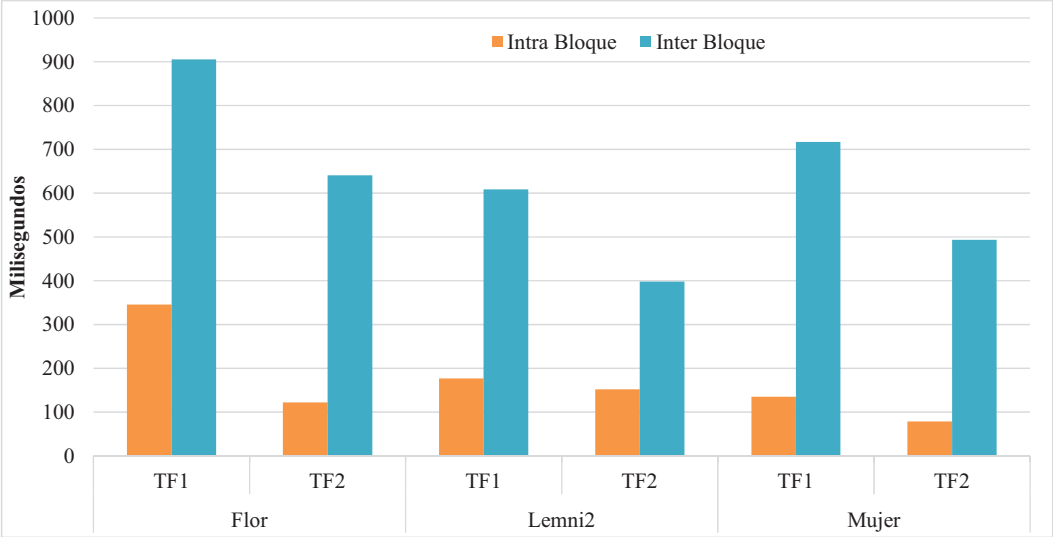


Figura 3.18: Comparación del tiempo de despliegue utilizando interpolación intra bloque e inter bloque.

resultados. Se utilizó un tamaño de la textura atlas de 1024 y tamaño de bloque de 32.

Debido que la segunda función de transferencia es más transparente que la primera, su tiempo de despliegue es mayor para todos los datasets, donde el tiempo de despliegue de los datasets opacos llegan a ser hasta $2,8\times$ más rápidos. Ahora, si comparamos la interpolación intra bloque con la inter bloque, podemos observar que la interpolación intra bloque siempre es más rápida, llegando hasta un máximo de $6,2\times$. Finalmente, no hay evidente relación entre los resultados y el tamaño del volumen, pero esto se debe a que los datasets no fueron desplegados desde el mismo ángulo ni con el mismo acercamiento.

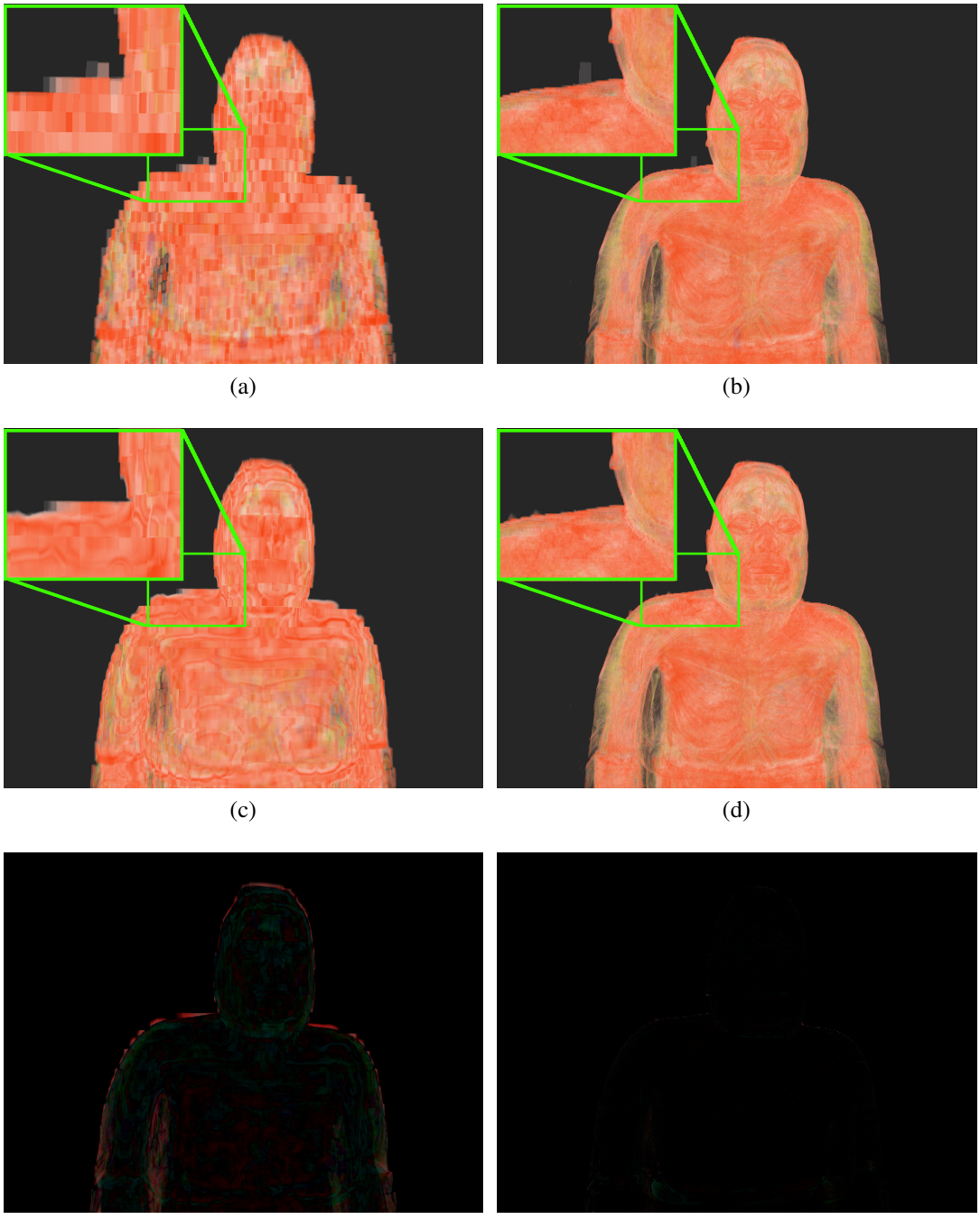
Por otro lado, la Figura 3.19 compara visualmente el despliegue usando la interpolación intrabloque y sin usarla, considerando dos tamaños de textura atlas diferentes. La Figura 3.19a y la Figura 3.19b, muestran la visualización de las imágenes sin el uso de la interpolación con una textura atlas de tamaño 64 y 512 respectivamente, con tiempos de despliegue de $367,44$ ms y $569,54$ ms respectivamente. La Figura 3.19c y la Figura 3.19d, muestran la visualización de las imágenes con el uso de la interpolación con una textura atlas de tamaño 64 y 512 respectivamente, con tiempos de despliegue de $1,92$ segundos y $2,03$ segundos respectivamente. Además, se comparó el resultados visual de las imágenes con uso de la

interpolación, con respecto a la Figura 3.20. Esta figura de referencia se desplegó utilizando una textura atlas de 1024 con interpolación entre bloques, la cual muestra el mejor resultado visual posible obtenido por la aplicación. La Figura 3.19e y la Figura 3.19f muestran las diferencia en el espacio CIEluv de la Figura 3.19c y la Figura 3.19d, con respecto a la imagen de referencia para poder comparar la calidad visual de los resultados obtenidos.

Podemos observar que la mejora visual obtenida por el uso la interpolación entre bloques es más notoria cuando el atlas tiene un tamaño de 64. Esto se debe a que al ser más pequeña la textura atlas, menos bloques pueden ser representados en su máxima resolución, y la interpolación entre bloques ayuda a reducir los artefactos existentes entre las fronteras. Además, el incremento en tiempo de despliegue es mayor con un atlas de 64 ($5{,}22\times$) que con un atlas de 512 ($3{,}57\times$). Esto se debe a que con bloques de menor resolución, la cantidad de muestras que deben ser obtenidas por medio de la interpolación entre bloques aumenta, por lo que también aumentan la cantidad de muestreos que se deben realizar sobre el volumen, y finalmente penalizan más el tiempo de despliegue. Por otro lado, se puede observar que incluso usando la interpolación entre bloques, la diferencia visual usando un atlas de 64 con respecto a la imagen de referencia es alta (Figura 3.19e). Sin embargo, con el uso de un atlas de 512, la diferencia mostrada con la imagen de referencia es pequeña, por lo que para este volumen particular, un atlas de 512 basta para representar el volumen con buena calidad visual.

Sin embargo, la interpolación entre bloques implementada no está exenta de algunos artefactos visuales, como puede ser visto en la Figura 3.21. En esta figura se muestra un dataset creado artificialmente de tamaño $96 \times 96 \times 96$ (creado especialmente para esta prueba), con un bloque de tamaño máximo 32 (teniendo $3x3x3 = 9$ bloques totales), y una textura atlas de 16, por lo que el volumen no cabe en su máxima resolución en el GPU. En la Figura 3.21a, vemos como el volumen no tiene interpolación entre bloques, por lo que posee bordes muy marcados en las fronteras entre bloques. La Figura 3.21b muestra también la interpolación entre bloques, pero nada más en el área de influencia del bloque, ya que estos no están en su máxima resolución. Finalmente, la Figura 3.21c muestra la interpolación entre bloques donde se observa una gama de colores en las fronteras entre los bloques. Esta gama de colores es producida debido a que para realizar la interpolación entre bloques, se interpolan los valores escalares de los vóxeles en las fronteras, cuyos valores al ser clasificados pueden originar colores distintos a los colores en los bordes, incluso llegando a ser transparentes, por lo cual no se ve un volumen continuo como en la Figura 3.21a.

Debido a que la interpolación entre bloques es costosa computacionalmente, se puede sugerir utilizarla únicamente cuando todas las prioridades de los bloques estén calculadas, y el algoritmo de refinamiento ya haya convergido, de manera de agilizar estos otros procedimientos, y que no se vea afectado su rendimiento por el cuello de botella que representa el despliegue. También se consideró la posibilidad de nada más utilizar interpolación entre bloques cuando el usuario no esté interactuando con el volumen o modificando la función de transferencia, de manera que las iteracciones sean más rápidas.



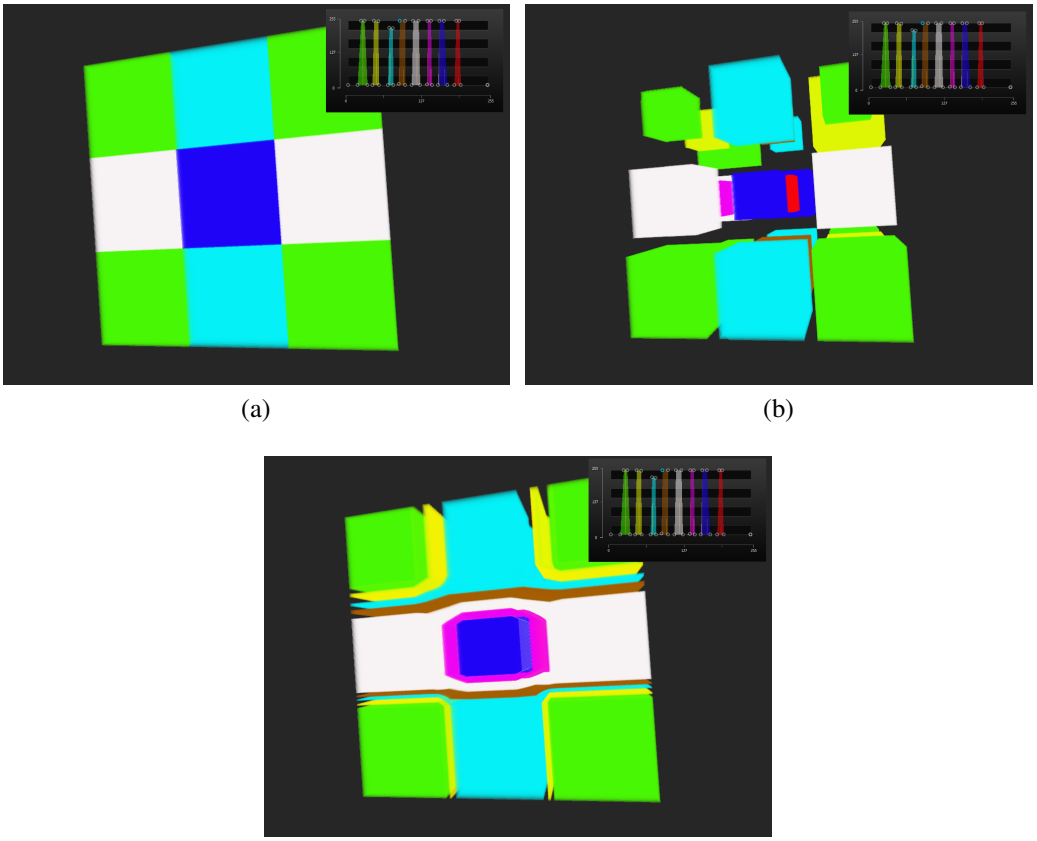
(a) (b)

(c) (d)

(e) (f)

Figura 3.19: Despliegue del dataset de la mujer con diferentes tamaños de textura atlas y métodos de interpolación. Se desplegaron las imágenes con: (a) sin interpolación entre bloques con textura atlas de tamaño 64, (b) sin interpolación entre bloques con textura atlas de tamaño 512, (c) con interpolación entre bloques con textura atlas de tamaño 64, (d) con interpolación entre bloques con textura atlas de tamaño 512, (e) diferencia de (c) en CIEluv con la Figura 3.20 y (f) diferencia de (d) en CIEluv con la Figura 3.20.

Figura 3.20: Despliegue de referencia de la mujer con una textura atlas de 1024.



(a) (b)

(c)

Figura 3.21: Artefactos del uso de interpolación entre bloques mostrado en un volumen con $3x3x3$ bloques, donde (a) muestra la interpolación intra bloques, (b) muestra la interpolación intra bloques solo en el área de influencia del bloque, y (c) muestra la interpolación entre bloques.

Capítulo 4

Conclusiones y Trabajos Futuros

Este trabajo de grado se dividió en dos partes para cumplir sus objetivos. Primero, se realizó un conjunto de implementaciones paralelas de un *ray casting* básico de volúmenes utilizando *fragment shader*, *compute shader*, OpenCL y CUDA. A partir de las pruebas realizadas, se determinó que el API paralelo que mejor se comportó en la mayoría de los casos fue el *compute shader*, además de no tener la necesidad de utilizar un API diferente a OpenGL para el uso de GPGPU. Por ello, para el resto del trabajo, los códigos GPGPU fueron implementados todos utilizando el *compute shader*. *Compute shader* mostró los mejores resultado siendo entre $1,03\times$ y $2,9\times$ más rápido que *fragment shader*, entre $1,5\times$ y $7,1\times$ más rápido que OpenCL, y entre $1,01\times$ y $1,3\times$ más rápido que CUDA. La siguiente mejor opción está entre el *fragment shader* y CUDA, donde CUDA muestra mejores resultados para volúmenes más grandes mientras que para volúmenes pequeños no hay un claro vencedor. Finalmente, OpenCL es el que muestra el desempeño más bajo entre todos los APIs.

Posteriormente, se realizó la implementación de un visualizador de volúmenes multi-resolución utilizando jerarquía de bloques y la textura atlas. Se propusieron varios algoritmos con el uso del GPU para la optimización de los procesos de submuestreo, de cálculo de la distorsión, y cálculo de la distancia de los bloques al ojo, demostrando que utilizando configuraciones de la aplicación que aprovechen el paralelismo del GPU, este ofrece mejores resultados que en el CPU. Por ejemplo, en el caso de la creación de la jerarquía los mejores resultados se obtuvieron utilizando el GPU con una textura atlas de $512 \times 512 \times 512$, debido a que es un tamaño lo suficientemente grande para poder submuestrear una cantidad considerable de bloques de manera paralela, obteniendo en la mayoría de los casos mejores resultados que el CPU. En el caso del cálculo de la distancia, los resultados indicaron que la versión en GPU puede llegar a ser incluso $6\times$ más rápida que la versión en CPU, siendo únicamente más lenta cuando la cantidad de bloques a procesar es muy pequeña y el costo en tiempo del paso de memoria al GPU es mayor a la aceleración obtenida en los cálculos. Por otro lado, en el caso de la distorsión paralela se observó que si se tiene una buena combinación de tamaño textura auxiliar/bloque la versión en GPU siempre tiene un mejor rendimiento a la versión en CPU, llegando a ser hasta $44,3\times$ más rápida. En cambio con un tamaño de

textura auxiliar menor a 8 veces el tamaño del bloque máximo, se permite a lo sumo solo calcular la distorsión de 7 bloques de manera paralela al mismo tiempo, desaprovechando el paralelismo, y siendo en estos casos mejor la versión en CPU. Además, se implementó un algoritmo de interpolación entre bloques sin copia de los valores entre frontera. Por medio de la interpolación se obtienen mejores resultados visuales, pero el algoritmo es costoso en tiempo, por lo que no se recomienda utilizar en los momentos en que el usuario esté interactuando con la aplicación, momentos en los que convendría un tiempo de respuesta más rápido.

La contribución más importante del trabajo es la implementación de un algoritmo para la actualización de la textura atlas utilizando el *Morton order*, donde se definió todas las estructuras y los movimientos de bloques necesarios para evitar la fragmentación de esta memoria. Para ello se propuso un conjunto de invariantes con las cuales se mantiene ordenada y desfragmentada la textura atlas, y así evitar un proceso de desfragmentación completo que sería más costoso. En cuanto a memoria, esta técnica solo posee una estructura de datos adicional a lo que normalmente se utiliza para el despliegue de volúmenes multi-resolución, la cual es un arreglo de listas de apuntadores a bloques, que sirve para mantener las invariantes propuestas en el algoritmo para order la textura atlas. Esta estructura es $O(N)$, donde N es el número de bloques totales, el cual es un costo en memoria aceptable. Con los resultados obtenidos por las pruebas, se demostró que en promedio esta técnica ofrece una sobrecarga en tiempo razonable para el refinamiento del *working set*, presentando un tiempo de ejecución promedio de aproximadamente de 30 ms por iteración. Por otro lado, a pesar de tener iteraciones con gran número de movimientos, la cantidad de memoria adicional que debe ser actualizada en la práctica para mantener la textura altas ordenada es aproximadamente del $230\,\%$, lo cual es aceptable considerando que la sobrecarga máxima teóricamente es mucho mayor, y que la sobrecarga de una desfragmentación completa de la textura atlas podría ser extremadamente costosa computacionalmente. Con este algoritmo se ofrece una posible solución factible a un problema poco atacado hasta ahora que es el manejo eficiente de la textura atlas.

Como trabajo futuro, sería interesante comparar las implementaciones del *ray casting* con un hardware gráfico más moderno, con mayor capacidad, especialmente con nuevas versiones de OpenCL y CUDA. Adicionalmente, en el despliegue de volúmenes es normal que muchos rayos hagan terminación temprana, dada la acumulación de opacidades a lo largo su recorrido. Por ello, el uso de métodos como *persistent threads* [67] puede ser beneficioso para el despliegue de volúmenes con funciones de transferencias opacas. También se propone optimizar el recorrido del rayo realizando el salto de espacios vacío en bloques que son completamente transparentes [18], o el uso de muestreo adaptativo para la toma de menor cantidad de muestras en bloques de baja resolución [68].

Por otro lado, muchos de los algoritmos en GPU planteados están optimizados para trabajar con una tarjeta de video Nvidia con microarquitectura Kepler, y tarjetas más actuales poseen capacidades de mayor tamaño de bloque de CUDA, mayor cantidad de bloques por SM, mayor cantidad de memoria compartida, entre otras cosas. Esto podría cambiar significativamente la programación de algunos kernels como los de submuestreo del volumen y la distorsión paralela. Además, a pesar de acelerar el cálculo de distorsión paralela con el GPU, el tiempo

de cálculo para todos los bloques sigue siendo costoso, por lo que se debe optimizar tratando de precalcular algunos resultados intermedios.

En cuanto al algoritmo de actualización de textura atlas, uno de los aspectos más importantes a optimizar es evitar la carga continua de bloques al GPU cuando se realizan movimientos dentro de la textura atlas. Al refinar un bloque, necesariamente este debe ser cargado desde la memoria principal, pero al realizar movimientos hacia la izquierda, movimientos hacia la derecha, sustituciones de bloques y colapsos, es posible realizarlos sin cargar directamente desde la memoria principal, debido a que toda la información ya se encuentra disponible en el GPU. Por ejemplo, se podría acelerar el tiempo de respuesta teniendo una lista con todos los movimientos a realizar, la cual será utilizada por un kernel en GPU para ejecutar la actualización paralela, y para aquellos casos en que se necesite espacio adicional para algún cálculo intermedio, se podría utilizar la textura auxiliar propuesta para otros algoritmos.

Apéndice A

Programación Paralela

La programación paralela es una forma de cómputo en la que muchas instrucciones se ejecutan simultáneamente, operando sobre el principio de que problemas grandes, a menudo se pueden dividir en unos más pequeños, que luego son resueltos simultáneamente (en paralelo).

Últimamente, este tipo de programación ha tenido un gran avance en las tarjetas gráficas. El uso de los procesadores gráficos para el cálculo de propósito general es conocido como GPGPU (*General-Purpose Computing on Graphics Processing Units*). El GPGPU tiene suma importancia ya que el GPU posee operaciones de punto flotante extremadamente rápidas y están diseñados desde sus inicios para trabajar en paralelo. Sin embargo, el GPU posee ciertas desventajas cuando se utiliza para cómputo de propósito general. La más importante es la lentitud de transferencia de información entre la memoria de GPU y la memoria principal. Además, en general la memoria del GPU es mucho más limitada que la memoria principal, lo que conlleva a tener que subdividir problemas muy grandes.

Actualmente hay una diversa cantidad de APIs para la utilización del GPU en aplicaciones de propósito general. Entre las más conocidas se encuentran: CUDA [21] [69] y OpenCL [22] [70] [71]. Además, OpenGL[23] [72] puede ser utilizado para computo de propósito general utilizando el *fragment shader* con algunas consideraciones especiales, o con el *compute shader*, que es más moderno y tiene mayor similitud con CUDA y OpenCL.

A continuación se expondrán las principales características de estos APIs de desarrollo de programación paralela.

A.1. CUDA

CUDA (*Compute Unified Device Architecture*), es una plataforma de computación paralela creada por NVidia. Permite a los desarrolladores utilizar tarjetas de video que soporten CUDA para realizar cómputo de propósito general, actuando como una capa que da acceso directo al

conjunto de instrucciones que posee el GPU y a sus elementos de computación paralela. Está disponible para las tarjetas gráficas NVidia superior o igual a la serie GeForce 8.

A.1.1. Procesamiento

El API de CUDA se constituye por un *host* (CPU) que se conecta a un *device* (GPU o CPU), el cual se encargará de realizar lo cálculos que le asignó el *host* a través de los *kernles*. Un *kernel* es una función compilada en el GPU, que fue escrita en lenguaje *C for CUDA*. Cuando un *kernel* se ejecuta, se crean varios hilos (*threads*), todos realizando las instrucciones indicadas en el *kernel*. A esto se le conoce como SIMT (*Single Instruction Multiple Thread*), en donde la capacidad multihilo es simulada por un conjunto de procesadores SIMD (*Single Instruction Mutiple Data*). Se pueden crear miles de hilos en pocos ciclos de reloj, casi sin costo alguno, a diferencia del CPU. Los hilos son agrupados en bloques (*blocks*) y los bloques en mallas (*grids*), como se puede observar en la Figura A.1. Tanto los bloques como las mallas pueden ser de una, dos o tres dimensiones, permitiendo una mejor organización lógica dependiendo de la aplicación que se esté realizando.

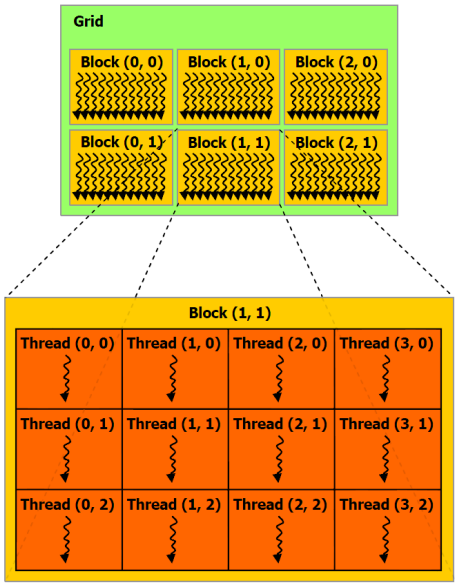


Figura A.1: Representación de los hilos, bloques y mallas. Se presenta una malla bidimensional compuesta por 6 bloques, donde cada bloque también es bidimensional y contiene 12 hilos. En total se tienen 72 hilos. Imagen tomada de [69].

También hay que agregar que CUDA posee un conjunto de instrucciones que permiten la sincronización entre programas y ejecuciones de los hilos. Además, los hilos dentro de los bloques son agrupados en conjuntos de 32 hilos, llamados *warps*, los cuales ejecutarán la misma instrucción al mismo tiempo. Esto quiere decir, que todos los hilos de un *warp* están

sincronizados, lo que permite ciertas optimizaciones a la hora de la programación. Sin embargo, si hay divergencias en el código, es importante que todos los hilos de un *warp* vayan por el mismo camino, sino el desempeño de la aplicación podría verse afectado.

A.1.2. Manejo de la Memoria

El espacio de memoria del *host* y el *device* es completamente separado. El *host* es el encargado de manejar la memoria del *device*, asignando y liberando la memoria, copiando los datos entre memoria de *host* y memoria del *device*, e incluso puede copiar datos entre memoria del *device* y memoria del *device*. El GPU posee una jerarquía de memoria (ver Figura A.2), donde se encuentran los siguientes tipos de memoria:

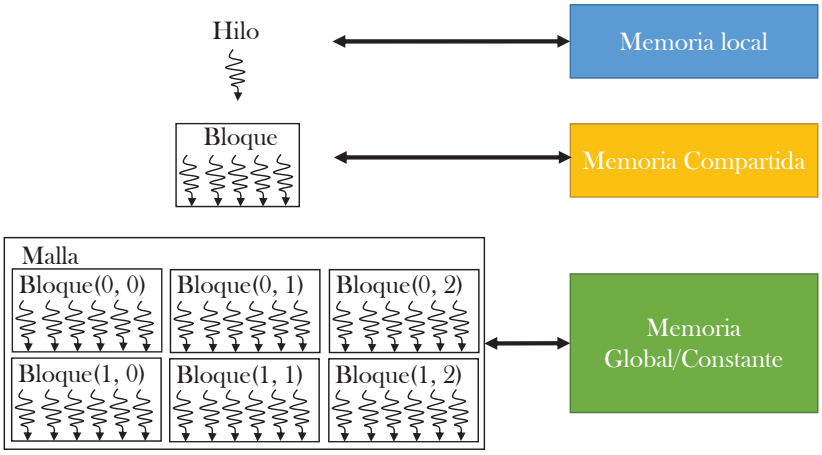


Figura A.2: Jerarquía de memoria del GPU en CUDA, donde puede observarse la memoria global, constante, compartida y local.

- Memoria global (*global memory*): en esta memoria se permite el acceso de lectura/escritura a todos los hilos de todos los bloques. Los hilos pueden leer y escribir cualquier elemento de un objeto de memoria de este tipo. Dependiendo de las capacidad del dispositivo, esta memoria puede poseer una caché, permitiendo realizar operaciones más rápidamente. El *host*, podrá escribir y leer en esta memoria para poder copiar la información al GPU.
- Memoria constante (*constant memory*): es una memoria global que se mantiene constante durante la ejecución de un *kernel*. El *host* tiene acceso a ella, y es el único que puede asignar e inicializar objetos en esta memoria.
- Memoria compartida (*shared memory*): la memoria compartida, es local a un bloque, y solo puede ser utilizada por los hilos pertenecientes a ese bloque. La velocidad de acceso a esta memoria desde un hilo es extremadamente más rápida que el uso de la memoria global y constante.

- Memoria local (*local memory*): es una memoria privada para un hilo. Las variables definidas en memoria local de un hilo no son visibles para otro hilo. 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: memoria de superficie (*surface memory*) y memoria de textura (*texture memory*). Una memoria de superficie almacena una colección unidimensional de elementos, mientras que una memoria de textura es usada para almacenar texturas, *frame buffers* o imágenes de dos o tres dimensiones.

Los elementos de una memoria de superficie pueden ser datos escalares (enteros, flotantes), datos de tipo vectorial o una estructura de datos definida por el usuario. La ventaja de utilizar una memoria de superficie es que son almacenados en memoria de forma secuencial y puede ser accedidos mediante punteros. En cambio la memoria de textura solo puede ser accedida y modificada mediante las funciones que provee CUDA para este tipo de datos. Sin embargo, la memoria de textura tiene una caché incorporada, permitiendo accesos rápidos en patrones propios utilizados para acceder texturas.

CUDA también incorpora un tipo de dato llamado CUDA *arrays*, el cual es un espacio manejado solo por funciones de CUDA, que al igual que la memoria de textura, puede ser definido de una, dos o tres dimensiones.

Por otra parte, dependiendo de la tarjeta de video, el hardware gráfico tiene diferentes capacidades, por lo que los programas paralelos pueden ser optimizados tomando en consideración las capacidades específicas de la tarjeta en la cual se está programando. La idea es utilizar los recursos como la cantidad de registros, memoria compartida, de hilos por bloque, entre otros, para permitir la ejecución paralela de la mayor cantidad de bloques de hilos, ocupando de la manera más eficiente los *SM* (*Stream multiprocessor*), los cuales se encargan del manejo y ejecución de los bloques. Determinar la configuración perfecta del programa, puede requerir mucha planificación y prueba.

Tabla A.1: Algunas características de una tarjeta CUDA con capacidad de cómputo 3.0.

Característica	**valor**
Número máximo de bloques residentes por SM	16
Número máximo de hilos residentes por SM	2048
Número máximo de *warps* residentes por SM	64
Número de registros por bloque	$64k$
Máxima capacidad de memoria compartida por SM	48 KB

Como ejemplo, para una tarjeta CUDA con capacidad de cómputo 3.0 se tienen las características mostradas en la Tabla A.1. Al diseñar un programa para esta tarjeta de video específica la idea es tratar de mantener los 2048 hilos que contiene un *SM* ocupados, y tratar de tener la mayor cantidad de bloques activos al mismo tiempo en el mismo *SM*. Por ejemplo, con bloques de 512 hilos es posible tener 4 bloques activos en un *SM* al mismo tiempo, si cumple

con las otras restricciones. Podría darse el caso de que tengamos bloques de 512 hilos, pero cada uno de ellos utilizando 24 KB de memoria compartida, lo que originaría que solo se puedan tener 2 bloques (2 bloques con 24 KB de memoria compartida ocupan los 48 KB totales disponibles para el *SM*), teniendo una ocupación del *SM* del $50\,\%$. Dado estos ejemplos, se puede observar que hay que realizar una configuración que trate de optimizar la ocupación tomando en cuenta la cantidad de hilos por bloques, la cantidad de memoria compartida por bloque, y los registros a utilizar. Esto tiene el beneficio de que a mayor ocupación de los *SM*, se tiene mayor paralelismo, ya que se encuentra una mayor cantidad de hilos corriendo al mismo tiempo de forma paralela. Sin embargo, en la práctica una mayor ocupación no simboliza necesariamente mayor eficiencia, pero al menos se garantiza el paralelismo de la solución.

Finalmente, para poder utilizar CUDA con una aplicación de despliegue gráfico en OpenGL, se necesitan utilizar instrucciones que permitan su interoperabilidad. Esto implica que posiblemente el paso de información entre CUDA y OpenGL sea costoso, ya que no manejan la memoria de la misma manera. Sin embargo, en aplicaciones donde no se realicen cálculos paralelos en todos los *frames*, el costo de la interoperabilidad es despreciable comparado con el costo constante del despliegue.

A.2. OpenCL

OpenCL (*Open Computing Language*) es un estándar abierto de programación paralela usando el GPU, mantenida por el grupo Khronos. Al ser abierto, no depende del hardware o sistema operativo en el que es utilizado, por lo que es multi-plataforma.

A.2.1. Procesamiento

El API de OpenCL posee características similares al de CUDA. Está constituido por un *host* (CPU) que se conecta a uno o más *devices* (GPU o CPU). Un *device* es el encargado de ejecutar el código paralelo a través de un *kernel*. El *host* se encarga de definir el contexto y administrar la ejecución. Cada *device* puede contener varias unidades de cómputo (*compute units*), las cuales pueden ser descompuestas en varios elementos de procesamiento.

Cuando un *kernel* es instanciado, pasa a ser un ítem de trabajo (*work-item*), ejecutando el mismo *kernel* pero en diferente data. A su vez, los ítems de trabajo se organizan en grupos de trabajo (*working group*), el cual es un concepto similar a los bloques en CUDA. Un grupo de trabajo es ejecutado de manera concurrente en un elemento de procesamiento de una unidad de cómputo. Finalmente los grupos de trabajos son conceptualmente ordenados en un espacio conocido como *NDRange*. *NDRange* puede ser concebido como una malla N-dimensional, la cual puede tomar los valores uno, dos o tres. Además, OpenCL también posee la capacidad de sincronización entre *work-items*.

A.2.2. Manejo de la Memoria

Para el manejo de la memoria de los dispositivos OpenCL hace uso de cuatro regiones de memoria accesibles por los ítems de trabajos, como podemos observar en la Figura A.3. Los tipos de memorias son:

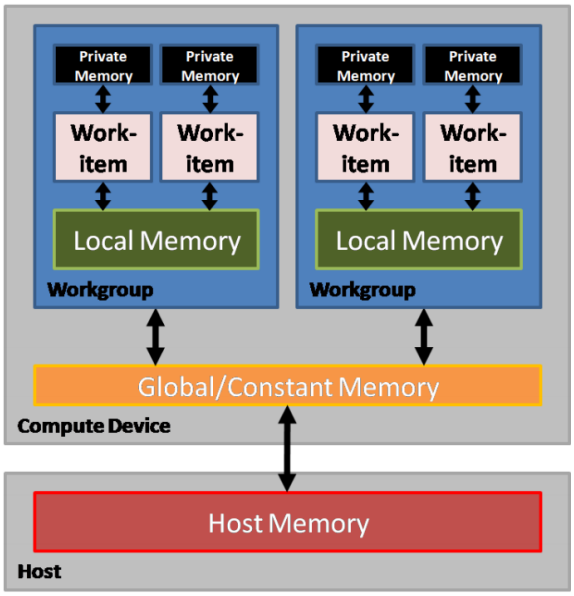


Figura A.3: Jerarquía de memoria de OpenCL. Imagen tomada de [70].

- Memoria global (*global memory*): es una memoria asociada a un *device* y todos los ítems de trabajo en todos los grupos de trabajo pueden leer y escribir en ella.
- Memoria constante (*constant memory*): es una memoria global que se mantiene constante durante la ejecución de un *kernel*.
- Memoria local (*local memory*): es una memoria compartida por todos los ítems de trabajo en un grupo de trabajo.
- Memoria privada (*private memory*): es una memoria privada para un ítem de trabajo. Las variables definidas en memoria local de un ítem de trabajo no son visibles para otro ítem de trabajo.

Adicionalmente OpenCL tiene soporte de dos tipos de objetos de memoria: *buffer* de objetos (*buffer object*) e imagen de objetos (*image object*). Un *buffer* de objetos almacena una colección unidimensional de elementos, mientras una imagen de objetos es usado para almacenar texturas, *frame buffers* o imágenes de dos o tres dimensiones. Los elementos de un *buffer* de objetos pueden ser datos escalares (enteros, flotantes), datos de tipo vectorial o una estructura de datos definida por el usuario.

La ventaja de utilizar un *buffer* de objetos es que son almacenados en memoria de forma secuencial y pueden ser accedidos mediante punteros. En cambio, las imágenes de objetos solo pueden ser accedidas y modificadas mediante las funciones que provee OpenCL para este tipo de datos. Al igual que en el caso de CUDA, OpenCL necesita instrucciones que permitan su interoperabilidad con OpenGL.

Por otro lado, a pensar de ser un API diferente al de CUDA, al final trabaja sobre el mismo hardware si se usa una tarjeta Nvidia, por lo que se pueden utilizar las mismas configuraciones para la cantidad de registros, memoria compartida, de hilos por bloque, entre otros, para tratar de optimizar el programa.

A.3. Uso de OpenGL para la Programación Paralela

OpenGL (*Open Graphics Library*) es un API multiplataforma, manejada por el grupo Khronos, que permite el despliegue de gráfico 3D, el cual hace uso del GPU para la aceleración por hardware. Aunque inicialmente no fue concebido como un API de cómputo general, con ciertas modificaciones, se ha podido utilizar para ello.

En sus inicios, OpenGL consistía de un *pipeline* de despliegue gráfico donde el usuario introducía datos geométricos como entrada y se producía una imagen de salida. Posteriormente, varias etapas de este *pipeline* han sido modificadas para permitir su programación en un lenguaje conocido como GLSL (*OpenGL Shading Language*). Entre las etapas modificables por medio de programas de sombreado (*shaders*), se encuentran: *vertex shader*, *tesselletion control shader*, *tesselletion evaluation shader*, *geometry shader* y *fragment shader* (ver Figura A.4).

Dado que esta librería permite el acceso al uso del GPU por medio de los *shaders*, los programadores comenzaron a realizar programas de cómputo de propósito general utilizando el *fragment shader*. Sin embargo, como este *shader* no está diseñado con este objetivo, realizar este tipo de cómputo tiene muchas limitantes. Por ello, recientemente el *compute shader* fue añadido a OpenGL como un nuevo programa de sombreado que permite realizar cómputo de propósito general.

En las siguientes subsecciones se va a explicar como se hace uso del *fragment shader* y el *compute shader* para el cómputo general. Además, se indicarán las ventajas y desventajas de cada uno.

A.3.1. *Fragment Shader*

Para entender el uso del *fragment shader* para cómputo general, primero debemos entender un poco el funcionamiento del *pipeline* gráfico. Hay que tomar en cuenta para un programa

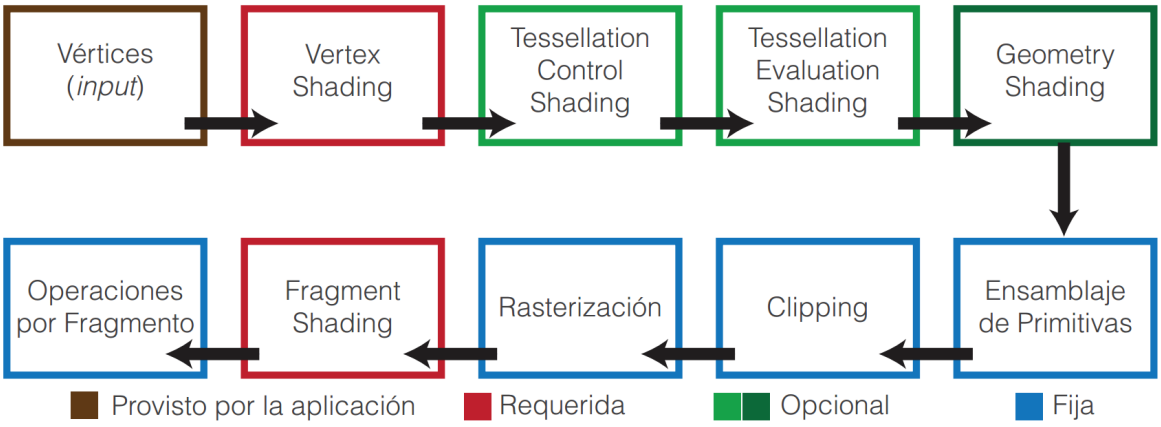


Figura A.4: Representación simplificada del *pipeline* de OpenGL mostrando sus etapas más importantes. En los cuadros con marco azul se encuentran las funcionalidades fijas; en rojo y verde las funcionalidades programables; y en marrón se encuentra la entrada del usuario al *pipeline*. El despliegue comienza cuando el usuario envía información a ser desplegada, pasando primero por el *vertex shader*, el *tessellation shader* (dos etapas) y *geometry shader* para luego llegar a la etapa de rasterizado. Esta etapa generará fragmentos para cualquier geometría que se encuentre en la región de recorte o *clipping*, y posteriormente se ejecutará un *fragment shader* por cada fragmento generado.

básico en el OpenGL moderno, es obligatorio utilizar GLSL con al menos el *vertex shader* y el *fragment shader*, por lo que no tomaremos las demás etapas de sombreado en consideración aquí. El proceso básico de despliegue puede ser visto en la Figura A.5. Inicialmente se tiene una geometría que quiere ser desplegada. Esta geometría está compuesta por vértices y uniones entre los vértices, las cuales terminan representando las figuras que se dibujarán. Cada uno de los vértices debe ser transformado desde el espacio de objeto al espacio de dispositivo. Para ello, OpenGL produce un programa *vertex shader* por cada vértice a ser procesado, el cual se encarga de hacer todas las transformaciones geométricas. Posteriormente, el hardware gráfico debe calcular la intersección de la figura proyectada con cada uno de los píxeles de la pantalla, y por cada píxel intersectado se produce un fragmento. Además, en cada fragmento los atributos de los vértices que forman la figura son interpolados. Este proceso es conocido como rasterización y no es programable. Una vez obtenidos todos los fragmentos, se ejecuta un *fragment shader* por cada uno de los fragmentos producidos, donde se une la información obtenida de la interpolación de los atributos de los vértices y se calcula el color final del fragmento. Finalmente, con los fragmentos resultantes se compone lo que es la imagen final.

Como podemos ver, tanto el *vertex shader* como el de *fragment shader* producen una cantidad de hilos de ejecución paralela. Sin embargo, el *vertex shader* no posee ninguna salida hacia el usuario, sino que produce transformaciones sobre los vértices, los cuales serán utilizados en el *fragment shader*. En cambio, el *fragment shader* tiene como salida el color del fragmento, el cual posteriormente será utilizado para pintar un *frame buffer*. El *frame buffer* por defecto es utilizado para el despliegue, pero el usuario puede indicarle a OpenGL que haga el despliegue

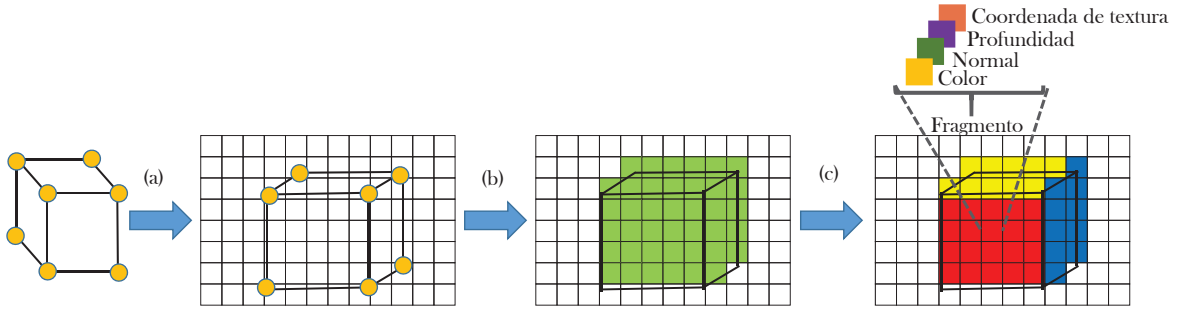


Figura A.5: Despliegue básico utilizando el *vertex shader* y el *fragment shader*. Como entrada se tiene una geometría (cubo) indicada por el usuario. En (a) los vértices son transformados con el *vertex shader*, para poder ser visualizados en el espacio de pantalla. Posteriormente, en (b) la geometría es rasterizada, se producen los fragmentos y se interpolan cada una de las propiedades de los vértices. Finalmente, el color final del fragmento se produce utilizando los atributos disponibles.

a un *frame buffer off-screen*, lo que resultará en hacer el despliegue sobre una textura. Además, es posible desplegar sobre varias texturas utilizando lo que se conoce como MRT (*Multiple Render Target*). De esta manera, es posible almacenar información que produce un *fragment shader* de tal manera que el usuario pueda posteriormente leerla y utilizarla.

Con lo explicado anteriormente, el proceso básico para utilizar el *fragment shader* para cómputo general puede observarse en la Figura A.6. La información a procesar de manera paralela debe ser codificada en uno o varias texturas OpenGL. Luego, un *buffer off-screen* debe ser creado para almacenar los resultados en una o varias texturas. En caso de utilizar varias texturas como resultado, se usa un MRT. Es importante que estos *buffers* posean un tamaño de resolución suficiente para almacenar el resultado del programa. Generalmente se colocan con la misma resolución que la textura que posee la información de entrada. Luego, para generar suficientes hilos paralelos para procesar la información, es necesario dibujar una geometría (que generalmente es un cuadrado) que ocupe toda la pantalla, para forzar a OpenGL a generar tantos fragmentos como téxeles tiene nuestra textura de entrada. Posteriormente, cada fragmento del cuadrado realizará un muestreo sobre la textura de entrada, decodificando la información almacenada en cada téxel para su procesamiento. Finalmente, el resultado obtenido es almacenado en el *frame buffer* asignando un color al fragmento.

Como puede observarse, todo este proceso no es el más natural para realizar cómputo general en la tarjeta de video, por lo que puede ser trabajoso configurar los *shaders* para realizar esta tarea. Además, el uso del *fragment shader* posee algunas desventajas con respecto a los demás APIs de programación paralela, entre las que se puede mencionar:

- El usuario no posee una manera directa de indicar cuantos hilos se van a ejecutar. Esto dependerá de cuantos fragmentos se generen.
- No existe una jerarquía accesible de memoria. Además, no se permite un agrupamiento de los hilos, y solo pueden ser trabajados en dos dimensiones.

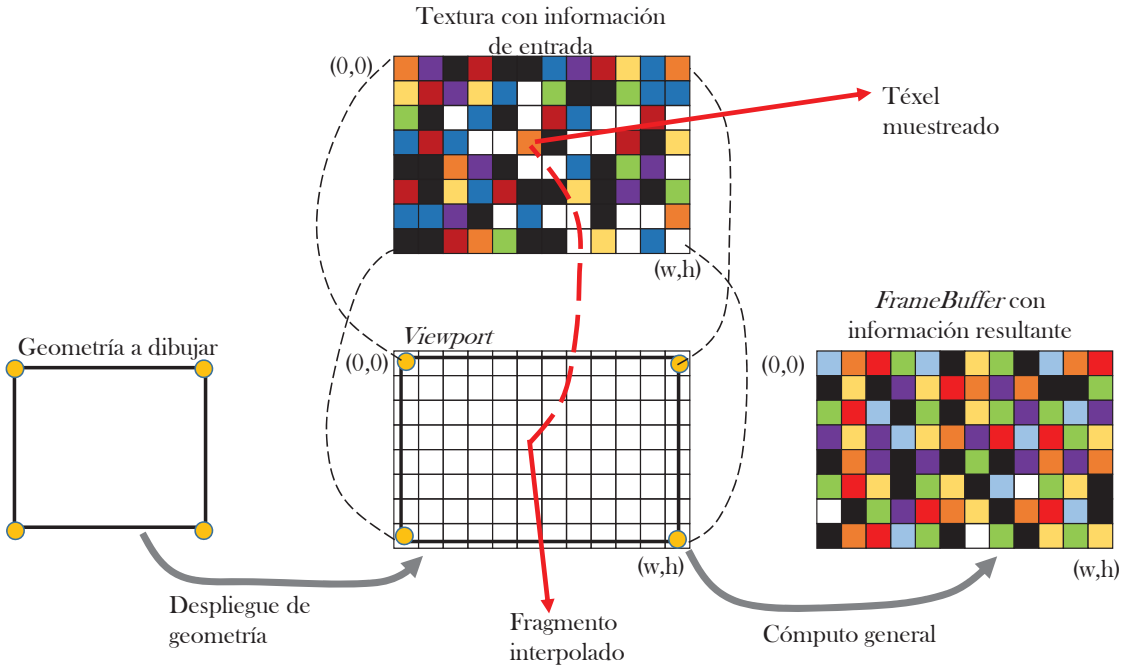


Figura A.6: Uso del *fragment shader* para cómputo de propósito general. Inicialmente se despliega un cuadrado que ocupe todo el *viewport*, para que se generen tantos fragmentos como píxeles totales en el *viewport*. Además, la información que quiere ser procesada de manera paralela es codificada en los colores de una textura. Las dimensiones de esta textura y las del *viewport* son las mismas, de tal manera de poder muestrear todos los valores de la textura. Finalmente, los cálculos de propósito general son realizados, codificando el resultado en el color final del fragmento el cual va a desplegarse.

- No existe comunicación ni sincronización entre los hilos. Esto obliga a que ciertos algoritmos paralelos básicos como *scan* y *compact* requieran muchas más llamadas que las que necesitaría una implementación en CUDA u OpenCL.
- Los datos de entrada y de salida deben ser codificados en texturas para poder ser trabajados en el *fragment shader*.
- Un *fragment shader* no puede escribir en las texturas que recibe como entrada, por lo que si el programa necesita actualizar los datos de entrada, se necesitará usar una textura *ping-pong*.
- Un hilo no podrá escribir en varias posiciones de una textura. En todo caso podría escribir en varias texturas diferentes utilizando MRT.

A.3.2. *Compute Shader*

El *compute shader* fue introducido en OpenGL con la versión 4.3. Consiste de un programa que permite utilizar el GPU y su paralelismo para cómputo general. Es utilizado en OpenGL

para tareas que son altamente paralelas pero no están ligadas al despliegue, como es el caso de las simulaciones físicas.

Aunque, como se mencionó anteriormente, ya hay APIs más especializados como CUDA y OpenCL para realizar GPGPU, estos están completamente separados de OpenGL. En cambio, el *compute shader* está directamente integrado en esta librería gráfica, lo que facilita la comunicación entre este programa y el resto del *pipeline* gráfico. Sin embargo, este *shader* no participar directamente en el proceso de despliegue, por lo que para ello depende de los otros *shaders*.

Sin embargo, su comportamiento no es igual a los demás *shaders*, ya que no responde a comandos de despliegue, e incluso cuenta con un conjuntos de instrucciones especializadas para su ejecución. A diferencia del *fragment shader*, el *compute shader* tienen características similares a CUDA o OpenCL. El número de invocaciones a este *shader* no está atado de ninguna manera al número de vértices o fragmentos a ser desplegados. La cantidad de invocaciones son definida por el usuario, el cual indica el número de grupos de trabajo a utilizar y el número de invocaciones dentro de cada grupo. Esto es similar a los bloques y los hilos de CUDA, respectivamente.

Además, estos *shaders* no poseen entradas predefinidas, sino que obtienen información directamente de la memoria utilizando diferentes funciones de acceso como las operaciones de lectura y escritura de imágenes, o vía *storage buffer objects*. Los resultados del programa son almacenados en los mismos objectos de entrada.

Por otro lado, a pensar de ser un API diferente al de CUDA, al final trabaja sobre el mismo hardware si se usa una tarjeta Nvidia, por lo que se pueden utilizar las mismas configuraciones de la cantidad de registros, memoria compartida, de hilos por bloque, entre otros, para tratar de optimizar el programa.

Apéndice B

Tabla Comparativa

La Tabla B.1 muestra un ejemplo de las pruebas realizadas para comparar el desempeño de los diferentes APIs paralelos para un solo dataset. Aquí todas las configuraciones de bloque son probadas, y se muestra el promedio del tiempo de ejecución en milisegundos para generar un *frame*. Una tabla similar fue generada para todos los datasets y todos los APIs paralelos considerados en la Sección 3.1 del Capítulo 3, para poder determinar la mejor configuración de bloque para el despliegue de dicho dataset. En este caso, podemos observar que la función de transferencia opaca (TF1) siempre es más rápida que la función de transferencia transparente (TF2), y que al añadir la iluminación siempre se ralentiza el proceso de despliegue. Además, la configuraciones de bloques con relación de aspecto $1:1$ o $1:2$, siempre presentan mejores opciones que configuraciones de bloques con relaciones de radio aspecto más rectangulares. La mejor configuración de bloque en este caso es 4×4 para el caso del volumen no iluminado, y 4×2 para el caso iluminado.

Tabla B.1: Tiempo por *frame* para el despliegue del dataset del hombre con resolución $512 \times 512 \times 1245$ usando *compute shader*. También se incluyen los resultados con iluminación.

Bloque	**TF**	**R**	**RB**	**R L**	**RB L**
4×1	**TF1**	108,53	107,50	118,89	116,81
	TF2	184,33	184,66		
2×2	**TF1**	102,13	102,45	111,93	109,83
	TF2	174,43	174,44		
8×1	**TF1**	64,08	63,82	82,19	81,33
	TF2	112,37	111,66		
4×2	**TF1**	58,46	57,91	66,34	65,87
	TF2	100,69	100,61		
16×1	**TF1**	66,92	67,11	166,13	170,38
	TF2	124,41	124,57		
8×2	**TF1**	41,69	41,36	96,59	97,95
	TF2	83,97	83,12		
4×4	**TF1**	37,91	36,99	80,28	81,12
	TF2	68,60	68,21		
32×1	**TF1**	107,43	107,80	282,62	279,19
	TF2	195,19	195,29		
16×2	**TF1**	77,30	77,53	191,44	190,53
	TF2	144,66	144,36		
8×4	**TF1**	59,37	59,74	146,71	147,27
	TF2	119,44	119,00		
64×1	**TF1**	120,91	121,23	330,48	322,41
	TF2	212,11	212,49		
32×2	**TF1**	119,05	119,61	326,05	319,22
	TF2	209,16	209,67		
16×4	**TF1**	93,69	94,10	241,13	234,87
	TF2	167,24	167,22		
8×8	**TF1**	75,95	75,65	191,10	188,76
	TF2	132,71	132,59		
128×1	**TF1**	123,33	123,69	348,21	347,23
	TF2	214,58	215,03		
64×2	**TF1**	121,99	123,14	363,13	362,94
	TF2	212,91	213,61		
32×4	**TF1**	121,30	122,22	363,70	363,79
	TF2	211,75	212,05		
16×8	**TF1**	97,40	99,27	273,41	272,89
	TF2	175,83	178,59		

Bibliografía

[1] P. Ljung, C. Lundström, and A. Ynnerman, "Multiresolution Interblock Interpolation in Direct Volume Rendering," in *Proceedings of the Eighth Joint Eurographics / IEEE VGTC Conference on Visualization*, ser. EUROVIS'06. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2006, pp. 259–266. 1, 2, 3, 10, 17, 33, 56, 57

[2] P. Bhanirantka and Y. Demange, "OpenGL Volumizer: a Toolkit for High Quality Volume Rendering of Large Data sets," in *Symposium on Volume Visualization and Graphics, 2002. Proceedings. IEEE / ACM SIGGRAPH*, Oct 2002, pp. 45–53. 1, 8

[3] E. LaMar, B. Hamann, and K. I. Joy, "Multiresolution Techniques for Interactive Texture-based Volume Visualization," in *Proceedings of the Conference on Visualization '99: Celebrating Ten Years*, ser. VIS '99, 1999, pp. 355–361. 1, 13, 15, 17

[4] K. López and R. Carmona, "Despliegue de Volúmenes Multi-resolución con Aceleración en GPU en el Cálculo de la Distorsión," in *II Simposio Científico y Tecnológico en Computación*, vol. 1, Caracas, Venezuela, May 2012, pp. 32–39. 1, 2, 3, 10, 11, 12, 16, 19, 20, 31

[5] J. Gao, C. Wang, L. Li, and H.-W. Shen, "A Parallel Multiresolution Volume Rendering Algorithm for Large Data Visualization," *Parallel Comput.*, vol. 31, no. 2, pp. 185–204, Feb. 2005. 1, 10

[6] Z. Fernández and A. Ramírez, "Visualización de Volúmenes Multi-resolución con Manejo Eficiente de la Segmenetación de la Textura Atlas," Trabajo Especial de Grado, Universidad Central de Venezuela, Caracas, Venezuela, May 2015. 1, 2, 3, 10, 12, 13, 16, 19, 20, 31, 33, 44

[7] R. Carmona, "Visualización Multi-Resolución de Volúmenes de Gran Tamaño," Ph.D. dissertation, Universidad Central de Venezuela, Caracas, Venezuela, 2008. 3, 6, 7, 9, 13, 16, 42, 79

[8] T. Fogal, A. Schiewe, and J. Krüger, "An Analysis of Scalable GPU-Based Ray-Guided Volume Rendering," in *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, Oct 2013, pp. 43–51. 3, 9, 10, 13, 16, 19, 31

[9] M. Hadwiger, J. Beyer, W.-K. Jeong, and H. Pfister, "Interactive Volume Exploration of Petascale Microscopy Data Streams Using a Visualization-Driven Virtual Memory Approach," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 12, pp. 2285–2294, Dec 2012. 3, 9, 10, 13, 16, 19, 31

[10] I. Jacobsen, M. Christerson, P. Jonsson, and G. Overgaard, *Object Oriented Software Engineering*, 1st ed. ACM Press, 1992. 3

[11] Git. git –fast-version-control. [Accedido: 04-05-2017]. [Online]. Available: http://git-scm.com/ 3

[12] Google. Google C++ Style Guide. [Accedido: 04-05-2017]. [Online]. Available: http://google-styleguide.googlecode.com/svn/trunk/cppguide.html 3

[13] ImGui. [Accedido: 04-05-2017]. [Online]. Available: https://github.com/ocornut/imgui 3

[14] F. Sans and R. Carmona, "Volume Ray Casting using Different GPU based Parallel APIs," in *2016 XLII Latin American Computing Conference (CLEI)*, Oct 2016, pp. 1–11. 4

[15] F. Sans and R. Carmona, "A Comparison between GPU-based Volume Ray Casting Implementations: Fragment Shader, Compute Shader, OpenCL, and CUDA." *CLEI electronic journa*, vol. 20, no. 2, August 2017. 4

[16] B. Preim and C. Botha, *Visual Computing for Medicine. Theory, Algorithms, and Applications*, 2nd ed. Elsevier, 2014. 5, 6

[17] P. Sabella, "A Rendering Algorithm for Visualizing 3D Scalar Fields," *SIGGRAPH Comput. Graph.*, vol. 22, no. 4, pp. 51–58, Jun. 1988. 6

[18] P. Lacroute and M. Levoy, "Fast Volume Rendering Using a Shear-warp Factorization of the Viewing Transformation," in *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '94. New York, NY, USA: ACM, 1994, pp. 451–458. 6, 7, 90

[19] H. Hans-Christian, H. Tobias, and S. Detlev, "Volume Rendering - Mathematical Models and Algorithmic Aspects," Konrad-Zuse-Zentrum Berlin, Berlin, Germany, Tech. Rep. TR 93-7, 1993. 7

[20] J. Krüger and R. Westermann, "Acceleration Techniques for GPU-based Volume Rendering," in *Proceedings of the 14th IEEE Visualization 2003 (VIS '03)*, ser. VIS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 38–. 7, 21

[21] Nvidia. CUDA. [Accedido: 04-05-2017]. [Online]. Available: http://la.nvidia.com/object/cuda_home_new_la.html 8, 92

[22] K. Group. OpenCL. [Accedido: 04-05-2017]. [Online]. Available: http://www.khronos.org/opencl 8, 92

[23] K. Group. OpenGL. [Accedido: 04-05-2017]. [Online]. Available: http://www.khronos.org/opengl 8, 92

[24] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel Computing Experiences with CUDA," *IEEE Micro*, vol. 28, no. 4, pp. 13–27, Jul. 2008. 8

[25] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi, "Evaluating Performance and Portability of OpenCL Programs," in *Proceedings of the Fifth international Workshop on Automatic Performance Tuning (iWAPT '10)*, Berkeley, USA, June 2010. 8

[26] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu, "An Application-centric Evaluation of OpenCL on Multi-core CPUs," *Parallel Comput.*, vol. 39, no. 12, pp. 834–850, Dec. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2013.08.009 8

[27] J. Fang, A. L. Varbanescu, and H. Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL," in *Proceedings of the 2011 International Conference on Parallel Processing*, ser. ICPP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 216–225. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2011.45 8

[28] K. Karimi, N. G. Dickson, and F. Hamze, "A Performance Comparison of CUDA and OpenCL," *CoRR*, vol. abs/1005.2581, 2010. 8

[29] N. Bombieri, S. Vinco, V. Bertacco, and D. Chatterjee, "SystemC Simulation on GP-GPUs: CUDA vs. OpenCL," in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '12)*. Tampere, Finland: ACM, 2012, pp. 343–352. 8

[30] C.-L. Su, P.-Y. Chen, C.-C. Lan, L.-S. Huang, and K.-H. Wu, "Overview and Comparison of OpenCL and CUDA Technology for GPGPU," in *2012 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, Kaohsiung, Taiwan, Dec 2012, pp. 448–451. 8

[31] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–10. 8

[32] R. Sachetto Oliveira, B. M. Rocha, R. M. Amorim, F. O. Campos, W. Meira, E. M. Toledo, and R. W. dos Santos, *Comparing CUDA, OpenCL and OpenGL Implementations of the Cardiac Monodomain Equations*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 111–120. 8

[33] T. I. Vassilev, "Comparison of Several Parallel API for Cloth Modelling on Modern GPUs," in *Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies*, ser. CompSysTech '10. Sofia, Bulgaria: ACM, 2010, pp. 131–136. 8

[34] N. Schubert and I. Scholl, "Comparing GPU-based Multi-volume Ray Casting Techniques," *Comput. Sci.*, vol. 26, no. 1-2, pp. 39–50, Feb. 2011. 8

[35] G. Kiss, E. Steen, J. P. Åsen, and H. G. Torp, "GPU volume rendering in 3D echocardiography: Real-time pre-processing and ray-casting," in *2010 IEEE International Ultrasonics Symposium*, Oct 2010, pp. 193–196. 8

[36] J. Mensmann, T. Ropinski, and K. Hinrichs, "An Advanced Volume Raycasting Technique using GPU Stream Processing," in *5th International Conference on Computer Graphics Theory and Applications (GRAPP '10)*, Anger, France, May 2010. 8

[37] A. Weinlich, B. Keck, H. Scherl, M. Kowarschik, and J. Hornegger, "Comparison of High-Speed Ray Casting on GPU using CUDA and OpenGL," in *Proceedings of the First International Workshop on New Frontiers in High-performance and Hardware-aware Computing*, vol. 1, November 2008, pp. 25–30. 8

[38] Z. Shi, C. J. Yi, and X. C. Hua, "Algorithm of Ray Casting Volume Rendering Based on CUDA," in *The 2nd International Conference on Industrial Application Engineering 2014 (ICIAE '14)*, Changshu, China, March 2014. 8

[39] E. W. Bethel, H. Childs, and C. Hansen, *High Performance Visualization – Enabling Extreme-Scale Scientific Insight.*, 2nd ed. Chapman & Hall, CRC Computational Science, 2012. 8

[40] J. Beyer, M. Hadwiger, and H. Pfister, "State-of-the-Art in GPU-Based Large-Scale Volume Visualization," *Computer Graphics Forum*, May 2015. 9

[41] C. Crassin, F. Neyret, S. Lefebvre, M. Sainz, and E. Eisemann, "Beyond Triangles: Gigavoxels Effects in Video Games," in *SIGGRAPH Talks 2009*. New York, NY, USA: ACM, 2009, pp. 78:1–78:1. 10, 13, 16

[42] X. Li and H.-W. Shen, "Time-critical Multiresolution Volume Rendering Using 3D Texture Mapping Hardware," in *Proceedings of the 2002 IEEE Symposium on Volume Visualization and Graphics*, ser. VVS '02. Piscataway, NJ, USA: IEEE Press, 2002, pp. 29–36. 10, 13, 15

[43] H. Younesy, T. Möller, and H. Carr, "Improving the Quality of Multi-resolution Volume Rendering," in *Proceedings of the Eighth Joint Eurographics / IEEE VGTC Conference on Visualization (EUROVIS '06)*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2006, pp. 251–258. 10

[44] R. Sicat, J. Krüger, T. Möller, and M. Hadwiger, "Sparse PDF Volumes for Consistent Multi-Resolution Volume Rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2417–2426, Dec 2014. 10

[45] C. Lux and B. Fröhlich, "GPU-Based Ray Casting of Multiple Multi-resolution Volume Datasets," in *Proceedings of the 5th International Symposium on Advances in Visual Computing: Part II*, ser. ISVC '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 104–116. 11, 12

[46] G. M. Morton, "A Computer Oriented Geodetic Data Base; and a New Technique in File Sequencing," Department of Computer Science, Michigan State University, Ottawa, Canada, Tech. Rep. MSU-CSE-00-2, 1996. 13, 32, 33

[47] D. Laur and P. Hanrahan, "Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering," *SIGGRAPH Comput. Graph.*, vol. 25, no. 4, pp. 285–288, Jul. 1991. 13

[48] I. Boada, I. Navazo, and R. Scopigno, "Multiresolution Volume Visualization with a Texture-Based Octree," *The Visual Computer*, vol. 17, no. 3, pp. 185–197, 2001. 13, 14, 15

[49] C. Wang, J. Gao, and H.-W. Shen, "Parallel Multiresolution Volume Rendering of Large Data Sets with Error-guided Load Balancing," in *Proceedings of the 5th Eurographics Conference on Parallel Graphics and Visualization*, ser. EGPGV '04. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2004, pp. 23–30. 13, 14

[50] P. Ljung, C. Lundstrom, A. Ynnerman, and K. Museth, "Transfer Function Based Adaptive Decompression for Volume Rendering of Large Medical Data Sets," in *Proceedings of the 2004 IEEE Symposium on Volume Visualization and Graphics (VV '04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 25–32. 13, 14

[51] C. Wang, A. Garcia, and H.-W. Shen, "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, Jan 2007. 13, 14, 42

[52] S. Guthe, M. Wand, J. Gonser, and W. Strasser, "Interactive Rendering of Large Volume Data Sets," in *Proceedings of the Conference on Visualization '02*, ser. VIS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 53–60. 13, 15, 17

[53] B. Chamberlain, T. DeRose, D. Lischinski, D. Salesin, and J. Snyder, "Fast Rendering of Complex Environments Using a Spatial Hierarchy," in *Proceedings of the Conference on Graphics Interface '96*, ser. GI '96. Toronto, Ont., Canada, Canada: Canadian Information Processing Society, 1996, pp. 132–141. [Online]. Available: http://dl.acm.org/citation.cfm?id=241020.241060 13, 15

[54] J. D. Foley, A. van Dam, S. K. Feiner, and john F. Hughes, *Computer Graphics: Principles and Practice*, 2nd ed. Addison-Wesley, 2017. 14, 41

[55] J. Plate, M. Tirtasana, R. Carmona, and B. Fröhlich, "Octreemizer: A Hierarchical Approach for Interactive Roaming Through Very Large Volumes," in *Proceedings of the Symposium on Data Visualisation (VISSYM '02)*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2002, pp. 53–ff. 16

[56] M. Weiler, R. Westermann, C. Hansen, K. Zimmermann, and T. Ertl, "Level-of-detail Volume Rendering via 3D Textures," in *Proceedings of the 2000 IEEE Symposium on Volume Visualization*, ser. VVS '00. New York, NY, USA: ACM, 2000, pp. 7–13. 17

[57] J. Beyer, M. Hadwiger, T. Möller, and L. Fritz, "Smooth Mixed-resolution GPU Volume Rendering," in *Proceedings of the Fifth Eurographics / IEEE VGTC Conference on Point-Based Graphics*, ser. SPBG'08. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008, pp. 163–170. 17

[58] R. Carmona, G. Rodríguez, and B. Fröhlich, "Reducing Artifacts Between Adjacent Bricks in Multi-resolution Volume Rendering," in *Proceedings of the 5th International Symposium on Advances in Visual Computing: Part I*, ser. ISVC '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 644–655. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-10331-5_60 18

[59] T. L. Kay and J. T. Kajiya, "Ray Tracing Complex Scenes," *SIGGRAPH Computer Graphics*, vol. 20, no. 4, pp. 269–278, Aug. 1986. 22

[60] T. Aila, S. Laine, and T. Karras, "Understanding the Efficiency of Ray Traversal on GPUs - Kepler and Fermi Addendum," NVIDIA Research, Tech. Rep., 2012. 22

[61] Levoy, Marc, "Display of Surfaces from Volume Data," *IEEE Comput. Graph. Appl.*, vol. 8, no. 3, pp. 29–37, May 1988. 30

[62] Google. Morton Encoding/Decoding Through Bit Interleaving: Implementations. [Accedido: 04-05-2017]. [Online]. Available: http://www.forceflow.be/2013/10/07/morton-encodingdecoding-through-bit-interleaving-implementations/ 33

[63] R. Carmona and B. Froehlich, "Error-Controlled Real-Time Cut Updates for Multi-Resolution Volume Rendering," *Computers & Graphics*, vol. 35, no. 4, pp. 931 – 944, 2011, semantic 3D Media and Content. 46

[64] M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein, "ROAMing terrain: Real-time Optimally Adapting Meshes," in *Visualization '97., Proceedings*, Oct 1997, pp. 81–88. 46

[65] U. N. L. of Medicine. Visible Human Project. [Accessed: 02-07-2015]. [Online]. Available: http://www.nlm.nih.gov/research/visible/visible_human.html. 60, 70

[66] Meister Eduard Gröller and Georg Glaeser and Johannes Kastner. (2005) Stag beetle. [Accessed: 03-08-2015]. [Online]. Available: https://www.cg.tuwien.ac.at/research/publications/2005/dataset-stagbeetle/ 60

[67] T. Aila and S. Laine, "Understanding the Efficiency of Ray Traversal on GPUs," in *Proceedings of the Conference on High Performance Graphics 2009*, ser. HPG '09. New York, NY, USA: ACM, 2009, pp. 145–149. 90

[68] P. Ljung, "Adaptive Sampling in Single Pass, GPU-based Raycasting of Multiresolution Volumes," in *Volume Graphics*, R. Machiraju and T. Moeller, Eds. The Eurographics Association, 2006. 90

[69] C. Nvidia, "Nvidia CUDA C Programming Guide," *NVIDIA Corporation*, vol. 120, p. 18, 2011. 92, 93

[70] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg, *OpenCL Programming by Example*, 2nd ed. Addison-Wesley, September 2012. 92, 97

[71] R. Tay, *OpenCL Parallel Programming Development Cookbook*, 1st ed. Packt Publishing Ltd., August 2013. 92

[72] D. Wolff, *OpenGL 4 Shading Language Cookbook*, 2nd ed. Packt Publishing Ltd., December 2013. 92