UNIVERSIDAD CENTRAL DE VENEZUELA
FACULTAD DE CIENCIAS
ESCUELA DE COMPUTACIÓN
CENTRO DE COMPUTACIÓN GRÁFICA

Visualización de volúmenes multi-resolución con manejo eficiente de la segmentación de la textura atlas

Trabajo Especial de Grado presentado ante

la ilustre Universidad Central de Venezuela

para optar al título de Licenciados en Computación

Br. Zilerimar Carolina Fernández Hensen

Br. Augusto Andrés Ramírez Colmenares

Tutor: **Prof. Rhadamés Carmona**

Caracas, Mayo 2015

UNIVERSIDAD CENTRAL DE VENEZUELA
FACULTAD DE CIENCIAS
ESCUELA DE COMPUTACIÓN
CENTRO DE COMPUTACIÓN GRÁFICA - CCG

ACTA

Quienes suscriben, miembros del jurado designado por el Consejo de la Escuela de Computación, para examinar el Trabajo Especial de Grado titulado **"Visualización de volúmenes multi-resolución con manejo eficiente de la segmentación de la textura atlas"** y presentado por los Brs. **Zilerimar Carolina Fernández Hensen (C.I. V-16.683.087) y Augusto Andrés Ramírez Colmenares (C.I V- 18.002.913)**, a los fines de optar al título de **Licenciados en Computación**, dejamos constancia de lo siguiente:

Leído como fue dicho trabajo, por cada uno de los miembros del jurado, se fijó el día __ de _____ de _____, a las _____ horas, para que el (los) autor(es) lo defendiera(n) en forma pública, lo que este (esta/estos) hizo (hicieron) en _____ de la Escuela de Computación, mediante una presentación oral de su contenido, luego de lo cual respondieron a las preguntas formuladas. Finalizada la defensa pública del Trabajo Especial de Grado, el jurado decidió aprobar con la nota de ____ puntos.

En fe de lo cual se levanta la presente Acta, en Caracas el día ___ de _____ de ____.

_____

Prof. Rhadamés Carmona

(Tutor)

_____ _____

Prof(a), Francisco Sans **Prof(a), Jaime Blanco**

(Jurado) **(Jurado)**

Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
CCG

Autores: Zilerimar Carolina Fernández Hensen y Augusto Andrés Ramírez Colmenares.

Tutor: Prof. Rhadamés Carmona.

Fecha: 15/05/2015.

RESUMEN

El despliegue de volúmenes es un área muy demandada en la actualidad. Con el desarrollo de nuevas tecnologías para la captura de volúmenes, se ha mejorado la resolución de los datos volumétricos, lo cual implica un aumento considerable en el tamaño de los mismos. Los computadores convencionales, e incluso las estaciones de trabajo, comúnmente no tienen la capacidad para desplegar estos volúmenes, a menos que se visualice solo una sub-área del volumen, o se utilicen técnicas multi-resolución. Entre las técnicas multi-resolución se destaca aquella basada en la jerarquía por bloques con despliegue de un pasada mediante la técnica de *Ray Casting*. Los bloques son almacenados en la memoria de tarjeta de video como un conjunto de texturas, empaquetadas en una única textura denominada textura atlas. Debido a que los bloques almacenados no son del mismo tamaño, la fragmentación de la memoria de textura es un aspecto importante a considerar, pues un uso ineficiente de esta memoria redundará en la pérdida de calidad global en el *rendering*. En este trabajo se presenta un algoritmo que mantiene un orden especial de los bloques en la textura atlas, el cual minimiza su fragmentación, y permite su fácil actualización conforme las prioridades de los bloques cambien entre cuadros de imagen. Los bloques son ordenados de forma decreciente dentro de la textura atlas a través de un algoritmo denominado *Z-Order*. Con este ordenamiento, las operaciones de refinamiento y reducción de resolución se efectúan eficientemente, con una fragmentación de la textura atlas insignificante. Se realizaron pruebas con volúmenes de gran tamaño, donde se obtuvieron resultados bastantes satisfactorios con respecto a tiempos de respuesta y fragmentación en la textura atlas.

Palabras Clave: despliegue de volúmenes, técnicas multi-resolución, *Ray Casting*, textura atlas, jerarquía por bloques, Refinamiento y Colapso, métrica de distancia, *Z-Order*.

Tabla de Contenido

Introducción

En el mundo de la computación gráfica existen diversas técnicas para visualizar volúmenes en una computadora convencional. Entre estas técnicas, tenemos dos vertientes: *Indirect Volume Rendering* (**IVR**) **[2]** o Rendering Indirecto de Volúmenes, en donde el volumen es desplegado mediante iso-superficies, y *Direct Volume Rendering* (**DVR**) **[4]** o Rendering Directo de Volúmenes, en donde se despliega el volumen usando trazos de rayos, tomando en cuenta la emisión y la absorción de la luz a lo largo del volumen. La técnica de **DVR** obtiene resultados de mejor calidad que en **IVR**, ya que permite definir características visuales como transparencia, color por material, etc., y además no se tiene limitaciones en cuanto a la cantidad de polígonos que pueden llegar a generarse cuando se despliega un volumen con la técnica de **IVR**.

Los volúmenes se consideran de gran tamaño cuando sobrepasan la capacidad de memoria disponible en el computador (ya sea la memoria del GPU o la memoria principal), y por ende requieren de un gran tiempo de procesamiento para su despliegue. Para estos volúmenes, se requieren de técnicas especiales para su procesamiento y despliegue. Diversos autores han propuesto algoritmos que pueden ser usados para lidiar con estos volúmenes. Como principales técnicas, tenemos *bricking* (despliegue por ladrillos) **[4]**, donde se particiona el volumen en un conjunto de sub-volúmenes de igual tamaño, en donde cada ladrillo sí puede ajustarse al espacio de GPU o CPU disponible. Así, el volumen es proyectado ladrillo a ladrillo, mientras se van mezclando para componer la imagen final. También se puede desplegar el volumen mediante un área de interés, en donde interactivamente podemos elegir una zona del volumen que se quiera visualizar. Otra solución consiste en desplegar el volumen mediante técnicas de despliegue multi-resolución. Estas técnicas asignan una resolución a cada área del volumen para luego ser desplegados. El pipeline gráfico para las técnicas multi-

resolución consta de la carga del volumen, generación de niveles de detalle por cada zona, criterio de selección de niveles de detalles y despliegue final del volumen. Las ventajas de las técnicas multi-resolución es que permiten darle mayor nivel de detalle a las áreas de interés en el volumen para el usuario y reducir el detalle en las áreas que no son de gran importancia para el mismo. Como principal desventaja, tenemos que estos algoritmos multi-resolución generan artefactos en la visualización final, además de requerir un overead en tiempo de cómputo para seleccionar las distintas resoluciones y hacer el despliegue.

R. Carmona **[4]** desarrolló un algoritmo óptimo polinomial, que obtiene la representación multi-resolución con mínimo error, cuando se utiliza una jerarquía multi-resolución de *Octree*. Implementa técnicas de aceleración como el salto de espacios vacíos y terminación temprana del rayo. K. López **[28]** propuso un algoritmo para desplegar volúmenes multi-resolución usando Ray *Casting* de una pasada, utilizando una jerarquía multi-resolución por bloques con tres niveles de detalles y una textura tridimensional denominada textura atlas, para almacenar los bloques a ser desplegados por el GPU.

En este trabajo se plantea utilizar una jerarquía por bloques similar a K. López [**28**], lidiando efectivamente con el problema de fragmentación de la textura atlas. Se propone utilizar un proceso de refinado compuesto por 2 sub-procesos encargados de realizar la reorganización de bloques dentro de la textura atlas, de manera tal de tener un control de la ubicación de cada bloque y cada espacio disponible en la misma. Estos sub-procesos determinan que bloques deben moverse para aprovechar los espacios de la mejor manera siempre intentando de minimizar la cantidad de bloques a mover. Como consecuencia de esto no es necesario implementar un algoritmo para realizar la compactación bloques en memoria.

Este trabajo especial de grado está estructurado por capítulos. En el primer capítulo se explica el problema principal para visualizar volúmenes multi-resolución utilizando la memoria disponible de manera adecuada y algunas propuestas para solucionarlo. El segundo capítulos explica los conceptos y definiciones necesarias para el desarrollo de la aplicación. El tercer capítulo describe la metodología y algoritmos realizados para la implementación de la aplicación. En el cuarto capítulo se presentan las pruebas

realizadas sobre 3 volúmenes, para luego finalizar con las conclusiones y trabajos futuros.

Capítulo I

Propuesta de Trabajo de Especial de Grado

En este capítulo se describe brevemente el problema principal del algoritmo de inserción de bloques en la textura atlas para volúmenes multi-resolución y se realiza una propuesta para su solución.

1.1 Situación actual

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 propuestos varios trabajos donde muestran una solución al despliegue de volúmenes usando técnicas multi-resolución. Con estas técnicas se logra hacer el despliegue del volumen en tiempo real; sin embargo, aún se presentan problemas de artefactos en la imagen final, y sub utilización de la memoria del GPU por fragmentación de la misma.

Para este trabajo especial de grado se ha tomado como caso de estudio el algoritmo de despliegue de volúmenes multi-resolución utilizando la técnica de *Ray Casting* de una pasada en GPU, utilizando una jerarquía por bloques. En esta técnica, el volumen multi resolución es almacenado en una textura de atlas, la cual es por lo general sub utilizada debido al problema de la fragmentación. La desfragmentación de la memoria puede limitar momentáneamente la interactividad en la visualización, por lo que definir una técnica que

evite la fragmentación de la memoria es de suma importancia. Evitando la fragmentación de la memoria atlas, permitirá mantener más información del volumen en la textura atlas, que redundará en una mejora en la calidad del rendering.

1.2 Planteamiento del problema

Existen diversas técnicas de almacenamiento para la visualización de volúmenes multi-resolución. Una de ellas utiliza una jerarquía por bloques. En una jerarquía por bloques, el volumen es inicialmente particionado en bloques de igual tamaño. Cada uno de estos bloques es representado en distintos niveles de detalle. Cada nivel de detalle tiene aproximadamente un octavo del espacio que ocupa su nivel de detalle inmediatamente superior. Así, por cada bloque del volumen original, se tiene una lista de bloques que representan el espacio ocupado por dicho bloque con distintos niveles de detalle. Durante el rendering, se toma la resolución adecuada de cada bloque según una métrica de error. Estos bloques son empaquetados en una única textura atlas (por ejemplo, de 0.5GB). Durante la interacción del usuario con el volumen, los bloques son reemplazados por otro nivel de detalle que requiere más o menos memoria que el bloque actualmente almacenado. Esto crea un problema de fragmentación de la memoria, con la desventaja de tener que realizar compactación o desfragmentación de la misma, ya que en el proceso de almacenamiento e inserción de bloques se desperdicia memoria que puede ser utilizada por otros bloques. Es por esto que surge la necesidad de mejorar el proceso de inserción, para utilizar el mayor espacio disponible de manera eficiente y minimizar el espacio perdido por la fragmentación en la memoria de textura. Por consiguiente, es de interés la implementación de un algoritmo de segmentación que permita organizar y mover bloques dentro del atlas, y que minimice la necesidad de desfragmentar dicha memoria.

1.3 Solución propuesta

En este trabajo especial de grado se plantea el desarrollo de una aplicación que permita el despliegue de volúmenes multi-resolución usando la técnica de *Ray Casting* de una pasada por GPU. Para el almacenamiento del volumen se desea implementar una jerarquía por bloques con distintos niveles de detalle; el volumen original es dividido en bloques y

cada nivel de detalle siguiente se obtiene de la interpolación de los bloques del nivel de detalle anterior.

Para la elección de bloques a desplegar se utilizará el punto de interés y la coordenada de ojo en espacio objeto como criterio de selección. Esto permite darle a cada bloque una prioridad calculada según ambas distancias. Una vez seleccionados los bloques a desplegar, se almacenan en una textura atlas indexada en otra textura de menor tamaño, conocida como textura de índices. Para la inserción de bloques en la textura atlas se implementará una indexación de bloques que permita su organización en la textura atlas de forma eficiente y permita aprovechar el máximo espacio disponible. Para el refinamiento y reducción de bloques se implementarán dos algoritmos que determinarán qué bloques deben ser movidos y en qué espacios deben ser reinsertados, siempre teniendo en cuenta que se debe hacer el mínimo de movimientos de bloques posible, sin afectar drásticamente la calidad visual del volumen.

1.4 Objetivo General

Desarrollar un sistema de despliegue de volúmenes multi-resolución almacenado en una jerarquía por bloques con la técnica de *Ray Casting* en GPU de una pasada, que permita organizar los bloques en una textura atlas usando la mayor cantidad de espacio disponible tratando de evitar la fragmentación de la memoria.

1.5 Objetivos Específicos

- Cargar en memoria el volumen y generar los niveles de detalles utilizando la jerarquía por bloques.
- Implementar el sistema de despliegue de volúmenes multi-resolución basado en *Ray Casting* en GPU de una pasada, basándose en un criterio de selección que tome en cuenta al menos la distancia a un punto de interés, refinando y reduciendo (*split* & *collapse*) bloques por una métrica de distancia.
- Implementar un algoritmo de inserción y reacomodo de bloques en la textura atlas que permita usar el espacio eficientemente, evitando la fragmentación de memoria

durante el proceso de reducción y refinado de bloques, minimizando la cantidad de movimientos de bloques.

- Realizar pruebas de rendimiento del sistema de despliegue, incluyendo la generación de los niveles de detalle, el algoritmo de segmentación y rendering.

1.6 Alcance

- Se requiere una tarjeta gráfica con soporte de texturas 3D, *OpenGL*® versión 4.0, *y* GLSL (*OpenGL Shading Language*) versión Y.Y. .
- El tamaño máximo de los datos volumétricos se restringe a un máximo de 3 GB de memoria, por lo cual se requiere como mínimo 4GB de RAM disponibles para la aplicación.
- El sistema será desarrollado y probado en una plataforma basada en Windows 8.1 de 64bits, utilizando como lenguaje de programación C++ bajo el entorno de desarrollo Visual C++ 2012. Se utilizará OpenMP para realizar también algunos procesos en paralelo en el CPU.

Capítulo II

Marco Teórico

Hoy en día existen diversos algoritmos para poder visualizar volúmenes en una computadora convencional. En este capítulo se estudian los conceptos básicos para el despliegue de volumen, explicando cada uno de los pasos a seguir para su adecuada visualización, los cuales son la carga del volumen, clasificación de los datos y el despliegue utilizando la técnica de *Ray Casting*.

2.1 Visualización de Volúmenes

La visualización de volúmenes ha adquirido una gran importancia a través de los años, ya que es una técnica muy utilizada en diversos campos científicos. Como por ejemplo, la sismología, la medicina, la física, la química, entre otros. Incluso puede ser utilizada en áreas que necesiten simular estructuras multidimensionales haciendo uso del computador.

La visualización de volúmenes tiene como objetivo proyectar un conjunto de datos multidimensionales, llamado *dataset*, en un plano de imagen bidimensional (ver **Figura 2.1**) con el propósito de entender la topología de la estructura contenida en los datos volumétricos **[1]**. Además ofrece la capacidad de manipular la estructura de manera rápida y con gran precisión según los parámetros definidos por el usuario. La estructura representada por el *dataset* es lo que se denomina volumen.

Un volumen puede ser definido como un conjunto de muestras de una función escalar continua, representado por una malla regular, compuesta por un conjunto de celdas, y almacenado en un arreglo tridimensional de escalares **[4]**.

Figura 2.1: Visualización de un *dataset* de una iguana usando técnicas de visualización de volúmenes.

2.2 Vóxeles y Celdas

Como mencionamos anteriormente el volumen no es más que un conjunto de datos multidimensionales, dentro de un arreglo tridimensional. Este conjunto de datos puede ser manipulado como un arreglo de elementos volumétricos **[1]**; éstos son conocidos como *vóxeles* (acrónimo de **vo**lumen y pí**xel**). Un vóxel puede considerarse como un píxel en un espacio tridimensional. Las celdas se componen por grupos de 8 vóxeles y un conjunto de celdas forman el *grid* (mallado) del volumen (ver **Figura 2.2**). En este trabajo, se define un vóxel como un punto correspondiente a una muestra del volumen.

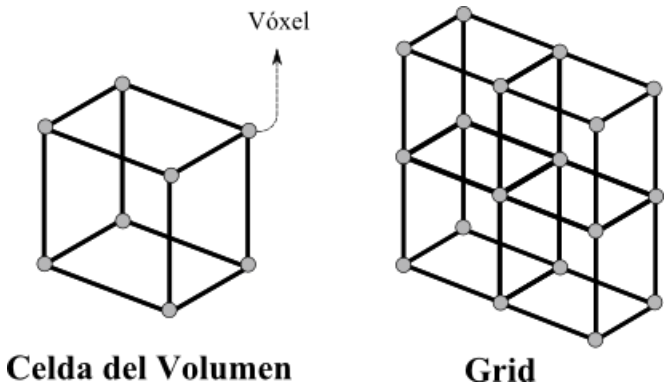


Figura 2.2: Representación gráfica de una Celda y un *Grid*

2.3 Tipos de Visualización

La técnica de visualización de volúmenes se puede clasificar en dos (2) tipos: (i) (**IVR**) *Indirect Volume Rendering* (Rendering Indirecto de Volúmenes) y (ii) (**DVR**) *Direct Volume Rendering* (Rendering Directo de Volúmenes). El primer tipo, **IVR**, consiste en transformar el volumen a una representación "intermedia" capaz de ser desplegada **[2]**. Esta representación "intermedia" es una aproximación a una superficie del volumen a través de la generación de geometrías (triángulos, cuadrados, etc.), es decir, El **IVR** es un proceso de correspondencia entre cada uno de los vóxeles y la geometría siguiendo una función de interpolación. Existen diversos algoritmos para el **IVR**, entre ellos tenemos: *Marching Cubes* (Cubos Marchantes), *Marching Tetrahedra* (Tetracubos Marchantes), *Contour-Connecting* (Conexión de Contornos) **[3]**, entre otros. El segundo tipo de visualización de volúmenes, **DVR**, consiste en desplegar el volumen sin hacer uso de la representación "intermedia" (ver **Figura 2.3**). Este trabajo está basado en **DVR**, el cual será explicado detalladamente en la siguiente sección **[4]**.

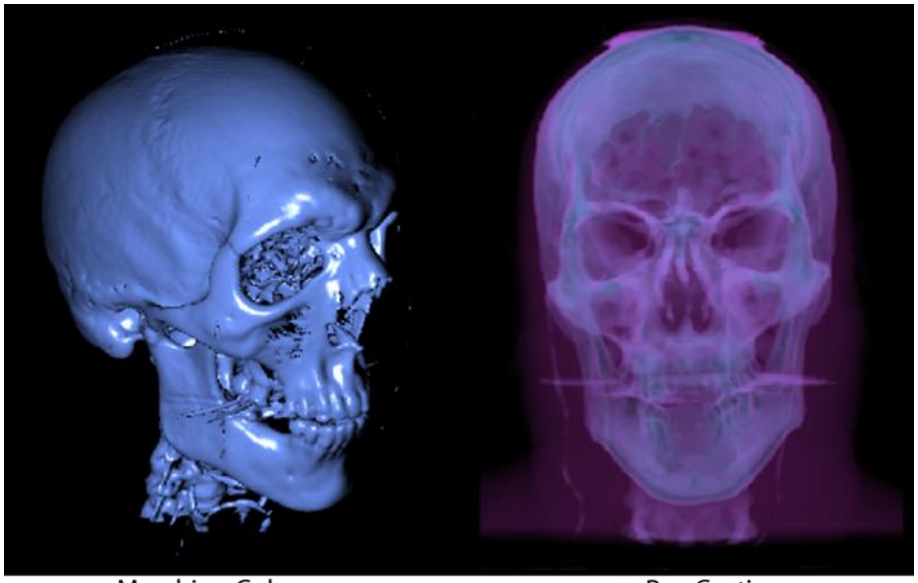


Figura 2.3: Despliegue de volúmenes usando Marching Cubes para IVR y el algoritmo de *Ray Casting* en GPU para el DVR.

2.4 Rendering Directo de Volúmenes

Este tipo de visualización (**DVR**) está basado en la composición de las propiedades visuales de un objeto semitransparente (volumen), simulando el paso de rayos de luz a través del mismo tomando en cuenta la emisión y absorción de la luz, con lo cual obtenemos los colores en cada píxel de la imagen a generar.

Para obtener el color de un píxel simulando el paso de un rayo a través del volumen se hace uso de la **Ec. 2.1 [4]**:

$$C = \int_0^D c(\lambda)t(\lambda)e^{-\int_0^\lambda t(\lambda')d\lambda'}\,d\lambda,$$

[Ec. 2.1]

donde C es el color resultante, D es la distancia que recorre el rayo dentro del volumen, $c(\lambda)$ y $t(\lambda)$ son el color y factor de absorción a una distancia λ de la entrada del rayo en el

volumen respectivamente. La integral representa la emisión de la luz desde la entrada del rayo ($\lambda = 0$) hasta la salida del mismo ($\lambda = D$). La exponencial representa el factor de extinción acumulado hasta ese punto **[3]**, que se puede interpretar como la transparencia $T(\lambda)$ del volumen hasta la distancia (λ) **[4]**. Note que con este factor de atenuación, a medida que el rayo avanza dentro del volumen se acumula más opacidad (menos transparencia) y la atenuación del color es mayor.

Como podemos observar, la **Ec. 2.1** es una ecuación continua, pero en el plano donde se proyectan los datos es discreto, y el volumen a su vez está dado por muestras discretas. Una manera de evaluar la ecuación es utilizar la *suma de Reimann* **[68]**, la cual a través de subdivisiones finitas rectangulares del área bajo la curva obtenemos una aproximación discreta de la ecuación.

Aplicando lo anterior obtenemos la siguiente ecuación discreta para la composición del color (ver **Ec. 2.2**) **[4]**:

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

[Ec. 2.2]

donde N representa la cantidad de muestras del volumen a evaluar, c_i representa la muestra i-ésima clasificada mediante una función de transferencia **[3]** y α_i es la opacidad de la muestra i-ésima en el rayo. En forma general, la opacidad acumulada la podemos definir como,

$$\alpha = 1 - T(\lambda),$$

[Ec. 2.3]

donde α se define como la opacidad acumulada del a una distancia λ**[4]**, donde $T(\lambda) = \prod(1 - \alpha_j)$.

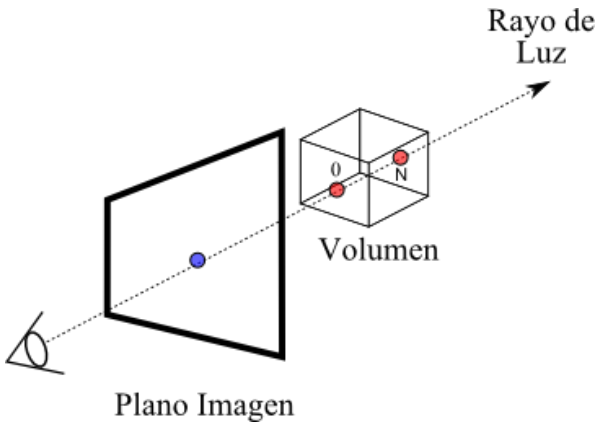


Figura 2.4: Trazo de un rayo de luz a través del volumen.

La **Ec. 2.2** se puede evaluar de dos maneras diferentes: *front to back* (de adelante hacia atrás) y *back to front* (de atrás hacia adelante). A continuación se describen estos dos enfoques.

2.4.1 Front To Back

Consiste en evaluar y acumular las muestras desde la más cercana a la más lejana. Esto puede expresarse mediante la **Ec. 2.4** de forma iterativa, donde C_n es equivalente al color C de la **Ec. 2.2**.

$$C_0 = 0, C_{i+1} = C_i + A_i \alpha_i c_i,,$$

$$A_0 = 1, A_{i+1} = A_i(1 - \alpha_{i-1}),$$

[Ec. 2.4]

C_i y A_i son el color y el factor de extinción acumulado respectivamente, después de evaluar i muestras **[3]**. El color final del píxel (C_n) es aquél que resulta de evaluar las N muestras.

2.4.2 Back To Front

Es el caso contrario de *front to back*; consiste en evaluar y acumular las muestras desde la más lejana a la más cercana esto puede expresarse iterativamente en la **Ec. 2.5**,

$$C_n = 0, C_i = \alpha_i c_i + C_{i+1}(1 - \alpha_i),$$

[Ec. 2.5]

donde C_i es el color acumulado cuando quedan i muestras por evaluar. El color final píxelC_0 es aquel obtenido cuando no quedan muestras por evaluar. Se puede demostrar que ambos procedimientos son equivalentes **[3]**.

2.5 Clasificación de los Datos Volumétricos

Para poder visualizar el volumen se requiere de la clasificación de los datos. Si estamos hablando de **IVR**, la clasificación se elige mediante un umbral para poder representar la superficie en el espacio 3D **[2]**. En el caso de **DVR**, a cada vóxel se le asigna su emisión c y absorción t, que se combinarán adecuadamente para generar la imagen final. Estas propiedades pueden ser asignadas al volumen mediante segmentación del volumen en partes específicas, o manipulando una función que típicamente es denominada función de transferencia **[28]**.

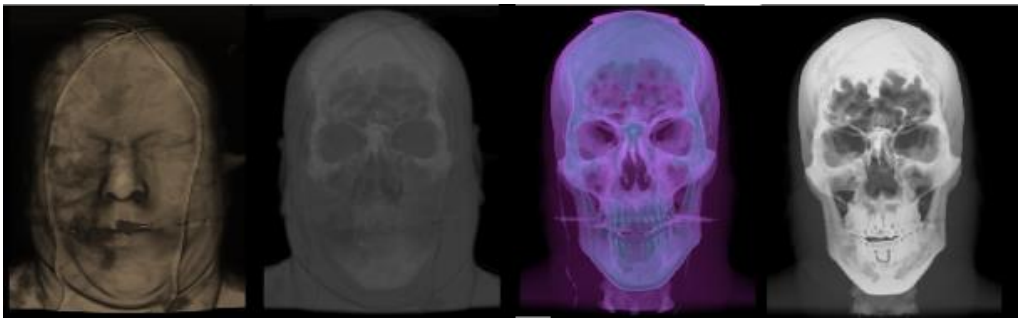


Figura 2.5: Un volumen proyectado con diferentes valores en la función de transferencia da como resultado la visualización de distintas partes del volumen.

La función de transferencia asigna el valor c y t dependiendo del valor escalar de cada vóxel en el volumen, dando como resultado un color (generalmente en formato $RGBA$) en cada muestra. Como podemos ver en la **Figura 2.5** con la función de transferencia podemos

visualizar diferentes partes de un volumen asignándole un color específico. Por lo general, se usan funciones de transferencia unidimensionales, que únicamente utilizan el valor del vóxel para su clasificación. También existen las funciones de transferencia multidimensionales, los cuales consideran entre otros parámetros a la normal del vóxel; estas permiten resaltar las fronteras entre los distintos materiales del volumen, pero requieren un nivel de complejidad mayor en su edición **[28]**.

Las funciones de transferencias unidimensionales son típicamente funciones lineales a trozos (Ver **Figura 2.6**), donde cada extremo de un trozo (punto de control) representa un color $RGBA$ con respecto al valor s que contiene el de un vóxel. El color y la absorción de los vóxeles que están entre un par de puntos de control son calculados mediante interpolación lineal por cada uno de los canales de color **[4]**.

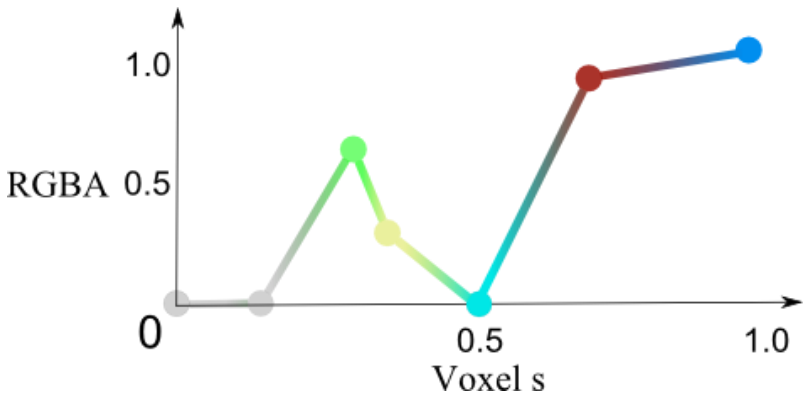


Figura 2.6: Los canales RGBA representan la coordenada $\boldsymbol{y}$ de la función, donde se puede apreciar que mientras más se aproxime a 0, más transparencia tiene el voxel. El eje $\boldsymbol{x}$ representa el voxel $\boldsymbol{s}$ evaluado en $\boldsymbol{s(x,y,z)}$ dentro del volumen. Entre cada par de puntos en la función de transferencia se realiza una interpolación entre cada canal de color.

En **DVR**, existen 3 tipos de clasificación de datos para el proceso de visualización de volúmenes. Pre-Clasificación, Post-Clasificación y Clasificación Pre-Integrada. A continuación se definirán cada una.

2.5.1 Pre-Clasificación

La Pre-Clasificación aplica la función de transferencia a los vóxeles del volumen antes de la interpolación entre sus muestras (como se observa en el trabajo **[4]** en la fase de reconstrucción), asignando a cada vóxel el valor $RGBA$ correspondiente de la función de

transferencia, dando como consecuencia la atenuación de las altas frecuencias en el despliegue final.

2.5.2 Post-Clasificación

La Post-Clasificación aplica la función de transferencia después de realizar la interpolación de las muestras escalares del volumen. Mediante esta técnica, se mantienen las altas frecuencias de la función de transferencia en la imagen final. La Pre y Post Clasificación producen diferentes resultados salvo cuando la función de transferencia es la función identidad, donde la interpolación de muestras puede conmutar con la función de transferencia **[5]**. Reescribiendo la **Ec. 2.1** para post-clasificación, se obtiene:

$$C = \int_0^D c\left(s\left(x(\lambda)\right)\right) t(s(x(\lambda)))e^{-\int_0^\lambda t(s(x(\lambda')))d\lambda'}\, d\lambda,$$

[Ec. 2.6]

donde $x(\lambda)$ representa un punto (x, y, z) a una distancia λ, y $s(x(\lambda))$ es la muestra interpolada del volumen en el punto $x(\lambda)$. De esta forma, se puede aplicar la función de transferencia para obtener su color *c* y absorción *t* **[4]**.

Siguiendo la **Ec. 2.6**, se debe discretizar el rayo para evaluar la integral numéricamente. Para esto, el rayo es dividido en $n = [D/h]$ segmentos, con un paso fijo h (Ver **Figura 2.7**) **[4]**.

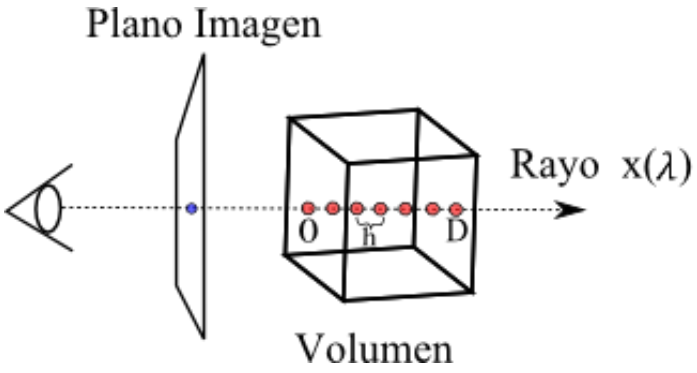


Figura 2.7: Discretizando el rayo con un paso *h*.

Una vez discretizado el rayo, se puede ver la **Ec. 2.6** como:

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

[Ec. 2.7]

Asumiendo que $s(x(\lambda))$ es constante en cada integral, $s\big(x(\lambda)\big)$ se sustituye por S_i, y asumiento que la función de transferencia es lineal a trozos, la **Ec. 2.7** se reescribe como:

$$C \approx \sum_{i=1}^{n-1} \prod_{j=0}^{i-1} e^{-ht(S_i)} c(S_i)(1 - e^{-ht(S_i)})$$

[Ec. 2.8]

Si sustituimos $c_{i=} \, c(S_i)$ y $\alpha_i = 1 - e^{-ht(S_i)}$ tenemos que:

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

[Ec. 2.9]

Si se quiere estudiar todo el proceso de desarrollo de la ecuación del rayo de luz en el volumen, se puede consultar el trabajo de Carmona **[4]**. Podemos evaluar la **Ec. 2.9** con el operador *under* iterativamente, tomando las muestras desde las más cercanas hasta las más lejanas del volumen **[8]** (*front to back*). La operación es definida de la siguiente manera:

$$c = (1 - \alpha)\alpha_i c_i + c$$

$$\alpha = (1 - \alpha)\alpha_i + \alpha$$

[Ec. 2.10]

En la **Ec. 2.9** también puede ser evaluada desde las muestras más lejanas hasta las más cercanas, con el operador *over* (*back to front*) **[9]**. De esta forma:

$$c = \alpha_i c_i + (1 - \alpha)c$$

$$\alpha = \alpha_i + (1 - \alpha_i)\alpha,$$

[Ec. 2.11]

donde c y α son valores acumulativos iniciados en 0. El operador *over* es el más usado debido a que saca mayor provecho del hardware gráfico, texturizando polígonos y desplegándolos desde el corte más lejano hasta el más cercano. Como ejemplo, estas operaciones son definidas en *OpenGL* mediante la función *glBlend* **[4]**.

2.5.3 Clasificación Pre-Integrada

Cuando aplicamos Post-Clasificación, en algunos casos, la cantidad de muestras procesadas en el rayo de visualización no son suficientes para reproducir las altas frecuencias de la función de transferencia. En este caso podemos aumentar significativamente la frecuencia de muestreo para captar todos los detalles de la función de transferencia, dando como resultado una imagen más fidedigna, pero con un rendimiento inferior. La clasificación pre-integrada, a diferencia de los otros tipos de clasificación, no evalúa cada una de las muestras individualmente. Busca pre-clasificar a segmentos delimitados por pares de muestras sf y sb, en donde por cada segmento cuantizado [s_f,s_b] se pre-calcula la integral del rayo y se almacena en una tabla bidimensional, donde posteriormente se puede acceder al color y a la opacidad dados los valores s_f y s_b (Ver **Figura 2.8**). Una forma eficiente de poder guardar esta tabla es en una textura 2D. Si la distancia h (distancia entre cada par de muestras) no es la misma (Muestreo Adaptativo) **[61]** se requiere una textura 3D para almacenar la tabla de pre-integración. En el trabajo **[4]** se puede conseguir en detalle todo el proceso de despliegue de volúmenes con una función de transferencia Pre-Integrada. Con la clasificación Pre-Integrada podemos ver que la calidad del rendering aumenta considerablemente con un pequeño *overhead* de procesamiento **[5]**.

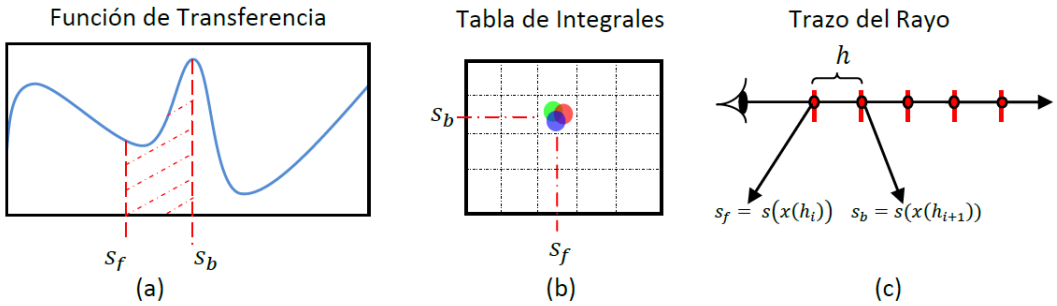


Figura 2.8: Despliegue de volúmenes utilizando clasificación pre-integrada. (a) Representaría el intervalo entre dos muestra en la función de transferencia. (b) son las integrales de cada canal (***RGBA***) entre todos los posibles pares de muestras s_f, s_b (c) Una ilustración de segmento de rayo cuyas muestras son s_f y s_b.

A continuación se mencionan las técnicas más comunes para el despliegue del volumen, tanto basados en software, como acelerados por el hardware gráfico.

2.6 Algoritmos para la Visualización de Volúmenes

Los algoritmos más usados para el despliegue de volúmenes son *Ray Casting*, *Splating* y *Shear-Warp* a nivel de software, aunque en investigaciones recientes se ha demostrado que pueden ser acelerados por hardware gráficos para mejorar el tiempo de respuesta. Las técnicas aceleradas por hardware están basadas en texturizado de polígonos **[4]**, en las cuales tenemos Planos Alineados al Objeto, Planos Alineados al *Viewport* y *Spherical Shells*.

En todas estas técnicas se pueden utilizar pre-clasificación, post-clasificación o clasificación pre-integrada para el despliegue del volumen.

En este trabajo se describe únicamente la técnica de *Ray Casting*, pues genera imágenes de gran calidad, y puede implementarse fácilmente con aceleración de GPU **[6]**.

2.6.1 Ray Casting

Este algoritmo básico de *Ray Casting* consiste en el lanzamiento de un rayo por cada píxel de la imagen final, desde el centro de proyección, atravesando el volumen, para evaluar la ecuación de composición volumétrica (e.g. **Ec. 2.9**) directamente. Al discretizar un rayo, se deben obtener las muestras escalares del volumen a lo largo del rayo, lo que

requiere de interpolación entre vóxeles para encontrar el valor de cada muestra. Las muestras clasificadas en $c(\lambda)$ y $t(\lambda)$ encontrados a lo largo del rayo se combinan para obtener la opacidad y el color del píxel. Las muestras dentro del rayo están distanciadas a una longitud constante h, aunque puede ser variable al considerar el muestreo adaptativo **[61]**.

El algoritmo de *Ray Casting* da como resultado una imagen de alta calidad pero presenta algunos inconvenientes. Entre estos está el problema de localidad espacial que se produce debido a que cuando se recorre el volumen en la dirección del rayo estamos accediendo a posiciones no contiguas en memoria, aumentando el tiempo de respuesta de despliegue. Este algoritmo puede ser acelerado por las técnicas de terminación temprana del rayo, muestreo adaptativo, entre otros **[61]**.

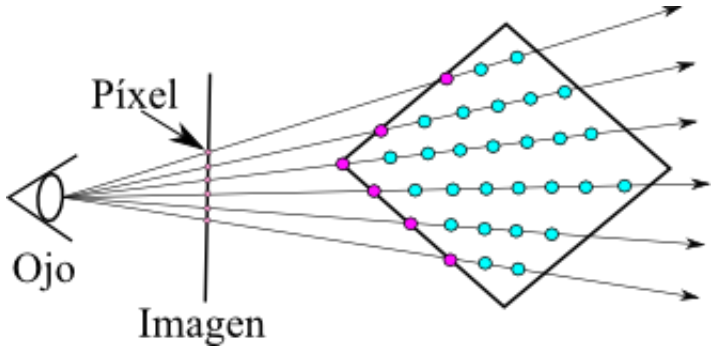


Figura 2.9: Recorrido del rayo desde la coordenada de ojo hasta cada posición final del volumen a desplegar. Los puntos morados representan la entrada del rayo.

El *Ray Casting* pude ser implementado tanto en software, como acelerada por hardware gráfico. Podemos ver en el trabajo **[6]** una técnica que aprovecha el procesador de fragmentos del GPU para realizar el algoritmo. Este algoritmo se puede dividir en 3 pasos. En el primer paso se determina el punto de entrada del rayo (Ver **Figura 2.9**). Para esto se despliega el *bounding box* del volumen como un cubo cromático. Al desplegar las caras frontales y traseras de este cubo, se obtiene posición inicial y final de cada rayo. En el segundo paso se determina la dirección de los rayos en el operador de fragmentos. Una vez que se tiene la dirección del rayo, se procede a recorrer el volumen para generar el color final por cada fragmento (Ver **Figura 2.10**).

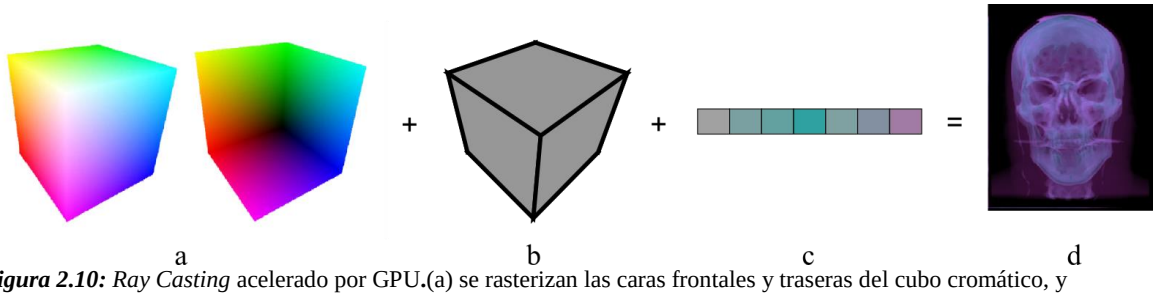


Figura 2.10: *Ray Casting* acelerado por GPU.(a) se rasterizan las caras frontales y traseras del cubo cromático, y considerando el volumen (b) como una textura 3D, y a la función de transferencia (c) como una textura 1D, se obtiene el resultado de *Ray Casting* en GPU en (d).

En el área de visualización de volúmenes, puede existir el caso de tener un *dataset* con miles de millones de datos, estos son conocidos como "Volumen de Gran Tamaño". Es por esto, que en el próximo capítulo se procederá a estudiar el despliegue de estos.

2.7 Volúmenes de gran tamaño

En la actualidad existen dispositivos de captura de imágenes de gran resolución, que pueden llegar a generar un volumen extremadamente grande. El procesamiento de estos volúmenes requiere de un algoritmo de mayor complejidad en computadores convencionales, debido a que el volumen puede superar en tamaño a la memoria principal y no es soportado por el hardware gráfico en memoria de textura. Algunos de los algoritmos para el despliegue de volúmenes de gran tamaño serán explicados a continuación en el documento, de los cuales tenemos la partición del volumen, área de interés y técnicas multi-resolución. Como ejemplos de estos volúmenes de gran resolución tenemos el proyecto del Humano Visible **[10]** y volúmenes obtenidos por simulaciones **[11]**.

2.8 Bricking

Bricking (despliegue por ladrillos) es una de las técnicas para desplegar un volumen de gran tamaño **[4]**. Consiste en particionar el volumen en pequeños sub-volúmenes llamados *bricks* (ladrillos), donde cada uno posee el mismo tamaño (Ver **Figura 2.11**). Para el despliegue se seleccionan típicamente los *bricks* desde el más lejano al más cercano en coordenadas de ojo, haciendo la composición de las muestras con el operador *over*.

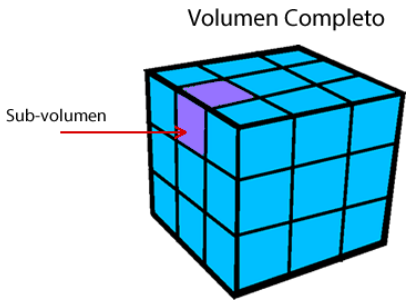


Figura 2.11: Técnica de Bricking. El volumen es dividido en sub-volúmenes que son desplegados desde el más lejano al más cercano. El cubo morado representa uno de los sub-volúmenes.

En esta técnica, cada *brick* se despliega independientemente. Sin embargo, esta independencia puede traer problemas en las fronteras de los *bricks*, pues para una consistente interpolación se requiere de los vóxeles frontera de *bricks* adyacentes. Debido a que un *brick* no tiene acceso a los *bricks* vecinos, se extiende el tamaño del *brick* para incluir los vóxeles fronterizos de cada *brick* adyacente. Como consecuencia, el tamaño total del volumen aumenta y produce cierto *overhead* (sobrecarga) en memoria (Ver **Figura 2.12**).

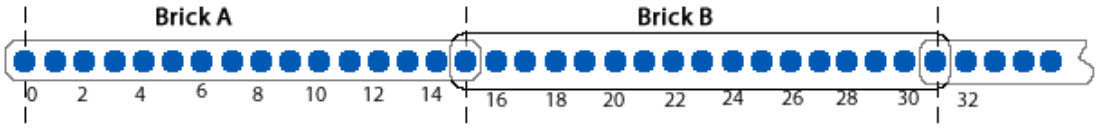


Figura 2.12: Los puntos azules representan los píxeles. Se puede observar cómo se comparte un vóxel en la frontera entre los *bricks* A y B. Por simplicidad se muestran los *bricks* en una dimensión.

Además de estas limitantes, tenemos también que esta técnica es muy poco práctica en tiempo real, debido a que se debe guardar en memoria principal cada uno de los *bricks* para subirlos a memoria de textura durante el *rendering*. Se sabe que el ancho de banda es limitado, y puede acarrear demoras en el tiempo de respuesta.

2.9 Volume Roaming

También conocida como despliegue de un área de interés, es una técnica donde solo se selecciona para visualizar una zona específica del volumen, no mayor al tamaño de memoria de textura **[12][13]**. Se puede seleccionar el área deseada de forma interactiva por el usuario, para luego ser cargada y procesada. El área de interés se especifica generalmente a través de un cubo, un punto o corte del volumen. Se puede acotar que al cambiar el área

de interés, ésta genera latencia en el despliegue, ya que se requiere de un nuevo sub-volumen en la memoria de textura. Para poder solucionar esto, se puede utilizar *Bricking* y coherencia *frame to frame* (cuadro a cuadro), ya que solo se cargan los nuevos *bricks* a ser desplegados y no todo un sub-volumen nuevo.

2.10 Visualización de Volúmenes Multi-Resolución.

Las técnicas de despliegue de volúmenes de gran tamaño más utilizadas son las llamadas multi-resolución, en donde se despliegan distintas zonas del volumen con distinto nivel de detalle en base a una prioridad, que incluye la capacidad de memoria.

Como podemos ver en la **Figura 2.13** los pasos a seguir para el despliegue de volúmenes multi-resolución se pueden resumir en de la siguiente forma: almacenamiento del volumen, organización en niveles de detalle, criterio de selección y despliegue del volumen.

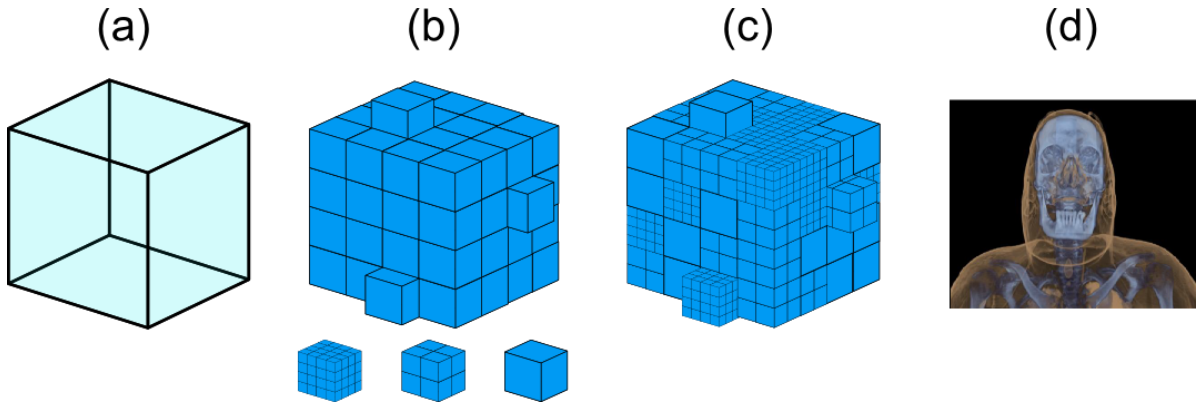


Figura 2.13: El proceso de despliegue de volúmenes utilizando técnicas multi-resolución consta de:(a) cargar el volumen en memoria, (b) almacenar los datos en una estructura de datos y organizarlo en niveles de detalles, (c) seleccionar los niveles de detalles a desplegar, y (d) desplegar del volumen en forma de multi-resolución.

2.10.1 Almacenamiento del Volumen

En la aplicación se debe almacenar el volumen de gran tamaño en una estructura de datos estable y eficiente para poder tener una visualización en tiempo real. Además debe soportar niveles de detalle. Entre los trabajos investigados se presentan principalmente dos jerarquías de almacenamiento del volumen: *Octree* (Árbol de ocho nodos hijos) **[14]** y *Blocks* (Bloques) **[16] [17] [28]**.

2.10.1.1 Jerarquía Octree

El volumen puede ser almacenado y organizado jerárquicamente en una estructura de datos *Octree* **[14][15]**, donde cada nivel del árbol corresponde un nivel de detalle del volumen. Cada nodo del árbol representa un *brick* del volumen a un determinado nivel de detalle. Los nodos hoja representan el nivel más fino de detalle, y los nodos internos se obtienen mediante la reducción del nivel de detalle de sus hijos (Ver **Figura 2.14**).

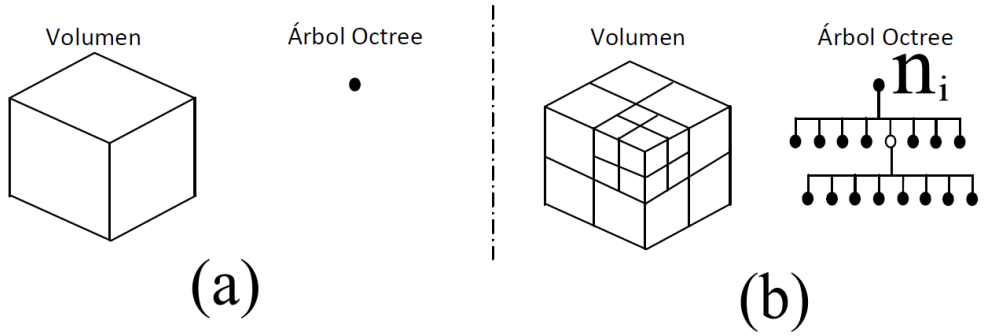


Figura 2.14: El volumen puede ser representado en un nivel de detalle burdo con un solo nodo del árbol *Octree* (a) o desplegando cada uno de los posibles cortes del *Octree* (b), donde cada nivel del árbol corresponde a un nivel de detalle del volumen. $n_i$representa el nivel del árbol. Los nodos hojas representan el nivel más fino de detalle.

2.10.1.2 Jerarquía por bloques

El volumen original se particiona inicialmente en bloques. Luego, cada bloque independientemente se sub-muestrea iterativamente generando todos sus niveles de detalle, hasta obtener el tamaño de bloque más pequeño que por lo general es 1x1x1 **[16][17]**. Para la visualización, el nivel de detalle que se le asigna a cada bloque es establecido a través de un conjunto de parámetros del criterio de selección. La ventaja de usar bloques, en comparación con *Octree*, es que los recorridos en la estructura de datos son más simples, y logra un particionamiento más fino en el volumen.

En esta jerarquía, la cantidad de vóxeles que tiene un bloque varía según el nivel de detalle (Ver **Figura 2.15**). Por el contrario, en la estructura del *Octree*, todos los bricks poseen la misma cantidad de vóxeles en todos los niveles del árbol, pero representando áreas de distintos tamaños dentro del volumen.

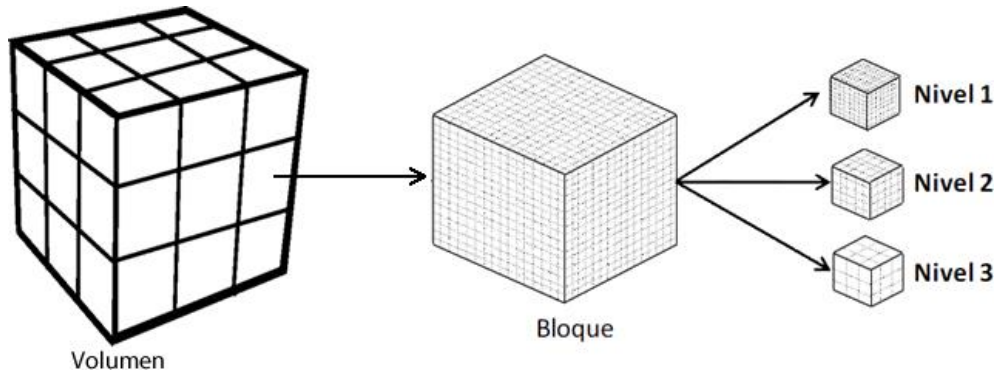


Figura 2.15: Ejemplo de una jerarquía por bloques. En un primer nivel tenemos la subdivisión en bloques del volumen, seguido se muestra los vóxeles de un bloque y tres niveles de detalles locales al bloque.

Tenemos que cada bloque de la estructura es independiente de los demás bloques, exceptuando la necesidad de compartir vóxeles vecinos en las fronteras de cada bloque. En la selección del LOD (*level of detail* o nivel de detalle), a cada bloque se le asigna una resolución independiente del resto de los bloques. En cambio, la jerarquía *Octree* no maneja los cambios de LOD por *bricks*, sino jerárquicamente por áreas representadas en la estructura de datos.

Para poder explicar de una forma sencilla el proceso por el cual se genera un bloque, se representará ésta en una estructura unidimensional en la **Figura 2.16**. Cuando se quieren generar todos los bloques en un volumen, se deben generar los LOD y almacenar cada uno de estos. Supongamos que el tamaño del bloque original es $n = 2^k$, para generar los siguientes bloques se debe ir dividiendo el tamaño en $\frac{n}{2}, \frac{n}{4}, \ldots$ y así sucesivamente hasta llegar al nivel de detalle donde un bloque ocupa un solo vóxel, teniendo $k + 1$ LOD.

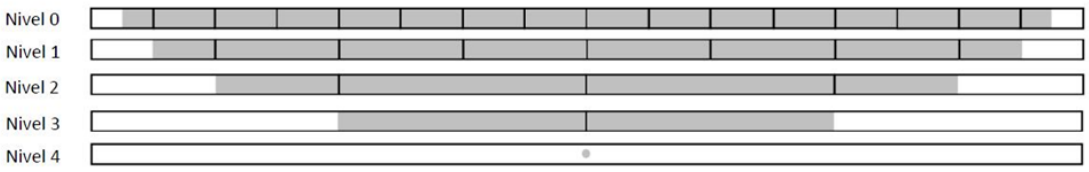


Figura 2.16: Un ejemplo de los niveles de detalles de un bloque 1D. La parte gris representa el área de la interpolación. El nivel 0 representa a la textura original y el 4 es el nivel de detalle más burdo. Así, cada vóxel de un nivel de detalle representa dos vóxeles de su nivel de detalle superior.

En la **Figura 2.16** se puede ver por cada nivel de detalle el dominio de interpolación el cual representa el bloque, en donde las fronteras que representan el dominio de

interpolación toman medio píxel de tamaño, ya que necesita el píxel de frontera del vecino para poder realizar la interpolación.

A nivel lógico, cada bloque representa el mismo espacio dentro del volumen. Lo que varía es la cantidad de píxeles de cada nivel, ocupando menos memoria para el despliegue. Por esta razón, basándonos en la **Figura 2.16**, hay que escalar cada uno de los bloques normalizando el dominio de interpolación en un rango de $[\frac{1}{2^n}, 1-\frac{1}{2^n}]$, donde n es el número de píxeles en el nivel de detalle. En la **Figura 2.17**, se puede ver cómo se lleva a cabo este proceso, dando como consecuencia que un píxel de un nivel de detalle es representado por más de dos píxeles del nivel de detalle superior, teniendo que generar un filtro muy complejo.

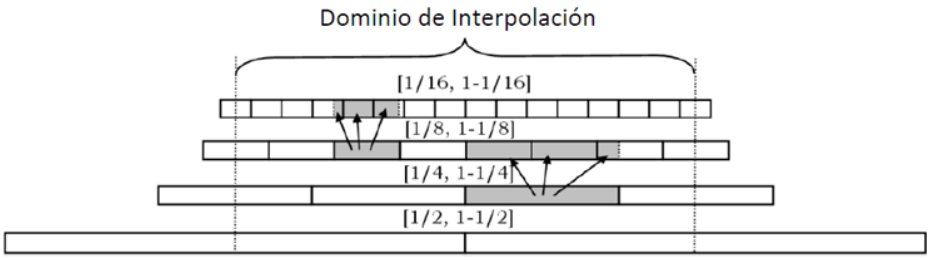


Figura 2.17: Niveles de detalles alineados al dominio de interpolación, por cada píxel del nivel de detalle intervienen **2** o **3** píxeles del nivel de detalles $\boldsymbol{d-1}$.

Este problema de gran complejidad se puede resolver si alineamos los bloques agregando un píxel de holgura. La resolución de un nivel de detalle d, no es exactamente la mitad del nivel de detalle $d-1$.

$$tamNivel(i,n) = (n\ div\ 2^i + 1)^3$$

[Ec. 2.12]

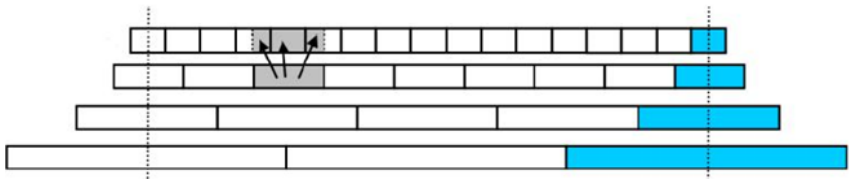


Figura 2.18: Niveles de detalle agregando un píxel de holgura por cada nivel de detalle.

Tenemos en la **Figura 2.18** que cada píxel del nivel d cubre un área de 2 píxeles del nivel $d-1$: el píxel central y dos medios píxeles (ver área grises). Los niveles $0, 1, 2\ y\ 3$ tienen $17, 9, 5, 3$ píxeles respectivamente. Siguiendo la **Ec. 2.12**, en cada nivel de detalle d se utiliza el píxel de holgura (ver áreas azules) para alinear los dominios de interpolación entre los niveles de detalle, y adicionalmente es compartido con el primer píxel del siguiente bloque para el proceso de interpolación entre vecinos.

A pesar de que esta solución no requiere de muchos cálculos, se debe utilizar un píxel de holgura en cada nivel de detalle en el caso 1D. Para el caso 3D, se puede ver en la **Ec. 2.13** que se requieren de $3 * n^2 + 3 * n + 1$ vóxeles de holgura por cada bloque.

$$(n+1)^3 - n^3 = (n^3 + 3 * n^2 + 3 * n + 1) - n^3 = 3 *\ n^2 + 3 * n + 1$$

[Ec. 2.13]

Entonces, si tenemos un bloque de 32^3 vóxeles, se requieren $3 * 32^2 + 3 * 32 + 1 = 3169$ vóxeles de holgura para el nivel de detalle más fino. Haciendo analogía, para un bloque de 16^3, tendríamos 817 vóxeles de holgura, para uno de 8^3, 217 de holgura, y así sucesivamente. Sumando los vóxeles de holgura para todos los niveles de detalle de un bloque, se obtiene 4290 vóxeles ocupando un 27% de memoria adicional **[28]**.

K. López utilizó únicamente el píxel central del nivel más fino sin aplicar ningún tipo de filtro (ver **Figura 2.18**). El objetivo de utilizar solo el píxel central es evitar la perturbación de los datos, ya que si se observa la **Figura 2.19a** el píxel central perdió su intensidad original cuando se mezcló con sus vecinos. Como consecuencia obtenemos un cambio en los datos provocando una clasificación errónea al momento de aplicar la función de transferencia. Este problema se puede observar con mayor frecuencia entre los contornos de algún objeto. Por eso al no mezclar con los vecinos, como en la **Figura 2.19b,** en el proceso de filtrado no se perturbarían los datos originales.

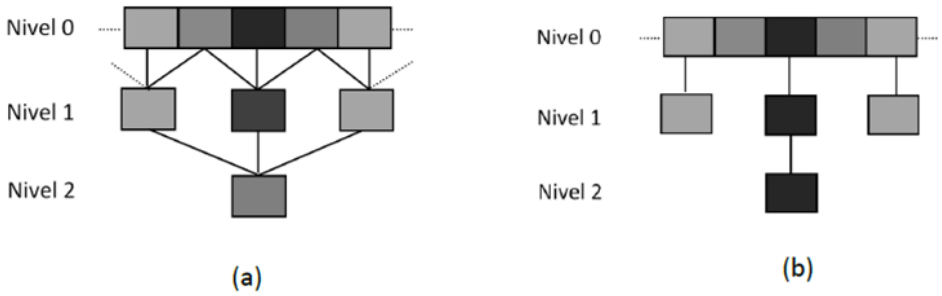


Figura 2.19: En la figura (a) se puede observar la generación de los niveles de detalle considerando los 3 píxeles que intervienen en el solapamiento. En la figura (b) los niveles de detalles son generados considerando solamente el píxel central. Observe como la intensidad del píxel central fue perdido por completo en la figura (a) **[28]**.

Ljung et. al [**26**] genera los bloques sin descartar el medio píxel en los bordes del dominio de interpolación. Esto es así ya que proponen un algoritmo para la interpolación de bloques entre fronteras, para evitar la redundancia de datos en la memoria Este algoritmo se divide en 2 etapas: en la primera etapa se despliegan de manera convencional los vóxeles que no forman parte de la frontera del bloque. En la segunda etapa, se despliegan los vóxeles que están en la frontera usando la interpolación entre bloques. Con esta técnica se le asigna un peso a cada uno de los bordes del bloque para hallar los pesos de cada bloque vecino. Con los pesos de los bloques ω_b y las muestras de cada bloque vecino φ_b podemos hallar el valor φ para generar la interpolación entre bloques. Para generar los niveles de detalles (**Figura 2.8**), Ljung et. al. [**26**] utiliza dos (2) píxeles en vez del píxel central y 2 medio píxeles de los píxel adyacentes ya que el dominio de interpolación no descarta los medio píxeles de las fronteras.

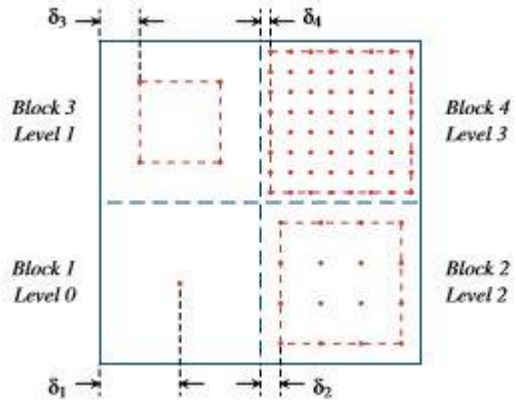


Figura 2.20: Niveles de detalle agregando un píxel de holgura por cada nivel de detalle.

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

2.10.2 Criterio de Selección

Después de almacenar el volumen en memoria y dividirlo en niveles de detalle, se debe seleccionar la resolución de cada área del volumen a desplegar. Los criterios que se deben tomar para seleccionar el nivel de detalle adecuado se indican mediante parámetros de visualización, como la distancia a un punto de interés, o a una región de interés. Otros criterios de selección son basados en los datos, como la distorsión de representar un área del volumen con un determinado nivel de detalle, la homogeneidad del *brick*, entre otros. Se debe tomar en cuenta las limitaciones del hardware para poder realizar este proceso, debido a que puede acaparar tiempo de cómputo necesario para hacer otras tareas como el despliegue. También se debe tener un mayor refinamiento de las áreas con mayor interés, sin perder tanta información del resto del volumen. A continuación se presentan trabajos donde explican las diferentes formas de realizar el criterio de selección.

LaMar et al. **[19]**realizaron la selección mediante dos aspectos, basado en la distancia con respecto a un punto de interés y la relación entre el ángulo de visión y el ángulo que genera la diagonal proyectada del *brick*. El algoritmo es recorrido en pre-orden; si el nodo esta fuera de la pirámide de visualización, no se despliega, en caso contrario, se despliega cumpliendo las siguientes condiciones (Ver **Figura 2.21**):

- La distancia desde el punto de interés al centro del *brick* debe ser mayor que la diagonal del *brick*, y el ángulo proyectado tiene que ser menor que la mitad del ángulo de visión.
- Se alcance un nodo hoja, ya que se alcanza el nivel de detalle con mayor resolución.

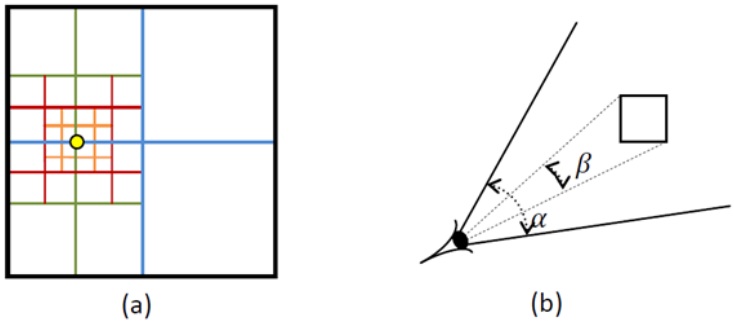


Figura 2.21: Criterio de selección basándose en la distancia al punto de interés y el ángulo de proyección del *brick*. En la imagen (a) se puede ver una representación de lo que sería un Octree en 2D. En este ejemplo, se seleccionaron sólo 34 *bricks*, cuando su representación más fina es de 256 *bricks*. En la imagen (b) se puede ilustrar la representación del ángulo proyectado del *brick* $\boldsymbol{\beta}$ y ángulo de visión $\boldsymbol{\alpha}$.

Boada et al. **[20]** proponen darle importancia a las regiones de interés del volumen, así como la homogeneidad del *brick* y al tamaño de memoria de textura. Se debe subdividir el volumen con el propósito de maximizar la obtención de los bloques homogéneos de mayor tamaño. Esto es para disminuir la cantidad de polígonos en el despliegue. Las áreas no homogéneas se representan con distintos niveles de detalles. Para el despliegue, se toman en cuenta un valor de predicción de la velocidad de despliegue del bloque, la opacidad del mismo, la distancia al ojo y el tamaño del bloque proyectado.

Guthe et al. **[21]**limitan la cantidad de bloques a ser desplegados en memoria de textura. Para esto se usa una cola de prioridad, donde se almacenan los bloques a memoria de textura. La prioridad P es otorgada a los bloques más cercanos al ojo Z y la medida de error del bloque E (distorsión), para tener la prioridad del bloque b como $P(b) = E(b)/Z(b)$. El brick con mayor prioridad es reemplazado por sus hijos. Este proceso es realizado hasta alcanzar el nivel de detalle más fino o llenar al tope la capacidad de memoria de textura El error es recalculado cada vez que cambia la función de transferencia.

Plate et al. **[22]**toma en cuenta la cantidad de *bricks* que se pueden cargar por cada *frame*, así como la capacidad de memoria de textura. Los *bricks* adyacentes que se seleccionan pueden diferir máximo en un nivel de detalle, teniendo como posición inicial las coordenadas del ojo. Usando coherencia *frame to frame*, se determina cuantos *bricks* serán cargados entre cada *frame*, con la finalidad de cargar los niveles más burdos primero, para luego ir refinando hasta alcanzar el umbral deseado que no exceda la capacidad de la memoria de textura.

Carmona **[4]** utiliza un algoritmo denominado *Split-and-Collapse*, un algoritmo voraz incremental en el campo de visualización de volúmenes multi-resolución, el cual utiliza coherencia *frame to frame* para actualizar la representación multi-resolución del volumen. Usa una función de prioridad, que indica la próxima operación a refinar *Split* (dividir) y reducción *Collapse* (colapso) a ejecutar para lograr la representación deseada. Cabe acotar que la cantidad de *bricks* que se transfieren al GPU en un frame dependerán del ancho de banda requerido. En este trabajo se consideran diversos parámetros para el criterio de selección, tales como la distancia al punto o área de interés, la distancia a la coordenada de ojo, la distorsión multi-resolución de los vóxeles clasificados y limitaciones de hardware tales como ancho de banda o capacidad de memoria de textura. Carmona **[4]** desarrolló en su trabajo un algoritmo óptimo para determinar la selección con mínimo error. A pesar de requerir mucho tiempo de cómputo, se utilizó para demostrar que su algoritmo voraz genera resultados cercanos al óptimo.

K. López [**28**] utilizó como criterio de selección la distancia con respecto a un punto de interés, la distorsión de los niveles de detalles considerando la función de transferencia y la unión de ambos. En un inicio los bloques son colocados en una cola de prioridad con el menor nivel de detalle. Los bloques se ordenan según una prioridad, que puede basarse en la distorsión y/o distancia al punto de interés. Luego, se aplica un proceso de refinamiento el cual consiste en extraer el primer bloque de la cola, subirle un nivel de detalle y reinsertarlo en la cola según su nueva prioridad. Este proceso es realizado hasta que se agote el espacio de la textura atlas o hasta que el volumen quede totalmente refinado. K. López [**28**] implementó un refinamiento *frame to frame* (cuadro a cuadro) similar al trabajo realizado por R. Carmona **[4]**. Es posible aplicar la unión de los criterios de distorsión y distancia al punto de interés, pero hay que considerar que la distorsión es proporcional a la prioridad, mientras que la distancia es inversamente proporcional. Así, la ecuación que mezcla ambos criterios es descrita en la **Ec. 2.14**:

$$prioridad = \frac{distancia + diag}{distorsión}$$

[Ec. 2.14]

Figura 2.21: En esta imagen se puede observar como a medida que transcurre el tiempo, el volumen va siendo refinado **[28]**.

En todos estos trabajos mencionados anteriormente, se genera una visualización multi-resolución del volumen, pero se generan muchos artefactos entre niveles de detalle. Por lo tanto, se deben desarrollar nuevas técnicas para minimizar los artefactos en estos volúmenes. A continuación se explicará con detalles como hacer el cálculo de las medidas de error para seleccionar los bricks con determinado nivel de detalle.

2.10.3 Métricas de Error

Estas métricas se utilizan para determinar la correcta selección de los niveles de detalles del *brick*. Están determinadas por varios parámetros, basado por los datos, en donde se mide la distorsión de los datos del volumen, basado en la imagen, que intenta captar la calidad perdida en la imagen final que percibe el usuario, y basado en la distancia a un punto de interés donde se da prioridad a los *bricks* más cercanos al punto.

2.10.4 Métricas Basadas en la Distancia

En este Trabajo Especial de Grado, se implementará la métrica basada en la distancia. A continuación se describen trabajos donde explican las diferentes formas de calcular las métricas de distancia.

Carmona **[4]** considera la distancia al ojo, el número de *bricks*, nivel de detalle y la distancia a un punto o a un área de interés. La función de prioridad ha sido definida de manera tal que: prioritize los *bricks* cercanos al punto de interés o área de interés, penalice la diferencia en nivel de detalle entre *bricks* adyacentes y priorice más suavemente a los *bricks* más cercanos al ojo, puesto que la prioridad principal es la distancia al punto o área de interés.

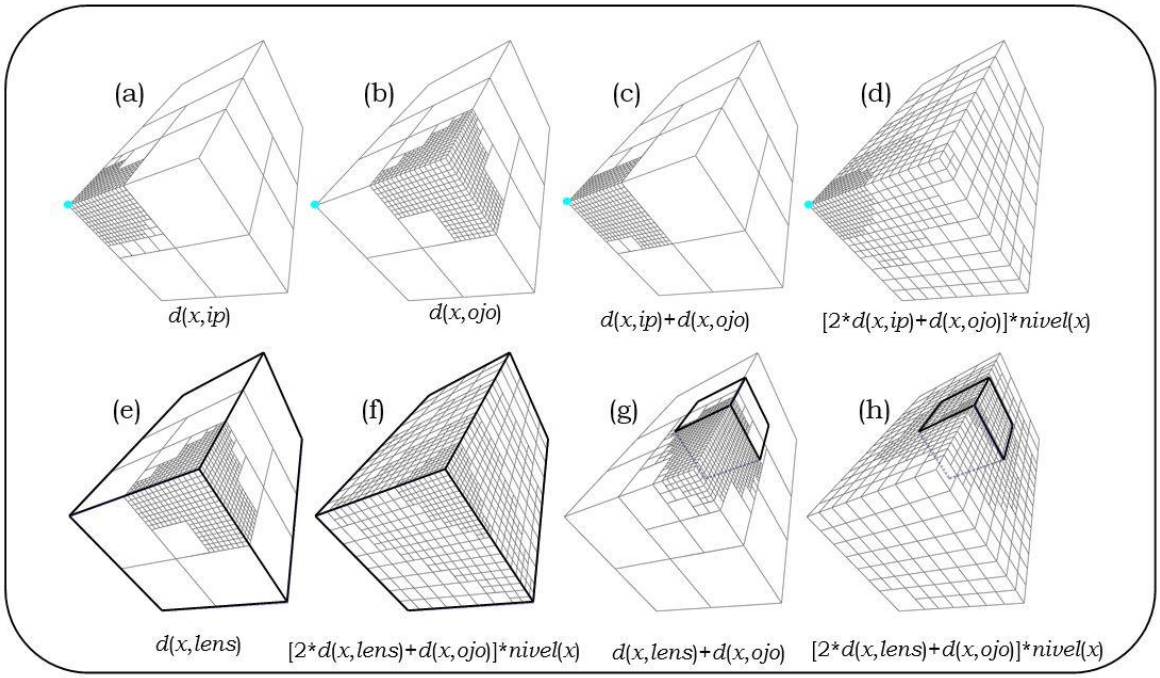


Figura 2.22: Criterio de selección basado en colas de prioridades. El valor de *d*(*x*,*y*)=*distancia*(*x*,*y*) representa la distancia del centro de *brick x* al objeto *y*, que puede ser el punto de interés o el *lens* o cubo de interés, o el ojo

En la **Figura 2.22** se muestran los volúmenes resultantes, dependiendo de la ecuación de métrica de distancia utilizada. Aceptables resultados han sido obtenidos mediante la siguiente función de penalización.

$$P(x) = (2 * distancia(x, ip) + distancia(x, ojo)) * nivel(x)$$

[Ec. 2.15]

en donde $distancia(x, y)$ es la distancia euclídea entre el centro del brick x y el objeto y, y $nivel(x)$ es el nivel del brick x en el *Octree* (0 para el nodo raíz que representa el nivel de

detalle más burdo). A menor penalización $P(x)$ mayor prioridad de refinar un brick x. Observe que los criterios (**Figura 2.22.a**) y (**Figura 2.22.c**) están reflejados en los términos $2 * distancia(x, ip)$ y $distancia(x, ojo)$, mientras que la diferencia de nivel de detalle en la multiplicación por $nivel(x)$.

K. López **[28]** considera que la prioridad de los bloques de menor resolución viene dada por la suma de la diagonal del bloque a dicha distancia (ver **Figura 2.23**). Debemos tener en cuenta que la diagonal es calculada en función al número de vóxeles del bloque.

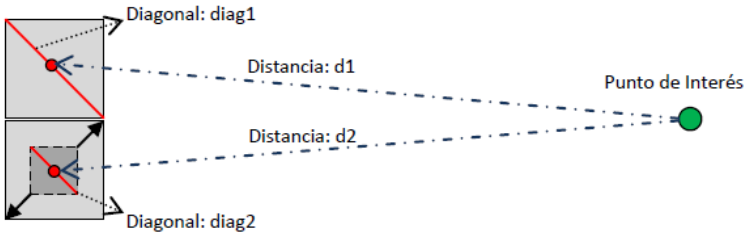


Figura 2.23: En la imagen se puede observar que la suma $\boldsymbol{diag1 + d1 > diag2 + d2}$, por lo tanto el bloque 2 tendrá mayor prioridad de refinamiento **[28].**

2.10.5 Proceso de Rendering

Para visualizar el volumen es necesario pasar por un proceso del despliegue el cual consiste en desplegar los datos que han sido seleccionados de manera ordenada. Luego, se aplica alguna técnica de visualización de volúmenes, (ver **Sección 2.6**). Existen varias formas de desplegar el volumen, entre las más usada se tiene el despliegue individual de cada *brick* (pero mezclándose en el búfer de color) **[4]** y el despliegue en una sola pasada **[28]**. Se puede mejorar su desempeño aplicando técnicas como terminación temprana del rayo y saltos de espacios vacíos **[61] [28]**.

Lux et al. **[32]** implementaron una forma diferente de almacenar el conjunto de texturas multi-resolución, al cual llamaron “Atlas”. Su sistema está basado en un (BSP) *Binary Space Partitioning* (Particionamiento de Espacio Binario) para realizar el despliegue de varios volúmenes en una misma escena, utilizando una jerarquía *Octree* para cada uno

de los volúmenes. Se construye una textura la cual es almacenada en el GPU (Atlas). En ella se organizan todos los *bricks* de los volúmenes seleccionados para el despliegue. Adicionalmente, crean un índice para identificar en qué posición del atlas se encuentra cada *brick* (ver **Figura 2.24**). Posteriormente utilizaron el algoritmo de *Ray Casting* basado en GPU, aplicando algunos ajustes para la visualización de varios volúmenes al mismo tiempo.

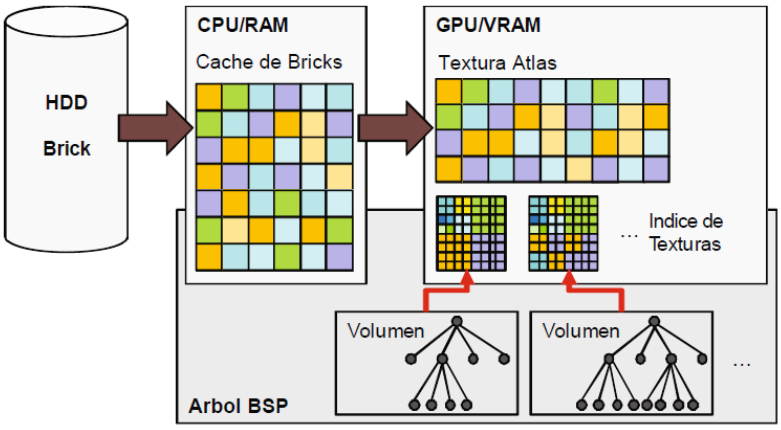


Figura 2.24: Atlas.

Debemos recordar que en una jerarquía *Octree* cada *brick* ocupa el mismo tamaño en memoria, sin importar el nivel de detalle que tenga. Por lo tanto no va haber fragmentación en la memoria de textura ni en la memoria principal a la hora de reemplazar los *bricks*.

K. López **[28]** adaptó el esquema propuesto por Lux et al. **[39]** a una jerarquía por bloques para generar la textura Atlas para el proceso de visualización. Este método consiste en tener los bloques que se van a mostrar almacenados en una sola textura e indexarlas en otra textura de índices (ver **Figura 2.25**).

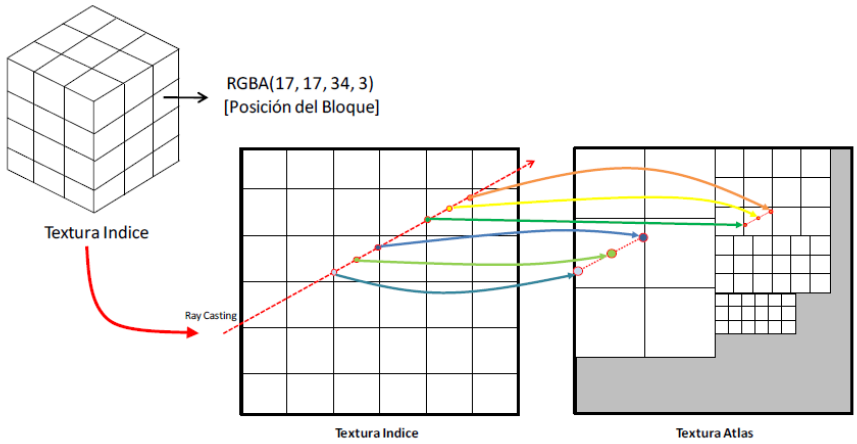


Figura 2.25: Cada vóxel de la textura de índices tiene una tupla *RGBA* (***px***, ***py***, ***pz***, ***lod***). Utilizando una representación 2D se puede observar la interacción entre la textura de índice y el atlas en cada paso del rayo.

La dimensión de la textura de índices viene dada por:

$$S(Tindice) = \left(\left[\frac{Sx-1}{n-1}\right], \left[\frac{Sy-1}{n-1}\right], \left[\frac{Sz-1}{n-1}\right]\right),$$

[Ec. 2.16]

donde Sx, Sy y Sz es el tamaño del volumen y n es el tamaño que va a tener el bloque de mayor nivel de detalle.

Una vez que todos los bloques son almacenados en la textura de atlas e indexados mediante la textura de índices, se realiza un *Ray Casting* de la textura de índices para recorrer adecuadamente la textura de atlas. Para realizar el *Ray Casting*, se le aplica la textura de índices a un cubo unitario y se despliegan las caras frontales para obtener el punto de entrada de cada rayo en el volumen **[6]**. La importancia de este método de visualización radica en que sólo se necesita realizar una pasada del *Ray Casting* para visualizar todo el volumen multi-resolución.

Para administrar el espacio de la textura atlas se generó un algoritmo básico que consiste en dividir la misma en distintas áreas dependiendo de los requerimientos del criterio de selección. Cuando se refina un bloque se reserva un espacio de memoria con la dimensión del nuevo nivel de detalle y se libera el espacio utilizado por el nivel burdo. Este algoritmo tiene como desventaja la fragmentación de la memoria Atlas que se genera a

medida que cambian los niveles de detalles. Como consecuencia se desaprovecha el espacio disponible por la desorganización de las áreas pequeñas que surgen de la subdivisión.

Como se puede observar hay distintas formas de realizar el despliegue de volúmenes multi-resolución. Sólo hay que tener en cuenta cuáles son los resultados esperados y saber que cada técnica tiene sus requerimientos de procesamiento que genera distintos tiempos de respuesta. Adicionalmente, la calidad de los resultados finales puede variar de técnica a técnica. Siguiendo con el punto anterior, se explicará a continuación cómo se fragmenta la textura atlas, cuando se utiliza una jerarquía multi-resolución por bloques.

2.11 Fragmentación del Atlas

En la **Figura 2.26** se puede ver con claridad la fragmentación que genera el algoritmo anteriormente descrito. Además, se puede observar que los espacios liberados por el refinamiento de bloques pasan a ser inutilizados si no hay más bloques por refinar que puedan ser almacenados en ese espacio disponible.

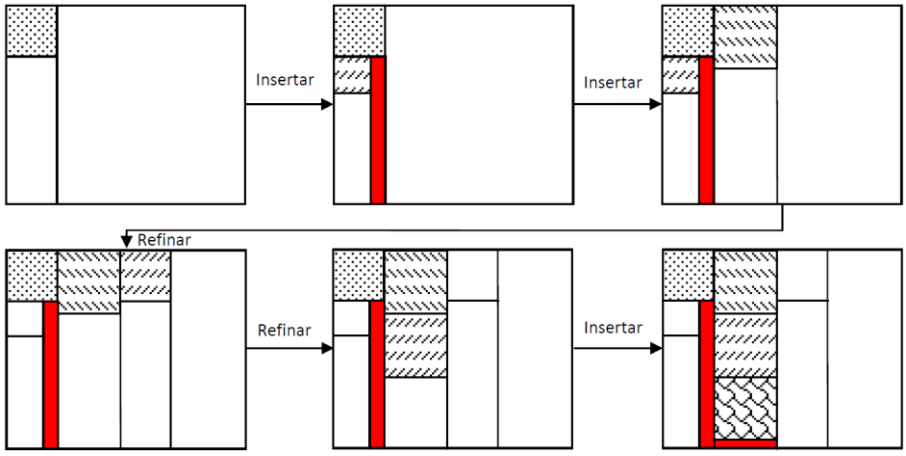


Figura 2.26: Proceso de inserción y refinamiento de los bloques en la textura Atlas. En la región roja se pueden notar las áreas que son demasiado pequeñas como para almacenar un bloque. En cada subdivisión se generan 2 nuevas áreas vacías, en el caso 3D se generarían 3 nuevas áreas.

K. López [**28**] desarrolló este método partiendo de la siguiente premisa, si el volumen multi-resolución no pudo ser refinado por completo y el atlas tiene espacio disponible pero fragmentado, se realiza un borrado y reinserción de todos los bloques, reagrupando los

bloques por orden de tamaño de manera descendente (del más grande al más pequeño). Posteriormente se continúa con el proceso de refinamiento. Esto se repite hasta que la cantidad de bloques que no pudieron ser refinados sea igual antes y después de una desfragmentación (ver **Figura 2.27**). El problema de esta técnica radica en que se cada vez que se realiza un borrado y reinserción de todos los bloques, se hace un uso excesivo del ancho de banda de la memoria de textura y genera ineficiencia en el despliegue.

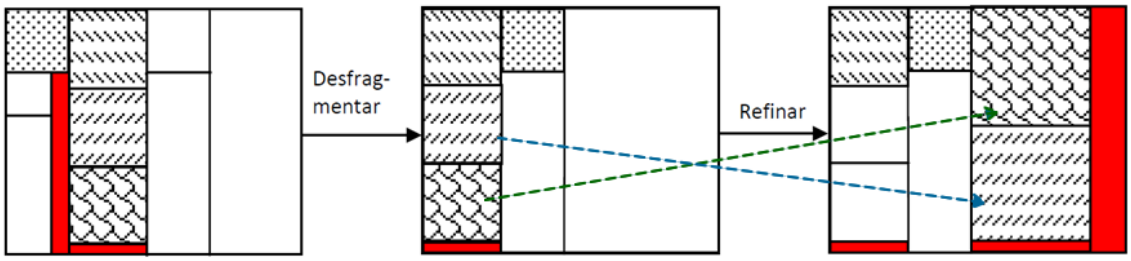


Figura 2.27: En la primera imagen podemos observar que no hay espacio suficiente para poder refinar más los bloques que tienen mayor tamaño. Aplicando el algoritmo de desfragmentación note que se acoplan los bloques permitiendo utilizar mejor el espacio restante del atlas y refinar más bloques. Las líneas de colores punteadas indican que bloques se han refinado luego de aplicar el algoritmo de desfragmentación.

2.12 Jerarquía de Memoria de Textura en GPU

Un factor importante que afecta al rendimiento de la GPU es el tiempo de acceso a los datos **[43]**. Por esto, es muy importante la gestión de memoria en las tarjetas gráficas de forma estructurada, eficiente y eficaz. Las jerarquías de memorias de las GPU's varían según el fabricante y el modelo de la tarjeta gráfica, así que se procederá a explicar de forma general por el fabricante Nvidia. Se deben tener en cuenta dos grandes aspectos en el manejo de memoria de textura: localidad espacial y el ancho de banda.

La memoria de textura permite almacenar arreglos 1D de color RGBA, imágenes 2D, volúmenes 3D, etc. Estas texturas pueden tener distintas funciones, pueden servir para darle mayor realismo a cualquier aplicación gráfica, almacenar información a ser procesada por el procesador de texturas, entre otros. Las tarjetas gráficas requieren un manejo sencillo de estas texturas, para que se puedan realizar todo tipo de transformaciones de tamaño y forma, para poder aplicarlas en un modelo tridimensional, sin que se vea afectado negativamente su aspecto final.

Además de almacenar un conjunto de valores de color RGBA, también se pueden almacenar los valores de luz de cada punto de textura, estas son llamadas texturas planas. También podemos usar los filtros que ofrece la tarjeta de video a nivel de hardware (ALUs) *Arithmetic Logic Units* (Unidades Aritméticas Lógicas) para hacer cálculo de redimensiones, filtrados de texturas, etc.

Las librerías más usadas para el uso de texturas son OpenGL y Direct3D. Ahora se procederá a explicar el manejo de texturas en OpenGL.

2.13 Usando Memoria de Textura con OpenGL

Hoy en día en los sistemas donde interviene el pipeline grafico se está buscando un mayor realismo en la imagen final de una escena. Para esto, OpenGL ofrece una solución para mapear texturas en las primitivas gráficas. *Texture Mapping* (Mapeo de Texturas) **[44]** tiene muchas funciones dentro de los sistemas gráficos. Su función más común es mapear una imagen 2D a un conjunto de primitivas para dar una sensación más real al render. También se pueden almacenar texturas 1D, que pueden ser usadas para almacenar un arreglo de colores en la memoria de textura, como por ejemplo la función de transferencia en Volume Rendering. Las texturas 3D también pueden ser almacenadas con OpenGL, pudiendo almacenar un *dataset* (conjunto de datos) de algún volumen.

Para usar mapeo de textura, se pueden seguir los siguientes pasos:

- Se crea un objeto del tipo *texture* (textura) y se especifica una textura para ese objeto.
- Se indica cómo va a ser aplicada la textura por cada píxel y los filtros a usar.
- Se habilita el mapeo de texturas.
- Se dibuja la escena, indicando las coordenadas geométricas y de textura.

Para almacenar una imagen o un volumen en la memoria de textura, se usa la función $glTexImagexD(\dots)$, donde xD es para diferenciar entre una textura 1D, 2D o 3D. Entre los parámetros a pasar a esta función tenemos el tipo de textura, nivel de textura usado (para múltiples resoluciones del mapeo de textura), formato de color de la textura, tamaño

del ancho y largo de la textura (por lo general debe ser en base a 2), el ancho del borde, el formato y el tipo de dato de los datos, y los píxeles de la imagen.

Figura 2.28: Mapeo de Textura.

También tenemos otras funciones tales como $gluScaleImage(...)$, para poder escalar el tamaño de la imagen, $glCopyTexImagexD(...)$, para crear una textura usando la data contenida en el Framebuffer para definir los téxeles.

De la misma forma en la que se almacena el volumen en la memoria de textura, se puede también modificar una parte de ella o reemplazarla. OpenGL define la función $glTexSubImagexD(...)$ para este fin, que a diferencia de $glTexImagexD(...)$, podemos remplazar toda la textura o una sub región de la misma, como se puede ver en la ***figura 2.28***.

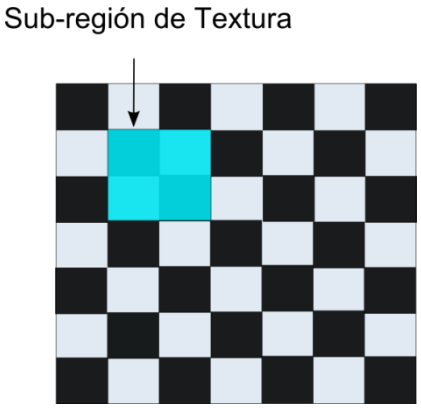


Figura 2.28: Mapeo de textura por Sub-Región

En la actualidad, debemos optimizar el espacio en la memoria de textura para poder tener un manejo más eficiente del mismo. Una forma de aprovechar el espacio es usando

texturas Atlas **[45]**, en donde podemos combinar muchas imágenes dentro de una misma textura, evitando declarar una textura nueva por cada imagen, y como resultado acelera el proceso de *render*.

Figura 2.29: Textura Atlas utilizadas para organizar un conjunto de texturas de forma eficiente.

2.14 Algoritmos de Empaquetamiento

Se puede optimizar el espacio de la textura Atlas a través de algoritmos de empaquetamiento, para poder insertar las imágenes en la textura minimizando el espacio inutilizado. Entre los algoritmos de empaquetamiento más usados, tenemos el algoritmo de *Bin Packing* (Empaquetamiento Binario) N-dimensional, un problema de optimización combinatoria NP-duro, en donde tenemos unidades de elementos finitos, y el objetivo es empaquetar todos los elementos en el número mínimo de contenedores. También tenemos el algoritmo de *Strip Packing* (Empaquetamiento por tiras), en donde solo se tiene una sola unidad de anchura dada, y el objetivo es empaquetar todos los elementos dentro de una altura mínima.

2.14.1 Strip Packing

Corman el al. **[46]** plantean que la mayoría de estos algoritmos son enfocados en *shelf – algoritms* (Algoritmos de Estantes), los cuales hacen el proceso de empaquetamiento en filas, de izquierda a derecha, creando un conjunto de shelfs por niveles (En el caso de dos dimensiones). Para poder empaquetar los elementos, se tienen tres estrategias de inserción en los shelfs (ver **Figura 2.30**):

(NFDH) *Next-Fit Decreasing Height* (Siguiente Ajuste de Altura Decreciente), en donde los elementos son empacados justificados en la izquierda del shelf actual, si encajan. En caso contrario, se crea un nuevo shelf y se ajusta el elemento a este.

(FFDH) *First-fit Decreasing Height* (Primer Ajuste de Altura Decreciente), donde encajamos el elemento en el primer shelf en donde se ajuste. Si no se encuentra ningún shelf, se procede a usar NFDH.

(BFDH)*Best-Fit Decreasing Height* (Mejor Ajuste de Altura Decreciente), el elemento es empaquetado en donde encaje, donde el espacio horizontal sea el mínimo posible. Igual que FFDH, si no encaja en ningún shelf, se usa NFDH.

Figura 2.30: En el NFDH se tiene que si no encaja un estante, se crea uno nuevo del tamaño del nuevo objeto. En el FFDH se inserta en el primer shelf en donde encaje y en BFDH en donde mejor encajen. Se insertan en orden del 1 al 6.

2.14.2 Bin Packing

Diversos autores han estudiado soluciones para resolver el problema de *Bin Packing*. Chung et al. **[47]** divide la solución en en dos fases, la primera en donde se usa el algoritmo (HFF) *Hybrid First-Fit* (Primer Ajuste Híbrido) que consiste en obtener un *strip* usando FFDH de la sección anterior. En la segunda fase se procede a empaquetar los strips en conjuntos finitos de contenedores usando el algoritmo (FFD) *First-Fit Decreacing* (Primer Ajuste Decreciente), en donde se insertan cada uno de estos en el primer lugar que encajen en orden decreciente en los contenedores. Lo mismo se puede aplicar con los algoritmos NFDH y BFDH.

Berkey y Wang **[48]**se basaron en HFF (ver **Figura 2.31**) para crear un algoritmo de dos fases llamado (FBS) *Finite Best-Strip* (Mejor Tira Finita), donde llevan a cabo como primera fase el algoritmo BFDH, y como segunda fase se utiliza un algoritmo de mejor ajuste decreciente: se inserta el estante actual en un contenedor de mayor tamaño (si existe), donde quepa y donde el espacio vertical inutilizado sea el menor, o se crea un nuevo contenedor para insertar el estante.

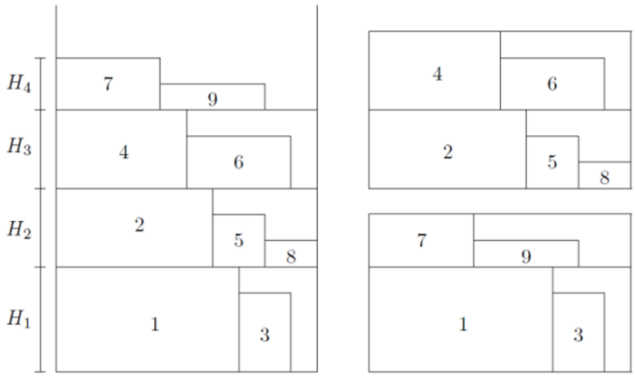


Figura 2.31: HFF

Otra estrategia del tipo *shelf packing* se basa en la solución adoptada en el problema de la mochila (KP) *Knapsack Packing* (Empaquetamiento por Mochila) propuesto por Lodi et al. **[49].** En un problema tradicional del mochilero se tiene que seleccionar un sub-conjunto de n elementos, cada uno tiene un beneficio y peso, por lo que el peso total no exceda la capacidad y ganancia total sea máxima. En la primera fase se inicializa un estante con un elemento de gran altura, en donde los pesos vienen denotados por el ancho de cada elemento, y la ganancia viene dada por el área del elemento. Se aplica el algoritmo KP, tratando de maximizar el número de elementos a insertar en un estante, con la mayor área ocupada posible. Una vez obtenido los *shelfs*, se insertan en un número finito de contenedores usando el algoritmo (BFD) *Best-Fit Decreacing* (Mejor Ajuste Decreciente).

Este algoritmo lo podemos desarrollar en una sola fase, trabajando con los *shelfs* directamente en los contenedores, Evaluados por Berkey y Wang **[48]**.

Se han propuestos diferentes algoritmos para trabajar con estantes, entre estos están (FNF) *Finite Next-Fit* y (FFF) *Finite First-Fit*. El algoritmo (FNF) *Finite Next-Fit*

(Siguiente Ajuste Finito) empaqueta directamente los elementos en números finitos de contenedores exactamente de la misma forma que HNF. Algoritmo (FFF) *Finite First-Fit* (Siguiente Ajuste Finito) (**Figura 2.32**) es muy parecida a FFDH. Se inserta el elemento en el estante más bajo en el primer contenedor donde encaje; si no encaja en ningún *shelf*, se debe crear en el primer contenedor adecuado.

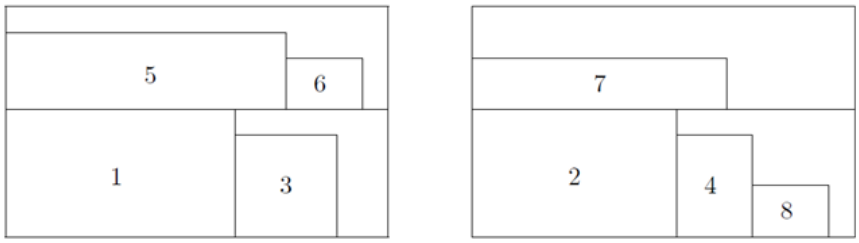


Figura 2.32: FFF

También podemos ver que existen algunas estrategias sin el uso de estantes para insertar elementos en los contenedores. La principal estrategia sin estantes es conocida como (BL) *Bottom-Left* (Abajo-Izquierda), y consiste en empacar los elementos actuales en la más baja posición posible, justificado a la izquierda. Baker et al. **[50]** analizaron el rendimiento del peor caso del algoritmo resultante, y obtuvieron lo siguiente: si los elementos no están ordenados, *BL* puede ser arbitrariamente malo; si los elementos están ordenado por anchura no-incremental, entonces se puede conseguir un algoritmo óptimo, con los límites ajustados.

Berkey y Wang **[48]**propusieron la estrategia *BL* para un número finito de contenedores. Su algoritmo es llamado (FBL) *Finite Bottom-Left* (Abajo-Izquierda Finito), donde inicialmente se ordenan los elementos por anchura no creciente. Luego, el elemento actual es empaquetado en la posición más baja del contenedor inicializado, justificado a la izquierda; Si ningún contenedor puede ser asignado, uno nuevo se inicializa. Este algoritmo produce un empaquetamiento en orden $O(n^2)$.

Martínez Bayona **[45]** utiliza un algoritmo de *Bin Packing* de dos dimensiones para empaquetar un conjunto de imágenes en una textura altas de forma optimizada. Para esto, utiliza la estrategia FFD, con la particularidad que utiliza un algoritmo para calcular el mínimo espacio que pueda tener la textura atlas, tomando en cuenta los tamaños de cada

una de las texturas de entrada. En este caso específico, podemos observar que a diferencia de las demás técnicas, ésta solo requiere un contenedor.

Para calcular el mínimo espacio que puede tener la textura atlas se toma como parámetro de entrada el conjunto de texturas $I = i_1, i_2 \dots i_n$. El tamaño inicial del ancho y largo de la textura será:

$$\alpha = log_2(\sum_{j=1}^{n} i_{width_j} \, x \, i_{height_j})$$

$$width = 2^{\left[\frac{\alpha}{2}\right]} \;,\; height = 2^{\alpha - \left[\frac{\alpha}{2}\right]}$$

[Ec. 2.17]

Esta fórmula retorna en potencia de dos el ancho y el largo del tamaño mínimo del atlas. Para incrementar el tamaño de la textura atlas, le sumas una unidad a α y se actualizan los tamaños.

Otra particularidad de este algoritmo de empaquetamiento, es la forma de insertar las texturas de entrada en el contenedor. Para esto se usa un árbol BSP. Cada nodo del árbol define una región rectangular de la textura atlas. La raíz define el espacio total a utilizar. Si un nodo es una hoja, este puede o no ser ocupado por una textura. En caso contrario, se tienen dos hijos que superpone todo el espacio del nodo padre, como se muestra en la **Figura 2.33**. El algoritmo propuesto tiene un tiempo de $O(log(n))$, donde n es la cantidad de nodos del árbol.

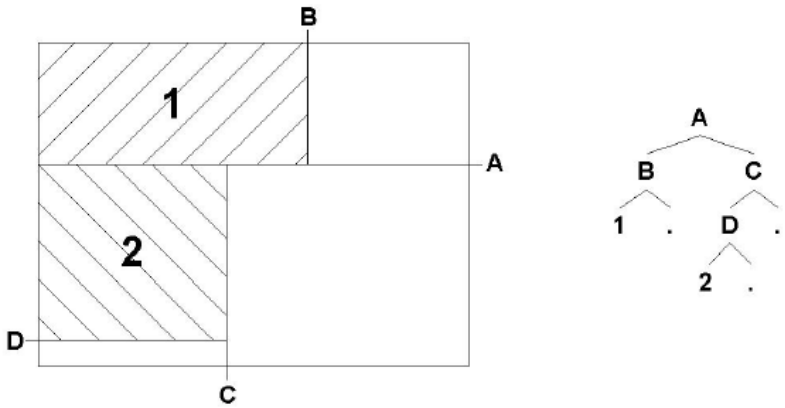


Figura 2.33: Árbol BSP generado para insertar en la textura atlas cada uno de los elementos a desplegar. Los nodos hoja son espacios vacíos o un espacio ocupado. Los demás nodos son las divisiones del contenedor.

Una vez insertado todas las texturas de entrada en la textura atlas, Martínez Bayona desarrolla dos algoritmos para optimizar el espacio de esta textura, extendiendo cada una de las texturas al tamaño total del atlas, y cambiando de lugar algunas texturas para ocupar el mayor espacio posible.

Gordon Jake **[51]** propone un algoritmo de empaquetamiento para utilizar una imagen compuesta por varias para auto-generar *CSS Sprites*. Al igual que Martínez Bayona, desarrolla un algoritmo de *Bin Packing* usando la heurística FFD y un árbol de particionamiento de espacio binario para realizar la inserción de los objetos. Para calcular el tamaño estimado del contenedor, siendo n el número de bloques a insertar:

$$width = avg(width) * \sqrt[2]{n} \ , \ height = avg(height) * \sqrt[2]{n}$$

[Ec. 2.18]

Gordon Jake desarrolla un algoritmo para generar el *CSS Sprites* sin tomar en cuenta el límite del contenedor. Al igual que Martínez Bayona, solo es requerido un solo contenedor. Tomando como base el tamaño del primer bloque, podemos ir aumentando el tamaño de la imagen por la derecha o por debajo del espacio ya particionado. No podemos insertar un bloque por debajo del espacio si el ancho del bloque a insertar es mayor al espacio particionado. Igualmente, si es mayor el largo del bloque al insertar por la derecha, no puede ser insertado el bloque. Si el bloque a insertar es mayor que el ancho y el largo del contenedor, no podemos insertar el bloque. Para esto, Gordon Jake propone que los bloques más grandes sean insertados primero.

El orden de los bloques es significativo para el resultado final. Podemos ordenar la lista de bloques a insertar por el tamaño del ancho, el largo, el área máxima, el lado máximo, y aleatorio (ver **Figura 2.34**).

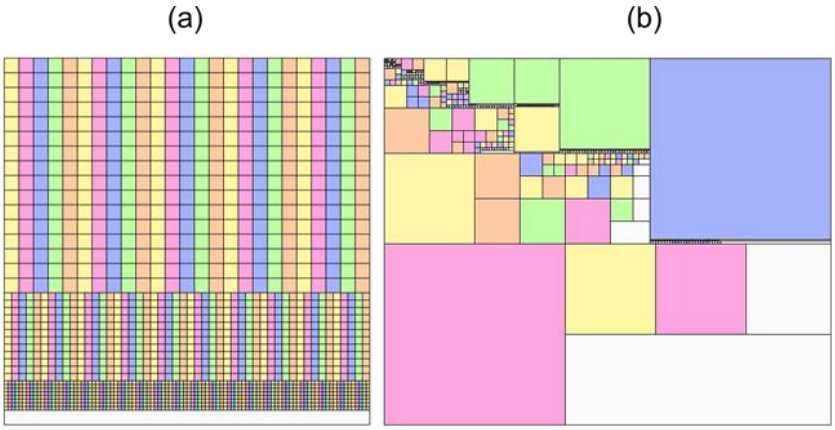


Figura 2.34: Resultados del Algoritmo de Bin Packing según el orden, en (a) podemos ver como esta ordenado el empaquetamiento por orden de tamaño, en caso contrario, (b) esta ordenado de forma aleatoria.

Con estas técnicas podemos insertar elementos en los contenedores de una forma sencilla, pero en nuestro caso, se debe aprovechar la mayor cantidad de espacio posible en un contenedor, para esto podemos usar técnicas meta heurísticas para minimizar el espacio inutilizado.

Crainic et. al [**69**] propone un algoritmo eficiente para empaquetar bloques a través de puntos extremos (ver **Figura 2.35**), los cuales generan puntos en los esquinas del área llena, donde a medida en que se vaya insertando, se van generando nuevos puntos. Con esto, se pueden explorar los espacios libres y comparar en base a los puntos adyacentes para saber si se puede insertar un bloque. Adicionalmente, proponen una heurística de inserción EP-FFD (*Extreme point First Fit Decreasing*) donde se insertan de forma decreciente los bloques en el primer punto extremo donde encajen. También proponen EP-BFD (*Extreme point Best Fit Decreasing*) el cual insertan en el área donde generen menos espacio residual mediante una función de mérito que busca la mejor área a insertar.

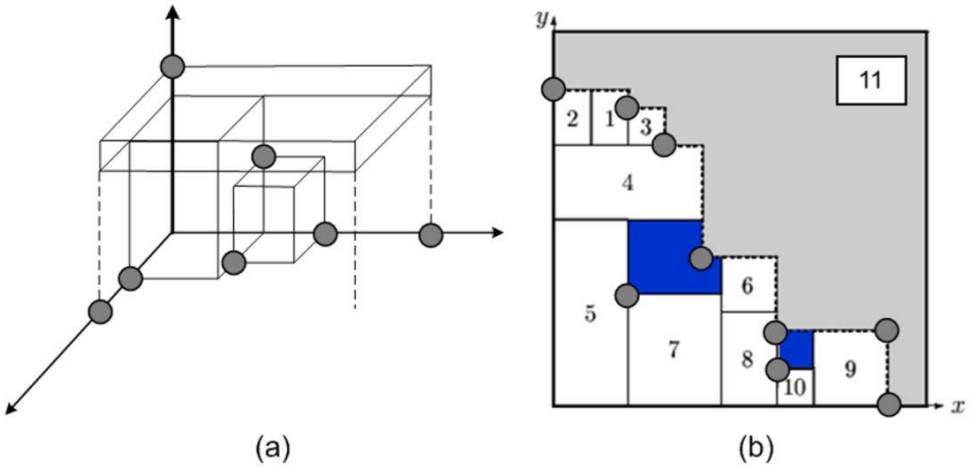


Figura 2.35: Algoritmo de puntos extremos, en (a) podemos ver el algoritmo de EP en 3D, en caso contrario, (b) representa el algoritmo en 2D, donde el área gris es el espacio residual.

El algoritmo de *Bin Packing* puede ser desarrollado bajo un ambiente 3D, teniendo en cuenta que se trabaja con un valor *Z* de profundidad. Puede ser utilizado para solucionar el problema de fragmentación de la textura atlas, con el propósito de poder insertar la mayor cantidad de bloques en la textura, dándole una mejor visualización al volumen.

2.15 Morton Order

Para el ordenamiento de una serie de bloques en un arreglo unidimensional a un ambiente 2D se puede usar el algoritmo propuesto por G. M. Morton [**70**]. *Morton Order*, que es una función que hace una correspondencia entre los datos multidimensionales a una dimensión mientras preserva la localidad de los puntos. El valor de *z* de un punto en un espacio multidimensional es simplemente calculado intercalando la representación binaria de sus valores de coordenadas. Una vez que la data está almacenada en este orden, cualquier estructura de datos unidimensional puede ser usada como un Árbol Binario de Búsqueda, *B-Tree*, Listas o Tablas Hash. El resultado de ordenar puede ser equivalentemente descrito como el orden que se obtendría de un recorrido en profundidad de un *quad-tree*.

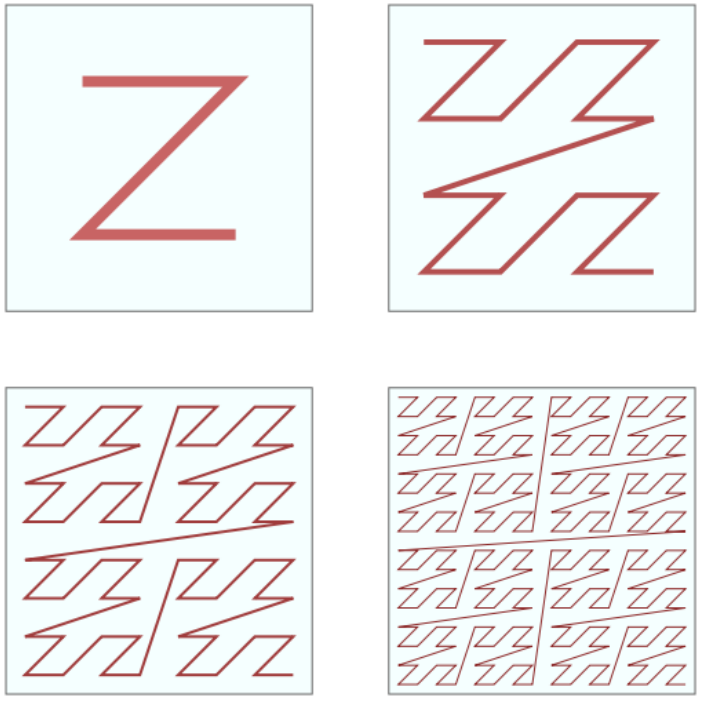


Figura 2.36: Representación en 2D del algoritmo *Morton Order*. Se puede ver como el orden converge a la forma de una letra **Z.**

En el próximo capítulo se procede a explicar la propuesta de trabajo especial de grado.

Capítulo 3

Marco Aplicativo

Ya estudiado los distintos métodos utilizados para el despliegue de volúmenes de gran tamaño, utilizando técnicas multi-resolución y el manejo de memoria de textura, se procederá a explicar la propuesta de tesis. Se explicará la metodología utilizada y cada uno de los aspectos resaltantes en el desarrollo.

3.1 Metodología de desarrollo

Para la gestión de la propuesta de tesis se usó como marco la metodología ágil *SCRUM* (K. Schwaber [**71**]), que hoy en día se puede decir que es la metodología ágil más popular utilizada para desarrollar productos de software.

Las características principales de *SCRUM* pueden resumirse en dos:

- El desarrollo software mediante **iteraciones incrementales** (entregas de pequeñas porciones de software en un máximo de 4 semanas).
- Las **reuniones** a lo largo del proyecto.

Como se indica K. Schwaber [**71**]**,** existen tres pilares en los que se basa:

- **Transparencia:** todos los aspectos del proceso que afectan al resultado son visibles para todos aquellos que administran dicho resultado. Por ejemplo, se

utilizan pizarras y otros mecanismos o técnicas colaborativas para mejorar la comunicación.

- **Inspección:** se debe controlar con la frecuencia suficiente los diversos aspectos del proceso para que puedan detectarse variaciones inaceptables en el mismo.
- **Revisión:** el producto debe estar dentro de los límites aceptables. En caso de desviación se procederá a una adaptación del proceso y el material procesado.

En las metodologías ágiles la descripción de estas necesidades se realiza a partir de las **Historias de Usuario** que son, principalmente, lo que el cliente o el usuario quiere que se implemente; es decir, son una descripción breve, de una funcionalidad software tal y como la percibe el usuario (M. Cohn, [**72**])**.**

En *SCRUM* a cada iteración se le denomina *Sprint*. *SCRUM* recomienda iteraciones cortas, por lo que cada *Sprint* durará entre 1 y 4 semanas (ver **Figura 3.1**). Y como resultado se creará un producto software potencialmente entregable, un prototipo operativo.

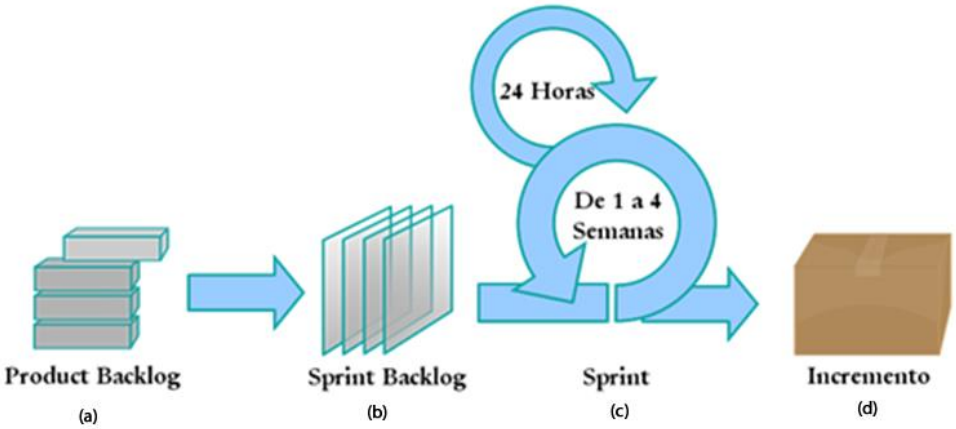


Figura 3.1: Como primera fase, tenemos el *Product Backlog* (a) que consiste en un listado de historias de usuario que se incorporarán al producto de software a medida en incremente el desarrollo. Seguido del *Sprint Backlog* (b), donde se apilan las historias de usuarios que se vayan a desarrollar a lo largo de la iteración. El *Sprint* es la iteración de desarrollo en sí (c), y al final de cada una de estas, se debe entregar un producto potencialmente operativo (d).

Uno de los aspectos más importantes en cualquier proyecto, y también en los proyectos ágiles, es el establecimiento del equipo. Los roles y responsabilidades deben ser claros y conocidos por todos los integrantes del mismo. Cada equipo *SCRUM* tiene tres roles:

- **Scrum Master**: es el responsable de asegurar que el equipo *SCRUM* siga las prácticas de *SCRUM*.
- **Propietario del Producto (*Product Owner*)**: es la persona responsable de gestionar las necesidades que serán satisfechas por el proyecto y asegurar el valor del trabajo que el equipo lleva a cabo.
- **Equipo de desarrollo**: El equipo está formado por los desarrolladores, que convertirán las necesidades del *Product Owner* en un conjunto de nuevas funcionalidades, modificaciones o incrementos del producto software final (ver **Figura 3.2**).

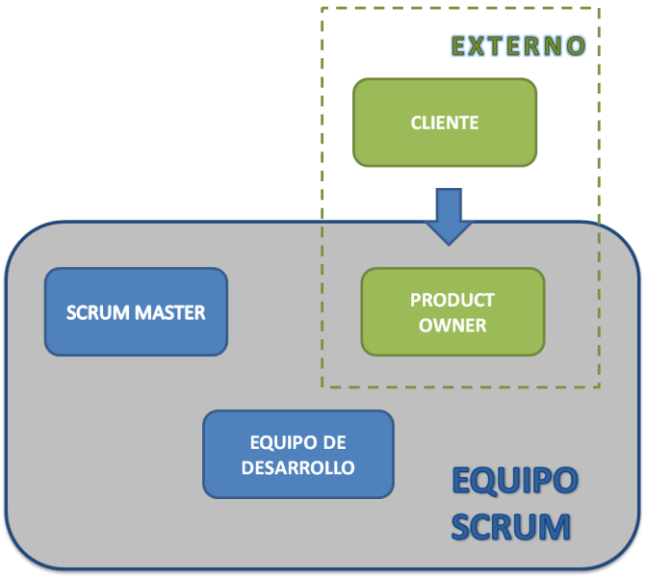


Figura 3.2: Conformación del equipo *SCRUM*. El Product Owner es un ente directamente relacionado con el cliente.

Las reuniones son un pilar importante dentro de *Scrum*. Se realizan a lo largo de todo el Sprint como muestra la **Figura 3.1**. Se definen diversos tipos de reuniones: Reunión de planificación del *Sprint* (*Sprint Planning Meeting*), que se lleva a cabo al principio de cada Sprint, definiendo en ella que se va a realizar en ese *Sprint*. Esta reunión da lugar al *Sprint Backlog*. En esta reunión participan todos los roles. El *Product Owner* presenta el conjunto de historias de usuario en el *Product Backlog* y el equipo de desarrollo selecciona las historias de usuario sobre las que se trabajará. El siguiente tipo es la reunión diaria**,** es de no más de 15 minutos en la que participan el equipo de desarrollo y el *Scrum Master*. En esta reunión cada miembro del equipo presenta lo qué hizo el día anterior, lo qué va a hacer hoy

y los impedimentos que se ha encontrado. También tenemos la reunión de revisión del *Sprint* (*Sprint Review Meeting*)**:** se realiza al final del *Sprint*. Durante la misma se indica qué ha podido completarse y qué no, presentando el trabajo realizado al *Product Owner*. Por su parte el *Product Owner* (y demás interesados) verifican el incremento del producto y obtienen información necesaria para actualizar el *Product Backlog* con nuevas historias de usuario. Por último se realiza retrospectiva del *Sprint* (*Sprint Retrospective*)**,** también al final del *Sprint*, sirve para que los integrantes del equipo *Scrum* y el *Scrum Master* den sus impresiones sobre el Sprint que acaba de terminar. Se utiliza para la mejora del proceso y normalmente se trabaja con dos columnas, con los aspectos positivos y negativos del *Sprint*. En la figura 3.3 se describe el ciclo de vida *Scrum*.

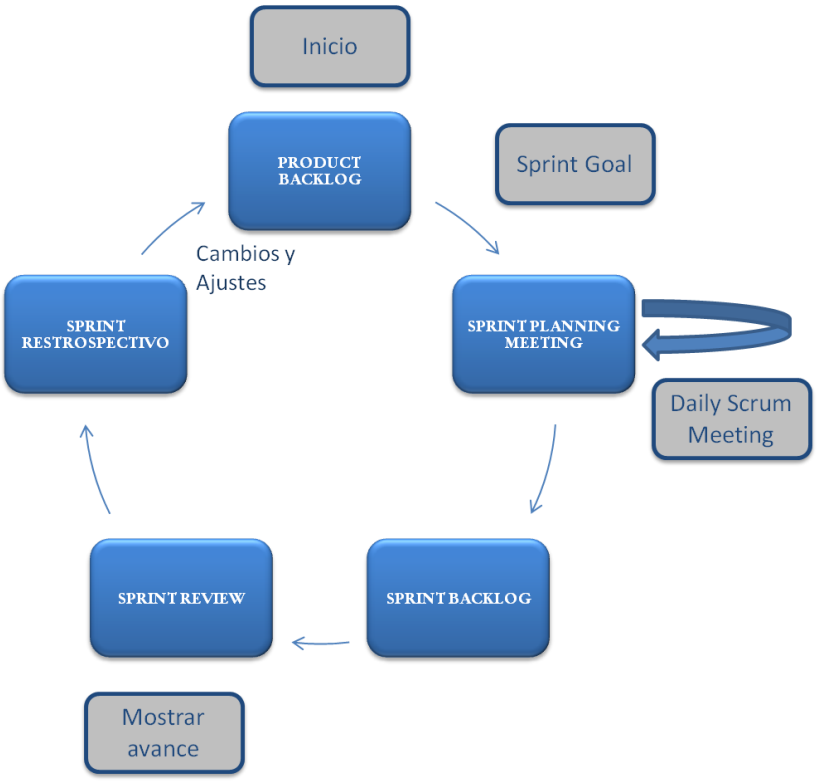


Figura 3.3: Ciclo de vida Scrum. Se resaltan las reuniones.

La propuesta de tesis se desarrolló a lo largo de 10 *Sprints*, con una duración de 4 semanas. A continuación desglosan cada uno de ellos:

- **Sprint 1**: Toma de decisiones, donde se crea el entorno de desarrollo para el proyecto, con su respectivo repositorio remoto y se crean las clases necesarias.

- **Sprint 2**: Montar y ejecutar las librerías del motor gráfico, también se desarrollaron clases para generar los resultados, para medición del tiempo y librerías matemáticas.
- **Sprint 3:** Se procede a la fase de carga del volumen, donde se generan los diferentes niveles de detalle del volumen en memoria *RAM*.
- **Sprint 4:** Se desarrolla un algoritmo para la inserción de bloques en la textura atlas recursivo en particiones fijas para luego comenzar a desarrollar el despliegue del volumen con *Ray Casting*.
- **Sprint 5:** Se implementa la función de transferencia, se desarrolla un algoritmo de selección con cola de prioridad sin colapso de bloques. Se termina de desplegar el volumen de forma correcta.
- **Sprint 6:** Se inicia el desarrollo de un algoritmo eficiente para el almacenamiento de bloques. Se elimina la recursividad a través de una estructura de pila y se desarrolla un intercambio de nodos en el atlas.
- **Sprint 7:** Se propone un algoritmo que genere un aumento del tamaño de la textura atlas para generar sub-estantes horizontales y verticales.
- **Sprint 8:** Se desarrolla una forma de inserción usando *Extreme Points*. Se comienza a desarrollar un algoritmo de apuntadores ordenado de mayor a menor. Se acomodan bugs de carga del volumen.
- **Sprint 9:** Se optimiza el algoritmo de apuntadores y se genera el algoritmo de selección para refinar y colapsar bloques.
- **Sprint 10:** Se hacen las pruebas pertinentes con tres tipos de volúmenes distintos.

A continuación se procede a explicar el proceso de visualización multi-resolución planteado en el trabajo especial de grado.

3.2 Visualización de volúmenes multi-resolución.

Como se mencionó anteriormente, existen diversas técnicas para la visualización de volúmenes multi-resolución. En este trabajo se implementó la técnica de *Ray Casting* en GPU de una pasada aplicada a una jerarquía por bloques almacenada en una textura atlas.

Los pasos para el despliegue del volumen multi-resolución son los siguientes (ver **Figura 3.4**): carga del volumen, almacenamiento de datos volumétricos en una estructura de bloques, generación de niveles de detalle, criterio de selección y finalmente el despliegue del volumen multi-resolución.

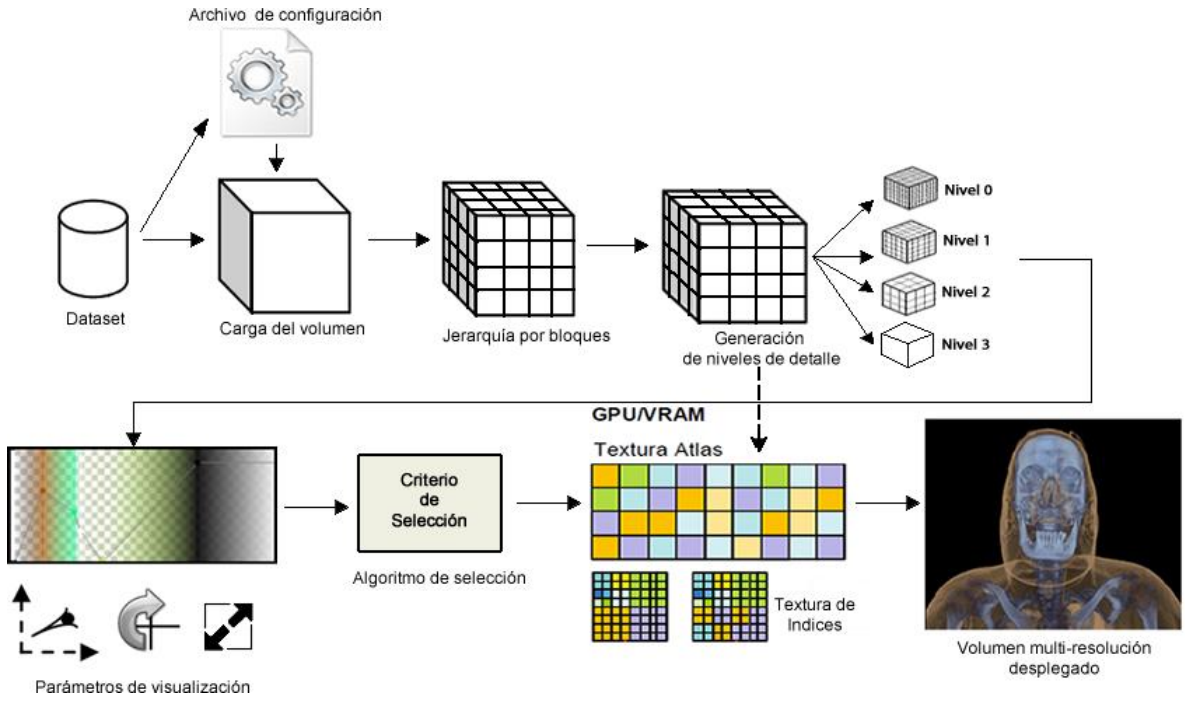


Figura 3.4: El proceso de despliegue de volúmenes multi-resolución utilizado consta de: cargar el volumen en memoria con los parámetros indicados en el archivo de configuración, almacenar los datos en una estructura de datos por bloques, generar los niveles de detalles y almacenarlos en la textura atlas, aplicar los parámetros de visualización, seleccionar los niveles de detalles a desplegar, buscar los bloques en la textura atlas y desplegar del volumen en forma de multi-resolución.

3.3 Carga de volumen

Los datos volumétricos se encuentran almacenados en un *dataset* que tiene un archivo de configuración que especifica los parámetros iniciales para la carga del volumen- Estos parámetros son: tipo de *dataset* (raw o pvm), ruta y nombre del *dataset*, tipo de datos (8 bits o 16 bits), dimensiones del volumen en $\boldsymbol{x}, \boldsymbol{y}$, y $\boldsymbol{z}$, tamaño del bloque, dimensiones de la textura atlas $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ y por último los parámetros de escalado del volumen en sus componentes $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$. Este archivo es leído por la aplicación para la carga de datos del volumen en memoria principal. Una vez cargada esta información se procede a organizarla en una jerarquía por bloques que se explicará a continuación.

3.4 Jerarquía multi-resolución basada en bloques

Para este trabajo especial de grado se utilizó una jerarquía por bloques para la representación del volumen multi-resolución. Inicialmente ésta jerarquía particiona el volumen en bloques (ver **Figura 3.5**), generando el nivel de detalle 0 del volumen, el cual representa los datos en su mayor resolución. Los bloques iniciales tienen un tamaño de $\boldsymbol{n^3}$, donde $\boldsymbol{n}$ representa el tamaño del bloque en una dimensión. Una vez obtenidos los bloques del primer nivel de detalle se procede a generar los niveles de detalle siguientes. Para este proceso se utilizó la técnica de interpolar los datos del nivel de detalle anterior como se describe en la **Figura 3.6 (a)**, donde 1 píxel representa 2 píxeles del nivel de detalle anterior en una representación unidimensional. La ventaja de usar esta técnica es que no es necesario utilizar el píxel de holgura, por lo que se elimina el *overhead* que conlleva compartir medio vóxel entre bloques adyacentes, dando así espacio para almacenar más información sin redundancia.

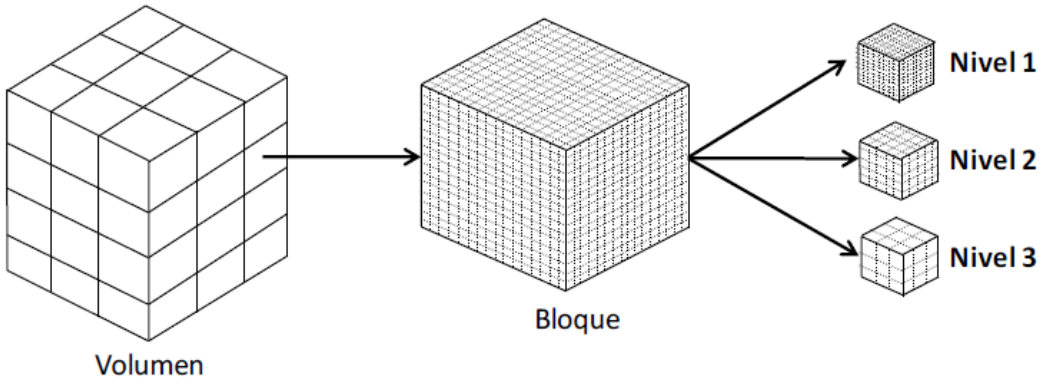


Figura 3.5: Representación de un volumen en jerarquía por bloques y 3 niveles de detalle por cada bloque del volumen.

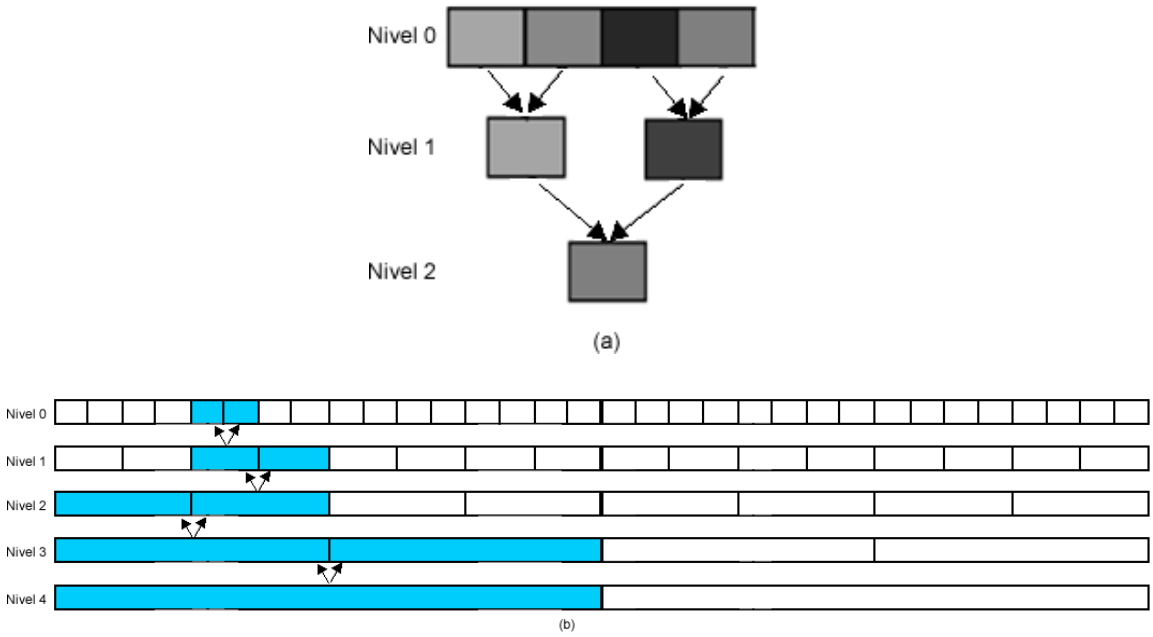


Figura 3.6: (a) Generación de niveles de detalle calculando el promedio de los píxeles del nivel de detalle anterior. (b) Representación 1D de los niveles de detalle en una jerarquía por bloques. Como se puede ver en la figura, un píxel del nivel de detalle $\boldsymbol{d + 1}$ representa 2 píxeles del nivel de detalle $\boldsymbol{d}$. En esta implementación, las coordenadas de textura en cada bloque será [0,1].

Siguiendo esto, se puede decir que se obtendrán $\frac{n^3}{2^{lod}}$ bloques por nivel de detalle, donde $\boldsymbol{lod}$ es un entero mayor o igual que 0 que representa el nivel de detalle.

Una vez que se obtienen todos los niveles de detalles se seleccionan los bloques que serán desplegados mediante un criterio de selección.

3.5 Criterio de selección

Para el despliegue de los bloques se utilizó como criterio de selección la distancia con respecto a un punto de interés, el cual es especificado por el usuario final en la interfaz de la aplicación (ver **Figura 3.7**). El criterio de selección permite determinar cuáles bloques deben ser desplegados y en qué resolución, teniendo en cuenta que el área de interés debe tener una mejor calidad de imagen que las áreas del volumen que están más alejadas del mismo.

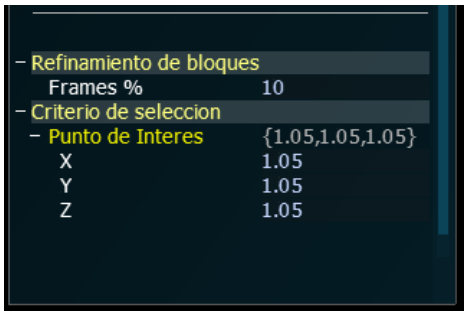


Figura 3.7: Opciones de criterio de selección y refinamiento de bloques en la interfaz gráfica de la aplicación.

Inicialmente los bloques del menor nivel de detalle son insertados en una cola de prioridad. La prioridad es dada según una distancia al punto de interés y la distancia al ojo en coordenadas de mundo, donde ordena los bloques desde el más cercano hasta el más lejano.

Cabe destacar que los bloques completamente transparentes son omitidos para el cálculo de la prioridad, ya que estos no aportan valor a la imagen final y ocupan un espacio en memoria que puede ser usado por otros bloques.

3.5.1 Punto de interés

La prioridad de cada bloque es calculada con la ecuación **Ec. 3.1** que es una relación entre la distancia del punto de interés al centro del bloque, la distancia al ojo en coordenadas de objeto y la diagonal del bloque (ver **Figura 3.8**). De esta manera se le dará mayor prioridad para ser refinado a los bloques de menor resolución, más cercanas al punto de interés y al ojo. Debemos tener en cuenta que la diagonal es calculada en función al número de vóxeles del bloque.

$$e_{dist} = diag \,/\, (diag \;+\; dist\,(c, ip))$$

[Ec. 3.1]

Donde $diag$ es la diagonal del bloque y $dist(c, ip)$ es la distancia desde el centro del bloque al punto de interés.

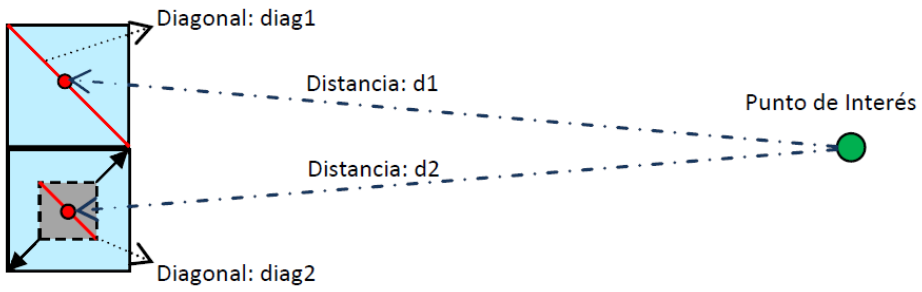


Figura 3.8: En la imagen se puede observar que la suma $\boldsymbol{diag1 + d1 > diag2 + d2}$, por lo tanto el bloque 2 tendrá mayor prioridad de refinamiento.

Una vez se hayan calculado las prioridades se inicia el proceso de refinamiento de bloques. Este proceso puede acarrear un gran tiempo de cómputo que es proporcional a la cantidad de bloques existente en la cola de prioridad. Es por esto que se ha implementado un refinamiento cuadro a cuadro (*frame to frame*) propuesto por Carmona **[4]**. El refinamiento cuadro a cuadro consiste en refinar solo hasta una cantidad máxima de memoria (indicado por el usuario) por cada cuadro. Este proceso se ejecuta hasta que el volumen quede completamente refinado o no se pueda refinar más (ver **Figura 3.9**).

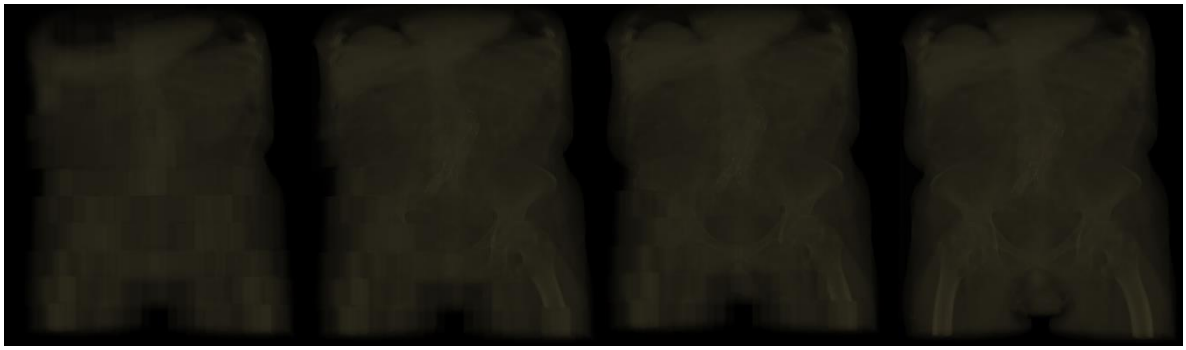


Figura 3.9: Ejemplo de refinamiento cuadro a cuadro (*frame to frame*) al 10% de un volumen.

3.5.2 Proceso de refinamiento y colapso de bloques

El proceso de refinamiento y colapso de bloques es el que se encarga de controlar la resolución que debe tener cada bloque que va a ser desplegado tomando en cuenta el área de interés y la capacidad de la memoria. Este proceso consta de dos (2) subprocesos, *refinar* y *colapsar*. El subproceso de *refinar* tiene como función cambiar el nivel de detalle

de los bloques a uno superior e insertarlos en la textura atlas siempre y cuando ésta tenga espacio disponible. El subproceso de *colapsar* hace lo contrario, cambia la resolución de los bloques a una menor. Idealmente hay que colapsar el bloque que menos reduzca la calidad en base a la métrica de error utilizada (ver **Figura 3.10**).

Los subprocesos *refinar* y *colapsar* son dependientes cuando no existe memoria suficiente para seguir refinando bloques, es decir, si se quiere refinar un bloque pero no hay memoria suficiente para realizarlo, el subproceso de colapsar debe reducir la resolución de algún o algunos bloques de manera que el refinamiento puede llevarse a cabo (ver **Figura 3.10**). Para esto se ha utilizado la métrica de distancia mencionada en la sección **3.5.1** ecuación **Ec.3.1** con el objetivo de determinar si el proceso de *refinar* y *colapsar* de bloques contribuye a mejorar la calidad de la imagen o no. Cuando no existe espacio suficiente para seguir refinando bloques y el proceso de *refinar* y *colapsar* de bloques no mejora la calidad de la imagen el algoritmo se detiene.

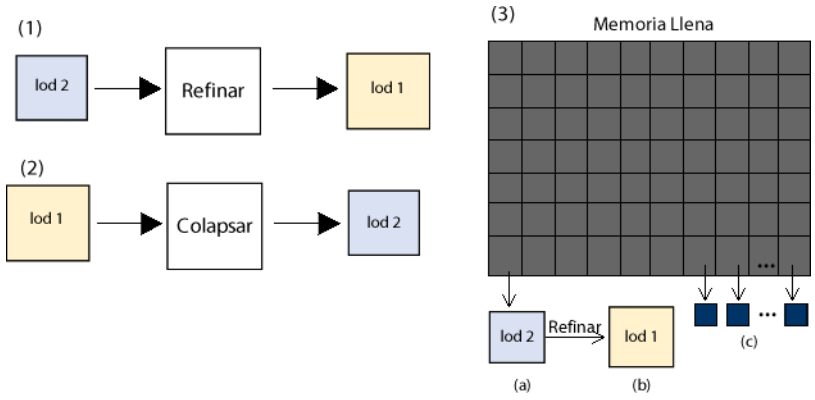


Figura 3.10: Representación de los subprocesos *refinar (1) y colapsar (2).* La figura *(3)* es una representación gráfica cuando los subprocesos son dependientes, cuando la memoria está llena y se quiere refinar un bloque (***figura 3.10.a***) se deben colapsar uno o más bloques (***figura 3.10.c***) para crear espacio y así poder insertar el bloque refinado (***figura 3.10.b***).

3.6 Despliegue del volumen

Para visualizar el volumen es necesario pasar por un proceso de despliegue. Los datos deben ordenarse y almacenarse en la textura atlas y luego recorrerse aplicando la técnica de *Ray Casting* en GPU de una pasada para obtener el volumen final. El despliegue se realizará cuadro a cuadro, procesando un porcentaje de bloques por cada cuadro.

3.7 Textura atlas

Se utilizó una textura atlas para insertar los bloques en memoria de textura. Las dimensiones de esta textura deben ser potencia de 2 en cada una de sus coordenadas (x, y) para un espacio bidimensional, donde x es el largo de la textura y y es la altura de la textura. Se usó un filtro de textura GL_NEAREST para evitar los artefactos generados por la interpolación entre las fronteras de los bloques. Se implementó una segunda textura más pequeña para indexar la textura atlas (ver **Sección 2.10.5**).

Cuando se insertan bloques en la textura atlas se puede dar el caso en que genere fragmentación o espacios inutilizados. Esto puede causar que no alcance el espacio para insertar bloques refinados y se tenga que realizar un proceso de desfragmentación de la textura atlas, dando como desventaja la transferencia de datos redundantes hacia la GPU. Para minimizar la fragmentación del atlas se explican diferentes algoritmos propuestos en la subsección siguiente.

3.8 Algoritmos propuestos para minimizar la fragmentación de la textura atlas.

En el presente trabajo se desarrollaron distintos algoritmos para el almacenamiento de bloques en la textura atlas. Cabe destacar que el último algoritmo propuesto es el usado en la solución de la tesis y es el que va a ser explicado con mayor énfasis. Todos los algoritmos fueron explicados en un ambiente de dos dimensiones, con motivo de facilitar al lector una mayor legibilidad de las propuestas.

3.8.1 Bin Packing recursivo

En esta solución se usó la textura atlas como único contenedor de bloques, a diferencia del problema de *Bin Packing* convencional, que crea un nuevo contendor cuando no hay espacio en ningún otro. Para esto, se divide el atlas en particiones (ver **Figura 3.11**) donde cada partición es creada dinámicamente a medida en que se van insertando los bloques.

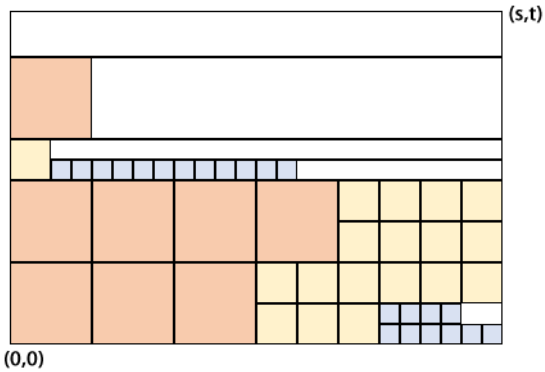


Figura 3.11: Representación de la textura atlas usando la técnica de *Bin Packing*.

Para la creación de particiones, se utiliza un árbol de particionamiento binario *BSP* donde se inserta a través de un algoritmo de recorrido Pre-Orden en el primer lugar que encaje (*First Fit*). Como se puede ver en la **Figura 3.12** el algoritmo reserva el área de tamaño del bloque a insertar, creando el nodo izquierdo como la nueva partición, y el área restante como el nodo derecho.

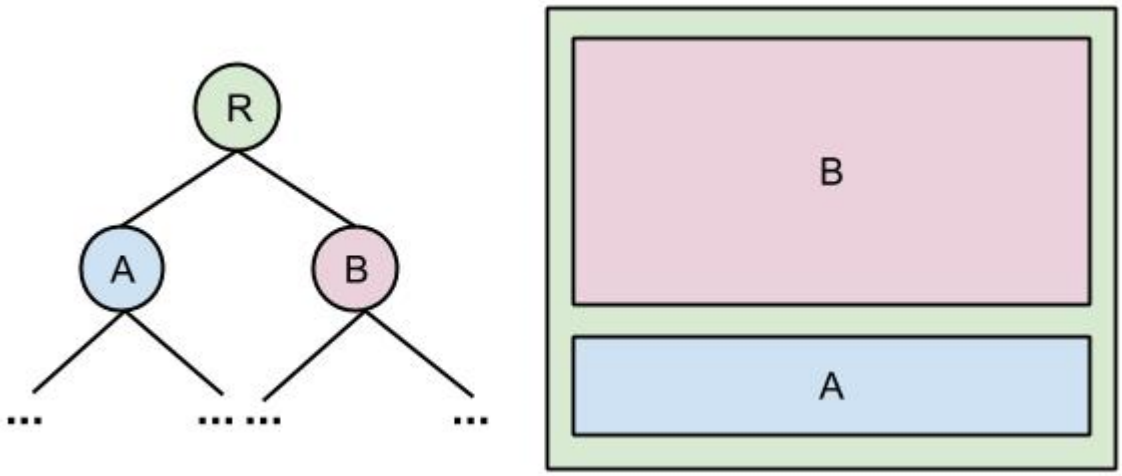


Figura 3.12: Proceso de particionamiento usando *Bin Packing*.

Cuando se trabaja con una gran cantidad de bloques, el algoritmo tiende a ser poco eficiente, esto es debido a que debe buscar desde la raíz a los nodos hojas el siguiente nodo con el suficiente espacio para insertar. Para optimizar el algoritmo, se cambió la recursividad por ciclos iterativos a través de una pila, el cual guarda solamente los nodos hoja con espacio disponible a insertar.

Para aprovechar mejor el espacio de la textura atlas, se implementó un algoritmo que va redimensionando de forma lógica el tamaño del atlas hasta alcanzar el límite s y t (dimensiones) del mismo. Se comenzó en un tamaño estándar (bloque con mayor nivel de detalle) y se fue redimensionando a través de sub particiones a lo largo de la textura (ver **Figura 3.13**).

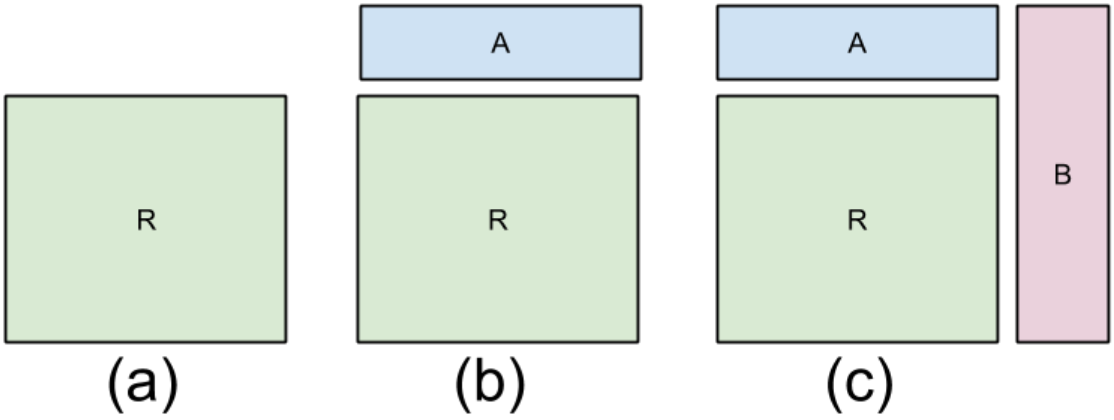


Figura 3.13: Redimensionamiento de la textura atlas, donde R es el espacio inicial **(a)**, cuando caben más bloques se procede a generar una partición en la parte superior del contenedor **(b)**, si se llena, genera una nueva partición en la parte derecha **(c)**.

Esta primera propuesta tiene como desventaja el alto costo al hacer reinserción de bloques, debido a que el algoritmo genera particiones de tamaño fija según el tamaño del bloque que debía ser insertado, si el siguiente bloque a ser insertado era de mayor tamaño que el anterior, debía crearse una nueva partición para insertarlo, dejando la partición anterior como un espacio libre de memoria inutilizado. En caso contrario, si el bloque a ser insertado era de menor tamaño, la partición generada debía ser de ese tamaño y cuando se debía refinar un bloque no podía ser insertado en esa área, generando gran pérdida de espacio. Para resolver este problema, se tenía que eliminar las particiones, generar nuevas, y reinsertar una gran cantidad de bloques en el atlas, sin garantía de que todos pudieran ser insertados.

3.8.1.1 Extreme Points

Se presentó una propuesta basada en *Extreme Points*, donde cada bloque puede generar nuevos puntos de inserción para los siguientes bloques (ver **Figura 3.14**). Estos puntos

tienen un orden específico *Bottom – Left* (de abajo hacia la izquierda) y son insertados en una lista de puntos. Para calcular estos puntos se toma en cuenta los bloques ya insertados, ya que se debe evaluar si no hay intersección de bloques en fronteras, si las consigue, entonces genera el punto en esa intersección.

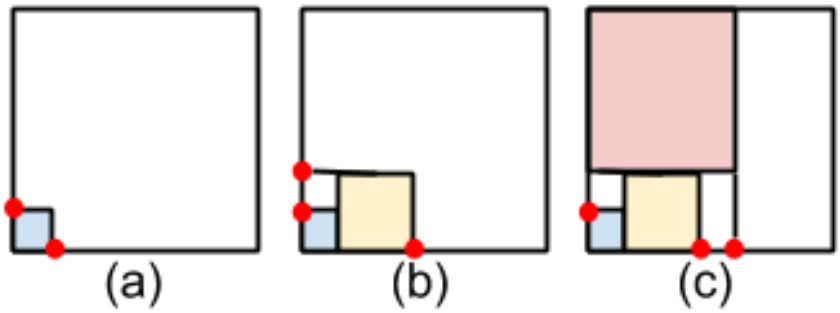


Figura 3.14: Generación de *Extreme Points.*

Existen dos heurísticas para inserción de bloques: *First Fit,* en donde se trata la lista de puntos como una cola ordenada de abajo hacia arriba, y *Best Fit,* que divide los espacios vacíos en sub – área a través de una función de mérito. Una vez obtenidos estos espacios, se elige la sub – área que pueda genera la menor pérdida de espacio y se inserta. Cabe destacar que para obtener un resultado óptimo, se debe ordenar de forma decreciente con respecto al tamaño del bloque, en caso contrario, puede generar pequeños espacios y el costo de evaluar el solapamiento de bloques es muy alto.

3.8.2 Strips and Pointers

La siguiente propuesta se denominó *Strips and Pointer* (Estantes y Apuntadores). Se crea una abstracción lógica de la textura atlas en un espacio unidimensional, representando su área en forma de un estante (*Strip*) de gran longitud, tomando como altura el tamaño del bloque en su resolución más fina. Los bloques serán insertados en este *Strip* bajo la regla de que siempre deben estar en orden decreciente según su tamaño. Se puede visualizar la representación de un *Strip* en la **Figura 3.15**, donde se pueden ver los bloques ordenados de mayor a menor según su nivel de detalle.

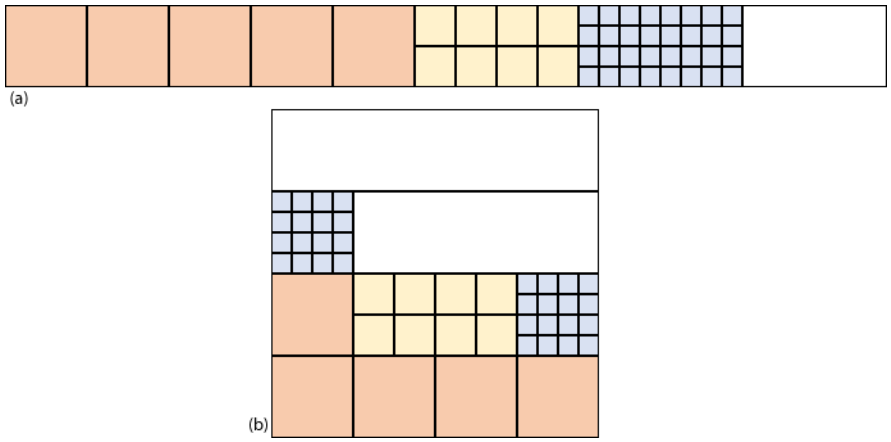


Figura 3.15: Representación de la textura atlas en un *Strip* (a). La figura (b) es la textura atlas correspondiente al Strip (a) en un espacio **2D**.

Para pasar de las coordenadas del *Strip* a las coordenadas del atlas, se desarrolló la **Ec. 3.3**,

$$x_{atlas} = x_{strip}\ mod\ dim_{atlas}$$

$$y_{atlas} = \left(\frac{x_{strip}}{dim_{atlas}}\right) mod \left(\frac{dim_{atlas}}{LOD_{max}}\right) + y_{strip}$$

[Ec. 3.3]

donde x_{atlas} y y_{atlas} son las coordenadas en la textura atlas, x_{strip} y y_{strip} son las coordenadas en el estante, dim_{atlas} es la dimensión del atlas y LOD_{max} es el tamaño del bloque en su representación más fina. Con ésta ecuación, podemos calcular las coordenadas finales del bloque en el atlas.

Para calcular las coordenadas del bloque en el *Strip*, se usan unas coordenadas base llamadas *Pointers*. Estas son pre-calculadas tomando en cuenta el área ocupada por los bloques en cada nivel de detalle y los anteriores, y se ubican al principio de cada nivel de detalle. Esto genera diferentes sub-áreas entre cada nivel de detalle delimitados por estos *Pointers*, ordenadas desde el nivel más fino al más bajo para validar el orden de los bloques.

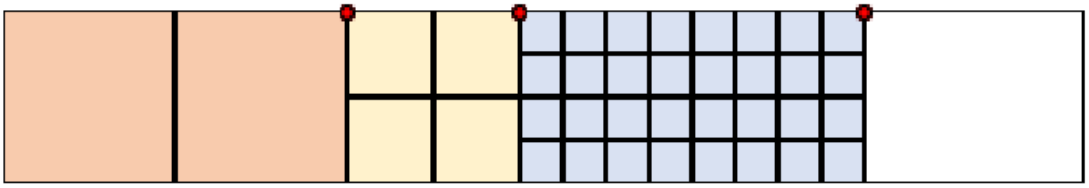


Figura 3.16: Representación de los Pointers (puntos rojos) en la técnica *Strip and Pointers*. Estos apuntadores solo se mueven solo en la coordenada ***x***.

Para insertar los bloques en el *Strip*, se deben enumerar los bloques por cada nivel de detalle. Esta indexación ordena los bloques en forma lineal y calcula su espacio en el *Strip* a través de su *Pointer*. Para calcular su posición, se presenta la **Ec. 3.4**,

$$x_{strip} = Pointer_x + \frac{Index_{block}}{dimY_{strip}}$$

$$y_{strip} = Pointer_y + (Index_{block}\ mod\ dimY_{strip})$$

[Ec. 3.4]

Donde x_{strip} y y_{strip} representa la coordenada del bloque en el *Strip*, $Pointer_x$ y $Pointer_y$ el *Pointer* del nivel de detalle del bloque, $Index_{block}$ el índice del bloque en la estructura de datos y $dimY_{strip}$ la dimensión del *Strip* en y. Los bloques están clasificados en listas independientes por niveles de detalles, como se pude visualizar en la **Figura 3.17,** donde cada lista empieza desde la posición 0.

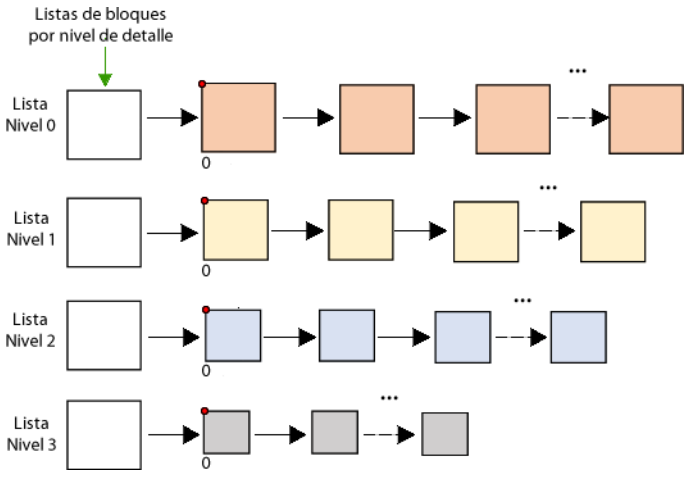


Figura 3.17: Lista de bloques por niveles de detalles. Cada lista tiene la coordenada del *Pointer* (puntos rojos) asociado a su nivel de detalle.

Cuando una sub-área de un nivel de detalle se encuentra llena, los *Pointers* deben realizar un proceso de traslación en el eje x para que el bloque pueda ser insertado en el *Strip*. Esta traslación debe ser de tamaño del bloque nuevo a insertar para que encaje en la sub-área. La traslación se puede hacer hacia la derecha o hacia la izquierda. Primero se realiza un cálculo de validación de traslación para determinar hacia qué lado debe hacerse. Cuando se realiza la traslación a la derecha (ver **Figura 3.18**) se seleccionan todos los bloques que puedan solaparse y se reinsertan en su sub-área si la misma tiene el espacio suficiente, en caso contrario, debe realizar el mismo proceso de traslación en el siguiente nivel de detalle menor hasta que puedan ser insertados todos los bloques. Si no hay espacio suficiente para trasladar y reinsertar a la derecha, se hace un proceso de inserción a la izquierda, siempre y cuando exista el espacio suficiente para insertar todos los bloques, de ser así, se reinsertan los bloques a la izquierda con sus nuevos índices y se traslada el *Pointer*.

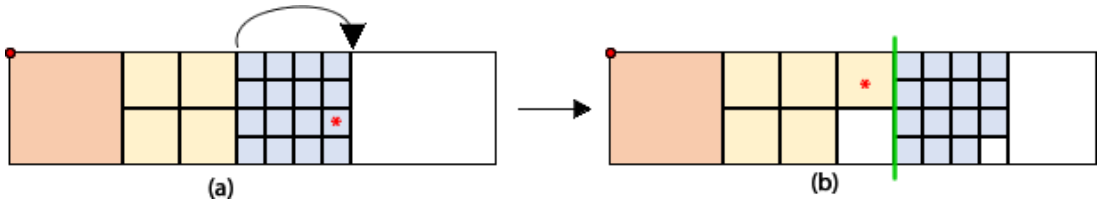


Figura 3.18: Translación de un *Pointer* hacia la derecha. Se deben mover 8 bloques para que se pueda insertar el bloque refinado.

El algoritmo genera mejores resultados que los algoritmos planteados anteriormente, debido a que la partición es representada por cada una de las sub-áreas por niveles de detalles, delimitadas entre los *Pointers*. Por otro lado, es costoso calcular si existen bloques que requieren ser removidos y reinsertados a las sub-áreas correspondientes, ya que la traslación se realiza en el eje x, y como consecuencia de no poder insertar un bloque de ese nivel de detalle, podría generar un gran movimiento de bloques, y lo que se quiere es trasladar el *Pointer* minimizando la cantidad de bloques a mover.

3.8.3 3D Z-order Strip

Basado en el algoritmo anterior, se desarrolló una propuesta donde se utiliza un *Strip* para insertar los bloques y un *Pointer* por cada nivel de detalle para delimitar los mismos.

Este algoritmo utiliza una indexación de bloques tomando en cuenta su tamaño y la cantidad de bloques existentes por cada nivel de detalle, el ordenamiento de los bloques en forma decreciente y las traslaciones de puntos en un *Strip* unidimensional, donde cada bloque en el *Strip* tendrá una posición correspondiente en la textura atlas, la cual es calculada con una fórmula de correspondencia.

Se ha implementado una estructura de ordenamiento de bloques la cual no es más que una lista de bloques para cada nivel detalle, donde cada bloque está indexado por el índice lógico del bloque en cada nivel de detalle multiplicado por el espacio que ocupa este, más la cantidad de bloques en los niveles de detalles anteriores y por el tamaño del bloque correspondiente a cada nivel, como se explica en la ecuación **Ec.3.5**. El último bloque de cada lista apunta al bloque inicial de la lista del nivel detalle siguiente, en caso de que ésta exista. La estructura de ordenamiento está representada en la **Figura 3.19**. Este ordenamiento e indexación tiene como ventaja organizar y almacenar la mayor cantidad de bloques en el *Strip* de forma eficiente.

$$b_i^{lod0} = i * tamBloque_{lod0}^3 \ \ , con\ 0 \leq i < n$$

$$b_j^{lod1} = n * tamBloque_{lod0}^3 + j * tamBloque_{lod1}^3 \ \ , con\ 0 \leq j < m\ y\ n \geq 0$$

$$b_k^{lod2} = n * tamBloque_{lod0}^3 + m * tamBloque_{lod1}^3 \ + k * tamBloque_{lod2}^3 \ ,$$

$$n, m \geq 0\ y\ 0 \leq k < p$$

[Ec. 3.5]

Donde b_i^{lodx} representa el índice del bloque con índice lógico i del nivel de detalle $lodx$ con $i, x \geq 0$ y n, m y p representan la cantidad de bloques existentes por cada nivel de detalle y $tamBloque_{lodx}^3$ es el tamaño del bloque del nivel de detalle x elevado al cubo para un ambiente tridimensional ($tamBloque_{lodx}^2$ para un ambiente bidimensional).

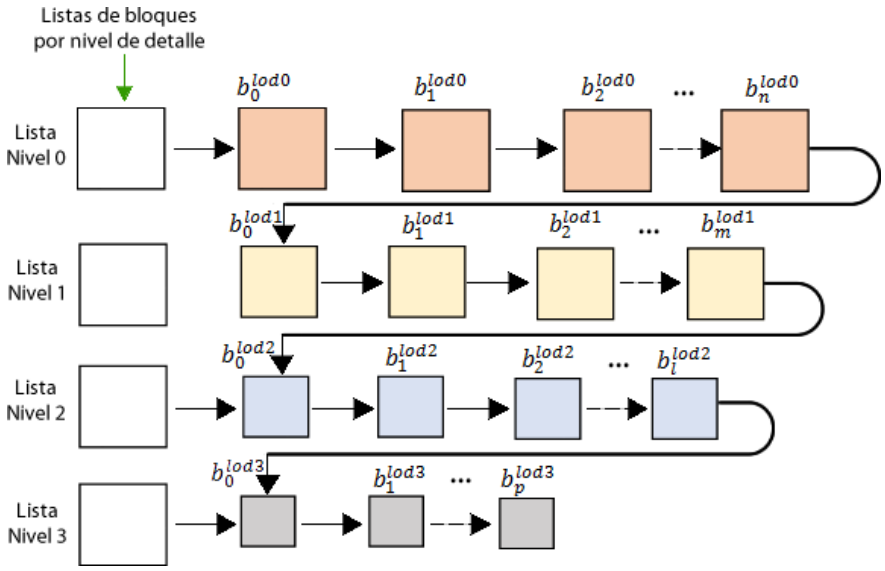


Figura 3.19: Descripción gráfica de las listas de bloques para su ordenamiento. La notación $\boldsymbol{b_i^{lodx}}$ representa el índice de cada bloque previamente calculado con la ecuación **[Ec. 3.5]**.

Una vez ordenados todos los bloques y calculado sus índices se procede a insertarlos en el *Strip* unidimensional respetando un orden específico. Este orden propuesto se denomina *Z-Order*, que consiste en ordenar los bloques indexados (ver **Figura 3.20**) en forma de una letra *Z*, donde se realiza una correspondencia de los índices de la estructura ordenada de bloques al *Strip* unidimensional.

Para calcular el orden *Z* de los bloques, se sub-divide el *Strip* en tiras de tamaño del nivel de detalle más fino, una vez que se ubique la tira correspondiente, se procede a generar una sub-división en forma de *Quad-Tree*, donde cada sub-área es representada en un rango específico, el cual determinará en qué posición debe ir el bloque en el *Strip* respetando el patrón del *Z-Order* (ver **Figura 3.21**). Según el sub-área que genere el algoritmo, se generan una cantidad movimientos en las coordenadas de la tira hasta llegar al nivel de detalle del bloque (ver **Algoritmo 3.1**). El valor de la sub-área dada por la sub-división indica el movimiento a realizar, como podemos ver en **Ec. 3.6**,

$$x = x + offset\ si\ (tile\ mod\ 2 == 1)$$

$$y = y + offset\ si\ (tile >= 2)$$

[Ec. 3.6]

donde se mueve en la coordenada x si la sub-área es impar y movemos en y si es mayor o igual que dos. Este algoritmo se realiza recursivamente hasta llegar a la coordenada requerida. Se pudo desarrollar una solución iterativa (ver **Algoritmo 3.1**) del algoritmo generando las sub-divisiones cuando se recorren los niveles de detalle.

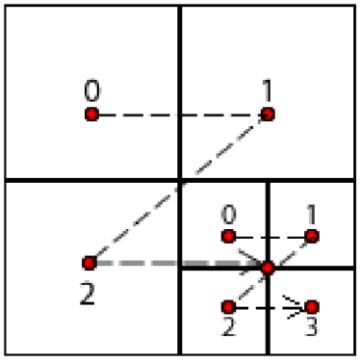


Figura 3.20: Representación del *Z-Order* en una tira en el *Strip*.

```
Funcion z-order(entero indice, entero lod, entero s=maxTamLOD) : retorna vector2
{
  //Valores x,y con la posición del bloque en coordenadas de Strip
  entero x, y;
  //Calculo de la posicion de la tira
  entero tira = indice /(s*s);
  //Se inicializan las coordenadas x,y
  x = tira * s; y = 0;
  //Se resta el área sobrante para simular el Quad-Tree
  indice -= tira*s*s;
  //Bucle para realizar los movimientos necesarios
  Para LOD maximo hasta LOD actual Hacer
  {
    //Se genera el nuevo LOD
    s >>= 1;
    //Se actualiza el valor de la tira
    tira = indice /(s*s);
    //Si la tira es mayor que 2, mueve en coordenada y
    Si tira >= 2 Entonces
      y += s;
    //Si la tira es impar, mueve en coordenada x
    Si tira % 2 == 1 Entonces
      x += s;
    //Se resta el área sobrante para simular el Quad-Tree
    indice -= tira*s*s;
  }
  retorna vector2(x, y);
}
```

Algoritmo 3.1: Algoritmo pseudo formal para el cálculo de las coordenadas bidimensionales de un bloque para la inserción en un *Strip*.

El resultado de ordenar los bloques a través de ésta técnica lo podemos visualizar en la **Figura 3.21**. Dicho orden se genera de forma eficiente y de tal forma de mover la mínima cantidad de bloques al momento de trasladar *Pointers*.

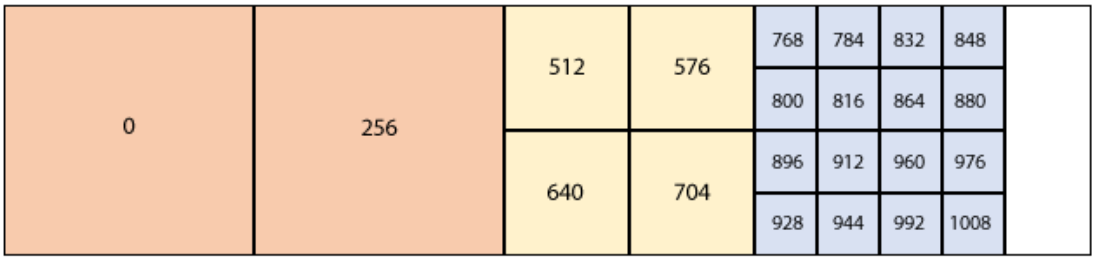


Figura 3.21: Los bloques ordenados por el algoritmo de *Z-Order*, se puede ver que los bloques están indexados por áreas para no depender del Pointer en cada nivel de detalle.

Por ejemplo, si se quiere hallar la coordenada del bloque en el índice 1008 de la **Figura 3.21**, con nivel de detalle 2, podemos hacer una corrida del algoritmo en la **Tabla 3.1**,

Índice 1008	**Tamaño 16**	**Tamaño 8**	**Tamaño 4**
Coordenadas (x,y)	(48,0)	(56,8)	(64,12)
Nivel de detalle	0	1	2
Tira	3	3	3
Espacio disponible	240	48	0

Tabla 3.1: Corrida en frio del **Algoritmo 3.1**.

Donde la coordenada inicial se encuentra en (48,0) y al evaluar la nueva sub-área, da como resultado que es impar y mayor que dos, por lo que se hacen movimientos de coordenadas en $\boldsymbol{x}$ y $\boldsymbol{y}$ en ocho (8) posiciones. Se recalcula la tira en base al espacio que nos hemos quitado y volvemos a evaluar en el siguiente nivel de detalle. Como la sub-área dio nuevamente tres (3), se vuelven a mover las coordenadas en los dos ejes en cuatro (4) posiciones por ser de nivel de detalle 2 para tamaño de bloque 16.

Cuando no sobra más espacio entre *Pointers* para insertar nuevos bloques, se debe hacer un proceso de traslación de *Pointers*. Con el *Z-Order*, a diferencia de la propuesta anterior que genera las coordenadas a través de estos *Pointers*, se pueden trasladar tanto en

el eje x como en el eje y, esto se debe a que este algoritmo ordena los bloques en el *Strip* de tal forma que se pueden mover solamente los bloques que ocupen el área necesaria para insertar el nuevo bloque refinado sin perder el orden en la estructura de datos.

Existen dos formas de trasladar *Pointers*, a la derecha y a la izquierda, en la lista unidimensional que contiene todos los bloques ordenados. La traslación hacia la derecha se da si no hay más espacio en la sub-área del nivel de detalle al que se desea refinar (ver **Figura 3.22**). Para realizar este proceso, se toman todos los bloques desde la primera posición que puedan ocupar el área del bloque a refinar. Una vez que se tienen todos los bloques que se deben mover, se inserta el bloque a refinar y se reinsertan los bloques en las sub-áreas restantes. Cabe destacar que se puede dar el caso en el que se deban mover más de un *Pointer* para que encajen todos los bloques, para esto se hace un proceso iterativo que recorre el *Strip* hasta el nivel más bajo de ser necesario, recalculando los espacios necesarios.

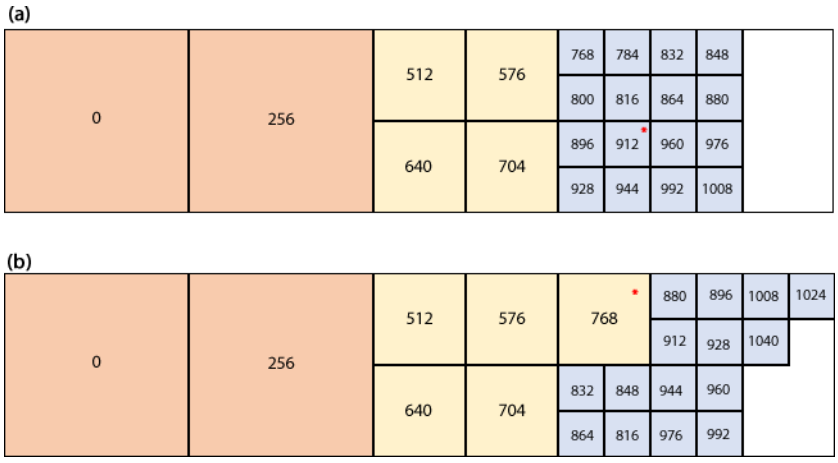


Figura 3.22: Representación gráfica de la traslación a la derecha, se quiere refinar el bloque con índice 912 (**a**) por lo que se mueven los 4 primeros bloques del nivel anterior y se inserta el bloque refinado cuyo índice pasa a ser 768 (**b**), los 4 bloques que fueron movidos se a los espacios libres correspondientes a su nivel de detalle. El bloque 768 de la figura (**a**) pasa a ser el 848 en la figura (**b**) ya que ocupa el espacio libre que dejó el bloque que se refinó, los bloques 784, 800 y 816 de la figura (**a**) ahora son los bloques 1008, 1024 y 1040 respectivamente, en la figura (**b**).

Para un mejor entendimiento de la traslación, se presenta el **Algoritmo 3.2**, donde se puede detallar como se realizó el proceso de traslación de una forma eficiente.

```
Funcion Traslado_der(bloque b) : retorna Booleano
{
  //Lista de bloques a insertar ordenados por nivel de detalle
  bloque lista_bloques_reinsertar [Cantidad_LOD][…];
```

```
  //Variable que indica el espacio necesario para insertar bloques del nivel actual
  entero mb = 0, numero_bloques= 1;
  //Validar si hay espacio suficiente a la derecha para hacer las traslaciones
necesarias
  Si No validar_espacio_derecha() Entonces
      retorna falso;
  //Bucle para realizar los movimientos necesarios
  Para LOD refinado hasta LOD minimo Hacer
  {
    //Calculamos área del nivel de detalle actual y del mas pequeño
    area_lod = pointers_LOD[LOD].area;
    area_lod_ant = pointers_LOD[LOD + 1].area;
    //Calculamos área del nivel de detalle actual y del mas pequeño
    pointers_LOD[LOD].pointer +=  mb + (area_lod* numero_bloques);
    //Se calcula área del nivel de detalle actual y del mas pequeño
    mb += area_lod * numero_bloques;
    //cantidad de bloques que se deben quitar del siguiente nivel de detalle
    entero cantidad_bloques = mb / area_lod_ant;
    //bucle que remueve los bloques de la lista para ser reinsertados
posteriormente
    Mientras cantidad_bloques <> 0 || mb <> 0 Hacer
    {
        lista_bloques_reinsertar [LOD + 1][i] = lista_bloques [LOD + 1][0];
        remover(lista_bloques [LOD + 1][0]);
    }
    //Insertamos los bloques del nivel de detalle actual
    Insertar_bloques(lista_bloques_reinsertar [LOD][…]);
    //Insertamos los bloques del anterior nivel de detalle
    Insertar_bloques(lista_bloques_reinsertar [LOD+1][…]);
    //Si el tamaño de la lista del ant. LOD no es cero, quiere decir que no
encajaron todos los bloques y se debe realizar una nueva movida de apuntador del
anterior nivel de detalle
    Si lista_bloques_reinsertar [LOD+1][…].tamaño == 0 Entonces
      Break;
    Sino
      numero_bloques = lista_bloques_reinsertar [LOD+1][…].tamaño;
  }
  retorna verdadero;
}
```

Algoritmo 3.2: Algoritmo pseudo formal para el traslado de bloques a la derecha.

Si la validación del espacio en el **Algoritmo 3.2** no se cumple, la función retorna falso porque no se tiene el suficiente espacio hacia la derecha para mover los bloques necesarios para insertar el bloque refinado. Si no se puede trasladar a la derecha, el algoritmo busca espacio a la izquierda (ver **Figura 3.23**) del nivel de detalle a refinar para insertar el bloque refinado. El proceso consiste en tomar el último espacio del nivel de detalle mayor al que se quiere refinar. Si el espacio es ocupado por un bloque, se debe mover este bloque a un espacio libre. Este proceso se ejecuta iterativamente, al igual que la traslación a la derecha, debido a que se puede mover más de un *Pointer* a la izquierda por falta de espacio, aunque se debe validar previamente que tenga el espacio suficiente para trasladar.

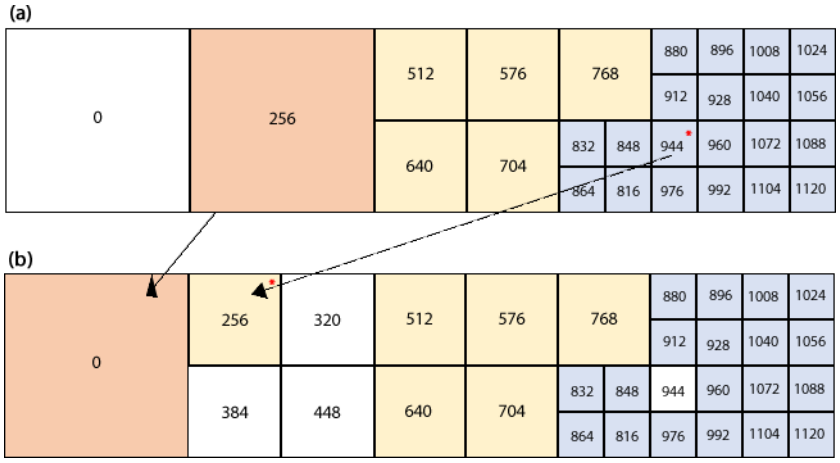


Figura 3.23: Representación gráfica de la traslación a la izquierda, se quiere refinar el bloque con índice 944 (**a**) por lo que se mueve el último bloque del nivel mayor y se inserta el bloque refinado (**b**) al final se insertan el bloque que fue movido a un espacio libre correspondiente a su nivel de detalle.

El **Algoritmo 3.3** se asemeja al planteado anteriormente con traslación a la derecha. Con diferencia que saca el último bloque de la lista nada más si en el espacio existe un bloque. Si la función retorna falso, no se puede realizar traslación a la izquierda y se requiere de un proceso de Colapso de bloques para que exista el espacio necesario en el *Strip*. Por cada colapso de bloque, se valida nuevamente el espacio requerido por la derecha y por la izquierda, dando como prioridad el área de la derecha.

```
Funcion Traslado_izq(bloque b) : retorna Booleano
{
  //Lista de bloques a insertar ordenados por nivel de detalle
  bloque lista_bloques_reinsertar [Cantidad_LOD][…];
  //Variable que indica el espacio necesario para insertar bloques del nivel actual
  entero mb = 0, numero_bloques= 1;
  //Validar si hay espacio suficiente a la derecha para hacer las traslaciones
necesarias
  Si No validar_espacio_izquierda() Entonces
      retorna falso;
  //Bucle para realizar los movimientos necesarios
  Para LOD refinado hasta LOD maximo Hacer
  {
    //Calculamos área del nivel de detalle actual y del mas pequeño
    area_lod = pointers_LOD[LOD].area;
    area_lod_sig = pointers_LOD[LOD-1].area
    //Calculamos área del nivel de detalle actual y del mas pequeño
    pointers_LOD[LOD].pointer -=  mb + (area_lod_sig* numero_bloques);
    //Se calcula área del nivel de detalle actual y del mas pequeño
    mb += area_lod * numero_bloques;
    //Si existe un bloque en el espacio, quitarlo para reinsertarlo posteriormente
    Si mover_bloque_izquierda() Entonces
    {
        lista_bloques_reinsertar [LOD-1][i] = lista_bloques [LOD-1][ultimo];
```

```
        remover(lista_bloques [LOD-1][ultimo]);
    }
    //Insertamos los bloques del nivel de detalle actual
    Insertar_bloques(lista_bloques_reinsertar [LOD][…]);
    //Insertamos los bloques del siguiente nivel de detalle
    Insertar_bloques(lista_bloques_reinsertar [LOD-1][…]);
    //Si el tamaño de la lista del sig. LOD no es cero, quiere decir que no encajo
el bloque que se quitó y se debe realizar una nueva movida de apuntador del
siguiente nivel de detalle
    Si lista_bloques_reinsertar [LOD-1][…].tamaño == 0 Entonces
      Break;
    Sino
      numero_bloques = lista_bloques_reinsertar [LOD+1][…].tamaño;
  }
  retorna verdadero;
}
```

Algoritmo 3.3: Algoritmo pseudo formal para el traslado de bloques a la izquierda.

Esta última propuesta fue la elegida para la implementación de la inserción y reajuste de los bloques en la textura atlas. Se puede evaluar en primera instancia que el algoritmo aprovecha la mayor cantidad de espacio por estar ordenado de forma decreciente, además de poder minimizar la cantidad de bloques a mover en caso de ser necesario y permite determinar la posición exacta de cada bloque en el atlas sin la necesidad de hacer un recorrido por cada bloque existente, obteniendo así una inserción de bloques eficiente. En el capítulo siguiente se desglosan los ambientes de pruebas con sus respectivos volúmenes y resultados obtenidos.

Capítulo IV

Pruebas y resultados

En este capítulo se presentan los ambientes de prueba, las pruebas realizadas y los resultados obtenidos del algoritmo implementado para el manejo eficiente de la segmentación de la textura atlas.

4.1 Ambiente de pruebas

El sistema de despliegue de volúmenes multi-resolución fue probado en 2 computadores con diferentes características de hardware para medir los tiempos de respuestas y niveles de fragmentación del sistema cargando diferentes volúmenes. A continuación se describen las especificaciones de hardware y software de ambos computadores.

4.2 Especificaciones de hardware

El Algoritmo de volúmenes multi-resolución requiere de una gran capacidad de Hardware para poder generar mejores resultados, por eso se eligieron dos equipos para hacer la comparación entre los tiempos de respuestas. Cabe acotar que se trabajó con dispositivos NVIDIA para la tarjeta gráfica y dispositivos Intel para Procesamiento.

- PC Intel i5 de 2.5 Ghz, con 16 Gb de RAM DDR3 y una tarjeta Gráfica Nvidia Gerforce 680GTX de 2Gb de RAM. Esta configuración será llamada “Equipo 1”.

- PC Procesador Intel Quad Core modelo 8400 de 2.4Ghz, con 8 Gb de RAM DDR3 y una tarjeta gráfica Nvidia Geforce 550 GTX de 1.0Gb de RAM. Esta configuración será llamada "Equipo 2".

4.3 Especificaciones de software

- Sistema Operativo: Windows 8.1 64 Bit para cargar volúmenes de gran tamaño
- Lenguaje de Programación: C++, lenguaje nativo del sistema para correr de forma eficiente el motor gráfico.
- IDE de desarrollo: Visual Studio 2013 para compilar y desarrollar la propuesta de tesis.
- Motor gráfico: OpenGL versión 4 con soporte para programar en el procesador de vértices y fragmentos a través de GLSL, con manejador de ventanas GLUT e interfaz gráfica AntTweakBar 1.15 **[60]**. Se utilizó Glew 1.5.8 para usar las extensiones del API de OpenGL.
- Repositorio remoto: GIT para almacenar en la web las versiones de la aplicación y poder obtener la última versión en cualquier PC.

4.4 Datasets

Para las pruebas se usaron 3 volúmenes diferentes, el primero es la mujer visible redimensionada, el segundo volumen será una muestra de una flor de nueces y el tercer volumen es una fruta haitiana. Todos los volúmenes desplegados serán generados con la técnica de *Ray Casting* de una pasada.

4.4.1 Volumen 1

Del proyecto Humano Visible se tienen los cortes fotografiados en RGB de la mujer visible convertidos a escala de grises (ver **Figura 4.1**). Por limitaciones de este trabajo, se utilizó únicamente una muestra de 867 MB que corresponde a una tomografía. Existe una muestra del volumen de 12 GB que son una colección de fotos convertidas a escala de grises. La dimensión de este sub volumen es de 512x512x1734 donde cada vóxel tiene 16 bits. De ahora en adelante se referirá a este como **"**Volumen 1".

Figura 4.1: Muestra de la mujer visible redimensionada.

4.4.2 Volumen 2

Para la prueba número 2 se utilizó el volumen de una flor (leucadendron rubrum) seca (ver **Figura 4.2**) de 1 GB, tomado de una micro tomografía computarizada. Sus dimensiones son 1024x1024x1024 y cada vóxel tiene 8 bits. A este volumen se le llamará "Volumen 2".

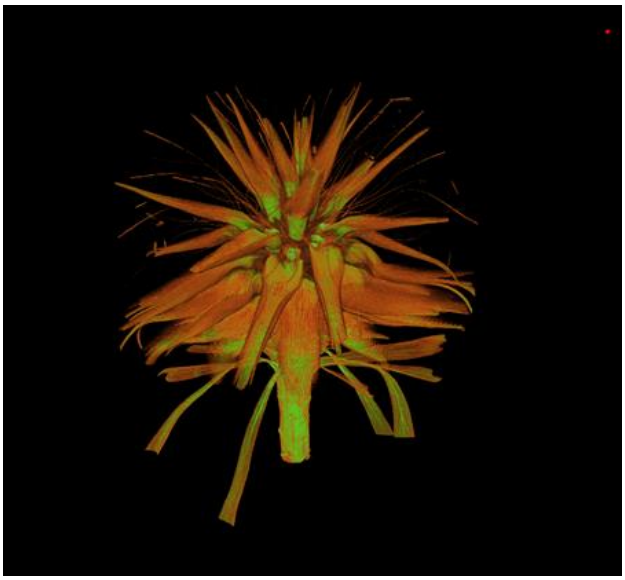


Figura 4.2: Volumen de flor seca.

4.4.3 Volumen 3

El tercer y último volumen de prueba será una fruta haitiana (ver **Figura 4.3**) de dimensión 1024x1024x1536 de 3.01GB con 16 bits por cada vóxel. Este volumen también fue generado por una micro tomografía computarizada y se identificará como "Volumen 3".

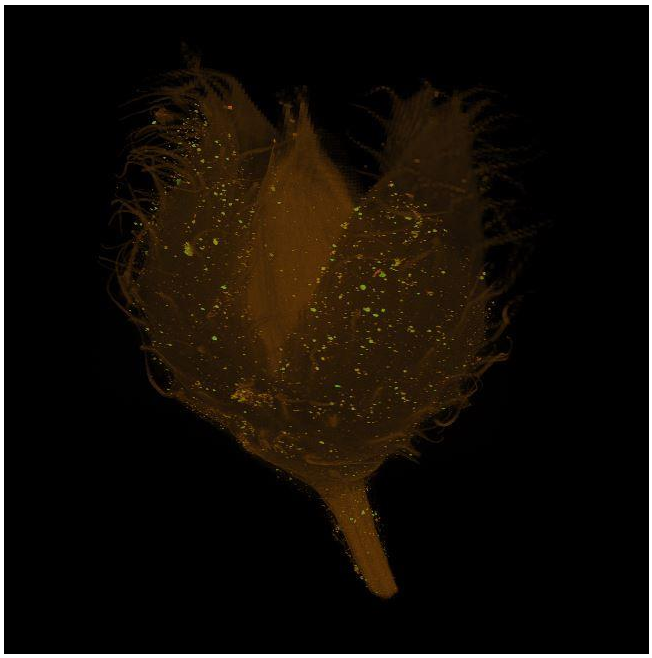


Figura 4.3: Volumen de fruta haitiana de 3.01GB

4.5 Resultados Cuantitativos

En esta sección se mostrarán los resultados cuantitativos, como mediciones de tiempo de carga, procesamiento y consumo de memoria obtenidos al desplegar cada volumen. Para estas pruebas se utilizó jerarquía en bloques con máximo seis niveles de detalle y la textura atlas con tres tamaños distintos.

4.5.1 Etapa de pre-procesamiento

En esta sección se mostrará una tabla donde se refleja la cantidad total de bloques que se generan, la memoria requerida para almacenar todo el volumen multi-resolución y el tiempo utilizado para la generación de los niveles de detalle (ver **Tabla 4.1**).

		Volumen 1		Volumen 2		Volumen 3	
		16^3	32^3	16^3	32^3	16^3	32^3
Bloques Generados		110.976	13.872	262.144	32.768	393.216	49.152
Memoria necesaria (MB)		990,82	990,85	1.170,25	1.170,28	3.510,75	3.510,84
Tiempo para generar los niveles de detalle (ms.)	Equipo 1	1.277,00	680,00	1.298,33	1.627,66	4.883,85	2.669,25
	Equipo 2	1.959,37	1.828,12	3.267,73	2.531,66	N/P	N/P

Tabla 4.1: Tabla descriptiva de los tiempos y memoria utilizada en la generación de los niveles de detalle.

Se puede notar que la cantidad de bloques generados para un tamaño de bloque igual a 32 es significativamente menor a la de 16, esto es así porque aunque se tiene un nivel de detalle adicional (uno para cada tamaño de bloque (32, 16, 8, 4, 2, 1)), cada bloque contiene mayor cantidad de vóxeles y esto es inversamente proporcional a la cantidad de bloques. Al tener menor cantidad de bloques el proceso para generar los niveles de detalle es más rápido.

4.5.2 Refinamiento

La siguiente tabla refleja la memoria utilizada y memoria desperdiciada al momento cuando de realizar el proceso de refinamiento y cuánto tiempo toma para refinar todos los bloques posibles (ver **Tabla 4.2**, **Tabla 4.3** y **Tabla 4.4**).

		Volumen 1		Volumen 1		Volumen 1	
Tamaño atlas		256x256x256 (16 MB)		512x512x512 (128 MB)		1024x1024x1024 (1GB)	
Tamaño de Bloques		16^3	32^3	16^3	32^3	16^3	32^3
Tiempo de carga (ms)		440	440	440	440	440	440
Memoria utilizada (MB)		15,999576	15,99512	127,998047	127,96094	212,914063	228,593750
Memoria libre (MB)		0,000424	0,00488	0,001953	0,03906	811,085937	795,40625
% de memoria utilizada		99,997350	99,969500	99,998474	99,969484	20,792389	22,323608
Tiempo de refinamiento (ms.)	Equipo 1	461,825	181,8525	550,675	242,2725	696,965	365,3425
	Equipo 2	979,11	501,42	1.259,28	661,64	1.733,31	803,77

Tabla 4.2: Tabla descriptiva de los tiempos y memoria utilizada del volumen 1 en el proceso de refinamiento de bloques con atlas de tamaño diferentes.

		Volumen 2		Volumen 2		Volumen 2	
Tamaño atlas		256x256x256 (16 MB)		512x512x512 (128 MB)		1024x1024x1024 (1GB)	
Tamaño de Bloques		16^3	32^3	16^3	32^3	16^3	32^3
Tiempo de carga (ms)		510	510	510	510	510	510
Memoria utilizada (MB)		16	15,997070	121,929688	127,976563	121,929688	130,218750
Memoria libre (MB)		0	0,00293	6,070312	0,023437	902,070312	893,78125
% de memoria utilizada		100,00	99,981688	95,257569	99,981690	11,907196	12,716675
Tiempo de refinamiento (ms.)	Equipo 1	256,8675	177,8825	302,8975	206,765	361,525	198,605
	Equipo 2	766,465	657,155	766,6975	690,3725	1021,855	613,7625

Tabla 4.3: Tabla descriptiva de los tiempos y memoria utilizada del volumen 2 en el proceso de refinamiento de bloques con atlas de tamaño diferentes.

		Volumen 3		Volumen 3		Volumen 3	
Tamaño atlas		256x256x256 (16 MB)		512x512x512 (128 MB)		1024x1024x1024 (1GB)	
Tamaño de Bloques		16^3	32^3	16^3	32^3	16^3	32^3
Tiempo de carga (ms)		15.624,93	15.624,93	15.624,93	15.624,93	15.624,93	15.624,93
Memoria utilizada (MB)		15,08941	16	127,999878	127,999512	1.023,99902	1.023,99609
Memoria libre (MB)		0,91059	0	0,000122	0,000488	0,00098	0,00391
% de memoria utilizada		94,308813	100,00	99,999905	99,999619	99,999904	99,999618
Tiempo de refinamiento (ms.)	Equipo 1	2.294,9575	368,39	2.584,425	590,7225	5.566,2775	1.471,2125
	Equipo 2	N/P	N/P	N/P	N/P	N/P	N/P

Tabla 4.4: Tabla descriptiva de los tiempos y memoria utilizada del volumen 3 en el proceso de refinamiento de bloques con atlas de tamaño diferentes. El volumen 3 no pudo ser probado (N/P=No Probado) en el equipo 2 por no tener la capacidad de memoria RAM necesaria para cargar 3.01GB de datos.

Se puede observar que el algoritmo Z-Order almacena los bloques eficientemente para usar el máximo espacio posible en la textura atlas. La textura atlas es usada casi en totalidad cuando su tamaño es significativamente menor al tamaño de volumen, ocupando entre el 99% y 100% de la memoria y en el peor de los casos es cercano al 94% de su espacio. Cuando el tamaño del volumen es menor que la textura atlas, el volumen necesita menos espacio para refinarse completamente, utilizando solo entre un 11% y 23% del tamaño disponible.

La siguiente tabla indica los tiempos que tarda la actualización de las prioridades de cada bloque cuando se mueve el punto de interés para volumen, así como el tiempo de *rendering*, inserción de bloques en la estructura de ordenamiento *Z-Order* y el tiempo de actualización de la textura atlas en GPU. Los tiempos son calculados en milisegundos y son el promedio de los tiempos tomados por cada 10 *frames* del *render*.

		Volumen 1		Volumen 1		Volumen 1	
Tamaño atlas		**256x256x256 (16 MB)**		**512x512x512 (128 MB)**		**1024x1024x1024 (1GB)**	
Tamaño de Bloques		**16^3**	**32^3**	**16^3**	**32^3**	**16^3**	**32^3**
Tiempo de cálculo de prioridades (ms.)	**Equipo 1**	0,01	0,01	0,01	0,01	0,01	0,01
	Equipo 2	0,02	0,00	0,02	0,00	0,02	0,00
Tiempo de *rendering* (ms.)	**Equipo 1**	49,08	4,81	33,34	5,40	61,13	9,04
	Equipo 2	71,65	12,40	72,38	12,60	117,15	23,00
Tiempo de inserción en estructura Z-Order (ms.)	**Equipo 1**	16,02	1,21	11,78	1,51	11,78	1,33
	Equipo 2	18,96	2,23	20,28	2,26	20,29	2,29
Tiempo de actualización de textura atlas en GPU (ms.)	**Equipo 1**	7,39	4,81	5,04	0,70	5,18	0,52
	Equipo 2	5,9	1,19	5,84	1,23	5,52	1,12

Tabla 4.5: Tabla descriptiva de los tiempos de cálculo en el volumen 1 de prioridades cuando se mueve el punto de interés, tiempo de *rendering*, inserción de bloques en la estructura de ordenamiento *Z-Order* y el tiempo de actualización de la textura atlas en GPU para el volumen de la mujer visible.

		Volumen 2		Volumen 2		Volumen 2	
Tamaño atlas		**256x256x256 (16 MB)**		**512x512x512 (128 MB)**		**1024x1024x1024 (1GB)**	
Tamaño de Bloques		**16^3**	**32^3**	**16^3**	**32^3**	**16^3**	**32^3**
Tiempo de cálculo de prioridades (ms.)	**Equipo 1**	0,01	0,00	0,01	0,00	0,01	0,00
	Equipo 2	0,01	0,00	0,01	0,00	0,01	0,00
Tiempo de rendering (ms.)	**Equipo 1**	45,17	2,68	17,90	3,35	34,66	5,14
	Equipo 2	120,08	11,96	50,46	7,09	34,02	12,98
Tiempo de inserción en estructura Z-Order (ms.)	**Equipo 1**	25,22	0,71	6,44	0,71	6,55	0,75
	Equipo 2	43,85	2,15	10,68	1,25	10,32	1,33
Tiempo de actualización de textura atlas en GPU (ms.)	**Equipo 1**	5,78	0,30	2,35	0,25	2,29	0,25
	Equipo 2	10,63	1,19	4,29	0,52	3,46	0,51

Tabla 4.6: Tabla descriptiva de los tiempos de cálculo en el volumen 2 de prioridades cuando se mueve el punto de interés, tiempo de *rendering*, inserción de bloques en la estructura de ordenamiento *Z-Order* y el tiempo de actualización de la textura atlas en GPU para el volumen de la flor seca.

	Volumen 3	Volumen 3	Volumen 3
Tamaño atlas	**256x256x256 (16 MB)**	**512x512x512 (128 MB)**	**1024x1024x1024 (1GB)**

Tamaño de Bloques		16^3	32^3	16^3	32^3	16^3	32^3
Tiempo de cálculo de prioridades (ms.)	Equipo 1	0,15	0,01	0,16	0,01	0,14	0,01
	Equipo 2	N/P	N/P	N/P	N/P	N/P	N/P
Tiempo de rendering (ms.)	Equipo 1	183,22	42,76	183,93	30,58	283,84	57,09
	Equipo 2	N/P	N/P	N/P	N/P	N/P	N/P
Tiempo de inserción en estructura Z-Order (ms.)	Equipo 1	142,45	14,16	141,71	10,80	140,89	11,03
	Equipo 2	N/P	N/P	N/P	N/P	N/P	N/P
Tiempo de actualización de textura atlas en GPU (ms.)	Equipo 1	33,06	7,12	33,02	4,48	34,33	4,97
	Equipo 2	N/P	N/P	N/P	N/P	N/P	N/P

Tabla 4.7: Tabla descriptiva de los tiempos de cálculo en el volumen 3 de prioridades cuando se mueve el punto de interés, tiempo de *rendering*, inserción de bloques en la estructura de ordenamiento *Z-Order* y el tiempo de actualización de la textura atlas en GPU para el volumen de la fruta haitiana de 3.01GB.

Se puede observar que la cantidad de bloques generados para cada volumen influye significativamente en los tiempos calculados, a menor cantidad de bloques menor tiempo de procesamiento. Los tiempos de cálculos de prioridad de los bloques son significativamente pequeños, debido a que la prioridad es calculada al principio de cada refinamiento y es O(1) ya que solo se requiere una fórmula para generar el error de cada bloque. El tiempo d actualización del atlas es mucho menor al tiempo de la actualización de la estructura de *Z-Order* debido a que el motor gráfico *OpenGL* genera un hilo de procesamiento para subir los bloques a la textura atlas y así no afectar el tiempo de *rendering*. Al utilizar la instrucción *glFlush*(), se pudo detallar que los tiempos de rendering son más elevados debido a que el motor gráfico espera a que los bloques se desplacen a la textura atlas para generar un cuadro de imagen. Por último, La estructura unidimensional de *Z-Order* es eficiente debido a que se guardan los apuntadores de los bloques en vez de una copia del mismo, y como el cálculo de la posición es en O(1), los bloques son insertados de forma muy rápida. Se puede ver que el tiempo para reacomodar los bloques en la estructura de datos *Z-Order* no genera mayor costo de tiempo.

4.6 Resultados cualitativos

Al compactar y organizar los bloques de manera eficaz y eficiente, el algoritmo implementado permite que se inserte una mayor cantidad de bloques de los mejores niveles

de detalle tratando siempre de no comprometer mucho la perdida de datos, con el fin de tener la mejor calidad de imagen posible con la memoria disponible. En la **Figura 4.4** y **Figura 4.5** se puede observar como varía la imagen al tener diferentes tamaños de memoria de textura atlas, generando mayores resoluciones con el atlas de 512x512x512 de tamaño porque se tiene mayor espacio para realizar el proceso de refinamiento .

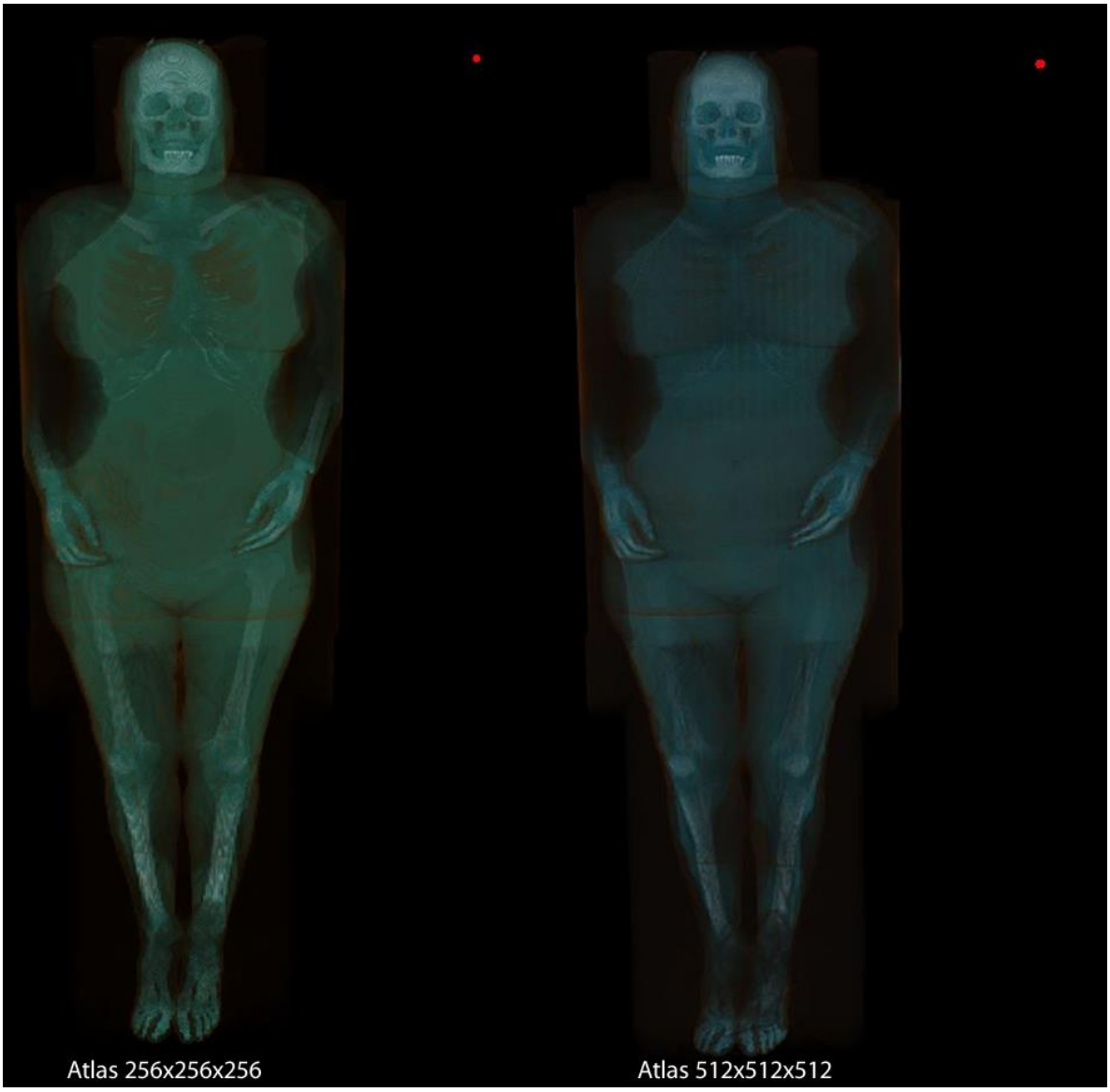


Figura 4.4: Muestra del volumen 1, con dos tamaños de atlas. En la figura se puede apreciar que el volumen con el atlas de 512x512x512 tiene una mejor resolución que el volumen con el atlas de 256x256x256.

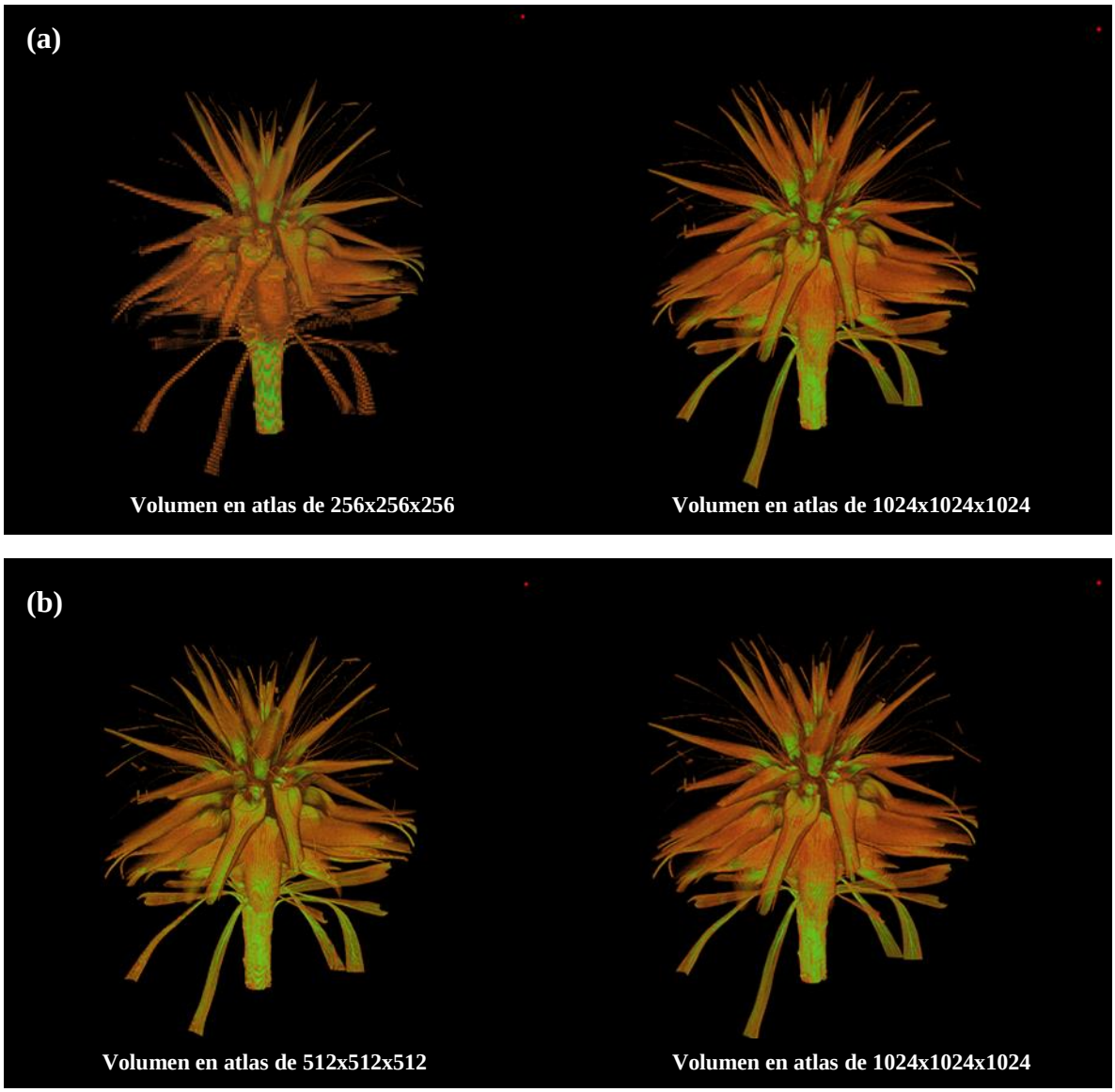


Figura 4.5: Muestra del volumen utilizando 3 tamaños de atlas diferentes. (a) Atlas de dimensiones 256x256x256. (b) Atlas de dimensiones 512x512x512 y atlas de dimensiones 1024x1024x1024. El punto rojo de la imagen representa el punto de interés.

En la **Figura 4.6** se tienen dos tamaños de bloques distintos para la generación de los niveles de detalles, de 16^3 y de 32^3 respectivamente. Se tiene que el volumen generado a 32^3 genera artefactos visuales tales como huecos en el volumen o picos en las fronteras del mismo. Se generan estos artefactos debido a que los bloques de mayor tamaño producen más perdida de datos cuando se crean los niveles de detalles menores.

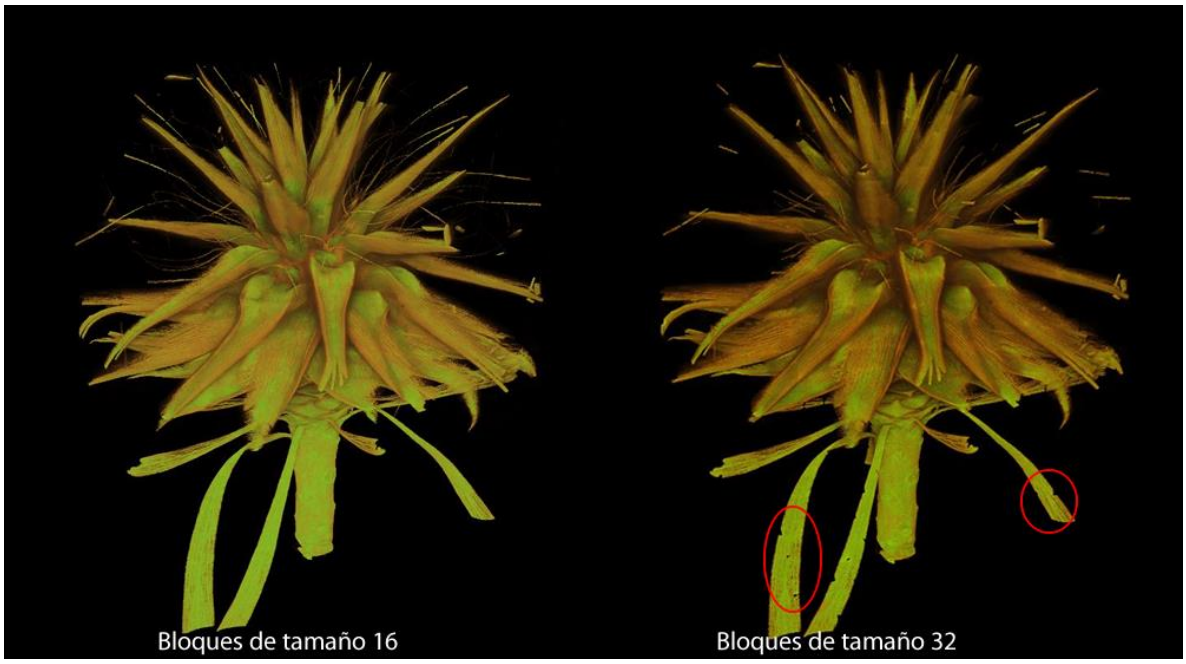


Figura 4.6: Muestra del volumen utilizando 2 tamaños de bloques diferentes. La figura de la izquierda tiene bloques de tamaño 16. La figura de la derecha tiene bloques de tamaño 32. Se puede observar como el volumen con bloques de 32 posee mayor cantidad de artefactos visuales.

Con respecto a los cambios de nivel de detalle, en la **Figura 4.7** se pueden visualizar los cambios de niveles de detalles por cada tamaño de bloques utilizados. En el tamaño de 32^3 los cambios de nivel de detalles son más toscos por el aumento de tamaño de nivel de detalle y la cantidad de bloques refinados al mayor nivel son menores. En el de 16^3 se puede ver un cambio más suavizado, porque el bloque más fino es de menor tamaño y se generan menos niveles de detalles.

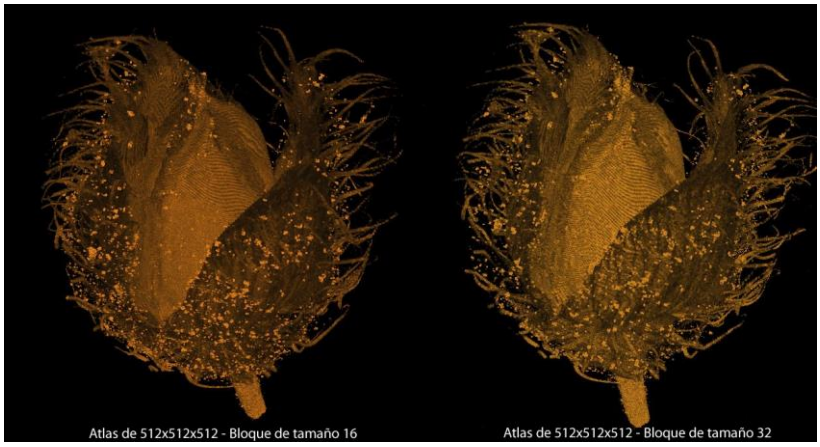


Figura 4.7: Volumen de fruta haitiana almacenada en un atlas de 128MB con dos tamaños de bloques, 16 y 32 respectivamente.

En las vistas de los resultados visuales se pueden distinguir que los volúmenes generan un pixelado entre cada una de las muestras. Esto se debe a que no se genera interpolación de muestras debido a que pueden generar artefactos entre frontera de bloques (ver **Figura 4.8**). Para solucionar esto, se debe realizar una interpolación entre bloques vecinos [**16**] para eliminar estos artefactos.

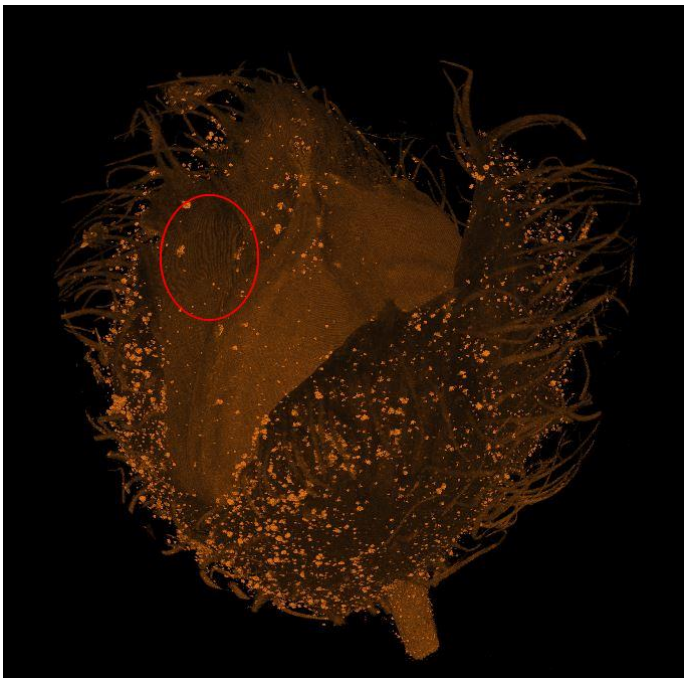
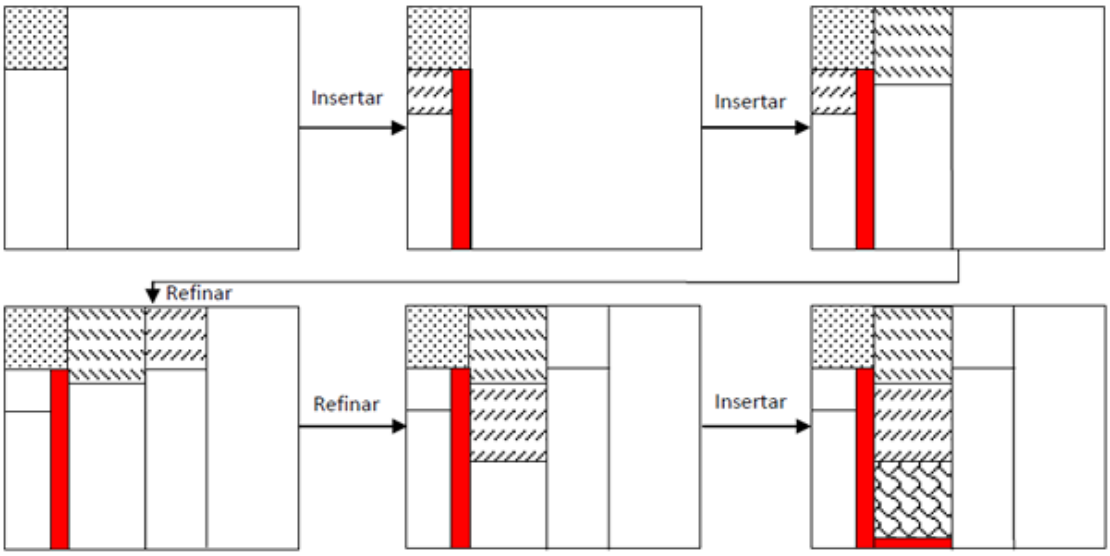


Figura 4.8: Muestra del volumen 3 en un atlas de 1024x1024x1024. Se puede observar que aunque el volumen tiene buena resolución, posee artefactos visuales por la falta de interpolación entre los bloques.

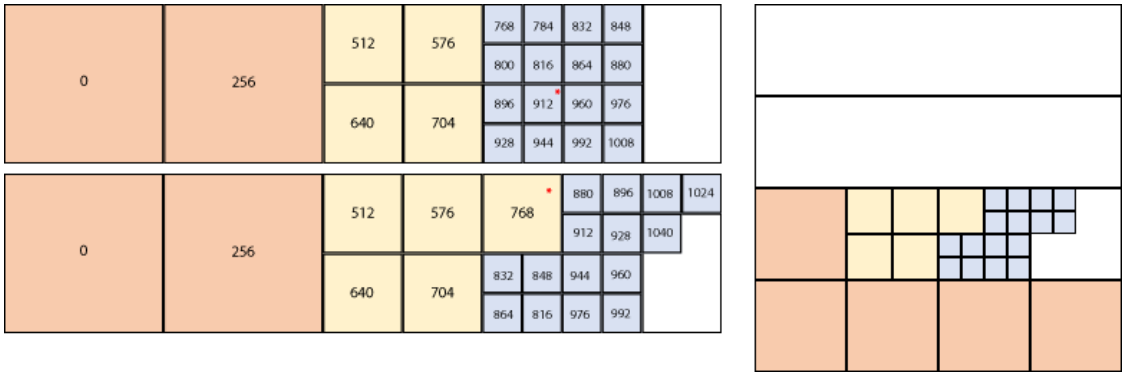
4.7 Comparación con trabajos previos

Para las comparaciones utilizamos el trabajo realizado por K. López [**28**]. Como se mencionó anteriormente, López utilizó una jerarquía basada en bloques usando un píxel de holgura entre las fronteras de cada bloque por lo que necesitaba 27% de memoria adicional (ver **Sección 2.10.1.2**), necesaria para insertar más bloques. Para la inserción de bloques en la textura atlas se particiona la textura según el tamaño del bloque que era insertado, como consecuencia, si se necesitaba insertar un bloque de mayor tamaño que el insertado anteriormente no era posible. Este particionamiento genera mucha fragmentación de

memoria, por lo que se debe pasar por un proceso de desfragmentación para borrar e insertar todos los bloques ordenados en forma decreciente. En este trabajo no se usó el píxel de holgura y la memoria restante se usó para insertar mayor cantidad de bloques. Para para la textura atlas se usó el algoritmo de *3D Z-Order Strip*, donde los bloques se reordenan en la textura atlas sin la necesidad de borrar todo y volver a insertar. Este algoritmo permite que los bloques se procesen en menos tiempo y haya mayor cantidad de bloques para un mejor despliegue del volumen.



(a)



(b)

Figura 4.9: Comparación entre el proceso de refinado de la textura atlas en el trabajo implementado por K. López (**a**) y el proceso de refinado de la textura atlas en este trabajo especial de grado (**b**).

Conclusiones y Trabajos Futuros

En este trabajo especial de grado se desarrolló una aplicación para la visualización de volúmenes multi-resolución usando una jerarquía basada en bloques con diversos niveles de detalle aplicando la técnica de *Ray Casting* en GPU de una pasada, utilizando una textura atlas para la inserción de bloques de manera eficiente.

La textura atlas es una estructura utilizada para ordenar otras texturas más pequeñas en un espacio limitado. El tamaño de los bloques en la textura atlas afecta considerablemente el tiempo de procesamiento, ya que mientras el tamaño de los bloques es mayor implica una menor cantidad de los mismos. Al tener menos bloques el proceso de refinamiento es más rápido, por el contrario, cuando se tienen bloques más pequeños el tiempo de refinamiento aumenta debido a que la cantidad de bloques es mayor. En conclusión, se puede decir que el tiempo de refinamiento es proporcional a la cantidad de bloques y a su vez la cantidad de bloques es inversamente proporcional al tamaño del bloque.

El algoritmo implementado, *3D Z-Order Strip*, permite hacer la inserción de bloques de forma eficiente, ya que su orden de complejidad es O(1), también, no requiere de un proceso de desfragmentación de memoria, ya que el algoritmo mantiene un orden de los bloques y los espacios disponibles dentro de la textura atlas. Como consecuencia, se tiene un control total sobre la posición de los bloques, y de esta manera se puede aprovechar el máximo espacio posible de la textura atlas para insertar la mayor cantidad de bloques posible según el punto de interés indicado por el usuario.

Muchos autores han propuestos diversas técnicas para el despliegue de volúmenes multi-resolución, ya sea para mejorar los artefactos visuales o para optimizar los tiempos de respuesta. A continuación se presentan algunas propuestas para mejorar el sistema de despliegue realizado en este trabajo especial de grado:

- Realizar un algoritmo de interpolación entre bloques para mejorar la calidad visual de la imagen y eliminar los artefactos visuales [**16**].

- Implementar un algoritmo para insertar más de un bloque a la vez de forma contigua en la memoria de textura, para minimizar las operaciones de subida de bloques y aprovechar el ancho de banda del GPU.
- Aplicar muestreo adaptativo en el despliegue del volumen [**61**] y aplicar clasificación pre-integrada para la mejora visual del volumen [**4**].

Referencias

[1] T. Elvins, "A Survey Of Algorithms For Volume Visualization", Advanced Scientific Visualization Laboratory, San Diego Supercomputer Center, vol. 26, 1992.

[2] E. Coto, "Volume Rendering", Presentación de Power Point, Centro de Computación Gráfica, Universidad Central de Venezuela, Facultad de Ciencias, Caracas, Venezuela, 2010.

[3] J.L. Bernadas Saragoza, "Tetraedrización De Intervalos De Volumen Mediante Modificación De Cubos Marchantes", Trabajo Especial de Grado, Biblioteca Alonso Gamero, Universidad Central de Venezuela, Facultad de Ciencias, 2009.

[4] R. Carmona, "Visualización Multi-Resolución de Volúmenes de Gran Tamaño", Tesis Doctoral, Biblioteca Alonso Gamero, Universidad Central de Venezuela, Facultad de Ciencias, Caracas, Venezuela, 2008.

[5] K. Engel, M. Kraus, T. Ertl, "High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Píxel Shading", Visualization and Interactive Systems Group, University of Stuttgart, Germany, 2001.

[6] J. Kruger, R. Westermann, "Acceleration Techniques for GPU-based Volume Rendering", in Computer Graphics and Visualization Group, Technical University Munich, 2003.

[7] M. Levoy, "Display Of Surfaces From Volume Data", Computer Science Department, University of North Carolina, 1988.

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

[9] P. Lacroute, M. Levoy, "Fast Volume Rendering Using Shear-Warp Factorization of the Viewing Transformation", Proc. SIGGRAPH '94, pp. 451-458, Orlando, Florida, 1994.

[10] The National Library of Medicine's. "Visible Human Project®".1987. [En Línea]. Disponible en: http://www.nlm.nih.gov/research/visible/visible_human.html

[11] A. A. Mirin et al., "Very High Resolution Simulation of Compressible Turbulence on the IBM-SP System", Proc. of the 1999 ACM/IEEE conference on Supercomputing, artículo No.70, 1999.

[12] P. Bhaniramka, Y. Demange, "OpenGL Volumizer: A Toolkit for High Quality Volume Rendering of Large Data sets", Proc. IEEE Symposium on Volume Visualization and Graphics, pp. 45-54, 2002.

[13] E. LaMar, M. Duchaineau, B. Hamann, K. Joy, "Multi-resolution Techniques for Interactive Texture-based Rendering of Arbitrarily Oriented Cutting Planes", en Data Visualization 2000, The Joint Eurographics and IEEE TVCG conference on Visualization, pp. 105-114, 2000,.

[14] E. LaMar, M. Duchaineau, B. Hamann, K. Joy, "Multi-resolution Techniques for Interactive Texture-based Volume Visualization", en Visualization '99, California-USA, pp. 355-361, 1999.

[15] I. Boada, I.Navazo, R. Sopigno, "A 3D Texture-Based Octree Volume Visualization Algorithm", en The Visual Computer, vol. 17, pp. 185-197, 2000.

[16] P. Ljung, C. Lundström, A. Ynnerman, "Multi-resolution Interblock Interpolation in Direct Volume Rendering", en Eurographics/IEEE-VGTC Symposium on Visualization, pp. 259-266, 2006.

[17] X. Li, H. Shen, "Time-Critical Multiresolution Volume Rendering Using 3D Texture Mapping Hardware", en IEEE Volume Visualization and Graphics Symposium '02, Boston-USA, 2002.

[18] B. Chamberlain, T. DeRose, D. Lischinski, David Salesin, and John Snyder, "Fast rendering of complex environments using a spatial hierarchy", pp. 132-141, 1996,.

[19] E. LaMar, M. Duchaineau, B. Hamann, K. Joy, "Multi-resolution Techniques for Interactive Texture-based Volume Visualization", en Visualization '99, California-USA, pp. 355-361, 1999.

[20] I. Boada, I. Navazo, R. Sopigno, "A 3D Texture-Based Octree Volume Visualization Algorithm", en The Visual Computer, vol. 17, pp. 185-197, 2000.

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

[22] J. Plate, M. Tirsana, R. Carmona, B. Froehlich, "Octreemizer: A Hierarchical Approach for Interactive Roaming Through Very Large Volumes", en Joint Eurographics - IEEE TCVG Symposium on Visualization '02, pp. 53-60, 2002.

[23] D. Laur, P. Hanrahan, "Hierarchical Splatting: A Progressive Refinement Algorithm For Volume Rendering", en SIGGRAPH Comput. Graph., pp. 285-288, 1991.

[24] I. Boada, I. Navazo, R. Sopigno, "Multi-resolution Volume Visualization With A Texture-Based Octree", en The Visual Computer, pp. 185-197, 2001.

[25] J. GaoGao, C. Wang, and Han-Wei Shen, "Parallel Multi-resolution Volume Rendering of Large Data Sets with Error-Guided Load Balancing", en Parallel Comput, pp. 185-204., 2005.

[26] P. Ljung, C. Lundstrom, A. Ynnerman, K. Museth, "Transfer Function Based Adaptive Decompression for Volume Rendering of Large Medical Data Sets," en IEEE Symposium on Volume Visualization and Graphics, pp. 25-32, 2004.

[27] C. Wang, A. Garcia, H. WeiShen, "Interactive Level-of-Detail Selection Using Image-Based Quality Metric For Large Volume Visualization," IEEE Transactions on Visualization.

[28] K. López, "Despliegue de Volúmenes Multi-resolución Basado en Ray Casting de una Pasada", Universidad Central de Venezuela, Tesis de Pregrado, 2011.

[29] Gabriel Rodríguez, "Despliegue Multi-resolución de Volúmenes con Reducción de Artefactos entre Sub-Volúmenes Adyacentes", Universidad Central de Venezuela, Tesis de Pregrado, 2008.

[30] Kurt Zimmermann, R. Westermann, T. Ertl, C. Hansen, M. Weiler, "Level-of-Detail Volume Rendering via 3D Textures", en Volume Visualization and Graphics, IEEE Symposium on, Los Alamitos, CA, USA, pp. 7-13, 2000.

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

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

[33] OpenMP®. The OpenMP® API Specification For Parallel Programming. 2010. [En Línea]. Disponible en: http://openmp.org/

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

[35] NVidiaCorporation. CUDA. 2012. [En Línea]. Disponible en: http://www.nvidia.com/object/cuda_home_new.html

[36] NVidia. NVIDIA ® CUDA™ Architecture. 2010 [En Línea]. Disponible en: http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf

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

[38] P. Ljung, C. Lundstrom, A. Ynnerman, K. Museth, "Transfer Function Based Adaptive Decompression for Volume Rendering of Large Medical Data Sets", en IEEE Symposium on Volume Visualization and Graphics, pp. 25-32, 2004.

[39] C. Lux, B. Frohlich, "GPU-Based Ray Casting of Multiple Multi-Resolution Volume Datasets," in ISVC '09 Proceedings of the 5th International Symposium on Advances in Visual Computing: Part II, pp. 104-116, 2009.

[40] P. B. Galvin, G. Gagne, A. Silberschats, "Fundamentos de Sistemas Operativos", Septima Edición, Editorial Mc Graw Hill, 2006.

[41] W. Stallings, "Sistemas Operativos", Segunda Edición, Editorial Pretice Hall, Inc., 2000

[42] J. Carretero Pérez, P. M. Anasagasti, Sistemas Operativos "Una visión aplicada", Editorial Mc Graw Hill, 2001.

[43] J. Martín Sáez, M. Hernández Huerta, J. González López, "Unidades de Acceso a Memoria de Texturas Estructura y Aplicaciones Dentro De Las UAMT", Universidad Rey Juan Carlos.

[44] D. Shreiner, "OpenGL Programing Guide Séptima Edición", Editorial Addison-Wesley, 2010.

[45] J. Martínez Bayona "Space-Optimized Texture Atlases" Universitat Politècnica De Catalunya Master Thesis, 2009.

[46] E. G. Corman, Jr., M. R. Garey, D. S. Johnson, R. E. Tarjan. "Performance Bounds For Level-Oriented Two-Dimensional Packing Algorithms". SIAM Journal on Computing, 9:801-826, 1980.

[47] F. K. R. Chung, M. R. Garey, D. S. Johnson. "On Packing Two-Dimensional Bins. SIAM Journal of Algebraic and Discrete Methods", 3:66-76, 1982.

[48] J. O. Berkey, P. Y. Wang. "Two Dimensional Finite Bin Packing Algorithms. Journal Of The Operational Research Society", 38:423-429, 1987.

[49] A. Lodi, S. Martello, D. Vigo. "Heuristic And Metaheuristic Approaches For A Class Of Two-Dimensional Bin Packing Problems". INFORMS Journal on Computing, 11:345-357, 1999.

[50] B. S. Baker, E. G. Co®man, Jr., and R. L. Rivest. "Orthogonal packing in two dimensions". SIAM Journal on Computing, 9:846-855, 1980.

[51] J. Gordon, "Binary Tree Bin Packing Algorithm", 2011, [En Línea]. Disponible en: http://codeincomplete.com/posts/2011/5/7/bin_packing/

[52] K. Dowsland, "Some Experiments With Simulated Annealing Techniques For Packing Problems", European Journal of Operational Research, 68:389–399. 1993

[53] S. Martello, P. Toth, "Lower Bounds and Reduction Procedures for the Bin Packing Problem. Discrete Applied Mathematics", 28(1), 59–70. 1990

[54] S. Jacobs, "On Genetic Algorithms For The Packing Of Polygons, European Journal Of Operational Research", 88:165–181. 1996.

[55] L. Cruz-Reyes, M. Quiroz C., A. C. F. Alvim, H. J. Fraire Huacuja, C. Gómez S. , J. Torres-Jiménez, "Heurísticas De Agrupación Híbridas Eficientes Para El Problema De Empacado De Objetos En Contenedores", Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional, México Computación y Sistemas Vol. 16 No.3, pp 349-360, 2012.

[56] F. Torres, D. Mauricio, L. Rivera, "Un modelo de Compactación de Objetos Irregulares UPG-FISI", Universidad Nacional Mayor de San Marcos (UNMSM) LCMAT-CCT, Universidade Estadual do Norte Fluminense (UENF) 2000.

[57] J. Sol Roo, J. P. D'Amato, E. Ferrante, "Optimización de Atlas de Textura Utilizando Compactación de Polígonos por Simulación Física", Facultad de Ciencias Exactas, Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Argentina, Consejo Nacional de Investigaciones Científicas y Técnicas CONICET, Bs. As., Argentina École Centrale de Paris, Paris, Francia.

[58] K. E. Kendall, J. E. Kendall, "Análisis Y Diseño De Sistemas", Editorial Pearson, 2005.

[59] Gestión Operativa De La Calidad Del Software "Extreme Programing" [En Línea]. Disponible En: http://maestria-modulo7.blogspot.com/2012/04/procesos-de-desarrollo-ligeros-vs.html

[60] AntTweakBar Library [En Línea]. Disponible En: http://anttweakbar.sourceforge.net/doc/

[61] R. Carmona, B. Froehlich, "Error-Controlled Real-Time Cut Updates For Multi-Resolution Volume Rendering", Universidad Central de Venezuela, Facultad de Ciencias, Centro de Computación Gráfica, Caracas, Venezuela, 2011.

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

[63] MPI [En Línea]. Disponible En: http://www.mcs.anl.gov/research/projects/mpi/

[64] OpenMP [En Línea]. Disponible En: http://openmp.org/wp/

[65] OpenCL [En Línea]. Disponible En: http://www.khronos.org/opencl/

[66] CUDA [En Línea]. Disponible En:

http://la.nvidia.com/object/cuda_home_new_la.html

[67] OpenGL [En Línea]. Disponible En: http://www.khronos.org/opengl

[68] E.Purcel, D. Varberg y E. Rigdon "Calculo Octava Edición", Pearson Educación, p. 234, 2001.

[69] T. Crainic, G. Perboli, R. Tadei, "Extreme Point-Based Heuristic for Three Dimensional Bin Packing", Université de Montréal, Quebéc, Canada, 2007.

[70] Z-order curve [En Línea]. Disponible En: http://en.wikipedia.org/wiki/Z-order_curve

[71] Schwaber, K., & Sutherland, J. (2010). Scrum guide. [En Línea]. Disponible En: http://www.scrumguides.org/

[72] Cohn, M. User stories applied: For agile software development. Addison Wesley. 2004.