Departamento de Ingeniería Informática
y Ciencias de la Computación
Universidad de Concepción

Aplicación de DACs en reemplazo del SVDAG para el almacenamiento de datos de Voxeles en la renderización en tiempo real

por
Catalina Isidora Jiménez Reyes

Memoria presentada para la obtención del título de
Ingeniero Civil Informático

Patrocinante: José Fuentes Sepúlveda
Comisión: Cecilia Hernández y Lilian Salinas

Concepción, 9 de septiembre de 2024

Agradecimientos

Esta memoria de título ha sido desarrollada en son de demostrarme que puedo aprender cosas nuevas desde cero y aplicarlas a un campo de estudio del cual no sabía hasta un año atrás. Pero, aun que me haya demostrado a mi misma lo que puedo hacer, esto no fue sin la ayuda de personas que aprecio, por ende quiero utilizar esta página para agradecer a quienes me apoyaron a lo largo de todo el proceso y más aún a quienes me extendieron una mano y compartieron palabras de apoyo en los momentos más bajos resultando en pequeños empujes necesarios para seguir superándome, hasta llegar a mi meta.

Primero quiero agradecer a mi profesor guía José Fuentes por haber sido una fuente de apoyo y conocimiento primordial para esta memoria de título en un área de la cual no poseía conocimiento previo. Agradezco a mi pareja Álvaro Castillo por apoyarme incondicionalmente y estar presente tanto en los buenos como malos momentos. Agradezco a mis amigos de infancia, Constanza Hawa, Pablo Stange y Daniela Martinez por siempre escuchar mis anecdotas,problemas y dudas brindandome su apoyo al necesitarlo. Agradezco a mi familia por su cariño incondicional, especialmente a mi hermano y hermana que siempre fueron honestos conmigo brindándome su conocimiento con respecto a pasar por etapas similares a la mía.
Mis hermanos fueron el principal motivante para demostrarme las cosas que puedo lograr y siempre les estaré agradecida: a mi hermana por siempre aconsejarme cuando lo he pedido y a mi hermano por aconsejarme aunque no se lo pida. No puedo pedir mejores roles modelo a seguir que ellos dos y especialmente mi hermana por compartir los mismos desafíos que yo actualmente, en cierta manera, mostrándome los caminos que puedo seguir. Gracias Francisca.

Resumen

En el área de la renderización en tiempo real, generalmente, las estructuras de datos que almacenan voxeles deben ser navegadas en GPU, la cual poca memoria. Esto último obliga a aprovechar lo que más se pueda el espacio. Debido a esto la estructura de datos mayormente utilizada en el área es el *SVDAG* (Sparse Voxel Directed Acyclic Graph) [6], ya que supone un mayor ahorro en espacio que estructuras anteriormente utilizadas. En otra área de las Ciencias de la Computación, existen las estructuras de datos compactas, las que tienen como objetivo principal el ahorro y buen manejo de la memoria, permitiendo operaciones eficientes a la vez que ahorran espacio. Hasta donde sabemos, no se ha usado una estructura de datos compactas en GPU, hasta esta memoria de título. Proponemos el uso de la estructura de datos compacta *DAC* (Directly Addressable Codes) para reemplazar a la representación SVDAG con el objetivo de que ahorre mayor espacio. Gracias a que el DAC posee acceso aleatorio, lo convierte en un reemplazo ideal para el SVDAG brindando un gran aporte en ambas áreas de estudio.

Índice

Índice de figuras

Índice de cuadros

1. Introducción

En el área de los videojuegos se destaca el uso de la renderización en tiempo real como un proceso que tiene por finalidad generar contenido gráfico en pantalla. La renderización es conocida por el resultado que generan los cálculos de la computadora para crear imágenes cada segundo, de manera que lo que se percibe en la pantalla sea fluido al mover la cámara. Para la renderización se utilizan varios algoritmos pero en la presente memoria de título se hará uso de uno llamado *raytracing*. El raytracing, también conocido como *trazado de rayos*, debido a cómo funciona su algoritmo, consiste en trazar rayos que son "disparados"desde un punto de origen, donde está ubicada la cámara, desde donde el usuario es capaz de visualizar un modelo. Estos rayos siguen su camino hasta que intersecta un objeto de la escena o hasta que no encuentra ningún objeto. Un ejemplo de esto se puede ver en la Figura 1.

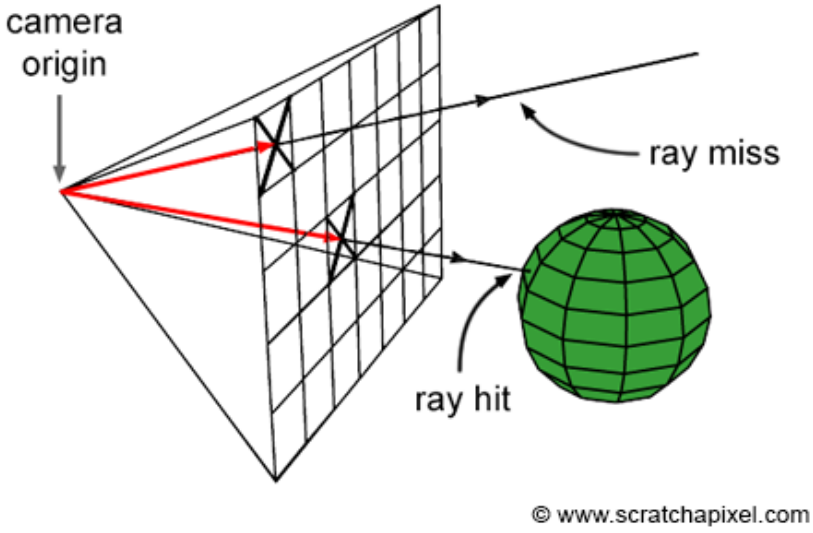


Figura 1: Ejemplo de raytracing [1]

En el contexto 3D, la unidad mínima que compone a un objeto tridimensional es un voxel y aquellos necesitan de estructuras de datos tridimensionales para almacenarlos. La estructura más utilizada es el *octree*, siendo el *Sparse Voxel Octree* (SVO) una de sus implementaciones usadas en GPU, la cual

busca optimizar la búsqueda de los datos dentro del árbol. Una mejor al SVO, de especial interés en esta memoria de título, es el *Sparse Voxel Directed Acyclic Graph* (SVDAG) que, por lo indicado por su nombre, convierte el árbol subyacente en un grafo dirigido sin ciclos.

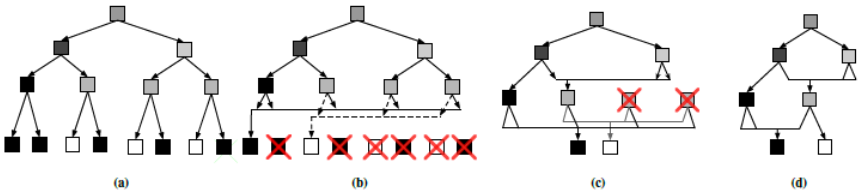


Figura 2: Proceso ilustrativo sobre como un octree pasa a ser un SVDAG [6]

La idea principal de cómo se lleva a cabo la conversión de SVO a SVDAG es que, aquellas hojas que sean idénticas dentro del árbol sean combinadas y luego se actualicen los punteros hacia sus nodos padres. Una representación de lo mencionado se puede apreciar en la Figura 2. El proceso de cómo estas hojas se juntan y eliminan nodos se explicará más adelante. Por otro lado, además de las estructuras de datos clásicas, hay una familia de ellas llamada *estructuras de datos compactas*, la cual es un área de investigación que busca representar datos en forma compacta, permitiendo realizar consultas sobre estos datos manteniendo un espacio reducido. Dentro de ellas, un ejemplo relevante es el k^3-*tree* [7], el cual corresponde a una versión compacta del octree, y vectores compactos, los cuales representan vectores de valores usando el espacio reducido, permitiendo a la vez acceso directo a sus elementos [10].

Esta memoria busca proveer evidencia de que las estructuras de datos compactas pueden juegan un papel importante en el procesamiento de información gráfica, permitiendo representar modelos 3D en espacio reducido. En particular, esta memoria de título se centra en el uso de una estructura de datos compacta para vectores de enteros, llamada *Directly Addressable Codes* (DAC), con la meta de usarla para almacenar el SVDAG. La propuesta del uso de DAC tiene como objetivo principal ahorrar significativamente más espacio que el SVDAG, optimizando aún más los recursos de memoria sin comprometer la calidad y velocidad de la renderización en tiempo real. El principal desafío radica en que no existen implementaciones para GPU del DAC, si no sólo existen para CPU.

1.1. Hipótesis

La siguiente memoria de título busca validar la hipótesis: "La estructura de datos compacta DAC permite un ahorro de espacio significativo en comparación con la estructura de datos SVDAG para la renderización de objetos en GPU". Para comprobar esta hipótesis, se pretenden comparar ambas estructuras DACs y SVDAG con respecto a su impacto sobre el ahorro de espacio y promedio de frames por segundo (FPS) por modelo.

1.2. Objetivos generales

El propósito de esta memoria de título es la comparación empírica de DACs y SVDAG, ambos construidos en CPU y recorridos en GPU a la hora de la renderización. Se busca evaluar el desempeño de ambas estructuras sobre el impacto de la renderización en tiempo real con la finalidad de demostrar que las estructuras de datos compactas demuestran una mejora significativa, en términos del uso de espacio, que la estructura de datos principal SVDAG.

1.3. Objetivos específicos

1. Realizar una revisión de la literatura en el área de estructuras de datos compactas para elegir y proponer una estructura en reemplazo del SVDAG.

2. Seleccionar y modificar un código que implemente el SVDAG para reemplazar su implementación por el DACs

3. Comparar ambas estructuras y analizar su desempeño a base de tests.

4. Proponer mejoras y optimizaciones para trabajo futuro.

2. Conceptos preliminares

Para la claridad de la presente memoria de título, se explicarán los siguientes conceptos clave:

2.1. Directly Addresable Codes (DAC)

Para una mejor comprensión de la estructura de datos DAC es necesario explicar primero lo que son las *estructuras de datos compactas*. En términos generales, la compresión busca convertir un string de bits que representa data a un string de bits más corto, de manera que la transmisión, almacenamiento o procesamiento de esa data pida menos requerimientos de espacio [2]. No obstante, en general, para poder utilizar los datos comprimidos, es necesario pasar por una etapa de descompresión, lo que puede ser costoso en términos de tiempo, especialmente cuando sólo necesitamos una pequeña porción de la data original. Por su lado, las estructuras de datos compactas tienen el objetivo de representar data usando la menor cantidad de espacio posible y, al mismo tiempo, soportar operaciones eficientes sobre dicha representación reducida en espacio. Aunque, en general, estas estructuras de datos requieren de algoritmos más complejos para su uso, son estructuras que tienden a mejorar el rendimiento general debido a que, con la reducción de espacio obtenido, operan en niveles rápidos en la jerarquía de memoria. Dentro de las estructuras de datos compactas más básicas tenemos los *bitmaps*, secuencias de símbolos definidos sobre un alfabeto binario $\{0, 1\}$. Dependiendo de la proporción de 0s y 1s, existen bitmaps más adecuados que otros [10]. Una de las operaciones más utilizadas sobre un bitmap es la operación $rank(B, i)$, la cual retorna en tiempo constante la cantidad de símbolos 1 en el bitmap desde su inicio hasta la posición i.

Los *Directly Addresable Codes* (DAC) son utilizados para codificar secuencias de enteros y permitir acceso aleatorio a ellos. Se basa en la reubicación de fragmentos del código binario que representa cada uno de los enteros de la secuencia en distintos niveles. La Figura 3 muestra un esquema de su funcionamiento. Asumiremos una secuencia C de enteros, donde cada entero es separado en fragmentos o trozos $C_{i,r}$. Estos trozos irán a 3 canales denotados como C_1, C_2 y C_3. C_1 contiene los trozos que estén más a la derecha, $C_{i,1}$; C_2 contiene los trozos siguientes a la izquierda, $C_{i,2}$, y así sigue con C_3. Como se puede apreciar, cada canal es separado en dos partes A y B: A corresponde a los bits de los trozos de C mencionados anteriormente, y B es un bitmap y se lee de la siguiente manera: un 1 en la posición $B[i]$ indica que hay más trozos en los canales siguientes del número representado en la posición $A[i]$, mientras que un 0 indica lo contrario.

Para extraer el valor $C[i]$ de la secuencia de enteros codificados en el DAC

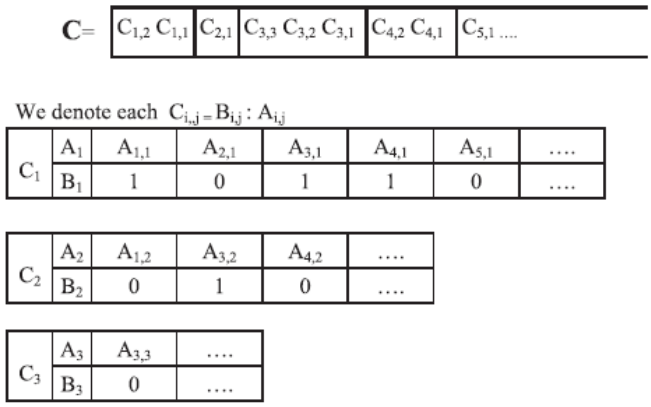


Figura 3: Imagen ilustrativa de la reubicación de códigos usando DAC [2]

se aplica el siguiente procedimiento: Con $i_1 = i$, se revisa el primer trozo $C_{i,1} = A_1[i_1]$ y se verifica $B_1[i_1]$. Si $B_1[i_1] = 0$, terminamos la búsqueda. En el caso contrario, se define $i_2 = rank(B_1, i_1)$, lo que nos dará la posición correcta del segundo trozo en C_2. Repetimos el proceso y extraemos $C_{i,2} = A_2[i_2]$. Si $B_2[i_2] = 0$ terminamos y retornamos el valor $A_1[i_1] + A_2[i_2] * 2^b$, donde b es el tamaño de los trozos. En caso contario, repetimos el procedimiento.

C_1	A_1	00	10	10	01	01	01	11
	B_1	1	0	1	0	1	1	1
C_2	A_2	01	10	01	01	00		
	B_2	0	0	1	0	1		
C_3	A_3	01	01					
	B_3	0	0					

Figura 4: Ejemplo utilización de DAC [2]

Los DACs poseen la particularidad de que el tamaño de los trozos, mencionados anteriormente, puede variar de canal a canal y no tienen que ser de un largo fijo. Gracias a esta flexibilidad de los DACs se puede obtener menor espacio.

Con respecto a la complejidad espacial de esta estructura, siendo N la suma del largo en bits de cada número de la secuencia, y agrupando b bits en cada canal, el tamaño total es de $O(\frac{N \log \log N}{b \log N})$ bits. Respecto a la comple-

jidad temporal, para acceder a cualquier valor de la secuencia necesitamos un tiempo de $O(\frac{\log M}{b})$ en el peor caso, siendo M el valor máximo de la secuencia (para ver los detalles de la explicación de esta complejidad, revisar la siguiente thesis [2]).

2.1.1. ¿Por qué se eligió el DAC?

Existen varias alternativas que entran en la categoría de vectores compactos [10], pero en general, tienen el problema de garantizar acceso aleatorio a cualquier valor del vector. La solución que ofrecen es un trade-off entre espacio y tiempo: Para proveer un menor tiempo de acceso, aumentan el tamaño de la representación, y viceversa. El DAC es una respuesta a este trade-off, ofreciendo acceso aleatorio en un tiempo reducido. Esto último lo hace ideal como reemplazo del SVDAG, ya que el acceso directo a nodos del árbol es crucial a la hora de aplicar el algoritmo raytracing. También, ya que DAC usa espacio proporcional a la magnitud de los valores almacenados en el vector de enteros, lo hace ideal para almacenar vectores donde la frecuencia de aparición de enteros pequeños es más frecuente que enteros más grandes, situación que tiende a ocurrir en el SVDAG.

2.2. GPU

GPU es la unidad de procesamiento gráfico de los computadores actuales y están formadas por pequeños núcleos de procesamiento. Aquellos núcleos trabajan junto a la unidad de procesamiento central. Las GPU, o unidades de procesamiento gráfico, son componentes clave en computadoras diseñadas para manejar tareas relacionadas con gráficos y renderización. A diferencia de las CPU, que están optimizadas para realizar una amplia variedad de tareas, las GPU están altamente especializadas en el procesamiento de gráficos y pueden manejar grandes cantidades de datos de manera simultánea, lo que las hace ideales para aplicaciones intensivas en gráficos como juegos y renderización.

2.2.1. OpenGL

Cuando se trata de escribir un código que trabaje en la GPU, una técnica común es utilizar OpenGL junto con shaders. OpenGL es una API de gráfi-

cos que proporciona una interfaz para interactuar con la GPU y renderizar gráficos en 2D y 3D [4]. Un shader es un pequeño programa de software utilizado en gráficos por computadora para definir la apariencia de objetos y superficies en una escena renderizada. Dentro de los shaders disponibles, los *compute shaders* son un tipo específico de shader utilizado en OpenGL que permite realizar cálculos de propósito general en la GPU. Son independientes del pipeline de gráficos (ver Figura 5), lo que los hace útiles para una variedad de aplicaciones además de la renderización.

2.2.2. Compute Shader

Para tener un mejor entendimiento de la comunicación entre CPU y GPU se debe entender el rol de los shaders en el computador, especialmente en el código. Como se mencionó en la sección anterior, los compute shader son un tipo de shader que no obedece la pipeline de gráficos, ya que actúa de manera independiente. Cada Shader conocido en la pipeline de gráficos cumple con distintas tareas.

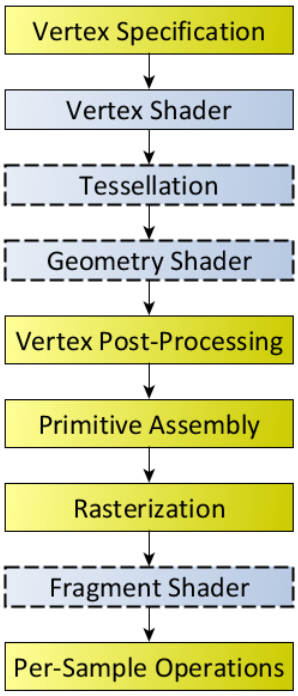


Figura 5: Ejemplo del pipeline de renderización [5]

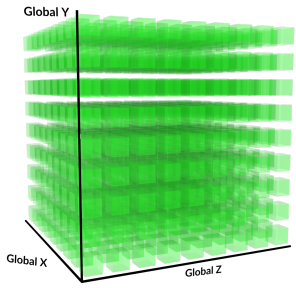


Figura 6: work groups globales

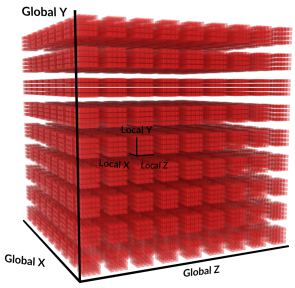


Figura 7: espacio local

Una de las razones por la cual el compute shader es independiente de la pipeline de gráficos es para brindar mayor flexibilidad a las tareas que se les puede asignar a una GPU. Cada vez se vuelve más común utilizar las tarjetas gráficas para ejecutar tareas de cálculo debido a su velocidad. A la vez, este aumento de velocidad trae consigo un desafío debido a que las GPU no son un procesador serial sino uno por streams. Un procesador de streams usa funciones (kernel) para ejecutar sets de inputs (stream) y así producir outputs como por ejemplo la imagen final vista en la pantalla del computador. Las GPU ejecutan elementos en paralelo y cada una de esas ejecuciones es independientes de la otra. Debido a estos puntos, trasladar un programa de CPU a GPU tiene sus desafíos y el primero es que el programa en cuestión sea paralelizable. Esto no es un problema para el raytracing debido a que cada rayo puede ser procesado de manera independiente.

Debido a la flexibilidad que poseen, los compute shader definen su propio espacio de trabajo y no tienen inputs definidos por usuario o outputs como los otros shaders. Debido a esto, el compute shader debe ir por los datos necesarios para la ejecución de sus tareas. La manera de lograr esto es utilizando lo que se conoce como *Buffer Objects*, conocidos coloquialmente como *buffers*, los cuales son la manera principal de guardar, enviar y recibir información respecto al lenguaje de shaders de OpenGL (GLSL). Hay distintos tipos de buffers pero el utilizado en la implementación del código de la presente memoria de título es el *Shader Storage Buffer Object* (SSBO). Retomando el concepto de *definir su propio espacio*, los compute shader operan en grupos de trabajos y estos son comprendidos por el mínimo tamaño de operaciones tipo compute que pueda ejecutar el usuario. Este espacio de trabajo es definido en 3D. La Figura 6 muestra un esquema, donde cada cuadrado verde es

un grupo de trabajo, y cada grupo de trabajo contiene múltiples invocaciones del compute shader en donde su número es definido por el tamaño local del grupo de trabajo. Debido a esto también son tridimensionales, como se aprecia en la Figura 7. El orden de ejecución de los grupos de trabajo varía, por ende el programa no debe depender del orden en el cual cada grupo es procesado.

3. Revisión bibliográfica

3.1. SVDAG

El SVDAG (Sparse Voxel Directed Acyclic Graph) es una estructura de datos utilizada principalmente en el área de renderización en tiempo real y es construido a partir de un SVO (Sparse Voxel Octree). Antes de explicar el proceso de transformación de SVO a SVDAG se debe entender la base del SVO, la cual es un octree, como lleva en su nombre. El Octree fue creado con fines de representación de datos en ambientes 3D y desarrollado por Donald Meagher [9].Un octree es una estructura de datos de tipo árbol que se utiliza principalmente para dividir un espacio tridimensional en volúmenes más pequeños. Es similar a un quadtree, que se utiliza para dividir espacios bidimensionales, pero en lugar de dividir un plano en cuatro partes, un octree divide un volumen en ocho partes iguales. Como toda estructura árbol, el octree posee una raíz, la cual representa el espacio completo que utiliza tridimensionalmente, también posee un espacio que es dividido recursivamente en ocho cajas representando a los hijos, esto repitiendose hasta llegar al criterio deseado de término, y como último punto también posee hojas que son las subdivisiones más pequeñas del árbol y son aquellas que contienen los datos del árbol.

El SVO corresponde a una manera de almacenar datos de manera similar al octree, ya que es un sucesor de este, entregando buena localidad espacial. Los datos almacenados en un subárbol del SVO se almacenan en orden DFS, permitiendo que nodos del mismo subárbol queden en áreas de memoria contigua, la que incluye la topología del octree, la geometría de los voxeles y sus sombras correspondientes, en caso de existir esa información. Finalmente, el SVO se almacena como un vector de enteros, donde cada entrada encripta información de navegación en sus bits. Lo que separa un Octree de un SVO

es su uso reducido de memoria en comparación con el octree y, por ende, es más eficiente; esto es gracias a que posee una "estructura parcial", lo cual es atribuido al nombre **Sparse** Voxel Octree, y significa que el SVO solo almacena y subdivide partes del espacio en donde contiene información relevante, por ende regiones vacías no son subdivididas y así ahorra más espacio.

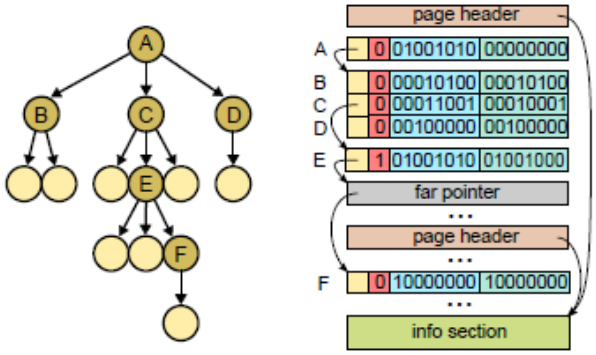


Figura 8: Izquierda: ejemplo de jerarquia de los voxeles. Derecha: ejemplo de arrays representando a los hijos, hay un array por cada voxel no hoja

Un ejemplo del SVO se puede ver en la Figura 8. Cada entrada del SVO corresponde a un nodo del árbol que representa, la cual a su vez es subdividida en rangos de bits con distintos significados. Para este trabajo nos interesarán las zonas marcadas en verde y en amarillo: la zona verde corresponde a los 8 bits menos significativos, donde se marcan con un bit en 1 los hijos que existen en el árbol, y en 0 los que no. Por su parte, la zona en amarillo corresponde a 15 bits que codifican la posición en el vector que almacena el SVO donde inicia la zona que representa a sus hijos. En el trabajo original, cada entrada del SVO consta de 64 bits, con 32 de ellos dedicados a describir el conjunto de vóxeles hijos y los otros 32 bits a otra información, como color, contorno, etc, del modelo que se está representando. Para esta memoria, se enfocará en los primeros 32 bits.

Para transformar un SVO a un SVDAG se puede usar un algoritmo bottom-up, donde cada nodo hoja es definido únicamente por su child mask (los 8 bits menos significativos), lo que permite identificar nodos hojas idénticos para luego fusionarlos y actualizar los punteros hijos a un nivel superior (ver Figura 2b). Luego, en el siguiente nivel superior se deben encontrar aque-

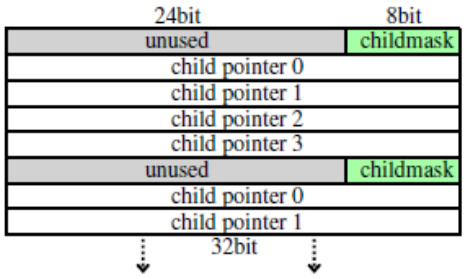


Figura 9: Ejemplo uso de memoria para los nodos de SVDAG [8]

llos nodos con child masks y punteros idénticos, lo que implica encontrar subárboles idénticos, y por ende, se pueden fusionar a nivel de sus raíces. Luego, este procedimiento se lleva a cabo repetitivamente hasta que no se encuentren más instancias que combinar. Al llegar a este punto el resultado final es el DAG más pequeño posible (Figura 2d). Este algoritmo es aplicable a SVOs de profundidades no uniformes como para DAGs parcialmente reducidos. El vector presente en la Figura 9 muestra como se ven los nodos hijos en memoria en donde hay 32 bits y 8 de ellos están en uso para la child mask, la cual codifica la existencia de punteros hijos, luego en la memoria siguiente se guardan aquellos punteros consecutivamente. La transformación de SVO a SVDAG permite que, en casos donde el SVO completo no cabe en memoria, el SVDAG sí lo haga. Esto es crucial dado que la memoria de la GPU es más limitada que la de la CPU, haciendo esencial el uso de estructuras que comprimen datos y permitan un acceso eficiente sin necesidad de descompresión.

4. Solución propuesta

Para el desarrollo de esta memoria usaremos como base la implementación provista en el paper High Resolution Sparse Voxel DAGs [6] [1], el cual es un renderizador de modelos en 3D, implementado en OpenGL, y que usa como estructura de datos subyacente el SVDAG. Como fue mencionado en capítulos anteriores, se utilizó un DAC para que reemplace al vector que almacena el SVDAG. El código final está disponible en `https://github.`

[1] Código disponible en `https://github.com/harrynull/voxel-raytracer/tree/main`

`com/Morrigan-berry/DAC-GPU-Raytracer`.

4.1. Código

A continuación se explicará la estructura del código, junto con sus algoritmos y estructuras correspondientes.

4.2. Loop principal

Hay tres clases importantes en el código: *Renderer*, *SVO* y *Window*, las cuales implementan distintas funcionalidades clave en renderización y construcción de estructuras de datos. La clase SVO se encargada de construir la estructura SVO a partir de modelos almacenados en archivos `.vox`. La clase Renderer se encarga de la comunicacion entre CPU y GPU para renderizar y comunicar la mayoría de las clases del código entre sí. Finalmente, la clase Window se encarga de llevar a cabo el loop principal y mantener abierta la ventana en donde se pueden visualizar los modelos 3D.

El código comienza con la creación de objetos de las clases Renderer y Window, creando una instancia de una ventana en donde se pueda ver la renderización. Luego, se ejecuta el loop principal, el cual se puede apreciar en el Algoritmo 1. Algo importante a destacar del algoritmo es el uso de los buffers (ver Sección 2.2.1), los cuales son la única manera de que la GPU pueda recibir información desde la CPU.

Algorithm 1 Loop principal

```
1: init Renderer
2: while Window no cierre do
3:     comienza estados del teclado
4:     actualiza Renderer
5:     renderiza Renderer
6:     swapBuffers
7: end while
```

4.2.1. Clase SVO

Esta clase se encarga de:

- Crear un SVO a partir de modelos de entrada o algunos modelos creados por defecto.
- Transformar el SVO en un SVDAG por medio de una función recursiva.
- Definir la estructura tipo *material*, la cual indica qué color posee y si es agua o no.

Cada modelo tiene un SVO construído a la hora de la renderización gracias a un constructor, el que utiliza a la función recursiva *set()*, encargada de insertar un voxel a la vez, insertando los datos del material a los nodos hoja necesarios. Luego, la función recursiva *toSVDAG()* se encarga de transformar el SVO a un SVDAG. Además del vector que representa al SVDAG, esta función también retorna un vector con la información de los materiales. Estas dos funciones serán utilizadas por la clase Renderer, una clase encargada de comunicar correctamente la mayoría de las clases en ella misma.

4.2.2. Window

Esta clase se encarga de:

- Inicializar la ventana principal
- Inicializar y mantener el loop principal

4.2.3. Renderer

Esta clase se encarga de:

- Conectar todas las clases encargadas de la renderización
- Inicializar y actualizar shaders
- Escuchar eventos del teclado
- Movimientos de cámara
- Cargar escenas

4.3. Modificaciones

A la implementación original de [6], se le aplicaron una serie de modificaciones, las que se detallan a continuación.

4.3.1. Buffers en CPU

La clase Renderer lleva consigo la mayor carga en términos funcionales y operativos durante la ejecución de la aplicación y es un pilar fundamental para el llamado y llenado de buffers. Por esto se debe entender el orden del llamado a las funciones antes de implementar cambios. La función principal es la llamada *init()* y se explica en Algoritmo 2.

Algorithm 2 Función init, clase Renderer

```
1: Inicializa clase Shader
2: activar compute shader
3: crear gui
4: crear vertices
5: generar array vertices
6: generar Buffers
7: bind Buffers
8: bind array vertex
9: activar compute shader
10: LoadScenes()
11: LoadSVO()
```

El enfoque principal se da en las últimas 2 líneas. Para cargar y crear el SVO, la función llama a *LoadScenes()*, la cual se encarga de cargar los modelos tipo .vox y crear un SVO para cada uno. Luego, Renderer llama a la función *loadSVO()*, representada en el Algoritmo 3. Esta última función fue la principal modificada en la presente memoria de título.

Las modificaciones por el lado de la CPU se llevan a cabo entre las líneas 2 y 8 del Algoritmo 3. Luego de llamar a la función *toSVDAG()* se crea el DAC copiando los elementos del SVDAG por medio del constructor del DAC. Posteriormente se consideró lo siguiente: el DAC es una estructura de datos que depende en su mayoría de punteros, el lenguaje de la GPU es cercano a C, pero no permite punteros. Además, considera un repertorio limitado de tipos de datos (int, float y bool). Considerando las limitantes de la GPU, la estructura de datos DAC tiene que ser modificada para poder ser usada en la GPU. Esto último se puede hacer hasta un punto fijo, DAC cuenta con dos estructuras en su código: FTRep y bitRankW32Int, la primera estructura contiene a la segunda y es la representación del DACs en código, contiene la

Algorithm 3 loadSVO

```
1: Crear vector svdag y material
2: Llamar toSVDAG()
3: if svdag y mat buffers existen then
4:     Borrar buffer
5: end if
6: Crear buffer
7: Asignar memoria a buffer
8: Bind buffer
9: Activar compute shader
10: Entregar tamaño del SVO a buffer
```

función *accesFT()* encargada de entregar acceso directo a la posición solicitada. La segunda estructura bitRankW32Int se encarga de proveer a FTRep con las funciones base de una estructura de datos compacta, la más importante en este caso es rank() mencionada en la sección 2.1. bitRankW32Int se modificó como se muestra en la Figura 10, donde todas las variables que no son punteros se copiaron de manera íntegra por medio de un buffer, salvo la variable *owner*, la cual se reemplazó por una variable tipo `int`. Para los datos que son punteros, se optó por traspasar sus datos a GPU por medio de un buffer cada uno. De la misma manera, cada elemento que fuera un puntero fue reemplazado por un arreglo en GPU, el cual fue llenado por los datos enviados en los buffers.

Siguiendo con los cambios, en la línea 3 del Algoritmo 3, se presenta el condicional que deben cumplir todos los buffer que van a compute shader, estar vacíos antes de entregarles información. Cada buffer nuevo cumple esta condición. En la línea 6 se crea un buffer por cada elemento que requiera información dentro del compute shader. La mayoría de los punteros de la estructura que implementa el DAC son punteros a arreglos, por ende su modificación en GPU es utilizar los arreglos de buffer storage a la hora de pasar y usar la información de esos punteros. Finalmente, en la línea 7 y 8 ocurre lo mismo con la línea mencionada anteriormente, por cada puntero se crea un buffer, se asigna la memoria que utilizará ese buffer y luego se hace bind al buffer creado para enviar la información desde la CPU a la GPU.

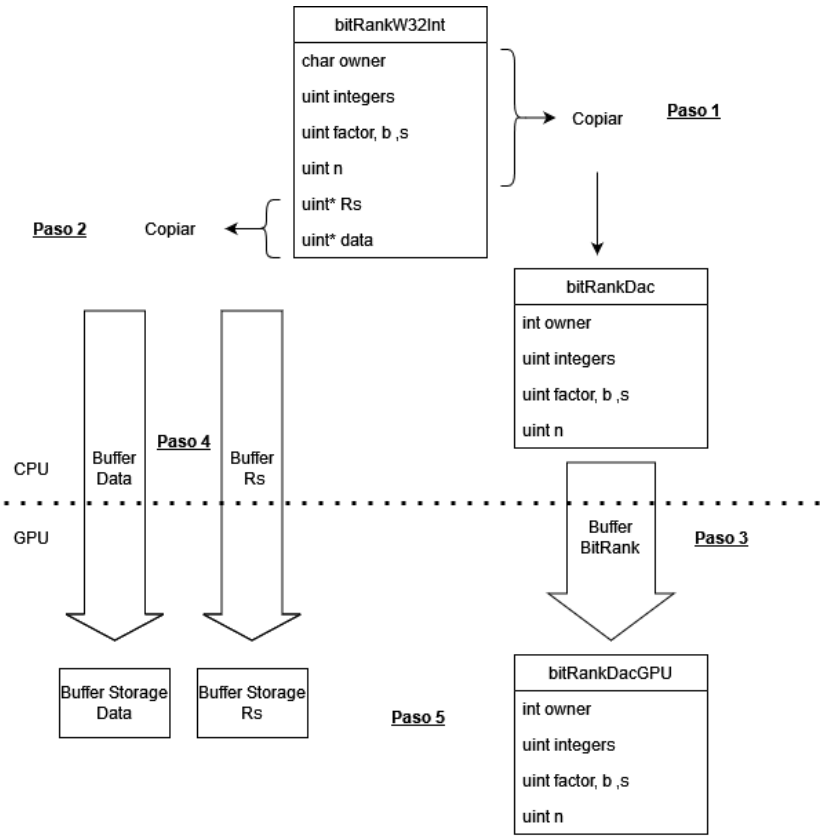


Figura 10: Diagrama explicando el paso de datos. *Paso 1*: copia de datos para la nueva estructura; *Paso 2*: copia de Rs y data a buffers; *Paso 3*: paso de datos del buffer de bitrank a GPU: *Paso 4*: paso de datos de los buffers data y Rs a GPU; y *Paso 5*: Buffer Storage de data y Rs, más la estructura creada en GPU bitRankDacGPU, reciben los datos para ser utilizados en GPU

4.3.2. Funciones en compute shader

Además de los punteros, se tuvo que modificar funciones de las estructuras FTRep y BitRank, las que implementan el DAC y el bitmap necesario para el DAC, respectivamente. En el caso de FTRep se modificó la función para acceder a un elemento de manera aleatorea, llamada *accessFT()*, la cual ahora debe usar los buffers pasados desde CPU. Además, se modificaron las funciones bitread y rank, las cuales recibían como entrada un puntero, el cual fue modificado. La implementación inicial del DAC y el soporte para bitmaps

contenían varíos componentes que dependían de punteros que fueron transformados a arreglos. Luego de modificarlos todos los punteros, el compute shader resultó con un total de 15 buffer storage, los cuales están en constante uso durante la ejecución.

Aquellas funciones mencionadas en el párrafo anterior fueron trasladadas a la GPU, especialmente accessFT() la cual consta de 2 funciones y todos los otros punteros enviados como buffer storage dentro de ella. Finalmente, dentro del compute shader se realizaron cambios en una función de raytracing, llamada *FindNodeAt()*, encargada de recorrer el SVDAG hasta una hoja. En lugar de usar el vector de enteros que implementaba el SVDAG, ahora esta función utiliza el DAC en GPU para acceder a los bitmasks y a los valores de material, si es que un nodo hoja existe.

4.3.3. Desafíos

La mayoría de los errores ocurrieron dentro del código de compute shader. El primer desafío, como fue mencionado anteriormente, es el rango limitado de tipos de dato que posee la GPU, limitado a int, float, bool, uint y double. Ya que la implementación original del DAC usaba variables tipo char, estas debieron ser modificadas a tipo int, lo que aumentó el espacio respecto a la implementación original. No obstante, la variante en GPU del DAC utiliza menos memoria que el SVDAG, por lo que es posible considerar esa memoria adicional como despreciable. El segundo desafío, y aquel que persiste, es el desempeño de la estructura DAC durante la ejecución del código, la cual es perjudicada por el alto número de buffers en constante uso. Esto último será detallado en la siguiente sección junto con los resultados de los tests aplicados a ambas estructuras.

5. Experimentos y resultados

5.1. Setup experimental

Los experimentos se llevaron a cabo en un computador con las siguientes características:

- Procesador Intel Core i5-12400
- Memoria RAM de 32 GB

- Sistema Operativo Windows 11 Pro Education
- compilador Microsoft Visual C++ ver. 1940
- AMD RX6600 8GB
- OpengGL versión 4.5

El dataset utilizado está compuesto por 8 modelos en formato .vox, los cuales eran parte del código original del SVDAG. Se puede apreciar su renderización en el Anexo A. Para medir experimentalmente el rendimiendo de las dos implementaciones, se han creado 3 movimientos de cámara automatizados para cada experimento. Cada experimento se repetió 20 veces, alcanzando un total de 960 experimentos independientes. Por cada experimento se extrajeron los fotogramas por segundo (FPS) y el espacio, en bytes, utilizado en memoria

Los movimientos de cámara utilizados consisten en:

- *Cámara 360 grados*: La cámara gira en 360 grados alrededor de cada modelo.
- *Cámara acercamiento*: Desde un origen lejano, la cámara se acerca al modelo hasta llegar cerca del borde del mismo.
- *Cámara por puntos*: La cámara sigue un movimiento aleatoreo, saltando desde un punto a otro.

Antes de analizar los resultados, la Tabla 1 muestra estadísticas de los modelos que se detallan a continuación:

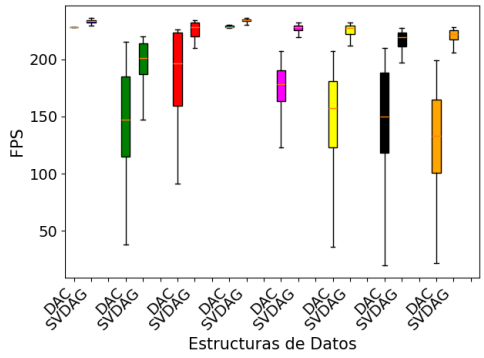
- *Test*: 4 cubos de distintos colores en posiciones asimétricas
- *Terrain*: Un terreno que contiene tierra y agua, su forma y cantidad de elementos varía con cada ejecución del código
- *Stairs*: Escaleras esculpidas sobre un cubo
- *Vox 3x3x3*: Un cubo con orificios simétricos
- *Deer*: Un ciervo

- *T-rex*: Un dinosaurio asimilandose a un tiranosaurio rex simplificado
- *Car*: Un auto
- *Castle*: Un castillo
- *Doom*: Figuras asimilando el aspecto 2D de personajes del videojuego DOOM
- *Teapot*: La tetera de la biblioteca de la Universidad de Utah

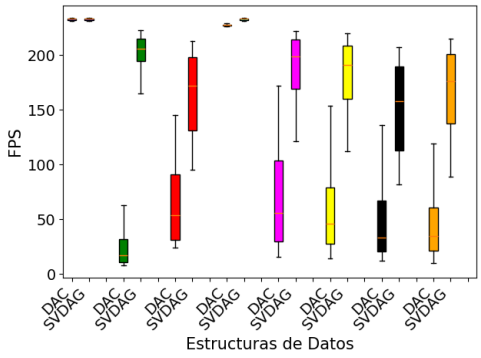
5.2. Resultados

En los gráficos de caja presentes en las Figuras 11(a), 11(b) y 11(c) representan el promedio de los FPS por cada dataset durante el movimiento de la cámara. Cabe mencionar la existencia de dos modelos adicionales que forman parte del dataset para la Figura 12, estos son Doom y Teapot, que no pudieron ser parte de todos los experimentos ya que debido a su gran tamaño se veía perjudicado el funcionamiento del computador al intentar ejecutar los experimentos de movimiento de cámara.

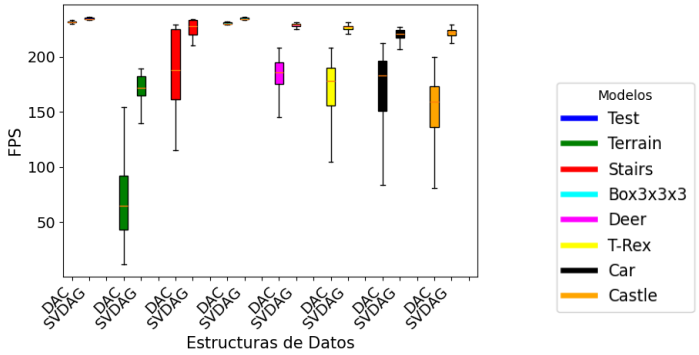
De los 3 gráficos se pueden apreciar la similitud de los resultados a lo largo de los 3 movimientos de cámara. Los gráficos son gráficos de caja, en general, representan la distribución de los datos y en este caso representa la distribución de FPS. La línea central de la caja indica la mediana de los FPS, los límites de la caja representan los percentiles 25 y 75 y su longitud (IQR), representando la diferencia entre esos dos percentiles. Las líneas que siguen después de las cajas, llamadas "bigotes", representan la varianza esperada de los FPS y se extienden hasta los valores mínimos y máximos de FPS. Los FPS son un indicador de la eficiencia del raytracing sobre la estructura de datos elegida. Aunque el algoritmo raytracing no es el centro de esta memoria de título, es necesario considerarlo para evaluar la utilidad de los DACs sobre el SVDAG, en términos de tiempo, en una aplicación real. Considerando lo anterior, mientras más rápido se recorra la estructura de datos, más FPS habrá. Ahora, si se toma de ejemplo las barras rojas del gráfico perteneciente a la Figura 11(c), se puede apreciar como en el uso de la estructura SVDAG hay mas FPS en general que la estructura DAC. Otro punto a mencionar es que los gráficos de caja de la estructura SVDAG son más cortos, en general, que aquellos perteneciente a DACs indicando la estabilidad que hay a lo largo de la ejecución en el momento que se recorre SVDAG con raytracing. Esto



(a) Movimiento de cámara 360 grados



(b) Movimiento de cámara acercamiento



(c) Movimineto de cámara por puntos

Figura 11: Experimentos de los 3 movimientos de camara automatizados sobre el dataset

lleva a la conclusión de que en términos de tiempo, el DAC es más ineficiente que el SVDAG.

Los resultados obtenidos de estos experimentos están dentro de lo esperado debido al mayor problema y desafío descrito en la Sección 4.3.3, destacando el gran número de buffers en comparación a los que utiliza el SVDAG en su implementación original. Como se explica en la Sección 4.3.2 se crearon un total de 15 buffer storage adicionales para la estructura *FTRep* debido a la falta de opciones para reemplazar punteros, que en este caso son utilizados como arrays. En la GPU se pueden crear arrays de la misma manera que

en C/C++, los cuales tienen sus limitantes dependiendo del uso que se le den. En este caso, su uso principal fue para el almacenamiento de datos de los buffers y para cumplir su rol no se requiere la declarar su tamaño al momento de crearlos [3]. El elevado número de buffers perjudicó el desempeño durante la ejecución, lo que se puede apreciar al ver las diferencias de FPS con modelos de gran tamaño.

Como último punto de análisis para los 3 gráficos, aunque el DAC sea más lento en cuestión de tiempo debido a los buffers, los FPS no indican con exactitud la verdadera rapidez con la que se accede a la información del DAC en GPU. Esto último considera la dificultad que hay para poder acceder a información dentro de la GPU, por ejemplo, no se puede pedir la información que posee una variable dentro de la GPU desde la CPU en un momento exacto de la ejecución, si se deseara la información que posee la variable hay distintas maneras, en el caso de esta memoria de título, se optó por copiar la implementación de funciones de GPU y adaptarlas a CPU para saber que valores eran guardados en las variables creadas en GPU, especialmente las funciones responsables del raytracing. El DAC tiene la capacidad de acceso aleatorio, lo cual indica que es directo como acceder a los datos de una posición de un array, por ende solamente los buffers perjudican el desempeño total debido a la gran cantidad de consultas que debe realizar la GPU a los buffers para acceder a la información, lo cual es un problema de código en GPU más que de la estructura DAC en sí.

La implementación de la estructura DAC cumplió las expectativas en términos de espacio ahorrado en memoria como se puede apreciar en la Figura 12. Se obtuvo, en promedio para los modelos más grandes, 62 % de ahorro de espacio. Este ahorro en espacio se calcula considerando la cantidad de elementos dentro de la estructura DAC en comparación con la estructura SVDAG. Las estructuras en la Figura 12 comparan el SVDAG DAC y DAC GPU, esta es considerando la memoria que utilizaría DAC dentro de la GPU que son 2 bytes más que la versión original, ya que para entregar la información de basebits, correspondiente a FTRep, a su correspondiente buffer, se tuvo que preparar memoria para un `uint` en vez de un `ushort`.

A pesar de su rendimiento temporal, el DAC ahorró una cantidad significativa de memoria en comparación con el SVDAG y este punto no es despreciable si se desea utilizar al DAC como reeemplazo del SVDAG para renderizar modelos grandes (ver Figura 12). En el caso de modelos muy pe-

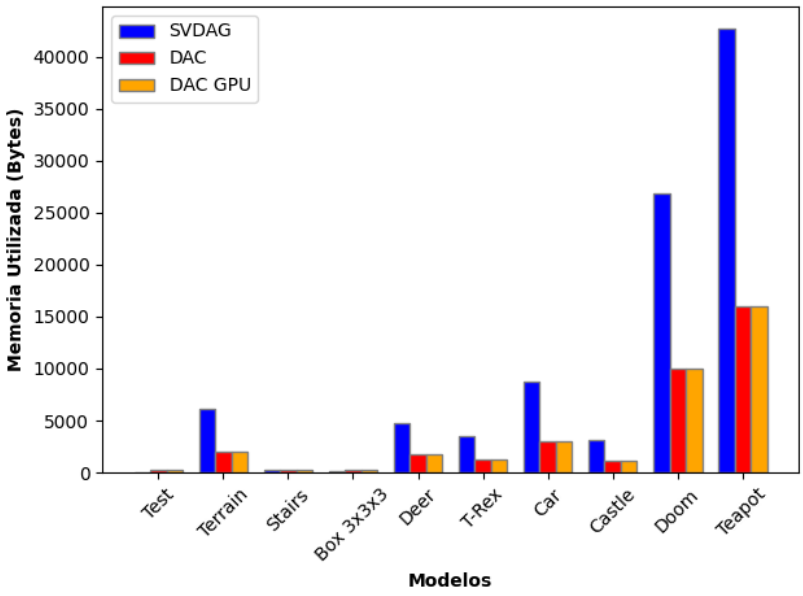


Figura 12: Comparación del uso en memoria de cada estructura de datos por modelo

queños, el ahorro de espacio del DAC no es significativo, por lo que no se recomienda en ese caso. Volviendo a lo mencionado en párrafos anteriores, el número elevado de buffers afectaba el rendimiento general de la estructura. Esta mejora da una idea sobre las mejoras a la implementación actual de DAC, la cual deberá tener como enfoque principal reducir el número de buffers. Una alternativa para conseguir esa mejora es combinar distintos arrays de la estructura FTRep, para de esa manera tener menos arrays en total y en consecuencia, necesitar menos buffers para su traspaso a la GPU.

6. Conclusiones y trabajo futuro

En conclusión, el DAC ocupa menos de la mitad de la memoria que usa el SVDAG. La eficiencia del DAC en uso de memoria se debe a que la implementación usa por cada valor la cantidad exacta o cercana de bits que dicho valor necesita para ser representado. Aun considerando lo anterior, el DAC posee un desempeño peor al SVDAG en términos de tiempo a la hora de renderizar, debido a que para permitir acceso aleatorio debe realizar va-

Dataset	**Tamaño**	**Cantidad vóxeles**
Test	4	64
Terrain	32	32.768
Stairs	32	32.768
Vox 3x3x3	4	64
Deer	32	32.768
T-rex	32	32.768
Cars	64	262.144
Castle	32	32.768
Doom	128	2.097.152
Teapot	128	2.097.152

Cuadro 1: Tabla demostrativa de los tamaños de los modelos representado en voxeles. La columna *Tamaño* corresponde a la magnitud del modelo en cada eje X, Y y Z. La columna *Cantidad vóxeles* es el tamaño máximo que podría ocupar cada modelo. Se obtiene al calcular $Tamaño^3$.

rias operaciones en bitmaps compactos. Esta ineficiencia debe ser el mayor enfoque para trabajo futuro si se desea mejorar su desempeño temporal.

A modo de trabajo futuro, se recomienda el estudio de otras plataformas que permitan codificar en GPU, como CUDA, e implementar DAC en dichas plataformas. Otro potencial trabajo futuro es, manteniendo el uso de OpenGL, juntar ciertos arrays de las estructuras *FTRep* y *bitRankW32Int*, las que implementan DAC, de manera que se puedan utilizar en un solo buffer sin comprometer como la estructura funciona. Ahora mismo, la implementación del DAC sigue una lógica de CPU y no de GPU.

Referencias

[1] cratchapixel. Overview of the ray-tracing rendering technique. https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview/ray-tracing-rendering-technique-overview.html, 2022. Accessed: 2024-08-02.

[2] Susana Ladra González. *Algorithms andCompressed Data Structures for Information Retrieval*. PhD thesis, Universidad de Coruña, 2011.

[3] Kronos Group. Data type (glsl). https://www.khronos.org/opengl/wiki/Data_Type_(GLSL), 2022. Accessed: 2024-08-14.

[4] Kronos Group. Opengl - getting started. https://www.khronos.org/opengl/wiki/Getting_Started, 2022. Accessed: 2024-08-01.

[5] Kronos Group. Rendering pipeline overview. https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview, 2022. Accessed: 2024-08-02.

[6] Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. High resolution sparse voxel dags. *ACM Trans. Graph.*, 32(4), jul 2013.

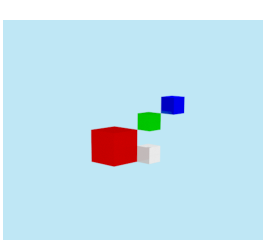
[7] Susana Ladra, José R. Paramá, and Fernado Silva-Coira. Compact and queryable representation of raster datasets. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*, SSDBM '16. ACM, July 2016.

[8] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, I3D '10. ACM, February 2010.

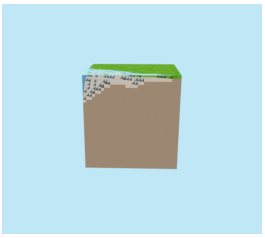
[9] Donald Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–147, June 1982.

[10] Gonzalo Navarro. *Compact Data Structures – A practical approach*, chapter 3, pages 39–59. Cambridge University Press, 2016.

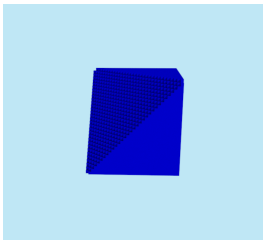
A. Modelos utilizados



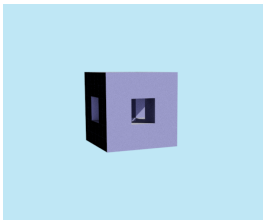
(a) Modelo *Test*



(b) Modelo *Terrain*

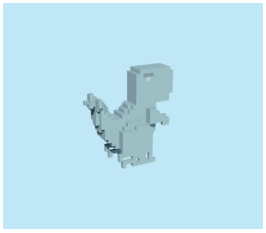


(c) Modelo *Stairs*

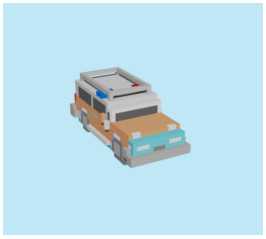


(d) Modelo *Vox 3x3x3*

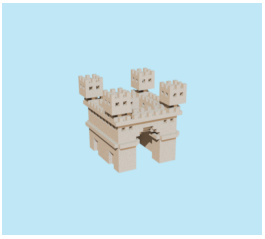
(e) Modelo *Deer*



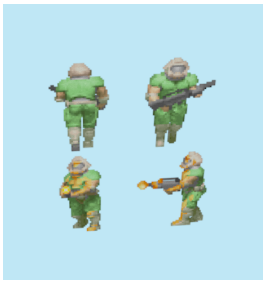
(f) Modelo *T-Rex*



(g) Modelo *Car*



(h) Modelo *Castle*



(i) Modelo *Doom*

(j) Modelo *Teapot*

Figura 13: Modelos utilizados en los experimentos