

Máster en Computación de Altas Prestaciones

Proyecto Fin de Máster

Point Cloud Manager

Sistema multirresolución para el tratamiento de grandes datasets de nubes de puntos 3D

Autor Alberto Jaspe Villanueva

Directores Emilio J. Padrón González

Javier Taibo Pena

A Coruña, 3 de Septiembre de 2012

RESUMEN

Con la reciente evolución de los equipos láser LIDAR y los escáneres 3D, las nubes de puntos (conjuntos de muestras espaciales, normalmente de superficies de objetos, con diversa información asociada) cada vez son más utilizadas en múltiples disciplinas, como la ingeniería civil, arquitectura, topografía o efectos especiales y animación 3D. Pero a medida que los requisitos de precisión aumentan, crece también el tamaño de las BBDD generadas, haciendo difícil su tratamiento. Además, con la generalización de uso de GPUs en equipos de escritorio comunes, y su evidente orientación hacia los cálculos tridimensionales, sería deseable poder aprovechar su enorme potencia de computación para procesar este tipo de *data-sets*.

Se presenta en esta memoria el sistema Point Cloud Manager, para la gestión, procesado y visualización de grandes *datasets* de nubes de puntos en equipos convencionales, cualesquiera sean sus capacidades. Para ello, se ha diseñado una estructura de datos espacial multirresolución y un sistema de caché software de dos niveles, capaz de procesar este tipo de datos, independientemente de su tamaño, así como una colección de herramientas para su gestión y visualización.

PALABRAS CLAVE

Nubes de puntos, LiDAR, escáner 3D, grandes *datasets*, GPGPU, caché software, jerarquía de memoria, estructuras de datos espaciales, sistemas multirresolución, niveles de detalle.

Quiero agradecer, en primer lugar, a los directores de este proyecto, **Javier Taibo** y **Emilio J. Padrón**, por confiar en mí, compartir conmigo su experiencia y animarme en los momentos de duda. Sin ellos este trabajo no hubiera sido posible.

Por supuesto, agradezco también a mi **familia y amigos** por estar siempre ahí, incluso a la vuelta de mis prolongados retiros.

Igualmente a mis **colegas de máster**, de los que he aprendido muchas cosas durante este año, y que han demostrado un gran nivel de compañerismo y solidaridad.

Finalmente, me gustaría además reconocer la ayuda prestada desde las siguientes instituciones y personas, que han apoyado incondicionalmente este proyecto:

- La **Xunta de Galicia** y su **Instituto de Estudios del Territorio**, por concederme acceso a sus vuelos LIDAR autonómicos, y particularmente a **Manuel Borobio** y **Nacho Varela**, por las gestiones realizadas con tal fin, y el apoyo y entusiasmo que muestran siempre hacia mis proyectos e investigaciones. Son un gran ejemplo de que la administración puede hacer las cosas realmente bien.

- **ENMACOSA S.A.**, por cederme varios *datasets* LIDAR terrestres para poder poner a prueba el sistema desarrollado en este trabajo, y particularmente a **Rubén Cifrián** y **Raúl Pernas**, porque creen, como yo lo creo, que uniendo puntos ganamos todos.

- A todos los integrantes del **laboratorio ViC del CRS4** (Cerdeña, Italia) porque sus novedosas investigaciones son parte clave de este proyecto, y por mostrarse siempre abiertos a atender mis consultas técnicas. Han demostrado además ser unos perfectos anfitriones y buenos amigos.

- A la **Universidad de Stanford**, por ceder sus escaneos 3D para testear el sistema desarrollado.

A todos vosotros, gracias.

Alberto Jaspe

ÍNDICE

1 INTRODUCCIÓN

Se presenta en este documento un nuevo sistema para el tratamiento de grandes *data-sets* de nubes de puntos 3D, denominado **Point Cloud Manager** (en adelante **PCM**). Este sistema cubre todas las fases necesarias para poder convertir, procesar y visualizar este tipo de datos tridimensionales, las nubes de puntos provenientes de escáneres láser que, como se verá en los próximos capítulos, constituyen hoy en día una de las herramientas más poderosas en numerosos campos, desde la ingeniería, arquitectura o topografía hasta la animación y los videojuegos.

La presente memoria se divide en los siguientes capítulos:

1. **Introducción**, donde se presenta un breve resumen del proyecto, así como sus motivaciones, objetivos y ámbitos de estudio.
2. **Contextualización y fundamentos**, con algunas bases que introducen al lector en los distintos ámbitos abarcados en este documento, desde el origen de los datos LIDAR hasta conceptos técnicos computacionales imprescindibles para la comprensión del funcionamiento del sistema.
3. **Estado del arte**, en el que se describen de forma escueta las técnicas multirresolución más utilizadas en la actualidad, y los sistemas software existentes en este ámbito.
4. **Análisis de la solución propuesta**, donde se ofrece una explicación funcional de la solución planteada, centrándose en la estructura de datos escogida y la jerarquía de memoria ideada.
5. **El paquete de software PCM**, en el que explicará la arquitectura general, el pipeline de trabajo según los roles de usuario y la metodología de trabajo.
6. **Diseño e implementación**, donde se mostrará con lenguaje UML el diseño software generado, y se explicará con detalle el funcionamiento de cada una de las partes que lo componen.
7. **Resultados**, para exponer algunos datos significativos del funcionamiento del sistema.
8. **Conclusiones y futuro de PCM**, donde se explicarán la proyección del proyecto, sus líneas de trabajo futuras, y en qué forma se han cumplido los objetivos expuestos.
9. **Bibliografía**, con referencias bibliográficas a los documentos más importantes en este tema.

1.1 MOTIVACIÓN

Hace ya varias décadas que todo tipo de información de carácter espacial, ya bien sea procedente de algoritmos o de herramientas de modelado, se utiliza en distintos procesos computacionales. Desde simulaciones atmosféricas para predicción meteorológica, procesos en estructuras moleculares, reconstrucción de edificios en CAD, hasta visualización de objetos en 3D. Pero recientemente, debido al abaratamiento de costes y la gran ganancia de precisión de los equipos, los sistemas **LIDAR** (*LIght Detection And Ranging*) se están introduciendo cada vez más en las tareas habituales en múltiples campos de la ingeniería. Estos sistemas permiten la adquisición automática de muestras puntuales de las superficies

de los objetos que lo rodean. Como se verá en el siguiente capítulo, estos equipos pueden además ser transportados sobre distintos medios (aéreos, terrestres, marítimos, etc.) y programados para escanear en múltiples pasadas y conseguir así tomas de espacios mayores y más complejos.

Dependiendo del tipo de LIDAR utilizado, o en conjunción con otros equipos sensores, cada una de las muestras adquiridas, además de su correspondientes coordenadas XYZ en el espacio, pueden llevar asociada multitud de información de carácter físico, como color del material, índice de reflectividad, temperatura, dureza, etc. Al conjunto combinado de todos estos datos (*dataset*), procedentes normalmente de múltiples escaneos de la zona de interés, se le denomina *Point Cloud* o **Nube de Puntos**.

Con los equipos de escaneo que existen en la actualidad, y dado que cada vez se utilizan para problemas más complejos y que requieren mayor precisión, un *dataset* de este tipo se puede componer de cantidades ingentes de puntos, que pueden llegar con fácilmente a órdenes de los **miles de millones**. En términos computacionales, esto puede llegar a suponer varias decenas o centenas de Gigabytes de información, que en las computadoras de consumo utilizadas en la actualidad, costaría tan sólo almacenar incluso en los últimos niveles de la jerarquía de memoria (discos duros). Sin embargo, el propósito de estos datos es habitualmente ser objetivo de algoritmos de carácter espacial, que puedan sustraer información relevante para el operario, como por ejemplo filtrar *outliers*, conseguir planos CAD, reconstruir superficies o curvas de nivel, analizar estructuras, etc. Incluso la visualización 3D más simple se torna problemática debido a su gran tamaño. Se puede aseverar que la tendencia de crecimiento de estos *datasets* resulta ser, como mínimo, tan pronunciada como las capacidades estándar de memoria RAM de los PCs, pero multiplicando éstas en varios niveles de magnitud [1].

Por otra parte, hasta la más sencilla de las *workstation* que suelen utilizar los operarios de estos ámbitos utilizan tarjetas gráficas dedicadas. Es estos últimos años, el uso de técnicas GPGPU se está estandarizando para procesos con altas necesidades computacionales, y gracias a lenguajes como ***CUDA*** u ***OpenCL***, los desarrolladores pueden contar con hardware de altas prestaciones en máquinas propias de usuarios de escritorio. Sin embargo, para conseguir el máximo rendimiento de esas tarjetas, los datos a procesar deben encontrarse en su propia zona de memoria (VRAM) que habitualmente es menor que la propia memoria del sistema (RAM).

Todo lo expuesto hasta el momento induce a pensar que la gestión y procesado de estos datos debe llevarse a cabo mediante el uso de supercomputadoras en el ámbito del "*Big Data*". **Este proyecto trata de ofrecer una solución para el tratamiento de grandes *datasets* de nubes de puntos 3D en PCs de escritorio**, mediante la **definición de una nueva estructura de datos especial**, y una **arquitectura software caché de dos niveles, que aproveche el carácter espacial de este tipo de información** para mover pequeños segmentos de datos entre el disco (HDD), la memoria de la CPU (RAM) y la memoria de la GPU (VRAM). De esta forma el usuario o el programador de procesos podrán trabajar con la nube de puntos completa en cualquier máquina, sean cuales sean las capacidades de memoria de ésta, realizando peticiones espaciales al sistema, que mantendrá en su propia jerarquía de memoria gestionada por software

una versión lo más detallada posible de la región del modelos sobre la que el usuario desee trabajar, ya bien sea esta un mínimo detalle o el dataset completo.

1.2 OBJETIVOS

Según las motivaciones descritas en la sección anterior, el objetivo principal de este proyecto es el estudio y creación de una solución software capaz de gestionar nubes de puntos 3D independientemente de las capacidades de la máquina y del tamaño del *dataset*, que pueda además aprovechar la potencia de computación que ofrecen las GPUs. Para ello, se listan los siguientes cuatro objetivos concretos:

1. Análisis y estudio del estado del arte en el campo de las estructuras de datos espaciales multirresolución, y desarrollo de una propia para su uso en el sistema.
2. Desarrollo teórico y construcción de un prototipo de gestión de la jerarquía de memoria gestionada por software que aproveche el carácter espacial de los datos.
3. Desarrollo de un paquete software que incluya las herramientas necesarias para convertir, visualizar y procesar los *datasets* de información provenientes de escáneres laser LIDAR u otras fuentes, utilizando el sistema descrito en punto anterior.
4. Implantación de una infraestructura de desarrollo y un sistema de información propio para el proyecto, capaz de soportar trabajo concurrente, y que pueda monitorizar el estado del proyecto, con vistas a su apertura a la comunidad.

El software descrito deberá además cumplir los siguientes requisitos:

- **Multiplataforma**, debe ser compatible con los sistemas operativos más utilizados en el momento, Microsoft Windows, Linux y Mac OSX.
- **Facilidad de uso**, tanto para usuarios como para programadores que quieran desarrollar sus algoritmos espaciales.
- **Generalidad**, de forma que pueda trabajar con distintos formatos de nubes de puntos, datos asociados, plataformas, etc.
- **Extensibilidad**, aprovechando los patrones de diseño [2], para que sea fácilmente ampliable a nuevos formatos y técnicas.

2 CONTEXTUALIZACIÓN Y FUNDAMENTOS

Este capítulo trata de enmarcar al lector en el ámbito del problema y ofrecer unos principios básicos en los distintitos ámbitos tocados por la solución propuesta. Para ello, se comenzará describiendo formalmente las nubes de puntos y sus formas de concepción, fundamentalmente de tipos escáneres LIDAR, aunque también de otras fuentes, y sus formatos. Por otra parte, se explicarán algunas nociones sobre sistemas multirresolución y niveles de detalle, y finalmente se comentarán las jerarquías de memoria en las máquinas actuales (incluyendo GPUs) y el papel jugado por las cachés software.

2.1 NUBES DE PUNTOS

Se denomina "nube de puntos" o *point cloud* a un conjunto de vértices en un sistema de coordenadas tridimensional, sin topología alguna que los relacione. Estos vértices se definen normalmente por sus coordenadas XYZ, y suelen representar muestras de la superficie externa de un objeto. Su densidad o distribución atiende exclusivamente al método con que haya sido creado el dataset.

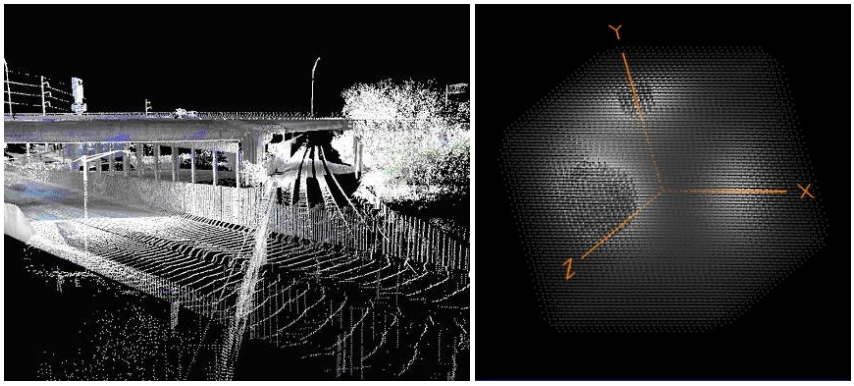


Ilustración 1. Visualización de nubes de puntos y su sistema de coordenadas.

En una nube de puntos, cada muestra o punto puede llevar asociado una cantidad arbitraria de información, como puede ser color del punto, la normal de la superficie, reflectividad del material, clasificación, temperatura, carga, etc. A efectos computacionales, puede decirse que el espacio ocupado por cada punto es la suma del espacio de sus tres coordenadas (simple o doble precisión) más la suma de los datos asociados. Por lo tanto, y de forma muy general, el tamaño en bytes de una nube de puntos sería

$$S_{pc} = N_p(12P + S_{data})$$

Donde S_{pc} sería el tamaño en bytes, N_p el número de puntos del *dataset, P* valdría 1 para simple precisión y 2 para doble, y S_{data} el tamaño en bytes de los datos asociados a cada punto. El número 12 sale de multiplicar 4 bytes de la simple precisión por las 3 coordenadas. De esta forma, por ejemplo, una nube de 500 millones de puntos, con color RGB, índice de reflectividad, vector normal, y un byte para una clasificación de hasta 256 clases, con todos los flotantes en simple precisión, ocuparía

$$S_{pc} = 500 \cdot 10^6\big(12 + (3 \cdot 4 + 4 \ + 3 \cdot 4 + 1)\big) = 20{,}5 \cdot 10^9 \ bytes \ \approx 19 \ GB$$

2.1.1 SISTEMAS LIDAR DE ADQUISICIÓN DE NUBES DE PUNTOS

Como se ha comentado en la introducción, **LIDAR** es el acrónimo de ***Light Detection and Ranging***, y hace referencia a la tecnología que permite determinar la distancia desde un emisor láser a un objeto o superficie utilizando un haz láser pulsado, midiendo la diferencia de tiempo entre su emisión y recepción. Además, en función de la intensidad y forma de haz recibido, se detecta también el índice de reflectividad del material muestreado. Automatizando este proceso, en una sesión de muestreo pueden tomarse hasta cientos o miles de estas muestras, cada una de las cuales tendrá unas coordenadas espaciales en el marco de referencia convenido inicialmente. Los errores de medición dependerán de la calibración y calidad del equipo utilizado.

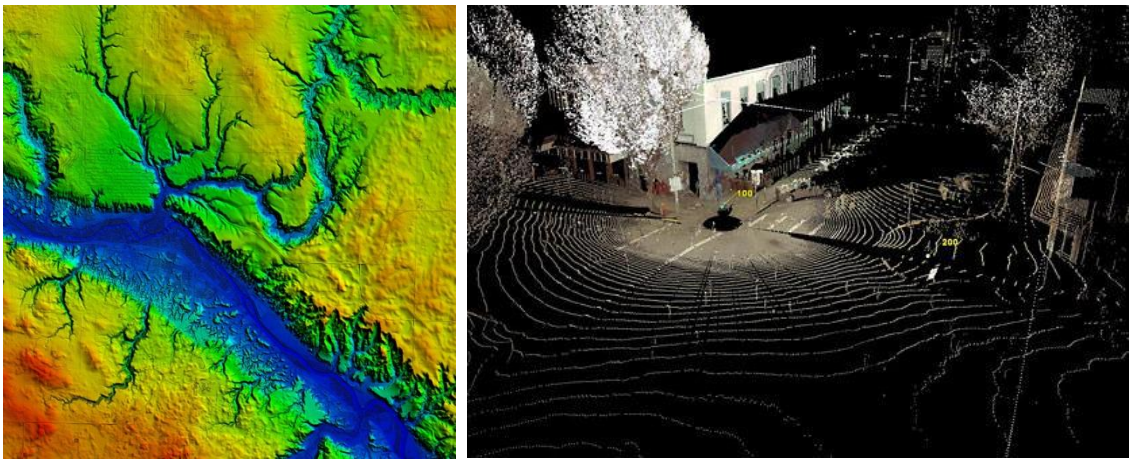


Ilustración 2. A la izquierda, una nube de puntos (como mapa de alturas, por colores) típica de un LIDAR aéreo. A la derecha, un entorno urbano adquirido con LIDAR terrestre.

En función del trabajo a realizar, y de la extensión de la zona a cubrir, se pueden realizar distintos montajes de equipos LIDAR. Algunos de ellos son los siguientes (Ilustración 3):

- **Terrestres estáticos**, que mediante un geoposicionamiento se sitúan en distintos lugares para obtener varios escaneos, que en un post-proceso serán conjuntados en un solo dataset. Es el más común para trabajos de edificaciones, patrimonio, etc.
- **Terrestre dinámico**, también conocido como *Mobile Mapping*, donde se montan el sistema LIDAR sobre algún vehículo terrestre (todoterrenos, quads, trenes, etc.) y con una equipación especial se realiza un escaneo continuo, que va generando un dataset en los alrededores de la ruta por la que pasa. Utilizado en mantenimiento de carreteras, paisajismo, urbanismo, etc.
- **Aéreo**, con un montaje en una aeronave especialmente adaptada, que va escaneando el terreno cenitalmente. En muchos casos, este tipo de equipamiento emite haces de gran diámetro, de forma que les permite clasificar las muestras en función de la cantidad de luz recibida de vuelta, para detectar masas arbóreas, agua (ríos, lagos, mar), etc. Muy utilizado en todos los campos de la topografía.
- **Batimétrico**, para escanear suelos marinos, con un montaje parecido al aéreo pero sobre un barco, y con tecnología adaptada para medir correctamente sobre un medio líquido.

Ilustración 3. A la izquierda, un equipo de LIDAR terrestre de la marca Leica. En la imagen central, visión explicativa de un vuelo LIDAR. A la derecha, un sistema de de *mobile mapping* (IPS2 de la casa Topcon).

Las posibilidades de aplicación de las nubes de puntos en ingeniería son prácticamente infinitas, y debido al abaratamiento y ganancia de precisión de los equipos, se están empezando a explotar es estos últimos años. Desde la generación de Modelos Digitales de Terreno (DTM) hasta cálculos de estructuras, simulaciones de carga, búsqueda de patrones y un largo etcétera de procesos, cualquiera de estos procesos tienen un claro carácter espacial, y exige poder ejecutar operaciones sobre distintas regiones del *dataset*.

2.1.2 OTRAS FUENTES

No sólo los equipos LIDAR pueden captar nubes de puntos del entorno real, existen otro tipo de tecnologías, que se denominan de forma genérica **escáneres 3D**, capaces de adquirir este tipo de *datasets*. Estos pueden estar basados en distintas tecnologías, aunque el funcionamiento suele ser parecido: emitir algún tipo de radiación o pulso y medir el tiempo de retorno. Algunos ejemplos son los equipos que utilizan ultrasonidos, luz infrarroja, ondas de radio, etc.

Un ejemplo muy en auge en estos momentos sería el ***Kinect* de *Microsoft***, dispositivo de entretenimiento capaz de captar los gestos y movimientos de los usuarios e interpretarlos por software para Interacción Computador – Humano (CHI). Su funcionamiento consiste en la emisión de un patrón de luz infrarroja emitida continuamente en una frecuencia particular, y la recepción mediante una cámara especial de los rebotes de dicha frecuencia. Su tecnología permite obtener una nube de puntos dinámica y en tiempo real, como un mapa de profundidades de los objetos que se sitúan frente a él. Puede verse este efecto en la Ilustración 4.

Ilustración 4. Nube de puntos adquirida en tiempo real con el dispositivo Kinect de Microsoft.

Pero además, las nubes de puntos pueden proceder de procesos computacionales o algoritmos, no necesariamente del mundo real. **Múltiples tipos de simulaciones trabajan con puntos para obtener muestras discretas de campos vectoriales**. Por ejemplo, las simulaciones de zonas límite de inestabilidad entre gases de *Richtmyer-Meshkov*, puede generar tantas muestras puntuales como precisión se requiera; o el sistema de renderizado utilizado por la compañía *Pixar* en sus películas de animación, *renderman*, que utiliza nubes de puntos para los cálculos intermedios de la iluminación indirecta [3].

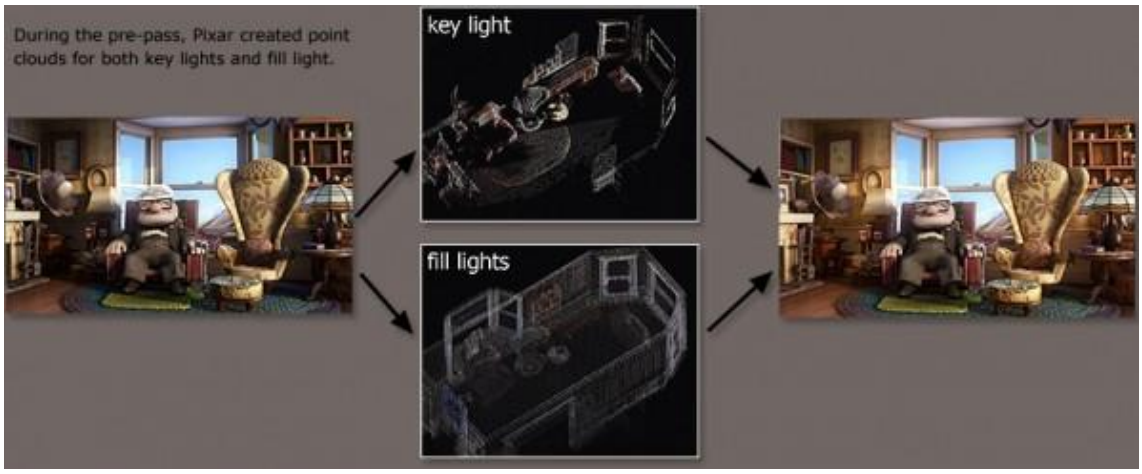


Ilustración 5. Nubes de puntos utilizadas como parte del proceso de iluminación de la compañía Pixar en la película "Up" (Fuente: Pixar Inc.)

2.1.3 FORMATOS

A pesar de la relativa sencillez de este tipo de datos, existen múltiples variantes de formatos utilizados, en función del fabricante del equipamiento, los sensores conectados al dispositivo, y los programas de post-procesado utilizados. Los más importantes son los siguientes:

- **XYZ**: el formato más básico, que almacena en ASCII solamente la información de coordenadas de los vértices.
- **PTS**: formato ASCII propietario de *Leica*, capaz de almacenar información asociada a cada vértice y de separar distintos conjuntos de puntos.
- **PTX**: el formato complementario al PTS, también de *Leica*, que almacena por separado cada una de las sesiones de escaneado y la información de calibración del escáner, así como las matrices de transformación del marco de coordenadas de referencia.
- **LAS**: el estándar de facto para los escaneos LIDAR aéreos, público y bien documentado. Capaz de almacenar múltiple información sobre los vuelos y escaneos, y de clasificar la información por capas, cuenta además con LAStools y LASlib, un paquete software *open-source* desarrollado por la Universidad de Carolina del Norte [4].
- **SD**: formato ampliamente utilizado para muestreos 3D, normalmente con escáneres de mano.

- **PLY**: formato extendido en el campo de la investigación, de descripción de objetos 3D de gran tamaño, desarrollado por la Universidad de Stanford, que proporciona también un conjunto de herramientas y librerías para su gestión [5].

Debido al tamaño que suelen alcanzar estos *datasets*, es común que suelan trocearse en varios ficheros, de entre 500MB y 1,5GB.

2.2 SISTEMAS MULTIRRESOLUCIÓN

Se denominan **sistemas multirresolución** (*multiresolution systems* o **MRS**) a aquellas arquitecturas y estructuras de datos capaces de almacenar y gestionar distintos niveles de detalle de un modelo matemático, permitiendo el análisis multirresolución (*multiresolution analysis* o **MRA**) o aproximaciones multiescala (*multiscale approximation* o **MSA**). Estos métodos de análisis, introducidos a finales de los 80's por *Stephane Mallat* e *Yves Meyer*, permiten obtener soluciones a procesos complejos cada vez más precisas utilizando distintas escalas de detalle del modelo (niveles de detalle, *levels of detail* o **LOD**s), y son muy conocidas por las técnicas concernientes a los *wavelets* [REF] que se utilizan ampliamente hoy en día en el tratamiento de imágenes digitales y en algoritmos de compresión y codificación.

En el campo de los Gráficos en Computación, los **sistemas multirresolución y niveles de detalle** constituyen una práctica habitual para casi cualquier motor de render [6]. Los modelos 3D que se utilizan habitualmente están compuestos por polígonos, y al margen de otras técnicas, el número de polígonos define la calidad visual del modelo. Pero debido al coste de procesado y ocupación de memoria, que suele ser bastante limitada, según la distancia al punto de vista desde el que se genera la imagen, se utilizan simplificaciones de éste. El mismo caso sucede con las texturas de los modelos, que son imágenes que se "adhieren" (mapean) a los polígonos de los modelos para llenarlos de detalle y color. Una textura "grande" (pongamos, 1024x1024) será una pérdida de tiempo de GPU y memoria si está aplicada sobre un objeto muy lejano, que apenas se ve en la proyección. Ejemplos de niveles de detalle tanto en polígonos como en texturas pueden verse en la Ilustración 6.

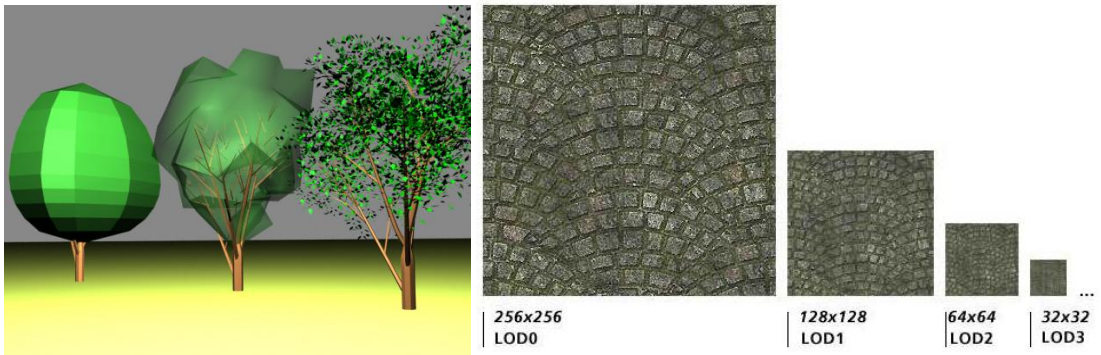


Ilustración 6. Niveles de detalle (LODs). A la izquierda, sobre un modelo 3D polignonal. A la derecha, sobre una imagen (textura), aplicando la técnica de *mipmap* (Fuentes: landscapemodeling.org y tomshardware.com)

La utilización de este tipo de sistemas produce un sobrecoste (que, obviamente, compensa respecto a su carencia) que se puede resumir en las siguientes tres necesidades:

- **Pre-proceso de los datos originales**, para crear a partir de éstos los modelos simplificados y generar la estructura de datos espacial que gestionará los niveles de detalle.
- **Aumento de los requisitos de memoria**, al menos en la parte inferior de la jerarquía de memoria (normalmente HDD), ya que hay que almacenar no sólo el modelo origina sino también los distintos niveles de detalle. Por ejemplo, en un caso de modelo bidimensional, como los LODs de texturas mostrados en la figura anterior, se estima aproximadamente un 33% adicional de memoria para almacenar las distintas reducciones de la imagen.
- **Gestión de los niveles de detalle en tiempo de ejecución**, para escoger en cada momento el más adecuado para la resolución del problema. Esto implica crear y mantener una estructura de datos de tipo espacial, a la que se pueda consultar en cada momento qué nivel debe escoger el algoritmo.

El primer punto es prácticamente insalvable, pero al producirse en un paso previo al tiempo de ejecución no resulta un grave problema. Se podría decir que es una etapa previa de preparación de los datos, y que normalmente basta con ser ejecutada una sola vez. De la misma forma, el segundo punto no supone un grave obstáculo, ya que normalmente la capacidad de almacenamiento estática (HDD) supera en varios órdenes de magnitud a la memoria del sistema (RAM), y por supuesto, a la memoria gráfica (VRAM). El tercer punto es quizá la clave de estos sistemas, y uno de los campos más importantes del presente trabajo; se introduce en la siguiente sección.

2.3 ESTRUCTURAS DE DATOS ESPACIALES

Cuando se trabaja con datos que guardan una relación espacial, lo habitual es que el tipo de algoritmos que se quiera ejecutar sobre ellos tenga también carácter espacial. Ésta es la hipótesis detrás de todo el desarrollo que existe sobre estructuras de datos espaciales. Este tipo de estructuras pre-procesa un conjunto de datos atendiendo a sus posiciones relativas y absolutas en el espacio, para generar una "meta-información" que permita accesos muy rápidos y facilite ciertas operaciones de álgebra espacial.

La estructura más comúnmente utilizada con este fin es la de árbol, cuya organización y jerarquía de nodos torna muy natural, subdividiendo recursivamente el espacio que ocupan cada conjunto de datos. En la **¡Error! No se encuentra el origen de la referencia.** se puede observar este hecho, con el tipo de árbol más sencillo, denominado *octree*: partiendo del volumen total que ocupa la información (denominado *bounding box*), éste se puede dividir en ocho sub-volúmenes diferentes, que serán los nodos hijos del nodo raíz en el árbol. Cada uno de ellos contendrá su propia *bounding box*, que podrá ser subdividida de nuevo. El criterio para continuar o no la subdivisión en cada nuevo nodo recae en la cantidad de información del conjunto de datos que está contenida en su propio volumen.

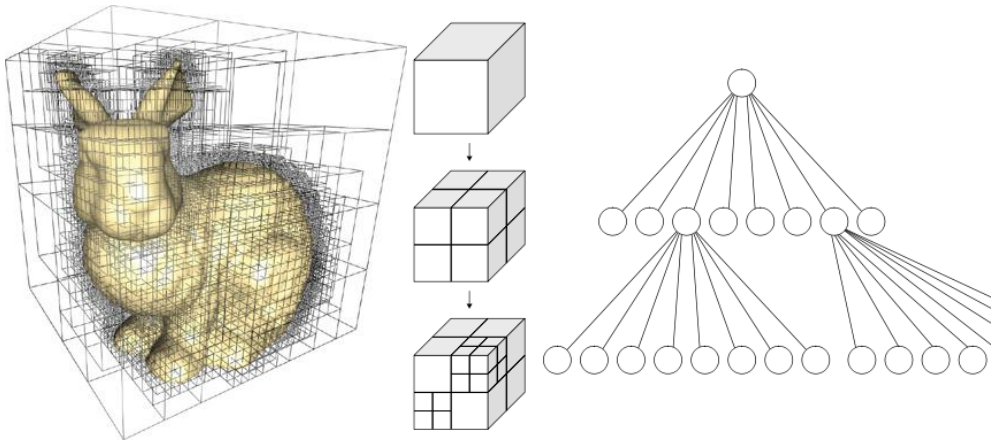


Ilustración 7. Subdivisión espacial recursiva para generar una estructura tipo árbol, en este caso, de tipo octree (Fuente: GPU Gems 2 y Wikipedia)

Sin una estructura de datos espacial, una operación tan simple como el acceso a los datos albergados en una determinada región del espacio tendría una complejidad computacional dada por la cota superior asintótica $O(n)$, es decir lineal, ya que habría que recorrer todos los datos comparando si caen o no en la zona solicitada. Sin embargo, en una estructura sencilla como la representada en la figura anterior, la complejidad se vuelve $O(1)$, es decir constante: un simple algoritmo recursivo como el siguiente alcanzaría inmediatamente la información.

```
FUNCTION compruebaSubNodo(nodo i, region) : datos
      IF ( contieneRegion(i, nodo, region) )
              IF ( esNodoHoja(i) ) return datos(i)
              ELSE FOR EACH j IN hijos(i) compruebaSubNodo(j, region)
END

// Llamada recursiva
datos = compruebaSubNodo(0, region)
```

Como se puede intuir, los algoritmos que se aplican sobre este tipo de estructuras son normalmente de naturaleza recursiva, y la acción recorrerlos jerárquicamente para acceder a los datos se denomina "*traversal*".

Existen numerosas estructuras jerárquicas además de los octrees, como las rejillas regulares, los árboles BSP, las jerarquías de bounding spheres, las jerarquías de geometría sólida (CSG), etc. en función del tipo de datos espaciales a albergar (puntos, polígonos, superficies, volúmenes, etc.) y del tipo de *traversals* que se quieran realizar. Para una información rápida y visual acerca de estas estructuras, se recomienda el curso de *Frank Pfenning*, de la *Carnegie Mellon University* [7], y para profundizar más en el campo, el artículo de *Hanan Sarnet* de la *University of Maryland* [8].

El presente trabajo se basa en una de estas estructuras, especialmente pensada para datos asociados a simples coordenadas en el espacio (puntos), denominada árbol K-Dimensional o ***kdtree***. En el siguiente capítulo se estudiará más en detalle.

2.4 JERARQUÍAS DE MEMORIA CON GPUS

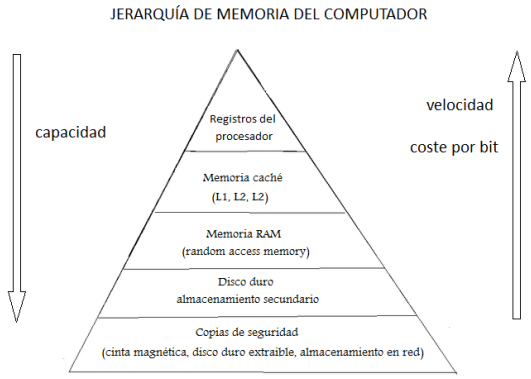


Ilustración 8. Jerarquía de memoria clásica.

Se denomina "jerarquía de memoria" de un sistema computerizado a la organización piramidal de la memoria en niveles, cuyo objetivo es conseguir el rendimiento de una memoria de gran velocidad al coste de una de baja velocidad, basándose en el principio de cercanía de referencias. Éste se refiere al agrupamiento de las lecturas de memoria por la CPU en direcciones relativamente cercanas entre sí.

Clásicamente se representa como en la Ilustración 8, donde se puede observar que, según se sube en la pirámide, el coste y la velocidad aumentan, mientras disminuye la capacidad. El principio de localidad temporal y espacial indica que, al subir un dato y sus vecinos en la jerarquía, estos serán requeridos de nuevo en un futuro próximo.

La evolución en los últimos años de los núcleos de las tarjetas gráficas (GPUs) y la aparición de nuevos lenguajes como CUDA u OpenCL ha generado un campo nuevo, denominado GPGPU (General Purpose GPU computing, o computación en GPU para propósito general), acercando enormemente el concepto que se conoce como Computación de Altas Prestaciones a los PCs de escritorio, e incluso a los equipos portátiles. Es por eso que el modelo clásico de jerarquía de memoria está mutando, para incorporar un nuevo nivel, de alguna forma solapado parcialmente con la CPU, y cuyo bus de acceso (en este momento) son las bandas PCI-e de la placa base, que permiten comunicar la CPU con la GPU. Este modelo "revisado"[1] se puede ver esquemáticamente en la Ilustración 9.

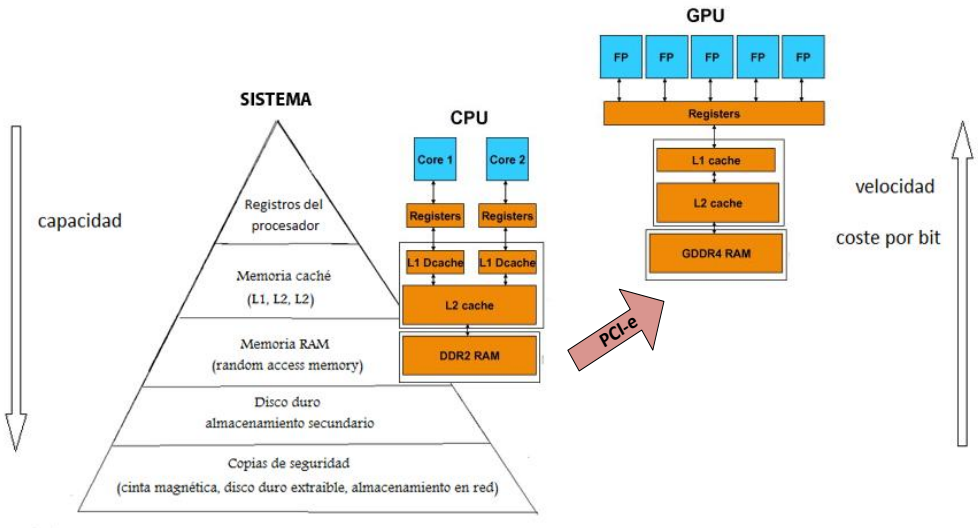


Ilustración 9. Modelo "revisado" de jerarquía de memoria en maquinas equipadas con GPU.

[1] Tómese este modelo extendido como una simple licencia del autor, con fines explicativos.

[2] El *blocking* aplicado a este tipo de datos cobra un nuevo sentido, ya que la división por bloques de los

En este modelo, cualquier dato susceptible de ser tratado en GPU debe ascender hasta la memoria del sistema, para después escoger un camino alternativo, y ser redireccionado a la memoria VRAM de la tarjeta de vídeo, donde seguirá su curso por las distintas cachés hardware de la GPU.

Si bien es cierto que lenguajes como CUDA han puesto a disposición de una gran masa de desarrolladores un enorme potencial computacional a bajo coste, también lo es que en muchas ocasiones han desvirtuado el propósito natural de las GPUs, que no es otro que la computación gráfica. Es por esto que muchos de los conceptos utilizados por estos lenguajes de alto nivel no son más que "fachadas" de conceptos o entidades propias de gráficos. Cuando un *toolkit* como CUDA hace referencia a subir un *array* de datos arbitrario a la memoria general de la GPU, lo que realmente genera es un ***Buffer Object***, propio del API gráfico **OpenGL**, que en este caso probablemente sea de uno de estos dos tipos:

- **Pixel Buffer Object (PBO)**: buffer ideado para almacenar imágenes, como conjuntos de píxeles.
- **Vertex Buffer Object (VBO)**: buffer que mantiene arrays de vértices y sus características (normal, color).

Cualquiera de estos dos *Buffer Objects* puede subirse a la memoria de la GPU (VRAM) de forma asíncrona, y dependiendo de la forma en que se configure, pueden incluso eliminarse de la memoria del sistema (RAM) mientras la tarjeta trabaja con ellos. Para ello, cada Buffer Object mantiene un identificador único que, una vez estén los datos en VRAM, permite a los lenguajes de alto nivel (CUDA, OpenCL) reconocerlos y trabajar con ellos. Además, los *Buffer Objects* permiten la asociación de información arbitraria. Estas características los convertirán en una parte fundamental del presente trabajo.

2.5 CACHÉS SOFTWARE

Por definición, una caché es cualquier tipo de componente (ya sea hardware o software) que, de forma transparente, almacene datos para que **las peticiones futuras de esos datos puedan ser atendidos rápidamente**. Existen gran variedad de sistemas caché, cuyas técnicas están muy estudiadas. Pero normalmente estas técnicas se centran en la casuística que pueda haber en un componente hardware particular, y para mantener la generalidad, no tienen en cuenta ningún tipo de relación entre los datos que manejan, más allá de sus posiciones relativas en los bancos de memoria física (como indica el ya mencionado principio de cercanía de referencias).

Pongamos el siguiente ejemplo: disponemos de un algoritmo que requiere ser alimentado por datos en un nivel de la jerarquía de memoria, datos que se encuentran en el nivel inferior. Pero por algún motivo arbitrario, a estos datos no se accede de forma secuencial, si no mediante la sucesión de Fibonacci, es decir, de esta forma: *datos[1], datos[1], datos[2], datos[3], datos[5], datos[8], datos[13], datos[21]*, etc. El principio de localidad temporal funcionará con el primero (se accede dos veces consecutivamente) y el de localidad espacial, con los primeros también (probablemente los cuatro o cinco primeros caigan en el mismo bloque). Pero de ahí en adelante, el funcionamiento de la caché, es decir, el cociente entre aciertos y peticiones, disminuirá estrepitosamente.

Cuando de antemano se conoce cuál es la relación fundamental entre los datos, y cuál será la forma de acceso a los mismos, entran en juego las cachés software. **Un problema suficientemente complejo, con un nivel de acceso a datos muy elevado, justifica diseñar un sistema ad-hoc de cachés entre los niveles de la jerarquía de memoria a los que tenga acceso el usuario.**

Como se verá en próximos capítulos, en este trabajo se ha diseñado e implementado un sistema de caché a dos niveles, que relaciona los niveles de memoria HDD – RAM – VRAM, teniendo en cuenta la relación espacial de los datos con los que trabaja, y cuya arquitectura permite además mantener la información del dataset no sólo en HDD, sino en cualquier otro sistema de almacenamiento que tenga un sistema de archivos, como por ejemplo, NFS vía una red de comunicaciones.

3 ESTADO DEL ARTE

En este capítulo se realiza un breve resumen de las técnicas en proceso de investigación y de los sistemas comerciales más relevantes en el momento actual, que utilicen arquitecturas multirresolución para el tratamiento y/o visualización de nubes de puntos.

3.1 TÉCNICAS EN INVESTIGACIÓN

Existen numerosos artículos de investigación sobre procesado de nubes de puntos. Pero normalmente se centran en un problema particular (clasificación, reconstrucción, detección de objetos, etc.) pasando por encima el problema de la aplicación sus algoritmos sobre grandes cantidades de datos. O bien presuponen que el *dataset* cabrá en memoria del sistema, o bien su método es suficientemente "poco complejo" ($< O(n)$) para permitirse realizar unas pocas pasadas por los datos en disco.

Inicialmente, se debe destacar el "***International Workshop on Point Cloud Processing***", organizado por el **INRIA** (*Institut National de Recherche en Informatique et en Automatique*, Francia) que dedica unas jornadas anuales al fomento de nuevas líneas interdisciplinares en los campos de Visión Artificial, Gráficos por Computador, Procesado Geométrico, Robótica y Fotogrametría.

Existen sin embargo algunas publicaciones que se acercan a este tema, sobre todo con el gran auge de artículos dedicados a la explotación de la GPGPU (*General Purpose GPU computing*) en diversos entornos. Podemos destacar el "*GPU-based cloud performance for LIDAR data processing*" de *R. Sugumaran* et al. (Universidad de Iowa Norte) [9], donde realizan una comparación de rendimiento entre algortimos paralelizados en CPU y en GPU con CUDA con una infraestructura *cloud*, o el "*Parallel processing of massive LiDAR point clouds using CUDA enabled GPU*" de *F. Qiu y C. Yuan* (Universidad de Texas) [10], donde dan algunas técnicas para el procesado paralelo basadas en *blocking* de estos *datasets*.

Es, sin embargo, en el campo de los **Computer Graphics** donde más inspiración ha encontrado este trabajo. Mientras que el campo de GPGPU es de relativa nueva creación, y la proliferación de lenguajes de alto nivel como CUDA y OpenCL permiten a investigadores de todos los ámbitos explorar las capacidades computacionales de las GPUs, los Gráficos en Computación llevan un recorrido mucho más amplio investigando estructuras de aceleración espaciales y multirresolución para resolver los problemas que plantea el render o visualización de grandes cantidades de datos, puntos en este caso [11].

En este sentido, y por su mayor influencia en el diseño del sistema planteado en este trabajo, se presentan los siguientes tres artículos y tesis doctoral. Por una parte, la "***QSplat: a multiresolution point rendering system for large meshes***", diseñada por *Szymon Rusinkiewicz* y *Marc Levoy* fue una revolución en el año 2000 [12]. Aparte del sistema de render, presentaban una estructura de datos basada en una jerarquía espacial de volúmenes esféricos que contenían recursivamente cantidades más pequeñas de puntos, que permitían una gran aceleración a la hora de buscar aquellos que se necesitaban visualizar en cada *frame*.

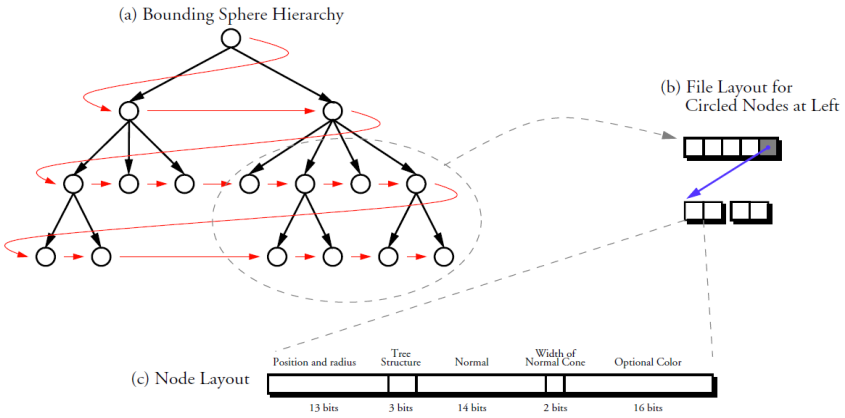


Ilustración 10. Estructura espacial de bounding-spheres del sistema QSplat [12].

Unos años más adelante, en el 2003, *Enrico Gobbetti* y *Fabio Marton* (ViC – CRS4, Italia) presentan la técnica de "***Layered Point Cloud***" [13], cuya estructura de datos espacial basada en *octrees* tiene como novedad que los niveles de detalle son aditivos, y que la estructura de datos guarda información en todos los nodos del árbol, no sólo en las hojas. Esta técnica, aunque modificada, es la que se utiliza en el sistema presentado, así que se hablará más en profundidad de ella en los siguientes capítulos.

A día de hoy, se podría decir que el *state-of-the-art* en estructuras espaciales multirresolución para nubes de puntos lo define la publicación "***An Efficient Multi-resolution Framework for High Quality Interactive Rendering of Massive Point Clouds using Multi-way kd-Trees***" de *P. Goswami, F. Erol, E. Pajarola* (EZH, Suiza) y *E. Gobbetti* (CRS4, Italia) [14] del año 2012. En ella utilizan una nueva estructura, denominada *multiway kd-tree*, que simplifica enormemente la gestión de memoria mediante la construcción de un árbol completamente balanceado y con nodos homogéneos. Esta idea también ha sido recogida en este trabajo.

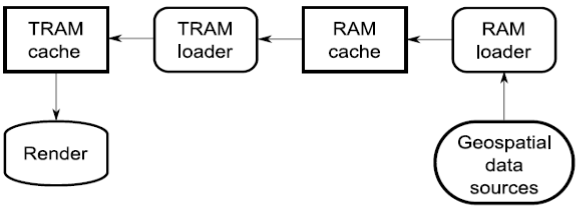


Figure 1: Overall architecture.

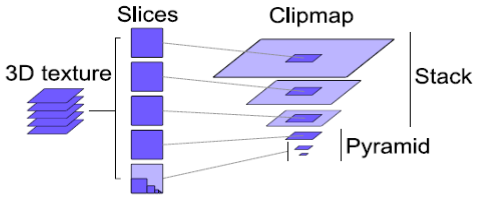


Figure 2: Clipmap structure on a 3D texture.

Ilustración 11. Sistema de caché software a dos niveles de J. Taibo y *dataset* piramidal de texturas.

Por otra parte, se destaca la tesis doctoral "***GeoTextura: Una arquitectura software para la visualización en tiempo real de información bidimensional dinámica georreferenciada sobre modelos digitales 3D de terreno basada en una técnica de mapeado de texturas virtuales***" de uno de los directores de este trabajo, *J. Taibo* (UDC, España), y presentada en el año 2010 [15]. A pesar de trabajar en un dominio completamente distinto al de las nubes de puntos (el de las texturas *georreferenciadas*), introduce el concepto de caché software a varios niveles entre HDD, RAM y VRAM para la visualización de grandes

cantidades de datos. En la Ilustración 11 se puede observar el diagrama representativo de su sistema, muy similar al utilizado en este trabajo, así como una representación del *dataset* de texturas *georrefe-renciadas*.

Como se demostrará más adelante, este tipo de estructuras y técnicas, utilizadas en el campo de los Gráficos por Computador, serán adaptadas en el sistema presentado para ofrecer una solución genérica, no sólo de visualización, sino también de procesado de grandes *datasets* de nubes de puntos.

3.2 SISTEMAS COMERCIALES

En general, se puede afirmar que los sistemas software comerciales proporcionados por los fabricantes de escáneres 3D y equipos LiDAR, implementan (cuando lo hacen) sistemas multirresolución muy sim-ples, además de ser arquitecturas cerradas que no admiten la programación de nuevos algoritmos de procesado. Sin embargo, están surgiendo algunas empresas que tratan de llevar el ámbito de la investi-gación a su uso comercial, con software innovador que si implementa tecnologías más avanzadas:

- Pointools de POINTOOLS Ltd
- Gexcel R^3 de GEXCEL
- Dielmo software de DIELMO
- MARS de MERRICK & COMPANY

Existen además una serie de iniciativas *open-source* para el tratamiento y visualización de nubes de puntos, como son las siguientes:

- MeshLab
- DielmoOpenLiDAR
- FullAnalyze
- Cloud Compare

Una mención aparte merece la **Point Cloud Library** (**PCL**) [16]. Esta librería, desarrollada en C++ bajo la licencia *open-source* BSD, es un proyecto de largo recorrido para el tratamiento de nubes de puntos, que contiene una extensa colección de algoritmos y tecnologías relacionadas con este campo, divididas en distintos módulos. Como se puede observar en la **¡Error! No se encuentra el origen de la referencia.**, esta librería trabaja también con estructuras espaciales como octrees y kd-trees, como base para la computación de sus algoritmos. Sin embargo, no provee de un API para trabajar con una jerarquía de memoria sencilla, quedando en manos del desarrollador su gestión. La inmensa cantidad de colaboradores y *partners* es una prueba de que este campo está en pleno auge, y en un estado de evolución aún muy temprano.

4 ANÁLISIS DE LA SOLUCIÓN PROPUESTA

En este capítulo se tratará de ofrecer una visión en conjunto de la solución planteada para cumplir los requisitos exigidos, así como de las distintas partes relevantes que la componen, a nivel meramente lógico y funcional. Los detalles de software (paquetes, diseño, implementación, etc.) serán expuestos en los siguientes capítulos.

4.1 REQUISITOS DEL SISTEMA

Se parte inicialmente de que los usuarios disponen de una nube de puntos, que contiene un número arbitrario de puntos o muestras, cada uno de ellos con una cierta cantidad de datos adjunta. Sobre ella se desean los siguientes requisitos de alto nivel:

- **La información debe ser accesible de forma rápida, independientemente del tamaño del dataset o de las características de la máquina**.
- Se podrán ejecutar **procesos de carácter espacial**, que permitan trabajar sobre distintas versiones del modelo, con mayor o menor detalle, ya bien sea mediante progresivos refinamientos o solicitando un nivel de calidad particular.
- Los procesos podrán ejecutarse de forma clásica en **CPU**, o bien aprovechar la potencia de la **GPU** del sistema.
- El usuario-desarrollador podrá programar procesos que actúen sobre el dataset completo de una forma **sencilla e intuitiva**, siendo transparente la forma del sistema de manejar los datos.

4.2 VISIÓN GENERAL DE LA SOLUCIÓN

El funcionamiento del sistema ha sido inspirado por las ideas sugeridas en algunos de los artículos de investigación comentados anteriormente, especialmente aquellos de *E. Gobbetti* (CRS4, Italia), *R. Pajarola* (UZH, Suiza) y *M. Gross* (ETHZ, Suiza) así como la tesis doctoral de uno de los directores de este trabajo, *J. Taibo* [15] (UDC, España). En realidad, la mayor parte de estas fuentes hacen referencia a técnicas propias de *render*, para la visualización de grandes *datasets*. En este trabajo se ha tratado de abstraer este tipo de estructuras de datos y algoritmos de forma que el sistema resultante sirva como una plataforma general de gestión de nubes de puntos, en el que se puedan lanzar procesos de carácter espacial, siendo el proceso de *render* o visualización uno más de ellos.

Dado que el mayor problema reside en hallar la forma de procesar una gran cantidad de datos en tiempo real, que tienen además una estructura heterogénea, y que deben estar cargados lo más rápido posible bajo demanda en memoria de sistema (RAM) o memoria de GPU (VRAM), los puntos clave de la solución se encontrarán en diseñar una correcta estructura de datos y una jerarquía de memoria que permita mover pequeñas porciones de estos datos entre todos los niveles de memoria (HDD – RAM – VRAM). Como el tipo de algoritmos que se quieren ejecutar para procesar estos datos son de carácter

espacial, este binomio estructura de datos – jerarquía de memoria debe ser diseñada siguiendo principios propiamente espaciales.

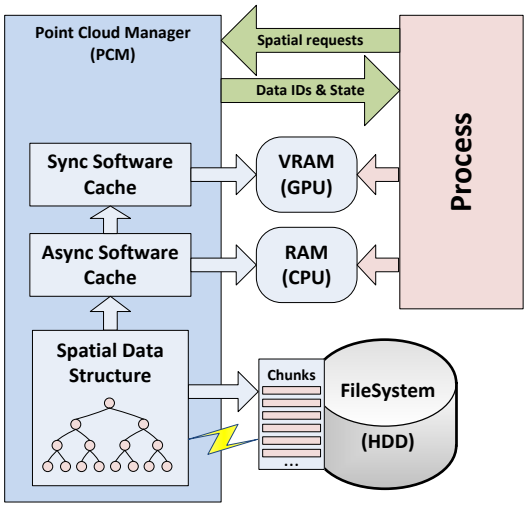


Ilustración 12. Diagrama abstracto de la solución propuesta.

En la Ilustración 12 se puede observar un diagrama resumido de la solución propuesta. El sistema desarrollado mantiene una **estructura de datos espacial**, especialmente pensada para la gestión de puntos, y cuyos nodos referencian pequeños subconjuntos del *dataset* original que residen en disco, que se denominarán ***data chunks***. Estas pequeñas unidades de información viajarán por un sistema de cachés software, compuesto por:

- **Caché asíncrona de 2º nivel** (en adelante, **L2**): responsable de consultar a la estructura de datos espacial para cargar de disco y mantener en memoria RAM un conjunto de chunks, normalmente de la zona del espacio en la que se esté trabajando. Tiene un funcionamiento asíncrono, de modo que, independientemente del proceso del sistema o de algoritmo en ejecutado, se mantendrá en un segundo plano cargando datos en RAM, mientras tenga peticiones pendientes.
- **Caché síncrona de 1º nivel** (en adelante, **L1**): en caso de que el proceso solicitado lo requiera, se encargará bajo demanda de cargar chunks desde la memoria RAM (aquellos previamente subidos por la caché L2) y componer y subir a VRAM *BufferObjects*, *arrays* de memoria especiales con los que puede trabajar la GPU.

El proceso que requiera realizar cualquier gestión sobre la nube de puntos, deberá únicamente solicitar en qué región del espacio va a trabajar. Tiene dos formas diferentes de hacerlo:

- **Restringido por tiempo**: se solicita que se cargue una región de la nube en un plazo de tiempo máximo. El sistema garantiza volver de la llamada en un tiempo menor o igual al solicitado y, dependiendo de las características del sistema, se habrá conseguido cargar una mayor o menor cantidad de datos de la zona requerida. Este método es sumamente útil para algoritmos de re-

finamiento que trabajen en tiempo real, como por ejemplo el propio *render* de la nube de puntos.

- **Restringido por nivel**: se solicita que se cargue una región de la nube hasta un nivel de detalle determinado. En este caso, el sistema volverá de la llamada o bien cuando se haya llegado a dicho nivel, o bien cuando se haya alcanzado la memoria máxima permitida, y no se pueda cargar más. Este otro método está diseñado para los procesos de "fuerza bruta" que quieran realizar un cómputo sobre el dataset completo, utilizando la técnica de *blocking*[2].

El sistema irá informando al proceso de los identificadores de los buffers cargados en RAM y/o VRAM, para que pueda trabajar con ellos en su código, así como el estado general del sistema (niveles de detalla alcanzados en cada momento, estado de las cachés, etc.).

Este sistema cumpliría todos los requisitos impuestos anteriormente, e incluso permitiría alguna ventaja adicional, como puede ser la posibilidad de servir los datos desde un sistema de archivos en red, en vez del disco duro local de la máquina. Los chunks, como pequeños conjuntos de datos binarios, son inherentemente perfectos candidatos para conseguir bajas latencias en su envío bajo una red TCP/IP. Además, como no se utilizan directamente, si no que son recibidos y gestionados por la caché L2, el sistema sería tolerante a fallos de comunicación.

En las siguientes secciones se describe como funcionan cada una de las partes del sistema en detalle.

4.3 ESTRUCTURA DE DATOS ESPACIAL MULTIRRESOLUCIÓN

Como se vio en el capítulo que repasaba el estado del arte, la mayor parte de los sistemas estudiados utilizan algún tipo de estructura jerárquica, normalmente árboles. En este proyecto, la estructura escogida se denomina **árbol K-dimensional** o ***kd-tree***. Cada uno de los nodos de este árbol, referenciará en el espacio una pequeña célula de información denominada ***data chunk***. La peculiaridad de este sistema, como se verá a continuación, es que la suma de los chunks de todos los nodos del árbol genera el *dataset* exacto, a pesar de que cada nivel del árbol representa la nube de puntos completa. Esto se logra mediante un proceso estadístico en la generación de la estructura.

4.3.1 KD-TREE

Un **árbol K-dimensional o kd-tree** es una **estructura de particionamiento espacial**, capaz de organizar puntos en un espacio de k dimensiones. Su construcción es muy sencilla, y puede verse reflejada en la Ilustración 13: cada nodo divide el espacio en dos mitades, de forma que dejen una agrupación de puntos a cada lado. El eje de particionado puede irse alternando, o escoger siempre el más corto, para un mejor balanceo (en ese caso, en cada nodo debe guardarse qué eje se utilizó). El proceso se repite re-

[2] El *blocking* aplicado a este tipo de datos cobra un nuevo sentido, ya que la división por bloques de los datos es, en realidad, como dividir en una rejilla las zonas del espacio, y trabajar sobre cada una de sus celdas.

cursivamente hasta la condición de parada, en este caso, que el número de puntos que quede bajo uno de los subespacios sea menor que un umbral dado, que se denomina **M**. Como se verá, este parámetro es, probablemente, el más importante e influyente en el funcionamiento del sistema.

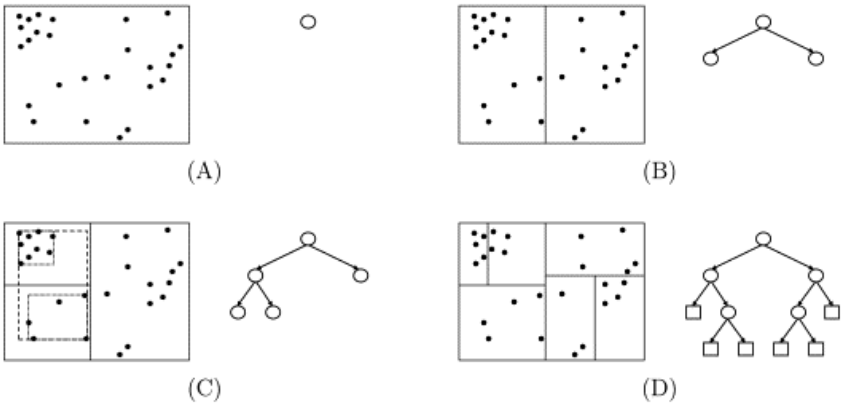


Ilustración 13. Construcción de un kd-tree. (A) El nodo raiz contiene el volumen completo. (B) El espacio original se particiona en dos mitades por el eje más largo. (C) y (D) el proceso se repite recursivamente hasta que el número de puntos de un nodo sea menor que un parámetro M dado, convirtiéndose en una hoja (Fuente: sciencedirect.com)

La operación de *traversal*, es decir, el recorrido del árbol en la búsqueda de un punto o región del espacio dada, puede verse en la Ilustración 14. Simplemente se debe escoger en cada nodo del árbol el camino que quede en el lado adecuado de su plano de corte.

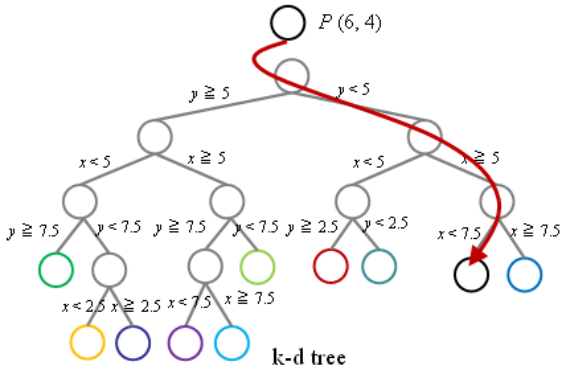


Ilustración 14. *Traversal* de un kd-tree (Fuente: frytail.seesaa.net)

La solución planteada implementa un kd-tree con una sutil variación, que será muy útil a la hora de gestionar una cantidad muy alta de nodos y niveles. Como se verá a continuación, durante el proceso de construcción, **a la hora de escoger el plano de corte de cada nodo, se buscará la mediana de las coordenadas de los puntos en dicho eje, para garantizar que el árbol generado sea, además de binario completo, perfecto**[3]. Esto, a nivel computacional, genera una ventaja sustancial: los nodos se pueden identificar secuencialmente, desde la raíz, que tendrá el número cero, hasta la última hoja (Ilustración 15).

[3] Se denomina árbol perfecto a aquel árbol completo en el cual, todas las hojas están en el mismo nivel. Se podría decir que, visualmente, semejaría una pirámide perfecta.

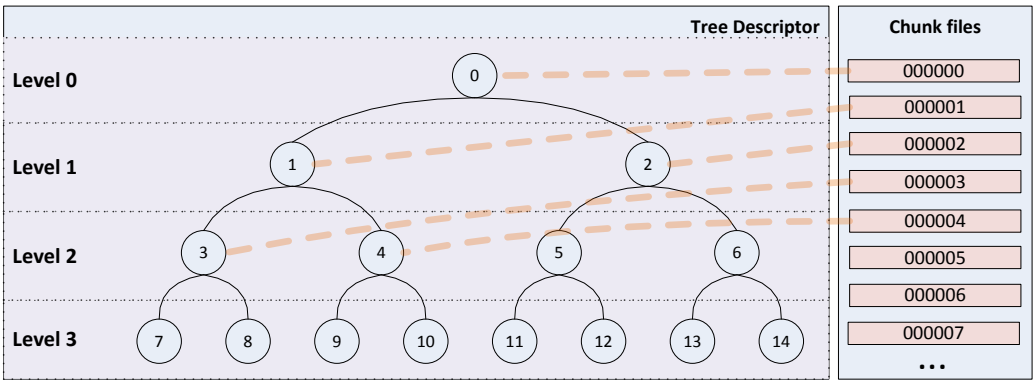


Ilustración 15. Árbol binario perfecto y su correspondencia lineal con los *chunks* de datos.

De esta forma, los nodos pueden almacenarse en *arrays* lineales, cuyo tamaño será siempre conocido y cuyos accesos a disco o memoria no deberán ser comprobados (siempre "acertarán), siguiendo las siguientes fórmulas:

$$\boldsymbol{Dado\ el\ nodo\ i\ y\ el\ nivel\ n\ \ldots}$$

$$Padre(i) = \frac{i-1}{2} \qquad N^{\underline{o}}\ Nodos\ Acumulados(n) = 2^{n+1} - 1$$

$$Hijo\ Izquierdo(i) = 2i + 1 \qquad N^{\underline{o}}\ Nodos\ Nivel(n) = 2^{n}$$

$$Hijo\ Derecho(i) = 2i + 2 \qquad Primer\ Nodo\ del\ Nivel(n) = 2^{n} - 1$$

$$Nivel(i) = log_2(i) + 1$$

4.3.2 ORGANIZACIÓN DE LOS DATOS Y NIVELES DE DETALLE

En el capítulo dedicado a fundamentos se explicó el funcionamiento habitual de los niveles de detalle, y se comentó cómo a partir del modelo original se creaban sucesivas simplificaciones hasta llegar a la más trivial. Esto implicaría el almacenamiento y gestión, además de la versión inicial, otras N versiones, tantos como niveles de detalle se deseen, que generarán necesariamente un considerable aumento en los requisitos de memoria, en varios niveles de la jerarquía.

Sin embargo, el caso tratado por este problema es singular en ese aspecto: una nube de puntos no suele ser más que un conjunto de muestras de la superficie de uno o varios objetos de un entorno. Partiendo de esta base, *Enrico Gobbetti* y *Fabio Marton*, en su artículo "***Layered Point Clouds***" [13], propusieron un sistema alternativo, pensando exclusivamente en la visualización y *render* del *dataset*. Ellos apuntan que, con un buen método estadístico para escoger puntos, los niveles de detalle pueden ser *aditivos*. Esto es, partiendo de un subconjunto del total de puntos bien distribuidos por todo el modelo (primer nivel de detalle) cada vez que se le añada otro subconjunto de puntos, se obtendrá un nuevo nivel de detalle, más refinado (con el doble de puntos). Esta idea, extrapolada al kd-tree visto anteriormente, se comprende perfectamente en visionando la Ilustración 16.

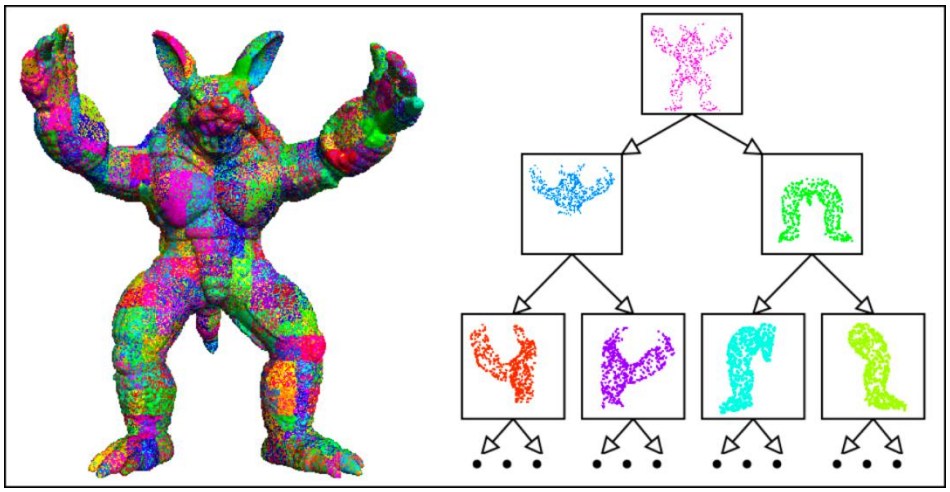


Ilustración 16. Distribución de los datos para generar niveles de detalle *aditivos,* siguiendo el esquema propuesto por E. Gobbetti y F. Marton en [17].

Normalmente, las estructuras espaciales almacenan datos exclusivamente en los nodos hoja, y el resto de nodos del árbol sirve únicamente para contener información espacial que ayude a los algoritmos de *traversal* a recorrer el árbol en las búsquedas. Sin embargo, **la estructura propuesta almacena un subconjunto de los datos en todos y cada uno de los nodos del árbol**. De esta forma, **la suma de los datos de todos los nodos conforma la nube de puntos completa**. Cada nodo hace entonces de "contenedor" de un pequeño subconjunto de datos, denominado ***data chunk*** (Ilustración 17).

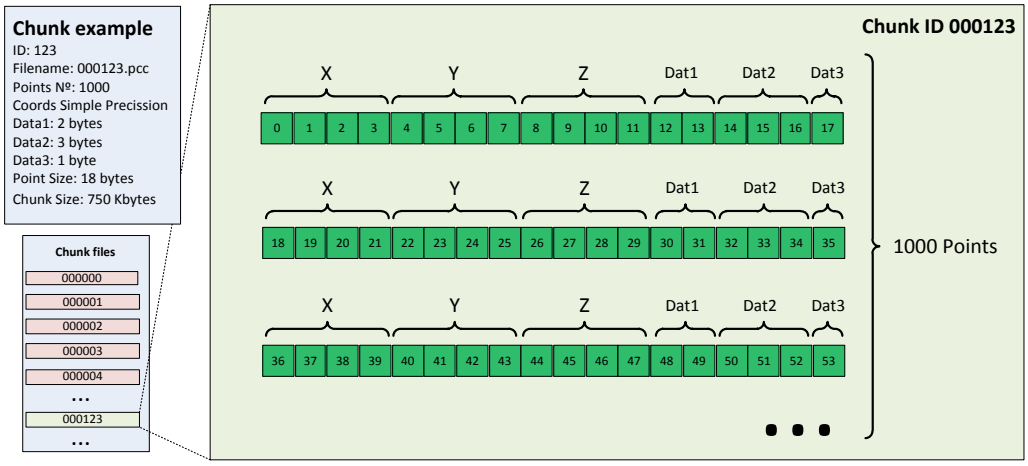


Ilustración 17. Estructura de un *data chunk*.

Esta idea es idónea para el tratamiento de grandes *dataset*, porque asegura que el tamaño en memoria de la BBDD generada es prácticamente el mismo que el del propio *dataset*, y que siempre se trabaja con datos reales, no con aproximaciones calculadas. Aunque, por otra parte, esto introduce a su vez una fuerte dependencia con la distribución original de los puntos, ya que influirá directamente en la forma de los niveles de detalle. En la práctica, esto no tiene por qué ser un gran problema, siempre que la construcción del árbol contemple un buen método estadístico o pseudo-aleatorio para escoger los puntos de cada nivel.

La selección del nivel de detalle dependerá del modo de *traversal* escogido. Cuando el proceso que trabaje con la nube de puntos defina una región sobre la cual quiere operar, se escogerá aquel nodo de nivel más bajo cuyo volumen encierre dicha región. Este nodo se definirá como el nodo base para la operación de *traversal*, y a partir del cual se generará una lista de todos aquellos nodos que contengan puntos de la región solicitada. Como veremos más adelante, el orden de esta lista establece las prioridades de carga en la jerarquía de memoria, así que, dependiendo de los propósitos del algoritmo, se han desarrollado cuatro órdenes diferentes, que se aprecian en la siguiente figura.

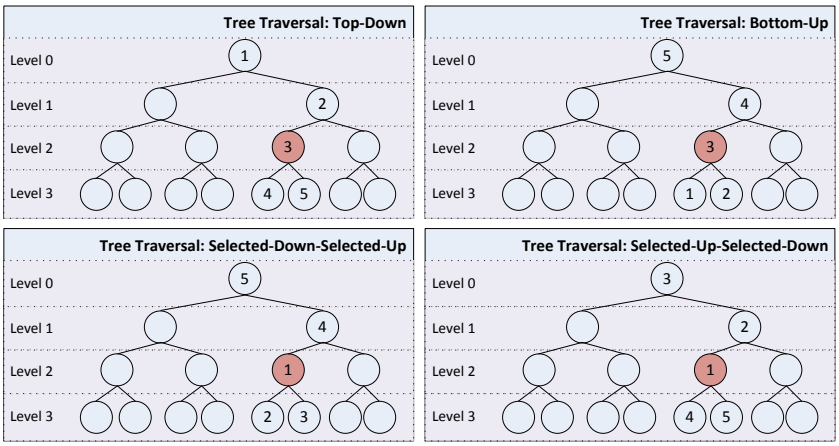


Ilustración 18. Modos de traversal definidos en el árbol.

4.3.3 CONSTRUCCIÓN DE LA ESTRUCTURA

Se define la estructura de datos como una colección de ficheros de datos (los *chunks*) junto con una descripción del *kd-tree* generado. Como se vio en anteriormente (Ilustración 15), el nombre de fichero de cada chunk corresponde con su identificador en el árbol. Su construcción es un proceso previo al uso del sistema, que sólo necesita ser ejecutado una vez, quedando la estructura guardada en disco.

Según lo visto en la anterior sección, **el proceso de construcción puede verse como una "simple" reordenación espacial y estadística de los datos, en los nodos del árbol que les correspondan**. El único parámetro que debe escoger el usuario es el denominado $\boldsymbol{M}$, que hace referencia al **número máximo de puntos que almacenará un nodo**. De alguna forma, podría decirse que *M* define la granularidad de la estructura de datos. De este umbral depende en gran medida el número de niveles y nodos del árbol, así como en el tamaño en bytes de los chunks. Por lo tanto, su influencia sobre el sistema es considerable:

- Los chunks son la célula básica de datos que recorre la jerarquía de memoria. Su tamaño influye directamente en el rendimiento de las trasferencias de memoria.
- El número de niveles del árbol es, a su vez, el número de niveles de detalle de la estructura. Cuanto mayor sea, más granularidad y menor salto de calidad entre niveles, pero a su vez mayor será el coste de gestión del árbol.

Es por esto que la selección del umbral M requiere cierta experiencia o consejo al usuario. En el capítulo dedicado a los futuros desarrollos se bocetarán ideas para su selección automática.

Por otra parte, se debe tener en cuenta en todo momento que el sistema debe trabajar con grandes cantidades de datos, y que, desde luego, el dataset original normalmente no cabrá en la memoria del sistema. Esto provoca dos grandes problemas que resolver:

- Cualquier operación de complejidad superior a $O(n)$ sobre el conjunto completo de los datos se vuelve extremadamente costosa y "engorrosa" (habría que realizar cargas y descargas de entre disco y RAM), así que deben seleccionarse cuidadosamente los métodos en la construcción del árbol, tanto para escoger los planos de corte, como para elegir los puntos que contendrá el nodo.
- El uso de ficheros temporales no puede ser exhaustivo, ya que no se puede presuponer un espacio en disco ilimitado.

El problema de acceso a los datos ha sido solucionado mediante una clase interfaz, que de cara al desarrollador funciona de forma similar a un *array* de datos en memoria, pero accediendo sin embargo a disco de una forma óptima. Los detalles de implementación de esta clase se explicarán en su correspondiente capítulo.

Para resolver el resto de las dificultades, se ha ideado el siguiente algoritmo recursivo. Supongamos que el dataset es un fichero binario de grandes dimensiones, con puntos desordenados, del que podemos extraer algunos de ellos y escribirlos en chunk a disco. Al número de puntos que tenga ese dataset en todo momento lo llamaremos N. Recordemos, por último, que M es el número máximo de puntos por nodo.

```
ConstrucciónRecursiva(dataset, i)
      Seleccionar eje de corte
      Iterar todos sus elementos (N), haciendo:
            →Búsqueda de la mediana en el eje de corte: coord_corte
            →Selección de M puntos
      Escritura del chunk con su nombre (i)
      Creación de dos nuevos sub-datasets: izquierdo y derecho
      Iterar de nuevo los puntos restantes del dataset (N-M)
            IF (punto.coord(eje) < coord._corte)
                  mover punto al sub-dataset izquierdo
            ELSE
                  mover punto al sub-dataset derecho
      Borrar dataset // debería estar vacío
      GuardarNodo(i, eje_corte, coord_corte, M)
      IF (nº_puntos_sub-dataset_izquierdo > M)
            ConstrucciónRecursiva(dataset_izquierdo, 2i+1)
      IF (nº_puntos_sub-dataset_derecho > M)
            ConstrucciónRecursiva(dataset_derecho, 2i+2)
End

// Llamada de construcción
ConstrucciónRecursiva(dataset_original, 0)
```

Este algoritmo construye la BBDD de ficheros chunk recursivamente, con la ventaja de que el máximo espacio en disco ocupado en cada momento, S_{max} cumple siempre la inecuación

$$S_{max} \leq 2\, S_{ds}$$

Siendo S_{ds} el tamaño en bytes del *dataset*. Además, en cada llamada a la función recursiva, solamente se itera el dataset completo dos veces, cumpliendo la restricción de ejecutar operaciones de complejidad mayor a $O(n)$. Las dos llamadas remarcadas con el símbolo → son las más relevantes, y requieren una explicación aparte.

Al final del algoritmo, además, la estructura del árbol queda guardada en memoria RAM. Será un conjunto de nodos, que contendrán cada uno de ellos la siguiente información:

- Eje de corte: X, Y, Z ó ninguno, en caso de que sea hoja (1 byte)
- Coordenada del plano de corte (4 bytes)
- Número de puntos que almacena ese nodo, que deberá ser ≤ M (4 bytes)
- Bounding Box de los puntos que almacena, una información a mayores que será muy útil para hacer varios cálculos espaciales (24 bytes)

Es importante contabilizar el tamaño en memoria de la estructura, como "contenedora" de los datos que estarán en disco, ya que esta estructura estará permanentemente almacenada en RAM. Dado el tamaño del nodo, se puede aproximar el tamaño del árbol como

$$\frac{N_{pc}}{M} S_n$$

Donde N_{pc} es el número de puntos del *dataset*, y S_n el espacio ocupado por un nodo, en este caso 33 bytes. Por ejemplo, un árbol de un dataset de 100 millones de puntos, con $M = 5000$ ocuparía algo menos de 640 KBytes de memoria, tamaño que podemos considerar despreciable para las capacidades de memoria RAM de hoy en día[4].

4.3.3.1 BÚSQUEDA DEL PLANO DE CORTE: APROXIMACIÓN DE LA MEDIANA

La primera decisión importante en el proceso de construcción es la selección del plano de corte en cada nodo, mediante la búsqueda de la mediana entre las coordenadas del eje seleccionado en cada punto del *sub-dataset*. Como se dijo anteriormente, cualquier algoritmo se vuelve complejo trabajando con cantidades grandes de datos. Una primera aproximación de cálculo de la mediana en un conjunto de números podría consistir en ordenar éstos y seleccionar el intermedio. Pero hasta el más rápido de los algoritmos de ordenación, el *quicksort* tiene una complejidad entre $O(n \log n)$ y $O(n^2)$, además de una cantidad de operaciones de lectura/escritura aleatorias absolutamente prohibitivas para un sistema que trabaja con datos en disco, no en memoria.

[4] Y así lo predijo ya Bill Gates, CEO de Microsoft, a mediados de los años 80's: «640KB deberían ser suficientes para cualquiera…»

Tras no pocas pesquisas sobre algoritmos de mediana y pseudo-mediana, la solución se halló en una ingeniosa idea propuesta por un usuario del destacado foro de desarrollo *Stack Overflow* [18]. *Rex Kerr* propone un algoritmo de bajo nivel, capaz de aproximar (con gran acierto) la mediana de un conjunto enorme de datos desordenados, con una complejidad $O(n)$. A grandes rasgos, consiste en generar en una primera pasada un histograma de 2^{16} muestras (262 KB de memoria) de la primera mitad de dígitos más representativos de las coordenadas. Después, contar el número de muestras hasta el punto medio, y quedarse con el "marcador" del histograma que lo contiene. En una segunda y última pasada, se repite la construcción del histograma, pero descartando aquellos números cuya parte mitad más significativa no coincida con la seleccionada en la primera pasada. El punto medio de este segundo histograma será la aproximación de la normal.

Además de su increíble eficiencia, los resultados de esta aproximación se han mostrado sorprendentemente precisos, fallando apenas por unidades sobre colecciones de decenas de millones de puntos.

4.3.3.2 SELECCIÓN PSEUDO-ALEATORIA DE LOS PUNTOS

Dado que los niveles de detalle de la estructura de datos van a depender únicamente de los puntos seleccionados en cada nodo, es necesario idear un sistema poco complejo pero a la vez eficaz, que sea capaz de escoger puntos al azar, pero procurando una distribución razonablemente equitativa entre el sub-dataset completo. Este tipo de métodos de *sampling* suelen denominarse pseudo-aleatorios. En este caso, el escogido puede verse representado en la Ilustración 19.

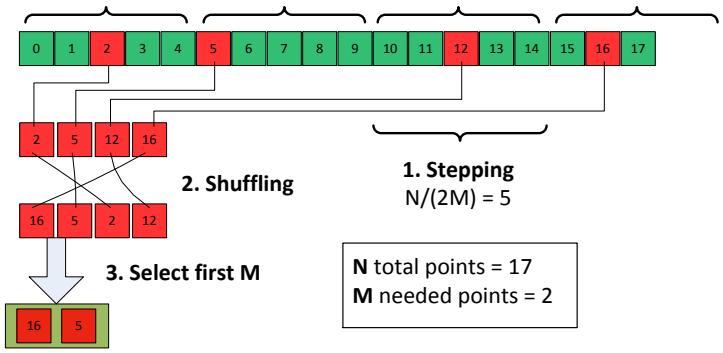


Ilustración 19. Método pseudo-aleatorio de selección de puntos para un nodo.

Aprovechando la iteración sobre los datos en la que se escogerá el plano de corte, cada $\frac{N}{2M} + X$ puntos, siendo $X\ aleatorio \in [0, \frac{M}{2})$, el algoritmo irá guardando una referencia de punto. Al final de la iteración, estas referencias se desordenarán, y se escogerán las M primeras para formar parte del nodo. Este sencillo algoritmo asegura una distribución aleatoria pero sin agrupaciones de puntos, evitando los patrones producidos por las distribuciones estadísticas fijas, y el ruido generado por las distribuciones completamente aleatorias.

4.4 JERARQUÍA DE MEMORIA

El segundo concepto importante del sistema que se presenta hace referencia a la gestión de la jerarquía de memoria, mediante un sistema de cachés software a dos niveles. Como se ha comentado en el capítulo dedicado a fundamentos, cuando se conoce de antemano una relación entre los datos con los que se trabaja, y el tipo de accesos que se van a realizar, la posibilidad de aumentar el rendimiento del sistema mediante la programación de cachés software ad-hoc es enorme. En este caso, existen claramente tres niveles de memoria a los que un desarrollador tiene acceso, y que serán los canales por donde irá pasando el flujo de unidades mínimas de información, los **chunks**. Estos son:

- **HDD**: la memoria de estado permanente, de alta capacidad pero baja velocidad, manejada por el sistema operativo mediante un sistema de archivos. Aquí residirá la base de datos completa de chunks, y el descriptor de la estructura espacial multirresolución (que será cargado en memoria principal al arranque del sistema). La abstracción proporcionada por el sistema de archivos permite que este nivel de la jerarquía esté en otra máquina, bajo una red de comunicaciones.
- **Memoria principal o RAM**: memoria de carácter volátil, relativamente baja capacidad (varios ordenes de magnitud menores que el anterior nivel) y alta velocidad. La mayor parte de algoritmos y aplicaciones trabajan con los datos en este nivel de memoria, mediante la CPU.
- **Memoria de vídeo o VRAM**: memoria volátil también, que suele bajar un orden de magnitud respecto a la RAM, y muy veloz, pero con la que sólo puede trabajar la GPU.

Partamos entonces de la siguiente figura, que muestra de forma esquemática el diseño de la jerarquía de memoria, para tratar de dar una visión general de la solución planteada (Ilustración 20).

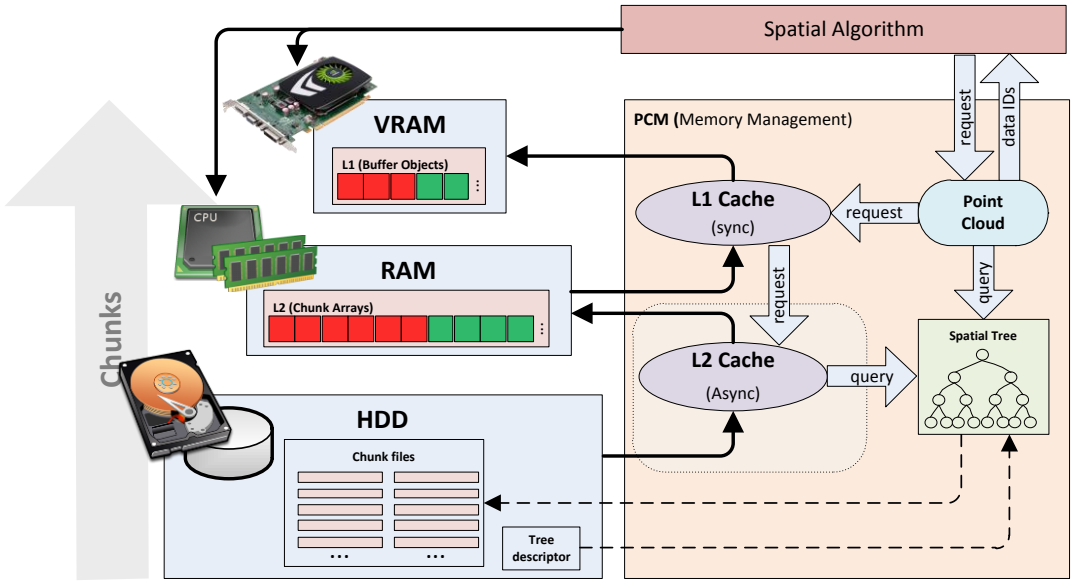


Ilustración 20. Sistema de gestión de la jerarquía de memoria.

En la parte izquierda de la figura, se puede observar la jerarquía de memoria con los datos del sistema, los chunks, en cada uno de los niveles. Como se comentaba, en el nivel más bajo residen los ficheros binarios correspondientes a cada chunk. Recordemos que cada nodo del árbol de la estructura espacial, es un "contenedor" de cada chunk, donde se almacenan los puntos y su información asociada. Por lo tanto, la correspondencia entre los nodos y los chunks es 1:1. En el siguiente nivel, la memoria RAM, se almacenará un subconjunto de chunks, en *arrays* del tamaño aquel con el que se haya configurado el sistema. Por último, en la memoria de vídeo o VRAM, los chunks sufren una transformación para adaptarse al formato de los *BufferObjects*, que como se vio son las unidades de memoria con las que puede trabajar la GPU.

En la parte derecha se ve una representación abstracta del sistema de gestión de memoria propuesto, donde destacan los dos niveles de caché. La sucesión de eventos sería de la siguiente manera:

1. Un proceso o algoritmo espacial mantiene una entidad genérica que representa una nube de puntos, "Point Cloud". En un momento dado, notifica que quiere trabajar sobre una zona del dataset y a un nivel de detalle particular.
2. La "*Point Cloud*", realiza una serie de consultas espaciales a la estructura de datos multirresolución para conocer qué nodos se corresponden a la zona que se le solicita, y con qué prioridad.
3. Conociendo esta lista ordenada por prioridad de nodos, realiza una petición de datos a la caché síncrona L1, para notificar que quiere trabajar con esa zona en la GPU (en caso de que se utilizase solamente la CPU, esta petición iría directamente a la caché L2). Como ya se ha comentado, esta petición se puede realizar de dos modos: o bien limitándola por tiempo, de forma que la el sistema cargue todos los datos que le sea posible en ese lapso de tiempo, y retorne la llamada; o bien limitada por nivel de detalle, en cuyo caso la llamada no retorna hasta que se haya alcanzado dicho nivel.
4. La caché L1, comprueba si esos datos están ya subidos a VRAM. En caso contrario, comprueba su disponibilidad en RAM:
    a. Si están disponibles, convierte los chunks en *BufferObjects* y los sube hasta que, o bien termine de subirlos todos, o bien alcance la máxima memoria VRAM permitida, configurada previamente en el sistema.
    b. Si no lo están, los solicita a la caché asíncrona L2.

En un hilo diferente de ejecución, y por lo tanto, de forma asíncrona, la caché L2 irá recibiendo notificaciones de cargar chunks de disco a RAM. Realizará una consulta al árbol de nodos, para conocer qué ficheros *chunk* se corresponden con los nodos solicitados, que serán cargados en memoria.

Este sistema plantea una serie de **ventajas**:

- **Los únicos parámetros con los que se ha de configurar el sistema son las cantidades máximas de memoria RAM y VRAM permitidas**.

- A su vez, esto **permite utilizar el sistema en cualquier equipo, sean cuales sean sus capacidades**.
- **La gestión de memoria es completamente transparente para el usuario**. Sus algoritmos espaciales funcionarán de la misma manera en cualquier sistema, aunque su eficiencia, evidentemente, dependerá de las capacidades del equipo donde se ejecute.

4.4.1 LA CACHÉ ASÍNCRONA DE 2º NIVEL (L2)

La caché L2 será básicamente la encargada de pasar datos (chunks) del disco a la memoria del sistema RAM. Es una caché con política de remplazo **LRU** (Menos Usada Recientemente o ***Least Recently Used***) que, como se ha demostrado en [15], es la más adecuada para este tipo de sistemas. Como la gestión del sistema de ficheros es del sistema operativo, esta caché funciona en un hilo de ejecución aparte del sistema. Esto implicará algunas complicaciones en su implementación (uso cuidadoso de MUTEX para evitar *bloqueos mutuos*[5] y mantener la coherencia en los datos) como se verá en su correspondiente capítulo.

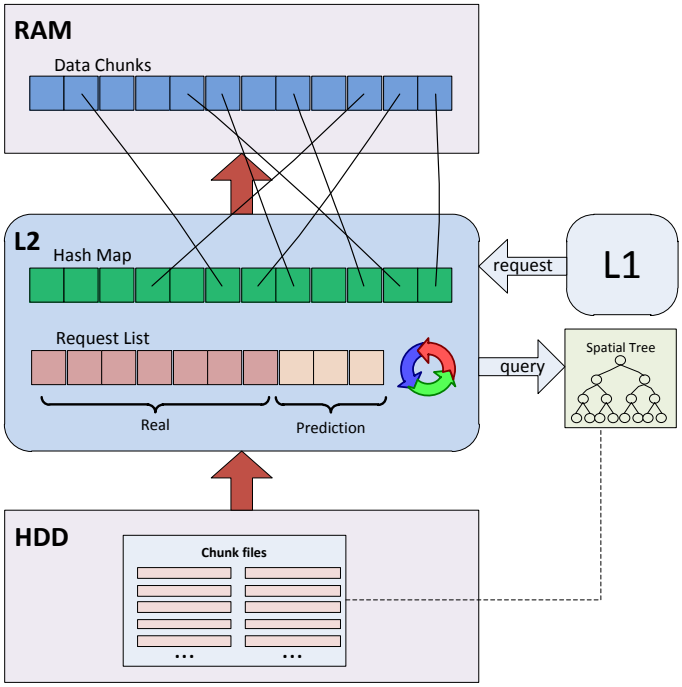


Ilustración 21. Estructura esquemática de la caché asíncrona L2.

Además tiene las siguientes responsabilidades:

- Actualizar la lista de peticiones de carga efectuadas por la caché inmediatamente superior (L1)

[5] Se denomina bloqueo mutuo o *deadlock* al bloqueo permanente de varios hilos de ejecución cuando están compitiendo por algún recurso común. Es uno de los principales riesgos que entraña la programación multi-hilo.

- Mantener un “repositorio” de chunks recientemente cargados, que debido a las características espaciales y la estructura en árbol de los datos, serán utilizados nuevamente con una alta probabilidad.
- Consultar a la estructura multirresolución qué ficheros corresponden a cada nodo de la lista de peticiones.
- Mantener la coherencia en la memoria del sistema, mediante un mapa de tipo hash que relacione los IDs únicos de los nodos con su chunk en memoria.

El bucle principal de hilo se puede resumir de una forma muy somera en el siguiente pseudocódigo. Básicamente se mantiene esperando una nueva lista de peticiones, y cuando le llega, va cargando ordenadamente cada uno de los chunks demandados. En caso de durante la carga aparezca una nueva lista, descarta la anterior y comienza de nuevo.

```
WHILE(!fin)
      IF(!listaPeticiones.vacia & !cacheLlena)
            cacheLlena := cargarNodo(listaPeticiones.siguiente())
            listaPeticiones.pop()
            IF(hayNuevaLista)
                  actualizarListaPeticiones()
                  cacheLlena := false
                  continue
      ELSE
            IF(!hayNuevaLista)
                  esperaNuevaLista()
```

La tasa de acierto de esta caché, medida en *hits/requests* es realmente alta, como se verá en el capítulo dedicado a resultados. Esto es debido a la topología del árbol de la estructura multirresolución: un algoritmo que vaya realizando pasadas por la nube de puntos, irá requiriendo llegar al máximo nivel en distintas zonas. Como los niveles de detalle son aditivos, para alcanzar el máximo nivel, debe sumarse la información contenida en toda la ruta de nodos (que denominamos *nodepath*) desde la hoja hasta la raíz. Pero en dos regiones del espacio cercanas, esto supondrá compartir una gran cantidad de nodos.

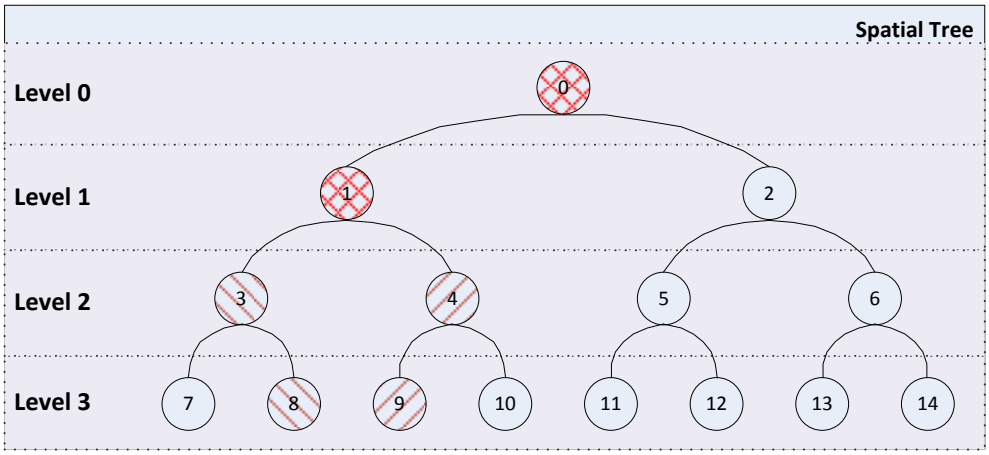


Ilustración 22. Dos rutas de nodos cercanos espacialmente (al nodo 8 y al 9), para alcanzar el nivel máximo de detalle (nivel 3), comparten los nodos 1 y 0.

En la Ilustración 22 puede verse este efecto. La sub-región cubierta en el nodo 8, en su máximo nivel de detalle, implicará puntos de los nodos 8, 3, 1 y 0. De la misma forma, la sub-región del nodo 9 implicará los puntos de los nodos 9, 4, 1 y 0. En estos dos *nodepaths*, la mitad de los nodos (el 0 y el 1) están compartidos. Si a la caché se realiza una petición para la región del 8, cuando realice la siguiente, del 9, tendrá ya los nodos 0 y 1 cargados en memoria del sistema, lo cual implicará un ratio de acierto de 2:4.

4.4.2 LA CACHÉ SÍNCRONA DE 1º NIVEL (L1)

La caché de primer nivel (L1) se encarga de gestionar las cargas de datos desde memoria del sistema (RAM) a memoria de vídeo (VRAM). A pesar de que su estructura es similar a la L1, por las características de los procesos que trabajan en GPU, su funcionamiento es de tipo síncrono. **Su mayor responsabilidad, además de la correspondiente gestión de memoria, consistirá en convertir los chunks de datos de RAM en *BufferObjets* de VRAM**, el tipo de unidades de información que maneja la GPU.

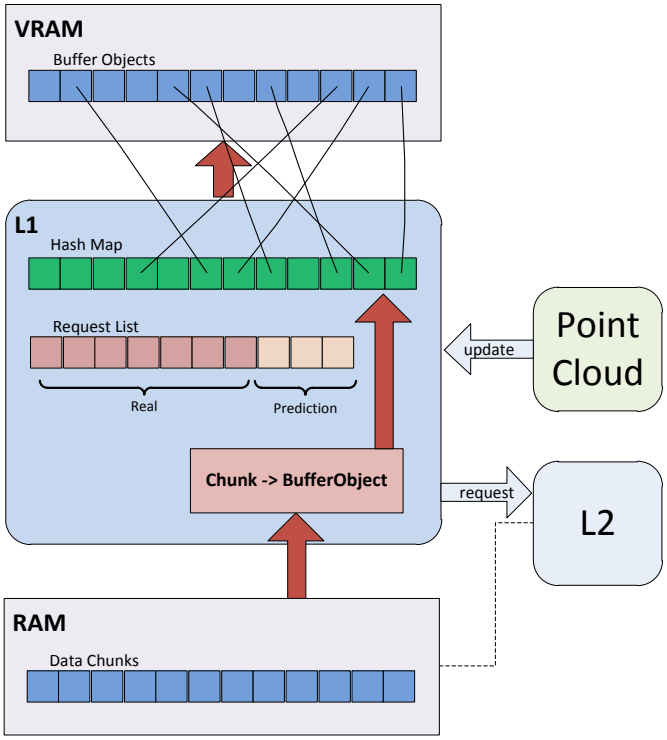


Ilustración 23. Estructura de la caché síncrona L1.

La política de remplazo de la caché L1 será de la siguiente manera: como la memoria VRAM es un recurso muy limitado y valioso, en cada actualización de estado, elimina todos aquellos *BufferObjects* de VRAM que no estén en la lista de solicitudes. De esta forma, cuando tiene que insertar un nuevo *BufferObject*, o bien hay sitio libre, y entonces lo inserta, o bien no lo hay, quedando la caché llena.

4.4.3 PREDICCIÓN

Dado el sistema de gestión de memoria que se acaba de explicar, añadir predicción para aumentar la ratio de aciertos, intuitivamente, tendrá que ver con las características espaciales de toda la estructura.

Nombremos la región sobre la que se esté trabajando en un momento dado como la región de interés. Esta región, definida por un volumen en el espacio tendrá un centro geométrico, que denominaremos centro de interés. Igual que se calcula en la física más básica, derivando los saltos que este centro de interés va dando por el espacio, según se requieren nuevas zonas, se obtiene un vector de velocidad. Así que, tal y como se representa en la Ilustración 24, a la lista de nodos requeridos que se solicita a la caché L1 (y que expandirá a la caché L2), se le acopla al final una nueva lista, generada a partir de la región interés predicha con el vector de velocidad.

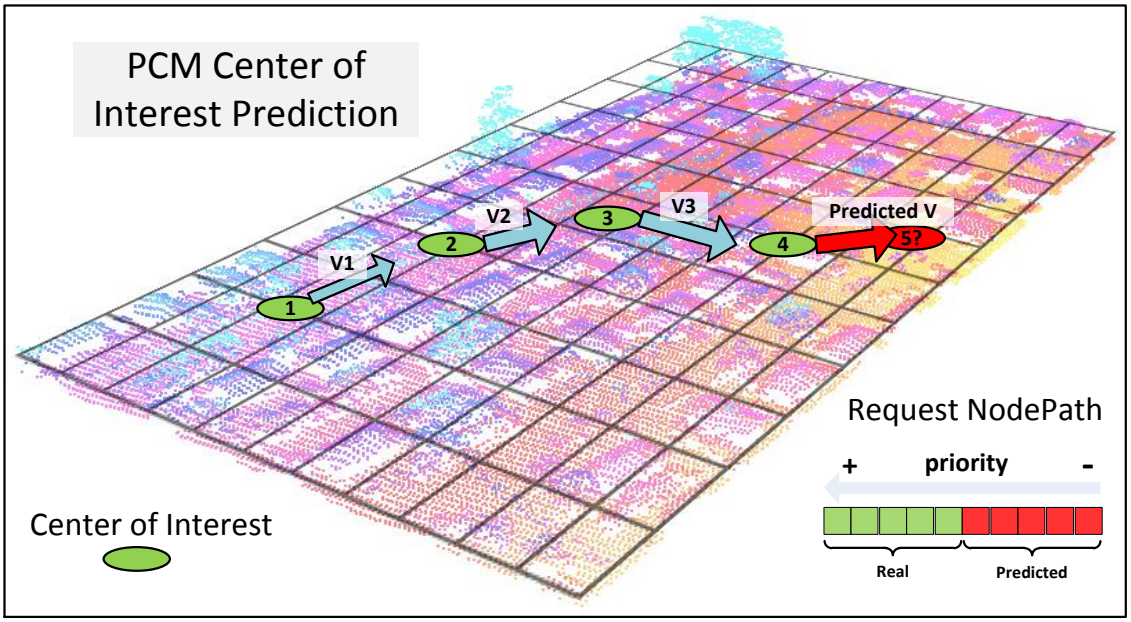


Ilustración 24. Sistema de predicción de PCM.

Como normalmente el acceso que se realizará a la estructura tendrá un movimiento suave (*blocking* en los algoritmos, movimiento de cámara lento en render, etc.) el sistema de predicción funcionará razonablemente bien, como se demuestra en los resultados. Sin embargo, para un acceso aleatorio, el sistema fallará estrepitosamente, ya que los vectores de velocidad del centro de interés carecerán de sentido alguno. En estos casos, será conveniente desactivar el sistema predictivo.

Por último, comentar que el hecho de duplicar la lista de nodos requeridos no supone ningún problema ni sobrecarga del sistema, incluso aunque existan peticiones de nodos repetidas en la misma lista. Por el orden prioritario de las listas, las cachés no comenzarán a cargar nodos de la predicción hasta que no hayan terminado con los nodos de la petición actual (la "real"), o hasta que éstas estén llenas; y en el caso de encontrarse nodos repetidos, comprobar si un nodo está cargado o no es inmediato gracias a los mapas hash, así que simplemente se descarta.

5 EL PAQUETE DE SOFTWARE PCM

En este capítulo se presentará **Point Cloud Manager** (en adelante **PCM**), como la solución integral software, conjunto de herramientas y librerías, donde se han implementado las ideas desarrolladas en el capítulo anterior. Este nuevo sistema nace con dos expectativas fundamentales:

- Por una parte, que sirva como **marco de trabajo para la investigación** en los campos del tratamiento y visualización de nubes de puntos:
    - En el propio núcleo del sistema, en los campo de estructuras de datos y jerarquías de memoria para gestionar grandes cantidades de información en computadoras con recursos limitados.
    - En el "extrarradio" del sistema, para los algoritmos de procesado de nubes de puntos con carácter espacial, donde se puedan aplicar todo tipo de técnicas (*Machine Learning*, *Data Mining*, etc.) para conseguir resultados en diversos ámbitos de la ingeniería aprovechándose de las capacidades que ofrece esta arquitectura.
- Por otra parte, como paquete **software con carácter comercial**, que pueda afrontar los problemas de ingeniería planteados en proyectos reales.

5.1 ROLES DE USUARIO

Antes de entrar en la estructura de PCM, se definen a continuación los tres tipos de roles que interactuarán con el software, ya que sus componentes están ideadas en función de las necesidades de cada uno de ellos.

- **Usuario de PCM no desarrollador**: utilizará principalmente las herramientas (*PCM tools*) que le permitirán convertir sus datos LIDAR en un dataset unificado, que podrá visualizar en 3D o al que podrá aplicar distintos procesos algorítmicos de su colección, para obtener resultados o informes. El número de parámetros a configurar para utilizar el sistema debe ser mínimo.
- **Usuario de PCM desarrollador**: mediante el interfaz de la librería (*PCM library*) podrá desarrollar sus propios procesos espaciales en librerías aparte, para resolver los problemas que desee sobre una nube de puntos de tamaño arbitrario. Además, podrá configurar hasta qué nivel de la jerarquía de memoria quiere utilizar, para valerse o bien únicamente de la CPU (jerarquía hasta la memoria RAM) o también de la GPU con lenguajes como CUDA o OpenCL (jerarquía hasta la memoria VRAM).
- **Desarrollador de PCM**: para los autores de la propia librería PCM, además se ofrece una colección de test unitarios y un test del sistema completo, de forma que puedan comprobar cómo afectan sus contribuciones al funcionamiento global de la librería, y sirva además como una primera barrera de control antes de incorporar las posibles mejoras.

5.2 ESTRUCTURA

En el siguiente diagrama (Ilustración 25) se representan los componentes básicos que componen el paquete de software y las formas de interacción de cada uno de los roles de usuario que se acaban de comentar.

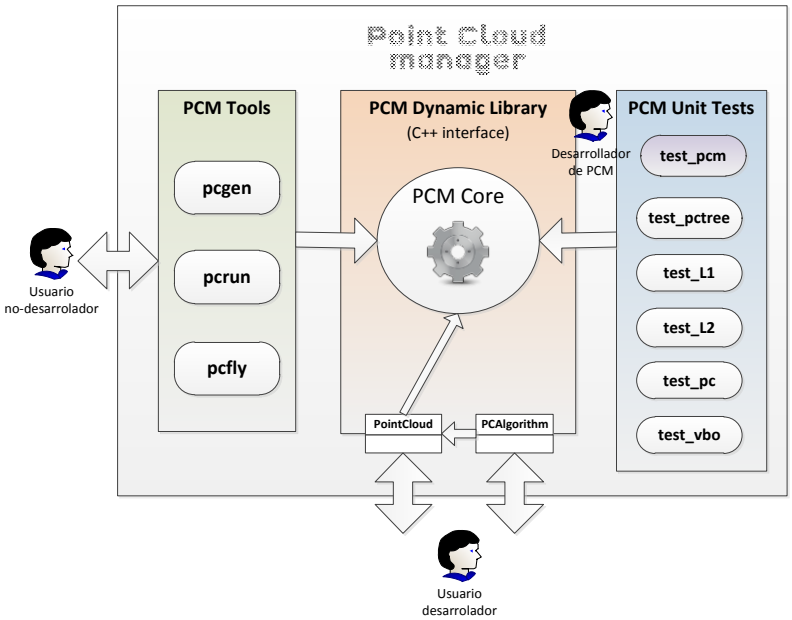


Ilustración 25. Arquitectura del paquete software PCM.

Se detallan a continuación cada uno de los componentes.

5.2.1 HERRAMIENTAS (*PCM TOOLS*)

Por motivos de compatibilidad y facilidad de uso, las herramientas que proporciona PCM están basadas en línea de comandos, y proporcionan un interfaz de cara al usuario con las que puede ejercer todas las operaciones necesarias (conversión de formatos, creación, información y visualización del dataset, ejecución de procesos espaciales, etc.) sin que sea necesaria una recompilación ni del sistema ni de los *plugins* anexos desarrollados por terceros. Todas ellas dependen de la *PCM Library*, y hacen un uso intensivo de distintas clases software contenidas en ésta.

Se detallan a continuación.

5.2.1.1 PCGEN

Esta herramienta, acrónimo de ***Point Cloud GENerator***, es la encargada de la lectura de distintos formatos LIDAR (y otros) y de la generación de la estructura de datos con la que trabajará el sistema, así como de proporcionar información relevante acerca de cualquiera de estos elementos. Esta herramienta utiliza fundamentalmente las clases software de la PCM Library dedicadas al tratamiento de grandes *streams* de datos en disco, como se verá en el siguiente capítulo.

Su funcionamiento se puede resumir en tres comandos fundamentales (para más información acerca de los formatos comentados, ver la siguiente sección acerca del pipeline de uso).

- **Convert**: capaz de leer distintos formatos LIDAR y modelos 3D para generar un formato unificado que se denomina BPC (Binary Point Cloud).
- **BuildTree**: que recibe como entrada el formato unificado de nubes de puntos binarios (BPC) y genera el dataset para su utilización, compuesto por un descriptor de árbol binario y una colección de ficheros binarios, a partir de una serie de parámetros de configuración.
- **Info**: que devolverá información relevante (número de puntos, tamaño de la información adjunta, niveles, nodos, etc.) sobre cualquiera de los ficheros o carpetas involucrados en el proceso.

5.2.1.2 PCFLY

Herramienta 3D para poder navegar por un *dataset* en tiempo real. Su nombre es un acrónimo de **Point Cloud FLY**. Valiéndose del API gráfico estándar **OpenGL**, y aprovechando la carga de datos en memoria VRAM, crea un espacio 3D donde se puede "volar" a través de la nube de puntos, con una doble misión:

- Analizar el modelo 3D generado por escaneo láser.
- Visualizar el funcionamiento del sistema de cachés y la forma de árbol del *dataset*.

En ella, se pueden realizar los movimiento básicos del espacio tridimensional (*pan*, *pitch* y *zoom*) utilizando el ratón, y monitorizar en tiempo real los datos concernientes a las cachés de primer y segundo nivel. También se pueden seleccionar distintos modos de *debug* para comprender mejor el sistema de jerarquía de memoria, de forma que coloree los distintos niveles de detalle, etc.

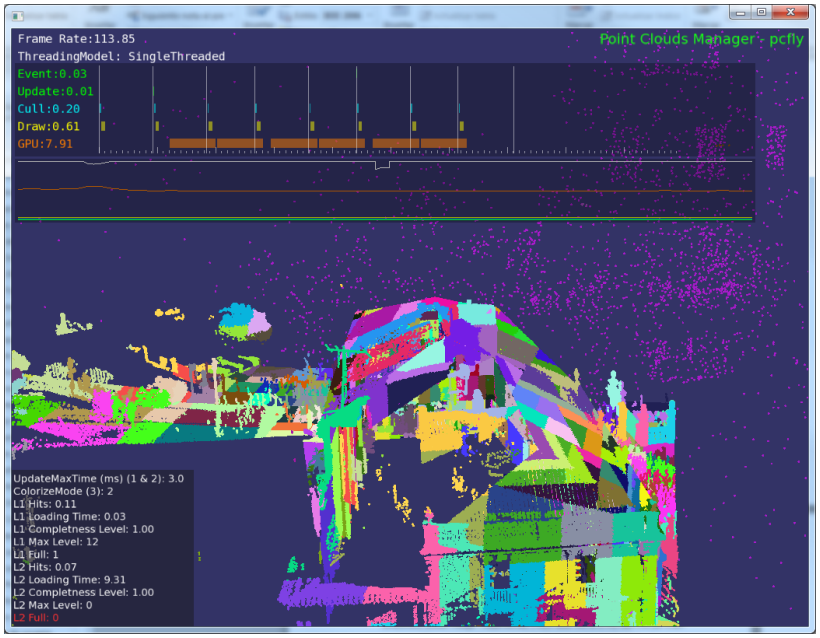


Ilustración 26. Captura de pantalla de la aplicación PCFLY en uso, monitorizando la carga e niveles y el estado de las cachés.

5.2.1.3 PCRUN

Herramienta para la ejecución de procesos sobre un *dataset* de nube de puntos. Solamente se le pasarán tres parámetros:

- Dataset sobre el que trabajar
- Librería dinámica donde se encuentre el algoritmo y su nombre
- Fichero XML de configuración del algoritmo

5.2.2 TESTS UNITARIOS (*PCM TEST*)

Estas herramientas, más pensadas para los desarrolladores del propio paquete PCM que para los usuarios finales, permiten realizar una serie de test unitarios de cada parte del software (caché L1, caché L2, árbol de datos, carga y descarga de *BufferObjets* a la GPU, etc.) y un test global del funcionamiento del sistema. La existencia de estos test viene dada fundamentalmente por dos motivos:

- Por una parte, existen ciertas entidades en el sistema que, a pesar de formar un conjunto bien cohesionado, pueden trabajar independientes, y cuyo desarrollo es a muy bajo nivel, por lo tanto muy sensibles a cambios. Por lo tanto resulta fundamental tener un conjunto de test que sean rápidamente aplicables y que sirvan como "*sandbox*" para probar las nuevas características desarrolladas.
- Por otro lado, será parte de la metodología propuesta la ejecución "mecánica" de estos test para mantener una consistencia y estabilidad en el software, de cara a los usuarios finales.

5.2.3 LIBRERÍA PCM (*PCM LIBRARY*)

El núcleo principal de la arquitectura reside en la librería software dinámica PCM. Esta librería, diseñada para ser compilable en una amplia variedad de sistemas operativos, contendrá fundamentalmente las funcionalidades necesarias para las siguientes tareas:

- Manejo de grandes *streams* de datos provenientes de disco duro
- Lectura/Escritura de:
    - Distintos formatos de modelos 3D y escaneos LIDAR
    - Formato propio de nubes de puntos "en crudo" (PCB)
    - Descriptor de árbol binario del *dataset*
    - *Chunks* de datos (pequeños ficheros binarios con subconjuntos del *dataset*)
- Gestión del *dataset* como árbol binario
- Gestión en tiempo real de la jerarquía de memoria
- Run-time propio para la ejecución de procesos sobre el *dataset*
- Proceso de render para la visualización del *dataset*

Los detalles de funcionamiento de la librería han sido comentados en el capítulo anterior, y su diseño e implementación, en el siguiente.

5.3 PIPELINE DE USO DEL SISTEMA

Para utilizar el sistema, existen una serie de pasos previos a realizar, que se centran en llegar desde los datos "en crudo" que los equipos LiDAR o escáneres 3D sacan, hasta la BBDD que puede manejar PCM. En el siguiente diagrama (Ilustración 27) puede verse el proceso completo: los datos del escáner son normalmente procesados por un software propio del fabricante (unión de escaneos, calibración, georreferenciación, etc.) que tendrán una salida estándar en alguno de los formatos vistos anteriormente. A partir de éstos, entra en juego la herramienta PCGEN, para convertirlos a un fichero único binario que compacte todos los puntos, y en cuya cabecera se almacena información de bajo nivel relevante, como número de puntos, tamaño en bytes de los datos asociados, tipo de precisión, etc. Por último, una nueva llamada a PCGEN generará la base de datos final en una carpeta, con todos los ficheros chunk y el descriptor de la estructura espacial multirresolución, que será con lo que trabajen el resto de herramientas de PCM.

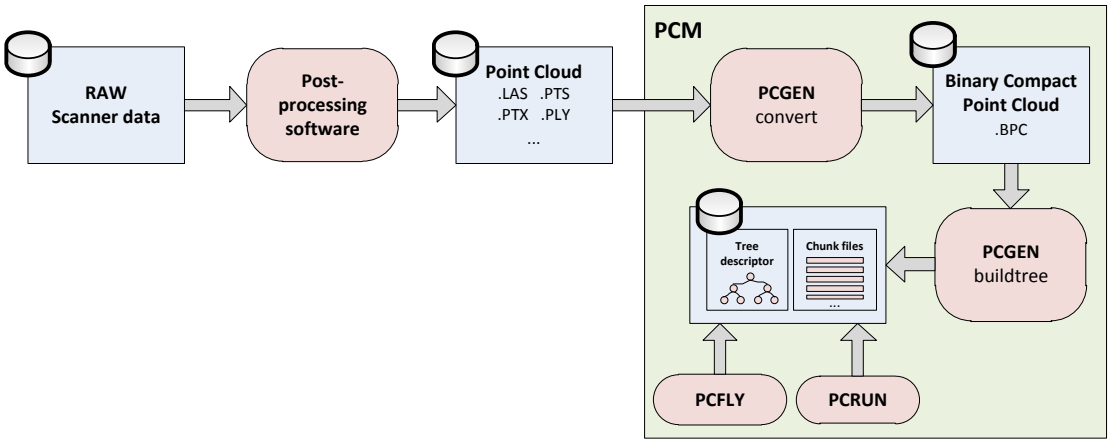


Ilustración 27. Proceso de conversión de datos para el uso del sistema PCM.

5.4 SEGUIMIENTO DEL PROYECTO

Como se verá en el último capítulo dedicado a las líneas futuras de PCM, se pretende que este trabajo tenga una proyección comercial, y que se pueda crear una comunidad de desarrollo entorno a él. Por lo tanto, se vuelve indispensable disponer de un sistema de gestión y de control de versiones, que tenga las siguientes características:

- Accesible vía Internet y multiplataforma, dado que el proyecto también lo es.
- Con control de versiones de código concurrente.
- Que gestione una BBDD de información del proyecto.
- Con guías de referencia para el API.
- Que permita hacer un seguimiento de hitos basado en tickets.

- Con control de usuarios.
- Fiel a los principios del manifiesto ágil [19].

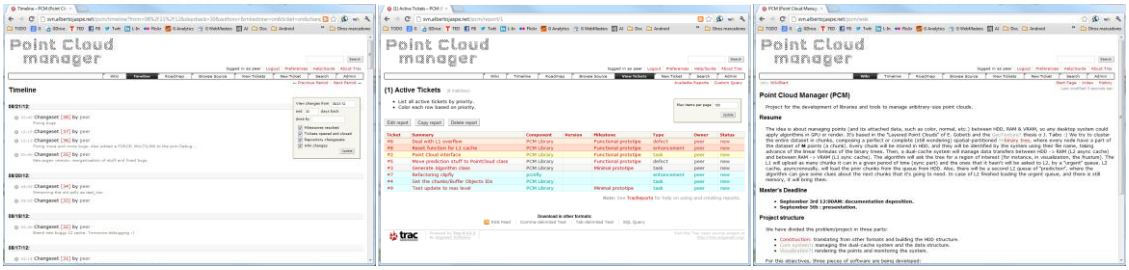


Ilustración 28. *Screenshots* del sistema web TRAC montado ex profeso para el desarrollo y seguimiento de PCM.

Tras barajar varias posibles opciones, se ha optado por una solución web basada en el binomio **TRAC + Subversion** [20], bajo un servidor Apache en una máquina Ubuntu. Esta configuración cumple con todos los requisitos expuestos, y además está completamente basada en software libre, por lo que su coste de implantación se mide exclusivamente en horas de trabajo.

Por ofrecer algunos datos de su utilización, desde la implantación del sistema hasta la presentación de esta memoria, se gestionan tres usuarios diferentes (autor y directores), casi un centenar de *commits* de código, varias páginas *wiki* con documentación y referencias creadas, y multitud de tickets que conforman dos hitos, el primero de los cuales está completo al 100% (versión funcional de PCM) y el segundo al 30% (primera *release*).

6 DISEÑO E IMPLEMENTACIÓN DE PCM

En este capítulo se describe el diseño actual de la librería PCM, en los aspectos que incumben a este trabajo, así como algunos detalles de implementación que puedan resultar interesantes. Para ello se presentarán algunos diagramas de clases, de las partes más relevantes del sistema, en el lenguaje estándar UML.

PCM está desarrollada siguiendo el paradigma de la Orientación a Objetos, y desarrollada con lenguaje C++. Además, se ha tratado en todo momento seguir en la medida de lo posible directrices de desarrollo conocidas como Patrones de Diseño [2], Los compiladores utilizados hasta el momento han sido Visual C++ 2010 en sistemas Windows y GCC/G++ 4 con el sistema de construcción CMake en sistemas Linux.

6.1 DIAGRAMA DE CLASES GLOBAL

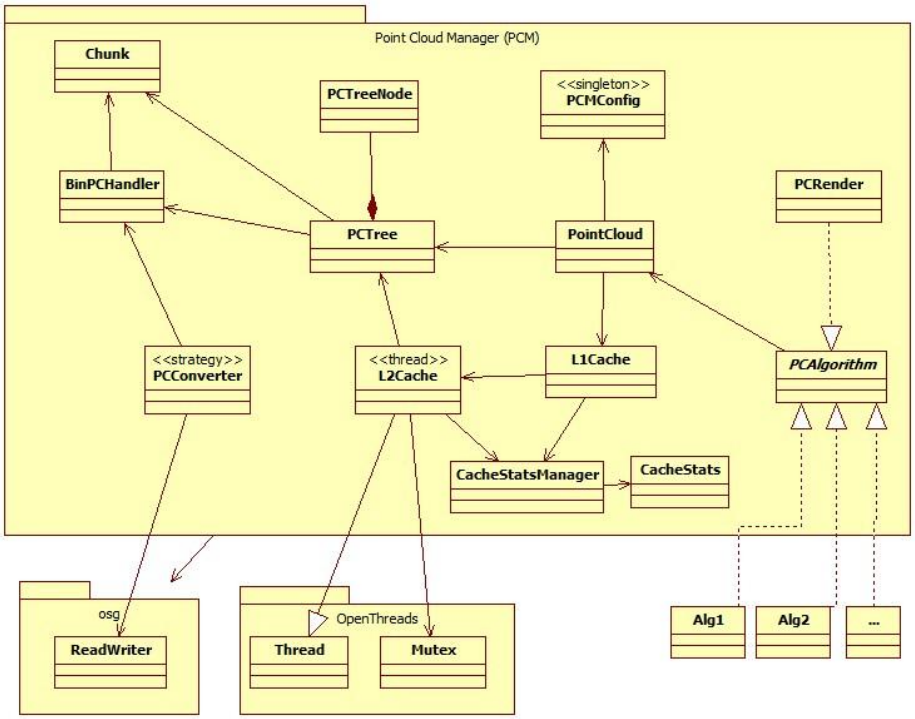


Ilustración 29. Diagrama de clases global (no detallado) de la librería PCM.

La librería PCM se compone (actualmente) de 13 clases. Como se aprecia en el diagrama, depende fundamentalmente de dos librerías:

- **OpenSceneGraph (OSG)** [21]: librería de gestión de un scenegraph gráfico para visualización en tiempo real. A pesar de que no se aproveche como visualizador en sí, contiene diversas clases que son de gran utilidad en el sistema relacionadas con álgebra espacial, y con la gestión de *BufferObjects*. Tiene una gran comunidad de usuario y desarrolladores detrás, es multiplataforma, y se licencia bajo LGPL.

- **Open Threads (OT)**: librería anexa a OpenSceneGraph, es un API de abstracción de Threads, muy parecido al API propuesto por Posix Threads. Tiene la gran ventaja de tener una implementación particular para cada plataforma, lo cual la hace muy versátil y eficiente.
- **Standard Template Library (STL)**: PCM hace un uso intensivo de las operaciones y contenedores que ofrece la librería estándar de C++.

Únicamente destacar aquí la clase ***PCConfig***, que encapsula la configuración global de la nube de puntos, y a la que tendrán por lo tanto acceso todas las clases del sistema. A continuación se resaltarán distintas partes de este diagrama, por áreas de la librería, con sus métodos y atributos más importantes[6].

6.2 CONVERSIÓN Y MANEJO DE DATASETS

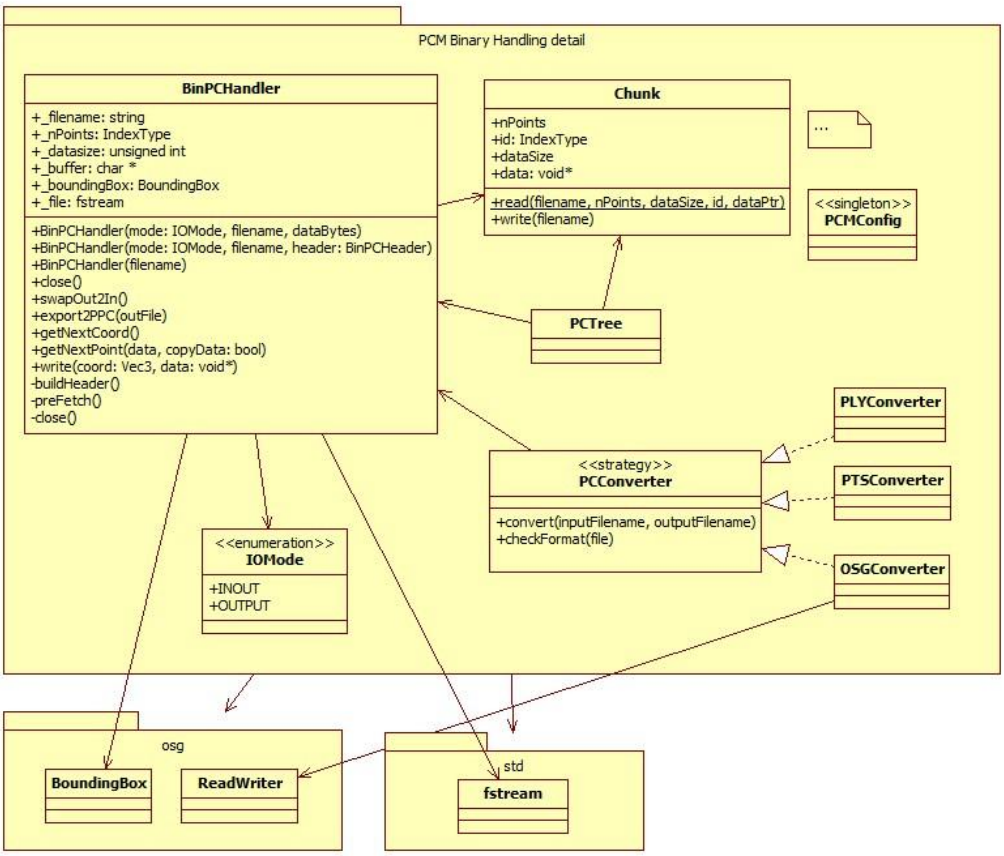


Ilustración 30. Diagrama de clases de PCM, resaltando la parte dedicada al manejo de ficheros binarios y conversión de formatos.

Destacaremos en este diagrama las siguientes clases:

- ***Chunk:*** representa la unidad mínima de información que se moverá por los distintos niveles de la memoria. Por lo tanto, su tamaño en bytes debe estar muy controlado (sus atributos son los estrictamente necesarios para gestionar la información que contiene). Sólo tiene métodos para lectura/escritura de disco.

[6] Los atributos y métodos mostrados no se exponen de forma estricta para facilitar su lectura.

- ***BinPCHandler***: esta clase representa el formato intermedio binario compactado (BPC) en la conversión de formatos y la construcción de la estructura espacial. Su objetivo es representar un array arbitrariamente grande de datos, como el *template vector* de la STL, pero que en vez de residir en memoria RAM, reside en disco. Su justificación es que, durante la construcción, se realizarán operaciones como la búsqueda de la mediana sobre el conjunto completo de puntos. Obviamente, en el caso general estos datos no cabrán en memoria RAM. Con esta clase, se puede operar sobre ellos "como si lo estuvieran". Para conseguir mejorar el rendimiento, hace *pre-fetching* y *buffering* de datos cada cierto número de peticiones, y además permite tanto lectura como escritura. Es ciertamente eficiente para el acceso secuencial a los datos, no así para el acceso aleatorio.
- ***PCConverter* y sus estrategias**: esta clase es la encargada de proporcionar una interfaz única para crear conversores de cualquier tipo de fichero de nube de puntos a al formato binario compacto BPC, siguiendo el patrón "estrategia". Su método "*checkFormat*" comprueba el formato y dirige la responsabilidad de conversión a la clase hija que le corresponda.

6.3 ESTRUCTURA DE DATOS

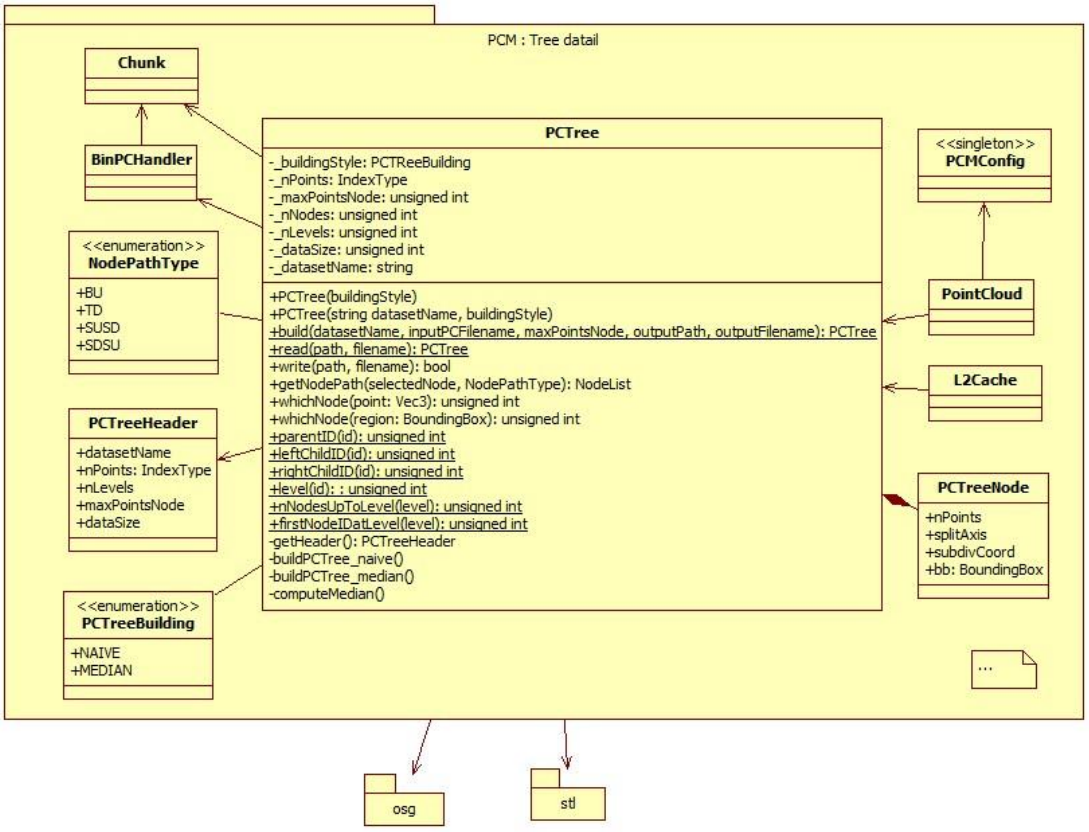


Ilustración 31. Diagrama de clases de PCM, resaltando la parte dedicada al manejo de la estructura de datos.

Como se aprecia en el diagrama, la clase fundamental en este caso es aquella que representa la propia estructura, ***PCTree***. La clase encapsula dos funcionalidades fundamentales:

- La construcción de la BBDD de chunks y de sí misma, efectuada en el proceso previo a la utilización de la librería, mediante “PCGEN buildtree”.
- La gestión del árbol multirresolución, con su operaciones tipo *traversal*, y construcción de *NodePaths*.

Los nodos del árbol, de la clase PCTreeNode, son mantenidos en un vector lineal, cuyo acceso aleatorio es realmente rápido ($O(1)$). Contiene además una lista de funciones *static*, válidas para cualquier árbol binario y por lo tanto, no dependientes de una instancia particular, para la navegación por la estructura (*leftChildNode*, *rightChildNode*, *parentNode*, *level*, *isLeaf*, etc.).Por otra parte, los métodos sobrecargados *whichNode* permiten seleccionar el nodo que corresponde a la región de interés, y *getNodePath*, a partir del nodo seleccionado, construye listas de prioridad de nodos, según los cuatro métodos de *traversal* vistos en capítulos anteriores.

6.4 JERARQUÍA DE MEMORIA

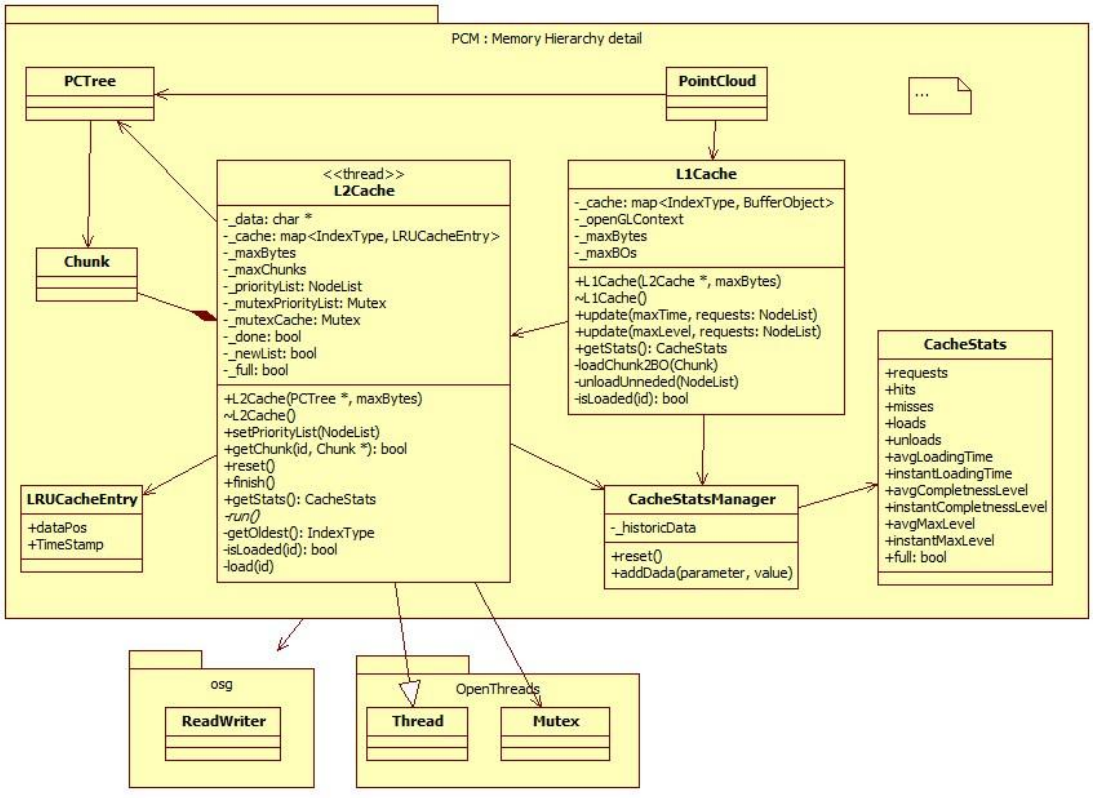


Ilustración 32. Diagrama de clases de PCM, resaltando la parte dedicada a la gestión y jerarquía de memoria.

En este caso, se muestran detalladas las clases que corresponden con las cachés y su monitorización.

La clase ***L2Cache***, responsable de la carga de datos entre HDD y RAM, hereda de ***OpenThreads::Thread*** para poder constituir un hilo de ejecución independiente. Como se comentó en el análisis, un mapa hash (atributo *_cache*) mantiene referencias a qué chunks están cargados y en que parte de la memoria,

apuntada por el atributo *_data*. La lista de nodos por prioridad se mantiene en todo momento en el atributo *_priorityList*. Ambas estructuras están protegidas por sendos *Mutex* para evitar la corrupción de los datos por acceso concurrente. El método *run()* es el que contiene el bucle principal de ejecución.

La clase ***L1Cache***, responsable de la gestión de datos hacia VRAM y de convertir chunks en *BufferObjects*, mantiene igualmente un mapa hash con las referencias de memoria. El método sobrecargado *update()* será el encargado de hacer las peticiones a la caché L2 y de convertir y subir los BufferObjects a VRAM, mediante el método *loadChunk2BO()*.

Ambas clases contienen una instancia del ***CacheStatsManager***. Esta otra clase se encarga de la monitorización de cada una de las cachés, y de sacar estadísticas de estado y funcionamiento, tales como el ratio de acierto, tiempos medios e instantáneos de carga, tiempos de vida de los chunks en caché, etc.

6.5 INTERFAZ EXTERIOR

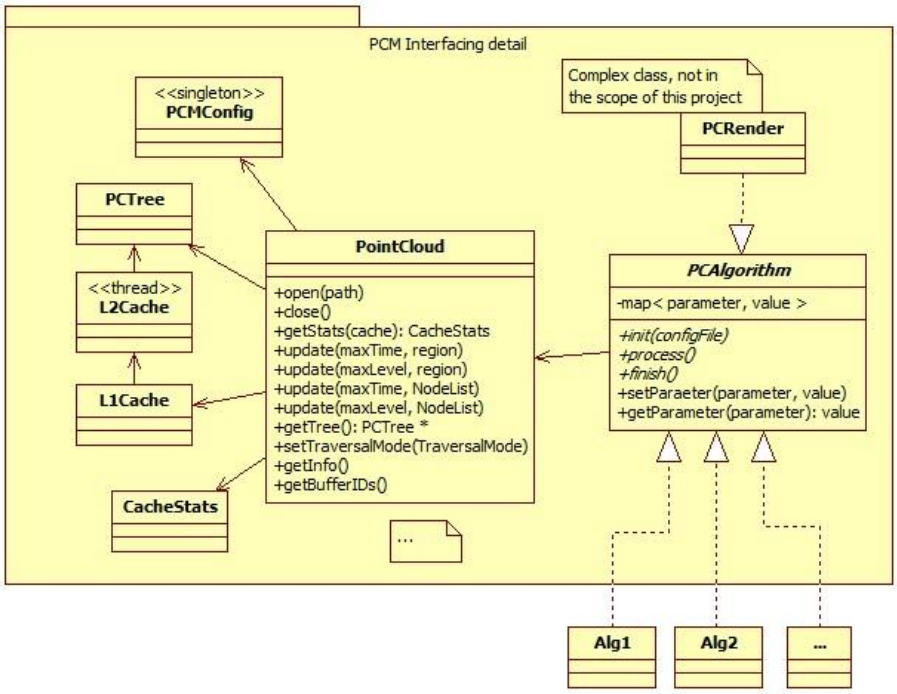


Ilustración 33. Diagrama de clases de PCM, resaltando la parte dedicada la interfaz del sistema de cara al exterior.

Se muestra en último lugar las clases que constituirán la interfaz visible de cara al exterior de la librería PCM. ***PointCloud*** representa un *dataset* completo. Como se puede observar, su interfaz es muy sencillo en intuitivo, respetando los requisitos originales con los que se ideó el sistema. Permite abrir o cerrar un dataset, pedir información sobre el mismo, solicitar la carga de una región (por volumen en el espacio o requiriendo unos nodos particulares) y da acceso a la estructura espacial, por si se quiere realizar alguna consulta especial sobre ella.

La clase virtual pura ***PCAlgorithm*** es la que dará acceso a la plataforma de procesado para los desarrolladores. Simplemente tendrán que hacer una clase que herede de ésta, y rellenar los siguientes métodos:

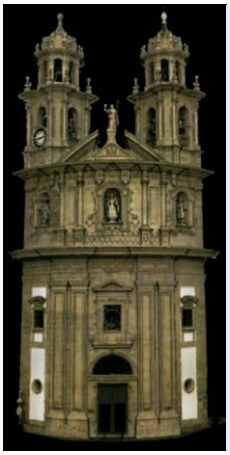
- ***init***(): de inicialización de sistema, variables, etc. Por ejemplo, en un proceso CUDA es donde se inicializaría el contexto.
- ***process***(): donde se crearía el algoritmo de procesado, con su propia condición de parada. Desde aquí se tendrá acceso a una variable protegida, _pointcloud, instancia de la clase PointCloud, con todo su API para trabajar sobre ella.
- ***finish***(): al terminar el proceso, se cerrarán aquí los contextos creados, y las liberaciones de memoria pertinentes.

Uno de los algoritmos de procesado interno de la librería, que sigue este método, es **PCRender**. Esta clase es la que utiliza PCFLY para visualizar con el API gráfico OpenGL en tres dimensiones la nube de puntos. Las técnicas de render con las que trabaja y su diseño e implementación no se explicarán en esta memoria, dado que quedan fuera del ámbito de este trabajo.

7 RESULTADOS

En este capítulo se mostrarán algunos resultados de tests ejecutados en el sistema presentado. El número de pruebas, monitorizaciones, etc. que se podrían obtener de una arquitectura como la mostrada es enorme. Las que se presentan tratarán de incidir en los parámetros clave, tanto la estructura de datos como del sistema de cachés.

En la Ilustración 34 se muestran todos los datos comunes de las pruebas, tanto del *dataset* utilizado (escaneo LIDAR terrestre, cortesía de ENMACOSA S.A.) como del PC utilizado en los ensayos.



Dataset

Iglesia de la Peregrina (Pontevedra)

Escaneo LIDAR: ENMACOSA S.A.
Propiedad: Ayuntamiento de Pontevedra
Tamaño del dataset original: ~ 3GB
Tamaño compactado del BPC: ~ 600 MB
Nº Puntos: 30.645.025
Bounding Sphere:
centro: (999, 1053, 118)
radio: 152 m
Precisión de las coordenadas: simple
Información asociada:
Color RGB (3 bytes)
Índice de reflectividad (4 bytes)

Test PC

Hardware
CPU: Intel Core i7-960 3330 MHz (4 cores)
RAM: 12GB DD3 1600
HDD: WD1002FAEX 1TB SATA3 7200RPM
GPU: nVIDIA GeForce GTX275 (GT200)

Software
OS: Windows 7 Ultimate 64bits
Compiler: Microsoft VC++ 2010

Ilustración 34. Dataset y características del PC utilizados en los tests.

7.1 APROXIMACIÓN DE LA MEDIANA

Como se ha explicado anteriormente, durante la construcción del árbol, utilizar la mediana de los puntos del nodo para escoger la coordenada de corte garantiza un árbol binario perfecto. Pero dada la elevada cantidad de puntos a tratar, el cálculo clásico resulta inviable por costoso, así que se ha descrito un algoritmo alternativo para aproximarla. El error de la aproximación resulta sumamente trascendente, puesto que de ser demasiado elevado, podría causar un desbalanceo en el árbol, que no valdría entonces para trabajar con la arquitectura diseñada.

La siguiente gráfica () muestra el "número de posiciones equivocadas" (absolutas) sobre un recorrido por una de las ramas del árbol, monitorizada de una construcción real. Se observa que, para un número muy elevado de puntos (30M) el error no llega a 40 posiciones, lo cual supone una error relativo por debajo del 0,00001%. Además, este error baja bruscamente a prácticamente cero y se mantiene, al calcular la mediana de 4M de puntos o menos. Esto es debido a la naturaleza del algorimo descrito, que trabaja a muy bajo nivel, con la mantisa y exponente de los números en formato flotante IEEE754.

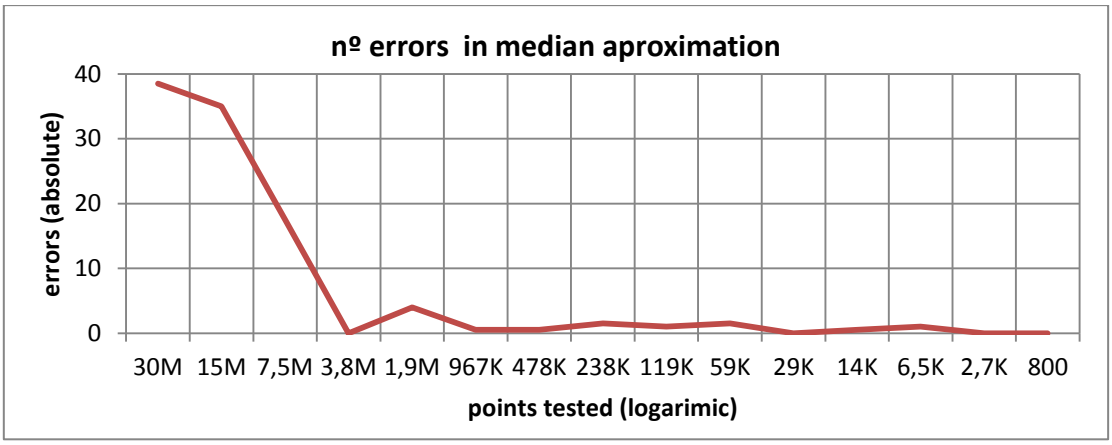


Ilustración 35. Gráfica de tasa de errores en la aproximación de la mediana, para la selección de planos de corte, en la construcción de la estructura de datos espacial.

La tasa de error se considera más que aceptable, y en la práctica suficientemente bueno para la construcción de cualquier árbol. No obstante, se han obtenido en muy raras ocasiones, algún árbol desbalanceado debido a este error.

7.2 CONSTRUCCIÓN DE LA ESTRUCTURA ESPACIAL

La construcción de la estructura espacial multirresolución, y concretamente la topología del árbol que genere, es una de las fases más importantes para el posterior funcionamiento del sistema. Como se ha visto, ésta depende de un único parámetro, al que denominábamos **M**, y que constituía el número máximo de puntos que podría albergar un nodo. De él dependen dos cuestiones:

- El tamaño en bytes de los chunks que se moverán por la memoria.
- El número de niveles del árbol, y por tanto, el número de niveles de detalle.

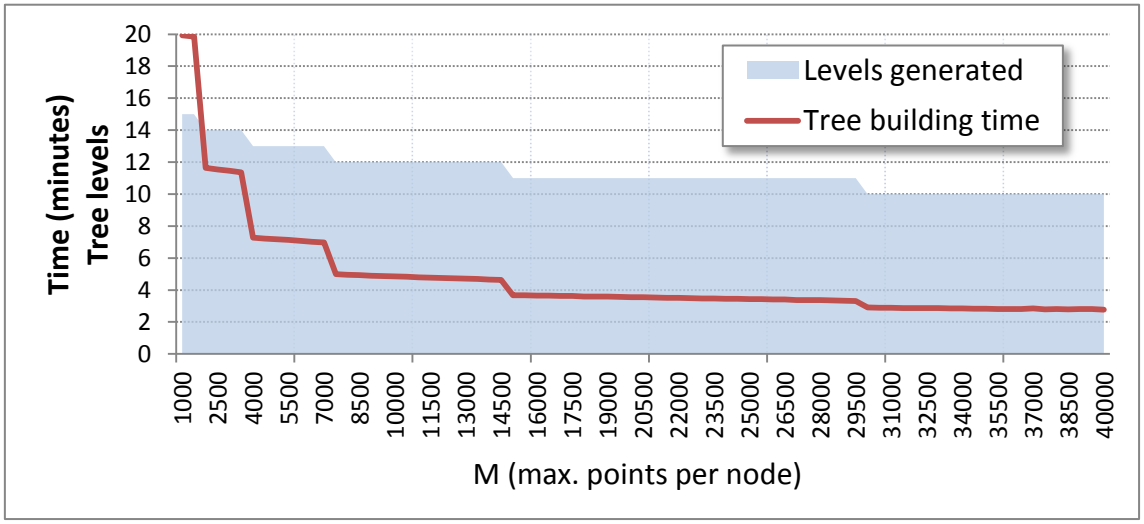
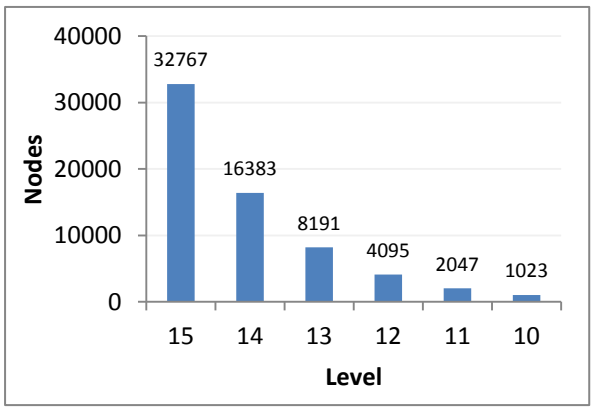


Ilustración 36. Gráfica comparando el parámetro M de máximos puntos por nodos con el tiempo de construcción y los niveles de detalle generados, en la construcción de la estructura espacial.



En la figura anterior (Ilustración 36) se puede observar el resultado de construir el *dataset* de pruebas con el parámetro M variando de 1000 a 40000. Aprovechando la similitud de escala, el eje de las abscisas muestra a la vez el tiempo de construcción en minutos, y el número de niveles de detalle generados. Se aprecia cómo el tiempo de generación de la estructura sigue un patrón logarítmico inverso, asintotado en los 3 minutos, aproximadamente, tiempo mínimo que tardan los algoritmos implicados en la construcción. El escalonado en los tiempos, coincidente con la cantidad de niveles generados, se explica por el carácter recursivo del algoritmo: el número de nodos crece exponencialmente cada vez que se genera un nuevo nivel, en un orden aproximado de un 33% más que el nivel anterior, que es precisamente el tiempo que cae en cada escalón.

7.3 CARGA ASÍNCRONA DE CHUNKS EN LA CACHÉ L2

Uno de los grandes motivos por los que la caché de segundo nivel es asíncrona, es para que no influyan las interrupciones del kernel del sistema operativo ni su gestión del sistema de ficheros en la estabilidad de la arquitectura propuesta. Se podría pensar que, dado que los chunks son ficheros organizados en disco, de similar tamaño, su carga a memoria del sistema llevará siempre un tiempo determinado. En la Ilustración 37 se muestra una monitorización de 5 segundos.

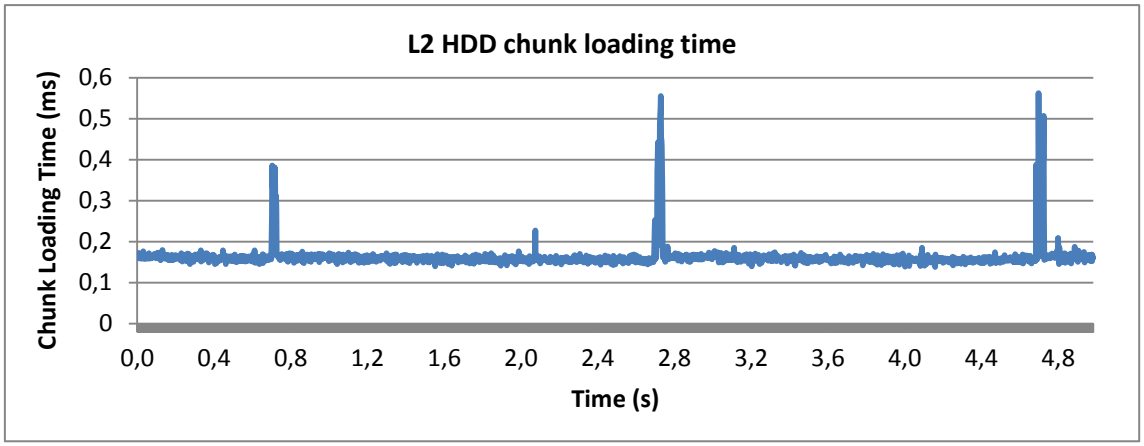


Ilustración 37. Gráfica de tiempos de carga de chunk, de HDD a RAM, por la caché L2 (M=5000).

Como se puede observar, si bien durante el lapso monitorizado, la mayor parte de las cargas tienen un tiempo estable (sobre 0,17 ms), existen unas claras fluctuaciones, que hacen doblar o incluso triplicar ese tiempo. Como decimos, son debidas principalmente al efecto *Buffer-Cache* del kernel del sistema operativo, y gracias a la arquitectura de dos niveles de caché de PCM, pasan inadvertidas.

7.4 RATIOS DE ACIERTO DE LAS CACHÉS

La forma más común de medir el rendimiento de una caché es mediante su tasa de aciertos, cociente entre el número de aciertos entre el número de peticiones. Se ha comentado anteriormente que el sistema propuesto tiene sobretodo en cuenta la estructura espacial de los datos. Un acceso a regiones "vecinas" de las anteriores, que podemos denominar "secuencial", debería ser mucho más eficiente que un acceso aleatorio.

En la siguiente tabla se muestran las tasas de acierto, y el tiempo de vida medio de un chunk en RAM, para un uso exclusivo de la caché L2, tanto en acceso secuencial como aleatorio, para los distintos modos *traversal* del árbol.

L2 Hit ratios					
Sequential Access			**Random Access**		
	Hit Ratio (%)	**Average Life (ms)**		**Hit Ratio (%)**	**Average Life (ms)**
TD	71.81	292.92	**TD**	44.23	186.04
BU	67.19	288.06	**BU**	37.23	178.55
SUSD	66.64	274.57	**SUSD**	35.43	175.43
SDSU	70.22	293.51	**SDSU**	40.23	188.63

En primer lugar, se puede decir que la caché funciona razonablemente bien, con unas tasas de acierto considerablemente altas. La caída de aciertos al pasar de acceso secuencial a aleatorio era la esperada, y por el mismo motivo, el tiempo medio que un chunk permanece en la caché. Por otra parte, es interesante comprobar que el comportamiento de la caché varía ligeramente en función del tipo de *traversal* escogido. El de mayor éxito es el Top-Down (TD), ya que en la lista de prioridades de los nodos siempre pone primero el raíz, y de ahí va construyendo la ruta hasta la hoja. Es previsible entonces, que **los nodos más altos del árbol se mantengan cargados siempre en caché, un comportamiento colateral del algoritmo y sumamente ventajoso**, porque siempre garantiza que hay un mínimo nivel de detalle cargado.

En la siguiente tabla, por último, se muestra las tasas de acierto, ya sólo en acceso secuencial, de ambas cachés funcionando al mismo tiempo. Se puede comprobar el efecto de la jerarquía, que hace que el mayor número de aciertos pase al primer nivel, y en el segundo, a donde llegan sólo los *requests* de nodos no encontrados en el primero, baje considerablemente.

L1 + L2 – Sequential access		
	L1 Hit Ratio (%)	**L2 Hit Ratio (%)**
TD	61.17	14.04
BU	49.45	7.71
SUSD	60.11	14.11
SDSU	48.44	7.50

8 CONCLUSIONES

En los capítulos dedicados a los fundamentos y estado del arte, se mostró que el campo del tratamiento de nubes de puntos es tremendamente útil en múltiples disciplinas, pero aun así un campo novel. Los software desarrollados por los fabricantes de equipamientos LiDAR y escáneres 3D permiten un conjunto de operaciones muy reducida y poco eficiente, y requieren potentes y costosas *workstations* para trabajar con *datasets* de tamaño elevado.

En este trabajo se ha presentado el paquete de software **Point Cloud Manager**, para el tratamiento de grandes *datasets* de nubes de puntos 3D, basado en un desarrollo teórico propio, consistente en un sistema de gestión de la jerarquía de memoria basado en cachés software de dos niveles, y una estructura de datos espacial multirresolución, que permite trabajar con *datasets* de un tamaño arbitrario en cualquier tipo de máquina, aprovechando la potencia de la GPU incorporada. Además, se han presentado un conjunto de herramientas, y un interfaz de programación que facilitan su utilización a los distintos perfiles de usuario. Para el desarrollo del proyecto, se ha elegido e implantado un sistema de seguimiento web, con control concurrente de versiones y base de conocimiento.

8.1 CUMPLIMIENTO DE REQUISITOS Y OBJETIVOS

En general, puede afirmarse que objetivos y requisitos de software planteados en el primer capítulo de este documento se han cumplido:

- Se ha realizado un estudio de las técnicas actuales capaces de gestionar este tipo de información, haciendo especial énfasis en el campo de los Gráficos por Computador, donde se han encontrado las más avanzadas.
- Se ha diseñado y construido una primera versión del sistema teórico y se ha demostrado su rendimiento y posibilidades.
- Se ha estructurado un paquete de software completo, teniendo en cuenta las necesidades específicas de los distintos roles de usuario, y capaz de trabajar con una variedad considerable de formatos de nube de puntos.
- Se ha elegido e implantado una plataforma web de seguimiento y desarrollo, con control concurrente de versiones y base de conocimiento

Además, el software generado resulta ser:

- Multiplataforma, gracias al uso de lenguajes estándar, librerías que a su vez también lo son, y a una cuidadosa metodología de trabajo.
- De fácil uso, debido a su arquitectura y diseño, definiendo de cara al usuario un API simple capaz de desacoplar el procesado de la nube de puntos de su gestión en memoria.

- Genérico, ya que permite trabajar con cualquier tamaño de dataset, en cualquier máquina, y para cualquiera que sea el formato de los datos adicionales incorporados en la nube de puntos.
- Extensible, gracias al uso de patrones de diseño, de forma que crear un nuevo conversor de formato, implementar una nueva técnica de construcción de la BBDD, o desarrollar un nuevo algoritmo de procesado, sea tan sencillo como heredar una clase y rellenar los métodos indicados.

Por otra parte, PCM es un proyecto ambicioso que aún se encuentra en una fase muy temprana de desarrollo, y desde luego, quedan muchas tareas por hacer. En este momento, la mayor necesidad del sistema es probablemente el testeo, por una parte con *datasets* reales procedentes de distintos dominios (ingeniería, edificación, topografía, patrimonio, etc.) y tecnologías (LiDAR aéreo, mobile mapping, escáneres 3D, etc.) para encontrar los posibles *bugs* o cuellos de botella que pueda tener. Por otra parte, de procesos que prueben los distintos lenguajes de GPGPU (CUDA, OpenCL) y que verifiquen que las técnicas de paralelización en GPU se pueden adaptar fácilmente al análisis multirresolución.

8.2 MEJORAS TÉCNICAS

Sin duda existen numerosas mejoras que se pueden (y deben) realizar en el sistema. Al tratarse de un prototipo de un desarrollo teórico nuevo, la experiencia de su diseño e implementación expone una serie escollos difíciles de haberse previsto. Entre todas ellas, se pueden destacar algunas inmediatas:

- Como se ha demostrado, el parámetro M (número máximo de puntos por nodo, en la estructura espacial) afecta demasiado a la construcción y funcionamiento del sistema, como para ser una opción que decida el usuario. Debe diseñarse una manera de automatizar la búsqueda de su valor óptimo.
- El sistema de predicción es tan sólo una primera aproximación, y a todas luces muy mejorables. Será necesario investigar la literatura para hallar métodos más avanzados que incrementen la ratio de acierto de las cachés.
- Actualmente, la estructura de datos espacial sólo soporta árboles binarios *perfectos*. A pesar de que se ha explicado que esto supone una ventaja de rendimiento, por otra parte, también crea una restricción a la hora de subdividir los sub-espacios de la nube de puntos, que siempre deben tener el mismo número de nodos a cada lado. En modelos con distribuciones de puntos muy heterogéneas, esto puede llegar a suponer un problema. Se debería buscar un método para, sin perder eficiencia, manejar árboles cuyas hojas caigan en distintos niveles (no perfectos).
- Las GPUs de los últimos años permiten nuevas formas de gestión de memoria, que podrían justificar hacer la caché de primer nivel L1 también asíncrona.
- Para datasets mayores de 64Gbytes, que es el tamaño máximo de archivo que pueden gestionar ahora mismo los sistemas de ficheros actuales, el formato propio BPC debe poder particionarse en varios ficheros.
- Paralelizar la construcción de la estructura espacial multirresolución.

- Desacoplar el tamaño en bytes de los chunks y los *BufferObjects*, para obtener el máximo rendimiento en la transferencia tanto a RAM como a VRAM.
- Estudiar una forma de integración con la librería PCL, para aprovechar toda su colección de procesos

8.3 FUTURO DE PCM

En las primeras páginas de este trabajo se comentaba que PCM nacía con un doble sentido: tener un marco para la investigación de técnicas y procesos en la gestión de grandes nubes de puntos, y una orientación comercial.

En el primer aspecto, a corto plazo, se proyecta realizar una publicación científica presentando las aportaciones de este sistema. A largo plazo, este campo se ve lo suficientemente interesante, amplio y no demasiado investigado ya, como para ser el inicio de una tesis doctoral.

El segundo aspecto, sobre la búsqueda de la explotación comercial del sistema, ya se encuentra en marcha. Actualmente existen dos propuestas con empresas del sector de la ingeniería, para tratar de afrontar proyectos que implicarán grandes escaneos y análisis LiDAR, aéreos por una parte, *y de mobile mapping* por otra. De llegar a acuerdo, PCM será la base de desarrollo del software de dichos proyectos, y en ese momento se decidirá el tipo de licencia más adecuada.

9 BIBLIOGRAFÍA

[1] G. Vosselman y H.-G. Mass, Airborne and Terrestrial Laser Scanning, 2010.

[2] E. Gamma, R. Helm, R. Johnson y J. Vissides, Design Patterns: Elemets of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994.

[3] Pixar Inc., «CGSociety - Pixar Points,» [En línea]. Available: http://www.cgsociety.org/index.php/CGSFeatures/CGSFeatureSpecial/pixar_points. [Último acceso: 2012].

[4] North Carolina University, Dep. Computer Science, «LAStools: software for rapid LiDAR processing,» [En línea]. Available: http://www.cs.unc.edu/~isenburg/lastools/. [Último acceso: 2012].

[5] Stanford University, «The PLY File Format,» [En línea]. Available: http://www.cc.gatech.edu/projects/large_models/ply.html. [Último acceso: 2012].

[6] D. Luebke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson y R. Huebner, Level of Detail for 3D Graphics: Application and Theory, Morgan Kaufmann, 2002.

[7] H. Samet, «Spatial Data Structures,» 2012.

[8] F. Pfenning, «Computer Graphics - Lecture 17: Spatial Data Structures,» Carnegie Mellon University, 2002. [En línea]. Available: http://www.cs.cmu.edu/~fp/courses/02-graphics/pdf-color/17-spatial.pdf.

[9] R. Sugumaran, D. Oryspayev y P. Gray, «GPU-based cloud performance for LiDAR data processing,» de *2nd International Conference on Computing for Geospatial Research & Applications*, 2011.

[10] F. Qiu y C. Yuan, «Parallel processing of massive LiDAR point clouds using CUDA enabled GPU,» de *AAG Annual Meeting*, 2012.

[11] M. Gross y H. Pfister, Point-Based Graphics, Morgan Kaufmann, 2007.

[12] S. Rusinkiewicz y M. Levoy, «QSplat: a multiresolution point rendering system for large meshes,» de *SIGGRAPH*, 2000.

[13] E. Gobbetti y F. Marton, «Layered Point Clouds,» de *Eurographics Symposium on Point Based Graphics*, Zurich, Switzerland, 2004.

[14] P. Goswami, Y. Zhang, R. Pajarola y E. Gobbetti, «High Quality Interactive Rendering of Massive Point Models using Multi-way kd-Trees,» de *18th Pacific Conference on Computer Graphics and Applications*, 2010.

[15] J. Taibo, GeoTextura: Una arquitectura software para la visualización en tiempo real de información bidimensional dinámica georreferenciada sobre modelos digitales 3D de terreno basada en una técnica de mapeado de texturas virtuales, A Coruña - España, 2009.

[16] Open Perception, «Point Clouds Library,» [En línea]. Available: http://pointclouds.org. [Último

acceso: 2012].

[17] E. Gobbetti y F. Marton, «Layered Point Clouds - a Simple and Efficient Multiresolution Structure for Distributing and Rendering Gigantic Point-Sampled Models,» *Computer & Graphics,* nº 28, pp. 815-826, 2004.

[18] R. Kerr, «Stack Overflow Forum,» 26 9 2010. [En línea]. Available: http://stackoverflow.com/questions/3572640/interview-question-find-median-from-mega-number-of-integers. [Último acceso: 2012].

[19] K. B. e. al., «Manifesto for Agile Software Development,» 2001. [En línea]. Available: http://agilemanifesto.org/.

[20] Edgewall Software, «The TRAC Project,» [En línea]. Available: http://trac.edgewall.org/. [Último acceso: 2012].

[21] R. Osfield, «Open Scene Graph,» [En línea]. Available: http://openscenegraph.org.