



UNIVERSIDADE DA CORUÑA

FACULTADE DE INFORMÁTICA

TRABALLO FIN DE GRAO

GRAO EN ENXEÑARÍA INFORMÁTICA

Mención en
Tecnoloxías da Información e Comunicaci3ns

Sistema multirresoluci3n de visualizaci3n 3D en tiempo real para grandes bases de datos LIDAR

Autor	Alberto Jaspe Villanueva
Director	Emilio J. Padr3n Gonz3lez

A Coru3a, Septiembre de 2013

*A mi familia y amigos,
que aún tan lejos están muy cerca*

Quiero agradecer en primer lugar, al director de este proyecto y buen amigo **Emilio J. Padrón**, por la confianza que siempre deposita en mí, y por toda la ayuda y apoyo que me ha prestado durante estos últimos años. Sin él este trabajo no hubiera sido posible.

Agradezco también a mis nuevos compañeros de trabajo en el **grupo de Visual Computing del CRS4** (Cerdeña – Italia), **Enrico, Fabio B., Fabio M., Marco, Antonio, Marcos, Gianni, Roberto, Alex, Emilio, Katia, Cinzia y Luca**, por acogerme y hacerme sentir como en casa.

Es mi deseo, además, agradecer particularmente a

- **Mis padres**, por tanto, tantísimo. Lo son todo para mí.
- **Raúl**, capaz de recorrerse más de 3.000 km en coche por un amigo. No lo olvidaré.
- **Fabio B.**, porque no se puede ser mejor persona, y porque me lo hace todo más fácil.
- **Mis hermanos, cuñados, tíos y primos**, por sus consejos y cariño incondicional, y por enseñarme lo importante que es la familia. La mía es la mejor.
- **Mis sobrinos**, la salsa de mi vida.
- **Javier, Julián, Rubén, Taibo, María, Brais, Isa(s), Liliana, Fátima y Carlos**, que aguantaron todas mis dudas en momentos muy difíciles para mí.
- **Jalley**, colega, amigo y confidente. Está siempre, en las buenas y en las malas.
- **Pau**, que cuida más de mi cuerpo que yo mismo.
- **Enrico G. y Fabio M.**, por compartir sus enormes conocimientos conmigo todos los días, y por hacerlo una forma tan honesta y humilde. Los mejores *research advisors* que se pueda desear.
- **Marcos**, por su cariñosa acogida en su casa a mi llegada a la isla.
- **Emma, Luca y Vale**, que me alegran y ayudan en mi nueva vida diaria.
- **Omar**, que está haciendo un gran trabajo.
- **Los integrantes de las reuniones de trabajo**, que tanto me han desestresado.
- **JuanRa**, gran investigador y mejor persona, comprensivo con los giros de la vida.
- **Quique y toda la gente de CEGA**, que siempre han confiado tanto en mí.
- **Manu**, por presentarme una isla maravillosa, por regalarme momentos increíbles que se quedarán para siempre en mi recuerdo, y porque la quiero.

A todos vosotros, gracias.

Alberto

RESUMEN

La tecnología **LIDAR** (*Laser Imaging Detection and Rangin*) consiste en el **escaneado con láseres de alta precisión para obtener un modelo digital 3D georreferenciado del mundo real**. Su uso se está estandarizando rápidamente en múltiples disciplinas de la ingeniería, ya bien sea en su versión aérea (escaneando la superficie terrestre desde un avión), terrestre (estacionado) o móvil (ubicado en un vehículo). Estos dispositivos generan un inmenso conjunto de *samples* con coordenadas e información asociada (como el color o el índice de reflectividad) que denominamos “**nubes de puntos**” o *point clouds*. La magnitud habitual de estos *datasets* es del orden de cientos o miles de millones de puntos, que se traducen en varios cientos de gigabytes de datos.

Conseguir una visualización 3D de estas nubes de puntos en PCs de escritorio requiere el uso de técnicas especiales denominadas multirresolución, basadas en estructuras de datos espaciales y jerarquías caché por software, así como complejos algoritmos de render programados a bajo nivel sobre las unidades de procesamiento gráfico o GPUs. Este proyecto pretende abordar este problema, como extensión de la librería Point Cloud Manager (PCM) creada por el mismo autor, realizando un estudio del estado de la cuestión y una implementación de visualización avanzada eficiente.

PALABRAS CLAVE

Nubes de puntos, LiDAR, escáner 3D, grandes *datasets*, real time render, visualización 3D, caché software, jerarquía de memoria, estructuras de datos espaciales, sistemas multirresolución, niveles de detalle, splat rendering.

ÍNDICE

Índice	1
1 Introducción	1
1.1 Motivación	2
1.2 Objetivos	4
1.3 Inspiración y continuación de trabajo.....	5
2 Contextualización y fundamentos	6
2.1 Nubes de puntos	6
2.1.1 Sistemas LiDAR de adquisición de nubes de puntos.....	7
2.1.2 Otras fuentes.....	9
2.1.3 Formatos	10
2.2 Render.....	11
2.2.1 Hardware gráfico (GPUs).....	13
2.2.2 Modelos tridimensionales.....	13
2.2.3 El Pipeline gráfico y Shaders	15
2.3 Sistemas multirresolución.....	17
2.4 Estructuras de datos espaciales	19
2.5 Jerarquías de memoria con GPUs	21
2.6 Cachés software	23
3 Estado del arte	25
3.1 Técnicas en investigación.....	25
3.2 Sistemas comerciales	28
4 Análisis general de la solución propuesta	29
4.1 Requisitos.....	29
4.2 Visión general de la solución.....	30
5 Estructura de datos espacial multirresolución	33
5.1 Árbol K-Dimensional (Kd-tree).....	33
5.2 Organización de los datos y niveles de detalle	35
5.3 Construcción de la estructura	38

5.3.1	Búsqueda del plano de corte: aproximación de la mediana.....	41
5.3.2	Selección pseudo-aleatoria de los puntos.....	42
6	Jerarquía de memoria	44
6.1	La caché asíncrona de 2º nivel (L2)	46
6.2	La caché síncrona de 1º nivel (L1)	49
6.3	Predicción.....	50
7	Visualización interactiva	52
7.1	Preparación de los datos.....	53
7.2	Test de nivel de detalle	56
7.3	Test de Culling (Frustum y backface)	57
7.3.1	View Frustum culling	58
7.3.2	Backface culling.....	60
7.4	Prioridad de las peticiones	60
7.5	Render en OpenGL.....	62
7.5.1	Render naive	62
7.5.2	Render avanzado: Splatting	63
8	El paquete de software PCM	69
8.1	Roles de usuario	69
8.2	Estructura	70
8.2.1	Herramientas (<i>PCM Tools</i>)	71
8.2.2	Tests unitarios (<i>PCM Test</i>).....	73
8.2.3	Librería pcm (<i>PCM Library</i>).....	74
8.3	Pipeline de uso del sistema.....	75
8.4	Seguimiento del proyecto	75
9	Diseño e implementación de PCM	78
9.1	Diagrama de clases global	78
9.2	Conversión y manejo de datasets	80
9.3	Estructura de datos	81
9.4	Jerarquía de memoria	83

9.5	Interfaz exterior	84
10	Resultados	87
10.1	Aproximación de la mediana	87
10.2	Construcción de la estructura espacial	88
10.3	Carga asíncrona de chunks en la caché L2	89
10.4	Ratios de acierto de las cachés	90
10.5	Estimación de normales	91
10.6	Datos de render	93
11	Conclusiones y futuros desarrollos	95
11.1	Cumplimiento de requisitos y objetivos	95
11.2	Mejoras técnicas	97
11.3	Futuro de PCM	98
12	Nota Legal	99
13	Bibliografía.....	100

1 INTRODUCCIÓN

El presente Trabajo Fin de Grado aborda el problema de la visualización tridimensional e interactiva de bases de datos LiDAR (nubes de puntos) con tamaño arbitrario. El documento se divide en los siguientes capítulos:

1. **Introducción**, donde se presenta un breve resumen del trabajo, 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, así como las distintas formas de abordar el *render*.
4. **Análisis de la solución propuesta**, donde se ofrece una breve explicación general de la solución planteada, así como unos requisitos propuestos.
5. **Estructura de datos espacial multirresolución**, donde se entra ya en materia funcional, y se comenta cómo es la estructura de datos, sus características de multirresolución y de niveles de detalle, así como su construcción.
6. **Jerarquía de memoria**, en el se propone un nuevo modelo de memoria ampliado hacia las tarjetas gráficas o GPUs, y dos cachés por software que aprovechen las características espaciales de los datos a tratar.
7. **Visualización interactiva**, donde se explicará cómo se realiza el render en tiempo real con el API para hardware gráfico OpenGL.
8. **El paquete de software PCM**, en se presenta la arquitectura de la solución software desarrollada, sus herramientas y funcionalidades, y el pipeline de trabajo para tratar con nubes de puntos.
9. **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.

10. **Resultados**, para exponer algunos datos significativos del funcionamiento del sistema.
11. **Conclusiones y futuros desarrollos**, donde se explicarán la proyección del proyecto, sus líneas de trabajo futuras, y en qué forma se han cumplido los objetivos expuestos.
12. **Bibliografía**, con referencias bibliográficas a los documentos más importantes en este tema.

1.1 MOTIVACIÓN

En los últimos años los sistemas de adquisición de datos tridimensionales (comúnmente denominados LiDAR, *Laser Imaging Detection And Ranging*) ha estado evolucionando de tal forma que ha causado una revolución en los métodos de trabajo de diversas disciplinas, desde la ingeniería civil e industrial, la arquitectura, la conservación del patrimonio cultural, hasta el cine o los videojuegos. Estos sistemas, como se verá en el siguiente capítulo, utilizan haces de luz láser y sensores fotográficos para muestrear las superficies de entornos espaciales, como objetos, habitaciones, edificios, calles, o ciudades enteras. Cada muestra es situada con una cierta precisión en un marco de referencia espacial virtual, mediante unas coordenadas (x,y,z) . Además, a cada una de ellas se le asocia una cierta cantidad de información, dependiendo de su propósito, como puede ser el color del objeto en esa muestra, su vector normal, índice de reflectividad de la luz, temperatura, etc. A la base de datos de una sesión de escaneo se le suele denominar *nube de puntos* o *point cloud*, y su tamaño es completamente arbitrario, dependiendo de las características del dispositivo y del número de escaneos, aunque se puede decir que actualmente, llegan con facilidad a las miles de millones de muestras. Se puede aseverar que la tendencia de crecimiento de estos *datasets* resulta ser, como mínimo, tan pronunciada como las capacidades estándar de memoria de sistema (RAM) de los PCs, pero multiplicando éstas en varios niveles de magnitud [1].

Una nube de puntos, por lo tanto, es el modelo digital de una realidad espacial, en un instante determinado del tiempo. Poder disponer de esta BBDD desde un computador

tiene innumerables ventajas y aplicaciones en una gran cantidad de ámbitos. Algunos ejemplos son:

- En **Ingeniería Civil**, se utilizan en todo tipo de simulaciones y cálculos para decir cómo acometer una obra, como planificar el trazado de una carretera (movimientos de tierra, taludes, desmontes, etc.), conocer el estado de conservación de una estructura, inventariado de elementos, etc. La ganancia en eficiencia, precisión y costes de este tipo de técnicas ha demostrado enorme respecto a la topología clásica.
- En **Ingeniería Industrial**, es ya prácticamente el estándar *de facto* para prototipado.
- En **Arquitectura**, se utiliza para cálculos de carga estructural, conservación de fachadas, simulaciones 3D para toma de decisiones de diseño, etc.
- En la **Conservación del Patrimonio Cultural**, también es prácticamente un estándar. Al tratar de clasificar y mantener objetos *únicos*, los escaneos 3D suponen una *copia digital* del bien cultural, de forma que su posterior estudio se puede realizar sin dañar el objeto real, por varios investigadores a la vez desde cualquier lugar del mundo.
- En **Cine**, ya que son la base de multitud de los efectos especiales donde se mezcla imagen real con imagen generada digitalmente.

En líneas generales, se pueden sintetizar las ventajas de esta técnica en dos grandes grupos. Por una parte, **la reducción de costes**, ya que en multitud de ocasiones, una nube de puntos sustituye acciones u operaciones que se deben tomar con instrumentos específicos, con su correspondiente desplazamiento de personal y material. Por otra parte, disponer de esta *copia* de forma digital permite **la aplicación de la ciencia y potencia computacional que disponemos actualmente para todo tipo propósitos**.

En prácticamente cualquiera de los casos de uso anteriormente descritos, el primer paso para poder trabajar con los datos es su visualización interactiva. Este término hace referencia al denominado *real-time render*, es decir, a la generación de imágenes con técnicas computacionales a una velocidad tal que el usuario tenga completa libertad para moverse

por la escena tridimensional para poder apreciarla desde cualquier punto de vista¹. Actualmente cualquier computadora personal viene dotada con un chip diseñado con este propósito (GPU) y con su propia memoria gráfica (*Video RAM* o *VRAM*). Pero existen **dos grandes dificultades** que impiden poder utilizar estos componentes de una manera estándar con nubes de puntos. Por una parte, **las primitivas de dibujo que las GPUs procesan habitualmente son polígonos, no puntos**. Por la otra, **el tamaño de los datasets obtenidos por dispositivos LiDAR pueden fácilmente multiplicar x100 o incluso x1000 tanto la memoria gráfica (VRAM) como la memoria del sistema (RAM)**.

Este proyecto estudia la problemática anteriormente descrita en todas sus fases y ofrecer una solución software eficiente, desde el tratamiento de estas bases de datos, la generación de estructuras multirresolución, la implementación de una nueva jerarquía de memoria y cachés software, hasta la visualización interactiva con técnicas avanzadas de render.

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*, para su posterior visualización en tiempo real utilizando técnicas de render avanzadas. 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. Análisis y estudio del estado del arte sobre el *render* interactivo de nubes de puntos, e implementación propia.

¹ Comúnmente se considera un *render* interactivo cuando se generan a partir de 25 *frames* por segundo.

4. Desarrollo de un paquete software que incluya las herramientas necesarias para convertir, procesar y visualizar los *datasets* de información provenientes de escáneres laser LIDAR u otras fuentes, utilizando el sistema descrito en punto anterior.
5. 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 de *render*.

1.3 INSPIRACIÓN Y CONTINUACIÓN DE TRABAJO

Este trabajo se presenta como una continuación o extensión de la librería Point Cloud Manager (PCM), introducida por el mismo autor en [3]. En esta tesis fin de máster se desarrolló un sistema capaz de trabajar con nubes de puntos de tamaño arbitrario, basándose en una estructura multirresolución inspirada en el campo de Computer Graphics. Este sistema trataba de aprovechar la naturaleza espacial de los datos, así como las técnicas de GPGPU posibles hoy en día en las tarjetas gráficas de consumo, para aplicar algoritmos sobre el *dataset*, como una forma de *High Performance Computing*.

El presente documento demuestra que esta estructura puede ser también válida para la visualización avanzada de nubes de puntos. Basándose en PCM, la estructura multirresolución puede ser adaptada para poder utilizar las partes programables de las GPUs y realizar un render de los paquetes de información que maneja.

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 así como de las las jerarquías de memoria en las máquinas actuales (incluyendo las nuevas unidades de procesamiento gráfico o *GPUs*) y el papel jugado por las cachés software. Por último, se comentarán las bases del *render* 3D en tiempo real para la visualización, y el papel de los *datasets* compuestos exclusivamente por puntos sin relaciones topológicas.

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.

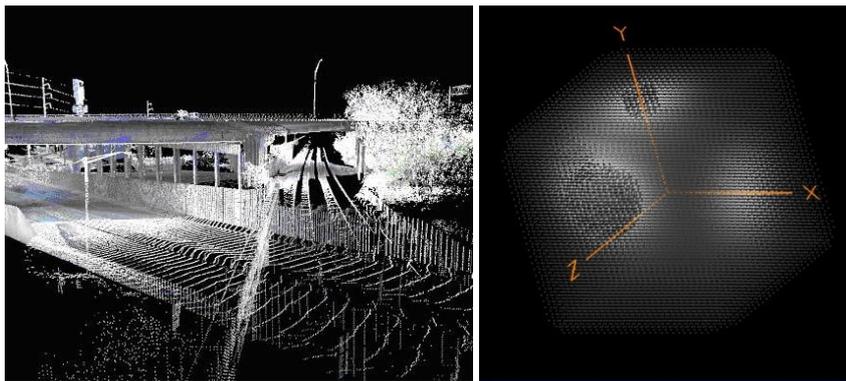


Ilustración 1. Visualización de nubes de puntos y su sistema de coordenadas.

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, reflecti-

vidad 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

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

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.

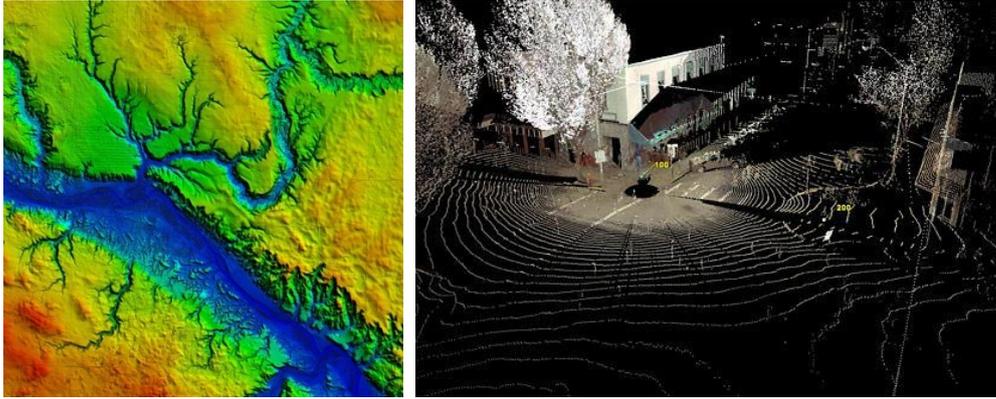


Ilustración 2. A la izquierda, una nube de puntos (como mapa de alturas, por colores) típica de un LIDAR aéreo. A la derecha, un entorno urbano adquirido con LIDAR terrestre.

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 un equipamiento 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.

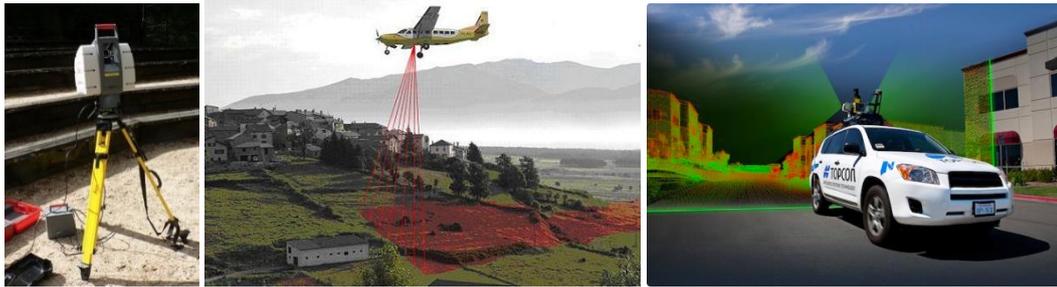


Ilustración 3. A la izquierda, un equipo de LIDAR terrestre de la marca Leica. En la imagen central, visión explicativa de un vuelo LIDAR. A la derecha, un sistema de de *mobile mapping* (IPS2 de la casa Topcon).

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 [4].

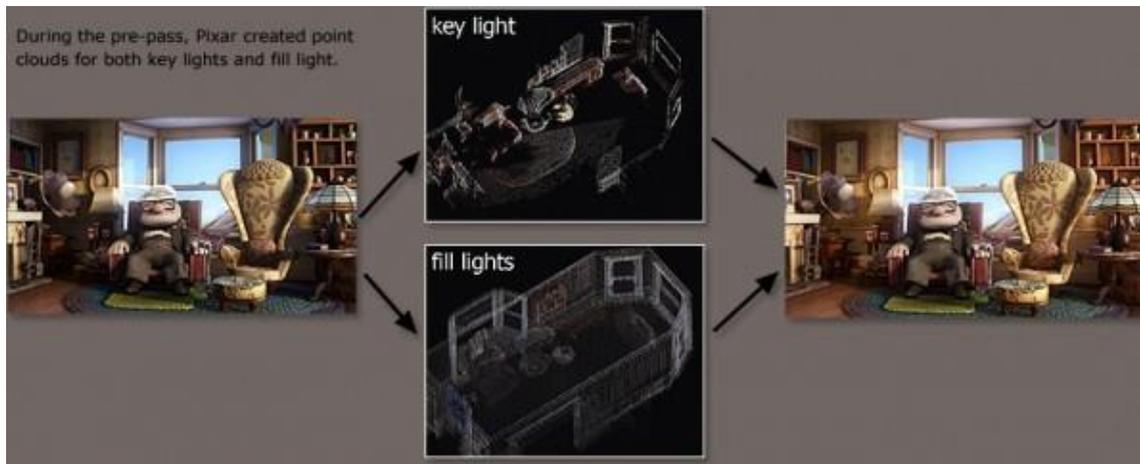


Ilustración 5. Nubes de puntos utilizadas como parte del proceso de iluminación de la compañía Pixar en la película "Up" (Fuente: Pixar Inc.)

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 [5].
- **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 [6].

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

2.2 RENDER

De forma general, se denomina render al proceso computacional de generación o síntesis de imágenes. Se puede afirmar que el estudio de dicho proceso es el núcleo de los Gráficos en Computación. Normalmente, dado un modelo 3D digital y una posición de la vista, este proceso consiste en aplicar ciertos algoritmos para obtener la imagen que se vería desde dicha posición. Existen numerosas tipos de render, según la naturaleza de los algo-

ritmos con los que trabaja, la intención de realismo, su propósito, los modelos que sintetiza, etc. Para este trabajo, es importante remarcar la siguiente diferencia:

- **Render *off-line*:** es aquel en el cual la visualización del resultado se produce *a posteriori* del proceso. Normalmente se trata de complejas simulaciones de luz sobre modelos tridimensionales de alta precisión, y requiere unas grandes capacidades de memoria y computacionales. La generación de cada imagen se suele medir en horas, y se llegan a utilizar *clusters* denominados *granjas de render*. Es el utilizado típicamente para conseguir infografías de arquitectura o películas de animación.
- **Render interactivo, *real-time* u *on-line*:** es aquel capaz de suministrar en tiempo real imágenes al usuario, de forma que éste tiene la capacidad de decidir en cada momento parámetros de la simulación, como la posición de la cámara, iluminación de la escena, etc. Para *engañar* al cerebro humano se consideran necesarias un mínimo de 12 imágenes por segundo (comúnmente denominado *frames por segundo*, o *fps*). Aunque el estándar en cine o TV suele ser de 25 o 30 fps, debido al ratio de actualización típico de los actuales dispositivos de visualización de los ordenadores, en Gráficos en Computación se considera un estándar los 60 fps. Esto quiere decir que el proceso tiene aproximadamente 16 ms para generar cada imagen. Este es el tipo de render utilizado en software de diseño (CAD o 3D), en simuladores y videojuegos.



Ilustración 6. Ejemplos de render *off-line*, fotorrealista (izquierda) y render interactivo (derecha). Puede apreciarse la diferencia de detalle e iluminación de la escena. El primero de ellos puede precisar varias horas de procesado, mientras que el segundo solamente algunas decenas de milisegundos en una GPU (Fuente: greyscalegorilla.com y dudka.cz)

Por supuesto, existen grandes diferencias a nivel computacional entre ambos tipos de render. Mientras el primero (off-line) puede disponer de grandes recursos tanto de procesado (enormes tiempos de CPU) como de memoria (puede realizar *swapping* entre memoria del sistema y almacenamientos persistentes), la segunda (*realtime*) se ve limitada al uso de la GPU durante unas decenas de milisegundos, y a la memoria que incluya la tarjeta de vídeo.

Este proyecto se centra en el render interactivo, para poder ser utilizado en software que trabaje con datos LIDAR en tiempo real, donde los usuarios puedan ver todos los detalles del escaneo, y tomar decisiones o aplicar operaciones sobre el mismo.

2.2.1 HARDWARE GRÁFICO (GPUS)

Hoy en día, cualquier ordenador personal posee hardware específico para conseguir este propósito, denominada comúnmente *tarjeta gráfica* o *tarjeta de vídeo*, compuesta de un chip de proceso (*Graphics Processing Unit* o *GPU*), una memoria propia (*Video RAM* o *VRAM*) y un interfaz contra el sistema (actualmente PCI-e). Este hardware está específicamente diseñado para procesar datos gráficos y tridimensionales, mediante dos APIs:

- **DirectX**, API propietario de Microsoft, válido sólo en sistemas Windows.
- **OpenGL**, estándar y multiplataforma².

Cualquiera de estos dos APIs definen unos tipos de datos tridimensionales y una forma estándar de procesarlos (implementada físicamente en el hardware) denominada *pipeline*. Se comentan separadamente en las siguientes subsecciones.

2.2.2 MODELOS TRIDIMENSIONALES

Como se acaba de mencionar, los chips gráficos (GPUs) tienen una implementación standard fijada en hardware para trabajar con una determinada descripción de datos tridimensionales, las *meshes*. Informalmente se pueden definir como *mallas* de triángulos que

² Siguiendo los objetivos planteados en el proyecto de uso de estándares y soporte multiplataforma, el API escogido en este desarrollo es OpenGL. En adelante, todas las referencias al API gráfico serán específicas de OpenGL.

crean objetos. Están formadas básicamente por vértices (puntos en el espacio) y caras o *faces*, que definen como se conectan los vértices en conjuntos de tres, formando triángulos. En la siguiente figura (Ilustración 7) se puede observar como ejemplo la descripción en esta forma de un cubo. La lista de vértices, cada uno con sus coordenadas (x,y,z), y la lista de los 12 triángulos, dos por cada cara del cubo. Esta información es *proyectada* al plano de la pantalla, y los píxeles son rellenados con el color asignado a cada cara.

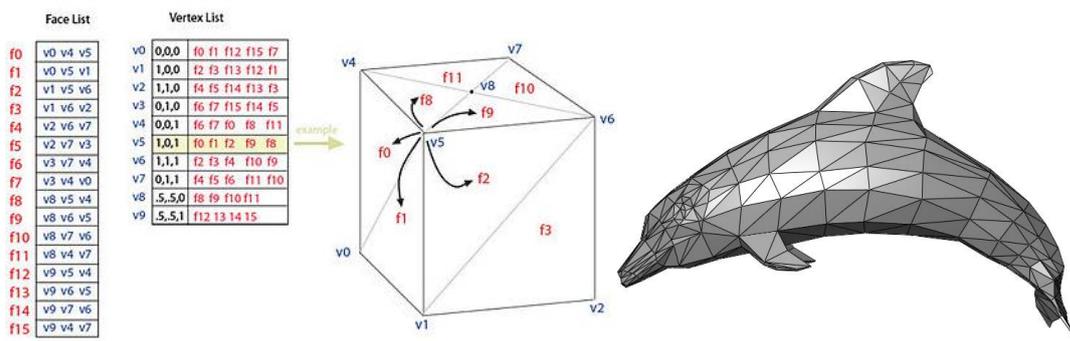


Ilustración 7. Meshes: en la figura de la izquierda, ejemplo de descripción de un cubo a partir de vértices y caras. A la derecha, una *mesh* básica completa (delfín). (Fuente: Wikipedia)

Para conseguir un acabado más detallado sobre las *meses*, se utilizan las texturas. Éstas son imágenes que, de alguna manera, *recubren* el objeto según un mapeado de cada vértice al pixel que le corresponda (Ilustración 8).

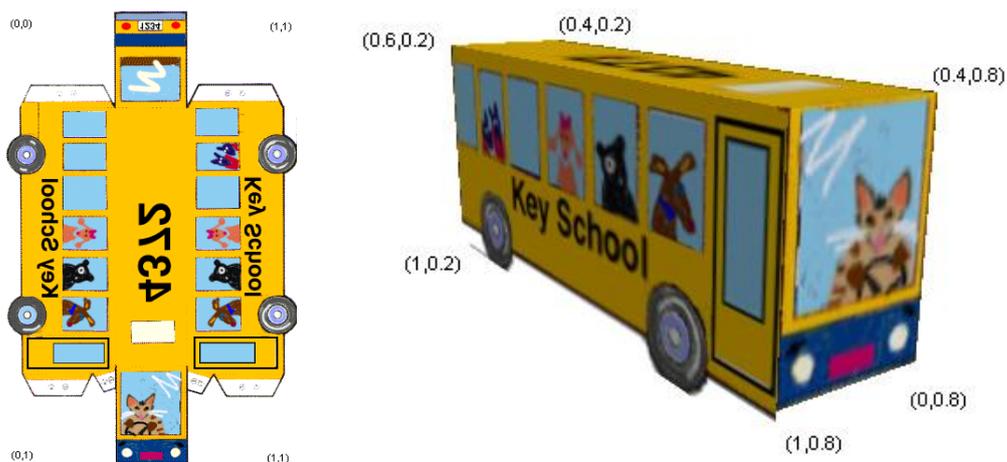


Ilustración 8. Mapeado de texturas sobre meshes (Fuente: resources.esri.com)

Toda esta información (descripción de las meshes y texturas) debe transferirse desde la memoria RAM del sistema a la memoria de la GPU (VRAM) en *arrays* de datos, para poder generar el render interactivo. Esto conlleva dos grandes problemas: por una parte, la velocidad del bus de datos entre estos dos componentes es limitada, así que se debe optimizar el número y tamaño de las transferencias para que el usuario no experimente “parones” o bajos *framerates*. Por otra parte, el tamaño de la memoria VRAM suele ser entre 8x y 20x veces menor que la memoria RAM (por ejemplo, un sistema con 4GBytes en RAM y 256MB en VRAM). Esto limita todavía más la cantidad de datos que se pueden mostrar.

Sin embargo, las GPUs no están preparadas *de facto* al tipo de datos que atañe a este trabajo, las nubes de puntos. Como se ha comentado ya, éstas no disponen de ninguna relación topológica, es decir, que los puntos no están conectados entre ellos, así que no forman los triángulos que la GPU necesita para poder realizar la visualización. Como se verá en el capítulo dedicado a render, la GPU puede dibujar cada punto como un simple vértice, dándole un radio de píxeles. Esta será la visualización *naive* de los datos LiDAR, que mejorará gracias a las posibilidades de programación de los nuevos *pipelines* gráficos.

2.2.3 EL PIPELINE GRÁFICO Y SHADERS

El *pipeline* de una GPU es el proceso implementado en hardware capaz de transformar la información vista en la subsección anterior en imágenes, de una forma muy eficiente. Consiste en una serie de etapas donde se realizan operaciones de álgebra lineal que tienen además en cuenta la posición, rotación y escala del punto de vista, objetos y luces, para definir finalmente de qué color dibujar cada píxel de la pantalla.

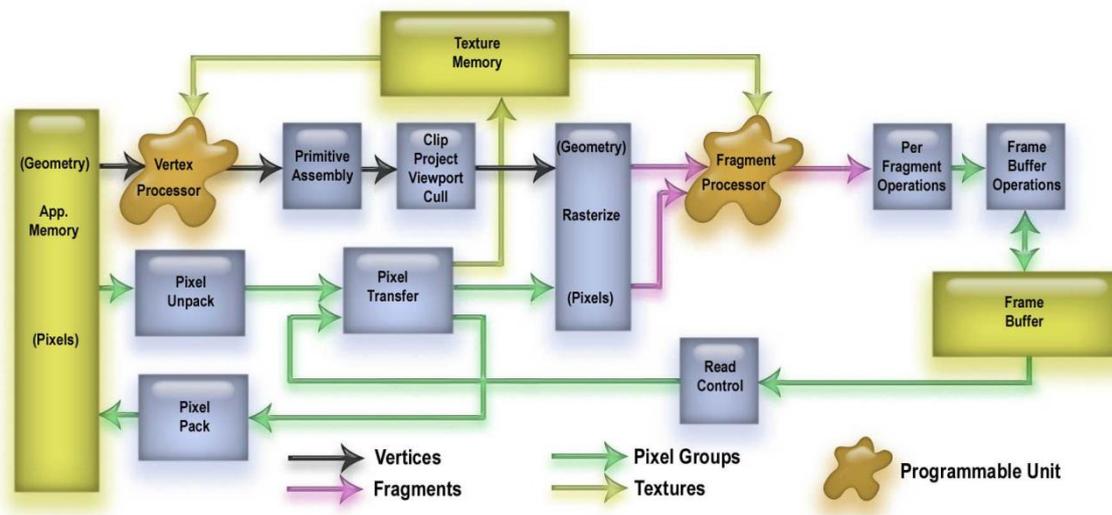


Ilustración 9. El pipeline de OpenGL (Fuente: "The OpenGL shading" book).

En las GPUs modernas, algunas de las etapas del *pipeline* se pueden modificar, mediante pequeños programas escritos en lenguaje GLSL (*GL Shading Language*), denominados *shaders*. En la anterior figura (Ilustración 9) se puede observar el *pipeline* de una versión moderna de OpenGL, donde los cuadros amarillos hacen referencia a *buffers* de datos, los azules a etapas *fijas* de procesado, y las moldeadas en forma de estrella, a etapas programables. El *Frame Buffer* es la imagen final que se mandará a la pantalla.

A pesar de las grandes limitaciones que existen a la hora de programar *shaders* (número de instrucciones, *ifs* y *loops* muy limitada, sólo algunas operaciones matemáticas, variables sin persistencia, etc.), éstos han provocado una gran revolución en el render interactivo, pasando de la cuestionable calidad visual que ofrecía el antiguo hardware *fijo* a imágenes fotorrealistas comparables al render off-line, en la mayor parte de los casos mediante *trucos algorítmicos* que imitan simulaciones complejas.



Ilustración 10. Diferencia de calidad entre un render con el clásico pipeline fijo tipo T&L (izquierda) y otro utilizando un pipeline programable con vertex y fragment shaders (derecha) (Fuente: electriccartilage.files.wordpress.com)

A pesar de que hoy en día existen más partes programables en las GPUs, como los geometry shaders, los principales y más utilizados son:

- **Vertex shaders:** es una función que recibe como parámetro un vértice. Sólo trabaja con un vértice a la vez, y no puede eliminarlo, sólo transformarlo. Para ello, modifica propiedades del mismo para que repercutan en la geometría del objeto al que pertenece.
- **Fragment shaders:** no interviene en el proceso de la definición de la geometría de la escena, sino que forma parte de la segunda etapa: la rasterización. Allí es donde se aplican las texturas y se tratan los píxeles que forman parte de ellas. Básicamente, un fragment shader especifica el color de un píxel.

Estas etapas programables correrán un papel fundamental a la hora de implementar un render de calidad de las nubes de puntos.

2.3 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 **LODs**), 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 multiresolución y niveles de detalle** constituyen una práctica habitual para casi cualquier motor de render [7]. 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 11.



Ilustración 11. Niveles de detalle (LODs). A la izquierda, sobre un modelo 3D poligonal. 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 original 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.

2.4 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 Ilustración 11 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.

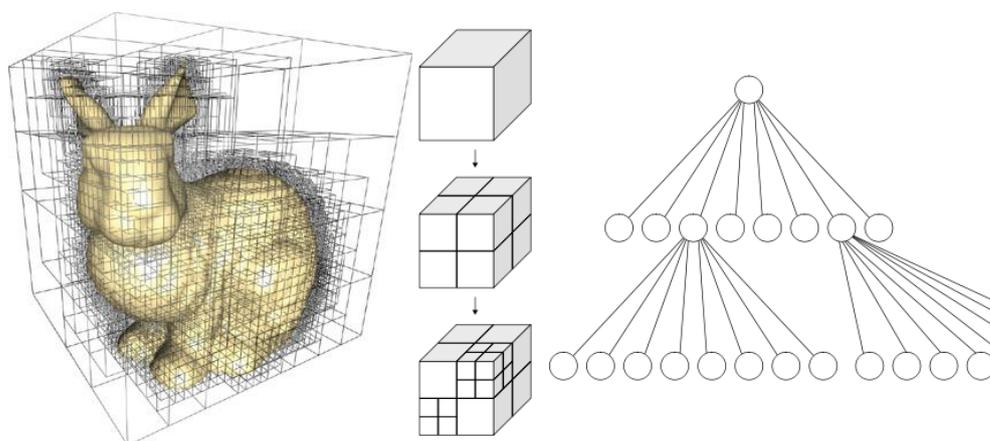


Ilustración 12. Subdivisión espacial recursiva para generar una estructura tipo árbol, en este caso, de tipo octree (Fuente: GPU Gems 2 y Wikipedia)

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 Pfening*, de la *Carnegie Mellon University* [8], y para profundizar más en el campo, el artículo de *Hanan Sarnet* de la *University of Maryland* [9].

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 *kd-tree*. En el siguiente capítulo se estudiará más en detalle.

2.5 JERARQUÍAS DE MEMORIA CON GPUS



Ilustración 13. Jerarquía de memoria clásica.

Se denomina “jerarquía de memoria” de un sistema computarizado 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 me-

moria por la CPU en direcciones relativamente cercanas entre sí.

Clásicamente se representa como en la Ilustración 13, 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"³ se puede ver esquemáticamente en la Ilustración 14.

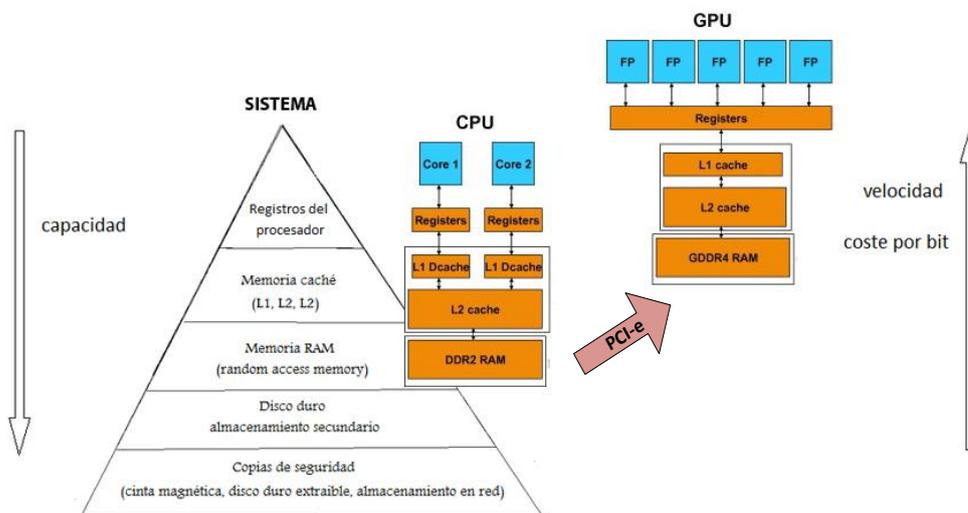


Ilustración 14. Modelo "revisado" de jerarquía de memoria en máquinas equipadas con GPU.

³ Tómese este modelo extendido como una simple licencia del autor, con fines explicativos.

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. Pero es además este tipo de paquetes de datos los que permiten actualmente a las GPUs *renderizar* eficientemente la información que necesitamos.

2.6 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 casuís-

tica 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 libres y comerciales más relevantes en el momento actual, que utilicen arquitecturas multirresolución para la visualización de nubes de puntos.

3.1 TÉCNICAS EN INVESTIGACIÓN

Existen numerosos artículos de investigación sobre procesado y render de nubes de puntos. Pero normalmente se centran en un problema particular (visualización, 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) [10], donde realizan una comparación de rendimiento entre algoritmos 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) [11], 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 bibliografía se puede encontrar sobre este tema. Mientras que el campo de GPGPU es de relativa nueva crea-

ción, 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 [12].

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 [13]. 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*.

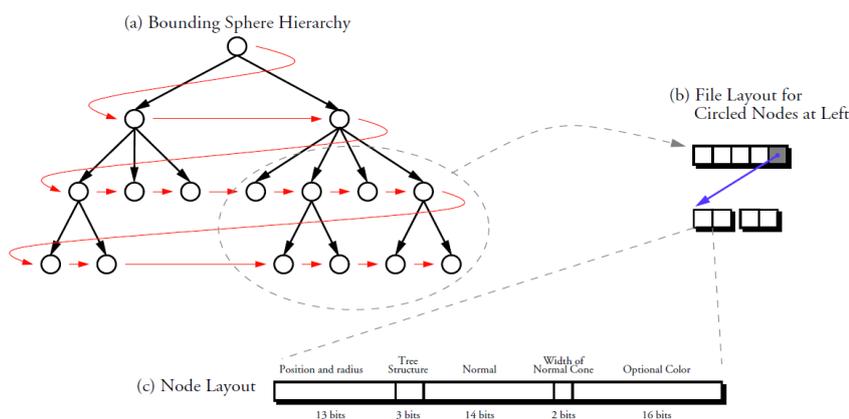


Ilustración 15. Estructura espacial de bounding-spheres del sistema QSplat [13].

Unos años más adelante, en el 2003, *Enrico Gobbetti* y *Fabio Marton* (ViC – CRS4, Italia) presentan la técnica de **“Layered Point Cloud”** [14], 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) [15] 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.

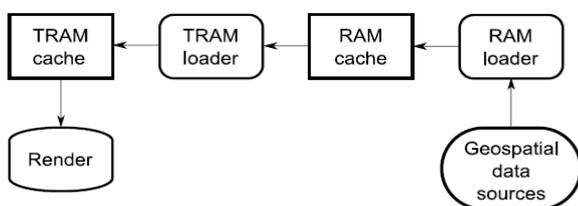


Figure 1: Overall architecture.

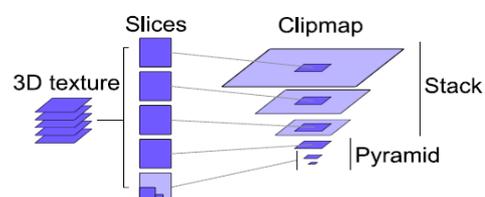


Figure 2: Clipmap structure on a 3D texture.

Ilustración 16. 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*” (Javier Taibo, UDC – España), presentada en el año 2010 [16]. 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 16 se puede observar el diagrama representativo de su sistema, muy similar al utilizado en este trabajo, así como una representación del *dataset* de texturas *georreferenciadas*.

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 procesamiento 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 simples, además de ser arquitecturas cerradas que no admiten la programación de nuevos algoritmos de procesamiento y visualizaciones bastante deficientes. Sin embargo, están surgiendo algunas empresas que tratan de llevar el ámbito de la investigación a su uso comercial, con software innovador que si implementa tecnologías más avanzadas:

- *Pointools* de POINTTOOLS Ltd
- *Gexcel R³* de GEXCEL
- *Dielmo* software de DIELMO
- *MARS* de MERRICK & COMPANY
- *Cyclon* de LEYCA

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)** [17]. 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. 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 GENERAL 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

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. La visualización de este *dataset* debe tener los siguientes requisitos de alto nivel:

- La visualización debe poder **funcionar en cualquier PC**, independientemente de su potencia y características.
- La visualización debe **consumir siempre los mismos recursos** (dependiendo de las capacidades de la máquina o de lo señalado por el usuario), **independientemente del tamaño o características del *dataset***.
- La **calidad del render debe ser progresivo**.
- El render debe soportar, al menos, el **color original** obtenida del escaneo LiDAR.
- Debe tener **soporte para distintos formatos de nube de puntos**, y extensibilidad a nuevos.

Para ello se desean los siguientes requisitos de sistema:

- **La información debe ser accesible de forma rápida, independientemente del tamaño del dataset o de las características de la máquina**, aprovechando toda la potencia de los distintos chips que ésta provea.
- 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 *J. Taibo* [16] (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.

En la Ilustración 17 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***.

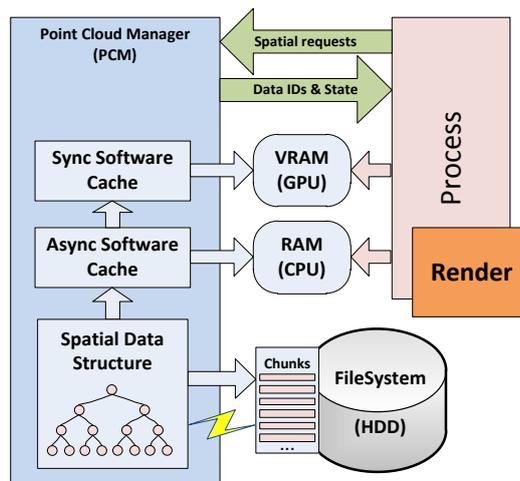


Ilustración 17. Diagrama abstracto de la solución propuesta.

Lo **chunks**, esas 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 refinamiento 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*⁴.

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, sino que son recibidos y gestionados por la caché L2, el sistema sería tolerante a fallos de comunicación.

⁴ 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.

5 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.

5.1 ÁRBOL K-DIMENSIONAL (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 18: 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 recursivamente 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.

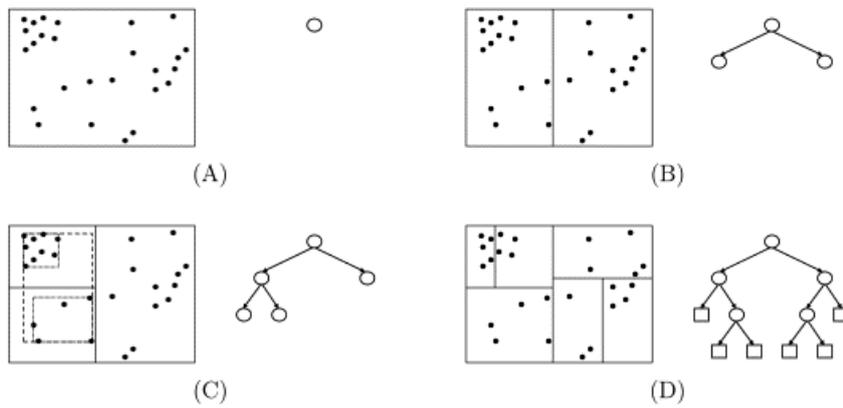


Ilustración 18. Construcción de un kd-tree. (A) El nodo raíz contiene el volumen completo. (B) El espacio original se particiona en dos mitades por el eje más largo. (C) y (D) el proceso se repite recursivamente hasta que el número de puntos de un nodo sea menor que un parámetro M dado, convirtiéndose 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 19. Simplemente se debe escoger en cada nodo del árbol el camino que quede en el lado adecuado de su plano de corte.

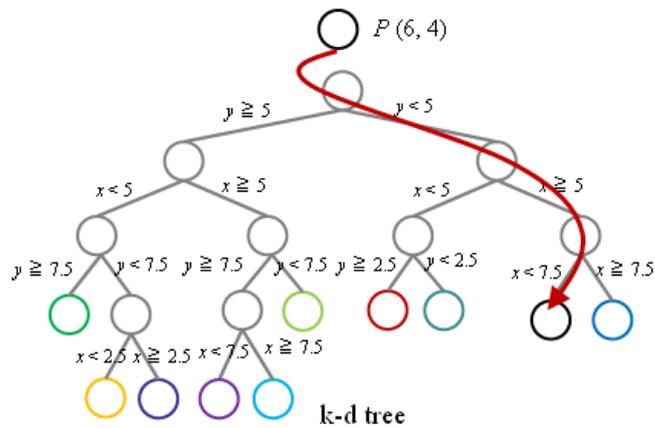


Ilustración 19. *Traversal* de un kd-tree (Fuente: frytail.seesaa.net)

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 garan-**

tizar que el árbol generado sea, además de binario completo, perfecto⁵. 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 20).

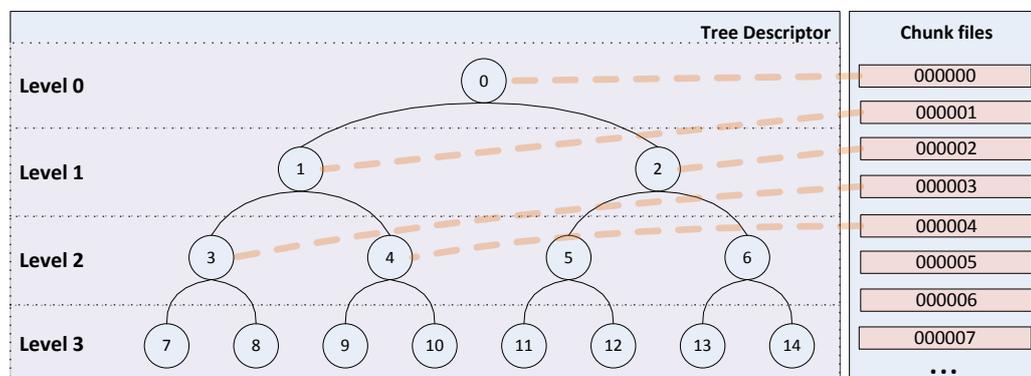


Ilustración 20. Árbol binario perfecto y su correspondencia lineal con los *chunks* de datos.

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:

Dado el nodo i y el nivel n ...

$$\text{Padre}(i) = \frac{i - 1}{2}$$

$$\text{N}^{\circ} \text{ Nodos Acumulados}(n) = 2^{n+1} - 1$$

$$\text{Hijo Izquierdo}(i) = 2i + 1$$

$$\text{N}^{\circ} \text{ Nodos Nivel}(n) = 2^n$$

$$\text{Hijo Derecho}(i) = 2i + 2$$

$$\text{Primer Nodo del Nivel}(n) = 2^n - 1$$

$$\text{Nivel}(i) = \log_2(i) + 1$$

5.2 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.

⁵ 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.

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*” [14], 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 añade 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 21.

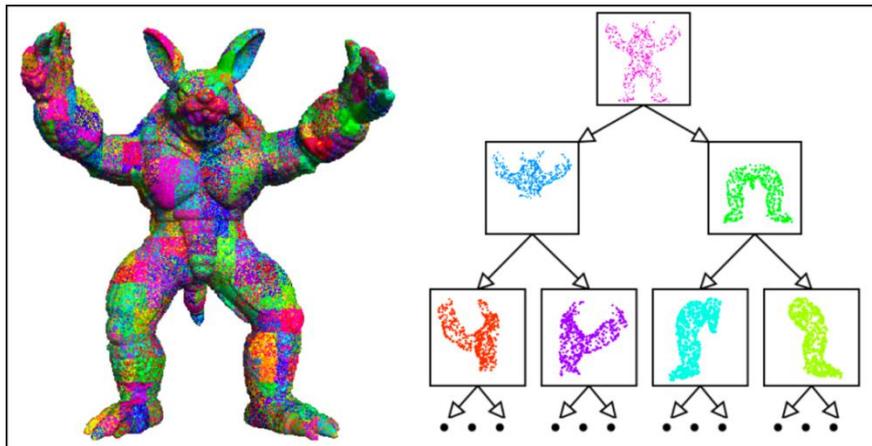


Ilustración 21. Distribución de los datos para generar niveles de detalle *aditivos*, siguiendo el esquema propuesto por E. Gobbetti y F. Marton en [18].

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 22).

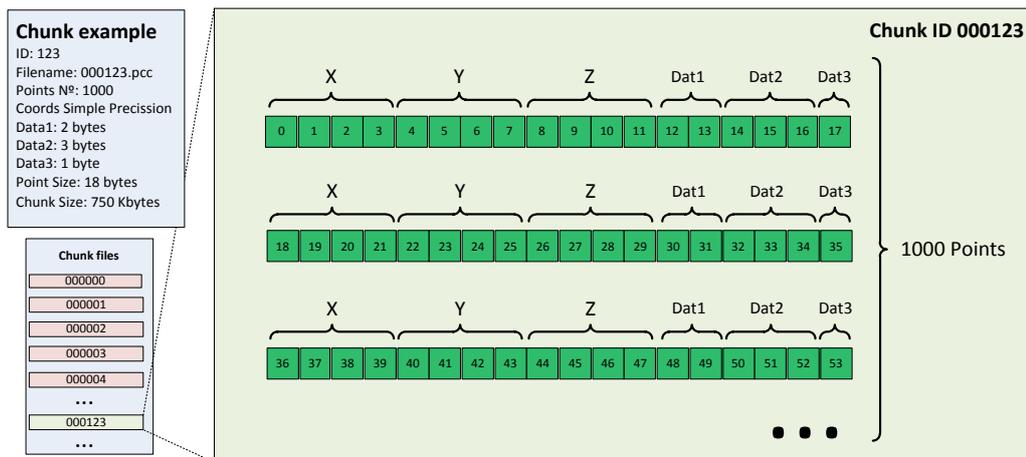


Ilustración 22. Estructura de un *data chunk*.

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.

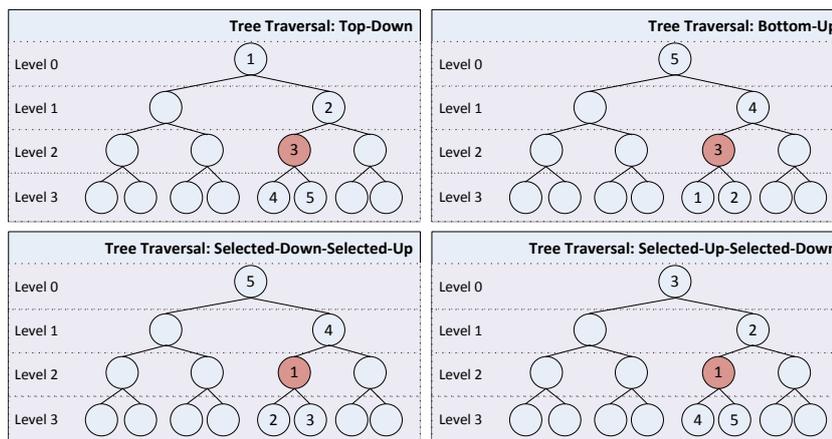


Ilustración 23. Modos de traversal definidos en el árbol.

5.3 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 20), 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 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 . Recuerde-mos, 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 $\leq M$ (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 día⁶.

5.3.1 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.

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* [19]. *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

⁶ Y así lo predijo ya Bill Gates, CEO de Microsoft, a mediados de los años 80's: «640KB deberían ser suficientes para cualquiera...»

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.

5.3.2 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 24.

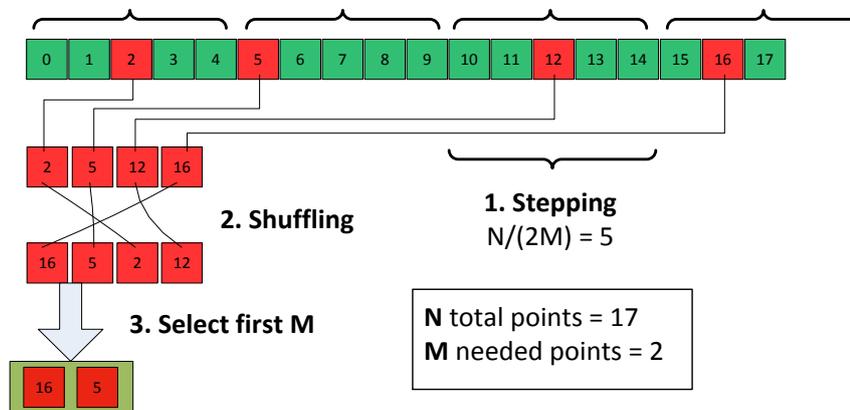


Ilustración 24. Método pseudo-aleatorio de selección de puntos para un nodo.

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.

6 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 25). 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.

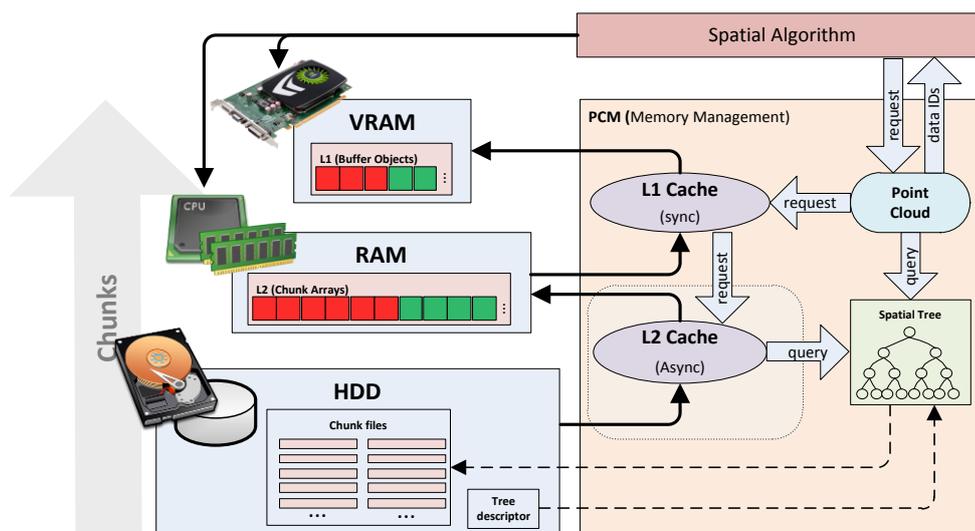


Ilustración 25. Sistema de gestión de la jerarquía de memoria.

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 al HDD).

te 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.

6.1 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 [16], 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*⁷ y mantener la coherencia en los datos) como se verá en su correspondiente capítulo.

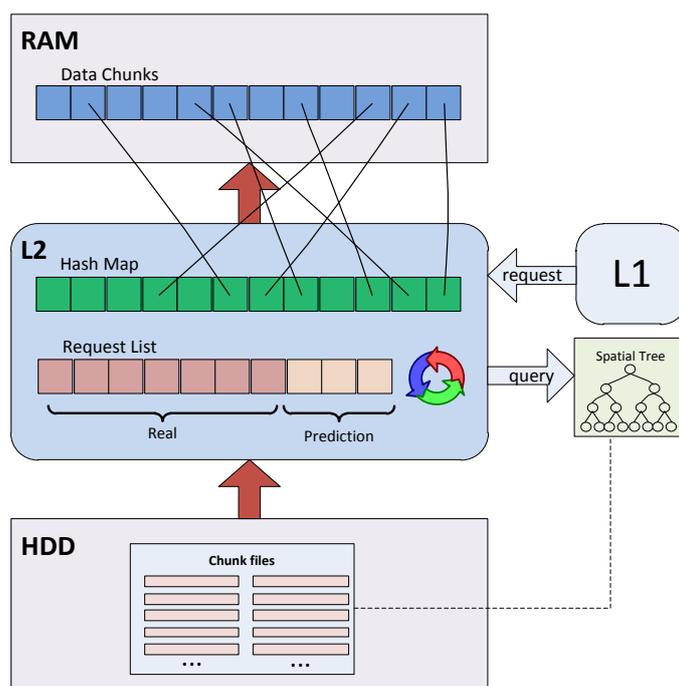


Ilustración 26. Estructura esquemática de la caché asíncrona L2.

Además tiene las siguientes responsabilidades:

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

⁷ 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 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.siguiete())
    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.

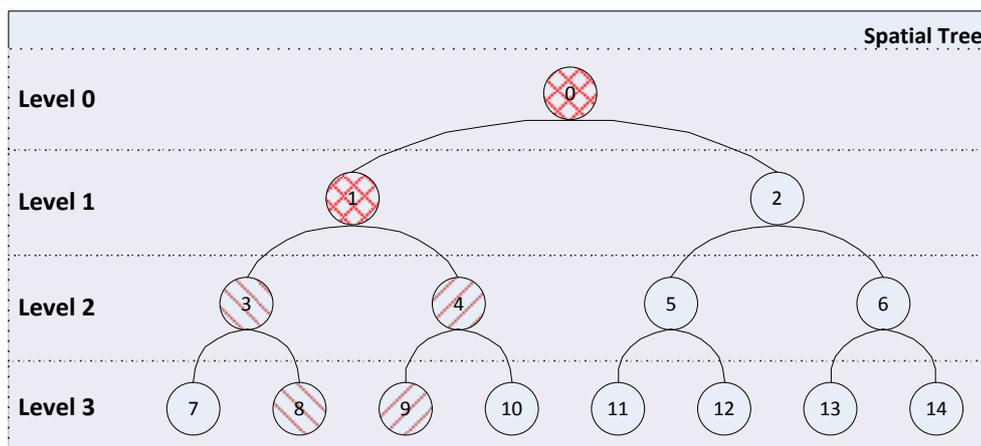


Ilustración 27. Dos rutas de nodos cercanos espacialmente (al nodo 8 y al 9), para alcanzar el nivel máximo de detalle (nivel 3), comparten los nodos 1 y 0.

En la Ilustración 27 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.

6.2 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.

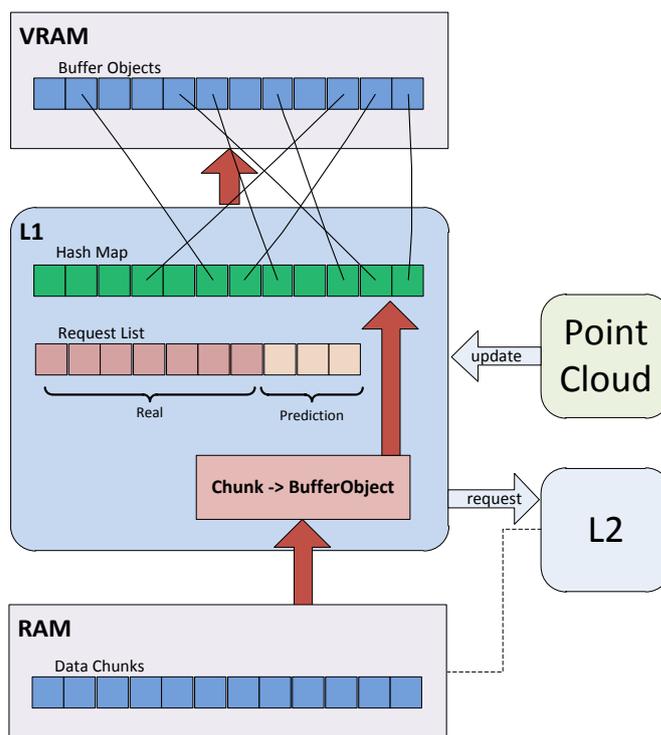


Ilustración 28. Estructura de la caché síncrona L1.

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.

6.3 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 29, 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.

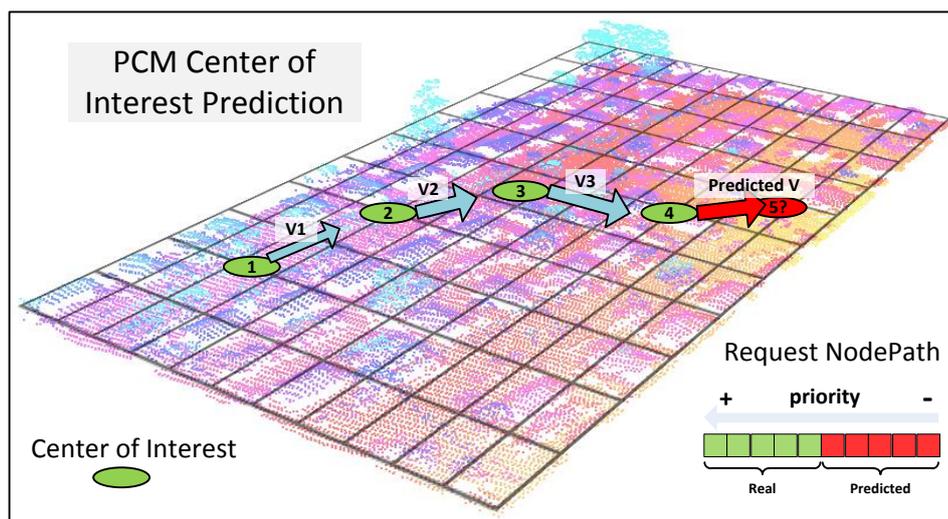


Ilustración 29. Sistema de predicción de PCM.

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.

7 VISUALIZACIÓN INTERACTIVA

Hasta ahora se han detallado las características del sistema multirresolución ideado para trabajar con nubes de puntos de tamaño arbitrario, de forma general, utilizando una jerarquía de memoria HDD – RAM – VRAM y cachés software. En este punto, los datos están particionados en *chunks* que podemos disponer en la GPU de una forma muy eficiente y progresiva, además en el tipo de dato nativo manejado por OpenGL (Vertex Buffer Objects).

En esta sección se describe finalmente cómo, a partir de las estructuras descritas, se puede llevar a cabo el proceso de render. En términos generales, éste constará de las siguientes fases, realizadas en cada *frame*:

1. Determinar las características del nuevo punto de vista (posición, rotación, proyección) definidas por la matriz de OpenGL *ViewProjection* (VP), variada por la entrada del usuario.
2. Escoger cual es el centro de detalle.
3. Realizar un *traversal* por el árbol, confeccionar una lista de los nodos del árbol de la estructura espacial que deberían dibujarse idealmente en este *frame*, pasando los siguientes tests (si un nodo no lo pasa, no se descartan todos sus hijos):
 - a. ¿Excede el nivel de detalle necesario para el punto de vista?
 - b. ¿Es visible desde el punto de vista? (*frustrum culling*)
 - c. ¿Está orientado hacia el punto de vista, es decir, es *la cara de delante?* (*backface culling*, opcional).
4. Ordenar esta lista de peticiones, según un algoritmo de prioridades.
5. Hacer la *query* de la lista a la jerarquía de memoria, con una limitación de tiempo que permita una visualización interactiva (unos pocos milisegundos).
6. Renderizar los datos disponibles en GPU (*dibujado* de los *chunks*).

En las siguientes secciones se comentarán los pasos más importantes de ese proceso, así como las soluciones escogidas para su implementación. Concretamente las siguientes:

- **Preparación de los datos**, donde se comentará cómo preprocesar la nube de puntos para inferir información relevante para el render, fundamentalmente el cálculo de las normales y de tamaños de punto.
- **Test de nivel de detalle**, explicando cómo conocer el nivel máximo requerido para un mejor compromiso calidad – coste.
- **Test de Culling (*frustum y backface*)**, para comentar cómo descartar partes del modelo según su orientación o visibilidad desde la cámara.
- **Prioridad de las peticiones**, sobre cómo ordenar la lista de peticiones de datos a la jerarquía de memoria, asegurando que se obtenga lo más rápido posible un detalle adecuado.
- **Render naive**, una primera aproximación del dibujado de los datos en OpenGL utilizando su *pipeline fijo*.
- **Render avanzado**, un algoritmo de dibujado más complejo para obtener una visualización de mayor calidad, mediante una algoritmos de bajo nivel incrustados en las partes programables del *pipeline* moderno de OpenGL.

7.1 PREPARACIÓN DE LOS DATOS

Algunos de los algoritmos que veremos en las próximas secciones, requieren información adicional en los puntos de la nube, concretamente:

- **Normal de los puntos**. Ya que cada uno de los puntos es una muestra de la superficie de un objeto del mundo real, éste tendrá asociado un vector normal, por definición ortogonal al plano tangente de la propia superficie (Ilustración 30). Este vector es de gran relevancia en la mayor parte de algoritmos de Gráficos en Computación.
- **Radio de los puntos**. La naturaleza del muestreo sobre las superficies de los objetos que realizan los escáneres láser, así como la combinación de varios escaneos para conseguir una única nube de puntos, generan densidades de puntos muy variables. Denominaremos tamaño de cada punto al radio de un círculo mínimo que, solapándose ligeramente con sus vecinos, cubra el máximo de la superficie del objeto muestreado. En la imagen derecha de la Ilustración 30 se puede obser-

var cómo en las zonas más densas, el radio de los puntos es menor (hay más puntos para cubrir la superficie). Sin embargo, en las zonas menos densas, el radio es mayor.

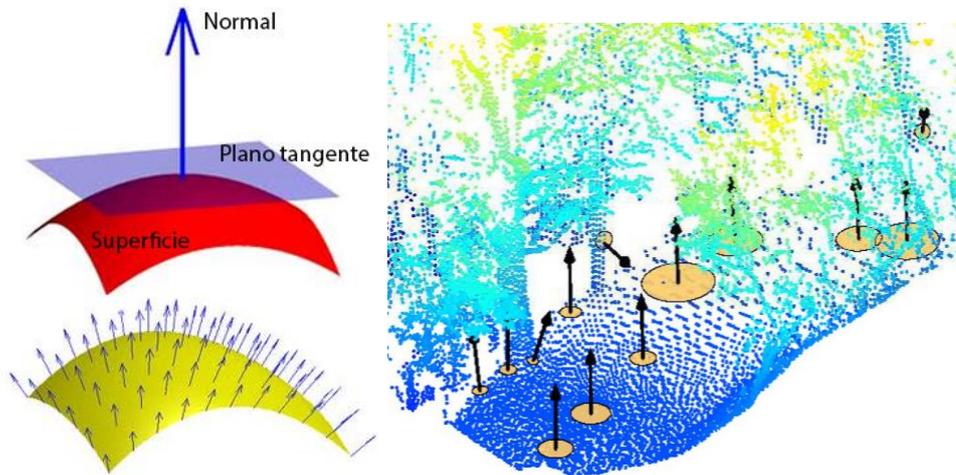


Ilustración 30. Información relevante inferida de la nube de puntos. Normal y radio de cada punto.

A pesar de que este trabajo implementa un procedimiento *naïve* para no depender de esta información, y poder visualizar la nube de puntos *según llega del escáner*, es interesante realizar un preproceso para inferirla y añadirla al conjunto de datos asociados a cada punto. Además, la estructura multirresolución vista anteriormente permite realizar este cálculo sobre nubes de puntos de cualquier tamaño, y aprovechando toda la potencia computacional de las GPUs con técnicas GPGPU y lenguajes como CUDA u OpenCL.

Existen varios métodos para aproximar las normales a cada punto (ver [12] y [20] para una exhaustiva revisión de varios de ellos). En este trabajo se ha optado por la más utilizada hoy en día, basada en la técnica estadística del **Análisis de Componentes Principales** (*Principal Component Analysis, PCA*), que reduce la dimensionalidad de un conjunto de datos y de una manera muy intuitiva halla las causas de su variabilidad y las ordena por importancia [21]. Matemáticamente, consiste en generar una matriz de covarianzas C entre los puntos más cercanos del punto analizado, y estudiar sus *autovectores* y *autovectores*.

$$C = \frac{1}{k} \sum_{i=1}^k \cdot (p_i - \bar{p}) \cdot (p_i - \bar{p})^T, C \cdot \vec{v}_j = \lambda_j \cdot \vec{v}_j, j \in \{0, 1, 2\}$$

Donde k es el número de vecinos considerado alrededor del punto del que se quiere calcular su normal p_i , \bar{p} representa el baricentro de estos puntos, λ_j es el j -ésimo *autovalor* de la matriz de varianza, y \vec{v}_j el j -ésimo *autovector*. De esta forma, se consiguen las bases ortogonales del espacio vectorial que representa a ese conjunto de datos, y cuyo tercer eje será la normal del punto p_i (Ilustración 31).

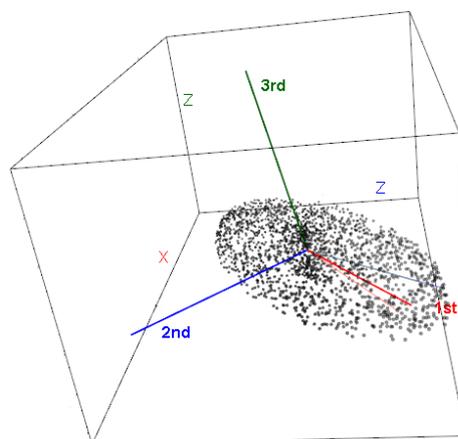


Ilustración 31. PCA aplicado a una nube de puntos para inferir la normal de uno de ellos.

Con este método se puede estimar la dirección de la normal, pero no su orientación, que será importante para la fase de *culling* (ver Sección 7.3). Existen dos formas de conocer el signo de esta normal:

- A partir de datos reales, conociendo los puntos de escaneo en el momento de adquisición del *dataset*. Por la naturaleza de los escáneres, las normales siempre deberán apuntar *hacia* donde se situaba el dispositivo.
- A partir de una nueva inferencia estadística, en este caso la ideada por H. Hoppes (*Microsoft Research*) en [22].

En cuanto a la estimación del tamaño de punto, aunque existen métodos más complejos (ver [12]), en nuestro sistema utilizaremos una más sencilla, basada simplemente en procesar cada punto, buscando la distancia mínima con sus vecinos. Dado el carácter multi-

rresolución del sistema y de carga progresiva de niveles de detalle, los radios estimados de esta forma se han comprobado suficientemente efectivos para los requisitos planteados.

7.2 TEST DE NIVEL DE DETALLE

En cualquier sistema multirresolución la selección del centro de detalle, así como el nivel de detalle necesario en cada instante es fundamental, principalmente para decidir cuáles son las partes a cargar en memoria del modelo.

En este caso, el centro de detalle así como el nivel máximo requerido en cada parte del kd-tree es uno de los primeros procesos realizados en cada *frame*, en función de las características del punto de vista, posición, rotación y proyección.

Existen algunos estudios sobre percepción humana observando una pantalla proyectada, para escoger las *zonas calientes* a donde un usuario mira con más asiduidad. Con ello se pueden elaborar complejas técnicas para escoger el centro de detalle del modelo. En este trabajo se ha optado por un método más simple, aunque eficaz. El nodo del árbol representante como centro de detalle será aquel *por delante* de la cámara, más cercano. Este nodo será utilizado como referencia en algunos de los algoritmos que se verán a continuación.

En realidad, es mucho más importante la selección del nivel máximo de detalle cuando se hace el *traversal* del árbol en la búsqueda de las peticiones óptimas a la caché L1. En la sección anterior se comentó el preproceso para estimar los radios de los puntos. Además de utilizarlos para el render, como se verá más adelante, juegan un papel fundamental en esta etapa. Durante la fase de construcción del árbol, cada nodo almacena un dato estático, la media de radios de los puntos contenidos en ese nodo, utilizando la hipótesis de que, como los nodos almacenan puntos cercanos entre sí (clusters), tienen una alta posibilidad de tener una densidad homogénea.

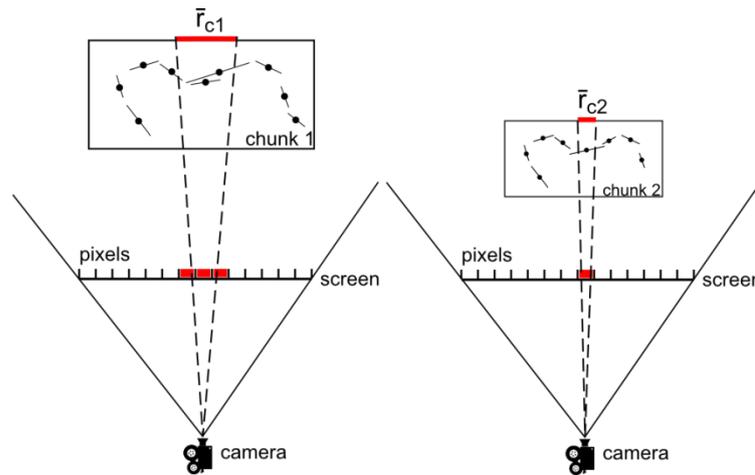


Ilustración 32. Test de nivel de detalle. A la izquierda, un nodo con sus puntos que pasa el test, ya que la proyección del radio medio de los puntos afecta a varios píxeles. A la derecha, un nodo que no pasa el test, ya que la misma proyección afecta a menos de un píxel de pantalla.

El test de nivel de detalle para un nodo, consiste en saber cuál será el grado de afectación de los puntos que contiene en los píxeles del *frame*. Si la proyección en pantalla de la media de los radios de los puntos tiene una dimensión sub-píxel, es decir, afectará a menos de un píxel, se descartará dicho nodo y todos sus hijos (que afectarán aún menos). En caso contrario, pasará el test y el *traversal* continuará por sus hijos (ver Ilustración 32).

7.3 TEST DE CULLING (FRUSTUM Y BACKFACE)

En el campo de los Gráficos en Computación se denomina *culling* al proceso de eliminar partes de un modelo tridimensional antes de la fase de dibujado, por uno de los siguientes motivos:

- Para ahorrar tiempo de CPU y GPU, así como memoria, ya que no afectarán de ninguna forma a la imagen final generada (denominaremos tipo 1).
- Tipo2, para decidir si se quieren dibujar partes del modelo con características concretas (denominaremos tipo 2).

Existen multitud de tipos de *culling*, revisados en [23], como por ejemplo:

- **View Frustum culling**, que revisa qué partes son vistas desde la cámara y cuales no (tipo 1).

- **Backface culling**, que decide si mostrar las *caras traseras* de las superficies (tipo 2).
- **Contribution culling**, que estima la contribución en la imagen final de partes del modelo, en función de su lejanía (tipo 1 y 2).
- **Occlusion culling**, que determina, según el punto de vista, qué partes del modelo quedan ocluidos por partes otras zonas opacas del modelo, que están por delante (tipo 1).

En este trabajo se han implementado versiones de los dos primeros, *View Frustum* y *Backface culling*. Se puede decir que el *contribution culling* es inherente a la naturaleza multiresolución de este sistema. Por último, el *occlusion culling* carece de importancia cuando se trabaja con modelos que representan objetos, pero supone una mejora en el caso de escenas con muchos elementos, como entornos urbanos o arquitecturales. Se establecerá como una mejora en el capítulo dedicado a futuros desarrollos.

A continuación se describen detalladamente los primeros.

7.3.1 VIEW FRUSTUM CULLING

Un *frustum*, también conocido como pirámide truncada, es una porción de una figura geométrica (usualmente un cono o una pirámide) comprendida entre dos planos paralelos. Este término es muy utilizado en Gráficos en Computación ya que define el espacio de visión de una cámara virtual (Ilustración 33).

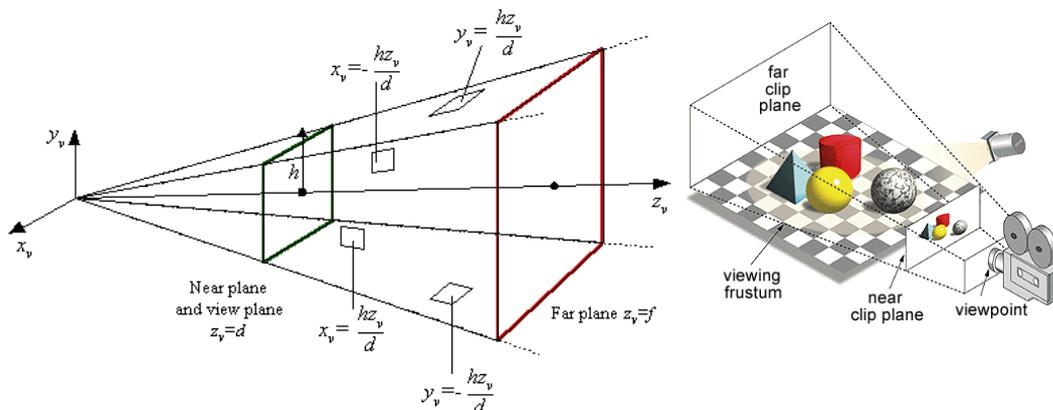


Ilustración 33. Frustum de una cámara y sus parámetros (Fuente: Wikipedia)

El *view frustum culling test* consiste en descartar en cada frame aquellas partes de la escena que no se ven desde el punto de vista. Dada la estructura espacial multirresolución utilizada, esta prueba se realiza durante el *traversal* del kd-tree, comprobando si la *bounding box* de cada nodo está contenida (completa o parcialmente) o no dentro del *frustum* de la cámara, según el siguiente algoritmo.

```
Function ViewFrustumCullingNodeTest(camFrustum, node, nodeList)
  contentness = cameraFrustum.content(node)
  if(contentness == COMPLETELY)
    nodeList.add(subtree(node))
  else if(contentness == PARTIALLY)
    nodeList.add(node)
    ViewFrustumCullingNodeTest(camFrustum, leftChild(node), nodeList)
    ViewFrustumCullingNodeTest(camFrustum, rightChild(node),
nodeList)
  else if(contentness == NONE)
    // nothing to do here
  return
End Function

// First recursive call
Function ViewFrustumCullingTest(camFrustum) = nodeList
  new nodeList
  ViewFrustumCullingNodeTest(camFrustum, 0, nodeList)
  return nodeList
End Function
```

De esta manera, el algoritmo es capaz de seleccionar los nodos que entran en la vista de la cámara, como se puede ver en la Ilustración 34.

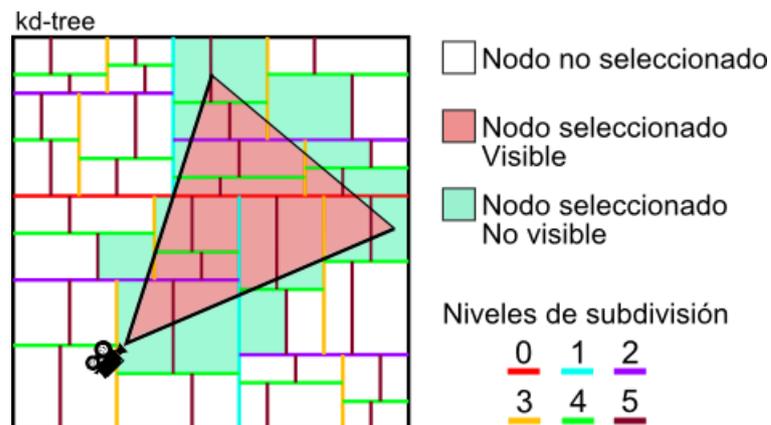


Ilustración 34. Representación gráfica de los nodos seleccionados en un *View Frustum Culling* para una cámara dada.

Cuanto mayor sea el número de niveles del kd-tree, mayor su *granularidad*, y más preciso el algoritmo a la hora de descartar puntos. Las matemáticas para averiguar si un nodo es contenido en el *frustum* son muy sencillas, y se basan simplemente en comprobar de qué lado de los planos que lo componen están cada una de las esquinas del *bounding box* del nodo.

7.3.2 BACKFACE CULLING

Como ya se ha comentado, el *backface culling* es opcional, y consiste en eliminar aquellos puntos del *dataset* que, desde un punto de vista dado, muestran su *cara trasera* en vez de su *delantera*. Esto puede ser interesante para eliminar *ruído* de ciertos modelos, con muchos objetos distintos, como entornos urbanos.

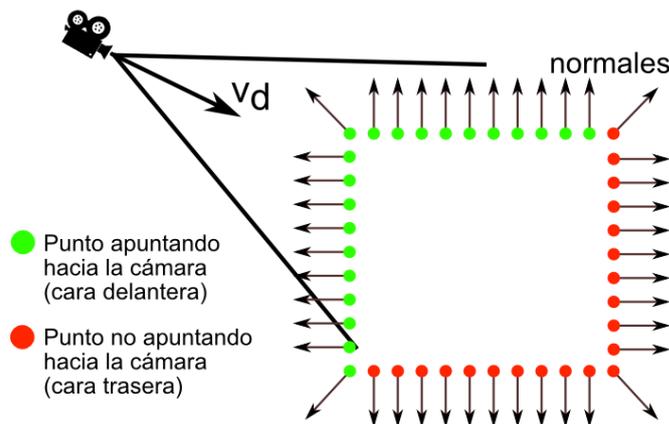


Ilustración 35. Backface culling.

Como se puede ver en la Ilustración 35, el test simplemente consiste en comprobar el signo del producto escalar entre el vector de dirección de la cámara (v_p) y la normal del punto (n).

$$\begin{cases} v_p \cdot n < 0 & \text{cara delantera} \\ v_p \cdot n \geq 0 & \text{cara trasera} \end{cases}$$

7.4 PRIORIDAD DE LAS PETICIONES

Una vez realizado el *traversal* del árbol, pasando todos los test anteriores según el punto de vista de la cámara, se tiene una lista de nodos que desearíamos estuviesen cargados

en la memoria VRAM de la GPU para poder ser dibujados. Esta lista deberá ser remitida al sistema multirresolución que, de forma asíncrona, se encargará de ir cargándolos para tenerlos disponibles en los sucesivos *frames* lo más rápido posible.

Pero, según se ha visto en capítulos anteriores, la orden con el que los nodos se sitúan en esa lista *query* definirá la prioridad con la que serán cargados; cuanto más al principio de la lista, más prioridad. Por lo tanto, se necesita establecer un criterio de ordenación de la lista antes de lanzar la *query*.

En este trabajo se han tenido en cuenta las siguientes heurísticas para confeccionar el algoritmo de ordenación de prioridad:

- **Se quiere dar prioridad a los niveles superiores del árbol** (modelo menos detallado). Si los niveles más altos (por lo tanto, más grandes, que ocupan mayor parte de la escena, pero con menor resolución) tienen siempre prioridad en las *queries*, las cachés software los mantendrán siempre cargados en memoria. De esta forma se evitarán *saltos* en la visualización cuando se mueva repentinamente la cámara, ganando en fluidez y suavidad. La contrapartida es que, para alcanzar la resolución más alta del modelo, el tiempo de espera será ligeramente mayor.
- **Dentro del mismo nivel de detalle, la prioridad será proporcional a la distancia a la cámara.** De esta forma la visualización cargará el detalle más rápidamente en las zonas cercanas al punto de vista, que son por las que generalmente se interesa el usuario.

De esta forma, se ha elaborado un algoritmo en dos etapas, a partir de la lista según llega del *traversal* (Ilustración 36):

1. Se ordena la lista por niveles de detalle. Este paso es trivial, ya que debido a la nomenclatura por IDs vista en el capítulo anterior, es una ordenación de números naturales.
2. Dentro de cada nivel, se sub-ordena por distancia del baricentro de su *bounding box* a la cámara.

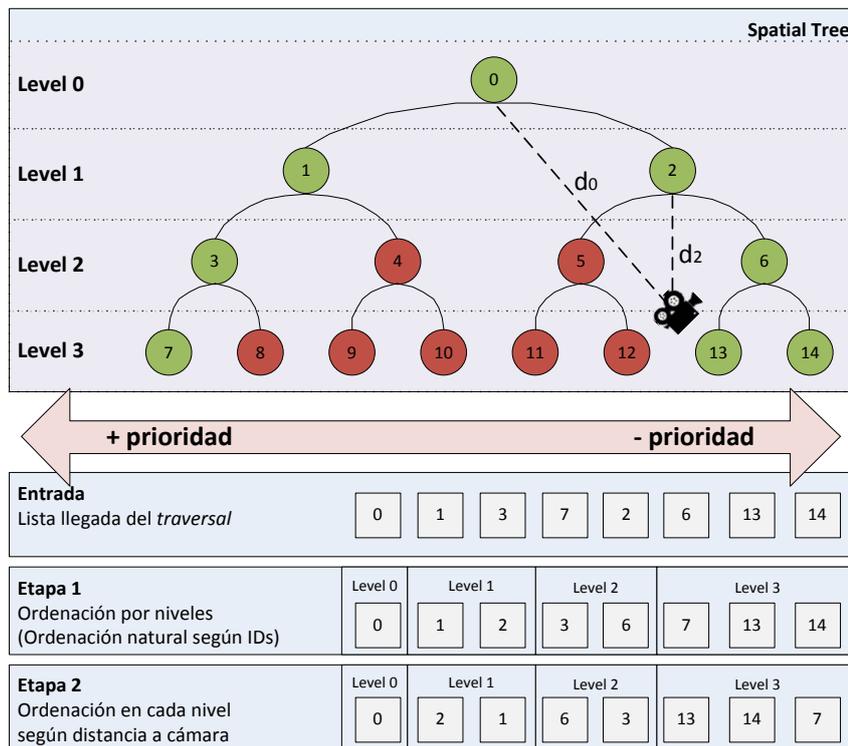


Ilustración 36. Ejemplo de aplicación del algoritmo de prioridad. Los nodos en verde son aquellos que han pasado los tests de *culling* y de nivel de detalle. Por comodidad, supongamos que las distancias a cámara son proporcionales a las distancias en el dibujo al icono de la cámara.

7.5 RENDER EN OPENGL

En esta sección se explicará la técnica de render con la que se pintarán los puntos que el sistema multiresolución va cargando en la GPU, según todo lo expuesto anteriormente.

En el capítulo dedicado a la contextualización se comentaron las bases del render en GPU, mediante el *pipeline* de OpenGL. Para poder comparar las mejoras que supone el uso de técnicas más avanzadas, se comenzará comentando cómo sería el render *naive* según la funcionalidad del *pipeline* fijo.

7.5.1 RENDER NAIVE

OpenGL permite renderizar puntos de una manera muy básica, mediante buffers de vértices con el estado `GL_POINTS`. El procedimiento que utiliza consiste en proyectar desde el espacio tridimensional los vértices a la pantalla, y pintar alrededor del punto de proyección un cuadrado de lado establecido.

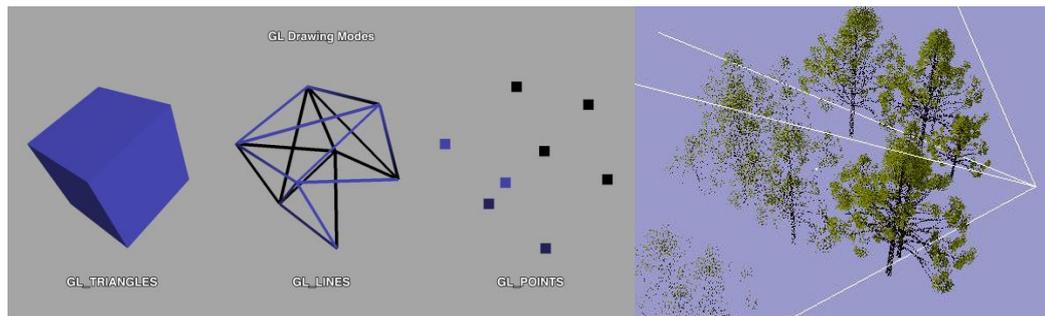


Ilustración 37. Izquierda: funcionamiento de `GL_POINTS` en contraposición a los estados normales de OpenGL. Derecha: ejemplo de render con `GL_POINTS`.

Esta técnica, además de ineficiente, produce resultados visuales de muy baja calidad debido a que:

- Los puntos tienen siempre forma cuadrada, independientemente de su orientación. Con ello se pierden todos los detalles de la superficie que se intenta representar.
- El tamaño de los puntos en pantalla no es variable (sin utilizar extensiones OpenGL especiales) eliminando toda sensación de profundidad.
- Los puntos se solapan entre ellos de una forma brusca, no permitiendo una sensación de continuidad en la superficie.
- El *pipeline* fijo no permite ningún control sobre la forma de pintar los puntos en pantalla, perdiendo toda opción de filtrado, efectos especiales, etc.

7.5.2 RENDER AVANZADO: SPLATTING

Como se ha visto en el capítulo de contextualización, el nuevo hardware gráfico (GPUs) suministra un pipeline con algunas partes programables, concretamente el procesado de vértices (*vertex programs*) y el procesado de fragmentos, los *píxeles virtuales* antes de ser lanzados a pantalla (*fragment programs*).

En la sección anterior se explicó por qué las opciones estándar de OpenGL para renderizado de puntos no son suficientemente buenas. En la siguiente imagen (Ilustración 38) se observa en la parte derecha, arriba, el procesado estándar por `GL_POINTS`, y abajo, el

resultado deseable, que utiliza la técnica que se presenta a continuación, el *splat rendering* [24].

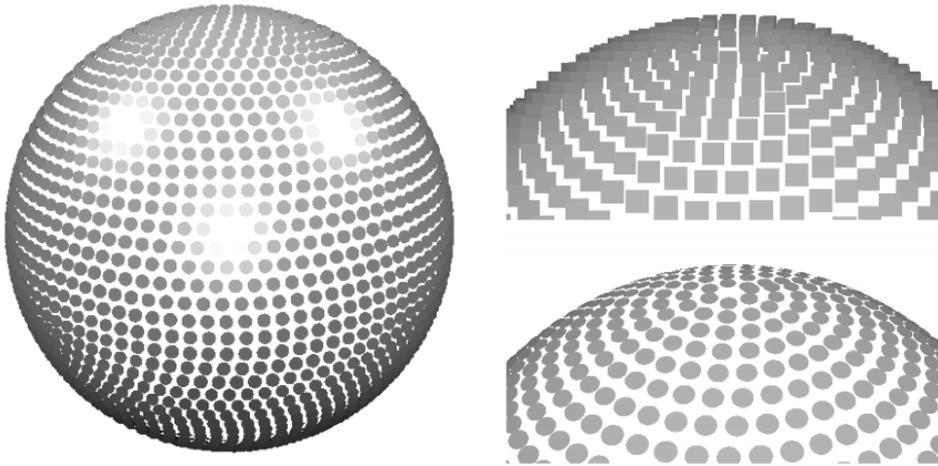


Ilustración 38. Comparación entre el render standard de OpenGL y el Splat Rendering (Fuente: [24]).

Esta técnica permite proyectar los puntos como elipses de distintos tamaños en pantalla, dependiendo de su posición, dirección y radio. Para ello, se basa, por una parte en las partes programables del pipeline, como veremos más adelante, y en el uso de un estado especial de OpenGL, denominado `GL_POINTSPRITES`. El funcionamiento estándar de esta primitiva es similar a `GL_POINTS`, pero permitiendo añadir una textura (imagen) al punto que se proyecta. **La técnica de *splatting* utiliza esta característica para calcular en la GPU cómo es la forma de la elipse proyectada (*vertex program*) y para cada pixel descartar aquellos que estén fuera de esa elipse (*fragment program*).**

Además, el tamaño de los puntos se puede multiplicar por un factor hasta conseguir que los splats cubran toda la superficie, eliminando los *huecos* intermedios (Ilustración 39).

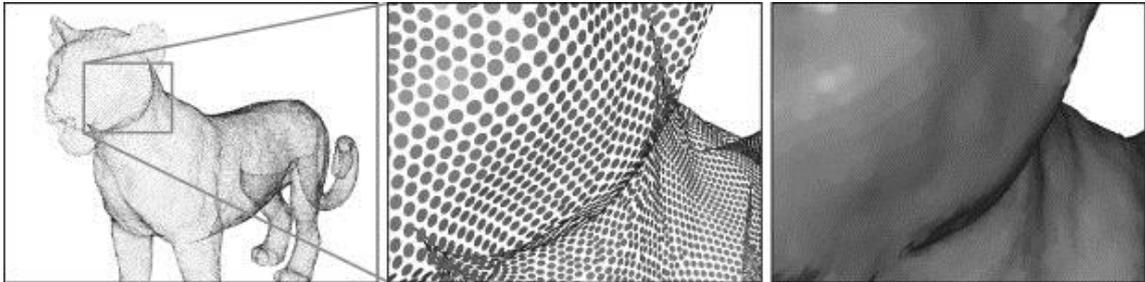


Ilustración 39. *Splatting* con diferente factor del tamaño de punto (Fuente: [24])

El funcionamiento de esta técnica se puede observar en la siguiente figura (Ilustración 40), donde se representa un punto de la nube siendo proyectado en pantalla. La figura bidimensional resultante será siempre una elipse, acotada por un cuadrado de lado *pointSize*, que depende de la proyección de la cámara y el radio precalculado r . Esta elipse queda definida por los parámetros de sus diagonales d_x y d_y , que pueden ser computados a partir de la normal precalculada del punto N .

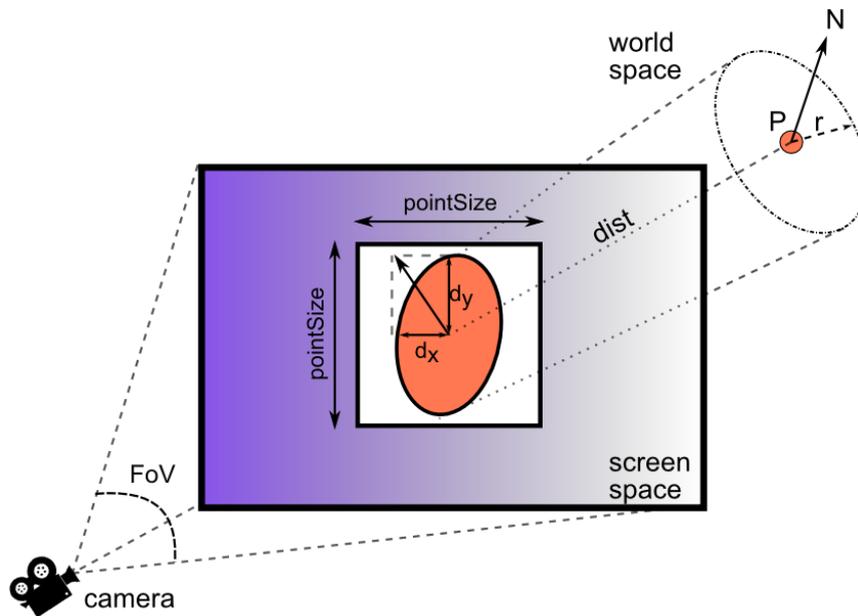


Ilustración 40. Definición de splat, parámetros y proyección en espacio pantalla.

Para poder implementar esta técnica, se debe hacer uso de las partes programables del pipeline gráfico visto en el capítulo de contextualización. En este caso, se compondrá de

dos partes: un programa por vértice, que se aplicará a todos los puntos individualmente cuando llegan en *stream* al *pipeline*, y un programa por fragmento (*potencial pixel*) que se aplica a cada uno de los fragmentos del cuadrado que acota la elipse de cada punto. Sus funciones se resumen en las siguientes tablas, donde se muestra la entrada de cada uno de los dos programas (ya bien sea a partir de las entradas estándar de OpenGL/GLSL, o de parámetros enviados desde la CPU, propios del *dataset* como el radio del punto o los datos de proyección).

Vertex Program (por punto)	
Recibe	Calcula
Coordenada Normal Color Radio Size Ventana Render Matriz ModelView	Proyecta la normal a la pantalla, y pointSize BackFace Culling Elipse (dx,dy)
Fragment Program (por pixel <i>potencial</i>)	
Elipse Coords. Textura	Si dentro de elipse Pinta Si no Descarta

Vista esta abstracción del funcionamiento sobre el pipeline programable, la definición de los programas o *shaders* es como sigue; dados los siguientes parámetros desde la aplicación:

- **splatRadio (r)**, el tamaño del punto precalculado anteriormente, en espacio mundo.
- **splatSizeFactor (s)**, un factor configurable para aumentar o disminuir el tamaño del splat.
- **projectionFactor (pf)**, el *aspect ratio* de la ventana de render.
- **windowHeight (wh)**, la altura de la ventana de render.

Y los siguientes parámetros de cada uno de los puntos del *dataset*:

- `modelViewProjection` (`mvp`), la matriz de transformación para ese frame.
- `pointPosition` (`p`), la posición en espacio mundo del punto.
- `pointNormal` (`n`), la normal en espacio mundo del punto.

En cada punto se pueden calcular los parámetros `dx` y `dy` que se corresponden con las diagonales de la supuesta elipse proyectada. Este proceso es realizado en el *vertex program* de la siguiente manera (lenguaje GLSL).

```

varying vec2 ellipse;

uniform float splatRadio;
uniform float splatSizeFactor;
uniform float projectionFactor;
uniform int windowHeight;

void main() {

    gl_Position = ftransform();
    gl_FrontColor = gl_Color;
    normal = normalize(gl_Normal);

    float pointDistance = abs((gl_ModelViewMatrix * gl_Vertex).z);
    float projectedDistance = proj_factor * splat_radio / pointDistance;
    float theoreticalSize = 2.0 * projectedDistance * windowHeight * splatSizeFactor;
    gl_PointSize = clamp(theoreticalSize, 1.0, 32.0);

    vec3 filteredNormal = gl_NormalMatrix * gl_Normal;

    // Point toward viewer
    if (filteredNormal.z < 0.0)
        filteredNormal *= -1.0;

    // Limit elliptical stretch
    if (filteredNormal.z < 0.1) {
        filteredNormal.z = 0.1;
        normalize(filteredNormal);
    }

    ellipse = vec2(
        filteredNormal.x / filteredNormal.z,
        filteredNormal.y / filteredNormal.z
    );
}

```

Después, en la etapa de proceso de fragmentos del pipeline, se deshace esta proyección, y teniendo las coordenadas de textura correspondientes a dicho fragmento, se puede saber si *caen* o no dentro de la elipse proyectada. En caso de que sea así, se pintarán con el color adecuado. En caso contrario, se descartarán.

```
varying vec2 ellipse;

void main()
{
    float dx = 2.0*(gl_TexCoord[0].x-0.5);
    float dy = 2.0*(gl_TexCoord[0].y-0.5);
    float dz = ellipse.x * dx - ellipse.y * dy;
    float R2 = dx*dx + dy*dy + dz*dz;

    if (R2>1.0) discard;
    else gl_FragColor = gl_Color;
}
```

Esta ingeniosa técnica, ideada por Bostch y Kobbalt en el 2003, sigue siendo a día de hoy el estado del arte en el render de puntos. Para mayores detalles y la demostración matemática, se ruega consultar [24].

Point Cloud manager

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 campos 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 procesamiento 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. Aquí también el programa visualizador interactivo, de forma que sea fácilmente acoplable a cualquier otro software con sólo realizar algunas llamadas o compartir un mismo contexto de OpenGL.
- Por otra parte, como paquete **software con carácter comercial**, que pueda afrontar los problemas de ingeniería planteados en proyectos reales.

8.1 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.

8.2 ESTRUCTURA

En el siguiente diagrama (Ilustración 41) 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.

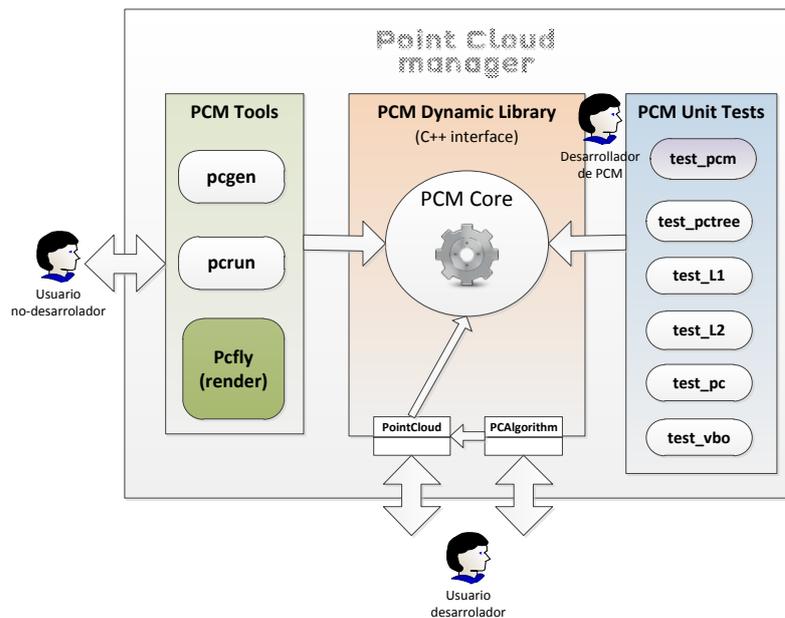


Ilustración 41. Arquitectura del paquete software PCM.

Se detallan a continuación cada uno de los componentes.

8.2.1 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.

8.2.1.1 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 cualquier-

ra 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.
- **NormEst:** capaz de realizar la estimación de las normales y tamaños de los puntos como paso previo a la construcción del árbol.

8.2.1.2 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 ca-

chés 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.

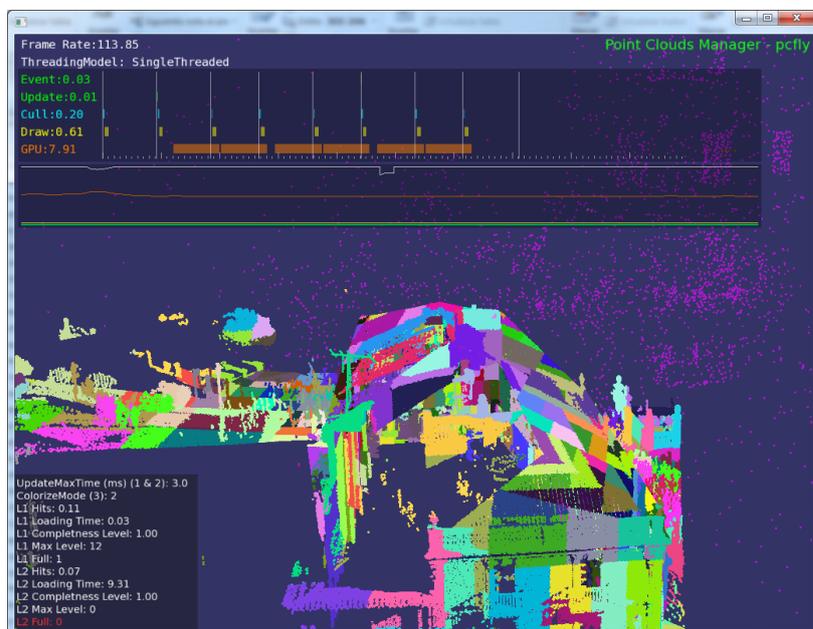


Ilustración 42. Captura de pantalla de la aplicación PCFLY en uso, monitorizando la carga e niveles y el estado de las cachés.

8.2.1.3 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

8.2.2 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,

Splat rendering, 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.

8.2.3 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.

8.3 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 43) 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. En este punto se puede lanzar también el comando *normest* que realizará la estimación estadística de normales y tamaños de punto.

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.

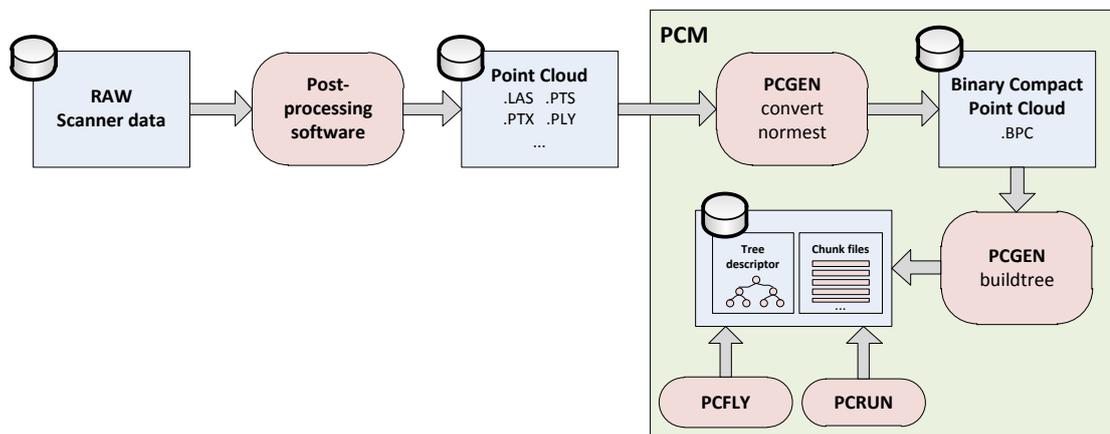


Ilustración 43. Proceso de conversión de datos para el uso del sistema PCM.

8.4 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 [25].

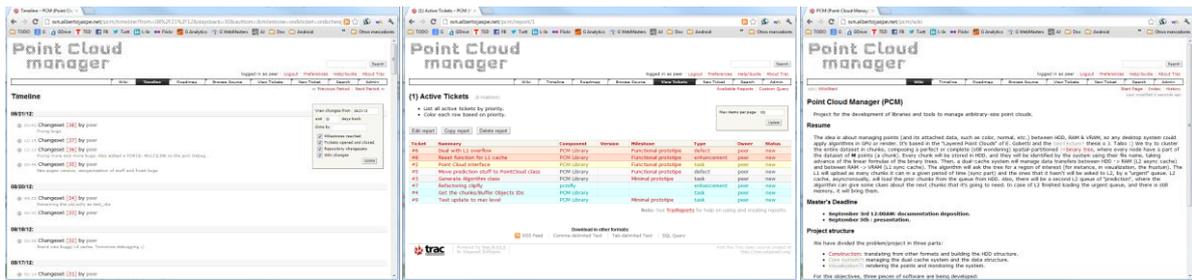


Ilustración 44. Screenshots del sistema web TRAC montado ex profeso para el desarrollo y seguimiento de PCM.

Tras barajar varias posibles opciones, se ha optado por una solución web basada en el binomio **TRAC + Subversion** [26], 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 referen-

cias 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*).

9 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, aunque el sistema podría compilar en diversas plataformas gracias a que utiliza el software de construcción CMake.

9.1 DIAGRAMA DE CLASES GLOBAL

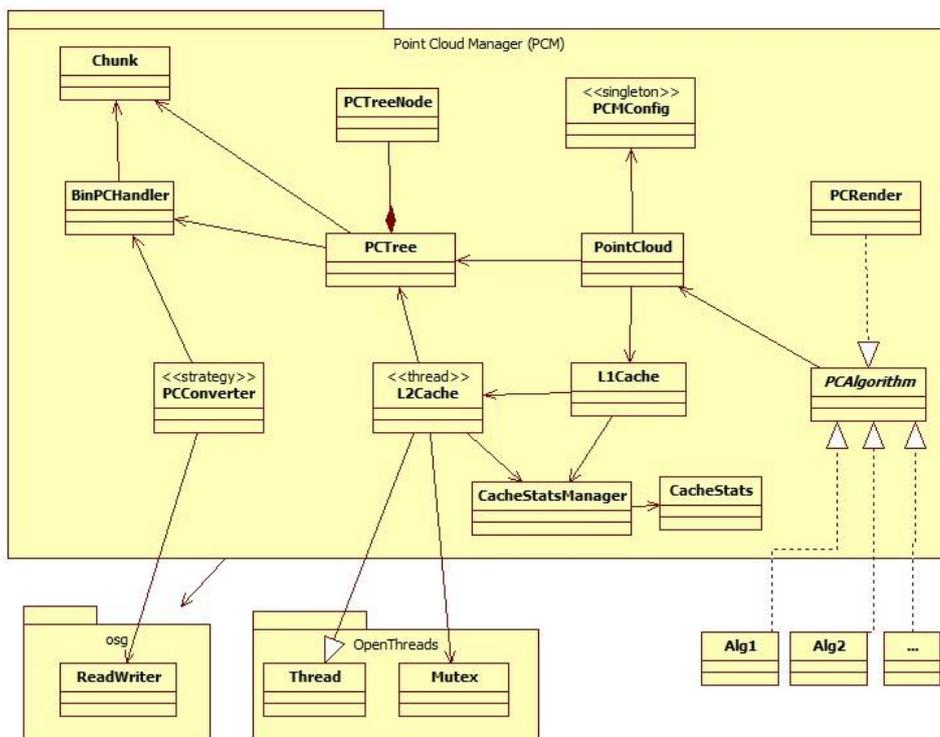


Ilustración 45. Diagrama de clases global (no detallado) de la librería PCM.

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

- **OpenSceneGraph (OSG)** [27]: 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 importantes⁸.

⁸ Los atributos y métodos mostrados no se exponen de forma estricta para facilitar su lectura.

9.2 CONVERSIÓN Y MANEJO DE DATASETS

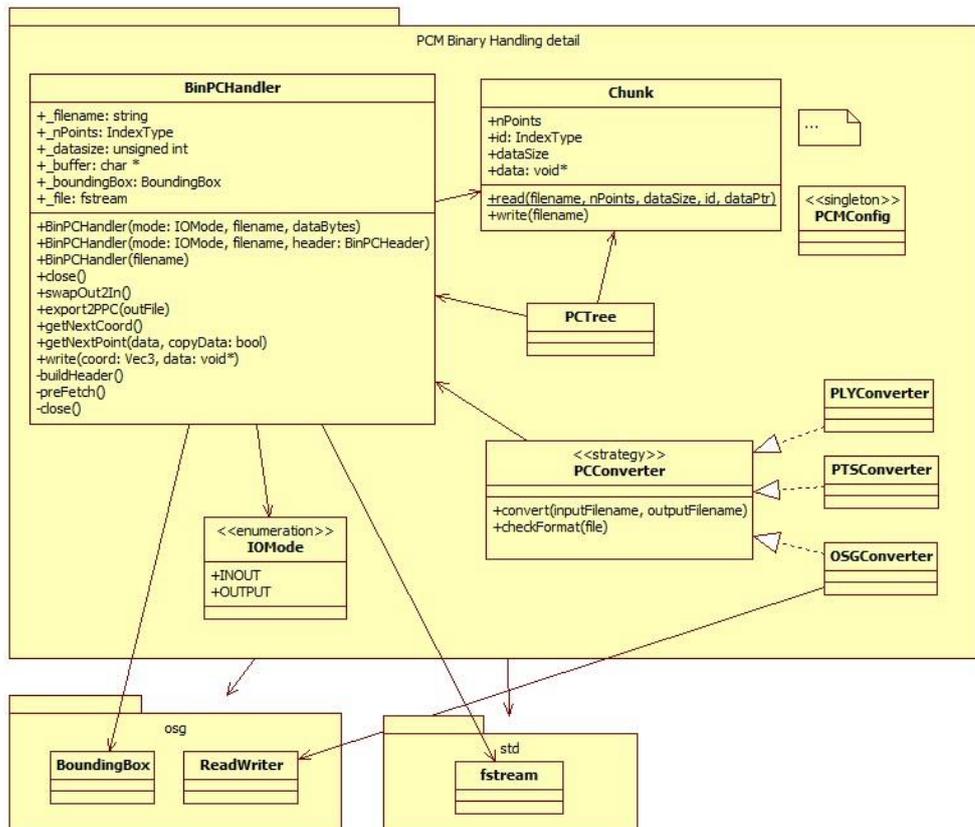


Ilustración 46. Diagrama de clases de PCM, resaltando la parte dedicada al manejo de ficheros binarios y conversión de formatos.

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.
- **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 *tem*-

plate 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.

9.3 ESTRUCTURA DE DATOS

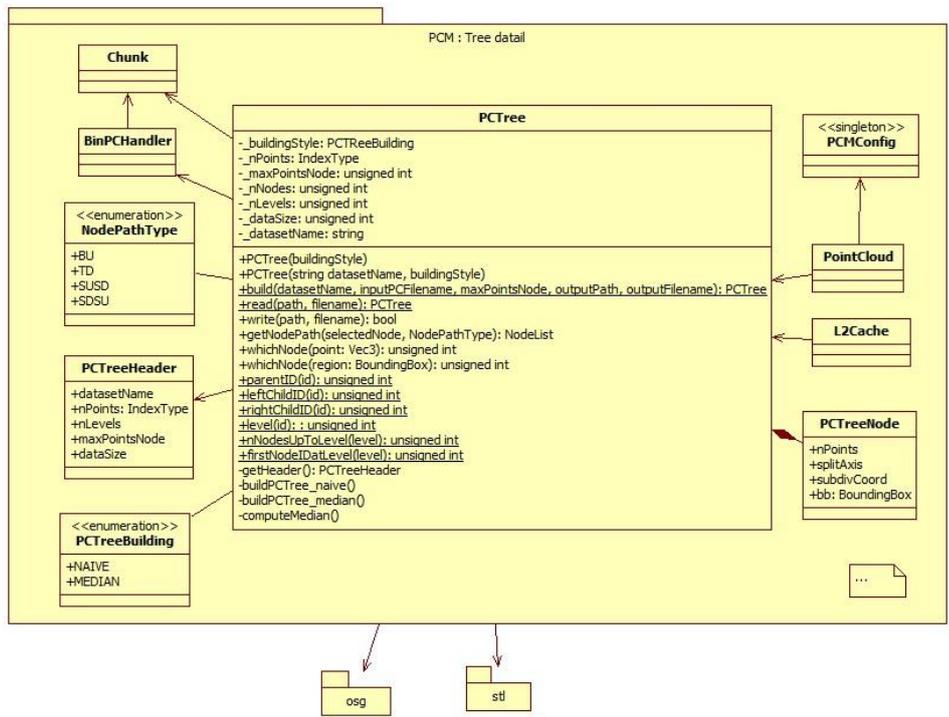


Ilustración 47. Diagrama de clases de PCM, resaltando la parte dedicada al manejo de la estructura de datos.

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 sus 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.

9.4 JERARQUÍA DE MEMORIA

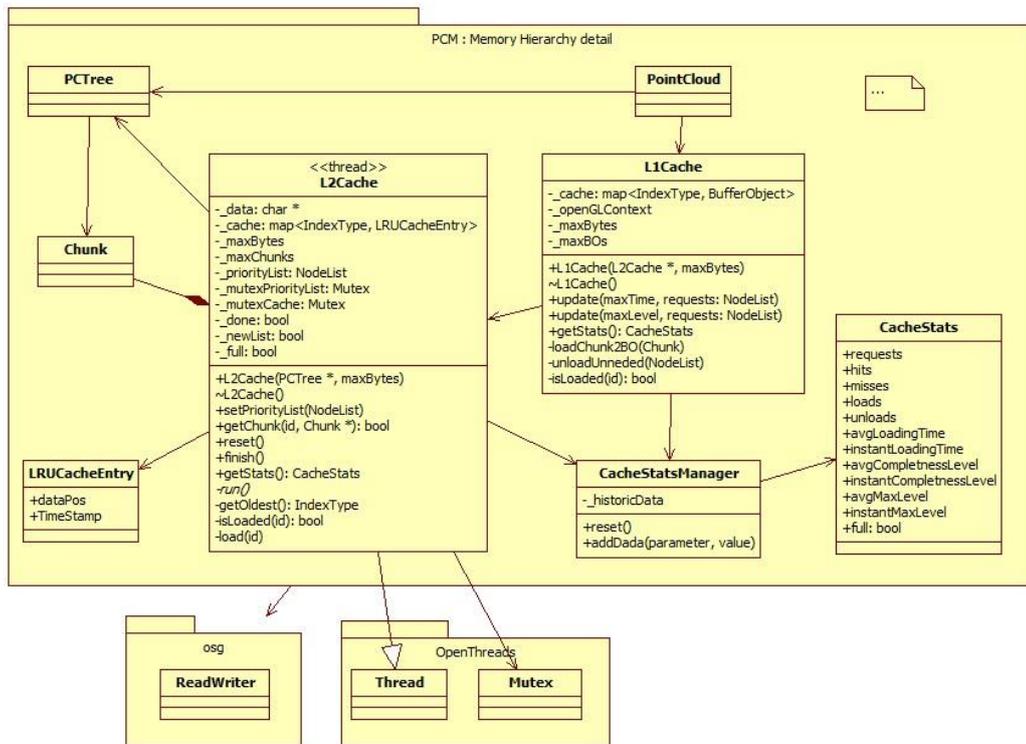


Ilustración 48. Diagrama de clases de PCM, resaltando la parte dedicada a la gestión y jerarquía de memoria.

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 **OpenTh-reads::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.

9.5 INTERFAZ 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 e 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.

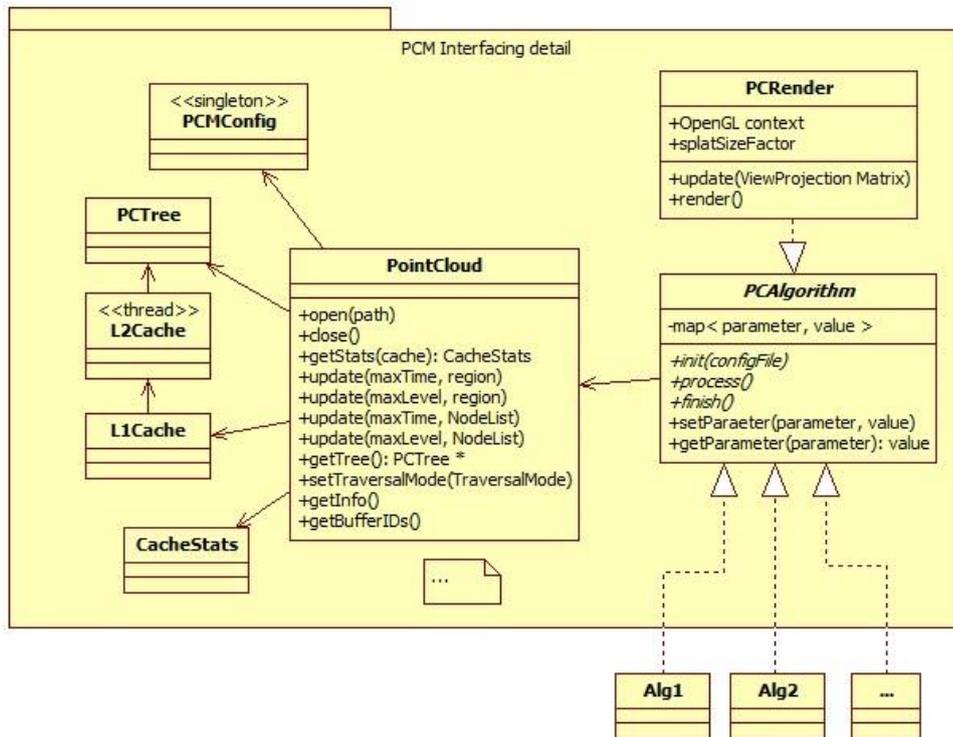


Ilustración 49. Diagrama de clases de PCM, resaltando la parte dedicada a la interfaz del sistema de cara al exterior.

La clase virtual pura **PCAlgorithm** es la que dará acceso a la plataforma de procesamiento 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 procesamiento, 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.

Dada la generalidad del sistema, el visualizador se plantea como uno de los algoritmos de procesamiento interno de la librería que sigue este método, **PCRender**. Esta clase es la que

utiliza PCFLY para visualizar con el API gráfico OpenGL en tres dimensiones la nube de puntos, según todo lo explicado en el capítulo de visualización interactiva.

10 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 50 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)	Test PC
	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)	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 50. Dataset y características del PC utilizados en los tests.

10.1 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 algoritmo descrito, que trabaja a muy bajo nivel, con la mantisa y exponente de los números en formato flotante IEEE754.

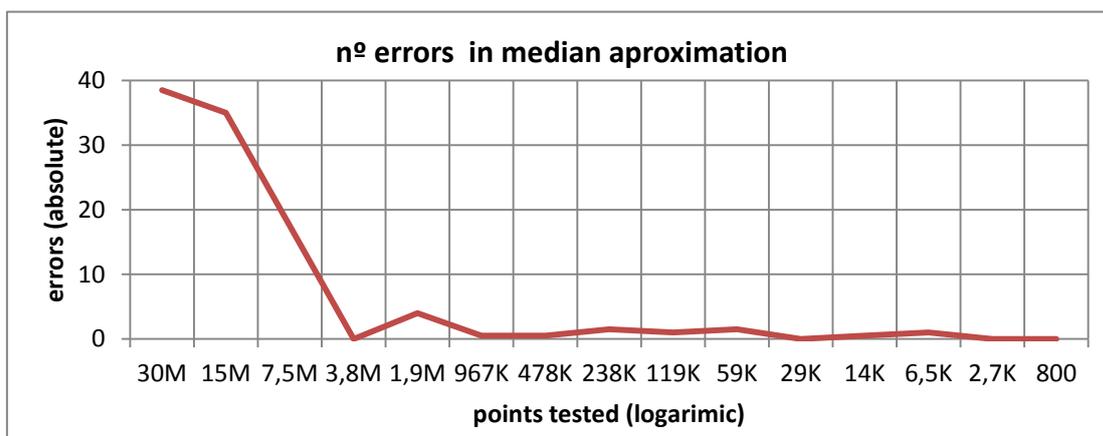


Ilustración 51. Gráfica de tasa de errores en la aproximación de la mediana, para la selección de planos de corte, en la construcción de la estructura de datos espacial.

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.

10.2 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.

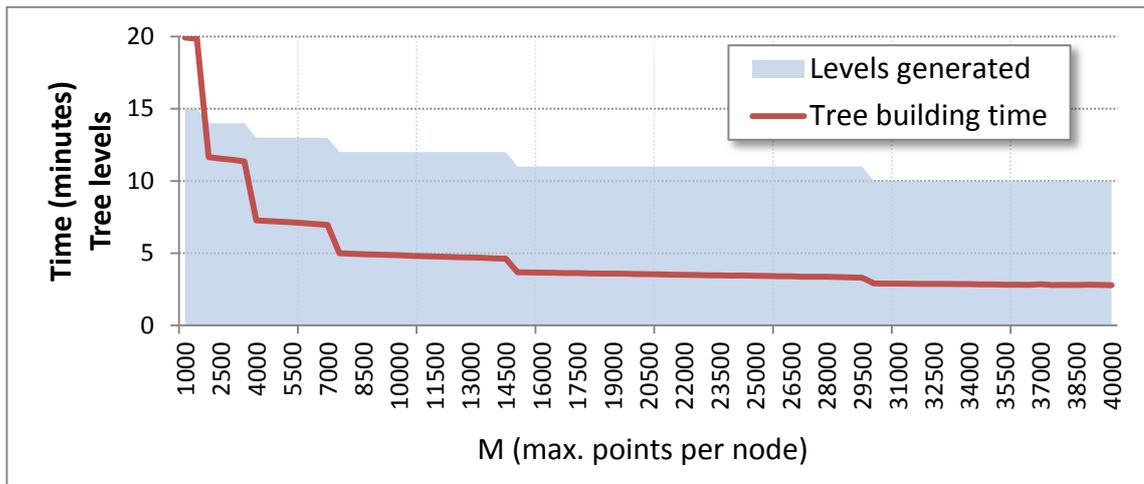
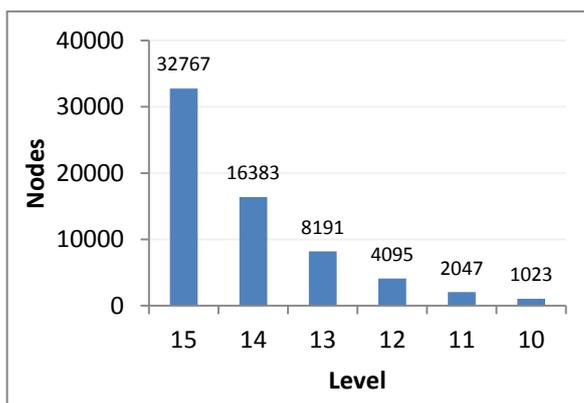


Ilustración 52. Gráfica comparando el parámetro M de máximos puntos por nodos con el tiempo de construcción y los niveles de detalle generados, en la construcción de la estructura espacial.



En la figura anterior (Ilustración 52) 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.

10.3 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 sis-

tema 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 53 se muestra una monitorización de 5 segundos.

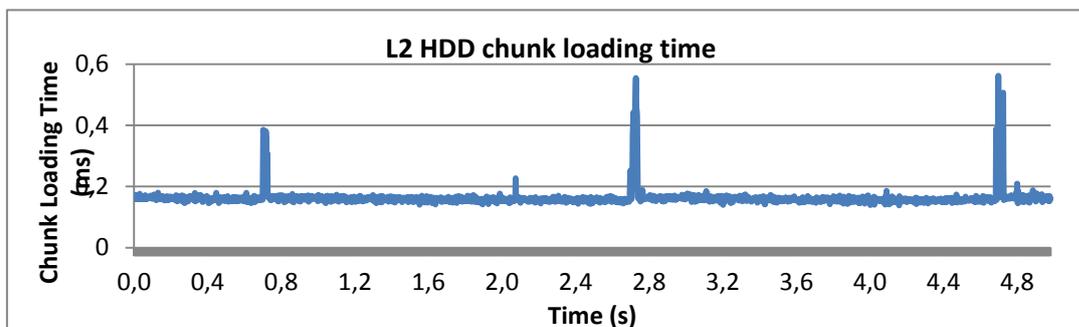


Ilustración 53. Gráfica de tiempos de carga de chunk, de HDD a RAM, por la caché L2 (M=5000).

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.

10.4 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

10.5 ESTIMACIÓN DE NORMALES

En esta sección se pone a prueba el estimador de normales mediante PCA, explicado en capítulos anteriores. Dado que las pruebas están realizadas con datos LIDAR reales, no es posible realizar una comprobación analítica. Pero una inspección visual puede ser suficiente para comprobar que la aproximación funciona correctamente.

Para ello, *pcfly* permite renderizar el modelo, en vez de con sus colores reales, con *el color de la normal*. Esto es, mapeando la normal al color RGB de la siguiente manera

$$(x, y, z) \rightarrow (r, g, b) \text{ donde } \begin{cases} r = 256x \\ g = 256y \\ b = 256z \end{cases}$$

Esto quiere decir, que si una normal está completamente en la dirección del eje x, se pintará el punto rojo. En el eje y, verde. En el eje z, azul. Las combinaciones producirán degradados entre estos tres colores. Los resultados pueden observarse en las siguientes imágenes (Ilustración 54).



Ilustración 54. Inspección visual de la aproximación de las normales de los puntos.

Nótese en las imágenes de arriba – izquierda y abajo – derecha el degradado dependiendo de la curvatura de la superficie. En la imagen de arriba –derecha, el edificio está perfectamente alineado con los ejes x e y. Por último, en la imagen de abajo – izquierda, los árboles, presentan ruido en su color. Esto es porque los puntos caen en la hojas, cada una orientada de una forma aleatoria.

10.6 DATOS DE RENDER

Por último, se muestran aquí algunos **datos analíticos de una sesión de render**. Están recogidos por el sistema de monitorización de PCM, durante unos pocos segundos, en los que se han realizado 4 movimientos bruscos de cámara, marcados con las flechas azules en las gráficas.

En primer lugar, se monitoriza el **traversal del árbol para la selección de nodos de la query** que se pedirá a la jerarquía de memoria (Ilustración 55). Se compara el número final de nodos seleccionados, aquellos que han pasado los test de nivel máximo de detalle y *culling*, en comparación con el coste en tiempo del *traversal*. Se puede observar una relación directa entre estos dos datos, sobre todo en las variaciones por movimientos bruscos de cámara.

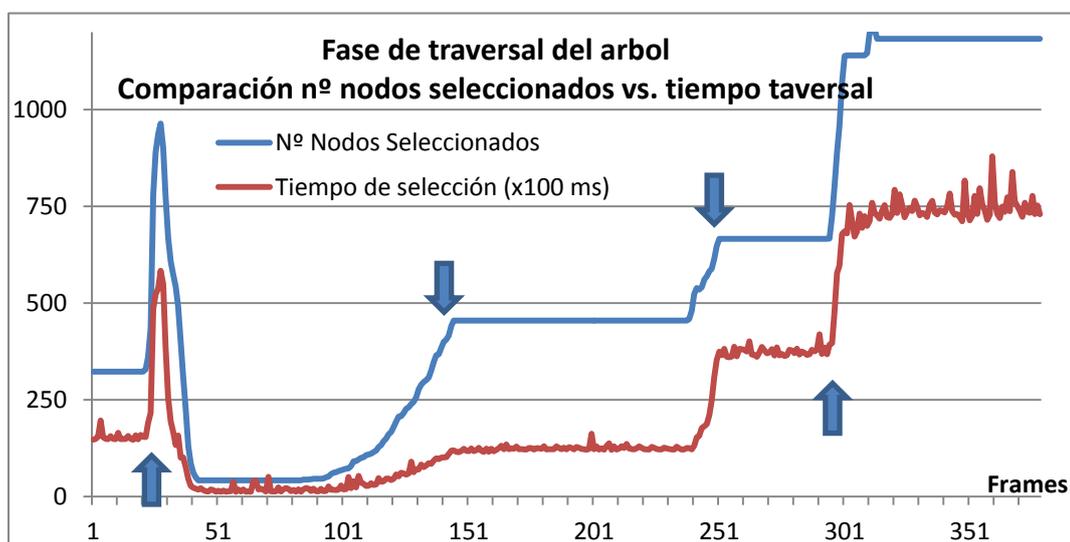


Ilustración 55. Monitorización de una sesión de render. Datos de *traversal* del árbol, para la selección de la *query* de nodos.

A continuación, algunos datos propios del proceso de *render* (Ilustración 56). En este caso, se comparan las relaciones entre la completitud del modelo cargado, es decir, qué porcentaje de resolución alcanza; el número de chunks (en *VertexBufferObjects* de OpenGL) que dibuja porque ya tiene disponibles en VRAM; y el *Framerate*, medido en términos de porcentaje respecto a la referencia de interactividad de 25 fps.

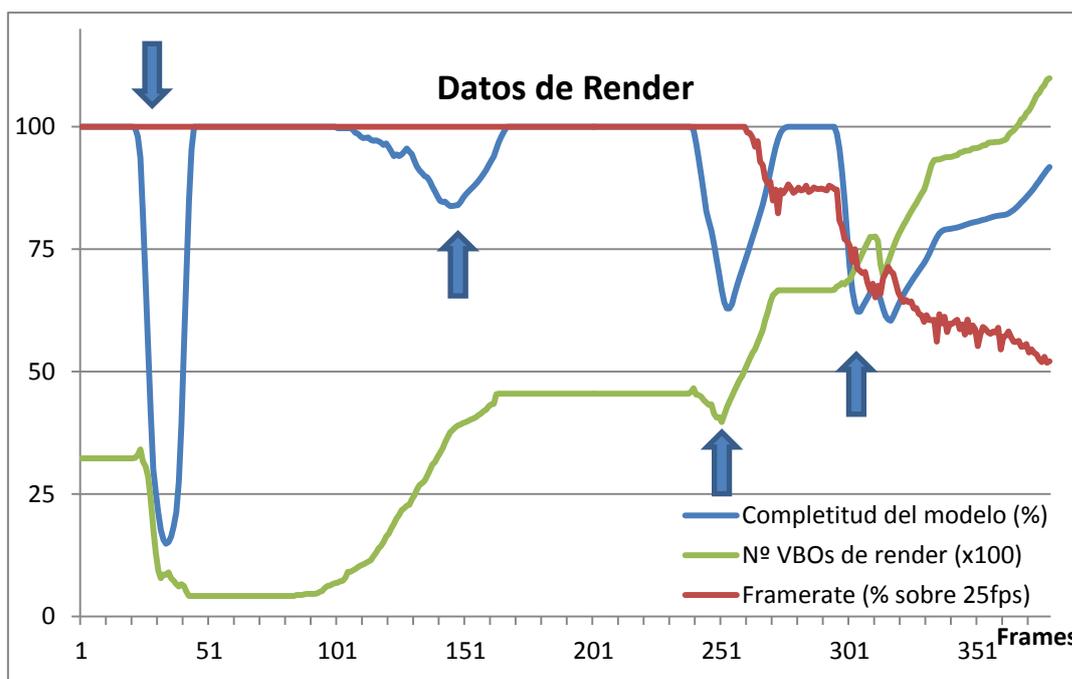


Ilustración 56. Monitorización de una sesión de render. Datos de de render.

También en este caso se puede observar una clara relación entre estos datos. El *framerate* se mantiene por encima del límite óptimo de interactividad (25fps) hasta que el número de VBOs a dibujar es excesivo, (en este caso, aproximadamente 7000; es decir unos 14M de puntos o splats).

Por otra parte, y lo más importante, es que **a pesar de los movimientos bruscos de la cámara, el framerate se mantiene líneal y suave, y la completitud (nivel de detalle) se carga progresiva y rápidamente hasta mostrar la máxima calidad**. Lo cual demuestra, en líneas generales, el correcto funcionamiento del sistema.

11 CONCLUSIONES Y FUTUROS DESARROLLOS

En los capítulos dedicados a introducción, 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.

Gracias a esta estructura, se ha desarrollado una técnica de visualización interactiva capaz de permitir al usuario explorar interactivamente y en tiempo real el *dataset*, independientemente de su tamaño o la potencia de la máquina utilizada.

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.

11.1 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.

El visualizador se encuentra también en fase de *testing*, y muchos de los algoritmos tienen una implementación *naive*, todavía poco optimizada.

11.2 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 subespacios 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.

Particularmente, en lo referido a la visualización interactiva:

- Implementar una detección de la potencia gráfica de la GPU del equipo donde se ejecute, para configurar más adecuadamente las posibilidades de render en tiempo de ejecución.
- Implementar el *occlusion culling*.
- Mejorar la detección del centro de detalle, según las heurísticas de percepción humana que se han comentado.
- Implementar filtrado EWA en el render de puntos [28].

11.3 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 encuentran en preparación diversas publicaciones científicas 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 la base de la tesis doctoral del autor. Además, está siendo una librería base en el desarrollo de otros Proyectos Fin de Carrera.

El segundo aspecto, sobre la búsqueda de la explotación comercial del sistema, en realidad actualmente es ya la base de un proyecto I+D financiado con fondos CDTI.

12 NOTA LEGAL



This work has been partially supported by **DIVA ITN** project funded by People Programme (Marie Curie Actions) of the European Union's Seventh Framework Programme FP7/2007-2013/ under REA grant agreement n°290227. (<http://diva-itn.ifi.uzh.ch/>).

El *dataset* LIDAR de la Iglesia Peregrina de Pontevedra (España) pertenece a la empresa **ENMACOSA** y ha sido cedido con fines científicos al autor de este trabajo, dentro del marco de colaboración impuesto por el proyecto **ToVIAS**, financiado por los fondos del CDTI del Ministerio de Economía del Gobierno de España.

13 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: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994.
- [3] A. Jaspe, «Point Cloud Manager. Sistema multirresolución para el tratamiento de grandes datasets de nubes en puntos 3D,» A Coruña, 2012.
- [4] Pixar Inc., «CGSociety - Pixar Points,» [En línea]. Available: http://www.cgsociety.org/index.php/CGSFeatures/CGSFeatureSpecial/pixar_points. [Último acceso: 2012].
- [5] 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].
- [6] Stanford University, «The PLY File Format,» [En línea]. Available: http://www.cc.gatech.edu/projects/large_models/ply.html. [Último acceso: 2012].
- [7] 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.
- [8] H. Samet, «Spatial Data Structures,» 2012.
- [9] 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>.
- [10] R. Sugumaran, D. Orispayev y P. Gray, «GPU-based cloud performance for LiDAR data processing,» de *2nd International Conference on Computing for Geospatial Research & Applications*, 2011.
- [11] F. Qiu y C. Yuan, «Parallel processing of massive LiDAR point clouds using CUDA enabled GPU,» de *AAG Annual Meeting*, 2012.
- [12] M. Gross y H. Pfister, *Point-Based Graphics*, Morgan Kaufmann, 2007.

- [13] S. Rusinkiewicz y M. Levoy, «QSplat: a multiresolution point rendering system for large meshes,» de *SIGGRAPH*, 2000.
- [14] E. Gobbetti y F. Marton, «Layered Point Clouds,» de *Eurographics Symposium on Point Based Graphics*, Zurich, Switzerland, 2004.
- [15] 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.
- [16] 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.
- [17] Open Perception, «Point Clouds Library,» [En línea]. Available: <http://pointclouds.org>. [Último acceso: 2012].
- [18] 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.
- [19] 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].
- [20] T. K Dey, G. Li y J. Sun, «Normal Estimation for Point Clouds: A Comparison Study,» de *Eurographics 2006*, 2006.
- [21] Open Perception Fundation, «Estimating Surface Normals in a PointCloud,» [En línea]. Available: http://pointclouds.org/documentation/tutorials/normal_estimation.php. [Último acceso: 2013].
- [22] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald y W. Stuetzle, «Surface reconstruction from unorganized points,» de *SIGGRAPH'90*, 1992.
- [23] UNC Charlotte, «Spatial Structures and Acceleration Algorithms,» 2010. [En línea]. Available: http://coitweb.uncc.edu/~krs/courses/5010/ged/lectures/cull_lod2.pdf.

- [24] M. Botsch y L. Kobbelt, «High-quality point-based rendering on modern GPUs,» *Computer Graphics and Applications*, 2003.
- [25] K. B. e. al., «Manifiesto for Agile Software Development,» 2001. [En línea]. Available: <http://agilemanifesto.org/>.
- [26] Edgewall Software, «The TRAC Project,» [En línea]. Available: <http://trac.edgewall.org/>. [Último acceso: 2012].
- [27] R. Osfield, «Open Scene Graph,» [En línea]. Available: <http://openscenegraph.org>.
- [28] L. Ren, H. Pfister y M. Zwicker, «Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering,» *Computer Graphics Forum*, 2002.