



UNIVERSIDADE DA CORUÑA

FACULTAD DE INFORMÁTICA

Departamento de Electrónica y Sistemas

Proyecto de Fin de Carrera de Ingeniería Informática

Sistema cliente-servidor para la visualización de nubes de puntos con WebGL

Autor: Javier Rey Neira

Directores: Emilio José Padrón González

Alberto Jaspe Villanueva

A Coruña, 16 de septiembre de 2013

Especificación

Título: Sistema cliente-servidor para la visualización de nubes de puntos con WebGL

Clase: Proyecto de Investigación

Autor: Javier Rey Neira

Directores: Emilio José Padrón González

Alberto Jaspe Villanueva

Tribunal:

Fecha de lectura:

Calificación:

A mis padres por todo su apoyo durante todos estos años.

Agradecimientos¹

A don Alberto Jaspe Villanueva y a don Emilio José Padrón González por toda la ayuda dada durante este proyecto.

¹en orden alfabético

Resumen

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

PCM es una biblioteca para la gestión y procesado de grandes datasets de nubes de puntos en equipos convencionales, independientemente de las capacidades hardware del equipo y del tamaño de la nube de puntos. Consiste en una estructura de datos espacial multirresolución y de un sistema de caché software de dos niveles. En este proyecto se usa una arquitectura cliente-servidor para, usando esta librería PCM, ampliar sus posibilidad de forma que el usuario no necesite ningún tipo de información y pueda acceder a los datasets desde un simple navegador web mediante HTML y WebGL.

Palabras clave

Nubes de puntos, caché software, niveles de detalle, arquitectura cliente-servidor, estructuras de datos espaciales, WebGL, protocolo de comunicación, WebSocket.

Índice general

1. Introducción	1
1.1. Motivación y objetivos	2
1.2. Organización de la memoria	3
2. Metodología y planificación	5
2.1. Metodologías ágiles	5
2.2. Metodología usada en el proyecto	8
2.3. Planificación	9
2.4. Herramientas utilizadas	9
3. Contextualización y fundamentos	11
3.1. Conceptos básicos	11
3.1.1. Nubes de puntos	11
3.1.2. Obtención de nubes de puntos	12
3.1.3. Sistemas multirresolución	12
3.1.4. Estructuras de datos espaciales	14
3.1.5. Jerarquías de memoria	14
3.2. Point Cloud Manager (PCM)	15
3.2.1. Arquitectura general	16
3.2.2. Estructura de datos	18
3.2.3. Jerarquía memoria	19
3.2.4. Cachés software	21
3.2.5. Sistema de cámaras	22

3.3. Paradigma Cliente-Servidor	22
3.3.1. WebSocket	23
3.3.2. WebGL	26
4. Estructura del sistema	27
4.1. Análisis	27
4.2. Reparto de tareas	29
4.3. Funcionamiento de la comunicación y flujo de mensajes	30
5. Protocolo	37
5.1. Estructura	37
5.2. Tipos de mensaje	37
5.2.1. Consulta de modelos	38
5.2.2. Inicio	38
5.2.3. Intercambio	40
5.2.4. Finalización	42
6. Servidor	43
6.1. Funcionamiento del servidor e integración con PCM	44
6.2. WebSocket	48
6.2.1. WebSocket++	48
6.3. Caché LRU	52
6.4. Diagrama de clases	54
7. Cliente	57
7.1. WebGL y funcionamiento general	57
7.2. Websocket	58
7.3. Web Workers	59
7.4. Diagrama de clases	59
8. Resultados	63
8.1. Mediciones tomadas	66

9. Conclusiones	69
9.1. Cumplimiento de requisitos y objetivos	69
9.2. Posibles mejoras	70
9.3. Futuro	71
9.4. Código fuente del proyecto	71

Índice de figuras

3.1. Diagrama de la Jerarquía de memoria	15
3.2. Arquitectura del paquete software PCM	16
3.3. Construcción de un kd-tree	19
3.4. Distribución de los datos para generar niveles de detalle aditivos.	20
3.5. Sistema de gestión de la jerarquía de memoria.	20
3.6. Flujo de comunicación cliente-servidor con AJAX.	24
3.7. Flujo de comunicación cliente-servidor con Long Polling.	24
3.8. Flujo de comunicación cliente-servidor con WebSocket.	25
4.1. Esquema general del sistema.	28
4.2. Captura del inicio de la aplicación web con la lista de los datasets.	31
4.3. Proceso a seguir en el servidor por cada petición de chunks del cliente.	33
4.4. Diagrama de secuencia de una comunicación común entre el cliente y el servidor.	35
6.1. Diagrama estructural de la relación de PCServer con PCM.	44
6.2. Diagrama de secuencia de una petición GET en la que todos los datos están en cache.	46
6.3. Diagrama de actividad del servidor ante una petición de chunks desde cliente.	47
6.4. Diagrama de clases PCServer.	55
7.1. Diagrama de clases del cliente web.	60

8.1. Captura del cliente web con el modelo Casas.	64
8.2. Captura del cliente web con el modelo Peregrina.	65

Índice de tablas

2.1. Planificación del proyecto	9
---	---

Capítulo 1

Introducción

Hace ya tiempo que el procesamiento de datos de carácter espacial se ha vuelto de uso común en múltiples disciplinas. No obstante, en los últimos años, gracias al abaratamiento y aumento en la precisión de los sistemas LIDAR (Light Detection and Ranging) [1], este tipo de tecnologías se han popularizado en numerosos campos de la ingeniería y otras áreas. Estos sistemas LIDAR pueden recoger distinta información de cada punto en el espacio aparte de sus coordenadas, como puede ser color, temperatura, índice de reflectividad, etc. formando todos juntos un gran dataset denominado Nube de Puntos (Point Cloud).

Todos estos datos pueden alcanzar cantidades miles de millones de puntos llegando a formar datasets de varios Gigabytes o incluso Terabytes cuyo procesamiento puede requerir del uso de supercomputadoras del ámbito de “Big Data”. En este contexto surge en 2012, como Proyecto de Fin de Master de esta facultad, Point Cloud Manager (PCM) [2], una librería que ofrece una solución para el tratamiento de grandes datasets de nubes de puntos 3D en PCs de escritorio.

En este documento se presenta una extensión a la librería PCM que se encargará de la transmisión de la información de los puntos a través de Internet para su visualización en un navegador Web. Este estará compuesto por una parte servidor, una parte cliente, en nuestro caso un cliente web, y un protocolo mediante el cual se comunicarán .

1.1. Motivación y objetivos

Aunque una biblioteca como PCM ofrece mediante una interfaz software sencilla y clara una abstracción de una nube de puntos arbitrariamente grande, permitiendo el manejo y la gestión de grandes datasets en equipos de uso común, su diseño presenta algunas desventajas en escenarios en los cuales no existe la posibilidad de un acceso de forma directa a los datos: bien porque no se dispone de los recursos suficientes para el almacenamiento de este tipo de información (pensemos en una biblioteca o archivo de varios de datasets, de muchos GigaBytes cada uno), bien porque el acceso a estos datos está restringido.

Así, el diseño original de PCM considera un único escenario en el que la máquina que tiene acceso a los datos es la misma máquina que necesita realizar los cálculos o visualización sobre esos datos. La motivación para este proyecto pasa por eliminar esa restricción extendiendo PCM con una arquitectura cliente-servidor que además permita la visualización de esos grandes datasets en dispositivos variados mediante un estándar abierto como es WebGL.

El objetivo por lo tanto es implementar una solución cliente-servidor mediante la cual un cliente se comunicará con un servidor que le ofrecerá los datos que necesita sin necesidad de enviarle todos los datos completos de la nube de puntos. Así el cliente podrá ir visualizando poco a poco, según los datos van llegando, almacenando únicamente los datos necesarios y sin tener que ni siquiera gestionar nada de la nube de puntos a visualizar.

Con esto se logra no solo que el cliente no requiera de una gran capacidad de almacenamiento sino que tampoco disponga de un gran nivel de procesado, ya que todo el peso de la computación y el acceso a los datos sería realizado por un servidor de altas prestaciones ya optimizado para ello. El cliente sólo necesitará mostrar los puntos reunidos a través de un cliente web, lo que requerirá cierta cantidad de computo gráfico pero totalmente administrable por cualquier PC actual.

Para alcanzar esa meta principal, que es la comunicación de datos de los puntos que sean necesarios entre un cliente y un servidor a través de Internet, se establecen los siguientes objetivos:

1. Creación de un protocolo para la transferencia bajo demanda de nubes de puntos de tamaño arbitrario.
2. Diseño e implementación de un servidor de nubes de puntos integrado en el paquete software PCM, que se beneficiará de la gestión de complejos datasets que ofrece la biblioteca PCM.
3. Diseño e implementación de un visualizador HTML5+WebGL para la visualización avanzada de los datasets desde una máquina remota.

Además el diseño y la implementación cumplirán los siguientes requisitos:

1. Multiplataforma, para seguir con la filosofía de PCM, se hará compatible con los principales sistemas operativos, Microsoft Windows, Linux y Mac OSX.
2. Multicliente, el servidor se diseñará de forma que acepte distintas conexiones desde diferentes clientes.

1.2. Organización de la memoria

La presente memoria se divide en los siguientes capítulos:

- **Capítulo 1: Introducción.** Se presenta una pequeña introducción al proyecto explicando su motivación, contexto y objetivos. También se describe la organización de la memoria.
- **Capítulo 2: Metodología y planificación.** En este capítulo se presenta la planificación que se ha llevado a cabo en este proyecto dando previamente una introducción al tipo de metodología que se ha usado, enmarcado en lo que se conoce como metodologías ágiles.
- **Capítulo 3: Contextualización y Fundamentos.** Se presentan las bases necesarias para la comprensión del presente proyecto. Se explican conceptos básicos previos, funcionamiento y arquitectura de PCM y la arquitectura cliente-servidor.

- **Capítulo 4: Estructura general del sistema.** En este capítulo se explica la estructura del proyecto desde un punto de vista general viendo cómo interactúan todas las partes que lo forman.
- **Capítulo 5: Protocolo.** Descripción del protocolo que es necesario para la transmisión de datos entre el cliente y el servidor.
- **Capítulo 6: Servidor.** Se explica detalladamente el funcionamiento del servidor y cómo este se integra con PCM.
- **Capítulo 7: Cliente.** Este capítulo explica ampliamente el funcionamiento de cliente, como este muestra la nube de puntos y se comunica con el servidor.
- **Capítulo 8: Resultados.** En este capítulo se exponen algunos datos significativos sobre el funcionamiento del sistema.
- **Capítulo 9: Conclusiones.** Se exponen las conclusiones a las que se ha llegado tras su realización y sus líneas de trabajo futuras.

Capítulo 2

Metodología y planificación

Uno de los factores más importantes a la hora de desarrollar software no es solo modelar con sus respectivas herramientas sino, previamente, establecer un esquema a seguir para la aplicación de los todos los elementos establecidos por una metodología que pueda asegurar el éxito del desarrollo de software.

2.1. Metodologías ágiles

Tradicionalmente se han usado metodologías formales, las cuales se han demostrado que son muy efectivas y necesarias para proyectos grandes, no obstante estas metodologías se convierten en ocasiones en excesivamente pesadas y rígidas, por culpa de su carácter normativo y por su dependencia a planificaciones previas al desarrollo, llegando a ser un obstáculo para desarrollos pequeños que son implementados por pocas personas.

Por este motivo, en 2011 se reúnen diecisiete críticos para tratar sobre las técnicas y procesos para desarrollar software rápidamente, intentando buscar una alternativa a los modelos tradicionales, muy costosos para desarrollos pequeños. En esta reunión se estableció el término “Métodos Ágiles” y con él se crea The Agile Alliance [3], organización dedicada a divulgar los términos establecidos para el desarrollo ágil de software.

Con este objetivo crean el Manifiesto Ágil [4], en el cual se establecen los siguientes valores:

- Individuos e interacciones sobre procesos y herramientas.
- Software funcionando sobre documentación extensiva.
- Colaboración con el cliente sobre negociación contractual.
- Respuesta ante el cambio sobre seguir un plan.

Estos valores inspiran los doce principios establecidos en el manifiesto:

- La mayor prioridad es satisfacer al cliente la entrada temprana y continua de software de valor.
- Se acepta que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos ágiles aprovechan el cambio para promocionar ventaja competitiva al cliente.
- Entregar software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.
- Los responsables de negocio y los desarrolladores tiene que trabajar juntos de forma cotidiana durante todo el proyecto.
- Los proyectos se desarrollan en torno a individuos motivados. Tienen que tener un entorno y el apoyo que necesiten, confiandoles la ejecución del trabajo.
- El método más eficiente y efectivo de comunicar información al equipo de desarrollo entre sus miembros es la conversación cara a cara.
- El software funcionando es la medida principal de progreso
- Los procesos ágiles promueven el desarrollo sostenible. Los promotores, desarrolladores y usuarios deben de ser capaces de mantener un ritmo constante de forma indefinida.
- La atención continua a la calidad técnica y el buen diseño mejoran la agilidad.
- La simplicidad es esencial.

- Las mejores arquitecturas, requisitos y diseños surgen de equipos auto-organizados.
- A intervalos regulares el equipo debe reflexionar sobre cómo ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia.

Los principales problemas presentados por las metodologías tradicionales, los cuales hicieron necesarios la necesidades de las nuevas metodologías ágiles, son:

- Fases previas de especificación de requisitos, análisis y diseño muy costosas. Además la corrección de cualquiera de estas presentará un coste muy alto.
- Proceso de desarrollo ralentizado por la necesidad de firmar los documentos.
- Desarrollo lento debido a la complejidad para los desarrolladores de entender el sistema en su globalidad.

Por estos motivos las metodologías ágiles adquieren gran importancia en proyectos que no están perfectamente definidos y que, por tanto, tendrán unos requisitos cambiantes. Si se divide el trabajo en módulos pequeños para minimizar los fallos y el coste, esta metodología se puede también utilizar en grandes sistemas, aunque donde adquieren mayor importancia es en equipos de desarrollo pequeños. Sus ventajas son:

- Capacidad de respuesta ante cambios de requisitos durante el desarrollo.
- Entrega continua en breves plazos de software funcional.
- Trabajo conjunto entre cliente y desarrolladores.
- Importancia de la simplicidad, eliminando el trabajo innecesario.
- Atención continua a la excelencia técnica y al buen diseño.
- Mejora continua de los procesos y del equipo de desarrollo.

Dentro de la metodologías ágiles podemos encontrar varios tipos de estas. Algunos ejemplos pueden ser: Agile Unified Process (AUP), Dynamic Systems Development Method (DSDM), Feature Driven Development (FDD) o SCRUM [5].

En este proyecto se usará una variante propia y simplificada de SCRUM. Esta metodología fue desarrollada por Ken Schwaber y Jeff Sutherland en los años 90 y más tarde, en 2001, Schwaber junto a Mike Beedle describieron el método en su libro “Agile Software Development with Scrum”. Esta metodología se ha venido usando en los últimos 10 años con gran éxito proyectos con cambios rápidos en sus requisitos. Sus dos principales características son los sprints y las reuniones. Los sprints consisten en pequeñas iteraciones, con una recomendación de duración de entre 1 y 4 semanas, en las cuales se obtiene un ejecutable válido para mostrarlo al cliente. Las reuniones se realizarán a lo largo del proyecto, destacando una reunión diaria de 15 minutos para coordinación e integración.

2.2. Metodología usada en el proyecto

En este proyecto es complejo realizar una planificación de largo plazo debido a su naturaleza de proyecto investigador y el desconocimiento de las tecnologías utilizadas. Por tanto, tras una planificación inicial muy somera, la idea consiste en ir poco a poco avanzando en pequeños progresos viendo que problemas van apareciendo y que mejoras se pueden ir implementando mediante progresa el proyecto. Esto lleva a unos frecuentes cambios de requisitos, por lo que establecer un plan y seguirlo estrictamente se convierte en una tarea muy compleja. Al mismo tiempo, debido a su corta duración, es importante que se planifique de forma simple y eliminar todo el trabajo que pueda ser innecesario.

Todo esto lleva a decantarnos por una metodología ágil. Exactamente se selecciona una versión de SCRUM relajada adaptada al proyecto según las necesidades. Cada iteración, denominada sprint, tendrá una duración aproximada entre 7 y 14 días. A causa del tipo de proyecto, en lugar de realizar una reunión diaria de 15 minutos se realizará una reunión cada finalización de un sprint y su duración será mayor. En estas se analizará la versión realizada en el último sprint, problemas aparecidos, posibles nuevas características y preparación para el próximo sprint. En total, en este proyecto, se realizaron 10 sprints los cuales se detallan a continuación.

2.3. Planificación

Al inicio de proyecto se realiza una planificación base para establecer un plan inicial a seguir y unos objetivos a cumplir, sin establecer un marco temporal fijo. En la siguiente tabla se describen las tareas realizadas en cada sprint, junto a su duración y fechas tanto de inicio como de finalización.

Tabla 2.1: Planificación del proyecto

Tareas	Duración	Fecha de inicio	Fecha de finalización
Estudio de la documentación relacionado con el proyecto.	15 días	1 mayo 2013	15 mayo 2013
Análisis del estado del arte y toma de decisiones.	15 días	16 mayo 2013	30 mayo 2013
Elaboración de la primera versión del protocolo para la comunicación entre cliente y servidor.	8 días	31 junio 2013	7 junio 2013
Diseño del servidor.	8 días	8 junio 2013	15 junio 2013
Implementación del servidor.	15 días	16 junio 2013	30 junio 2013
Diseño del cliente.	8 días	1 julio 2013	8 julio 2013
Implementación del cliente.	15 días	9 julio 2013	23 julio 2013
Integración cliente servidor.	15 días	24 julio 2013	7 agosto 2013
Revisión y optimización tanto de código del cliente como de la parte servidor.	19 días	8 agosto 2013	26 agosto 2013
Realización de la memoria.	19 días	26 agosto 2013	13 septiembre 2013

2.4. Herramientas utilizadas

Las herramientas utilizadas se dividen en dos partes, servidor y cliente. Para la parte servidor se ha utilizado el lenguaje de programación C++, empleando el IDE Visual Studio 2010, aunque cualquier compilador convencional para código C++ en cualquier SO es válido. Las librerías necesarias son PCM y WebSocket++ [6]. La librería websocket++ es una librería que implementa el protocolo WebSocket [7] y también las funcionalidades básicas de comunicación del servidor. Para la parte de comunicación se basa en Boost Asio[8], por lo que esta librería también será necesaria. En la parte cliente será necesario simplemente cualquier navegador web que sea compatible con Websocket, WebGL y WebWorkers [9], como por ejemplo Chrome.

Una herramienta de las más útiles para la organización del proyecto ha sido Git [10]

[11]. Git es un software de control de versiones diseñado por Linux Torvalds válido para cualquier tipo de proyecto, independientemente de su tamaño, con una gran velocidad y eficacia. Git es software libre y se distribuye bajo una licencia GNU GPL v2 [12]. No depende de un repositorio central y a cada programador se le da una copia local del proyecto y se le permite crear nuevas ramas locales sin que esta afecte al resto del desarrollo, pudiendo después ser fusionados con cualquier otra rama. Esto ha resultado muy útil ya que se ha podido trabajar paralelamente con PCM, el cual sigue en desarrollo, creando ramas distintas para más tarde combinarlas.

Por último, la presente memoria ha sido confeccionada con LaTeX [13], un sistema de composición de documentos basada en un lenguaje de marcado de documentos.

Capítulo 3

Contextualización y fundamentos

Este capítulo explica los conceptos básicos en los distintos ámbitos que enmarca este proyecto. Primero se describen los fundamentos básicos sobre nubes de puntos, cómo estas pueden ser recogidas por escáneres LIDAR y otros conceptos importantes para la comprensión de estas. Se continuará comentando la solución dada por PCM para tratar estas nubes de puntos, explicando tanto su funcionamiento como estructura. Por último se describirán los conceptos básicos sobre el modelo de arquitectura cliente-servidor propuesto para extender PCM, basado en la comunicación mediante WebSockets y la visualización con WebGL.

3.1. Conceptos básicos

A continuación se definen unos conceptos previos necesarios para la comprensión de PCM y de este proyecto.

3.1.1. Nubes de puntos

Se llama “nube de puntos” o “point cloud” a un conjunto formado por vértices en un sistema de coordenadas tridimensional sin ningún tipo de topología que los relacione. Estos vértices se definen por sus coordenadas XYZ y suelen representar muestras de las superficies externas de los objetos.

En una nube de puntos cada muestra o punto tiene asociada una cantidad de infor-

mación como puede ser color del punto, normal de la superficie, reflectividad, temperatura, etc. Es decir, el tamaño total de un punto no es sólo su posición sino también toda la información asociada a dicho punto, que puede ser arbitraria.

3.1.2. Obtención de nubes de puntos

LIDAR es una tecnología capaz de determinar la distancia desde un emisor láser a un objeto o una superficie utilizando un haz láser pulsado, midiendo la diferencia de tiempo entre la emisión y recepción de este haz. A mayores, según la intensidad y forma en la que se ha recibido, se puede detectar también el índice de reflectividad del material muestreado. Si se automatiza este proceso, en una sesión podemos tomar cientos o miles de muestras, cada una de las cuales tendrá unas coordenadas espaciales en el marco de referencia establecido inicialmente. Los posibles errores de medición estarán determinados por la calibración y calidad del equipo utilizado.

Pero no sólo existen los LIDAR para capturar nubes de puntos, existen otros tipos de tecnologías denominadas escáneres 3D capaces de adquirir estos tipos de datasets. Aunque pueden estar basados en distintas tecnologías en general todos se basan en un funcionamiento parecido: emisión y recepción de algún tipo de radiación o pulso.

Un ejemplo de estos sistemas, que recientemente ha estado en auge gracias a su bajo coste, sería el Kinect de Microsoft. Para la captura de puntos se emite un patrón de luz infrarroja emitida continuamente en una determinada frecuencia y se captura el rebote de estas emisiones mediante una cámara especial. Con esto logra poder obtener una nube de puntos dinámica en tiempo real.

Las nubes de puntos no tienen porque ser obtenidas siempre desde el mundo real, también pueden proceder de procesos computacionales o algoritmos y como resultado por ejemplo de una simulación.

3.1.3. 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, permitiendo el análisis multirresolución (multiresolu-

tion analysis o MRA) o aproximaciones multiescala (multiscale approximation o MSA). Estos métodos permiten obtener soluciones cada vez más precisas a procesos complejos utilizando distintas escalas de detalle del modelo (niveles de detalle, levels of detail o LODs), usados muy comúnmente en tratamiento de imágenes digitales y en algoritmos de compresión y codificación.

Los sistemas multirresolución y niveles de detalle forma una práctica habitual para casi cualquier motor de render [14]. En los modelos 3D uno de los parámetros importantes que establece la calidad visual del modelo son la cantidad de polígonos que se muestran. Pero estos tiene un gran coste de procesado y ocupan una gran cantidad de memoria. Por este motivo son necesarios estos sistemas para usar simplificación del modelo según la distancia que se encuentran los polígonos desde el punto de vista. En el caso de las texturas de los modelos pasa exactamente lo mismo, procesar una textura de gran tamaño resultaría una gran pérdida de tiempo si esta después está situada en un objeto muy lejano que casi no se ve desde el punto de vista actual.

Estos sistemas producen un sobre coste, siempre menor al coste ahorrado por su utilización, que se puede resumir en tres necesidades:

1. Pre-proceso de los datos originales: es necesario crear modelos simplificados a partir de los originales y generar una estructura de datos espacial que gestionará los niveles de detalle.
2. Aumento de los requisitos de memoria: aunque se ahorrara tamaño de memoria en las partes altas de la jerarquía de memoria, menor consumo de memoria RAM, el consumo de memoria en la parte inferior de la jerarquía (HDD) aumenta considerablemente. Esto se debe a que hay que almacenar no sólo el modelo original sino también los distintos niveles de detalle.
3. Gestión de los niveles de detalle en tiempo de ejecución: para poder escoger en cada momento el nivel de detalle más adecuado es necesario crear y mantener una estructura de datos de tipo espacial.

El primer punto es insalvable pero no presenta un gran problema, es un proceso que se produce antes de la ejecución y que basta con ejecutarlo solamente la primera

vez. El segundo punto tampoco es un factor crítico, a día de hoy la capacidad de almacenamiento estática (HDD) alcanza grandes cantidades y siempre es superior en varios órdenes de magnitud a la memoria del sistema (RAM).

3.1.4. Estructuras de datos espaciales

Si tenemos datos que tiene alguna relación espacial entre ellos lo normal es que los algoritmos que tratan estos datos tenga también un carácter espacial. Para esto se hace necesario una estructura que tengan en cuenta estas relaciones en cuanto sus posiciones en el espacio, la pre-procese y genere una meta-información capaz de facilitar accesos más rápidos que faciliten los operaciones sobre los datos.

La estructura más utilizada para esto es el árbol, la cual nos facilita de forma natural, gracias a la división recursiva de sus nodos, una organización espacial. Si no se utilizará ningún tipo de estructura espacial la complejidad de las operaciones sería mucho más alto. Por ejemplo, si queremos determinar los datos que contiene una determinada región del espacio tendríamos que ir comprobando dato a dato si está dentro de la región que nos interesa, complejidad lineal $O(n)$. Con una estructura en árbol la complejidad sería constante, $O(1)$, porque gracias a la recursión a través de sus nodos es posible establecer un camino en las ramas del árbol que contiene la región indicada.

Existen otros tipos de estructuras jerárquicas según el tipo de dato (puntos, polígonos, superficies, etc.) y según como se quieran recorrer estos. Algunos ejemplos puede ser rejillas regulares, árboles BSP, jerarquías de geometría sólida, jerarquías de bounding spheres, etc. [15] [16]

PCM hace uso de la estructura árbol K-Dimensional (kdtree), especialmente pensada para datos asociados solo a coordenadas en el espacio. Esta estructura es especialmente adecuada para datos LIDAR.

3.1.5. Jerarquías de memoria

La jerarquía de memoria es una organización piramidal (Figura 3.1) de la memoria utilizada por los ordenadores para conseguir el rendimiento de una memoria de gran

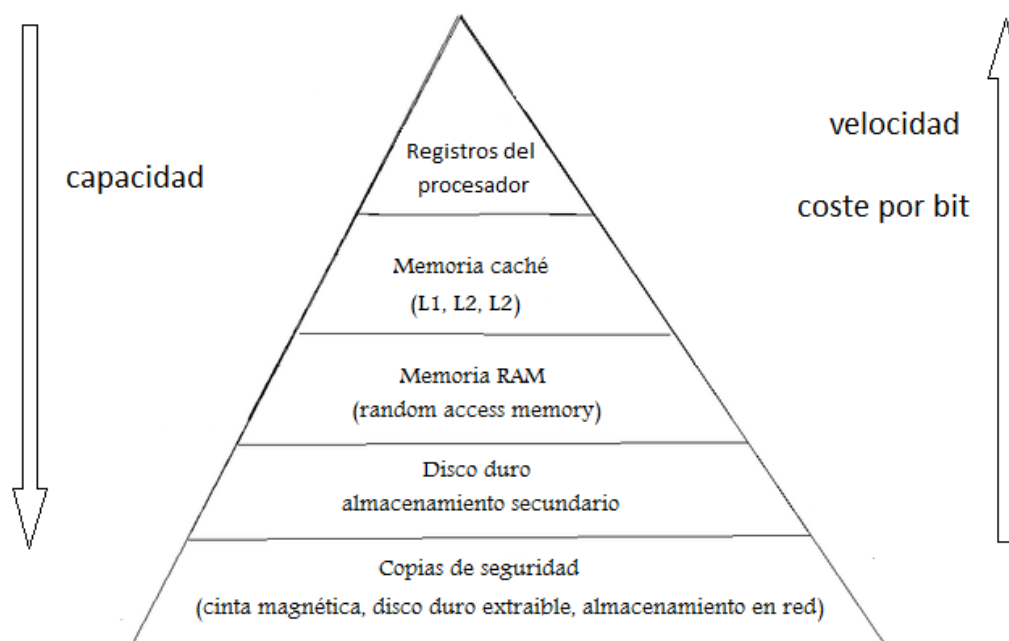


Figura 3.1: Diagrama de la Jerarquía de memoria

velocidad al coste de una de baja velocidad basándose en el principio de cercanía de referencias cercanas entre sí.

Su organización se basa en que, según se sube en la pirámide, el coste por bit y velocidad aumenta pero si bajamos estos se ven reducidos. De esta forma, como el principio de localidad temporal establece que si un dato es utilizado este posiblemente será requerido de nuevo próximamente, cada vez que un dato es necesitado se puede ir subiendo a través de la jerarquía de memoria. Además, el principio de localidad espacial nos indica que probablemente los datos vecinos también hagan falta próximamente.

3.2. Point Cloud Manager (PCM)

Point Cloud Manager [2] consiste en una solución software capaz de convertir, visualizar y procesar datasets que provengan de escáneres láser LIDAR u otras fuentes. A continuación se explica toda su arquitectura dándole especial importancia a la estructura de datos y a las cachés que implementa ya que este proyecto se basa en esos

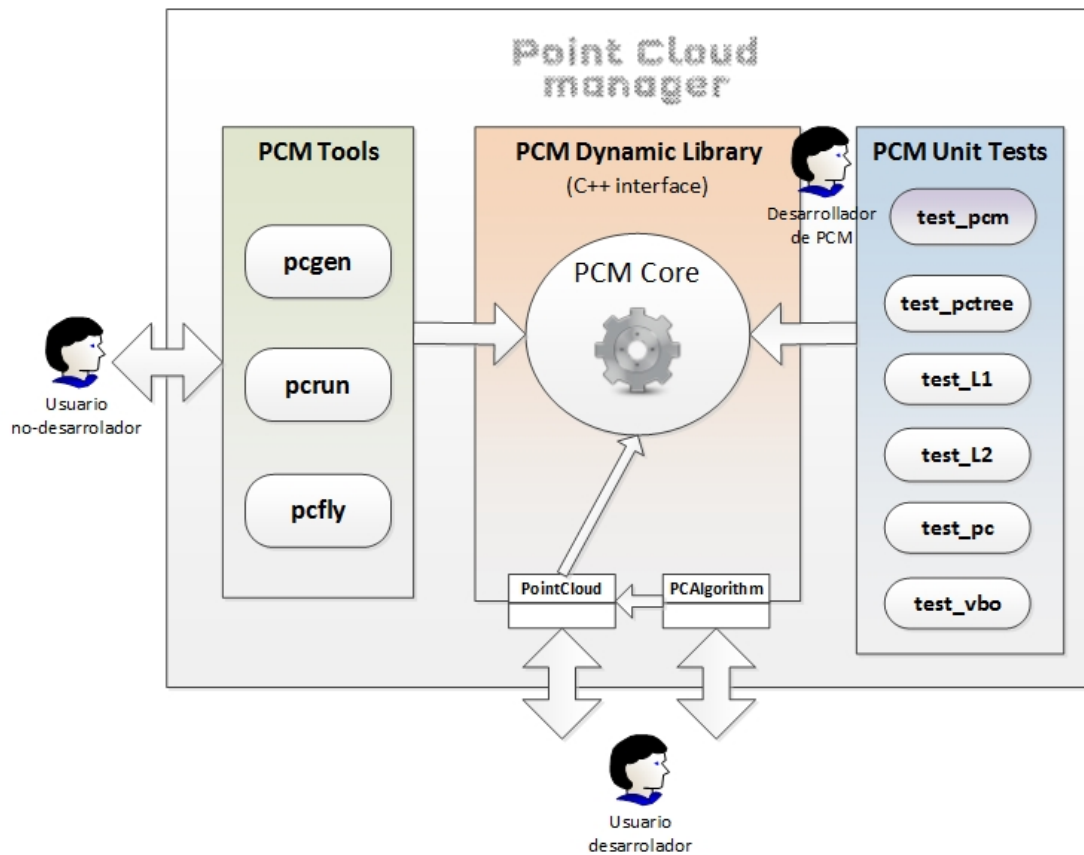


Figura 3.2: Arquitectura del paquete software PCM

aspectos.

3.2.1. Arquitectura general

El paquete software PCM está compuesta por tres componentes básicos y con la que cada tipo de usuario interactuará con ellas dependiendo de su rol, Figura 3.2). Los usuarios no desarrolladores utilizan las herramientas dadas por PCM para poder visualizar o procesar los datasets. El rol usuario de PCM desarrollador consiste en aquellos que quieran desarrollar sus propios procesos espaciales usando la librería que PCM ofrece. Por último, los desarrolladores de PCM disponen de una batería de test unitarios y un test del sistema completo para que puedan comprobar cómo afectan sus contribuciones a la librería PCM.

El núcleo principal de la arquitectura de PCM es su librería. Esta se encarga de la

siguientes tareas:

- manejo de grandes streams de datos provenientes de disco duro.
- lectura/escritura de formatos de modelos 3D y escaneos LIDAR, formato propio de nubes de puntos (PCB), descriptor del árbol binario del dataset y conjuntos (*chunks*) de datos.
- 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.

Sobre esta librería, PCM construye unas herramientas para proporcionar una interfaz al usuario para que pueda realizar todas las operaciones que quiera como conversión de formatos, creación, información y visualización del dataset, ejecución de procesos espaciales, etc. Estas herramientas son las tres siguientes:

1. PCGEN: herramienta encargada de la lectura de los diferentes formatos LIDAR posibles y de otros tipos para la generación de la estructura de datos con la que el sistema trabajará. También aporta información importante sobre cualquiera de estos elementos. Su funcionamiento se basa en tres comandos:
 - Convert: lee la nube de puntos original en formato LIDAR u otro para generar uno en formato unificado denominado BPC (Binary Point Cloud)
 - BuildTree: recibe de entrada el formato creado previamente (BPC) y, a partir de unos parámetros de configuración, genera el dataset final formado por un descriptor de árbol binario y una colección de ficheros binarios.
 - Info: muestra información importante sobre cualquiera de los ficheros anteriores.
2. PCFLY: herramienta para la visualización 3D de un dataset en tiempo real utilizando la API gráfica OpenGL.

3. PCRUN: herramienta para la ejecución de procesos sobre un dataset.

En este proyecto se usará PCM como usuario desarrollador por lo que haremos uso de la librería PCM. A pesar de esto también se hará uso de una herramienta, *pcgen*, que será necesaria para la preparación del dataset que gestiona la librería PCM.

A continuación se explicará algunos aspectos fundamentales sobre el funcionamiento interno de la librería PCM.

3.2.2. Estructura de datos

Como se explicó previamente, para poder gestionar datos espaciales es necesario algún tipo de estructura de datos para gestionarlos. PCM utiliza el denominado árbol K-dimensional o *kd-tree*. Cada nodo de este árbol hace referencia a una célula de información, llamada *chunk*, que representa una pequeña parte del espacio.

Un árbol K-dimensional o *kd-tree* es una estructura de particionamiento espacial que organiza los puntos en un espacio compuesto por k dimensiones. Para su formación cada nodo divide el espacio en dos mitades, dejando un grupo de puntos a cada lado. Se repite esto progresivamente, como se muestra en el ejemplo de la Figura 3.3, hasta que la condición de parada se cumple. En el caso de PCM esta parada será cuando el número de puntos del subespacio sea menor a un umbral preestablecido denominado M . Este parámetro tiene una gran influencia en el sistema ya que es el que establece el tamaño máximo de cada *chunk*.

Además PCM añade una pequeña variación a *kd-tree* para conseguir que el árbol generado sea binario y perfecto (todas las hojas están al mismo nivel). De esta forma consigue que cada nodo se pueda identificar secuencialmente desde la raíz, que será el nodo 0, hasta la última hoja.

Con respecto a los niveles de detalle, PCM sigue la base establecida por Enrico Gobbetti y Fabio Marton en su artículo “Layered Point Clouds” [17], la cual establece que con un buen método estadístico que seleccione bien los puntos se puede hacer que los niveles sean aditivos. Esto quiere decir que en el primer nodo almacenará un subconjunto de puntos bien distribuido a través de todo el espacio total del modelo, formando el primer nivel de detalle. Los siguientes nodos irán añadiendo mayor detalle.

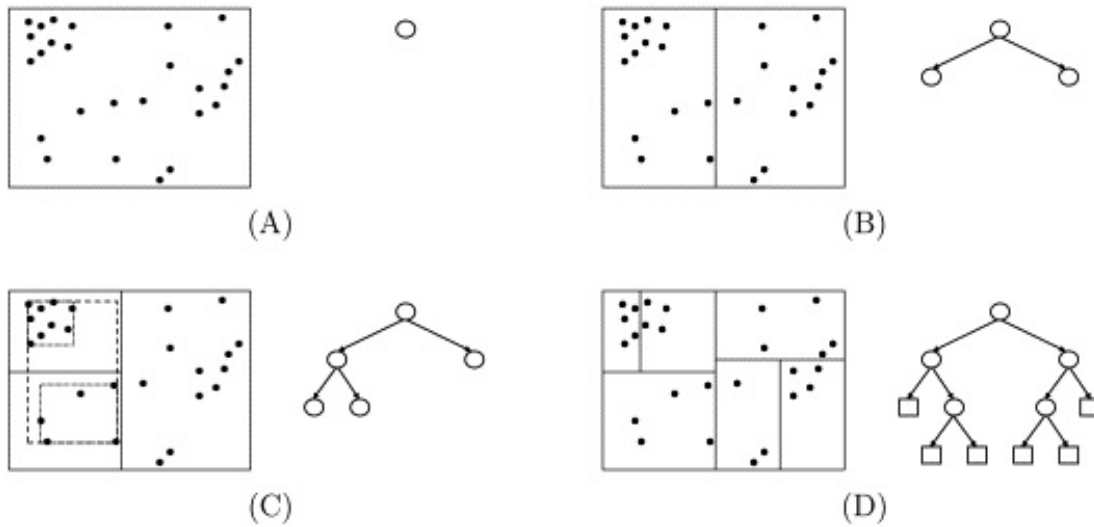


Figura 3.3: Construcción de un kd-tree

Esto se puede apreciar perfectamente en la Figura 3.4.

En conclusión, al contrario que en muchas implementaciones de kdtrees, no se almacena toda la información solo en los nodos hoja, con lo que el resto de nodos del árbol únicamente sirven para mantener información espacial. En PCM se almacena un subconjunto de datos, denominado data chunk, en cada uno de los nodos. De esta forma si se suman los datos de todos los nodos se consigue la nube de puntos completa.

3.2.3. Jerarquía memoria

PCM gestiona la jerarquía de memoria mediante un sistema de cachés software a dos niveles que se integrarán con los tres niveles de memoria que utiliza. Estos son:

- PHDD: memoria de estado permanente con alta capacidad pero baja velocidad. Aquí se guarda la base de datos completa de chunks y el descriptor de la estructura espacial multirresolución, que será cargada en memoria principal una vez arranque el servidor.
- PMemoria principal (RAM): memoria volátil de baja capacidad y alta velocidad.
- PMemoria de vídeo (VRAM): memoria volátil de menor capacidad pero mayor velocidad con respecto a la RAM con la que solo trabaja la GPU.

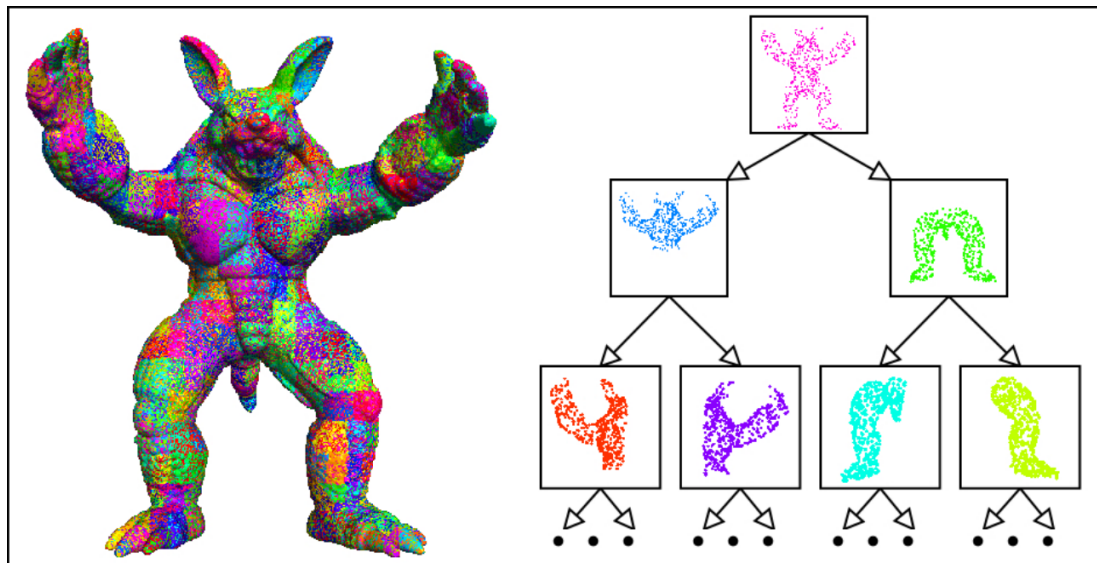


Figura 3.4: Distribución de los datos para generar niveles de detalle aditivos.

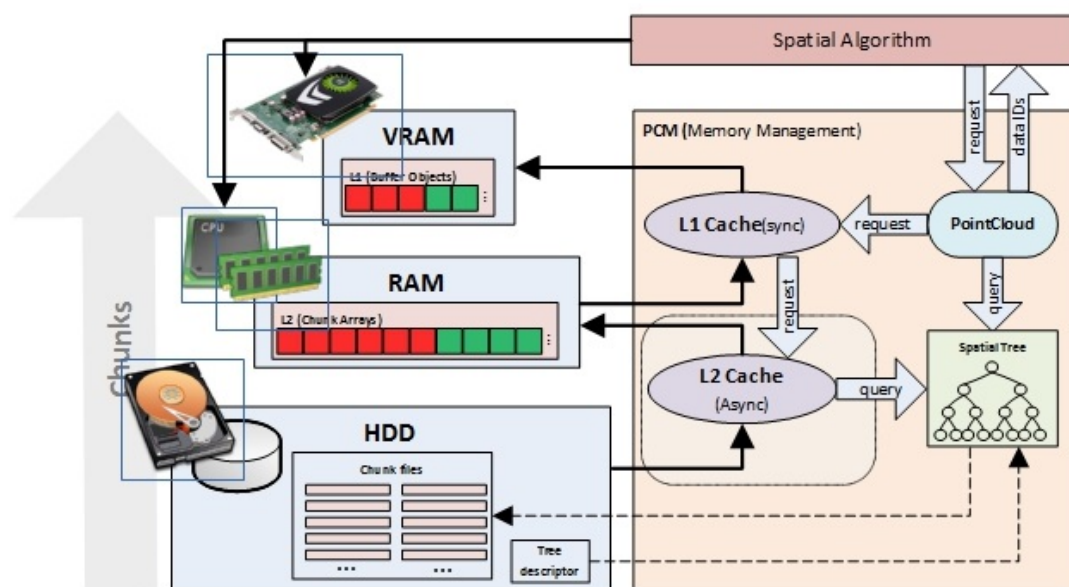


Figura 3.5: Sistema de gestión de la jerarquía de memoria.

En la imagen de la Figura 3.5 podemos ver cómo las dos cachés que implementa PCM, parte derecha, interactúan con la jerarquía de memoria, en la parte izquierda. La dos cachés implementadas por PCM son:

- L1 Cache. Caché de tipo síncrono, es la encargada de gestionar la carga de datos entre la memoria del sistema (RAM) y la memoria de vídeo (VRAM). Su otro objetivo, aparte de la gestión, es también convertir los datos de los chunks en datos válidos para la VRAM.
- L2 Cache. Encargada de pasar los chunks desde disco a la memoria del sistema RAM. Su política de reemplazo es del tipo LRU (Menos Usada Recientemente o “Least Recently Used”). Sus principales responsabilidades son:
 - Actualizar la lista de peticiones de carga efectuadas por la caché inmediatamente superior.
 - Mantener los chunks recientemente cargados y que serán utilizados nuevamente con una alta probabilidad.
 - Consultar a la estructura multirresolución qué ficheros corresponden a cada nodo de la lista de peticiones.
 - Mantener la coherencia en la memoria del sistema relacionando mediante los IDs únicos de los nodos su chunk correspondiente en memoria.

3.2.4. Cachés software

Una caché es cualquier tipo de complemento que consiga el almacenamiento de datos para que las próximas peticiones sobre estos puedan ser atendidos rápidamente. Podemos definir dos tipos de cachés. Las de tipo hardware se realizan a través de algún componente hardware en particular y están centradas únicamente en la posición de los datos en los bancos de memoria física sin tener en cuenta ningún tipo de relación que pueda haber entre los datos en sí. Para esto son necesarias las cachés software. Cuando se conoce la relación de los datos y cual será la posible forma de acceso a estos se puede justificar el diseño de una caché software que se situará entre los niveles de la jerarquía de memoria para poder mejorar el rendimiento del sistema.

3.2.5. Sistema de cámaras

El sistema de cámaras que maneja OpenGL y WebGL se basa en dos matrices que tiene asociadas cada cámara: “view matrix” y “projection matrix”.

“View matrix” es la matriz encargada de guardar los valores necesarios para saber la situación de la cámara. Esta compuesta por tres valores: “eye”, punto donde está situada la cámara, “up”, vector que determina la dirección “hacia arriba” de la escena, y “look-at”, punto al que mira la cámara.

Por otro lado, “projection matrix” es la matriz a partir de la cual se puede generar el “frustum” de la escena. Se denomina “frustum” a la región del espacio que es mostrada en la escena. Para calcularlo son necesarios cuatro valores: “fovy”, ángulo de visión de la cámara, “aspect”, proporción entre ancho y alto, “near”, indica a qué distancia desde la cámara es empieza a ser visible la escena, y “far”, indica a qué distancia deja de ser visible la escena.

3.3. Paradigma Cliente-Servidor

El modelo cliente-servidor es una estructura de aplicación distribuida basada en la repartición de tareas entre un proveedor de recursos o servicios, llamado servidor, y los solicitantes de estos recursos o servicios, llamados clientes. En general estos estarán en hardware totalmente separados y se comunicarán a través de algún tipo de red, aunque también pueden estar situados en el mismo hardware.

La repartición de tareas entre servidor y cliente se puede realizar de distintas maneras. Dependiendo de nuestro interés podemos recaer mayor carga en el servidor o en el cliente. El primer caso tiene la ventaja de que el servidor es generalmente una máquina de prestaciones superiores a los del cliente por lo que ejecutará las tareas mucho más rápido. No obstante se requiere un gran envío de datos entre cliente y servidor la red puede retrasar tanto que se ejecutará más rápido directamente sobre el cliente. En el segundo, por el contrario, las tareas se realizarán mayoritariamente en el cliente lo que significa mayor lentitud pero sin la necesidad de utilizar de forma intensiva una red que puede llegar a ser muy lenta. Lo ideal sería buscar un punto intermedio según el tipo

de servicio, prestando especialmente interés en la calidad de red que puedan disponer los clientes.

Para poder comunicarse ambos, servidor y cliente, se hacen necesarios múltiples protocolos. Para establecer algún tipo de organización en 1980 la Organización Internacional para la Estandarización (ISO) estableció el modelo OSI (Modelo de Interconexión de sistemas abiertos o Open System Interconnection). Este describe varias capas en la que cada uno tendrá un protocolo formando una especie de “pila” de protocolos. Estas capas son física, enlace de datos (Mac), de red (IP), de transporte (TCP/UDP), de sesión, de presentación, y de aplicación.

Para las comunicaciones a través de Internet actualmente las capas física (Ethernet), de datos (Mac) y de red (IP) tienen un protocolo que es usado en prácticamente la totalidad de las comunicaciones. En el caso del transporte, en este proyecto se usará TCP, que garantiza que los datos serán entregados sin errores en el mismo orden que fueron enviados.

El protocolo que se especificará más adelante dentro de este proyecto se enmarca dentro de la capa de aplicación e irá sobre otro protocolo llamado WebSocket. Este protocolo es complejo de definir dentro del modelo OSI pero podríamos considerarlo dentro de la capa de sesión aunque a su vez usa HTTP (capa de aplicación) para establecer la conexión y TLS (capa de presentación) para seguridad, aunque esto último es optativo.

3.3.1. WebSocket

Tradicionalmente las aplicaciones web de Internet se han basado en el paradigma de solicitud/respuesta de HTTP. En un inicio todo era estático y no se podía enviar nada hasta que el usuario realizaba algún tipo de evento. La primera aproximación a un mayor dinamismo se realizó con AJAX, pero en esta tecnología se requería igualmente interacción por parte del cliente preguntando al servidor de forma regular si los datos estaban cargados como se ve en la Figura 3.6.

Desde hace unos años existen ya existen opciones para enviar datos al cliente una vez estén disponibles. Estas se conocen como “Push” o “Comet”. Su funcionamiento

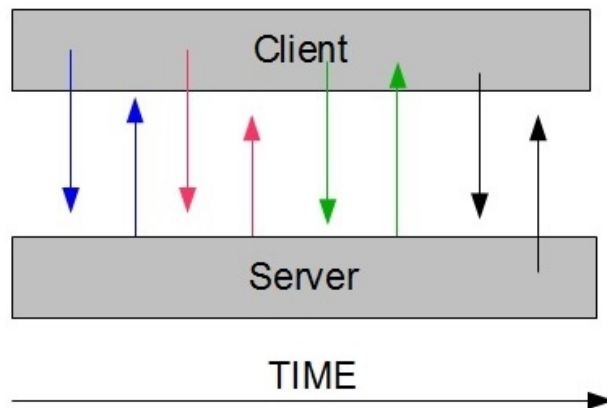


Figura 3.6: Flujo de comunicación cliente-servidor con AJAX.

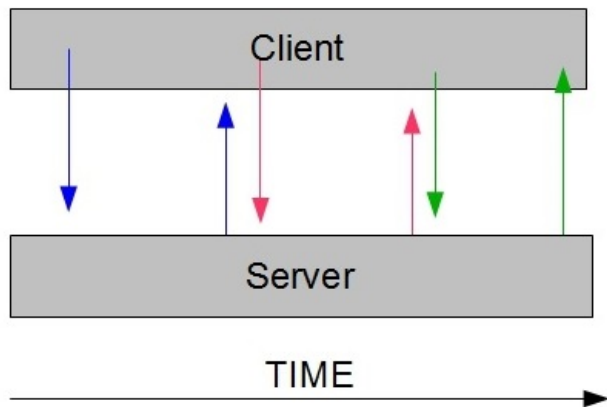


Figura 3.7: Flujo de comunicación cliente-servidor con Long Polling.

normalmente se basa en lo que se conoce como “Long Polling”. Long Polling (sondeo largo) consiste en un pequeño truco en el cual se simula un sistema *push*, la petición de una transacción la origina el servidor, cuando este no es posible. Para hacer esto posible el cliente inicia una petición de información de manera normal pero si el servidor no tiene los datos preparados, en lugar de enviar una respuesta vacía, no contestará hasta que tenga la información disponible. Acto seguido, el cliente puede volver a realizar otra solicitud para que el servidor tenga siempre una petición en espera y así poder responder a un evento como se ve en la Figura 3.7.

Pero esta solución tiene un gran problema: producen un exceso de peticiones y

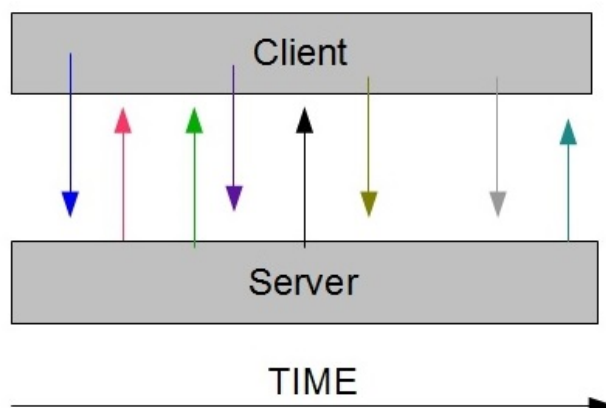


Figura 3.8: Flujo de comunicación cliente-servidor con WebSocket.

tráfico HTTP, lo que lo hace inviable para las aplicaciones de baja latencia como la de este proyecto. Por este motivo nace la necesidad de utilizar algún tipo de alternativa tecnológica, en nuestro caso WebSocket.

Websocket es una tecnología web que proporciona una comunicación full-duplex sobre una única conexión TCP sin sobrecargar el protocolo HTTP. El protocolo WebSocket está estandarizado por IETF [18] y la API WebSocket por el W3C [7]. Con WebSocket, una vez establecida la conexión, cualquiera puede enviar un mensaje sin necesidad de que la otra parte tenga que ser encuestada, Figura 3.8.

Según un análisis de Kaazing Corporation [19], puede llegar a reducir el tráfico HTTP entre 500 y 1000 veces y reducir la latencia hasta un tercio con respecto a las tecnologías tradicionales basadas en HTTP. Esto lo convierte en la opción perfecta para aplicaciones que requieran actualizaciones en tiempo real.

Además la comunicación se realiza sobre el puerto 80 del protocolo TCP, lo cual es un gran beneficio para entornos en los cuales se bloquean las conexiones no habituales de Internet mediante el uso de firewalls.

Su funcionamiento es sencillo. Primero el cliente realiza una negociación a través de una petición “upgrade” del protocolo HTTP de forma que el servidor interpreta esta petición como HTTP para después cambiar a modo WebSocket si este lo admite. Una vez establecida la conexión se pueden empezar a enviar los datos en la dirección que se

quiera, tanto de cliente a servidor como de servidor a cliente.

Un detalle importante a tener en cuenta es que, aunque WebSocket está diseñado para ser implementado en navegadores web y servidores web, este puede ser usado por cualquier tipo de aplicación cliente-servidor.

3.3.2. WebGL

En 1992, Silicon Graphics Inc. (SGI) inicia el desarrollo de OpenGL (Open Graphics Library), una especificación estándar que define una API multilenguaje y multiplataforma para aplicaciones de gráficos 2D y 3D.

A partir de OpenGL, en 2006 nace el primer prototipo de WebGL a manos de Vladimir Vukicevic en Mozilla. Al principio Mozilla y Opera empiezan creando sus propias implementaciones independientes pero en 2009 se crea el WebGL Working Group con el objetivo de crear una implementación unificada. WebGL es una API JavaScript para renderizado interactivo de gráficos 3D y 2D en un navegador web compatible sin la necesidad de ningún tipo de *plug-in* o complemento adicional.

El diseño de WebGL está basado en OpenGL ES 2.0, una variante de OpenGL pensada para dispositivos empotrados como teléfonos móviles, videoconsolas, etc. Se puede introducir en un HTML perfectamente junto a otros elementos a través del elemento Canvas que nos proporciona HTML5, al que se puede acceder usando Document Object Model (MOD).

A día de hoy, la gran mayoría de navegadores, como Mozilla Firefox, Google Chrome y Opera, ya han implementado WebGL y son perfectamente compatibles e incluso algunos navegadores móviles también lo implementa. Esto ofrece una compatibilidad muy alta con un gran número de dispositivos sin la necesidad de tener que programar específicamente para cada uno, sin importar el S.O. o tipo de dispositivo que sea.

Para más información técnica sobre WebGL se recomienda la lectura de “WebGL Programming Guide” [20] y para gráficos interactivos basados en OpenGL “Interactive Computer Graphics” [21].

Capítulo 4

Estructura del sistema

En este capítulo se mostrará desde una perspectiva general la estructura de todas las partes del proyecto para después, en siguientes capítulos, explicar más detalladamente cada componente individualmente.

4.1. Análisis

En un escenario inicial tenemos la biblioteca PCM que está pensada para la gestión de grandes datasets de puntos e incluye una herramienta para la visualización de nubes de puntos para PC de uso común. Este escenario se sitúa todo sobre una misma máquina, tanto la gestión como la visualización. El objetivo que se busca es separarlo, siendo una parte realizada por una máquina, el servidor, que se encarga de la gestión de la nube de puntos, y la otra parte por otra máquina diferente, el cliente, que será quien visualice la nube de puntos en tiempo real enviados a través de un protocolo, véase Figura 4.1.

Gracias a la forma en que fue diseñado, PCM esto es fácilmente adaptable. Para ello solo es necesario implementar un servidor que sea capaz de interactuar con la estructura multiresolución y la caché de PCM, de forma que pueda extraer la información necesaria y enviar esta al servidor. El servidor deberá funcionar de una forma parecida a como lo hace la caché L1 con respecto a como esta gestiona la carga de datos desde la memoria del sistema (RAM). La diferencia que tendrá es que en lugar de convertir los chunks a

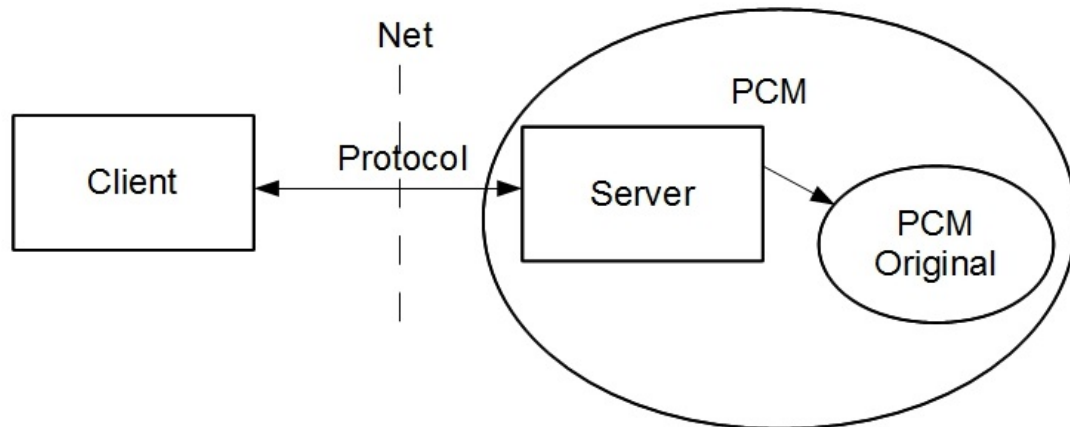


Figura 4.1: Esquema general del sistema.

una forma que entienda la VRAM su objetivo será transformarlos de forma que puedan ser enviados por una red y que puedan ser fácilmente recogidos y analizados por un cliente remoto.

Para poder hacer que el servidor pueda enviar los datos al cliente es necesario establecer algún tipo de protocolo mediante el cual ambos se puedan comunicar y enviar los datos que sean necesarios. Este protocolo, creado exclusivamente para la comunicación de las nubes de puntos, se sitúa en la capa de aplicación y es totalmente independiente de los protocolos de transporte sobre los que se envía. Es decir, si en un futuro se quisiera cambiar la forma en que se envían los datos el protocolo que se establece en este proyecto seguiría siendo totalmente válido. Esto además también ofrece la posibilidad de ir enriqueciendo el protocolo con nuevos comandos sin influir en las capas inferiores de transporte.

La parte cliente se encargará de recibir los datos desde el servidor y mostrar en 3D esta información de forma similar a como lo hacía PCM en local. Desde el punto de vista del protocolo, y del servidor, el tipo de cliente que recibe los datos no es relevante.

En este proyecto se ha utilizado un cliente web basado en WebGL pero se podría hacer perfectamente con otro tipo que se quisiera como, por ejemplo, una aplicación nativa en OpenGL similar a la de PCM pero en lugar de obtener los datos de la caché L1, los obtendría de un servidor a través de una red.

4.2. Reparto de tareas

Un factor importante, a considerar al introducir esta arquitectura cliente-servidor en PCM, es cómo se reparte cada parte las tareas que debe realizar y qué información deben compartir el cliente y el servidor. La información que debe manejar el servidor a través de PCM es el dataset total de chunks y la estructura de datos de esta.

El envío del dataset completo se descarta completamente ya que es inviable a causa del gran tamaño que estos ocupan, además que uno de los objetivos es evitar que el cliente tenga que tener toda la información de la nube de puntos. El gran dilema que surge es quien debe tener y controlar la estructura de datos encargada de facilitar los accesos al dataset. Dos opciones son posibles:

- Compartir estructura de datos: se tendría que enviar esta estructura justo tras establecer conexión. Con esta opción el cliente sería capaz de saber qué chunks son los indicados para mostrar y tendría que solicitar estos directamente al servidor. El servidor recibiría esta petición, recuperaría el chunk solicitado (de HDD o de caché L2 según como fuera el caso) y sería enviado al cliente.
- No compartir la estructura de datos: en este caso el cliente no conoce absolutamente nada del dataset, solamente su nombre para poder solicitarlo al cliente. Por este motivo el cliente tiene que informar sobre la posición de su cámara para que el cliente pueda saber que chunks son los que realmente necesita y así poder enviarlos.

En este proyecto se opta por la segunda opción. Con esta opción conseguimos separar el cliente de forma que tenga que realizar la mínima tarea posible. Es cierto que con esta opción se pueden generar mayor número de peticiones por parte del servidor, hay que tener en cuenta que el cliente tendrá que informar cada cierto tiempo sobre su cámara ya que si esta ha cambiado puede ser necesario enviar nuevos chunks, pero estas son de pequeño tamaño y no implican una gran pérdida de rendimiento. Por el contrario, la otra opción requiere enviar la estructura de datos que puede llegar a ocupar bastante y puede retrasar el inicio de la comunicación, además de exigir más procesamiento al cliente, que tendría que gestionar la estructura multiresolución.

Nuestra aproximación consigue así desplazar el grueso de la carga computacional al servidor, que es el único que conoce y mantiene la nube de puntos y la estructura multiresolución, encapsulando en el toda la lógica de control sobre la gestión del mismo. De esta forma conseguimos un cliente ligero, totalmente independiente de toda la estructura de aceleración y que puede implementarse en plataformas poco “exigentes” desde un punto de vista hardware, como dispositivos móviles o PCs con pocos recursos.

4.3. Funcionamiento de la comunicación y flujo de mensajes

La comunicación se inicia siempre desde la parte cliente. Para poder hacer esto el cliente accede a una web en la que se muestra una lista con todos los modelos que tiene el servidor. Esta lista no es estática sino que se hará una consulta al servidor a través de WebSocket preguntando cuáles son los modelos de los que dispone el servidor. El protocolo provee por tanto, como veremos, de comandos para realizar esta consulta. Una vez recibidos se crea una lista simple en HTML mostrándolos de forma que el cliente solo necesita hacer “click” en uno de los elementos de la lista para que ese modelo se inicie, Figura 4.2.

Tras la selección del modelo por parte del cliente se procede al inicio de la transmisión de este. Antes de esto es necesario que se envíe un mensaje para iniciar la sesión desde el cliente al servidor indicando el nombre del modelo, el tamaño caché del que dispone el cliente para almacenar toda la información que se reciba y qué versión del protocolo maneja el cliente. Con este mensaje el servidor tiene que ser capaz de iniciar sesión creando toda la información que necesita (L2 Caché, estructura de datos, etc.) y si ha sido todo creado correctamente este envía al cliente un mensaje para comunicarle que se ha iniciado correctamente el modelo y en el que se le indica qué información contiene cada punto que va a ser enviado. Si existe algún problema durante este inicio de sesión será comunicado por la parte servidor indicando qué problema hubo mediante un código de error.

Una vez el inicio haya sido correcto se procede al intercambio de datos entre el

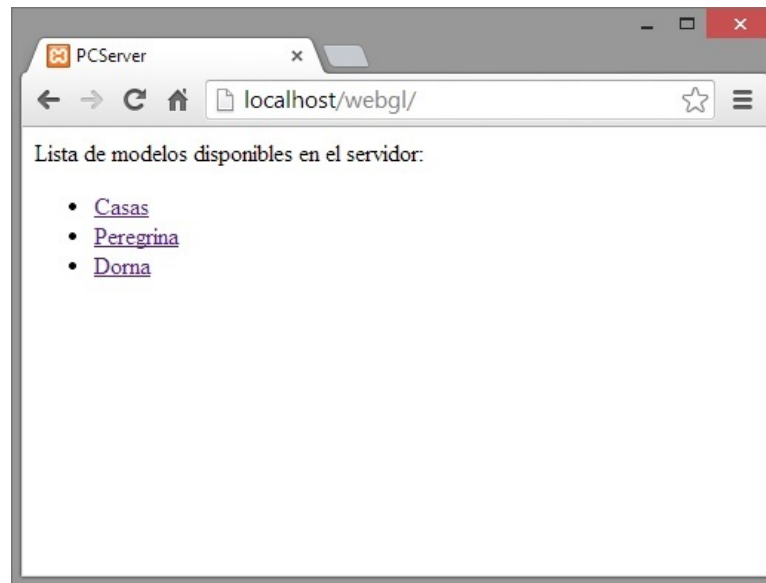


Figura 4.2: Captura del inicio de la aplicación web con la lista de los datasets.

cliente y el servidor. El cliente se encarga de solicitar la información a través de un mensaje en el que se indica la cámara actual que tiene, incluyendo toda la información de esta. Esto se realiza con un mensaje que incorpora todos estos datos y que se realiza cada cierto tiempo, preestablecido por el cliente. Este tiempo es un factor importante para el rendimiento y precisión en la visualización, cuando el cliente mueve la cámara es necesario que el servidor sea informado para que puede enviar los nuevos datos referentes a esa nueva posición. Cómo determinar si la cámara ha sido movido “poco” o “mucho” es un proceso complejo se ha optado por establecer que cada cierto tiempo se realice una petición sin comprobar si la cámara ha sido movida. Esto puede parecer a priori un factor muy negativo para el rendimiento por el gran número de mensajes que puede llegar a enviar el cliente pero gracias a que estos mensajes de petición no son de gran tamaño y a como se ha implementado el servidor, LRU caché para no volver a enviar datos de los que ya dispone el cliente y anulación de la petición anterior ante la llegada de una nueva, no repercute en el rendimiento final.

Ante cada una de estas peticiones por parte del cliente el servidor consulta a PCM qué información es la más apropiada para ser enviada con la cámara actual que le ha sido proporcionada desde el cliente. El servidor obtiene una lista con los IDs de los

chunks ordenados de forma óptima según su posición en el espacio con respecto a la cámara y a su nivel de detalle. Por tanto este orden es el adecuado con el cual el servidor procede a su envío.

Con estos IDs el servidor es capaz de recuperar sus chunks correspondientes en la caché L2 de PCM. Cada chunk es enviado en un único mensaje con todos los puntos que contenga y con toda la información correspondiente a estos. Como el cliente tiene un límite de información que puede aceptar el servidor tiene que ser capaz de saber cuándo tiene que dar la orden de eliminación para que el cliente pueda recibir nuevos datos. Esto se consigue con una caché LRU que nos permite saber cuando es necesario eliminar algo y qué chunks son los apropiados para su eliminación. La gestión de esta LRU del cliente es llevada a cabo por parte del servidor, que en todo momento sabe los chunks que tiene el cliente en su poder. Con esto, además también se aporta una característica importante para el rendimiento. Como el servidor sabe qué chunks son exactamente los que tiene el cliente el servidor puede ahorrar el envío de estos con lo que consigue eliminar una gran carga en la red.

En la Figura 4.3 podemos ver este proceso. Cada vez que se recibe una petición de solicitud de chunks desde el cliente el servidor obtiene de PCM una lista con cuáles son los que debe que enviar. A continuación, por cada elemento, comprueba si ya está en la caché LRU. Si existe en ella es que el cliente ya tiene ese elemento, por lo que no es necesario enviarlo. En caso contrario procede a realizar el envío del chunk correspondiente, no sin antes comprobar si el cliente tiene espacio suficiente. Si esto es así, se le envían los IDs de los chunks que deben ser borrados para que pueda hacer espacio antes de que reciba la información del chunk actual. Este proceso se repite hasta que sean enviados todos los chunks de la lista devuelta por PCM.

En el diagrama de secuencia de la Figura 4.4 podemos observar el flujo de mensajes entre el cliente y el servidor en una sesión normal. Primero el cliente inicia sesión. A continuación realiza una primera solicitud con su cámara actual. Tras esto, el servidor empieza a contestar enviando los chunks necesarios según la cámara recibida. Antes de que el servidor termine de contestar, el cliente realiza una nueva petición. El cliente anula la petición antigua y empieza a contestar la última que ha recibido desde el cliente.

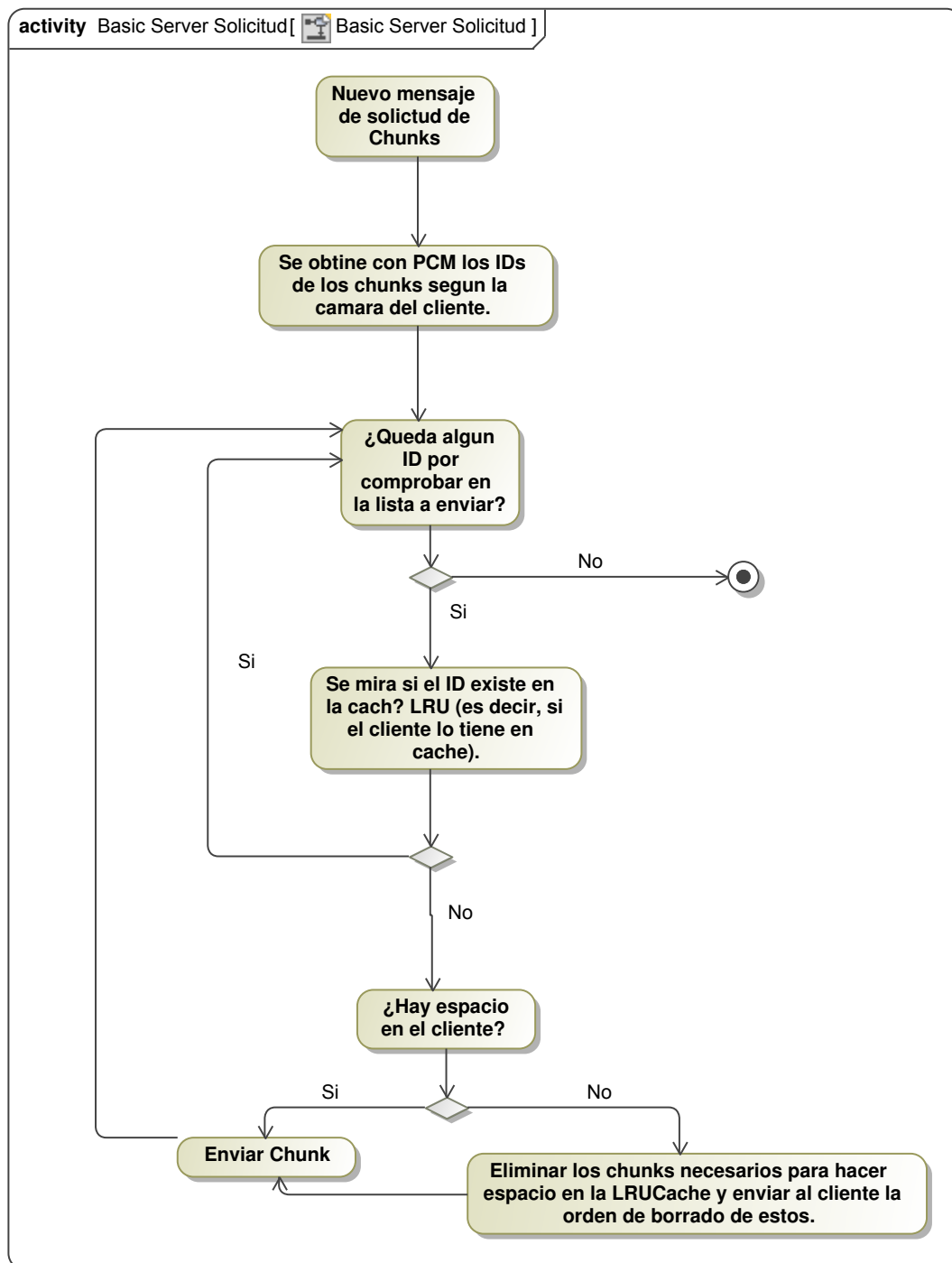


Figura 4.3: Proceso a seguir en el servidor por cada petición de chunks del cliente.

Por último, el cliente decide terminar la sesión aunque también podría ser terminada por parte del servidor.

Con todos estos mensajes el cliente consigue recibir exactamente los chunks que necesita para así poder mostrarlos. Para evitar que exista una carga innecesaria en la red se renuncia al uso de mensajes de confirmación o similares por parte del cliente, por lo que el servidor asume que todos los mensajes que son enviados son aceptados y se actúa en consecuencia a ellos por parte del cliente. Esto es un factor muy importante para conseguir que el cliente sea lo más ligero posible ya que no tendrá que calcular que tiene que hacer sino que es ordenado desde el servidor.

Una vez se quiere finalizar la sesión, cualquiera de los dos puede realizar esta acción indicando el porqué de este cierre con un código. Aunque se puede acabar la conexión cerrando directamente el canal de WebSocket sin enviar el mensaje establecido en el protocolo es recomendable realizarlo para que la parte, cliente o servidor, que no decide el cierre sepa el porqué de este.

Como se puede observar la comunicación es muy simple y el principal objetivo del servidor es de hacer de intermediario entre PCM y el cliente para ofrecerle los datos que realmente se necesitan. En el capítulo correspondiente al servidor se profundiza más en el proceso de recuperar los chunks y con qué clases de PCM se interactúa.

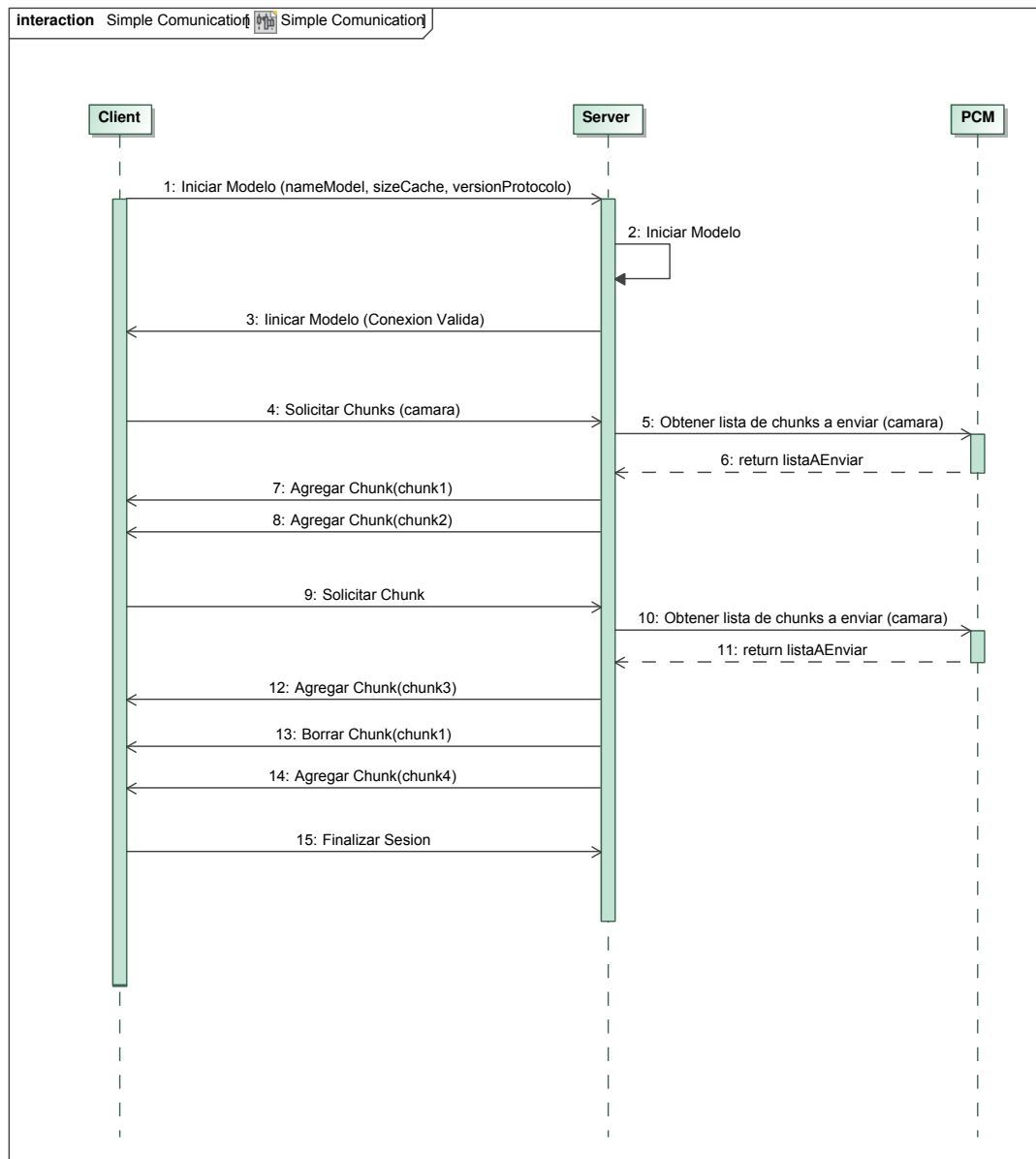


Figura 4.4: Diagrama de secuencia de una comunicación común entre el cliente y el servidor.

Capítulo 5

Protocolo

En este capítulo se mostrará desde una perspectiva general la estructura de todas las partes del proyecto para después, en siguientes capítulos, explicar más detalladamente cada componente individualmente.

5.1. Estructura

En este capítulo se presenta el protocolo de comunicación necesario para poder intercambiar datos entre la parte servidor y la parte cliente. Este protocolo ha sido diseñado para que sea fácilmente analizable y revisable en posible futuras versiones.

5.2. Tipos de mensaje

Los mensajes de este protocolo se pueden clasificar en cuatro tipos de mensajes. El primero es un tipo de mensaje especial que no necesita ningún inicio de sesión y puede ser recibido en cualquier momento y que se utiliza para encuestar que modelos tiene este servidor. El resto de mensajes forma parte del proceso de comunicación, formado por: inicio de sesión, intercambio de datos y finalización de la sesión.

5.2.1. Consulta de modelos

La consulta de modelos es realizada por el cliente y simplemente está formada por una cabecera con el valor “INF”.

Cabecera (INF)

Cuando el servidor recibe este mensaje debe consultar qué modelos están en su directorio y enviar el nombre de estos con los metadatos que tenga sobre estas. El mensaje que envía el servidor como contestación debe ser:

Cabecera (INF)	Nombre modelo	Metadatos del modelo
----------------	---------------	----------------------

La cabecera deberá tener el valor ‘INF’ para indicar que el mensaje es una respuesta a una consulta sobre los modelos disponibles. Los siguientes campos son:

- Nombre modelo: indica el nombre asociado al modelo.
- Metadatos del modelo: toda la información de interés que pueda haber sobre el modelo. Esta información es totalmente variable y cada modelo puede tener unos tipos de datos totalmente diferentes. Por eso la forma de enviar estos datos será la siguiente. Cada dato sobre el modelo se escribe con el formato “dato:valor” dividiendo los distintos datos con un separador “,”. De esta forma será fácilmente analizable por parte del cliente.

Estos dos valores se repetirán tantas veces como modelos haya en el servidor

5.2.2. Inicio

El inicio de la comunicación se realizará con un mensaje por la parte cliente con la siguiente estructura:

Cabecera (INI)	Versión protocolo	Tamaño Caché	Nombre del modelo
----------------	-------------------	--------------	-------------------

La cabecera deberá tener el valor ‘INI’ para indicar el mensaje que es, los siguientes valores son:

- Versión Protocolo: Establece qué versión del protocolo usa el cliente. Este campo será útil para que en versiones futuras haya compatibilidad hacia atrás, de forma que versiones antiguas de clientes sigan siendo compatibles con servidores actualizados.
- Tamaño caché: Tamaño caché con el que cuenta el cliente. Será usada por el servidor para saber cuándo el cliente tiene su caché llena y es necesaria la eliminación de datos.
- Nombre del modelo: Cada servidor puede tener varias nubes de puntos por lo que es necesario indicar en el inicio de la comunicación cuál es la solicitada por el cliente.

A continuación el servidor deberá contestar con un mensaje con la siguiente estructura:

Cabecera (INI)	Aceptación	Código de estado	Centro del modelo
----------------	------------	------------------	-------------------

- Aceptación: indica si se acepta o rechaza la conexión solicitada por el cliente, “0” indicaría que la rechaza y “1” que la acepta.
- Código de estado: código para indicar cualquier anotación sobre la aceptación o rechazo. En caso de rechazo indicará el motivo por el cual y si se acepta se indicará cuales son los datos que se enviarán por cada chunk u otros motivos.
- Centro del modelo: Se indica en qué posición se encuentra el centro del modelo para que el cliente sepa cómo colocar su cámara. Serán tres valores correspondientes a su posición X, Y, Z en dicho orden y divididos por un carácter separador “,”.

Para los códigos de estado se establecen algunos de ejemplo que se podrán ir ampliando según sean necesarios. Si se rechaza:

0000	Error no definido.
0001	Nombre de modelo no encontrado.
0002	Tamaño de caché no válido.
0003	Número máximo de sesiones alcanzado.
0004	Error al iniciar el modelo.
0005	Protocolo no válido.

Si se acepta:

0000	Solo posiciones en el espacio(XYZ).
0001	Posiciones (XYZ) y color (RGB).
0002	Posiciones, color y normal.
0003	Posiciones, color, normal y radio.

5.2.3. Intercambio

Una vez se ha establecido la conexión se procederá al intercambio de información. El proceso es muy simple, el cliente solicitará el envío de chunks a través de un mensaje en el que se informa de la posición actual de la cámara del cliente. Con esta posición el servidor puede saber cuales son los chunks necesarios que deberá enviar al cliente. A partir de este mensaje empezará a transmitir esos chunks hasta que finalice o hasta que el cliente envíe otra solicitud del mismo tipo tras la cual el servidor deberá parar la anterior, volver a consultar que chunks son necesarios con la nueva posición de cámara y transmitir estos. El mensaje de inicio que deberá enviar el cliente tiene que ser de la siguiente forma:

Cabecera (GET)	viewMatrix	projectionMatrix
----------------	------------	------------------

La cabecera deberá tener el valor ‘GET’ para indicar el mensaje que es, los siguientes valores son:

- ViewMatrix: View Matrix de la cámara del cliente. Esta matriz está formada por los vectores de eye, center y up, cada uno compuesto de 3 valores. Estos valores irán divididos por un carácter separador “,” en el orden que se han comentado,

es decir, primero los tres valores de eye seguidos de los de center y por último los de up.

- **ProjectionMatrix:** Projection Matrix de la cámara del cliente. Esta matriz está formada por los valores fov, aspect, near, far. En este caso cada uno de estos valores está compuesto solo por un único número y se enviarán de la misma forma que en el caso anterior, uno seguido de otro separados por un carácter “,”.

Después cada chunk deberá ser enviado de la siguiente forma:

Cabecera (CHU)	Chunk ID	Número de puntos	Información de puntos
----------------	----------	------------------	-----------------------

La cabecera deberá tener el valor ‘CHU’ para indicar el mensaje que es, los siguientes valores son:

- **Chunk ID:** identificador del chunk que se envía.
- **Número de puntos:** número total de puntos que tiene este chunk y que se envían en este mensaje.
- **Información de puntos:** Incluye toda la información de todos los puntos del chunk que se envía. En este caso se usarán dos tipos de separadores. Uno separará cada punto que se envía, que será un carácter “|” y dentro de cada punto se usará “,” para separar cada dato del punto. Los datos que se envían sobre el dato se establecen cuando se inicia la sesión.

Como el cliente tiene una memoria limitada es necesario que el servidor pueda informa que chunks son los que tiene que borrar. Para estos mensajes se establece la siguiente estructura:

Cabecera (DEL)	Chunk ID
----------------	----------

La cabecera deberá tener el valor ‘DEL’ para indicar el mensaje que es, los siguientes valores son:

- **Chunk ID:** identificador del chunk que se tiene que borrar.

5.2.4. Finalización

Este tipo de mensaje es el único que puede ser iniciado tanto por el cliente como por el servidor. Consiste en un único mensaje para indicar el final de la sesión y que no se enviarán ya nuevos datos. No necesita confirmación por parte del receptor. Acto seguido del envío se procederá al cierre del canal de comunicación utilizado. La estructura será:

Cabecera (END)	Código
----------------	--------

La cabecera deberá tener el valor ‘DEL’ para indicar el mensaje que es, los siguientes valores son:

- Código: código usado para indicar el motivo por el cual se finaliza la sesión.

Tabla de códigos para finalizar sesión:

0000	Error desconocido.
0001	Finalizado correctamente.
0002	Error inesperado, memoria del cliente sobrepasada.

Capítulo 6

Servidor

El servidor será la parte encargada de conectar el cliente con la librería PCM. A través de los datos enviados por el cliente usará la librería PCM para poder saber qué chunks son los óptimos para la visualización en el cliente y se los enviará a este.

El servidor controla toda la información que el cliente necesita sobre la nube de puntos y le dirá a este qué es lo que tiene que hacer y qué chunks tiene que visualizar. El cliente tiene una memoria limitada, esto provoca que en la parte servidor sea necesario mantener un control sobre qué información mantiene el cliente. De esta forma, cuando se llena la memoria del cliente, el servidor sabe qué chunks son los que debe eliminar el cliente. Como el cliente no tiene ninguna información sobre la estructura multiresolución de la nube de puntos, el servidor debe también informar al cliente que datos tiene que borrar. Además esto aporta una ventaja, gracias a que el servidor sabe que tiene el cliente puede evitar enviar los chunks que el cliente ya tenga. Además, gracias a que el servidor sabe en todo momento que es lo que tiene el cliente, ahorra el envío de aquellos chunks que ya hayan sido enviados previamente. Para poder hacer esto posible, el servidor mantendrá una caché LRU por cada cliente, que se explicará después más detalladamente.

Para el envío de mensajes se usará el protocolo previamente detallado sobre el protocolo websocket, el cual nos proporcionará un canal bidireccional de comunicación entre cliente y servidor.

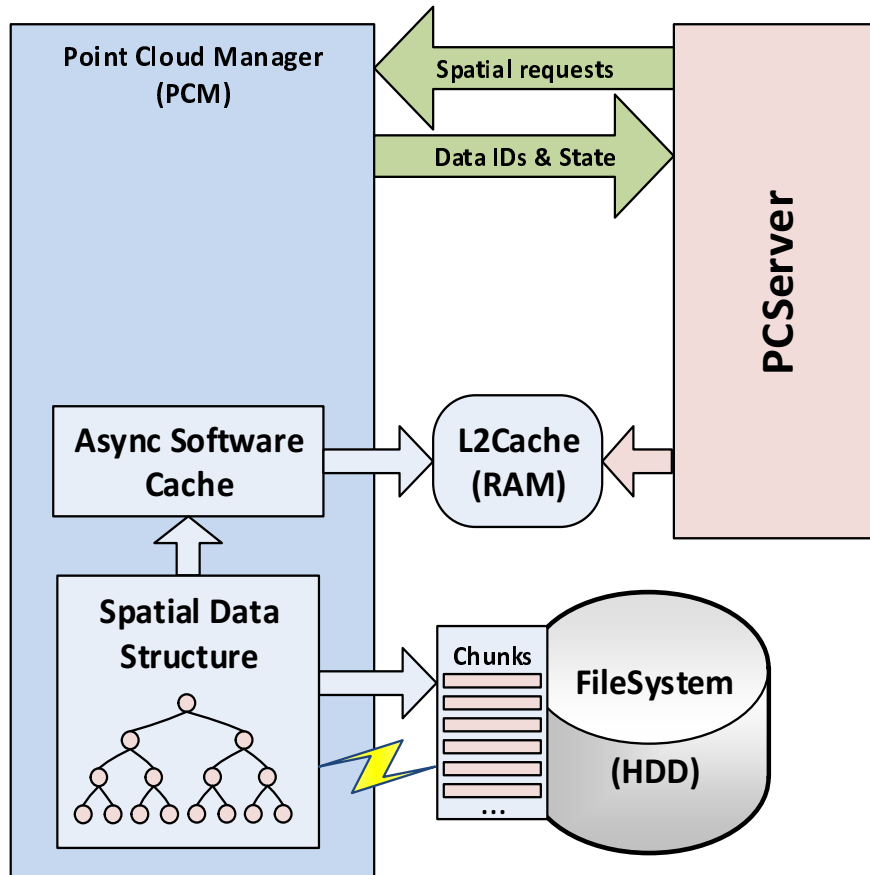


Figura 6.1: Diagrama estructural de la relación de PCServer con PCM.

6.1. Funcionamiento del servidor e integración con PCM

El Servidor se relaciona con PCM a través de su caché L2, Figura 6.1. Esta caché es la encargada de traer todos los chunks que sean necesarios y que posiblemente vayan a ser utilizados próximamente desde el disco duro hasta la memoria RAM.

Antes de esto el servidor tiene que iniciar la sesión cuando el cliente lo solicita. En este inicio es cuando se crea la caché L2 de PCM. Para crear esta caché es necesario primero la estructura de datos correspondiente (PCTree) al modelo seleccionado por el cliente. Con esta estructura el servidor ya puede crear e iniciar la caché L2 seleccionando antes su tamaño, que es elegido por servidor de forma arbitraria.

Una vez iniciada la caché L2 de PCM el servidor no puede recuperar directamente un chunk desde esta. Si un chunk no está cargado en la caché antes tiene que indicarle

que es necesario que cargarlo añadiendo su ID a la lista de prioridad de la caché L2. De esta forma, la caché prioriza intentando cargar primero los chunks indicados.

Para saber que chunks son los que se deben enviar al cliente PCM ofrece una función llamada “getLODPath” en la estructura de datos que con la cámara actual es capaz de calcular que chunks se deben mostrar primero según su cercanía y los devuelve ordenados en una lista por su nivel de detalle. Eso facilita mucho el trabajo del servidor, que solo tiene que enviar los chunks en el orden que le devuelve esta función. A veces este orden no tiene porque cumplirse, si el servidor ve que un chunk no esta cargado en RAM no se queda esperando a que la caché L2 lo cargue sino que intenta recuperar los chunks siguientes y solo envía directamente los que están cargados. Cuando acabe volverá a consultar si los chunks que no fueron enviados ya han sido cargados y repetirá el proceso hasta que todos los chunks devueltos por “getLODPath” hayan sido enviados. Con este planteamiento se consigue ampliar el rendimiento de forma que la caché L2 puede ir recuperando los chunks mientras el servidor ya va enviando aquellos que ya han sido cargados aunque estos tengan menos preferencia. La Figura 6.2 muestra un ejemplo de una comunicación en la que todos los datos estarían en caché.

Como ya se ha explicado, el cliente no tiene una memoria infinita y tendrá cierto límite para almacenar la información que es enviada desde el servidor. Para controlar esto se implementa una caché LRU en la que se guarda qué información tiene el cliente. Además, como sabemos en todo momento que almacena el cliente, nos aporta que no necesitamos reenviar un chunk que ya tiene el cliente. Con esto el servidor solo tiene que comprobar en la caché LRU, guardando una por cada cliente que inicie sesión en el servidor, si el cliente ya tiene el chunk que se va a enviar. Esto se realiza directamente después de recuperar la lista de la llamada “getLODPath” y antes de consultar nada en la caché L2. En caso de que el cliente ya lo tenga no se realiza ninguna acción más y se pasa al siguiente chunk, si por el contrario el cliente no lo tiene antes hay que comprobar si va a tener espacio para él. En caso de tenerlo se procede como ya se explicó, recuperandolo de la caché L2, sino antes de hacer esto es necesario que el servidor haga hueco en la caché LRU y enviar al cliente que chunks son los que necesita borrar. Este proceso se puede ver en la Figura 6.3.

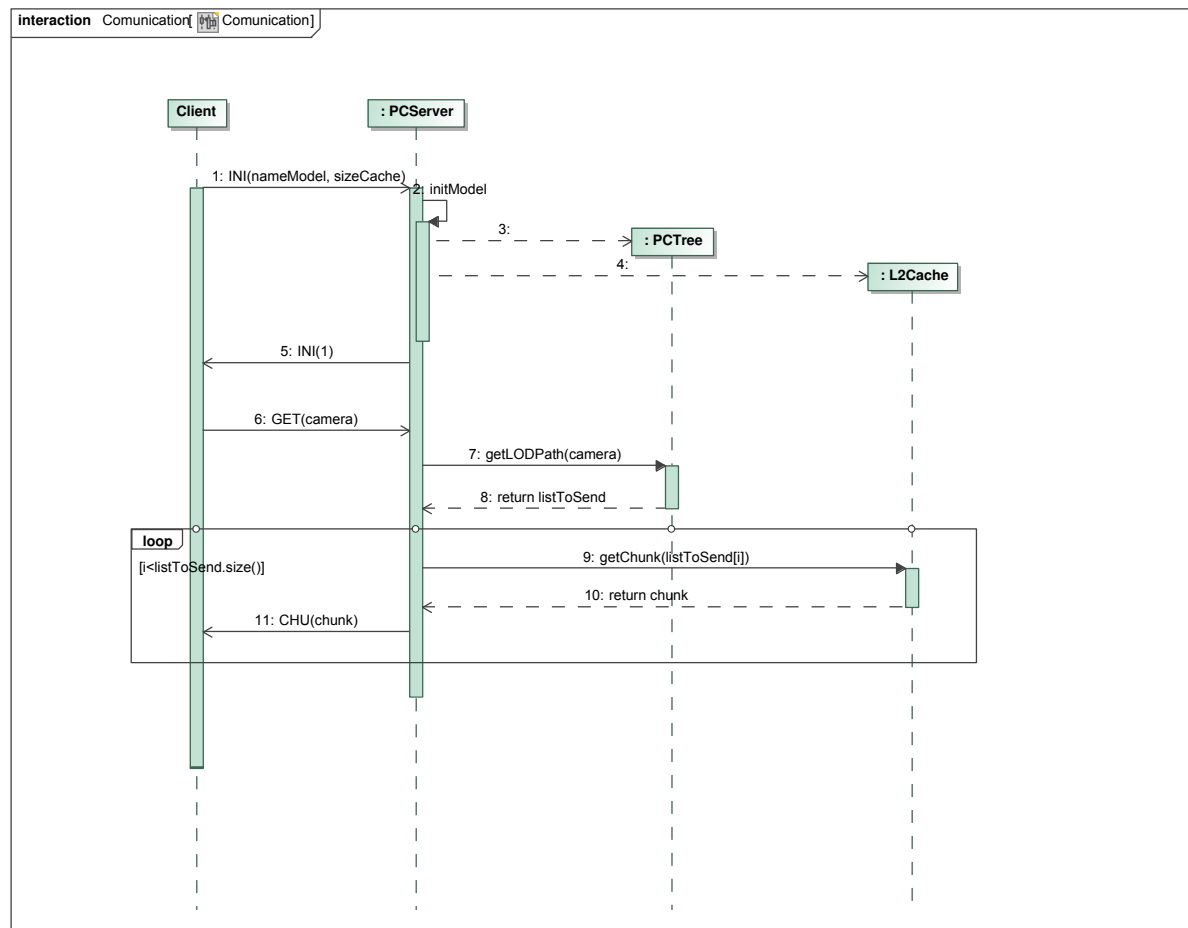


Figura 6.2: Diagrama de secuencia de una petición GET en la que todos los datos están en cache.

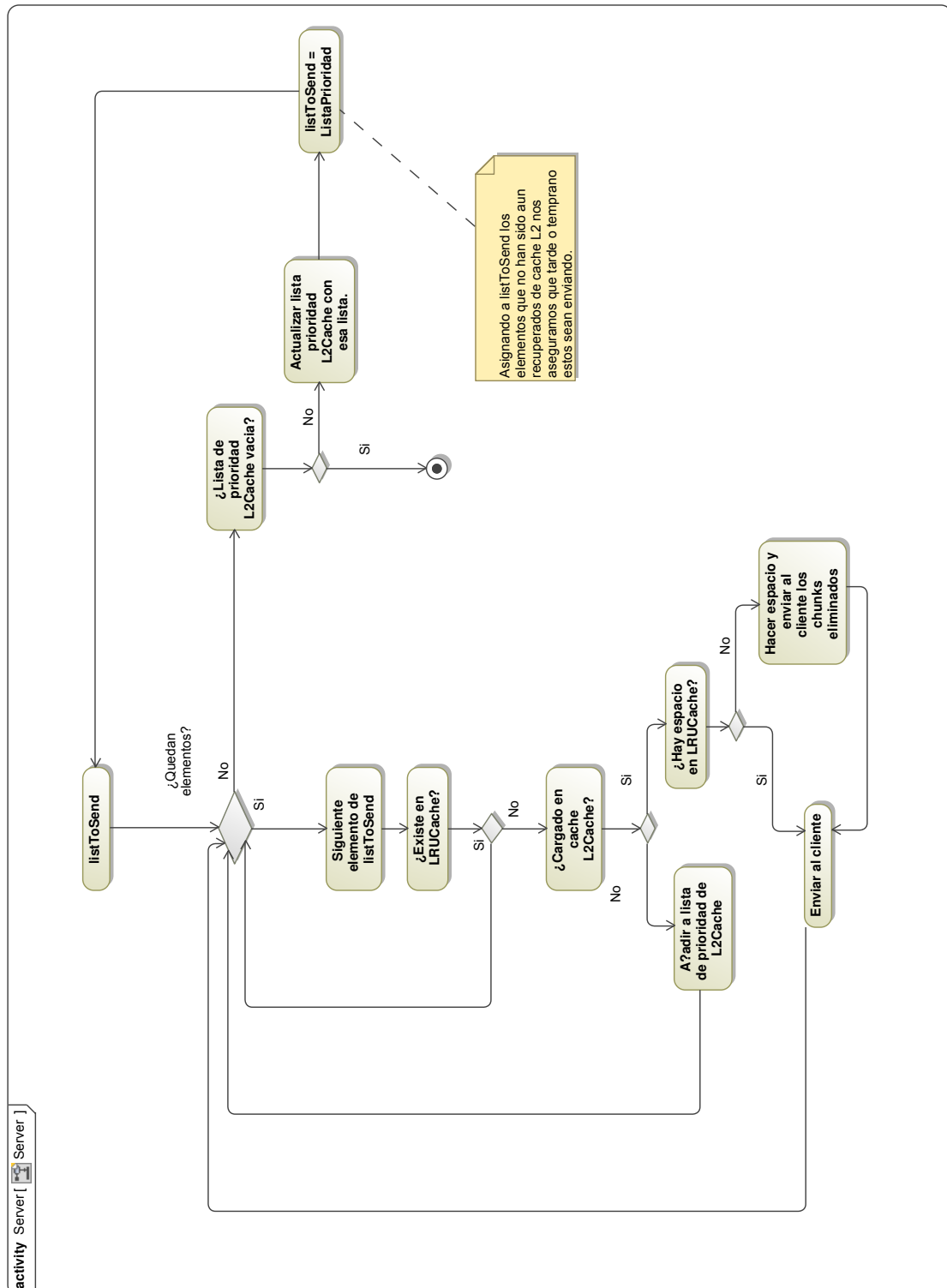


Figura 6.3: Diagrama de actividad del servidor ante una petición de chunks desde cliente.

Así se puede conseguir un gran rendimiento evitando enviar el mayor número posible de mensajes con información de los chunks desde el servidor.

Todo este proceso es el que ejecuta el servidor cuando recibe un mensaje “Get” desde el cliente. En general se va a dar el caso que, mientras el servidor aún está enviando datos, el cliente mueve su cámara y necesita hacer otra petición con la nueva posición de la cámara. Cuando esto ocurre el servidor anula completamente el resto de la petición anterior que no haya sido realizado y pasa directamente a ejecutar la nueva petición. Esto debe hacerse así porque si la cámara se mueve los chunks de la petición anterior pueden ser de una posición de cámara totalmente diferente y que no son de ninguna utilidad para el cliente.

6.2. WebSocket

Como ya se ha explicado anteriormente, Websocket está diseñado para ser implementado en navegadores web y servidores web pero puede ser usado por cualquier tipo de aplicación cliente-servidor. Este es uno de los principales motivos por los que se elige esta opción. El objetivo principal es que el servidor pueda comunicarse con un cliente de cualquier tipo, ya sea una aplicación web o tradicional. Otras opciones como sockets tradicionales u otras alternativas no son válidos para aplicaciones web o requiere la instalación de algún tipo de plugin en el navegador, lo cual quería evitarse y hacer que fuera lo más simple posible de cara al usuario.

6.2.1. WebSocket++

Websocket++ es una librería creada por Peter Thorson[X8] que implementa el protocolo WebSocket. Esta integra también funcionalidades de cliente y servidor en C++ que nos permitirá crear el servidor sin preocuparnos en la capa de transporte.

Una gran ventaja que ofrece WebSocket++ es la fácil configuración a través de archivos cabecera pudiendo así en un futuro cambiar el medio de transporte de datos, concurrencia, configuración de ficheros log, etc. sin complicaciones y convirtiéndolo en una muy buena opción para un sistema escalable.

Esta librería se utiliza a través de un objeto proporcionada por ella llamado “server”. Este objeto nos proporciona todas las funcionalidades básicas para poder enviar y recibir mensajes. Su funcionamiento se basa en unas funciones que son llamadas cuando ocurre determinado el evento correspondiente a dicha función. Estas funciones son llamadas manejadores (“handlers”). Aunque Websocket++ proporciona varios manejadores en este proyecto serán usados cuatro: “open”, cuando se abre una conexión nueva de WebSocket, “close”, cuando se cierra una conexión WebSocket, “message”, se activa cuando llega un mensaje nuevo, “interrupt”, fuerza a que los “send” realizados sean enviados.

Los manejadores “open” y “close” se usan para iniciar y liberar determinados recursos correspondientes a cada sesión WebSocket. Los dos manejadores más importantes son “message” e “interrupt”. En el primero se analiza el mensaje llegado y según el tipo de mensaje se trata según corresponda. Mientras cuando es un mensaje de finalizar, no se responde con ningún mensaje, o iniciar, solo es necesario un mensaje de contestación, no presenta ningún problema, cuando llega un mensaje de tipo “Get” aparece un pequeño problema. Por defecto, cuando llamas a la función “send” para enviar un mensaje en WebSocket++ se realiza de forma asíncrona para no bloquear el servidor. Esto quiere decir que cuando se llama a “send” no se realiza directamente el envío sino que hasta que termina la función que ha llamado a “send” no se enviará nada. Esto provoca que, tal como se ha diseñado PCServer, no se empezaría a enviar ningún mensaje hasta que todos los chunks de una petición “Get” hayan sido preparados. Para evitar esto se necesita crear un “thread” que es el que se encarga de realizar todos los envíos.

Cada vez que se quiere enviar un mensaje, en lugar de llamar directamente a “send”, se añadirá el mensaje que se quiere enviar a una cola. Un hilo complementamente separado de la ejecución principal del servidor se encarga de ir leyendo esa cola e ir enviando uno a uno los mensajes en el orden que han sido insertados. Siempre que se tiene hilos hay que tener cuidado con las variables compartidas, en este caso la cola que almacena las peticiones es una variable en conflicto. Para evitarlo siempre que uno de los dos hilos de ejecución intente modificarla debe bloquearla para que el otro espere.

A continuación se muestra pseudocódigo de la implementación de este thread. “accion” es una estructura que sirve para almacenar los mensajes junto al ID de la conexión a la que hay que enviarlo. “send” es la función que debe ser llamada por el server para enviar cualquier mensaje. “procesar_mensajes” es la función que ejecuta el thread continuamente.

```
1 struct accion{
2     accion(identificadorConexion id, string mensaje) : id(id), msg(mensaje) {}
3     identificadorConexion id;
4     string msg;
5 }
```

```
1 void send(identificadorConexion id, string mensaje){
2     mutex.lock (accion_lock);
3     lista_acciones.push(accion(id, mensaje);
4     mutex.unlock (accion_lock);
5     accion_condicion.notificar();
6 }
```

```
1 void procesar_mensajes(){
2     mutex.lock (accion_lock);
3     while (1){
4         while(lista_acciones.empty()){
5             action_condition.esperar(mutex);
6         }
7         accion ac = lista_acciones.front();
8         lista_acciones.pop();
9         mutex.unlock (accion_lock);
10
11     server.send(ac.id, ac.mensaje);
12 }
13 }
```

Ahora ya no es necesario tener que esperar que toda la función encargada de preparar los chunks termine para que el cliente pueda ir recibiendo los que ya están listos. Pero aún queda una cosa más para que funcione como se ha definido. Si se el

bucle encargado de preparar los chunks lo ejecuta el propio servidor, hasta que este termine no podrá atender otra petición lo que provoca que no se puede cancelar una petición “Get” vieja ante la llegada de una nueva y ni siquiera se podría atender a otro cliente diferente. Para solucionar esto se presentan dos soluciones: crear otro hilo diferente o usar la función “interrupt” comentada anteriormente. Por simplicidad se opta por la segunda opción ya que, en este caso, implementar un thread implicaría tener que controlar varias variables que no se usadas al mismo tiempo por diferentes hilos. La función “interrupt” lo que aporta es que cada vez que esta es llamada WebSocket++ se encarga de atender otras llamadas del servidor. Así cada vez que se añade un mensaje para enviar “interrupt” es llamado para que se compruebe si ha llegado un nuevo mensaje y atender a este o si se debe seguir con la ejecución anterior.

El siguiente pseudocódigo es el encargado de ejecutar las llamadas a interrupt. Cuando un nuevo GET es recibido se llama a “interrupt”, que estará llamandose recursivamente enviando un chunk por cada ejecución.

```
1 void message (identificadorConexion id) {  
2     switch(tipo_mensaje){  
3         ...  
4         case GET:{  
5             InfoConexion data = recuperarInfo( id);  
6             data->tree.getLODPath(camara, data->listToSend);  
7             data->nextChunk = data->listToSend.begin();  
8             interrupt(id);  
9             break;  
10        }  
11    ...  
12    }  
13 }
```

```
1 void interrupt(identificadorConexion id){  
2     InfoConexion data = recuperarInfo ( id);  
3     iterador nextChunk = data->nextChunk;  
4     if (LRUCache.exist( nextChunk )){  
5         Chunk c;  
6         if (L2cache.getChunk(nextChunk, c)!=NULL){
```

```

7      listaIDBorrados = LRUCache.insert(nextChunk, c.tamano);
8      while (!listaIDBorrados.empty()) {
9          send(id, prepararMensajeBorrado(listaIDBorrados.front()));
10         listaIDBorrados.pop_front();
11     }
12     send(id, prepararMensajeEnviarChunk(c));
13 } else {
14     data->priorityList.add(nextChunk);
15     L2cache.setPriorityList(data->priorityList);
16 }
17 } else {
18     LRUCache.update(nextChunk);
19 }
20 data->nextChunk++;
21 interrupt(id);
22 }

```

Uno de los objetivos de este proyecto era que el servidor estuviera preparado para multicitente, es decir que varios clientes se puedan conectar a este al mismo tiempo. Para lograr esto WebSocket++ asocia a cada conexión un identificador único. Con este identificador se puede indexar la información de cada conexión en un “map” para que así en cada mensaje se recupere la información correspondiente a esa conexión y poder actuar según el estado de la sesión. Además como cada iteración del bucle que se encarga de preparar los chunks se llama a “interrupt” logra que ante cada llamada se vaya intercalando entre los distintas sesiones que están ejecutando esa función.

6.3. Caché LRU

Lo ideal en aplicaciones red con gran número de transferencia de datos es evitar el mayor número de posible de envíos sin que esto repercuta gravemente en el rendimiento final.

PCM nos ofrece una lista de los Chunks que se deben visualizar según la cámara del cliente ordenados por su nivel de importancia, es decir, los primeros serán los que forman una primera imagen de las escena mientras que los siguientes la irán haciendo

más detallada. Las consultas a PCM se realizan con una cierta frecuencia determinada por el cliente pero lo que siempre ocurrirá es que un gran número de chunks estarán repetidos. Esto es un factor importante para optimizar el rendimiento y en el cual se basa la motivación por la que se implementa una caché de tipo LRU.

Una caché LRU (Menos Usada Recientemente o “Least Recently Used”) [22] se basa en guardar los datos ordenados según su uso y una vez esta esté llena y sea necesario realizar el borrado será eliminado el último elemento que se haya usado el seleccionado. En nuestro caso habrá una pequeña diferencia. PCM incluye una caché, L2Cache, que ya hace la función de guardar los elementos y por tanto no es necesario que se guarden en la caché LRU ya que esto solo aportaría redundancia en los datos incrementando altamente el tamaño total por cada sesión del cliente. Por este motivo, LRU no guardará los chunks sino simplemente su identificador para poder recuperarlo de la L2Cache y el orden en el que han sido utilizados.

Su funcionamiento es muy simple y se basa solo en dos únicas operaciones: insertar y actualizar.

- `insert(identificador,tamaño)`: se insertará ese identificador con el tamaño indicado dentro de la caché. Dicho elemento no debe existir en ella y debe ser comprobado antes de llamar a esta función. En caso de que no haya espacio en la caché se eliminará los elementos que hagan falta y se devolverá una lista con los identificadores de los elementos borrados para proceder a su envío de estos al cliente.
- `update(identificador)`: actualiza el orden de la caché colocando como primer elemento el indicado.

Su funcionamiento interno se basa en una lista y en un map. En el primero se guardan los identificadores según el orden de uso. En el map se asocia con el identificador del chunk un par formado por el tamaño de este y por un iterador que indica la posición en la lista. De esta forma cuando se actualiza la lista no hace falta recorrer la lista en búsqueda del elemento sino que se recupera su posición desde el map, lo que nos aporta un rendimiento mucho mejor ya que la búsqueda en map tiene una complejidad menor que en una lista. Esto solo es necesario en los casos de actualización cuando se inserta un

elemento se añade al inicio de la lista y cuando se borra se elimina el último elemento, siempre acordandose de añadir o eliminar su entrada correspondiente en el map.

En conclusión, cada vez que se tiene un chunk se comprobará si existe o no en la caché LRU. Si no existe se insertará en ella y se enviara ese chunk al cliente. Si fue necesario borrar algún elementos se enviará el identificador de estos para que el cliente pueda hacer espacio. Si el chunk ya es estaba en la caché no se realizará ningún tipo de envío y simplemente se actualiza su posición.

6.4. Diagrama de clases

PCServer ha sido desarrollado bajo el paradigma de Orientación a Objetos y realizado completamente con el lenguaje C++.

En este diagrama de la Figura 6.4 podemos observar todas las clases que forman PCServer y también aquellas que se usan de librerías externas usadas. Los objetos propios de PCServer son los siguientes:

- PCServer: esta es la clase principal del proyecto y la cual se encarga de manejar tanto la conexión con el cliente como las acciones llevadas a cabo por los mensajes recibidos mediante la librería WebSocket++. Los métodos `on_acción` serán llamados por su evento correspondiente:
 - “`on_open`”: cuando se abre una conexión con un cliente.
 - “`on_close`”: cuando se cierra una conexión con un cliente.
 - “`on_message`”: cada vez que llega un mensaje de cualquier cliente
 - “`on_interrupt`”: evento que fuerza que se compruebe si hay alguna petición nueva y atenderla.
- Parser: esta clase nos ofrece todos los métodos relacionados con el protocolo para la preparación y análisis de todos los mensajes posibles.
- ConnectionData: estructura de datos que tiene una conexión. Cada conexión tendrá una caché L2Cache, caché de PCM que guarda en memoria los chunks más

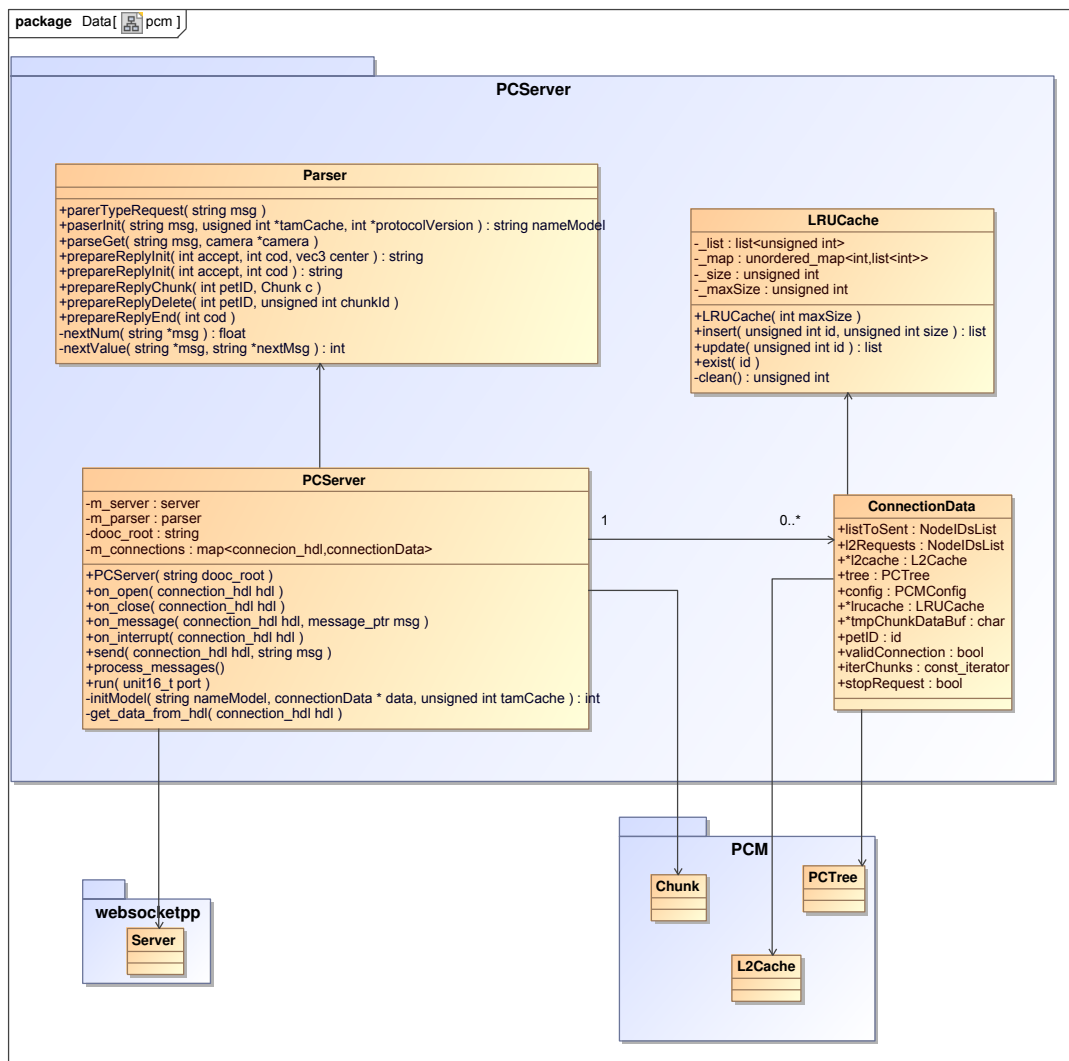


Figura 6.4: Diagrama de clases PCServer.

recientes, y una estructura PCTree de PCM que guarda información de la estructura de la nube de puntos. El resto de parámetros son simplemente para poder mantener la sesión entre distintos mensajes.

- LRUCache: caché que almacena que chunks tiene actualmente el cliente y que se ha sido explicada más detalladamente previamente.

Capítulo 7

Cliente

En este capítulo se explicará cómo se implementa el cliente, que dependencias tiene y cual es su estructura. Este cliente es un ejemplo concreto y está pensado para que sea accesible al mayor número posible de usuarios, por eso es un servicio web que solo requiere un navegador, pero podría implementarse otro tipo de cliente de forma parecida.

7.1. WebGL y funcionamiento general

Para la visualización de la nubes de puntos se usa WebGL. A mayores se utilizará una librería de alto nivel THREE.js que trabaja sobre WegGL. Esta librería nos ofrece ciertas características como cámaras, escenas, geometrías, etc. que facilitaran altamente la implementación.

En el caso de este proyecto hay que hacer especial hincapié en el sistema de partículas que nos ofrece THREE.js . Así, cuando un cliente reciba un nuevo chunk desde el servidor, se creará un sistema de partículas que contendrá todos los puntos pertenecientes a ese chunk . Además se guardará en un array el ID del chunk y un puntero a este sistema de partículas, lo que aportará una forma sencilla y rápida de localizar un sistema según su chunk para poder eliminarlo cuando sea necesario.

El resto se implementará de forma muy sencilla. Se crea una escena, como se hace normalmente, y a esta escena se le añadirá un Object3D, figura de la librería THREE.js,

que contendrá los sistemas de partículas que se han comentado previamente y que se irán añadiendo según lleguen los mensajes. Además se añadirán también algunas luces básicas también dadas por THREE.js. Todo esto se ejecuta una vez según se inicia la web. Tras esto WebGL llama continuamente a la función “render”, que es la encargada de renderizar la escena continuamente con todos los objetos que se hayan añadido a ella. Además en esta función también es la encargada de iniciar las peticiones del cliente. Se comprueba continuamente cuánto tiempo ha pasado desde la última vez que una petición ha sido enviada y cuando este es superior al tiempo establecido se realiza una nueva petición con la cámara que tiene actualmente.

7.2. WebSocket

Para el envío y recibo de datos se usará WebSocket como ya se ha explicado previamente. La parte cliente se basa en dos tipos de peticiones. La primera se realizará cuando se inicia la web y será simplemente para iniciar el modelo que se quiera. La otra petición se realizará cada un tiempo preestablecido arbitrariamente y será una petición de chunks al servidor enviando la posición de la cámara actual.

Para la recepción se ejecutará como se explicó previamente, cada vez que llegue un chunk se añadirá como un sistema de partículas a un objeto que está en la escena que se muestra.

El funcionamiento de WebSocket para enviar y recibir mensajes es parecido a como se hacía el servidor. Existen ciertas funciones que una vez implementadas en nuestro código será llamadas cuando un determinado evento ocurra, como por ejemplo cuando se recibe un mensaje o se cierra una conexión.

La recepción y envío de datos se realizará mediante una especie de threads para JavaScript llamados Web Workers, que serán explicados a continuación. De esta forma se separa claramente la parte encargada de la transmisión de datos y la encargada de la renderización y se evita el máximo posible de trabajo de esta última. Con esto se consigue intentar evitar que la aplicación no tenga un framerate bajo a causa de una pérdida de tiempo en la recepción y de envío de datos mientras estos son preparados.

7.3. Web Workers

Uno de los grandes problemas de JavaScript, ya desde su definición, es que se ejecuta en un subproceso único y no permite la ejecución de varios comandos al mismo tiempo. Antiguamente si se quería hacer esto había que optar a pequeños trucos para intentar simular una cierta simultaneidad. Para evitar esto y establecer una forma estándar para conseguir esto W3C define Web Socket [9].

Existen dos tipos de Workers, dedicados y compartidos. La mayor diferencia entre ellos es alcance de estos, mientras que los dedicados solo son alcanzables por la página que fue creada, los compartidos pueden ser alcanzable desde cualquier página creada desde un mismo origen (dominio). Por simplicidad y como en este proyecto sólo existe una página que necesita workers se ha optado por el uso de workers dedicados.

Para iniciar un Worker solo necesitamos tener su código separado en un fichero aparte y crear un Worker con dicho código desde el JavaScript principal o desde donde deseemos crear uno. Una vez creado, la página que lo ha creado y el worker se comunicarán a través de mensajes en lugar de tener zonas de memoria compartida. Una de las intenciones es que la parte encargada de renderizar el modelo no tuviera que, ni si quiera, preparar los datos sino que estos fueran directamente añadidos por el Worker en una variable compartida. Así el “render” solo se preocuparía de leer donde tiene los chunks añadidos y pintarlos en pantalla. Esto finalmente no ha sido posible por como funcionan los workers, que como no comparten ninguna zona de memoria sino que se comunican a través de mensajes implica que el “render” tiene que recoger el mensaje, analizarlo y añadir el chunk. Aun así se decide seguir con su uso ya que así no se pierde tiempo en la recepción de datos.

7.4. Diagrama de clases

El cliente ha sido desarrollado bajo JavaScript el cual es ejecutado desde un HTML que crea un contenedor canvas para WebGL y ejecuta el script principal.

En el diagrama de clases de la Figura 7.1 podemos ver la clase Main, encargada de renderizar la escena con WebGL. En ella se puede ver los elementos de THREE.js

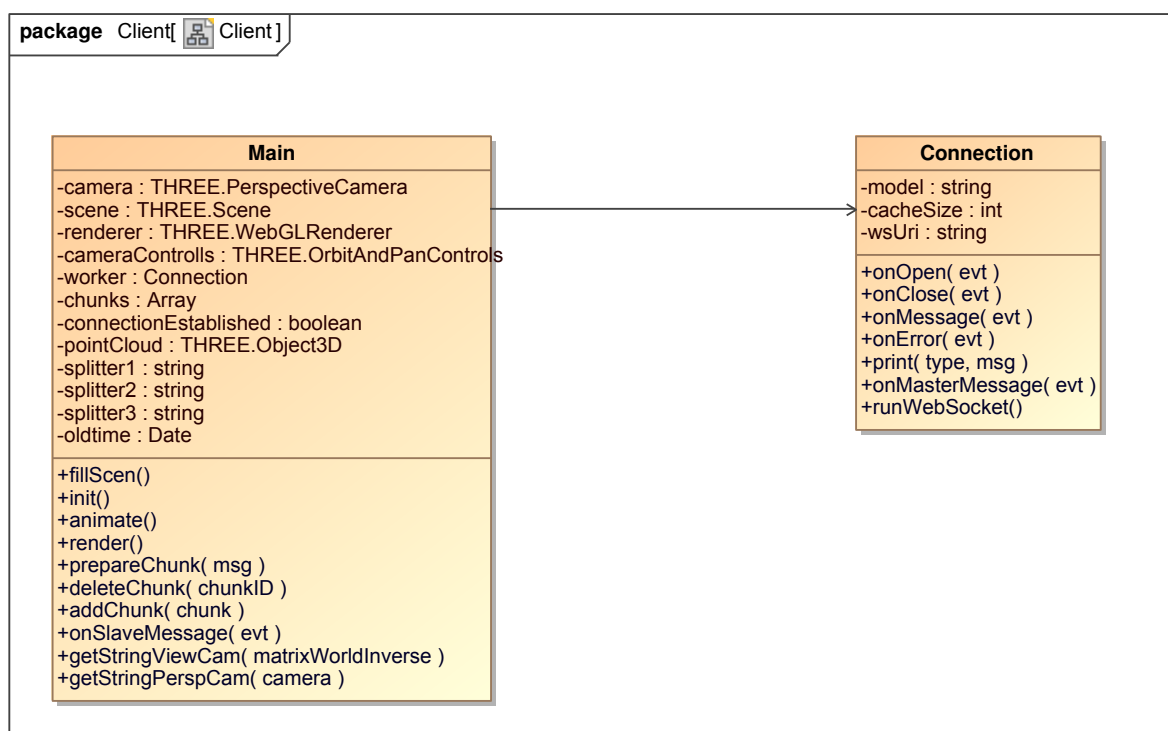


Figura 7.1: Diagrama de clases del cliente web.

necesarios para crear la escena, la nube de puntos (pointCloud) y el array que facilita su búsqueda (chunks). Entre las funciones se puede destacar:

- “onSlaveMessage”: encargada de recibir los mensajes del Worker.
- “getStringViewCam”: convierte la View Camera en String en el formato establecido por el protocolo.
- “getStringPerspCam”: convierte la Perspective Camera en String en el formato establecido por el protocolo.
- “prepareChunk”, “deleteChunk”, “addChunk”: son las funciones encargadas de gestionar (añadir y borrar) los chunks recibidos dentro de la escena.
- El resto de funciones se encargan de renderizar la escena WebGL.

Por otra parte, la clase Connection contiene en sus atributos el nombre del model (model), el tamaño máximo de su cache (cacheSize) y la dirección del servidor (wsUri). Entre sus funciones podemos ver todos los métodos necesarios para los eventos de WebSocket, una función para mandar a su creador que imprima algo por consola y el método que inicia su ejecución (runWebSocket).

Capítulo 8

Resultados

En este capítulo se muestran y analizan las pruebas realizadas sobre el sistema implementando, incluyendo algunos datos de rendimiento tanto en la parte cliente como en la parte servidor.

En la Figura 8.1 y en la Figura 8.2 se pueden ver dos ejemplos de ejecución.

Lo primero que se puede observar cuando se ejecuta es que, a día de hoy, el envío de tanta información por red es un procesos costoso y que requiere cierto tiempo para hacerlo. Esto se deduce viendo como los chunks van apareciendo poco a poco a causa del retardo que se produce. Este retardo viene dado por los siguientes procesos:

1. Recuperación del chunk en el servidor desde la caché L2 de PCM.
2. Preparación del mensaje en el servidor para ser enviado.
3. Tiempo de transporte a través de la red.
4. Recepción y preparación en el cliente.

La ventaja es que estos datasets son estáticos por lo que, aunque se tarde en recibir la información, sirve perfectamente para poder ver y analizar el modelo. En el caso de que fuera dinámicos dependiendo de algún tipo de acción desde el cliente, como por ejemplo un videojuego, si que seria inviable, el retardo entre que el cliente ejecuta una acción y recibe las acciones de este sería muy alto.

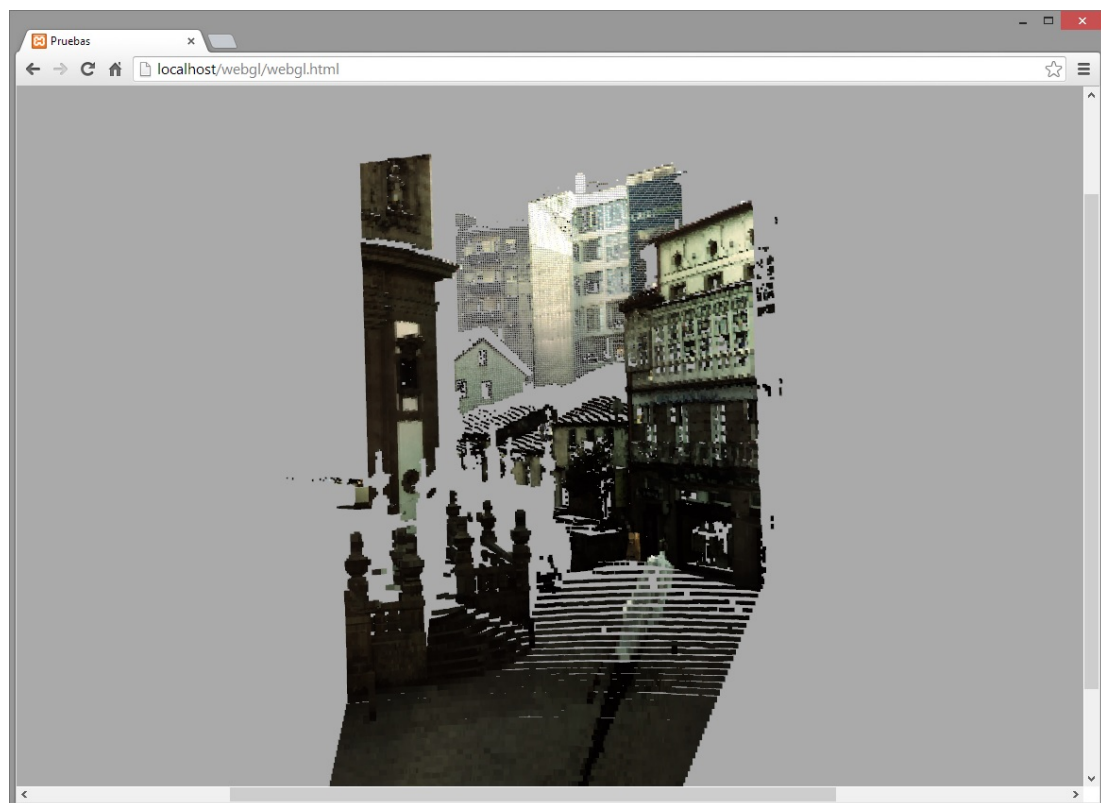


Figura 8.1: Captura del cliente web con el modelo Casas.

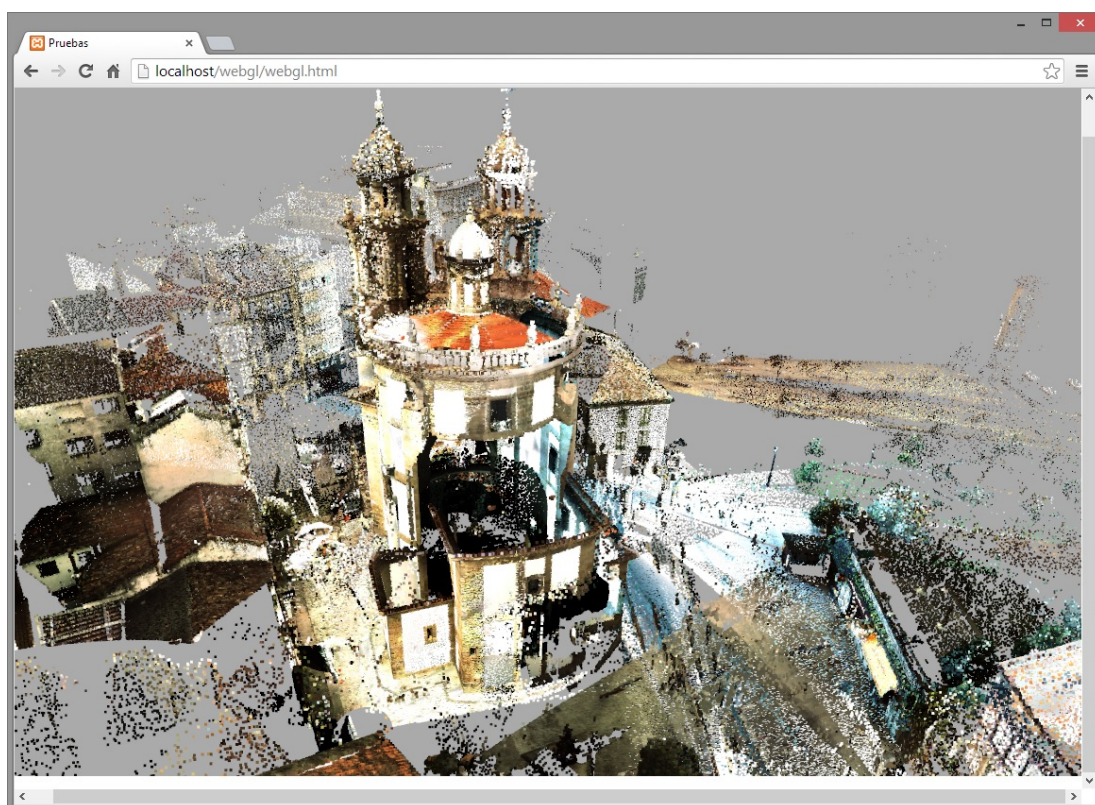


Figura 8.2: Captura del cliente web con el modelo Peregrina.

A continuación se muestra ciertos tiempos tomados en el cliente y servidor para ver el rendimiento que se obtiene. La máquina en la que se ha ejecutado el cliente y el servidor y se han tomado los datos es la siguiente:

- CPU: Intel Core i5 3570K 3400MHz (4 cores)
- RAM: 16 GBytes DDR3 1600
- SDD: m4 crucial 128GB SATA3
- GPU: nVIDIA GeForce GTX 660 Ti
- OS: Windows 8 Professional 64 bits
- Compilador: Microsof VC++ 2010

El dataset utilizado es el siguiente:

- Nombre: Iglesia de la Peregrina (Pontevedra)
- Escaneo LIDAR: ENMACOSA
- Propiedad: Ayuntamiento de Pontevedra
- Tamaño del dataset original: 3GB
- Tamaño compactado del BPC: 600MB
- N° Puntos: 30.645.025

8.1. Mediciones tomadas

En la parte servidor se pueden obtener dos tiempos, el tiempo máximo y el tiempo medio que tarda en recuperar un chunk de la L2Cache y en prepararlo para su envío.

$$T_{max} = 0,435ms$$

$$T_{med} = 0,117ms$$

El tiempo máximo corresponde siempre cuando los datos no están en RAM y la caché L2 necesita traerlos a memoria, mientras que el tiempo medio los datos ya están en RAM por lo que el tiempo que se tarda es considerablemente menor. Podemos observar como la caché L2 de PCM reduce casi en cuatro el tiempo. Un factor a tener en cuenta con respecto al tiempo máximo es en el tipo de disco duro que se realiza la prueba. Un SSD (“solid-state drive”) es un tipo de disco duro que puede llegar a mejorar en más de cuatro veces el tiempo de lectura con respecto a los discos duros tradicionales.

En la parte cliente se ha obtenido la media de preparación de un mensaje, desde que este es recogido hasta que es añadido en la escena. El tiempo obtenido es:

$$T_{med} = 13,89ms$$

Este tiempo es importante que nunca supere los 18 ms. En gráficos de computación una de las mayores metas es siempre que tu aplicación se muestre a 60 “frames” por segundo. Esto quiere decir que tienes unos 18 segundos para computar todo lo que necesites pero si tardas más de ese tiempo tendrás problemas y tu aplicación se verá como con retardo. En este caso se cumple esto y de media se puede conseguir los 60 “frames” aunque en algunos casos hay ciertos picos que hacen caer la tasa de imágenes por segundo pero que no agravan el rendimiento.

Capítulo 9

Conclusiones

Hoy en día la directriz general en el sector es ir desviando todas las gestiones posibles hacía grandes servidores especializados, lo que actualmente se denomina como “la nube”, que gracias a sus mayores recursos consiguen un mejor rendimiento que los PCs comunes. Aunque no todo es posible por el retardo que generan las redes, gracias a la gran evolución de estas, día a día se van consiguiendo mayores progresos. Este proyecto se basa en esa idea para ofrecer una solución a uno de los grandes problemas los datasets de nubes de puntos, su gran tamaño.

Con esto se ha implementado una solución cliente-servidor mediante la cual, prácticamente cualquier tipo de cliente que disponga de un navegador podrá visualizar cualquier nube de puntos a través de una red sin necesidad de tener el dataset en su propia máquina.

9.1. Cumplimiento de requisitos y objetivos

Se puede decir que tantos los objetivos como los requisitos establecidos en el primer capítulo se pueden dar como cumplidos:

1. Se ha creado un protocolo perfectamente válido para la transferencia de nubes de puntos de gran tamaño bajo demanda. Además se ha buscado que este protocolo sea fácilmente ampliable para futuros requisitos.

2. Se ha diseñado e implementado un servidor que, integrado con el paquete software PCM y su librería, es capaz de enviar los datos necesarios según los solicita el cliente.
3. Se ha diseñado e implementado un cliente web HTML que recoge los datos que le llegan desde el servidor y los muestra en pantalla con WebGL.

Además el software también es:

1. Multiplataforma. Continuando el camino que sigue PCM, se han usado solo librerías que sean perfectamente válidas en los principales sistemas operativos.
2. Multicliente. Aunque quizá no perfectamente optimizado para ello, el servidor es capaz de aceptar y tratar

9.2. Posibles mejoras

Como en todo proyecto de investigación siempre existen numerosas mejoras que se puede implementar para intentar mejorar el rendimiento. En este proyecto podemos destacar:

- Seguridad. Actualmente el servidor no aporta ningún tipo de seguridad, cualquiera que acceda a la web podrá ver el modelo que quiera. Una posible mejora es aumentar esta seguridad con la creación de cuentas por usuario para que así cada usuario solo pueda acceder a unos determinados datasets. Además WebSocket esta completamente preparado para conexiones cifradas.
- Comprensión. En esta versión los mensajes se mandan sin ningún tipo de comprensión para que sea fácil ver que información se manda y poder detectar posibles errores pero esto aumenta el tiempo que tarda en realizarse la transmisión de los datos.
- Mejoras multicliente. Aunque el servidor es multicliente, cada sesión es tratada por un único hilo por lo que si se realizan múltiples conexiones cada cliente tardará más en recibir los datos del cliente.

- Mejoras visuales en el cliente. El cliente implementado en este proyecto es el más básico posible enseñando solo los puntos en su posición correcta con su color. A mayores se podría añadir mejoras en el control de la cámara, mayor muestra de información y mejoras en la muestra de puntos.

9.3. Futuro

En la actualidad el proyecto PCM continúa en pleno desarrollo, al ser la base de un proyecto I+D financiado con fondos CDTI (proyecto ToVIAS). Todo el código aportado por este proyecto, con la aproximación cliente-servidor y el visualizador WebGL se ha incorporado en dicho proyecto. A corto plazo se encuentran en preparación diversas publicaciones científicas, que pretenden presentar las aportaciones relevantes de este sistema. A largo plazo se tanteará la búsqueda de la explotación comercial del sistema.

Y no solo eso, el ritmo de progreso de las redes crece a ritmo continuo y a pasos agigantados, logrando que con el tiempo el traspaso de grandes cantidades de datos sea mucho más óptimo y fluido, otorgando una gran mejora

Todos estos factores pueden lograr que PCServer cobre una gran importancia y este puede seguir en desarrollo implementando las mejoras explicadas previamente.

9.4. Código fuente del proyecto

El repositorio con el código fuente se encuentra accesible en http://tovias.citic.udc.es/projects/pcm/repository?rev=dev_webgl

Bibliografía

- [1] “Lidar—light detection and ranging—is a remote sensing method used to examine the surface of the earth..” Available: <http://oceanservice.noaa.gov/facts/lidar.html>.
- [2] Jaspe, A., *Point Cloud Manager. Sistema multirresolución para el tratamiento de grandes satasets de nubes de puntos 3D*. Proyecto Fin de Máster, 2012.
- [3] Alliance, T. A. Available: <http://www.agilealliance.org/>.
- [4] Alliance, T. A., “Manifesto for agile software development.” Available: <http://agilemanifesto.org/>.
- [5] “Scrum.” Available: <http://www.scrum.org/>.
- [6] “Websocket++.” Available: <http://www.zaphoyd.com/websocketpp>.
- [7] W3C, “Api websocket.” Available: <http://www.w3.org/TR/websockets/>.
- [8] “Boost. asio.” Available: http://www.boost.org/doc/libs/1_54_0/doc/html/boost_asio.html.
- [9] W3C, “Api web workers.” Available: <http://www.w3.org/TR/workers/>.
- [10] “git –fast-version-control.” Available: <http://git-scm.com/>.
- [11] Chacon, S., *Pro Git*. Apress, 2009.
- [12] “Gnu general public license v2.” Available: <http://www.gnu.org/licenses/gpl-2.0.html>.

-
- [13] “L^AT_EX - a document preparation system.” Available: <http://www.latex-project.org/>.
- [14] Luebke, D., Reddy, M., Cohen, J. D., Varshney, A., Watson, B., and Huebner, R., *Level of Detail for 3D Graphics. Application and Theory*. Morgan Kaufmann, 2002.
- [15] Samet, H., *Spatial Data Structure*. 2012.
- [16] Pfenning, F., “Computer graphics - lecture 17: Spatial data structures.” Available: <http://www.cs.cmu.edu/~fp/courses/02-graphics/pdf-color/17-spatial.pdf>.
- [17] Gobbetti, E. and Marton, F., “Layered point clouds,” *Eurographics Symposium on Point Based Graphics*, 2007.
- [18] IETF, “The websocket protocol.” Available: <http://tools.ietf.org/html/rfc6455>.
- [19] Lubbers, P. and Greco, F., “A quantum leap in scalability for the web,” tech. rep., Kaazing Corporation.
- [20] Matsuda, K. and Lea, R., *WebGL Programaming Guide. Interactive 3D Graphics Programming with WebGL*. Addison-Wesley, 2013.
- [21] Angel, E. and Shreiner, D., *WebGL Programaming Guide. Interactive 3D Graphics Programming with WebGL*. Addison-Wesley, 2012.
- [22] Day, T., “LRU cache implementation in C++.” Available: <http://timday.bitbucket.org/lru.html#x1-20001>.