Proyecto de Grado

Cámaras Heterogéneas

InCo – Laboratorio de medios

Marzo 2016

Integrantes	
Rodrigo Alvarez	5.003.515-6
Gonzalo Martinez	4.532.461-3
Rodrigo Cardozo	4.669.734-0

Tutores
Christian Clark
Tomas Laurenzo

Índice

1 Introducción

El proyecto Cámaras Heterogéneas se originó bajo la necesidad de tener un sistema de código abierto, fácilmente extensible, que permita reunir los datos obtenidos a través de un conjunto de cámaras, ya sean RGB o sensores de profundidad, para formar una representación tridimensional de una escena.

Es decir que se podría montar una escena de cualquier tipo, en cualquier ambiente, fijar un conjunto de cámaras estratégicamente posicionadas y filmar la escena para verla, incluso en tiempo real, desde cualquier ángulo.

Las posibles aplicaciones de este proyecto son muy diversas, desde la más directa que sería la producción de películas realmente 3D observables desde cualquier punto, pasando por la vigilancia tridimensional, hasta la generación de juegos de realidad virtual que incluyan escenarios de la vida real de cualquier parte del mundo.

Por las características y lo amplio del proyecto, y sin entrar en detalles técnicos, hubo múltiples aspectos y desafíos tecnológicos que debieron ser resueltos; tales como la calibración de las cámaras, la recolección de los datos, la distribución del procesamiento, la comprensión de los datos, la generación de la información tridimensional, el renderizado, la texturización, entre otras.

Para lograr el objetivo se hizo uso de ciertas tecnologías de software que se encuentran disponibles, a saber: openFramework, OpenGL, PCL, OpenNI, VCG library, entre otras que se detallarán más adelante. La razón de utilizarlas es que ellas ya abordan y ofrecen una solución a algunos de los subproblemas que identificamos en este proyecto.

Si bien la arquitectura se explicará más adelante, ya queda claro que el proyecto se compondrá de diversos módulos. A grandes rasgos la idea fue dividir el sistema en tres aplicaciones principales. En un extremo, los Clientes, que se dedican a recopilar los datos. En el otro extremo, el Reproductor, que renderiza la escena. Y en medio de éstos, el Servidor, encargado del procesamiento más pesado de la información. Estos tres módulos deben estar conectados para cumplir el requerimiento del procesamiento en tiempo real. Además, se implementó un cuarto y quinto módulo, uno para la detección de las cámaras y otro para la calibración de las mismas. Este último módulo proporcionará datos de entrada para los primeros tres.

Diagrama 1-1 *Diagrama de componentes del sistema.*

En las siguientes páginas el objetivo será desarrollar todas estas ideas, explicar cómo se hizo para lograrlas y explorar, a su vez, el estado del arte en este campo de la computación.

El informe se desarrolla bajo cinco grandes títulos: Descripción del problema, Estado del arte, Solución propuesta, Conclusiones y Trabajos futuros, Proceso de desarrollo. A su vez, adicionalmente se incluye un Glosario y una Bibliografía.

La primera de esas secciones, planteando los principales problemas relacionados al diseño e implementación del sistema, pero sin entrar en los detalles específicos ni en la solución elegida en cada uno de ellos.

En la segunda sección, se exploran los principales conceptos y tecnologías que rodean al proyecto. Éstos, serán de vital importancia para entender más adelante algunos de los subproblemas que componen al sistema. Ya sea porque los conceptos definen algunos de estos subproblemas o porque le dan solución a los mismos.

La tercera, aborda el diseño y la implementación de la solución propiamente dicha. Esta sección explica las cuestiones técnicas relacionadas a la arquitectura y detalla cada uno de los módulos fundamentales que conforman el sistema.

La siguiente sección, trata sobre los resultados a los que se llegó con la implementación entregada. Es aquí donde se analizan métricas para medir el desempeño final del sistema. Además, se enumeran y explican cada uno de los puntos en los que se podría mejorar la implementación actual, mejoras que no implicarían rehacer el sistema gracias a que el proyecto fue pensado para que fuese extensible.

La sección Proceso de desarrollo cuenta el camino que siguió el grupo en la elaboración de la solución, desde el estudio de las tecnologías relacionadas al proyecto hasta la documentación final.

La última sección es un Glosario, que expone una breve descripción sobre los principales conceptos relacionados al proyecto y que figuran en este informe.

2 Descripción del problema

El Proyecto Cámaras Heterogéneas es un sistema para la grabación y reproducción de escenas tridimensionales, a partir de un conjunto heterogéneo de cámaras RGB y de profundidad.

El objetico de esta sección es describir los problemas más importantes que tomaron parte en la realización de este proyecto.

A modo de resumen, el proyecto tiene que ser un sistema distribuido, multiplataforma, extensible y de código abierto, capaz de interactuar con un número indeterminado de dispositivos de grabación de imágenes y nubes de puntos; generar un modelo tridimensional texturizado, en tiempo real, que pueda ser visualizado por medio de una interfaz que permita la navegación por la escena, con un framerate aceptable.

El primer desafío implica restringir el abanico de tecnologías que se pueden utilizar, a aquellas que son de código abierto. Esto plantea la necesidad de implementar ciertas partes del proyecto, que no están resueltas de manera directa por alguna librería o proyecto de código abierto.

El siguiente desafío es el requerimiento no funcional de que el sistema sea multiplataforma y performante. Haciendo que la elección del lenguaje de programación sea un punto importante. Por estas razones, el lenguaje seleccionado para la implementación del proyecto es C++, que es multiplataforma y permite tener un gran control sobre los recursos del sistema.

Hacer que el sistema sea extensible requiere que el proyecto parta de un buen diseño inicial de la arquitectura. El objetivo con este requerimiento es tener la posibilidad de reemplazar, en el futuro, módulos enteros del proyecto y que el mismo siga funcionando, pese a las modificaciones introducidas.

Relacionado al punto anterior se encuentra el hecho de que el sistema, por cuestiones de performance, tiene que distribuir la carga entre los diferentes módulos. Esto hace que el pasaje de la información entre ellos, imponga ciertas restricciones. Siendo necesario en algunas ocasiones utilizar memoria compartida y en otros la red. Y a su vez, en el caso de este último, hay veces que será conveniente utilizar una implementación de TCP y en otras una implementación de UDP.

Una de los desafíos más grandes que tiene el proyecto, y que le da nombre al mismo, es el requerimiento no funcional de que el sistema pueda tomar los datos de diferentes tipos de cámaras, ya sean RGB o de profundidad. Donde ambos tipos, recolectan información muy distinta entre sí de la escena. Para el caso de las cámaras RGB se espera como resultado una matriz de píxeles, y para el caso de las cámaras de profundidad, una matriz de puntos. Más aún, las cámaras pueden ser de diferentes marcas, siempre y cuando los datos que provean cumplan ciertas restricciones, definidas en una interfaz.

El sistema debe proporcionar la opción de persistir la información con la que se está trabajando. Es decir, todas las imágenes que toman las cámaras RBG y todas las nubes de puntos que provienen de los sensores de profundidad, deben poderse guardar en disco para su posterior procesamiento, en caso de que se quiera que el sistema funcione en modo diferido. A su vez, tiene que existir la opción de que la salida del sistema también se persista en disco, para que pueda ser reproducida en futuras oportunidades.

Por cuestiones de precisión, el sistema debe contar con un módulo de calibración intrínseca. El objetivo de este tipo de calibración es aplicar ciertas correcciones por software a las imágenes que las cámaras toman. Esas correcciones son necesarias debido a imperfecciones que son propias de las cámaras que se utilizan durante el proceso de filmación.

Otro de los desafíos que plantea el proyecto, y que se desprende de la utilización de varias cámaras para filmar una misma escena, es hacer la unión de los datos recogidos por las diferentes cámaras, ya sea que éstas provean imágenes o matrices de puntos. Este proceso se conoce como calibración extrínseca y tiene por objetivo determinar la posición de las cámaras en el espacio, para poder aplicar después, las transformaciones adecuadas que hacen coincidir las nubes de puntos y las texturas.

Hay otros aspectos que deben ser tenidos en cuenta a la hora de la implementación, pero que escapan a los que respectan a la calibración intrínseca y extrínseca. Uno de ellos, es la corrección de color en las imágenes que toman las cámaras RGB, debido a los cambios de iluminación durante el proceso de filmación, que pueden afectar de diferente manera a cada una de ellas. Este fenómeno, logra provocar inconsistencias en la texturización durante el proceso de renderización, en especial en los bordes que se generan en las uniones de las imágenes.

El siguiente problema que surge, debido a la utilización de más de un sensor de profundidad, es que los mismos generan interferencia entre sí, cuando al menos dos de ellos filman la misma área de la escena. Este fenómeno es propio de los escáneres de luz estructurada, como Kinect. La solución al problema pasa por utilizar motores en la base de los sensores, para que los hagan vibrar en patrones diferentes y con ello evitar la interferencia.

La sincronización de los datos que provienen de las diferentes cámaras, es otro de los desafíos en el procesamiento de la información. El sistema hace uso de una serie de computadoras, que, a su vez, están conectadas a un conjunto de cámaras que recogen la información de la escena que se está filmando. Cada una de esas cámaras funciona a un framerate independiente, que incluso puede variar a lo largo del tiempo. El objetivo de la sincronización es ordenar esos frames y agruparlos de forma coherente, generando instantáneas completas de la escena a partir de los fragmentos parciales recogidos por las cámaras.

Para cumplir con el requerimiento no funcional de que el sistema final cuente con un framerate relativamente bueno, es necesario hacer uso de la paralelización del procesamiento. Esto quiere decir que, cada una de las etapas por las que pasa la información, desde que es capturada por las cámaras hasta que es mostrada en pantalla, debe ser optimizada para no generar cuellos de botella. Una forma de hacer esta optimización es dividir las tareas, para que corran en procesos diferentes y así hacer un mayor uso de los recursos del sistema.

Ligado al requerimiento anterior y debido a que el sistema en su conjunto maneja constantemente una gran cantidad de datos, la comunicación entre sus partes tiene que estar optimizada para ser capaz de soportar la transferencia masiva de información. Una forma de atacar este problema, es hacer uso de la compresión, ya sea con pérdida o sin pérdida, dependiendo del tipo de datos que se quiere transmitir. Teniendo en cuenta a su vez, el tiempo extra en el que se incurre por aplicar este tipo de algoritmos. Es decir que, el tiempo que toma hacer la compresión, sumado al tiempo que toma mandar los datos comprimidos, debe ser menor al tiempo que demora mandar la información, pero sin comprimir.

El sistema se compone de una secuencia de módulos interconectados en serie, donde la información que recogen los Clientes es enviada al Servidor, quien se encarga de procesarla para después reenviarla al Reproductor. Es por ello que, el framerate en el que ejecuta el proyecto dependerá finalmente del framerate más lento de estos tres módulos. Entonces, implementar algún mecanismo para coordinar los trabajos que hacen tanto los Clientes como el Servidor y el Reproductor es fundamental, para no desperdiciar recursos haciendo tareas que no contribuirán en nada a mejorar el desempeño del sistema en su conjunto.

Uno de los requerimientos funcionales más importantes del proyecto es que el sistema cuente con un programa para visualizar la escena tridimensional en tiempo real. Esto significa que se debe implementar una aplicación que sea capaz de consumir los datos de la escena, por algún medio en tiempo real, y hacer la renderización correspondiente. Esos datos se componen, entre otras cosas, de una malla de polígonos y un juego de imágenes sincronizadas. Entonces, la malla de polígonos debe actualizarse lo más rápido posible, a medida que llegan los datos, y a su vez responder a los movimientos que hace el usuario desde la interfaz, para visualizar la escena desde el ángulo deseado.

Ligado al punto anterior aparece el problema de la texturización, es decir, cómo mapear las imágenes al modelo tridimensional. O lo que es lo mismo, determinar con qué textura se debe pintar cada una de las caras de la malla de polígonos, que se generó a partir de la nube de puntos. Este problema se vuelve más complejo cuando se tiene en cuenta que la malla se va transformando durante la filmación de la escena, es decir, que la posición y el número de caras cambia a lo largo del tiempo.

Todos estos puntos son algunos de los muchos desafíos que supone este proyecto. Problemas que se irán detallando y resolviendo con mayor detenimiento en las diferentes secciones de este informe.

3 Estado del arte

Esta sección presenta el estado actual de las tecnologías aplicadas durante el desarrollo del proyecto. En una primera parte se enfoca en el estudio sobre los dos tipos de dispositivos de entrada utilizados: las cámaras RGB y los sensores de profundidad.

Dentro de las cámaras RGB se estudia dos problemáticas conocidas que son la calibración intrínseca y extrínseca. Además, se analizan diferentes formatos existentes para el almacenamiento de las imágenes generadas. En el caso de los escáneres 3D, o sensores de profundidad, se estudian las categorías en las que se dividen, la salida que generan, y se hace particular énfasis en el Kinect de Microsoft, dado que fue el sensor más utilizado durante la realización del proyecto.

Una vez conocida la salida provista por los dispositivos mencionados, es necesario determinar la manera en que éstas se integraran al proyecto. En el caso de los sensores de profundidad, su salida es una nube de puntos, por esta razón es que se presentan algoritmos que permiten formar mallas de polígonos y diferentes formatos para poder almacenarlas. En el caso de las imágenes, por otro lado, se deben conocer técnicas de texturización para poder aplicarlas sobre estas mallas.

También se tratarán los frameworks y librerías que permitieron la implementación del proyecto, facilitando el desarrollo del mismo. Entra ellas se destacan openFrameworks [1], OpenGL [2], OpenNI [3], PCL [4] y VCG library [5].

3.1 Cámaras RGB

Actualmente existe una variedad muy amplia de cámaras, pero todas ellas siguen manteniendo un mismo objetivo: capturar la imagen que se encuentra en el campo visual.

A grandes rasgos éstas constan de un compartimiento oscuro cerrado con una abertura en uno de sus extremos para dejar pasar la luz, conocida como apertura, cuyo diámetro puede modificarse, y una superficie plana de formación de la imagen o de visualización para capturar la luz en el otro extremo. Además de eso, las cámaras suelen contar con un dispositivo, que se encuentra por delante de la abertura que se mencionó antes, que contiene un conjunto de lentes convergentes y divergentes que se denomina objetivo, cuya misión es controlar la luz entrante y enfocar la imagen.

Mientras la apertura controla la cantidad de luz que entra en la cámara, el obturador controla el tiempo en el que la luz incide sobre la superficie de grabación durante la toma de la fotografía [6].

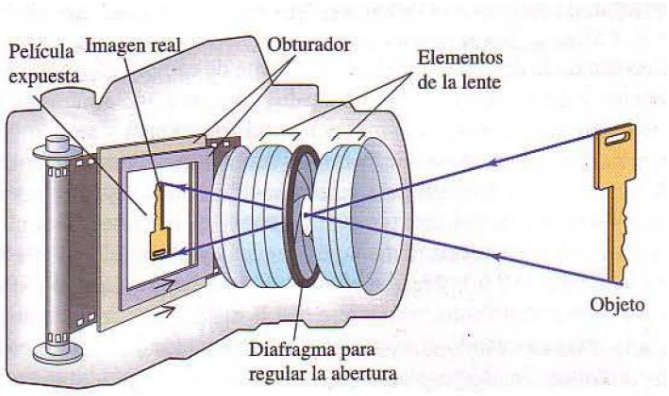


Ilustración 3-1 *Elementos fundamentales de una cámara fotográfica* [7]

Las cámaras se pueden catalogar en múltiples grupos de acuerdo al criterio de clasificación que se utilice. Existen las cámaras compactas, las cámaras réflex, las cámaras digitales, las cámaras estereoscópicas, etcétera. En este proyecto las cámaras que se utilizarán serán las que entran en la categoría de digitales. Una cámara digital es un dispositivo electrónico usado para capturar y almacenar fotografías digitales en lugar de usar películas fotográficas.

Debido a cómo se diseñan los lentes de las cámaras y desajustes técnicos en el montaje de las mismas, estos dispositivos generan distorsiones en las imágenes que capturan. Dicha distorsión tuvo que ser contemplada a la hora de tratar con las imágenes que las cámaras le proveen al sistema, puesto que en este proyecto la precisión es fundamental para lograr buenos resultados. El problema en cuestión se conoce como calibración intrínseca e implica hacer una corrección por software de las imágenes. La tarea será delegada a una de las librerías de OpenCV, que ya dispone de un algoritmo para este propósito. De todas formas, en la sección 4.2.2.1 se detallará con mayor detenimiento el proceso detrás de la calibración intrínseca.

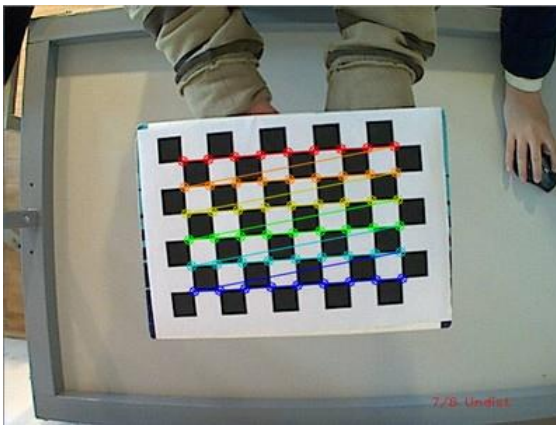


Imagen 3-1 *Imagen tomada por una cámara RGB cualquiera, que no ha sido tratada por el algoritmo de Calibración intrínseca.*

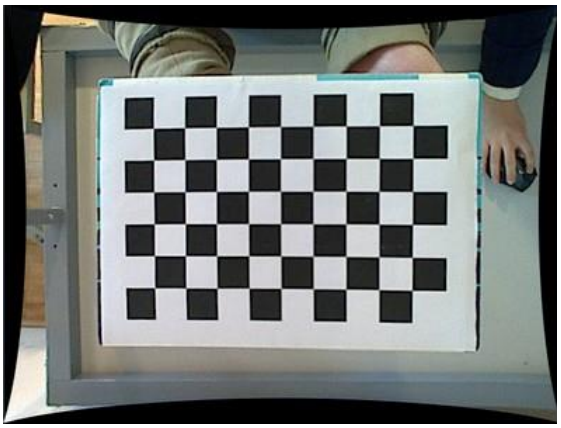


Imagen 3-2 *Imagen tomada por la misma cámara RGB que la anterior, después de ser tratada por el algoritmo de Calibración intrínseca.*

Otro desafío totalmente diferente, pero que se relaciona con las imágenes de las cámaras, es la calibración extrínseca, que tiene que ver con cómo se ajusta la imagen que toma la cámara A con la imagen que toma la cámara B, ambas ubicadas en diferentes posiciones espaciales. Los detalles detrás de este proceso se explicarán en la sección 4.2.2.2.

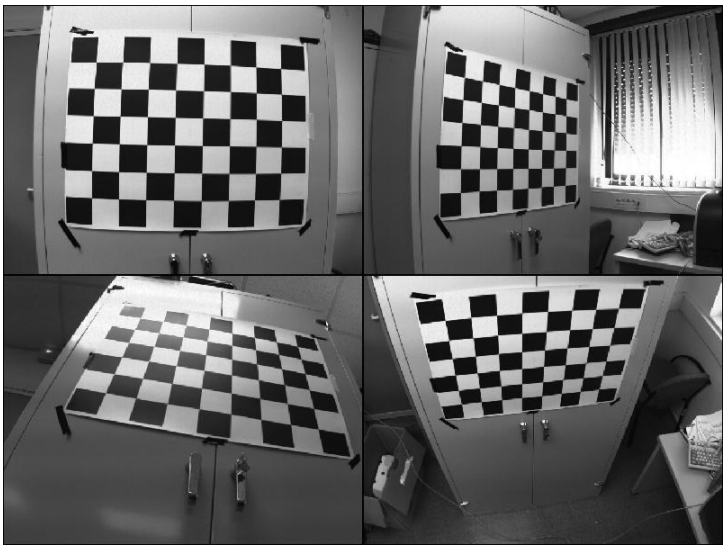


Imagen 3-3 *Imágenes de una misma escena tomadas por una serie de cámaras RGB posicionadas en lugares y con ángulos diferentes.*

Por último, es importante conocer la manera de almacenar las imágenes capturadas por las cámaras. En la siguiente sección se explican algunos formatos de almacenamiento conocidos y sus características más importantes.

3.1.1 Formato de almacenamiento de imágenes

Hay diferentes formatos digitales para el almacenamiento de imágenes, cada uno de ellos con características que permiten determinan cuál de estos es el más indicado para utilizar, dadas las necesidades del usuario. Algunos se almacenan sin compresión y sin pérdida de información, otros en cambio, se comprimen y pueden llegar a perder calidad. Algunos de los formatos más conocidos son:

- **PNG**: es un formato de almacenamiento sin pérdida de información. Utiliza patrones de la imagen para poder comprimir su tamaño. El algoritmo de compresión utilizado es perfectamente reversible con lo que la imagen se recupera exactamente como es. Además, PNG soporta transparencias en las imágenes.
- **JPG**: está optimizado para la fotografía, para imágenes que tengan una amplia gama de colores. Puede tener niveles de compresión muy altos manteniendo imágenes de gran calidad. El grado de compresión es ajustable. Almacena información con 24 bits de colores.
- **BMP**: es un formato sin compresión creado por Microsoft. Las imágenes almacenadas suelen ocupar más espacio en disco que con los otros formatos, esto se debe a que para guardar una imagen como bitmap no se utiliza ningún algoritmo de compresión.

- **GIF**: permite hasta 256 colores, si la imagen tiene menos colores, GIF puede representarla exactamente como es, en cambio, si contiene más colores se debe utilizar un algoritmo para aproximarlos a los 256 que se tienen disponibles. El algoritmo de compresión de GIF funciona en dos etapas: primero, reduce el número de colores, por lo tanto, reduce el número de bits necesarios por pixel para cada imagen, segundo, reemplaza grandes patrones del mismo color por una abreviación de qué color es y cuantas veces se repite.

Elegir el formato adecuado para almacenar imágenes depende mucho del uso que se le dé a las mismas, por esta razón es necesario conocer las características que presentan cada uno de ellos [8].

3.2 Escáner 3D

Los escáneres 3D se encuentran en diferentes formas y tamaños para adaptarse a la escala y a los detalles del objeto que tienen como objetivo medir. Están creados para recolectar información de forma y apariencia, dado que a partir de ella habitualmente se construyen modelos digitales en tres dimensiones. Suelen adquirir coordenadas X, Y, Z de la superficie del objeto que miden, cada coordenada X, Y, Z es conocida como un punto. La unión de todos los puntos obtenidos se denomina nube de puntos.

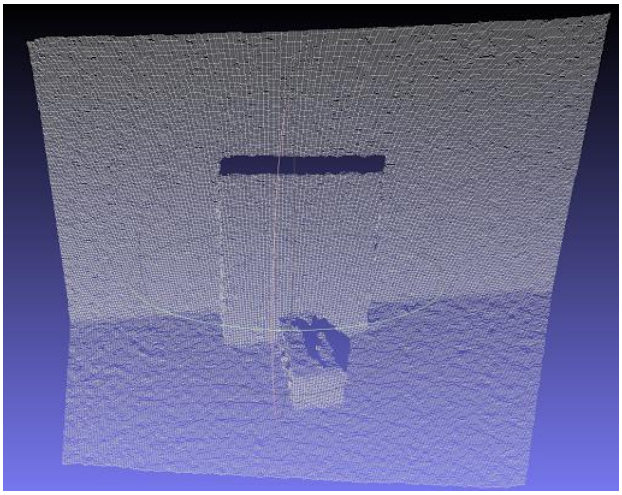


Imagen 3-4 *Nube de puntos obtenida con un Kinect visualizada en MeshLab.*

Las diferentes tecnologías con las que están construidos los escáneres les dan a estos sus limitaciones, ventajas y costos. Es necesario saber qué características tienen los objetos con los cuales se va trabajar para poder determinar el escáner que más se adapta a estas, por ejemplo, los que usan tecnologías ópticas, encuentran muchas dificultades en ambientes con mucho brillo, espejos y objetos transparentes.

La finalidad de un escáner 3D puede ser crear una nube de puntos de una escena objetivo. Estos puntos pueden ser utilizados para extrapolar la forma del objeto de análisis. Si además se

le agrega una cámara con información RGB también es posible que el objeto tenga información del color de su superficie.

Los escáneres 3D tienen grandes similitudes con las cámaras tradicionales, al igual que estas, poseen un campo de visión y sólo pueden obtener información de los objetos que no quedan ocultos. Mientras que una cámara obtiene información del color de los objetos de su campo de visión, un escáner 3D obtiene la distancia que hay a cada objeto para cada punto de la imagen. Esto permite un posicionamiento 3D de cada pixel.

Un solo escáner en una posición fija, no es capaz de producir un modelo 3D completo de un objeto. Para tomar información desde todos sus lados, es necesario contar con varios escáneres 3D o bien posicionar un escáner en diferentes puntos. Es importante tener en cuenta que el segundo caso únicamente funciona si el objeto se encuentra quieto en la escena. Los datos producidos por estos escáneres deben estar en un mismo sistema de referencia y a su vez deben estar alineados de tal manera de unir los datos y que dé como resultado el modelo completo. Un conjunto de escáneres está en un mismo sistema de referencia, si se encuentran en un mismo espacio tridimensional donde se respetan las posiciones de cada uno con respecto a los otros, para así poder representar la escena u objeto 3D correctamente.

Los diferentes tipos de tecnologías que utilizan los escáneres 3D principalmente se pueden clasificar en dos, los que requieren de contacto con el objeto y los que no. Dentro de las tecnologías que no requieren de contacto se pueden dividir en dos categorías, activas y pasivas.

3.2.1 Tecnología de contacto

Brevemente, la tecnología de contacto es la que como lo dice su nombre, entra en contacto con el objeto a medir. Son lentos comparados con los sistemas ópticos y pueden modificar la estructura del objeto a estudiar, por esta razón no es recomendable utilizarlos para el estudio de objetos frágiles. Un ejemplo de escáner 3D de contacto es un CMM. Un CMM, Coordinate Measuring Machines de sus siglas en inglés, es una máquina que por medio de un brazo tantea el objeto a estudiar para obtener sus medidas [9].

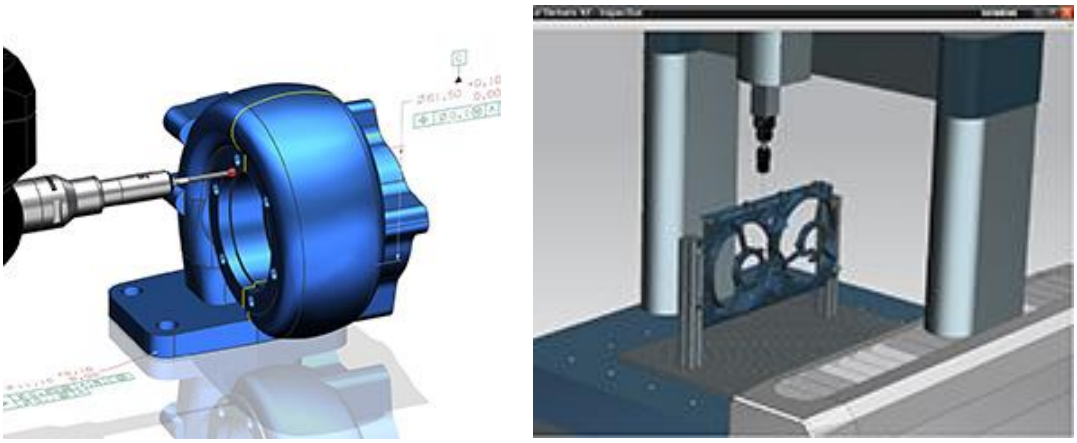


Imagen 3-5 *CMM estudiando la superficie de un objeto* [9].

3.2.2 Tecnología sin contacto

3.2.2.1 Activos

Los escáneres activos emiten algún tipo de luz o radiación que por medio de una cámara detectan la reflexión sobre un objeto. Los posibles tipos de emisión son: luz, láser y rayos X. Algunos ejemplos de esta clase de escáner son [10]:

- **Tiempo de Vuelo**: Mide el tiempo de viaje de ida y vuelta de un pulso de luz. Posee un largo alcance.
- **Triangulación**: Un haz de luz incide sobre un objeto y se utiliza una cámara para buscar la ubicación del punto del láser. Se denomina triangulación porque el punto del láser, la cámara y el emisor del láser forman un triángulo. La distancia entre el emisor del láser y la cámara es conocida, el ángulo del vértice del emisor del láser también y el ángulo del vértice de la cámara puede ser determinado por la ubicación del punto del láser en la cámara. Dado que se tiene un lado y los dos ángulos adyacentes se pueden determinar las otras medidas del triángulo conformado por los tres puntos.

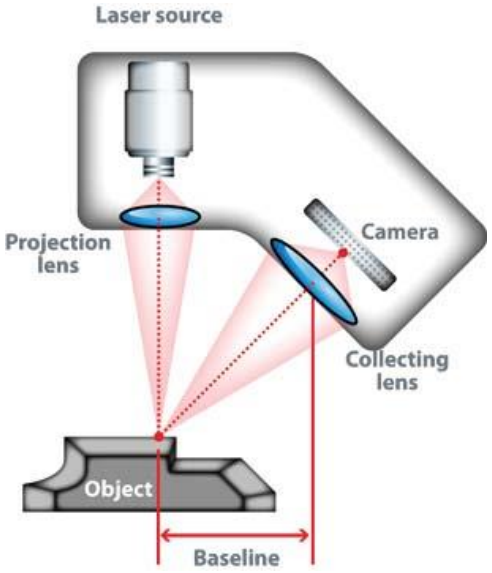


Imagen 3-6 *Triangulación entre el láser, la cámara y el punto en un objeto* [11].

- **Diferencia de fase**: Mide la diferencia de fase entre la luz emitida y la luz recibida, utilizando dicha medida para estimar la distancia del objeto. Se encuentra en un punto intermedio en cuanto a precisión y alcance de los dos métodos mencionados anteriormente.
- **Luz estructurada**: Proyectan un patrón de luz y capturan la deformación del patrón en el objeto proyectado. Generalmente cuentan con el emisor del patrón y una cámara que lo captura calibrada entre ellos. Un ejemplo de este tipo de escáner es el Kinect del cual se explica su funcionamiento en la siguiente sección [10].

- **Volumétricos**: La tomografía computada es un ejemplo usado en el área médica, genera una imagen tridimensional del interior de un objeto con una gran cantidad de imágenes de rayos X de dos dimensiones.

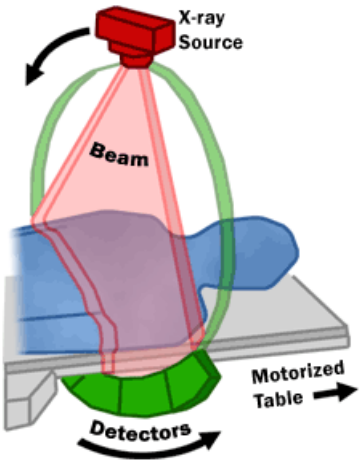


Imagen 3-7 *Tomografía computada* [12].

3.2.2.2 Pasivos

No emiten ningún tipo de radiación por sí mismos, en lugar de esto detectan la radiación reflejada en el ambiente. Algunos utilizan luz infrarroja, suelen ser métodos baratos debido a que no necesitan grandes prestaciones de hardware. Algunos tipos son:

- **Estereoscópico**: Utiliza dos cámaras de vídeo, ligeramente separadas que miran la misma escena, analizando la diferencia que hay entre la información de las dos cámaras se determina la distancia de cada punto de la imagen. Es el mismo principio que utiliza la vista humana.
- **Silueta**: La información se toma sacando una sucesión de imágenes alrededor de un objeto contra un fondo contrastado. No es una técnica muy precisa.

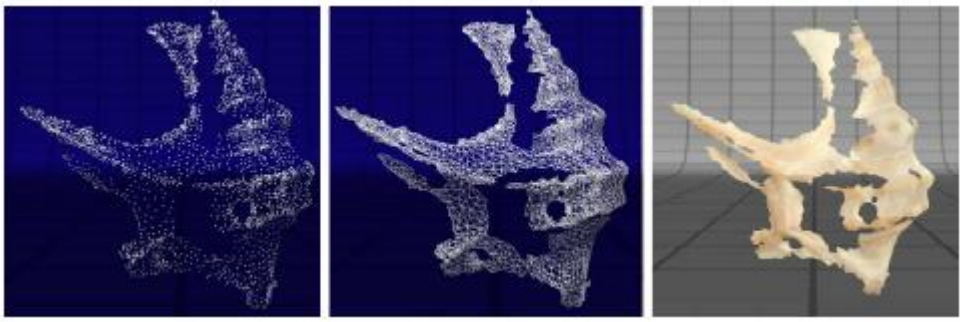


Imagen 3-8 *Nube de puntos, malla generada a partir de ella y malla texturizada* [13].

3.2.3 Kinect

Dado que el Kinect es el escáner 3D utilizado en la realización del proyecto, en esta sección se detalla cómo está constituido y cómo es su funcionamiento.

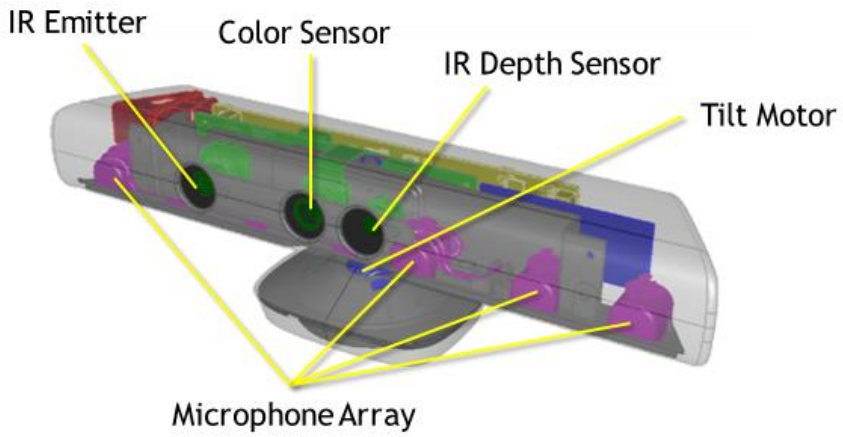


Imagen 3-9 *Componentes del Kinect* [14].

El Kinect se ha convertido en uno de los escáneres 3D más populares a nivel mundial, debido a su bajo costo, confiabilidad y su capacidad para reconstruir escenas 3D, reconocimiento de personas y objetos.

Un Kinect está compuesto por un proyector IR (infrarrojo) y una cámara IR las cuales son utilizadas para generar nubes de puntos en el espacio. Funciona como una cámara de profundidad unido a una cámara de color (RGB), las que se utilizan para reconocer contenidos en la imagen y texturizar las nubes de puntos. Genera tres salidas, una imagen IR, una imagen RGB y una imagen de profundidad. Por otro lado posee un conjunto de cuatro micrófonos que permiten capturar el audio en varias direcciones y además posee un acelerómetro que permite saber en qué posición está el Kinect [15].

El emisor de infrarrojos emite un patrón de puntos y el sensor de profundidad es el que lee la distorsión de los mismos en la escena. Los datos obtenidos son convertidos en medidas de profundidad de la distancia entre un objeto y el sensor.

Kinect, como se explicó anteriormente, es un sensor que emite un rayo de luz infrarroja para obtener la información de profundidad. Como tal tiene una serie de problemas que se encuentran asociadas a esta manera de recabar la información. En el caso en que el objeto tenga transparencias, los rayos infrarrojos lo atraviesan, retornando la información del objeto que se encuentra por detrás. Para el caso en que el objeto tenga reflejos, la información retornada también puede ser distorsionada por los mismos. Otro punto a tener en cuenta es que si dos Kinect se encuentran escaneando la misma escena en la misma dirección, los rayos infrarrojos que emiten generan una interferencia, la cual provoca que los datos se pierdan o sean devueltos con mucho ruido [14].

3.2.3.1 Interferencia entre múltiples cámaras de luz estructurada.

La interferencia se produce cuando dos o más escáneres de luz estructurada, como puede ser un Kinect, se encuentran enfocando la misma área de una escena. Estos están basados en una cámara de infrarrojos, la luz infrarroja pasa a través de un elemento óptico de difracción que proyecta un patrón de puntos en la escena. La disparidad entre el patrón de iluminación conocido y los puntos observados se utiliza para calcular la profundidad a través del campo de visión de la cámara.

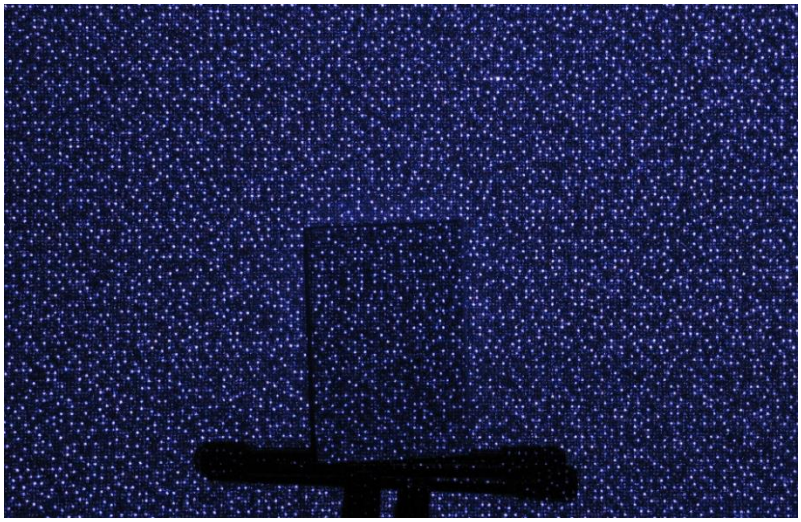


Imagen 3-10 *Patrón de iluminación IR del Kinect* [16].

El problema se produce entonces, cuando ambos escáneres de luz estructurada proyectan cada uno su patrón sobre la escena; la cámara no es capaz de distinguir fácilmente cuál de los patrones corresponde a cada una, esto produce que el resultado tengo mucho ruido.

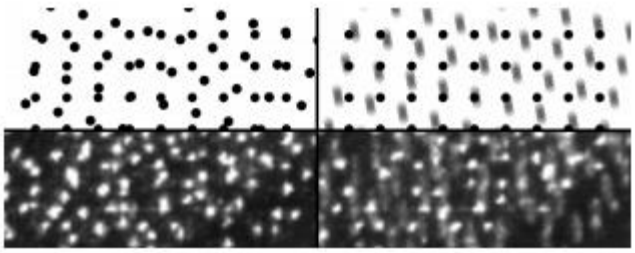


Imagen 3-11 *Las imágenes de la izquierda muestran dos patrones superpuestos, las de la derecha muestra cómo se genera la desambiguación de los patrones al introducir una vibración en los dispositivos* [17].

La solución para este problema se detalla en el paper "*Shake'n'Sense: Reducing Interference for Overlapping Structured Light Depth Cameras*" [18]. Ésta consiste en hacer vibrar sutilmente los dispositivos para lograr que se distorsione el patrón proyectado de uno con respecto al

otro. Como en el caso de los Kinect, cada uno tiene la cámara y el sensor infrarrojo en el mismo dispositivo, cuando a este se le induce una vibración, tanto la cámara como el sensor IR se mueven en armonía, lo que permite calcular la distancia de manera correcta. En los otros dispositivos sucede lo mismo, teniendo que el único patrón que cada uno logra ver con claridad es el que él mismo genera, de esta manera se logra disminuir notoriamente el ruido provocado por la interferencia de múltiples sensores de luz estructurada.

3.2.4 Datos generados

La mayoría de los escáneres 3D generan nubes de puntos, las cuales pueden ser utilizadas directamente. Sin embargo, generalmente se utilizan reconstrucciones poligonales de las escenas [13].

Las **mallas de polígonos** modelan las superficies curvas con pequeñas caras planas, por ejemplo, una esfera sería representada como algo similar a una bola de espejos. Por intermedio de diversos algoritmos, dada una nube de puntos, se logra encontrar la secuencia de vértices adyacentes entre sí que generan la malla.

Los **modelos de superficies** consisten en representar las curvas como funciones matemáticas, en este caso una esfera sería representada como una esfera matemática.

3.3 Algoritmos de reconstrucción de superficies

Los algoritmos de reconstrucción de superficies son problemas bien estudiados en computación gráfica, reciben como entrada un conjunto de puntos que describen la forma o topología de un objeto en tres dimensiones. Estos algoritmos convierten dichos puntos en modelos 3D. En particular durante el desarrollo del proyecto fue necesario convertir las nubes de puntos obtenidas de los escáneres de profundidad en mallas 3D, para esto es que se utilizaron algunos de los siguientes algoritmos.

3.3.1 Método *Tangent Plane Estimation*

Es un método introducido por Boissonnat en 1984 [19]. Selecciona los vecinos de un punto en una nube de puntos, luego proyecta estos puntos en un plano tangente a una estimación de la superficie en ese punto y computa la triangulación de Delaunay en el área proyectada. Cuando la nube es suficientemente densa y lisa, los vecinos del punto seleccionado no deberían desviarse mucho de las proyecciones en el plano tangente, por lo tanto la superficie de la nube de puntos se puede estimar [19].

3.3.1.1 Triangulación de Delaunay

Una triangulación es una subdivisión de un área en triángulos. La triangulación de Delaunay [20] para un conjunto de puntos en un plano D(P), es aquella en la que ningún punto se encuentra dentro de la circunferencia circunscrita de cualquier triángulo en D(P).

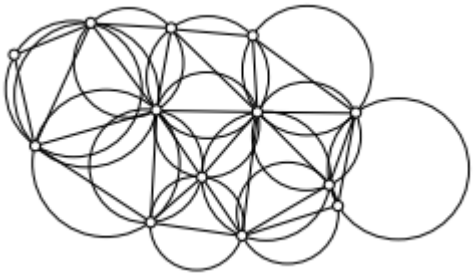


Imagen 3-12 *Triángulos circunscriptos en la triangulación de Delaunay* [20].

Esta técnica maximiza el ángulo mínimo de todos los ángulos de los triángulos en la triangulación, esto evita que haya triángulos “muy finos”.

Para un conjunto de puntos P en un espacio Euclidiano n-dimensional, la triangulación de Delaunay D(P) consiste en que ningún punto en P está dentro de la hiperesfera circunscrita de ningún simplex (tetraedro n-dimensional).

3.3.2 Método *Alpha Shape*

Este método utiliza la triangulación de Delaunay y un radio dado por el usuario de valor alfa. Alpha Shape [19] es un concepto geométrico bien definido que consiste en la generalización de la envolvente convexa y la triangulación de Delaunay para un conjunto de puntos. Partiendo de la triangulación de Delaunay se pueden obtener una familia de formas donde alfa es el coeficiente que controla el nivel de detalle de la misma.

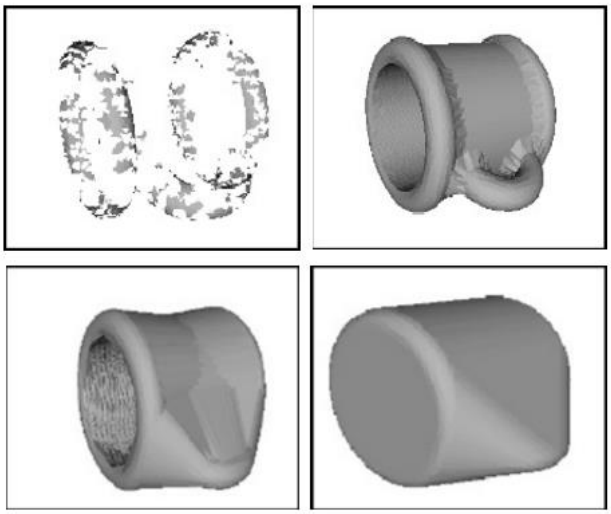


Imagen 3-13 *Cambio en la reconstrucción del algoritmo Alpha Shape variando alfa de 0 a 10* [19].

Alfa toma un valor entre cero e infinito, cuando el valor es cero el Alfa Shape de un punto es el punto en sí mismo, mientras que a medida que alfa va creciendo se va transformando en un envolvente convexo cada vez más grande. El conjunto de todos los Alpha Shapes posibles para un punto es denominado familia de Alpha Shapes. El orden de este algoritmo es $O(n^2)$, donde *n* es la cantidad de puntos de la nube de puntos.

3.3.3 Método "Ball Pivoting"

El algoritmo Ball Pivoting (BPA por sus siglas en inglés) genera una malla de triángulos a partir de la interpolación de una nube de puntos. El principio del BPA es que para que tres puntos formen un triángulo, una bola de radio *r* especificado por el usuario debe tocarlos sin incluir ningún otro punto. Empezando por un triángulo como semilla, la bola va pivoteando una arista hasta que toca otro punto para así formar otro triángulo. Este proceso continúa hasta que todas las aristas alcanzables han sido probadas, luego se empieza con otro triángulo como semilla hasta que todos los puntos hayan sido considerados.

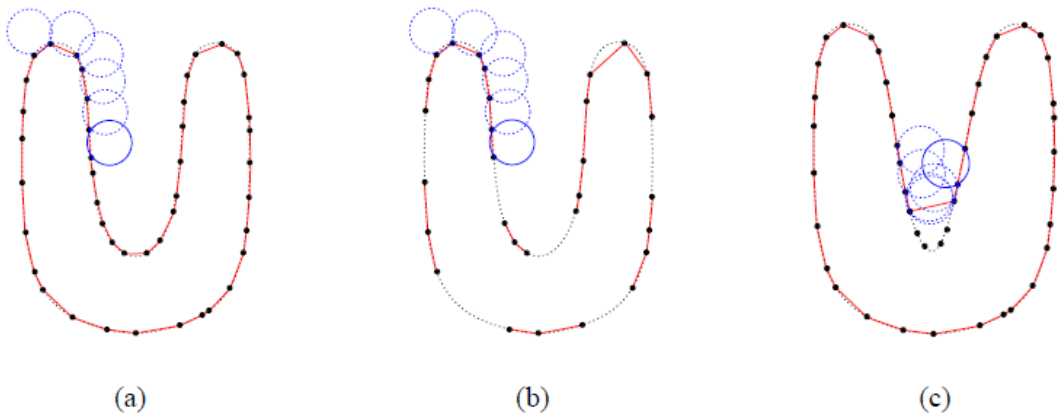


Imagen 3-14 *(a) Se muestra como el círculo va recorriendo los lados de la nube para unir sus puntos. (b) Se muestra qué pasa si hay huecos de una distancia mayor al diámetro de la circunferencia, el resultado es que quedan grupos de puntos disjuntos. (c) Muestra lo que pasa si hay ángulos muy pronunciados en la nube de puntos, como se observa hay puntos que quedan aislados debido a que la circunferencia tiene un diámetro mayor a la distancia que hay entre dos puntos de los lados del mismo* [21].

El ejemplo anterior es para dos dimensiones, se puede hacer fácilmente una analogía con una esfera que va formando una malla a medida que va uniendo los puntos. Ambos problemas mencionados se mantienen para el caso de tres dimensiones.
Al no tener que realizar la triangulación de Delaunay, Ball Pivoting es más rápido que los algoritmos que se basan en Delaunay para generar la malla, pero presenta el problema de dejar huecos cuando la nube de puntos no es suficiente densa.

3.3.4 Métodos de interpolación de superficies

Los métodos de interpolación de superficies utilizan funciones matemáticas específicas para interpolar las nubes de puntos. El resultado es una malla de triángulos obtenida de la función

de interpolación. Comparado con otros algoritmos, el método de interpolación puede ser utilizado para reducir el ruido en la nube de puntos, lo que se conoce como suavizado o *smooth* en inglés, o disminuir la densidad de la nube para resolver el problema de almacenamiento, conocido como *downsampling*.

Algunas funciones conocidas:

- Radial Basis Functions (RBF) [22].
- Moving Least Square (MLS) [23].
- Adaptive Moving Least Square (AMLS) [24].

3.3.5 Comparación de algoritmos

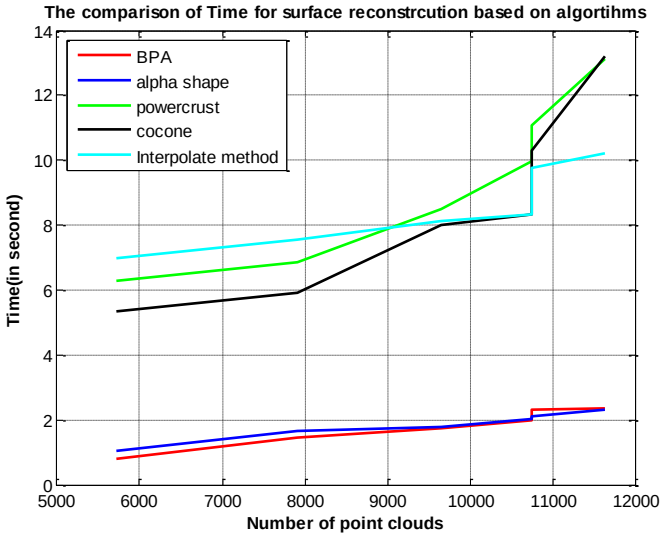


Imagen 3-15 *Grafico comparativo de algoritmos de generación de superficies* [19].

En el grafico superior se puede ver el tiempo que demoran los diferentes algoritmos en reconstruir una superficie, dependiendo del número de puntos que contenga la nube. Se observa que la relación entre estos datos es directamente proporcional. Los métodos basados en Delaunay o Voronoi como *Alpha Shape*, *Powercrust* y *Cocone*, son visiblemente más lentos dado que tienen mayor cantidad de cálculos por estar basados en las técnicas antes mencionadas. Los métodos de interpolación en cambio, trabajan con matrices que aumentan de tamaño en la medida que lo hace la cantidad de puntos, ocupando mayor espacio en memoria y aumentando el tiempo de procesamiento. Dado que Ball Pivoting no requiere de cálculos de Delaunay o Veroni, este algoritmo es más rápido que los otros [19].

3.3.6 El problema con los algoritmos de reconstrucción de superficies

La elección de un algoritmo para reconstruir una superficie es un problema común en la computación gráfica. El método de la estimación de tangente puede fallar reconstruyendo

superficies cuando la nube de puntos tiene ruido [19]. En tiempo real, el ruido no se puede evitar dado que las medidas siempre contienen errores, por lo tanto, este método estimará mal el plano tangente.

Los algoritmos basados en Delaunay como Alpha Shape, vienen acompañados de garantías teóricas pero estas solamente se dan bajo ciertas condiciones, si estas condiciones no se cumplen el comportamiento es indeterminado [19]. La densidad de la nube es una de estas condiciones, por lo cual, garantizar una densidad alta en tiempo real es muy difícil.

El tiempo que insume generar la malla es otro factor a considerar, escanear en tiempo real siempre produce un gran número de puntos. El cálculo de la triangulación de Delaunay para estos casos puede ser muy costoso en tiempo [19].

3.4 Malla de Polígonos

Una malla de polígonos consiste en un conjunto de vértices, un conjunto de aristas y un conjunto de caras. Los vértices se conectan entre sí por intermedio de aristas, formando polígonos que generalmente son triángulos o cuadriláteros, pero también pueden ser figuras geométricas planas con mayor número de lados [25].

Los distintos tipos de archivos utilizados para almacenar una malla tienen que permitir obtener no solo los diferentes vértices, sino también la manera en la que ellos se interconectan para formar los polígonos de la malla. A una malla poligonal se le pueden aplicar operaciones como: suavizados, simplificaciones y operaciones booleanas lógicas [26].

Los vértices pueden estar acompañados de otra información, como color, una normal y una coordenada de textura [27]. La mayoría del hardware para dibujado soporta caras de tres o cuatro lados, con lo que los polígonos son representados por múltiples caras. Desde un punto de vista matemático, las mallas de polígonos son consideradas como grafos sin dirección, con propiedades topológicas, de forma y geometría.

3.4.1 Estructura de los datos

Las mallas de polígonos pueden estar representadas de diferentes maneras, usando diversos métodos para almacenar los vértices, aristas y caras. Algunas de las formas son las siguientes:

- **Vértices compartidos**: Esta estructura consiste en un arreglo de vértices y un arreglo de polígonos con referencias a los vértices. Permite encontrar los vértices de una cara en orden uno [28].

Vértices	(X,Y,Z)
1	(0,0,0)
2	(0,0,1)
3	(0,1,0)
4	(0,1,1)
5	(1,0,0)
6	(1,0,1)
7	(1,1,1)
...	...

Tabla 3-1 *Arreglo de vértices.*

Vértices	Polígonos
1 4 3	(0,0,0)(0,1,1)(0,1,0)
1 2 4	(0,0,0)(0,0,1)(0,1,1)
2 7 4	(0,0,1)(1,1,1)(0,1,1)
...	...

Tabla 3-2 *Arreglo de polígonos.*

- **Arreglo de polígonos**: Representan un arreglo de polígonos con las coordenadas de los vértices que lo componen. Es una representación muy simple con mucha redundancia, dado que almacena las coordenadas de los vértices tantas veces como en polígonos se encuentren [28].

Polígonos
(0,0,0)(0,1,1)(0,1,0)
(0,0,0)(0,0,1)(0,1,1)
(0,0,1)(1,1,1)(0,1,1)
...

Tabla 3-3 *Arreglo de polígonos.*

- **Winged-edge**: El centro de esta estructura es la arista. Los vértices tienen referencias a las aristas que componen, mientras que los polígonos tienen referencias a las aristas que los componen. Esta representación tiene como ventaja que es fácil de modificar si algún vértice cambia y tiene como desventaja que requiere mayor espacio de almacenamiento que las dos anteriores [29].

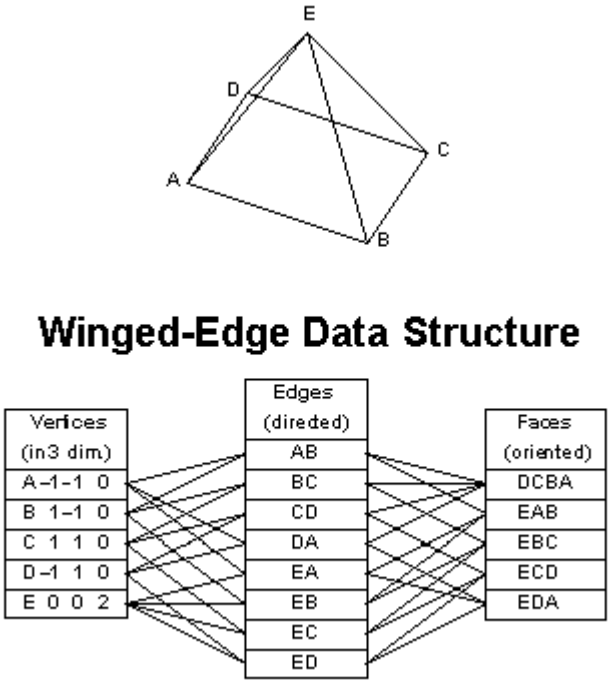


Imagen 3-16 *Estructura de Winged-Edge* [30].

3.4.1.1 Formatos para almacenar mallas

Los formatos existentes son utilizados dependiendo de la manera en la que se quiera trabajar con la malla, los datos que se quieran almacenar de la misma y el tamaño de archivo que se desea tener, entre otras consideraciones. Algunos formatos de ejemplo son los siguientes:

- **RAW**: Es un formato que se encuentra únicamente en ASCII. Las líneas del archivo contienen información de los tres vértices necesarios para formar cada triángulo de la malla, de la forma X1 Y1 Z1 X2 Y2 Z2 X3 Y3 Z3. Dado que no contiene indexación los archivos tienen un gran tamaño en disco. Es del tipo arreglos de polígonos [31].
- **PLY**: Es un formato que se encuentra tanto en ASCII como en binario, consiste de un cabezal con la información general del archivo, y como mínimo una lista de vértices y una lista de caras con referencias a los vértices que la componen. Es del tipo vértices compartidos [32].
- **OBJ**: Formato de texto plano en ASCII donde se representa una lista de vértices y una lista de caras con referencias a los vértices, opcionalmente por cada vértice se puede poner normales, coordenadas de texturas y otras propiedades. Es del tipo vértices compartidos [33].
- **3DS**: Es un formato patentado por Autodesk, permite ingresar una gran cantidad de información a la malla o modelo 3D, como material, color ambiente, color difuso, color especular, textura, reflectividad y demás [27].
- **DAE**: también conocido como COLLADA, es un formato basado en XML, donde al igual que en el formato 3DS se pueden especificar varias propiedades a las mallas [34].

3.4.2 Malla de triángulos

Las mallas de triángulos son un tipo particular de malla de polígonos, donde como su nombre lo indica, los polígonos son triángulos. Es más eficiente operar sobre triángulos que están agrupados en una malla que sobre el mismo número de triángulos independientes [35]. Esto es típicamente porque se hacen operaciones sobre los vértices de las esquinas de los triángulos, que son conocidos como vértices compartidos. En una malla grande, un vértice puede estar compartido por un número arbitrario de triángulos, pudiendo ser procesado una sola vez para todos los triángulos que lo comparten, fraccionando el trabajo realizado y obteniendo el mismo resultado [35].

A continuación, se presentan algunas representaciones soportadas por los frameworks encargados del dibujado de gráficos 2D y 3D como OpenGL [36] y DirectX [37].

- **Tiras de triángulos**: En una tira de triángulos donde cada triángulo comparte una arista con un vecino y otra arista con el siguiente.

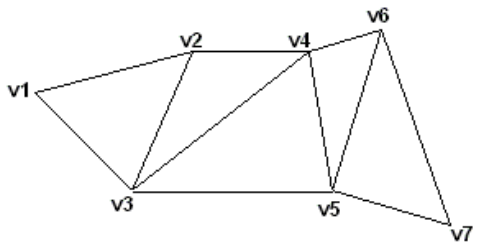


Imagen 3-17 *Ejemplo de representación como tiras de triángulos* [37].

- **Abanico de triángulos**: Otra manera de representación es con un abanico en donde un conjunto de triángulos comparte un vértice central.

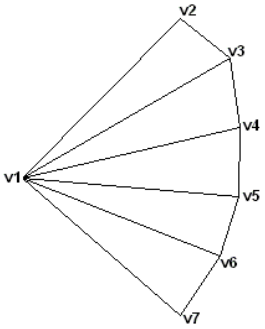


Imagen 3-18 *Ejemplo de representación como abanico de triángulos* [38].

Tanto con este método como con el anterior, el conjunto de vértices necesario para procesar una cantidad N de triángulos es de N+2, como se puede apreciar en las imágenes [37].

- **Arreglos Indexados**: Es una manera eficiente de representar cualquier conjunto arbitrario de triángulos. Una malla es representada con dos arreglos separados, un arreglo que contiene los vértices y otro arreglo que contiene el conjunto de tres índices al arreglo de vértices que componen cada triángulo [39].

3.5 Texturización

La texturización surge de la necesidad de dar realismo a los modelos 3D. Para lograr un mayor grado de realismo es necesaria mayor complejidad o por lo menos dar la apariencia de tenerla. El mapeado de texturas es una manera eficiente de dar la apariencia de complejidad sin tener que modelar y dibujar todos los detalles 3D de una superficie [40].

Una textura puede ser definida popularmente como las características o apariencias de una tela dadas por la delgadez de sus hilos. También es conocida como la apariencia, consistencia o tacto de una superficie o sustancia [41]. En el ámbito de la computación gráfica se puede definir como una imagen utilizada para mostrar las características de una superficie [42]. Algunos ejemplos de textura son:

- **Imágenes de repetición**, es un patrón detallado que se repite varias veces sin que notemos que se está repitiendo, debido a que las uniones coinciden entre sí para dar una sensación de unidad.

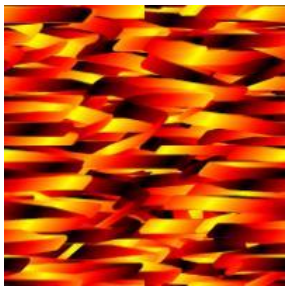


Imagen 3-19 *Textura de repetición* [43].

- **Textura para un modelo 3D**, es una textura creada para un modelo 3D particular, donde cada cara del modelo es mapeada con un área de la textura.

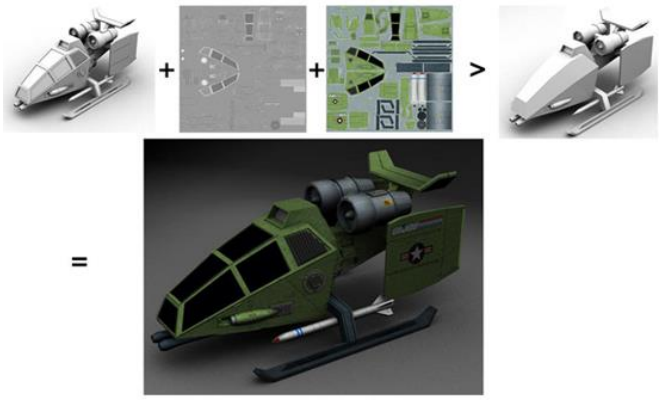


Imagen 3-20 *Textura de un objeto 3D especifico* [44].

- Una imagen tomada del mundo real con una cámara RGB también puede ser una textura. En el desarrollo de este proyecto se utilizan estas imágenes para texturizar las

mallas de triángulos generadas a partir de los datos obtenidos de los sensores de profundidad.

3.5.1 Técnicas de texturización

Algunas técnicas utilizadas para texturizar un objeto son:

- **Mapeado de texturas**: es el proceso de proyectar una textura (imagen 2D) en la superficie de un objeto 3D. Puede ser utilizado para manipular una variedad de características de la superficie como el color, transparencia, especularidad, incandescencia, etc. [45]. Consiste en el mapeado de una función en una superficie 3D. El dominio de la función puede ser de una, dos o tres dimensiones y puede estar representado por un arreglo o por una función matemática.
  Para realizar la correspondencia de un objeto 3D y una imagen 2D es que se crean las coordenadas de mapeado de textura (U, V), permitiendo que un objeto 3D sea pintado con los colores de una imagen.

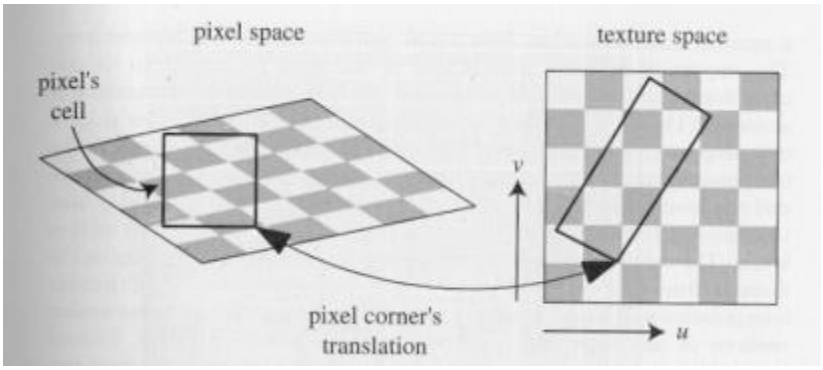


Imagen 3-21 *Correspondencia entre la textura con el objeto utilizando las coordenadas de mapeado de texturas (U,V).*

- **Mapeado en dos partes**: esta técnica es utilizada para crear coordenadas de mapeado de texturas (U, V) para polígonos. Consiste de dos pasos:
  - Paso 1: Una textura 2D es mapeada en la superficie de una forma geométrica 3D simple como un plano, un cubo, un cilindro o una esfera.
  - Paso 2: La textura 3D creada en el paso 1 es mapeada en la superficie de un objeto.

  Los métodos más utilizados comúnmente son proyecciones planas, cúbicas, cilíndricas o polares. De esta manera se puede dividir a un objeto en múltiples partes para seleccionar y aplicar el método más apropiado para cada una y permitir que todas las partes compartan un mapeado UV.

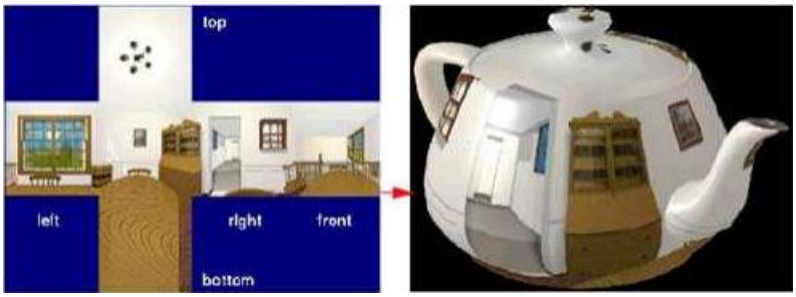


Imagen 3-22 *Mapeado Cubico* [46].

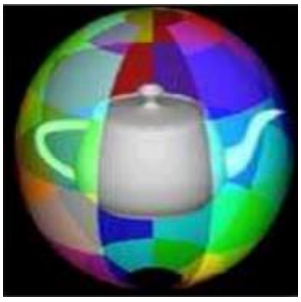


Imagen 3-23 *Mapeado Esférico* **[46]**.

Imagen 3-24 *Mapeado Cilíndrico* **[46]**.

- **Mip-mapping**: Se utiliza cuando una gran área de textura es mapeada a una pequeña área de la imagen, o lo que es lo mismo, cuando un gran número de texeles (es la unidad mínima de textura aplicada a una superficie) son mapeados a un simple pixel de la pantalla. Si se utiliza el color de un simple texel para pintar un pixel entonces aparece el aliasing y como resultado el mapeado de textura no luce bien. Para evitar esto es necesario promediar el color de los texeles que se van a aplicar. Por lo cual determinar el color de un pixel, requiere de mucho tiempo computacional [45].

  El mip-mapping es una técnica que pre-filtra las imágenes, creando varias copias de la textura, todas promediando los texeles y generando nuevas texturas de inferior resolución. Cada textura es generada con exactamente la mitad de la resolución de la anterior. A la hora del dibujado se elige la imagen más apropiada.

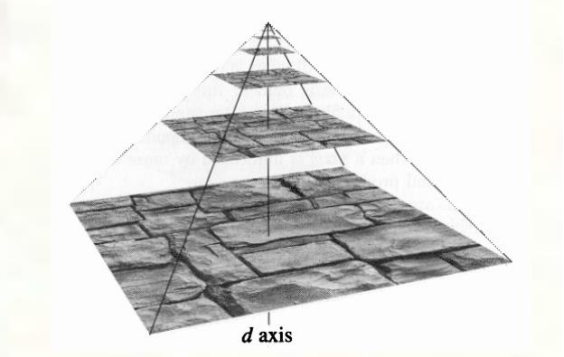


Imagen 3-25 *Técnica de Mip-mapping aplicada a una imagen.*

- **Projective texture**: El mapeado de textura proyectada permite que una imagen de textura sea proyectada en la escena como si se tratara de un proyector sobre una superficie. En el siguiente punto se explica con mayor profundidad este método.

3.5.1.1 Projective Texture

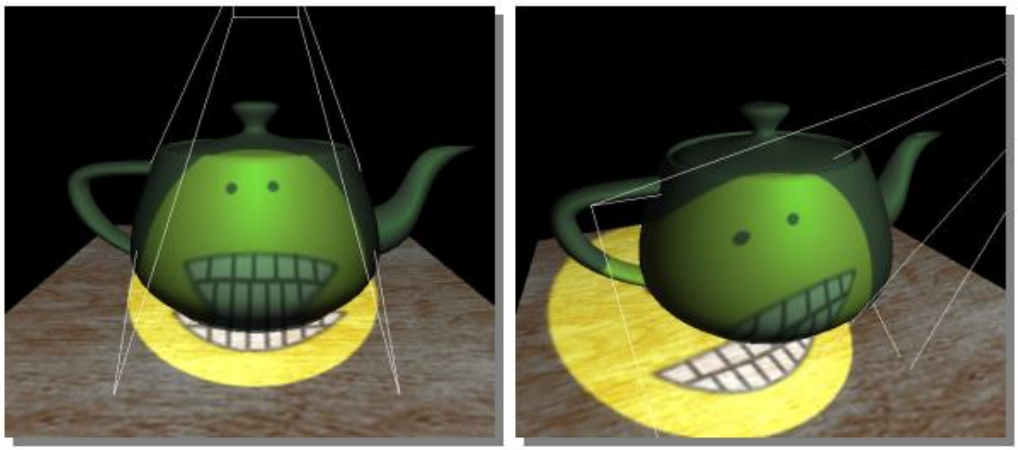


Imagen 3-26 *Imagen proyectada desde dos puntos diferentes* [47].

Se suele asociar a la texturización como la asignación explícita de coordenadas de textura a una superficie, sin embargo, esta no es la única manera de texturizar. La imagen superior muestra un ejemplo de textura proyectada sobre un objeto, como se puede observar no existe la noción de que un objeto bloquea a otro, por lo tanto, se texturizan todos los objetos en el frustrum del proyector.

Una textura proyectada es una textura como cualquier otra, donde la única diferencia radica en la forma en la que es aplicada a la geometría. Como el texturizado convencional, este método requiere que cada triángulo en la escena tenga las coordenadas de textura apropiadas, si se obtienen las coordenadas correctamente el resultado va a hacer que la parte adecuada de la "luz del proyector" texturice cada triángulo.

Las transformaciones del *pipeline* gráfico hacen que los triángulos se dibujen en el área de visualización. Si se piensa en el área de visualización como una textura, entonces lo que resulta es que cada vértice se mapea con un texel de la imagen. Por ejemplo, en lugar de mapear los vertices sobre una pantalla, son mapeados sobre una imagen que actua como una textura. Este es el concepto básico para entender cómo es que funciona este método de texturización.

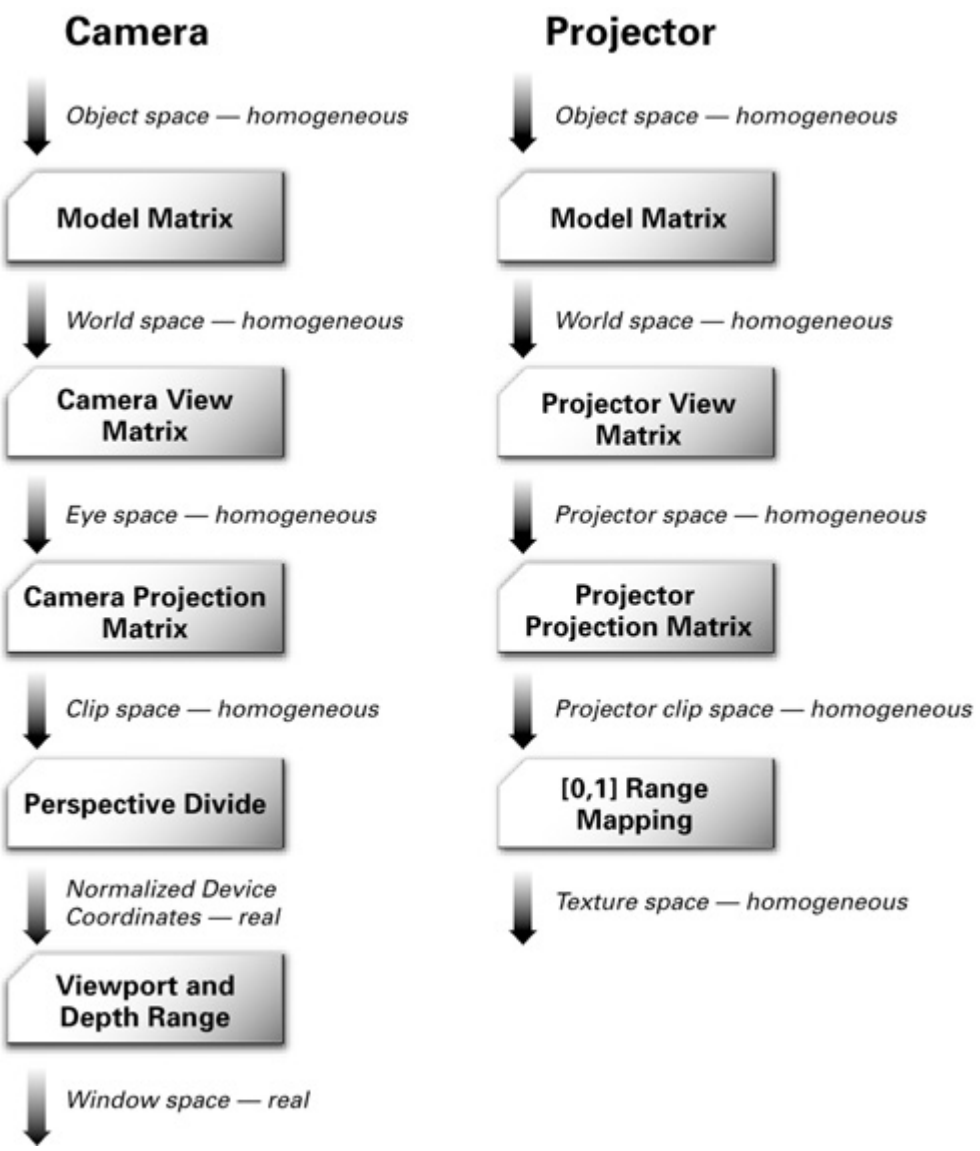


Diagrama 3-1 *Comparativa entre pipeline gráfico y pipeline de projective texture* [48].

La técnica de proyección de texturas tiene dos problemas principales:

- El primero de ellos es que no realiza un chequeo de los objetos que obstruyen la proyección. La textura es aplicada a todos los triángulos que se encuentran en el frustum de la luz del proyector. Esto no respeta la noción intuitiva de que un triángulo le hace sombra a otro, como sí ocurre con un proyector de la vida real. Este comportamiento no resulta extraño considerando que el hardware simplemente transforma vértices de acuerdo a la matriz que recibe y utiliza estas transformaciones para texturizar. Dado que el hardware transforma cada triángulo, este no tiene ningún conocimiento de la relación que poseen los triángulos entre ellos. Una manera de evitar este problema es realizando una técnica de *Occlusion Culling* [49] a la hora de aplicar la proyección para quedarse únicamente con los triángulos que son visibles por el proyector.

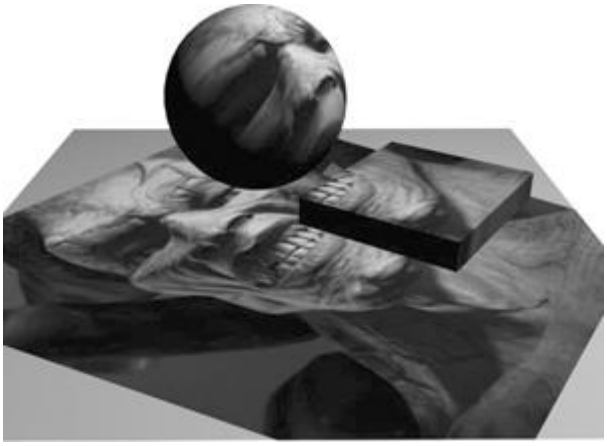


Imagen 3-27 *La imagen muestra como todos los triángulos del frustrum se texturizan* [48].

- El segundo problema es la proyección hacia atrás. Quiere decir que la proyección también aparece sobre objetos que se encuentran por detrás del proyector. Existen diferentes maneras para evitar que se realice esta proyección, usar *Culling* para hacer que se dibujen únicamente las geometrías que se encuentran en frente del proyector, otra podría ser acotar la visión del proyector con un plano que se ubique por detrás del mismo [48].

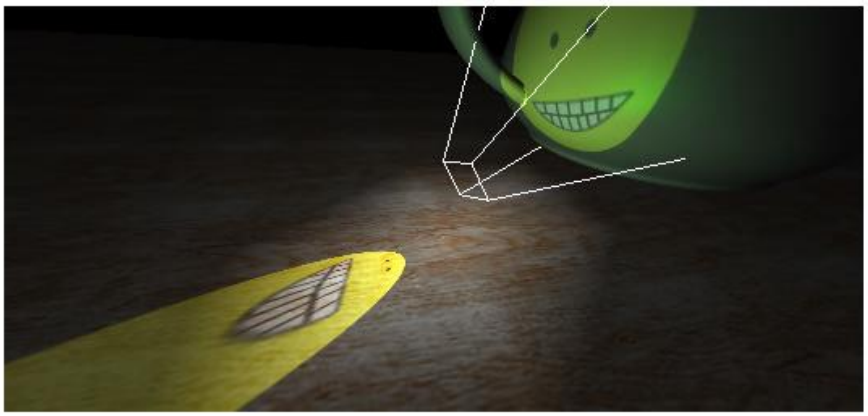


Imagen 3-28 *Ejemplo de proyección hacia atrás* [47].

3.6 Programación con GPU

La unidad de procesamiento gráfico, GPU por sus siglas en inglés, se está convirtiendo hoy en día en parte fundamental del procesamiento que se realiza en una computadora [50]. Tanto es así que en los últimos años las características de las GPUs se han ido incrementando considerablemente.

El futuro de la computación va hacia el paralelismo [50]. Si se considera la evolución que ha tenido la velocidad de las CPU se tiene clara evidencia que se ha estancado en el entorno de los 3GHz, lo que sí ha aumentado es la cantidad de núcleos por procesador, algo de lo que las GPUs siempre han sacado partido.

La GPU está diseñada para una clase particular de aplicaciones con las características que se listan a continuación.

- **Grandes requerimientos computacionales**: El dibujado en tiempo real requiere miles de pixeles por segundo, donde cada uno de ellos requiere cientos o más de operaciones. Las GPUs deben invertir enormes cantidades de cálculos computacionales para satisfacer la demanda en tiempo real de la aplicación.
- **Paralelismo**: Afortunadamente el *pipeline* gráfico está bien adecuado para el paralelismo. Las operaciones sobre vértices, son bien divisibles lo que permite estar estrechamente emparejadas con la programación en paralelo. Esto es aplicable a muchos otros dominios en la computación [50].
- **El rendimiento es más importante que la latencia**: La GPU prioriza el rendimiento por sobre la latencia en el pipeline gráfico. La vista humana opera en milésimas de segundo, mientras que los procesadores modernos lo hacen en el orden de nanosegundos. Los seis órdenes de magnitud que hay de diferencia hacen que la latencia de cualquier operación individual sea insignificante [50].

3.7 Plataformas evaluadas

Existen actualmente varias soluciones o arquitecturas que permiten atacar el problema de interactuar con cámaras RGB y de profundidad. Muchas de estas arquitecturas pertenecen a una corriente frecuentemente denominada como “Codificación Creativa”, Creative Coding en su idioma original, y las más populares son:

- openFrameworks
- Processing
- Cinder

Todas estas arquitecturas son potentes, open-source y cuentan con una comunidad de desarrolladores alrededor del mundo muy activa.

Basar el desarrollo en cualquiera de ellas, permite incorporar complementos desarrollados por la comunidad que resuelven varios de los problemas generales que se deben atacar, y ayuda a poner foco en los problemas específicos del proyecto.

Las tres arquitecturas cuentan con distintos niveles de performance y madurez, tanto de la plataforma en sí como de su comunidad. La utilización de openFrameworks para el desarrollo del sistema fue parte de los prerrequisitos del proyecto.

A continuación, se detallarán algunas características de cada plataforma:

openFrameworks: está desarrollada en C++, lo cual permite tener un mayor control sobre los recursos del sistema, y a su vez cuenta con una comunidad muy grande de desarrolladores que ha aportado varios complementos para conectar con una vasta cantidad de tecnologías [1].

Processing: está desarrollada en Java y, también es una muy buena opción para varios proyectos. Cuenta también con una gran comunidad activa de desarrolladores publicando complementos [51].

Cinder: Al igual que openFrameworks está desarrollada en C++ y fue liberada por primera vez a mediados del 2010. Si bien esta es una plataforma que también ha tenido crecimiento, al momento de comenzar el proyecto (2013), tanto el nivel de participación de la comunidad de desarrolladores como la cantidad de complementos disponible, no es tan completa como la de openFrameworks [52].

La versión de openFrameworks utilizada al comenzar el proyecto es la 0.7.3.

3.8 OpenNI

OpenNI fue liberado por primera vez en 2010 por la empresa PrimeSense y el objetivo fue crear una librería integral para estandarizar la comunicación con todos los dispositivos y metodologías de interacción natural, como ser, comandos de voz, gestos y traqueo del cuerpo humano.

En noviembre de 2013, esta organización fue adquirida por Apple y el proyecto se volvió privado, pero hasta aquel momento, OpenNI fue una librería de código abierto que permitía operar con múltiples dispositivos, entre ellos, varios sensores de profundidad, como Kinect.

Como ocurrió con muchas librerías de código abierto, la comunidad de openFrameworks no demoró en liberar un addon para trabajar con OpenNI. Tal como es la convención en openFramework, el addon se llama ofxOpenNI y se utilizó la última versión disponible.

Esta librería permite acceder a toda la información de los sensores de profundidad: nube de puntos, imagen RGB de la cámara, gestos, cuerpos en la escena, audio, entre otros. Para los objetivos de nuestro proyecto, los datos que nos resultan relevantes son las nubes de puntos y las imágenes RGB.

Luego de su privatización, la última versión del código y sus binarios es mantenida y hosteada por la empresa Structure [53].

3.9 VCG Library (Visualization and Computer Graphics Library)

Es una librería desarrollada por el ISTI (Italian National Research Council Institute) que se encuentra bajo el licenciamiento GPL (General Public License). Permite trabajar de una gran variedad de formas con mallas de triángulos, brindando diferentes tipos de funcionalidades para el procesamiento de las mismas, por ejemplo:

- Estructuras eficientes de consultas espaciales, grillas uniformes, kdtrees, grillas hasheadas, etc [5].
- Algoritmos de suavizado, Laplacian.
- Reconstrucción de superficies, Ball Pivoting.
- Optimización de coordenadas de textura.

- Herramientas y algoritmos para obtener muestras de puntos distribuidas más uniformemente, Poisson Disk.

Dentro de las aplicaciones más conocidas que se encuentran basadas en este conjunto de librerías se encuentra MeshLab [26].

3.10 PCL (Point Cloud Library)

Es un proyecto de código abierto para el procesamiento de imágenes y puntos 2D/3D. Se encuentra bajo el licenciamiento BSD license que es libre para investigación y uso comercial.

PCL tiene soporte para las plataformas más importantes que hay en la actualidad que son: Windows, Linux, MacOS, Android e iOS. Además, posee soporte para ejecutar algunas tareas con GPUs Nvidia de última generación [4].

Se encuentra dividido en pequeñas librerías de código para simplificar el desarrollo. Esta división en módulos es importante para distribuir PCL en las plataformas reduciendo tamaño y costo computacional. Entre los módulos más destacados, se encuentran:

- Visualización.
- Reconocimiento de patrones.
- Reconstrucción de superficies.
- Comunicación con dispositivos de entrada y salida.
- Seguimiento de puntos.
- Filtros.

Los datos se pueden obtener tanto por sensores de profundidad como el Kinect, cámaras de tiempo de vuelo o pueden ser generados sintéticamente por un programa. PCL implementa nativamente las interfaces OpenNI 3D, permitiendo soporte para PrimeSense, Microsoft Kinect o Asus XTionPro [4].

4 Solución Propuesta

El proyecto se compone de cuatro módulos principales: Calibrador, Cliente, Servidor y Reproductor. Los mismos, de manera muy general, se comunican entre sí de acuerdo al siguiente diagrama.

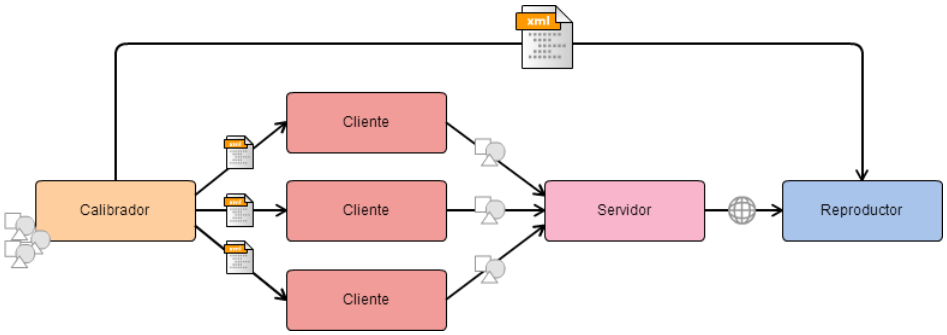


Diagrama 4-1 *Diagrama de componentes del sistema. Las flechas indican el tipo de datos que cada módulo pasa al módulo siguiente.*

En las siguientes líneas se procederá a explicar mejor cómo es que el sistema funciona y la forma en que sus módulos se comunican. El orden en el que se presentan estos módulos se relaciona con el flujo de trabajo típico que haría un usuario que utilice el sistema.

El primer paso para usar el sistema es montar la escena y la red de dispositivos necesaria para filmarla. Esto último incluye conectar cada dispositivo, ya sean cámaras 2D o 3D, a las computadoras correspondientes. Entonces, en cada una de esas computadoras se ejecutará un programa denominado DeviceDetector. El programa facilitará la obtención de una imagen por cada cámara RGB y una nube de puntos por cada cámara de profundidad. Los archivos generados servirán como entrada para el primero de los cuatro grandes módulos del sistema: el Calibrador.

El Calibrador es un programa que puede ejecutar en cualquier computadora, independientemente de si tiene o no conectado algún dispositivo. Todo lo que el programa necesita son las imágenes y las nubes de puntos obtenidas en el paso anterior. Pero además puede recibir opcionalmente una matriz de pre-calibración que contenga las transformaciones de cada una de las nubes de puntos, simplificando el proceso de calibración final. Así como matrices de calibración intrínseca para corregir las deformaciones de las imágenes.

La calibración es realizada manualmente por el usuario en dos etapas. En la primera etapa, se hace coincidir las nubes de puntos después de una serie de traslaciones y rotaciones. En la segunda etapa, una vez terminada la calibración de los sensores de profundidad, el programa generará una malla de polígonos que se utilizará como lienzo para colocar las texturas y con ello completar la calibración de las cámaras RGB. Este proceso se realizará una única vez y supondrá el mayor trabajo para el usuario.

La herramienta tiene como objetivo generar un XML que sirva como entrada para cada uno de los programas Cliente y el programa Reproductor.

Los Clientes comenzarán a grabar la información proveniente de las cámaras asignadas a cada uno, en el XML de configuración, y transmitirla al Servidor.

El Servidor almacena temporalmente los frames provenientes de los Clientes para luego procesarlos y generar por cada uno de ellos una malla. Dichas mallas, junto con sus imágenes asociadas, conforman la salida del sistema.

En una última instancia, toda la información que el Servidor genere es publicada para que pueda ser renderizada por el Reproductor. La información que se comparte es: la malla de polígonos, las imágenes de las cámaras RGB y un identificador para cada uno de esos bloques de datos.

Para lograr la renderización, el Reproductor se vale de uno de los archivos de configuración generados por el Calibrador y los datos que el Servidor comparte. El usuario puede interactuar en cualquier momento con el sistema por medio del teclado y el ratón para navegar por la escena.

En la siguiente sección se desarrollarán los aspectos más importantes referentes a la arquitectura de la solución propuesta.

4.1 Arquitectura

El sistema a construir debe poder manejar una gran carga de procesamiento, permitir un crecimiento escalable, mantener el framerate de salida más alto posible y ser flexible para adaptarse al hardware disponible. Dado ese contexto se decidió construir el sistema en una arquitectura Cliente/Servidor.

Una de las principales ventajas que ofrece esta arquitectura es que permite distribuir el trabajo pesado entre el Cliente y el Servidor, logrando así un mejor aprovechamiento de los recursos de hardware y evitando que disminuya la performance de los equipos que realizan las tareas más críticas.

Esta arquitectura permite además solucionar el problema de la escalabilidad, ya que, estableciendo pautas de configuración bien definidas, facilita incorporar una gran cantidad de Clientes al sistema con sus correspondientes cámaras, desviando los problemas de cuello de botella al hardware disponible.

La distribución de tareas permite asignar los recursos de hardware más potentes a la ejecución del Servidor, mientras que los Clientes podrán operar en equipos de menor capacidad ya que su trabajo será de una intensidad menos demandante.

Por las características planteadas, para mantener un *framerate* de salida lo más alto posible, es importante hacer foco en optimizar los principales cuellos de botella del sistema. Por un lado, la generación de las mallas supone la necesidad de manejar un alto nivel de procesamiento, mientras que el flujo de datos entre los Clientes y el Servidor, por otro lado, implican problemas de velocidad y ancho de banda. Tal como se verá más adelante, la optimización del primer punto se logró paralelizando el proceso de generación de las mallas a través de múltiples hilos que ejecutan simultáneamente. Mientras que para mejorar el flujo de la información entre el Cliente y el Servidor se puso énfasis en minimizar el tamaño de datos que

se trasmiten entre ambos. Para esto se incorporó un módulo que comprime los datos antes de enviarlos desde el Cliente y los descomprime al recibirlos en el Servidor.

En ambos casos la performance está fuertemente influida por el hardware disponible. Por esa razón se diseñó el sistema de una forma flexible, de modo que ninguna característica que resulte clave en el rendimiento esté atada a una única implementación. Tanto la cantidad de hilos de ejecución involucrados en la generación de las mallas, como el uso o no de compresión al transmitir los datos, son configurables al iniciar el sistema. Se utilizó el mismo criterio para modificar el comportamiento de otros puntos clave y de esa forma adaptarse fácilmente a las prestaciones del hardware.

4.1.1 Componentes del sistema

Los dos componentes principales del sistema son el Cliente y el Servidor. Tal como se mencionó anteriormente, el diseño escalable de la arquitectura permite que varios Clientes ejecuten y transmitan datos simultáneamente, mientras que el Servidor recibe y procesa información proveniente de múltiples orígenes.

Los Clientes tienen la responsabilidad de instanciar las cámaras RGB y sensores de profundidad configuradas en el equipo, obtener los datos provenientes de éstos y transmitirlos al Servidor. Cada Cliente puede manejar una cantidad arbitraria de estas cámaras.

La información de las cámaras recuperada por el Cliente es ordenada cronológicamente y agrupada en estructuras de datos para ser persistida en disco o enviada por red al Servidor, dependiendo de la configuración inicial.

El Servidor, por su parte, es responsable de consumir los datos que cada Cliente le envía, clasificarlos cronológicamente, unir la información en una única estructura y procesarla para obtener, a partir de las nubes de puntos, una malla de polígonos.

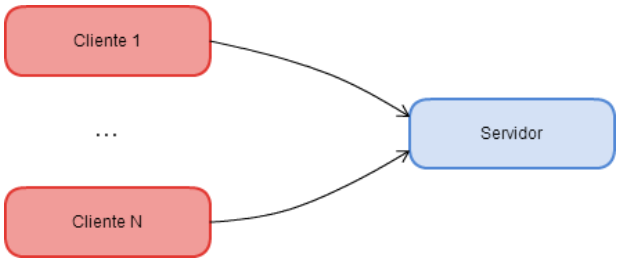


Diagrama 4-2 *Diagrama de conexión entre los Clientes y el Servidor.*

Adicionalmente el sistema cuenta con otros componentes que simplifican funciones específicas, como la identificación de las cámaras disponibles, la calibración entre ellas y la visualización de la salida del sistema. Los primeros dos componentes ejecutan del lado de los Clientes, mientras que el último complementa al Servidor consumiendo y desplegando la salida de los datos.

En el siguiente listado se describen los componentes complementarios del sistema:

- *Device Detector:* Este componente se ejecuta como paso inicial en el proceso de configuración del sistema. Tiene como finalidad detectar las cámaras conectadas al equipo y obtener de ellas una imagen RGB. En caso de tratarse de una cámara de profundidad, obtiene además su nube de puntos. Toda la información recolectada es persistida en disco para ser utilizada posteriormente por el Calibrador.
- *Calibrador:* Toma como entrada la información generada por el Device Detector permitiendo calibrar las cámaras y obtener las transformaciones relativas entre ellas. La salida del Calibrador es también impactada en disco y utilizada luego como entrada para el Cliente y el Reproductor.
- *Reproductor:* Por último, este componente se ubica al final de la cadena y consume la salida del Servidor. Permite visualizar de forma gráfica el resultado de las mallas generadas y aplica las texturas correspondientes a la escena.

El componente Device Detector reconoce todas las cámaras conectadas a una computadora dada. En general un Cliente controla todas las cámaras de una misma computadora, por lo que normalmente, se ejecuta un Device Detector por cada Cliente que se quiera incorporar. De todos modos, en algunas oportunidades puede ocurrir que se desee ejecutar dos Clientes en la misma máquina. Ese escenario también está permitido por la arquitectura, y en tal caso, con una única ejecución del componente, se detectarían las cámaras de dos o más Clientes.

En el siguiente diagrama se ilustra el orden de conexión de todos los componentes descritos anteriormente. Se representa únicamente el caso más general del Device Detector, que corresponde a una ejecución por Cliente.

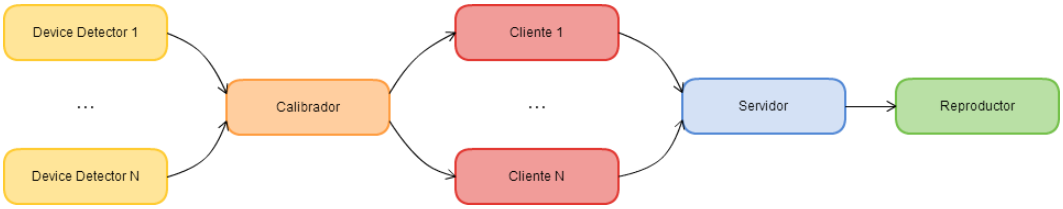


Diagrama 4-3 *Se ilustra el orden de conexión de todos los componentes del sistema. Se representa únicamente el caso más general del Device Detector, que corresponde a una ejecución por Cliente.*

4.1.2 Característica modular del sistema

Existen múltiples enfoques que permiten resolver cada uno de los puntos involucrados en el sistema a construir. Tal es el caso de la calibración de las cámaras, compresión de los datos, generación de las mallas, texturización y visualización del resultado. Si bien, como se verá más adelante, la implementación que se propone plantea una solución concreta para cada problema, estos caminos no pretenden ser los únicos o los más eficientes. Sin embargo, desde la arquitectura se plantea el sistema en su totalidad como un framework de trabajo, que determina una secuencia y flujo de información, pero cuyas piezas fundamentales pueden ser sustituidas posteriormente por mejores soluciones.

Esta característica modular del sistema permite resolver el problema global del proyecto, dejando el camino abierto a implementaciones diferentes que resuelvan de forma más eficaz cada punto e incrementen la performance general. Para lograrlo, se encapsuló los algoritmos más críticos en archivos DLL fácilmente sustituibles.

Actualmente el sistema cuenta con cinco DLLs que resuelven problemas concretos. A continuación, se detallan las funciones de cada una.

- *Compresión/Descompresión de imágenes:* Una forma de reducir el tamaño del arreglo de bytes que envía el Cliente al Servidor, es reduciendo el tamaño de las imágenes de cada cámara. Este módulo permite aplicar una transformación a las imágenes antes de enviarla y deshacerla al recibirla. La implementación actual de la DLL, comprime las imágenes utilizando el formato JPG. Este módulo se utiliza tanto en el Cliente como en el Servidor.
- *Compresión/Descompresión de arreglo de bytes:* Otro modo de reducir la cantidad de bytes que se envían es comprimiendo el arreglo antes de enviarlo y descomprimiéndolo al recibirlo. La implementación propuesta realiza la compresión utilizando la librería ZLIB y se utiliza tanto en el Cliente como en el Servidor [54].
- *Generación de mallas:* Este módulo se utiliza únicamente en el Servidor y es instanciado por cada hilo involucrado en la generación de las mallas. En la implementación de este módulo se seleccionó particularmente el algoritmo que demostró tener mejor rendimiento y resultados en las pruebas realizadas.
- *Publicación de los datos:* Actualmente el Servidor y el Reproductor se comunican a través de memoria compartida. El Servidor escribe la salida en un formato específico, y el Reproductor la consume de ese modo. Tanto la lógica de escritura y lectura como la implementación a través de memoria compartida pueden ser modificadas y sustituidas por otra solución.
- *Renderización:* Este módulo es utilizado tanto en el Calibrador como en el Reproductor. Tiene la finalidad de encapsular la lógica de visualización de la malla y aplicar las texturas sobre ellas.

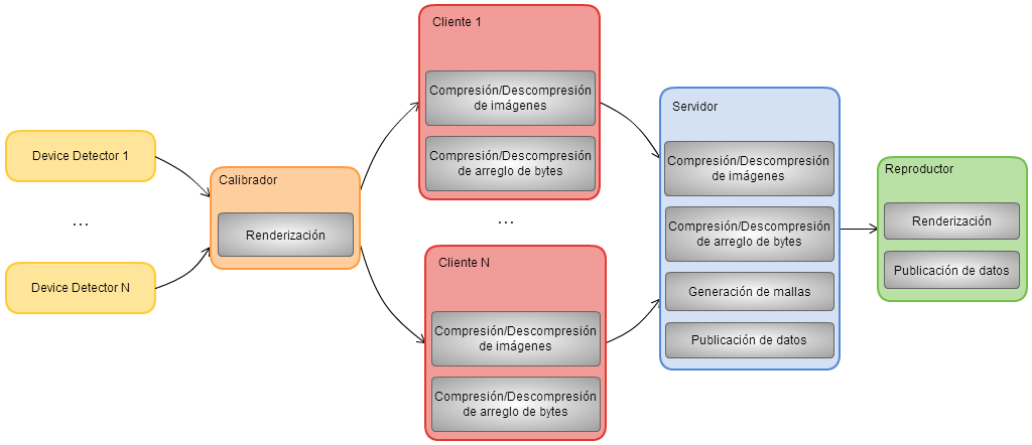


Diagrama 4-4 *El diagrama muestra cómo se relacionan los módulos entre sí y en qué lugar de la cadena del flujo de datos están ubicados.*

4.2 Implementación

4.2.1 Device Detector

Se trata de un sistema auxiliar que provee de datos al Calibrador y que funciona como el primer paso de la secuencia necesaria para generar una configuración de Cliente. Detecta los dispositivos, tanto cámaras RGB como sensores de profundidad, que están conectados al equipo donde estará alojado el Cliente.

Por limitaciones de la versión de openFrameworks utilizada en el proyecto, no es posible detectar automáticamente las cámaras RGB. Por tal motivo, la cantidad de cámaras RGB presentes en el equipo deberá ser establecida en un archivo denominado *detector_settings.xml* a través del parámetro *totalRGBDevices*.

Es importante notar que se requiere únicamente el número de cámaras RGB y no así la cantidad de sensores de profundidad. Este dato es proporcionado directamente por el addon ofxOpenNI.

Una vez iniciado, el Device Detector instancia la primera cámara RGB conectada y muestra en pantalla las imágenes recibidas. Presionando la barra espaciadora, se almacena en disco una captura de la cámara actual y se instancia la siguiente cámara. Este proceso continúa hasta abarcar todas las cámaras RGB, para comenzar luego con los sensores de profundidad. En el caso de estos sensores, no solamente se almacena la imagen RGB, sino también la nube de puntos asociada en el momento de tomar dicha imagen.

En todo momento el programa muestra en pantalla el tipo de cámara, RGB o de profundidad, el identificador del dispositivo y el total de cámaras disponibles.

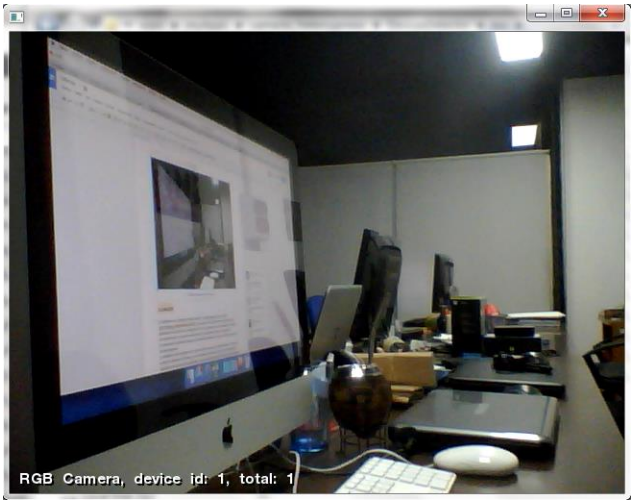


Imagen 4-1 *Interfaz del programa Device Detector durante la detección de la imagen de una cámara RGB.*

Como salida, el Device Detector genera dos subdirectorios denominados *depth_devices* y *rgb_devices*. En el primero, *depth_devices*, se almacena la información de imágenes y nubes de puntos recogida de cada sensor de profundidad. En el segundo, *rgb_devices*, se almacenan las imágenes obtenidas a partir de las cámaras RGB.

En ambos directorios, las imágenes que se almacenan están codificadas en formato JPG y las nubes de puntos en un archivo de texto plano con las coordenadas X, Y, Z de los puntos.

Esta información sirve como entrada para el Calibrador, que se explicará en la siguiente sección.

4.2.2 Calibrador

El Calibrador es un programa basado en OpenGL y openFrameworks, con el addon ofxXmlSettings y las librerías de VCG, cuyo objetivo es proporcionar una interfaz para la generación de archivos de configuración *XML*, que contienen entre otras cosas las matrices de transformación, que serán la entrada para los programas Clientes y Reproductor.

A continuación, se explicarán algunos fundamentos teóricos en los que se basó el desarrollo de este programa.

La calibración de las cámaras se utiliza para determinar el conjunto de parámetros que describen el mapeo entre la realidad de tres dimensiones y la imagen de la cámara de dos dimensiones. Estos parámetros se pueden subdividir en dos grupos:

- Los **parámetros intrínsecos** describen la geometría interna de la cámara, incluidos los coeficientes de distorsión radial y tangencial.
- Los **parámetros extrínsecos** describen la relación entre el sistema de coordenadas tridimensional de la cámara y un sistema de coordenadas dado, definido por la posición y orientación.

Los parámetros intrínsecos definen la geometría interna de la cámara. Este conjunto de parámetros incluye las longitudes focales, el punto principal (el centro de la imagen) y los coeficientes de distorsión. Los mismos son independiente de la posición y orientación de la cámara en relación al sistema de coordenadas. La calibración intrínseca se utiliza para reducir o eliminar la distorsión de la lente. La calibración extrínseca depende de los resultados y exactitud de la calibración intrínseca, para la estimación de la matriz de transformación de coordenadas [55].

Los parámetros extrínsecos definen la relación que existe entre el sistema tridimensional de coordenadas de la cámara y un sistema determinado de coordenadas del mundo, representada mediante una matriz de transformación. Ésta se compone de dos submatrices, la matriz de rotación y la matriz de traslación [56].

Puesto que la calibración extrínseca depende de la calibración intrínseca, ésta última debe hacerse previa a la primera. Después de realizada la calibración extrínseca, la posición de las cámaras, con respecto al sistema de coordenadas que se utilizó como referencia, no puede cambiar.

4.2.2.1 Calibración Intrínseca

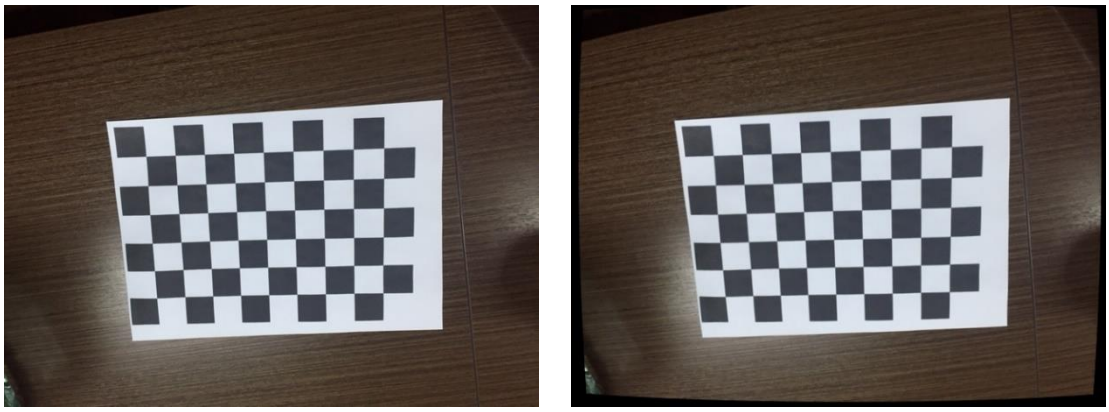


Imagen 4-2 *Imagen del antes y del después de una foto tomada por una cámara RGB cualquiera que pasó por el proceso de Calibración intrínseca.*

Luego de investigar las prestaciones de openFrameworks, se encontró que OpenCV, un addon de openFrameworks, ya cuenta con instrumentos para la calibración de cámaras.

Ahora se pasará a resumir la teoría que hay detrás del proceso de calibración que OpenCV implementa para la calibración intrínseca.

En este texto no se explicará cómo es que se derivan las ecuaciones siguientes, puesto que no es el objetivo del informe ahondar en la matemática que implica la lógica que sigue el algoritmo utilizado por OpenCV, para determinar la distorsión producto de estos parámetros intrínsecos de las cámaras [57].

Para la distorsión de las imágenes, OpenCV tiene en cuenta los factores radial y tangencial. El primero se produce por la curvatura del lente y el segundo porque el lente de la cámara no se encuentra perfectamente paralelo al plano de la imagen.

Para el factor radial, OpenCV se vale de las siguientes ecuaciones para hacer la corrección:

$$x_{corrected} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$y_{corrected} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Por otro lado, la distorsión tangencial puede ser corregida a partir de este otro par de ecuaciones:

$$x_{corrected} = x + [2p_1 xy + p_2(r^2 + 2x^2)]$$

$$y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$

Se tienen entonces dos pares de ecuaciones, de las que surgen cinco coeficientes de distorsión:

$$Distortion_{coefficients} = (k_1 \quad k_2 \quad p_1 \quad p_2 \quad k_3)$$

A su vez, para la conversión de unidades se utiliza una matriz, denominada matriz de cámara, que está compuesta por cuatro parámetros a determinar (f_x, f_y, c_x, c_y) que dependen de la resolución de la imagen.

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

En este caso se explica la presencia de w debido al uso de coordenadas homogéneas. Los parámetros f_x y f_y corresponden a la distancia focal y los parámetros c_x y c_y corresponden a los centros ópticos expresados en píxeles.

Determinar estas dos matrices es fundamental para calibrar una cámara. El cálculo de los parámetros antes mencionados se hace a partir de ecuaciones geométricas que dependen de los objetos que se utilicen para la calibración, en este caso, un tablero de ajedrez. OpenCV además soporta imágenes de otras superficies como método de entrada para las ecuaciones.

Herramienta de Calibración Intrínseca

Para esta parte del proceso de calibración se decidió utilizar una herramienta de código abierto que ya ataca este problema, GML Camera Calibration Toolbox. La herramienta utiliza OpenCV y con ello las mismas ecuaciones que se describieron antes [58].

Antes de comenzar con este proceso, que se repetirá para cada una de las cámaras, es necesario tomar una serie de fotografías del tablero de calibración, para convertirlas en valores numéricos que formarán parte de las ecuaciones mencionadas. Mientras más imágenes se utilicen, mayor será el número de ecuaciones y por ende mejor el resultado de calibración que se obtendrá.

Los pasos a seguir para hacer la calibración intrínseca con la herramienta se encuentran en una sección del Anexo.

Como resultado de este proceso, el programa despliega, la matriz de cámara y los coeficientes de distorsión. Los valores serán colocados en un archivo de texto, separados por espacios y en el siguiente orden: k_1, k_2, k_3, k_4, f_x, f_y, c_x, c_y. Estos archivos, uno por cada una de las cámaras, serán tomados opcionalmente como entrada del programa Calibrador, quien incluirá los datos en los archivos de configuración correspondientes a dichas cámaras.

4.2.2.2 Calibración Extrínseca

La calibración extrínseca constituye el punto fundamental en la unión de los datos recogidos por las diferentes cámaras, ya sea que éstas provean imágenes o matrices de puntos. El objetivo detrás de este proceso es determinar la posición de las cámaras en el espacio para poder aplicar después las transformaciones adecuadas para hacer coincidir las nubes de puntos y las texturas.

La lógica que se utilizó para atacar este punto se verá con mayor detenimiento en las siguientes dos secciones, a continuación se detallarán los conceptos teóricos que hacen esto posible.

Cada cámara del sistema se encuentra en una posición fija del espacio tridimensional con una orientación determinada. En un principio el objetivo es determinar dichas posiciones y orientaciones para un origen de coordenadas, pero la solución más adecuada es descubrir las

posiciones y orientaciones relativas de una cámara con respecto a todas las demás. Es decir, la solución que se adoptó pasó por dar una matriz que diga que la cámara B se encuentra a distancia (x, y, z) de la cámara A, y no en una posición (x', y', z') respecto a un origen arbitrario. Con ello, se necesitan n-1 matrices de transformación, puesto que la primera es la identidad, y la escena resulta ser independiente del lugar en el que se encuentra.

Esas transformaciones, que hacen coincidir la cámara A con la cámara B pueden expresarse por medio de una matriz de transformación. Es decir, que al final del proceso de calibración se obtendrá una matriz que hace coincidir la cámara A, con la cámara B; otra matriz que hace coincidir la A con la C; otra para la A con la D y así sucesivamente.

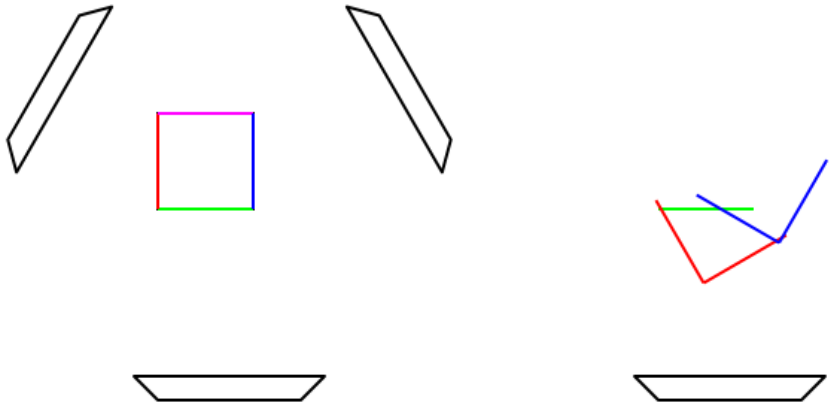


Ilustración 4-1 *Representación vista de arriba de tres cámaras y una caja. En la ilustración de la izquierda se aprecia la posición real de las cámaras y la caja. En la ilustración de la derecha, la unión de las partes que ven cada una de las cámaras. El objetivo de la calibración extrínseca, es encontrar las matrices de transformación para cada una de esas partes que reconstruyen la escena.*

Esas matrices serán utilizadas más tarde por los programas Clientes para reorientar las nubes de puntos, por el programa Servidor para unir correctamente dichas nubes, y por el programa Reproductor para aplicar con exactitud las texturas.

El Calibrador divide la calibración extrínseca en dos etapas. Por un lado, la calibración de las nubes de puntos de las cámaras de profundidad, que se denomina calibración 3D, y por otro lado la calibración de las imágenes de las cámaras RGB, que se denomina calibración 2D. El orden de los pasos no puede invertirse dado que la calibración de las cámaras RGB requiere como entrada una malla de polígonos, generada ésta a partir de la calibración de las nubes de puntos, que será utilizada como lienzo para proyectar sobre ellas las texturas.

El proceso que debe seguir el usuario para utilizar con éxito este programa se describe en la siguiente sección.

4.2.2.3 Calibración 3D

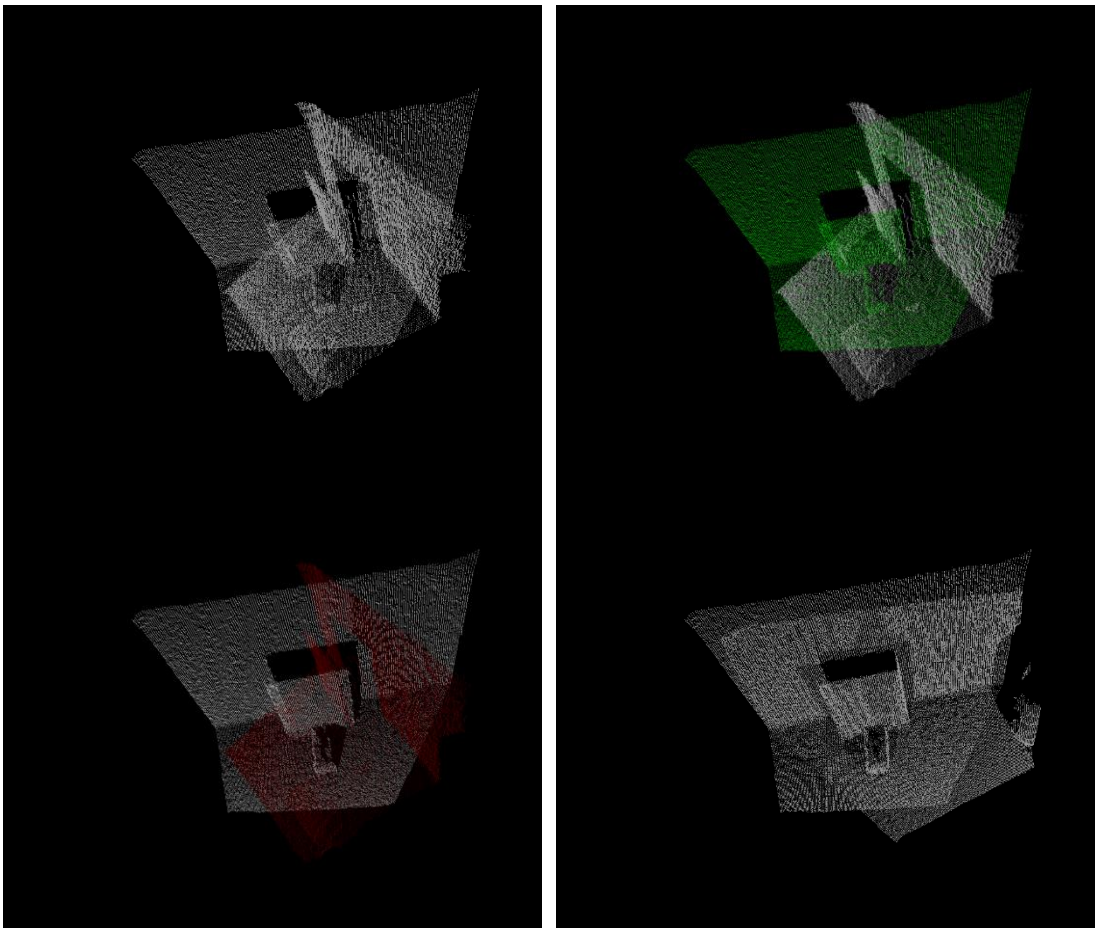


Imagen 4-3 *En la imagen de arriba a la izquierda, se muestra la unión de dos nubes de puntos tal y como son provistas por las cámaras de profundidad. En la imagen de arriba a la derecha y abajo a la izquierda, se visualizan las nubes de puntos seleccionadas para su transformación manual. En la imagen de abajo a la derecha, se muestra el resultado final de ambas nubes de puntos transformadas.*

Antes de iniciar el programa de calibración, el usuario debe colocar bajo el directorio *data* una carpeta por cada uno de los clientes del sistema. A su vez, dentro de cada directorio cliente debe existir una carpeta por cada cámara conectada a dicho Cliente. Los directorios de las cámaras pueden contener un archivo de imagen (en formatos BMP, JPG o PNG), un archivo de nube de puntos (en formato XYZ) o ambos, si es que la cámara captura tanto imágenes como matrices de profundidad.

Opcionalmente, se puede incluir una serie de archivos *TXT* con matrices de pre-calibración 3D. La explicación de esto es que el usuario puede contar con información de calibración, referente a la posición relativa entre las nubes de puntos, si es ese el caso, solo debe tener un archivo de éstos por cada nube de puntos, con la matriz correspondiente.

Existen algunos aspectos a tener en cuenta a la hora de realizar esta parte de la calibración y obtener buenos resultados.

- El primero de ellos es evitar, en la medida de lo posible, incluir en la escena objetos en movimiento durante la calibración, si bien durante el resto del proceso de grabación su presencia no supondrá ningún problema.
- El segundo aspecto, es que todas las nubes de puntos deberían de haber sido tomadas al mismo tiempo, para evitar inconsistencias a la hora de unirlas si es que en la escena existen objetos que están en movimiento.
- El tercer aspecto, es disponer las cámaras de tal manera que, al menos una parte de la nube de puntos que cada cámara captura, se encuentre comprendida en la nube de puntos de otra cámara, de esta manera se consiguen vértices en común entre las diferentes nubes de puntos, que serán utilizados como referencia para la correcta unión de las mismas.

Cuando el Calibrador se inicia, tanto los archivos de nube de puntos, en formato XYZ, como los archivos de imagen, en formato JPG, son precargados, fijando así automáticamente parámetros como la cantidad de cámaras, el tipo de cada una de ellas y la distribución de las mismas entre los diferentes clientes, todo a partir del número y tipo de archivos que se encuentren en la raíz del programa.

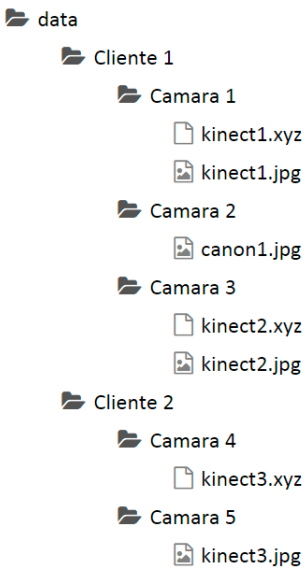


Diagrama 4-5 *En el diagrama se muestra un ejemplo de la estructura de carpetas y archivos que hay en la raíz del Calibrador. En este caso, la estructura determina que hay dos Clientes, uno con tres y otro con dos cámaras respectivamente. Donde la Cámara 1 es RGB y de profundidad, la cámara 2 es RGB, la cámara 3 es RGB y de profundidad, la cámara 4 es de profundidad y la cámara 5 es RGB.*

Al principio, el Calibrador hace el renderizado de la unión de las nubes de puntos para mostrar al usuario todos los vértices de la escena. El objetivo de éste será entonces reorganizar dichos vértices para conseguir una representación virtual que se asemeje a la realidad.

Una vez el usuario esté conforme con el resultado obtenido, puede pasar a la modalidad de calibración 2D. En este punto el programa generará dinámicamente una malla de polígonos a partir de la unión de las nubes de puntos modificadas por el usuario.

4.2.2.4 Calibración 2D

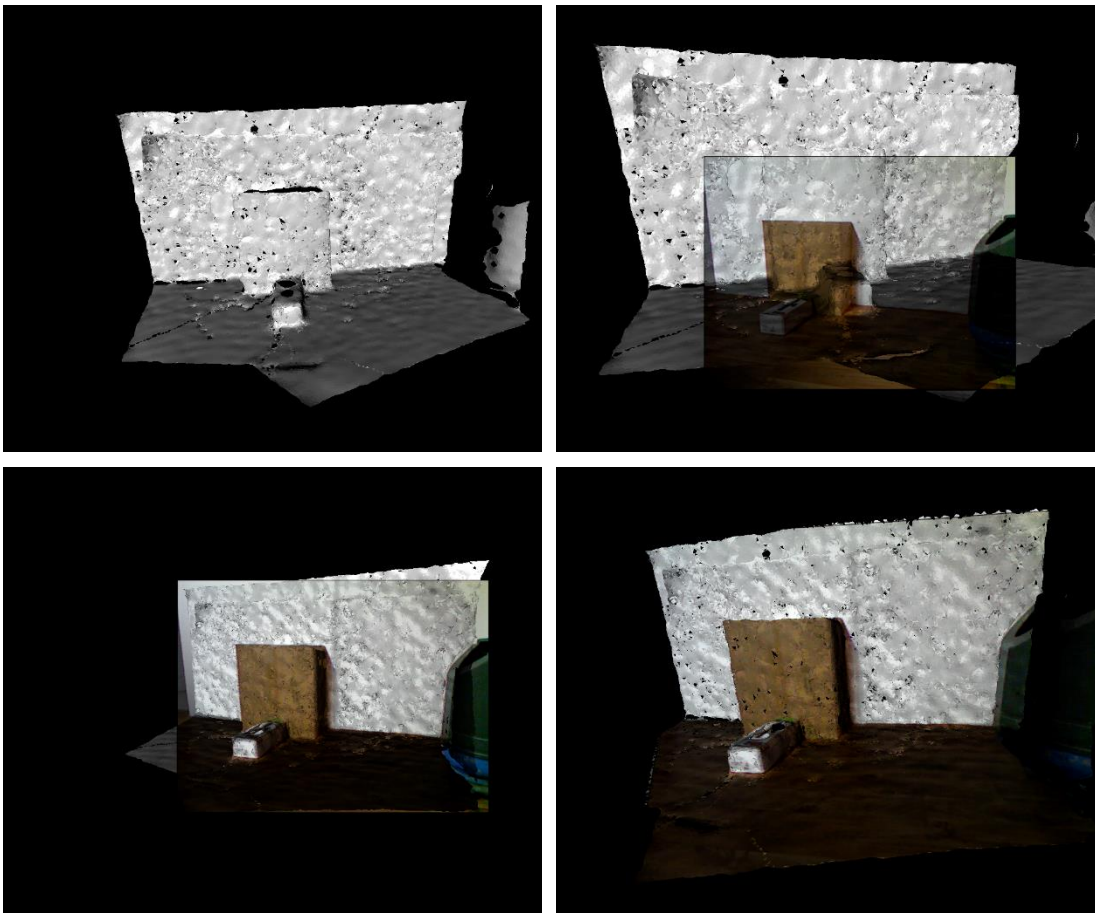


Imagen 4-4 *En la imagen de arriba a la izquierda, se muestra la malla de polígonos sin texturizar. En la imagen de arriba a la derecha y abajo a la izquierda, se visualiza la malla de polígonos junto a la textura seleccionada, que se dibuja por sobre la malla, para su calibración manual. En la imagen de abajo a la derecha, se muestra el resultado final de ambas nubes de puntos transformadas.*

Una vez que el Calibrador pasa al modo de calibración 2D, el programa genera la malla de polígonos a partir de las nubes de puntos transformadas, resultado del proceso anterior.

Lo siguiente que hará el programa será proyectar cada una de las imágenes de las cámaras RGB sobre la renderización del modelo 3D. Entonces, en este punto, el usuario solo debe ajustar la posición de dicho modelo, a través de traslaciones y rotaciones a la imagen que se está proyectando sobre la escena.

Se podría decir entonces que la cámara que se está calibrando hace las veces de un proyector colocado en una posición estática del espacio, y que lo que se mueve no es el proyector, sino que es la propia escena, para que ésta coincida con la imagen que el proyector emite. Este proceso se repite para cada una de las cámaras de forma independiente, por lo que la calibración de una cámara no afecta la calibración de las cámaras anteriores.

Una vez el usuario esté conforme con la texturización del modelo, éste puede persistir las transformaciones de las nubes de puntos y las imágenes en archivos *XML*.

4.2.2.5 Algoritmo de Calibración

Cuando comienza la ejecución del Calibrador, se establecen las configuraciones iniciales de la interfaz de usuario y se procede a buscar en el directorio local los archivos de entrada, como las nubes de puntos, las imágenes y las matrices de pre-calibración.

El próximo paso consiste en crear un juego de estructuras, para almacenar la información relacionada a la calibración de cada uno de los Clientes.

Lo siguiente que hace el Calibrador es renderizar todas las nubes de puntos, habilitando una serie de opciones de teclado y ratón para moverlas.

Durante la renderización se dibujan las nubes de puntos, tanto la que se encuentra seleccionada, identificable a través de un color único, como la unión de las demás nubes, en color gris.

Una vez que el usuario haya concluido la calibración 3D y decida pasar al modo de calibración 2D, el sistema genera una malla de polígonos, a partir de la unión de las nubes de puntos, habilitando una serie de opciones de teclado y ratón para mover dicha malla.

Las opciones que ofrece el programa en el modo de calibración 2D no difieren mucho de las del modo de calibración 3D. Sin embargo, la lógica de renderización es completamente diferente.

Al igual que en la primera etapa de la calibración, el Calibrador cuenta con dos modos para esta segunda etapa. El modo de calibración 2D y el modo de visualización final.

En el primer modo, el usuario puede trasladar y rotar la malla de polígonos para hacerla coincidir con la imagen proyectada por una de las cámaras en particular. Por otro lado, en el segundo modo, el usuario tiene la posibilidad de navegar por la escena ya texturizada.

Es durante el proceso de calibración 2D, que se determina la posición y ángulo de cada una de las cámaras RGB, así como qué caras de la malla de polígonos deben ser pintadas por una u otra textura.

4.2.3 Cliente

El Cliente es un programa basado en openFrameworks que tiene por objetivo obtener los datos de cámaras RGB y sensores de profundidad, pre-procesarlos y transmitirlos por red al Servidor.

Toma como entrada el archivo *XML* de nombre *settings.xml* generado por el Calibrador para las cámaras asociadas a éste, y por cada una, se inicia un hilo de ejecución que es responsable de instanciarlas y consumir sus datos.

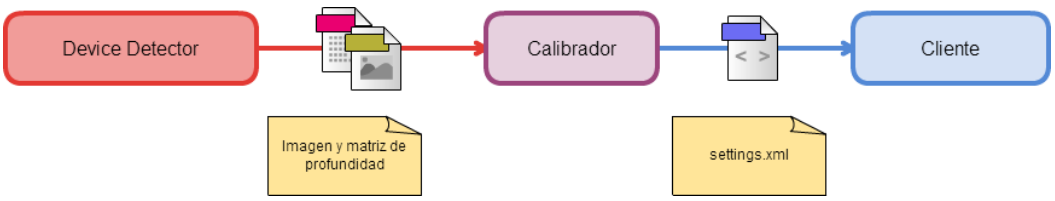


Diagrama 4-6 *Representación del pasaje de datos entre los módulos Device Detector, Calibrador y Cliente, durante el proceso de calibración.*

Como se verá más adelante, el Cliente puede ser empleado tanto para transmitir y enviar a procesar la información hacia un Servidor en tiempo real, como para persistir los datos al disco para ser utilizados posteriormente.

Utiliza internamente un conjunto de hilos de ejecución, que ayudan a dividir el problema en tres áreas específicas: lectura de datos, pre-procesamiento y trasmisión. A continuación, se detalla cómo se comunican estos hilos y cómo fluye la información entre ellos.

4.2.3.1 Clasificación de hilos por área

Se puede dividir los hilos involucrados en el trabajo del Cliente en tres categorías: lectura de datos (Thread2D, Thread3D y ThreadONI), pre-procesamiento (Grabber) y transmisión (Trasmitter). Se detallarán las categorías y los hilos que las componen.

4.2.3.2 Hilos de lectura de datos

El Cliente acepta información de entrada de tres fuentes de datos distintas: Cámaras RGB, Sensores de profundidad y archivos ONI. Existe una implementación específica para atender a cada una de estas entradas.

Se denomina Thread2D, a los hilos que consumen información específicamente de cámaras RGB. Por su parte, los hilos Thread3D, toman datos de los sensores de profundidad. Por último, los archivos ONI son leídos por hilos denominados ThreadONI.

Thread2D

En el Cliente existe un hilo Thread2D por cada cámara RGB configurada en el *XML* de entrada *settings.xml*. Cada hilo Thread2D en ejecución se encarga de instanciar una cámara y recolectar su información.

La comunicación con cada dispositivo es realizada a través del complemento de openFrameworks denominado ofVideoGrabber. Este complemento permite, a través de un identificador, solicitar e instanciar una cámara específica, indicando sus dimensiones y la velocidad esperada de actualización de fotogramas.

La velocidad máxima de iteraciones que intentará alcanzar el bucle principal de este hilo se establece a través del parámetro *FPS*.

Cada hilo Thread2D mantiene una variable compartida conteniendo el arreglo de bytes de la última imagen leída desde la cámara. Como se explicará más adelante, esta variable compartida se accede luego y su contenido será unido y transformado junto a la información proveniente de los otros hilos de lectura.

Thread3D

De forma análoga a lo descrito previamente sobre los hilos Thread2D, existe un hilo Thread3D por cada sensor de profundidad configurado en el *XML* de entrada. Cada sensor de profundidad indicado en dicho *XML* será instanciado y accedido por un único hilo Thread3D.

Los datos de cada dispositivo son obtenidos a través del addon ofxOpenNI. Tal como se explicó anteriormente, este complemento proporciona una interfaz para trabajar con la librería OpenNI que permite instanciar y comunicarse con un sensor de profundidad.

Cada hilo Thread3D mantiene en todo momento dos variables compartidas. La primera de ellas contiene el arreglo de bytes de la nube de puntos obtenida mientras que la segunda, mantiene la imagen correspondiente recuperada por la cámara. Ambas variables serán accedidas luego para unir sus datos junto con la información proveniente de otros hilos de lectura.

Thread ONI

Los hilos denominados ThreadONI, análogamente a los hilos Thread2D y Thread3D descritos anteriormente, tienen como objetivo consumir datos de un tipo de entrada específica, en este caso, archivos *ONI*. Cada hilo ThreadONI consume datos de un archivo *ONI* específico, y estos archivos, son configurados en el *XML* de configuración.

Los archivos de tipo ONI son leídos utilizando la librearía OpenNI, que cuenta con funciones específicas para manejar este tipo de archivos y retorna la información en el mismo formato estándar que lo hace cuando el origen es un dispositivo físico.

Cada hilo ThreadONI mantiene dos variables compartidas en las que contiene, por un lado, el arreglo de bytes de la nube de puntos obtenida, y por el otro, la imagen RGB recuperada desde el archivo. Ambas variables serán accedidas luego para unir sus datos junto con la información proveniente de otros procesos de lectura.

Estos hilos son particularmente útiles para simplificar tareas que, o bien consumen mucho tiempo, o bien requerirían de una gran cantidad de cámaras RGB o de profundidad. Los casos identificados fueron los siguientes:

- El tiempo que se debe invertir para montar una escena con la suficiente cantidad de elementos, sumado al hecho de tener que conectar las cámaras RGB y sensores de profundidad en los lugares adecuados, puede ascender a varios minutos. Para reducir estos tiempos repetitivos, es posible grabar toda la escena una única vez en archivos *ONI* y utilizarlos luego para reproducirla en cualquier momento sin la necesidad de montar nuevamente la escena cada vez.

- Analizar y mejorar el proceso de generación de una malla requiere, no sólo tiempo, sino que implica necesariamente tener que comparar los resultados obtenidos a partir del mismo conjunto de datos. Grabar las escenas en formato *ONI* simplifica esta tarea ya que permite utilizar el mismo conjunto de datos y procesarlo múltiples veces, pudiendo así comparar la salida generada por el sistema.
- Entre los aspectos a medir al momento de evaluar la robustez del sistema, resulta importante analizar su comportamiento cuando se lo somete al procesamiento de una gran cantidad de cámaras de profundidad de forma simultánea. En general esto no es sencillo de lograr ya que la cantidad de dispositivos a menudo es limitada. Sin embargo, no existe una limitación en cuanto a la cantidad de archivos *ONI* que se pueden asignar como entrada a un Cliente. Por tanto, es posible grabar varios ángulos de una misma escena en formato *ONI*, y configurarlos luego como entrada en los Clientes para simular una gran cantidad de dispositivos de profundidad.

4.2.3.3 Pre-procesamiento

Tal como se detalló en la sección anterior, la información recuperada por los procesos de lectura de datos se mantiene en variables compartidas para ser consumidas. El proceso Grabber mantiene referencias hacia los procesos activos de lectura, y es el responsable de recolectar la información que estos almacenan en cada iteración. Los datos aportados por los procesos que leen pueden ser de tipo imagen o nubes de puntos, y dependiendo del caso, se le aplica un pre-procesamiento distinto.

Las imágenes RGB son re-escaladas según lo indique el atributo *resolutionDownSample* en el *XML* de configuración.

Las nubes de puntos en cambio sufren cuatro transformaciones:

- En primer lugar, se produce una reducción uniforme de los puntos. Esto significa que se genera una nueva nube, con una cantidad $\frac{1}{pointsDownSample^2}$ de los puntos originales. El valor *pointsDownSample* se establece desde el archivo *XML* de configuración. Un valor de 1, redunda en que la nube se copiará sin cambios, mientras que un valor de 2, implica que se utilizará la cuarta parte de los puntos, y así sucesivamente.
- Una vez obtenida la nube de puntos filtrada del paso anterior, se remueven los puntos que no aportan información valiosa, ya sea por estar muy cerca de la cámara o por tener algún valor nulo.
- Hasta este momento, los valores de la nube de puntos se encuentran en representación proyectiva. En una representación proyectiva, el espacio tridimensional completo se representa a través de una imagen bidimensional vista desde un punto en particular, mientras que una representación en “mundo real”, posiciona al objeto en el espacio 3D.
  A los puntos de la nube se les aplica una la transformación que cambia su representación de coordenadas proyectivas a coordenadas en el mundo real [59].

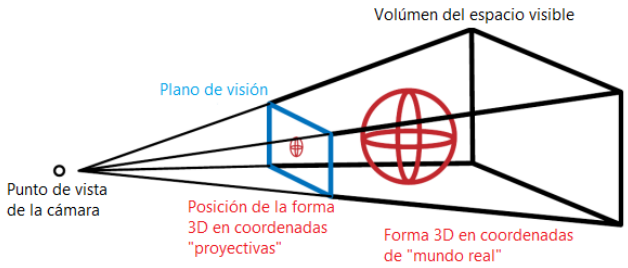


Diagrama 4-3 *Para la Imagen: Relación entre coordenadas proyectivas y de mundo real. Las coordenadas de mundo real posicionan el objeto en el espacio 3D. Las coordenadas proyectivas describen cómo se vería en un plano de visión.*

- Por último, a los puntos resultantes se les aplica una transformación compuesta por una rotación y una traslación. El objetivo de esta transformación es el de posicionar las nubes de tal forma que se unan correctamente con las nubes de otras cámaras al ser unidas posteriormente en el Servidor y lograr así una representación fidedigna de la escena. De no aplicarse esta transformación, los puntos de las nubes permanecerían posicionados de forma relativa a cada cámara, y la unión con otras nubes resultaría en una superposición de puntos que no sería representativa de la escena grabada. En la definición de cada cámara de profundidad que figura en el *XML* de configuración del Cliente se encuentran los datos necesarios para aplicar la traslación y rotación de sus puntos.

Culminado el pre-procesamiento, se unifica la información en una sola estructura almacenándola hasta que el proceso encargado de la transmisión esté libre y en condiciones de utilizar los datos.

Cuando el Grabber finaliza su ejecución, libera toda la memoria utilizada como resultado del pre-procesamiento de datos e indica a los procesos de lectura y de trasmisión que también se detengan.

4.2.3.4 Transmisión

El proceso Transmitter es el último proceso en la secuencia de ejecución del Cliente, y tiene como responsabilidad mantener abierta la conexión con el Servidor y enviar la información de cada frame.

Mantiene el estado de la conexión en una variable global, que puede tener los valores: desconectado, conectado, finalizado o reintentando.

Presionando el botón “Conectar” desde la interfaz del Cliente, el proceso Transmitter intentará comenzar la conexión utilizando los datos contenidos en los atributos *serverIp* y *serverPort* incluidos en el *XML* de configuración.

El proceso monitorea de forma constante el estado de la conexión y, en caso de que ocurra un error, intenta reconectarse automáticamente para continuar transmitiendo. Luego de una caída, el Transmitter espera tres segundos antes de intentar recuperar la conexión.

Una vez que logra establecer la conexión, toma el último *frame* generado por el Grabber, calcula el tamaño en bytes a enviar y particiona el paquete en fragmentos más pequeños, tal como se indica en la sección 4.2.4.3.

De forma adicional, existen dos últimas transformaciones opcionales que el Trasmitter puede aplicar a los datos con el fin de reducir su tamaño:

- A través del parámetro resolutionDownSample, se reduce la calidad de la imagen a transmitir, disminuyendo por consecuencia su tamaño en bytes. Para realizar esta reducción de tamaño se utiliza una librería dinámica externa denominada imageCompression.dll. Se amplía más información al respecto en la sección 4.2.3.7.
- El XML de configuración cuenta además con el parámetro denominado allowCompression que permite aplicar un algoritmo de compresión a cada *frame* antes de enviarlo. Este parámetro acepta los valores: 0 para deshabilitarlo o 1 para habilitarlo. Para realizar la compresión se utiliza una librería dinámica externa denominada FrameCompression.dll, que se describe en la sección 4.2.3.7. El XML de configuración del Servidor también cuenta con el parámetro *allowCompression* y siempre debe tener el mismo valor que el estipulado en los Clientes para que el sistema funcione correctamente.

4.2.3.5 Modos de funcionamiento Persistence y Real Time

En esta sección se detallarán las dos modalidades principales de ejecución soportadas por el sistema, que se definen con los nombres Persistence y Real Time.

En modo Persistence, se ejecuta el Cliente sin la necesidad de una instancia de Servidor, ya que en ningún momento envía información a procesar. El único cometido de ejecutar el Cliente en modo Persistence es el de almacenar en disco la información de las cámaras RGB y de profundidad para ser procesadas luego. El sistema genera archivos *MP4* y *ONI* correspondientes a la información recibidas por las cámaras, y las almacena en la siguiente estructura de carpetas:

Diagrama 4-4 *Estructura del directorio en el que el Cliente persiste los datos en modo Persistencia.*

Esta modalidad se activa cambiando el valor de la bandera *persistence* a 1 en el archivo de configuración *settings.xml* del Cliente.

A diferencia del modo Persistence, en el modo Real Time es necesario contar con una instancia de Servidor en ejecución. El Cliente recopila toda la información recibida de las cámaras y la envía al Servidor para ser procesada.

Esta modalidad se activa cambiando el valor de la bandera *realTime* a 1. Se puede encontrar una descripción más detallada en la sección "Settings.xml del Cliente" del Anexo.

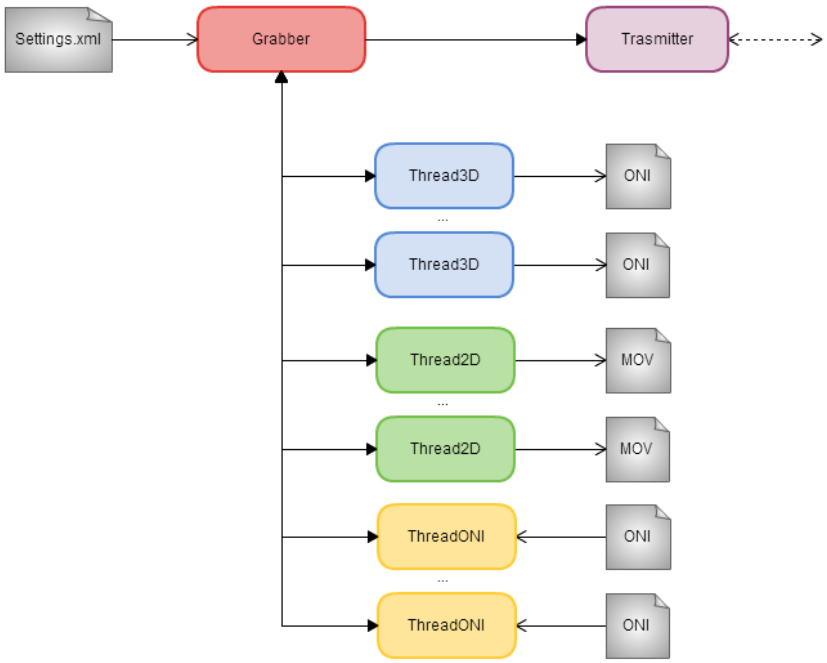


Diagrama 4-5 *Descripción del modo en que se conectan los hilos dentro del Cliente durante la trasmisión al Servidor y los archivos involucrados. La cantidad de Thread3D, Thread2D y ThreadONI depende exclusivamente de los datos del Settings.xml.*

4.2.3.6 Interfaz del Cliente

La interfaz del cliente muestra los siguientes datos y funcionalidades:

- **FPS**: muestra el *framerate* de ejecución al que están trabajando el Cliente y el Servidor. Existe un parámetro denominado FPS en la configuración de ambos programas que indica el *framerate* objetivo al que se espera que ejecuten. De todos modos, el *framerate* real tiende a ser siempre menor al objetivo.
- **CLI ID**: muestra el ID de Cliente. Este parámetro es tomado también de la configuración inicial, y se muestra en la interfaz para simplificar la identificación de cada uno de los Clientes, particularmente para los casos en los que hay varios de éstos ejecutando simultáneamente.
- **Botón Connect**: permite iniciar la comunicación con el Servidor y sirve para monitorear la conexión. Si ocurre algún problema en la red, o el Servidor se desconecta por algún

motivo, el Cliente se desconecta y el estado se ve reflejado en el botón automáticamente.

- **Botón Exit**: cierra la ejecución del Cliente, desconectándolo del Servidor, finalizando todos los procesos activos (Transmitter, Thread2D, Thread3D y ThreadONI) y saliendo del programa.
- **Vista previa de las cámaras**: al momento de comenzar a transmitir, en el Cliente se muestran las imágenes obtenidas de las cámaras configuradas en el archivo settings.xml. Debajo de cada imagen se muestra el tipo de entrada (ONI, DEPTH, RGB) y el id de cámara especificado (ID CAM).

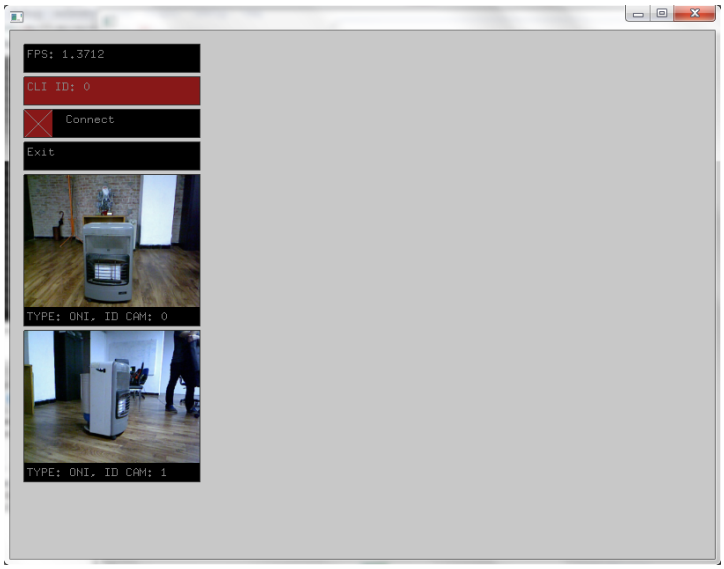


Imagen 4-5 *Esta imagen muestra la interfaz gráfica del Cliente. En este ejemplo, se encuentra trasmitiendo la información de dos cámaras al Servidor.*

4.2.3.7 Compresión de datos

Tal como se mencionó anteriormente, el proceso Trasmitter cuenta con la posibilidad de aplicar dos operaciones para reducir el tamaño de un *frame* antes de enviarlo al Servidor.

La primera de ellas plantea el enfoque de bajar la calidad de las imágenes que se envían, produciendo en consecuencia una reducción del tamaño del *frame* en su conjunto.

Se desarrolló para esto una librería dinámica externa denominada *imageCompression.dll* que encapsula una implementación concreta que soluciona este problema. La implementación utilizada se basó en el proyecto *Jpeg-Compressor* [60].

Este mecanismo de compresión se habilita modificando el valor del parámetro *resolutionDownSample* en la configuración del Cliente y acepta valores decimales en el rango de 0 a 1 inclusive. Los valores más próximos a 1 implican una menor reducción de la calidad de la imagen y los próximos a 0, por el contrario, provocan una baja de calidad mayor. Si el parámetro es estipulado en 1, no se aplicará ninguna reducción de calidad y se enviará la imagen original.

El segundo enfoque incluido para reducir el tamaño de los *frames* apunta a que sean comprimidos antes de enviarlos y descomprimirlos al momento de recibirlos. Se desarrolló con este fin una librería dinámica externa denominada *FrameCompression.dll*. Al igual que en el caso anterior, propone una solución concreta al problema pero deja la puerta abierta a nuevas implementaciones que mejoren la performance. La implementación incluida con el proyecto utiliza la librería *Zlib* [54].

La compresión de los *frames* con esta librería se habilita mediante el parámetro denominado *allowCompression* presente tanto en el *XML* de configuración del Cliente como en el del Servidor. Un valor de 1 significa que está habilitada y un valor de 0 desactiva la compresión. Es importante notar que el valor de este parámetro debe ser siempre el mismo, ya sea 1 o 0, en el Cliente y el Servidor.

El objetivo de incluir estas implementaciones dentro de librerías externas es que sean fácilmente sustituibles en un futuro por cualquier otra implementación que logre mejores resultados.

4.2.4 Servidor

El Servidor tiene como responsabilidad procesar los datos recolectados de los Clientes, sincronizarlos, unir las nubes de puntos recibidas pertenecientes a un instante de tiempo dado, obtener una malla a partir de ellas y brindar las imágenes apropiadas para poder texturizarla.

Cuenta con un proceso principal que está constantemente a la espera de nuevas conexiones. Cuando una nueva conexión arriba, el Servidor le asigna un número de puerto al Cliente e instancia un proceso ThreadServer que atenderá exclusivamente a dicho Cliente en el puerto asignado. El objetivo de hacer esta separación es lograr que el Servidor utilice al máximo todos sus recursos y no interferir con el procesamiento de los datos de un Cliente cuando ocurren nuevas conexiones.

Cada ThreadServer recibe y almacena los *frames* de los clientes en un buffer circular. La decisión de utilizar un buffer de este estilo implica que, al llenarse, se descartarán los *frames* más viejos evitando así que la cola de *frames* crezca indefinidamente. En el apartado “Procesos Receptores” del capítulo 4.2.4.1 se detalla su funcionamiento en profundidad.

El proceso principal del Servidor consulta a todas sus instancias de ThreadServer en busca de nuevos *frames*. En cada iteración toma los datos recibidos, los organiza cronológicamente y los combina en un solo *frame*. En este punto la información de todos los clientes queda unificada, sincronizada y en condiciones de ser procesada. Una vez finalizado el procesamiento, se obtiene como resultado una malla de polígonos.

Como salida del sistema, el Servidor publica la malla final generada y las imágenes para texturizarla. Las aplicaciones que quieran hacer uso de esta información deberán consumir los datos según se indica en el apartado 4.2.4.5.

El proceso de generación de la malla, es el punto más costoso de todo el sistema y por este motivo se decidió paralelizarlo según se explicará en un apartado más adelante.

Toda la actividad que ocurre en el Servidor puede ser registrada en un archivo de log llamado *server_log.txt*. La cantidad de información volcada a este archivo dependerá del atributo *logLevel* que figura en el *XML* de configuración.

4.2.4.1 Categorización de procesos del Servidor

Podemos agrupar a los procesos del Servidor en las siguientes categorías dependiendo de su participación en la generación de las mallas: Receptores, Generadores y Publicación.

Procesos receptores

En esta categoría se encuentran los procesos Server y ThreadedServer mencionados anteriormente. Estos procesos están involucrados directamente en la comunicación con los clientes, tanto en la recepción de los datos de cada uno, como en el establecimiento y administración de la conexión con ellos.

El proceso Server tiene además la responsabilidad de realizar la inicialización de todos los procesos involucrados en la generación de las mallas, controlar la interfaz y realizar las eliminaciones correspondientes al momento de cerrar el sistema, dejando la memoria liberada y en el estado original.

Tal como se mencionó anteriormente, el proceso ThreadServer por su parte tiene la responsabilidad de mantener abierta la conexión y recibir los datos que le envíe su Cliente asignado. Existe en el sistema una instancia de ThreadServer activa por cada Cliente que monitorea constantemente la conexión hasta que arriba algún dato. Los primeros datos que recibe de cada *frame*, determinan la cantidad de paquetes que deberá esperar.

Una vez recibida la totalidad de paquetes, los datos son mapeados a su estructura original y almacenados en el buffer circular a la espera de ser procesados por el Servidor.

El proceso continúa ejecutando hasta que ocurre una de dos cosas: o bien el usuario indica que desea cerrar el Servidor, o bien el Cliente envía una flag señalando que desea dejar de transmitir.

Procesos generadores

Para solucionar el problema del alto costo computacional requerido por el procesamiento de las nubes de puntos, se utilizó un enfoque basado en una estrategia de Divide & Conquer que consistió en separar el procesamiento de los *frames* entre varios procesos que ejecutan en paralelo.

Los procesos generadores están vinculados de forma directa a la obtención de las mallas a partir de las nubes de puntos recibidas por parte de los Clientes. Dentro de esta categoría se encuentran los procesos Mesh Generator y Mesh Threaded Generator que se describirá a continuación.

El Mesh Generator tiene dos funciones principales: por un lado, inicia la ejecución de los procesos que se encargan de la generación de la malla propiamente dicha, denominados Mesh Threaded Generator. El parámetro *totalFreeCores* dentro de la configuración del Servidor indica la cantidad de procesos Mesh Threaded Generator que serán iniciados.

En segundo lugar, mantiene registros de los Mesh Threaded Generator iniciados y monitorea su actividad a través de un bucle principal, asignándoles nuevos *frames* para procesar en los momentos en que estos procesos se encuentran ociosos.

Los *frames* que se asignan para ser procesados, son *frames* finales, es decir que ya contienen toda la información proveniente de las cámaras de los distintos clientes unificada en una estructura.

Al terminar su ejecución, el Mesh Generator recorre el listado de procesos Mesh Threaded Generator que inició, los finaliza y libera la memoria asignada.

Por su parte los procesos Mesh Threaded Generator tienen la responsabilidad exclusiva de generar una malla a partir de una nube de puntos. Se encuentran en ejecución continua ya sea procesando o a la espera del arribo de nueva información.

Los algoritmos que utiliza para la generación de la malla están encapsulados en una librería externa que es invocada dinámicamente. Se optó por este camino para que resultara más sencillo a la hora de modificar el procedimiento o los algoritmos escogidos.

Con el ingreso de cada nuevo *frame* para procesar, recibe un identificador único que será publicado inalterado junto a la malla generada. Este valor se utiliza posteriormente para ordenar las respuestas que los Mesh Threaded Generator activos generan y obtener una secuencia de mallas cronológicamente correcta.

El Mesh Threaded Generator maneja un indicador que puede tomar tres posibles valores. Estos valores reflejan su estado y sirve para diferenciar la etapa en las cual se encuentra la generación de la malla actual.

- **Idle**: el proceso se encuentra libre, esperando una nube de puntos para procesar.
- **Processing**: tiene lugar cuando se está trabajando en la generación de una malla. En esta etapa se invoca y utiliza la librería externa.
- **Complete**: indica que el procesamiento de la malla fue completado y avisa al MeshCollector que existe una malla generada para ser enviada a la salida del sistema.

Procesos de publicación

El único proceso ubicado en esta categoría es Mesh Collector y se encuentra ubicado al final de la secuencia de pasos en la generación de las mallas e implementan la salida del sistema, publicando los datos procesados para ser consumidos por aplicaciones externas.

El Mesh Collector mantiene referencias a cada uno de los Mesh Threaded Generators que están en ejecución y los revisa constantemente en espera de que alguno cambie su estado a Complete, en dicho caso toma la información y la almacena. La información leída incluye un conjunto de imágenes (una por cada una de las cámaras), la malla generada y el identificador único de cada *frame*.

Dado que los procesos Mesh Threaded Generator ejecutan en forma paralela, no es posible verificar que su orden de completitud sea el orden cronológico correcto. Por este motivo, el proceso debe reordenar la información leída para asegurarse que cumplan la secuencia original. Dicho ordenamiento se realiza en base al identificador único de cada *frame*, utilizando el criterio de menor a mayor.

Una vez reordenada la información, es divulgada a través de otra librería externa denominada *SharedMemory.dll*. Esta librería implementa la publicación de los datos por medio del uso de memoria compartida.

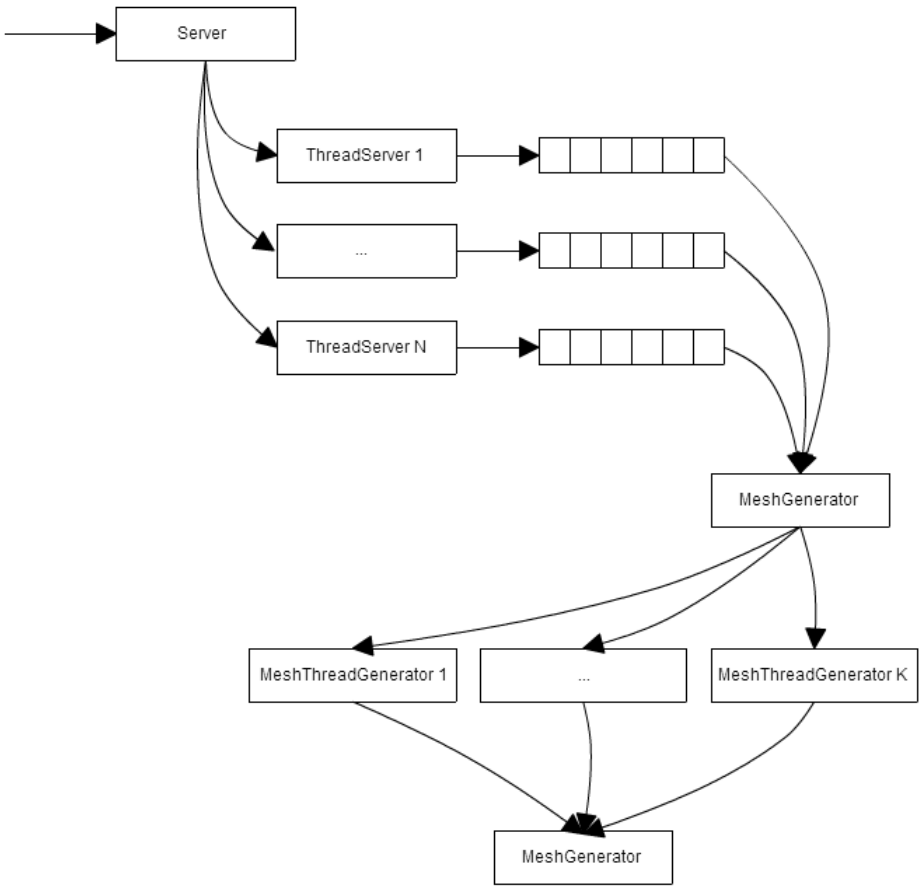


Diagrama 4-6 *Descripción del flujo de datos en el Servidor e interconexión de los hilos involucrados en el proceso de generación de una malla.*

4.2.4.2 Interfaz del Servidor

La interfaz del Servidor muestra los siguientes datos y funcionalidades:

- **FPS**: muestra el framerate objetivo de ejecución indicado en el *XML* de configuración inicial. Este valor tiende a disminuir al momento en el que el Cliente se conecta con el Servidor y comienza a transmitir datos.
- **IP**: muestra la dirección IP del Servidor. Este valor puede ser útil al momento de configurar los clientes.

- **Port**: muestra el puerto actual en el que el Servidor está esperando recibir conexiones, este valor se modifica desde el *XML* de configuración.
- **Desired FPS**: muestra el *framerate* objetivo que se indicó en el *XML* de configuración.
- **Max Package Size**: muestra el tamaño máximo de paquete en el que se subdividen los *frames* que se le envía desde el Cliente al Servidor.
- **Max Threads**: muestra el valor especificado en el archivo de configuración, que indica la cantidad de clientes en paralelo que podrán ser atendidos.
- **Max Cli Buffer**: indica el tamaño establecido como valor máximo para el buffer circular que se utilizan para recibir los *frames* de los clientes.
- **Botón Exit**: cierra la ejecución del Servidor, finalizando todos los procesos activos ThreadServer, MeshGenerator, MeshThreadedGenerator y MeshCollector.
- **Vista previa de las cámaras**: se muestra una preview de las imágenes que están llegando, mostrando adicionalmente el id del Cliente y la cámara dentro de este.

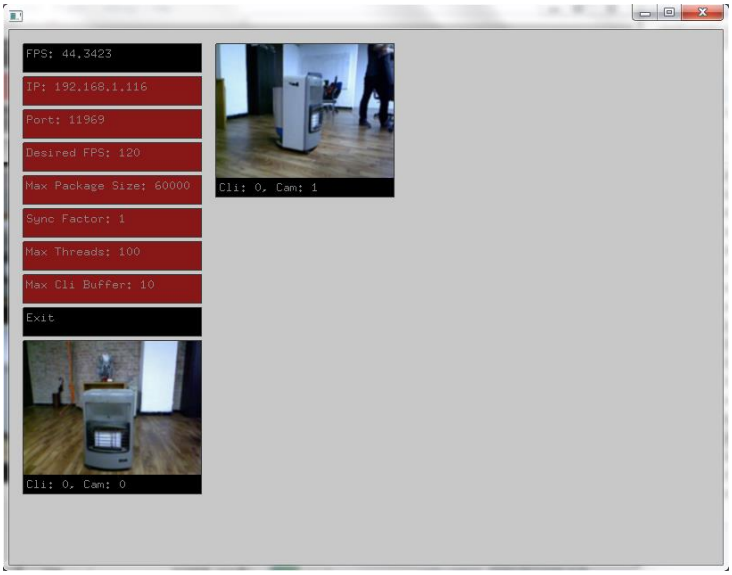


Imagen 4-6 *En la imagen de arriba se puede ver la interfaz gráfica del Servidor. En este ejemplo, se encuentra procesando la información recibida de dos cámaras a través de un Cliente.*

4.2.4.3 Protocolo de comunicación entre el Cliente y el Servidor

Para establecer la comunicación entre el Cliente y el Servidor es necesario ejecutar una serie de pasos entre ambas partes.

Al inicio, el Cliente es el único que cuenta con datos para conectarse al Servidor. En su configuración, existen las etiquetas *serverIp* y *serverPort*, que indican respectivamente la IP y puerto en los que el Servidor espera nuevas conexiones.

El Cliente solicita entonces una conexión al Servidor utilizando estos datos y, una vez conectado, le envía un mensaje identificándose con su número de Cliente.

El Servidor por su parte, al recibir la identificación del nuevo Cliente, instancia un proceso ThreadServer para atenderlo, asignándole un puerto exclusivo y cierra la conexión.

Posteriormente, el nuevo proceso ThreadServer inicia una conexión con el Cliente y, una vez establecida, este último comienza a enviar los *frames* uno por uno.

Tanto el Cliente como el Servidor pueden finalizar la conexión cuando lo deseen. Para hacerlo, envían un mensaje a la contraparte indicando que se disponen a cerrar la conexión. Esto permite que ambos sistemas estén listos y liberen los recursos necesarios antes que la conexión finalice.

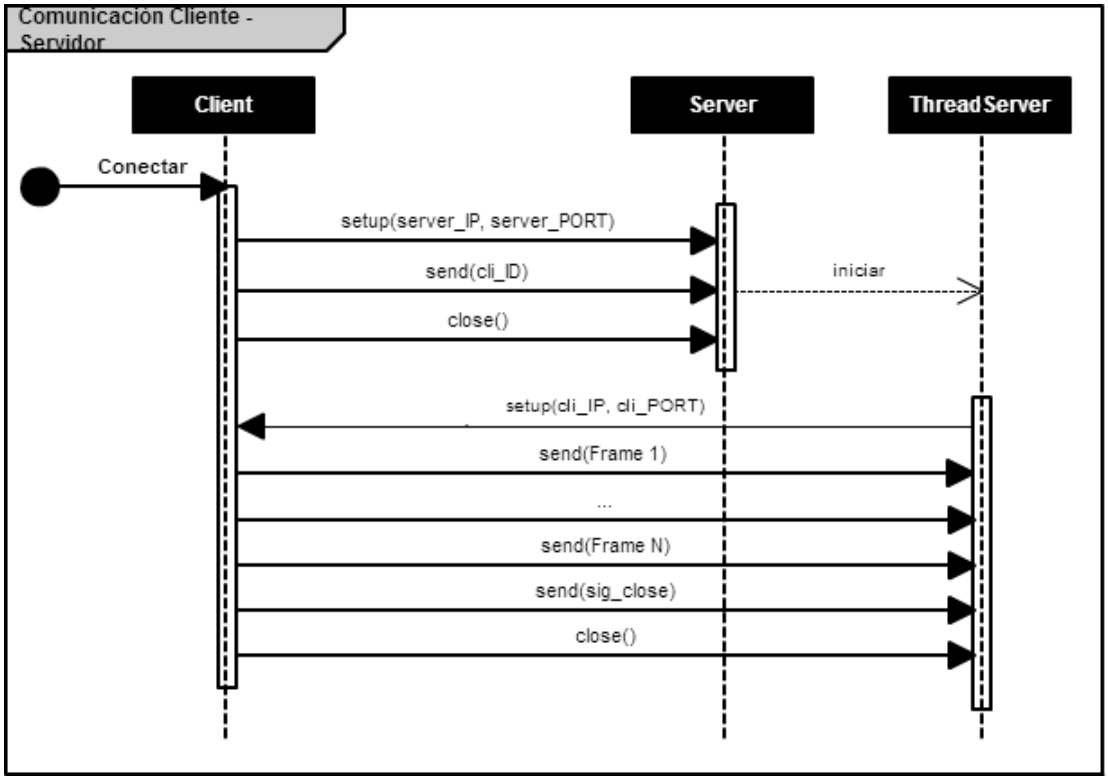


Diagrama 4-8 *Diagrama de comunicación entre el Cliente y el Servidor.*

Estructura y tamaño del paquete a enviar

El tamaño de los *frames* excede ampliamente el tamaño máximo de mensaje que se puede transmitir por red en una conexión TCP (65535 bytes). Por tanto, es necesario dividir la información en mensajes más pequeños para poder ser enviados. Del mismo modo, al recibir los datos es necesario conocer cuántos mensajes fueron transmitidos originalmente.

El máximo tamaño en bytes que tendrán los mensajes que envíe el Cliente al Servidor es un valor que se encuentra definido en la configuración y se denomina maxPackageSize. Este valor debe estar idéntico tanto en el *XML* del Cliente como en el del Servidor.

La metodología empleada para enviar cada *frame* se define en los siguientes dos pasos:

- Paso 1: Dado el valor *maxPackageSize* definido en el *XML* de configuración y el tamaño en bytes del *frame* a enviar, se calculan dos valores:
  - Por un lado, se calcula la cantidad de mensajes de tamaño *maxPackageSize* a enviar. Esto se hace quedándose con la parte entera de la división directa entre el tamaño del *frame* y *maxPackageSize*. En adelante se llamará a este valor "*cantidad_mensajes*".
  - El segundo valor que se calcula es el resto de la división anterior. Se llamará a este valor "*tamaño_paquete_final*" y por la forma de calcularlo siempre será de tamaño menor a *maxPackageSize*.

  Ambos valores son calculados y enviados al Servidor antes de enviar el *frame* correspondiente.

- Paso 2: el Servidor lee estos dos valores y con ellos ya es capaz de calcular qué cantidad de memoria necesita reservar para poder recibir el *frame* completo. La memoria reservada será un espacio de tamaño $(maxPackageSize \times cantidad\_mensajes) + tamaño\_paquete\_final$.

Tamaño del arreglo de bytes a enviar

Al momento de transmitir el *frame* entre los Clientes y el Servidor se ejecutan los siguientes tres pasos:

- Paso 1: se recolectan los nuevos datos a partir de las cámaras conectadas. Por cada cámara, se toman la última imagen capturada y su nube de punto, si corresponde.
- Paso 2: se genera el arreglo de bytes a transmitir y se calcula su tamaño. Para esto, se toman en cuenta dos componentes:
  A. El tamaño del arreglo de bytes de imágenes y nubes de puntos.
  B. El tamaño de los atributos adicionales necesarios para poder reconstruir la estructura del arreglo desde el Servidor (ancho y alto de cada imagen, tamaño de cada nube de puntos, id de cliente, id de cámaras, tipo de cámaras, etc.).

  Todos estos valores se suman y obtenemos el tamaño total del *frame* al que se llamará *FRAME_SIZE*.

- Paso 3: se procede a enviar los datos en series de tamaño *maxPackageSize*. Se toma el arreglo de bytes, se divide en sub paquetes de tamaño $\frac{FRAME\_SIZE}{maxPackageSize}$ y se envían consecutivamente.

Formatos de frame enviado

Como se explica en la sección anterior, el Cliente envía una estructura de datos al Servidor que se denomina *frame*. Esta estructura empaqueta la información actualizada de todas las cámaras configuradas en ese Cliente.

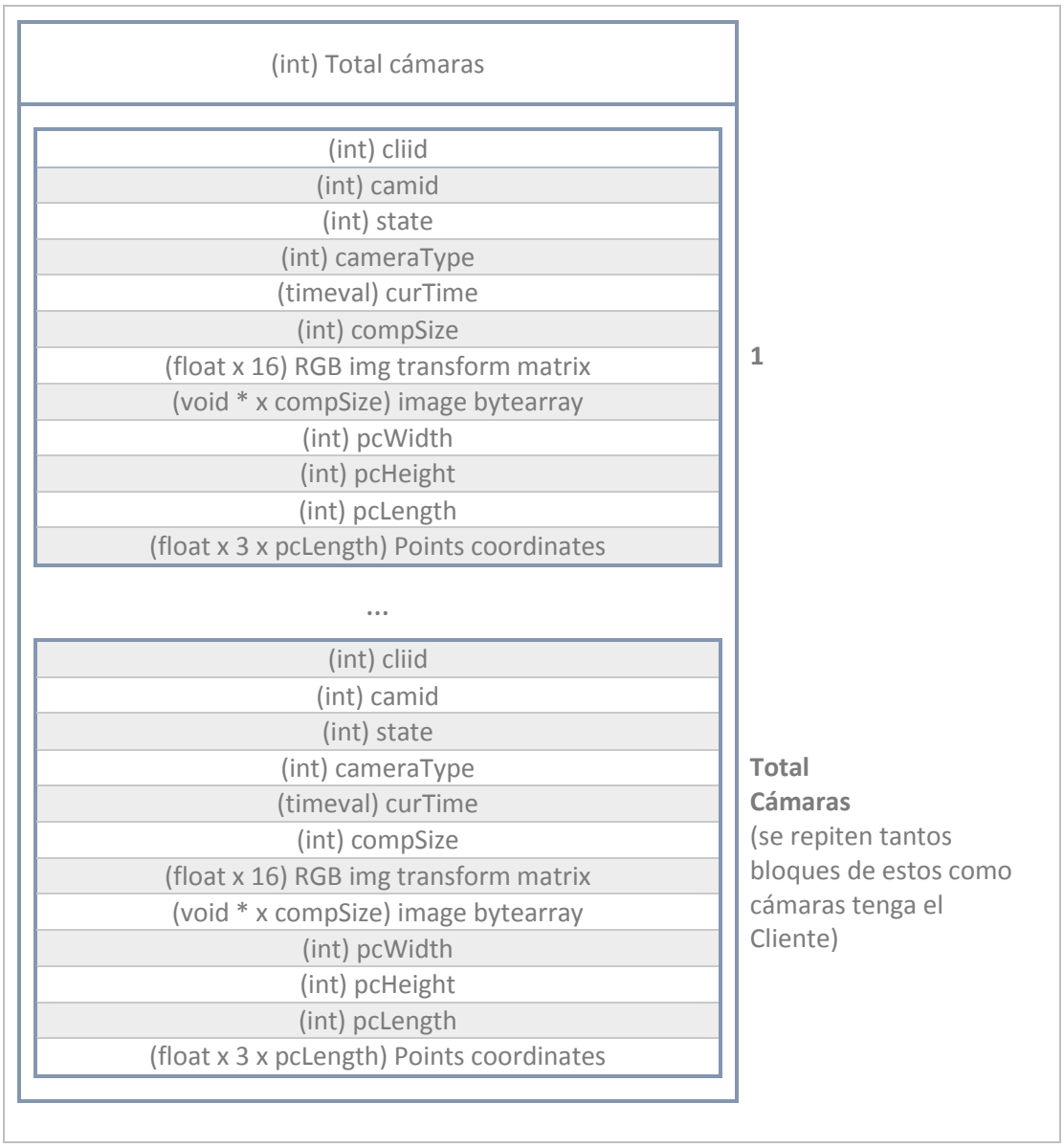


Diagrama 4-7 *Estructura interna de los frames que se envían entre el Cliente y el Servidor.*

4.2.4.4 Generación de la malla

Para la generación de las mallas, se construye un *frame*, el cual contiene la unión de las nubes de puntos de todos los sensores de profundidad, obtenida en el paso de la combinación de los datos mencionado anteriormente.

Los pasos necesarios para generar la malla a partir de la nube de puntos se encuentran encapsulados en la librería *MeshGenerator.dll*. El algoritmo que se implementó en esta librería, consiste en una técnica para la selección de los puntos, una técnica para generar la malla y opcionalmente otra para alisarla. A continuación, se detallan cada uno de los pasos:

- **Selección de los puntos**: la técnica utilizada es *Poisson Disk Sampling* [61]. Consiste en distribuir los puntos para que queden ubicados a una distancia dada, logrando así construir una malla con una distribución más homogénea de los puntos. Este paso

permite que la malla se genere más rápidamente debido a la disminución de puntos y a la distribución de los mismos.

- **Generación de la malla**: la técnica seleccionada para generar la malla fue Ball-Pivoting. Como se vio en el estado del arte, este algoritmo es uno de los que brinda mejores resultados en términos de tiempo.
- **Alisado de la malla**: dado que la malla resultante de aplicar las dos técnicas anteriores puede generar una superficie muy irregular, se decidió emplear una técnica para alisarla denominada *Laplacian Smooth* [62].

Como se mencionó, la generación de la malla se encuentra en una librería llamada de forma dinámica dentro del programa. Esto permite que si en algún momento se desea cambiar el algoritmo de generación de las mallas, únicamente se debe implementar una librería que respete la entrada y salida de los datos.

Como entrada se debe usar la siguiente estructura:

```
struct PointCloud {
    float* x;
    float* y;
    float* z;
    int lenght;
};
```

Esta estructura permite representar una nube de puntos en tres dimensiones de un largo específico. Donde x, y, z son arreglos de floats de tamaño determinado por la variable lenght.

Como salida se debe respetar la siguiente estructura:

```
struct FaceStruct {
    float p1[3];
    float p2[3];
    float p3[3];
};
```

Un arreglo de esta estructura representa una malla. Donde p1, p2, p3 son los tres puntos que forman una cara. Cada punto está compuesto por un arreglo de tres coordenadas x, y, z en las posiciones 0, 1, 2 respectivamente. La cantidad de caras de la malla es pasada como otro parámetro, así como el identificador del *frame* de la escena que representa esa malla. La firma de la función queda dada de la siguiente forma:

```
meshGenerate(PointsCloud* nbIN, FaceStruct** faces, int* numberFaces, int
nroFrame);
```

4.2.4.5 Salida del Sistema

A partir de cada *frame* procesado, el Servidor genera una salida que consiste en una malla y un conjunto de imágenes. Los programas que consuman la salida del Servidor pueden utilizar cada grupo de imágenes para texturizar la malla asociada.

Tal como se mencionó en la arquitectura, y del mismo modo que se realizó con los puntos más sensibles del sistema, la implementación concreta utilizada para publicar los resultados del Servidor se encapsuló en una librería dinámica externa, en este caso de nombre *SharedMemory.dll*.

La solución planteada en esta versión de la librería utiliza memoria compartida para publicar el resultado y viene provista de cuatro funciones que permiten escribir y leer la salida. Las operaciones de escritura denominadas ShareMesh y ShareImage, son utilizadas por el Servidor para publicar la malla e imágenes respectivamente, mientras que las operaciones ReadSharedMesh y ReadSharedImage fueron incluidas para simplificar el trabajo de los programas que quieran consumir la salida y permiten recuperar los datos de memoria compartida.

A continuación se detallará cuál es el modo correcto de acceder a los datos publicados por el Servidor a través de esta librería.

Para poder acceder a la salida del sistema en memoria compartida es imprescindible conocer el ID de cada Cliente y el de las cámaras asociadas a cada uno. Los datos son utilizados para generar las claves con las que se accede a la memoria compartida. La nomenclatura de dichas claves es la siguiente:

- Imágenes RGB:
  - **Clave de imagen**: “ImageId” + ID_CLIENTE + TIPO_DE_CAMARA (1-RGB, 2-Depth Camera) + ID_CAMARA. Aquí se publica el número de *frame*, también denominado “ID MOMENTO”. Este valor le sirve al sistema que consume los datos para hacer un chequeo de los *frames*, ya que dos *frames* son diferentes sólo si tienen distinto “ID MOMENTO”.
  - **Clave de ancho de imagen**: “ImagePixelsW” + ID_CLIENTE + TIPO_DE_CAMARA + ID_CAMARA. Bajo este tag se publica el ancho de la imagen que se está compartiendo.
  - **Clave de alto de imagen**: “ImagePixelsH” + ID_CLIENTE + TIPO_DE_CAMARA + ID_CAMARA. Bajo este tag se publica el alto de la imagen que se está compartiendo.
  - **Clave de datos de imagen**: “ImagePixels” + ID_CLIENTE + TIPO_DE_CAMARA + ID_CAMARA. Bajo este tag se publica el arreglo de bytes propiamente dicho de la imagen que se está compartiendo.

  Esta información se publica por cada *frame* generado y por cada cámara disponible. El único modo de consumirlo es conociendo las claves de cada dato. Por ese motivo es muy importante contar con los ID correctos de cada cámara y Cliente.

- Malla generada:
  - **Clave “MeshId1”**: aquí se publica el número de malla. El concepto es similar al de “ID MOMENTO”. Este valor le sirve al sistema que consume los datos para hacer un chequeo de los *frames*, ya que dos mallas son diferentes solo si tienen distinto número.

- **Clave "MeshNumberFaces1"**: bajo este tag se publica el número total de caras que contiene la malla.
- **Clave "MeshFaces" + ID_MOMENTO**: bajo este tag se accede a la estructura de la malla propiamente dicha. La estructura que se obtiene es un array de largo "valor(MeshNumberFaces1)" donde cada elemento es de tipo FaceStruct.

4.2.5 .Prototipo: Reproductor

Para visualizar el resultado final del procesamiento del Servidor, se implementó un programa que renderiza en tiempo real el video 3D generado por el propio Servidor. Esta aplicación es un programa basada en OpenGL y openFramework (con el addon ofxXmlSetting) cuyo objetivo es proporcionar una interfaz para la visualización de la malla texturizada. El Reproductor le facilitará al usuario navegar por la escena que se está filmando en tiempo real.

Es importante aclarar en este punto que el Reproductor no permite cargar un archivo de disco ni ofrecer navegabilidad en el tiempo, sino que en realidad es un visualizador, que posibilita ver los frames de la escena generados por el Servidor mientras éste los publica. Es necesario destacar que el código del Reproductor es completamente extensible, y que los cambios para que el Reproductor tome la información de disco y permita navegar en el tiempo por estos archivos son relativamente pequeños.

La principal razón de que no se implementara un Reproductor que tomase la información de disco y la renderización fue que, dado el estado actual de la tecnología, dicho proceso tomaría mucho tiempo y degradaría considerablemente el desempeño del Reproductor.

Como ya se mencionó, la lógica del Reproductor es apenas diferente de la del Calibrador, salvo por el pasaje de la información y el hecho de que se recalculan las texturas para la malla de polígonos cada vez que un nuevo frame de la escena es proveída por el Servidor. Se pasará a detallar a continuación los procesos que difieren del Calibrador.

4.2.5.1 Lectura de los datos

Actualmente, el pasaje de los datos entre el Servidor y el Reproductor se hace mediante memoria compartida. Esos datos comprenden la malla de polígonos, la cantidad de puntos, las texturas y sus dimensiones. Los demás valores tales como los identificadores de clientes y cámaras, así como las transformaciones, entre otros, son recogidos al inicio del programa cuando se leen de uno de los archivos de configuración generados por el Calibrador.

La razón de que los valores de configuración sean leídos de un archivo *XML*, es porque si se desea extender este programa dichos valores necesariamente deberían estar en un archivo. Se puede decir entonces, que la obtención de los parámetros de configuración, minimizan los cambios que el reproductor debería sufrir en caso de ser modificado.

El punto más importante de esta sección es la comunicación entre el Servidor y el Reproductor mediante memoria compartida. Dicha comunicación está encapsulada en una librería que podría ser reemplazada si se desease agregar otras funcionalidades diferentes a las que ya se

implementaron. Por ejemplo, se podría cambiar el tamaño de la malla de polígonos, el color de las imágenes, entre otras propiedades.

Los datos compartidos son a grandes rasgos los siguientes.

- **Identificador del *frame***: un entero que determina de qué Cliente, de qué cámara y cuál es el índice del *frame* que se está enviando.
- **Malla de polígonos**: esta estructura es un arreglo de caras, donde cada una de ellas es una terna de vértices. Note que el tamaño del arreglo cambia en cada *frame* dado que constantemente aparecen y desaparecen caras en la escena mientras los objetos en ella sufren cambios.
- **Cantidad de caras**: un entero que indica cuántas caras tiene la malla.
- **Imágenes**: un conjunto de arreglos de píxeles, uno por cada cámara RGB de la escena. Cada píxel se compone de tres valores entre 0 y 255, con la codificación RGB.
- **Tamaño de las imágenes**: un conjunto de enteros con los anchos y largos de las imágenes. Note que el tamaño de las imágenes forma parte del *XML* de configuración, sin embargo, se decidió compartir estos valores por si el servidor decide modificar el tamaño de las imágenes dinámicamente.

Si bien no se ha mencionado aún, el Reproductor y el Servidor deben de estar en la misma computadora dado que la comunicación entre ellos se realiza mediante memoria compartida.

Este punto representó un gran desafío en el proyecto. La razón es que, si los dos programas corren en la misma terminal, el desempeño de ambos se degrada en un factor nada despreciable. Pero, si ambos programas se ejecutan en terminales separadas, la comunicación entre ellos se ve limitada por el considerable volumen de los datos que necesitan ser compartidos. Se analizará con mayor detenimiento estas dos alternativas.

Si los programas ejecutan en terminales separadas, la comunicación entre ellos debe hacerse mediante la red. Ahora bien, existen básicamente dos protocolos de red: TCP y UDP. El primero asegura que todos los datos enviados lleguen al destinatario, el segundo no asegura eso pero proporciona una tasa de transferencia de datos mucho mayor.

Dado que la estructura del *frame* que el servidor envía tiene valores críticos tales como el instante de tiempo, los tamaños de la malla de puntos y las dimensiones de las imágenes, UDP no supone ser una buena opción. Entonces, TCP sería la solución para la comunicación entre los programas, pero como el volumen de datos es muy grande, y los tiempos entre un *frame* y otro deben de ser mínimos, TCP queda definitivamente descartado.

La solución ideal para esta arquitectura distribuida de Reproductor y Servidor, entonces sería utilizar TCP para enviar los datos críticos y UDP para los datos pesados, los puntos y píxeles. Dicha implementación supone desafíos técnicos como son la sincronización y la consistencia de los datos. Una alternativa mejor sería incluir algún tipo de compresión, ya sea con o sin pérdida de información, para minimizar el tamaño del paquete de datos.

De ser resueltos estos problemas, el sistema podría utilizar esta solución ya que la comunicación entre los programas se encuentra encapsulada en una librería que podría ser reemplazada por una mejor, como la que se está sugiriendo aquí y mejorar así el desempeño del sistema en lo que a la reproducción respecta.

Si los programas ejecutan en la misma computadora, la solución ideal es la memoria compartida. Utilizando memoria compartida se consigue lo mejor de ambos enfoques, los

datos se transmiten rápidamente y la consistencia está asegurada, por lo que el pasaje de los datos entre el Servidor y el Reproductor está resuelto.

Esta arquitectura introduce un nuevo factor a tener en cuenta: el tiempo de procesamiento. Porque ahora tanto el procesamiento que hace el Servidor como el procesamiento que hace el Reproductor tiene lugar en la misma computadora.

La decisión final de utilizar esta última solución, radicó en el hecho de que la implementación de la memoria compartida, es sumamente sencilla frente a la complejidad de una arquitectura distribuida, con compresión de la información y múltiples protocolos de comunicación para los diferentes datos que se comparten.

Nuevas y mejores implementaciones pueden hacerse para resolver la comunicación entre el Servidor y el Reproductor, e integrarlas al sistema, sin necesidad de cambiar prácticamente nada de la lógica de ambos. La solución actual supone ser nada más que un prototipo que pruebe la utilidad del sistema.

4.2.5.2 Visualización de los datos

La renderización de la malla texturizada es el principal objetivo del Reproductor. Afortunadamente, y dejando de lado los aspectos que tienen que ver con la obtención de los datos, la lógica que hay detrás de este proceso es muy similar a la del Calibrador en su etapa de visualización, salvo por algunos detalles que se explicarán a continuación.

Para fijar algunas ideas, cada vez que un evento de teclado, o del ratón se dispara, la escena se redibuja en la pantalla nuevamente. A su vez, cada vez que un *frame* nuevo le llega al Reproductor, también se redibuja la escena, salvo que en este último evento se deben de hacer una serie de cálculos adicionales que se detallarán más adelante.

Cada vez que la escena se redibuja, se aplica un conjunto de transformaciones al modelo y a la posición de la cámara para ajustar las texturas a la malla de polígonos. La idea es la siguiente. Se recorre cada una de las cámaras RGB y para cada una de ellas se establece el visor de la escena en la posición de la cámara actual, entonces se proyecta la imagen de ésta sobre la malla, solo para las caras que están visibles del modelo desde la posición actual del visor. Una vez hecho esto, se obtiene una malla de polígonos texturizada limitada sólo a las caras que fueron pintadas por la imagen de alguna de las cámaras. Finalmente se traslada y rota el modelo para que se ajuste a la posición establecida por el usuario, a través de los comandos del ratón y del teclado en su navegación por la escena.

Para agilizar los cálculos, se mantiene en un arreglo, cuáles son las caras visibles de la malla para cada una de las cámaras, y dado que la posición de las cámaras es fija, dicho cálculo sólo debería hacerse al inicio del programa si no fuese porque la malla de polígonos cambia, es decir que las caras cambian, por lo que cada vez que la malla se actualiza, es necesario recalcular qué caras son las que están visibles para cada una de las cámaras RGB.

El algoritmo para calcular qué caras son las que están visibles se denomina Occlusion Culling. La eliminación selectiva de oclusión (Occlusion Culling) consiste en evitar la renderización de los objetos cuando estos no están vistos por la cámara ya que son cubiertos por otros objetos. Esto no sucede automáticamente en la renderización 3D ya que la mayor parte del tiempo los objetos más alejados de la cámara son dibujados primeros y los objetos más cercanos son

dibujados encima de ellos, esto se llama overdraw. La oclusión selectiva, Occlusion Culling, es diferente de Frustum Culling. Éste último solamente evita la renderización de los objetos que están afuera del área visual de la cámara, pero no evita renderizar algo que está oculto por un overdraw [49].

4.2.5.3 Algoritmo de Renderización

Como ya se mencionó, dado que el Reproductor es, al igual que el Calibrador, un programa de openGL, casi toda la lógica del mismo radica en un main que se ejecuta al principio, un bucle principal de dibujo y en los eventos de teclado.

En el main se establecen los atributos de la ventana como su posición, tamaño y título. Pero también se asignan las funciones destinadas a escuchar los eventos del ratón, del teclado, los cambios de tamaño y el redibujado de la pantalla.

Después de establecer las configuraciones iniciales, se obtienen los datos, es decir, la malla de polígonos y las imágenes, que se encuentran en memoria compartida. Esos datos resultan del procesamiento que se hizo en el Servidor.

El último paso en el main es inicializar un conjunto de estructuras, para almacenar las transformaciones que haga el usuario en su navegación por la escena. Al mismo tiempo, se instancia un arreglo de texturas, a partir de las imágenes que se obtuvieron del paso anterior. Estos objetos son los que se utilizarán más adelante durante el proceso de renderizado.

A partir del momento en el que termina de ejecutar el main, la lógica del programa recae sobre los métodos: keys, mouse, mouseMove, timer y display. El primero, contiene la lógica que sigue el sistema cuando el usuario manipula el teclado. El segundo y tercero, la lógica para los eventos del ratón. El cuarto de los métodos es el que pregunta cada cierto tiempo si hay un nuevo frame en memoria. Y el último de los métodos es el que se encarga de la renderización de la escena en la pantalla, que tiene lugar después de que ocurra alguno de los eventos anteriores.

En el Reproductor, las teclas que tienen sentido son: W, S, A, D, Q y E para las rotaciones del modelo; M, B, H, N, J y G para las traslaciones; + y - para regular la velocidad del cambio provocado por las teclas anteriores; P para habilitar o deshabilitar las luces; entre otras teclas, cuya acción resulta ser de menor importancia.

Por otro lado, los eventos del ratón que contempla el Reproductor son: el botón izquierdo presionado para trasladar en el eje X; el botón derecho presionado para trasladar en el eje Y; el botón central presionado para trasladar en el eje Z.

Después de que ocurrió alguno de estos dos tipos de eventos, los cambios son trasladados a una estructura que almacena las transformaciones y se invalida la representación visual, con lo que se invoca al evento display, que es donde tiene lugar la renderización y la mayor parte de la lógica del Reproductor.

La función que ejecuta el temporizador ocurre cada intervalos regulares de tiempo. Y sin ella no se podrían actualizar las imágenes ni la malla de polígonos. Lo que hace la función es preguntar si hubo cambios en los identificadores de las imágenes o el identificador de la malla, de haberlo habido, se copian los datos asociados a las estructuras que el programa destina

para renderizar la escena. En ese pasaje de información se regeneran las texturas correspondientes a las imágenes que cambiaron y se vuelve a invalidar la representación visual. Sin embargo, esta vez, el llamado que se hace al display implica una renderización mucho más compleja que la que tiene lugar después de las traslaciones y rotaciones.

Lo primero que hace el display es inicializar el sistema de luces. La siguiente tarea es preguntar si la renderización es simple o compleja. El primer caso tiene lugar cuando el usuario desencadenó un evento de transformación, como una rotación o una traslación. El segundo ocurre cuando el Servidor procesó y compartió un nuevo *frame*, por lo que la malla de polígonos que hay que renderizar ya no es la misma. Finalmente se dibujan las etiquetas de texto que aparecen en la pantalla, que contienen información tal como la posición del modelo.

La renderización simple consiste en recorrer el arreglo de imágenes y dibujar las caras de la malla que son alcanzadas por la proyección de esa imagen sobre el modelo. Una vez hecho esto, se aplica la textura correspondiente.

La renderización compleja tiene que ver con obtener los datos que determinan qué cara debe dibujarse y con qué textura debe hacerse. Así como trasladar esos cálculos a la pantalla, como hace el display en modo de renderización simple.

El primer paso en este proceso es establecer algunos parámetros e inicializar las estructuras necesarias, para después descartar las caras del modelo que sobresalen de la pantalla e iterar sobre las restantes, haciendo las consultas de occlusion culling correspondientes. Como resultado del paso anterior, se obtiene el número de incidencias que tuvieron las consultas sobre cada una de las caras. Si el número de incidencias fue mayor a cero, significa que la cara es visible desde la perspectiva de la cámara actual, en dicho caso, la textura asociada a la cámara es candidata para ser la que pinte esa cara.

5 Conclusiones y Trabajos Futuros

Se realizaron pruebas para medir el rendimiento general del sistema, que permitieron detectar los cuellos de botella e identificar los valores óptimos de configuración para aprovechar al máximo las características del equipo.

Las pruebas efectuadas hicieron foco específicamente en las variables *pcDownSample* y *allowCompression*, que combinadas influyen directamente sobre el tamaño de los *frames* transmitidos y la cantidad de nubes procesadas en el Servidor.

5.1 Análisis de performance

5.1.1 Configuración del ambiente de prueba

Las pruebas fueron realizadas utilizando dos máquinas con conexión directa a través de cable UTP. En una de ellas se ejecutó el Cliente y en la otra el Servidor junto al Player. Se llamará de aquí en adelante Maquina A al equipo en el que se ejecutó el Cliente y Maquina B en la computadora donde se ejecutó el Servidor y el Player.

Características de las máquinas de prueba:

- *Maquina A*
  - SO: Windows 7 de 64 bits
  - CPU: Intel Core i3-2350M CPU 2.30 GHz
  - RAM: 4 GB
  - Modelo: HP Pavilon g6 Notebook PC
  - Tarjeta de red: Realtek RTL8188CE 802.11b/g/n WiFi Adapter
  - Tarjeta de video: Intel HD Graphics Family
  - Disco: 500 Gb

- *Maquina B*
  - SO: Windows 8.1
  - CPU: Intel Core i7-4702MQ 2.20GHz
  - RAM: 12.0 GB
  - Modelo: HP Envy 15-j108la
  - Tarjeta de red: LAN Gigabit Ethernet 10/100/1000 (conector RJ-45)
  - Tarjeta de video: NVIDIA GeForce GT 750M
  - Disco: Kingston SSD 480GB

Para realizar las pruebas que se describen en las siguientes secciones, se utilizó siempre el mismo set de datos. Todas las pruebas fueron realizadas utilizando el archivo *test.oni* que se encuentra adjunto a los materiales del proyecto. Este archivo *ONI* fue grabado utilizando un dispositivo Microsoft Kinect versión 1.

5.1.2 Variables e índices medidos

La variable *pcDownSample* se utilizar para filtrar la nube de puntos original, obteniendo una resultante con una cantidad $\frac{1}{pcDownSample^2}$ de puntos. En las pruebas realizadas hicimos variar el valor de este parámetro en el rango 1 a 8.

La variable *allowCompression* permite activar la compresión en formato *ZIP*. Habilitar esta variable implica que cada *frame* es comprimido antes de ser enviado por un Cliente y descomprimido al arribar al Servidor. Todas las pruebas de performance fueron realizadas dos veces, una con la compresión habilitada y otra con esta opción desactivada.

Estos dos parámetros fueron los únicos que sufrieron modificaciones durante las pruebas realizadas. El resto de las variables se mantuvieron siempre estáticas, tomando los valores: *persistence* en 0, *logLevel* en 5, *fps* en 20, *maxPackageSize* en 1000000, *resolutionX* en 640, *resolutionY* en 480 y *useRGBCompression* en 0.

A partir de los datos relevados, se analizan las siguientes características:

- Variación de la cantidad de puntos en la nube en función del parámetro *pcDownSample*.
- Tamaño en bytes del *frame* enviado en función de la cantidad de puntos y la compresión.
- Variación de la cantidad de *frames* enviados por segundo.
- Variación de la cantidad de mallas generadas por segundo.

5.1.3 Valores obtenidos

A continuación, se muestran dos tablas con los resultados de las pruebas realizadas. En la primera se pueden ver las mediciones obtenidas al variar el parámetro *pcDownSample* ejecutando el sistema con la compresión habilitada, mientras que en la segunda se encuentran los valores relevados al realizar las pruebas sin compresión.

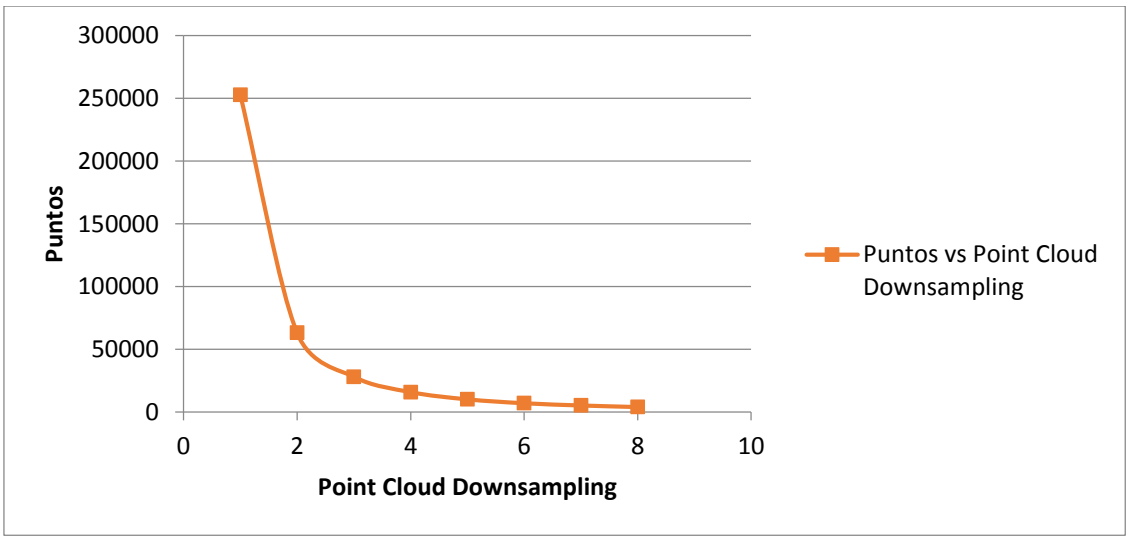
ID	Compression	PC Downsample	Tamaño frame (Bytes)	Cant puntos	Arribos p/seg	Mallas p/seg	Duración promedio de cada generación
1	1	1	3306229	252846	1,25	1,25	1817,4
2	1	2	1286755	63130	3,34	1,78	3281,18
3	1	3	913155	28132	4,82	3,08	1831,53
4	1	4	782410	15860	5,76	2,72	2188,65
5	1	5	720776	10096	6,40	6,36	727,58
6	1	6	687666	6988	6,58	6,97	318,25
7	1	7	668463	5208	6,85	6,85	190,18
8	1	8	654978	3946	6,96	6,91	123,32

ID	Compression	PC Downsample	Tamaño frame (Bytes)	Cant puntos	Arribos p/seg	Mallas p/seg	Duración promedio de cada generación
9	0	1	3956285	252880	1,84	0,79	6126,82
10	0	2	1679375	63138	4,84	1,45	3999,05
11	0	3	1259259	28128	6,62	2,83	2028,05
12	0	4	1112062	15862	7,68	2,45	2379,44
13	0	5	1042880	10097	8,21	6,05	875,53
14	0	6	1005575	6988	8,66	8,74	349,96
15	0	7	984208	5207	8,90	9,41	195,1
16	0	8	969060	3945	9,12	9,12	121,04

5.1.4 Cantidad de puntos en la nube en función del parámetro Point Cloud Downsampling

El eje horizontal representa la variación del parámetro *pcDownSample* en el rango 1 a 8, mientras que en el eje vertical se indica la cantidad de puntos en las nubes enviadas desde el Cliente al Servidor con esa configuración.

Es importante destacar que el Cliente realiza un filtro adicional, eliminando los valores de la nube de puntos que no cuentan con datos válidos, es por eso que los resultados pueden variar levemente al utilizar otra fuente de datos que no sea el archivo *ONI* utilizado en las pruebas.

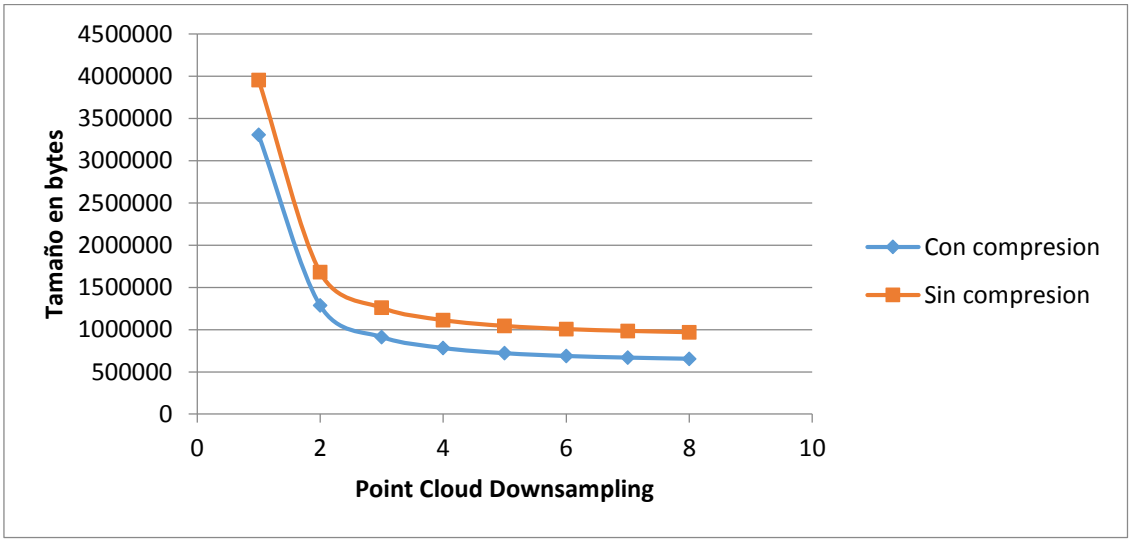


Conclusión: Por el modo en el que está implementada la reducción de puntos en el proyecto, las modificaciones en el parámetro *pcDownSample* afectan de forma exponencial la cantidad de puntos presentes en la nube enviada.

5.1.5 Tamaño en bytes de un *frame* con y sin compresión

La siguiente imagen muestra el tamaño en bytes de los *frames* que se envían desde el Cliente al Servidor diferenciando los casos de ejecución con y sin compresión.

En el eje horizontal se muestra la variación del parámetro *pcDownSample* en el rango del 1 al 8, mientras que en el eje vertical se representa el tamaño en bytes de cada *frame* enviado.



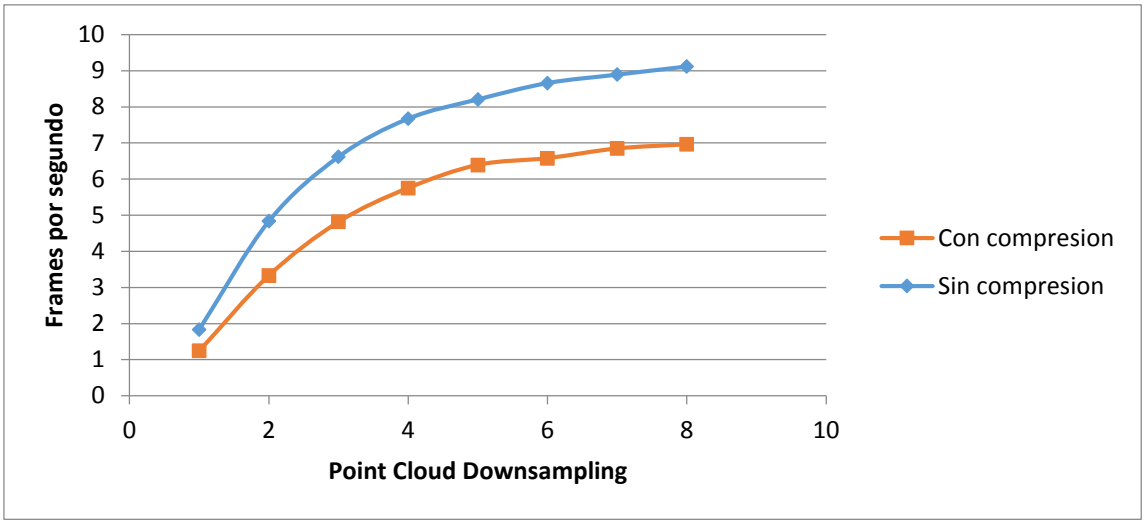
Conclusiones:

- Utilizando un valor de *pcDownSample* entre 1 y 2, el tamaño en bytes de cada *frame* está fuertemente determinado por el tamaño de la nube de puntos, mientras que con un valor de 3 o superior, el tamaño pasa a estar influenciado casi exclusivamente por el peso de la imagen.
- El método utilizado para la compresión, permite reducir hasta un 40% del tamaño total del *frame* en el caso más favorable.

5.1.6 Tasa de arribos con y sin compresión de datos

En el siguiente gráfico se analiza la tasa de envío de *frames* desde el Cliente al Servidor diferenciando los casos de ejecución del sistema con y sin compresión, en naranja y celeste respectivamente.

En el eje horizontal se representa la variación del parámetro *pcDownSample*, mientras que el eje vertical, se muestra la cantidad de *frames* recibidos por segundo con dicha configuración.



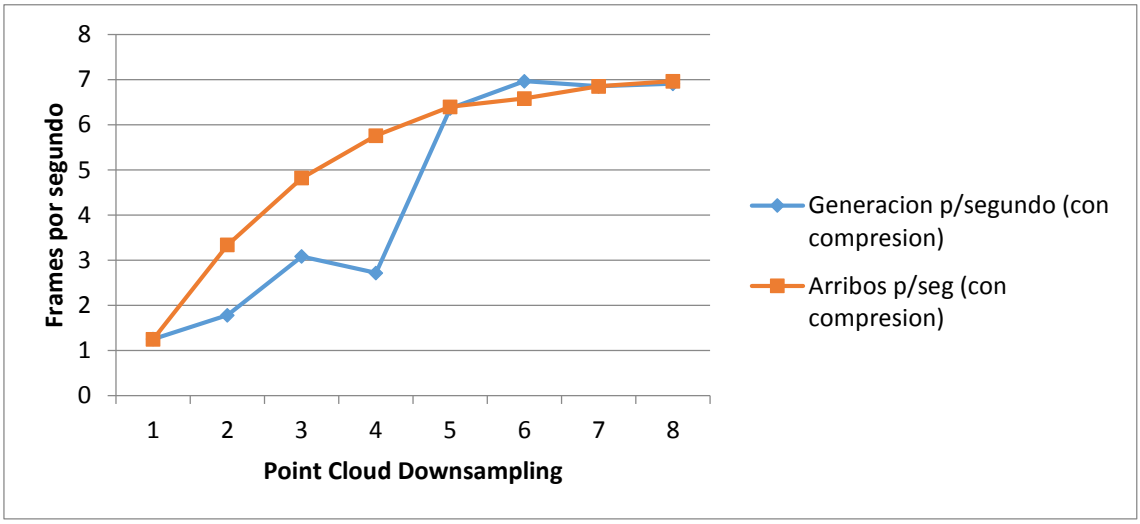
Conclusión:

La penalización, en términos de tiempo, que provoca comprimir y descomprimir los *frames* entre el Cliente y el Servidor es tal, que reduce la performance general del sistema provocando una menor tasa de generación de mallas.

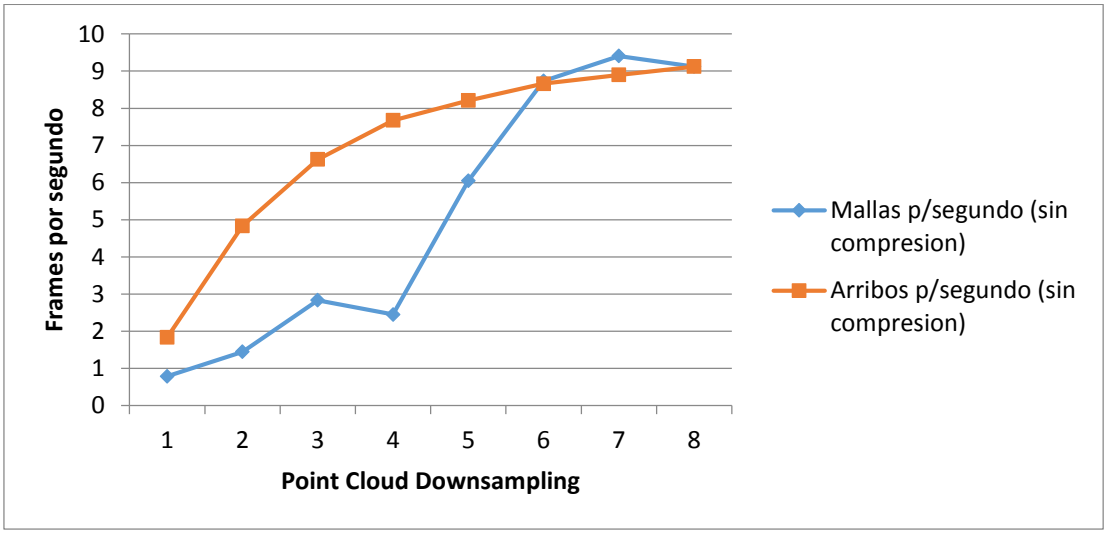
5.1.7 Generación de mallas por segundo

En este apartado se analiza particularmente el total de mallas generadas por segundo en función de la cantidad de puntos en la nube. Nuevamente en el eje horizontal se encuentra representada la variación del parámetro *pcDownSample*, mientras que, en el eje vertical, se muestra la cantidad de *frames* por segundo generados.

La primera imagen muestra los resultados obtenidos ejecutando el sistema con compresión. Se contrasta, por un lado, la gráfica de mallas generadas, utilizando el color celeste, y por otro, la cantidad de envíos de *frames* por segundo entre el Cliente y el Servidor, representada en color naranja.



En esta segunda imagen se ve el resultado de realizar las mismas pruebas, pero ejecutando el sistema sin compresión. Nuevamente en celeste se representa el total de mallas generadas por segundo y en naranja la cantidad de *frames* enviado entre el Cliente y el Servidor.



Conclusiones:

- En el caso de utilizar compresión, la configuración que permite aprovechar mejor los recursos se produce estableciendo el parámetro *pcDownSample* entre 5 y 7. Utilizando estos valores, la tasa de arribos y generación de mallas coinciden en un promedio de 6,5 ~ 7, provocando que el sistema no descarte *frames*.
- En caso de ejecutar el sistema sin compresión, la configuración que facilita aprovechar mejor los recursos se alcanza estableciendo el parámetro *pcDownSample* entre 6 y 8. Con estos valores, la tasa de arribos y generación de mallas coinciden en un promedio de 8,5 ~ 9,5.

5.2 Trabajos Futuros

Debido a lo amplio del proyecto, a la cantidad de puntos que se tocaron, y el número de factores que tuvieron que ser tenidos en cuenta para conseguir un buen resultado final, la solución implementada siempre pretendió ser, más que un sistema completamente terminado, un prototipo funcional. Es por ello que, durante la implementación del proyecto, se decidió dejar puertas abiertas para futuras extensiones y mejoras a procesos críticos dentro del flujo de trabajo del sistema, tales como el pasaje de los datos y la generación de la malla 3D.

En las siguientes secciones se abordará con mayor detenimiento algunos de los puntos que el grupo consideró más importantes para la mejora del proyecto. Muchas de esas mejoras vienen relacionadas al desempeño final del sistema, es decir, al número de *frames* por segundo que es capaz de procesar. Otras, sin embargo, tienen que ver con funcionalidades útiles para el usuario final que no fueron incluidas en el entregable final del proyecto.

5.2.1 Homogeneizar colores de Luz

Dos cámaras, posicionadas en lugares diferentes de una escena, que apuntan a un mismo sitio, pueden obtener imágenes con tonalidades ligeramente diferentes. Este suceso provoca que durante el proceso de texturización, existan partes de un mismo objeto que son pintadas con una textura más clara y otras partes que son pintadas con una textura más oscura, generando una falta de continuidad en la texturización general del objeto.

Las razones de este fenómeno se deben a las características propias de las cámaras RGB y a factores de iluminación en la escena.

Una posible mejora del sistema en el área de visualización podría implicar hacer un algoritmo de corrección de colores que haga más uniforma la texturización, en especial, en las partes de la malla donde ocurre una transición entre una textura y otra.

5.2.2 Optimización por GPU

El sistema final se compone de un grupo de módulos interconectados que se acoplan perfectamente unos con otros para conseguir el objetivo del proyecto: grabar en tiempo real escenas en tres dimensiones. En la serie de etapas por la que pasan los datos, desde la grabación de las imágenes y las nubes de puntos hasta la renderización de la malla texturizada, se conjugan una serie de procesos, unos más complejos que otros, que logran transformar esos datos en una representación visual acorde a la realidad de lo que se está filmando.

Entre los procesos más críticos, y sin dudas el más costoso, está la generación de la malla de polígonos. Debido a que actualmente este paso supone el cuello de botella en el incremento del número de *frames* por segundo que se consigue en la salida del sistema. Corresponde tratar el tema con mayor detenimiento y ahondar en posibles soluciones.

En este proyecto, la transformación de la nube de puntos en una malla de polígonos se delega a las librerías de VCG. Dado que las librerías de VCG son de código abierto, sumado a lo fácil que resultó integrarlas al proyecto del Servidor, que ya se tenía desarrollado en openFramework, y al hecho de que existe una serie de aplicaciones reconocidas en el área del tratamiento de las mallas tridimensionales tales como MeshLab, que utilizan VCG, se decidió integrar estas librerías al proyecto.

El problema ocurre básicamente por la falta de soporte que tiene VCG para el procesamiento por GPU. Pese a los intentos que se realizaron por resolver esta situación, no se logró optimizar los tiempos en que la librería convierte la nube de puntos en una malla de polígonos. Es por ello que esta etapa del procesamiento se encapsuló en una librería externa para que fuese fácilmente reemplazable por otra con un método mejor.

La alternativa que se implementó fue lanzar un conjunto de subprocesos, cada uno asociado a una nube de puntos de un *frame* diferente, para realizar la mayor cantidad de trabajo concurrente en el menor tiempo posible.

Otro aspecto sobre el que se puede hacer una mejora significativa del rendimiento del sistema viene por el lado del número de puntos que se toman como referencia para generar la malla.

Actualmente, el sistema permite, mediante el archivo *XML* de configuración, bajar esa cantidad de puntos, multiplicándola por un factor de $1/2, 1/4, 1/8$ y así sucesivamente. Pero el problema surge en el hecho de que los puntos que son descartados se encuentran a intervalos regulares en la matriz de puntos que proporcionan los sensores de profundidad, sin seguir una lógica un poco más inteligente. Por ejemplo, dado que en una escena suele haber superficies planas, tales como paredes o mesas, sería ideal quitar puntos intermedios en los planos que forman esas superficies y no en las áreas de la malla que representan cosas más complejas como rostros o vegetación.

Entonces, la introducción de un algoritmo de selección de puntos podría introducir grandes mejoras, siempre que el procesamiento que supone esa lógica no degrade más el desempeño del sistema.

5.2.3 Transferencia de Datos

Como ya se mencionó, cada uno de los aspectos críticos del proyecto fue atacado de la mejor manera posible, y en aquellos casos en donde el resultado no fue lo suficientemente bueno, se decidió encapsular la lógica correspondiente en una librería. Cada de una de ellas fácilmente reemplazable en el futuro por otra con resultados mejores, y que quizás, ataque el problema de manera completamente diferente. Uno de esos casos es la lógica para publicar los resultados del procesamiento que hace el Servidor. Este punto constituye el puente entre el Servidor y el Reproductor, y es gracias a él que se consigue la visualización en tiempo real.

La decisión que se tome aquí podría limitar la forma en la que funciona el sistema en la parte más importante para el usuario final, que es la visualización de la escena. A grandes rasgos hay dos opciones posibles para compartir los datos, una es utilizar la red y la otra es memoria compartida. Conviene analizar un poco que supone cada una de estas dos alternativas.

Si se utiliza la red como medio para transferir los datos entre un programa y otro, se posibilita que éstos corran en máquinas diferentes, dejando que ambos hagan uso de una mayor cantidad de recursos para trabajar. Pero tiene como desventaja que los datos que se transfieren son susceptibles a errores. También genera que la información se transfiera más lentamente e introduce cotas en la cantidad de datos que se pueden transmitir en simultáneo.

Si se utiliza la memoria compartida, el problema de la introducción de errores durante la transmisión, las limitaciones de cantidad de información y la velocidad de transmisión dejan de ser un problema. La desventaja radica en la capacidad de procesamiento que se puede asignar a un programa y a otro, dado que ejecutan en una misma terminal.

Actualmente, el proyecto utiliza esta última arquitectura para el pasaje de los datos entre el Servidor y el Reproductor. Pero si se logran resolver los problemas mencionados de un sistema distribuido, se podría conseguir un resultado muy bueno, balanceando la carga de trabajo.

Para atacar el problema de la introducción de errores, se podría pensar en utilizar alternativas separadas para enviar los diferentes tipos de datos. Por un lado, se podría emplear TCP para enviar los datos críticos, como los identificadores de *frame*, las dimensiones de las imágenes y el número de caras de la malla. Por otro lado, se podría emplear UDP para transferir el arreglo de puntos, caras y píxeles, que pueden contener valores con error, sin que ello arruine la renderización de la escena en un instante dado.

El problema del volumen de datos transferidos puede solucionarse si se utiliza algún tipo de algoritmo de compresión, tanto para el arreglo de puntos, caras y píxeles, que representan prácticamente el cien por ciento del volumen de datos que se transmiten en cada *frame*. La compresión puede ser con pérdida o sin pérdida, cuyo último caso podría incluir a los datos críticos que se mencionaron antes. La compresión con pérdida sigue siendo una alternativa viable, dado que la introducción de pequeños errores en las coordenadas de los puntos o en el valor RGB de las imágenes, no afecta tanto la renderización final que hace el Reproductor de la escena.

5.2.4 Persistencia del resultado final

Tal como se describió anteriormente, en la versión actual del proyecto el único modo de consumir la salida del sistema es a través de la librería externa *SharedMemory.dll*. Esto implica, que solo se puede consumir la información si tanto el Servidor como los Clientes están en ejecución y existen *frames* siendo procesados.

Sería deseable agregar una funcionalidad que permitiera persistir la información de forma tal que esta pudiera ser leída posteriormente sin la necesidad de mantener el sistema en ejecución.

La implementación de esa funcionalidad implica definir primeramente un formato de archivo que permita almacenar tanto la malla como las texturas.

Dado el tipo de contenido, es de suponer que el tamaño del archivo que se genera crece a una tasa muy alta, por lo que sería necesario contemplar algún mecanismo de compresión.

Por otra parte, resulta importante que la escritura de dicho archivo tenga el menor impacto posible en la performance del Servidor. La utilización del disco siempre es costosa, por lo que es importante implementar una solución que no genere un cuello de botella.

Por último, un formato de archivo con estas características debería estar pensado de forma tal que sea simple de leer, y contemple también la posibilidad de hacer saltos en el tiempo hacia adelante y hacia atrás.

5.3 Conclusiones

Dado que uno de los requerimientos originales del proyecto fue utilizar la mayor cantidad de cámaras posibles, encontramos que la elección de openFramework fue una buena decisión ya que permitió incorporar complementos que simplificaron varios aspectos del desarrollo.

Haber basado el proyecto exclusivamente en Microsoft Kinect, hubiese repercutido en obtener un resultado más performante, dado que existen librerías desarrolladas específicamente para este hardware que implementan parte del trabajo realizado.

El alcance del proyecto fue demasiado extenso y terminó sobrepasando ampliamente la estimación original de horas planificadas.

Si bien la utilización de la librería VCG Library simplificó la generación de las mallas, la falta de soporte para GPU limitó la performance del sistema.

El hecho de generar toda la malla en cada *frame*, simplificó la implementación del sistema, pero perjudicó el framerate final.

La implementación del Calibrador y el Reproductor por intermedio de OpenGL resultó ser mucho más compleja que la implementación inicial desarrollada en Unity.

La utilización de C++ en la implementación implicó la necesidad de tener que administrar la memoria del sistema, haciendo que el desarrollo fuese más complicado.

Haber escogido una arquitectura distribuida permitió dividir el procesamiento en diferentes terminales, mejorando así el desempeño del sistema.

La decisión de adoptar una arquitectura modular simplificó el proceso de desarrollo ya que permitió concentrarse en la solución global del sistema y generar implementaciones concretas fácilmente sustituibles por otras más performantes.

6 Proceso de Desarrollo

El proyecto comenzó con un estudio de las diferentes tecnologías que se encontraban disponibles para atacar las problemáticas presentadas.

En un primer momento se definieron cuáles eran los puntos más importantes a tratar, de ahí surgieron tres temas: las cámaras RGB, los sensores de profundidad y las herramientas de software que permitían trabajar con estos dos dispositivos.

Con las cámaras RGB se comenzó estudiando los problemas de calibración intrínseca y extrínseca. Por el lado de los sensores de profundidad se investigó sobre el Kinect, dado que era el sensor que se tenía disponible, pero teniendo en cuenta que el proyecto debía soportar múltiples tipos de sensores. Buscando sobre estos puntos, se encontró openFramwork que posee addons de OpenNI y OpenCV, para el manejo de sensores 3D y cámaras RGB respectivamente, además de facilitar la implementación con C++.

Una de las primeras decisiones que se tomaron fue determinar con qué entorno de desarrollo se iba a trabajar, se evaluaron dos opciones: Visual Studio o CodeBlocks. La herramienta que determinó que entorno utilizar en ese momento fue PCL, la cual surgió del estudio de las nubes de puntos, y la manera de tratar con los problemas que estas planteaban. Dado que era compatible con Visual Studio y openFramework, se decidió utilizar este entorno para el desarrollo.

Teniendo el IDE y las herramientas necesarias para interactuar con los dispositivos se comenzaron a realizar los primeros prototipos. El objetivo era obtener la nube de puntos y las imágenes que los mismos proporcionaban. Se realizaron pruebas con PCL utilizando las diferentes operaciones que la librería facilita: generación de mallas, alisados, filtros, disminución de resolución y compresión. Los últimos dos puntos atacaban otro de los grandes problemas del proyecto, el tamaño de los datos manejados.

Realizando pruebas de generación de mallas bajo diferentes configuraciones, se halló otro inconveniente, el tiempo que se necesitaba para generar la malla de un solo *frame* con la información de un solo sensor, era del orden de segundos. Considerando que para realizar un video es necesario tener varios *frame* por segundo, éste era un gran problema a atacar. Pensando una solución, surge la discusión de crear un algoritmo propio y más eficiente para la generación de una malla.

Luego de investigar, se llegó a la conclusión de que pensar este algoritmo se iba del alcance del proyecto, donde si bien esto era parte importante, no era la totalidad del mismo. Otra opción era continuar buscando librerías que facilitaran la creación de mallas. Meshlab fue una de las opciones encontradas, si bien es un programa con interfaz de usuario, brinda la posibilidad de generar mallas por medio de scripts ejecutados desde consola. Meshlab fue una herramienta muy utilizada durante el desarrollo del proyecto, dado las diferentes operaciones que permite aplicarle a una malla y la capacidad de poder visualizarlas en tiempo real. Tras haber realizado diversas pruebas de algoritmos y configuraciones, se encontró uno en particular denominado Ball Pivoting, que daba resultados satisfactorios en un tiempo razonable. Se implementó un prototipo que invocaba por intermedio de un script el algoritmo para poder ejecutarlo sin interacción de usuario.

En este punto, al haber descartado temporalmente PCL y dado la gran cantidad de problemas de compatibilidad que estaba dando Visual Studio, se optó por continuar el desarrollo del sistema con CodeBlocks. Se incorporó en este momento GitHub, lo que permitió al equipo trabajar de una manera más ordenada. Al igual que las otras herramientas utilizadas, tuvo una curva de aprendizaje.

En paralelo al estudio del procesamiento de la información de los sensores y la generación de las mallas, fue necesario investigar técnicas de texturización. Surgió la idea de unificar las imágenes de las diferentes cámaras para generar una única textura que cubriera la malla. Esta idea se descartó luego de analizar las dificultades que presentaba unificar las imágenes y posicionarlas en la malla. Las técnicas más conocidas de mapeado de texturas requieren saber la correspondencia de la imagen con la figura 3D a texturizar, para el problema planteado esto es algo que cambia constantemente y dificulta mucho determinar dicha correspondencia. Investigando sobre otras técnicas, fue que se encontró la idea de proyectar la imagen sobre la malla, como si se tratase de un proyector.

En este momento del proyecto, luego de haber realizado el estudio de algunos puntos críticos y tener una panorámica de las tecnologías a utilizar, fue necesario visualizar como estaría compuesto el sistema. Aquí surgen el Cliente y el Servidor. El Cliente es el que se encarga de obtener la información de las cámaras y sensores, mientras que el Servidor de procesarla. Con estos dos subsistemas no era posible que un usuario final viera el resultado, para esto fue necesario crear el Reproductor, que sería el prototipo del proyecto. De esta arquitectura surgen nuevos problemas: la comunicación entre los componentes, el procesamiento de la información, la sincronización, el almacenamiento, la performance, entre otros.

La transferencia de información entre los componentes fue uno de los temas atacados, dado que era demasiado trabajo para una sola computadora tener todo el procesamiento de los Clientes, el Servidor y el Reproductor. Por esta razón fue necesaria una arquitectura distribuida. Con este nuevo enfoque, se definió que iban a haber un conjunto de Clientes, donde cada uno tendría conectado un conjunto de cámaras RGB y sensores de profundidad. En un principio, cada uno de estos agruparía la información de sus dispositivos y los enviaría al Servidor sin un mayor procesamiento. El paquete enviado tendría información para la sincronización en el tiempo y un identificador para cada dispositivo.

Si bien se consideró UDP como protocolo de transferencia por su velocidad, se decidió usar TCP porque garantiza el envío del identificador y el momento de cada paquete, datos que no se pueden perder. El Servidor, por su parte, tendría un hilo por cada Cliente, el cual recibe la información, unificándola con la de los otros Clientes para un instante dado. Además de unir los datos, es el encargado de generar las mallas y publicar la información para el Reproductor.

Al mismo tiempo que se desarrollaban el Cliente y el Servidor, se investigó la manera de unificar los datos de los diferentes dispositivos, para que convivan en un mismo sistema de referencia. Era fundamental que las nubes de puntos de los diferentes sensores estuvieran calibradas, para que la malla de como resultado una representación en 3D de la escena grabada. El desarrollo del Calibrador inició como un prototipo de Unity [63] dada la facilidad que brinda al momento de trabajar con objetos 3D. Se realizó un lector de archivos *XYZ* para cargar las nubes de puntos de los sensores, posteriormente fue necesario transformarlas para ubicarlas en el mismo sistema de referencia. Encontrar estas transformaciones no fue una tarea trivial.

La primera idea que surgió fue seleccionar tres puntos de una nube, y sus correspondientes tres puntos en la otra, para poder hallar la transformación entre ellos. El problema con este método, era que seleccionar los mismos puntos en ambas nubes se tornaba una tarea extremadamente difícil por la cantidad de puntos que había. Otra idea era seleccionar un conjunto de puntos y promediarlos, pero esto introducía mucho error, lo que hacía que al transformar las nubes no solamente se trasladaran y rotaran, sino que también se deformaran. Por último, se optó por transformar las nubes, rotándolas y trasladándolas con el teclado hasta que una nube coincida con la otra, si bien la solución no deformaba las nubes, lograr calibrarlas no resulta practico.

Una discusión que se dio en la primera mitad del proyecto era si la aplicación iba a poder soportar el tiempo real o no. La cantidad de procesamiento que implicaba generar un *frame* daba la impresión de que era una tarea demasiado compleja poder mostrar una escena en tiempo real, por lo que se optó por generar una escena grabada y luego levantarla desde disco.

Implementando la idea de generar un sistema que leyera archivos de disco, el equipo descubrió que en realidad realizarlo en tiempo real no era demasiado diferente. Las razones eran que, para hacerlo en diferido, se necesitaba un programa que levante la información de disco y otro que lo reproduzca. Lo cual requería implementar un nuevo sistema encargado de cargar la información y publicarla. Esta idea se descartó y surgió la que sería la idea final del proyecto. El Cliente iba a tener dos modos, uno que lee la información de los dispositivos y otro que lee la información almacenada en disco y la trasmite al Servidor. Esto trajo como consecuencia la optimización del sistema.

Retomando la implementación del Calibrador, se decidió dejar de utilizar Unity dado que, si bien la implementación que se hizo solucionaba el problema, al ser un software que no era de código abierto, iba en contra de uno de los requerimientos del proyecto. Sin embargo, las ideas realizadas con Unity se mantuvieron, pero esta vez se desarrollaron con OpenGL. De nuevo, esto implicaba aprender a utilizar otra herramienta, que a diferencia de la anterior, el desarrollo requiere mayor esfuerzo.

Una vez obtenidas las mismas funcionalidades que se tenían con Unity, fue importante saber de qué manera calibrar las cámaras RGB. Luego de discutir algunas alternativas, se decidió que la calibración consistía en rotar y trasladar la malla para cada imagen obtenida, guardando la transformación en la que la imagen texturiza la parte de la escena correcta. La calibración de las imágenes da como resultado una matriz por cada cámara RGB. Los dos tipos de calibración definieron la estructura del Calibrador de forma natural. En primer lugar, se realiza la calibración de las nubes de puntos, para luego con la malla unida, realizar la calibración de las texturas. La información de calibración se graba en un archivo *XML* de configuración, leído por los Clientes y el Reproductor.

En paralelo al Reproductor, el Servidor se fue implementando con las tareas acordadas: obtener los datos transformados de los Clientes, ordenarlos cronológicamente según el momento en el que fueron grabados, generar una malla unificada y publicarla junto a las imágenes. En los casos en que no haya información de un Cliente para un instante determinado, el Servidor utiliza el último *frame* recibido de éste para completar la nube. Esta decisión se tomó para darle continuidad al Reproductor y que no existan parpadeos en la salida generada.

Hasta el momento, la generación de la malla implicaba que el Servidor guardara en un archivo en disco la nube de puntos unificada y por intermedio de un script se ejecutara el algoritmo de generación para Meshlab Server. Luego, la malla generada se cargaba en memoria nuevamente y quedaba pronta para publicarse. Este método es muy poco performante debido a los accesos a disco, por lo que se debió buscar una solución alternativa. Aquí fue donde investigando se encontró que Meshlab estaba basado en un conjunto de librerías denominadas VCG libraries. Inspeccionando el código fuente de Meshlab se pudo obtener el método que utilizaba para generar la malla y de esta manera se logró optimizar el trabajo evitando los accesos a disco.

Para la parte de la publicación de los datos, se pensó en almacenar archivos en disco para que el Reproductor los leyera. Con el mismo criterio utilizado anteriormente, esta opción se descartó y surgieron dos alternativas, publicar la información en un puerto o utilizar memoria compartida. La primera se eliminó porque no se quería introducir más latencia al procesamiento, por lo que la otra opción resultó la más correcta. Sin embargo, ésta tenía la desventaja que el Servidor y el Reproductor debían ejecutar en la misma máquina.

En este punto del proyecto se tenían varias partes que estaban sujetas a decisiones que no resultaban ser las más optimas o que se debían mejorar, pero que dada la extensión del proyecto no se tenía el tiempo para investigarlas. Aquí surgió la idea de encapsular estas partes en librerías cargadas dinámicamente, para que fuesen fácilmente remplazables. Se decidió que la generación de la malla, la publicación y transferencia de datos no quedaran atadas a la solución propuesta.

Avanzando con la implementación del Reproductor, el cual nació inicialmente como una extensión del Calibrador que luego se separó. El Reproductor usa la librería encargada de leer los datos de memoria compartida, y junto con la configuración obtenida del Calibrador dibuja los *frames* en pantalla. La principal diferencia con el Calibrador es que ya no se debe dibujar únicamente un solo *frame*, sino que se deben dibujar todos los que el Servidor publica.

Atacando la texturizacion, se encontró que las imágenes compartidas no se encontraban con el formato correcto para ser utilizadas como texturas de OpenGL, lo que llevó a investigar sobre este tema y corregir las dimensiones de las imágenes. La técnica de proyección de texturas tiene dos problemas conocidos: se texturizan todos los triángulos en el frustum y ambas caras de los triángulos de la malla. Para el primer problema, se investigó sobre el método denominado Oclussion Culling y posteriormente se realizó su implementación. El segundo problema se resolvió calculando las normales de las caras y dibujando únicamente el lado que se encuentra en dirección a la cámara. Resueltos ambos desafíos, se observó que el tiempo necesario para aplicar este método a cada *frame* era muy alto y limitaba enormemente el framerate, utilizando el mismo criterio que en otras partes, se encapsuló esta lógica en una librería.

Si bien Oclussion Colling se encapsuló en una librería, se buscó optimizar el Reproductor de diferentes maneras. En primer lugar, cuando la escena es movida por el usuario para verla desde diferentes ángulos, se bloquea la actualización de los *frames*. En segundo lugar, se dividió la escena en cubos, los cuales se dibujan en tiempos diferentes dependiendo que tanto cambia la malla comprendida en esos cubos. En tercer lugar, el dibujado de una cara se hace en base al resultado del Oclussion Culling de sus vecinas, si las vecinas se dibujan, esa cara automáticamente también se dibuja. Finalmente, caras aisladas tampoco se dibujan. Con estas

técnicas se buscó disminuir la cantidad de cálculos por *frame* y así aumentar el framerate del Reproductor.

Las optimizaciones del sistema Cliente y Servidor se dividieron en dos partes, por un lado, distribuir el procesamiento lo mejor posible entre Cliente y Servidor, y por otro optimizar cada componente. En primer lugar, se decidió que cada Cliente sea el responsable de transformar cada una de sus nubes de puntos con la matriz de transformación generada por el Calibrador, ahorrándole procesamiento al Servidor. Por otro lado, el grupo decidió paralelizar la generación de las mallas con varios hilos de ejecución. La cantidad de hilos se hizo configurable al igual que otras características, para sacar mayor ventaja de las prestaciones de la maquina donde está ejecutando el sistema.

Se probó también aplicar compresión para disminuir las tasas de transferencias de datos. El tiempo que requería la compresión y descompresión con los algoritmos utilizados, no logro mejorar el tiempo general de transferencia, por esto es que se decidió encapsular esta funcionalidad en una librería. Al Servidor, además, se le agregó un manejo de distintos niveles de registro de errores, para facilitar el seguimiento del estado del mismo, así como también, una interfaz para que los usuarios puedan ver las imágenes de las cámaras RGB que está recibiendo.

La calibración intrínseca y la utilización de PCL, se retomaron en la recta final del proyecto. La primera de éstas, se incorporó agregando las matrices para las correcciones de las lentes de las cámaras. Si bien se había investigado al comienzo y se llegó a implementar un algoritmo de calibración intrínseca utilizando OpenCV, los resultados obtenidos no eran los esperados. Por lo que se optó por utilizar el programa GML C++ Camera Calibration para obtener las matrices. Por otro lado, fue posible incorporar opcionalmente PCL gracias a la libertad que brinda tener la generación de la malla encapsulada en una *DLL*. Como se mencionó anteriormente, PCL funciona con Visual Studio, por lo tanto, la librería se realizó con esta herramienta.

Por último, a lo largo del proyecto, sobre todo en la primera y última parte, se realizó la documentación. En un principio se documentó el estado del arte con los diferentes puntos estudiados y al final se entró en detalle en la implementación. Además, se hizo un estudio de performance de la transferencia de datos entre Cliente y Servidor. El estudio se hizo modificando la cantidad de datos y se obtuvo la cantidad optima de los mismos en la que se debe transferir.

7 Glosario

7.1 Frustum

Es una porción del espacio delimitada por dos planos paralelos que atraviesan un sólido. Los tipos de frustum más conocidos son los piramidales y los cónicos.

El frustum de vista es un volumen 3D que define como los modelos son proyectados desde la cámara al espacio de proyección. Los objetos deben estar dentro del volumen 3D para ser visibles.

Se utiliza en la proyección de perspectiva, lo que hace que un objeto que se encuentra más cerca de la cámara se vea más grande que los objetos más distantes. Para este tipo de proyección de perspectiva el frustum es representado como una pirámide acotada por dos planos paralelos.

El frustum de vista está definido por un campo de visión, FOV de sus siglas en inglés, y la distancia al plano más cercano y al más lejano [64].

7.2 Add-on

Un add-on es una extensión de software que añade funciones adicionales a un programa. Se pueden extender ciertas funcionalidades ya existentes o añadir funcionalidades completamente nuevas [65].

7.3 Pipeline

El concepto de *pipeline* aplicado a un trabajo, consiste en dividir el mismo en un conjunto de tareas secuenciales [66]. Dado que puede haber diferentes actores encargados de cada tarea, esto permite ejecutar trabajos en paralelo.

7.4 Frame de datos

Es la estructura de datos que se trasmite del Cliente al Servidor. Contiene el conjunto de imágenes y nubes de puntos de todas las cámaras conectadas al Cliente en un instante dado, así como los metadatos adicionales que permiten identificar esta información en el sistema.

7.5 Frame de video

Un video es una secuencia de imágenes donde cada imagen sucede a la anterior en la línea del tiempo en la que fue tomado el video. Cada una de estas imágenes son denominadas *frame* [67].

7.6 Framerate

El termino framerate refiere a la cantidad de *frames* por segundo que se muestran en un video [68].

7.7 TCP, UDP

Ambos son protocolos de la capa de trasporte del modelo de capas de la red. UDP está enfocado en la velocidad de entrega de los paquetes que se envían, mientras que TCP se basa en la garantía de que los paquetes lleguen. La elección de que protocolo utilizar depende del tipo de aplicación que se esté construyendo. El factor decisivo a la hora de optar por uno de los dos, es si se requiere de una garantía de entrega o no [69].

8 Bibliografía

[44] 3DVia, «Modeling for games, Why does it look so good?,» [En línea]. Available: http://www.3dvia.com/blog/modeling-for-games-why-does-it-look-so-good/. [Último acceso: 29 Setiembre 2015].

[27] A. Ltd, *3D-Studio File Format,* 1997.

[69] B. P. Kamal Hyder, Embedded Systems Design using the Rabbit 3000 Microprocessor: Interfacing, Networking, and Application Development, OXFORD: ELSEVIER.

[47] C. Everitt, «Projective Texture Mapping,» 2001.

[45] C. o. A. a. S. The Ohio State University, «Mapping Techniques,» 13 Abril 2004. [En línea]. Available: http://accad.osu.edu/~midori/Materials/texture_mapping.htm. [Último acceso: 3 Setiembre 2015].

[30] C. S. Division, «Berkeley University of California,» [En línea]. Available: https://www.cs.berkeley.edu/~sequin/CS184/IMGS/Breps.GIF. [Último acceso: 29 Setiembre 2015].

[52] Cinder, «Cinder,» Cinder, 29 Setiembre 2015. [En línea]. Available: http://libcinder.org. [Último acceso: 17 Octubre 2015].

[67] CISCO, «Cisco Video and TelePresence Architecture Design Guide,» 30 3 2012. [En línea]. Available: http://www.cisco.com/c/en/us/td/docs/voice_ip_comm/uc_system/design/guides/videodg/vidguide/basics.html#wp1056105. [Último acceso: 9 12 2015].

[25] D. C.-K. Shene, «Department of Computer Science Michigan Technological University,» [En línea]. Available: http://www.cs.mtu.edu/~shene/COURSES/cs3621/SLIDES/Mesh.pdf. [Último acceso: 03 12 2015].

[11] D. Editors, «3D Scanning 101,» deskeng, 14 Diciembre 2010. [En línea]. Available: http://www.deskeng.com/de/3d-scanning-101/. [Último acceso: 18 Setiembre 2015].

[16] D. Hoiem, «How the Kinect Works,» University of Illinois, Illinois, 2011.

[31] D. Lawlor, «3D Objects: Mesh Modeling & File Formats,» College of Engineering & Mines, 2013. [En línea]. Available: https://www.cs.uaf.edu/2013/spring/cs493/lecture/02_07_meshes.html. [Último acceso: 17 Octubre 2015].

[9] D. R. McMurtry, «Coordinate measuring machine». US Patente US4333238 A, 1982 Junio 8.

[10] F. B. L. C. G. G. a. M. R. J-Angelo Beraldin, «Active 3D sensing,» *Centro di Ricerche Informatiche per i Beni Culturali, SCUOLA NORMALE SUPERIORE PISA,* vol. QUADERNI 10, pp. 1-10, 2000.

[5] F. G. Paolo Cignoni, «VCG Library,» Visual Computing Lab of the Italian National Research Council - ISTI, [En línea]. Available: http://vcg.isti.cnr.it/vcglib/index.html. [Último acceso: 22 Setiembre 2015].

[21] F. I. T. J. W. R. C. Y. H. N. U. Bernardini, J. Mittleman, H. Rushmeier, C. Silva y G. Taubin, «The ball-pivoting algorithm for surface reconstruction,» *Visualization and Computer Graphics, IEEE Transactions on,* vol. 5, nº 4, pp. 349 - 359, 1999.

[58] G. a. M. Lab, «graphics.cs.msu.ru,» Graphics and Media Lab, 31 Marzo 2014. [En línea]. Available: http://graphics.cs.msu.ru/en/node/909. [Último acceso: 18 Setiembre 2015].

[59] G. Borenstein, «User Detection,» de *Making Things See: 3D vision with Kinect, Processing, Arduino, and MakerBot*, Canada, Makermedia, 2012, p. 204.

[60] G. Code, «JPEG image compressor and decompressor classes in two C++ source files,» 20 Diciembre 2015. [En línea]. Available: https://code.google.com/p/jpeg-compressor/.

[54] G. R. a. M. Adler, «zlib Home Site,» 20 Diciembre 2015. [En línea]. Available: http://www.zlib.net/.

[32] G. Turk, *PLY polygon files,* Stanford: The Board of Trustees of The Leland Stanford, 1994.

[43] Gimp, «TILABLE TEXTURES,» [En línea]. Available: http://www.gimp.org/tutorials/Tileable_Textures/. [Último acceso: 29 Setiembre 2015].

[17] H. F. Andrew Maimone, «Reducing Interference Between Multiple Structured Light Depth Sensors Using Motion,» *Virtual Reality Short Papers and Posters (VRW), 2012 IEEE,* nº 10.1109/VR.2012.6180879, pp. 51 - 54, 2012.

[66] I. L. Zebo Peng, «Linkoping Universitet - Department of Computer and Information Science,» 28 10 2011. [En línea]. Available: https://www.ida.liu.se/~TDTS08/lectures/11/lec3.pdf. [Último acceso: 9 12 2015].

[68] J. J. P. Dan Oja, Computer Concepts: Illustrated Introductory, 8 edicion ed., Boston, Massachussetts: CENGAGE Learning.

[36] J. Richard S. Wright, «Open GL Super Bible,» [En línea]. Available: http://opengl.czweb.org/ch06/145-149.html. [Último acceso: 03 12 2015].

[6] J. U. K. K. B. B. Barbara London, Photography (7th Edition), London: Prentice Hall, 2001.

[35] L. K. a. H.-P. S. Swen Campagna, «Directed Edges - A Scalable Representation for Triangle Meshes,» *Journal of Graphics tools,* vol. 3, nº 4, pp. 1-11, 1998.

[34] M. B. a. E. L. Finch, *COLLADA,* Sony Computer Entertainment Inc., 2008.

[62] M. Ben-Chen, «Geometry Processing Algorithms,» [En línea]. Available: http://graphics.stanford.edu/courses/cs468-12-spring/LectureSlides/06_smoothing.pdf. [Último acceso: 6 01 2016].

[28] M. Ben-Chen, «Stanford Computer Graphics Laboratory,» [En línea]. Available: http://graphics.stanford.edu/courses/cs468-12-spring/LectureSlides/02_Mesh_Data_Structures.pdf. [Último acceso: 26 Setiembre 2015].

[22] M. D. Buhmann, «Radial basis functions,» *Acta Numerica 2000 ,* vol. 9, pp. 1-38, 2000.

[50] M. H. D. L. S. G. E. S. a. J. C. P. By John D. Owens, «GPU Computing,» *Graphics Processing UnitsVpowerful, programmable, and highly parallelVare,* vol. 96, nº 5, 2008.

[15] M. J. a. T. P. Jan Smisek, «3D with Kinect,» *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on,* nº 12504009, pp. 1154 - 1160, 2011.

[48] M. J. K. Randima Fernando, The Cg Tutorial, Addison-Wesley Professional, 2003.

[13] M. R. C. S. Branislav SOBOTA, «3D SCANNER DATA PROCESSING,» *Journal of Information, Control and Management Systems,* vol. 7, nº 2, pp. 1-2, 2009.

[26] MeshLab, «MeshLab,» [En línea]. Available: http://meshlab.sourceforge.net/. [Último acceso: 22 Setiembre 2015].

[14] Microsoft, «Kinect for Windows Sensor Components and Specifications,» Microsoft, [En línea]. Available: https://msdn.microsoft.com/en-us/library/jj131033.aspx. [Último acceso: 16 Setiembre 2015].

[38] Microsoft, «Triangle Fans (Direct3D 9),» [En línea]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/bb206271(v=vs.85).aspx. [Último acceso: 29 Setiembre 2015].

[37] Microsoft, «Triangle Strips,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/bb206274(v=vs.85).aspx. [Último acceso: 29 Setiembre 2015].

[64] Microsoft, «What Is a View Frustum?,» Microsoft, [En línea]. Available: https://msdn.microsoft.com/en-us/library/ff634570.aspx. [Último acceso: 17 Octubre 2015].

[39] N. Connio, «Computación Gráfica Avanzada: Técnicas Poligonales,» 1 Setiembre 2010. [En línea]. Available: http://www.fing.edu.uy/inco/cursos/cga/Clases/2010/TecnicasPoligonales.pdf. [Último acceso: 29 Setiembre 2010].

[57] OpenCV, «docs.opencv.org,» docs.opencv.org, 2 Agosto 2015. [En línea]. Available: http://docs.opencv.org/doc/tutorials/calib3d/camera_calibration/camera_calibration.html. [Último acceso: 16 Setiembre 2015].

[1] openFrameworks, «openframeworks,» openFrameworks, 24 Setiembre 2015. [En línea]. Available: http://openframeworks.cc/. [Último acceso: 14 Octubre 2015].

[2] openGL, «opengl.org,» Khronos Group, 3 Octubre 2015. [En línea]. Available: https://www.opengl.org/. [Último acceso: 14 Octubre 2015].

[41] Oxford, «Oxforddictionaries,» [En línea]. Available: http://www.oxforddictionaries.com/es/definicion/ingles_americano/texture. [Último acceso: 3 Setiembre 2015].

[40] P. Heckbert, «Survey of Texture Mapping,» *Computer Graphics and Applications, IEEE,* vol. 6, nº 11, pp. 56 - 67, 1986.

[23] P. L. a. K. Salkauskas, «Surfaces generated by moving least squares methods,» *Math. Comp,* vol. 37, pp. 141-158, 1981.

[46] P. N. y. W. S. Néstor Calvo, Escritor, *Modelado de la terminación superficial.* [Performance]. Centro de Investigación de Métodos Computacionales, 2011.

[56] parsley, «code.google.com,» parsley, 10 Setiembre 2010. [En línea]. Available: https://code.google.com/p/parsley/wiki/ExtrinsicCalibration. [Último acceso: 13 Setiembre 2015].

[55] parsley, «code.google.com,» parsley, 23 Setiembre 2010. [En línea]. Available: https://code.google.com/p/parsley/wiki/IntrinsicCalibration. [Último acceso: 13 Setiembre 2015].

[51] Processing, «Processing,» Processing, 10 Octubre 2015. [En línea]. Available: http://processing.org/. [Último acceso: 17 Octubre 2015].

[61] R. Bridson, «Fast Poisson Disk Sampling in Arbitrary Dimensions,» 2007.

[8] R. Matthews, «Wake Forest University,» 15 Junio 2013. [En línea]. Available: http://users.wfu.edu/matthews/misc/graphics/formats/formats.html. [Último acceso: 13 Setiembre 2015].

[19] S. H. M. Z. Renoald Tang, «Surface Reconstruction Algorithms: Review and Comparison,» 2013.

[18] S. I. O. H. D. M. S. H. a. D. K. D.A. Butler, «Shake'n'sense: reducing interference for overlapping structured light depth cameras,» *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems,* p. 1933–1936, 2012.

[53] Structure, «OpenNI,» Structure, 9 Setiembre 2015. [En línea]. Available: http://structure.io/openni. [Último acceso: 17 Octubre 2015].

[3] Structure, «structure.io,» Structure, 9 Setiembre 2015. [En línea]. Available: http://structure.io/openni. [Último acceso: 14 Octubre 2015].

[24] T. K. D. a. J. Sun, «An Adaptive MLS Surface for Reconstruction with Guarantees,» *Symposium on Geometry processing,* 2005.

[20] T. K. D. J. R. S. Siu-Wing Cheng, Delaunay Mesh Generation, London: CRC Press, 2012.

[65] T. Terms, «techterms.com,» techterms.com, 22 Diciembre 2008. [En línea]. Available: http://techterms.com/definition/addon. [Último acceso: 16 Setiembre 2015].

[42] The Ohio State University - Computer Science and Engineering, [En línea]. Available: http://web.cse.ohio-state.edu/~whmin/courses/cse5542-2013-spring/15-texture.pdf. [Último acceso: 03 12 2015].

[12] U. F. a. D. Administration, «U.S. Food and Drug Administration,» 04 Marzo 2014. [En línea]. Available: http://www.fda.gov/Radiation-EmittingProducts/RadiationEmittingProductsandProcedures/MedicalImaging/MedicalX-Rays/ucm115318.htm. [Último acceso: 19 Setiembre 2015].

[63] Unity, «Unity,» [En línea]. Available: https://unity3d.com/es. [Último acceso: 24 Octubre 2015].

[49] Unity3D, «Unity3D,» [En línea]. Available: http://docs.unity3d.com/es/current/Manual/OcclusionCulling.html. [Último acceso: 03 12 2015].

[4] W. G. S. Library), «PCL,» PCL, [En línea]. Available: http://www.pointclouds.org/about/. [Último acceso: 8 Setiembre 2015].

[29] W. O. Cochran, «Widged-Edge Data Structure,» WSU Vancouver, 18 Octubre 2011. [En línea]. Available: http://ezekiel.vancouver.wsu.edu/~cs442/archive/archive/lectures/winged-edge/winged-edge.pdf. [Último acceso: 17 Octubre 2011].

[33] W. Technologies, «Object Files (.obj),» [En línea]. Available: http://www.cs.utah.edu/~boulos/cs3505/obj_spec.pdf. [Último acceso: 17 Octubre 2015].

[7] Z. Y. F. Sears, «Física Universitaria,» de *Física Universitaria*, Ciudad de Mexico, Pearson, 2004, p. 1313.