UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

EVALUACIÓN AUTOMÁTICA DE CALIDAD DE IMPRESIÓN 3D

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

PAULO JOSE SANDOVAL SUAZO

PROFESOR GUÍA:
BENJAMIN EUGENIO BUSTOS CARDENAS

MIEMBROS DE LA COMISIÓN:
DANIEL PEROVICH GEROSA
MAURICIO EDUARDO PALMA LIZANA

SANTIAGO DE CHILE
2018

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR: PAULO JOSE SANDOVAL SUAZO
FECHA: ENERO 2018
PROF. GUÍA: BENJAMIN EUGENIO BUSTOS CARDENAS

EVALUACIÓN AUTOMÁTICA DE CALIDAD DE IMPRESIÓN 3D

Bajo el contexto del concurso Beauchef Proyecta dentro de la Facultad de Cientas Físicas y Matemáticas de la Universidad de Chile se diseña una Impresora 3D que imprime concreto, este proyecto debe tener una validación respecto a la salida de la impresora, tomando en cuenta que la construcción de la impresora se realizará iterativamente tener un feedback preciso y eficaz ayuda a mejorar la manufactura de este dispositivo.

Para comenzar se realizó una evaluación respecto al estado del arte de las técnicas para describir y comparar volúmenes en modelos 3D y se llegó a la conclusión que la voxelización fue la ruta a seguir para la evaluación de la impresora utilizando comparación volumétrica.

Luego de haber fijado la voxelización como la manera de comparar volúmenes se procedió a diseñar un algoritmo que preprocese los modelos para normalizar las estructuras a comparar. Se utiliza la librería PCL para la implementación.

Se realizó experimentación para determinar si las hipótesis y los criterios determinados para el funcionamiento del algoritmo funcionan correctamente bajo un experimento sintético utilizando modelos de un *benchmark* y se determinó que no cumple con los estándares para la verificación de calidad de una impresora. Luego de esto se realizó un segundo experimento con hipótesis refinadas para considerar la alineación de los modelos utilizando los mismos modelos que el experimento anterior.

Se llegó a resultados exitosos bajo el segundo experimento, lo que logró determinar una métrica bajo el algoritmo para la evaluación de modelos 3D y sus respectivas impressiones utilizando alguna impresora y un dispositivo de adquisición de datos 3D. Esta métrica se determinó respecto a las medidas de los objetos a comparar.

La impresora 3D de concreto al momento de la redacción de este documento está en construcción.

A los que me siguen y a los que no. "Veni, vidi, vici"

Agradecimientos

Me gustaría agradecer primero que todo a mis amigos José Manuel y Jonathan, los cuales hace más de 6 años brindan apoyo incondicional para llevar adelante este largo proceso. Sin duda sin ellos no podría tener la oportunidad de estar en este momento redactando esto. Agradezco infinitamente la entrega durante los momentos buenos, malos y tristes de la carrera.

Además un agradecimiento a mucha gente de la Universidad que fuera de su formación académica lograron enseñarme cosas más allá de los libros y la teoría. Mención especial a "La Pepita" por tantos momentos en poco tiempo.

Finalmente agradezco a mi familia, amigos y a mi polola por alentarme durante este extenso camino para poder cerrar esta etapa y comenzar a brindar frutos con lo aprendido.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Índice de Código

Capítulo 1

Introducción

1.1. Motivación

A principios del presente año, se abrió dentro de la Facultad un concurso de proyectos para Tesis Multidisciplinarias llamado Beauchef Proyecta. Bajo dicho concurso, un grupo de estudiantes postuló a la realización de una Impresora 3D que imprime Concreto (Hormigón). Dicha impresora estará enfocada en la creación de elementos estructurales de construcción.

Bajo el contexto de la aplicación de ingeniería en el proceso de impresión de objetos 3D, se pueden encontrar varios problemas dentro de la implementación de nuevas tecnologías de impresión. Por ejemplo, algún cambio en la materia prima utilizada por la impresora podría hacer que los resultados finales no sean lo suficientemente fieles al objeto. Un problema dentro de la construcción de una impresora es verificar qué tan precisa es al momento de realizar la impresión, esta verificación consiste en analizar el objeto una vez impreso y medir qué tan similar es al modelo 3D original.

Para el caso de la impresora 3D de concreto, es importante tener mucha exactitud puesto que el tipo de modelo que va a ser utilizado en la impresora es del tipo estructural para la construcción de obras más grandes como casas, puentes, etc. Si bien la exactitud del modelo es importante, las propiedades del material como firmeza, resistencia, conductividad, etc también lo son, puesto que conllevan a errores que pueden costar vidas humanas. Tener la exactitud volumétrica sólo asegura el ensamble perfecto del sistema de piezas en conjunto.

En esta memoria, se propone la automatización del proceso de verificación. Esto permitirá que los objetos producidos por la Impresora 3D de Concreto sean validados con menos intervención humana y que el proceso sea eficiente. Finalmente, el proyecto de la Impresora 3D de Concreto fue aceptado por Beauchef Proyecta para el año 2017.

1.2. Objetivos

1.2.1. Objetivo General

El objetivo general de esta memoria es desarrollar un algoritmo que compara objetos reales con modelos 3D y una métrica relacionada a la calidad de la similitud entre el objeto real y el modelo 3D para evaluar la efectividad de la impresora. Este algoritmo estará enfocado en particular a la solución de la Impresora 3D de Concreto, por lo que se buscará implementar un algoritmo eficiente dado el tipo de objetos 3D a imprimir.

1.2.2. Objetivos Específicos

1. Encontrar una descripción adecuada para el objeto real a comparar
   Este objetivo consiste en desarrollar un algoritmo que permita describir el objeto real impreso para poder compararlo con el modelo 3D correspondiente. De manera preliminar se utilizará la Voxelización [23], el cual es una discretización de un plano 3D en una grilla. El voxel se puede considerar como un homólogo a los pixeles pero en un ambiente tridimensional.
2. Definir una medida de evaluación para comparar el objeto real con el modelo 3D
   En primera instancia, una técnica de evaluación será superponer volúmenes de Voxels y medir la diferencia de volúmen. Esta medida deberá ser un porcentaje o un número entre 0 y 1 para poder discriminar numéricamente si la impresión es buena o mala.
3. Comprobar la eficacia de la impresora
   Dado varios objetos con distintas propiedades, determinar si la Impresora 3D realmente imprime adecuadamente los modelos 3D comparando la ejecución del algoritmo con el modelo 3D asociado al objeto.

1.3. Metodología

La metodología utilizada para realizar este trabajo de memoria consta de los siguiente pasos:

1. Describir el estado del arte en métodos para representar y trabajar con objetos 3D
   Esto será un estudio [17] y posterior análisis dado ciertas suposiciones de los objetos que la impresora 3D de Concreto utilizará en su funcionamiento.
   Uno de los posibles métodos a estudiar es la Voxelización [23] la cual, como se mencionó anteriormente, consiste en tomar objetos 3D y pasarlos a una grilla 3D, haciendo que los objetos se representen por medio de "píxeles" 3D. De esta manera los objetos 3D pasan a estar discretizados y pueden ser procesados por distintos algoritmos. Los resultados de este estudio darán lugar a dos algoritmos que estén acorde al contexto de la Impresora 3D a validar. En la Figura 1.1 podemos ver la superficie de un conejo armado con voxels.

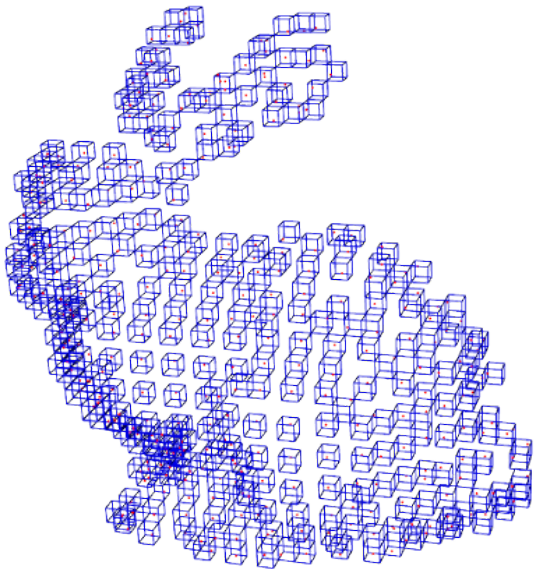

Figura 1.1: Superficie de Conejo armado con Voxels

2. Implementar y probar la Evaluación Automática

   La implementación del algoritmo anterior se llevará a cabo utilizando librerías que simplifiquen la implementación y ejecución del algoritmo. Un ejemplo de estas librerías es la Point Cloud Library [29] la cual se encarga de procesos relacionados a geometrías 3D. La idea es no perder tiempo ni recursos reinventando la rueda, sino que tratar de utilizar bibliotecas existentes para desarrollar la metodología de evaluación automática propuesta.

3. Diseñar y correr experimento para evaluar Impresora 3D

   Dado el algoritmo de evaluación, se espera que éste devuelva un valor que mida la distancia (cercanía) del objeto impreso con el modelo 3D original. El experimento consistirá en la impresión y digitalización de un modelo 3D para luego hacer uso del algoritmo de evaluación con el objetivo de procesar el modelo digitalizado y así verificar la similitud de éste respecto al modelo original.

   En un principio, como no se dispondrá de objetos producidos por la Impresora 3D de Concreto, se propone comenzar esta experimentación con perturbaciones de modelos 3D digitalizados. De esta forma, se podrá validar la eficacia del algoritmo de evaluación sin la necesidad de utilizar objetos reales y/o el scanner 3D.

4. Evaluar Resultados

   El algoritmo se probará con modelos 3D obtenidos de un *dataset* estándar, para luego establecer una métrica para validar si un objeto fue impreso correctamente y determinar si la Impresora está apta para la fabricación de objetos 3D a partir de modelos.

Capítulo 2

Marco Teórico

2.1. Impresión 3D

La impresión 3D es una tecnología que se basa en la fabricación de un objeto físico tridimensional por adición de material desde un diseño digitalizado. Las impresoras 3D ofrecen a los desarrolladores de productos una manera rápida de construir montajes, partes y piezas para la construccion de prototipos de distintos materiales a utilizarse en la industria, de esta manera agilizando el proceso de manufactura y puesta en marcha de sus productos.

2.1.1. Funcionamiento

Para la creación de un objeto físico impreso es necesario comenzar por el diseño 3D digital, la cual es la representación de la superficie (en general) de un objeto tridimensional. Estos diseños se modelan en general utilizando herramientas de diseño asistido por computadora (CAD), a partir del cual se puede imprimir. La impresión de un modelo 3D varía según el tipo de impresora, lo que todas estas impresoras tienen en común es el funcionamiento aditivo del material, es decir, el material empleado para crear los objetos siempre se añade y nunca se quita (durante el proceso de impresión).

Los modelos 3D no pueden ser impresos sin antes ser tratados bajo algún algoritmo que determine las instrucciones de posición y adición del material, por ejemplo se le ordena a la impresora estar en la posición (3,2,0) y luego depositar un poco de material, para luego moverse a la posición (3,3,0) para depositar otro poco de material. El funcionamiento aditivo por capas se puede ver en la Figura 2.1. Asimismo, las impresoras pueden manejar de manera distinta la altura a la cual se deposita el material, podrían ser de base trasladable o el cabezal de impresión ser de altura variable [21]. De esta forma tenemos varios grados de libertad para la impresora, es decir la posición <X,Y,Z> y la adición del material.

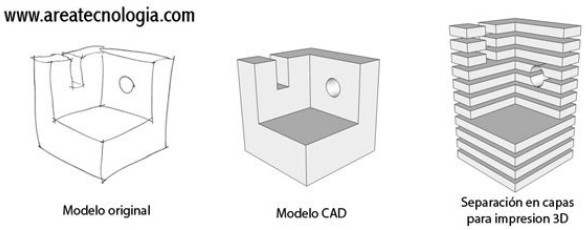

Figura 2.1: Funcionamiento de Impresora 3D por capas
Fuente: Sitio web "AreaTecnologia"
`http://www.areatecnologia.com/informatica/impresoras-3d.html`

El proceso de impresión varía según el tipo de impresora. Las más comunes son las impresoras de plástico PLA en el cual el plástico es un filamento muy delgado con la ventaja que se puede derretir a temperaturas no tan altas y no emite gases al hacerlo. Las impresoras de PLA deben manejar un carril continuo de filamento PLA para mantener siempre caliente una porción del material e ir calentando más al ir depositando material al objeto final. Es recomendable utilizar algún tipo de cubierta para el depósito del material, en general se utiliza una cinta azul (como se puede ver en la Figura 2.2) para que la plataforma que reciba el material no reciba directamente el plástico caliente y no se deforme el objeto final.

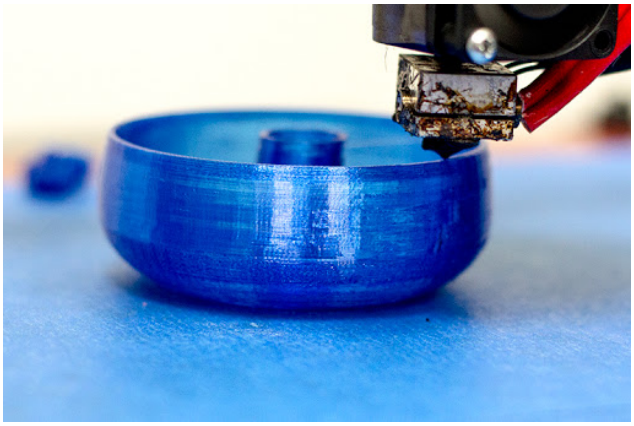

Figura 2.2: Impresión utilizando PLA
Fuente: Sitio Web "MatterHackers"
`https://www.matterhackers.com/articles/how-to-succeed-when-printing-in-pla`

También existen impresoras de ABS que son más utilizadas en la industria de juguetes, automotriz, etc. A diferencia del PLA, el ABS tiene un punto de fusión más elevado que el PLA y emite gases tóxicos al utilizar el material, pero a la vez tiene la capacidad de ser reutilizable una vez que se vuelve a calentar el material.

Una vez finalizado el depósito del material dependiendo de la calidad y resolución de la impresora puede que sean necesarios ciertos postprocesos para completar la fabricación del producto final, como por ejemplo lacado, lijado o adhesión de otro objeto.

Tecnologías similares se utilizan para las máquinas CNC [31] (computer numerically con-

trolled), con la diferencia fundamental que no funcionan con adición sino que con sustracción de material (se comienza con un bloque de material y se extrae con una máquina rotatoria). También existen proyectos que fusionan la tecnología CNC de extrusión con las impresoras 3D como por ejemplo la impresora de la Figura 2.3, la cual realiza extrusiones con taladro y también puede realizar depósito de material para generar objetos utilizando PLA, ABS entre otros.

Figura 2.3: Máquina CNC estándar, utilizando un Dremel
Fuente: Sitio Web "KickStarter"
`https://www.kickstarter.com/projects/stepcraft2/stepcraft-2-universal-desktop-cnc-3d-printer-for-e`

2.1.2. Beneficios

De acuerdo a Leydy Gómez Reyes et al. [27], se pueden observar las siguientes ventajas de la impresión 3D respecto a la creación común de objetos:

"

- ***Reciclaje**: Los científicos han logrado desarrollar una máquina con la capacidad de reciclar los residuos domésticos de plástico para convertirlos en material de trabajo para las impresoras 3D, esa máquina es llamada Filabot y permite que mediante el reciclaje de plástico se genere un ahorro en la compra de ABS.*
- ***Versatilidad**: Una sola impresora tiene la posibilidad de realizar infinidad de productos distintos a diferencia de la forma en la que se fabricaba anteriormente, pues para producir un objeto tenía que haber una máquina específica, y si el producto cambiaba se tenía que readaptar la máquina.*
- ***Reducción de Costos**: Los costos de producción disminuyen porque son más rápidos los tiempos de fabricación y se pueden realizar en casa.*
- ***Personalización**: Una de las ventajas más atractivas es la posibilidad de que cada usuario puede diseñar sus propios artículos de acuerdo a sus necesidades y gustos.*

- ***Creación de Prototipos****: Los diseñadores industriales podrán realizar prototipos de los objetos industriales en menor tiempo y costo.*
- ***Construcción****: Esta herramienta se aplica también en la arquitectura pues facilita la construcción de edificios y demás estructuras ya que se pueden hacer impresiones a gran escala y por medio del modelado capa por capa. Esta aplicación actualmente se está experimentando para crear estructuras en planetas distintos a la tierra.*

" [27]

2.2. Objetos/Modelos 3D

Los objetos 3D por definición son más complejos de representar que los objetos 2D.

Tomemos por ejemplo una grilla de pixels, sabemos por definición que esta grilla tiene una única representación en un display 2D (asumiendo aspect ratio constante) puesto que la vista 2D de un objeto 3D sobre un plano 2D es una representación perfecta. Las únicas posibles ambiguedades podrían darse en la representación de los colores de aquél arreglo de pixels pero nunca en su forma. Podemos ver una representación de pixels en la Figura 2.4, donde hay muy poca ambiguedad en su representación final. De esta manera, los modelos 3D buscan representar objetos de la vida real utilizando representaciones que tengan la menor ambiguedad posible. El problema es que no se puede representar la totalidad de un objeto 3D sobre un plano 2D, puesto que no se pueden mantener los datos sobre las distintas perspectivas posibles y las posibles superficies que conlleva el objeto 3D no son suficientes para determinar la totalidad del objeto 3D, dado que los objetos 3D en particular poseen ...

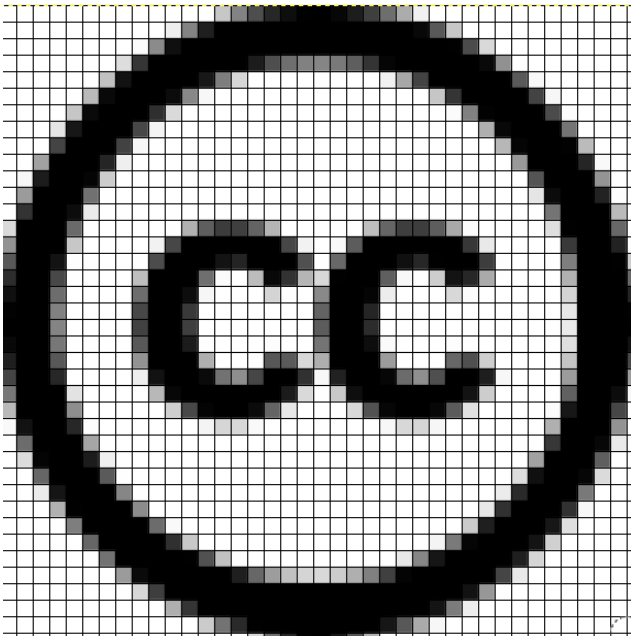

Figura 2.4: Grilla de Pixels 2D
Logo de Creative Commons bajo grilla de pixeles

Las dificultades de representar un objeto 3D son:

- Distintas fuentes de datos 3D de un mismo objeto (escaneo, topografía) no producen representaciones exactas.
- Las distintas aplicaciones (de diseño, salud, etc) no consideran las mismas representaciones del objeto 3D.
- Cambiar de una representación a otra es un proceso complejo.

El modelado 3D es el proceso de desarrollo de una representación matemática de cualquier objeto tridimensional (ya sea inanimado o vivo) a través de un software especializado. Al producto se le llama modelo 3D. Se puede visualizar como una imagen bidimensional mediante un proceso llamado renderizado 3D o utilizar en una simulación por computadora de fenómenos físicos. El modelo también se puede crear físicamente usando dispositivos de impresión 3D.

Los modelos 3D pueden representarse en forma de la carcasa/superficie o el sólido que contienen, por lo general son representados por la superficie o contorno puesto que al ser visualizados por alguna herramienta computacional definen de manera más específica la forma que el objeto representa. La representación de sólido es menos utilizada puesto que no es tan útil para ser visualizada, pero sí para hacer pruebas y simulaciones físicas sobre el modelo. [24]

2.3. Representación de Objetos 3D

Los objetos 3D en general se pueden representar de distintas maneras, las que serán tratadas dentro de este estudio son [19]:

- Nube de Puntos.
- Malla Poligonal.
- Voxels.

2.3.1. Representación: Nube de Puntos

Una nube de puntos es un conjunto de puntos bajo cierto sistema de coordenadas, en el caso de los objetos 3D se utilizan las coordenadas X, Y y Z. Las nubes de puntos son generadas mayoritariamente por *Scanners* 3D, utilizando herramientas de medición logran medir una serie de puntos en la superficie del objeto y luego guardando una nube como salida. Podemos apreciar una Nube de Puntos en la Figura 2.5, donde los puntos forman la superficie de un Torso humano.

Si bien las nubes de puntos pueden ser útiles para inspeccionar visualmente los objetos, no son tan útiles para las mediciones de fenómenos físicos y relacionados, por lo cual generalmente se convierten a mallas de polígonos u otras conversiones según se requiera.

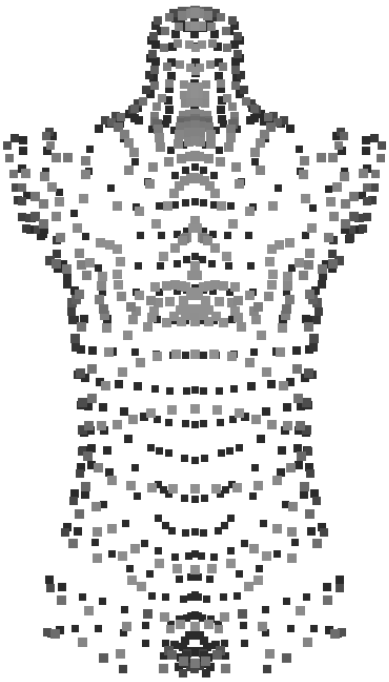

Figura 2.5: Nube de puntos de un Torso

2.3.2. Representación: Malla Poligonal

Triangulación

De acuerdo a Francisca Gallardo [20], “

Una triangulación en dos dimensiones sobre un conjunto de puntos, se define como una subdivisión del plano en caras triangulares cuyos vértices corresponden a los puntos del conjunto inicial. En tres dimensiones, una triangulación es un conjunto de triángulos que aproximan una superficie en el espacio a partir de un conjunto de puntos sobre esa superficie. Ambos tipos de triangulación se construyen formando caras planas triangulares, uniendo pares de puntos mediante aristas que nunca se cruzan entre sí.

Las triangulaciones también se conocen por el nombre de mallas de triángulos. En la Figura 2.6 se muestran dos ejemplos de mallas, una sobre el plano y otra sobre una superficie en tres dimensiones.

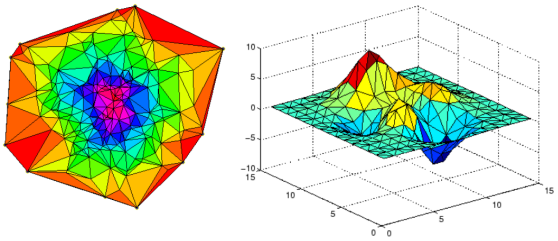

Figura 2.6: Aproximación de superficies mediante mallas de triángulos, en dos y tres dimensiones [20]

Se dice que una triangulación o malla de triángulos es válida si cumple con las siguientes 4 características:

1. *Todos los triángulos construidos poseen área mayor que cero*
2. *La intersección del interior de dos triángulos cualesquiera del conjunto es siempre vacía.*
3. *La intersección de dos triángulos cualesquiera del conjunto corresponde a una arista común o a un vértice común.*
4. *El conjunto define una superficie conexa, abierta o cerrada.*

" [20]

Triangulación de Delaunay

De acuerdo a Francisca Gallardo [20], "

La triangulación de Delaunay ha sido ampliamente estudiada en el ámbito de la geometría computacional debido a sus aplicaciones en diferentes áreas de la ciencia y la ingeniería. Su definición formal se presenta a continuación:

*Definición: **Triangulación de Delaunay**. Una triangulación del conjunto P de puntos sobre el plano es de Delaunay, si y solo si el circuncírculo de cualquier triángulo de la malla no contiene un punto de P en su interior (ver Figura 2.7)*

La triangulación de Delaunay satisface las siguientes propiedades:

1. *La triangulación maximiza el menor ángulo de la malla, es decir, el menor de los ángulos internos de los triángulos que la conforman.*
2. *La frontera de la triangulación es la envolvente convexa de los puntos, en otras palabras, las aristas del borde de la triangulación forman un polígono convexo que contiene todos los demás puntos.*
3. *La triangulación es única cuando ningún borde de circunferencia circunscrita contiene más de tres vértices de la malla.*

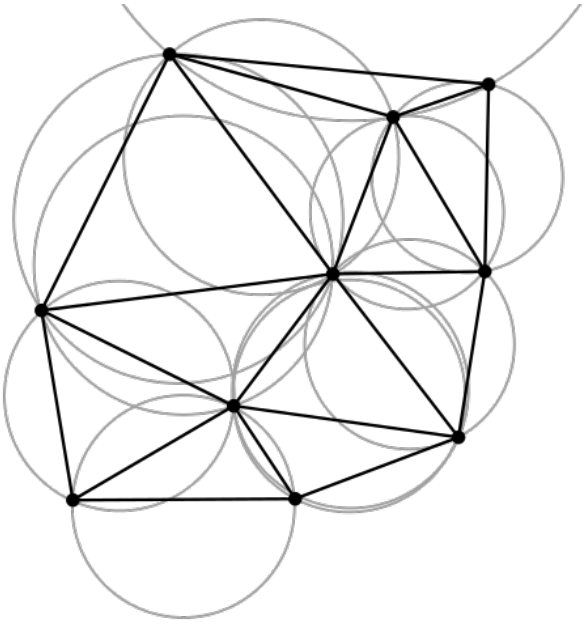

Figura 2.7: Triangularización de Delaunay sobre un conjunto de 10 puntos [20]

La Figura 2.7 Existen varios métodos para convertir de una Nube de Puntos a una Malla Poligonal. La triangularización de Delaunay es la más utilizada.

" [20]

2.3.3. Malla Poligonal

Un *scanner* 3D produce una nube de puntos 3D con ninguna relación entre ellos, por lo que es ineficiente para ver propiedades físicas o para ser mostrado en una pantalla.

Para calcular volúmenes y otras mediciones físicas se procede a convertir esta nube de puntos a una malla poligonal, la cual es una representación de la superficie a base de planos (generalmente triangulares) que ayuda a obtener información volumétrica del objeto 3D.

En particular esta malla 3D contiene puntos 3D (vértices) que están relacionados por aristas para formar facetas poligonales (ver Figura 2.8).

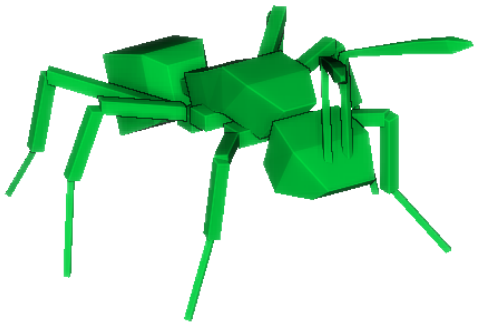

Figura 2.8: Malla de Hormiga de Princeton Benchmark [30]

El problema de las mallas poligonales es que tienen un buen comportamiento de forma local pero al aplicar deformaciones al cuerpo completo se deben recalcular todas las facetas. Asimismo tener una malla discreta con cierto número de vértices y facetas es discreto pero aumentar la precisión implica un gran aumento en la cantidad de datos.

2.3.4. Representación: Voxels

En este modelo volumétrico se utilizan píxeles volumétricos (voxels) en una grilla tridimensional. Esta grilla puede contener voxels con información booleana o algún número relativo a la densidad del volumen que representa.

El objeto se determina respecto a una matriz tridimensional de tamaño fijo, con cada elemento representando una región cúbica en el espacio 3D. En la Figura 2.9 tenemos una malla de polígonos siendo convertida a voxels, representando unidades volumétricas en vez de sólo la superficie del conejo.

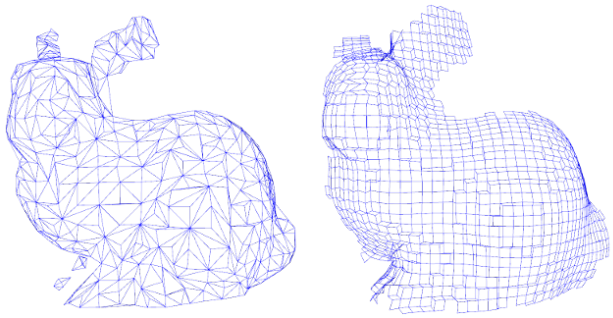

Figura 2.9: Conversión de Malla a Voxels

2.4. Adquisición de Datos

El objetivo de la adquisición de datos es tener una representación lo más legítima del objeto de la vida real. En general hay dos tipos de adquisición, las cuales son adquisición por superficie (*Scanner* 3D) o por volumen (Tomografía). En esta investigación utilizaremos sólo la tecnología de Scanner 3D por su fácil acceso dentro de la Facultad.

2.4.1. Scanner 3D

El *Scanner* 3D funciona similar a una cámara fotográfica pero midiendo distancia relativa al objeto en vez de información respecto a la intensidad de colores del objeto. Esta información de distancia es la profundidad de un punto respecto a la cámara principal, por lo que con sólo una toma no se puede reconstituir el objeto de la vida real. Para esto hay que realizar distintas tomas del mismo objeto en distintos ángulos para luego realizar un alineamiento entre todos los modelos para formar un único modelo 3D.

Una vez tomada la información de las distintas tomas, se produce una única Superficie o Nube de Puntos representando el objeto real. Estas tomas deben ser mezcladas y alineadas dentro del mismo sistema de coordenadas.

2.4.2. Alineamiento

El alineamiento es un problema bastante recurrente dentro del uso de modelos 3D digitalizados. Un caso de uso muy común es tratar de unir dos piezas o dos modelos 3D bajo el contexto de una arista o una cara, para esto es necesario que la orientación de ambos modelos sea compatible. En la Figura 2.10 podemos ver el posible alineamiento de dos figuras para luego ser unidas bajo esos puntos de contacto.

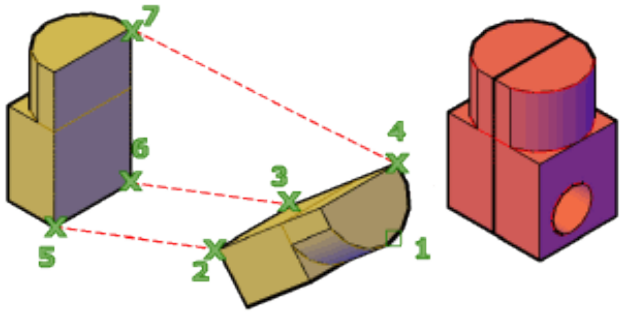

Figura 2.10: Alineamiento de puntos en programa CAD
Imagen extraída de manual de AutoCAD [2]

Para ayudar al alineamiento existen algoritmos que intentan asistir a los usuarios ofre-

ciéndoles la mejor orientación posible relativa a los puntos a asociar. Un algoritmo bastante utilizado en este ámbito es el ICP (Iterative Closest Point) [16].

2.4.3. Algoritmo ICP

El algoritmo Iterative Closest Point (ICP de ahora en adelante) es utilizado fundamentalmente para asistir al alineamiento de dos o más objetos. Los objetos para ser correctamente digitalizados deben tener coordenadas de referencia previamente establecidas de manera tal de no perder ningún detalle espacial en la digitalización.

Figura 2.11: Alineamiento de puntos para generar un modelo único
Fuente: Domingo Gallardo López (2012), Sesión 6: Alineamiento 3D a partir de nubes de puntos, Documento publicado en el Departamento de Ciencia de la Computación e Inteligencia Artificial, archivado en
http://www.dccia.ua.es/dccia/inf/asignaturas/Vision/vision-tema6.pdf

Dados dos modelos observados desde dos puntos de vista diferentes, en nuestro caso, dos nubes de puntos con su respectivo sistema de referencia local, el problema consiste en encontrar la transformación rígida (rotación y traslación) entre los dos sistemas de referencia que alinea puntos correspondientes de las dos nubes (scan matching). Podemos ver en la Figura 2.11 tres tomas del mismo objeto en distintas orientaciones, se seleccionan puntos de referencia para lograr unificar un modelo completo del objeto (en este caso la casa)

El algoritmo ICP consiste en minimizar la diferencia entre dos nubes de puntos, utilizando combinaciones de transformaciones (rotación, traslación) hasta encontrar la que tenga el error mínimo. Este algoritmo fue publicado por Besl y McKay [16] en el año 1992 y es utilizado para reconstruir superficies 2D o 3D a partir de varios scans de distintas tomas.

Los pasos a seguir del algoritmo [28] son:

- Por cada punto en la nube de puntos original, encontrar el punto más cercano en la nube de puntos de la referencia.
- Estimar la combinación de rotación y traslación utilizando "minimización de distancia métrica RMS punto a punto" que alinee de mejor forma el punto original a su match

encontrado en el paso anterior luego de hacer tratamientos de *weighting* y eliminar *outliers*.

- Transformar los puntos originales utilizando la transformación anterior.

Dentro de la PCL (ver Sección 2.5.1) existe una implementación de ICP [9]. Esta implementación recibe dos nubes de puntos para comenzar a iterar y al utilizar el método *align* de esta biblioteca se obtiene el resultado final de las iteraciones.

Dadas dos nubes de puntos, se procede a recrear el objeto 3D uniendo las distintas vistas utilizando ICP. Para esto, se trabaja de a pares de nubes para obtener una única nube de puntos que contenga todas las vistas escaneadas. El algoritmo ICP entrega como resultado un puntaje de convergencia por lo cual se revisará que la nube de puntos final sea coherente con el objeto escaneado

2.5. Bibliotecas y Lenguajes de Programación utilizados

En esta seccion se detallan las herramientas computacionales utilizadas durante este trabajo de memoria.

2.5.1. PCL y C++

Point Cloud Library [29] (PCL de ahora en adelante) es una librería de código abierto basada en algoritmos para el procesamiento de imágenes 2D y 3D. Es un proyecto a gran escala publicado bajo la licencia BSD, lo cual lo hace libre de uso para uso comercial y de investigación.

Los algoritmos que contiene PCL están relacionados al estado del arte respecto a las técnicas desarrolladas de búsqueda de *features*, reconstrucción de superficies, extracción de *keypoints* entre otros problemas del mundo 2D y 3D. Utilizando estos algoritmos se pueden crear soluciones de visión por computadora, filtrar *outliers*, generar descriptores, crear superficies desde nubes de puntos y más.

Es importante notar que la librería PCL es Multiplataforma, es decir compatible con Windows, MacOSX, Linux, Android y iOS. Además la librería está siendo mantenida y financiadas por grandes empresas del área de desarrollo e ingeniería como Google, NVidia, Toyota, Intel, entre otras.

Para la PCL todos los modelos 3D se pueden representar como nubes de puntos, siendo esta nube contenedora la estructura básica para el procesamiento de modelos. Generalmente estos puntos se representan utilizando coordenadas geométricas X, Y, Z además de información de color siempre y cuando esté disponible. Estas nubes de puntos pueden ser adquiridas desde diversos dispositivos como cámaras, scanners 3D o creados sintéticamente utilizando una computadora. PCL también es capaz de comunicarse directamente con dispositivos como Microsoft Kinect para la adquisición de puntos.

2.5.2. Hardware y Sistema Operativo

El hardware utilizado para la instalación del ambiente de compilación es una máquina virtual de 4 cores utilizando VirtualBox. Esta máquina corre Ubuntu 14.04 64bits, con 8GB de RAM asignados.

En la computadora utilizada para el desarrollo del software de esta memoria sólo funciona Windows 10 y para crear programas utilizando la librería PCL es necesario tenerlo en un ambiente de compilación de C++, como en Windows esto no es nativo se optó por la opción de utilizar Ubuntu sobre una máquina virtual para evitar conflictos de hardware y poder comenzar a desarrollar en el menor tiempo posible.

Hubo muchas dificultades para hacer funcionar PCL bajo Windows por lo que se eligió Ubuntu debido a la facilidad de instalar aplicaciones con el Gestor de Paquetes interno. Así la versión 1.7.2 de PCL fue instalada sin mayores dificultades bajo Ubuntu, mientras que en Windows no se lograba realizar una compilación de ejemplo.

La compilación bajo Linux se realizó con el programa CMake, el cual es un set de herramientas para la construcción de aplicaciones multiplataforma con generación de makefiles. Fue creada para la utilización en proyectos de código abierto

En particular se utilizó la guía de instalación de ROS (Robot operating System) [26] en Ubuntu 14.04 [12] puesto que ROS utiliza librerías de PCL para su funcionamiento estándar. Además la instalación de ROS tiene incluídas bastantes dependencias de librerías de C++ adicionales a PCL.

2.5.3. Software Adicional

Para el desarrollo y manejo de los modelos 3D, se utilizaron dos aplicaciones requeridas para la conversión de archivos y la visualización de los modelos respectivamente. El primero es CloudCompare [22] y el segundo es MeshLab [18]

MeshLab

"the open source system for processing and editing 3D triangular meshes" [5]

MeshLab provee un set de herramientas para edición, limpieza, restauración, inspección, rendering, textura y conversión de mallas. Ofrece características para procesar datos "crudos" producidos por dispositivos de digitalización 3D y también características para preparar modelos para su impresión 3D.

CloudCompare

"3D point cloud and mesh processing software. Open Source Project"[3]

CloudCompare es un software procesador de Nubes de Puntos 3D y mallas triangulares. Ha sido originalmente diseñado para realizar comparasiones entre dos nubes de puntos densas o entre una nube de puntos y una malla triangular. Se basa en una estructura específica de *octree* dedicada a esta tarea. Más tarde se ha extendido a ser un procesador de nubes de puntos genérico, incluyendo varios algoritmos en sus herramientas.

2.6. Impresión 3D y Evaluación de la calidad

Balletti et al. [15] a mediados del año 2017 mencionan que el esparcimiento de las impresoras 3D (sólidas) ha resaltado el problema de construir modelos digitales que necesitan ser impresos y, por otra parte, de la conformidad de la copia con el modelo original.

Para identificar las diferentes aplicaciones como reproducciones, copias, etc. es necesario determinar y verificar la exactitud métrica del modelo físico y digital. Sólo de esta manera la copia va a reemplazar efecticamente el modelo original en muchos aspectos, especialmente en el evento que éste debe ser preservado y su uso es limitado a exposición. Los modelos físicos de hecho son cada vez más y más frecuentes en ocasiones en cual el contacto físico es necesario; por ejemplo, exhibiciones de museos dedicados a público infantil o para gente con discapacidades visuales.

2.6.1. Problema

La creación de objetos 3D a partir de modelos digitalizados se ha vuelto muy popular en la última década, es por esto que muchos equipos de ingenieros y arquitectos van iterando respecto a los diseños de las impresoras y/o materiales de construcción.

Es importante para el proceso iterativo tener un *feedback* constante, de calidad y que no sea costoso (ni en tiempo ni en dinero). En particular, para el desarrollo de la Impresora 3D de Concreto es necesario contar con una métrica de comparación de eficacia o exactitud y no tanto su eficiencia (entendiendo eficiencia como el tiempo/recursos que toma hacer una tarea) puesto que para las etapas iniciales del funcionamiento de la impresora los recursos a utilizar no estarán optimizados.

El problema específico a resolver es la comparación entre el modelo 3D digital y el objeto impreso real, los cuales no tienen las mismas unidades de medida y en el caso de una impresora en desarrollo, probablemente no tenga simetría ni sean 100 % equivalentes.

Para esto se espera diseñar una solución computacional que resuelva tanto el problema de comparar un modelo 3D digitalizado con uno escaneado para ver temas de eficacia y también para hacer controles de calidad en el futuro.

2.6.2. Solución Planteada

Para el proceso de validación de la Impresora, hasta donde sabemos, no hay ningún método documentado a la fecha, por lo cual en esta memoria se diseñó un método coherente para el uso en el área de Ingeniería.

Para poder evaluar la impresión 3D se requiere obligatoriamente digitalizar el objeto real nuevamente hacia un modelo 3D digital. Este proceso requiere intervención humana para lograr la correcta digitalización del objeto, en la implementación inicial de la solución se considerará un scanner de barrido para la digitalización del objeto real hacia el modelo 3D.

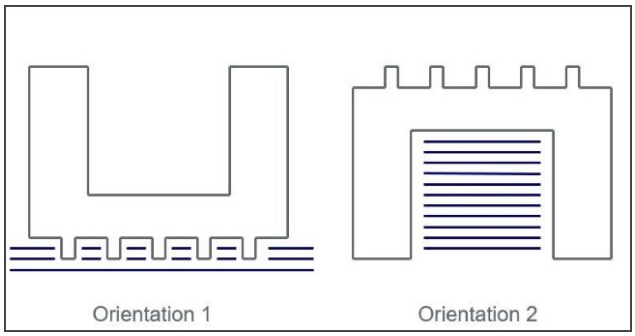

Figura 2.12: Problemas de orientación
Al digitalizar un objeto real, hay que considerar posibles pérdidas de información respecto a la orientación con la cual estamos adquiriendo los datos, puesto que pueden perderse detalles importantes en este proceso

Existen ciertos aspectos muy importantes a considerar durante la digitalización del objeto real. Entre ellos el alineamiento y orientación. Si consideramos un modelo 3D bajo cierta orientación, compararlo con otro modelo 3D con distinta orientación es un problema complejo, dada la cantidad de orientaciones posibles en un ambiente tridimensional (ver Figura 2.12). Para reducir este tipo de problemas existen algoritmos como el ICP (Iterative Closest Point)[16] que utilizado con algunos métodos que se basan en la topografía del modelo original logran calcular un calce estimado mucho más rápido.

Los problemas de orientación y alineamiento en principio no fueron considerados en la solución planteada, dado que los casos de uso para la Impresora 3D de Concreto contemplan generalmente objetos con simetría axial dentro del eje Z y al digitalizar los objetos debería utilizarse esta misma orientación dentro del *Scanner*/Dispositivo de adquisición de Datos. Dicho esto, los problemas de escala y posicionamiento del objeto si están contemplados dentro del análisis en este trabajo, puesto que sólo conllevan traslaciones y escalamiento de las coordenadas de los puntos pertenecientes a la nube.

Capítulo 3

Algoritmo Propuesto

La evaluación de los objetos 3D se realizará en 4 pasos:

- Escaneo 3D.
- Preprocesamiento.
- Voxelización.
- Comparación.

3.1. Escaneo 3D

El Escaneo 3D consiste en digitalizar el objeto real para luego ser procesado y comparado con el objeto 3D original. Este Escaneo puede ser realizado con cualquier Scanner 3D siempre y cuando se tengan en cuenta los archivos de salida de los Scanners.

En general la mayoría de los Scanners funciona con técnicas de barrido de superficie y sus archivos de salida son del formato STL o OFF, estos formatos son muy populares dentro del manejo de objetos 3D pero nuestra librería funciona de manera óptima con archivos del tipo PCD (PointCloud Data) [6], por lo que es necesario convertir los modelos digitalizados a PCD antes de comenzar el procesamiento por el algoritmo utilizando ciertos programas externos.

La conversión se realiza en dos pasos. La primera conversión es de STL/OBJ/PLY a VTK y luego de VTK a PCD. La conversión a VTK la realiza CloudCompare en modo consola, puesto que lo hace automatizable, y la segunda conversión un programa incluído en la librería PCL llamado pcl_vtk2pcd, el cual recibe un VTK como input y genera un PCD (PointCloudData) [6]

Una vez terminado el Escaneo 3D y verificada la conversión de los archivos de salida del Scanner se pasa a la etapa de preprocesamiento y Voxelización.

3.2. Preprocesamiento

La etapa de preprocesamiento consiste en normalizar la escala y la posición de los archivos PCD. En particular es necesario que ambos objetos tengan características similares para que las posteriores voxelización y comparación tengan sentido. La escala se normaliza respecto a la distancia máxima en el eje Z, dado que los objetos a trabajar son del tipo estructurales y por ende tienden a tener un eje de simetría en ese eje. La normalización de la posición consiste en trasladar el centroide hacia el punto de origen.

No obstante, se asume que el objeto digitalizado está orientado (en rotación, respecto a ejes) de la misma manera que el objeto original, de manera tal que el preprocesamiento sólo se encarga de traslaciones y escalas. Luego, la librería PCL logra hacer este procesamiento utilizando Matrices de Transformación las cuales se pueden programar bajo una estructura de matriz o especificando las acciones de traslación y escala directamente a una matriz.

Vale notar que para la utilización de las funciones **transform** [11] y **scale** [11], es necesaria una conversión del objeto contenedor de la nube de puntos. En la implementación más moderna del objeto PointCloud (pcl::PointCloud2) no se almacenan de la misma manera los puntos, por lo que recorrer la nube de puntos requiere una conversión de pcl::PointCloud2 a pcl::PointCloud(<pcl::PointXYZ>) dado que en este último tipo de objeto los puntos se pueden acceder e iterar directamente.

Finalmente, se obtiene una nube de puntos lista para ser procesada en la etapa de Voxelización. El código 3.1 muestra la transformación aplicada para el primer experimento, manteniendo el centroide del modelo en el origen del sistema y escalando respecto al eje Z.

```
1  pcl::PCLPointCloud2::Ptr transform(pcl::PCLPointCloud2::Ptr cloud, bool scale)
       {
2      // Calcular Centroide y aplicar Trasaci n
3      pcl::PointCloud<pcl::PointXYZ>::Ptr vertices(new pcl::PointCloud<pcl::
       PointXYZ>);
4      pcl::fromPCLPointCloud2(*cloud, *vertices);
5
6      Eigen::Vector4f centroid = compute_centroid(vertices);
7      float centroidX = centroid[0];
8      float centroidY = centroid[1];
9      float centroidZ = centroid[2];
10
11     // Hasta aqui estan los valores de la matriz de traslaci n
12     // Definimos la Transformacion
13     Eigen::Affine3f transform = Eigen::Affine3f::Identity();
14
15     if (scale) {
16         pcl::PointXYZ p_min;
17         pcl::PointXYZ p_max;
18         pcl::getMinMax3D(*vertices, p_min, p_max);
19         // Suponemos un cubo de tama o 10 para las comparaciones
20
21
22         float sx = 1; // factor X de escala
23         float sy = 1; // factor Y de escala
24         float sz = 1; // factor Z de escala
```

```
25
26          float dist_max = p_max._PointXYZ::data[2] - p_min._PointXYZ::data[2];
27          float scale = 10 / dist_max;
28          sx = scale;
29          sy = scale;
30          sz = scale;
31
32          Eigen::Vector3f escala;
33
34          escala[0] = sx;
35          escala[1] = sy;
36          escala[2] = sz;
37
38          transform.translation() << -centroidX*sx, -centroidY*sy, -centroidZ*sz
    ;
39          transform.scale(escala);
40      } else {
41          transform.translation() << -centroidX, -centroidY, -centroidZ;
42      }
43
44      // Definicion de nube nueva
45      pcl::PointCloud<pcl::PointXYZ>::Ptr transformed_cloud(new pcl::PointCloud<
    pcl::PointXYZ> ());
46      // Aplicar la Transformaci n
47      pcl::transformPointCloud(*vertices, *transformed_cloud, transform);
48
49      pcl::PCLPointCloud2::Ptr transformed(new pcl::PCLPointCloud2());
50      pcl::toPCLPointCloud2(*transformed_cloud, *transformed);
51
52      return transformed;
53  }
```

Código 3.1: Código de Preprocesamiento

3.3. Voxelización

Dadas las distintas formas de representar objetos 3D, la Voxelización fue elegida para representar y comparar dos representaciones de objetos. En particular, la voxelización representa discretamente el volumen de un sólido en entidades separables y enumerables. La comparación de superficies involucra trabajar directamente con las nubes de puntos infiriendo muchos datos en su ejecución, por lo que los Voxels tienen más sentido para el caso de uso de nuestra Impresora 3D de Concreto.

Una vez terminado el preprocesamiento se debe volver a convertir el objeto de la nube de puntos de pcl::PointCloud a pcl::PointCloud2 para su Voxelización, esto es porque la funcionalidad de la librería PCL que procesa la nube de puntos como Voxels utiliza este tipo de objeto.

La libreria PCL contiene un filtro de grilla de Voxels (PCL::VoxelGrid) [7] la cual filtra nubes de puntos entregando puntos individuales que representan voxels posicionados en el centroide de los puntos contenidos dentro de la grilla de voxels.

De esta manera tenemos una nube de puntos reducida en cantidad respecto a la nube de puntos original pero conteniendo información relevante respecto a los voxels que representan la superficie manteniendo la estructura de nube de puntos.

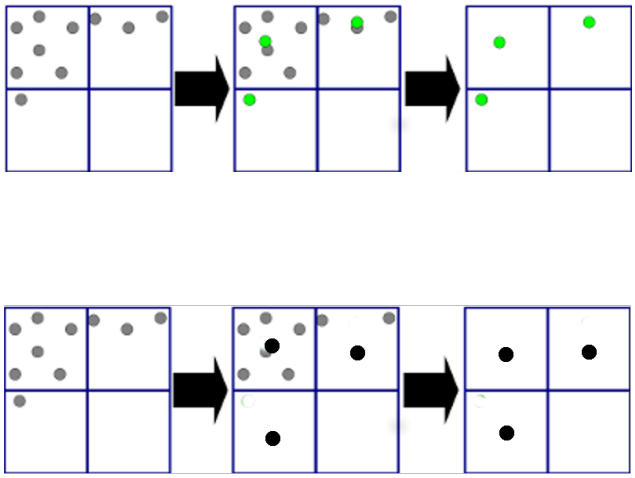

Figura 3.1: Explicación de VoxelGrid
El primer diagrama representa el método VoxelGrid que implementa PCL, marcando el centroide como punto referencial del Voxel.
El segundo diagrama representa una grilla de Voxels con un output estandar (El centroide del cubo es marcado como punto referencial del Voxel)

VoxelGrid [8] crea una grilla virtual de cubos de tamaño definido sobre la nube de puntos. Sobre cada cubo se calcula el centroide de todos los puntos que contiene su interior, que no necesariamente calza con el centroide del cubo. Con esta implementación de VoxelGrid se obtienen superficies más fidedignas a la nube original pero su cálculo es más costoso que la implementación original, la cual es utilizar el centro del cubo como representación del voxel sólo si existen puntos dentro del cubo [7]. Para ver la diferencia entre implementaciones ver Figura 3.1. La implementación utilizada en el algoritmo es la siguiente (ver Código 3.2)

```
1
2  // Retorna nube de puntos filtrada segun Voxels de tama o size
3  pcl::PCLPointCloud2::Ptr voxel_cloud(pcl::PCLPointCloud2::Ptr cloud, float
       size) {
4      pcl::PCLPointCloud2::Ptr cloud_filtered = transform(cloud, true);
5      // Create the filtering object
6      pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
7      sor.setInputCloud(cloud_filtered);
8      sor.setLeafSize(size, size, size);
9      sor.filter(*cloud_filtered);
10     return cloud_filtered;
11 }
```

Código 3.2: Código de Voxelización

Una vez que la nube de puntos termina esta etapa, se puede pasar a la etapa de comparación.

3.4. Algoritmos de Comparación

Una vez definidas las nubes de puntos que representan los centroides de los voxels se procede a iterar respecto a los puntos de la nube comparando cada una con los puntos de la nube vecina, buscando la distancia menor (distancia euclideana) para considerar éxito en la búsqueda de puntos más cercanos siempre y cuando esta distancia sea menor al radio de la esfera cincunscrita (ver Figura 3.2) en el volumen virtual del voxel.

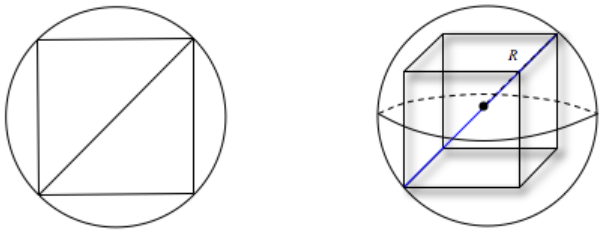

Figura 3.2: Esfera Circunscrita en un Cubo
Fuente: Sitio Web "Aula Mágica" `https://aulamagica.wordpress.com/2008/06/19/volumen-de-una-esfera-circunscrita-en-un-cubo/`

El algoritmo de comparación tiene como resultado la cantidad de aciertos y porcentaje de aciertos respecto a la primera nube. Esta búsqueda se hace en los dos sentidos (nube A ->nube B y nube B ->nube A) y con estos resultados se determina la similitud utilizando el mínimo porcentaje entre ambas comparaciones.

Una vez determinado la similitud, se considera una escala (por definir más adelante) para denotar si los objetos son o no suficientemente similares para considerar eficaz la impresión. En el Código 3.3 podemos ver la implementación de la distancia entre puntos y la comparación de nubes de puntos.

```
1  // calcula distancia entre 2 PointXYZ
2  float dist_pcl_points(pcl::PointXYZ v1, pcl::PointXYZ v2) {
3      float x1, x2, y1, y2, z1, z2;
4
5      x1 = v1._PointXYZ::data[ 0 ];
6      y1 = v1._PointXYZ::data[ 1 ];
7      z1 = v1._PointXYZ::data[ 2 ];
8      x2 = v2._PointXYZ::data[ 0 ];
9      y2 = v2._PointXYZ::data[ 1 ];
10     z2 = v2._PointXYZ::data[ 2 ];
11
12     return std::sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2) + (z1 - z2)*(z1
       - z2));
13 }
14
15 // compara 2 clouds de PointXYZ
16 float compare(pcl::PointCloud<pcl::PointXYZ>::Ptr vertices1, pcl::PointCloud<
       pcl::PointXYZ>::Ptr vertices2, float size) {
17     int contador = 0; // Habra que cambiarlo puedo que maxint = 32767
18     int comp = (int) vertices1->size();
```

```
    float size_cube = std::sqrt( (3*(size*size)/4));
    // access each vertex
    for (int id1 = 0; id1 < vertices1->size(); id1++) {
        float min = 1000000;
        pcl::PointXYZ v1 = vertices1->points[ id1 ];
        for (int id2 = 0; id2 < vertices2->size(); id2++) {
            pcl::PointXYZ v2 = vertices2->points[ id2 ];
            float dist = dist_pcl_points(v1, v2);
            if (min >= dist) {
                min = dist;
            }
        }
        if (min <= size_cube) {
            contador++;
        }
    }
    return contador;
}

// compara 2 pointcloud3 dado cierto size
float compare_clouds(pcl::PCLPointCloud2::Ptr cloud1, pcl::PCLPointCloud2::Ptr
    cloud2, float size) {

    pcl::PointCloud<pcl::PointXYZ>::Ptr vertices1(new pcl::PointCloud<pcl::
    PointXYZ>);
    pcl::PointCloud<pcl::PointXYZ>::Ptr vertices2(new pcl::PointCloud<pcl::
    PointXYZ>);
    pcl::fromPCLPointCloud2(*cloud1, *vertices1);
    pcl::fromPCLPointCloud2(*cloud2, *vertices2);

    float c1 = compare(vertices1, vertices2, size);
    float c2 = compare(vertices2, vertices1, size);
    int tot1 = (int) vertices1->size();
    int tot2 = (int) vertices2->size();
    float hit1 = (float) (c1 * 100 / tot1);
    float hit2 = (float) (c2 * 100 / tot2);

    // Retornar o imprimir algo con los aciertos y totales
    return (hit1 >= hit2) ? hit2 : hit1 ;

}
```

Código 3.3: Algoritmo de Comparación

3.5. Análisis de Complejidad Temporal

Para la determinación de la escala se utiliza la funcion *getMinMax3D* de la librería PCL. Esta implementación recorre todos los puntos de la nube en tiempo $O(n)$ donde n es el número de puntos del modelo 3D, solo compara si es mayor menor o igual a los datos de cada coordenada.

Para la comparación entre Voxelizaciones se requiere de dos recorridos para la búsqueda

de los puntos similares, la cual está actualmente implementada en tiempo $O(n^2)$ (o $n \cdot m$, con n y m los tamaños de las nubes de puntos).

Si consideraramos recorridos de puntos uno a uno en ambas comparaciones, asumiendo un modelo 3D con gran densidad de puntos ($\sim$ 10.000 puntos) tendríamos aproximadamente $10.000^{\,2} = 100.000.000$ comparaciones para el algoritmo de comparación, lo cual no es rápido de procesar. Para esto la voxelización realiza un gran trabajo en simplificar la cantidad de puntos de la nube. Esta simplificación no es predecible según la cantidad de puntos pero si respecto a la disposición de los modelos, por lo que es importante tener el tamaño de voxel ideal para ciertos tipos de modelos.

Para un objeto de 10.000 puntos, el preprocesamiento utilizando la voxelización toma aproximadamente 8 milisegundos, mientras que la comparación de nubes utilizando los voxels toma 43 milisegundos. Si quitamos algunos elementos del preprocesamiento (el filtro de voxels), la comparación de puntos se dispara a 160 segundos, por lo que se asume que el filtrado de la voxelización sirve para la simplificación del tiempo de ejecución además de la comparación volumétrica.

Bajo la suposición que los datos de los voxels se almacenen en una estructura de árbol (Por ejemplo Octree), se pueden disminuir los tiempos de búsqueda de cierto punto a una distancia determinada respecto a otro, dado que la estructura de Octree logra converger a los puntos deseados más rápido, pagando parte de ese costo de procesamiento en la inserción de un punto nuevo al árbol.

Capítulo 4

Evaluación Experimental

Dado el estado actual de la Impresora 3D de Concreto (no está terminada su construcción) se diseñaron experimentos sintéticos para evaluar la efectividad del algoritmo deformando el objeto original dado ciertos porcentajes de deformación.

4.1. Descripción del Experimento Sintético

El experimento sintético consiste en lo siguiente:

Dados ciertos modelos de prueba, aplicar el algoritmo de voxelización sobre el modelo original y un modelo levemente deformado aleatoriamente. Luego de esto aplicar el algoritmo de comparación sobre el objeto original con los objetos deformados y concluir sobre los resultados obtenidos.

Como vimos en el capítulo anterior, el algoritmo de voxelización incluye etapas de escala y traslación respecto al centroide de la nube de puntos. La experimentación se basará en variar el porcentaje de deformación y el tamaño del voxel. Los tamaños de voxel variarán entre [10 cm, 50cm y 1m], mientras que los porcentajes de deformación variarán entre [1.01, 1.02, 1.03 y 1.05], este tipo de deformación implica deformar la distancia respecto al origen en ese factor. Por ejemplo si la distancia desde el origen a un punto es 5, la deformación lo dejará en 5.05, 5.10, etc. alejandolo siempre desde el origen. Asimismo la escala se realiza con un cálculo de distancias del eje Z, para realizar el crecimiento/decrecimiento de toda la nube de puntos manteniendo como invariante que el tamaño del eje Z va a ser siempre 10 metros.

Bajo estas hipótesis relacionadas a este experimento, se desea encontrar un tamaño de voxel ideal para cada una de las deformaciones controladas, con lo cual se busca tener un buen porcentaje de acierto (aceptación) para gran parte de las deformaciones. El experimento se realizará usando una serie de modelos 3D extraídos de SHREC [14]

4.2. SHREC

SHREC es una serie de *benchmarks* (3D Shape Retrieval Contest) [14] del cual se busca encontrar la efectividad de algoritmos de recuperación de forma, SHREC'17 es la doceava edición del *benchmark*. El *benchmark* está organizado en conjunto con Eurographics Workshop on 3D Object Retrieval, en el cual se especifican varias disciplinas de competencia, por ejemplo:

1. Point-Cloud Shape Retrieval of Non-Rigid Toys [13]

2. Large-scale 3D Shape Retrieval from ShapeNet Core55 [4]

3. 3D Hand Gesture Recognition Using a Depth and Skeletal Dataset [1]

De la colección de datos de prueba del *benchmark* Large-scale 3D Shape Retrieval [4] se tomaron cuatro modelos para la evaluación experimental del algoritmo. En general ninguno de estos modelos supera las dimensiones de 1 metro en ninguna de las coordenadas <X,Y,Z>, por lo que son ideales para considerar pruebas dentro de una impresora.

La especificación de cada uno de los modelos seleccionados es la siguiente:

1. Bote

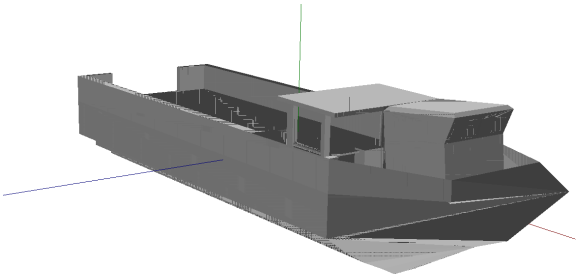

Figura 4.1: Modelo de Bote

   El modelo de bote utilizado es un modelo bastante detallado en puntos y formas por lo que lleva a una gran cantidad de puntos para analizar. Se espera que las iteraciones con tamaño de voxel más grande tengan más exactitud.

2. Botella

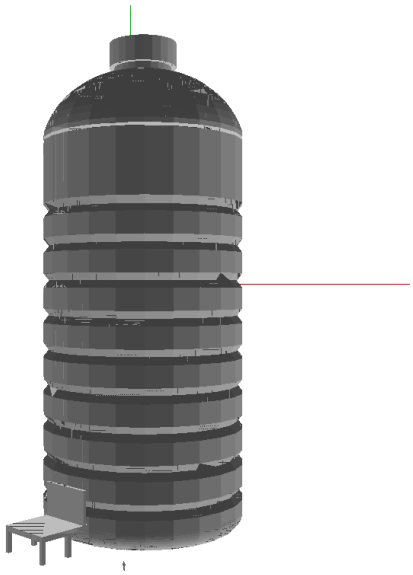

Figura 4.2: Modelo de Botella

Este modelo tiene la particularidad que tiene gran parte del modelo con una simetría en el eje Z, según el algoritmo actual de preprocesamiento y voxelización, se calcula una escala de los modelos según la distancia máxima en el eje Z por lo que el modelo es un muy buen caso para el algoritmo actual. Además se compone de una cantidad de puntos por lo que se espera tener una gran cantidad de aciertos.

3. Starbucks

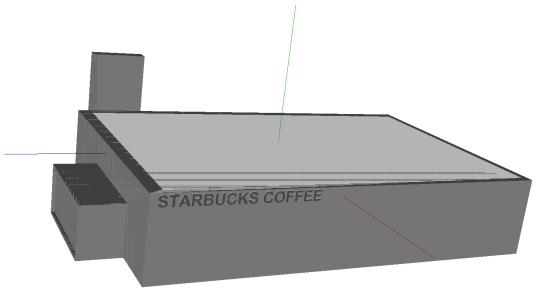

Figura 4.3: Modelo de Starbucks

Contiene un modelo de edificio comercial de Starbucks, en particular tiene muebles en los interiores del edificio lo cual podría considerarse como ruido dentro del modelo original pero se adapta bien a las nubes de puntos.

Se espera que el algoritmo tenga una gran tasa de aciertos por la alta densidad de los puntos contenidos en este modelo.

4. Tubo

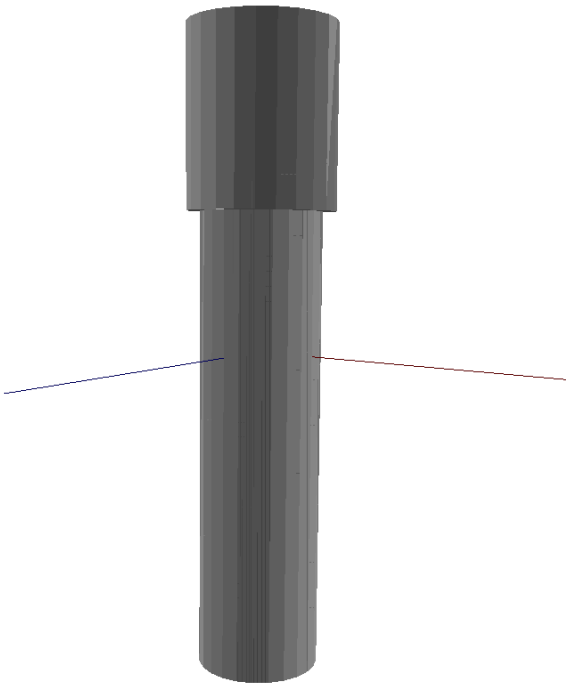

Figura 4.4: Modelo de Tubo

Modelo de un tubo, similar a un telescopio. Este modelo también posee simetría axial en el eje Z, por lo que se adaptaría muy bien al algoritmo. No posee gran densidad de puntos por lo que podría no tener tan buena tasa de aciertos para tamaños de voxels pequeños.

4.3. Implementación

La implementación del experimento descrito anteriormente se hizo gracias a la automatización de varios procesos.

Primero, se declara un arreglo contenedor de PCLPointClouds para almacenar el modelo 3D original y sus posteriores deformaciones.

Luego, se procede a iterar sobre un arreglo de números (1.01, 1.02, 1.03 y 1.05) que representan el grado de deformación (factor de deformación) que sufrirá el modelo original. De esta forma cada factor afectará azarosamente cada uno de los puntos de la nube de puntos original.

Asimismo, se crea un arreglo para almacenar el cálculo de las voxelizaciones con la voxelización del modelo original y también voxelizaciones de sus deformaciones (las deformaciones descritas anteriormente), para luego ser utilizados por el algoritmo comparador.

Finalmente, se compara el modelo original con el resto del arreglo contenedor de nubes,

los cuales representan las deformaciones del modelo original. De manera que la ejecución del algoritmo computa el nivel de acierto entre el modelo original y cada una de las deformaciones computadas.

4.4. Resultados

En este experimento se representará el porcentaje de acierto de la comparación del modelo 3D original contra el modelo 3D deformado. El número que se ve en la tabla es el porcentaje de aciertos del algoritmo, bajo los parámetros de tamaño de voxel y el porcentaje de deformación (error). Para cada configuración de parámetros se realizan diez iteraciones sobre el mismo modelo y se obtiene el promedio del porcentaje de aciertos.

Este experimento debe mostrar un crecimiento importante en el porcentaje de aciertos para las configuraciones de parámetros mientras vaya creciendo el tamaño del voxel. Puesto que mientras crece el tamaño del voxel, se pierden detalles volumétricos de la figura. También se espera que a mayor porcentaje de deformación, menor sean los porcentajes de aciertos fijando el tamaño del voxel.

Un algoritmo que tenga aciertos bajos para la primera tanda de errores es aceptable pero no es aceptable que no cumpla las hipótesis de crecimiento respecto al tamaño del voxel y al valor de deformación. Finalmente el algoritmo se ejecuta con un script utilizando *Bash* y la salida estándar del *Terminal*.

Una vez que se ejecuta el algoritmo de corrupción sintético sobre los modelos y recopilando el output de la ejecución del algoritmo los resultados obtenidos son los siguientes:

Tubo Voxelsize = 0.1 m

	Error		
Iteracion	1.01	1.02	1.05
1	46.875	4.16667	0
2	46.875	4.16667	0
3	46.875	4.16667	0
4	46.875	4.16667	0
5	48.9583	47.9167	4.16667
6	48.9583	47.9167	4.16667
7	48.9583	47.9167	4.16667
8	48.9583	47.9167	4.16667
9	48.9583	47.9167	4.16667
10	48.9583	47.9167	4.16667
Average	47.91665	26.041685	2.083335

Tabla 4.1: Resultados Experimento 1 para Tubo (a)

Tubo Voxelsize = 0.5 m

	Error		
Iteracion	1.01	1.02	1.05
1	97.9167	62.5	16.6667
2	97.9167	62.5	16.6667
3	97.9167	62.5	16.6667
4	97.9167	62.5	16.6667
5	97.9167	62.5	16.6667
6	97.9167	62.5	16.6667
7	97.9167	62.5	16.6667
8	97.9167	62.5	16.6667
9	97.9167	62.5	16.6667
10	97.9167	62.5	16.6667
Average	97.9167	62.5	16.6667

Tabla 4.2: Resultados Experimento 1 para Tubo (b)

Tubo Voxelsize = 1 m

	Error		
Iteracion	1.01	1.02	1.05
1	95.7447	77.6596	53.012
2	100	87.0968	89.011
3	100	87.0968	89.011
4	100	87.0968	89.011
5	100	87.0968	89.011
6	100	87.0968	89.011
7	100	87.0968	89.011
8	100	87.0968	89.011
9	100	87.0968	89.011
10	100	87.0968	89.011
Average	85.91715	85.91715	84.511125

Tabla 4.3: Resultados Experimento 1 para Tubo (c)

Starbucks Voxelsize = 0.1 m

	Error		
Iteracion	1.01	1.02	1.05
1	94.2342	92.973	77.9313
2	93.7538	93.8739	78.1778
3	93.8739	93.6336	78.4076
4	94.0541	93.8138	78.2763
5	93.8138	93.3934	77.5801
6	93.994	94.1141	77.9812
7	93.8739	93.8138	78.228
8	93.6937	94.1742	78.1703
9	93.3333	93.7538	78.3267
10	92.8529	93.3934	77.8414
Average	93.911425	93.723725	78.094075

Tabla 4.4: Resultados Experimento 1 para Starbucks (a)

Starbucks Voxelsize = 0.5 m

	Error		
Iteracion	1.01	1.02	1.05
1	100	99.1597	91.9118
2	100	99.1597	91.9118
3	100	99.1597	93.2773
4	100	99.1597	93.2773
5	100	99.1597	93.2773
6	100	99.1597	93.2773
7	100	97.479	92.7007
8	100	97.479	92.7007
9	100	97.479	92.7007
10	100	98.3333	94.0299
Average	100	98.739525	92.791775

Tabla 4.5: Resultados Experimento 1 para Starbucks (b)

Starbucks Voxelsize = 1 m

	Error		
Iteracion	1.01	1.02	1.05
1	100	100	94.7368
2	100	100	94.7368
3	100	97.2973	97.2222
4	100	97.2973	97.2222
5	100	97.2973	97.2222
6	100	95.9459	96.0526
7	100	95.9459	96.0526
8	100	95.9459	96.0526
9	100	95.9459	90.9091
10	100	95.9459	90.9091
Average	100	97.4662	96.16225

Tabla 4.6: Resultados Experimento 1 para Starbucks (c)

Bote Voxelsize = 0.1 m

	Error		
Iteracion	1.01	1.02	1.05
1	96.7756	96.8216	77.3177
2	97.1537	96.8691	76.6447
3	97.0588	96.7742	76.3071
4	97.1537	96.8691	84.0315
5	97.2011	96.7268	76.5265
6	97.2486	96.722	75.8533
7	96.7268	96.6793	75.9341
8	96.6793	96.9193	76.8976
9	97.064	96.6319	94.6573
10	96.77	96.5196	77.6152
Average	96.9997	96.797675	77.4390625

Tabla 4.7: Resultados Experimento 1 para Bote (a)

Bote Voxelsize = 0.5 m

	Error		
Iteracion	1.01	1.02	1.05
1	93.9297	78.3237	72.0994
2	93.6102	79.2398	72.0222
3	93.6102	76.0807	70.7865
4	94.2492	76.3158	69.7143
5	94.2492	76.3158	69.7143
6	93.6102	76.9911	76.7313
7	94.2492	76.1765	74.7191
8	94.8882	76.5396	72.0548
9	94.5687	78.5507	75.5556
10	94.5687	78.5507	75.5556
Average	94.0495125	76.997875	72.2302375

Tabla 4.8: Resultados Experimento 1 para Bote (b)

Bote Voxelsize = 1 m

	Error		
Iteracion	1.01	1.02	1.05
1	93.956	94.709	77.561
2	96.1326	94.8186	75.8454
3	95.0549	93.6508	77.2947
4	95.0549	95.0549	73.5294
5	95.0549	95.0549	73.5294
6	93.956	93.75	77.1144
7	96.0894	92.1053	76.3033
8	96.1326	94.5055	79.8077
9	95.6044	91.4894	76.1421
10	95.6044	91.4894	76.1421
Average	95.1789125	94.206125	76.3731625

Tabla 4.9: Resultados Experimento 1 para Bote (c)

Botella Voxelsize = 0.1 m

	Error		
Iteracion	1.01	1.02	1.05
1	94.744	50.9215	51.5358
2	94.198	50.3754	53.6519
3	95.0685	52.1088	54.198
4	94.8805	46.6894	47.7816
5	94.8805	46.6894	47.7816
6	94.6648	54.8123	53.5836
7	94.198	50.5119	53.3788
8	94.2662	54.2662	48.8055
9	94.471	49.2507	50.3754
10	94.471	49.2507	50.3754
Average	94.6125625	50.7968625	51.3396

Tabla 4.10: Resultados Experimento 1 para Botella (a)

Botella Voxelsize = 0.5 m

	Error		
Iteracion	1.01	1.02	1.05
1	88.9474	88.7689	83.4773
2	88.9474	88.7689	83.4773
3	88.3369	88.6609	83.2613
4	88.3369	88.6609	83.2613
5	88.1209	89.0929	82.7214
6	88.1209	89.0929	82.7214
7	88.1209	89.0929	82.7214
8	88.4861	89.1236	83.8013
9	88.4861	89.1236	83.8013
10	89.4572	88.2911	81.5335
Average	88.427175	88.9077375	83.1803375

Tabla 4.11: Resultados Experimento 1 para Botella (b)

Botella Voxelsize = 1 m			
	Error		
Iteracion	1.01	1.02	1.05
1	94.6	93.014	92.7644
2	94.6	93.014	92.7644
3	94.6	93.014	92.7644
4	94.6	93.014	92.7644
5	97.7778	95.4	93.6803
6	97.7778	95.4	93.6803
7	97.7778	95.4	93.6803
8	97.7778	95.4	93.6803
9	97.7778	95.4	93.6803
10	97.7778	95.4	93.6803
Average	96.1889	94.207	93.22235

Tabla 4.12: Resultados Experimento 1 para Botella (c)

En base a estos resultados podemos destacar lo siguiente:

Como el ejercicio de la corrupción funciona de manera aleatoria, no siempre se obtiene el mismo output, por lo que es importante hacer más de una ejecución del algoritmo para obtener resultados precisos. Al hacer 10 veces el mismo experimento sobre la misma configuración de modelo, tamaño de voxel y porcentaje de corrupción, tenemos que algunos resultados no cambian mucho a lo largo del tiempo. Esto puede ser debido a la poca densidad de puntos del modelo, sin muchos puntos para modificar teniendo pocos puntos cercanos a otros.

En particular mientras más grande era el tamaño del voxel, mayor el porcentaje de aciertos. La división de voxels en un objeto con poca densidad de puntos locales genera mayores desaciertos en general. Las deformaciones de puntos si bien eran aleatorias, involucraron aumentos de posición relativos a su mismo vector de posición. Por ejemplo si el punto ya estaba lo suficientemente alejado del eje Z, bajo este concepto el punto se aleja aún más, haciendo muy dificil su acierto dentro del algoritmo de comparación.

Por alguna razón en especial, el tamaño de voxel 50cm tiene resultados relativamente peores (Ver Figuras 4.2, 4.5, 4.8 y 4.11) en la mayoría de los casos que el tamaño de voxel 10cm. Este claramente es un resultado no esperado puesto que según la implementación del algoritmo de comparación, debería dar una mayor ventana de error para la aceptación de los vóxels presentes.

4.5. Retrospectiva y Nuevo Experimento

Dados estos resultados no tan satisfactorios respecto a los tamaños de voxels y resultados, se decide refinar el experimento considerando lo siguiente:

1. La aleatoriedad del experimento es deficiente, debe ser mejorada (Se propone mejorar

la eficacia de la funcion aleatoria)

2. Descartar la normalización de escala y posición, para pasar al ajuste de modelos (alineamiento) utilizando el algoritmo ICP. Con el experimento anterior un tren de tamaño real al ser comparado con un tren de juguete, tendrían un calce muy positivo, lo cual es incorrecto para el uso real del algoritmo. Con el alineamiento se busca un calce bueno para la comparación de dos nubes sin alterar escala.
3. La deformación realizada de modo más local. Agregar una ventana de error determinada arbitrariamente a un valor similar al posible error de una impresora (alrededor de 1cm a 5cm) para luego agregar ese error como una suma sobre cada una de las coordenadas de un punto. Este error en general va a ser más pequeño de las deformaciones anteriores.

Por lo que se modificó el preprocesamiento utilizando la siguiente funcionalidad como se muestra en el Código 4.1:

```
pcl::PointCloud<pcl::PointXYZ>::Ptr vanilla_icp(pcl::PointCloud<pcl::PointXYZ>::Ptr sourceCloud, pcl::PointCloud<pcl::PointXYZ>::Ptr targetCloud){

    pcl::PointCloud<pcl::PointXYZ>::Ptr finalCloud(new pcl::PointCloud<pcl::PointXYZ>);
    // ICP object.
    pcl::IterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ> registration;
    registration.setInputSource(sourceCloud);
    registration.setInputTarget(targetCloud);
    // Set the max correspondence distance to 5cm (e.g., correspondences with higher distances will be ignored)
    registration.setMaxCorrespondenceDistance (0.1);
    // Set the maximum number of iterations (criterion 1)
    registration.setMaximumIterations (50);
    // Set the transformation epsilon (criterion 2)
    registration.setTransformationEpsilon (1e-8);
    // Set the euclidean distance difference epsilon (criterion 3)
    registration.setEuclideanFitnessEpsilon (1);

    registration.align(*finalCloud);
    return finalCloud;
}
```

Código 4.1: Alineamiento utilizando PCL

Así como cambia la función de preprocesamiento, también cambia la función de deformación para tener una implementación más cercana a las ventanas de errores que normalmente tendría una impresora. Esto se puede ver en la implementación nueva de la función *corrupt2* en el Código 4.2

```
float RandomFloat(float a, float b) {
    float random = ((float) rand()) / (float) RAND_MAX;
    float diff = b - a;
    float r = random * diff;
    return a + r;
}

// Corrupta clouds de PointXYZ
void corrupt2( pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, float error) {
    for (int i = 0; i < cloud->size(); i++) {
```

```
11          if(0 == (rand() %2)) {
12              cloud->points[i]._PointXYZ::data[ 0 ] = (float) cloud->points[i].
    _PointXYZ::data[ 0 ] + RandomFloat(-error, error);
13              cloud->points[i]._PointXYZ::data[ 1 ] = (float) cloud->points[i].
    _PointXYZ::data[ 1 ] + RandomFloat(-error, error);
14              cloud->points[i]._PointXYZ::data[ 2 ] = (float) cloud->points[i].
    _PointXYZ::data[ 2 ] + RandomFloat(-error, error);
15          }
16      }
17  }
```

Código 4.2: Deformación nueva de Puntos

Dados estos cambios tenemos los siguientes resultados del nuevo experimento, teniendo en cuenta que el voxelsize iterará desde 1cm a 10cm y el error entre 2cm y 5cm.

Tubo Voxelsize = 0.01 m

	Error		
Iteracion	0.02	0.03	0.05
1	48.9583	39.5833	36.4583
2	56.25	37.5	33.3333
3	52.0833	51.0417	28.125
4	46.875	51.0417	39.5833
5	32.2917	43.1579	31.5789
6	59.375	47.9167	44.7917
7	58.3333	39.5833	28.125
8	43.617	45.8333	55.2083
9	31.25	47.9167	15.625
10	43.75	45.8333	45.8333
AVG	47.27836	44.94079	35.86621

Tabla 4.13: Resultados Experimento 2 para Tubo (a)

Tubo Voxelsize = 0.05 m

	Error		
Iteracion	0.02	0.03	0.05
1	93.1818	91.3793	58.5714
2	94	92.3077	77.6119
3	90.9091	90	74.2424
4	92.1569	90	64.6154
5	95.4545	86.2069	53.6232
6	92	80.3571	67.1429
7	88.4615	83.9286	58.3333
8	96.0784	85.9649	52.1739
9	94.1176	87.2727	71.6667
10	89.0909	86.7924	65.0794
AVG	92.54507	87.42096	64.30605

Tabla 4.14: Resultados Experimento 2 para Tubo (b)

Tubo Voxelsize = 0.1 m

	Error		
Iteracion	0.02	0.03	0.05
1	100	94.7368	95.2381
2	100	85.7143	89.1892
3	100	94.4444	85.7143
4	96.4286	97.0588	90.4762
5	100	94.1176	91.1765
6	96.4286	94.2857	92.1053
7	100	96.9697	90.2439
8	100	91.1765	97.2222
9	100	100	90.4762
10	92.8571	91.1765	92.3077
AVG	98.57143	93.96803	91.41496

Tabla 4.15: Resultados Experimento 2 para Tubo (c)

Starbucks Voxelsize = 0.01 m

	Error		
Iteracion	0.02	0.03	0.05
1	42.1957	30.7514	21.0892
2	42.1009	31.1426	20.6279
3	42.3922	30.5114	21.2352
4	42.3109	30.9017	21.2473
5	42.3958	32	20.4262
6	42.6344	31.0278	21.4043
7	42.6674	30.1134	21.1287
8	41.8337	30.7595	20.3917
9	42.5199	30.2824	20.6886
10	41.5481	31.6283	20.6528
AVG	42.2599	30.91185	20.88919

Tabla 4.16: Resultados Experimento 2 para Starbucks (a)

Starbucks Voxelsize = 0.05 m

	Error		
Iteracion	0.02	0.03	0.05
1	90.9091	82.1862	33.7182
2	91.0526	77.2201	34.2169
3	90.3743	80.0781	33.4118
4	90	72.093	35.5609
5	91.3043	74.6154	33.2553
6	89.8936	76.0784	38.1418
7	87.8307	72.8302	34.5238
8	89.8936	77.5194	33.8061
9	90.7104	74.8031	33.7321
10	93.0636	77.8656	32.7945
AVG	90.50322	76.52895	34.31614

Tabla 4.17: Resultados Experimento 2 para Starbucks (b)

Starbucks Voxelsize = 0.1 m

	Error		
Iteracion	0.02	0.03	0.05
1	92.7536	89.8551	86.9565
2	89.8551	89.8551	88.4058
3	91.3043	89.8551	85.5072
4	88.4058	91.3043	85.5072
5	86.9565	89.8551	88.4058
6	88.4058	89.8551	84.058
7	88.4058	88.4058	89.8551
8	91.3043	89.8551	85.5072
9	91.3043	88.4058	85.5072
10	91.3043	89.8551	84.058
AVG	89.99998	89.71016	86.3768

Tabla 4.18: Resultados Experimento 2 para Starbucks (c)

Bote Voxelsize = 0.01 m

	Error		
Iteracion	0.02	0.03	0.05
1	20.6193	11.9982	6.3811
2	21.454	13.2261	6.08925
3	21.2811	13.0947	6.38298
4	20.582	12.7813	6.32209
5	21.9178	13.2763	6.51401
6	20.67	13.6182	6.41652
7	20.2981	12.2721	6.30009
8	20.8039	12.4332	5.9099
9	21.4487	12.9329	6.58307
10	19.8441	13.7439	6.42222
AVG	20.8919	12.93769	6.332123

Tabla 4.19: Resultados Experimento 2 para Bote (a)

Bote Voxelsize = 0.05 m

	Error		
Iteracion	0.02	0.03	0.05
1	89.0411	88.8112	48.2412
2	93.1507	90.411	46.6981
3	93.1034	90.411	51.9048
4	93.1034	87.9699	47.1963
5	93.1507	91.7808	50.2415
6	89.0411	86.5248	54.067
7	91.7808	86.5672	52.2388
8	92.3729	90.2098	46.2264
9	91.6667	91.7808	50.7389
10	93.1507	90.411	46.89
AVG	91.95615	89.48775	49.4443

Tabla 4.20: Resultados Experimento 2 para Bote (b)

Bote Voxelsize = 0.1 m

	Error		
Iteracion	0.02	0.03	0.05
1	100	100	95.4545
2	100	100	94.0299
3	100	100	93.8462
4	100	100	96.875
5	100	100	92.4242
6	100	100	92.5373
7	100	100	96.8254
8	100	100	89.5522
9	100	97.9592	95.3125
10	100	100	93.6508
AVG	100	99.79592	94.0508

Tabla 4.21: Resultados Experimento 2 para Bote (c)

Botella Voxelsize = 0.01 m

	Error		
Iteracion	0.02	0.03	0.05
1	50.4419	44.095	38.7915
2	49.1479	45.0352	42.0168
3	51.9375	46.3492	41.8829
4	51.3292	44.1683	41.3606
5	49.8314	45.2518	39.7576
6	49.187	43.4728	42.1433
7	47.0867	43.0114	43.1408
8	52.6244	40.4053	40.887
9	47.258	43.9924	42.5519
10	50.9804	45.108	40.8563
AVG	49.98244	44.08894	41.33887

Tabla 4.22: Resultados Experimento 2 para Botella (a)

Botella Voxelsize = 0.05 m

	Error		
Iteracion	0.02	0.03	0.05
1	95.3092	93.6538	77.1523
2	95.935	94.7859	76.4895
3	96.4286	93.7023	77.5081
4	96.162	94.3074	78.4893
5	96.1905	93.7743	79.7771
6	96.9048	91.6828	76.9481
7	96.0581	94.636	77.5444
8	96.1864	92.381	78.0096
9	95.7576	92.9791	77.1475
10	95.5466	93.346	75.7025
AVG	96.04788	93.52486	77.47684

Tabla 4.23: Resultados Experimento 2 para Botella (b)

Botella Voxelsize = 0.1 m

	Error		
Iteracion	0.02	0.03	0.05
1	99.3151	99.2	98.3051
2	100	98.7179	98.324
3	100	99.2	98.8636
4	99.2	99.2	97.6608
5	99.3056	97.351	98.8701
6	99.2	99.3421	98.2659
7	100	99.2	98.3333
8	99.2	98.4	98.2955
9	99.2	100	98.2857
10	100	97.973	98.3051
AVG	99.54207	98.8584	98.35091

Tabla 4.24: Resultados Experimento 2 para Botella (c)

Dado los resultados de los experimentos, podemos ver que para un tamaño de voxel 5cm (ver Tablas 4.14, 4.17, 4.20 y 4.23) el porcentaje de aceptación se acerca a 95 % por lo que se ve mucha mejora respecto al algoritmo de la primera experimentación. En particular no se ve la anomalía de la pérdida de efectividad que se veía en el tamaño de vóxel intermedio en el experimento 1 (ver Tablas 4.2, 4.5, 4.8 y 4.11).

El resto de las tablas nos muestran información coherente al tamaño de los voxels y la deformación. Para los tamaños pequeños con deformación alta (ver Tablas 4.13, 4.16, 4.19 y 4.22), los errores son más propensos a manifestarse dentro del algoritmo de comparación, puesto que los voxels no tienen la capacidad de enmendar estos errores y a la vez, necesitamos que los algoritmos detecten este tipo de comportamientos de manera exacta. Si utilizamos un tamaño de voxel muy pequeño tenemos grandes posibilidades de no poder compararlos

de manera efectiva (y tener un score bajo), mientras que si el tamaño es muy grande (ver Tablas 4.15, 4.18, 4.21 y 4.24) perdemos detalles importantes de la superficie original.

Bajo estas hipótesis y tomando en cuenta que los tamaños de los objetos utilizados para el *benchmark* no superan un metro en cada uno de sus ejes, se concluye que el **Experimento de Voxelización y Comparación utilizando Alineamiento con ICP y Deformación Local** posee una buena métrica para medir la efectividad de los modelos impresos para un tamaño de voxel entre 1 y 5 cm dependiendo del nivel de detalle que se quiere alcanzar.

Capítulo 5

Conclusiones

5.1. Resultados

Durante el trabajo de Memoria se buscó encontrar una manera de evaluar la exactitud de la salida de una Impresora 3D de Concreto. Para eso, se investigó maneras de representar y comparar volúmenes en modelos 3D. Esto llevó a tomar la voxelización como el método a utilizar para este trabajo, utilizando los voxels como elementos a comparar para la evaluación de la impresora.

Bajo la voxelización se definen experimentos dentro de un *dataset* de modelos 3D estándar, ajustando la escala de la nube de puntos, variando el tamaño de voxel a utilizar y deformando las figuras bajo un factor multiplicativo, variando los tamaños de voxels y el factor de deformación.

De los resultados experimentales obtenidos se observa que los resultados del experimento que tiene tamaño de voxel medio (50cm) poseen menos porcentaje de aciertos que los de tamaño pequeño, lo cual rompe una de las hipótesis del experimento. También se nota que la transformación asociada a la escala dentro del preprocesamiento no funciona para normalizar en el caso de la comparación de modelos impresos puesto que un modelo que representa un objeto pequeño sería equivalente a un modelo de un objeto grande. Esto es inaceptable para el contexto de evaluación de calidad de una Impresora 3D.

Luego de la primera iteración del experimento, notando los resultados inconsistentes y a la luz de las posibles falencias se cambian las hipótesis del experimento para mejorar el desempeño del algoritmo y acercarlo a un caso de uso real, tal cual sería una impresora con fallas locales de magnitudes bajas.

Con esto se obtuvieron mejoras en la exactitud del modelo de voxelización, ya que errores pequeños afectan menos a las voxelizaciones de tamaño grande. Además se mejora el preprocesamiento utilizando ICP, el cual tiene la particularidad de alinear de manera exitosa 2 modelos utilizando transformaciones de rotación y traslación, a la vez cumpliendo las hipótesis anteriormente descritas.

En conclusion se proponen algoritmos de evaluación para evaluar impresoras 3D, estos algoritmos siendo probados con un *dataset* de modelos 3D estándar y utilizando la voxelización con el algoritmo ICP para mejorar la exactitud de la comparación y considerar posibles cambios de orientación en la adquisición de los datos.

Bajo este experimento se proponen medidas de evaluación, las cuales llegan a una medida de aceptación de un modelo bajo los estándares del procesamiento de los voxels. Finalmente no se logró medir la efectividad de la impresora 3D de concreto puesto que a la fecha no se ha terminado la construcción ni puesta en marcha de esta.

5.2. Trabajo Futuro

Como trabajo futuro se proponen los siguientes puntos:

- La experimentación con distintas fuentes de adquisición de imágenes podría provocar muchas diferencias en la cantidad y densidad de los puntos de un modelo 3D, por lo que habría que extender el modelo para soportar modelos que tengan diferencias considerables en la cantidad de puntos para detallar ciertas zonas. Por ejemplo la Kinect al ser utilizado como fuente de nube de puntos tiene menos resolución y certeza que otras fuentes de información [25], por lo que si o si habría diferencia entre la cantidad y calidad de los puntos extraídos de Kinect con el modelo original.
- Probar el algoritmo desarrollado para medir la efectividad real de la Impresora 3D de Concreto una vez esté construída y tomar la medición como validación de las impresiones que esta haga.
- Utilizar una mejor estructura de datos para almacenar datos volumétricos y hacer consultas sobre estos, una posible implementación sería con *octree* para mejorar las búsquedas dentro de las Nubes de Puntos. En general algunas comparaciones funcionarían de manera más rápida que en la implementación actual.

Bibliografía

[1] 3d hand gesture recognition using a depth and skeletal dataset. http://www-rech.telecom-lille.fr/shrec2017-hand/, 2017. [Online; accessed 19-July-2017].

[2] 3dalign (command). https://knowledge.autodesk.com/support/autocad/learn-explore/caas/CloudHelp/cloudhelp/2016/ENU/AutoCAD-Core/files/GUID-26823DD2-BAA5-4494-81A0-713B746DD94B-htm.html, 2017. [Online; accessed 19-July-2017].

[3] Cloudcompare - open source project. http://www.cloudcompare.org/, 2017. [Online; accessed 19-July-2017].

[4] Large-scale 3d shape retrieval from shapenet core55. https://shapenet.cs.stanford.edu/shrec17/, 2017. [Online; accessed 19-July-2017].

[5] Meshlab. http://www.meshlab.net/, 2017. [Online; accessed 19-July-2017].

[6] The pcd (point cloud data) file format. http://pointclouds.org/documentation/tutorials/pcd_file_format.php, 2017. [Online; accessed 19-July-2017].

[7] PCL - Documentation. http://docs.pointclouds.org/1.7.1/classpcl_1_1_voxel_grid.html#details, 2017. [Online; accessed 19-July-2017].

[8] PCL - Downsampling a PointCloud using a VoxelGrid filter. http://pointclouds.org/documentation/tutorials/voxel_grid.php, 2017. [Online; accessed 19-July-2017].

[9] PCL - How to use iterative closest point. http://pointclouds.org/documentation/tutorials/iterative_closest_point.php, 2017. [Online; accessed 19-July-2017].

[10] PCL - Point Cloud Library (PCL). http://pointclouds.org/, 2017. [Online; accessed 19-July-2017].

[11] PCL - Using a matrix to transform a point cloud. http://pointclouds.org/documentation/tutorials/matrix_transform.php, 2017. [Online; accessed 19-July-2017].

[12] Pcl 1.7.2: Ubuntu 14.04 installation guide. https://github.com/hsean/Capstone-44-Object-Segmentation/wiki/PCL-1.7.2:-Ubuntu-14.04-Installation-Guide, 2017. [Online; accessed 19-July-2017].

[13] Point-cloud shape retrieval of non-rigid toys. https://www.cs.york.ac.uk/cvpr/pronto/, 2017. [Online; accessed 19-July-2017].

[14] Shrec2017 - 3d shape retrieval contest 2017. http://www.shrec.net/, 2017. [Online; accessed 19-July-2017].

[15] Caterina Balletti, Martina Ballarin, and Francesco Guerra. 3d printing: State of the art and future perspectives. *Journal of Cultural Heritage*, 2017.

[16] Paul J Besl, Neil D McKay, et al. A method for registration of 3-d shapes. *IEEE Transactions on pattern analysis and machine intelligence*, 14(2):239–256, 1992.

[17] Benjamin Bustos, Daniel A Keim, Dietmar Saupe, Tobias Schreck, and Dejan V Vranić. Feature-based similarity search in 3d object databases. *ACM Computing Surveys (CSUR)*, 37(4):345–387, 2005.

[18] Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. Meshlab: an open-source mesh processing tool. In *Eurographics Italian Chapter Conference*, volume 2008, pages 129–136, 2008.

[19] Jean-Luc Dugelay, Atilla Baskurt, and Mohamed Daoudi. *3D object processing: compression, indexing and watermarking.* John Wiley & Sons, 2008.

[20] Francisca Daniela Gallardo Palacios. Software de comparación de algoritmos delaunay de refinamiento de triangulaciones. 2012.

[21] Charles Gantt. Tweaktown's guide to 3d printing: Part 1 - what makes up a 3d printer? https://www.tweaktown.com/guides/5287/tweaktown-s-guide-to-3d-printing-part-1-what-makes-up-a-3d-printer-/index5.html, 2017. [Online; accessed 19-July-2017].

[22] Daniel Girardeau-Montaut. Cloudcompare-open source project. *OpenSource Project*, 2011.

[23] Arie Kaufman, Daniel Cohen, and Roni Yagel. Volume graphics. *Computer*, 26(7):51–64, 1993.

[24] Haresh Khemani. Applications of cad software: What is solid modeling? http://www.brighthubengineering.com/cad-autocad-reviews-tips/19623-applications-of-cad-software-what-is-solid-modeling/, 2008. [Online; accessed 19-July-2017].

[25] Kourosh Khoshelham and Sander Oude Elberink. Accuracy and resolution of kinect depth data for indoor mapping applications. *Sensors*, 12(2):1437–1454, 2012.

[26] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, 2009.

[27] Leydy Gómez Reyes. Análisis documental de las ventajas de la impresión 3d/documentary analysis of the advantages of 3d printing. *RECI Revista Iberoamericana de las Ciencias Computacionales e Informática*, 6(11):1–12, 2017.

[28] Szymon Rusinkiewicz and Marc Levoy. Efficient variants of the icp algorithm. In *3-D Digital Imaging and Modeling, 2001. Proceedings. Third International Conference on*, pages 145–152. IEEE, 2001.

[29] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[30] Philip Shilane, Patrick Min, Michael Kazhdan, and Thomas Funkhouser. The princeton shape benchmark. In *Shape modeling applications, 2004. Proceedings*, pages 167–178. IEEE, 2004.

[31] XW Xu and Stephen T Newman. Making cnc machine tools more open, interoperable and intelligent—a review of the technologies. *Computers in Industry*, 57(2):141–152, 2006.

Para replicar los experimentos es necesario tener el ambiente de compilación (CMake, PCL y dependencias) previamente configurado y funcionando.

Con estas instrucciones se logra ejecutar el experimento 1. El programa acepta inputs PCD (hay que hacer conversiones previas).

Para la compilación en CMake se debe usar un archivo con nombre "CMakeLists.txt", el cual contiene las dependencias de la librería PCL.

```
cmake_minimum_required(VERSION 2.6 FATAL_ERROR)

project(pcl-interactive_icp)

find_package(PCL 1.5 REQUIRED)

include_directories(${PCL_INCLUDE_DIRS})
link_directories(${PCL_LIBRARY_DIRS})
add_definitions(${PCL_DEFINITIONS})

add_executable (interactive_icp interactive_icp.cpp)
target_link_libraries (interactive_icp ${PCL_LIBRARIES})
```

Código 5.1: CMakeLists.txt

Luego de esto se procede a especificar el código del primer experimento. El primer experimento recibe como argumentos:

- El nombre del modelo en PCD (archivo.pcd)
- El tamaño del voxel (en metros)

Se adjunta primero el archivo header (que contiene la mayoría de las funciona auxiliares) y luego el código fuente del experimento.

```
#ifndef AUX_FUN_INCLUDE
#define AUX_FUN_INCLUDE


#include <iostream>
#include <fstream>
#include <sstream>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/common/common.h>
#include <pcl/common/transforms.h>


/* Your function statement here */

void ToString(std::string& out, float value);

Eigen::Vector4f compute_centroid(pcl::PointCloud<pcl::PointXYZ>::Ptr v1);

pcl::PCLPointCloud2::Ptr transform(pcl::PCLPointCloud2::Ptr cloud, bool scale)
    ;
```

```
22
23 void corrupt( pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, float percent);
24
25 float dist_pcl_points(pcl::PointXYZ v1, pcl::PointXYZ v2);
26
27 float compare(pcl::PointCloud<pcl::PointXYZ>::Ptr vertices1, pcl::PointCloud<
    pcl::PointXYZ>::Ptr vertices2, float size);
28
29 pcl::PCLPointCloud2::Ptr voxel_cloud(pcl::PCLPointCloud2::Ptr cloud, float
    size);
30
31
32 float compare_clouds(pcl::PCLPointCloud2::Ptr cloud1, pcl::PCLPointCloud2::Ptr
     cloud2, float size);
33
34 pcl::PCLPointCloud2::Ptr traslate_centroid(pcl::PCLPointCloud2::Ptr cloud);
35
36 // Devuelve un string a out
37 void ToString(std::string& out, float value){
38     std::ostringstream ss;
39     ss << value;
40     out = ss.str();
41 }
42
43 // Retorna Vector de Centroide
44 Eigen::Vector4f compute_centroid(pcl::PointCloud<pcl::PointXYZ>::Ptr v1) {
45     Eigen::Vector4f centroid;
46     pcl::compute3DCentroid(*v1, centroid);
47     return centroid;
48 }
49
50 // Transforma con traslacion y escala (si es aplicable) una Nube
    PCLPointCloud2
51
52 pcl::PCLPointCloud2::Ptr transform(pcl::PCLPointCloud2::Ptr cloud, bool scale)
     {
53     // Calcular Centroide y aplicar Trasaci n
54     pcl::PointCloud<pcl::PointXYZ>::Ptr vertices(new pcl::PointCloud<pcl::
    PointXYZ>);
55     pcl::fromPCLPointCloud2(*cloud, *vertices);
56
57     Eigen::Vector4f centroid = compute_centroid(vertices);
58     float centroidX = centroid[0];
59     float centroidY = centroid[1];
60     float centroidZ = centroid[2];
61
62     // Hasta aqui estan los valores de la matriz de traslaci n
63     // Definimos la Transformacion
64     Eigen::Affine3f transform = Eigen::Affine3f::Identity();
65
66     if (scale) {
67         pcl::PointXYZ p_min;
68         pcl::PointXYZ p_max;
69         pcl::getMinMax3D(*vertices, p_min, p_max);
70         // Suponemos un cubo de tama o 10 para las comparaciones
71
```

```
72
73          float sx = 1; // factor X de escala
74          float sy = 1; // factor Y de escala
75          float sz = 1; // factor Z de escala
76
77          float dist_max = p_max._PointXYZ::data[2] - p_min._PointXYZ::data[2];
78          float scale = 10 / dist_max;
79          sx = scale;
80          sy = scale;
81          sz = scale;
82
83          Eigen::Vector3f escala;
84
85          escala[0] = sx;
86          escala[1] = sy;
87          escala[2] = sz;
88
89          transform.translation() << -centroidX*sx, -centroidY*sy, -centroidZ*sz
    ;
90          transform.scale(escala);
91      } else {
92          transform.translation() << -centroidX, -centroidY, -centroidZ;
93      }
94
95      // Definicion de nube nueva
96      pcl::PointCloud<pcl::PointXYZ>::Ptr transformed_cloud(new pcl::PointCloud<
    pcl::PointXYZ> ());
97      // Aplicar la Transformaci n
98      pcl::transformPointCloud(*vertices, *transformed_cloud, transform);
99
100     pcl::PCLPointCloud2::Ptr transformed(new pcl::PCLPointCloud2());
101     pcl::toPCLPointCloud2(*transformed_cloud, *transformed);
102
103     return transformed;
104
105
106
107 }
108
109 // Corrupta clouds de PointXYZ
110 void corrupt( pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, float percent) {
111     // access each vertex without assigning anything
112     for (int i = 0; i < cloud->size(); i++) {
113         if(0 == (rand() %2)) {
114             cloud->points[i]._PointXYZ::data[ 0 ] = (float) cloud->points[i].
    _PointXYZ::data[ 0 ] * percent;
115             cloud->points[i]._PointXYZ::data[ 1 ] = (float) cloud->points[i].
    _PointXYZ::data[ 1 ] * percent;
116             cloud->points[i]._PointXYZ::data[ 2 ] = (float) cloud->points[i].
    _PointXYZ::data[ 2 ] * percent;
117         }
118     }
119 }
120
121 // calcula distancia entre 2 PointXYZ
122 float dist_pcl_points(pcl::PointXYZ v1, pcl::PointXYZ v2) {
```

```
123     float x1, x2, y1, y2, z1, z2;
124 
125     x1 = v1._PointXYZ::data[ 0 ];
126     y1 = v1._PointXYZ::data[ 1 ];
127     z1 = v1._PointXYZ::data[ 2 ];
128     x2 = v2._PointXYZ::data[ 0 ];
129     y2 = v2._PointXYZ::data[ 1 ];
130     z2 = v2._PointXYZ::data[ 2 ];
131 
132     return std::sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2) + (z1 - z2)*(z1
    - z2));
133 }
134 
135 // compara 2 clouds de PointXYZ
136 float compare(pcl::PointCloud<pcl::PointXYZ>::Ptr vertices1, pcl::PointCloud<
    pcl::PointXYZ>::Ptr vertices2, float size) {
137     int contador = 0; // Habra que cambiarlo puedo que maxint = 32767
138     int comp = (int) vertices1->size();
139     float size_cube = std::sqrt( (3*(size*size)/4));
140     // access each vertex
141     for (int id1 = 0; id1 < vertices1->size(); id1++) {
142         float min = 1000000;
143         pcl::PointXYZ v1 = vertices1->points[ id1 ];
144         for (int id2 = 0; id2 < vertices2->size(); id2++) {
145             pcl::PointXYZ v2 = vertices2->points[ id2 ];
146             float dist = dist_pcl_points(v1, v2);
147             if (min >= dist) {
148                 min = dist;
149             }
150         }
151         if (min <= size_cube) {
152             contador++;
153         }
154     }
155     return contador;
156 }
157 
158 // Retorna nube de puntos filtrada segun Voxels de tama o size
159 pcl::PCLPointCloud2::Ptr voxel_cloud(pcl::PCLPointCloud2::Ptr cloud, float
    size) {
160     pcl::PCLPointCloud2::Ptr cloud_filtered = transform(cloud, true);
161     // Create the filtering object
162     pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
163     sor.setInputCloud(cloud_filtered);
164     sor.setLeafSize(size, size, size);
165     sor.filter(*cloud_filtered);
166     return cloud_filtered;
167 }
168 
169 // compara 2 pointcloud3 dado cierto size
170 float compare_clouds(pcl::PCLPointCloud2::Ptr cloud1, pcl::PCLPointCloud2::Ptr
     cloud2, float size) {
171 
172     pcl::PointCloud<pcl::PointXYZ>::Ptr vertices1(new pcl::PointCloud<pcl::
    PointXYZ>);
173     pcl::PointCloud<pcl::PointXYZ>::Ptr vertices2(new pcl::PointCloud<pcl::
```

```
    PointXYZ>);
174     pcl::fromPCLPointCloud2(*cloud1, *vertices1);
175     pcl::fromPCLPointCloud2(*cloud2, *vertices2);
176 
177     float c1 = compare(vertices1, vertices2, size);
178     float c2 = compare(vertices2, vertices1, size);
179     int tot1 = (int) vertices1->size();
180     int tot2 = (int) vertices2->size();
181     float hit1 = (float) (c1 * 100 / tot1);
182     float hit2 = (float) (c2 * 100 / tot2);
183 //    std::cerr << "Aciertos " << c1 << " " << c2 << "\n";
184 //    std::cerr << "Puntos totales " << tot1 << " " << tot2 << "\n";
185 
186     // Retornar o imprimir algo con los aciertos y totales
187     return (hit1 >= hit2) ? hit2 : hit1 ;
188 
189 }
190 
191 // Recibe un pointcloud2 y se transforma en un pointcloud2 centrado en el
    centroide
192 pcl::PCLPointCloud2::Ptr traslate_centroid(pcl::PCLPointCloud2::Ptr cloud) {
193     // Calcular Centroide y aplicar Trasaci n
194     pcl::PointCloud<pcl::PointXYZ>::Ptr vertices(new pcl::PointCloud<pcl::
    PointXYZ>);
195     pcl::fromPCLPointCloud2(*cloud, *vertices);
196 
197     Eigen::Vector4f centroid = compute_centroid(vertices);
198     float centroidX = centroid[0];
199     float centroidY = centroid[1];
200     float centroidZ = centroid[2];
201 
202     // Hasta aqui estan los valores de la matriz de traslaci n
203     // Definimos la Transformacion
204     Eigen::Affine3f transform = Eigen::Affine3f::Identity();
205     transform.translation() << -centroidX, -centroidY, -centroidZ;
206 
207     // Definicion de nube nueva
208     pcl::PointCloud<pcl::PointXYZ>::Ptr transformed_cloud(new pcl::PointCloud<
    pcl::PointXYZ> ());
209     // Aplicar la Transformaci n
210     pcl::transformPointCloud(*vertices, *transformed_cloud, transform);
211 
212     pcl::PCLPointCloud2::Ptr transformed(new pcl::PCLPointCloud2());
213     pcl::toPCLPointCloud2(*transformed_cloud, *transformed);
214 
215     return transformed;
216 
217 }
218 
219 
220 
221 #endif
```

Código 5.2: Header File 1 (“aux_functions.h”)

```
1 #include "aux_functions.h"
2 
```

```
3
4  int main (int argc, char** argv) {
5      // Mejor Random
6        struct timeval time;
7        gettimeofday(&time,NULL);
8        srand((time.tv_sec * 1000) + (time.tv_usec / 1000));
9        // End Mejor Random
10       if (argc < 3) {
11         std::cerr << "========== HELP ==========" << std::endl;
12         std::cerr << "Usage: " << argv[0] << " <filename.pcd> <size>" << std::
       endl;
13         std::cerr << "========== /HELP ==========" << std::endl;
14         return 1;
15     }
16
17     pcl::PCLPointCloud2::Ptr cloud (new pcl::PCLPointCloud2 ());
18
19     // Replace the path below with the path where you saved your file
20     std::string fullname = argv[1];
21
22     // Fill in the cloud data
23     pcl::PCDReader reader;
24     reader.read(fullname, *cloud);
25
26
27     // tama o del voxel grid
28     std::string size = argv[2];
29     float size_to_float = std::atof(size.c_str());
30
31     // trasladar por centroide
32     pcl::PCLPointCloud2::Ptr centered = transform(cloud, false);
33
34     pcl::PCLPointCloud2::Ptr pcl_array[4];
35     pcl_array[0] = centered;
36
37     int arr_size = 5;
38     // Arreglo de porcentajes de Corrupcion
39     float array[5] = {1.00 , 1.01, 1.02, 1.03, 1.05 };
40
41     std::cerr << "Porcentajes de Corrupci n : ";
42
43     for (int i = 0; i < arr_size; i++){
44         std::cerr << array[i] << " ";
45     }
46
47
48     std::cerr << std::endl;
49
50
51     // Iterar la corrupci n y almacenarlo a pcl_array
52     for (int i = 1; i <= arr_size; i++) {
53         pcl::PCLPointCloud2::Ptr transformed_cloud(new pcl::PCLPointCloud2());
54         transformed_cloud = transform(cloud,false);
55         pcl::PointCloud<pcl::PointXYZ>::Ptr vertices(new pcl::PointCloud<pcl::
       PointXYZ>);
56         pcl::fromPCLPointCloud2(*transformed_cloud, *vertices);
```

```
57
58          float percent = array[i-1];
59
60          if (percent != 1.0){
61              // random de error. se multiplica por percent
62              float error = 0.1 * percent;
63              corrupt(vertices, percent);
64          }
65
66          // Volver a PointCloud2
67          pcl::PCLPointCloud2::Ptr cloud_filtered(new pcl::PCLPointCloud2());
68          pcl::toPCLPointCloud2(*vertices, *cloud_filtered);
69
70          // Almacenar en Arreglo de PCL2
71          pcl_array[i] = cloud_filtered;
72
73      }
74      std::cerr << "Modelos Corruptos Procesados" << std::endl;
75      std::cerr << "Resultados Finales ";
76      // Aplicar Voxelizaci n sobre todo pcl_array
77      pcl_array[0] = voxel_cloud(pcl_array[0], size_to_float);
78      for (int i = 1; i <= arr_size; i++){
79         pcl_array[i] = voxel_cloud(pcl_array[i], size_to_float);
80         float c1 = compare_clouds(pcl_array[0], pcl_array[i], size_to_float);
81         std::cerr << c1 << " " ;
82      }
83      std::cerr << std::endl;
84
85
86      // // Comparar pcl_array[0] con [1] [2] y [3]
87      // float c1 = compare_clouds(pcl_array[0], pcl_array[1], size_to_float);
88      // float c2 = compare_clouds(pcl_array[0], pcl_array[2], size_to_float);
89      // float c3 = compare_clouds(pcl_array[0], pcl_array[3], size_to_float);
90
91
92
93
94      return (0);
95  }
```

Código 5.3: Experimento 1

Luego el experimento 2 soporta archivos OBJ, puesto que los fuentes del dataset de SHREC eran OBJ cambia el input del experimento a OBJ (sin convertir como en el experimento anterior)

Finalmente se especifica el código del segundo experimento. El segundo experimento recibe como argumentos:

- El nombre del modelo en OBJ
- -v o –voxel si se desea utilizar filtro de voxelización
- -i o –icp si se desea utilizar alineamiento con ICP
- El tamaño del voxel (en metros) para la comparación volumétrica
- El tamaño de las deformaciones generadas aleatoriamente

Se adjunta primero el archivo header (que contiene la mayoría de las funciona auxiliares) y luego el código fuente del experimento.

```
#ifndef AUX_FUN_INCLUDE
#define AUX_FUN_INCLUDE


#include <iostream>
#include <fstream>
#include <sstream>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/common/common.h>
#include <pcl/common/transforms.h>


/* Your function statement here */

void ToString(std::string& out, float value);

Eigen::Vector4f compute_centroid(pcl::PointCloud<pcl::PointXYZ>::Ptr v1);

pcl::PCLPointCloud2::Ptr transform(pcl::PCLPointCloud2::Ptr cloud, bool scale)
    ;

void corrupt( pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, float percent);

float dist_pcl_points(pcl::PointXYZ v1, pcl::PointXYZ v2);

float compare(pcl::PointCloud<pcl::PointXYZ>::Ptr vertices1, pcl::PointCloud<
    pcl::PointXYZ>::Ptr vertices2, float size);

pcl::PCLPointCloud2::Ptr voxel_cloud(pcl::PCLPointCloud2::Ptr cloud, float
    size);


float compare_clouds(pcl::PCLPointCloud2::Ptr cloud1, pcl::PCLPointCloud2::Ptr
     cloud2, float size);

pcl::PCLPointCloud2::Ptr traslate_centroid(pcl::PCLPointCloud2::Ptr cloud);


float RandomFloat(float a, float b) {
    float random = ((float) rand()) / (float) RAND_MAX;
    float diff = b - a;
    float r = random * diff;
    return a + r;
}

// Devuelve un string a out
void ToString(std::string& out, float value){
    std::ostringstream ss;
    ss << value;
    out = ss.str();
}

```

```
51 // Retorna Vector de Centroide
52 Eigen::Vector4f compute_centroid(pcl::PointCloud<pcl::PointXYZ>::Ptr v1) {
53     Eigen::Vector4f centroid;
54     pcl::compute3DCentroid(*v1, centroid);
55     return centroid;
56 }
57
58 // Transforma con traslacion y escala (si es aplicable) una Nube
      PCLPointCloud2
59
60 pcl::PCLPointCloud2::Ptr transform(pcl::PCLPointCloud2::Ptr cloud, bool scale)
       {
61     // Calcular Centroide y aplicar Trasaci n
62     pcl::PointCloud<pcl::PointXYZ>::Ptr vertices(new pcl::PointCloud<pcl::
      PointXYZ>);
63     pcl::fromPCLPointCloud2(*cloud, *vertices);
64
65     Eigen::Vector4f centroid = compute_centroid(vertices);
66     float centroidX = centroid[0];
67     float centroidY = centroid[1];
68     float centroidZ = centroid[2];
69
70     // Hasta aqui estan los valores de la matriz de traslaci n
71     // Definimos la Transformacion
72     Eigen::Affine3f transform = Eigen::Affine3f::Identity();
73
74     if (scale) {
75         pcl::PointXYZ p_min;
76         pcl::PointXYZ p_max;
77         pcl::getMinMax3D(*vertices, p_min, p_max);
78         // Suponemos un cubo de tama o 10 para las comparaciones
79
80
81         float sx = 1; // factor X de escala
82         float sy = 1; // factor Y de escala
83         float sz = 1; // factor Z de escala
84
85         float dist_max = p_max._PointXYZ::data[2] - p_min._PointXYZ::data[2];
86         float scale = 10 / dist_max;
87         sx = scale;
88         sy = scale;
89         sz = scale;
90
91         Eigen::Vector3f escala;
92
93         escala[0] = sx;
94         escala[1] = sy;
95         escala[2] = sz;
96
97         transform.translation() << -centroidX*sx, -centroidY*sy, -centroidZ*sz
      ;
98         transform.scale(escala);
99     } else {
100         transform.translation() << -centroidX, -centroidY, -centroidZ;
101     }
102
```

```
103     // Definicion de nube nueva
104     pcl::PointCloud<pcl::PointXYZ>::Ptr transformed_cloud(new pcl::PointCloud<
        pcl::PointXYZ> ());
105     // Aplicar la Transformaci n
106     pcl::transformPointCloud(*vertices, *transformed_cloud, transform);
107 
108     pcl::PCLPointCloud2::Ptr transformed(new pcl::PCLPointCloud2());
109     pcl::toPCLPointCloud2(*transformed_cloud, *transformed);
110 
111     return transformed;
112 
113 
114 
115 }
116 
117 // Corrupta clouds de PointXYZ
118 void corrupt( pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, float percent) {
119     // access each vertex without assigning anything
120     for (int i = 0; i < cloud->size(); i++) {
121         if(0 == (rand() %2)) {
122             cloud->points[i]._PointXYZ::data[ 0 ] = (float) cloud->points[i].
        _PointXYZ::data[ 0 ] * percent;
123             cloud->points[i]._PointXYZ::data[ 1 ] = (float) cloud->points[i].
        _PointXYZ::data[ 1 ] * percent;
124             cloud->points[i]._PointXYZ::data[ 2 ] = (float) cloud->points[i].
        _PointXYZ::data[ 2 ] * percent;
125         }
126     }
127 }
128 
129 // calcula distancia entre 2 PointXYZ
130 float dist_pcl_points(pcl::PointXYZ v1, pcl::PointXYZ v2) {
131     float x1, x2, y1, y2, z1, z2;
132 
133     x1 = v1._PointXYZ::data[ 0 ];
134     y1 = v1._PointXYZ::data[ 1 ];
135     z1 = v1._PointXYZ::data[ 2 ];
136     x2 = v2._PointXYZ::data[ 0 ];
137     y2 = v2._PointXYZ::data[ 1 ];
138     z2 = v2._PointXYZ::data[ 2 ];
139 
140     return std::sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2) + (z1 - z2)*(z1
        - z2));
141 }
142 
143 // compara 2 clouds de PointXYZ
144 float compare(pcl::PointCloud<pcl::PointXYZ>::Ptr vertices1, pcl::PointCloud<
        pcl::PointXYZ>::Ptr vertices2, float size) {
145     int contador = 0; // Habra que cambiarlo puedo que maxint = 32767
146     int comp = (int) vertices1->size();
147     float size_cube = std::sqrt( (3*(size*size)/4));
148     // access each vertex
149     for (int id1 = 0; id1 < vertices1->size(); id1++) {
150         float min = 1000000;
151         pcl::PointXYZ v1 = vertices1->points[ id1 ];
152         for (int id2 = 0; id2 < vertices2->size(); id2++) {
```

```
153             pcl::PointXYZ v2 = vertices2->points[ id2 ];
154             float dist = dist_pcl_points(v1, v2);
155             if (min >= dist) {
156                 min = dist;
157             }
158         }
159         if (min <= size_cube) {
160             contador++;
161         }
162     }
163     return contador;
164 }
165 
166 // Retorna nube de puntos filtrada segun Voxels de tama o size
167 pcl::PCLPointCloud2::Ptr voxel_cloud(pcl::PCLPointCloud2::Ptr cloud, float
        size) {
168     pcl::PCLPointCloud2::Ptr cloud_filtered(new pcl::PCLPointCloud2());
169     *cloud_filtered = *cloud;
170     // Create the filtering object
171     pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
172     sor.setInputCloud(cloud_filtered);
173     sor.setLeafSize(size, size, size);
174     sor.filter(*cloud_filtered);
175     return cloud_filtered;
176 }
177 
178 // compara 2 pointcloud3 dado cierto size
179 float compare_clouds(pcl::PCLPointCloud2::Ptr cloud1, pcl::PCLPointCloud2::Ptr
         cloud2, float size) {
180 
181     pcl::PointCloud<pcl::PointXYZ>::Ptr vertices1(new pcl::PointCloud<pcl::
        PointXYZ>);
182     pcl::PointCloud<pcl::PointXYZ>::Ptr vertices2(new pcl::PointCloud<pcl::
        PointXYZ>);
183     pcl::fromPCLPointCloud2(*cloud1, *vertices1);
184     pcl::fromPCLPointCloud2(*cloud2, *vertices2);
185 
186     float c1 = compare(vertices1, vertices2, size);
187     float c2 = compare(vertices2, vertices1, size);
188     int tot1 = (int) vertices1->size();
189     int tot2 = (int) vertices2->size();
190     float hit1 = (float) (c1 * 100 / tot1);
191     float hit2 = (float) (c2 * 100 / tot2);
192 
193     // Retornar o imprimir algo con los aciertos y totales
194     return (hit1 >= hit2) ? hit2 : hit1 ;
195 
196 }
197 
198 // Recibe un pointcloud2 y se transforma en un pointcloud2 centrado en el
        centroide
199 pcl::PCLPointCloud2::Ptr traslate_centroid(pcl::PCLPointCloud2::Ptr cloud) {
200     // Calcular Centroide y aplicar Trasaci n
201     pcl::PointCloud<pcl::PointXYZ>::Ptr vertices(new pcl::PointCloud<pcl::
        PointXYZ>);
202     pcl::fromPCLPointCloud2(*cloud, *vertices);
```

```
203
204     Eigen::Vector4f centroid = compute_centroid(vertices);
205     float centroidX = centroid[0];
206     float centroidY = centroid[1];
207     float centroidZ = centroid[2];
208
209     // Hasta aqui estan los valores de la matriz de traslaci n
210     // Definimos la Transformacion
211     Eigen::Affine3f transform = Eigen::Affine3f::Identity();
212     transform.translation() << -centroidX, -centroidY, -centroidZ;
213
214     // Definicion de nube nueva
215     pcl::PointCloud<pcl::PointXYZ>::Ptr transformed_cloud(new pcl::PointCloud<
        pcl::PointXYZ> ());
216     // Aplicar la Transformaci n
217     pcl::transformPointCloud(*vertices, *transformed_cloud, transform);
218
219     pcl::PCLPointCloud2::Ptr transformed(new pcl::PCLPointCloud2());
220     pcl::toPCLPointCloud2(*transformed_cloud, *transformed);
221
222     return transformed;
223
224 }
225
226 pcl::PCLPointCloud2::Ptr transform2(pcl::PCLPointCloud2::Ptr cloud, bool scale
        ) {
227     // Calcular Centroide y aplicar Trasaci n
228     pcl::PointCloud<pcl::PointXYZ>::Ptr vertices(new pcl::PointCloud<pcl::
        PointXYZ>);
229     pcl::fromPCLPointCloud2(*cloud, *vertices);
230
231     Eigen::Vector4f centroid = compute_centroid(vertices);
232     float centroidX = centroid[0];
233     float centroidY = centroid[1];
234     float centroidZ = centroid[2];
235
236     // Hasta aqui estan los valores de la matriz de traslaci n
237     // Definimos la Transformacion
238     Eigen::Affine3f transform = Eigen::Affine3f::Identity();
239
240     if (scale) {
241         pcl::PointXYZ p_min;
242         pcl::PointXYZ p_max;
243         pcl::getMinMax3D(*vertices, p_min, p_max);
244         // Suponemos un cubo de tama o 10 para las comparaciones
245
246
247         float sx = 1; // factor X de escala
248         float sy = 1; // factor Y de escala
249         float sz = 1; // factor Z de escala
250
251         float dist_max = p_max._PointXYZ::data[2] - p_min._PointXYZ::data[2];
252         float scale = 10 / dist_max;
253         sx = scale;
254         sy = scale;
255         sz = scale;
```

```
256
257         Eigen::Vector3f escala;
258
259         escala[0] = sx;
260         escala[1] = sy;
261         escala[2] = sz;
262
263         transform.translation() << -centroidX*sx, -centroidY*sy, -centroidZ*sz
    ;
264         transform.scale(escala);
265     } else {
266         transform.translation() << -centroidX, -centroidY, -centroidZ;
267     }
268
269     // Definicion de nube nueva
270     pcl::PointCloud<pcl::PointXYZ>::Ptr transformed_cloud(new pcl::PointCloud<
    pcl::PointXYZ> ());
271     // Aplicar la Transformaci n
272     pcl::transformPointCloud(*vertices, *transformed_cloud, transform);
273
274     pcl::PCLPointCloud2::Ptr transformed(new pcl::PCLPointCloud2());
275     pcl::toPCLPointCloud2(*transformed_cloud, *transformed);
276
277     return transformed;
278
279 }
280
281 // Corrupta clouds de PointXYZ
282 void corrupt2( pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, float error) {
283     // access each vertex without assigning anything
284     for (int i = 0; i < cloud->size(); i++) {
285         if(0 == (rand() % 2)) {
286             cloud->points[i]._PointXYZ::data[ 0 ] = (float) cloud->points[i].
    _PointXYZ::data[ 0 ] + RandomFloat(-error, error);
287             cloud->points[i]._PointXYZ::data[ 1 ] = (float) cloud->points[i].
    _PointXYZ::data[ 1 ] + RandomFloat(-error, error);
288             cloud->points[i]._PointXYZ::data[ 2 ] = (float) cloud->points[i].
    _PointXYZ::data[ 2 ] + RandomFloat(-error, error);
289         }
290     }
291 }
292
293 pcl::PointCloud<pcl::PointXYZ>::Ptr vanilla_icp(pcl::PointCloud<pcl::PointXYZ
    >::Ptr sourceCloud, pcl::PointCloud<pcl::PointXYZ>::Ptr targetCloud){
294
295     pcl::PointCloud<pcl::PointXYZ>::Ptr finalCloud(new pcl::PointCloud<pcl::
    PointXYZ>);
296     // ICP object.
297     pcl::IterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ> registration;
298     registration.setInputSource(sourceCloud);
299     registration.setInputTarget(targetCloud);
300     // Set the max correspondence distance to 5cm (e.g., correspondences with
    higher distances will be ignored)
301     registration.setMaxCorrespondenceDistance (0.1);
302     // Set the maximum number of iterations (criterion 1)
303     registration.setMaximumIterations (50);
```

```
304     // Set the transformation epsilon (criterion 2)
305     registration.setTransformationEpsilon (1e-8);
306     // Set the euclidean distance difference epsilon (criterion 3)
307     registration.setEuclideanFitnessEpsilon (1);
308 
309     registration.align(*finalCloud);
310 
311     if (registration.hasConverged()){
312         // std::cout << "ICP converged." << std::endl
313                   // << "The score is " << registration.getFitnessScore() <<
    std::endl;
314         // std::cout << "Transformation matrix:" << std::endl;
315         // std::cout << registration.getFinalTransformation() << std::endl;
316     }
317 
318     // else std::cout << "ICP did not converge." << std::endl;
319 
320     return finalCloud;
321 }
322 
323 #endif
```

Código 5.4: Header File 2 (“aux_functions.h”)

```
1  #include <iostream>
2  #include <string>
3  
4  #include <pcl/io/obj_io.h>
5  #include <pcl/io/ply_io.h>
6  #include <pcl/point_types.h>
7  #include <pcl/registration/icp.h>
8  #include <pcl/visualization/pcl_visualizer.h>
9  #include <pcl/console/time.h>   // TicToc
10 
11 #include "aux_functions.h"
12 
13 typedef pcl::PointXYZ PointT;
14 typedef pcl::PointCloud<PointT> PointCloudT;
15 
16 bool has_suffix(const std::string &str, const std::string &suffix)
17 {
18     return str.size() >= suffix.size() &&
19            str.compare(str.size() - suffix.size(), suffix.size(), suffix) ==
    0;
20 }
21 
22 
23 int main (int argc, char* argv[]){
24 
25   if (argc < 6  ) {
26       std::cerr << "========== HELP ==========" << std::endl;
27       std::cerr << "Usage: " << argv[0] << " <filename.obj|ply> <-v|--voxel>
    <-i|--i> <size_of_voxel_m> <max_error>" << std::endl;
28       std::cerr << "========== /HELP ==========" << std::endl;
29       return 1;
30   }
31 
```

```
32    // The point clouds we will be using
33    PointCloudT::Ptr cloud_in (new PointCloudT);  // Original point cloud
34    PointCloudT::Ptr cloud_out (new PointCloudT);  // Transformed point cloud
35    PointCloudT::Ptr cloud_icp (new PointCloudT);  // ICP output point cloud
36
37  // Mejor Random
38    struct timeval time, time2;
39    gettimeofday(&time,NULL);
40    srand((time.tv_sec * 1000) + (time.tv_usec / 1000));
41    // End Mejor Random
42
43    int iterations = 1;  // Default number of ICP iterations
44
45  // Leer modelos calos
46    if(has_suffix(argv[1],".obj")){
47      pcl::io::loadOBJFile(argv[1], *cloud_in);
48    }
49    else if(has_suffix(argv[1],".ply")){
50      pcl::io::loadPLYFile (argv[1], *cloud_in);
51    }
52    else{
53      PCL_ERROR ("Error loading cloud %s.\n", argv[1]);
54      return (-1);
55    }
56
57    // std::cout << cloud_in->size () << " points " <<  std::endl;
58
59    *cloud_out = *cloud_in;
60
61
62    std::string arg2(argv[2]);
63    std::string arg3(argv[3]);
64    std::string size(argv[4]);
65    std::string error_str(argv[5]);
66
67    float size_to_f = std::atof(size.c_str());
68
69    // Deformar la forma
70    // float error = 0.02f;
71    float error = std::atof(error_str.c_str());
72    corrupt2(cloud_out, error);
73    // std::cout << "Corrupci n Completa con deformacion = " << error_str <<
       std::endl;
74
75
76    // Parametros de Distancias
77    pcl::PointXYZ p_min;
78    pcl::PointXYZ p_max;
79    pcl::getMinMax3D(*cloud_in, p_min, p_max);
80
81    float dist_max_x = p_max._PointXYZ::data[0] - p_min._PointXYZ::data[0];
82    float dist_max_y = p_max._PointXYZ::data[1] - p_min._PointXYZ::data[1];
83    float dist_max_z = p_max._PointXYZ::data[2] - p_min._PointXYZ::data[2];
84
85    // std::cout << "Distancias M ximas: " << dist_max_x << " " << dist_max_y
       << " " << dist_max_z << std::endl;
```

```
86
87
88  if (arg3 == "-i" || arg3 == "--icp"){
89    // Voxelizar el source y el corrupto
90    // std::cout << "Aplicando ICP " << std::endl;
91    // cloud out => source
92    // cloud in => target
93    // queremos que el cloud_out se transforme en cloud_in
94    cloud_out = vanilla_icp(cloud_out, cloud_in);
95    gettimeofday(&time2,NULL);
96
97    // imprimir diferencia de tiempos diciendo que fue del ICP
98    std::cout << ((time2.tv_sec - time.tv_sec)*1000000L+time2.tv_usec) - time.
     tv_usec << " Microsegundos para el ICP" << std::endl;
99
100   gettimeofday(&time,NULL);
101 }
102
103
104 pcl::PCLPointCloud2::Ptr transformed_cloud1(new pcl::PCLPointCloud2());
105 pcl::PCLPointCloud2::Ptr transformed_cloud2(new pcl::PCLPointCloud2());
106 pcl::toPCLPointCloud2(*cloud_in, *transformed_cloud1);
107 pcl::toPCLPointCloud2(*cloud_out, *transformed_cloud2);
108
109 // Trasladar Centroides al origen
110 // transformed_cloud1 = traslate_centroid(transformed_cloud1);
111 // transformed_cloud2 = traslate_centroid(transformed_cloud2);
112
113 if (arg2 == "-v" || arg2 == "--voxel"){
114   // Voxelizar el source y el corrupto
115   // std::cout << "Voxelizando con tama o " << size << std::endl;
116   transformed_cloud1 = voxel_cloud(transformed_cloud1, size_to_f);
117   transformed_cloud2 = voxel_cloud(transformed_cloud2, size_to_f);
118
119   gettimeofday(&time2,NULL);
120   std::cout << ((time2.tv_sec - time.tv_sec)*1000000L+time2.tv_usec) - time.
     tv_usec << " Microsegundos para la voxelizacion" << std::endl;
121   // imprimir diferencia de tiempos diciendo que fue de la voxelizacion de
     las nubes
122
123   gettimeofday(&time,NULL);
124 }
125 else{
126   // Nada por ahora jaja
127 }
128
129
130
131
132 // Comparacion de Nube de puntos
133 std::cout << "Porcentaje de Acierto: " << compare_clouds(transformed_cloud1,
      transformed_cloud2, size_to_f) << std::endl;
134
135 // std::cout << contador << std::endl;
136
137 gettimeofday(&time2,NULL);
```

```
138   std::cout << ((time2.tv_sec - time.tv_sec)*1000000L+time2.tv_usec) - time.
        tv_usec << " Microsegundos para la comparaci n de puntos" << std::endl;
139   // imprimir diferencia de tiempos de la comparaci n de puntos
140
141   return (0);
142 }
```

Código 5.5: Experimento 2