UNIVERSIDAD POLITÉCNICA DE VALENCIA

ESCUELA POLITÉCNICA SUPERIOR DE GANDIA

I.T. Telecomunicación (Sonido e Imagen)

UNIVERSIDAD POLITECNICA DE VALENCIA

ESCUELA POLITECNICA SUPERIOR DE GANDIA

“Desarrollo de un procedimiento para la creación de imágenes 3D a partir de imágenes 2D”

TRABAJO FINAL DE CARRERA

Autor/es:
Jorge Palos Cuesta

Director/es:
D. Jaime García Rupérez

GANDIA, 2012

ÍNDICE

1. INTRODUCCIÓN 1
1.1. Objetivo del proyecto 1
2. LA RECONSTRUCCIÓN 3D 4
2.1. Introducción 4
2.2. Definiciones 5
2.3. Sistemas de coordenadas 6
2.3.1. Sistema de coordenadas Local 6
2.3.2. Sistema de coordenadas View 6
2.4. Número de vistas 7
2.5. Reconstrucción 3D a partir de vistas múltiples 8
2.5.1. Métodos BRep 9
2.5.2. Métodos CSG 9
2.6. Reconstrucción 3D a partir de vista única 10
2.6.1. Método de Etiquetado 10
2.6.2. Método del Espacio Gradiente 11
2.6.3. Método de Programación Lineal 11
2.6.4. Método Perceptual 12
2.7. Adquisición de datos 3D 13
2.7.1. Adquisición desde sensores de datos 13
2.7.2. Adquisición desde imágenes 2D 13
3. MODELADO EN 3D 15
3.1. Introducción 15
3.2. Modelos 3D 15
3.3. Representación 16
3.5. Coordenadas homogéneas 17
3.7. Proceso de Modelado en 3D 18
3.7.1. Modelado Poligonal 18
3.7.2. Modelado Curvo 18
3.7.3. Esculpido Digital 18

4. DESCRIPCIÓN DEL ALGORITMO 20

4.1. Introducción 20

4.1.1. Fases del algoritmo 20

4.1.2. Detección de características 21

4.2. Fase 1. Detección de Bordes 22

4.2.1. Definición 22

4.2.2. Algoritmo de detección de bordes 23

4.3. Fase 2. Detección de líneas rectas 27

4.3.1. Definición 27

4.3.2. Algoritmo de detección de líneas rectas 28

4.4. Fase 3. Refinamiento 40

4.4.1. Definición 40

4.4.2. Algoritmo de refinamiento 40

4.5. Fase 4. Detección de esquinas 43

4.5.1. Definición 43

4.5.2. Algoritmo de detección de esquinas 43

4.6. Fase 5. Virtualización y Renderizado 51

4.6.1. Definición 51

4.6.2. Transformaciones en el Modelado en 3D 51

4.6.2.1. Escalado 51

4.6.2.2. Traslación 53

4.6.2.3. Rotación 54

4.6.2.4. Proyección Perspectiva 56

4.6.3. Algoritmo de virtualización y renderizado 59

4.7. Fase 6. Deslizadores 73

4.7.1. Definición 73

4.7.2. Algoritmo de los deslizadores 73

4.8. Fase 7. Generación del par estéreo de imágenes 74

4.8.1. Definición 74

4.8.2. Creación del par estéreo de imágenes 74

4.8.3. Algoritmo de generación del par estéreo de imágenes 77

5. FUNCIONAMIENTO DE LA INTERFAZ GRÁFICA 79

5.1. Introducción.......... 79
5.2. Software *GUIVirtual3D* (Manual de Usuario).......... 79
5.2.1. Zonas en la ventana principal 80
5.2.2. Mensajes emergentes 82
5.2.3. Botones de la aplicación 83
6. RESULTADOS DEL ALGORITMO.......... 87
6.1. Imágenes con procesado correcto.......... 87
6.2. Imágenes con error de procesado.......... 90
6.3. Imágenes con procesado incorrecto 93
7. CONCLUSIONES Y LINEAS DE TRABAJO FUTURO 96
7.1. Conclusiones 96
7.2. Líneas de trabajo futuro.......... 96
8. BIBLIOGRAFÍA 98
ANEXO I: ALGORITMO DE FUNCIONES PROPIAS.......... 100
ANEXO II: RESULTADOS DIFERENTES EN DIFERENTES VERSIONES DE MATLAB.......... 138

ÍNDICE DE IMÁGENES

Figura 1: Sistema de coordenadas locales. 6
Figura 2: Sistema de coordenadas View... 7
Figura 3: Tetera de Utah. 17
Figura 4: Imagen de entrada. 23
Figura 5: Gradiente bidimensional. 23
Figura 6: Gradiente horizontal.. 23
Figura 7: Gradiente Vertical. 23
Figura 8: Gradiente componente R de cubo. 24
Figura 9: Gradiente componente R de pirámide... 24
Figura 10: Gradiente componente G de cubo. 24
Figura 11: Gradiente componente G de pirámide. 24
Figura 12: Gradiente componente B de cubo. 25
Figura 13: Gradiente componente B de pirámide... 25
Figura 14: Gradiente componente S de cubo. 25
Figura 15: Gradiente componente S de pirámide. 25
Figura 16: Gradiente final del cubo... 25
Figura 17: Gradiente final de la pirámide... 25
Figura 18: Gradiente final umbralizado. 26
Figura 19: Gradiente final umbralizado. 26
Figura 20: Imagen Entrada Algoritmo.. 26
Figura 21: Resultado. Bordes del objeto. 26
Figura 22: Imagen Entrada Algoritmo.. 27
Figura 23: Resultado. Bordes del Objeto. 27
Figura 24: Imagen de entrada cubo. 28
Figura 25: Transformada de Hough de cubo. 28
Figura 26: Imagen de entrada pirámide. 29
Figura 27: Transformada de Hough de pirámide.. 29
Figura 28: Transformada de Hough con todas las líneas detectadas de cubo. 30
Figura 29: Transformada de Hough con todas las líneas detectadas de pirámide.......... 30
Figura 30: Espectro de la transformada de Hough. 31
Figura 31: Zoom espectro transformada de hough. 31
Figura 32: Proyección y envolvente eje Y (rho)... 33
Figura 33: Proyección y envolvente eje X (theta). 33
Figura 34: Picos encontrados en proyección rho y división en tantas partes como picos encontrados. 34
Figura 35: Picos en la proyección theta (abajo) a partir de las divisiones de proyección rho (arriba). 34
Figura 36: División de proyección theta y búsqueda de picos en proyección rho. 35
Figura 37: Transformada de Hough con todas las líneas detectadas de cubo. 37
Figura 38: Transformada de Hough con las líneas reales del objeto cubo. 37
Figura 39: Líneas detectadas sobre la frontera del objeto. 38

Figura 40: Transformada de Hough con todas las líneas detectadas pirámide............... 38
Figura 41: Transformada de Hough con las líneas reales del objeto pirámide............... 39
Figura 42: Líneas detectadas sobre la frontera del objeto pirámide. 39
Figura 43: Líneas antes del refinamiento. 42
Figura 44: Líneas después del refinamiento. 42
Figura 45: Líneas antes del refinamiento. 42
Figura 46: Líneas después del refinamiento. 42
Figura 47: Imagen cubo con todos los puntos de corte existentes................................. 45
Figura 48: Imagen pirámide con todos los puntos de corte existentes. 45
Figura 49: Imagen cubo con puntos de corte sobre la frontera. 46
Figura 50: Imagen pirámide con puntos de corte sobre la frontera. 47
Figura 51: Imagen cubo con puntos de corte de la frontera sin repetir. 48
Figura 52: Imagen pirámide con puntos de corte de la frontera sin repetir. 48
Figura 53: Imagen cubo con todos los vértices detectados. 49
Figura 54: Imagen pirámide con todos los vértices detectados. 50
Figura 55: Cambio de escala uniforme. 52
Figura 56: Cambio de escala no uniforme. 52
Figura 57: Traslación de un objeto en el espacio. 54
Figura 58: Rotación de un objeto alrededor de un eje paralelo a sistema de referencia global. 55
Figura 59: Objeto rotado alrededor de su eje X’. 56
Figura 60: Los objetos del mundo real parecen más pequeños a medida que están más lejos.. 56
Figura 61: Proyección Perspectiva Estándar. 57
Figura 62: Proyección perspectiva de un objeto. (Objeto con transformación de perspectiva).. 57
Figura 63: Transformación perspectiva de un objeto. 58
Figura 64: Objeto antes (izquierda) y después (derecha) de aplicar la transformación perspectiva. 59
Figura 65: Coordenadas 3D del cubo. 60
Figura 66: Objeto y distancias para calcular α y β. 61
Figura 67: Objeto y distancias para calcular α y β. 61
Figura 68: Cubo 3D y distancia para calcular β. 61
Figura 69: Cubo 3D con la rotación α, β y µ. 61
Figura 70: Coordenadas 3D de la pirámide. 66
Figura 71: Objeto y distancias para calcular α y β. 67
Figura 72: Objeto y distancias para calcular α y β. 67
Figura 73: Pirámide y distancia para calcular β. 67
Figura 74: Pirámide 3D con la rotación α, β y µ. 67
Figura 75: Renderizado 2D del cubo. 70
Figura 76: Virtualización 3D del cubo. 70
Figura 77: Renderizado 2D de la pirámide. 70
Figura 78: Virtualización 3D de la pirámide. 70
Figura 79: Imagen original. 71

Figura 80: Modelo 3D del objeto original. 71
Figura 81: Imagen original. 71
Figura 82: Modelo 3D del objeto original. 71
Figura 83: Disparidad Binocular. Muestra los ojos, objeto, distancias y ángulos..... 75
Figura 84: Trigonometría para calcular el ángulo 76
Figura 85: Relación distancia objeto-observador y ángulo entre ojos..... 76
Figura 86: Imagen del ojo Izquierdo de cubo. 78
Figura 87: Imagen del ojo Derecho de cubo. 78
Figura 88: Imagen del ojo Izquierdo de pirámide. 78
Figura 89: Imagen del ojo Derecho de Pirámide 78
Figura 90: Ventana principal de la aplicación. 80
Figura 91: Ventana gráfica de Entrada. 80
Figura 92: Ventana gráfica de Salida. 81
Figura 93: Deslizadores en reposo..... 81
Figura 94: Deslizadores desplazados..... 81
Figura 95: Cuadro de texto 'Ruta' vacío. 82
Figura 96: Cuadro de texto 'Ruta' indicando ruta de imagen cargada. 82
Figura 97: Mensaje emergente de Espera. 82
Figura 98: Mensaje emergente de finalizado. 82
Figura 99: Mensaje emergente de error. 83
Figura 100: Diálogo para abrir archivo 'imagen de entrada'. 83
Figura 101: Ventana principal del programa cuando se ha cargado la imagen. 84
Figura 102: Ventana principal del programa cuando se pulsa el botón 'Procesar imagen'. 85
Figura 103: Aspecto de la ventana principal cuando se ha procesado el objeto. 85
Figura 104: Ventana principal del programa al pulsar el botón 'Obtener Par Estéreo'. 86
Figura 105: Procesado de la imagen cubo_amarillo.jpg..... 87
Figura 106: Procesado de la imagen cubo_azulD.jpg 88
Figura 107: Procesado de la imagen cubo_verde2.jpg 88
Figura 108: Procesado de la imagen cubo_rojo.jpg 89
Figura 109: Procesado de la imagen piramide.jpg 89
Figura 110: Procesado de la imagen piramide4.jpg 90
Figura 111: Imagen cubo_azulG.jpg 91
Figura 112: Imagen con detección de bordes, líneas y vértices. 91
Figura 113: Imagen cubo_azulC.jpg 92
Figura 114: Imagen con detección de bordes, líneas y vértices. 92
Figura 115: Imagen cubo_verde.jpg..... 93
Figura 116: Imagen con detección de bordes, líneas y vértices. 93
Figura 117: Imagen cubo_azulE.jpg..... 93
Figura 118: Objeto virtual incorrecto. 93
Figura 119: Imagen con fase de detección y refinamiento. 94
Figura 120: Imagen cubo_azulA.jpg 94
Figura 121: Objeto virtual incorrecto. 94
Figura 122: Detección de vértices correcta. 95

1. INTRODUCCIÓN

La descripción de objetos tridimensionales en un plano, utilizando proyecciones bidimensionales de éstos fue inventada hace más de dos mil años por el matemático francés Gaspard Monge[1]. Monge fue el primero que sistematizó y simplificó los métodos existentes para dar lugar a la geometría descriptiva.

En geometría descriptiva y proyectiva hace tiempo que se estudian las proyecciones bidimensionales, donde las investigaciones se han centrado en cómo describir un objeto en el plano biodimensional.

A finales de los 60, y con el avance de los ordenadores digitales empezó a captar la atención de los matemáticos la dificultad de cómo reconstruir de forma automática la estructura, tanto geométrica como topológica, de un objeto tridimensional a partir de su proyección bidimensional. El problema oculto de pasar de dos dimensiones a tres dimensiones es tanto teórico como matemático.

Este problema se llama “reconstrucción”. La reconstrucción implica determinar la relación geométrica y topológica de las partes de un objeto. Este término no debe confundirse con el de reconocimiento, que implica la identificación de un objeto mediante algún sistema de acoplamiento de plantillas.

Los algoritmos desarrollados hasta el momento, se debaten entre el coste computacional y la calidad del mallado obtenido. La eficiencia del algoritmo es la que define la calidad final del mallado. Si suponemos un conjunto de puntos mal representado, existirán puntos definidos que no cumplan las condiciones óptimas para el mallado. Los puntos que se encuentran muy cercanos entre sí, los puntos ruidosos y los puntos redundantes, no ofrecen ninguna información para la reconstrucción. Por ejemplo, si queremos representar un cubo en el espacio, simplemente con siete u ocho puntos sería suficiente, el resto de la información sería redundante.

1.1. Objetivo del proyecto

El Plan de Estudios de la Titulación de Ingeniería Técnica de Telecomunicación especialidad Sonido e Imagen, de la Escuela Politécnica Superior de Gandía (EPSG) en la Universidad Politécnica de Valencia (UPV), incluye la asignatura de Sistemas Multimedia en la intensificación de Tecnología Audiovisual y la asignatura troncal de Tratamiento Digital de Imágenes. Estas asignaturas son impartidas por el profesor Jaime

[1] O'Connor, John J.; Robertson, Edmund F., «Biografía de Gaspard Monge», *MacTutor History of Mathematics archive*, Universidad de Saint Andrews.

García Rupérez y el profesor Ignacio Bosch Roig, respectivamente, pertenecientes al área de Teoría de la Señal y Comunicaciones del Departamento de Comunicaciones.

Con el objetivo de poner en práctica los contenidos de las dos asignaturas, se pensó en la posibilidad de desarrollar un procedimiento y un conjunto de herramientas en el entorno Matlab que sirvieran para crear imágenes tridimensionales (3D) a partir de imágenes en dos dimensiones (2D) con unas propiedades específicas de translación, escalado, rotación y perspectiva, y que a partir de esta conversión se pudiera obtener las imágenes que corresponden a los dos ojos para una posterior conversión estereoscópica.

El resultado es una interfaz gráfica como herramienta para el modelado tridimensional de imágenes 2D. La opción escogida fue conseguir un algoritmo que, a partir de una imagen 2D, tomada con una cámara fotográfica, llegue a crear un modelo tridimensional de un objeto.

Aunque obtener proyecciones 2D para un objeto 3D dado es sencillo, la operación inversa llega a ser algo complicada. Estas dificultades se originan por la pérdida de información cuando un objeto 3D es representado con proyecciones 2D, tal y como ocurre por ejemplo en una fotografía. Este proceso de obtener un objeto 3D a partir de una o varias proyecciones 2D es lo que se llama *Reconstrucción 3D*.

La reconstrucción tridimensional tiene varias aplicaciones, como la navegación de un robot permitiéndole conocer en qué parte de la escena se encuentra y poder planificar sus movimientos sin necesidad de ayuda humana. También es útil, en el tema que nos ocupa, para la creación de una imagen estereoscópica, ya que nos permitirá obtener las dos imágenes 2D visualizadas por cada ojo.

Para conseguir dicho objetivo se optó por una reconstrucción 3D dispersa basada en puntos de interés, al proporcionar una solución robusta y rápida. A continuación se analizaron los principales detectores de puntos de interés (esquinas, bordes y líneas), implantando algunos de ellos como el detector basado en el Gradiente y el basado en la transformada de Hough, para la detección de bordes y líneas rectas. Estas técnicas se compararon y probaron con distintas imágenes. Una vez estudiados e implementados los algoritmos de los detectores de puntos de interés, se llevó a cabo la programación de un algoritmo que permitiera crear un modelo 3D.

El paso siguiente fue dotar al objeto reconstruido de una serie de movimientos, en los que el usuario, a través de una serie de deslizadores puede aplicarle y ver su efecto en tiempo real.

Como se ha comentado en párrafos anteriores, éste sería un posible paso intermedio a la conversión estereoscópica, por lo que por último se ha añadido un algoritmo en el que, una vez reconstruido el modelo del objeto, se pueden obtener dos imágenes bidimensionales del modelo desde diferente perspectiva visual, imágenes

pertenecientes al ojo derecho y al ojo izquierdo, que serían utilizadas en una posterior creación de imagen estereoscópica.

En resumen, el algoritmo desarrollado constará de las siguientes fases:

- Detección de los bordes de los objeto.
- Identificación de las líneas correspondientes a estos bordes.
- Búsqueda de los puntos de corte de las líneas para identificar los vértices del objeto.
- Identificación de la forma básica representada por el objeto a partir de la distribución espacial de los vértices.
- Determinación de los parámetros constitutivos del objeto básico identificado de cara a generar su modelo 3D.
- Obtención de las imágenes bidimensionales correspondientes a cada ojo para la representación estereoscópica del objeto.

El trabajo se ha sido realizado para la identificación de formas básicas, inicialmente cubos y pirámides. Para comprobar si el algoritmo era eficaz se realizaron varias pruebas con varias imágenes tomadas por una cámara fotográfica.

El presente trabajo demuestra que es posible realizar un algoritmo que reconstruya un objeto en el espacio, dejando como trabajo futuro su optimización para todo tipo de objetos y para una conversión estereoscópica.

2. LA RECONSTRUCCIÓN 3D

2.1. Introducción

En la visión por ordenador y en gráficos por ordenador, la reconstrucción 3D es el proceso mediante el cual objetos reales son reproducidos en la memoria de una computadora capturando sus características físicas, como la forma, el volumen, las dimensiones y la apariencia. Existen multitud de técnicas de reconstrucción y métodos de mallado 3D, cuyo objetivo principal es obtener un algoritmo que sea capaz de realizar la conexión del conjunto de puntos representativos del objeto en forma de elementos de superficie, ya sean triángulos, cuadrados o cualquier otra forma geométrica.

El proceso de reconstrucción puede ser llevado a cabo mediante métodos activos o pasivos. En estos métodos es necesario un mecanismo de digitalización del objeto 3D real. Un punto interesante de estos métodos es la digitalización del objeto 3D a partir de una imagen 2D del mismo.

A continuación se distingue entre estos dos métodos de reconstrucción 3D:

- **Métodos Activos:** Estos métodos interfieren de forma activa con el objeto reconstruido, ya sea mecánica o radiométricamente, para obtener la información del objeto. Un ejemplo sencillo de un método mecánico sería un medidor de profundidad para medir la distancia a un objeto en rotación puesto en una plataforma giratoria. Otro ejemplo de métodos radiométricos aplicables sería un emisor de radiación hacia el objeto que mida su parte reflejada. Los ejemplos van desde las fuentes de luz en movimiento, la luz visible coloreada, tiempo de vuelo láser, hasta las microondas o los ultrasonidos.

- **Métodos Pasivos:** Los métodos pasivos de la reconstrucción 3D no interfieren con el objeto reconstruido, éstos sólo utilizan un sensor para medir la radiación reflejada o emitida por la superficie del objeto para inferir su estructura 3D. Normalmente, el sensor es un sensor de imagen en una cámara sensible a la luz visible y la entrada para el método es un conjunto de imágenes digitales (uno, dos o más) o de video. En este caso hablamos de la imagen basada en la reconstrucción y el resultado es un modelo 3D.

2.2. Definiciones

Puesto que los términos que se usan en la reconstrucción a menudo se solapan y en algunos casos tienen significados que difieren de su uso común, los definiremos brevemente aquí:

Objeto: Un objeto es un cuerpo sólido tridimensional. En muchos casos usamos el término “sólido” como sinónimo de objeto.

Frontera: La frontera de un objeto es el objeto menos su interior.

Mundo del objeto: Es el sistema de Coordenadas del Mundo (World Coodinates) en el que se define al objeto.

Modelo: Un modelo es un objeto de forma primitiva predefinida, como un bloque paralelepipédico o una esfera. Un modelo se puede definir en el mundo del objeto o en su sistema de coordenadas locales.

Cara: Una cara de un objeto es una superficie no vacía, delimitada, no necesariamente plana y cerrada, con su primer y último vértice perfectamente coincidentes, que pertenece a la frontera de ese objeto.

Polígono: Cuando una cara de un objeto es plana se le puede denominar polígono.

Arista: Una arista es el segmento lineal compartido por dos caras. Las aristas no tienen que ser necesariamente rectas; dependerá de si las caras compartidas son o no planas.

Vértice: Un vértice es un punto terminal de una arista.

Escena: Una escena es un conjunto finito de objetos.

Dibujo lineal: Un dibujo lineal es la proyección de una escena 3D.

Línea: Una línea es la proyección de una arista en un dibujo lineal.

Unión: Una unión es la proyección de un vértice en un dibujo lineal.

Sistema de coordenadas: Una escena puede identificarse con las coordenadas en tres dimensiones del espacio en las cuales tiene lugar la representación. Este espacio se llama sistema de coordenadas universal o World Coordinates.

2.3. Sistemas de coordenadas

Al operar con los objetos de la escena se puede utilizar diferentes sistemas de coordenadas. Este sistema se debe elegir para la posterior transformación del objeto, la cual se aplica respecto el sistema de coordenadas elegido.

Los sistemas utilizados son el sistema de coordenadas local y el sistema de coordenadas de la vista –View-. Por defecto se establece este último sistema.

2.3.1. Sistema de coordenadas Local

En el sistema de coordenadas local, cada objeto tiene su propio sistema de coordenadas, donde el origen del sistema viene dado por el pivote del objeto.

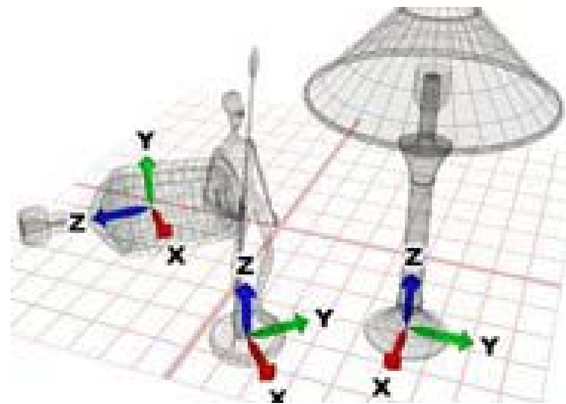

Figura 1: Sistema de coordenadas locales.

2.3.2. Sistema de coordenadas View

En este sistema de coordenadas los ejes de coordenadas se intercambian en función del visor que empleemos. Entonces se establece que:

- El eje X siempre apunta hacia la derecha de la vista
- El eje Y siempre apunta hacia arriba de la vista
- El eje Z siempre apunta hacia el usuario

En la *Figura 2* se puede ver cómo cambia el sistema de coordenadas en función de la vista empleada.

Figura 2:

1) Sistema de coordenadas vista Arriba
2) Sistema de coordenadas vista Frontal
3) Sistema de coordenadas vista Izquierda
4) Sistema de coordenadas vista Perspectiva

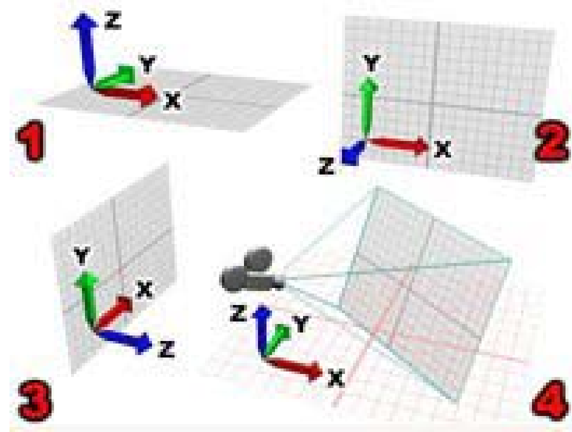

Figura 2: Sistema de coordenadas View.

2.4. Número de vistas

Por el número de vistas utilizadas en la entrada de datos, existen dos categorías: métodos de vistas múltiples y métodos de vista única.

Los métodos basados en vistas múltiples están más adelantados. Obviamente es mucho más fácil reconstruir objetos 3D a partir de proyecciones de vistas múltiples que a partir de proyecciones de vista única. No obstante dichos métodos suelen estar limitados a considerar las vistas en alzado, planta y perfil.

La reconstrucción a partir de una única vista presenta más ambigüedades y da más soluciones.

Premisas

Para simplificar el problema de la reconstrucción, generalmente se asume que en cualquier proyección de una escena sólo se representan las aristas y los contornos. Es decir, se trabaja con vistas "normalizadas" o "puras" (en el sentido de que los objetos se representan por medio de aristas y contornos). No se consideran la textura, la gama, el sombreado y otros parámetros adicionales que se están usando en el reconocimiento de objetos. Algunos sistemas distinguen entre aristas vistas y ocultas; otros no distinguen, y los hay que exigen que no se empleen dichas aristas ocultas.

En este trabajo se asume que en cualquier proyección de una escena se representan las aristas y los contornos "normalizados", y además la única característica física con la que se trabaja es la del color -caracterizado en componentes Rojo-Verde-

Azul (RGB)-. Además, se distingue entre aristas visibles y ocultas para la representación del modelo 3D.

2.5. Reconstrucción 3D a partir de vistas múltiples

La reconstrucción 3D a partir de varias imágenes es la creación de modelos tridimensionales a partir de un conjunto de imágenes. Es el proceso inverso de la obtención de imágenes en 2D a partir de escenas en 3D.

La esencia de una imagen es una proyección de una escena 3D en un plano 2D, proceso durante el cual se pierde la profundidad. El punto 3D correspondiente a un punto específico de la imagen se ve obligado a estar en la línea de visión. De una sola imagen, es imposible determinar exactamente qué punto en esta línea de visión corresponde al punto de la imagen. Si se dispone de dos imágenes, entonces la posición de un punto 3D se puede encontrar como la intersección de los dos rayos de la proyección. Este proceso se conoce como triangulación. La clave para este proceso es la relación entre varias vistas que transmiten la información de la estructura de los correspondientes conjuntos de puntos y esta estructura está relacionada con la posición y la calibración de la cámara.

En las últimas décadas, hay una demanda importante para el contenido 3D de los gráficos por ordenador, la realidad virtual y la comunicación, lo que provocó un cambio en el énfasis de los requisitos. Muchos de los sistemas existentes para la construcción de modelos 3D están construidos alrededor de un hardware especializado (por ejemplo, equipos de estéreo), resultando en un alto costo, que no puede satisfacer la exigencia de sus nuevas aplicaciones. Esta brecha estimula el uso de las instalaciones digitales de imagen (como una cámara). La ley de Moore nos dice también que cada vez se puede realizar más trabajo mediante software. Un método temprano fue propuesto por Tomasi y Kanade.

Los modelos obtenidos tras el proceso de reconstrucción son generalmente de tipo "Geometría Constructiva de Sólidos" (CSG) o "Representación de Fronteras" (BRep "*Boundary Representation*"). Actualmente, la mayor parte de los trabajos de reconstrucción totalmente automática usan BRep, mientras que las representaciones CSG son más habituales en los procesos de reconstrucción guiada.

2.5.1. Métodos BRep

Este método se basa en la descripción algebraica de los sólidos, asumiendo que están delimitados por un conjunto de caras, que pertenecen a superficies orientables y cerradas. La orientación implica que es posible distinguir la cara exterior de la interior al solido.

El método surge a partir de los modelos poliédricos usados para la representación de objetos con eliminación de líneas y caras ocultas. En él los sólidos se describen dando la geometría de su superficie (frontera), normalmente formada por un conjunto de caras planas, y las relaciones topológicas existentes. Para ello se distingue entre entidades geométricas (puntos, curvas, superficies) y topológicas (vértices, aristas y caras). La superficie de una cara es el plano que sustenta la cara.

La secuencia de pasos que siguen estos métodos se resumen a continuación:

1. Transformación de vértices 2D a vértices 3D.
2. Generación de aristas a partir de vértices 3D y de segmentos lineales de partida.
3. Construcción de caras a partir de aristas.
4. Formación del modelo 3D, a partir de las caras.

Las principales diferencias entre los distintos métodos BRep residen en las técnicas que se usan para resolver estos cuatro pasos.

Aunque este método se refiere a la reconstrucción a partir de vista múltiple, en este trabajo se ha intentado modelar el objeto 3D de un modo similar al que se explica en este método con la particularidad de que se utiliza la reconstrucción a partir de vista única.

2.5.2. Métodos CSG

La geometría constructiva de sólidos (CSG) no representa explícitamente la geométrica del sólido, si no que lo describe como una expresión booleana, en la que intervienen una combinación de primitivas (sólidos básicos predefinidos) siguiendo una jerarquía determinada.

Mas formalmente, un modelo CSG se puede expresar como

$$Sólido = T_{g1}(Sólido_1)\ OpB\ T_{g2}(Sólido_2)\ |\ T_{g3}\ (primitiva)$$

Donde las Tg son transformaciones geométricas y OpB es una operación booleana (∪, ∩ o -).

Obviamente la representación es muy concisa, pero a costa de no almacenar explícitamente la geometría, lo que hace que la visualización no sea trivial.

La geometría constructiva de sólidos es un método de modelado no ambiguo, con un dominio amplio (condicionado por el repertorio de primitivas disponibles) y válido. La principal debilidad del método es la no unicidad de las representaciones, ya que hay infinidad de formas de representar cualquier solido.

2.6. Reconstrucción 3D a partir de vista única

A continuación se describen algunos de los métodos típicos que se usan en el tema de la reconstrucción a partir de una vista única. Estos métodos se clasifican según sus modelos.

Dos de los métodos, el método del etiquetado y el del espacio *gradiente*, son métodos de interpretación más que de reconstrucción. Sirven para la interpretación de dibujos lineales 2D y generan información 3D que se puede usar posteriormente en el trabajo de reconstrucción. Pero por si mismos no hacen el trabajo de reconstrucción: no se obtiene ninguna representación de objetos 3D con estos métodos.

2.6.1. Método de Etiquetado

El primer método de etiquetado válido para poliedros lo establecieron Huffman y Clowes. Los métodos de etiquetado posteriores trataron de ampliarlo a casos más generales, incluyendo dibujos con aristas ocultas y dibujos con objetos curvos. Todos se basaban en reglas heurísticas, hasta que Malik publicó su trabajo. El método de Malik trata los dibujos lineales de escenas formadas por objetos sólidos regulares opacos delimitados por piezas con superficies suaves, con lo que solucionó el problema de etiquetado general.

Los métodos de etiquetado son métodos de interpretación más que de reconstrucción: ofrecen sólo las condiciones necesarias para que un dibujo lineal 2D represente un sólido 3D válido. Además, un dibujo lineal que se puede etiquetar adecuadamente no necesariamente representa un sólido 3D verdadero.

2.6.2. Método del Espacio Gradiente

El método del gradiente parte del método de Mackworth, que interpreta los dibujos lineales construyendo la imagen de cada plano en el espacio de gradientes. Su programa determina primero el tipo de aristas: las aristas enlazadas y las ocultas. Después resuelve si las aristas enlazadas son cóncavas o convexas mediante la construcción de la imagen de gradientes. Mackworth también demuestra que un dibujo lineal etiquetado puede considerarse incorrecto si no admite una figura recíproca. Por ello, su método del espacio de gradientes se puede usar para detectar dibujos lineales no realizables. No obstante, algunos dibujos lineales no son realizables incluso aunque se puedan construir sus imágenes de gradientes.

Wei amplió la idea de Mackworth, dando las cuatro condiciones necesarias y suficientes para que exista un poliedro en R^3. La diferencia estriba en el hecho de que el gradiente de Wei es un vector tridimensional, mientras que el de Mackworth es sólo bidimensional. Basándose en estas cuatro restricciones, dio las reglas para hallar la solución recursivamente. Wei también estudió posteriormente como ampliar su algoritmo a dibujos no perfectos. Sin embargo, la limitación del método de Wei es que sólo puede manejar poliedros con cuatro grados de libertad. Los poliedros con más grados de libertad son descompuestos en varios poliedros, cada uno de los cuales no puede tener más de cuatro grados de libertad.

2.6.3. Método de Programación Lineal

El método de programación lineal se basa en plantear y resolver las ecuaciones lineales que describen las condiciones que un poliedro debe satisfacer. Sugihara consiguió dar una condición necesaria y suficiente que permitía que un dibujo lineal representase un objeto poliédrico en términos del problema de programación lineal. Su formulación permite resolver el problema de discriminar entre dibujos lineales correctos e incorrectos. Sin embargo, el método de Sugihara no es práctico para realizar la reconstrucción. La condición es tan precisa matemáticamente que algunos dibujos son rechazados simplemente porque los vértices se desvían ligeramente de las posiciones correctas. La dificultad estriba en el hecho de que hay ecuaciones redundantes en el sistema. Sugihara soluciona esta dificultad identificando las ecuaciones redundantes y permitiendo extraer un subconjunto. El método corregido da una solución completa al problema de la reconstrucción de un objeto poliédrico 3D, obteniendo su representación BRep, a partir de una proyección 2D de vista única. Su limitación más evidente es que sólo trata objetos poliédricos. La implementación del método requiere la solución de un gran problema de programación lineal de 3m+n variables, donde m es el número de caras visibles del poliedro y n el número de vértices visibles.

2.6.4. Método Perceptual

Lamb y Bandopadhy presentaron un sistema para extraer la estructura 3D, a partir de un dibujo lineal impreciso. La idea es dar a los diseñadores que utilizan CAD la capacidad de visualizar interactivamente la interpretación más probable de los dibujos disponibles en cada momento, los cuales pueden ser imprecisos, para permitir los cambios pertinentes cuando la interpretación de los mismos sea incorrecta.

Este "*método perceptual*" difiere de los otros en que no usa métodos numéricos. El algoritmo toma como dato un dibujo axonométrico. Y utiliza los invariantes que aseguran que las líneas paralelas aparecen paralelas, y que las aristas paralelas a los ejes principales se dibujan con longitudes proporcionales a las dimensiones reales. El algoritmo preprocesa el dibujo generando un gráfico de adyacencia (un mapa de vértices y aristas) y después etiqueta el dibujo usando el método de Waltz. Se le asignan al dibujo los ejes principales, los cuales, a su vez, forman los planos principales. También se selecciona el origen.

El objetivo del algoritmo es asignar coordenadas 3D a todos los vértices del gráfico de adyacencia. Esto se hace mediante los siguientes pasos:

1. Seleccionar la "mejor" región para la asignación de las coordenadas. La "mejor" región se determina mediante la evaluación de la información existente.

2. Seleccionar un punto de referencia para la región. Todas las coordenadas de esta región se asignarán en relación a este punto de referencia. Si ésta es la primera región seleccionada, tiene sentido seleccionar la del origen. Si no, se espera que la "mejor" región contenga algún(os) vértice(s) ya asignados.

3. Asignar las coordenadas a todos los vértices de la región. Si la región es paralela a un plano principal, entonces es fácil asignar las coordenadas ya que la longitud de una arista refleja la distancia real. Si la región es paralela a un plano identificado, la información sobre el plano conocido y la longitud se pueden usar para determinar las coordenadas. Si la región no es paralela a ningún plano conocido, pero se pude aplicar una regla de simetría, entonces se usa la información dada por la simetría, así como otras reglas heurísticas (un paralelogramo visto como un rectángulo inclinado, por ejemplo) para determinar las coordenadas. Si no se cumple nada de lo anterior, el algoritmo necesita que el usuario especifique las coordenadas.

4. Repetir los 3 pasos anteriores hasta que se les hayan asignados las coordenadas a todos los vértices visibles.

Dependiendo de la precisión del dibujo y el orden en el que se procesen las regiones, es posible que a dos vértices se les asignen coordenadas que sean incompatibles con la ecuación del plano. Cuando esto ocurre, el algoritmo retrocede y modifica uno de los valores de las coordenadas.

2.7. Adquisición de datos 3D

La adquisición de datos se realiza a través de sensores de datos. Las técnicas y teorías, en general, trabajan con la mayoría o la totalidad de los tipos de sensores, incluyendo sensores ópticos, acústicos, escaneado láser, radares, térmicos y sísmicos.

La adquisición de los datos del objeto real puede ocurrir por una multitud de métodos, entre los más utilizados están los sensores de datos y las imágenes en 2D.

2.7.1. Adquisición desde sensores de datos

Una forma de adquirir datos semiautomáticamente para la reconstrucción 3D es a partir de datos LIDAR (Luz de Detección y Rango) e imágenes de alta resolución o a través de sistemas de sensores láser. Este sistema permite modelar el objeto sin moverlo físicamente o mover su emplazamiento. Con los datos LIDAR, se pueden generar Modelos Digitales de Superficie (DSM). Basándose en el conocimiento general acerca de los objetos, se extrae la información sobre las características geométricas tales como el tamaño, la altura y la forma.

2.7.2. Adquisición desde imágenes 2D

La adquisición de datos en 3D y la reconstrucción del objeto se puede realizar a partir de imágenes 2D del objeto, ya sea utilizando una imagen única o bien múltiples imágenes. El método más común es la reconstrucción a partir de pares estéreo del objeto.

La fotogrametría estéreo o la fotogrametría basada en un bloque de imágenes superpuestas es el enfoque principal del modelado en 3D y la reconstrucción de objetos utilizando imágenes 2D. La fotogrametría de rango cercano también ha madurado hasta el nivel en el cual las cámaras o cámaras digitales se utilizan para capturar la apariencia de objetos en distancias cercanas, por ejemplo, edificios, y reconstruirlos usando la misma teoría que la fotogrametría aérea.

Un método semiautomático para la adquisición de datos 3D topológicamente estructurados de imágenes 2D estéreo aéreas fue presentado por Sisi Zlatanova[2]. El proceso consiste en la digitalización manual de un número de puntos necesarios para la reconstrucción automática de los objetos 3D. Cada objeto reconstruido es validado por la superposición de su estructura en el modelo estéreo. Los datos 3D topológicamente estructurados se almacenan en una base de datos para su posterior uso en la visualización de los objetos.

Franz Rottensteiner desarrolló un método semiautomático para la reconstrucción de objetos, además de un concepto para almacenar modelos de objetos junto con sus datos en un sistema de información. Su enfoque se basa en la integración de parámetros estimados del modelo en el proceso de fotogrametría aplicando un esquema de modelado híbrido. Los modelos se descomponen en un conjunto de primitivas simples que se reconstruyen de forma individual y luego se combinan mediante operadores lógicos. La estructura interna de datos tanto de esas primitivas simples como de los modelos de construcción compuestos se basa en métodos de representación de contorno.

Para la reconstrucción a partir de múltiples imágenes se utiliza la propuesta de Zeng. La idea principal de este enfoque es explorar la integración de los datos 3D y de las imágenes 2D. Esta propuesta está motivada por el hecho de que sólo se reconstruyen en el espacio los puntos sólidos y precisos que sobreviven un escrutinio geométrico. Este escrutinio da lugar a una insuficiencia de datos estéreo que luego debe ser rellenado utilizando información de múltiples imágenes.

La idea es, pues, construir primero pequeñas zonas superficiales de los puntos estéreo, y luego ir propagando progresivamente sólo las zonas fiables vecinas a partir de imágenes de toda la superficie.

Existen nuevas técnicas de medición que también se emplean para obtener medidas de y entre los objetos a partir de imágenes individuales, mediante el uso de proyecciones o sombras, así como una combinación de ambas. Esta tecnología está ganando fuerza debido a su rápido tiempo de procesamiento, y un costo mucho más bajo que las mediciones estéreo.

[2] S. Zlatanova, J. Paintsil, K. Tempfli. "3D Object reconstruction from aerial stereo images" International Institute for Aerospace Survey and Earth Sciences.

3. MODELADO EN 3D

3.1. Introducción

El modelado en 3D es el proceso de elaboración de una representación matemática de cualquier superficie tridimensional de un objeto (ya sea inanimado o vivo) a través de software. El producto final es lo que se llama un modelo 3D y se puede visualizar como una imagen de dos dimensiones a través de un proceso llamado renderizado 3D (representación 3D) o utilizarlo en un equipo de simulación de fenómenos físicos. El modelo también puede ser físicamente creado mediante dispositivos de impresión tridimensional. Los modelos pueden ser creados manualmente o de forma automática.

3.2. Modelos 3D

Los modelos 3D representan un objeto 3D utilizando una colección de puntos en el espacio tridimensional, conectados de diversas formas geométricas como triángulos, líneas, superficies curvas, etc. Siendo una colección de datos (puntos y otra información), los modelos en 3D pueden ser creados a mano, algorítmicamente (modelado de procedimientos), o mediante escaneado.

Los modelos 3D son ampliamente utilizados en los gráficos en 3D. En realidad, su uso es anterior al uso generalizado de los gráficos 3D en los ordenadores personales. Muchos juegos de ordenador utilizados pre-renderizan imágenes de modelos en 3D como sprites[3] antes que las computadoras puedan representarlos en tiempo real.

Hoy en día, los modelos 3D se utilizan en una amplia variedad de campos. La industria médica utiliza modelos detallados de los órganos. La industria del cine los usa como personajes y objetos para la animación e imágenes en movimiento de la vida real. La industria de los videojuegos los usa como activos para los juegos de ordenador y videojuegos. El sector de la ciencia los usa como modelos muy detallados de los compuestos químicos. La industria de la arquitectura utiliza modelos 3D para demostrar los edificios propuestos y los paisajes. La comunidad de la ingeniería los utiliza como los diseños de los nuevos dispositivos, vehículos y estructuras, así como una serie de

[3]Sprites: (en inglés 'duendecillos') Se trata de un tipo de mapa de bits dibujados en la pantalla de ordenador por hardware gráfico especializado. A menudo son pequeños y parcialmente transparentes, dejándoles así asumir otras formas a la del rectángulo, por ejemplo puntero de ratón en ordenadores de la serie Amiga.

otros usos. En las últimas décadas la comunidad de la ciencia de la tierra ha empezado a construir modelos geológicos en 3D.

3.3. Representación

Casi todos los modelos 3D se pueden dividir en dos categorías, sólidos o fronteras. A continuación se explican las diferencias principales entre ellas:

- **Sólido**: Estos modelos definen el volumen del objeto que representan (como una roca). Estos son más realistas, pero más difíciles de construir. Los modelos sólidos se utilizan sobre todo para las simulaciones no visuales, tales como simulaciones de medicina e ingeniería, para CAD y aplicaciones especializadas visuales, tales como el trazado de rayos y de la geometría sólida constructiva

- **Frontera o Concha**: Estos modelos representan la superficie, por ejemplo, el límite del objeto, no su volumen (como una cáscara de huevo infinitamente delgada). Estos son más fáciles de trabajar que con los modelos sólidos. Casi todos los modelos visuales que se utilizan en los juegos y las películas son los modelos de concha.

Debido a que la apariencia de un objeto depende en gran medida del exterior del objeto, las representaciones de contorno son comunes en los gráficos por ordenador. Las superficies bidimensionales son una buena analogía para los objetos utilizados en los gráficos. Dado que las superficies no son finitas, una aproximación digital discreta es necesaria: mallas poligonales (y, en menor medida subdivisión de superficies) son, con diferencia, la representación más común, aunque las representaciones basadas en puntos ha ido ganando cierta popularidad en los últimos años. Los conjuntos de nivel son una representación útil para deformar las superficies que sean objeto de muchos cambios topológicos, tales como los líquidos.

El proceso de transformación de las representaciones de objetos, tales como la coordenada del punto medio de una esfera y un punto de su circunferencia en una representación poligonal de una esfera, se llama teselación. Este paso se utiliza en la representación poligonal, donde los objetos se descomponen a partir de representaciones abstractas ("primitivas"), tales como esferas, conos, etc, erróneamente llamadas *mallas*, que son redes de triángulos interconectados. Las mallas de triángulos (en lugar de, por ejemplo cuadrados) son populares, ya que han demostrado ser fáciles de representar haciendo uso de la representación lineal. Las representaciones poligonales no se utilizan en todas las técnicas de representación, y en estos casos la teselación no está incluida en la transición de la representación abstracta a la escena renderizada.

La tetera de Utah es uno de los modelos más comunes utilizados en la enseñanza de gráficos 3D. Fue desarrollada por Martin Newell en 1975. Una representación moderna de la icónica tetera se puede ver en la Figura 3 como ejemplo de Modelo 3D renderizado.

Figura 3: Tetera de Utah.

3.5. Coordenadas homogéneas[4]

En geometría proyectiva, los puntos se representan mediante coordenadas homogéneas.

Será útil sustituir las coordenadas (x, y, z) por las coordenadas (x_h, y_h, z_h, h), llamadas coordenadas homogéneas, donde

$$x = x_h/h, \qquad y = y_h/h, \qquad z=z_h/h$$

de donde

$$(x_h, y_h, z_h, h) = (h\cdot x, h\cdot y, h\cdot z, h)$$

Existe un número finito de representaciones homogéneas equivalentes para cada punto de coordenadas (x, y, z) seleccionando un valor no cero para h.

Por conveniencia, se escoge h = 1, con lo que cada posición tridimensional se representará con las coordenadas homogéneas (x, y, z, 1).

Expresar posiciones en coordenadas homogéneas nos permite representar todas las ecuaciones de transformación geométrica como multiplicaciones de matriz. Se

[4] En matemáticas, se utiliza el término coordenadas homogéneas para referirse al efecto de esta representación de ecuaciones cartesianas. Cuando se convierte un punto cartesiano (x, y) a una representación homogénea (x_h, y_h, h) las ecuaciones que contienen x, y se convierten en ecuaciones homogéneas de tres parámetros x_h, y_h y h. Esto significa que si se sustituye cada uno de los 3 parámetros con cualquier valor $\boldsymbol{v}$ veces ese parámetro, el valor $\boldsymbol{v}$ se puede factorizar fuera de las ecuaciones.

representan las coordenadas con vectores de columna de 4 elementos y las operaciones de transformación se expresan como matrices de 4 por 4.

3.7. Proceso de Modelado en 3D

La etapa de modelado consiste en dar forma a objetos individuales que luego serán utilizados en la escena.

De todas las maneras de crear un modelo 3D, tres de ellas son las más utilizadas, en los siguientes párrafos se realiza una breve descripción de los tres métodos que se utilizan en el proceso:

3.7.1. Modelado Poligonal

Los puntos en el espacio 3D, llamados vértices, están conectados por segmentos de línea para formar una malla poligonal. La gran mayoría de los modelos 3D de hoy se construyen como modelos poligonales con textura, ya que son flexibles y porque las computadoras pueden hacerlo rápidamente. Sin embargo, los polígonos son planos y solo pueden aproximar superficies curvas utilizando muchos polígonos.

3.7.2. Modelado Curvo

Las superficies están definidas por curvas, que son influenciadas por puntos de control ponderados. La curva sigue (pero no necesariamente interpolada) los puntos. Un aumento del tamaño de un punto desplazará la curva más cerca de ese punto. Los tipos de curvas que se incluyen son curvas B-spline (NURBS), Splines, parches y primitivas geométricas.

3.7.3. Esculpido Digital

Todavía es un método de modelado relativamente nuevo, el esculpido en 3D se ha vuelto muy popular en los pocos años de vida que tiene. En este momento hay 3 tipos de escultura digital: *desplazamiento,* que es el más utilizado entre las aplicaciones en este momento, *volumétrica* y la *teselación dinámica*.

El *desplazamiento* utiliza un modelo de densidad (a menudo generada por subdivisión de superficies de una malla de control de polígonos) y almacena las nuevas ubicaciones para las posiciones de los vértices mediante el uso de un mapa de imagen

de 32 bits que almacena los lugares acordados. El tipo *volumétrico* se basa en Vóxels[5] que tienen capacidades similares como el desplazamiento, pero no se deterioran cuando no hay suficientes polígonos en una región para lograr una deformación. La *Teselación dinámica* es similar al volumétrico, pero divide la superficie utilizando triangulación para mantener una superficie lisa y permitir detalles más finos.

Estos métodos permiten una exploración muy artística, ya que el modelo tendrá una nueva topología creada sobre sí mismo una vez hayan sido esculpidos la forma de los modelos y, posiblemente, los detalles. La nueva malla generalmente tiene la malla original de alta resolución, la información se transfiere a datos de desplazamiento o a un mapa normal de datos.

[5] Vóxel: (Volumetric pixel) es la unidad cúbica que compone un objeto tridimensional. Constituye la unidad mínima procesable de una matriz tridimensional y es, por tanto, el equivalente del pixel en un objeto 3D.

4. DESCRIPCIÓN DEL ALGORITMO

4.1. Introducción

Un algoritmo es un conjunto de instrucciones bien definidas, ordenadas y finitas que permite hallar la solución de un problema. También se puede definir como una lista de instrucciones que especifican una secuencia de operaciones deterministas que llevadas a cabo por un agente ejecutor (modelo computacional) da una solución a cualquier instancia de un problema determinado en un tiempo finito.

Seguidamente vamos a ver las fases del algoritmo creado para resolver el problema de la reconstrucción 3D de objetos a partir de una imagen 2D. En posteriores capítulos se desarrollarán las diferentes fases del algoritmo.

4.1.1. Fases del algoritmo

El proyecto a realizar consta de cinco fases relacionadas entre sí, de manera que los datos de salida de una fase corresponden a los datos de entrada de la siguiente. No siempre debe seguirse este orden en las fases ni siempre deben ser estas fases, a veces puede variar según los métodos utilizados. En este caso las fases son:

1. Detección de bordes.
2. Detección de líneas rectas (aristas).
3. Refinamiento.
4. Detección de vértices.
5. Virtualización 3D y Renderizado.

La primera fase tiene como datos de entrada la imagen 2D del objeto, obtenida con una cámara fotográfica. De esta fase se obtendrán los bordes del objeto, datos necesarios para poder definir los vértices y las aristas en las posteriores fases del algoritmo.

Una vez obtenidos estos datos se utilizará un algoritmo de búsqueda de líneas rectas para poder detectar todas las líneas del objeto a reconstruir, que estará basado en la Transformada de Hough.

En tercer lugar se aplicará un algoritmo de refinamiento o de ajuste donde se comprobará que las líneas obtenidas en las fases de detección de líneas son las

correctas, en caso de que alguna de las rectas no sea la correcta, el algoritmo corrige las líneas inexactas.

Posteriormente se aplicará el algoritmo de búsqueda de vértices o esquinas, cuyos datos de entrada son los datos de salida de la fase anterior y que encontrará todos los puntos de corte entre las líneas obtenidas en la fase anterior y después descartará los puntos de corte que no pertenecen a los vértices del objeto.

La fase quinta obtiene como datos entrada la lista de vértices y lineas derivadas de las fases anteriores con el fin de renderizar el objeto virtualmente. Para ello en esta fase se llevan a cabo unos cálculos para conseguir las características físicas del objeto (posición en la escena, tamaño, color y líneas visibles y ocultas) y reconstruirlo con la mayor fidelidad posible.

Hay una siguiente fase en la que el usuario, mediante el uso de deslizadores, puede rotar el objeto en el espacio. Esta fase se basa en repetir la fase quinta de virtualización y renderizado con los nuevos valores de los ángulos de rotación tantas veces como se desplacen los deslizadores.

Existe además una última fase en el algoritmo que no depende de las demás, si bien sus datos de entrada son los datos de salida de la fase quinta de virtualización 3D y renderizado, es una fase del algoritmo independiente y no de obligatoria ejecución. Es la fase de *Generación del par estéreo de imágenes*.

En esta fase se genera el par estéreo de imágenes del objeto renderizado. Para generarlas se utiliza un algoritmo de cálculo de la rotación que se ha de aplicar al objeto para simular la visión humana, rotación que es equivalente al ángulo con que los ojos del ser humano ven una escena.

4.1.2. Detección de características

La mayor parte del algoritmo de reconstrucción 3D está basado en la detección de características, por lo que es de significativa importancia en la reconstrucción 3D.

En procesado de imágenes, la detección de características se refiere a la obtención de información de la imagen. Las características resultantes serán subconjuntos del dominio de la imagen, a menudo bajo la forma de puntos aislados, curvas continuas o regiones conectadas.

Aunque no existe una definición exacta de qué es una característica, se puede definir, como una parte de interés de una imagen. Se suelen usar como punto de partida para muchos algoritmos de visión por ordenador.

Debido a que el algoritmo se basa en dichas características, el algoritmo será tan bueno como sea su detector. Otra cosa a tener en cuenta, es que un buen detector de

característica debe detectar la misma característica en dos o más imágenes diferentes, es decir, una de las propiedades que debe tener un detector de características es la repetición.

Otras propiedades que debe tener un detector de característica es la exactitud (ya que debe detectar la característica en el píxel correcto) y la estabilidad (debe detectar la característica después de que la imagen haya sufrido algún tipo de transformación geométrica como rotación o cambio de escala).

Existen varios detectores de características ya desarrollados, que varían en el tipo de característica a detectar, la complejidad computacional y la repetitividad. Los detectores más importantes son los detectores de esquinas, de bordes y de líneas rectas.

4.2. Fase 1. Detección de Bordes

4.2.1. Definición

La detección de bordes es un proceso en el análisis digital de imágenes que detecta los cambios en la intensidad de luz.

Los bordes de una imagen digital se pueden definir como transiciones entre dos regiones de niveles de gris significativamente distintos. Suministran una valiosa información sobre las fronteras de los objetos y puede ser utilizada para segmentar la imagen, reconocer objetos, etc.

La detección de bordes es clave en la reconstrucción tridimensional porque se puede extraer información importante de la imagen, como pueden ser las formas de los objetos que la componen. Los bordes indican dónde están los objetos, su forma, su tamaño, y también ofrecen información sobre su textura.

La mayoría de las técnicas para detectar bordes emplean operadores locales basados en distintas aproximaciones discretas de la primera derivada (Gradiente) de los niveles de grises de la imagen.

En este algoritmo se utiliza el gradiente numérico, el algoritmo calcula el gradiente de una imagen (matriz) en sus dos dimensiones obteniendo los bordes por separado de cada dimensión para luego combinarlas y conseguir los bordes de la imagen completa.

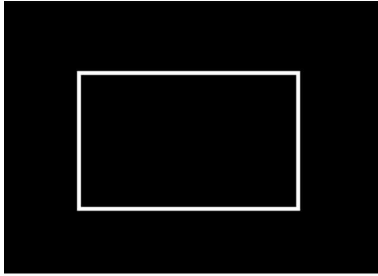

Figura 4: Imagen de entrada.

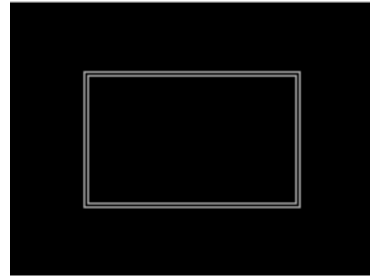

Figura 5: Gradiente bidimensional.

Figura 6: Gradiente horizontal.

Figura 7: Gradiente Vertical.

En el siguiente apartado se desarrolla el algoritmo utilizado en esta fase de detección de bordes así como una representación de los resultados que se obtendrían si se ejecutase el algoritmo.

4.2.2. Algoritmo de detección de bordes

Lo primero que se hace en esta fase una vez están definidos los datos de entrada es separar las componentes RGB (Red-Blue-Green) y HSI (Hue-Saturation-Intensity) de la imagen para su posterior utilización.

```
R = im_rgb(:,:,1); % im_rgb es la imagen de entrada precargada.
G = im_rgb(:,:,2);
B = im_rgb(:,:,3);
im_hsv = rgb2hsv(im_rgb);
H = im_hsv(:,:,1);
S = im_hsv(:,:,2);
I = im_hsv(:,:,3);
```

En lugar de mirar el nivel de HSI o RGB para umbralizar[6] la imagen, se hace una búsqueda de bordes sobre las componentes R, G, B y S, que es donde hay mayor diferencia entre el objeto y el fondo. Se calcula el gradiente para obtener el borde del objeto:

```
[gradient_Rx gradient_Ry]=gradient(double(R)/255);
```

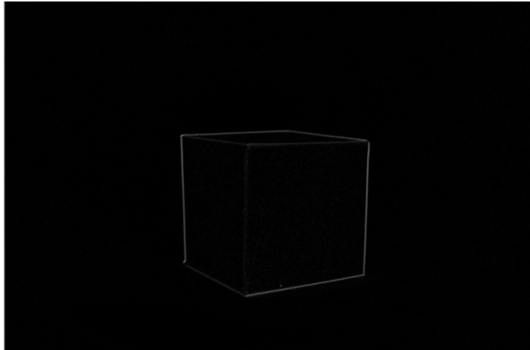

Figura 8: Gradiente componente R de cubo[7].

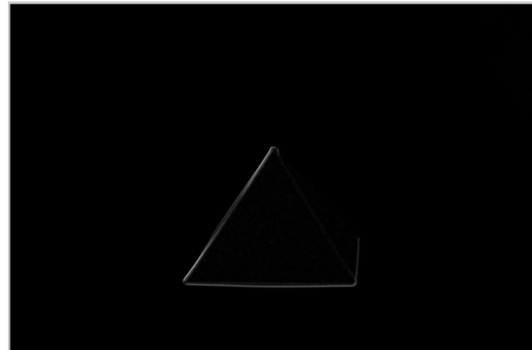

Figura 9: Gradiente componente R de pirámide[8].

```
[gradient_Gx gradient_Gy]=gradient(double(G)/255);
```

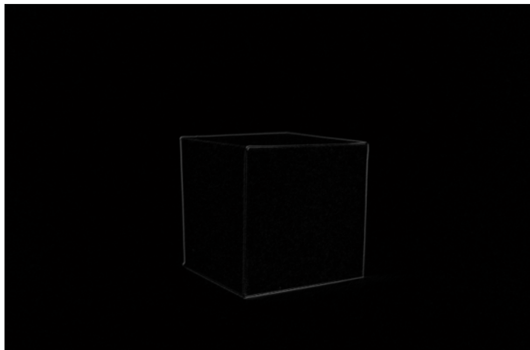

Figura 10: Gradiente componente G de cubo.

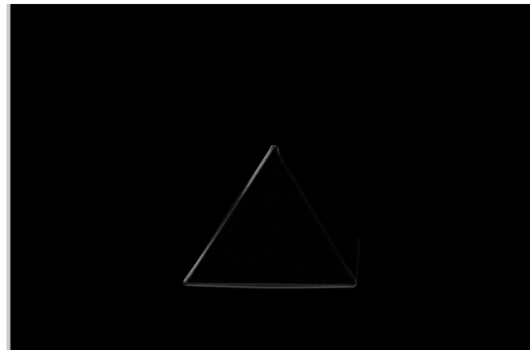

Figura 11: Gradiente componente G de pirámide.

```
[gradient_Bx gradient_By]=gradient(double(B)/255);
```

[6] En tratamiento de imágenes es la técnica mediante la cual los valores de los pixeles de una imagen se cambian dependiendo de uno o varios umbrales elegidos.

[7] Cubo es una imagen tomada con una cámara fotográfica que se utiliza de ejemplo y como apoyo visual para desarrollar paso a paso el algoritmo.

[8] Pirámide es una imagen tomada con una cámara fotográfica que se utiliza de ejemplo y como apoyo visual para desarrollar paso a paso el algoritmo.

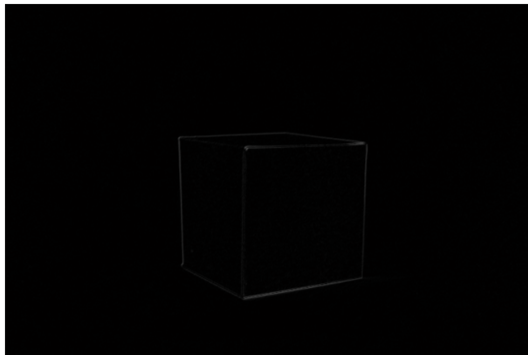

Figura 12: Gradiente componente B de cubo.

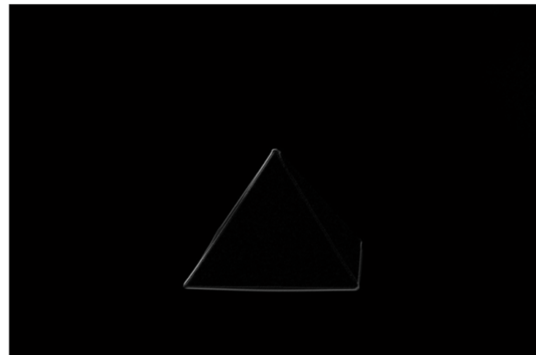

Figura 13: Gradiente componente B de pirámide.

```
[gradient_Sx gradient_Sy]=gradient(double(S));
```

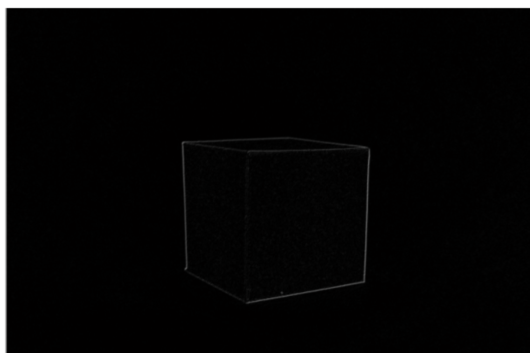

Figura 14: Gradiente componente S de cubo.

Figura 15: Gradiente componente S de pirámide.

```
gradient_image=abs(gradient_Rx)+abs(gradient_Ry)+abs(gradient_Gx)+ ...
...abs(gradient_Gy)+abs(gradient_Bx)+abs(gradient_By)+abs(gradient_Sx)
...+abs(gradient_Sy);
```

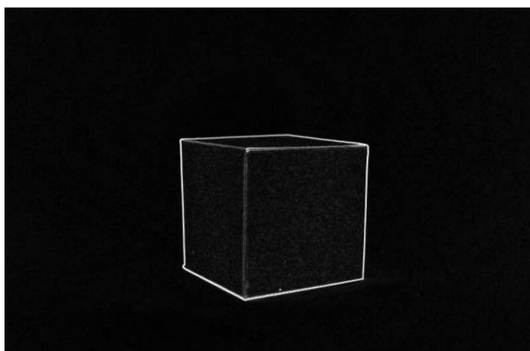

Figura 16: Gradiente final del cubo.

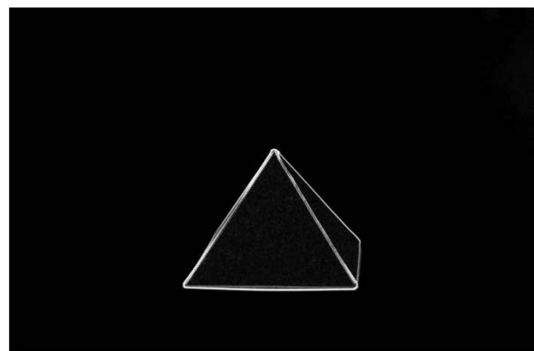

Figura 17: Gradiente final de la pirámide.

El gradiente final es una combinación entre el resultado de los cuatro gradientes de las componentes, ya que se ha comprobado que de esta forma se obtienen mejores resultados frente a otras alternativas que se han probado.

El cálculo del gradiente tiene como resultado una imagen con valores en sus pixeles decimales entre 0 y 1, a continuación se umbraliza la imagen gradiente para

obtener una imagen resultante con los pixeles en valores binarios. Los pixeles mayores a 0.15 pasarán a valer 1 y los menores de 0.15 pasaran a valer 0.

```
poss=find(gradient_image>0.15);
gradient_imageA=zeros(size(gradient_image));
gradient_imageA(poss)=1;
```

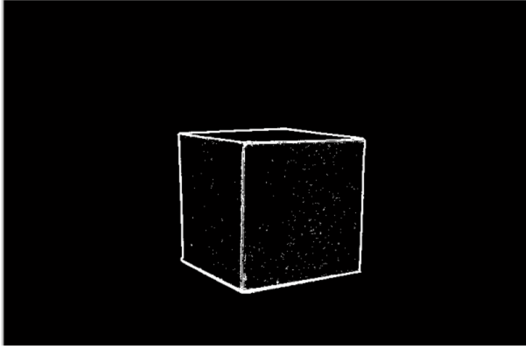

Figura 18: Gradiente final umbralizado.

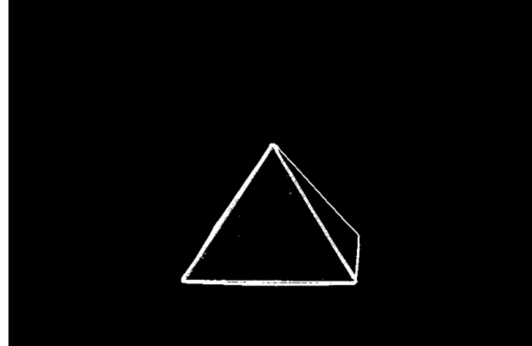

Figura 19: Gradiente final umbralizado.

El último paso del algoritmo es eliminar el ruido que pueda tener la imagen después de la búsqueda de bordes y los posibles píxeles que queden por la imagen fuera de los bordes del objeto. El ruido se quita usando un *filtro de mediana* y los pixeles sueltos mediante el operador morfológico *open.*

```
gradient_imageB = medfilt2(gradient_imageA,[3 3]);
se=strel('square',2);
gradient_imageB=imopen(gradient_imageB,se);
```

A continuación se pueden observar el resultado de eliminar el ruido y por tanto el resultado final de dos imágenes utilizadas como entrada del algoritmo y el resultado de aplicar el algoritmo sobre las imágenes, con lo que se puede comprobar el buen funcionamiento de éste.

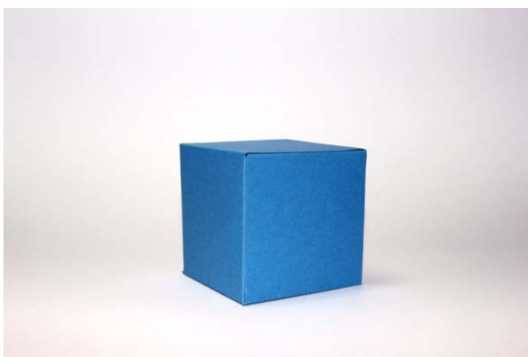

Figura 20: Imagen Entrada Algoritmo.

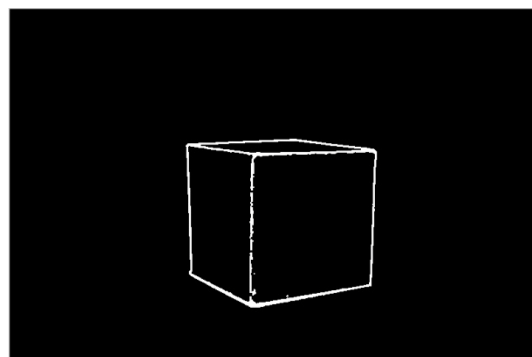

Figura 21: Resultado. Bordes del objeto.

Figura 22: Imagen Entrada Algoritmo.

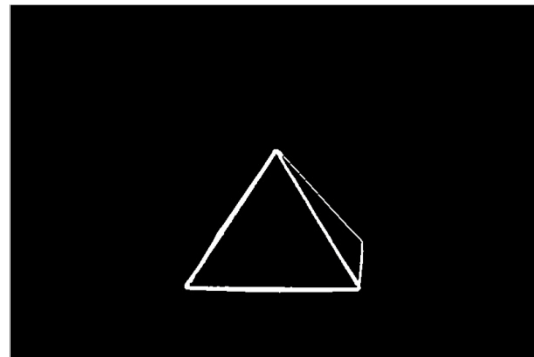

Figura 23: Resultado. Bordes del Objeto.

4.3. Fase 2. Detección de líneas rectas

4.3.1. Definición

La detección de líneas es un proceso que permite encontrar puntos alineados que puedan existir en la imagen, es decir, líneas rectas. El algoritmo que se ha utilizado en este trabajo está basado en la Transformada de Hough. Es necesario que a la detección de líneas le preceda la detección de bordes ya que necesita como datos de entrada una imagen binaria (i.e., únicamente valores de blanco (1) o negro (0)) de los pixeles que forman parte de la frontera del objeto.

La Transformada de Hough es una técnica muy robusta frente al ruido y a la existencia de huecos en la frontera del objeto. Es un algoritmo empleado en reconocimiento de patrones en imágenes que permite encontrar ciertas formas dentro de una imagen, como líneas, círculos, etc. La versión más simple consiste en encontrar líneas rectas. Su modo de operación es principalmente estadístico y consiste en que para cada punto que se desea averiguar si es parte de una línea se aplica una operación dentro de cierto rango, con lo que se averiguan las posibles líneas de las que puede ser parte el punto. Esto se continúa para todos los puntos en la imagen, al final se determina qué líneas fueron las que más puntos posibles tuvieron y esas son las líneas en la imagen.

En la versión simple el objetivo de la transformada de Hough es encontrar puntos alineados que puedan existir en la imagen, es decir, puntos en la imagen que satisfagan la ecuación de la recta, para distintos valores de ρ y θ. La ecuación de la recta en forma polar:

$$\rho = x \cdot \cos(\theta) + y \cdot \sin(\theta)$$

Por tanto hay que realizar una transformación entre el plano imagen (coordenadas x-y) y el plano o espacio de parámetros (ρ,θ).

La ventaja de este método es que evita singularidades, como por ejemplo rectas de pendiente infinita. Si se representa *y* en un plano cartesiano, una recta queda determinada mediante un punto con coordenadas (ρ,θ), mientras que un punto, se representa como una función senoidal. Si por ejemplo tenemos dos puntos, tendremos dos senoides desfasadas alfa grados dependiendo de las coordenadas de los puntos. Dichas senoides se irán cruzando cada 180º. La interpretación geométrica de este hecho, es que la función seno de cada punto, representa las infinitas rectas que pasan por cada punto, cuando dos puntos comparten la misma recta, sus representaciones senoidales se cruzan, se obtiene un punto. Cada vez que se da media vuelta (=180º) se vuelve a repetir la misma recta, por lo que volvemos a obtener otro punto, que de hecho es la misma recta.

En el siguiente apartado se explica el algoritmo utilizado en la detección de líneas rectas.

4.3.2. Algoritmo de detección de líneas rectas

El algoritmo de detección de líneas calcula la transformada de Hough de la imagen mediante la función de Matlab *hough.m.* Posteriormente busca a través de la función de Matlab *houghpeaks.m* los picos que existen en el espectro de la transformada. Después se realizan una serie de operaciones para detectar qué líneas son válidas y cuáles no lo son.

Se calcula la Transformada de Hough de la imagen de entrada de donde se obtendrán las líneas del objeto:

```
[H T R]=hough(gradient_imageB,'RhoResolution',1,'ThetaResolution',1)
```

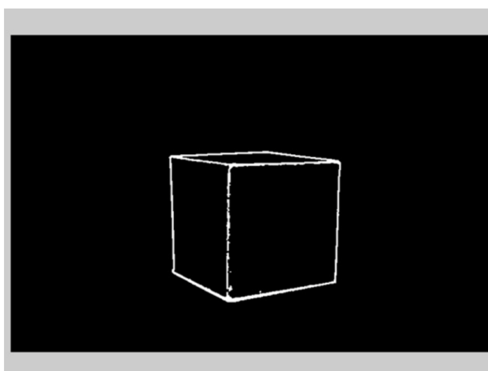

Figura 24: Imagen de entrada cubo.

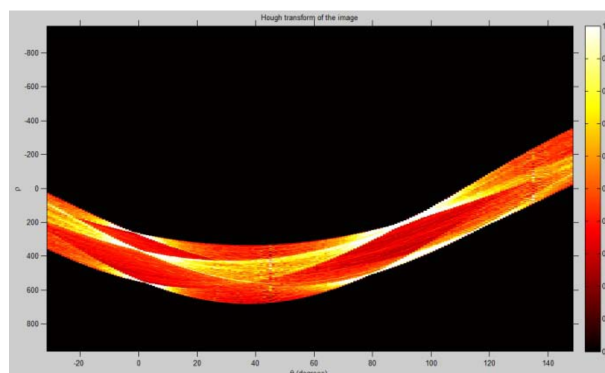

Figura 25: Transformada de Hough de cubo.

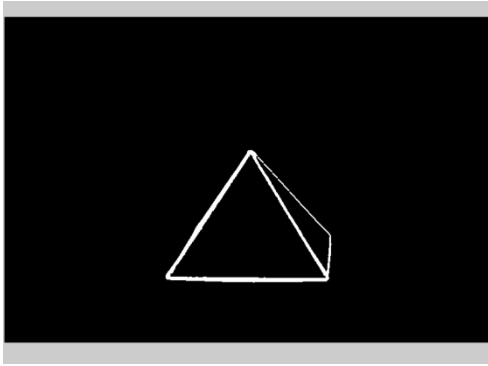

Figura 26: Imagen de entrada pirámide.

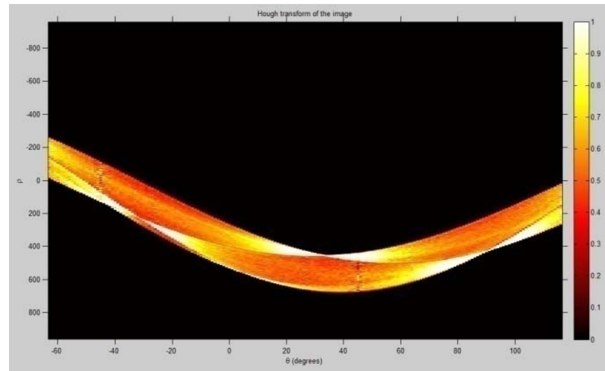

Figura 27: Transformada de Hough de pirámide.

Se calcula el umbral de la función *houghpeaks* y se buscan los picos de la transformada de Hough. Cada uno de los picos que detecte la función *houghpeaks* será una línea. *Valor* es el umbral a partir del cual la función encuentra los picos en la matriz transformada (H). Este umbral depende del objeto que aparezca en la imagen por lo que la función *decide_umbral*[9] detecta el tipo de objeto y selecciona el umbral en función de éste.

```
valor=decide_umbral(gradient_imageB);
P=houghpeaks(H,100,'Threshold',valor*max(H(:)));
```

Se guardan y ordenan los puntos de hough de menor a mayor theta (T) de cara al descarte de líneas que se hace después. A continuación se completa la simetría respecto a la otra parte del espectro de la transformada de forma que se hace un corte a la matriz de la transformada (H) y la primera parte de ésta se coloca detrás de la última parte para que en caso de haber puntos de hough cerca de los extremos queden dentro de la matriz. Además se definen los nuevos ejes de la matriz y se vuelven a detectar los nuevos picos de hough.

```
points_hough = [T(P(:,2)); R(P(:,1))]';
points_hough = sortrows(points_hough,1);

resta = diff(points_hough,1);
pos = find(resta(:,1)>25);
theta_div = (points_hough(pos(1),1)+points_hough(pos(1)+1,1))/2;
pos = find(T>theta_div);
```

[9] El algoritmo de las funciones creadas no pertenecientes a Matlab se puede ver en el Anexo I del presente documento.

```
H=[H(:,pos+1:end) flipud(H(:,1:pos))];
T=[T(pos+1:end) 180+T(1:pos)];
R=R;


P=houghpeaks(H,100,'Threshold',valor*max(H(:)));

points_hough = [T(P(:,2)); R(P(:,1))]';
points_hough = sortrows(points_hough,1);
```

Los picos se representan sobre la transformada de Hough en forma de puntos con coordenadas ρ y θ. Las líneas son la parte más brillante de la transformada y como se puede apreciar hay varias zonas perfectamente definidas donde puede haber una o varias líneas.

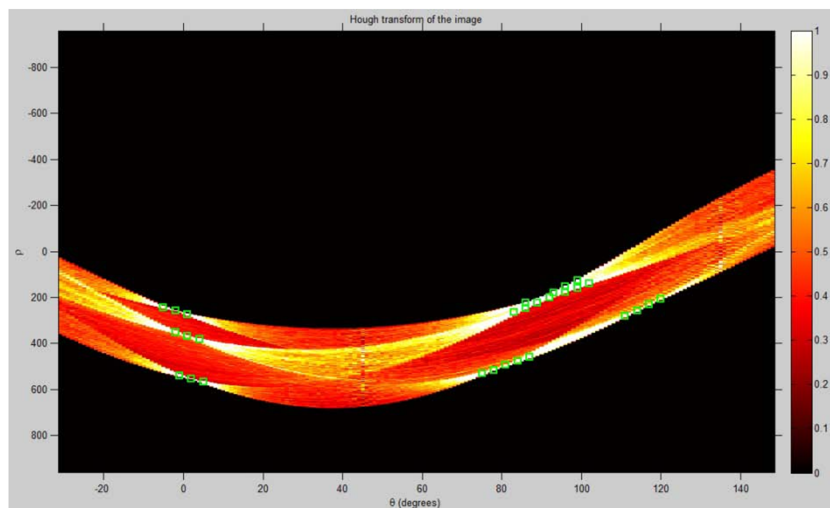

Figura 28: Transformada de Hough con todas las líneas detectadas de cubo.

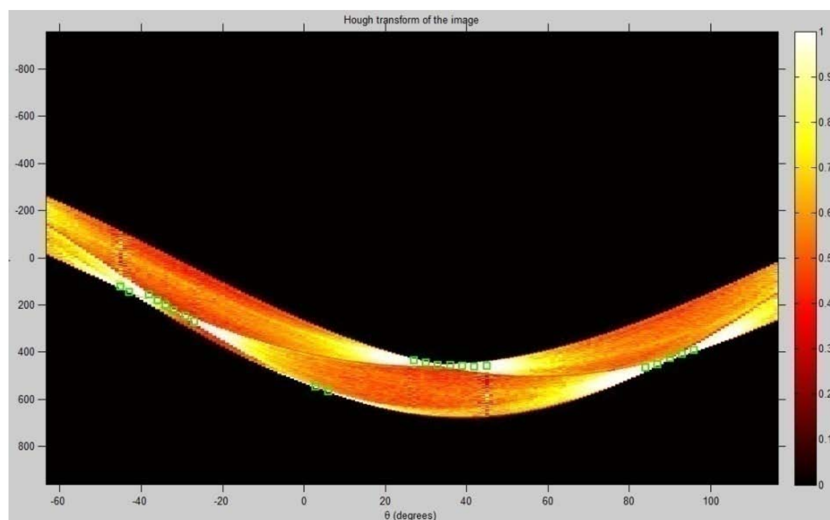

Figura 29: Transformada de Hough con todas las líneas detectadas de pirámide.

No todos los puntos detectados por la función de Matlab corresponden realmente con picos que representan rectas. El problema que se encuentra es que generalmente la detección produce varios picos en torno al pico que realmente pertenece a una recta. La causa de este problema está en la forma "pulsada" que tiene el espectro de la transformada de Hough. En las siguientes figuras puede verse el espectro de la transformada:

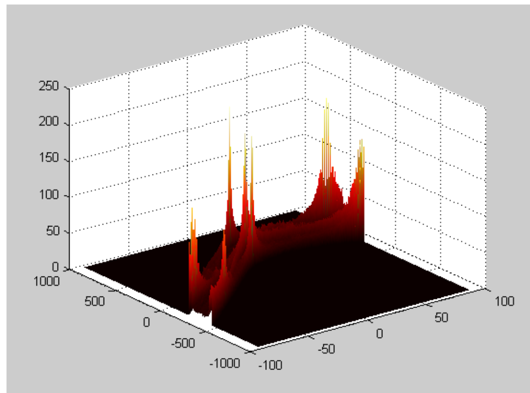

Figura 30: Espectro de la transformada de Hough.

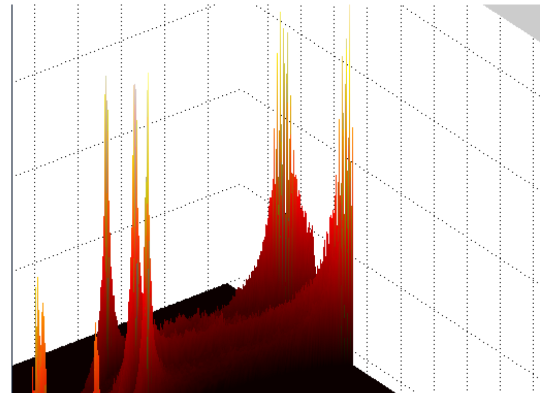

Figura 31: Zoom espectro transformada de hough.

Para solucionar el problema de la detección masiva de puntos se localizan las zonas de la transformada de Hough donde hay picos, para hacer luego un análisis más en detalle de cada zona y eliminar los picos que no correspondan a rectas.

Para considerar que varios puntos corresponden a una misma zona, donde luego se mirará si hay uno o más puntos, se considera que:

- Los puntos son de la misma zona si la theta solo cambia menos de +-5º.
- Y además los puntos son de la misma zona si rho cambia menos de +-50.

Como los puntos están ordenados, para crear la zona el algoritmo mira un punto y los siguientes si están cerca en theta y luego en rho.

```
theta_limites=5;
rho_limites=50;
points_hough2=[];
zonas_hough_inicio=[];
zonas_hough_fin=[];
puntos_por_procesar=points_hough;


while size(puntos_por_procesar,1)~=0
    H_aux=zeros(size(H));
    zonas_hough_inicio=[zonas_hough_inicio ;
```

```
    puntos_por_procesar(1,:)];
    zonas_hough_fin=[zonas_hough_fin; puntos_por_procesar(1,:)];
    puntos_por_procesar=puntos_por_procesar(2:end,:);
    pos_theta=find((T>(zonas_hough_inicio(end,1)- ...
 ...theta_limites))&(T<(zonas_hough_inicio(end,1)+theta_limites)));
    pos_rho=find((R>(zonas_hough_inicio(end,2)- ...
 ...rho_limites))&(R<(zonas_hough_inicio(end,2)+rho_limites)));
    H_aux(pos_rho,pos_theta)=H(pos_rho,pos_theta);
    buscar2=1;
    while (buscar2 & size(puntos_por_procesar,1)~=0)
        pos=find((puntos_por_procesar(:,1)>=zonas_hough_fin(end,1)-...
     ...theta_limites)&(puntos_por_procesar(:,1)<=...
     ...zonas_hough_fin(end,1)+theta_limites)&...
     ...(puntos_por_procesar(:,2)<zonas_hough_fin(end,2)+...
     ...rho_limites)&(puntos_por_procesar(:,2)>...
     ...zonas_hough_fin(end,2)-rho_limites));

        if isempty(pos)
          buscar2=0;
        else
          zonas_hough_fin(end,:)=puntos_por_procesar(pos(1),:);
          puntos_por_procesar=[puntos_por_procesar(1:pos-1,:); ...
       ...puntos_por_procesar(pos+1:end,:)];
          pos_theta=find((T>(zonas_hough_fin(end,1)-...
       ...theta_limites))&(T<(zonas_hough_fin(end,1)+theta_limites)));
          pos_rho=find((R>(zonas_hough_fin(end,2)-...
       ...rho_limites))&(R<(zonas_hough_fin(end,2)+rho_limites)));
          H_aux(pos_rho,pos_theta)=H(pos_rho,pos_theta);
   end
end

    inicio_theta=zonas_hough_inicio(end,1)-theta_limites;
    final_theta=zonas_hough_fin(end,1)+theta_limites;
    rhos=[zonas_hough_inicio(end,2) zonas_hough_fin(end,2)];
    inicio_rho=min(rhos)-rho_limites;
    final_rho=max(rhos)+rho_limites;

    pos_rho=find((R>inicio_rho)&(R<final_rho));
    pos_theta=find((T>inicio_theta)&(T<final_theta));

    H_zona=H_aux(pos_rho,pos_theta);
    T_zona=T(pos_theta);
    R_zona=R(pos_rho);
```

Una vez definida la zona y sus ejes de coordenadas, comienza su procesado, buscará los puntos de Hough dentro de esa zona que son válidos, es decir, buscará qué líneas pertenecen a la frontera del objeto, almacenará estas líneas en una variable en forma de dos coordenadas polares (θ,ρ).

Para obtener un mejor resultado al realizar la discriminación de puntos se gira la zona hasta que quede de forma vertical, esto conlleva un recalculo de las coordenadas que posteriormente habrá que deshacer con el fin de obtener las coordenadas correctamente.

```
angulo_out = detectar_angulo_Hough(H_zona);
if length(angulo_out)>1
   pos=find(angulo_out>0);
   angulo_out=angulo_out(pos);
end
H_zona2 = imrotate(H_zona,angulo_out,'bilinear','loose');
```

Mediante el uso de la función *detectar_picos_Hough* se realiza una búsqueda más exhaustiva de los puntos y se descartan los que no pertenezcan a un pico del espectro. A continuación se desarrolla la forma en que trabaja dicha función:

```
posiciones_rot=detectar_picos_Hough(H_zona2,1,size(H_zona2,2),...
...1,size(H_zona2,1))
```

Esta función obtiene la proyección del espectro de la zona en una de las dimensiones, concretamente en el eje ρ, obtiene su envolvente y realiza una búsqueda de picos de forma que sólo detecta los picos que tienen entre ellos una distancia mayor a la del umbral establecido, como se ve en la *figura 33*, y además, que su amplitud esté por encima de un segundo umbral, de esta manera descarta los picos que estén alrededor del pico más alto que es realmente el pico que busca.

Si encuentra un solo pico, realiza el mismo proceso con la proyección de la otra dimensión θ. Por lo que encuentra el mismo pico en las dos proyecciones descartando los picos menores como puede verse en las siguientes figuras:

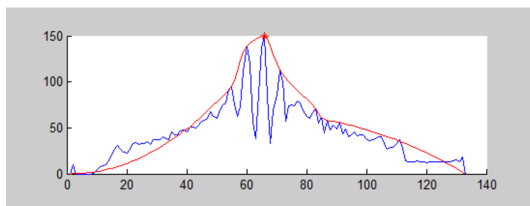

Figura 32: Proyección y envolvente eje Y (rho).

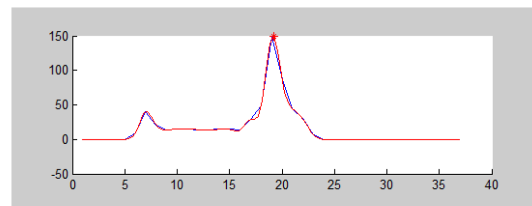

Figura 33: Proyección y envolvente eje X (theta).

Puede ocurrir que en lugar de encontrar un solo pico encuentre más de uno, por lo que para realizar correctamente la detección divide la proyección en tantas partes como picos encuentra, y vuelve a realizar el mismo proceso explicado para hacer una búsqueda iterativa hasta que encuentra un solo pico en ambas proyecciones Este proceso se repite tantas veces como picos encuentra. Puede verse un esquema del proceso en las siguientes imágenes:

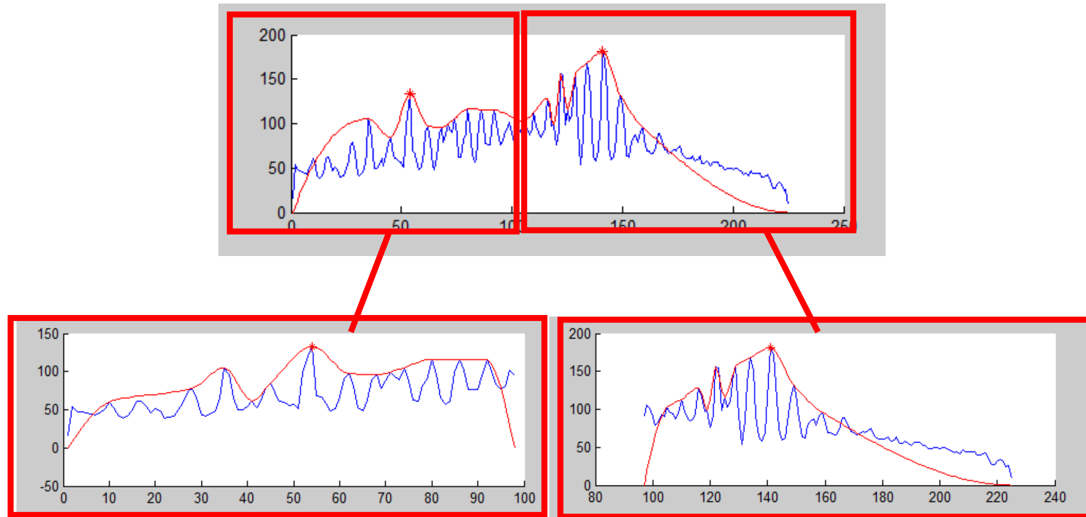

Figura 34: Picos encontrados en proyección rho y división en tantas partes como picos encontrados.

Una vez hecha la división de la proyección rho se buscan los picos de la proyección theta para cada una de las partes de la división.

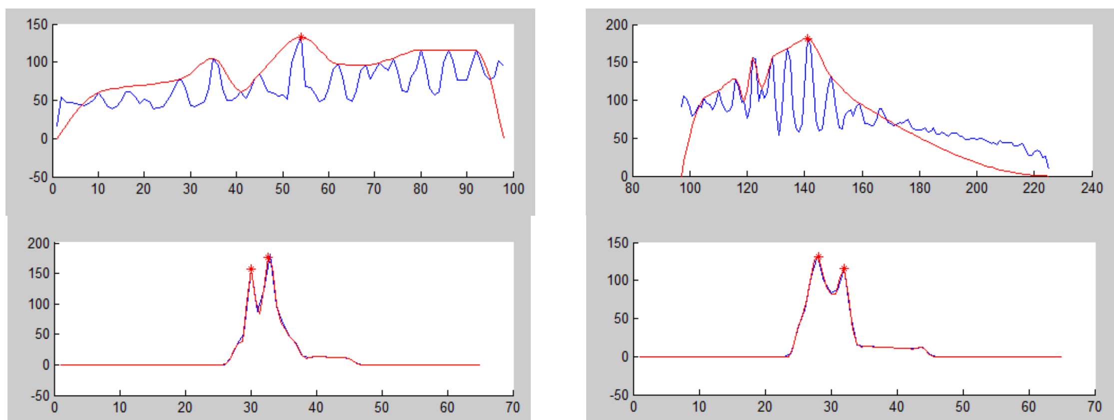

Figura 35: Picos en la proyección theta (abajo) a partir de las divisiones de proyección rho (arriba).

Al buscar picos en la proyección de theta se observa como se han encontrado dos picos posibles para cada una de las proyecciones en rho. Estos picos pueden ser puntos de hough reales y por tanto habría cuatro puntos o puede que alguno de ellos no pertenezca a ninguna recta real, para ello se realiza la búsqueda iterativa de picos y se vuelve a realizar el proceso de división. Cuando hace la división busca en la otra proyección el pico para encontrar la coordenada que falta, en caso que encuentre pico

tiene el punto completo con sus dos coordenadas y por el contrario si no encuentra pico en la otra proyección ese punto no es real y por tanto no es válido y es descartado:

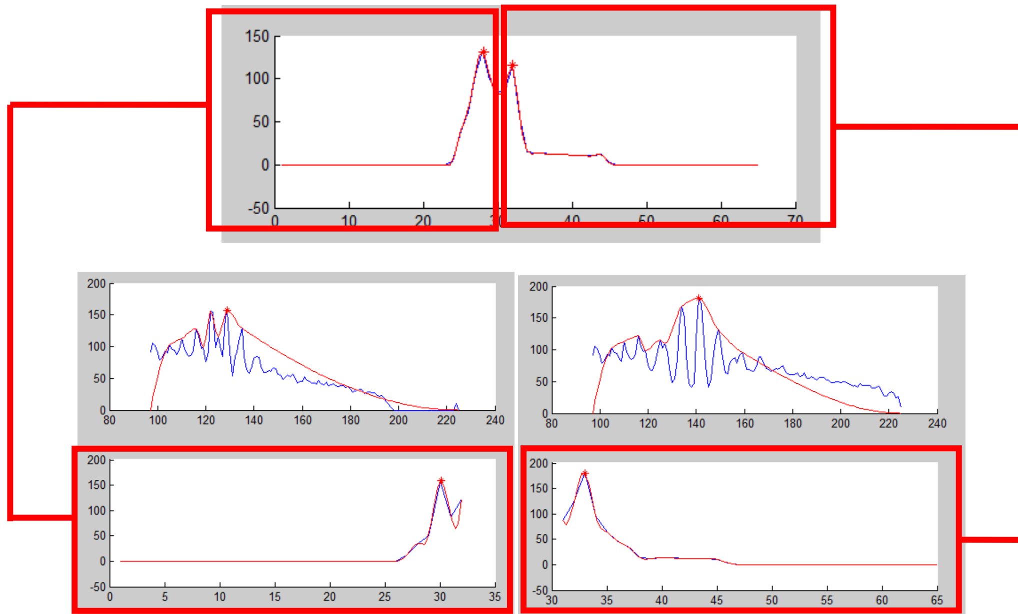

Figura 36: División de proyección theta y búsqueda de picos en proyección rho.

En el caso de la *figura 36* los picos son reales ya que encuentra pico en las dos proyecciones después de la división.

Finalmente, realizando la búsqueda iterativa de picos en las proyecciones de la zona del espectro de la transformada de hough donde hay puntos, la función obtiene solo los puntos reales de esa zona.

A continuación, el algoritmo de detección de líneas deshace la rotación anteriormente aplicada para obtener las coordenadas reales de los puntos de hough.

```
    posiciones_orig=zeros(size(posiciones_rot));
    centro_rot=fliplr(floor(size(H_zona2)/2));
    centro_orig=fliplr(floor(size(H_zona)/2));

    for ind2=1:size(posiciones_rot,1)
        posiciones_rot(ind2,:)=posiciones_rot(ind2,:)-centro_rot;
        radio=sqrt(sum(posiciones_rot(ind2,:).^2));
        alpha_rot=atan(posiciones_rot(ind2,2)/posiciones_rot(ind2,...
     ...1))*180/pi;


        if (sign(posiciones_rot(ind2,1))==-1) &&...
...(sign(posiciones_rot(ind2,2))==-1)
          alpha_rot=alpha_rot-180;
```

```
      end

      if (sign(posiciones_rot(ind2,1))==-1) &&...
...(sign(posiciones_rot(ind2,2))==1)
         alpha_rot=alpha_rot-180;
      end

      alpha_orig=alpha_rot+angulo_out;
      posiciones_orig(ind2,:)=...
   ...radio*[cos(alpha_orig*pi/180) sin(alpha_orig*pi/180)];

      posiciones_orig(ind2,:)=round(posiciones_orig(ind2,:)+...
   ...centro_orig);

      punto=[T_zona(posiciones_orig(ind2,1)),...
      ...R_zona(posiciones_orig(ind2,2))];

      points_hough2=[points_hough2; punto];
    end
end
```

Cabe la posibilidad de que haya dos líneas que sean idénticas, por tanto en caso de que las haya se elimina una de ellas:

```
points_hough4=points_hough2(1,:);
for ind1=2:size(points_hough2,1)
    pos_repetido=find(points_hough2(ind1,1)==points_hough4(:,1) &...
  ...points_hough2(ind1,2)==points_hough4(:,2));
    if length(pos_repetido)==0
       points_hough4=[points_hough4; points_hough2(ind1,:)];
    end
end
points_hough2=points_hough4;
```

Una vez se tienen todos los puntos de hough que pertenecen a rectas del objeto se crean las ecuaciones de dichas rectas:

```
m_param2=-1./tan(points_hough2(:,1)*pi/180);
b_param2=points_hough2(:,2)./sin(points_hough2(:,1)*pi/180);
n_lines2=length(m_param2)
x_vector=linspace(1,im_width,10);
for ind1 = 1:n_lines2
    [m_param2(ind1) b_param2(ind1)]
```

```
    points_hough2(ind1,:)
    y=m_param2(ind1)*x_vector+b_param2(ind1);
end
```

Y una vez eliminadas las líneas que no sean válidas así como las líneas repetidas o las que no pertenecen a ninguna arista del objeto los resultados serían los siguientes:

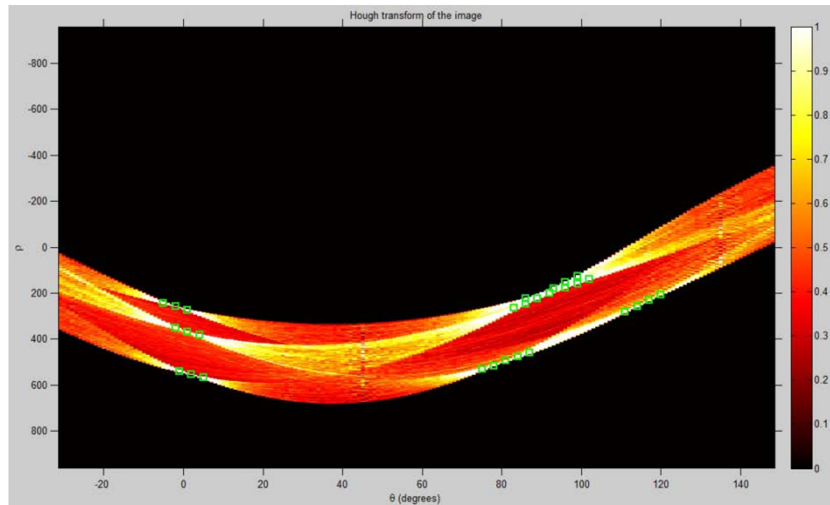

Figura 37: Transformada de Hough con todas las líneas detectadas de cubo.

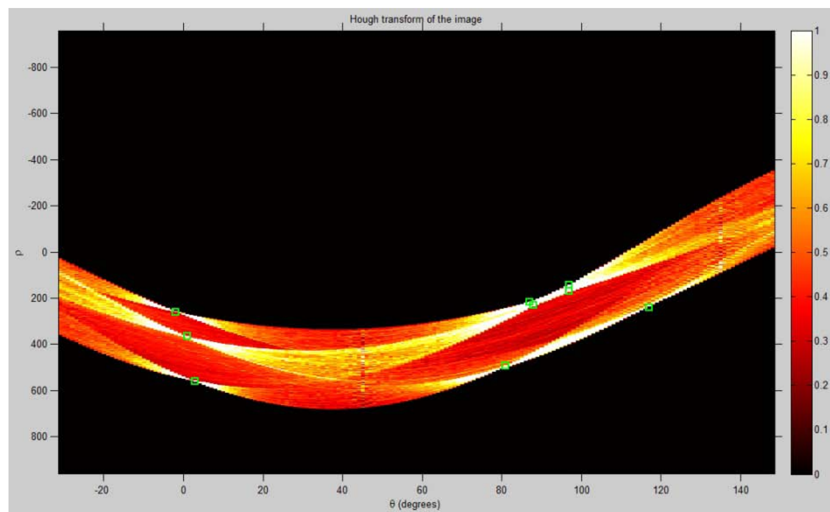

Figura 38: Transformada de Hough con las líneas reales del objeto cubo.

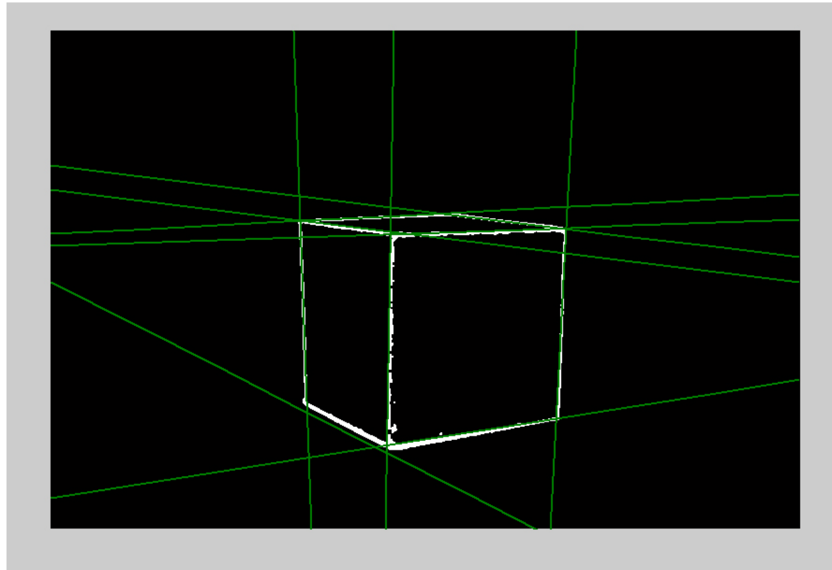

Figura 39: Líneas detectadas sobre la frontera del objeto.

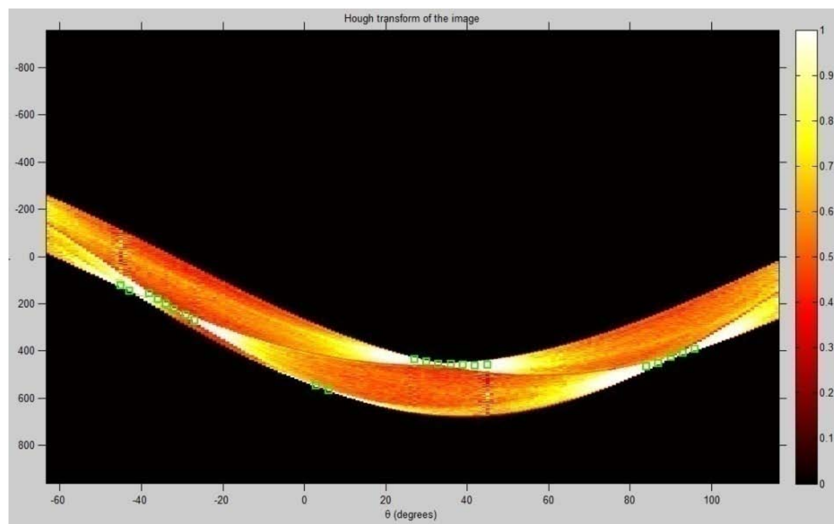

Figura 40: Transformada de Hough con todas las líneas detectadas pirámide.

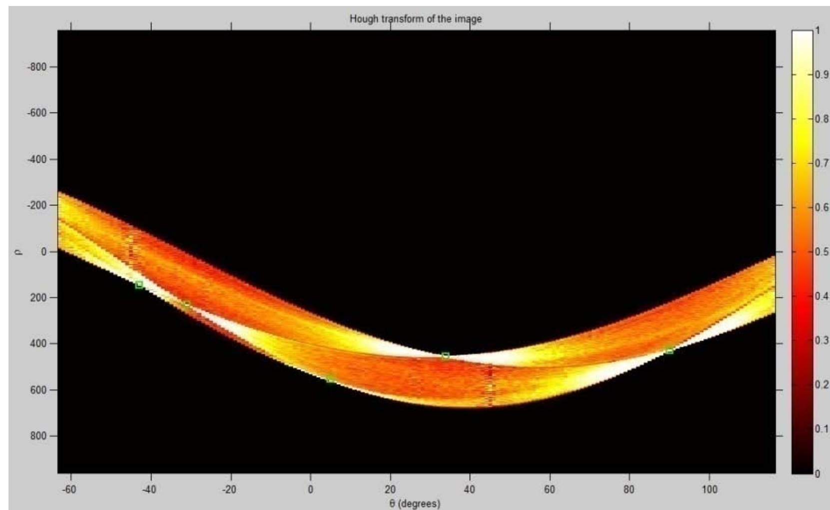

Figura 41: Transformada de Hough con las líneas reales del objeto pirámide.

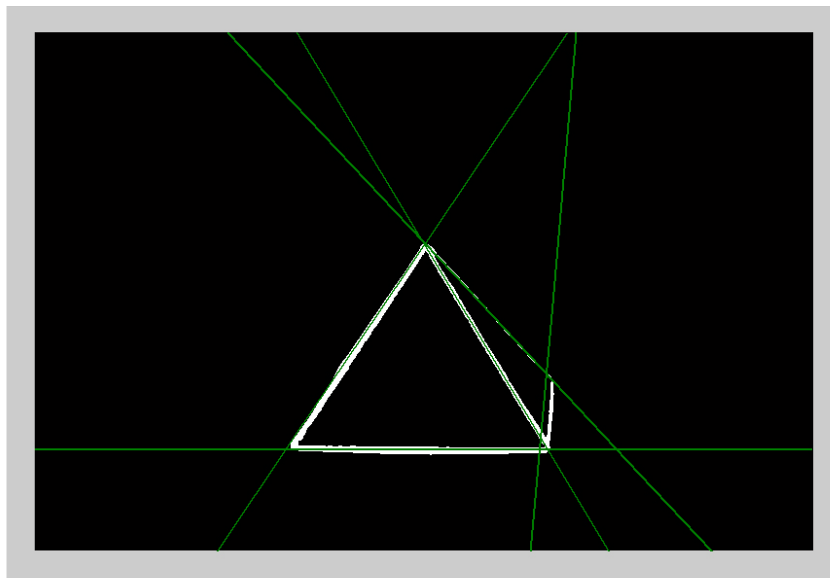

Figura 42: Líneas detectadas sobre la frontera del objeto pirámide.

Como se puede apreciar en la *figura 39* y en la *figura 42*, no todas las líneas han sido detectadas perfectamente sino que alguna está desplazada. Es por ello que se crea el algoritmo de Refinamiento, que se desarrolla en el punto siguiente.

4.4. Fase 3. Refinamiento

4.4.1. Definición

El refinamiento es un proceso mediante el cual se perfecciona la detección de las dos fases anteriores con el fin de reproducir con mayor fidelidad el modelo 3D del objeto en reconstrucción.

La detección de líneas puede no ser totalmente efectiva y cabe la posibilidad que no detecte alguna línea correctamente y por tanto la detección de vértices también será errónea. Mediante esta fase de refinamiento se consigue que las líneas no detectadas y/o las detectadas incorrectamente se localicen y/o se corrijan y por tanto mejore la reconstrucción del objeto.

4.4.2. Algoritmo de refinamiento

El algoritmo de refinamiento hace una búsqueda más exhaustiva de las líneas ya detectadas para corregir líneas que en la detección no hayan sido correctas.

Este algoritmo busca, para cada una de las líneas encontradas en la fase anterior de detección, si alrededor de la línea existe alguna otra que esté definida con mayor intensidad en la transformada de Hough (mayor valor del pico en la transformada de Hough); si encuentra alguna cambiará la línea, ya que ésta nueva será más precisa que la que había encontrado anteriormente.

```
points_hough2aux=zeros(size(points_hough2));
points_hough3=[];
n_points_hough = n_lines2;
for ind1=1:n_points_hough
    refinar=1;
    if refinar==1
       inicio_theta=points_hough2(ind1,1)-5;
       final_theta=points_hough2(ind1,1)+5;
       inicio_rho=points_hough2(ind1,2)-7;
       final_rho=points_hough2(ind1,2)+7;

       pos_rho=find((R>inicio_rho)&(R<final_rho));
       pos_theta=find((T>inicio_theta)&(T<final_theta));
```

```
        H_zona=H(pos_rho,pos_theta);
        T_zona=T(pos_theta);
        R_zona=R(pos_rho);

        [f,c]=find(H_zona==max(max(H_zona)));
        punto_refinado=[T_zona(c(1)), R_zona(f(1))];
        punto_antes=points_hough2aux(ind1,:);
        distancia=sqrt((punto_refinado(:,1)-...
    ...punto_antes(1)).^2+(punto_refinado(:,2)-punto_antes(2)).^2);
        points_hough2aux(ind1,:)=punto_refinado;
         if distancia<5
             refinar=0;
         end
    end

    points_hough3=[points_hough3; points_hough2aux(ind1,:)];

end

pos=find(abs(points_hough3(:,1))<0.1);
points_hough3(pos)=0.1;
```

Cabe la posibilidad de que hayan dos líneas que sean idénticas, en cuyo caso se eliminará una de ellas. Después se crean las ecuaciones de las nuevas rectas:

```
points_hough4=points_hough3(1,:);
for ind1=2:size(points_hough3,1)
    pos_repetido=find(points_hough3(ind1,1)==points_hough4(:,1) &...
   ...points_hough3(ind1,2)==points_hough4(:,2));
    if length(pos_repetido)==0
        points_hough4=[points_hough4; points_hough3(ind1,:)];
    end
end

points_hough3=points_hough4;

m_param3=-1./tan(points_hough3(:,1)*pi/180);
b_param3=points_hough3(:,2)./sin(points_hough3(:,1)*pi/180);
```

```
n_lines3=length(m_param3)
x_vector=linspace(1,im_width,10);
for ind1 = 1:n_lines3
    [m_param3(ind1) b_param3(ind1)]
    points_hough3(ind1,:)
    y=m_param3(ind1)*x_vector+b_param3(ind1);
end
```

El resultado de esta fase se puede ver en las siguientes imágenes junto con la imagen de las líneas de entrada, donde se comprueba que ajusta más las líneas a los bordes reales del objeto.

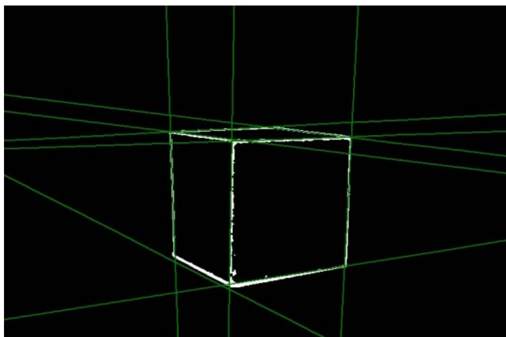

Figura 43: Líneas antes del refinamiento.

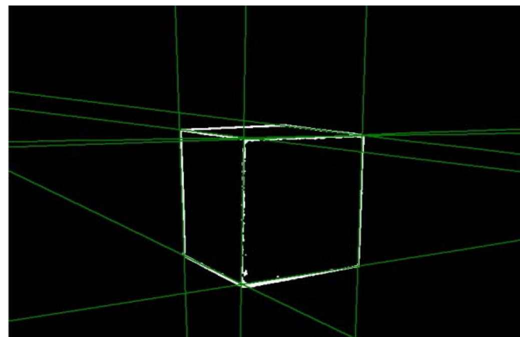

Figura 44: Líneas después del refinamiento.

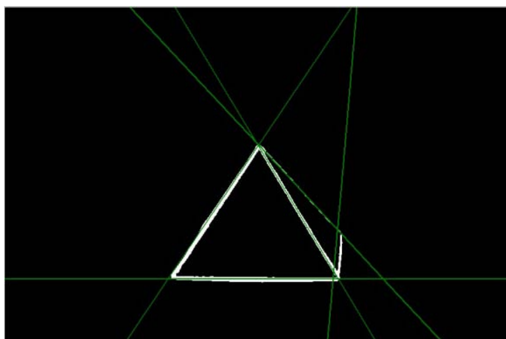

Figura 45: Líneas antes del refinamiento.

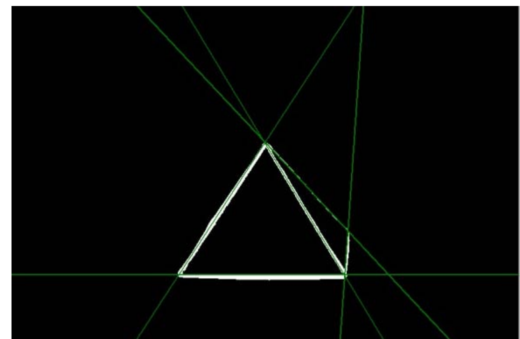

Figura 46: Líneas después del refinamiento.

Como se puede ver en las figuras anteriores las líneas que estaban desplazadas antes de la fase de refinamiento han sido corregidas tras el refinamiento.

Aunque puede ocurrir que después del refinamiento alguna línea correcta haya sido desplazada esto ya no será un problema a la hora de detectar los vértices.

4.5. Fase 4. Detección de esquinas

4.5.1. Definición

Una esquina puede ser definida como la intersección de dos bordes o como un punto en el que hay dos bordes con direcciones distintas y dominantes en la zona cercana al punto. Otra forma de definir una esquina es como una zona donde las variaciones de intensidad en las direcciones x e y son grandes o dicho de otra manera una región donde la intensidad varía en ambas direcciones.

La detección de esquinas es un proceso fundamental en muchas aplicaciones además de la reconstrucción 3D. Las esquinas en imágenes representan información útil y son muy importantes para describir objetos para su reconocimiento e identificación.

Un detector de esquinas requiere que se cumplan determinados requisitos. En primer lugar, todas las esquinas verdaderas deben ser detectadas y ninguna esquina falsa. En segundo lugar, las esquinas detectadas tienen que estar correctamente localizadas. Además el detector debe tener repetitividad (estabilidad), ser robusto ante ruido y ser computacionalmente eficiente. El algoritmo más utilizado en la detección de esquinas es el algoritmo de Harris.

Actualmente los algoritmos para la detección de esquinas presentan algún tipo de limitación como puede ser susceptibilidad al ruido, detección de esquinas falsas, imposibilidad para encontrar algunos puntos de esquina, alto costo computacional en consumo de tiempo y recursos de máquina, no identificación del tipo de esquina…

Debido a estas limitaciones en este trabajo no se utiliza ningún algoritmo preestablecido de detección de esquinas, sino que se utiliza un algoritmo que obtiene los vértices a partir de la fase anterior de detección de líneas. Este algoritmo utiliza como datos de entrada las líneas conseguidas en la detección de líneas y busca los vértices como los puntos de corte entre las líneas descartando los puntos de corte que encuentra fuera de la frontera del objeto y conservando los puntos que se encuentran en la frontera del objeto y que son vértices reales.

4.5.2. Algoritmo de detección de esquinas

El algoritmo de la detección de esquinas se divide en dos partes, que a la vez son dos funciones creadas específicamente que realizan el proceso de buscar todos los puntos de corte entre las líneas, que son los posibles vértices del objeto, y luego eliminar los puntos innecesarios.

Antes que nada se dilata la frontera del objeto para que posteriormente considere que los vértices encontrados se corresponden con los del objeto pese a que estén un poco desplazados respecto a la posición ideal. Posteriormente se utilizan las funciones creadas para la detección de los vértices:

```
se=strel('square',9);
gradient_imageC=imdilate(gradient_imageB,se);

pos=find(abs(points_hough2(:,1))<0.2);
points_hough2(pos)=0.2;

vertices_fin=detecta_vertices(gradient_imageC,b_param3,m_param3,...
...points_hough3,n_lines3);

[vertices_figura figura] = decide_figura(gradient_imageC,...
...vertices_fin, points_hough3,handles.gradient_imageB);
```

A continuación se va a desarrollar el algoritmo de las funciones utilizadas. La primera de las dos funciones *detecta_vertices* realiza cuatro pasos:

1. Detecta todos los puntos de corte entre las rectas (recordar que las rectas son infinitas por lo que existen puntos de corte que no aparecerán en la imagen porque quedarán fuera de las dimensiones de ésta).

```
vertices=[];
vertices_x=[];
vertices_y=[];
for ind1 = 1:n_lines2
  if ind1>1
   for ind2=1:ind1-1
     x_vertice=(b_param2(ind1)-...
  ...b_param2(ind2))/(m_param2(ind2)-m_param2(ind1));
     vertices_x=[vertices_x; round(x_vertice)];
     if isnan(x_vertice)~=1
       y_vertice=m_param2(ind1)*x_vertice+b_param2(ind1);
       vertices_y=[vertices_y; round(y_vertice)];
       vertices=[vertices; round(x_vertice) round(y_vertice)];
     end
   end
  end
end
```

El resultado de este paso de la detección de vértices se ve en la siguiente imagen:

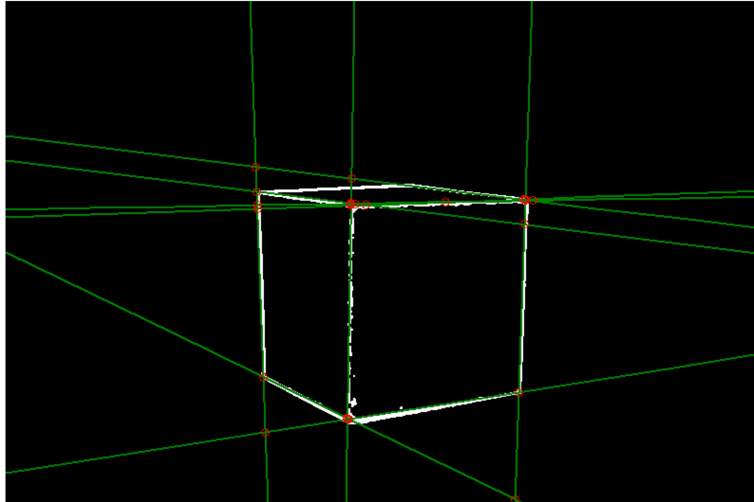

Figura 47: Imagen cubo con todos los puntos de corte existentes.

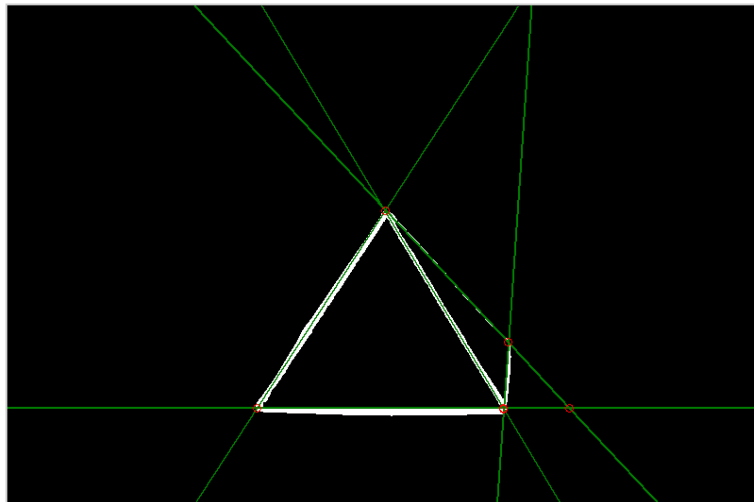

Figura 48: Imagen pirámide con todos los puntos de corte existentes.

2. Elimina los puntos de corte que quedan fuera de las dimensiones de la imagen.

```
size_im=size(image_in);
size_y=size_im(1);
size_x=size_im(2);
pos=find(vertices_x(:)>0 & vertices_x(:)<size_x &...
...vertices_y(:)>0 & vertices_y(:)<size_y);
vertices_x_new=vertices_x(pos);
vertices_y_new=vertices_y(pos);
```

El resultado de este paso no es visible debido a que los puntos que elimina no son visibles en la imagen, por tanto el resultado perceptible sería el mismo que en el paso anterior y que se aprecia en la *figura 47* y *figura 48*.

3. Elimina los puntos de corte que quedan dentro de las dimensiones de la imagen, pero que no pertenecen a la frontera del objeto.

```
vertices_final=[];
for ind2=1:length(vertices_x_new)
    if image_in(vertices_y_new(ind2),vertices_x_new(ind2))~=0
        vertices_final=[vertices_final;...
     ...vertices_x_new(ind2) vertices_y_new(ind2)];
    end
end
```

Como se ve en la *figura 49* y *figura 50*, en este paso el algoritmo se queda solo con los puntos de corte que pertenecen al borde del objeto:

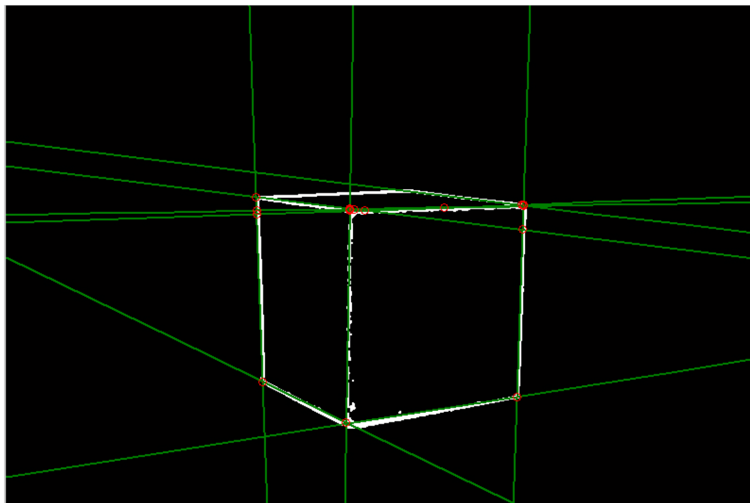

Figura 49: Imagen cubo con puntos de corte sobre la frontera.

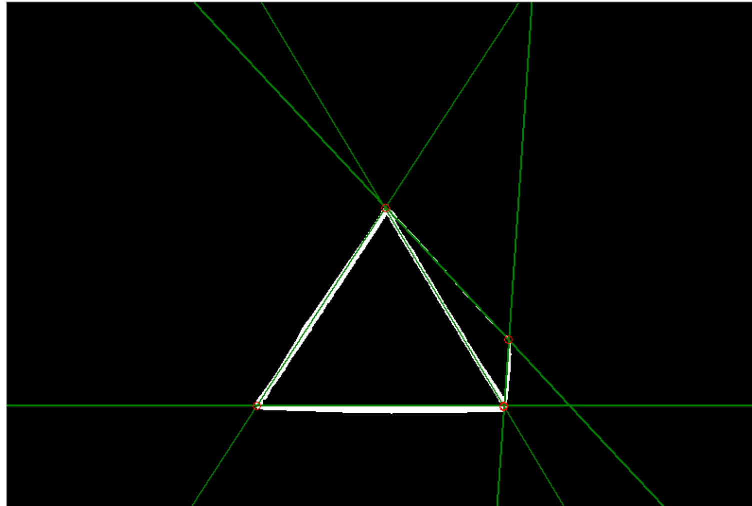

Figura 50: Imagen pirámide con puntos de corte sobre la frontera.

4. Hace un reconocimiento de zonas donde hay puntos de corte para eliminar los puntos muy próximos que puedan ser vértices repetidos.

```
vertices_final=sortrows(vertices_final);
vertices_fin=[];
vertices_por_procesar=vertices_final;

while size(vertices_por_procesar,1)~=0
    vertices_fin=[vertices_fin ; vertices_por_procesar(1,:)];
    vertices_por_procesar=vertices_por_procesar(2:end,:);
    if size(vertices_por_procesar,1)~=0
       v1=vertices_por_procesar;
       v2=vertices_fin(end,:);
       distancia=sqrt((v1(:,1)-v2(1)).^2+(v1(:,2)-v2(2)).^2);
       pos=find(distancia<10);

       if isempty(pos)==0
          vertices_fin(end,:)=mean([vertices_fin(end,:) ;
          vertices_por_procesar(pos,:)]);
          vertices_por_procesar(pos,:)=[];
       end
    end
end
```

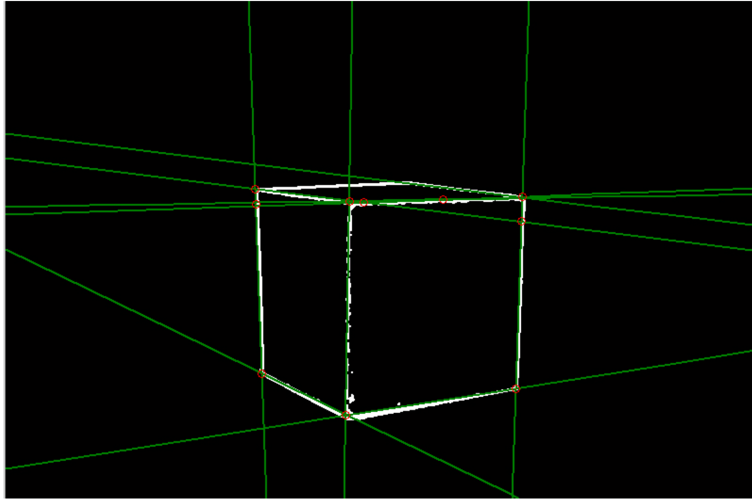

Figura 51: Imagen cubo con puntos de corte de la frontera sin repetir.

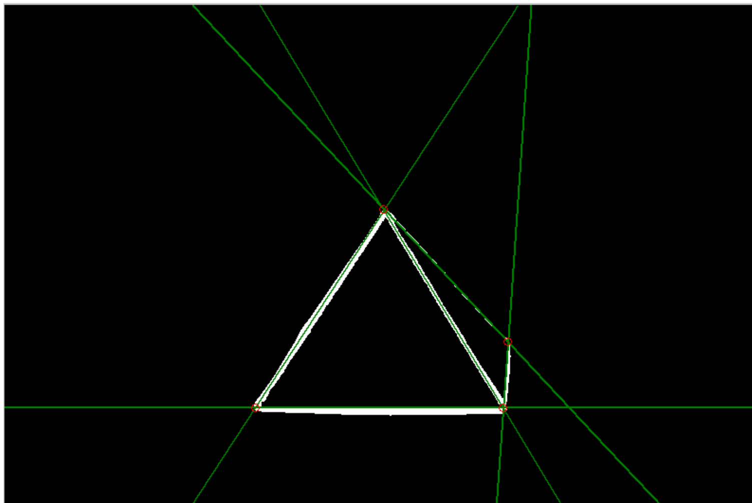

Figura 52: Imagen pirámide con puntos de corte de la frontera sin repetir.

La segunda función *decide_figura* realiza una serie de comprobaciones y cálculos para quedarse únicamente con los puntos que son vértices reales del objeto en función del tipo de figura, así como dependiendo del número de líneas que ha obtenido el detector de líneas de la fase anterior.

Asimismo, en caso de que la detección de vértices no haya funcionado a la perfección y falte algún vértice por obtener, como ocurre con el objeto cubo que se utiliza como ejemplo, la función *decide_figura* lo detecta y obtiene todos los vértices finales del objeto. Estos vértices "estimados" pueden obtenerse teniendo en cuenta que se sabe con qué tipo de objeto se trabaja (cubo o pirámide) y que se tienen otros vértices

del objeto ya determinados mediante el análisis realizado previamente (por tanto, se puede estimar dónde deberían estar los vértices que el algoritmo no ha sido capaz de determinar)

Debido a la larga extensión de esta función se ha optado por incluir su algoritmo en el *Anexo I* contenido al final de este documento. No obstante el resultado de esta función, y por ende el resultado de la detección de vértices, se muestra en las siguientes figuras. Por ejemplo, para el caso del cubo, se puede ver como finalmente no se han considerado los vértices erróneos que aparecían en los lados verticales izquierdo y derecho, así como en el borde frontal derecho de la cara superior del cubo. Además, también se ha estimado de forma correcta la posición del vértice trasero de la cara superior del cubo, el cual no se obtendría a partir del cálculo de vértices realizado previamente.

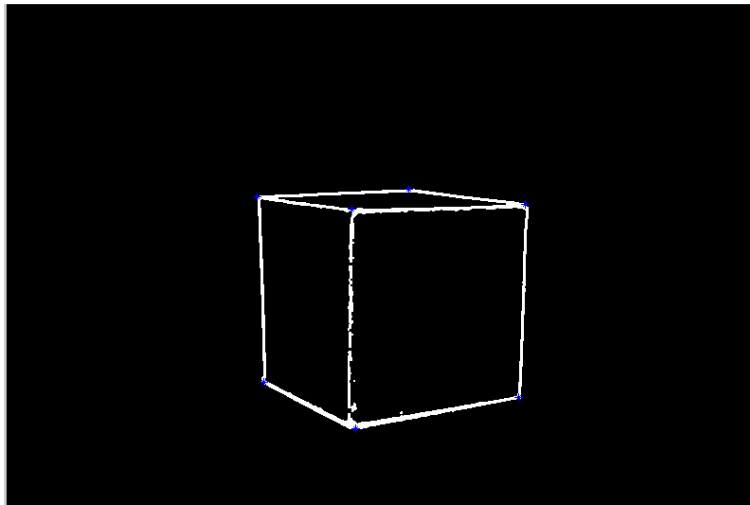

Figura 53: Imagen cubo con todos los vértices detectados.

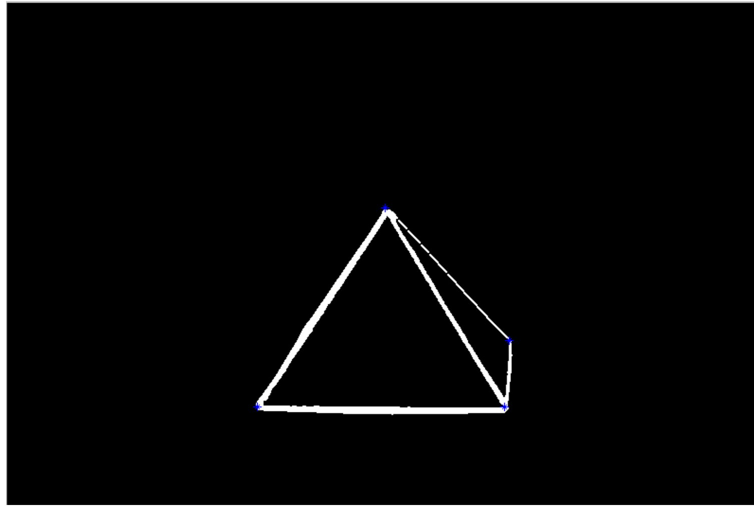

Figura 54: Imagen pirámide con todos los vértices detectados.

4.6. Fase 5. Virtualización y Renderizado

4.6.1. Definición

La virtualización es la creación de un modelo 3D de la escena u objeto. El renderizado es el proceso de interpretar la virtualización, es decir, interpretar la escena en tres dimensiones y representarla en una imagen bidimensional.

Una vez está el objeto virtualizado, para que el renderizado sea lo más parecido al objeto real de entrada se realizan una serie de cálculos de las características físicas del objeto, como puede ser posición en la escena, tamaño, color y líneas ocultas y visibles. El cálculo de estas características físicas solo es posible tras la detección de vértices.

Para detectar qué vértices y aristas están ocultos se hace uso del método de triangulación. En términos generales triangular supone cotejar al menos tres puntos de referencia para el conocimiento de la posición de un objeto. En este caso los vértices se cotejan mediante este método (siendo los puntos de referencia los vértices de las caras) para conocer si están ocultos o visibles. Si un vértice está dentro de alguna cara del objeto estará oculto, mientras que si queda fuera de la cara estará visible.

4.6.2. Transformaciones en el Modelado en 3D

La variación de la posición y/o el tamaño de los objetos, con respecto a los sistemas de referencia, se hace mediante *transformaciones lineales*.

La manera más fácil de conseguir las transformaciones básicas es utilizando matrices de transformación.

Aplicando ligeros cambios a las matrices, se pueden combinar para conseguir que una sola matriz resultante sirva para varias de estas transformaciones o se pueden aplicar dichas matrices por separado consiguiendo el mismo resultado. Para aplicar las matrices de transformación se utilizan coordenadas homogéneas ya que esto ayuda a obtener los resultados deseados.

Las transformaciones más utilizadas, escalado, traslación, rotación y proyección perspectiva, se detallan a continuación.

4.6.2.1. Escalado

Dentro de un espacio de referencia los objetos pueden modificar su tamaño relativo en uno, dos, o los tres ejes.

Para escalar un objeto al tamaño del vector de escalado $v = (v_x, v_y, v_z)$, cada coordenada homogénea del punto $p = (p_x, p_y, p_z, 1)$ tiene que ser multiplicada por su matriz de transformación de escalado:

$$S_v = \begin{bmatrix} v_x & 0 & 0 & 0 \\ 0 & v_y & 0 & 0 \\ 0 & 0 & v_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Como se muestra a continuación, la multiplicación da el resultado esperado:

$$p' = S_v \cdot p = \begin{bmatrix} v_x & 0 & 0 & 0 \\ 0 & v_y & 0 & 0 \\ 0 & 0 & v_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} v_x p_x \\ v_y p_y \\ v_z p_z \\ 1 \end{bmatrix}$$

Las coordenadas del punto final *p'* serán $p'=(v_x \cdot p_x, v_y \cdot p_y, v_z \cdot p_y, 1)$.

La escala es uniforme si y sólo si los factores de escala son iguales $v_x=v_y=v_z$. Si todos excepto uno de los factores de escala son iguales a 1, tenemos escalado direccional.

En el caso donde $v_x=v_y=v_z=k$ el escalado también se llama *ampliación* o *dilatación por un factor k*, aumentando el área por un factor de k^2 y el volumen por un factor de k^3.

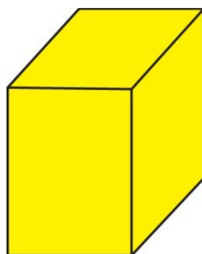

Figura 55: Cambio de escala uniforme.

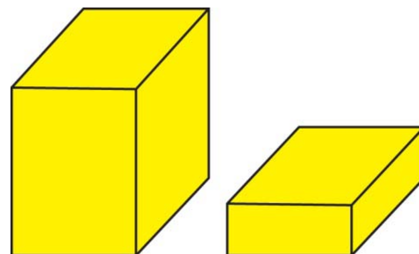

Figura 56: Cambio de escala no uniforme.

Para obtener el cambio de escala inverso al realizado aplicando la matriz S_v, basta con multiplicar los puntos finales por la matriz inversa de S_v, o sea, por la matriz S_v^{-1}, que se obtiene al sustituir v_x, v_y, v_z en S_v por $1/v_x$, $1/v_y$, $1/v_z$, respectivamente. Por lo que la matriz de transformación sería,

$$S_v^{-1} = \begin{bmatrix} \frac{1}{v_x} & 0 & 0 & 0 \\ 0 & \frac{1}{v_y} & 0 & 0 \\ 0 & 0 & \frac{1}{v_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Los cambios de escala no uniformes deforman los objetos, por lo que pueden resultar interesantes (siempre que se realicen de manera controlada); por el contrario, el escalado uniforme no deforma los objetos, por lo que se emplea con mayor frecuencia.

Esta operación de escalado hay que realizarla con todos los puntos del objeto, es decir, se ha de aplicar la matriz S_v a todos los vértices del objeto. Si no se toman precauciones, los cambios de escala, además de suponer una variación en las proporciones de los objetos, también implican una traslación de los mismos un efecto colateral no deseado en muchas ocasiones.

4.6.2.2. Traslación

La translación de un objeto consiste en moverlo cierta distancia, en una dirección determinada.

Para trasladar un objeto a la posición del vector de traslación t = (t_x, t_y, t_z), cada coordenada homogénea del punto p = (p_x, p_y, p_z, 1) tiene que ser multiplicada por su matriz de transformación de translación:

$$T_t = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

Como se muestra a continuación, la multiplicación da el resultado esperado:

$$p' = T_t \cdot p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} \cdot \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} t_x + p_x \\ t_y + p_y \\ t_z + p_z \\ 1 \end{bmatrix}$$

Las coordenadas del punto final *p'* serán p'= (t+p_x, t_y+p_y, t_z+p_y, *1)*.

Cabe recordar que para trasladar un objeto, al igual que en el escalado, se ha de aplicar la matriz T_t a todos los vértices del objeto. Es importante observar que al hacer la traslación de un objeto sus proporciones no varían, puesto que todos los vértices se mueven la misma distancia, en la misma dirección.

Para realizar la traslación inversa a la efectuada mediante la matriz T, se ha de aplicar la matriz inversa, es decir, la T^{-1}, que se obtiene cambiando el signo (multiplicando por −1) el vector de traslación. Por tanto,

$$T_t^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -t_x & -t_y & -t_z & 1 \end{bmatrix}$$

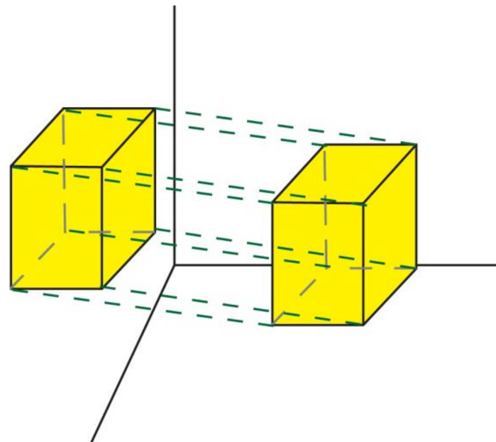

Figura 57: Traslación de un objeto en el espacio.

4.6.2.3. Rotación

Para generar una transformación de rotación, debemos designar un eje de rotación respecto del cual girará el objeto, y la cantidad de rotación angular, es decir, un ángulo (θ).

Una rotación tridimensional se puede especificar alrededor de cualquier línea en el espacio, aunque los ejes de rotación más fáciles de manejar son aquellos paralelos a los ejes de coordenadas.

Los ángulos de rotación positiva producen giros en el sentido opuesto a las manecillas del reloj con respecto al eje de una coordenada, si el observador se encuentra viendo a lo largo de la mitad positiva del eje hacia el origen de coordenadas.

La matriz de rotación respecto al eje x se especifica como

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

La matriz de rotación respecto al eje y se especifica como

$$R_y = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y la matriz de rotación respecto al eje z se especifica como

$$R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Para rotar un objeto cada coordenada homogénea del punto $p = (p_x, p_y, p_z, 1)$ tiene que ser multiplicada por su matriz de transformación proyectiva correspondiente.

Se forma una matriz de rotación inversa al sustituir el ángulo de rotación θ por –θ. Los valores negativos para los ángulos de rotación generan rotaciones en una dirección en el sentido del reloj.

Para comprender como se efectúa la rotación en estos ejes se puede visualizar la *figura 58*, donde se muestra un objeto en un sistema de referencia local (X', Y', Z'), cuyos ejes locales son paralelos a los ejes del sistema de coordenadas global (X, Y, Z).

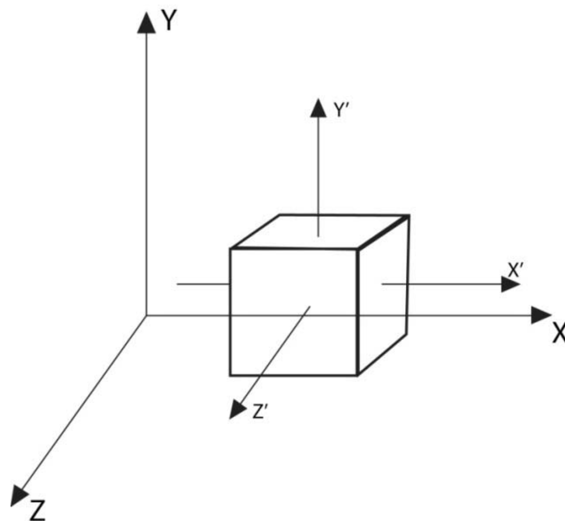

Figura 58: Rotación de un objeto alrededor de un eje paralelo a sistema de referencia global.

Para rotar un objeto en torno a su eje X local (X') se ha de girar sobre este eje un ángulo θ, utilizando la matriz de transformación correspondiente, en este caso R_x. Como se ha desarrollado anteriormente hay que multiplicar todos los puntos del objeto por la matriz. El resultado final del giro sobre el eje X' sería algo similar a lo mostrado en la *figura 59*.

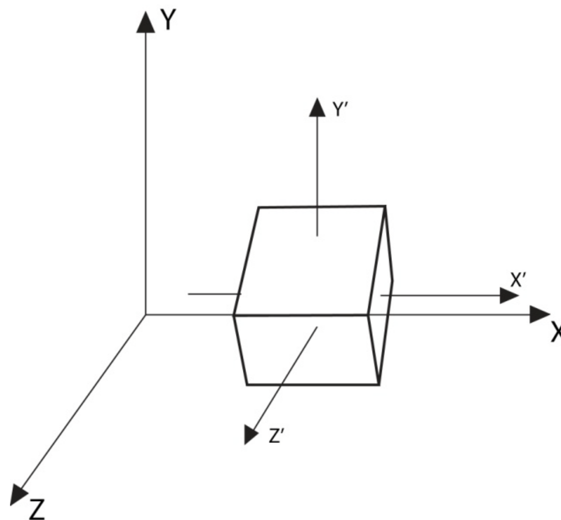

Figura 59: Objeto rotado alrededor de su eje X'.

4.6.2.4. Proyección Perspectiva

La transformación de proyección perspectiva se utiliza para producir un objeto realista. Cuando se ven las escenas en la vida cotidiana, los elementos lejanos aparecen pequeños en relación con los elementos cercanos. Un efecto secundario de la proyección en perspectiva es que las líneas paralelas parecen converger en un *punto de fuga*. Una característica importante de la perspectiva es que conserva las líneas rectas.

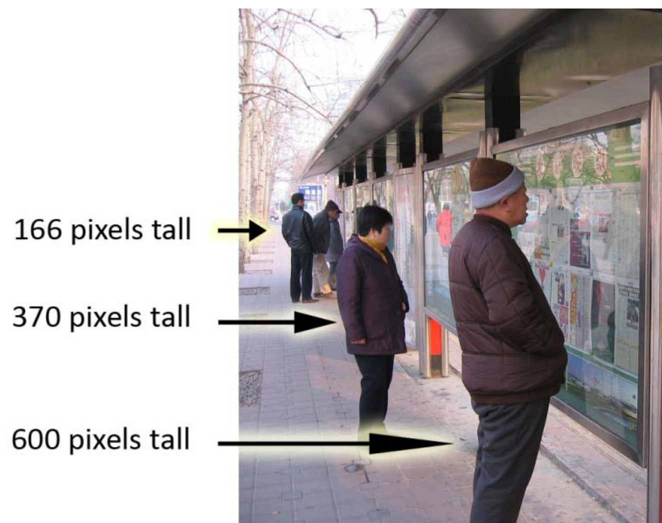

Figura 60: Los objetos del mundo real parecen más pequeños a medida que están más lejos.

Después de Durero[10] y los pintores del Renacimiento, *la proyección de perspectiva* se puede definir de la siguiente manera (ver figura 61):

> El *centro de la proyección* está en el origen *O* del sistema de referencia del espacio 3D (global). El plano de la imagen Π es paralelo al plano XY y desplazado una distancia *f* (*distancia focal*) a lo largo del eje Z desde el origen. Los puntos *P* tridimensionales proyectan hasta el plano Π el punto *p*. La proyección ortogonal de O en Π es el *punto principal O'*, y el eje Z que corresponde a esta línea de proyección es el *eje principal* (a veces llamado *eje óptico*, aunque no hay óptica en absoluto).

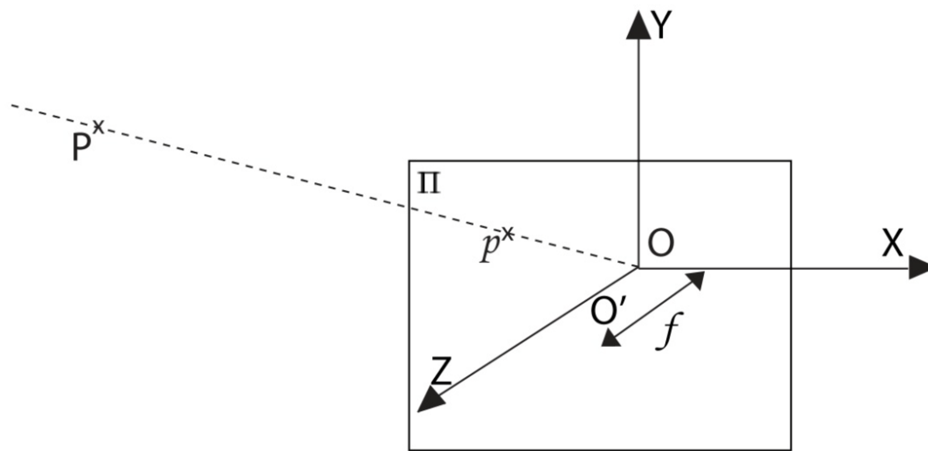

Figura 61: Proyección Perspectiva Estándar.

Como se puede ver en el ejemplo básico de la *Figura 62*, para realizar la transformación proyección perspectiva de un objeto, se aplica este proceso para cada uno de los puntos.

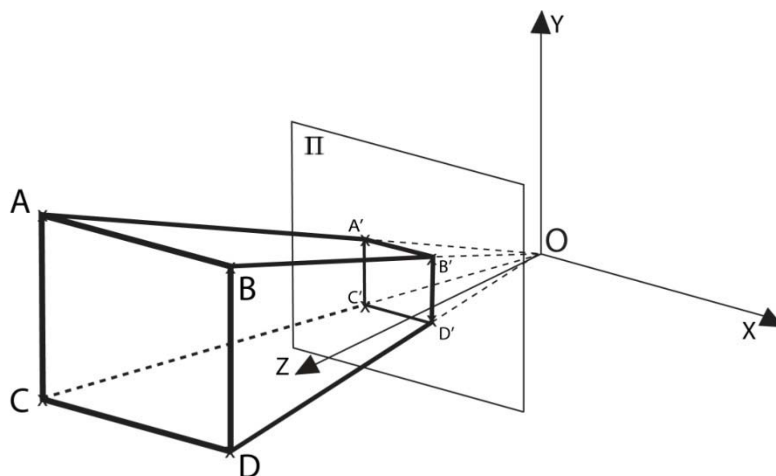

Figura 62: Proyección perspectiva de un objeto.
(Objeto con transformación de perspectiva)

[10] Mueller, Peter O. (1993). *Substantiv-Derivation in Den Schriften Albrecht Durers*.

En efecto, lo que se ha explicado es un ejemplo sencillo y válido, pero para generar esta transformación se requieren unos parámetros adicionales, como son el FOV (Campo de Visión), planos cercano y lejano.

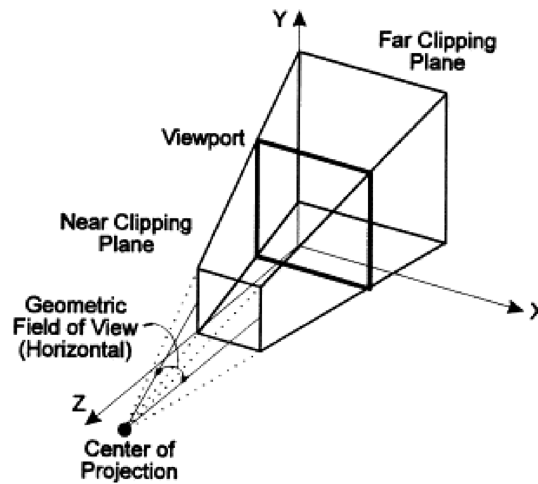

Figura 63: Transformación perspectiva de un objeto.

Para aplicar esta transformación cada coordenada homogénea del punto $p = (p_x, p_y, p_z, 1)$ tiene que ser multiplicada por su matriz de transformación proyectiva:

$$P = \begin{bmatrix} \frac{F}{aspect} & 0 & 0 & 0 \\ 0 & F & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2 \cdot f \cdot n}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Donde f es la distancia más lejana al plano Viewport, n es la distancia más cercana al plano Viewport, $aspect$ es la relación de aspecto de la imagen y $\mathcal{F}$ es la cotangente de FOV. Además existen dos planos importantes: el plano cercano y el plano lejano. El cercano es aquel a partir del cual los objetos que vemos están demasiado cerca para poder percibirlos en 3 dimensiones y el plano lejano es el plano a partir del cual todos los objetos que se encuentran a más alejados de él son percibidos sin efecto estereoscópico.

En la *figura 64* se puede observar un objeto antes y después de aplicarle la transformación de perspectiva.

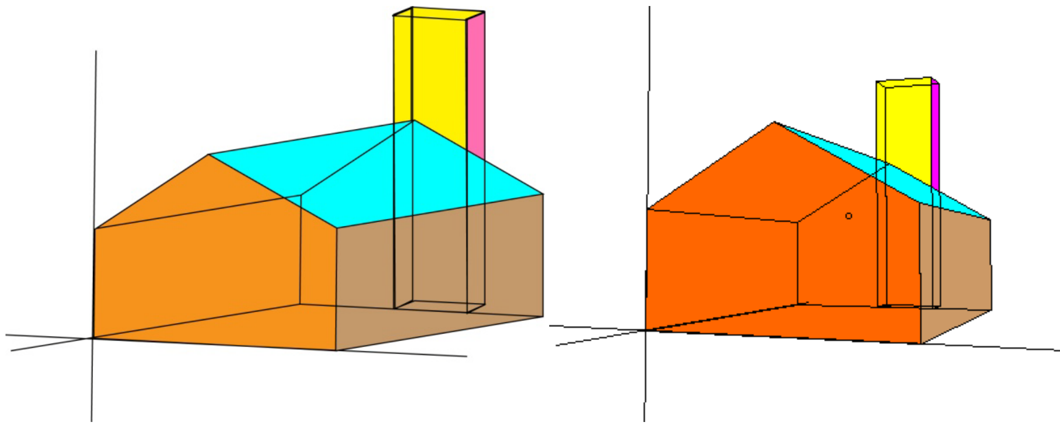

Figura 64: Objeto antes (izquierda) y después (derecha) de aplicar la transformación perspectiva.

4.6.3. Algoritmo de virtualización y renderizado

El algoritmo está dividido en dos partes, la virtualización y después el renderizado.

El primer paso que realiza el algoritmo es calcular las características del objeto real. La función *calculaPosicion* calcula la posición en el espacio 3D, es decir los valores de escalado, traslación y rotación que hay que aplicar para que el modelo virtualizado esté en la misma posición que el objeto.

```
[pitch_orig yaw_orig roll_orig]=calculaPosicion(vertices_figura,...
...figura);
```

La función *calculaPosicion* se desarrolla a continuación:

```
function [pitch yaw roll]=calculaPosicion(vertices_imagen,figura)
```

El algoritmo efectua operaciones diferentes según el tipo de figura con la que se trabaje.

```
if figura==1 % es cubo
```

Se calcula el tamaño del lado del objeto, L, utilizando el teorema de Pitágoras entre los vértices 1 y 3 del objeto. Además se obtiene la posición del centro (punto O en la *figura 65*) en la escena, para su posterior traslación.

```
L=sqrt((vertices_imagen(2,1)-vertices_imagen(1,1))^2+...
...(vertices_imagen(2,2)- vertices_imagen(1,2))^2);

object_position=mean(vertices_imagen);
object_position(2)=-object_position(2);
```

Se genera el modelo 3D desde el origen de coordenadas y tomando como origen el centro del cubo.

```
vertices = [
    -L/2, -L/2, -L/2;
    -L/2,  L/2, -L/2;
    L/2, -L/2, -L/2;
    L/2,  L/2, -L/2;
    L/2, -L/2,  L/2;
    L/2,  L/2,  L/2;
    -L/2, -L/2,  L/2;
    -L/2,  L/2,  L/2];
```

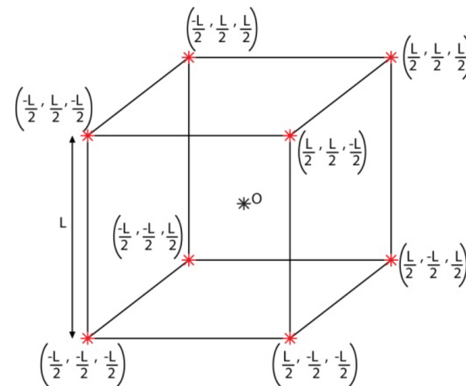

Figura 65: Coordenadas 3D del cubo.

```
vertices_coord=[vertices ones(size(vertices,1),1)];
```

Se genera y se aplica a los vértices ya creados la matriz de translación, esto es desplazar el modelo 3D generado a la posición calculada.

```
translation_matrix = [
1 0 0 0;
0 1 0 0;
0 0 1 0;
object_position 0 1];

vertices_coord=vertices_coord*translation_matrix;
```

Se genera y se aplica a los vértices ya trasladados la matriz de escalado, en este caso la figura quedará al mismo tamaño debido a que el factor de escalado es 1.

```
scaling_factor=1;
scaling_matrix = [
    scaling_factor 0 0 0;
    0 scaling_factor 0 0;
    0 0 scaling_factor 0;
    0 0 0 1];

vertices_coord=vertices_coord*scaling_matrix;
```

A continuación se realizan los cálculos de la rotación que se ha de aplicar. Estos cálculos son diferentes según en que posición se encuentre el vértice trasero del objeto respecto de la arista vertical central. Por tanto si el vértice superior trasero se encuentra a la izquierda de la arista vertical central se realizan unos cálculos y si se encuentra a la derecha se realizan otros diferentes. Asimismo, también se encuentran casos concretos en caso de que las aristas inferiores del objeto sean practicamente horizontales. Por lo general el procedimiento que se sigue es el siguiente:

- Se calcula la rotación del eje vertical (yaw - α) utilizando las distancias A y B si el vértice trasero está a la derecha, y las distancias F y E si el vértice esta a la izquierda. Esa rotación es:

α=-2*α' si el vértice trasero está a la derecha, siendo $\alpha' = \tan^{-1}\left(\frac{A}{B}\right)$

o α=-α' si el vértice trasero está a la izquierda, siendo $\alpha' = \tan^{-1}\left(\frac{F}{E}\right)$

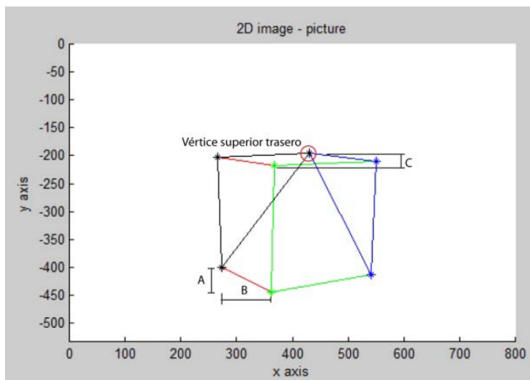

Figura 66: Objeto y distancias para calcular α y β.

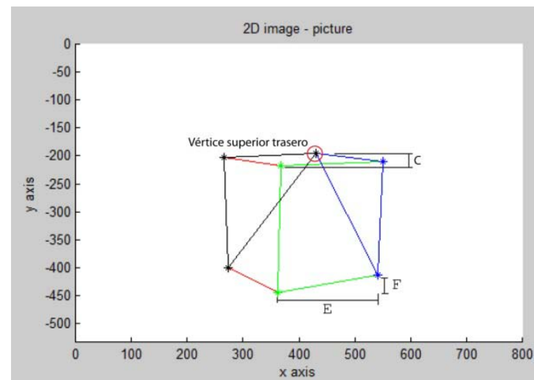

Figura 67: Objeto y distancias para calcular α y β.

- Una vez aplicada esa rotación se ha de calcular la inclinación, eje Y (pitch - β), y a partir de ésta obtener la rotación del eje Z (roll - μ):

β=2β' y μ=-β si el vértice trasero está a la derecha o β=β' y μ=-β/2 si está a la izquierda, siendo $\beta' = \sin^{-1}\left(\frac{C}{D}\right)$

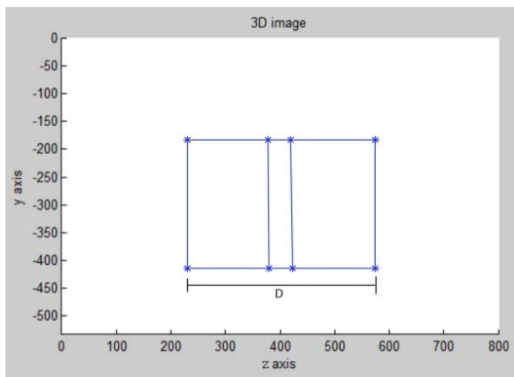

Figura 68: Cubo 3D y distancia para calcular β.

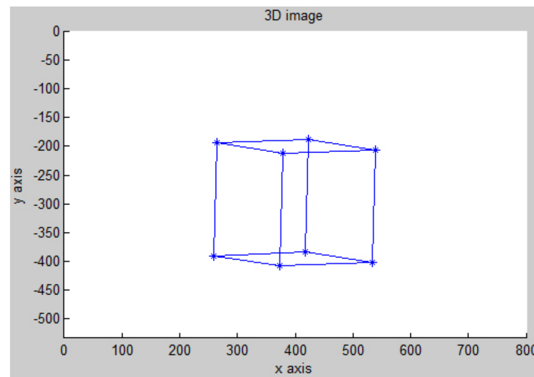

Figura 69: Cubo 3D con la rotación α, β y μ.

```
if vertices_imagen(4,1)<vertices_imagen(7,1)

    B=abs(vertices_imagen(1,1)-vertices_imagen(3,1));
    A=abs(vertices_imagen(1,2)-vertices_imagen(3,2));
    yaw_in=atan(A/B);
```

```
yaw=-2*yaw_in;

if (0.1745<yaw_in && yaw_in<1.0472)
    pitch = 0*pi/180;
    roll =  0*pi/180;

    rotation_matrix_pitch = [
        1 0 0 0;
        0 cos(pitch) sin(pitch) 0;
        0 -sin(pitch) cos(pitch) 0;
        0 0 0 1];

    rotation_matrix_yaw = [
        cos(yaw) 0 -sin(yaw) 0;
        0 1 0 0;
        sin(yaw) 0 cos(yaw) 0;
        0 0 0 1];

    rotation_matrix_roll = [
        cos(roll) sin(roll)  0 0;
        -sin(roll) cos(roll) 0 0;
        0 0 1 0;
        0 0 0 1];

    vertices_coord=vertices_coord*rotation_matrix_pitch*...
 ...rotation_matrix_yaw*rotation_matrix_roll;

    min_x=min(vertices_coord(:,1));
    max_x=max(vertices_coord(:,1));

    D=abs(max_x-min_x);
    C=abs(vertices_imagen(7,2)-vertices_imagen(4,2));

    pitch=asin(C/D);
    pitch=2*pitch;

    roll=-pitch;

end
if yaw_in>1.0472
    B=abs(vertices_imagen(3,1)-vertices_imagen(5,1));
    A=abs(vertices_imagen(3,2)-vertices_imagen(5,2));

    yaw=atan(A/B);
    yaw=2*yaw;
    pitch = 0*pi/180;
    roll =  0*pi/180;

    rotation_matrix_pitch = [
        1 0 0 0;
        0 cos(pitch) sin(pitch) 0;
        0 -sin(pitch) cos(pitch) 0;
        0 0 0 1];

    rotation_matrix_yaw = [
        cos(yaw) 0 -sin(yaw) 0;
        0 1 0 0;
        sin(yaw) 0 cos(yaw) 0;
        0 0 0 1];
```

```
        rotation_matrix_roll = [
            cos(roll) sin(roll)  0 0;
            -sin(roll) cos(roll) 0 0;
            0 0 1 0;
            0 0 0 1];

        vertices_coord=vertices_coord*rotation_matrix_pitch*...
     ...rotation_matrix_yaw*rotation_matrix_roll;
        min_x=min(vertices_coord(:,1));
        max_x=max(vertices_coord(:,1));
        D=abs(max_x-min_x);
        C=abs(vertices_imagen(7,2)-vertices_imagen(4,2));
        pitch=asin(C/D);
        roll=0;
    end
    if yaw_in<0.1745 % si yaw_in<10º
        B=abs(vertices_imagen(3,1)-vertices_imagen(6,1));
        A=abs(vertices_imagen(3,2)-vertices_imagen(6,2));
        yaw=atan(A/B);
        yaw=-yaw;
        pitch = 0*pi/180;
        roll =  0*pi/180;

        rotation_matrix_pitch = [
            1 0 0 0;
            0 cos(pitch) sin(pitch) 0;
            0 -sin(pitch) cos(pitch) 0;
            0 0 0 1];

        rotation_matrix_yaw = [
            cos(yaw) 0 -sin(yaw) 0;
            0 1 0 0;
            sin(yaw) 0 cos(yaw) 0;
            0 0 0 1];

        rotation_matrix_roll = [
            cos(roll) sin(roll)  0 0;
            -sin(roll) cos(roll) 0 0;
            0 0 1 0;
            0 0 0 1];

        vertices_coord=vertices_coord*rotation_matrix_pitch*...
     ...rotation_matrix_yaw*rotation_matrix_roll;
        min_x=min(vertices_coord(:,1));
        max_x=max(vertices_coord(:,1));
        D=abs(max_x-min_x);
        C=abs(vertices_imagen(7,2)-vertices_imagen(4,2));
        pitch=asin(C/D);
        roll=-pitch;
    end

end
if vertices_imagen(4,1)>vertices_imagen(7,1)
    B=abs(vertices_imagen(3,1)-vertices_imagen(5,1));
    A=abs(vertices_imagen(3,2)-vertices_imagen(5,2));
    yaw_in=atan(A/B)
    if 0.5236<yaw_in
        yaw=-yaw_in;
        pitch = 0*pi/180;
```

```
    roll =  0*pi/180;

    rotation_matrix_pitch = [
        1 0 0 0;
        0 cos(pitch) sin(pitch) 0;
        0 -sin(pitch) cos(pitch) 0;
        0 0 0 1];

    rotation_matrix_yaw = [
        cos(yaw) 0 -sin(yaw) 0;
        0 1 0 0;
        sin(yaw) 0 cos(yaw) 0;
        0 0 0 1];

    rotation_matrix_roll = [
        cos(roll) sin(roll)  0 0;
        -sin(roll) cos(roll) 0 0;
        0 0 1 0;
        0 0 0 1];

    vertices_coord=vertices_coord*rotation_matrix_pitch*...
 ...rotation_matrix_yaw*rotation_matrix_roll;
    min_x=min(vertices_coord(:,1));
    max_x=max(vertices_coord(:,1));
    D=abs(max_x-min_x);
    C=abs(vertices_imagen(7,2)-vertices_imagen(4,2));
    pitch=asin(C/D);
    roll=-pitch/2;
end
if (0.2618<yaw_in && yaw_in<0.5236)
    B=abs(vertices_imagen(3,1)-vertices_imagen(1,1));
    A=abs(vertices_imagen(3,2)-vertices_imagen(1,2));
    yaw=atan(A/B);
    yaw=-2*yaw
    pitch = 0*pi/180;
    roll =  0*pi/180;

    rotation_matrix_pitch = [
        1 0 0 0;
        0 cos(pitch) sin(pitch) 0;
        0 -sin(pitch) cos(pitch) 0;
        0 0 0 1];

    rotation_matrix_yaw = [
        cos(yaw) 0 -sin(yaw) 0;
        0 1 0 0;
        sin(yaw) 0 cos(yaw) 0;
        0 0 0 1];

    rotation_matrix_roll = [
        cos(roll) sin(roll)  0 0;
        -sin(roll) cos(roll) 0 0;
        0 0 1 0;
        0 0 0 1];

    vertices_coord=vertices_coord*rotation_matrix_pitch*...
 ...rotation_matrix_yaw*rotation_matrix_roll;
    min_x=min(vertices_coord(:,1));
    max_x=max(vertices_coord(:,1));
    D=abs(max_x-min_x);
```

```
            C=abs(vertices_imagen(7,2)-vertices_imagen(4,2));
            pitch=asin(C/D)
            roll=-pitch/2
        end

        if yaw_in<=0.2618

            H=(abs(vertices_imagen(7,1)-vertices_imagen(4,1)))/100;
            yaw=45-H;
            yaw=-deg2rad(yaw);
            pitch = 0*pi/180;
            roll =  0*pi/180;

            rotation_matrix_pitch = [
                1 0 0 0;
                0 cos(pitch) sin(pitch) 0;
                0 -sin(pitch) cos(pitch) 0;
                0 0 0 1];

            rotation_matrix_yaw = [
                cos(yaw) 0 -sin(yaw) 0;
                0 1 0 0;
                sin(yaw) 0 cos(yaw) 0;
                0 0 0 1];

            rotation_matrix_roll = [
                cos(roll) sin(roll)  0 0;
                -sin(roll) cos(roll) 0 0;
                0 0 1 0;
                0 0 0 1];

            vertices_coord=vertices_coord*rotation_matrix_pitch*...
         ...rotation_matrix_yaw*rotation_matrix_roll;
            min_x=min(vertices_coord(:,1));
            max_x=max(vertices_coord(:,1));
            D=abs(max_x-min_x);
            C=abs(vertices_imagen(7,2)-vertices_imagen(4,2));
            pitch=asin(C/D);
            roll=-pitch/2;
        end
    end

end
```

A continuación se sigue el mismo prodecimiento para realizar los cálculos en caso de que el objeto sea una pirámide:

Se calcula el tamaño del lado del objeto, L, utilizando el teorema de Pitágoras entre los vértices 1 y 2 del objeto. Además se obtiene la posición del centro (punto O en la *figura 70*) en la escena para su posterior traslación, y la altura de la pirámide, H, como la distancia entre las coordenadas Y del vértice 1 y 2.

```
  L=sqrt((vertices_imagen(2,1)-vertices_imagen(1,1))^2+...
...(vertices_imagen(2,2)-vertices_imagen(1,2))^2)
```

```
H= vertices_imagen(2,2)-vertices_imagen(1,2);
H=-H;
object_position = mean(vertices_imagen)
object_position(2)=-object_position(2);
```

Se genera el modelo 3D desde el origen de coordenadas y tomando como origen el centro de la pirámide.

```
vertices = [
      -L/2, -H/2, -L/2;
         0,  H/2,  0;
       L/2, -H/2,  L/2;
       L/2, -H/2, -L/2;
      -L/2, -H/2,  L/2];
```

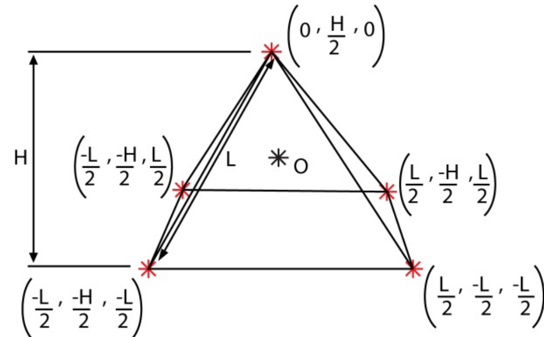

Figura 70: Coordenadas 3D de la pirámide.

```
vertices_coord=[vertices ones(size(vertices,1),1)];
```

Se genera y se aplica a los vértices ya creados la matriz de translación, esto es desplazar el modelo 3D generado a la posición calculada.

```
  translation_matrix = [
  1 0 0 0;
  0 1 0 0;
  0 0 1 0;
  object_position 0 1];

vertices_coord=vertices_coord*translation_matrix;
```

Se genera y se aplica a los vértices ya trasladados la matriz de escalado, en este caso la figura quedará al mismo tamaño debido a que el factor de escalado es 1.

```
scaling_factor=1;
scaling_matrix = [
    scaling_factor 0 0 0;
    0 scaling_factor 0 0;
    0 0 scaling_factor 0;
    0 0 0 1];

vertices_coord=vertices_coord*scaling_matrix;
```

A continuación se realizan los cálculos de la rotación que se ha de aplicar. Estos cálculos son diferentes según en que posición se encuentre el objeto. Por lo general el procedimiento que se sigue es el siguiente:

Se realiza siempre el cálculo de la rotación del eje vertical (yaw - α) a partir de las medidas E y F, siendo $\alpha' = \tan^{-1}\left(\frac{E}{F}\right)$.

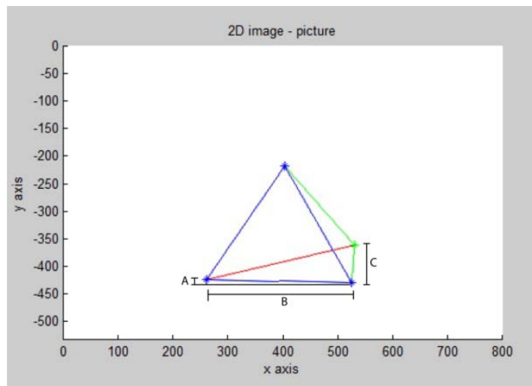

Figura 71: Objeto y distancias para calcular α y β.

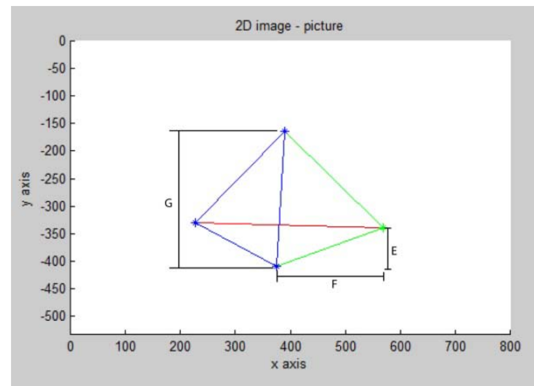

Figura 72: Objeto y distancias para calcular α y β.

A partir de este cálculo trabaja el algoritmo de una forma u otra:

- Si 10º < α' < 80º , entonces: α=2*α'
- Si α' > 80º, el algoritmo recalcula α', siendo $\alpha' = \tan^{-1}\left(\frac{A}{B}\right)$, entonces: α=-α'.

Una vez aplicada esa rotación se ha de calcular la inclinación, eje Y (pitch - β), y a partir de ésta obtener la rotación del eje Z (roll - μ):

- Si α=2*α', entonces β=β'/2 y μ=β/2, siendo $\beta' = \sin^{-1}\left(\frac{G}{D}\right)$.
- Si α=-α', entonces β=β' y μ=0º, siendo $\beta' = \sin^{-1}\left(\frac{C}{D}\right)$.

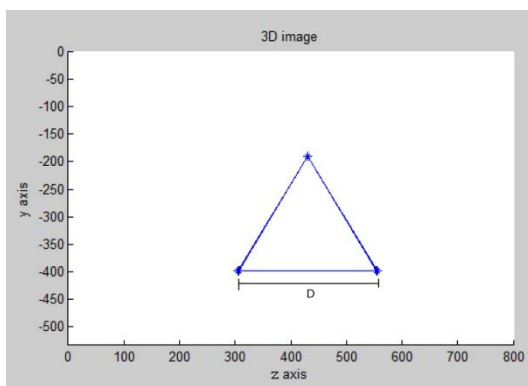

Figura 73: Pirámide y distancia para calcular β.

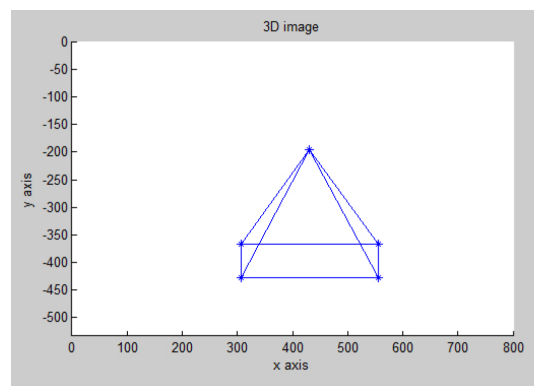

Figura 74: Pirámide 3D con la rotación α, β y μ.

```
if figura==0 % es pirámide

            B=abs(vertices_imagen(4,1)-vertices_imagen(3,1));
            A=abs(vertices_imagen(4,2)-vertices_imagen(3,2));
            yaw_in=atan(A/B);

            if 0.1745<yaw_in<1.3963
                yaw=2*yaw_in;
```

```
                pitch = 0*pi/180;
                roll =  0*pi/180;

                rotation_matrix_pitch = [
                    1 0 0 0;
                    0 cos(pitch) sin(pitch) 0;
                    0 -sin(pitch) cos(pitch) 0;
                    0 0 0 1];

                rotation_matrix_yaw = [
                    cos(yaw) 0 -sin(yaw) 0;
                    0 1 0 0;
                    sin(yaw) 0 cos(yaw) 0;
                    0 0 0 1];

                rotation_matrix_roll = [
                    cos(roll) sin(roll)  0 0;
                    -sin(roll) cos(roll) 0 0;
                    0 0 1 0;
                    0 0 0 1];


vertices_coord=vertices_coord*rotation_matrix_pitch*...
...rotation_matrix_yaw*rotation_matrix_roll;
                min_x=min(vertices_coord(:,1));
                max_x=max(vertices_coord(:,1));
                D=abs(max_x-min_x);
                C=abs(vertices_imagen(3,2)-vertices_imagen(2,2));
                pitch=asin(C/D);
                pitch=pitch/2;
                roll=pitch/2;

            end
            if yaw_in>=1.3963
                B=abs(vertices_imagen(3,1)-vertices_imagen(1,1));
                A=abs(vertices_imagen(3,2)-vertices_imagen(1,2));
                yaw=-atan(A/B);
                pitch = 0*pi/180;
                roll =  0*pi/180;

                rotation_matrix_pitch = [
                    1 0 0 0;
                    0 cos(pitch) sin(pitch) 0;
                    0 -sin(pitch) cos(pitch) 0;
                    0 0 0 1];

                rotation_matrix_yaw = [
                    cos(yaw) 0 -sin(yaw) 0;
                    0 1 0 0;
                    sin(yaw) 0 cos(yaw) 0;
                    0 0 0 1];

                rotation_matrix_roll = [
                    cos(roll) sin(roll)  0 0;
                    -sin(roll) cos(roll) 0 0;
                    0 0 1 0;
                    0 0 0 1];

            vertices_coord=vertices_coord*rotation_matrix_pitch*...
         ...rotation_matrix_yaw*rotation_matrix_roll;
```

```
                min_x=min(vertices_coord(:,1));
                max_x=max(vertices_coord(:,1));
                D=abs(max_x-min_x);
                C=abs(vertices_imagen(4,2)-vertices_imagen(3,2));
                pitch=asin(C/D);
                roll=0;
            end
end
```

También se obtiene mediante la función *detectaColor* su color.

```
[NivR NivG NivB]=detectaColor(im_rgb,gradient_imageB);
```

Esta función descompone la imagen en sus tres componentes, después crea una máscara del objeto para eliminar el resto de la imagen que no sea objeto y obtiene el valor de cada componente de un pixel del centro del objeto. A co0ntinuación se desarrolla la función:

```
function [NivelR NivelG NivelB]=detectaColor(imagen_in,gradiente)

 imagen_in=double(imagen_in)/255;

 R = imagen_in(:,:,1);
 G = imagen_in(:,:,2);
 B = imagen_in(:,:,3);

 mascara=imfill(gradiente);

 R_mask=R.*mascara;
 posR=find(R_mask~=0);
 G_mask=G.*mascara;
 posG=find(G_mask~=0);
 B_mask=B.*mascara;
 posB=find(B_mask~=0);

 NivelR=R_mask(posR(floor(length(posR)/2)));
 NivelG=G_mask(posG(floor(length(posG)/2)));
 NivelB=B_mask(posB(floor(length(posB)/2)));
```

Una vez obtenido el color del objeto mediante la función `detectaColor` se calcula el color complementario para dibujar las aristas.

```
Niv_r=1-NivR;
Niv_g=1-NivG;
Niv_b=1-NivB;
```

Posteriormente se genera el modelo virtual utilizando la función *generar_figura_virtual* de donde se obtienen las coordenadas tridimensionales para su renderizado.

```
vertices_virtuales=generar_figura_virtual(vertices_figura,figura...
...pitch_orig,yaw_orig,roll_orig);
```

El efecto de la virtualización se puede percibir a continuación:

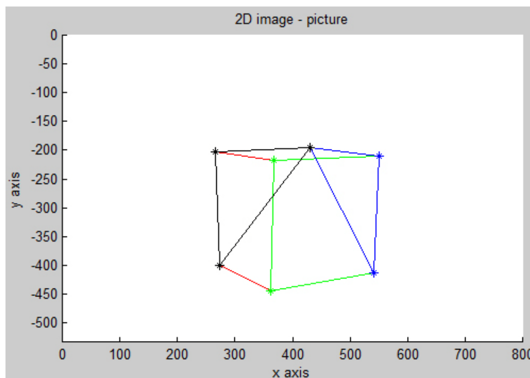

Figura 75: Renderizado 2D del cubo.

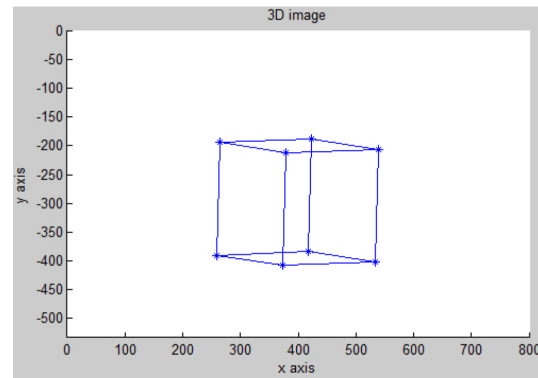

Figura 76: Virtualización 3D del cubo.

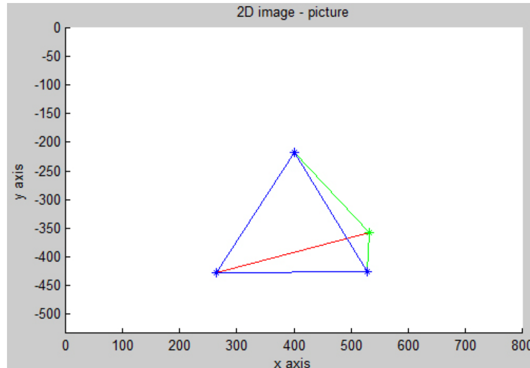

Figura 77: Renderizado 2D de la pirámide.

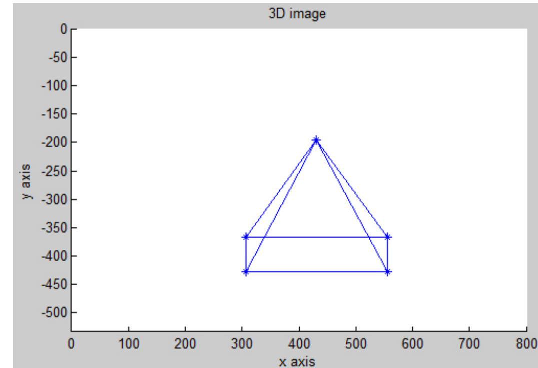

Figura 78: Virtualización 3D de la pirámide.

La *figura 75* y *figura 77* muestran la virtualización del objeto real de la *figura 76* y *figura 78* respectivamente. Se puede apreciar que el objeto virtualizado no es exactamente igual al objeto real. Esto es debido a que no se ha implementado la transformación de perspectiva debido a su complejidad ya que se está trabajando en la virtualización 3D a partir de vista única y el cálculo de esta transformación es muy complejo.

El segundo paso es la *renderización*. Esto consiste en representar en 2D el modelo 3D, por tanto se obtienen las coordenadas 2D del modelo a partir de las coordenadas 3D. Basta con quedarse con las coordenadas X e Y y despreciar la coordenada Z para tener un objeto bidimensional. Consecutivamente se utiliza la función *rellenaFiguraVirtual2* para detectar que vértices y que aristas son las ocultas y representar la figura correctamente.

```
vertices_figura_final=[];

vertices_figura_final(:,1)=vertices_virtuales(:,1);
vertices_figura_final(:,2)=(-1)*vertices_virtuales(:,2);



 vertice_oculto=rellenaFiguraVirtual2(vertices_virtuales,figura,...
...Niv_r,Niv_g,Niv_b,NivR,NivG,NivB)
```

En las siguientes imágenes se puede ver el resultado final del algoritmo a la vez que se compara con la imagen original:

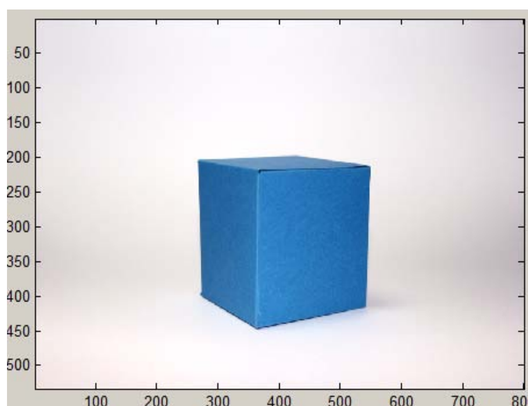

Figura 79: Imagen original.

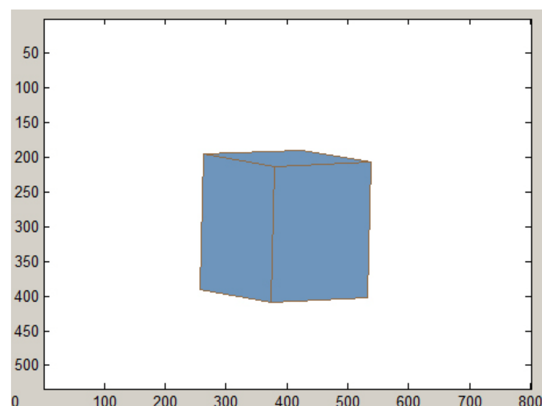

Figura 80: Modelo 3D del objeto original.

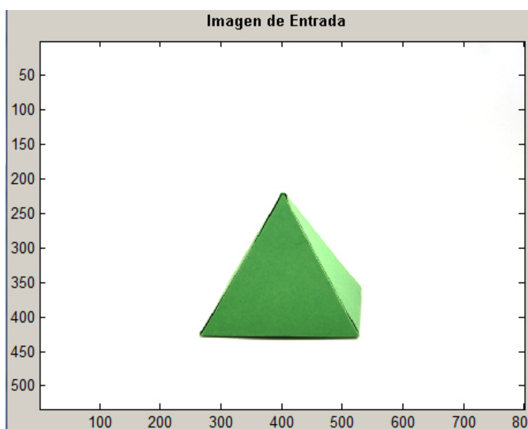

Figura 81: Imagen original.

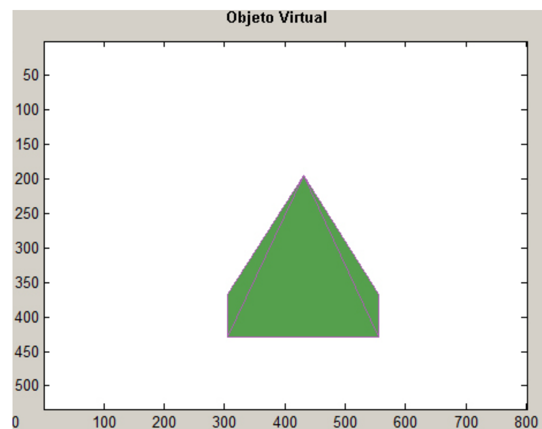

Figura 82: Modelo 3D del objeto original.

4.7. Fase 6. Deslizadores

4.7.1. Definición

Los deslizadores permiten interactuar en tiempo real con el objeto 3D rotándolo en sus tres direcciones del espacio, de forma que no es necesario cambiar cada vez el valor de las rotaciones sino que el deslizador realiza esa función automáticamente cuando se desplaza.

Es por ello que cada deslizador tiene su propio algoritmo. Lo que hace el algoritmo de los deslizadores es repetir la fase anterior de virtualización y renderizado con la finalidad de volver a dibujar el modelo 3D con los nuevos datos de rotación que adquieren los deslizadores independientemente uno de otro.

4.7.2. Algoritmo de los deslizadores

Se obtienen los valores de los deslizadores para posteriormente realizar la rotación.

```
pitch_current_deg=get(slider_pitch,'Value');

yaw_current_deg=get(slider_yaw,'Value');

roll_current_deg=get(slider_roll,'Value');
```

Se genera la figura virtual con los valores calulados.

```
vertices_virtuales=generar_figura_virtual(vertices_figura_final,...
...figura,pitch_current,yaw_current,roll_current);
```

Se desvirtualiza y se obtienen las coordenadas 2D de la figura para posteriormente representarla bidimensionalmente.

```
vertices_figura_final(:,1)=vertices_virtuales(:,1);
vertices_figura_final(:,2)=(-1)*vertices_virtuales(:,2);


vertice_oculto=rellenaFiguraVirtual2(vertices_virtuales,figura,...
...Niv_r,Niv_g,Niv_b,NivR,NivG,NivB)
```

Este código fuente aparece en el algoritmo final repetido tres veces, una vez por cada uno de los deslizadores que contiene el programa. De esta manera cada vez que un

deslizador sea desplazado el algoritmo vuelve a representar el modelo 3D con la nueva rotación.

4.8. Fase 7. Generación del par estéreo de imágenes

4.8.1. Definición

En la virtualización a partir de varias vistas es propio el uso de pares estéreo de imágenes, de esta forma es más fácil definir un objeto con una sola imagen.

Como en este trabajo se trabaja con virtualización a partir de vista única se puede generar el par estéreo de imágenes a partir del modelo tridimensional de una forma bastante aproximada. Si para capturar el par de imágenes estéreo con una cámara se tiene que desplazar o girar la cámara un ángulo diferente para cada imagen del par, este algoritmo realiza ese procedimiento matemáticamente con una diferencia, en vez de rotar la cámara se rota el objeto y el efecto será el mismo. Se obtendrán dos imágenes ligeramente rotadas.

4.8.2. Creación del par estéreo de imágenes

Para la creación del par estéreo se hace una aproximación al sistema visual humano. El sistema visual humano es capaz de ver en tres dimensiones principalmente porque posee visión binocular.

La visión binocular tiene lugar porque los dos ojos (separados unos centímetros) miran al mismo objeto desde ángulos ligeramente distintos, obteniendo como resultado dos imágenes muy parecidas, pero no iguales.

Muchos animales, como por ejemplo los conejos, peces y pájaros, tienen los ojos a cada lado de la cabeza, enfocando hacia direcciones opuestas. Estos animales pueden ver todo su entorno casi sin mover la cabeza, pero no pueden percibir dos imágenes parecidas del mismo objeto. Es por este motivo que no pueden disfrutar de la variedad de formas en relieve que nos proporciona la visión binocular a los humanos.

Disparidad binocular

La disparidad binocular o retinal es la ligera diferencia entre los dos puntos de vista proporcionados por ambos ojos. La disparidad binocular es la forma más utilizada

de percibir profundidad y relieve por el cerebro humano, y es la que permite ser más manipulada, convirtiéndose en la base para la creación de imágenes 3D en superficies planas. El cerebro coge estos dos puntos de vista distintos y los integra, creando así un objeto en tres dimensiones.

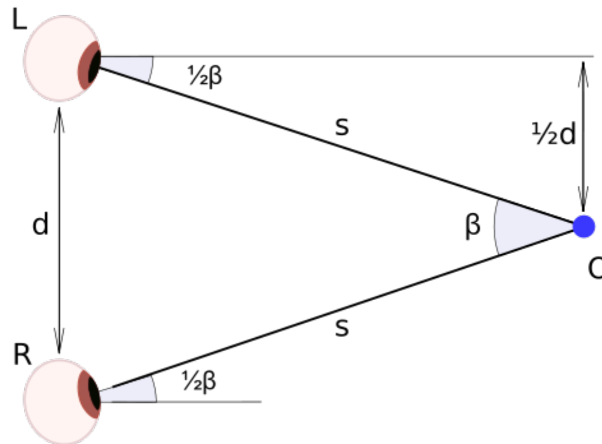

Figura 83: Disparidad Binocular. Muestra los ojos, objeto, distancias y ángulos.

Distancia interocular

La distancia interocular o interpupilar es, como su propio nombre indica, la distancia que separa los centros de rotación de los ojos. También se puede utilizar este nombre para hablar de la distancia que hay entre los objetivos de dos cámaras. En el caso del ser humano la distancia que se toma como estándar es de 65 milímetros.

Una vez se ha obtenido el modelo 3D y haciendo uso de los conceptos de disparidad binocular y distancia interocular se puede obtener el par estéreo de imágenes.

Para comprender como se efectúan los cálculos y los datos que se utilizan se puede visualizar la *figura 83*, el modelo 3D será rotado respecto a su eje vertical un ángulo equivalente al ángulo de visión del ojo humano $\frac{1}{2}\beta$. Se aplicará una rotación positiva de $\frac{1}{2}\beta$ para obtener la imagen del ojo izquierdo y una rotación negativa de $-\frac{1}{2}\beta$ para obtener la imagen del ojo derecho. Es decir, se ha de calcular la mitad del ángulo existente entre los ojos cuando observan el objeto (β).

Para realizar el cálculo se ha de conocer la distancia *S,* distancia entre el objeto y el observador (una persona o un par de cámaras como es el caso) y la distancia interocular, *d*.

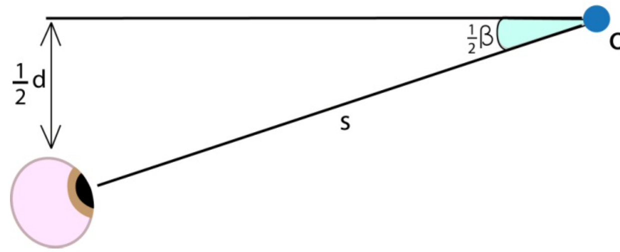

Figura 84: Trigonometría para calcular el ángulo

Y por trigonometría,

$$\sin\left(\frac{1}{2}\cdot\beta\right) = \frac{{}^{1}\!/_{2}\cdot d}{S}$$

De donde,

$$\frac{1}{2}\cdot\beta = \sin^{-1}\left(\frac{{}^{1}\!/_{2}\cdot d}{S}\right)$$

Hay que destacar que, como puede verse reflejado en la *figura 85*, el ánguloβ será diferente según sea la distancia entre el objeto y el observador. Cuanto más cerca esté el objeto del observador mayor será el ángulo de visión entre los dos ojos.

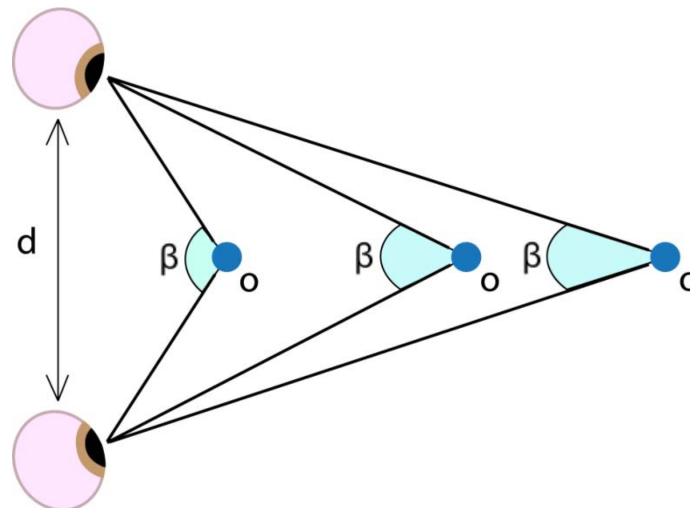

Figura 85: Relación distancia objeto-observador y ángulo entre ojos.

4.8.3. Algoritmo de generación del par estéreo de imágenes

El proceso que realiza el algoritmo está basado en la visión humana, por lo que se utilizan parámetros de la visión humana como es la *distancia interocular* que se establece en 65mm (0.065m). Debido a que las distancias deben ser distancias de píxeles porque se está trabajando con imágenes, es de utilidad conocer la resolución de la imagen así como el factor de conversión entre pixeles y la unidad métrica. También es necesario el conocimiento de la distancia entre la cámara y el objeto.

Así, el primer paso es la conversión de las distancias a píxeles y el cálculo del ángulo a utilizar:

```
info_imagen=imfinfo(FILENAME);
pixperinch=info_imagen.XResolution;


distInterocular=0.065; %en metros
inch2cm=0.0254; % equivalencia de una pulgada en metros.

distInterocular_inch=distInterocular/inch2cm;
distInterocular_pix=distInterocular_inch*pixperinch;

distCam_Objeto=abs(im_heigth-vertices_figura(3,2));


angle=atan((distInterocular_pix/2)/distCam_Objeto);
```

A continuación se calcula el ángulo que hay que rotar el objeto respecto del eje vertical, este ángulo es el ángulo actual que tiene el objeto además del ángulo que se ha calculado, el cual es negativo para una imagen del par y positivo para la otra. También se comprueba que el ángulo obtenido no quede fuera del rango [-90,90], en caso de que quede fuera se calcula su complementario que queda dentro y ese será el giro a aplicar:

```
giroDerecho=yaw_current-angle;


if giroDerecho<-1.5708
    giroDerecho=1.5708-(abs(giroDerecho)-1.5708)
elseif giroDerecho>1.5708
    giroDerecho=-1.5708+(abs(giroDerecho)-1.5708)
end
```

```
giroIzquierdo= yaw_current+angle;

if giroIzquierdo<-1.5708
    giroIzquierdo=1.5708-(abs(giroIzquierdo)-1.5708)
elseif giroIzquierdo>1.5708
    giroIzquierdo=-1.5708+(abs(giroIzquierdo)-1.5708)
end
```

Finalmente se obtiene el par estéreo de imágenes, el resultado se puede observar en las siguientes imágenes:

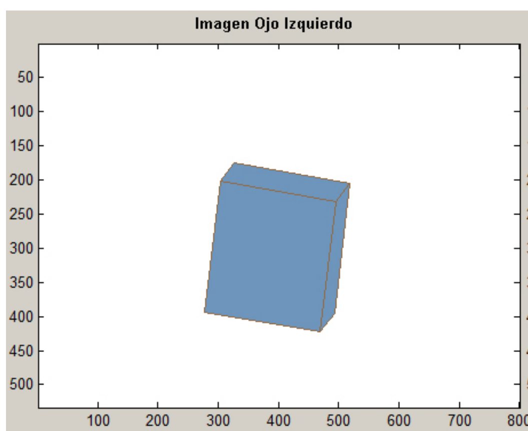

Figura 86: Imagen del ojo Izquierdo de cubo.

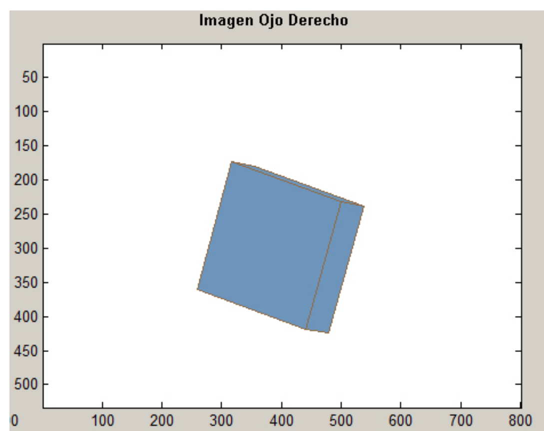

Figura 87: Imagen del ojo Derecho de cubo.

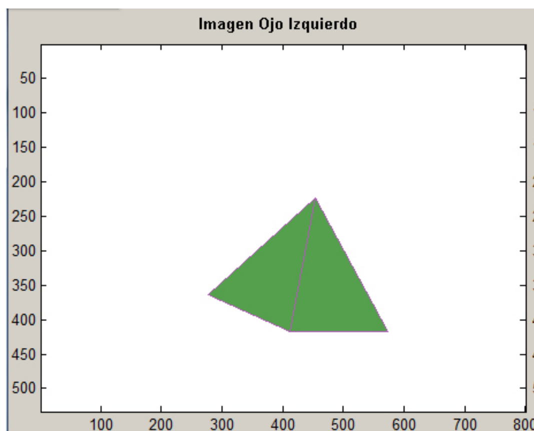

Figura 88: Imagen del ojo Izquierdo de pirámide.

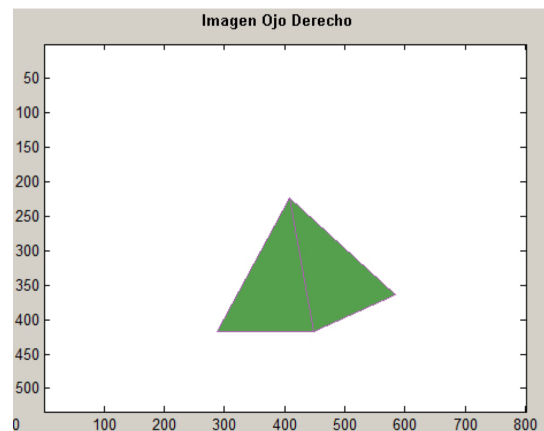

Figura 89: Imagen del ojo Derecho de Pirámide

5. FUNCIONAMIENTO DE LA INTERFAZ GRÁFICA

5.1. Introducción

Con la finalidad de realizar todo el proceso de forma automática y transparente para el usuario y que éste pudiera interactuar con el programa de una forma más sencilla, se decide crear una interfaz gráfica utilizando GUIDE (Graphic User Interface) en Matlab.

GUIDE es un entorno de programación visual disponible en Matlab para realizar y ejecutar programas que necesiten ingreso continuo de datos. Tiene las características básicas de todos los programas visuales como Visual Basic o Visual C++.

Esta sección se ha escrito como manual de usuario para la aplicación guide; cubre todas las opciones posibles con las que el usuario puede interactuar, explicando todos los parámetros que pueden ser introducidos para las diferentes opciones, así como todos los mensajes y botones.

5.2. Software *GUIVirtual3D* (Manual de Usuario)

La *figura 90* es una impresión de pantalla de la ventana principal de la aplicación:

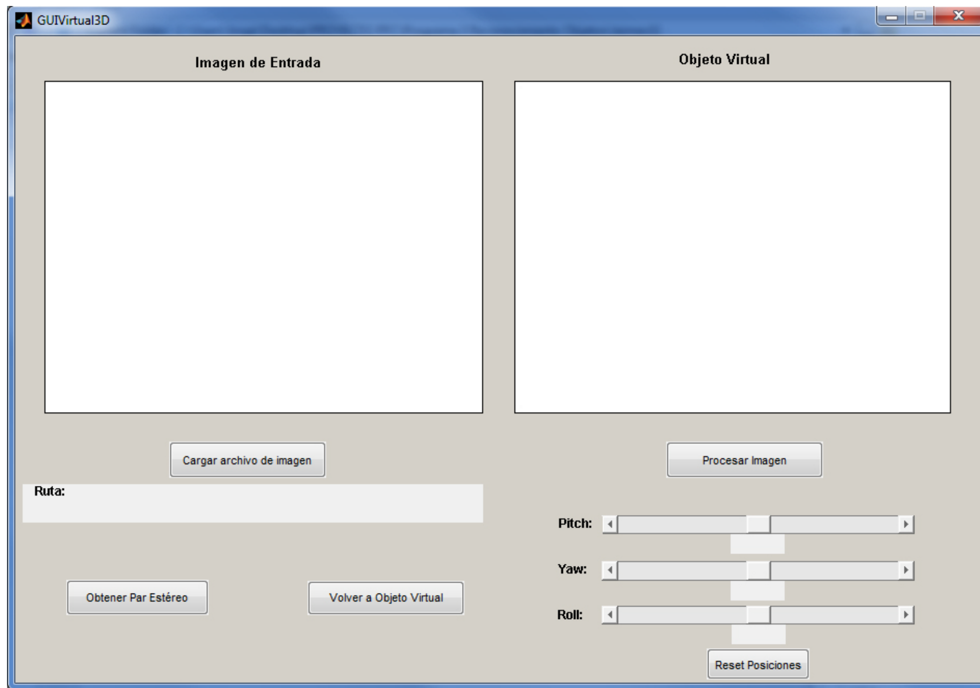

Figura 90: Ventana principal de la aplicación.

5.2.1. Zonas en la ventana principal

a) Ventana gráfica de entrada (Imagen de entrada):

En esta ventana se representa la imagen de entrada cuando el usuario la carga en el programa.

Figura 91: Ventana gráfica de Entrada.

b) Ventana gráfica de Salida (Objeto Virtual):

En esta ventana se representa el objeto virtual una vez acabado el procesado de la imagen de entrada.

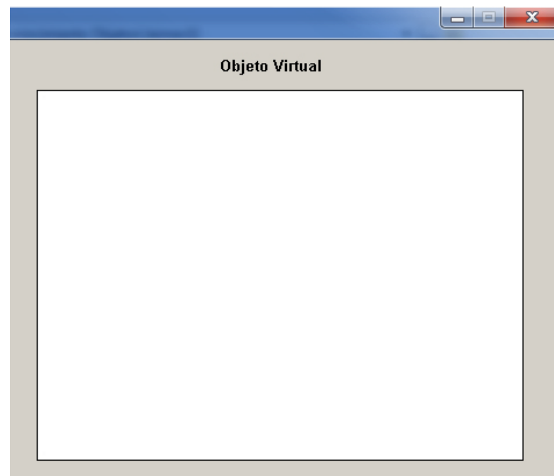

Figura 92: Ventana gráfica de Salida.

c) Deslizadores o Barras de desplazamiento:

Los deslizadores permiten rotar el objeto virtual en los tres ejes del espacio, éstos varían el ángulo de rotación desde -90º hasta 90º, siendo 0º el centro de la barra de desplazamiento. Los ejes han sido llamados Pitch, Yaw y Roll, lo que equivale a eje Y, eje X y eje Z respectivamente, siendo estos ejes los paralelos a los planos de igual nombre.

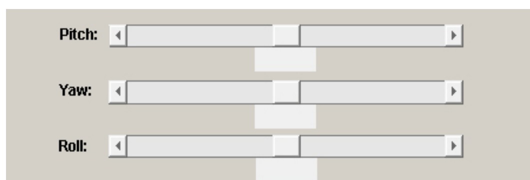

Figura 93: Deslizadores en reposo.

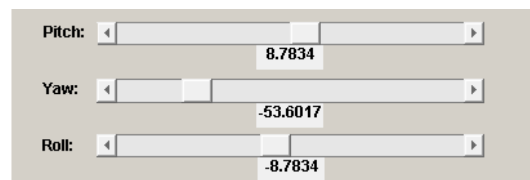

Figura 94: Deslizadores desplazados.

Los cuadros de texto bajo cada una de las barras de desplazamiento indica el valor del ángulo que marca el deslizador.

d) Cuadro de texto '*Ruta*':

En este cuadro de texto aparece la ruta de la imagen de entrada cuando el usuario la carga.

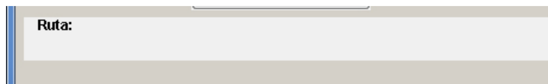

Figura 95: Cuadro de texto 'Ruta' vacío.

Figura 96: Cuadro de texto 'Ruta' indicando ruta de imagen cargada.

5.2.2. Mensajes emergentes

La interfaz realiza un seguimiento al usuario del estado del procesado de su imagen. Así pueden aparecen los siguientes mensajes en la ventana principal del programa:

1. El mensaje 'Espera. Tu imagen se está procesando…' aparece cuando el usuario pulsa el botón 'Procesar Imagen' (explicado más adelante).

Figura 97: Mensaje emergente de Espera.

2. El mensaje 'Imagen procesada.' emerge cuando el programa ha acabado todo el procesamiento y ha representado el objeto de salida (Objeto Virtual) en la Ventana gráfica de Salida.

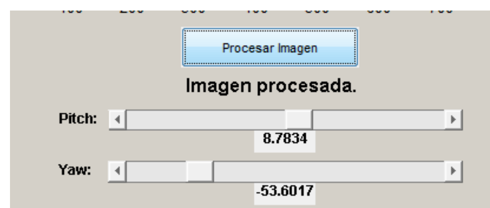

Figura 98: Mensaje emergente de finalizado.

3. El mensaje 'No se ha podido procesar la imagen.' aparece cuando el programa a encontrado un error en el procesamiento de la imagen y no ha podido por cualquier razón conseguir representar el objeto virtual.

Figura 99: Mensaje emergente de error.

5.2.3. Botones de la aplicación

a) Cargar archivo de imagen:

Al pulsar sobre este botón se realiza la siguiente función:

-Se abrirá un diálogo con el directorio correspondiente a la carpeta donde se encuentra el programa y las imágenes con las extensiones programadas en el código fuente:

```
direc=cd;
[FILENAME, PATHNAME, FILTERINDEX] = uigetfile('*jpeg;*jpg;*tiff;...
...*gif;*.bmp;*.png;','Elige la imagen a procesar','Multiselect',...
...'off');
im_rgb = imread(FILENAME);
```

El usuario podrá elegir entre todos estos formatos de imágenes. Como se muestra en la figura siguiente:

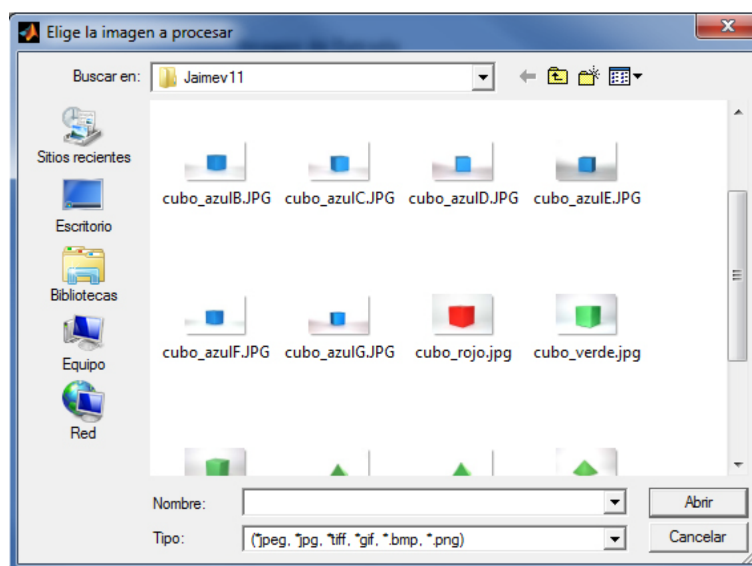

Figura 100: Diálogo para abrir archivo 'imagen de entrada'.

Cuando el usuario haya cargado la imagen seleccionada se mostrará en la ventana gráfica de entrada de la interfaz:

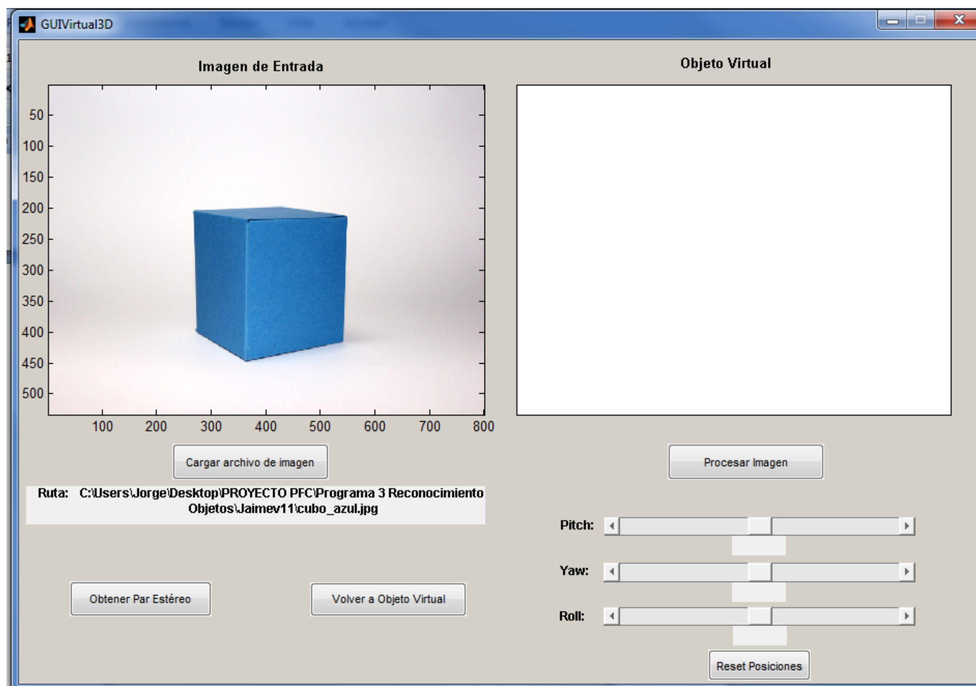

Figura 101: Ventana principal del programa cuando se ha cargado la imagen.

b) Procesar Imagen:

Al pulsar sobre este botón comienza el procesado de la imagen abierta previamente. Este procesado es el algoritmo desarrollado en la sección anterior, que incluye las fases 1 a 5, esto es, la detección de bordes, detección de líneas rectas, refinamiento, detección de esquinas y virtualización y renderizado.

Un ejemplo de lo que mostraría la interfaz cuando es pulsado el botón sería lo siguiente:

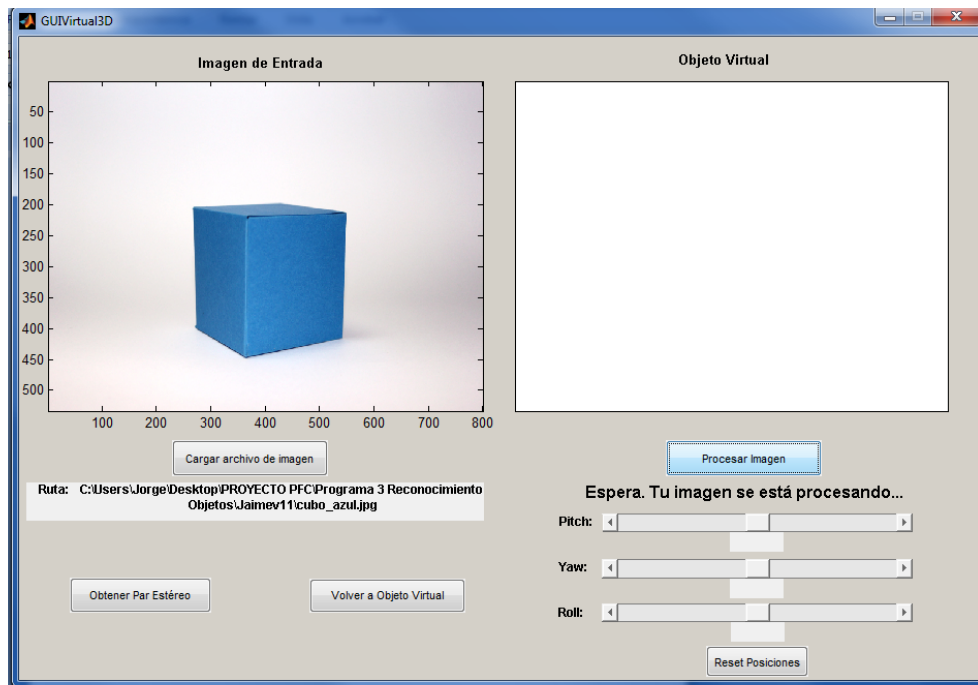

Figura 102: Ventana principal del programa cuando se pulsa el botón 'Procesar imagen'.

Una vez procesado el objeto, éste se mostrará en la ventana gráfica de salida y la ventana principal del programa tendrá un aspecto como sigue:

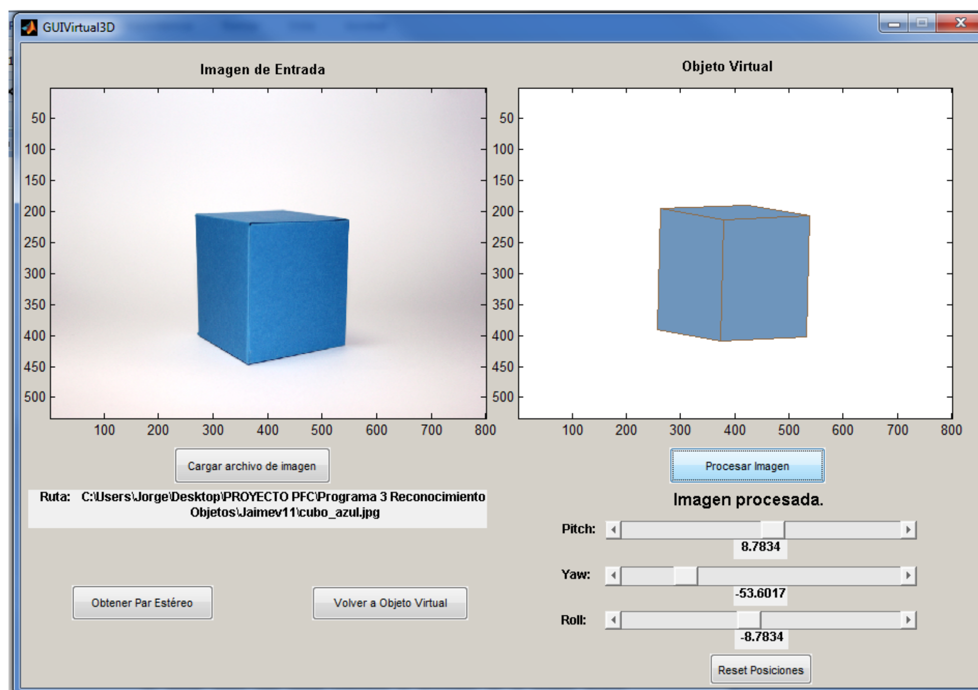

Figura 103: Aspecto de la ventana principal cuando se ha procesado el objeto.

c) Reset posiciones:

Cuando se pulsa este botón devuelve la figura a su posición inicial en caso de que previamente hayan sido desplazados los deslizadores de posición por el usuario.

d) Obtener Par Estéreo:

Al pulsar sobre este botón se calcula y se representa el par estéreo del objeto virtual que en ese momento haya sido procesado. En la ventana gráfica de la izquierda se representa la imagen del ojo izquierdo y en la ventana gráfica de la derecha se representa la imagen del ojo derecho.

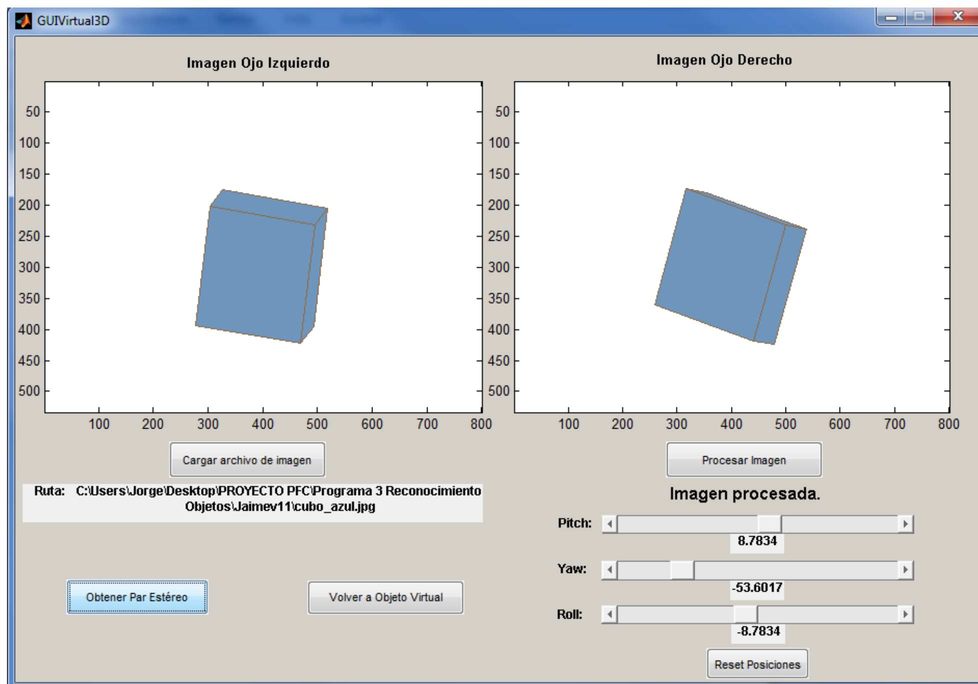

Figura 104: Ventana principal del programa al pulsar el botón 'Obtener Par Estéreo'.

e) Volver a Objeto Virtual:

Si se pulsa este botón el programa vuelve a representar la imagen de entrada y el objeto virtual en la ventana principal del programa, siempre que se haya pulsado anteriormente el botón 'Obtener Par Estéreo'.

6. RESULTADOS DEL ALGORITMO

En este capítulo se mostrarán los resultados obtenidos tras la aplicación del algoritmo mediante el uso del programa *GUIVirtual3D* a varias imágenes de prueba.

Existen tres grandes casos bastante diferenciados después de hacer las pruebas:

- Imágenes con procesado correcto.
- Imágenes con error de procesado.
- Imágenes con procesado incorrecto.

A continuación se desarrollan estos casos y se intenta encontrar una explicación a los casos que no funcionan correctamente.

6.1. Imágenes con procesado correcto

La mayor parte de las imágenes de prueba que se han procesado en el software han sido con resultado satisfactorio. De un total de dieciséis imágenes, once de ellas han sido procesadas correctamente. Algunos de los resultados se pueden ver seguidamente:

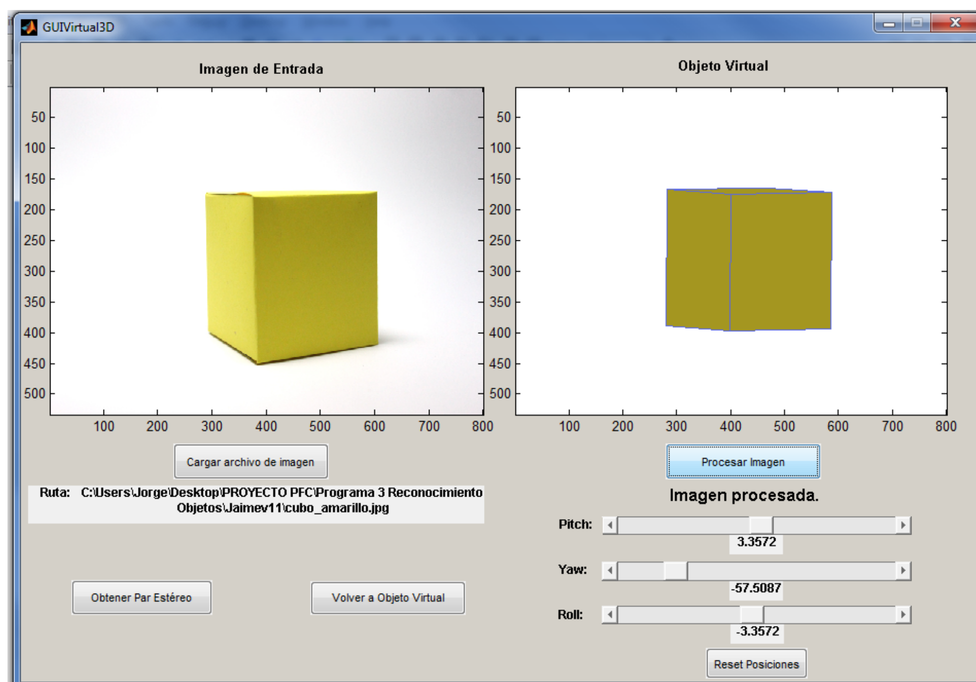

Figura 105: Procesado de la imagen cubo_amarillo.jpg

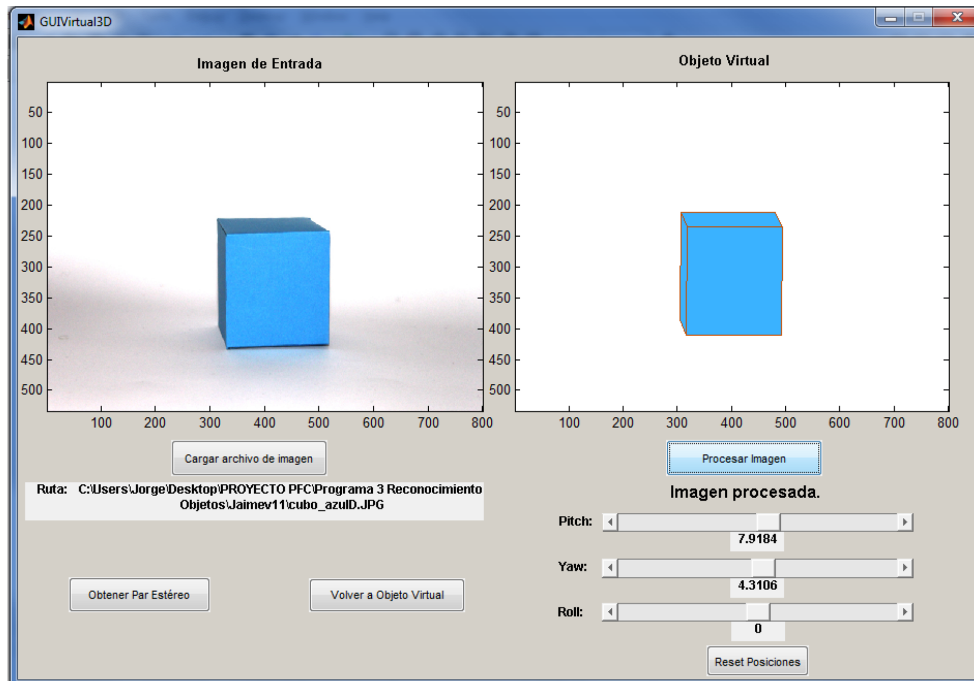

Figura 106: Procesado de la imagen cubo_azulD.jpg

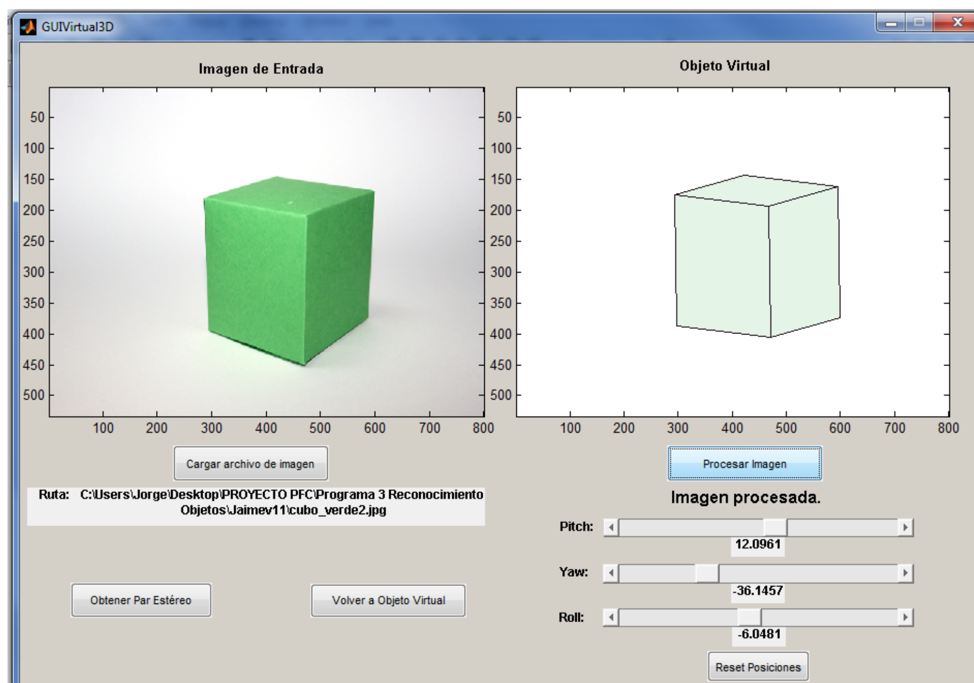

Figura 107: Procesado de la imagen cubo_verde2.jpg

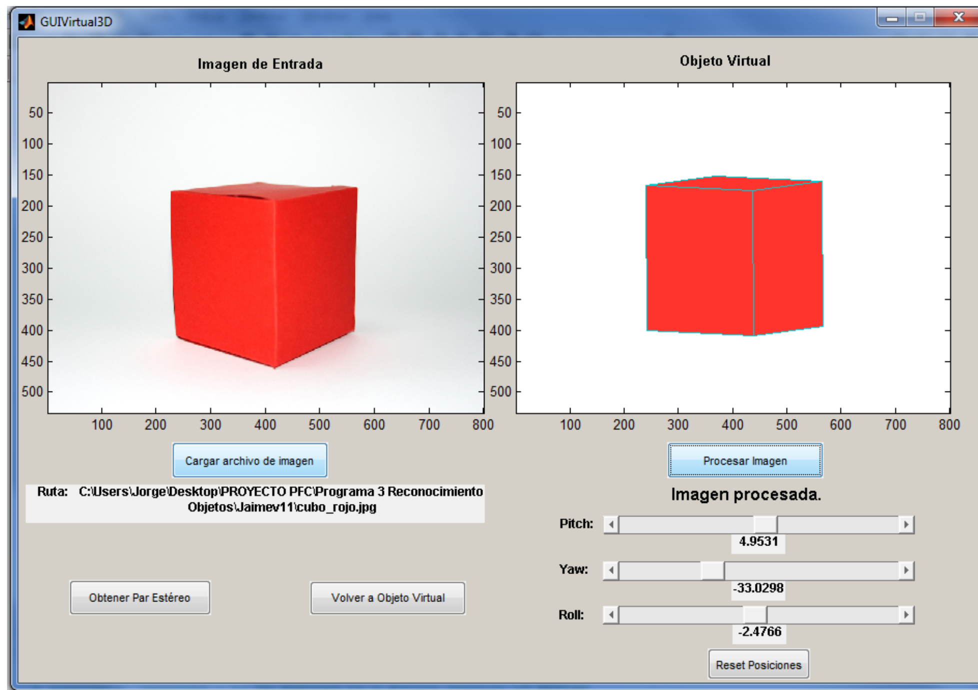

Figura 108: Procesado de la imagen cubo_rojo.jpg

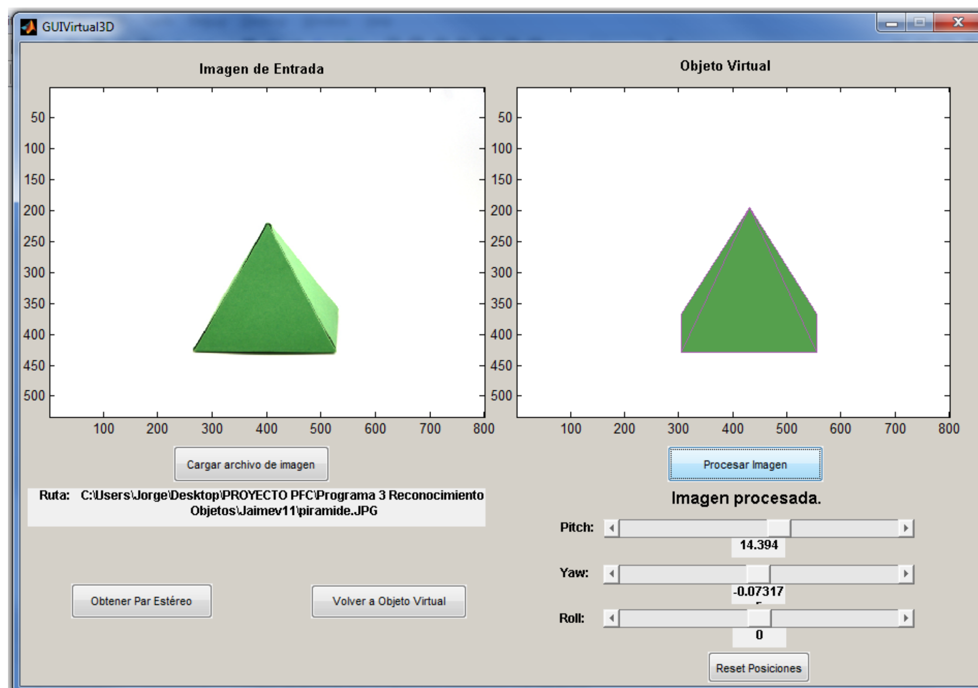

Figura 109: Procesado de la imagen piramide.jpg

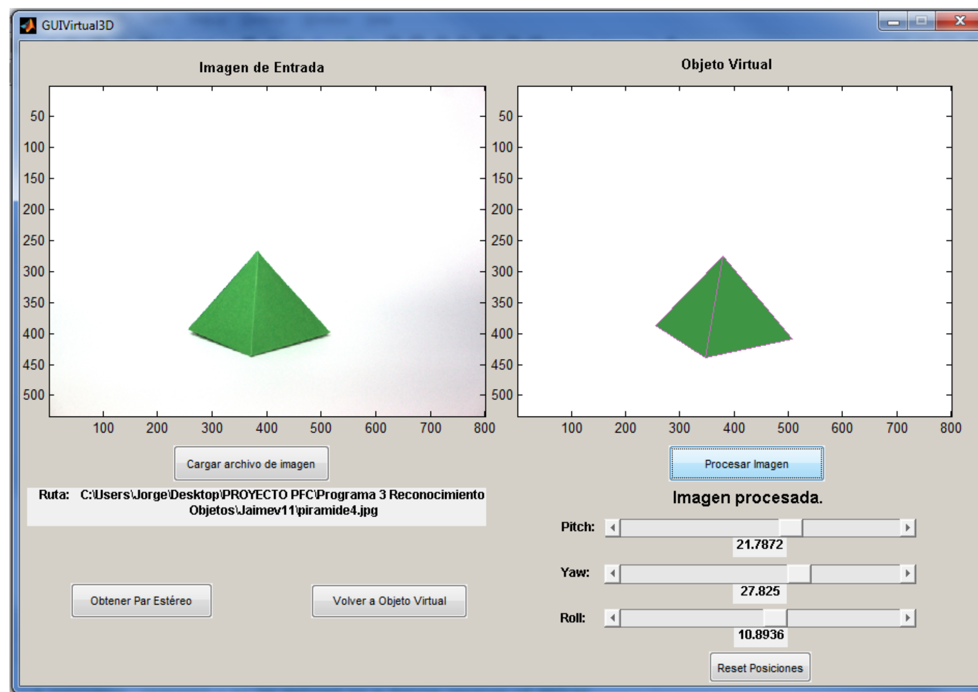

Figura 110: Procesado de la imagen piramide4.jpg

Como puede apreciarse en las imágenes anteriores los resultados obtenidos son bastante buenos si se tiene en cuenta la limitación de la transformación perspectiva que no ha sido incluida en el algoritmo.

6.2. Imágenes con error de procesado

Al probar el algoritmo también se encuentran imágenes que no son capaces de ser procesadas y el programa muestra un mensaje de error. Aunque en el marco de este proyecto no se va a implementar una mejora del algoritmo, en las posteriores líneas se va a intentar encontrar el problema y dar una explicación de forma que en un futuro pueda ser solucionado.

El primer caso encontrado es la imagen *cubo_azulG.jpg:*

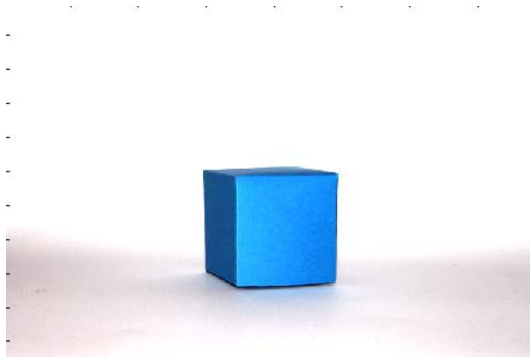

Figura 111: Imagen cubo_azulG.jpg

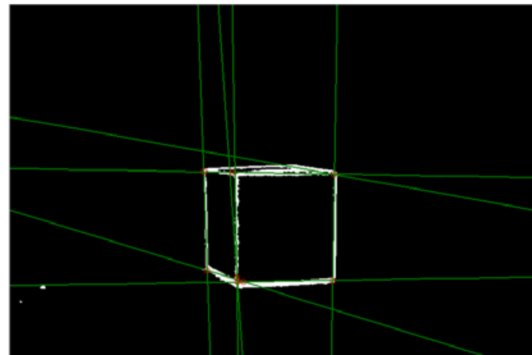

Figura 112: Imagen con detección de bordes, líneas y vértices.

En las figuras anteriores se aprecia como la detección de líneas rectas es deficiente y el refinamiento no ha mejorado la detección. Además detecta vértices bastante desplazados y es aquí donde se encuentra el error. El algoritmo no encuentra vértices sobre las líneas por lo que no puede continuar y no procesa la imagen.

El error que muestra el algoritmo es el que sigue:

```
vertices_final =

     []

Error in ==> decide_figura at 9

lineas_vert_pos=find(points_hough(:,1)>-5& points_hough(:,1)<5);
```

Es complicado encontrar explicación a este error, el error es debido a que no detecta líneas verticales entre -5º y +5º, cuando, como se puede ver en la siguiente lista de rectas detectadas, sí que las hay:

```
points_hough3 =

  -4    315
  -2    285
  -1    337
   1    496
  91    247
 107    296
  89    424
 100    167
```

En el siguiente caso la detección de líneas y vértices es bastante precisa, aunque se obtiene un error cuando el algoritmo va a eliminar los puntos de corte que no son vértices debido a que no encuentra puntos de corte sobre una de las rectas.

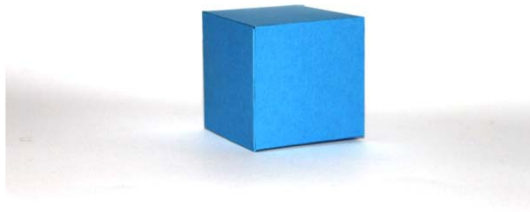

Figura 113: Imagen cubo_azulC.jpg

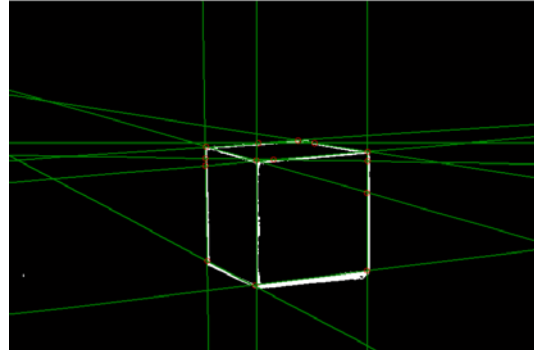

Figura 114: Imagen con detección de bordes, líneas y vértices.

El error que presenta el algoritmo es el siguiente:

```
vertices_recta =

     []

??? Error using ==> sortrows at 64

COL must be a vector of column indices into X.
```

Una solución rápida de este error sería modificar el umbral que tiene por defecto el algoritmo de refinamiento, de esta forma se procesará esta imagen aunque algunas otras no funcionaran.

La mejor solución pasa por mejorar el algoritmo de refinamiento.

El último caso que se encuentra entre los que dan error y que es un error diferente a los anteriores es el siguiente:

Figura 115: Imagen cubo_verde.jpg

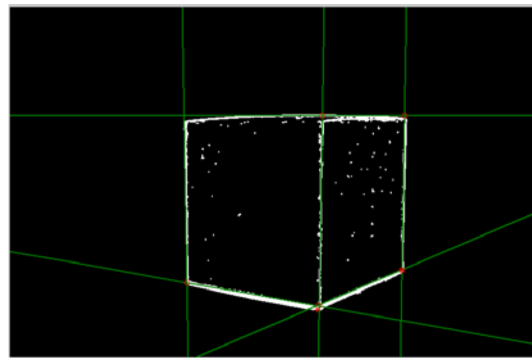

Figura 116: Imagen con detección de bordes, líneas y vértices.

Observando la figura se puede ver como la detección de bordes no es muy buena. Esto se debe a que la posición de la cámara que ha tomado la imagen ha obtenido una posición un tanto complicada para detectar los bordes. Es por esto que la imagen no se procesa correctamente porque le faltan vértices para poder dibujar el objeto.

6.3. Imágenes con procesado incorrecto

Entre los resultados obtenidos mediante el uso del software *GUIVirtual3D* también se encuentran imágenes que no son orientadas correctamente, por lo que son procesadas de manera incorrecta. Esto puede ser debido a varias causas.

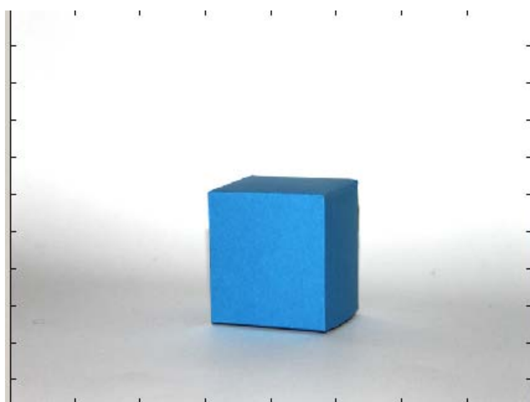

Figura 117: Imagen cubo_azulE.jpg

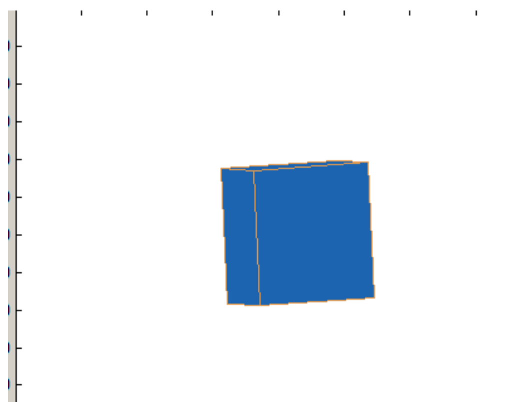

Figura 118: Objeto virtual incorrecto.

El algoritmo del cálculo de la orientación no realiza los cálculos correctamente debido a la incorrecta detección de líneas y vértices. Véase el resultado de las fases de detección y refinamiento en la siguiente imagen:

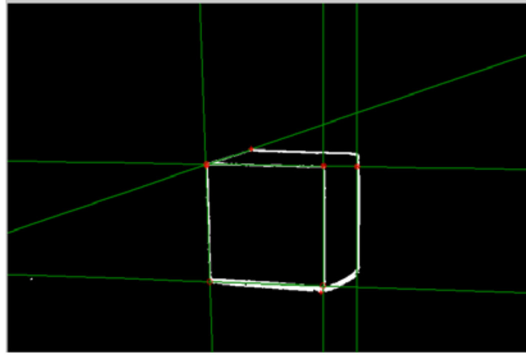

Figura 119: Imagen con fase de detección y refinamiento.

Otro procesado incorrecto es el que se encuentra al probar la imagen *cubo_azulA.jpg*

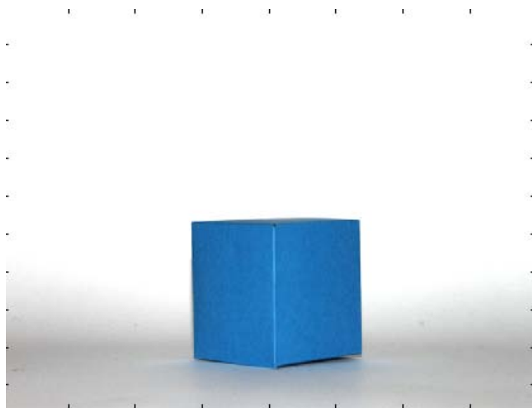

Figura 120: Imagen cubo_azulA.jpg

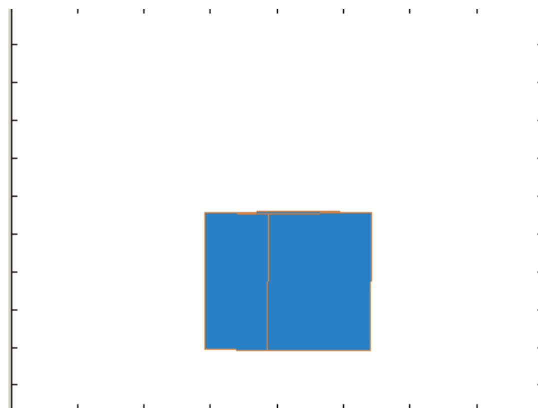

Figura 121: Objeto virtual incorrecto.

En esta ocasión el error en los cálculos de orientación de la imagen se encuentra en el propio algoritmo. A pesar de que detecta todos los vértices correctamente, como puede verse en la figura siguiente, los cálculos que realiza no son correctos y esto produce una virtualización y renderización incorrectas.

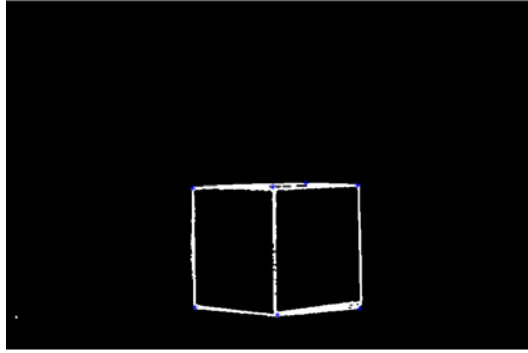

Figura 122: Detección de vértices correcta.

Este problema podría ser solucionado mejorando el algoritmo de cálculo, aunque también hay que tener en cuenta que los cálculos no pueden ser exactos ya que no se tienen datos sobre la profundidad en la imagen debido a que se trabaja con virtualización 3D a partir de vista única y por tanto los datos obtenidos a partir de la imagen 2D en el eje Z son aproximados.

7. CONCLUSIONES Y LINEAS DE TRABAJO FUTURO

7.1. Conclusiones

Con este proyecto se ha conseguido implementar un software de reconstrucción 3D, en el que se ha hecho lo posible por automatizar todas las tareas de tratamiento de imágenes y hacerlo lo más sencillo posible.

Se ha estudiado la problemática de ocultar caras y líneas, la detección de características de un objeto y la virtualización y representación de ese objeto. Para ello se ha utilizado el software matemático MATLab.

Durante el desarrollo ha habido ciertas partes que han resultado más problemáticas, como la complejidad de la transformación perspectiva, que ha requerido un estudio más en profundidad y que finalmente no ha sido incluida en el algoritmo por la dificultad que ello conlleva.

Además se ha ampliado el objetivo inicial de la reconstrucción 3D consiguiendo aplicar un método de eliminación de vértices y aristas ocultas en los modelos 3D, utilizando el método de triangulación.

Se ha descrito un algoritmo de reconstrucción 3D de modelos BRep para objetos a partir de una sola vista. El algoritmo ha sido implementado y han sido reconstruidos varios ejemplos con éxito. Se ha comprobado que funciona para la mayor parte de los objetos.

7.2. Líneas de trabajo futuro.

Como futuras líneas de trabajo se han pensado varias posibilidades, encaminadas a ampliar el software que proporciona este proyecto y que sirva de base a futuras ampliaciones.

Algunas de estas ampliaciones pueden ser las siguientes:

- Modificaciones futuras para GUIVirtual3D

Un aspecto interesante sería incorporar como opción de imagen de entrada la adquisición de datos reales en tiempo real con ayuda de un periférico como puede ser una webcam. Esto podría realizarse mediante la herramienta Data Tool Box de Matlab.

- Implementar la transformación de perspectiva de forma que el objeto virtualizado sea lo más análogo al objeto real.

- Implementar un algoritmo de conversión estereoscópica a partir del par estéreo de imágenes obtenido.

- Ampliar los tipos de figuras a reconstruir. Además de sólidos con líneas rectas se podría introducir poliedros con líneas curvas.

- Implementar un algoritmo de reconstrucción a partir de múltiples vistas. De esta manera se podrían comprobar los resultados de las dos técnicas de reconstrucción 3D.

- Implementar un método general de ocultar líneas en pirámides cuando son visibles todos los vértices del objeto. Este proceso es costoso debido a que los objetos puede ser movidos alrededor de los tres ejes del espacio y a que no se ocultan vértices y sus aristas, sino que hay que encontrar únicamente aristas ocultas sin sus vértices.

8. BIBLIOGRAFÍA

- "3D Scanning Advancements in Medical Science". Konica Minolta. Retrieved 24 October 2011.

- "3D Marketplace 3D Model Licensing Documentation". Retrieved 30 October 2011.

- C. Tomasi and T. Kanade, “Shape and motion from image streams under orthography: A factorization approach”, International Journal of Computer Vision, 9(2):137-154, 1992.

- R. Mohr and E. Arbogast. It can be done without camera calibration. Pattern Recognition Letters, 12:39-43, 1991.

- O. Faugeras. “What can be seen in three dimensions with an uncalibrated stereo rig?” In Proceedings of the European Conference on Computer Vision, pages 563-578, Santa Margherita L., 1992.

- S. J. Maybank and O. Faugeras. “A theory of self-calibration of a moving camera.” International Journal of Computer Vision, 8(2):123-151, 1992.

- O. Faugeras and S. Maybank. “Motion from point matches: multiplicity of solutions.” International Journal of Computer Vision, 4(3):225-246, June 1990.

- R. I. Hartley. “Kruppa's equations derived from the fundamental matrix.” IEEE Transactions on Pattern Analysis and Machine Intelligence, 19(2):133-135, February 1997.

- R. Hartley and A. Zisserman. “Multiple view geometry in computer vision.” Cambridge University Press, 2nd edition, 2003.

- José García Resta: "Reconstrucción 3D de objetos poliédricos a partir de su representación axonométrica oblicua 2D". Proyecto Fin de Carrera, 1997.

- Foley J.D.; van Dam A.; Feiner S.K.; Hughes J.F.: "Computer Graphics. Theory and Practice". Addison-Wesley 1990.

- Mantyla, M.; "An introduction to solid modeling". Computer Science Press 1988.

- "Práctica 4ª: Segmentación.Transformada de Hough". Universidad de Jaén, 2005.

- "Práctica 3ª: Detección de bordes en una imagen". Universidad de Jaén, 2005.

ANEXO I: ALGORITMO DE FUNCIONES PROPIAS

En las siguientes páginas de este anexo se encuentra el código fuente de las funciones externas a la toolbox de Matlab creadas premeditadamente para desarrollar correctamente el algoritmo principal del software *GUIVirtual3D*.

ÍNDICE

1. **FUNCIÓN *DETECTAR_ANGULO_HOUGH*............................102**
2. **FUNCIÓN *DETECTAR_PICOS_HOUGH*................................102**
3. **FUNCIÓN *DECIDE_FIGURA*..104**
4. **FUNCIÓN *GENERAR_FIGURA_VIRTUAL*.............................111**
5. **FUNCIÓN *RELLENAFIGURAVIRTUAL2*................................114**
6. **FUNCIÓN *BUSCAR_PICOS_VECTOR*.....................................135**
7. **FUNCIÓN *VERTEXTRASERODELANTERO*..........................136**
8. **FUNCIÓN *VERTEXTRASERODELANTERO2*........................136**
9. **FUNCIÓN *VERTICEABAJOARRIBA*..137**

1. Función *detectar_angulo_Hough*

```
function angulo_out = detectar_angulo_Hough(H_zona)

   pixels_proyeccion=[];
   angulos=linspace(-90,90,50);
   H_zona2=zeros(size(H_zona));
   pos=find(H_zona>0.5*max(max(H_zona)));
   H_zona2(pos)=1;
   for angle=angulos
   I = mat2gray(H_zona2);
   I_rot = imrotate(I,angle,'bilinear','loose');
   pixels_blanco=max(I_rot,[],1);
   pixels_proyeccion=[pixels_proyeccion, sum(pixels_blanco)];
   end
   pos=find(pixels_proyeccion==min(pixels_proyeccion));
   angulo_out=angulos(pos);
```

2. Función *detectar_picos_Hough*

```
function picos_out=detectar_picos_Hough(H_zona,inicio_x,fin_x,...
...inicio_y, fin_y)

   picos_out=[];
   H_y=max(H_zona,[],2);
   H_y(2)=H_y(2)+10;
   H_y(end-1)=H_y(end-1)+10;
   subplot(2,1,1), hold on
   eje_y=inicio_y:fin_y;
   plot(eje_y,H_y)
   pos=busca_picos_vector(H_y, 0.1);
   y_aux=eje_y;
   if length(pos)>1
   H_aux=interp1([eje_y(1) eje_y(pos) eje_y(end)],...
   ...[0;H_y(pos);0],y_aux,'pchip');
   elseif length(pos)<=1
   H_aux=spline(eje_y,H_y,y_aux);
   end
   minpeakdis=round(0.9*70/(y_aux(2)-y_aux(1)))
   minpeakdis=30;
   minpeakhei=0.5*max(H_aux);

   if minpeakdis>=length(y_aux)
       minpeakdis=length(y_aux)-1;
   else
       minpeakdis
```

```
    end

    [pico,pos_y]=findpeaks(H_aux,'MINPEAKHEIGHT',minpeakhei,...
    ...'MINPEAKDISTANCE',minpeakdis,'SORTSTR','descend');

    if sum(pos_y)==0
        return
    end

    if length(pos_y)==1
        H_x=max(H_zona,[],1);
        eje_x=inicio_x:fin_x;
        x_aux=linspace(eje_x(1), eje_x(end), 100);
        H_aux=spline(eje_x,H_x,x_aux);
        [pico,pos_x]=findpeaks(H_aux,'MINPEAKHEIGHT',minpeakhei);
       if sum(pos_x)==0
            return
       end

       if length(pos_x)==1
          pico_actual=[x_aux(pos_x), y_aux(pos_y)];
          picos_out=[picos_out ; pico_actual];
       else
          picos_Hough=[];
          puntos_x_division=[x_aux(1) (x_aux(pos_x(1:end-1))+...
       ...x_aux(pos_x(2:end)))/2 x_aux(end)];
          puntos_x_division=sort(puntos_x_division);
          for ind=1:length(pos_x)
             pos_zona=find((eje_x>(puntos_x_division(ind)-1))...
          ...&(eje_x<(puntos_x_division(ind+1)+1)));
             H_zona_trozo=H_zona(:,pos_zona);

            picos_Hough_actual=detectar_picos_Hough(H_zona_trozo,...
         ...eje_x(pos_zona(1)),eje_x(pos_zona(end)),inicio_y,fin_y);
               picos_Hough = [picos_Hough; picos_Hough_actual];
           end
            picos_out=[picos_out ; picos_Hough];
       end

    else
       picos_Hough=[];
       puntos_y_division=[y_aux(1) (y_aux(pos_y(1:end-1))+...
    ...y_aux(pos_y(2:end)))/2 y_aux(end)];
       puntos_y_division=sort(puntos_y_division);
       for ind=1:length(pos_y)
          pos_zona=find((eje_y>(puntos_y_division(ind)-1))...
       ...&(eje_y<(puntos_y_division(ind+1)+1)));
         H_zona_trozo=H_zona(pos_zona,:);
         picos_Hough_actual=detectar_picos_Hough(H_zona_trozo,...
      ...inicio_x, fin_x, eje_y(pos_zona(1)), eje_y(pos_zona(end)));
         picos_Hough = [picos_Hough; picos_Hough_actual];
     end
     picos_out=[picos_out ; picos_Hough];
```

```
   end
```

3. Función *decide_figura*

```
function [vertices_finales figura] = decide_figura(mascara_imagen,...
...vertices_inicial, points_hough,gradiente)

lineas_vert_pos = find(points_hough(:,1)>-5 & points_hough(:,1)<5);
lineas_horz_pos = find(points_hough(:,1)>87 & points_hough(:,1)<93);
if length(lineas_vert_pos)>=2
disp('la figura es un cubo.')
figura=1;
else
disp('la figura es una pirámide.')
figura=0;
end

size_im=size(mascara_imagen);
x_vector=linspace(1,size_im(2),10);
if figura==1

vertices_final=[];
lineas_vert=points_hough(lineas_vert_pos,:);
lineas_vert=sortrows(lineas_vert,2);

for ind1=1:length(lineas_vert_pos)
linea_vert_actual=lineas_vert(ind1,:);
m_param=-1./tan(linea_vert_actual(:,1)*pi/180);
b_param=linea_vert_actual(:,2)./sin(linea_vert_actual(:,1)*pi/180);
y=m_param*x_vector+b_param;
vertices_recta=[];
for ind2=1:length(vertices_inicial)
y_recta_actual = m_param*vertices_inicial(ind2,1)+b_param;
y_vertice_actual = vertices_inicial(ind2,2);
if (abs(y_recta_actual-y_vertice_actual)<400)
vertices_recta=[vertices_recta; vertices_inicial(ind2,:)];
end
end

vertices_recta=sortrows(vertices_recta,2)
if (length(lineas_vert_pos)==3 && (ind1==1 | ind1==3))&&...
...length(lineas_horz_pos)<3

vertices_final=[vertices_final; vertices_recta(end,:)];
vertices_final=[vertices_final; vertices_recta(1,:)];
end
```

```
if (length(lineas_vert_pos)==3 && ind1==2)&&...
...length(lineas_horz_pos)<3

vertice_delantero=vertexTraseroDelantero(gradiente,0);

if vertice_delantero(1,1)<vertices_final(1,1)
vertices_final=[vertices_final; vertices_recta(end,:)];
else
vertices_final=[vertices_final; vertice_delantero];
end

dif_altura=[abs(vertices_final(1,2)-vertices_final(2,2))]
pos_y_vertice=vertices_final(3,2)-dif_altura
dif_pos=abs(vertices_recta(:,2)-pos_y_vertice)
pos_vert_bueno=find(dif_pos==min(dif_pos))
vertices_final(4,:)=vertices_recta(pos_vert_bueno(1),:)
end

if length(lineas_vert_pos)==3 && length(vertices_final)==6 &&...
...length(lineas_horz_pos)<3

if ind1==3
      vertices_finales=zeros(7,2);
      vertices_finales(3,:)=vertices_final(3,:);

if abs(vertices_final(1,2))<=abs(vertices_final(5,2)-100)
      vertice_mayor=max(vertices_final(1,2),vertices_final(5,2));
      pos_vertice_mayor=find(vertices_final(:,2)==vertice_mayor);

if vertices_final(pos_vertice_mayor,:)==vertices_final(1,:)
      vertices_finales(1,:)=vertices_final(pos_vertice_mayor,:);
elseif vertices_final(pos_vertice_mayor,:)==vertices_final(5,:)
      vertices_finales(5,:)=vertices_final(pos_vertice_mayor,:);
end

else
      vertices_finales(1,:)=vertices_final(1,:);
      vertices_finales(5,:)=vertices_final(5,:);
end

if abs(vertices_final(2,2))>=abs(vertices_final(6,2)+100)
      vertice_menor=min(vertices_final(2,2),vertices_final(6,2));
      pos_vertice_menor=find(vertices_final(:,2)==vertice_menor);

if vertices_final(pos_vertice_menor,:)==vertices_final(2,:)
      vertices_finales(2,:)=vertices_final(pos_vertice_menor,:);
elseif vertices_final(pos_vertice_menor,:)==vertices_final(6,:)
      vertices_finales(6,:)=vertices_final(pos_vertice_menor,:);
end
else
      vertices_finales(2,:)=vertices_final(2,:);
      vertices_finales(6,:)=vertices_final(6,:);
```

```
end

if ((abs(vertices_final(1,2))<=abs(vertices_final(5,2)-100)) ||...
...(abs(vertices_final(2,2))>=abs(vertices_final(6,2)+100)))
      break
else
      vertices_finales(4,:)=vertices_final(4,:);
end
end
end

if length(lineas_vert_pos)==3  && ind1==3  &&...
...length(lineas_horz_pos)<3

      vertice_trasero=vertexTraseroDelantero(gradiente,1);
      vertices_finales(7,:)=vertice_trasero;
end

if length(lineas_vert_pos)==3 && length(lineas_horz_pos)==3

if ind1==1
      distA=abs(vertices_inicial(1,1)-vertices_inicial(3,1));
if length(vertices_inicial)>=6
      distB=abs(vertices_inicial(3,1)-vertices_inicial(6,1));
else
      distB=abs(vertices_inicial(3,1)-vertices_inicial(5,1));
end

if distA<distB
      orientacion=1;
elseif distA>distB
      orientacion=2;
end
end

if orientacion==1
      vertices_finales=zeros(7,2);

if ind1==1
      [vertice_abajo vertice_arriba]=verticeAbajoArriba(gradiente,1);
      vertices_final=[vertices_final; vertice_abajo];
      vertices_final=[vertices_final; vertice_arriba];
end

if ind1==2
      vertices_final=[vertices_final; vertices_recta(end,:)];
      vertices_final=[vertices_final; vertices_recta(2,:)];
end

if ind1==3
      vertices_final=[vertices_final; vertices_recta(end,:)];
      vertices_final=[vertices_final; vertices_recta(1,:)];
```

```
      vertice_trasero=vertexTraseroDelantero2(gradiente,1);
      vertices_final=[vertices_final; vertice_trasero];
      vertices_finales(1,:)=vertices_final(1,:);
      vertices_finales(2,:)=vertices_final(2,:);
      vertices_finales(3,:)=vertices_final(3,:);
      vertices_finales(4,:)=vertices_final(4,:);
      vertices_finales(5,:)=vertices_final(5,:);
      vertices_finales(6,:)=vertices_final(6,:);
      vertices_finales(7,:)=vertices_final(7,:);
end
elseif orientacion==2
      vertices_finales=zeros(7,2);
if ind1==1
      vertices_final=[vertices_final; vertices_recta(end,:)];
      vertices_final=[vertices_final; vertices_recta(1,:)];
end
if ind1==2
      vertices_final=[vertices_final; vertices_recta(end,:)];
      vertices_final=[vertices_final; vertices_recta(1,:)];
end
if ind1==3
      [vertice_abajo vertice_arriba]=verticeAbajoArriba(gradiente,3);
      vertices_final=[vertices_final; vertice_abajo];
      vertices_final=[vertices_final; vertice_arriba];
      vertice_trasero=vertexTraseroDelantero2(gradiente,1);
      vertices_final=[vertices_final; vertice_trasero];
      vertices_finales(1,:)=vertices_final(1,:);
      vertices_finales(2,:)=vertices_final(2,:);
      vertices_finales(3,:)=vertices_final(3,:);
      vertices_finales(4,:)=vertices_final(4,:);
      vertices_finales(5,:)=vertices_final(5,:);
      vertices_finales(6,:)=vertices_final(6,:);
      vertices_finales(7,:)=vertices_final(7,:);
end
end
end
end
if length(lineas_vert_pos)==2
      vertices_rectas2=[];
for ind1=1:length(lineas_vert_pos)
   linea_vert_actual=lineas_vert(ind1,:);
   m_param=-1./tan(linea_vert_actual(:,1)*pi/180);
   b_param=linea_vert_actual(:,2)./sin(linea_vert_actual(:,1)*pi/180);
   y=m_param*x_vector+b_param;
for ind2=1:length(vertices_inicial)
      y_recta_actual = m_param*vertices_inicial(ind2,1)+b_param;
      y_vertice_actual = vertices_inicial(ind2,2);
if (abs(y_recta_actual-y_vertice_actual)<400)
      vertices_rectas2=[vertices_rectas2; vertices_inicial(ind2,:)];
end
end
end
min_vertice=min(vertices_rectas2(:,1));
```

```
max_vertice=max(vertices_rectas2(:,1));
vertices_medios=find(vertices_inicial(:,1)>min_vertice &...
...vertices_inicial(:,1)<max_vertice);

vertices_finales=zeros(7,2);

if length(vertices_medios)>0

for ind1=1:length(lineas_vert_pos)
   linea_vert_actual=lineas_vert(ind1,:);
   m_param=-1./tan(linea_vert_actual(:,1)*pi/180);
   b_param=linea_vert_actual(:,2)./sin(linea_vert_actual(:,1)*pi/180);
   y=m_param*x_vector+b_param;
   vertices_recta=[];

for ind2=1:length(vertices_inicial)
      y_recta_actual = m_param*vertices_inicial(ind2,1)+b_param;
      y_vertice_actual = vertices_inicial(ind2,2);
if (abs(y_recta_actual-y_vertice_actual)<400)
      vertices_recta=[vertices_recta; vertices_inicial(ind2,:)];
end
end
      vertices_recta=sortrows(vertices_recta,2)

if ind1==2
      vertice_delantero=vertexTraseroDelantero(gradiente,0);
      vertices_finales(3,:)=vertice_delantero;
end

if ind1==2
      dif_altura=abs(diff(vertices_finales(:,2)));
      dif_altura=dif_altura(1);
      pos_y_vertice3=vertices_finales(3,2)-dif_altura;
      pos_vertices_posibles=find(vertices_inicial(:,1)>...
      ...vertices_finales(3,1)-50 & vertices_inicial(:,1)<...
      ...vertices_finales(3,1)+50);
      vertices_posibles=vertices_inicial(pos_vertices_posibles,:);
      dif_pos=abs(vertices_posibles(:,2)-pos_y_vertice3);
      pos_vert_bueno=find(dif_pos==min(dif_pos));
      vertices_finales(4,:)=vertices_posibles(pos_vert_bueno,:);
end

if ind1==1
      vertices_finales(1,:)=vertices_recta(end,:);
      vertices_finales(2,:)=vertices_recta(1,:);
end

if ind1==2
      vertices_finales(5,:)=vertices_recta(end,:);
      vertices_finales(6,:)=vertices_recta(1,:);
end

if ind1==2
```

```
      vertice_trasero=vertexTraseroDelantero(gradiente,1);
      vertices_finales(7,:)=vertice_trasero;
end
end
end

      vertices_derecha=find(vertices_inicial(:,1)>max_vertice);

if length(vertices_derecha)>0

for ind1=1:length(lineas_vert_pos)
   linea_vert_actual=lineas_vert(ind1,:);
   m_param=-1./tan(linea_vert_actual(:,1)*pi/180);
   b_param=linea_vert_actual(:,2)./sin(linea_vert_actual(:,1)*pi/180);
   y=m_param*x_vector+b_param;
   vertices_recta=[];
for ind2=1:length(vertices_inicial)
      y_recta_actual = m_param*vertices_inicial(ind2,1)+b_param;
      y_vertice_actual = vertices_inicial(ind2,2);
if (abs(y_recta_actual-y_vertice_actual)<400)
      vertices_recta=[vertices_recta; vertices_inicial(ind2,:)];
end
end

      vertices_recta=sortrows(vertices_recta,2)

if ind1==2
      vertice_delantero=vertexTraseroDelantero(gradiente,0);
      vertices_finales(3,:)=vertice_delantero;
end
if ind1==2
      dif_altura=abs(diff(vertices_finales(:,2)));
      dif_altura=dif_altura(1);
      pos_y_vertice3=vertices_finales(3,2)-dif_altura;
      pos_vertices_posibles=find(vertices_inicial(:,1)>...
      ...vertices_finales(3,1)-50 & vertices_inicial(:,1)<...
      ...vertices_finales(3,1)+50);
      vertices_posibles=vertices_inicial(pos_vertices_posibles,:);
      dif_pos=abs(vertices_posibles(:,2)-pos_y_vertice3);
      pos_vert_bueno=find(dif_pos==min(dif_pos));
      vertices_finales(4,:)=vertices_posibles(pos_vert_bueno,:);
end

if ind1==1
      vertices_finales(1,:)=vertices_recta(end,:);
      vertices_finales(2,:)=vertices_recta(1,:);
end

if ind2==2
      vertices_prob=vertices_inicial(vertices_derecha,:);
      pos_alto=find(vertices_prob(:,2)<vertices_finales(2,2)+50...
      ...& vertices_prob(:,2)>vertices_finales(2,2)-50);
      vertices_finales(6,:)=vertices_prob(pos_alto,:);
```

```
      pos_bajo=find(vertices_prob(:,2)<vertices_finales(1,2)+50...
      ...& vertices_prob(:,2)>vertices_finales(1,2)-50);
      vertices_finales(5,:)=vertices_prob(pos_bajo,:);
end

if ind1==2
      vertice_trasero=vertexTraseroDelantero(gradiente,1);
      vertices_finales(7,:)=vertice_trasero;
end
end
end

      vertices_izquierda=find(vertices_inicial(:,1)<min_vertice);

if length(vertices_izquierda)>0

for ind1=1:length(lineas_vert_pos)
   linea_vert_actual=lineas_vert(ind1,:);
   m_param=-1./tan(linea_vert_actual(:,1)*pi/180);
   b_param=linea_vert_actual(:,2)./sin(linea_vert_actual(:,1)*pi/180);
   y=m_param*x_vector+b_param;
   vertices_recta=[];

for ind2=1:length(vertices_inicial)
      y_recta_actual = m_param*vertices_inicial(ind2,1)+b_param;
      y_vertice_actual = vertices_inicial(ind2,2);
if (abs(y_recta_actual-y_vertice_actual)<400)
      vertices_recta=[vertices_recta; vertices_inicial(ind2,:)];
end
end

      vertices_recta=sortrows(vertices_recta,2)

if ind1==2
      vertice_delantero=vertexTraseroDelantero(gradiente,0);
      vertices_finales(3,:)=vertice_delantero;
end

if ind1==2
      vertices_finales(5,:)=vertices_recta(end,:);
      vertices_finales(6,:)=vertices_recta(1,:);
end

if ind2==2
      vertices_prob=vertices_inicial(vertices_izquierda,:);
      pos_alto=find(vertices_prob(:,2)<vertices_finales(6,2)+50...
      ...& vertices_prob(:,2)>vertices_finales(6,2)-50);
      vertices_finales(2,:)=vertices_prob(pos_alto,:);
      pos_bajo=find(vertices_prob(:,2)<vertices_finales(5,2)+50...
      ...& vertices_prob(:,2)>vertices_finales(5,2)-50);
      vertices_finales(1,:)=vertices_prob(pos_bajo,:);
end
```

```
if ind1==2
      dif_altura=abs(vertices_finales(2,2)-vertices_finales(1,2));
      pos_y_vertice3=vertices_finales(3,2)-dif_altura;
      pos_vertices_posibles=find(vertices_inicial(:,1)>...
      ...vertices_finales(3,1)-50 & vertices_inicial(:,1)<...
      ...vertices_finales(3,1)+50);
      vertices_posibles=vertices_inicial(pos_vertices_posibles,:);
      dif_pos=abs(vertices_posibles(:,2)-pos_y_vertice3);
      pos_vert_bueno=find(dif_pos==min(dif_pos));
      vertices_finales(4,:)=vertices_posibles(pos_vert_bueno,:);
end

if ind1==2
      vertice_trasero=vertexTraseroDelantero(gradiente,1);
      vertices_finales(7,:)=vertice_trasero;
end
end
end
end

elseif figura==0
      vertices_finales=vertices_inicial;
if vertices_inicial(2,2)>vertices_inicial(3,2)
      vertice_2=vertices_inicial(2,:);
      vertice_3=vertices_inicial(3,:);
      vertices_finales(3,:)=vertice_2;
      vertices_finales(2,:)=vertice_3;
end
end
```

4. Función *generar_figura_virtual*

```
function vertices_virtuales=generar_figura_virtual(vertices_imagen,...
...figura,pitch,yaw,roll)

if figura==1
    image_size=[800, 533];
    L=sqrt((vertices_imagen(2,1)-vertices_imagen(1,1))^2+...
...(vertices_imagen(2,2)-vertices_imagen(1,2))^2)

    object_position = mean(vertices_imagen)
    object_position(2)=-object_position(2);

    vertices = [
        -L/2, -L/2, -L/2;
        -L/2, L/2, -L/2;
        L/2, -L/2, -L/2;
```

```
        L/2, L/2, -L/2;
        L/2, -L/2, L/2;
        L/2, L/2, L/2;
        -L/2, -L/2, L/2;
        -L/2, L/2, L/2];

    vertices_coord=[vertices ones(size(vertices,1),1)];

    % ----------------------------------------------------------------
    % ---------- escalado del objeto
    % ----------------------------------------------------------------
    scaling_factor=1;
    scaling_matrix = [
        scaling_factor 0 0 0;
        0 scaling_factor 0 0;
        0 0 scaling_factor 0;
        0 0 0 1];

    vertices_coord=vertices_coord*scaling_matrix;

    % ----------------------------------------------------------------
    % ---------- rotación del objeto
    % ----------------------------------------------------------------
    % ahora distinguiré entre 3 rotaciones para cada uno de los ejes

    rotation_matrix_pitch = [
        1 0 0 0;
        0 cos(pitch) sin(pitch) 0;
        0 -sin(pitch) cos(pitch) 0;
        0 0 0 1];

    rotation_matrix_yaw = [
        cos(yaw) 0 -sin(yaw) 0;
        0 1 0 0;
        sin(yaw) 0 cos(yaw) 0;
        0 0 0 1];

    rotation_matrix_roll = [
        cos(roll) sin(roll)  0 0;
        -sin(roll) cos(roll) 0 0;
        0 0 1 0;
        0 0 0 1];

    vertices_coord=vertices_coord*rotation_matrix_pitch*...
 ...rotation_matrix_yaw*rotation_matrix_roll;

    % ----------------------------------------------------------------
    % ---------- traslación del objeto
    % ----------------------------------------------------------------
    translation_matrix = [
        1 0 0 0;
        0 1 0 0;
        0 0 1 0;
        object_position 0 1];

    vertices_coord=vertices_coord*translation_matrix;
    vertices_virtuales=vertices_coord;

elseif figura==0
```

```
    image_size=[800, 533];
    L=sqrt((vertices_imagen(2,1)-vertices_imagen(1,1))^2+...
 ...(vertices_imagen(2,2)-vertices_imagen(1,2))^2);

    H= vertices_imagen(2,2)-vertices_imagen(1,2)
    H=-H;
    object_position = mean(vertices_imagen);
    object_position(2)=-object_position(2);

    vertices = [
    -L/2, -H/2, -L/2;
       0,  H/2,  0;
     L/2, -H/2,  L/2;
     L/2, -H/2, -L/2;
    -L/2, -H/2,  L/2];

    vertices_coord=[vertices ones(size(vertices,1),1)];

    centro=mean(vertices_coord);

    cambio=vertices_coord(:,2)-centro(2);
    vertices_coord(:,2)=cambio;

    mean(vertices_coord);

    % -----------------------------------------------------------------
    % ---------- escalado del objeto
    % -----------------------------------------------------------------
    scaling_factor=1;
    scaling_matrix = [
        scaling_factor 0 0 0;
        0 scaling_factor 0 0;
        0 0 scaling_factor 0;
        0 0 0 1];

    vertices_coord=vertices_coord*scaling_matrix;

    % -----------------------------------------------------------------
    % ---------- rotación del objeto
    % -----------------------------------------------------------------
    % ahora distinguiré entre 3 rotaciones para cada uno de los ejes

    rotation_matrix_pitch = [
        1 0 0 0;
        0 cos(pitch) sin(pitch) 0;
        0 -sin(pitch) cos(pitch) 0;
        0 0 0 1];

    rotation_matrix_yaw = [
        cos(yaw) 0 -sin(yaw) 0;
        0 1 0 0;
        sin(yaw) 0 cos(yaw) 0;
        0 0 0 1];

    rotation_matrix_roll = [
        cos(roll) sin(roll)  0 0;
        -sin(roll) cos(roll) 0 0;
        0 0 1 0;
        0 0 0 1];
```

```
    vertices_coord=vertices_coord*rotation_matrix_pitch*...
 ...rotation_matrix_yaw*rotation_matrix_roll;

    % ---------------------------------------------------------------
    % ---------- traslación del objeto
    % ---------------------------------------------------------------
    translation_matrix = [
        1 0 0 0;
        0 1 0 0;
        0 0 1 0;
        object_position 0 1];

    vertices_coord=vertices_coord*translation_matrix;
    vertices_virtuales=vertices_coord;

end
```

5. Función *rellenaFiguraVirtual2*

```
function vertices_ocultos=rellenaFiguraVirtual2(vertices_virtuales,...
...figura,Niv_r,Niv_g,Niv_b,NivR,NivG,NivB)

vertices_virtuales2=vertices_virtuales;
vertices_virtuales2(:,2)=abs(vertices_virtuales(:,2));

if figura==1
    num_caras=6;
    cara1=[vertices_virtuales2(1,:);vertices_virtuales2(2,:);...
...vertices_virtuales2(4,:);vertices_virtuales2(3,:)];
    cara2=[vertices_virtuales2(5,:);vertices_virtuales2(6,:);...
...vertices_virtuales2(4,:);vertices_virtuales2(3,:)];
    cara3=[vertices_virtuales2(7,:);vertices_virtuales2(8,:);...
...vertices_virtuales2(6,:);vertices_virtuales2(5,:)];
    cara4=[vertices_virtuales2(7,:);vertices_virtuales2(8,:);...
...vertices_virtuales2(2,:);vertices_virtuales2(1,:)];
    cara5=[vertices_virtuales2(5,:);vertices_virtuales2(7,:);...
...vertices_virtuales2(1,:);vertices_virtuales2(3,:)];
    cara6=[vertices_virtuales2(6,:);vertices_virtuales2(8,:);...
...vertices_virtuales2(2,:);vertices_virtuales2(4,:)];
    vertices_ocultos=[];

    for ind1=1:num_caras

         if ind1==1
            triangulo1=[cara1(1,:);cara1(2,:);cara1(3,:)];
            triangulo2=[cara1(3,:);cara1(4,:);cara1(1,:)];

            vertices_virtuales_aux=[vertices_virtuales2(5,:);
```

```
                                    vertices_virtuales2(6,:);
                                    vertices_virtuales2(7,:);
                                    vertices_virtuales2(8,:)];

           orientacion_triangulo1=(triangulo1(1,1)-triangulo1(3,1))*...
        ...(triangulo1(2,2)-triangulo1(3,2))-(triangulo1(1,2)-...
        ...triangulo1(3,2))*(triangulo1(2,1)-triangulo1(3,1));

           orientacion_triangulo2=(triangulo2(1,1)-triangulo2(3,1))*...
        ...(triangulo2(2,2)-triangulo2(3,2))-(triangulo2(1,2)- ...
        ...triangulo2(3,2))*(triangulo2(2,1)-triangulo2(3,1));

             if orientacion_triangulo1<0
                 orientacion1=0;
             else
                 orientacion1=1;
             end
             if orientacion_triangulo2<0
                 orientacion2=0;
             else
                 orientacion2=1;
             end

             for ind2=1:length(vertices_virtuales_aux(:,1))


trianguloA=[cara1(1,:);cara1(2,:);vertices_virtuales_aux(ind2,:)];

trianguloB=[cara1(2,:);cara1(3,:);vertices_virtuales_aux(ind2,:)];

trianguloC=[cara1(3,:);cara1(1,:);vertices_virtuales_aux(ind2,:)];


trianguloD=[cara1(3,:);cara1(4,:);vertices_virtuales_aux(ind2,:)];

trianguloE=[cara1(4,:);cara1(1,:);vertices_virtuales_aux(ind2,:)];

trianguloF=[cara1(1,:);cara1(3,:);vertices_virtuales_aux(ind2,:)];

orientacion_trianguloA=(trianguloA(1,1)-trianguloA(3,1))*...
...(trianguloA(2,2)-trianguloA(3,2))-(trianguloA(1,2)-...
...trianguloA(3,2))*(trianguloA(2,1)-trianguloA(3,1));

orientacion_trianguloB=(trianguloB(1,1)-trianguloB(3,1))*...
...(trianguloB(2,2)-trianguloB(3,2))-(trianguloB(1,2)-...
...trianguloB(3,2))*(trianguloB(2,1)-trianguloB(3,1));

orientacion_trianguloC=(trianguloC(1,1)-trianguloC(3,1))*...
...(trianguloC(2,2)-trianguloC(3,2))-(trianguloC(1,2)-...
...trianguloC(3,2))*(trianguloC(2,1)-trianguloC(3,1));

orientacion_trianguloD=(trianguloD(1,1)-trianguloD(3,1))*...
...(trianguloD(2,2)-trianguloD(3,2))-(trianguloD(1,2)-...
...trianguloD(3,2))*(trianguloD(2,1)-trianguloD(3,1));

orientacion_trianguloE=(trianguloE(1,1)-trianguloE(3,1))*...
...(trianguloE(2,2)-trianguloE(3,2))-(trianguloE(1,2)-...
trianguloE(3,2))*(trianguloE(2,1)-trianguloE(3,1));

orientacion_trianguloF=(trianguloF(1,1)-trianguloF(3,1))*...
```

```
...(trianguloF(2,2)-trianguloF(3,2))-(trianguloF(1,2)-...
trianguloF(3,2))*(trianguloF(2,1)-trianguloF(3,1));

                if orientacion_trianguloA<0
                    orientacionA=0;
                else
                    orientacionA=1;
                end
                if orientacion_trianguloB<0
                    orientacionB=0;
                else
                    orientacionB=1;
                end
                if orientacion_trianguloC<0
                    orientacionC=0;
                else
                    orientacionC=1;
                end
                if orientacion_trianguloD<0
                    orientacionD=0;
                else
                    orientacionD=1;
                end
                if orientacion_trianguloE<0
                    orientacionE=0;
                else
                    orientacionE=1;
                end
                if orientacion_trianguloF<0
                    orientacionF=0;
                else
                    orientacionF=1;
                end

    if (orientacion1==orientacionA && orientacion1==orientacionB &&...
...orientacion1==orientacionC)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara1(:,3));
if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end

elseif (orientacion2==orientacionD && orientacion2==orientacionE &&...
...orientacion2==orientacionF)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara1(:,3));

if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end

elseif (orientacion_trianguloA==0 || orientacion_trianguloB==0 ||...
...orientacion_trianguloC==0 || orientacion_trianguloD==0 ||...
...orientacion_trianguloE==0 || orientacion_trianguloF==0)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
```

```
Zmin=min(cara1(:,3));

if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end

elseif isempty(vertices_ocultos)
end
end
end

if ind1==2

triangulo1=[cara2(1,:);cara2(2,:);cara2(3,:)];
triangulo2=[cara2(3,:);cara2(4,:);cara2(1,:)];

vertices_virtuales_aux=[vertices_virtuales2(1,:);
                        vertices_virtuales2(2,:);
                        vertices_virtuales2(7,:);
                        vertices_virtuales2(8,:)];

orientacion_triangulo1=(triangulo1(1,1)-triangulo1(3,1))*...
...(triangulo1(2,2)-triangulo1(3,2))-(triangulo1(1,2)-...
...triangulo1(3,2))*(triangulo1(2,1)-triangulo1(3,1));

orientacion_triangulo2=(triangulo2(1,1)-triangulo2(3,1))*...
...(triangulo2(2,2)-triangulo2(3,2))-(triangulo2(1,2)-...
...triangulo2(3,2))*(triangulo2(2,1)-triangulo2(3,1));

            if orientacion_triangulo1<0
                orientacion1=0;
            else
                orientacion1=1;
            end
            if orientacion_triangulo2<0
                orientacion2=0;
            else
                orientacion2=1;
            end

            for ind2=1:length(vertices_virtuales_aux(:,1))


trianguloA=[cara2(1,:);cara2(2,:);vertices_virtuales_aux(ind2,:)];
trianguloB=[cara2(2,:);cara2(3,:);vertices_virtuales_aux(ind2,:)];
trianguloC=[cara2(3,:);cara2(1,:);vertices_virtuales_aux(ind2,:)];
trianguloD=[cara2(3,:);cara2(4,:);vertices_virtuales_aux(ind2,:)];
trianguloE=[cara2(4,:);cara2(1,:);vertices_virtuales_aux(ind2,:)];
trianguloF=[cara2(1,:);cara2(3,:);vertices_virtuales_aux(ind2,:)];

orientacion_trianguloA=(trianguloA(1,1)-trianguloA(3,1))*...
...(trianguloA(2,2)-trianguloA(3,2))-(trianguloA(1,2)-...
...trianguloA(3,2))*(trianguloA(2,1)-trianguloA(3,1));

orientacion_trianguloB=(trianguloB(1,1)-trianguloB(3,1))*...
...(trianguloB(2,2)-trianguloB(3,2))-(trianguloB(1,2)-...
...trianguloB(3,2))*(trianguloB(2,1)-trianguloB(3,1));

orientacion_trianguloC=(trianguloC(1,1)-trianguloC(3,1))*...
```

```
...(trianguloC(2,2)-trianguloC(3,2))-(trianguloC(1,2)-...
...trianguloC(3,2))*(trianguloC(2,1)-trianguloC(3,1));

orientacion_trianguloD=(trianguloD(1,1)-trianguloD(3,1))*...
...(trianguloD(2,2)-trianguloD(3,2))-(trianguloD(1,2)-...
...trianguloD(3,2))*(trianguloD(2,1)-trianguloD(3,1));

orientacion_trianguloE=(trianguloE(1,1)-trianguloE(3,1))*...
...(trianguloE(2,2)-trianguloE(3,2))-(trianguloE(1,2)-...
......trianguloE(3,2))*(trianguloE(2,1)-trianguloE(3,1));

orientacion_trianguloF=(trianguloF(1,1)-trianguloF(3,1))*...
...(trianguloF(2,2)-trianguloF(3,2))-(trianguloF(1,2)-...
...trianguloF(3,2))*(trianguloF(2,1)-trianguloF(3,1));

                if orientacion_trianguloA<0
                    orientacionA=0;
                else
                    orientacionA=1;
                end
                if orientacion_trianguloB<0
                    orientacionB=0;
                else
                    orientacionB=1;
                end
                if orientacion_trianguloC<0
                    orientacionC=0;
                else
                    orientacionC=1;
                end
                if orientacion_trianguloD<0
                    orientacionD=0;
                else
                    orientacionD=1;
                end
                if orientacion_trianguloE<0
                    orientacionE=0;
                else
                    orientacionE=1;
                end
                if orientacion_trianguloF<0
                    orientacionF=0;
                else
                    orientacionF=1;
                end

if (orientacion1==orientacionA && orientacion1==orientacionB &&...
...orientacion1==orientacionC)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara2(:,3));

if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end

elseif (orientacion2==orientacionD && orientacion2==orientacionE &&...
...orientacion2==orientacionF)
```

```
posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara2(:,3));

if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end

elseif (orientacion_trianguloA==0 || orientacion_trianguloB==0 ||...
...orientacion_trianguloC==0 || orientacion_trianguloD==0 ||...
...orientacion_trianguloE==0 || orientacion_trianguloF==0)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara2(:,3));

if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end

elseif isempty(vertices_ocultos)
end
end
end

if ind1==3
            triangulo1=[cara3(1,:);cara3(2,:);cara3(3,:)];
            triangulo2=[cara3(3,:);cara3(4,:);cara3(1,:)];

            vertices_virtuales_aux=[vertices_virtuales2(1,:);
                                    vertices_virtuales2(2,:);
                                    vertices_virtuales2(3,:);
                                    vertices_virtuales2(4,:)];

orientacion_triangulo1=(triangulo1(1,1)-triangulo1(3,1))*...
...(triangulo1(2,2)-triangulo1(3,2))-(triangulo1(1,2)-...
...triangulo1(3,2))*(triangulo1(2,1)-triangulo1(3,1));

orientacion_triangulo2=(triangulo2(1,1)-triangulo2(3,1))*...
...(triangulo2(2,2)-triangulo2(3,2))-(triangulo2(1,2)-...
...triangulo2(3,2))*(triangulo2(2,1)-triangulo2(3,1));

            if orientacion_triangulo1<0
                orientacion1=0;
            else
                orientacion1=1;
            end
            if orientacion_triangulo2<0
                orientacion2=0;
            else
                orientacion2=1;
            end

            for ind2=1:length(vertices_virtuales_aux(:,1))


trianguloA=[cara3(1,:);cara3(2,:);vertices_virtuales_aux(ind2,:)];

trianguloB=[cara3(2,:);cara3(3,:);vertices_virtuales_aux(ind2,:)];
```

```
trianguloC=[cara3(3,:);cara3(1,:);vertices_virtuales_aux(ind2,:)];


trianguloD=[cara3(3,:);cara3(4,:);vertices_virtuales_aux(ind2,:)];

trianguloE=[cara3(4,:);cara3(1,:);vertices_virtuales_aux(ind2,:)];

trianguloF=[cara3(1,:);cara3(3,:);vertices_virtuales_aux(ind2,:)];

orientacion_trianguloA=(trianguloA(1,1)-trianguloA(3,1))*...
...(trianguloA(2,2)-trianguloA(3,2))-(trianguloA(1,2)-...
...trianguloA(3,2))*(trianguloA(2,1)-trianguloA(3,1));

orientacion_trianguloB=(trianguloB(1,1)-trianguloB(3,1))*...
...(trianguloB(2,2)-trianguloB(3,2))-(trianguloB(1,2)-...
...trianguloB(3,2))*(trianguloB(2,1)-trianguloB(3,1));

orientacion_trianguloC=(trianguloC(1,1)-trianguloC(3,1))*...
...(trianguloC(2,2)-trianguloC(3,2))-(trianguloC(1,2)-...
...trianguloC(3,2))*(trianguloC(2,1)-trianguloC(3,1));

orientacion_trianguloD=(trianguloD(1,1)-trianguloD(3,1))*...
...(trianguloD(2,2)-trianguloD(3,2))-(trianguloD(1,2)-...
...trianguloD(3,2))*(trianguloD(2,1)-trianguloD(3,1));

orientacion_trianguloE=(trianguloE(1,1)-trianguloE(3,1))*...
...(trianguloE(2,2)-trianguloE(3,2))-(trianguloE(1,2)-...
...trianguloE(3,2))*(trianguloE(2,1)-trianguloE(3,1));

orientacion_trianguloF=(trianguloF(1,1)-trianguloF(3,1))*...
...(trianguloF(2,2)-trianguloF(3,2))-(trianguloF(1,2)-...
...trianguloF(3,2))*(trianguloF(2,1)-trianguloF(3,1));

                if orientacion_trianguloA<0
                    orientacionA=0;
                else
                    orientacionA=1;
                end
                if orientacion_trianguloB<0
                    orientacionB=0;
                else
                    orientacionB=1;
                end
                if orientacion_trianguloC<0
                    orientacionC=0;
                else
                    orientacionC=1;
                end
                if orientacion_trianguloD<0
                    orientacionD=0;
                else
                    orientacionD=1;
                end
                if orientacion_trianguloE<0
                    orientacionE=0;
                else
                    orientacionE=1;
                end
                if orientacion_trianguloF<0
                    orientacionF=0;
```

```
                else
                    orientacionF=1;
                end

if (orientacion1==orientacionA && orientacion1==orientacionB &&...
...orientacion1==orientacionC)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara3(:,3));

if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end

elseif (orientacion2==orientacionD && orientacion2==orientacionE &&...
...orientacion2==orientacionF)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara3(:,3));

if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end

elseif (orientacion_trianguloA==0 || orientacion_trianguloB==0 ||...
...orientacion_trianguloC==0 || orientacion_trianguloD==0 ||...
...orientacion_trianguloE==0 || orientacion_trianguloF==0)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara3(:,3));

if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end
elseif isempty(vertices_ocultos)
end
end
end

if ind1==4
            triangulo1=[cara4(1,:);cara4(2,:);cara4(3,:)];
            triangulo2=[cara4(3,:);cara4(4,:);cara4(1,:)];

            vertices_virtuales_aux=[vertices_virtuales2(3,:);
                                    vertices_virtuales2(4,:);
                                    vertices_virtuales2(5,:);
                                    vertices_virtuales2(6,:)];

orientacion_triangulo1=(triangulo1(1,1)-triangulo1(3,1))*...
...(triangulo1(2,2)-triangulo1(3,2))-(triangulo1(1,2)-...
...triangulo1(3,2))*(triangulo1(2,1)-triangulo1(3,1));

orientacion_triangulo2=(triangulo2(1,1)-triangulo2(3,1))*...
...(triangulo2(2,2)-triangulo2(3,2))-(triangulo2(1,2)-...
...triangulo2(3,2))*(triangulo2(2,1)-triangulo2(3,1));

            if orientacion_triangulo1<0
                orientacion1=0;
```

```
            else
                orientacion1=1;
            end
            if orientacion_triangulo2<0
                orientacion2=0;
            else
                orientacion2=1;
            end

            for ind2=1:length(vertices_virtuales_aux(:,1))


trianguloA=[cara4(1,:);cara4(2,:);vertices_virtuales_aux(ind2,:)];
trianguloB=[cara4(2,:);cara4(3,:);vertices_virtuales_aux(ind2,:)];
trianguloC=[cara4(3,:);cara4(1,:);vertices_virtuales_aux(ind2,:)];


trianguloD=[cara4(3,:);cara4(4,:);vertices_virtuales_aux(ind2,:)];
trianguloE=[cara4(4,:);cara4(1,:);vertices_virtuales_aux(ind2,:)];
trianguloF=[cara4(1,:);cara4(3,:);vertices_virtuales_aux(ind2,:)];

orientacion_trianguloA=(trianguloA(1,1)-trianguloA(3,1))*...
...(trianguloA(2,2)-trianguloA(3,2))-(trianguloA(1,2)-...
...trianguloA(3,2))*(trianguloA(2,1)-trianguloA(3,1));

orientacion_trianguloB=(trianguloB(1,1)-trianguloB(3,1))* ...
...(trianguloB(2,2)-trianguloB(3,2))-(trianguloB(1,2)- ...
...trianguloB(3,2))*(trianguloB(2,1)-trianguloB(3,1));

orientacion_trianguloC=(trianguloC(1,1)-trianguloC(3,1))* ...
...(trianguloC(2,2)-trianguloC(3,2))-(trianguloC(1,2)- ...
...trianguloC(3,2))*(trianguloC(2,1)-trianguloC(3,1));

orientacion_trianguloD=(trianguloD(1,1)-trianguloD(3,1))* ...
...(trianguloD(2,2)-trianguloD(3,2))-(trianguloD(1,2)- ...
...trianguloD(3,2))*(trianguloD(2,1)-trianguloD(3,1));

orientacion_trianguloE=(trianguloE(1,1)-trianguloE(3,1))* ...
...(trianguloE(2,2)-trianguloE(3,2))-(trianguloE(1,2)- ...
...trianguloE(3,2))*(trianguloE(2,1)-trianguloE(3,1));

orientacion_trianguloF=(trianguloF(1,1)-trianguloF(3,1))* ...
...(trianguloF(2,2)-trianguloF(3,2))-(trianguloF(1,2)- ...
...trianguloF(3,2))*(trianguloF(2,1)-trianguloF(3,1));

                if orientacion_trianguloA<0
                    orientacionA=0;
                else
                    orientacionA=1;
                end
                if orientacion_trianguloB<0
                    orientacionB=0;
                else
                    orientacionB=1;
                end
                if orientacion_trianguloC<0
                    orientacionC=0;
                else
                    orientacionC=1;
                end
```

```
                if orientacion_trianguloD<0
                    orientacionD=0;
                else
                    orientacionD=1;
                end
                if orientacion_trianguloE<0
                    orientacionE=0;
                else
                    orientacionE=1;
                end
                if orientacion_trianguloF<0
                    orientacionF=0;
                else
                    orientacionF=1;
                end

                if (orientacion1==orientacionA &&...
...orientacion1==orientacionB && orientacion1==orientacionC)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara4(:,3));
if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end
elseif (orientacion2==orientacionD && orientacion2==orientacionE &&...
... orientacion2==orientacionF)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara4(:,3));
if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end
elseif (orientacion_trianguloA==0 || orientacion_trianguloB==0 ||...
... orientacion_trianguloC==0 || orientacion_trianguloD==0 ||...
... orientacion_trianguloE==0 || orientacion_trianguloF==0)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara4(:,3));
if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end
elseif isempty(vertices_ocultos)
end
end
end

if ind1==5
            triangulo1=[cara5(1,:);cara5(2,:);cara5(3,:)];
            triangulo2=[cara5(3,:);cara5(4,:);cara5(1,:)];

            vertices_virtuales_aux=[vertices_virtuales2(2,:);
                                    vertices_virtuales2(4,:);
                                    vertices_virtuales2(6,:);
                                    vertices_virtuales2(8,:)];

orientacion_triangulo1=(triangulo1(1,1)-triangulo1(3,1))* ...
...(triangulo1(2,2)-triangulo1(3,2))-(triangulo1(1,2)- ...
...triangulo1(3,2))*(triangulo1(2,1)-triangulo1(3,1));
```

```
orientacion_triangulo2=(triangulo2(1,1)-triangulo2(3,1))* ...
...(triangulo2(2,2)-triangulo2(3,2))-(triangulo2(1,2)- ...
...triangulo2(3,2))*(triangulo2(2,1)-triangulo2(3,1));

            if orientacion_triangulo1<0
                orientacion1=0;
            else
                orientacion1=1;
            end
            if orientacion_triangulo2<0
                orientacion2=0;
            else
                orientacion2=1;
            end

            for ind2=1:length(vertices_virtuales_aux(:,1))


trianguloA=[cara5(1,:);cara5(2,:);vertices_virtuales_aux(ind2,:)];
trianguloB=[cara5(2,:);cara5(3,:);vertices_virtuales_aux(ind2,:)];
trianguloC=[cara5(3,:);cara5(1,:);vertices_virtuales_aux(ind2,:)];

trianguloD=[cara5(3,:);cara5(4,:);vertices_virtuales_aux(ind2,:)];
trianguloE=[cara5(4,:);cara5(1,:);vertices_virtuales_aux(ind2,:)];
trianguloF=[cara5(1,:);cara5(3,:);vertices_virtuales_aux(ind2,:)];

orientacion_trianguloA=(trianguloA(1,1)-trianguloA(3,1))* ...
...(trianguloA(2,2)-trianguloA(3,2))-(trianguloA(1,2)- ...
...trianguloA(3,2))*(trianguloA(2,1)-trianguloA(3,1));

orientacion_trianguloB=(trianguloB(1,1)-trianguloB(3,1))* ...
...(trianguloB(2,2)-trianguloB(3,2))-(trianguloB(1,2)- ...
...trianguloB(3,2))*(trianguloB(2,1)-trianguloB(3,1));

orientacion_trianguloC=(trianguloC(1,1)-trianguloC(3,1))* ...
...(trianguloC(2,2)-trianguloC(3,2))-(trianguloC(1,2)- ...
...trianguloC(3,2))*(trianguloC(2,1)-trianguloC(3,1));

orientacion_trianguloD=(trianguloD(1,1)-trianguloD(3,1))* ...
...(trianguloD(2,2)-trianguloD(3,2))-(trianguloD(1,2)- ...
...trianguloD(3,2))*(trianguloD(2,1)-trianguloD(3,1));

orientacion_trianguloE=(trianguloE(1,1)-trianguloE(3,1))* ...
...(trianguloE(2,2)-trianguloE(3,2))-(trianguloE(1,2)- ...
...trianguloE(3,2))*(trianguloE(2,1)-trianguloE(3,1));

orientacion_trianguloF=(trianguloF(1,1)-trianguloF(3,1))* ...
...(trianguloF(2,2)-trianguloF(3,2))-(trianguloF(1,2)- ...
...trianguloF(3,2))*(trianguloF(2,1)-trianguloF(3,1));

                if orientacion_trianguloA<0
                    orientacionA=0;
                else
                    orientacionA=1;
                end
                if orientacion_trianguloB<0
                    orientacionB=0;
                else
                    orientacionB=1;
```

```
                end
                if orientacion_trianguloC<0
                    orientacionC=0;
                else
                    orientacionC=1;
                end
                if orientacion_trianguloD<0
                    orientacionD=0;
                else
                    orientacionD=1;
                end
                if orientacion_trianguloE<0
                    orientacionE=0;
                else
                    orientacionE=1;
                end
                if orientacion_trianguloF<0
                    orientacionF=0;
                else
                    orientacionF=1;
                end

if (orientacion1==orientacionA && orientacion1==orientacionB && ...
...orientacion1==orientacionC)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara5(:,3));

if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end

elseif (orientacion2==orientacionD && orientacion2==orientacionE &&...
... orientacion2==orientacionF)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara5(:,3));
if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end
elseif (orientacion_trianguloA==0 || orientacion_trianguloB==0 ||...
... orientacion_trianguloC==0 || orientacion_trianguloD==0 ||...
... orientacion_trianguloE==0 || orientacion_trianguloF==0)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara5(:,3));
if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end
elseif isempty(vertices_ocultos)
end
end
end

         if ind1==6
            triangulo1=[cara6(1,:);cara6(2,:);cara6(3,:)];
            triangulo2=[cara6(3,:);cara6(4,:);cara6(1,:)];
```

```
            vertices_virtuales_aux=[vertices_virtuales2(1,:);
                                    vertices_virtuales2(3,:);
                                    vertices_virtuales2(5,:);
                                    vertices_virtuales2(7,:)];

orientacion_triangulo1=(triangulo1(1,1)-triangulo1(3,1))* ...
...(triangulo1(2,2)-triangulo1(3,2))-(triangulo1(1,2)- ...
...triangulo1(3,2))*(triangulo1(2,1)-triangulo1(3,1));

orientacion_triangulo2=(triangulo2(1,1)-triangulo2(3,1))* ...
...(triangulo2(2,2)-triangulo2(3,2))-(triangulo2(1,2)- ...
...triangulo2(3,2))*(triangulo2(2,1)-triangulo2(3,1));

            if orientacion_triangulo1<0
                orientacion1=0;
            else
                orientacion1=1;
            end
            if orientacion_triangulo2<0
                orientacion2=0;
            else
                orientacion2=1;
            end

            for ind2=1:length(vertices_virtuales_aux(:,1))


trianguloA=[cara6(1,:);cara6(2,:);vertices_virtuales_aux(ind2,:)];
trianguloB=[cara6(2,:);cara6(3,:);vertices_virtuales_aux(ind2,:)];
trianguloC=[cara6(3,:);cara6(1,:);vertices_virtuales_aux(ind2,:)];

trianguloD=[cara6(3,:);cara6(4,:);vertices_virtuales_aux(ind2,:)];
trianguloE=[cara6(4,:);cara6(1,:);vertices_virtuales_aux(ind2,:)];
trianguloF=[cara6(1,:);cara6(3,:);vertices_virtuales_aux(ind2,:)];

orientacion_trianguloA=(trianguloA(1,1)-trianguloA(3,1))* ...
...(trianguloA(2,2)-trianguloA(3,2))-(trianguloA(1,2)- ...
...trianguloA(3,2))*(trianguloA(2,1)-trianguloA(3,1));

orientacion_trianguloB=(trianguloB(1,1)-trianguloB(3,1))* ...
...(trianguloB(2,2)-trianguloB(3,2))-(trianguloB(1,2)- ...
...trianguloB(3,2))*(trianguloB(2,1)-trianguloB(3,1));

orientacion_trianguloC=(trianguloC(1,1)-trianguloC(3,1))* ...
...(trianguloC(2,2)-trianguloC(3,2))-(trianguloC(1,2)- ...
...trianguloC(3,2))*(trianguloC(2,1)-trianguloC(3,1));

orientacion_trianguloD=(trianguloD(1,1)-trianguloD(3,1))* ...
...(trianguloD(2,2)-trianguloD(3,2))-(trianguloD(1,2)- ...
...trianguloD(3,2))*(trianguloD(2,1)-trianguloD(3,1));

orientacion_trianguloE=(trianguloE(1,1)-trianguloE(3,1))* ...
...(trianguloE(2,2)-trianguloE(3,2))-(trianguloE(1,2)- ...
...trianguloE(3,2))*(trianguloE(2,1)-trianguloE(3,1));

orientacion_trianguloF=(trianguloF(1,1)-trianguloF(3,1))* ...
...(trianguloF(2,2)-trianguloF(3,2))-(trianguloF(1,2)- ...
...trianguloF(3,2))*(trianguloF(2,1)-trianguloF(3,1));

                if orientacion_trianguloA<0
```

```
                    orientacionA=0;
                else
                    orientacionA=1;
                end
                if orientacion_trianguloB<0
                    orientacionB=0;
                else
                    orientacionB=1;
                end
                if orientacion_trianguloC<0
                    orientacionC=0;
                else
                    orientacionC=1;
                end
                if orientacion_trianguloD<0
                    orientacionD=0;
                else
                    orientacionD=1;
                end
                if orientacion_trianguloE<0
                    orientacionE=0;
                else
                    orientacionE=1;
                end
                if orientacion_trianguloF<0
                    orientacionF=0;
                else
                    orientacionF=1;
                end

if (orientacion1==orientacionA && orientacion1==orientacionB && ...
...orientacion1==orientacionC)
posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara6(:,3));

if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end
elseif (orientacion2==orientacionD && orientacion2==orientacionE &&...
...orientacion2==orientacionF)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara6(:,3));
if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end
elseif (orientacion_trianguloA==0 || orientacion_trianguloB==0 ||...
...orientacion_trianguloC==0 || orientacion_trianguloD==0 || ...
...orientacion_trianguloE==0 || orientacion_trianguloF==0)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara6(:,3));
if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end
elseif isempty(vertices_ocultos)
end
end
```

```
end
end

         % Vamos a buscar qué vértice de los 8 que hay es el oculto

posicion_vertice_virtual_oculto=find(vertices_virtuales2(:,1)==vertice
s_ocultos(:,1) & vertices_virtuales2(:,2)==vertices_ocultos(:,2));

%AHORA RELLENAMOS LA FIGURA Y DIBUJAMOS LAS LINEAS SEGÚN QUÉ VERTICE
SEA EL OCULTO (puede ser de 1 a 8).
```

El código fuente del relleno y el lineado ha sido omitido por su larga extensión. Se utilizan los comandos plot y fill para dibujar y rellenar las caras del objeto según el vértice oculto.

```
elseif figura==0

    num_caras=5;

    cara1=[vertices_virtuales2(1,:);vertices_virtuales2(2,:); ...
...vertices_virtuales2(5,:)];
    cara2=[vertices_virtuales2(5,:);vertices_virtuales2(2,:); ...
...vertices_virtuales2(3,:)];
    cara3=[vertices_virtuales2(4,:);vertices_virtuales2(2,:); ...
...vertices_virtuales2(1,:)];
    cara4=[vertices_virtuales2(4,:);vertices_virtuales2(2,:); ...
...vertices_virtuales2(3,:)];
    cara5=[vertices_virtuales2(1,:);vertices_virtuales2(4,:); ...
...vertices_virtuales2(3,:);vertices_virtuales2(5,:)];
    vertices_ocultos=[];

    for ind1=1:num_caras

         if ind1==1
            triangulo1=[cara1(1,:);cara1(2,:);cara1(3,:)];

            vertices_virtuales_aux=[vertices_virtuales2(3,:);
                                    vertices_virtuales2(4,:)];

orientacion_triangulo1=(triangulo1(1,1)-triangulo1(3,1))* ...
...(triangulo1(2,2)-triangulo1(3,2))-(triangulo1(1,2)- ...
...triangulo1(3,2))*(triangulo1(2,1)-triangulo1(3,1));

            if orientacion_triangulo1<0
                orientacion1=0;
            else
                orientacion1=1;
            end

            for ind2=1:length(vertices_virtuales_aux(:,1))


trianguloA=[cara1(1,:);cara1(2,:);vertices_virtuales_aux(ind2,:)];
trianguloB=[cara1(2,:);cara1(3,:);vertices_virtuales_aux(ind2,:)];
trianguloC=[cara1(3,:);cara1(1,:);vertices_virtuales_aux(ind2,:)];
```

```
orientacion_trianguloA=(trianguloA(1,1)-trianguloA(3,1))* ...
...(trianguloA(2,2)-trianguloA(3,2))-(trianguloA(1,2)- ...
...trianguloA(3,2))*(trianguloA(2,1)-trianguloA(3,1));

orientacion_trianguloB=(trianguloB(1,1)-trianguloB(3,1))* ...
...(trianguloB(2,2)-trianguloB(3,2))-(trianguloB(1,2)- ...
...trianguloB(3,2))*(trianguloB(2,1)-trianguloB(3,1));

orientacion_trianguloC=(trianguloC(1,1)-trianguloC(3,1))* ...
...(trianguloC(2,2)-trianguloC(3,2))-(trianguloC(1,2)- ...
...trianguloC(3,2))*(trianguloC(2,1)-trianguloC(3,1));

                if orientacion_trianguloA<0
                    orientacionA=0;
                else
                    orientacionA=1;
                end
                if orientacion_trianguloB<0
                    orientacionB=0;
                else
                    orientacionB=1;
                end
                if orientacion_trianguloC<0
                    orientacionC=0;
                else
                    orientacionC=1;
                end

if (orientacion1==orientacionA && orientacion1==orientacionB &&...
... orientacion1==orientacionC)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara1(:,3));

if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end
elseif isempty(vertices_ocultos)
end
end
end

if ind1==2
    triangulo1=[cara2(1,:);cara2(2,:);cara2(3,:)];

            vertices_virtuales_aux=[vertices_virtuales2(1,:);
                                    vertices_virtuales2(4,:)];

orientacion_triangulo1=(triangulo1(1,1)-triangulo1(3,1))* ...
...(triangulo1(2,2)-triangulo1(3,2))-(triangulo1(1,2)- ...
...triangulo1(3,2))*(triangulo1(2,1)-triangulo1(3,1));
            if orientacion_triangulo1<0
                orientacion1=0;
            else
                orientacion1=1;
            end


            for ind2=1:length(vertices_virtuales_aux(:,1))
```

```
trianguloA=[cara2(1,:);cara2(2,:);vertices_virtuales_aux(ind2,:)];
trianguloB=[cara2(2,:);cara2(3,:);vertices_virtuales_aux(ind2,:)];
trianguloC=[cara2(3,:);cara2(1,:);vertices_virtuales_aux(ind2,:)];


orientacion_trianguloA=(trianguloA(1,1)-trianguloA(3,1))* ...
...(trianguloA(2,2)-trianguloA(3,2))-(trianguloA(1,2)- ...
...trianguloA(3,2))*(trianguloA(2,1)-trianguloA(3,1));

orientacion_trianguloB=(trianguloB(1,1)-trianguloB(3,1))* ...
...(trianguloB(2,2)-trianguloB(3,2))-(trianguloB(1,2)- ...
...trianguloB(3,2))*(trianguloB(2,1)-trianguloB(3,1));

orientacion_trianguloC=(trianguloC(1,1)-trianguloC(3,1))* ...
...(trianguloC(2,2)-trianguloC(3,2))-(trianguloC(1,2)- ...
...trianguloC(3,2))*(trianguloC(2,1)-trianguloC(3,1));


                if orientacion_trianguloA<0
                    orientacionA=0;
                else
                    orientacionA=1;
                end
                if orientacion_trianguloB<0
                    orientacionB=0;
                else
                    orientacionB=1;
                end
                if orientacion_trianguloC<0
                    orientacionC=0;
                else
                    orientacionC=1;
                end


if (orientacion1==orientacionA && orientacion1==orientacionB &&...
... orientacion1==orientacionC)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara2(:,3));

if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end
elseif isempty(vertices_ocultos)
end
end

end

if ind1==3
    triangulo1=[cara3(1,:);cara3(2,:);cara3(3,:)];


   vertices_virtuales_aux=[vertices_virtuales2(3,:);
                            vertices_virtuales2(5,:)];
```

```
orientacion_triangulo1=(triangulo1(1,1)-triangulo1(3,1))* ...
... (triangulo1(2,2)-triangulo1(3,2))-(triangulo1(1,2)- ...
...triangulo1(3,2))*(triangulo1(2,1)-triangulo1(3,1));

            if orientacion_triangulo1<0
                orientacion1=0;
            else
                orientacion1=1;
            end

            for ind2=1:length(vertices_virtuales_aux(:,1))


trianguloA=[cara3(1,:);cara3(2,:);vertices_virtuales_aux(ind2,:)];
trianguloB=[cara3(2,:);cara3(3,:);vertices_virtuales_aux(ind2,:)];
trianguloC=[cara3(3,:);cara3(1,:);vertices_virtuales_aux(ind2,:)];


orientacion_trianguloA=(trianguloA(1,1)-trianguloA(3,1))* ...
...(trianguloA(2,2)-trianguloA(3,2))-(trianguloA(1,2)- ...
...trianguloA(3,2))*(trianguloA(2,1)-trianguloA(3,1));

orientacion_trianguloB=(trianguloB(1,1)-trianguloB(3,1))* ...
...(trianguloB(2,2)-trianguloB(3,2))-(trianguloB(1,2)- ...
...trianguloB(3,2))*(trianguloB(2,1)-trianguloB(3,1));

orientacion_trianguloC=(trianguloC(1,1)-trianguloC(3,1))* ...
...(trianguloC(2,2)-trianguloC(3,2))-(trianguloC(1,2)- ...
...trianguloC(3,2))*(trianguloC(2,1)-trianguloC(3,1));


                if orientacion_trianguloA<0
                    orientacionA=0;
                else
                    orientacionA=1;
                end
                if orientacion_trianguloB<0
                    orientacionB=0;
                else
                    orientacionB=1;
                end
                if orientacion_trianguloC<0
                    orientacionC=0;
                else
                    orientacionC=1;
                end


if (orientacion1==orientacionA && orientacion1==orientacionB &&...
... orientacion1==orientacionC)
posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara3(:,3));
if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end
elseif isempty(vertices_ocultos)
end
```

```
end
end

if ind1==4
            triangulo1=[cara4(1,:);cara4(2,:);cara4(3,:)];

            vertices_virtuales_aux=[vertices_virtuales2(1,:);
                                    vertices_virtuales2(5,:)];

orientacion_triangulo1=(triangulo1(1,1)-triangulo1(3,1))* ...
...(triangulo1(2,2)-triangulo1(3,2))-(triangulo1(1,2)- ...
...triangulo1(3,2))*(triangulo1(2,1)-triangulo1(3,1));

            if orientacion_triangulo1<0
                orientacion1=0;
            else
                orientacion1=1;
            end

            for ind2=1:length(vertices_virtuales_aux(:,1))

trianguloA=[cara4(1,:);cara4(2,:);vertices_virtuales_aux(ind2,:)];
trianguloB=[cara4(2,:);cara4(3,:);vertices_virtuales_aux(ind2,:)];
trianguloC=[cara4(3,:);cara4(1,:);vertices_virtuales_aux(ind2,:)];

orientacion_trianguloA=(trianguloA(1,1)-trianguloA(3,1))* ...
...(trianguloA(2,2)-trianguloA(3,2))-(trianguloA(1,2)- ...
...trianguloA(3,2))*(trianguloA(2,1)-trianguloA(3,1));

orientacion_trianguloB=(trianguloB(1,1)-trianguloB(3,1))* ...
...(trianguloB(2,2)-trianguloB(3,2))-(trianguloB(1,2)- ...
...trianguloB(3,2))*(trianguloB(2,1)-trianguloB(3,1));

orientacion_trianguloC=(trianguloC(1,1)-trianguloC(3,1))* ...
...(trianguloC(2,2)-trianguloC(3,2))-(trianguloC(1,2)- ...
...trianguloC(3,2))*(trianguloC(2,1)-trianguloC(3,1));


                if orientacion_trianguloA<0
                    orientacionA=0;
                else
                    orientacionA=1;
                end
                if orientacion_trianguloB<0
                    orientacionB=0;
                else
                    orientacionB=1;
                end
                if orientacion_trianguloC<0
                    orientacionC=0;
                else
                    orientacionC=1;
                end

if (orientacion1==orientacionA && orientacion1==orientacionB &&...
... orientacion1==orientacionC)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara4(:,3));
```

```
if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end
elseif isempty(vertices_ocultos)
end
end
end

if ind1==5
            triangulo1=[cara5(1,:);cara5(2,:);cara5(3,:)];
            triangulo2=[cara5(3,:);cara5(4,:);cara5(1,:)];

            vertices_virtuales_aux=[vertices_virtuales2(2,:)];

orientacion_triangulo1=(triangulo1(1,1)-triangulo1(3,1))* ...
... (triangulo1(2,2)-triangulo1(3,2))-(triangulo1(1,2)- ...
...triangulo1(3,2))*(triangulo1(2,1)-triangulo1(3,1));

orientacion_triangulo2=(triangulo2(1,1)-triangulo2(3,1))* ...
... (triangulo2(2,2)-triangulo2(3,2))-(triangulo2(1,2)- ...
...triangulo2(3,2))*(triangulo2(2,1)-triangulo2(3,1));

            if orientacion_triangulo1<0
                orientacion1=0;
            else
                orientacion1=1;
            end
            if orientacion_triangulo2<0
                orientacion2=0;
            else
                orientacion2=1;
            end

            for ind2=1:length(vertices_virtuales_aux(:,1))


trianguloA=[cara5(1,:);cara5(2,:);vertices_virtuales_aux(ind2,:)];
trianguloB=[cara5(2,:);cara5(3,:);vertices_virtuales_aux(ind2,:)];
trianguloC=[cara5(3,:);cara5(1,:);vertices_virtuales_aux(ind2,:)];

trianguloD=[cara5(3,:);cara5(4,:);vertices_virtuales_aux(ind2,:)];
trianguloE=[cara5(4,:);cara5(1,:);vertices_virtuales_aux(ind2,:)];
trianguloF=[cara5(1,:);cara5(3,:);vertices_virtuales_aux(ind2,:)];

orientacion_trianguloA=(trianguloA(1,1)-trianguloA(3,1))* ...
... (trianguloA(2,2)-trianguloA(3,2))-(trianguloA(1,2)- ...
...trianguloA(3,2))*(trianguloA(2,1)-trianguloA(3,1));

orientacion_trianguloB=(trianguloB(1,1)-trianguloB(3,1))* ...
... (trianguloB(2,2)-trianguloB(3,2))-(trianguloB(1,2)- ...
...trianguloB(3,2))*(trianguloB(2,1)-trianguloB(3,1));
orientacion_trianguloC=(trianguloC(1,1)-trianguloC(3,1))* ...
...(trianguloC(2,2)-trianguloC(3,2))-(trianguloC(1,2)- ...
...trianguloC(3,2))*(trianguloC(2,1)-trianguloC(3,1));

orientacion_trianguloD=(trianguloD(1,1)-trianguloD(3,1))* ...
...(trianguloD(2,2)-trianguloD(3,2))-(trianguloD(1,2)- ...
...trianguloD(3,2))*(trianguloD(2,1)-trianguloD(3,1));

orientacion_trianguloE=(trianguloE(1,1)-trianguloE(3,1))* ...
```

```
... (trianguloE(2,2)-trianguloE(3,2))-(trianguloE(1,2)- ...
...trianguloE(3,2))*(trianguloE(2,1)-trianguloE(3,1));

orientacion_trianguloF=(trianguloF(1,1)-trianguloF(3,1))* ...
... (trianguloF(2,2)-trianguloF(3,2))-(trianguloF(1,2)- ...
...trianguloF(3,2))*(trianguloF(2,1)-trianguloF(3,1));

                if orientacion_trianguloA<0
                    orientacionA=0;
                else
                    orientacionA=1;
                end
                if orientacion_trianguloB<0
                    orientacionB=0;
                else
                    orientacionB=1;
                end
                if orientacion_trianguloC<0
                    orientacionC=0;
                else
                    orientacionC=1;
                end
                if orientacion_trianguloD<0
                    orientacionD=0;
                else
                    orientacionD=1;
                end
                if orientacion_trianguloE<0
                    orientacionE=0;
                else
                    orientacionE=1;
                end
                if orientacion_trianguloF<0
                    orientacionF=0;
                else
                    orientacionF=1;
                end

if (orientacion1==orientacionA && orientacion1==orientacionB &&...
... orientacion1==orientacionC)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara5(:,3));
if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end
elseif (orientacion2==orientacionD && orientacion2==orientacionE &&...
... orientacion2==orientacionF)

posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara5(:,3));
if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end
elseif (orientacion_trianguloA==0 || orientacion_trianguloB==0 ||...
... orientacion_trianguloC==0 || orientacion_trianguloD==0 ||...
...orientacion_trianguloE==0 || orientacion_trianguloF==0)
```

```
posible_vertice_oculto=vertices_virtuales_aux(ind2,:);
Zposible_vertice_oculto=posible_vertice_oculto(:,3);
Zmin=min(cara5(:,3));
if Zmin>Zposible_vertice_oculto
vertices_ocultos=[vertices_ocultos; vertices_virtuales_aux(ind2,:)];
end
elseif isempty(vertices_ocultos)
end
end
end
end

    % Vamos a buscar qué vértice de los 5 que hay es el oculto
    if isempty(vertices_ocultos)
        %si está vacio creamos también la posición vacia
        posicion_vertice_virtual_oculto=[];
    else % si no está vacio buscamos la posición del vértice oculto

posicion_vertice_virtual_oculto=find(vertices_virtuales2(:,1)==vertice
s_ocultos(:,1) & vertices_virtuales2(:,2)==vertices_ocultos(:,2));
    end

    % AHORA RELLENAMOS LA FIGURA Y DIBUJAMOS LAS LINEAS SEGÚN QUÉ
VERTICE SEA EL OCULTO(puede ser de 1 a 5) O SI NO HAY VÉRTICE OCULTO.
```

El código fuente del relleno y el lineado ha sido omitido por su larga extensión. Se utilizan los comandos plot y fill para dibujar y rellenar las caras del objeto según el vértice oculto. En caso de no existir vértice oculto se representan todas las aristas del objeto, tanto ocultas como visibles.

6. Función *buscar_picos_vector*

```
function pos_picos_out = busca_picos_vector(datos, umbral)

datos_norm = datos/max(datos);
derivada = diff(datos_norm);
signo_derivada = sign(derivada);
derivada2 = diff(signo_derivada);
pos_picos1 = find(derivada2 == -2) + 1;
pos_picos1 = pos_picos1(2:end-1);
nivel_pico = (datos_norm(pos_picos1)-...
...datos_norm(pos_picos1+1))+(datos_norm(pos_picos1)-...
...datos_norm(pos_picos1-1));
pos_picos2 = find(nivel_pico>umbral);

pos_picos_out=pos_picos1(pos_picos2);
```

7. Función *VertexTraseroDelantero*

```
Function vertice_elegido=vertexTraseroDelantero(gradiente,...
...vert_elegido)
% si vert_elegido=0 calculará el vertice delantero
% si vert_elegido=1 calculará el vertice trasero

mascara=imfill(gradiente);
proy_y=sum(mascara,2);
proy_y=proy_y./max(proy_y);
if vert_elegido==1
    pos_y=find(proy_y>0);
    pos_y(1);
    pos_x_tras=find(mascara(pos_y(1),:)==1);
    if length(pos_x_tras)>1
        tam=floor(length(pos_x_tras)/2);
        pos_x_tras=pos_x_tras(tam);
    else
    end
    vertice_elegido=[pos_x_tras pos_y(1)];
end
if vert_elegido==0
    pos_y=find(proy_y>0);
    pos_y(end)
    pos_x_del=find(mascara(pos_y(end),:)==1);
    if length(pos_x_del)>1
        tam=floor(length(pos_x_del)/2);
        pos_x_del=mean(pos_x_del);
    else
    end
    vertice_elegido=[pos_x_del pos_y(end)];
end
```

8. Función *vertexTraseroDelantero2*

```
function vertice_elegido=vertexTraseroDelantero2(gradiente,...
...vert_elegido)
% si vert_elegido=0 calculará el vertice delantero
% si vert_elegido=1 calculará el vertice trasero

mascara=imfill(gradiente);
proy_y=sum(mascara,2);
proy_y=proy_y./max(proy_y);

if vert_elegido==1
    pos_y=find(proy_y>0);
    pos_y(1);
    pos_x_tras=find(mascara(pos_y(1),:)==1);
    if length(pos_x_tras)>1
        pos_x_tras=pos_x_tras(end);
    else
    end
    vertice_elegido=[pos_x_tras pos_y(1)];
```

```
end
if vert_elegido==0
    pos_y=find(proy_y>0);
    pos_y(end)
    pos_x_del=find(mascara(pos_y(end),:)==1);
    if length(pos_x_del)>1
        tam=floor(length(pos_x_del)/2);
        pos_x_del=mean(pos_x_del);
    else
    end
    vertice_elegido=[pos_x_del pos_y(end)];
end
```

9. Función *verticeAbajoArriba*

```
function [verticeAbajo verticeArriba]=verticeAbajoArriba(gradiente...
...,orientacion)
% si orientacion=1 es la 1ª linea la que buscará el punto de abajo
% si orientación=3 es la 3ª linea la que buscará el punto de abajo

mascara=imfill(gradiente);
se=strel('square',3);
mascara=imopen(mascara,se);

proy_x=sum(mascara,1);
proy_x=proy_x./max(proy_x);
if orientacion==1
    pos_x=find(proy_x>0);
    pos_x=pos_x(3);
    pos_y=find(mascara(:,pos_x)==1);
    pos_y(end);

    verticeAbajo=[pos_x pos_y(end)];
    verticeArriba=[pos_x pos_y(1)];
end

if orientacion==3
    pos_x=find(proy_x>0);
    pos_x=pos_x(end)-3;
    pos_y=find(mascara(:,pos_x)==1);
    pos_y(end);
    verticeAbajo=[pos_x pos_y(end)];
    verticeArriba=[pos_x pos_y(1)];
end
```

ANEXO II: RESULTADOS DIFERENTES EN DIFERENTES VERSIONES DE MATLAB.

El programa *GUIVirtual3D* ha sido ejecutado en dos versiones de Matlab diferentes. Las versiones del software matemático han sido la versión R2009b y la versión R2007a, ambas instaladas en el mismo sistema operativo, siendo este Microsoft Windows 7.

De esta doble ejecución se puede concluir que el software *GUIVirtual3D* no tiene el mismo comportamiento en ambas versiones, es decir, se han obtenido diferentes resultados para las mismas imágenes de entrada, siendo los resultados de la versión R2009b de Matlab los que más se aproximan al objeto real.

Sin entender la razón por la cual los resultados son diferentes utilizando las mismas funciones de la toolbox de Matlab y las funciones propias, en las siguientes páginas se puede ver una comparación[11] con los resultados obtenidos en las dos versiones del software matemático.

[11] Esta comparación está hecha mostrando todas las aristas del modelo 3D, es decir, no hay vértices y líneas ocultas en la comparación.

ÍNDICE DE IMÁGENES

FIGURA 1: CUBO AMARILLO EN MATLAB R2007A. 140
FIGURA 2123: CUBO AMARILLO EN MATLAB R2009B. 140
FIGURA 3: CUBO AZUL EN MATLAB R2007A. 141
FIGURA 4: CUBO AZUL EN MATLAB R2009B. 141
FIGURA 5: CUBO AZUL A EN MATLAB R2007A. 142
FIGURA 6: CUBO AZUL A EN MATLAB R2009B. 142
FIGURA 7: CUBO AZUL B EN MATLAB R2007A. 143
FIGURA 8: CUBO AZUL B EN MATLAB R2009B. 143
FIGURA 9: CUBO AZUL C EN MATLAB R2007A. 144
FIGURA 10: CUBO AZUL C EN MATLAB R2009B. 144
FIGURA 11: CUBO AZUL D EN MATLAB R2007A. 145
FIGURA 12: CUBO AZUL D EN MATLAB R2009B. 145
FIGURA 13: CUBO AZUL E EN MATLAB R2007A. 146
FIGURA 14: CUBO AZUL E EN MATLAB R2009B. 146
FIGURA 15: CUBO AZUL F EN MATLAB R2007A. 147
FIGURA 16: CUBO AZUL F EN MATLAB R2009B. 147
FIGURA 17: CUBO AZUL G EN MATLAB R2007A. 148
FIGURA 18: CUBO AZUL G EN MATLAB R2009B. 148
FIGURA 19: CUBO ROJO EN MATLAB R2007A. 149
FIGURA 20: CUBO ROJO EN MATLAB R2009B. 149
FIGURA 21: CUBO VERDE EN MATLAB R2007A. 150
FIGURA 22: CUBO VERDE EN MATLAB R2009B. 150
FIGURA 23: CUBO VERDE 2 EN MATLAB R2007A. 151
FIGURA 24: CUBO VERDE 2 EN MATLAB R2009B. 151
FIGURA 25: CUBO PIRÁMIDE EN MATLAB R2007A. 152
FIGURA 26: CUBO PIRÁMIDE EN MATLAB R2009B. 152
FIGURA 27: CUBO PIRÁMIDE 2 EN MATLAB R2007A. 153
FIGURA 28: CUBO PIRÁMIDE 2 EN MATLAB R2009B. 153
FIGURA 29: CUBO PIRÁMIDE 3 EN MATLAB R2007A. 154
FIGURA 30: CUBO PIRÁMIDE 3 EN MATLAB R2009B. 154
FIGURA 31: CUBO PIRÁMIDE 4 EN MATLAB R2007A. 155
FIGURA 32: CUBO PIRÁMIDE 4 EN MATLAB R2009B. 155

En la *figura 1* se puede observar como la orientación no es muy correcta, siendo la *figura 2*, en la versión R2009b, la que más se parece al objeto real.

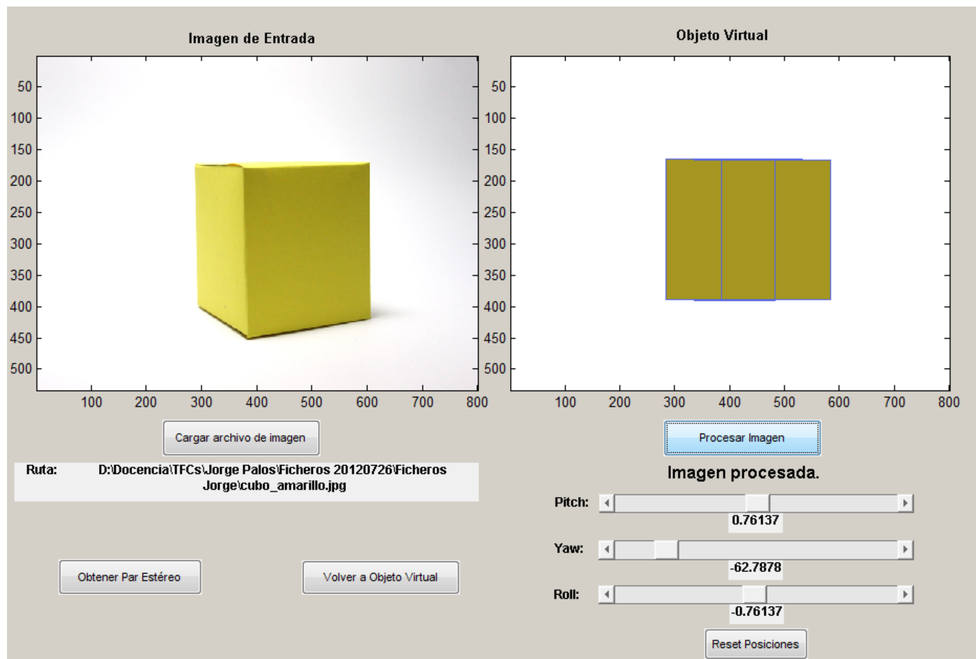

Figura 124: Cubo amarillo en Matlab R2007a.

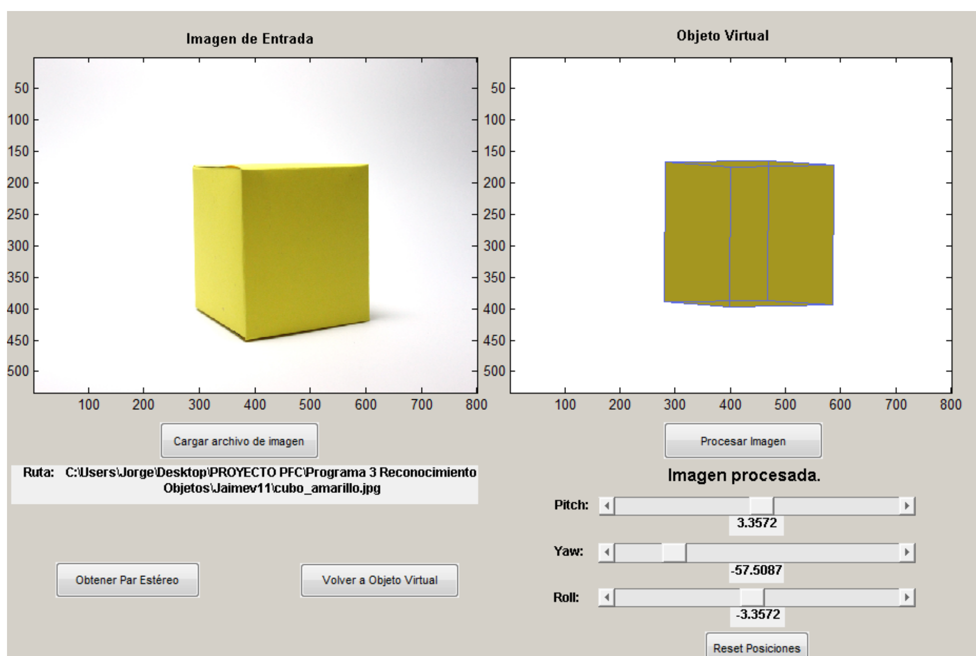

Figura 125: Cubo amarillo en Matlab R2009b.

En el caso de la imagen siguiente funciona correctamente en ambas versiones, sin embargo se puede apreciar como las rotaciones (*pitch, yaw, roll*) son diferentes para ambas versiones.

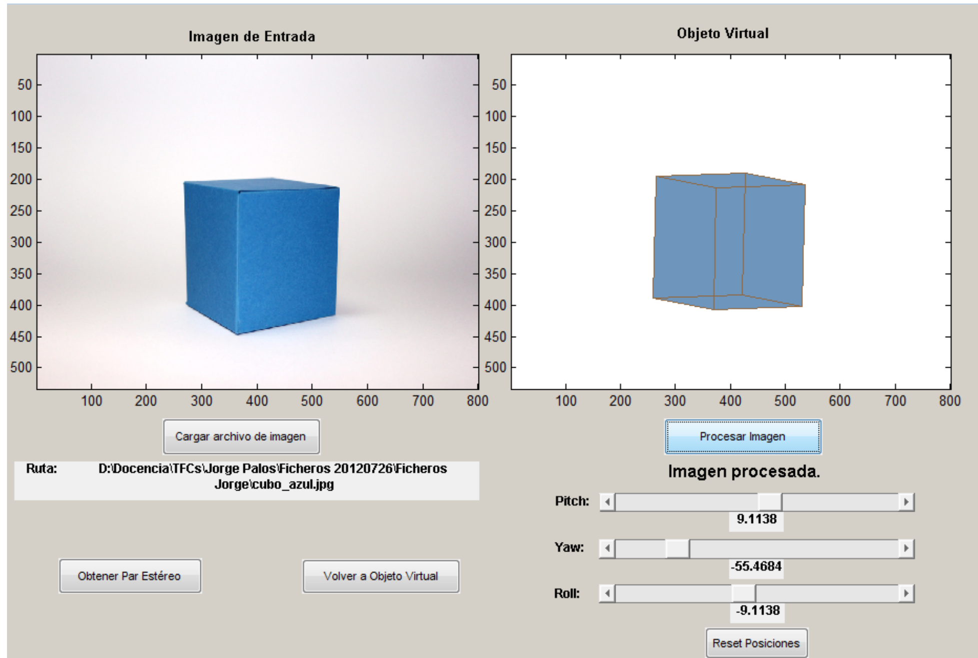

Figura 126: Cubo azul en Matlab R2007a.

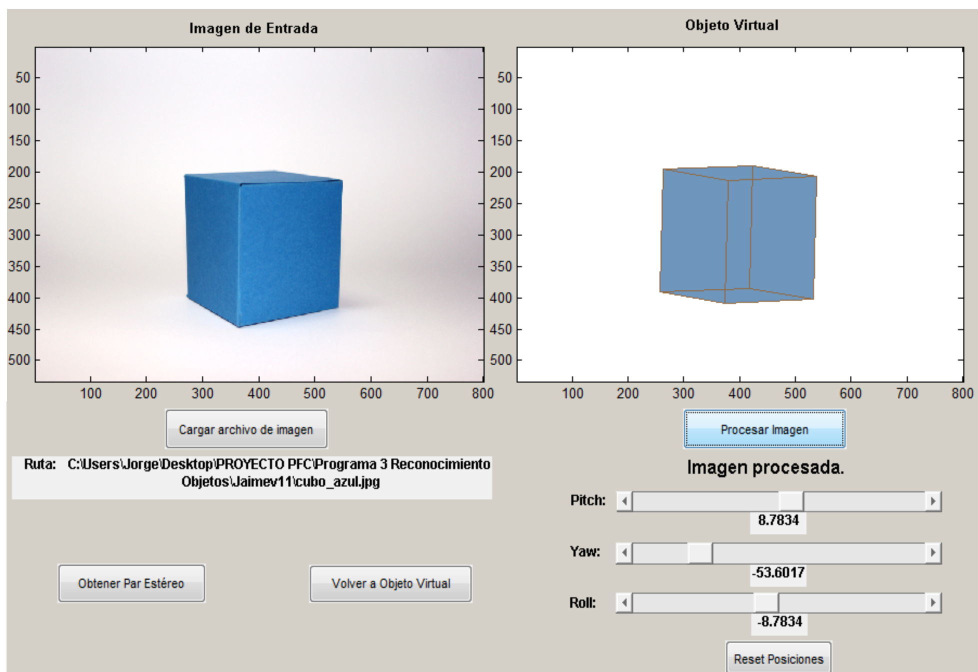

Figura 127: Cubo azul en Matlab R2009b.

En la *figura 5* se obtiene un error al procesar la imagen mientras que en la *figura 6* ha sido procesada, no obstante el procesado se ha realizado incorrectamente ya que la orientación del modelo 3D no es correcta.

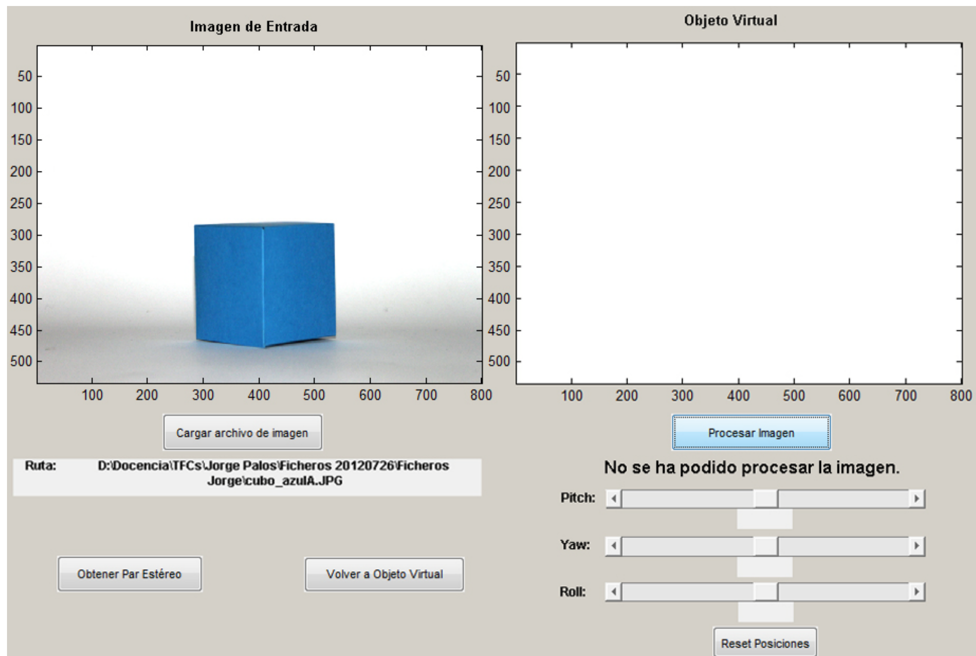

Figura 128: Cubo azul A en Matlab R2007a.

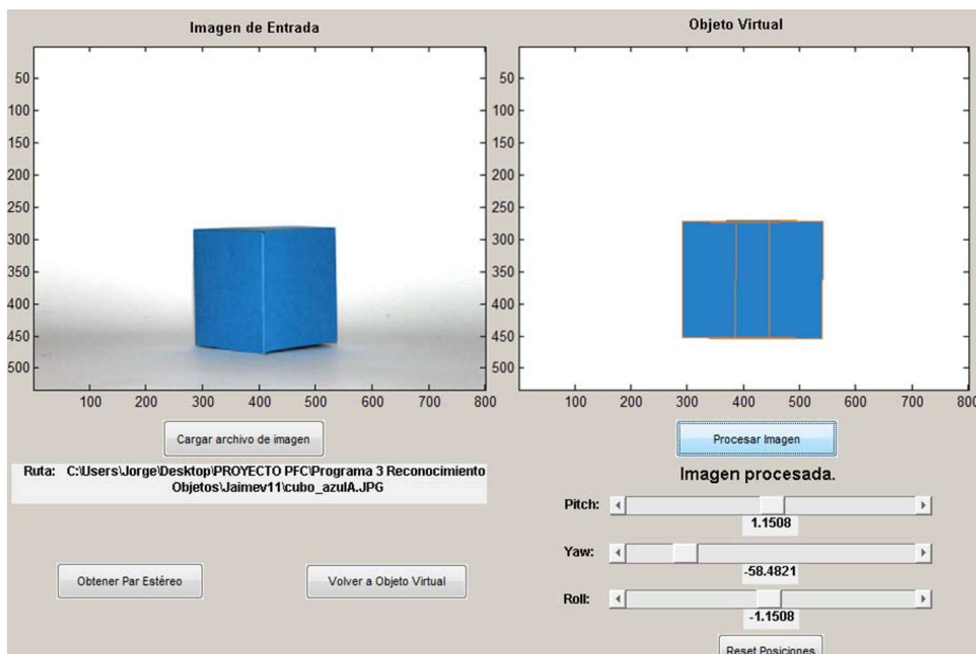

Figura 129: Cubo azul A en Matlab R2009b.

En este caso se obtiene prácticamente el mismo resultado en las dos versiones de Matlab, incluyendo los valores de orientación.

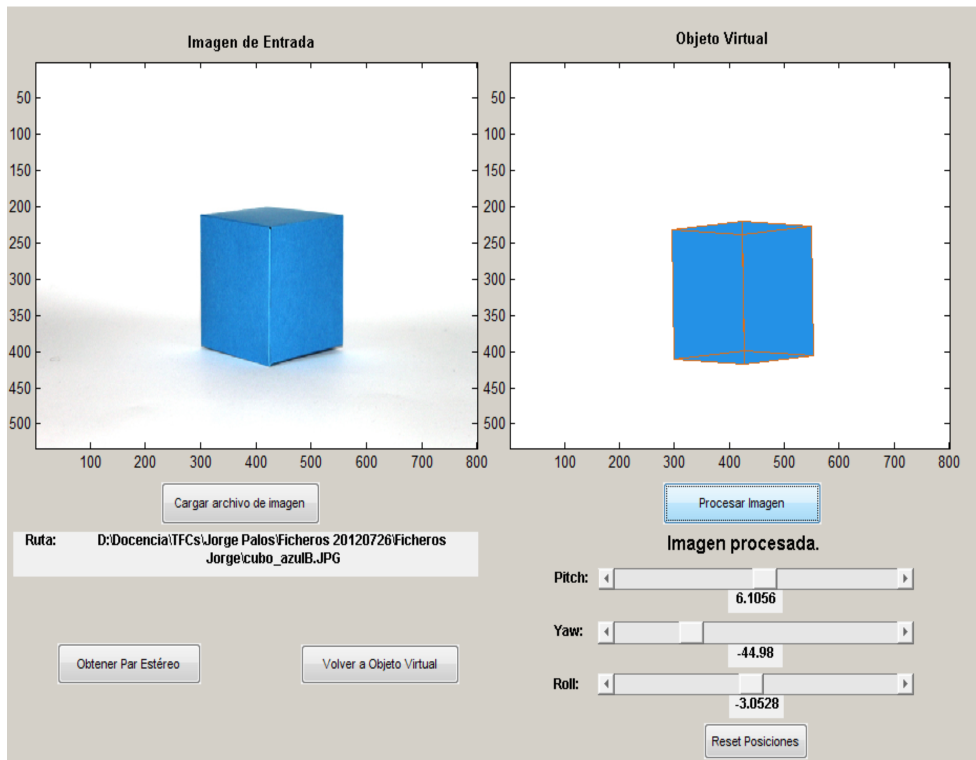

Figura 130: Cubo azul B en Matlab R2007a.

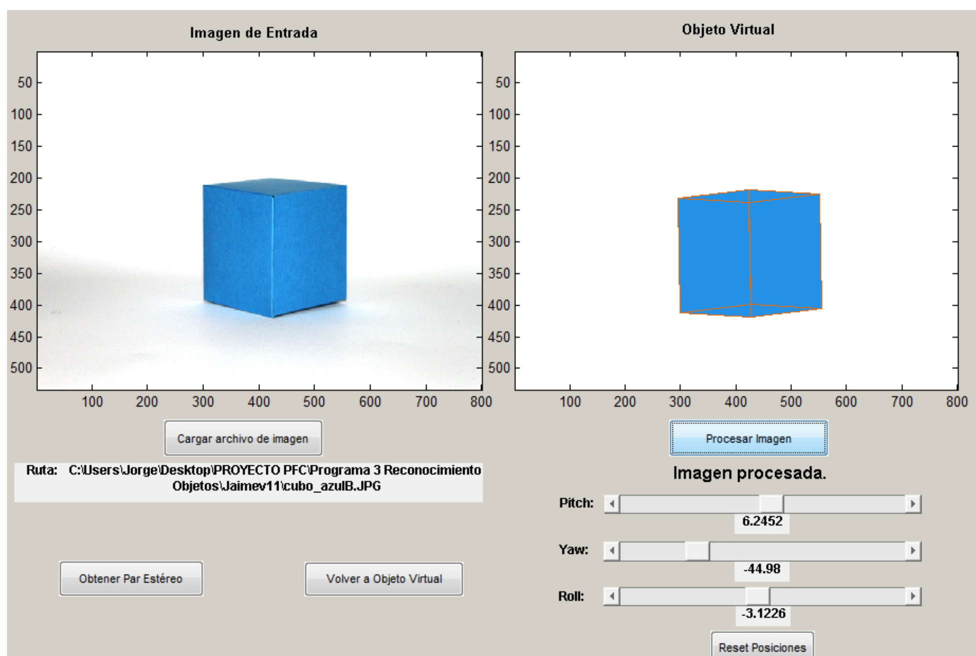

Figura 131: Cubo azul B en Matlab R2009b.

En la *figura 9* se observa como se ha procesado la imagen con valores de orientación incorrectos, mientras en la *figura 10* se ve como el programa no puede procesar a imagen y muestra de nuevo el mensaje de error.

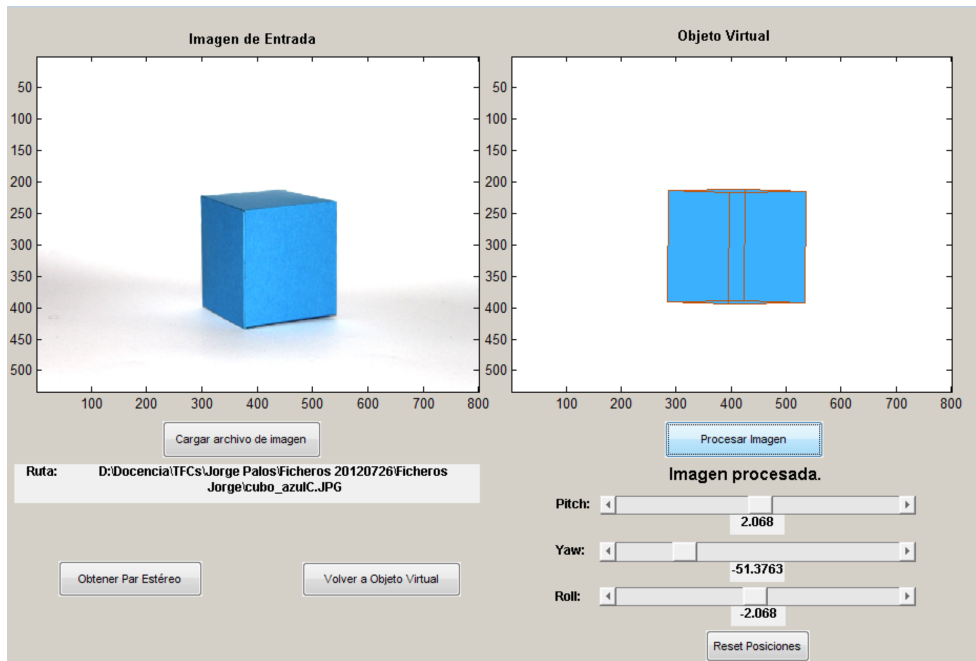

Figura 132: Cubo azul C en Matlab R2007a.

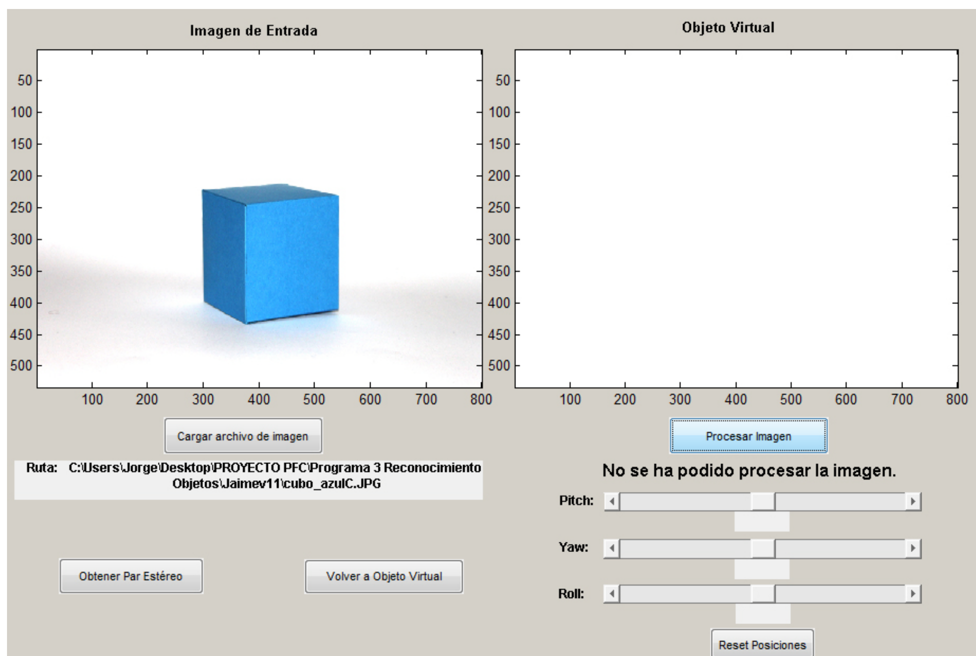

Figura 133: Cubo azul C en Matlab R2009b.

En las *figura 11* y *figura 12* se vuelve a tener el caso en que se procesa correctamente en las dos versiones aunque los valores de orientación calculados son distintos, siendo más preciso en el caso de la versión R2009b.

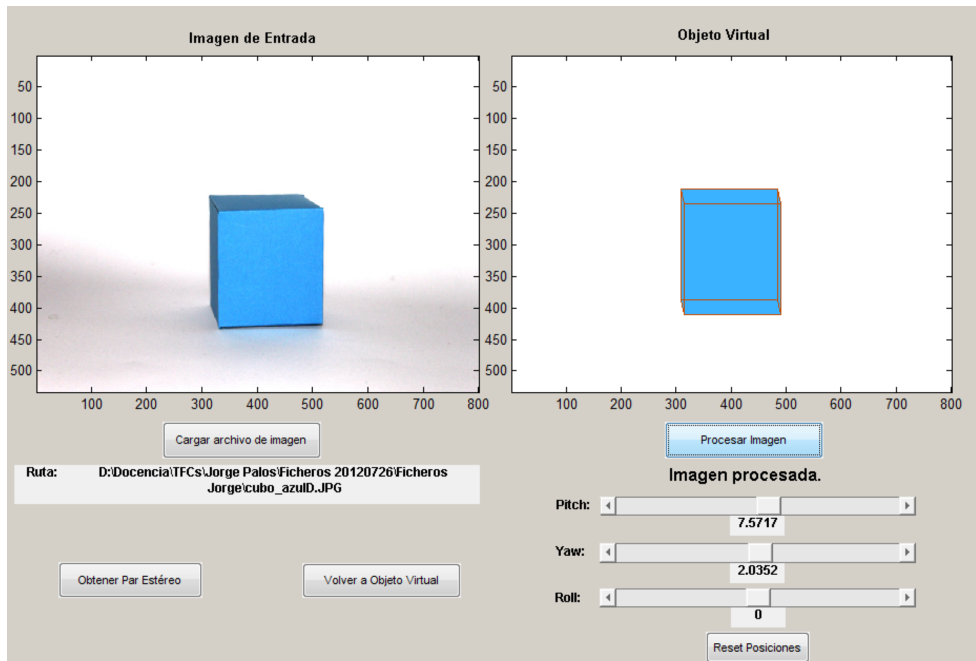

Figura 134: Cubo azul D en Matlab R2007a.

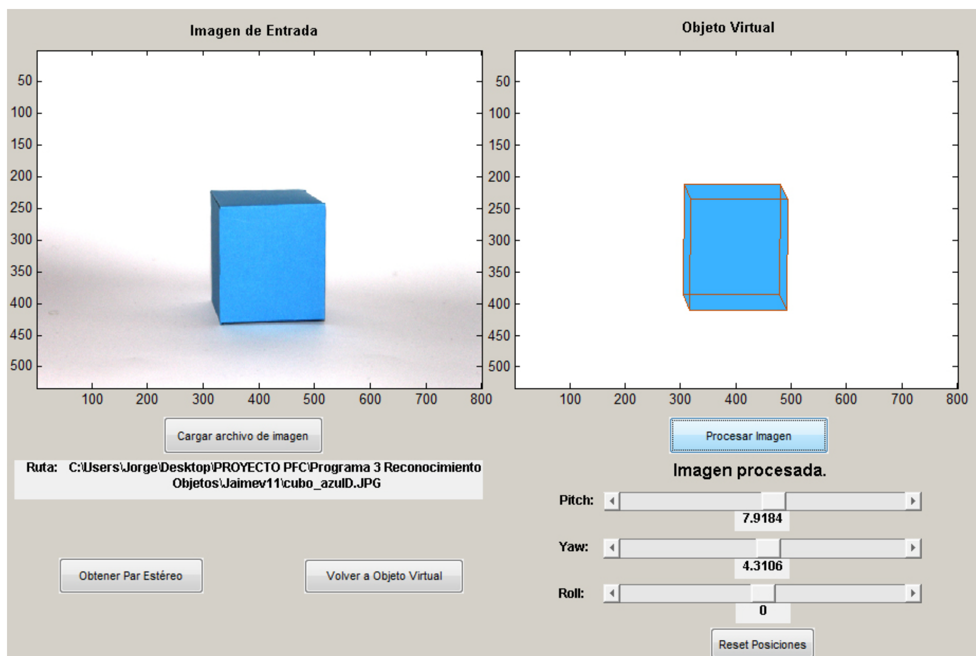

Figura 135: Cubo azul D en Matlab R2009b.

La imagen cubo azul E es procesada con orientación incorrecta por el software en la versión de Matlab 2009b (véase *figura 14*) mientras que no se procesa en la versión R2007a (*figura 13*).

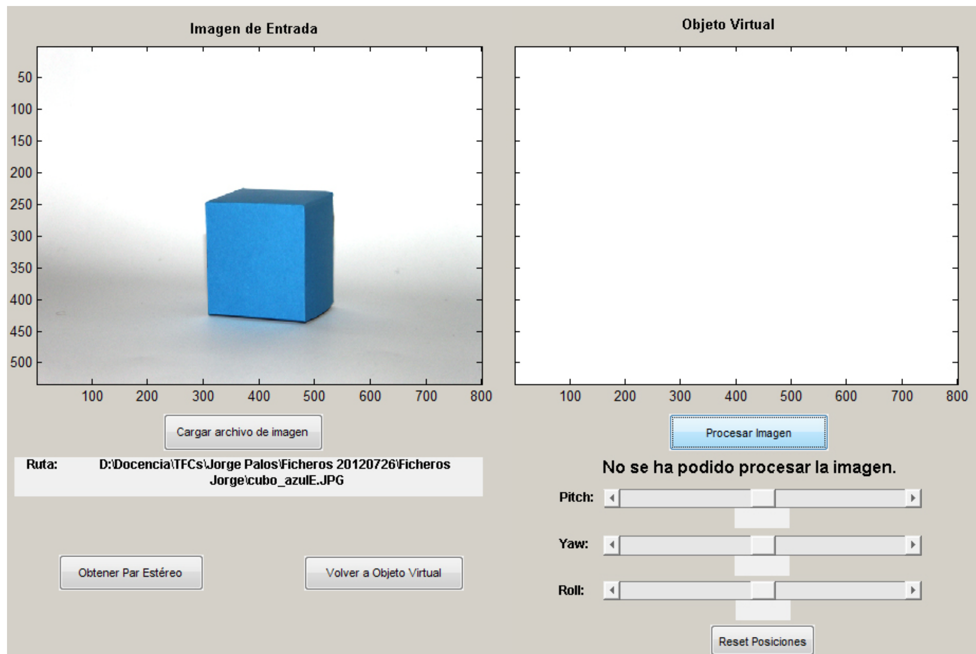

Figura 136: Cubo azul E en Matlab R2007a.

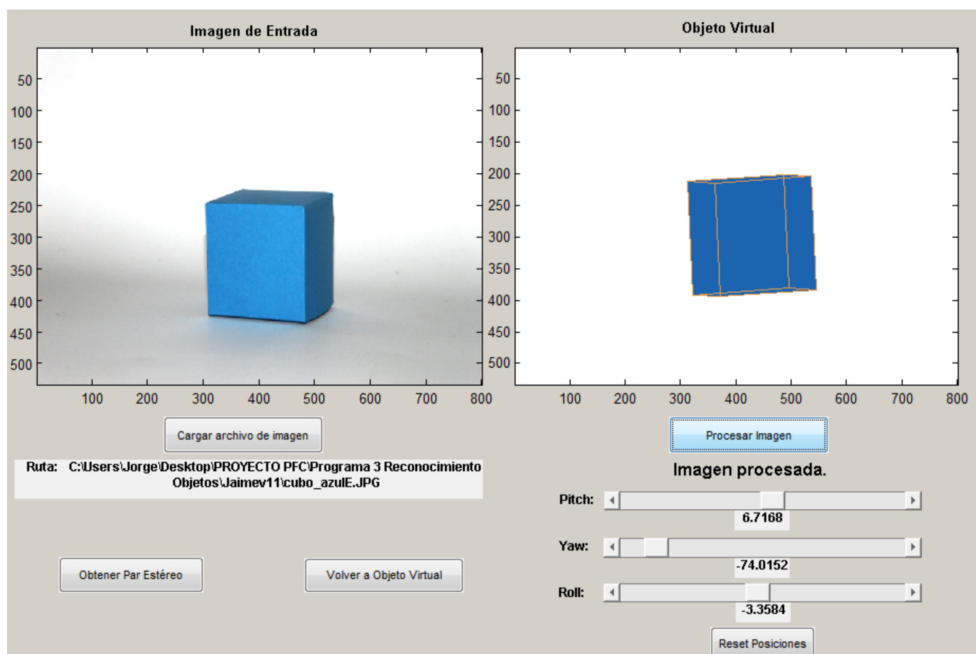

Figura 137: Cubo azul E en Matlab R2009b.

La imagen cubo azul F se procesa correctamente en ambas versiones, además los cálculos de orientación son correctos en las dos versiones de Matlab.

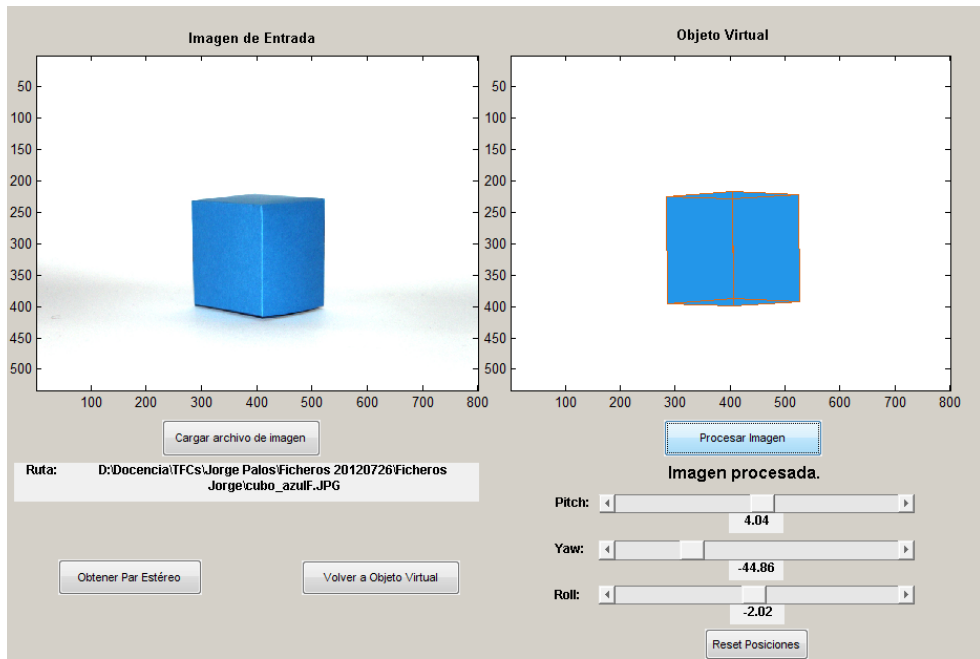

Figura 138: Cubo azul F en Matlab R2007a.

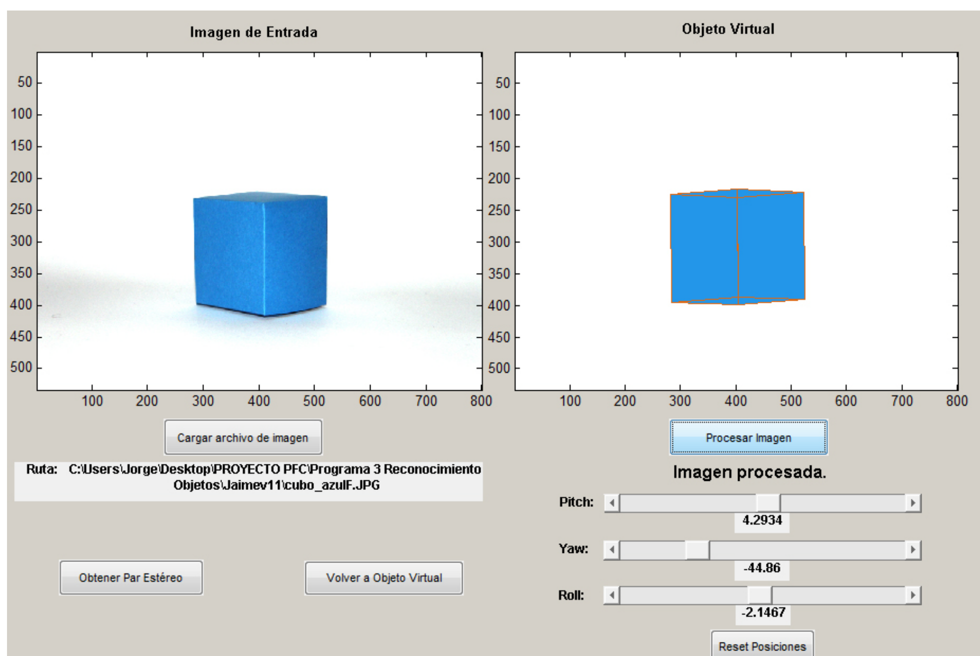

Figura 139: Cubo azul F en Matlab R2009b.

La imagen que a continuación se muestra en las *figura 17* y *figura 18* no se procesa en ninguna de las versiones.

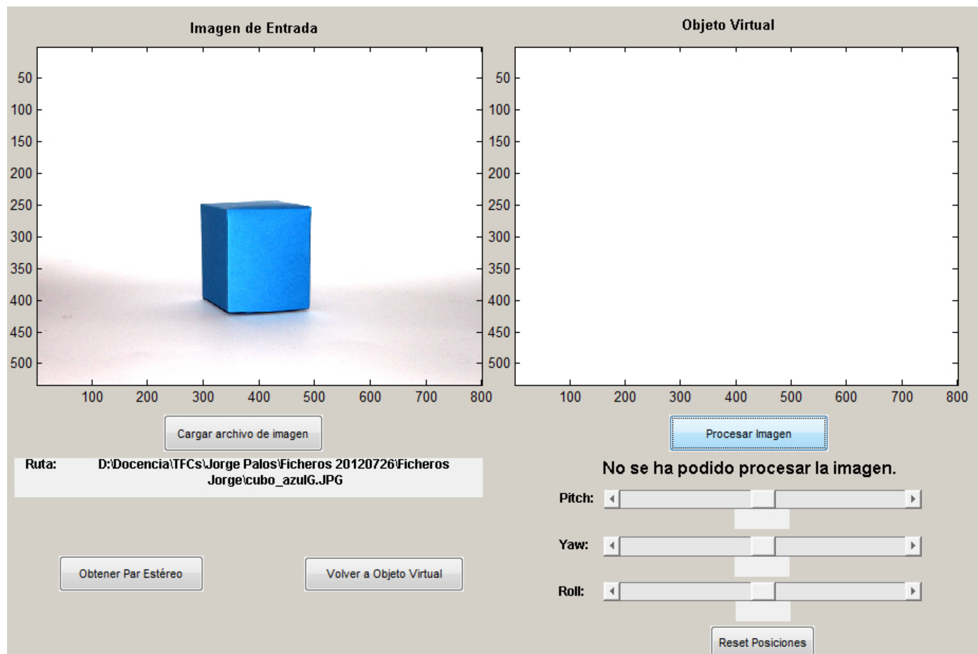

Figura 140: Cubo azul G en Matlab R2007a.

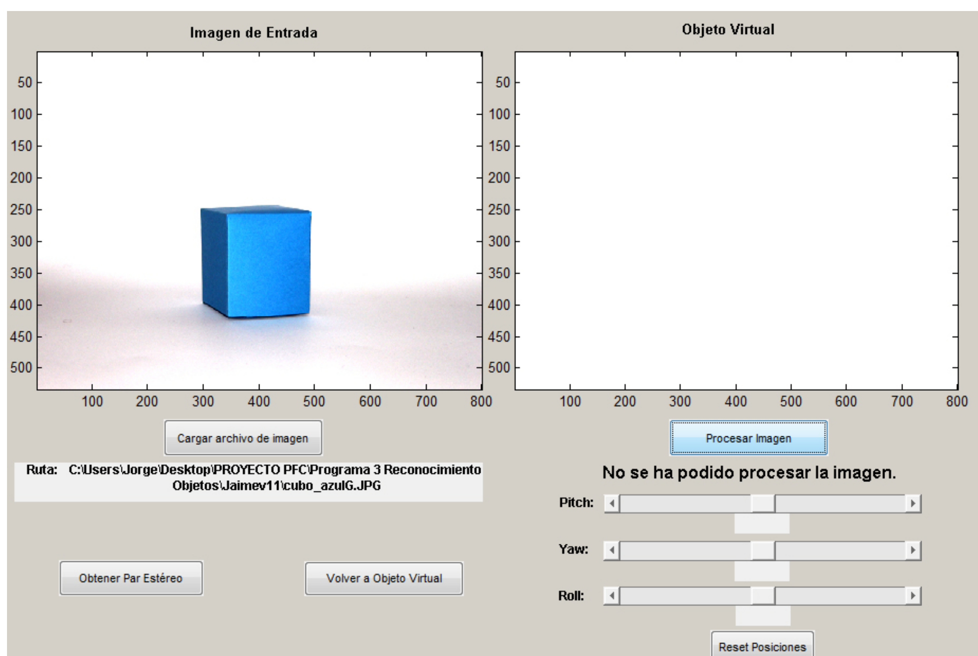

Figura 141: Cubo azul G en Matlab R2009b.

La imagen cubo rojo se procesa correctamente en las dos versiones, también lo hace el cálculo de la orientación.

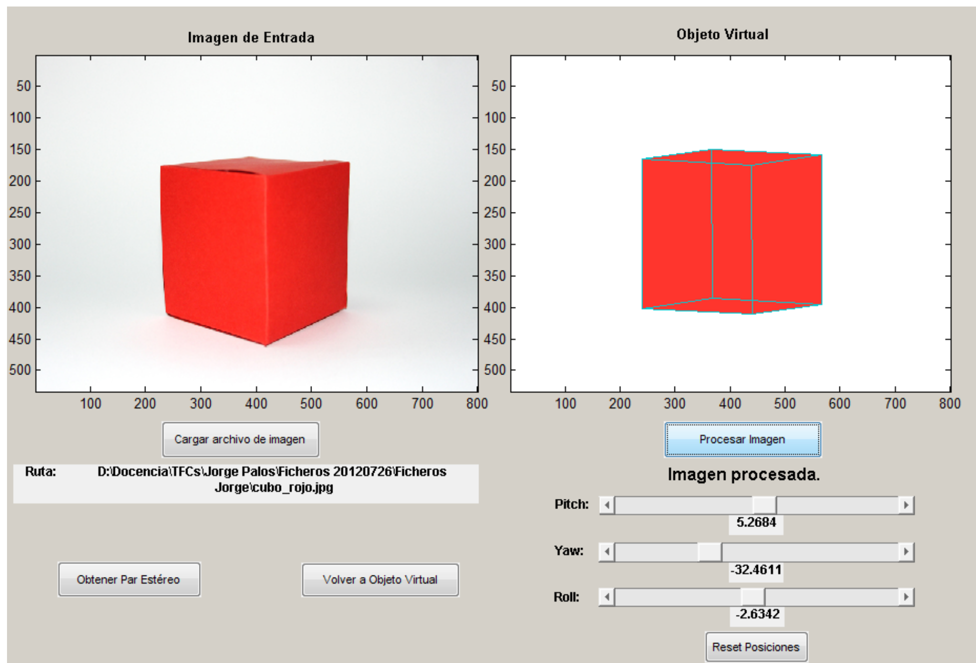

Figura 142: Cubo rojo en Matlab R2007a.

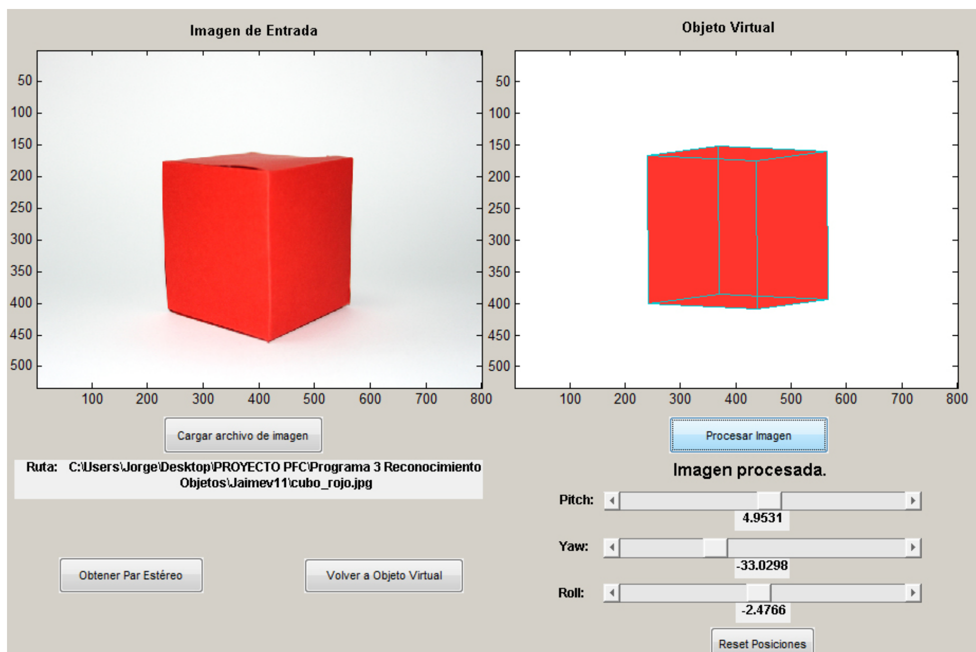

Figura 143: Cubo rojo en Matlab R2009b.

En las siguientes figuras vuelve a funcionar por igual en las dos versiones el software, muestra un error de procesado.

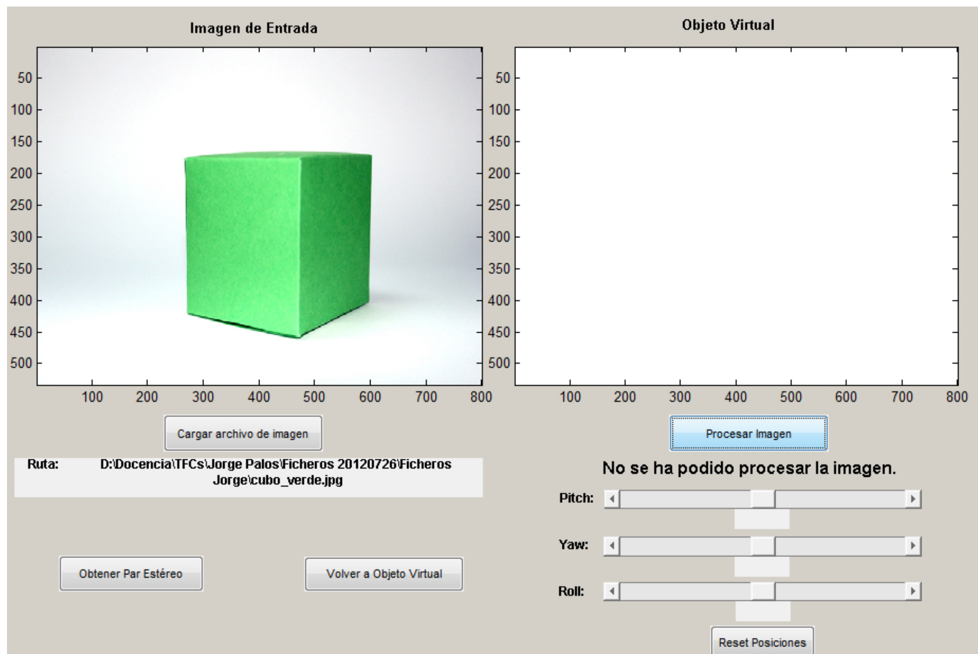

Figura 144: Cubo verde en Matlab R2007a.

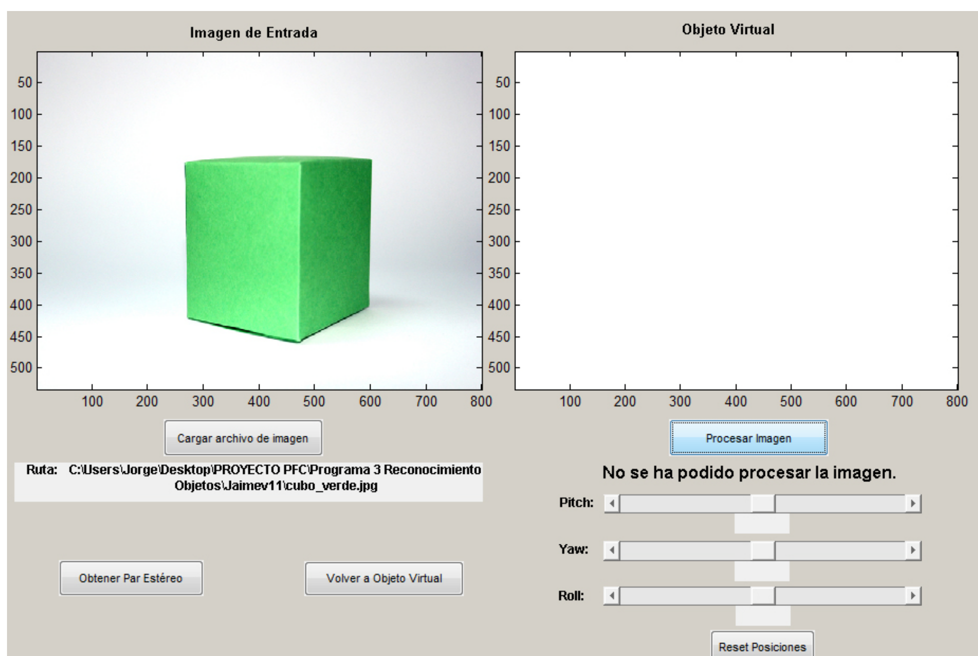

Figura 145: Cubo verde en Matlab R2009b.

La imagen cubo verde 2 se procesa correctamente en las dos versiones, también lo hace el cálculo de la orientación.

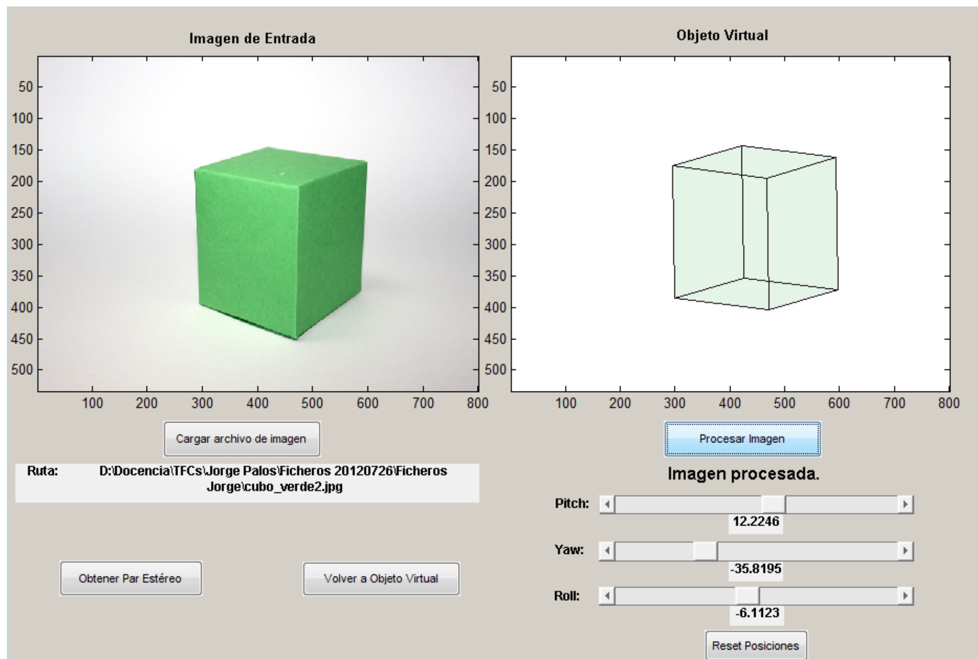

Figura 146: Cubo verde 2 en Matlab R2007a.

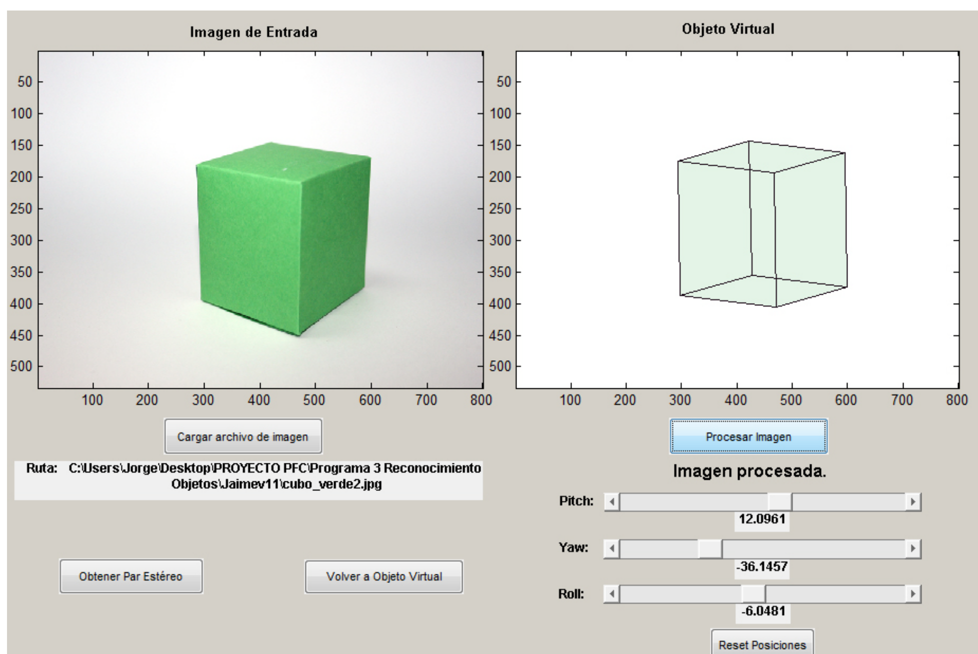

Figura 147: Cubo verde 2 en Matlab R2009b.

La imagen pirámide se procesa correctamente en las dos versiones, también lo hace el cálculo de la orientación.

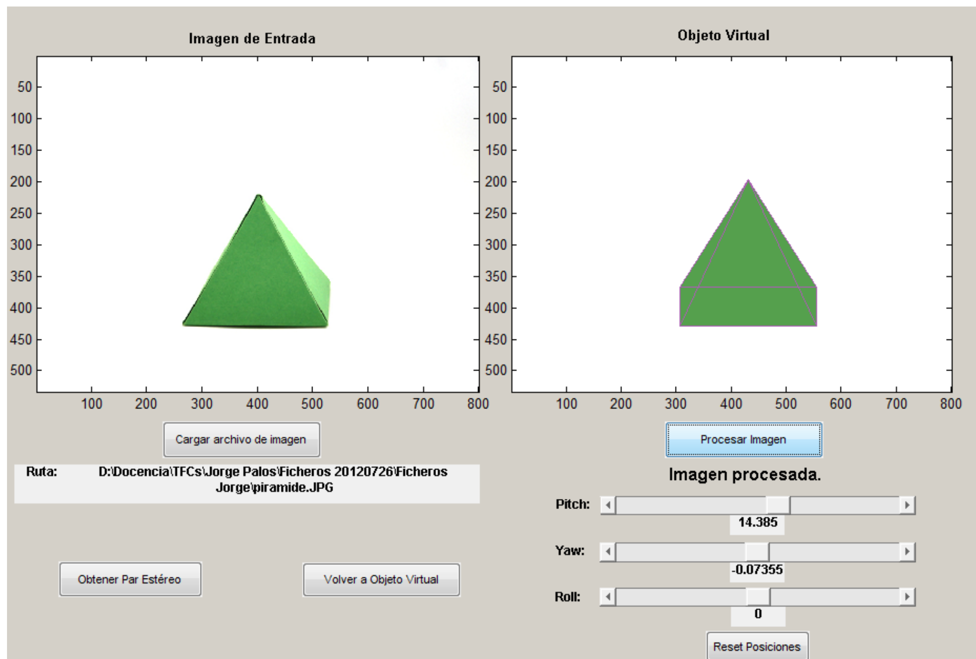

Figura 148: Cubo pirámide en Matlab R2007a.

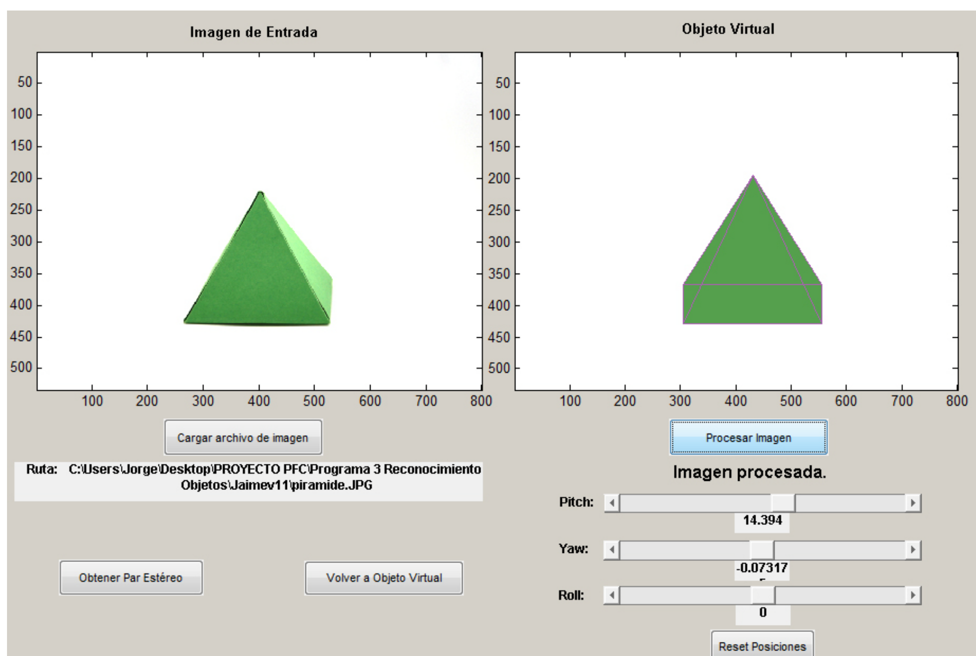

Figura 149: Cubo pirámide en Matlab R2009b.

Tal como se observa a continuación, en la versión más actual de Matlab el software funciona correctamente, mientras que en la otra se obtiene un error en el procesado.

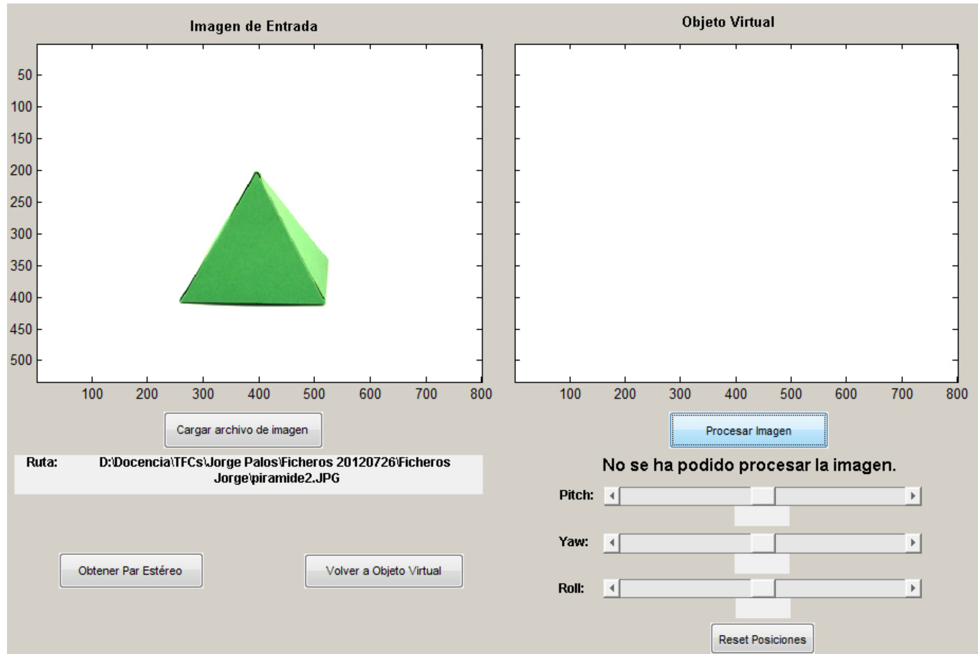

Figura 150: Cubo pirámide 2 en Matlab R2007a.

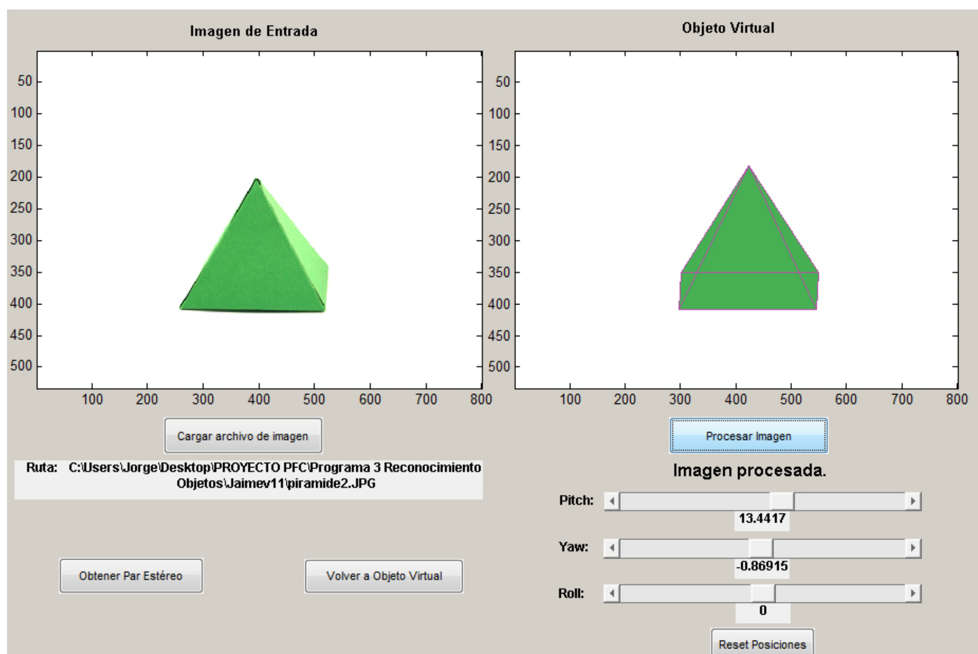

Figura 151: Cubo pirámide 2 en Matlab R2009b.

La imagen pirámide 3 se procesa correctamente en las dos versiones, también lo hace el cálculo de la orientación (hay una mínima diferencia en la rotación de un eje).

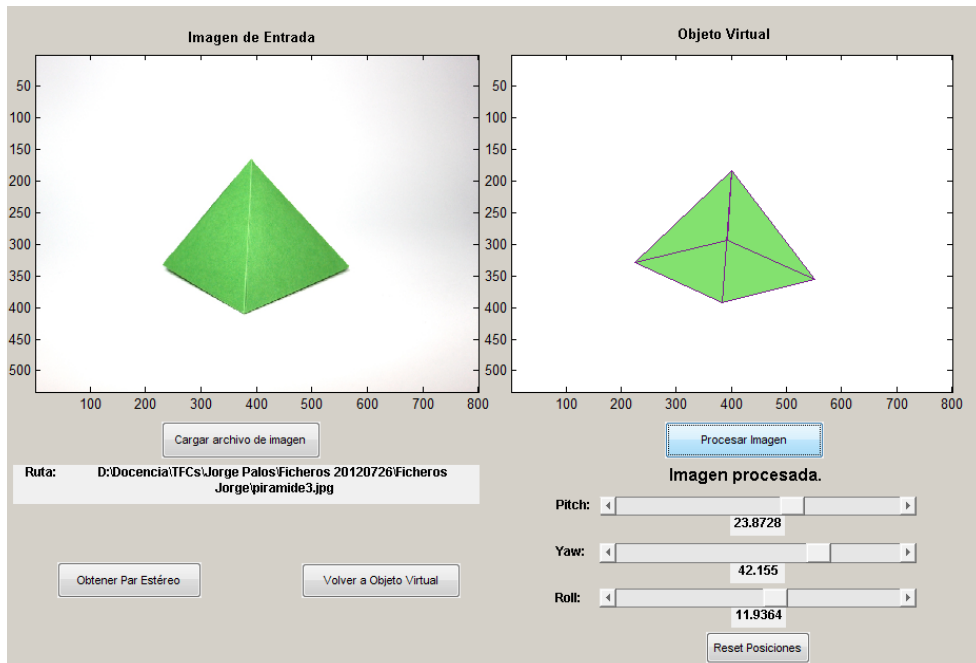

Figura 152: Cubo pirámide 3 en Matlab R2007a.

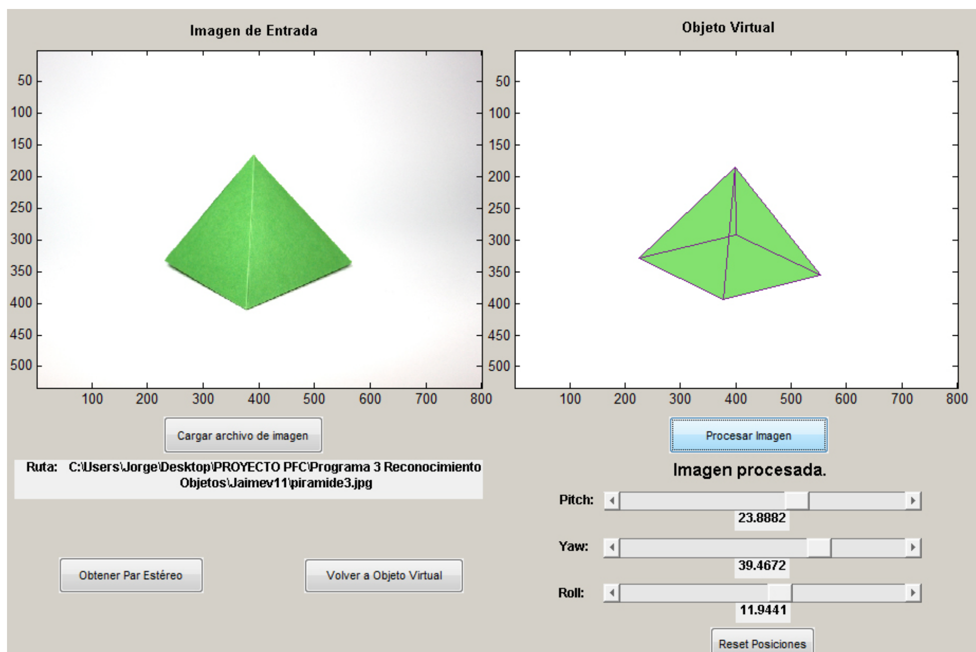

Figura 153: Cubo pirámide 3 en Matlab R2009b.

Tal como se observa en las *figura 31* y *figura 32*, en la versión más actual de Matlab el software funciona correctamente, mientras que en la otra se obtiene un error en el procesado.

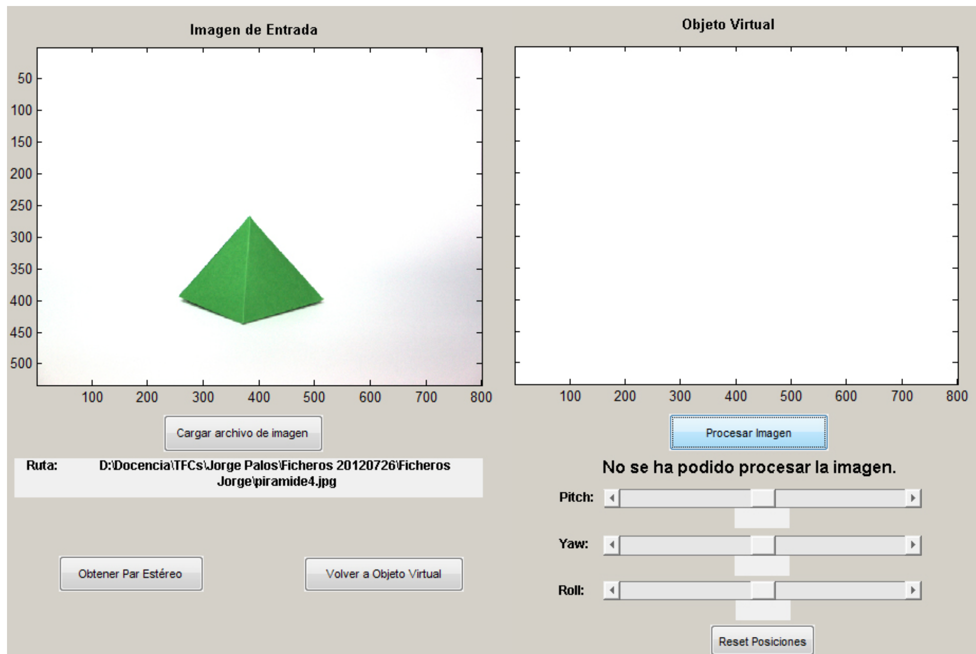

Figura 154: Cubo pirámide 4 en Matlab R2007a.

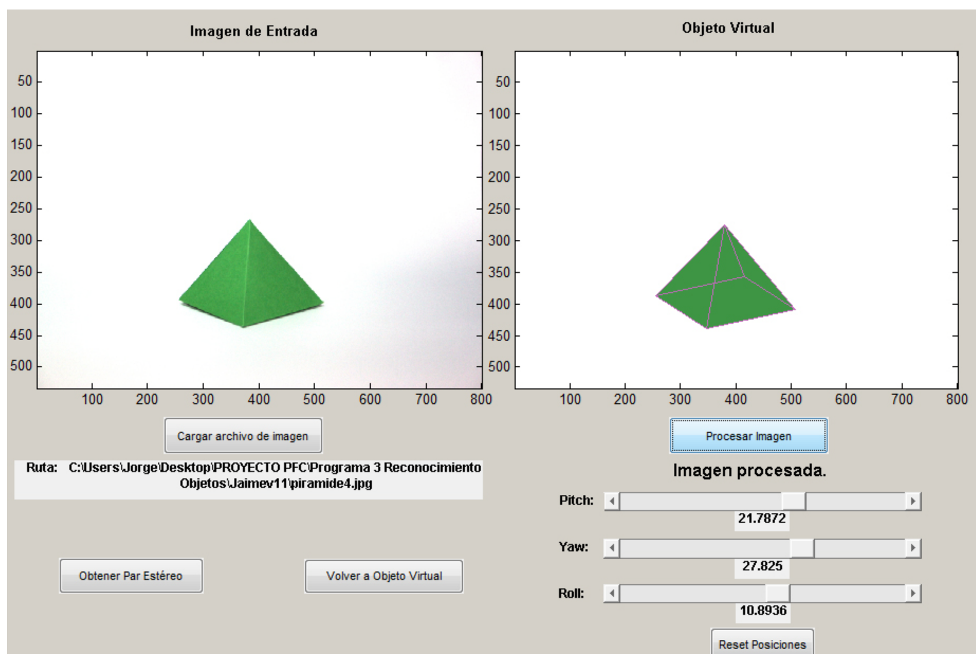

Figura 155: Cubo pirámide 4 en Matlab R2009b.